

Quicksort Algorithm

Quicksort Algorithm

Given an array of n elements (e.g., integers):

- If array only contains one element, return
- Else
 - pick one element to use as *pivot*.
 - Partition elements into two sub-arrays:
 - Elements less than or equal to pivot
 - Elements greater than pivot
 - Quicksort two sub-arrays
 - Return results

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	10	0
----	----	----	----	----	----	---	----	----	---

Pick Pivot

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	10	0
----	----	----	----	----	----	---	----	----	---

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements $<$ pivot
2. Another sub-array that contains elements \geq pivot

The sub-arrays are stored in the original data array.

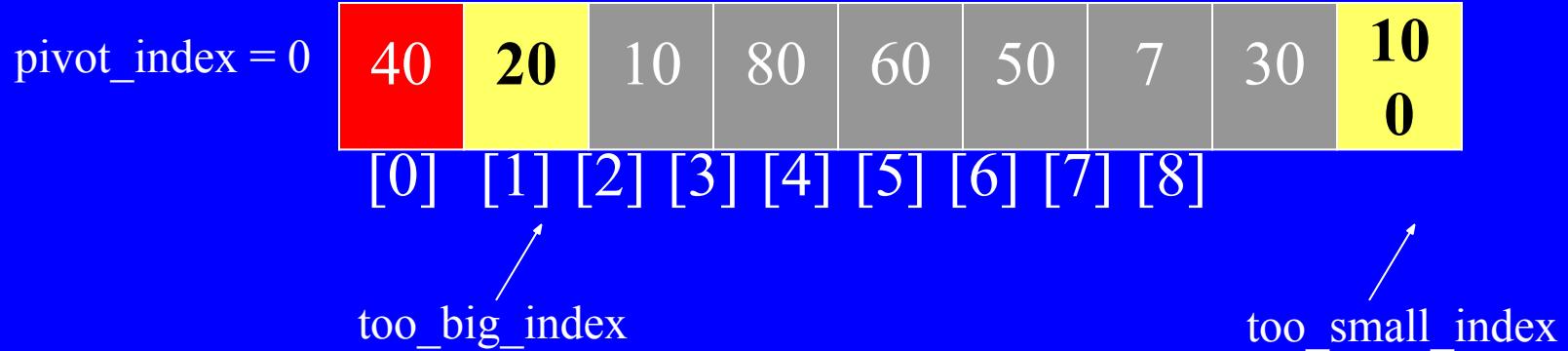
pivot_index = 0

40	20	10	80	60	50	7	30	10 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

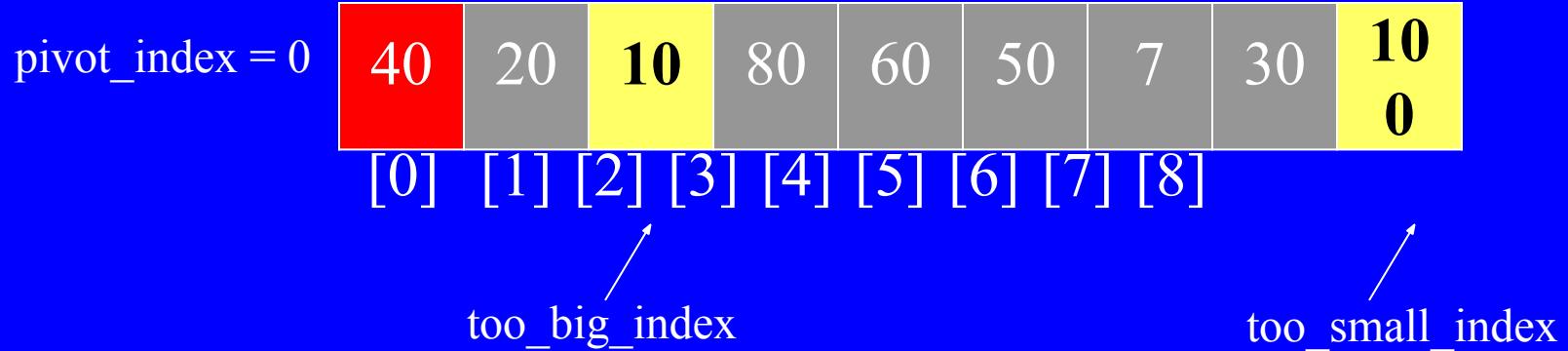
too_big_index

too_small_index

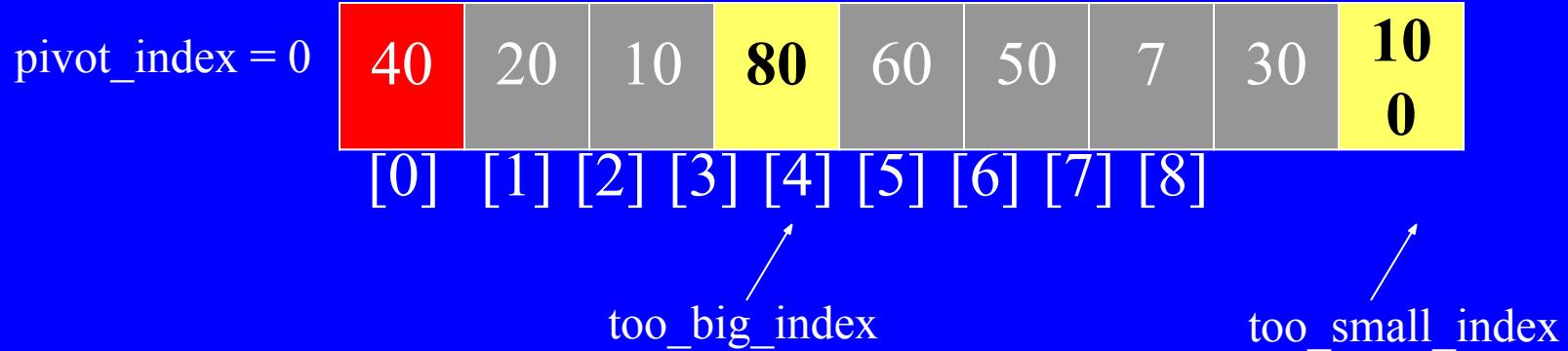
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index



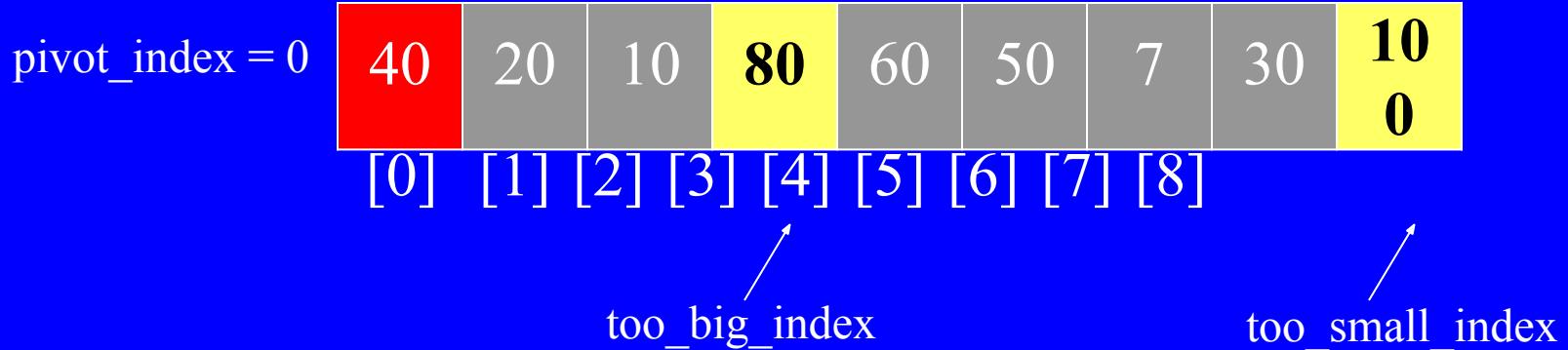
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index



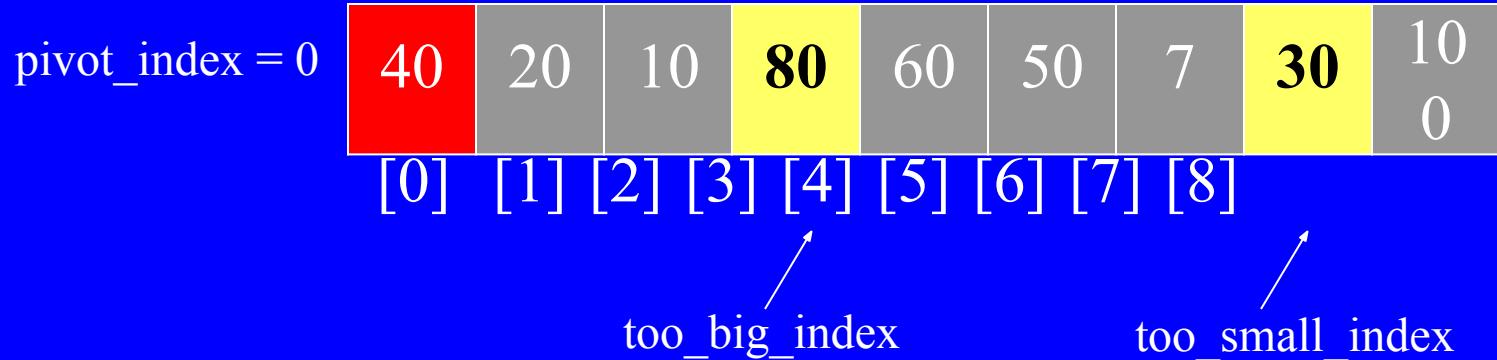
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index



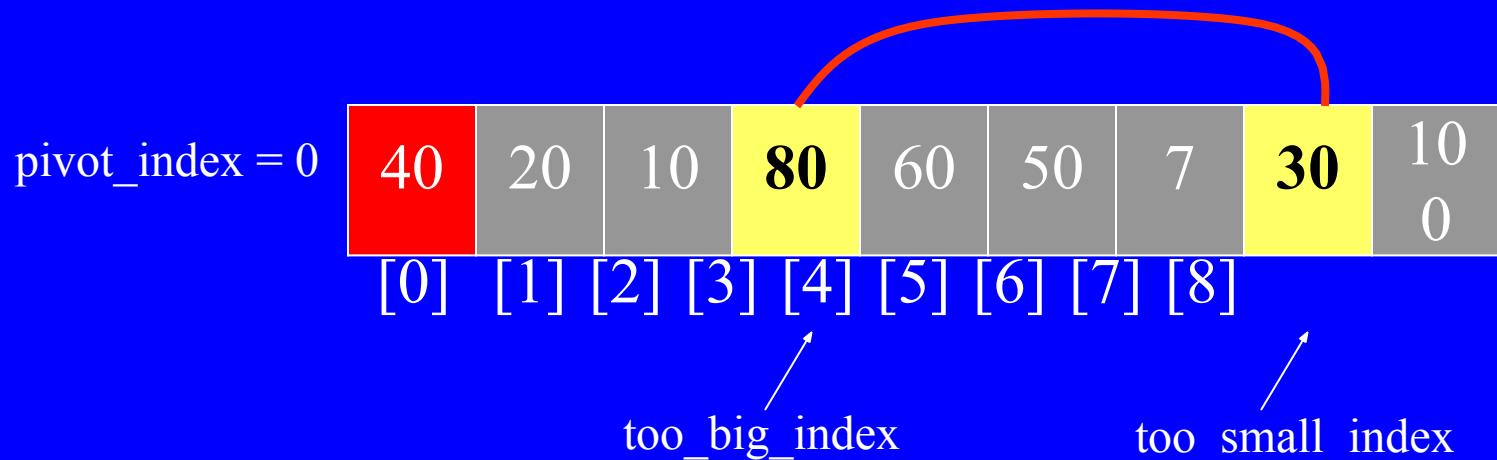
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 ++too_big_index
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 --too_small_index



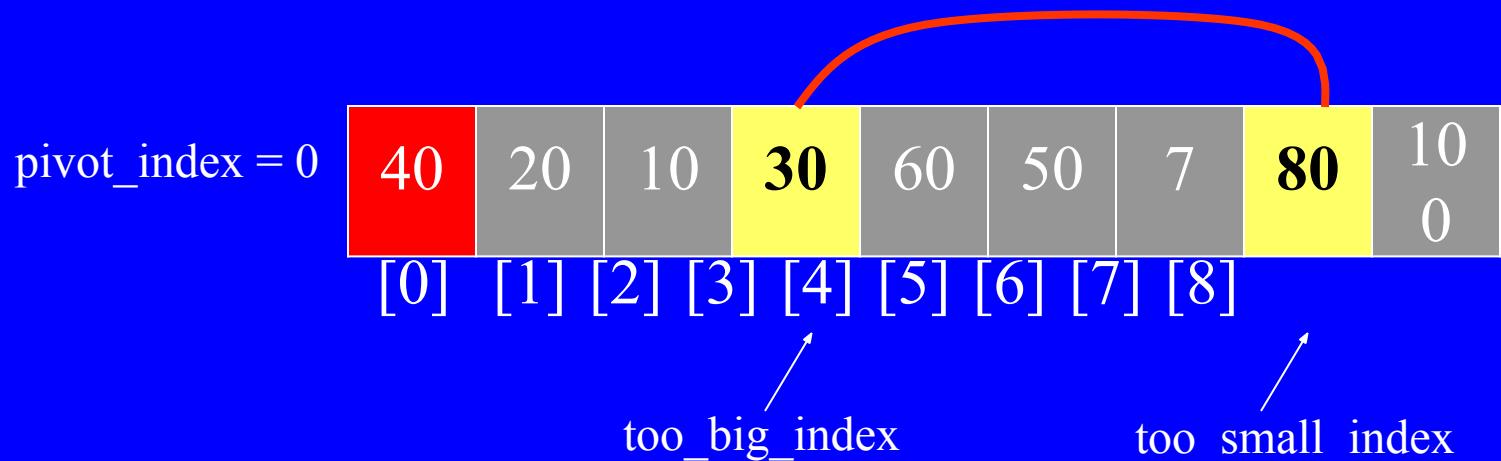
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$



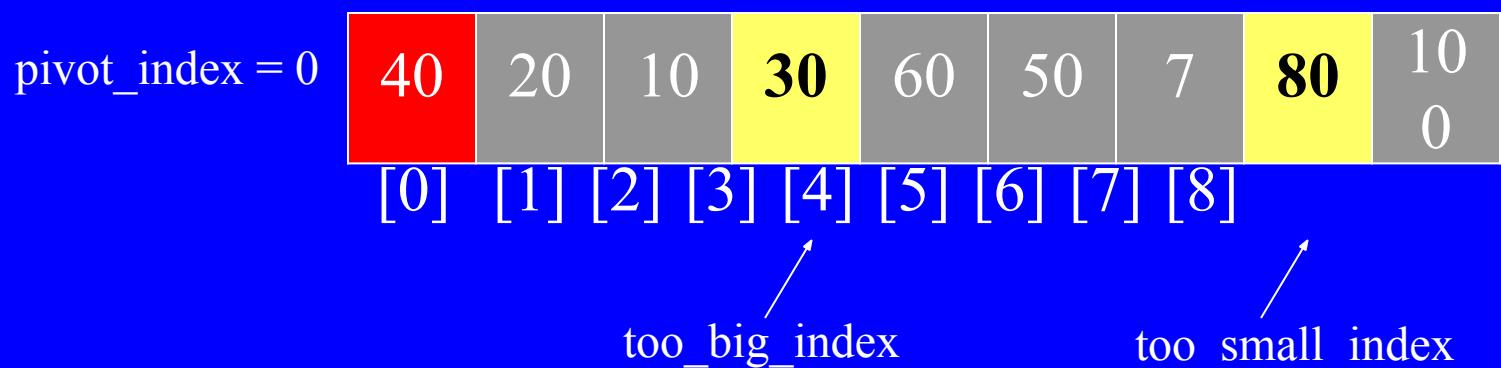
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

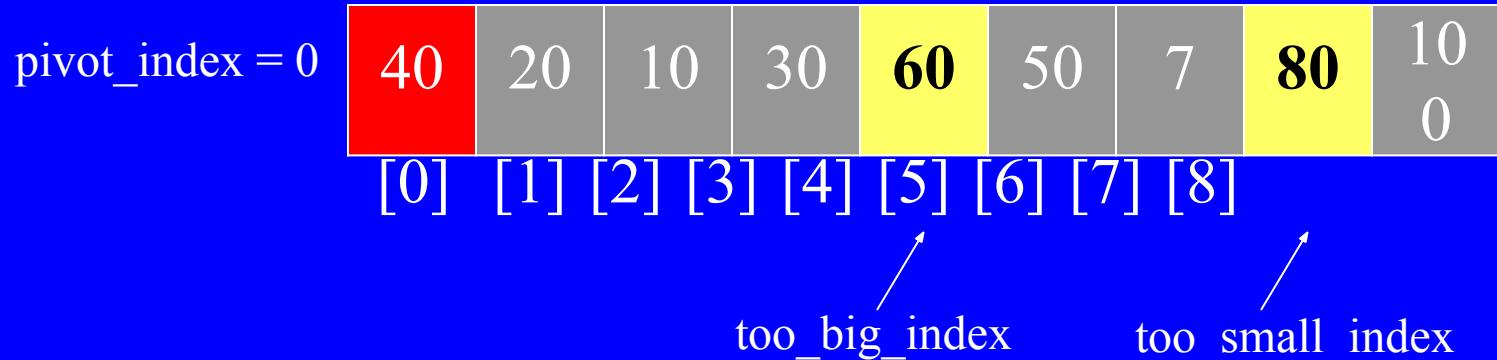


- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

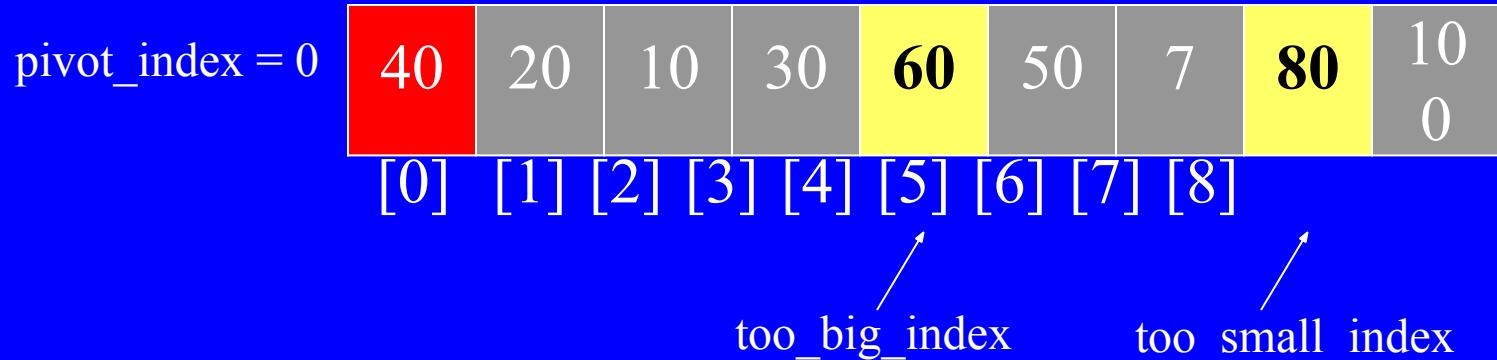
pivot_index = 0	40	20	10	30	60	50	7	80	10 0
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

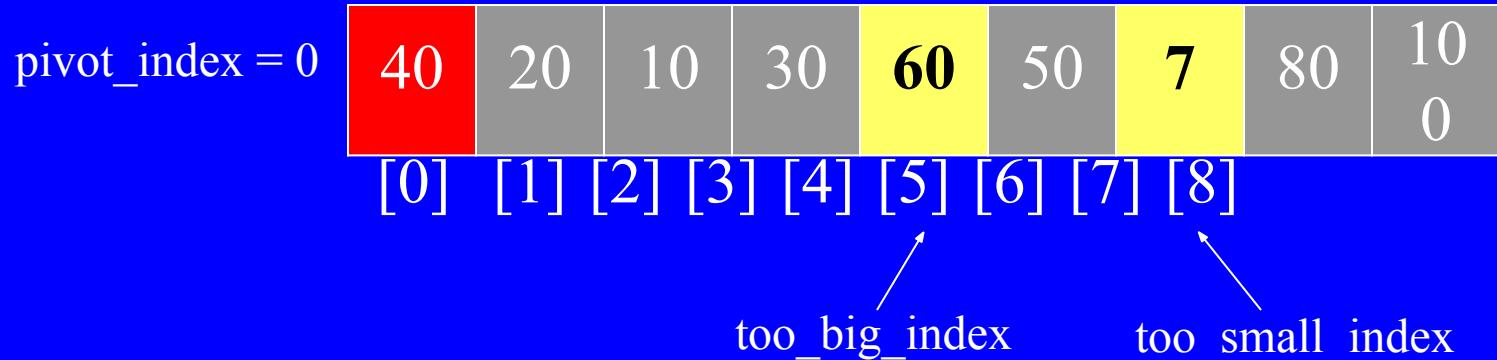
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



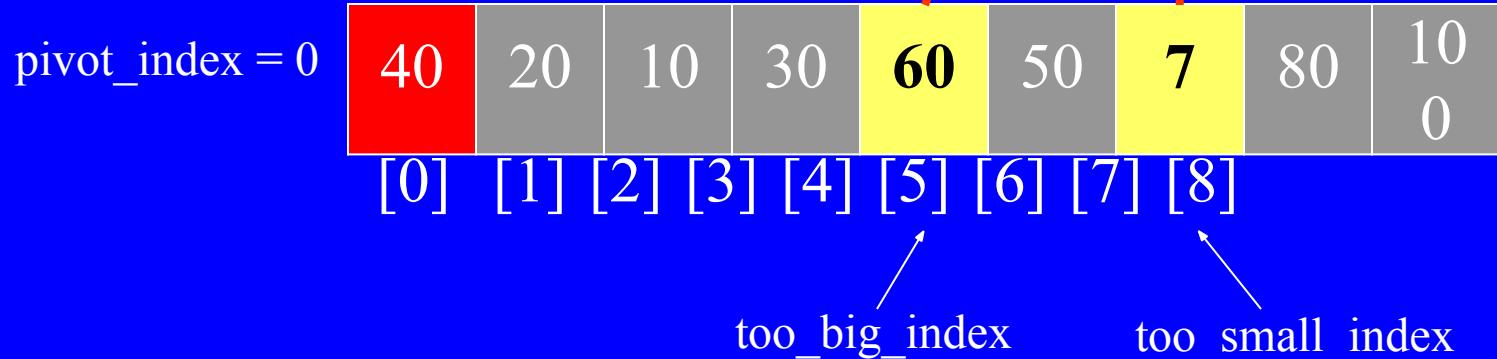
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



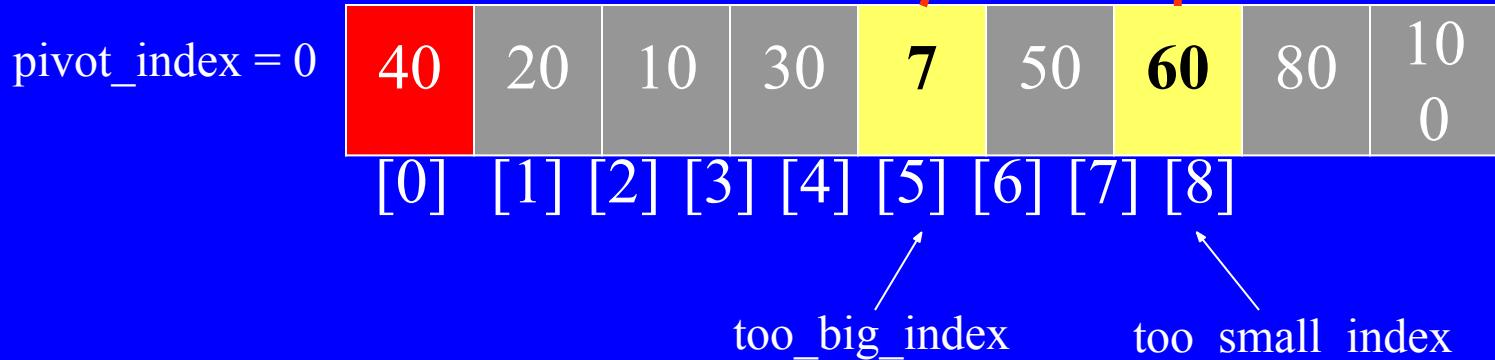
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

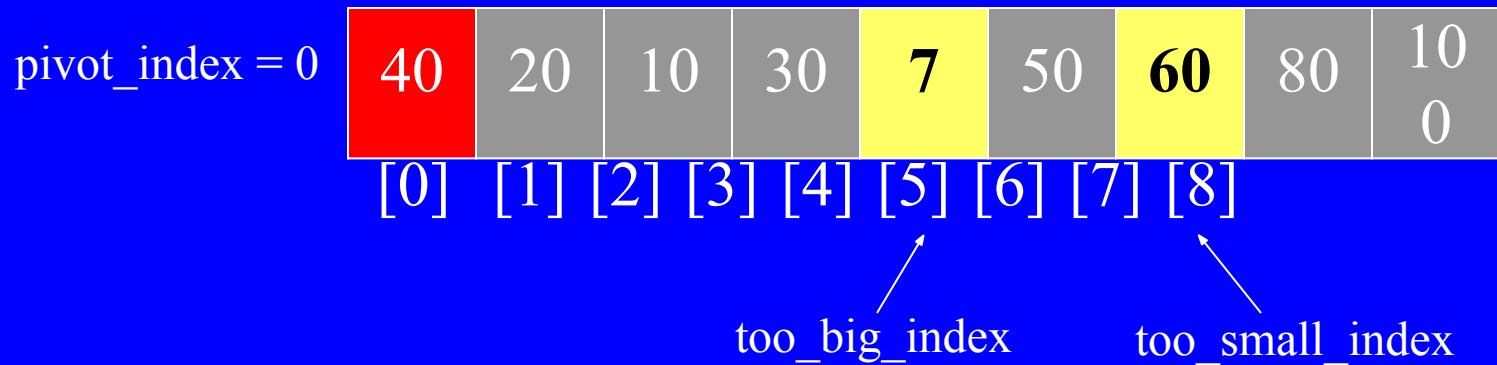


1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

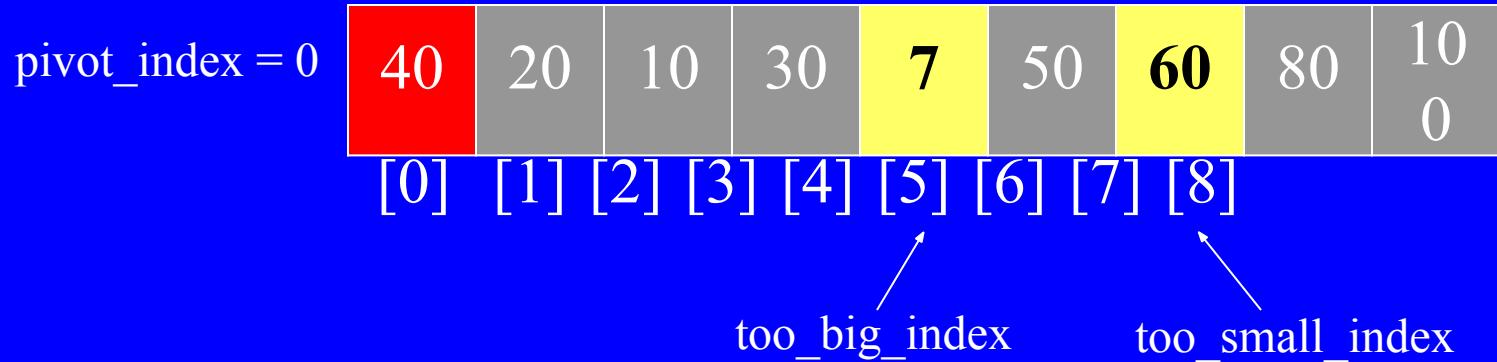


1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$

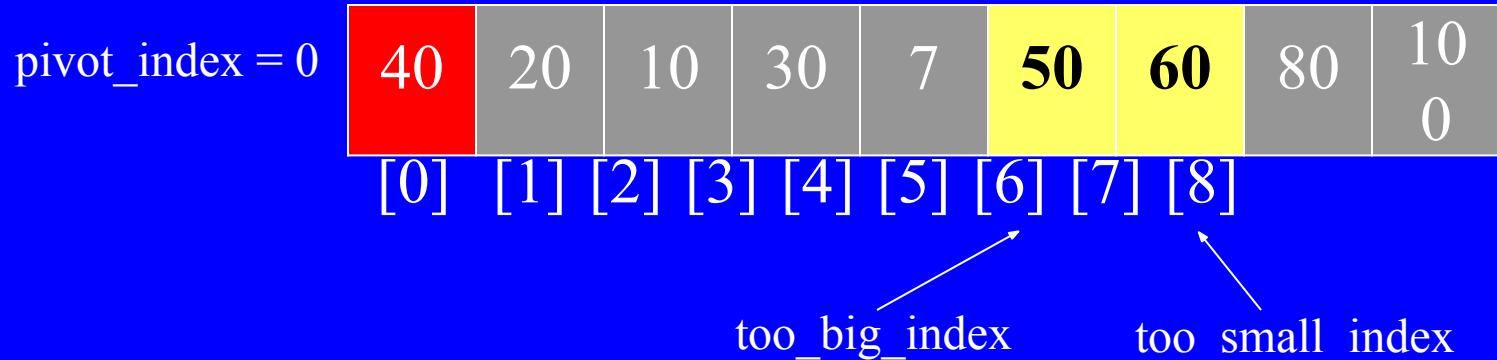
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



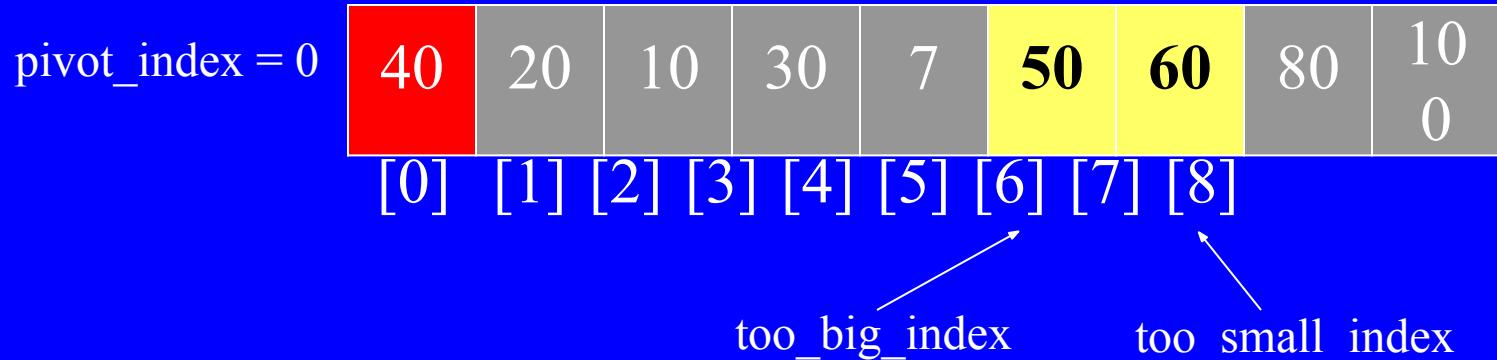
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

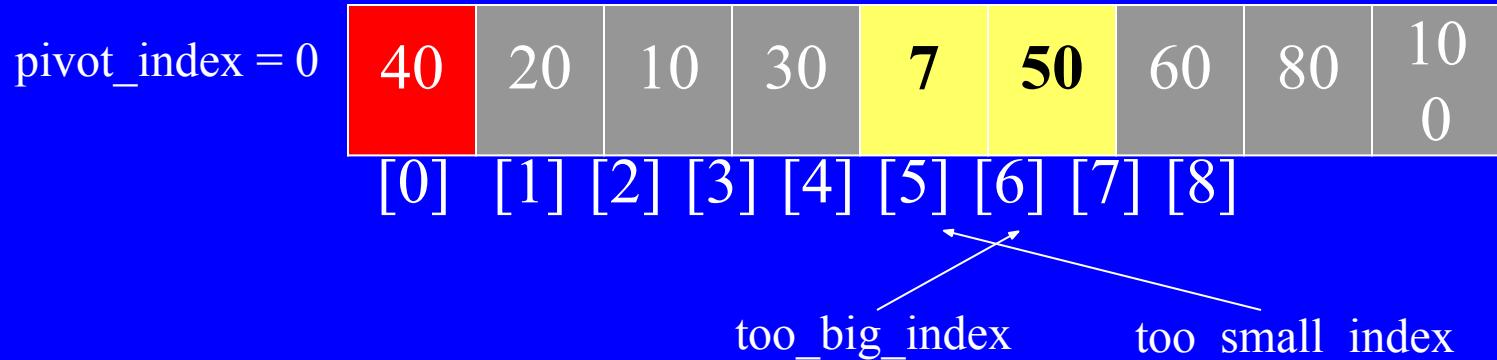
pivot_index = 0

40	20	10	30	7	50	60	80	10 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

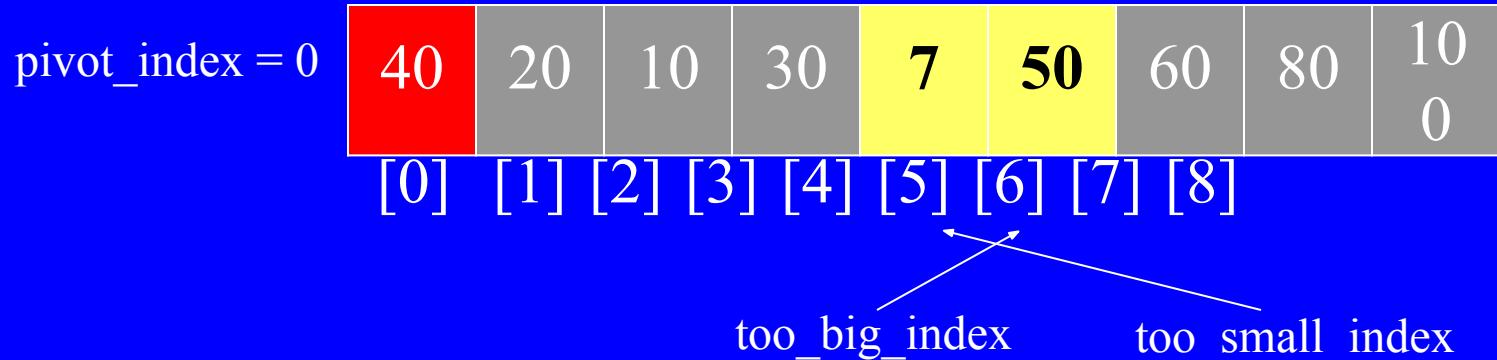
too_big_index too_small_index

```
graph TD; A[40] --- B[20]; B --- C[10]; C --- D[30]; D --- E[7]; E --- F[50]; F --- G[60]; G --- H[80]; H --- I[10]; I --- J[0];
```

1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
- 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.

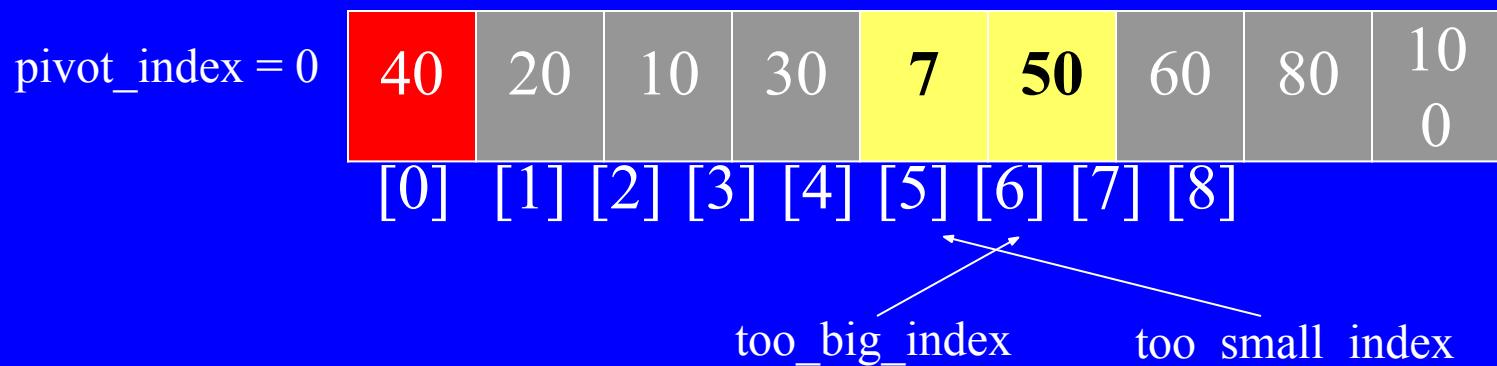


1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
- 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$

4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

$\text{pivot_index} = 0$



[0] [1] [2] [3] [4] [5] [6] [7] [8]
 ↑
 too_big_index
 ↑
 too_small_index

1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$

`pivot_index = 4`

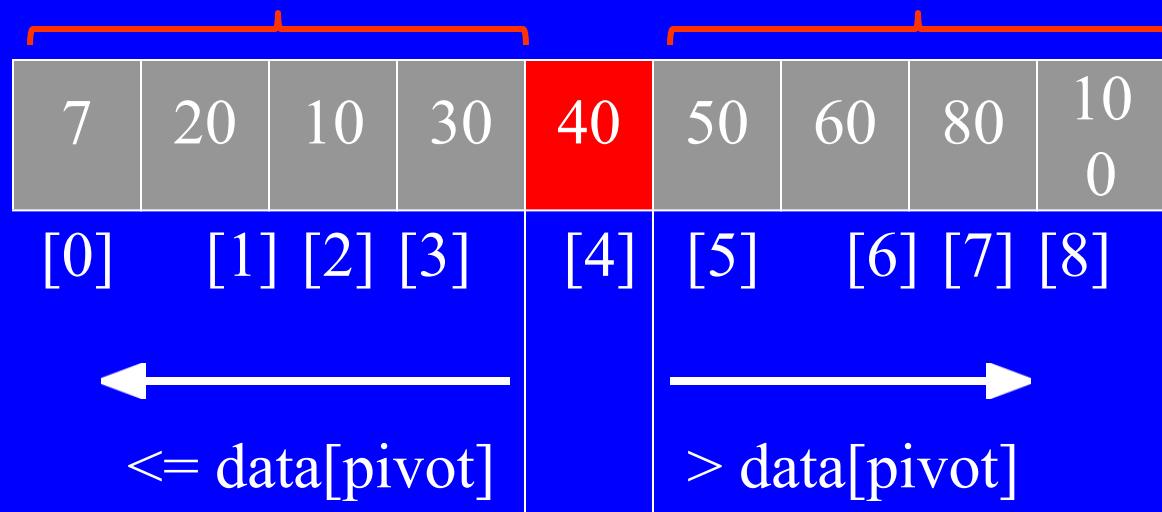
7	20	10	30	40	50	60	80	10 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

`too_big_index` `too_small_index`

Partition Result

7	20	10	30	40	50	60	80	10 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
←					→			
<= data[pivot]					> data[pivot]			

Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays of size $n/2$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses in partition? $O(n)$

Quicksort Analysis

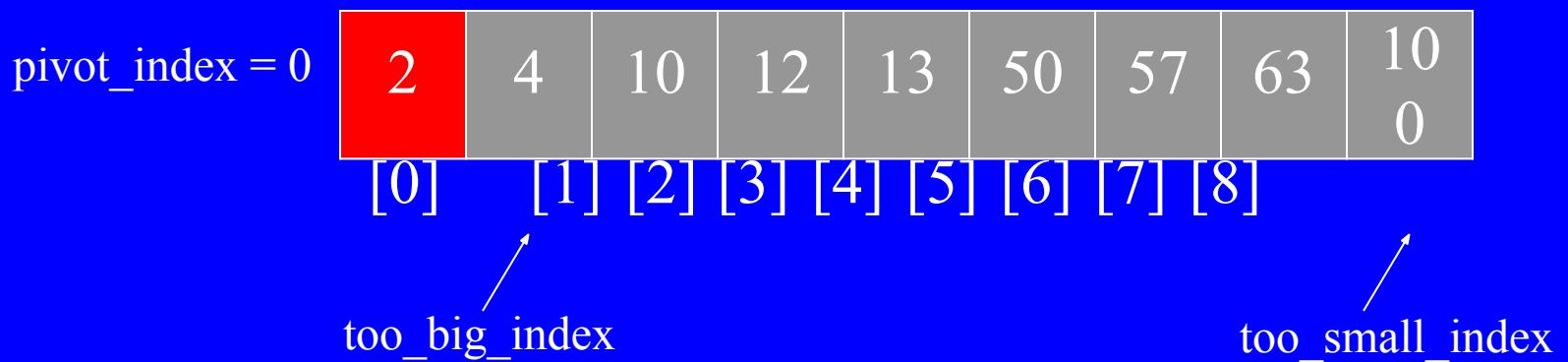
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

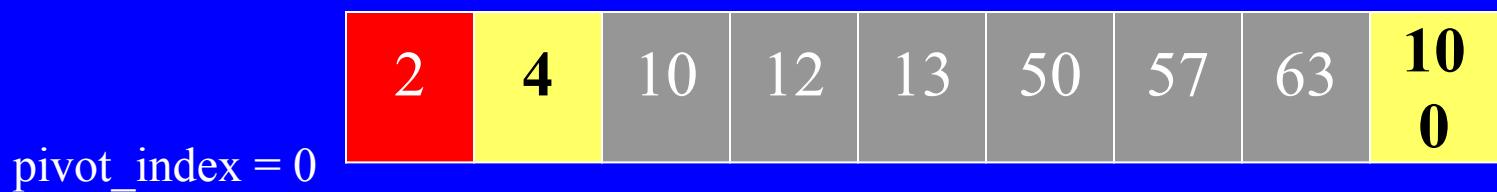
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

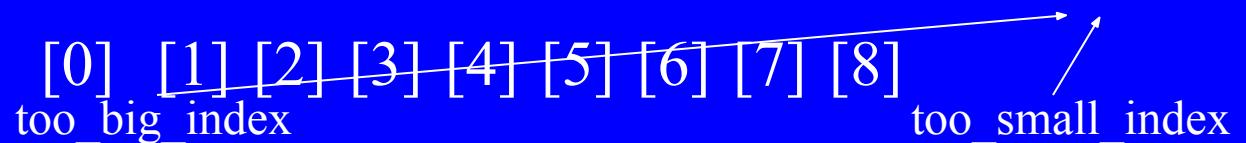
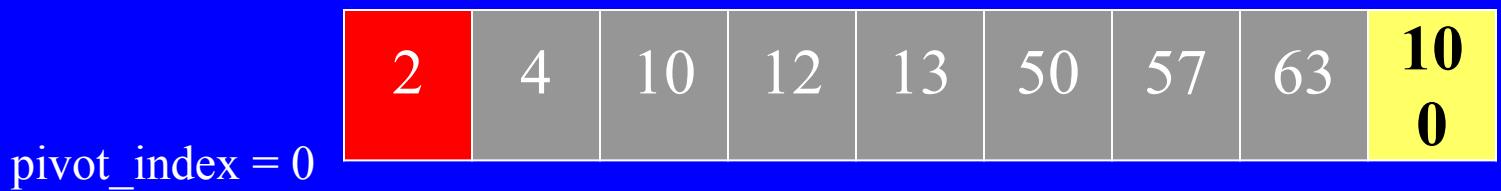
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



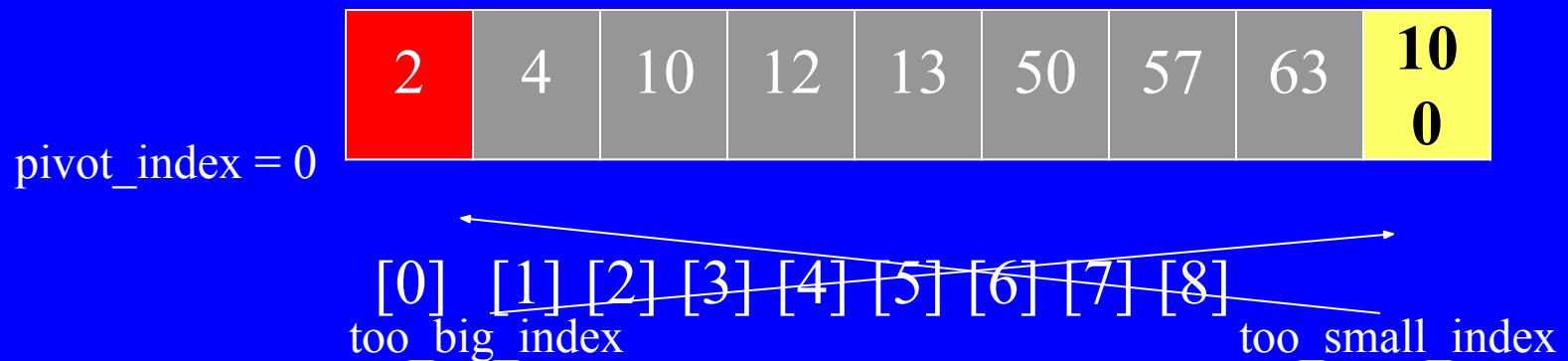
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad ++\text{too_big_index}$
 - 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad --\text{too_small_index}$
 - 3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
 - 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
 - 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



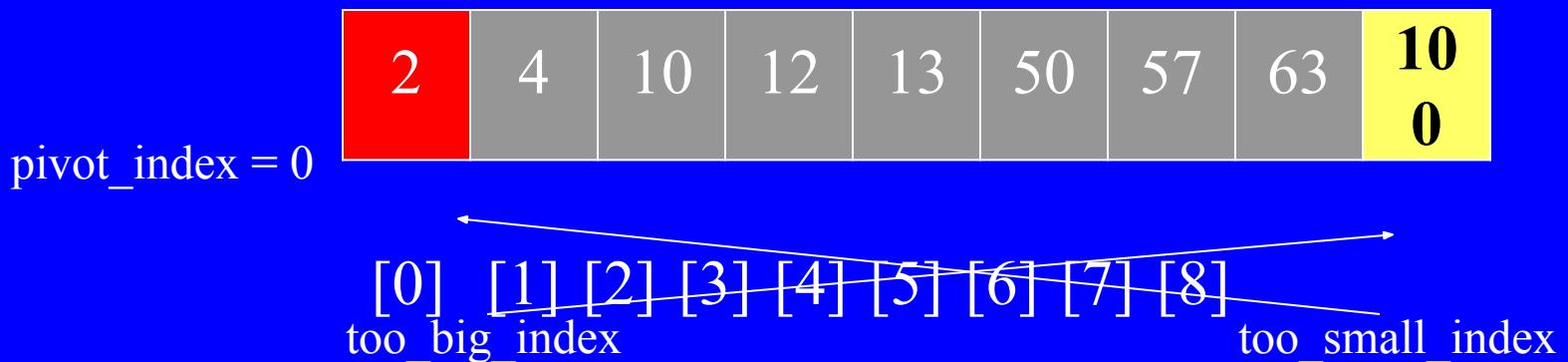
- 1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



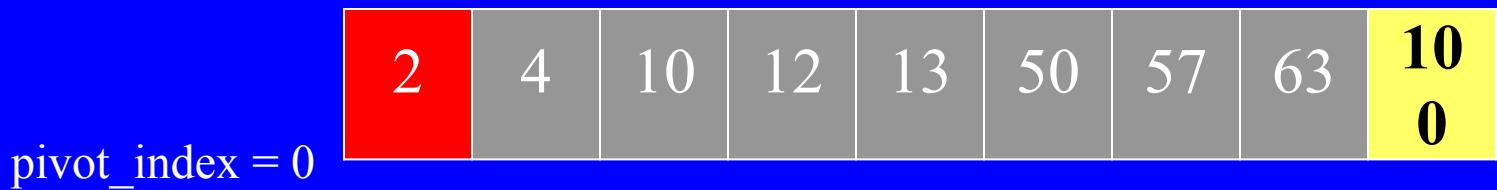
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
 - 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
 3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



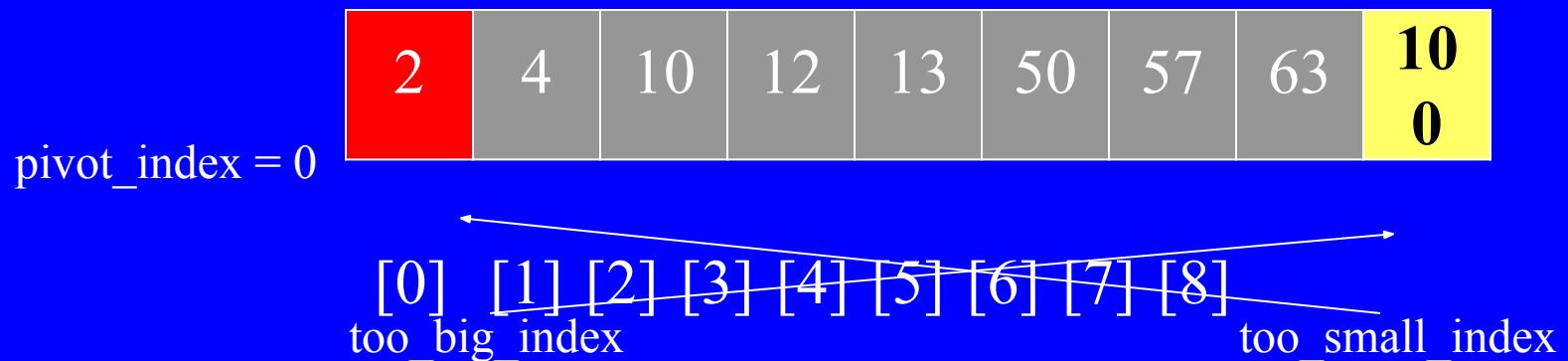
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
 - 3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



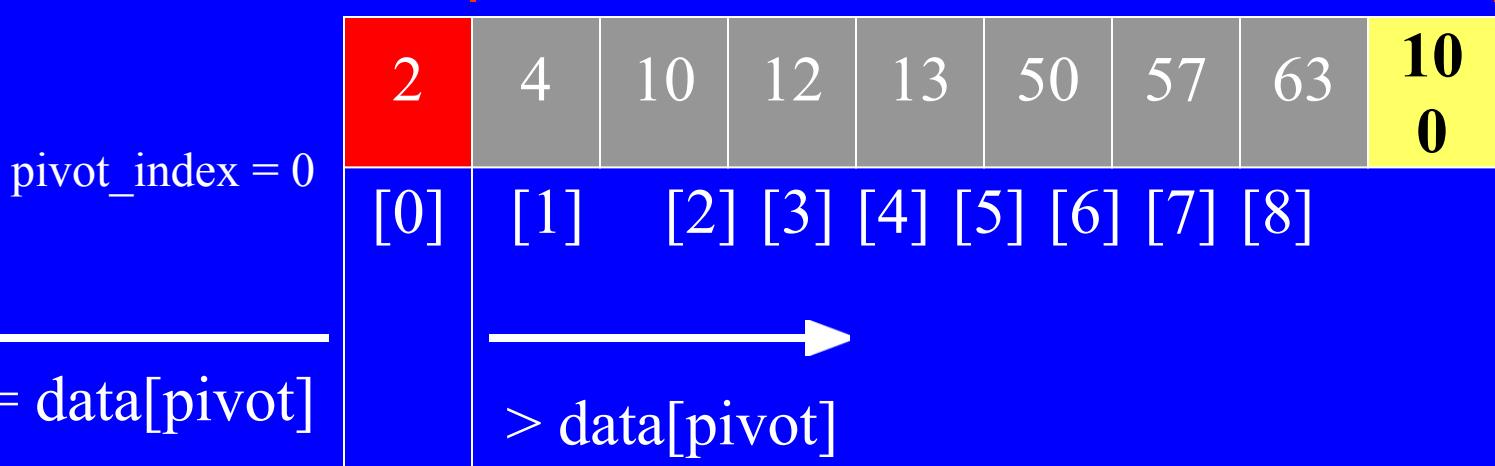
1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
 2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
 3. If $\text{too_big_index} < \text{too_small_index}$
 $\quad \quad \quad \text{swap } \text{data}[\text{too_big_index}] \text{ and } \text{data}[\text{too_small_index}]$
 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
 5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{++too_big_index}$
2. While $\text{data}[\text{too_small_index}] > \text{data}[\text{pivot}]$
 $\quad \quad \quad \text{--too_small_index}$
3. If $\text{too_big_index} < \text{too_small_index}$
 swap $\text{data}[\text{too_big_index}]$ and $\text{data}[\text{too_small_index}]$
- 4. While $\text{too_small_index} > \text{too_big_index}$, go to 1.
5. Swap $\text{data}[\text{too_small_index}]$ and $\text{data}[\text{pivot_index}]$



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)!!!$
- What can we do to avoid worst case?

Improved Pivot Selection

Pick mid value from data array: $\text{data}[n/2]$.

Use this mid value as pivot.

Improved Pivot Selection

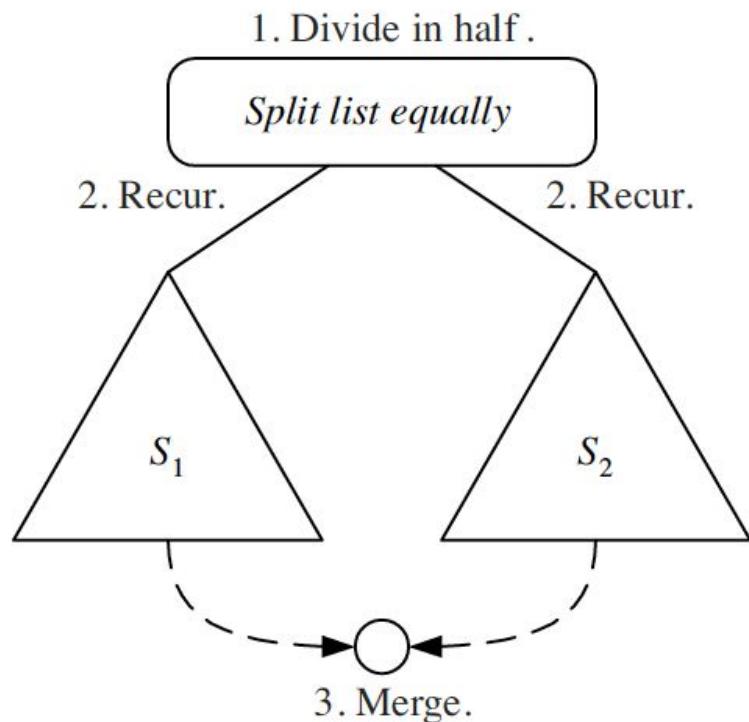
Pick median value of three elements from data array:
 $\text{data}[0]$, $\text{data}[n/2]$, and $\text{data}[n-1]$.

Use this median value as pivot.

Merge Sort

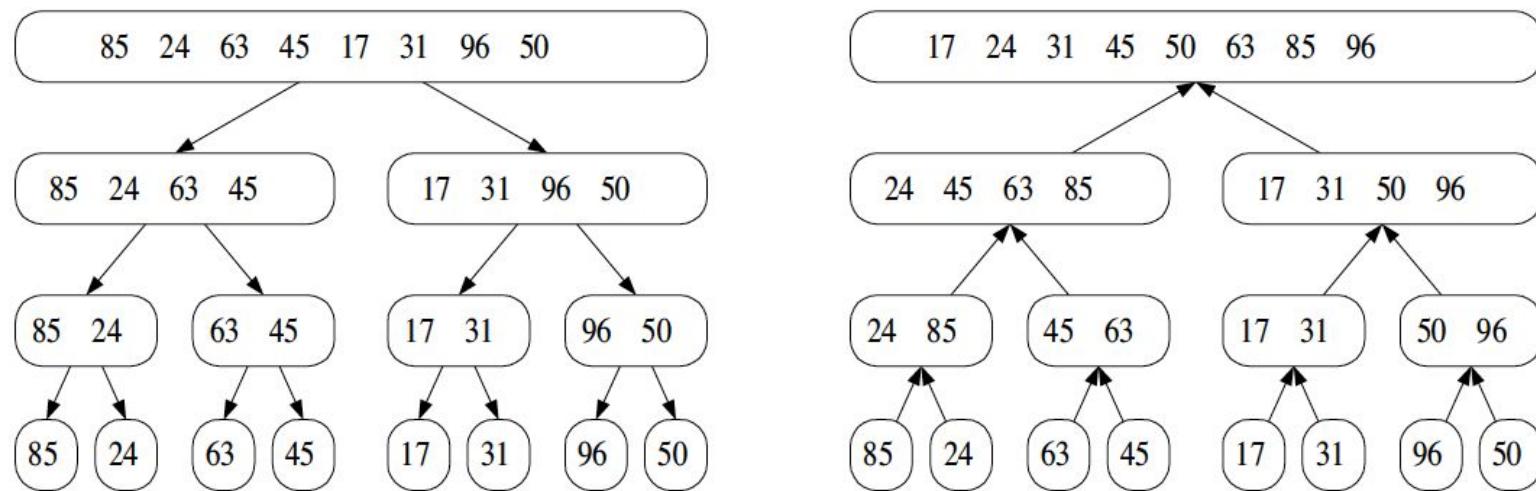
Divide-and-Conquer

- ◆ Divide-and conquer is a general algorithm design paradigm:
 - **Divide:** divide the input data S in two disjoint subsets S_1 and S_2
 - **Recur:** solve the subproblems associated with S_1 and S_2
 - **Conquer:** combine the solutions for S_1 and S_2 into a solution for S
- ◆ The base case for the recursion are subproblems of size 0 or 1



Merge-Sort

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm



The Merge-Sort Algorithm

Merge-sort on an input sequence S with n elements consists of three steps:

- **Divide:** partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- **Recur:** recursively sort S_1 and S_2
- **Conquer:** merge S_1 and S_2 into a unique sorted sequence

Merging Two Sorted Sequences

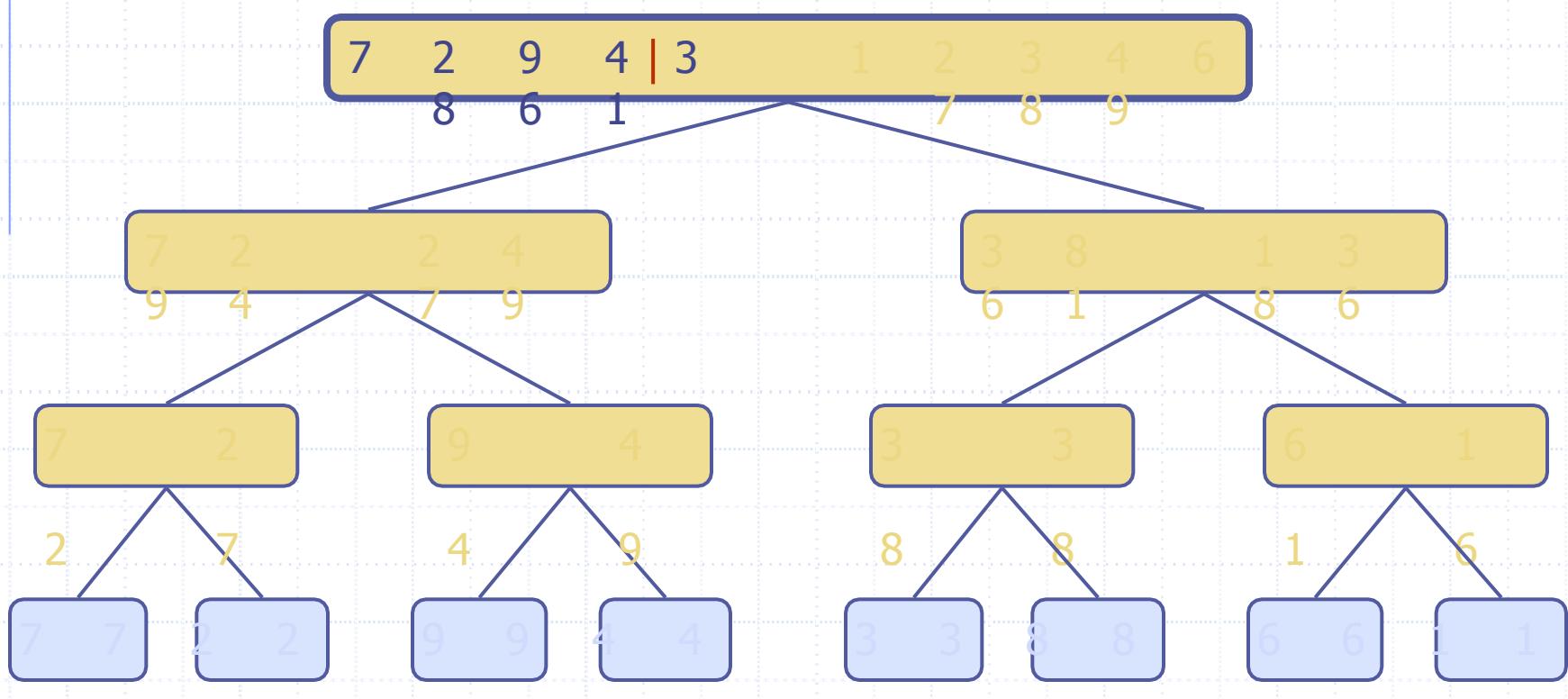
MERGE_SORT(arr, start, end)

```
if start < end
    set mid = (start + end)/2
    MERGE_SORT(arr, start, mid)
    MERGE_SORT(arr, mid + 1,
               end)
    MERGE (arr, start, mid, end)
end of if

END MERGE_SORT
```

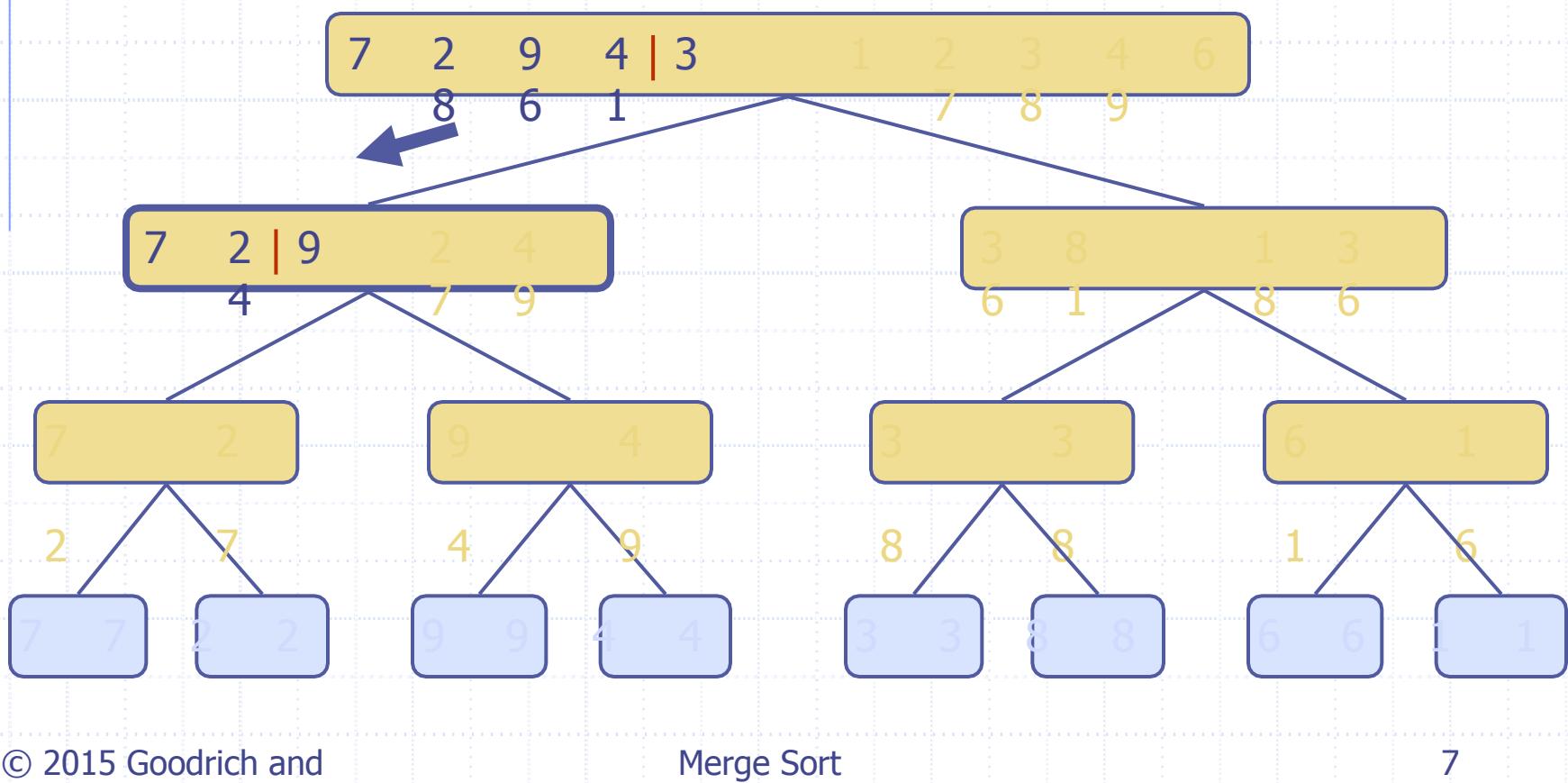
Execution Example

Partition



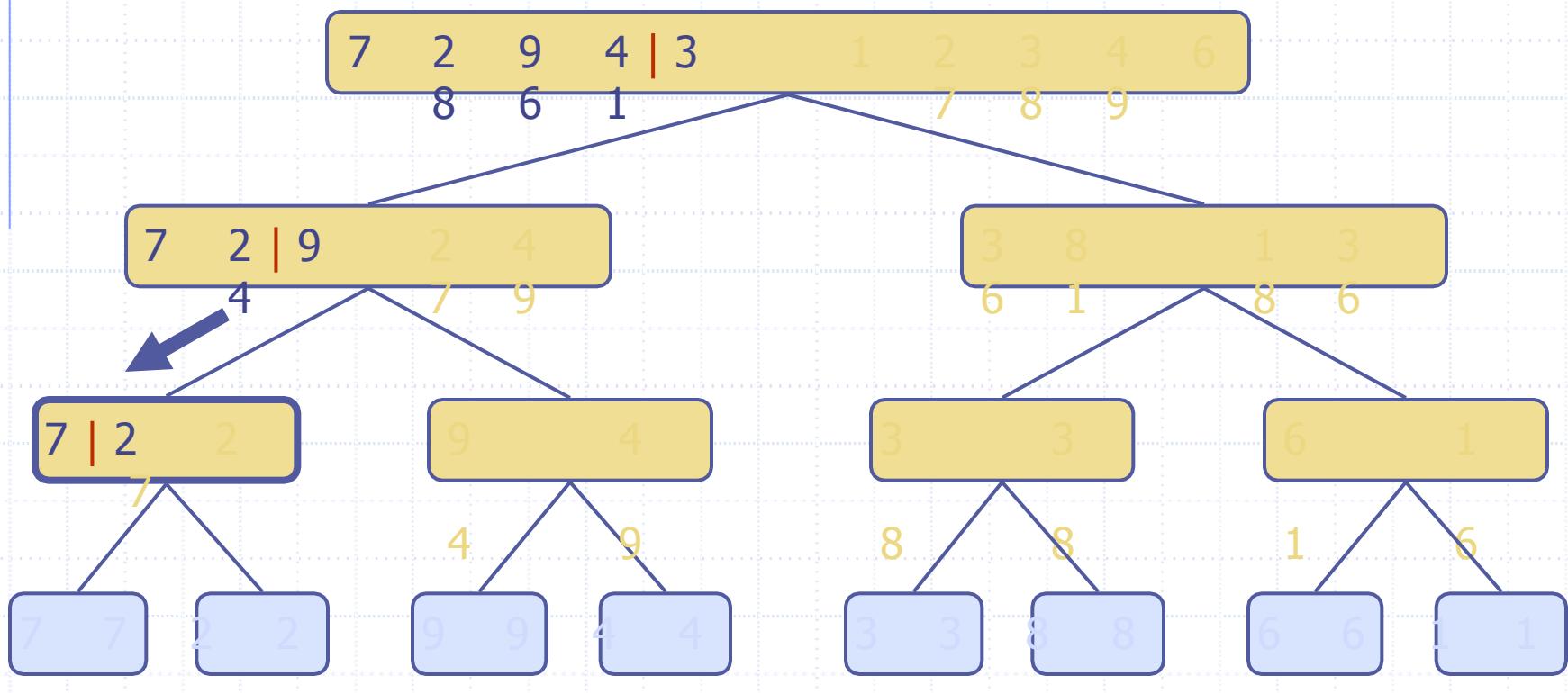
Execution Example

(cont.) Recursive call, partition



Execution Example

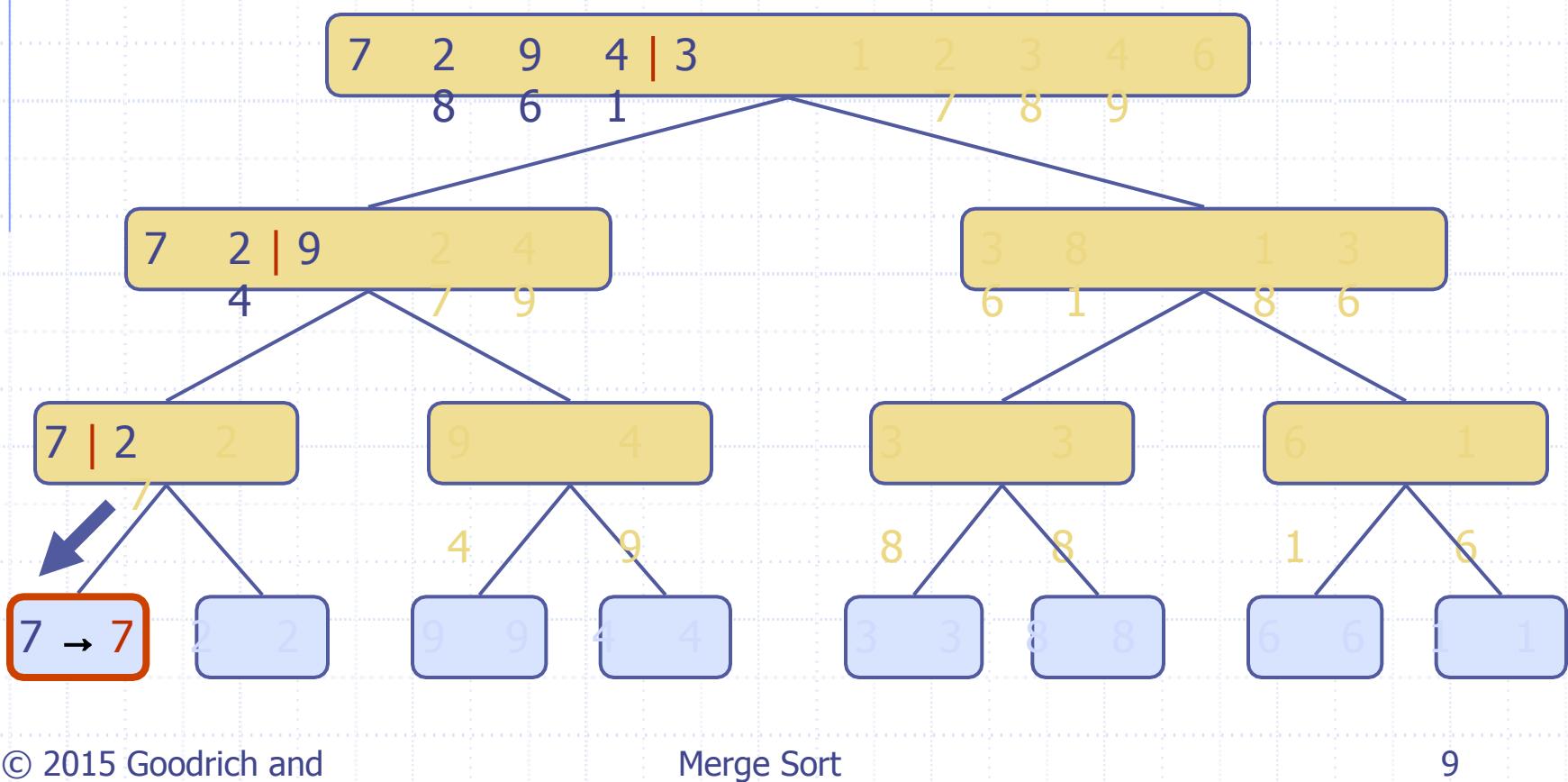
(cont.) Recursive call, partition



Execution Example

(cont.)

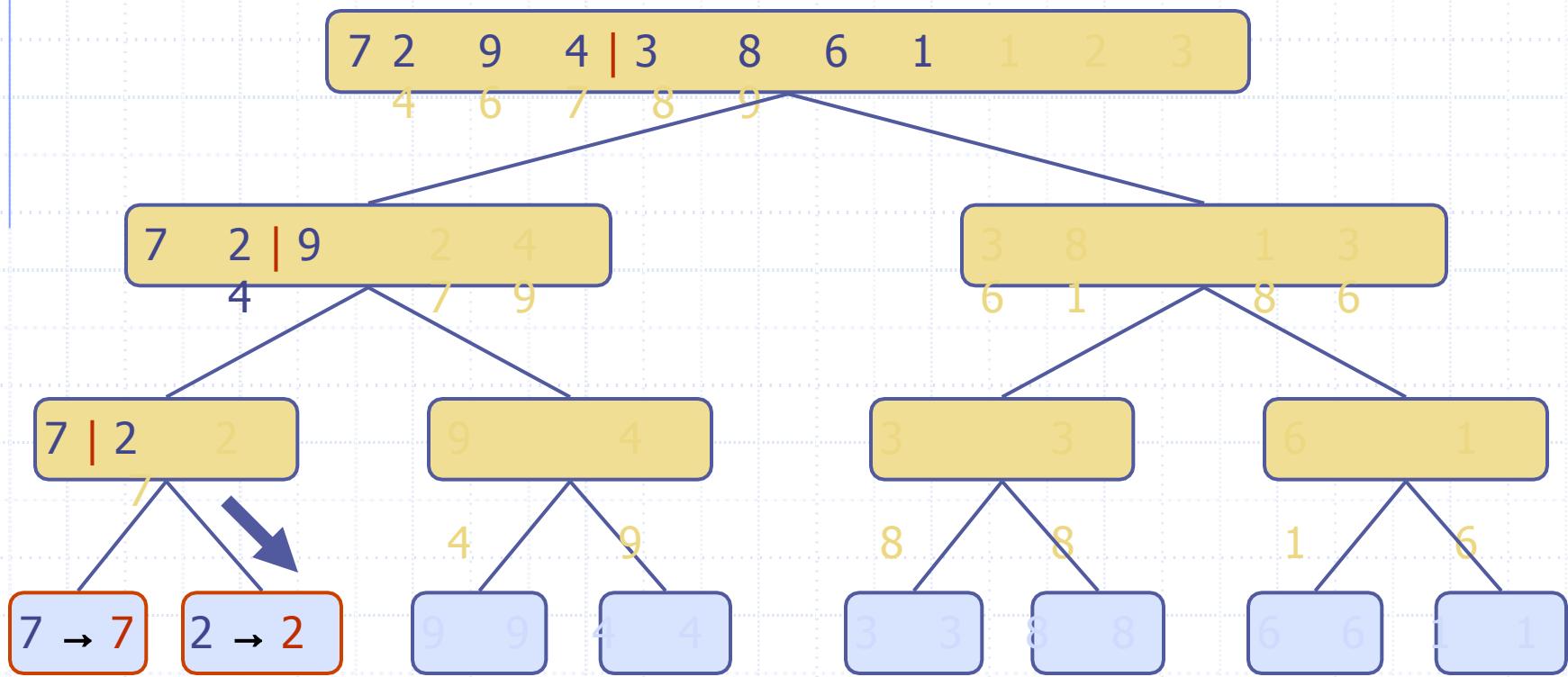
Recursive call, base case



Execution Example

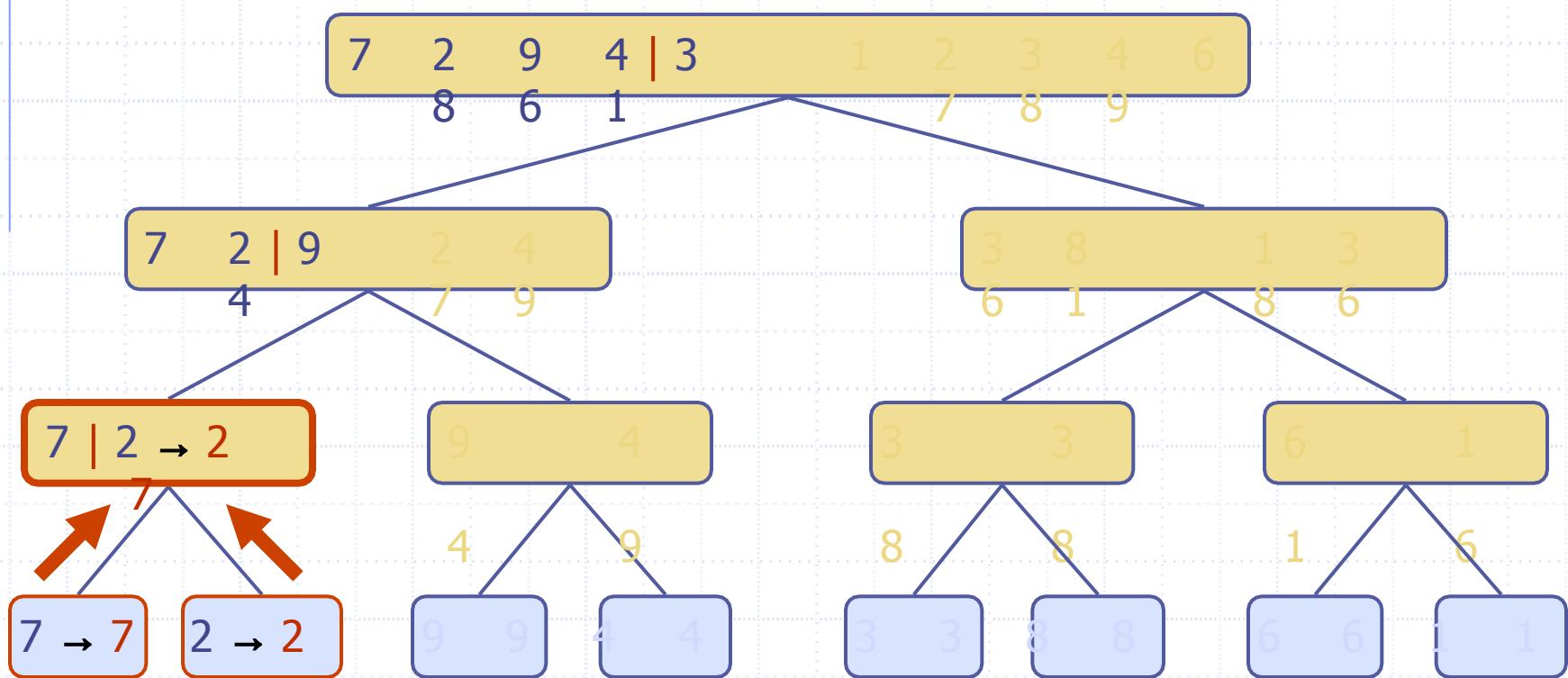
(cont.)

Recursive call, base case



Execution Example

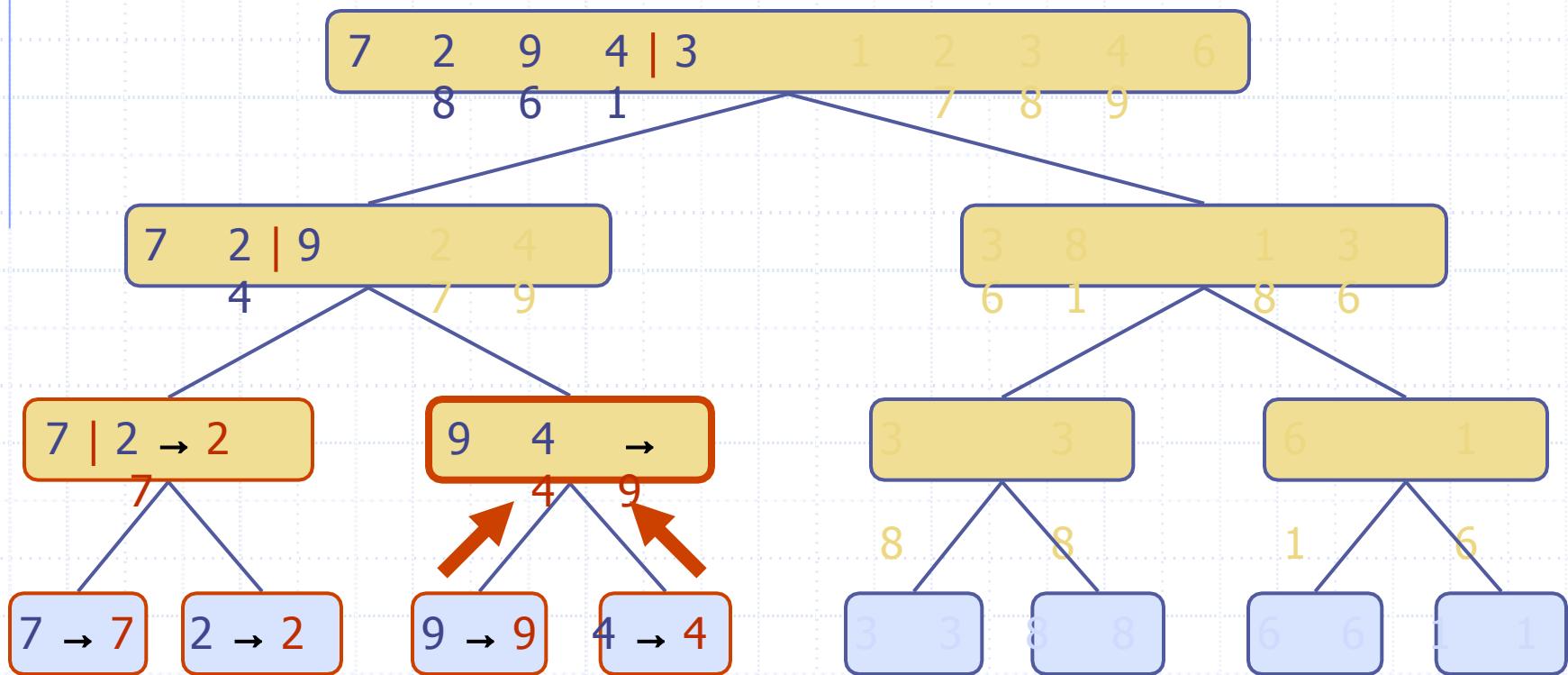
(cont.)
Merge



Execution Example

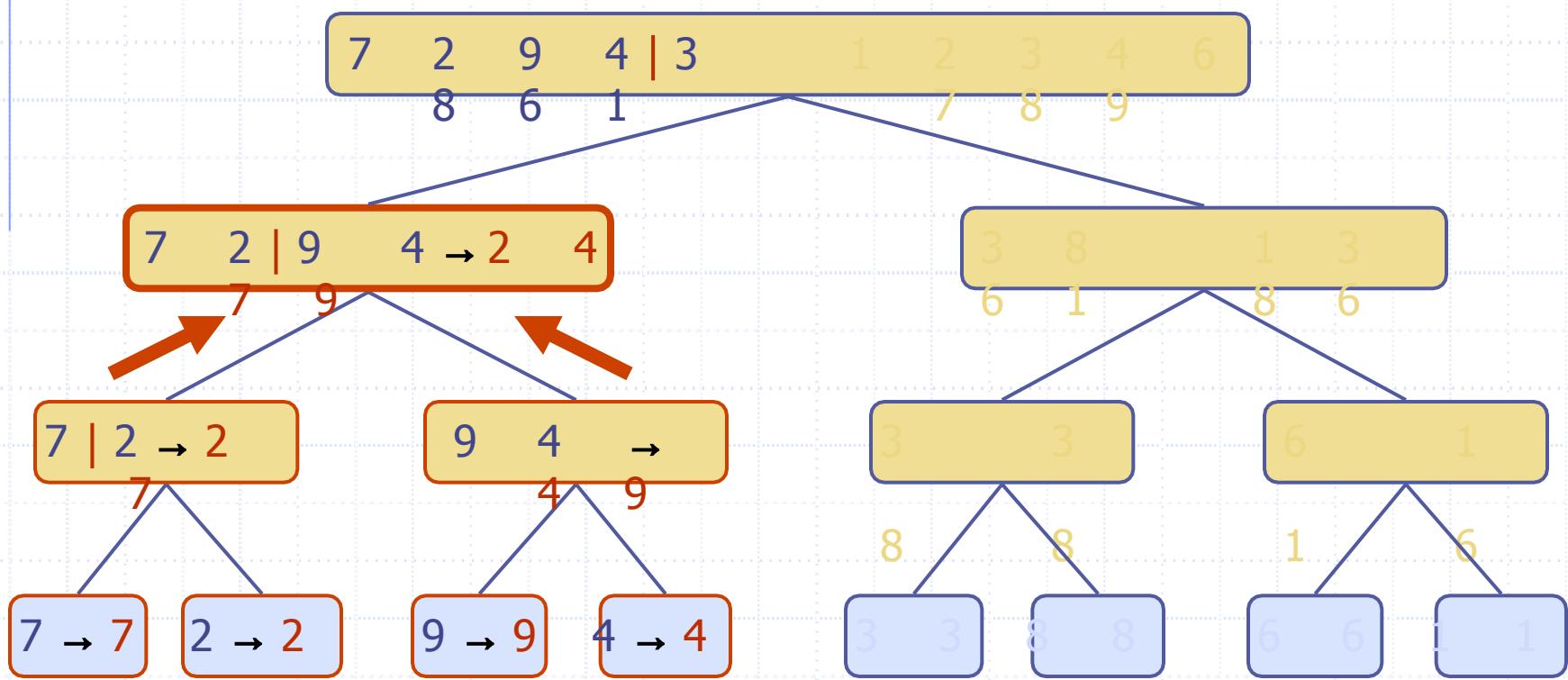
(cont.)

Recursive call, ..., base case, merge



Execution Example

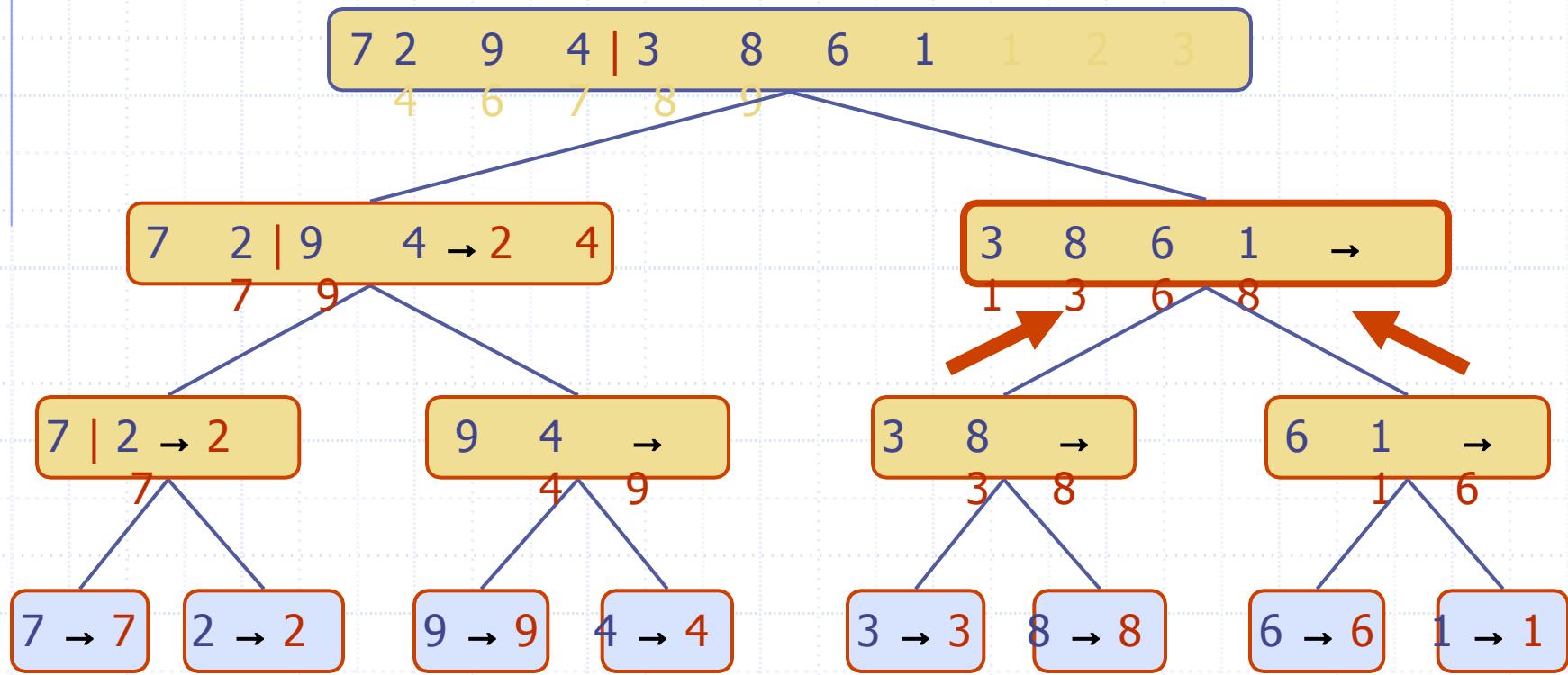
(cont.)
Merge



Execution Example

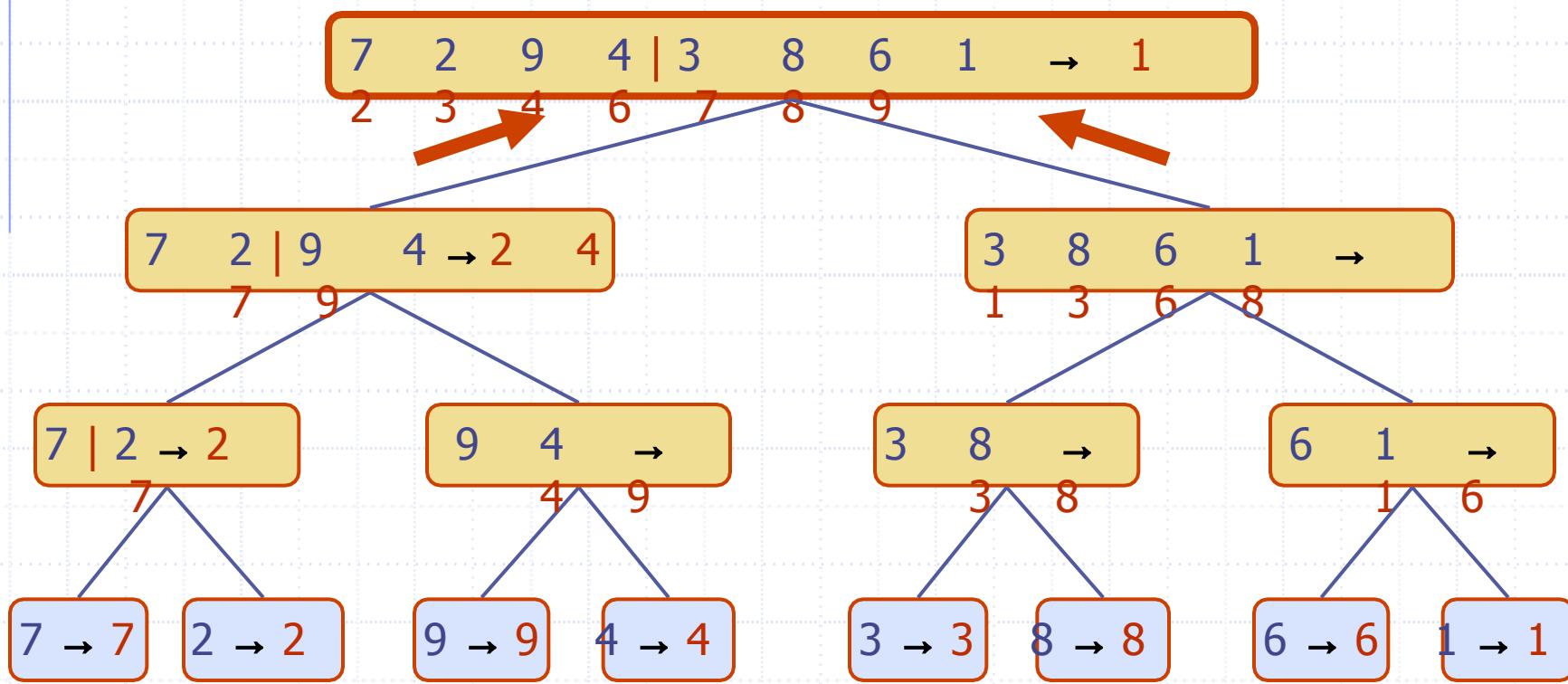
(cont.)

Recursive call, ..., merge, merge



Execution Example

(cont.)
Merge



Analysis of Merge-Sort

◆ The height h of the merge-sort tree is $O(\log n)$

- at each recursive call we divide in half the sequence,

◆ The overall amount or work done at the nodes of depth i is $O(n)$

- we partition and merge 2^i sequences of size $n/2^i$
- we make 2^{i+1} recursive calls

◆ Thus, the total running time of merge-sort is $O(n \log n)$

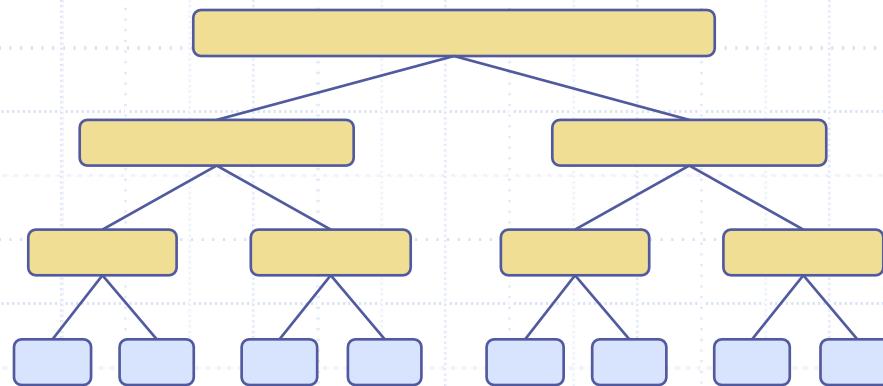
depth	#seqs	size
-------	-------	------

0	1	n
---	---	-----

1	2	$n/2$
---	---	-------

i	$n/2^i$	$\frac{n}{2^i}$
-----	---------	-----------------

...
-----	-----	-----



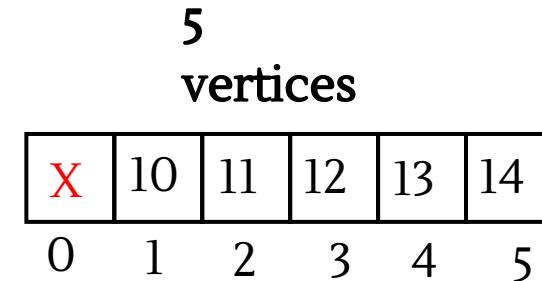
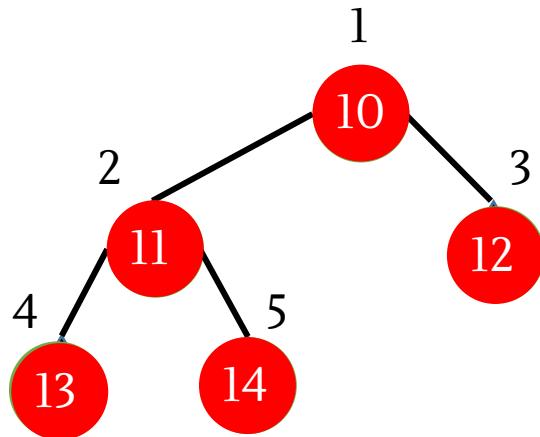
Merge Sort

Heap Sort & Priority Queue

Properties of a Complete Binary Tree

- Must be filled from left to right
- No missing elements in an array representation

Representation of Complete Binary Tree



Left child of i^{th} node is:

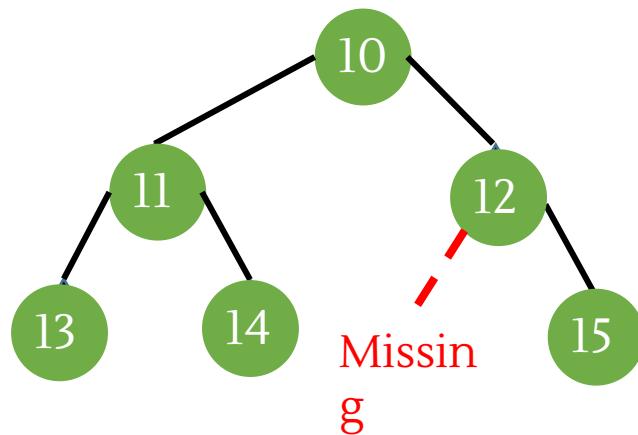
$2*i$
Right child of i^{th} node is: $2*i + 1$

Parent of i^{th} node is:

$\text{floor}(i/2)$

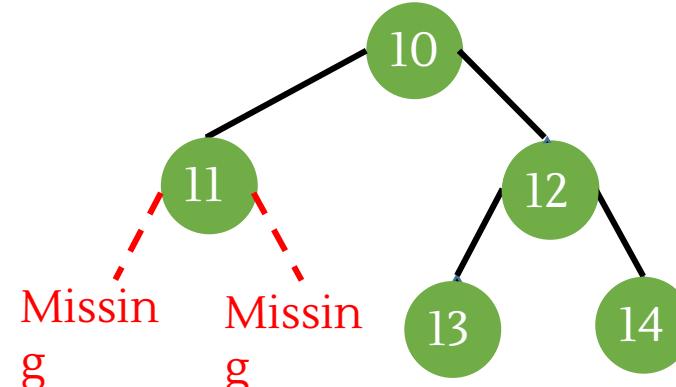
What is a Complete Binary Tree?

- ☐ Filled from left to right



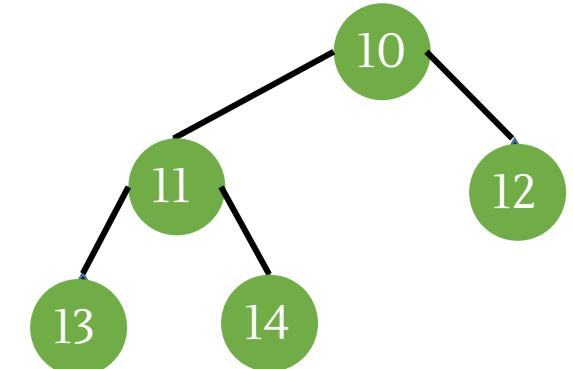
Not a Complete Binary Tree

10	11	12	13	14	--	15
----	----	----	----	----	----	----



Not a Complete Binary Tree

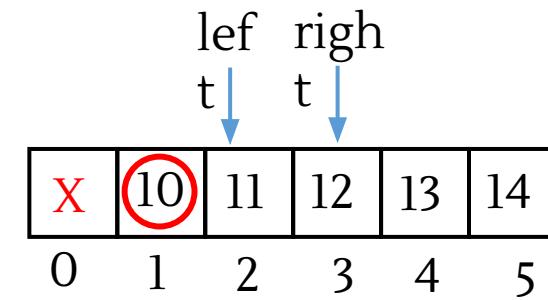
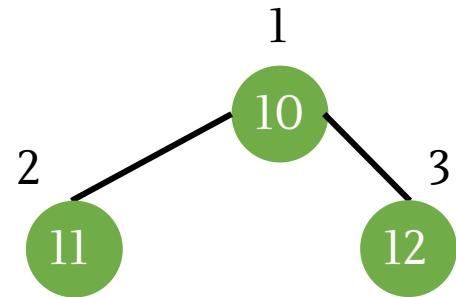
10	11	12	--	--	13	14
----	----	----	----	----	----	----



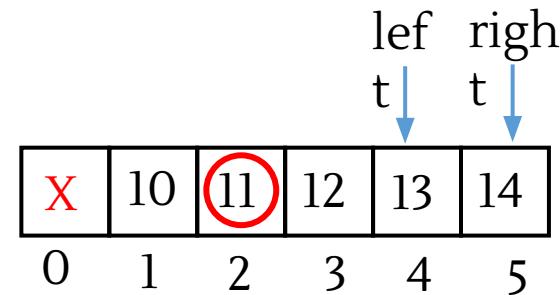
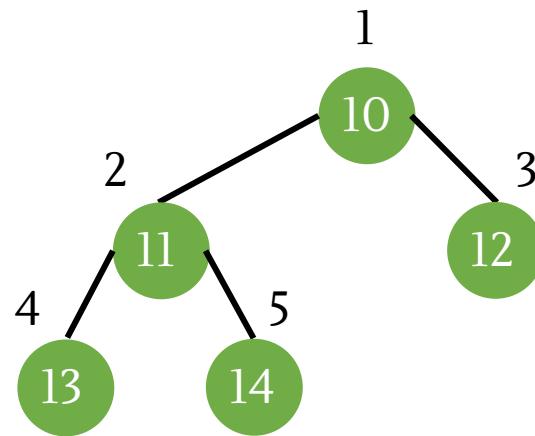
Complete Binary Tree

10	11	12	13	14
----	----	----	----	----

Representation of Complete Binary Tree From Array



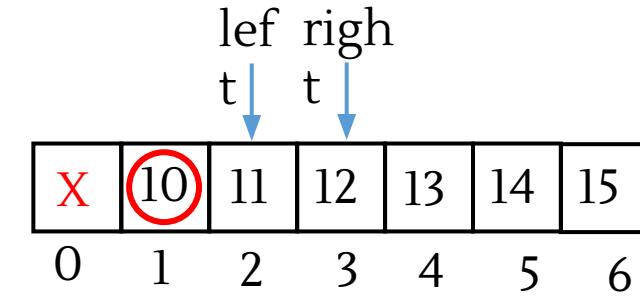
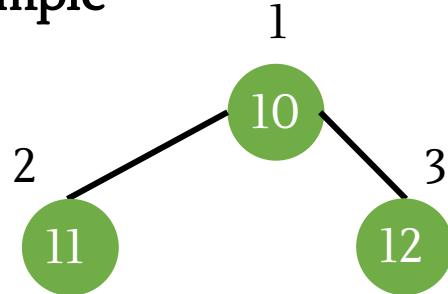
Representation of Complete Binary Tree From Array



- How many vertices need to be explored to generate the complete binary tree?
- Because only 2 vertices have child (**internal vertices**)
- Rest ($5-2$) = 3 vertices have no child (**external vertices**)
- Summary:** If a complete binary tree having n vertices, it's first floor($n/2$) vertices are **internal**, rest are **leaf** vertices

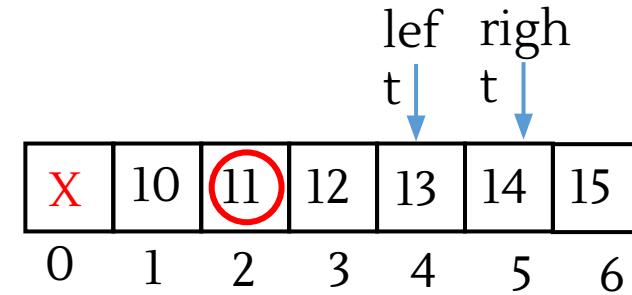
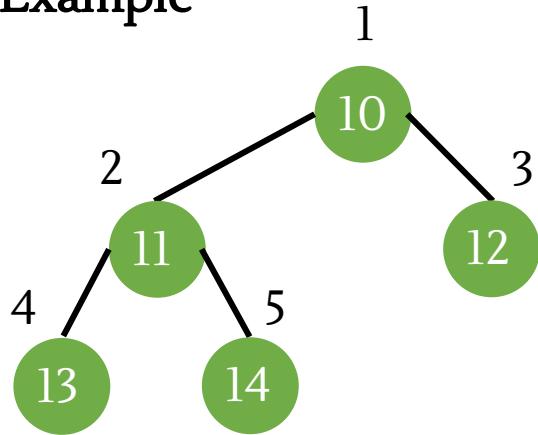
Representation of Complete Binary Tree From Array

- Another Example



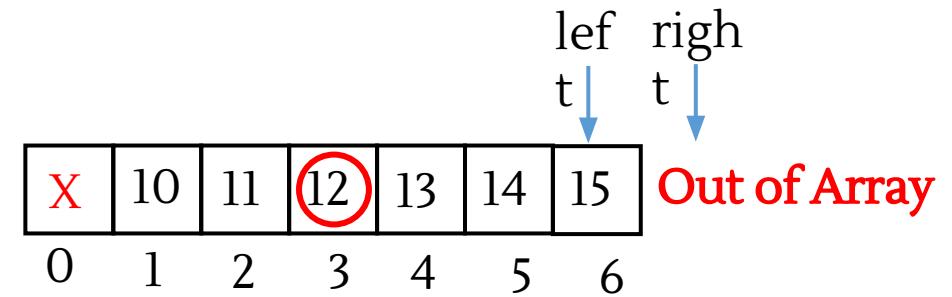
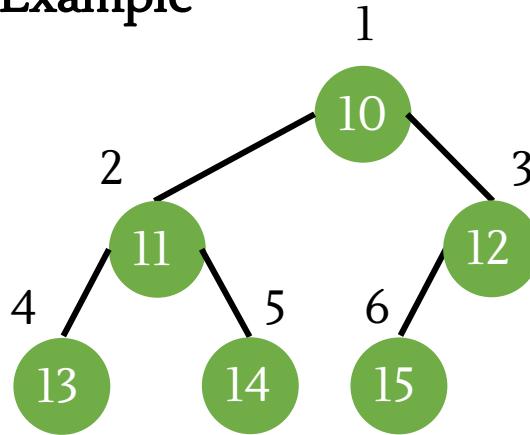
Representation of Complete Binary Tree From Array

- Another Example



Representation of Complete Binary Tree From Array

- Another Example



- How many vertices need to be explored to generate the complete binary tree?
✓
↳ $\frac{n}{2}$
- Maintains the same formula for **even** value
↳ $\frac{n}{2}$
- First $\text{floor}(\frac{n}{2})$ vertices are *internal* and rest vertices are *external*(leaf)

Heap

Max Heap

- parent \geq left child && parent \geq right
- All the sub trees maintain the ^{child} same
- Is this a Max Heap?

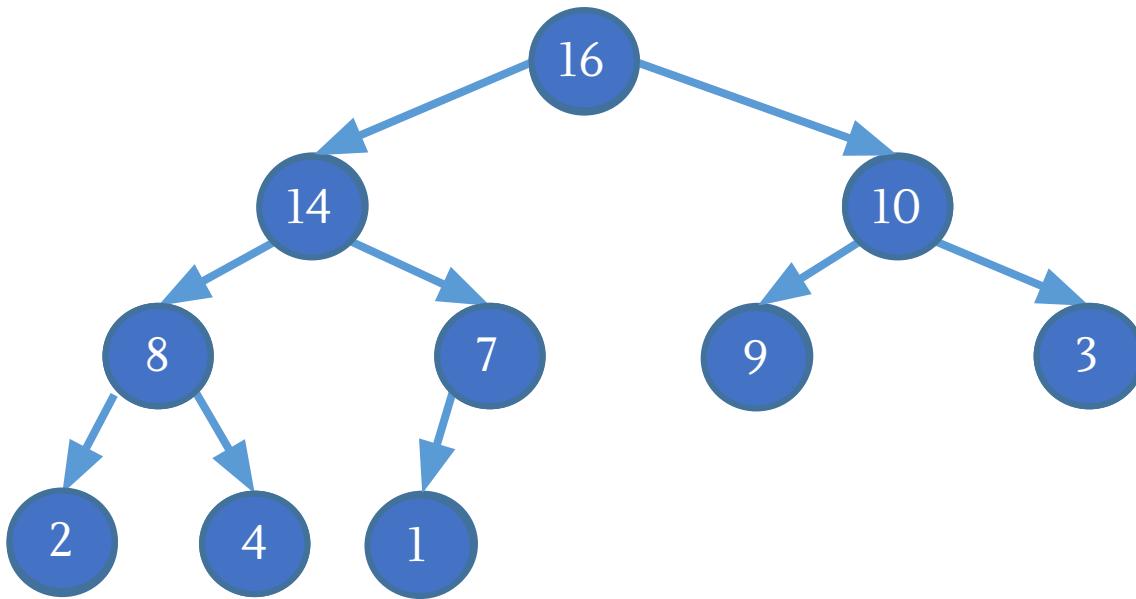
21	20	18	19	17	13	15
----	----	----	----	----	----	----

Min Heap

- parent \leq left child && parent \leq right
- All the sub trees maintain the same

Max Heap

Example



Array
Representation

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

Height of a Heap

- 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 **$O(\log n)$** .

Basic Operations of a Max-Heap

1. Max-Heap-Insert (Insertion)
2. Heap-Increase-Key (Increase the value of a current node)
3. Heap-Extract-Max (Remove the root element)
4. Heap-Maximum (Show the maximum element of the heap ↳ root)
5. Heap Sort

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.

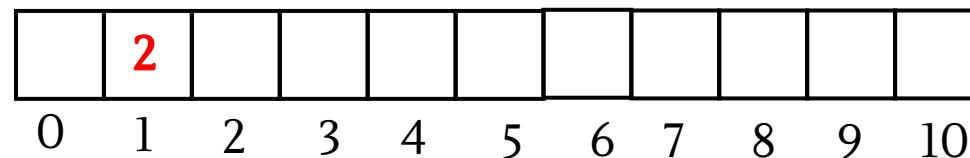
Insertion: ($O(n \log n)$ or $O(n)$)

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

Insertion

2

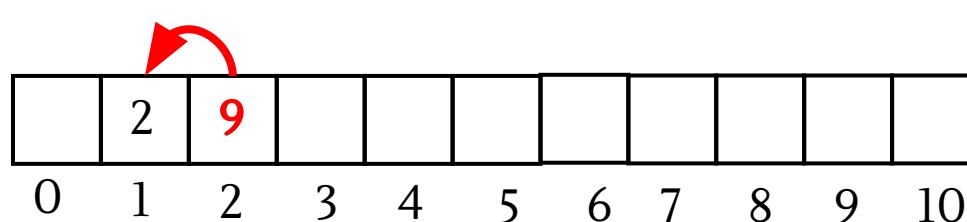
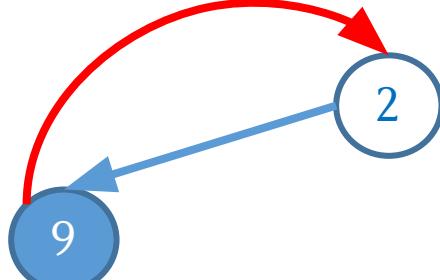
2



Insertion

2 9

Compare with
parent swa
p

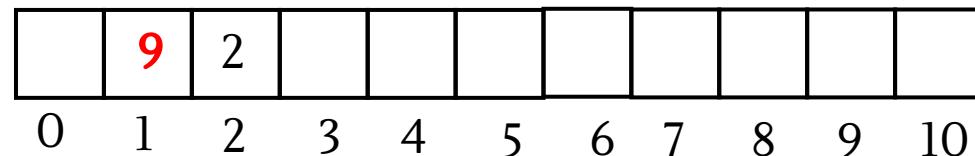
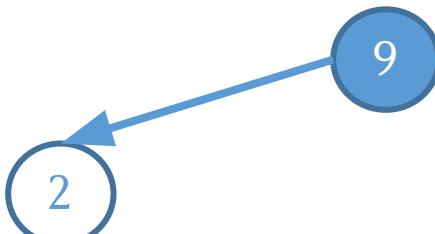


Insertion

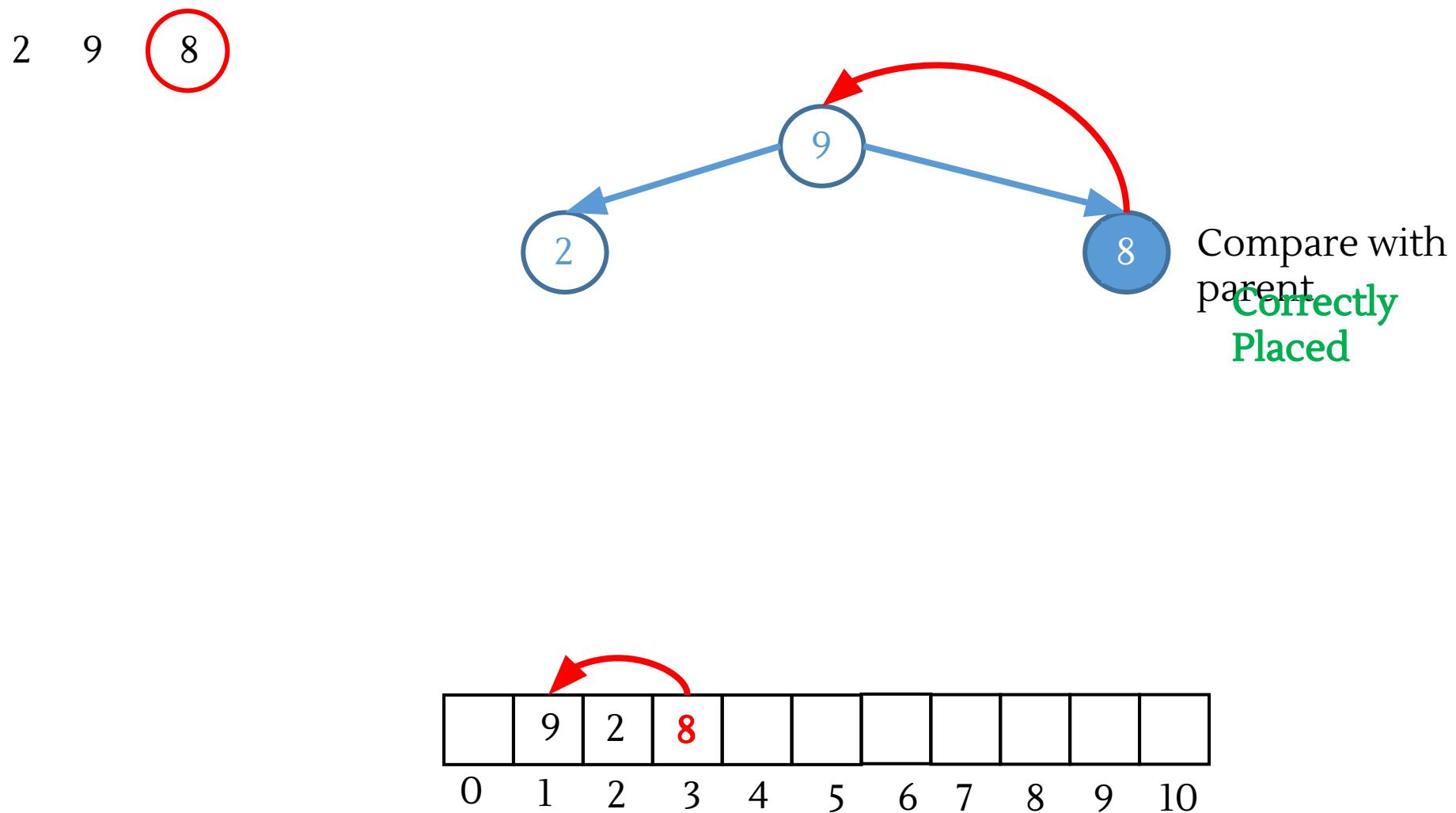
2 9

Compare with
parent swa

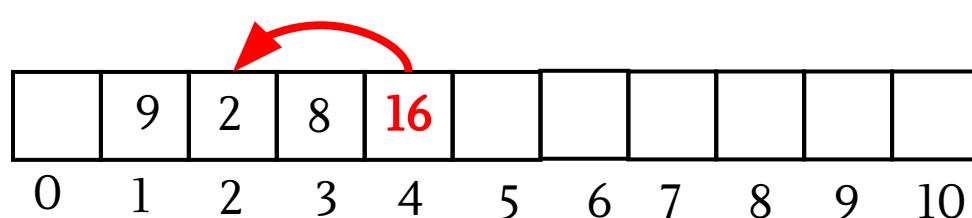
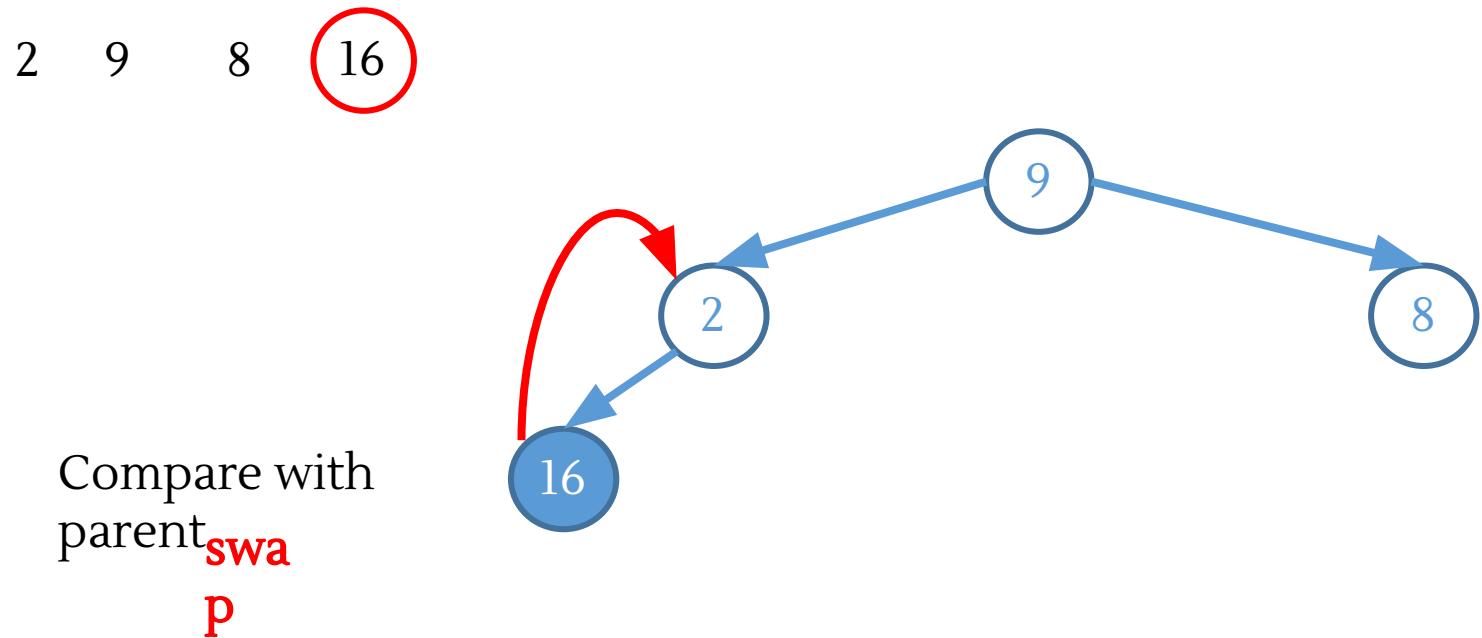
p



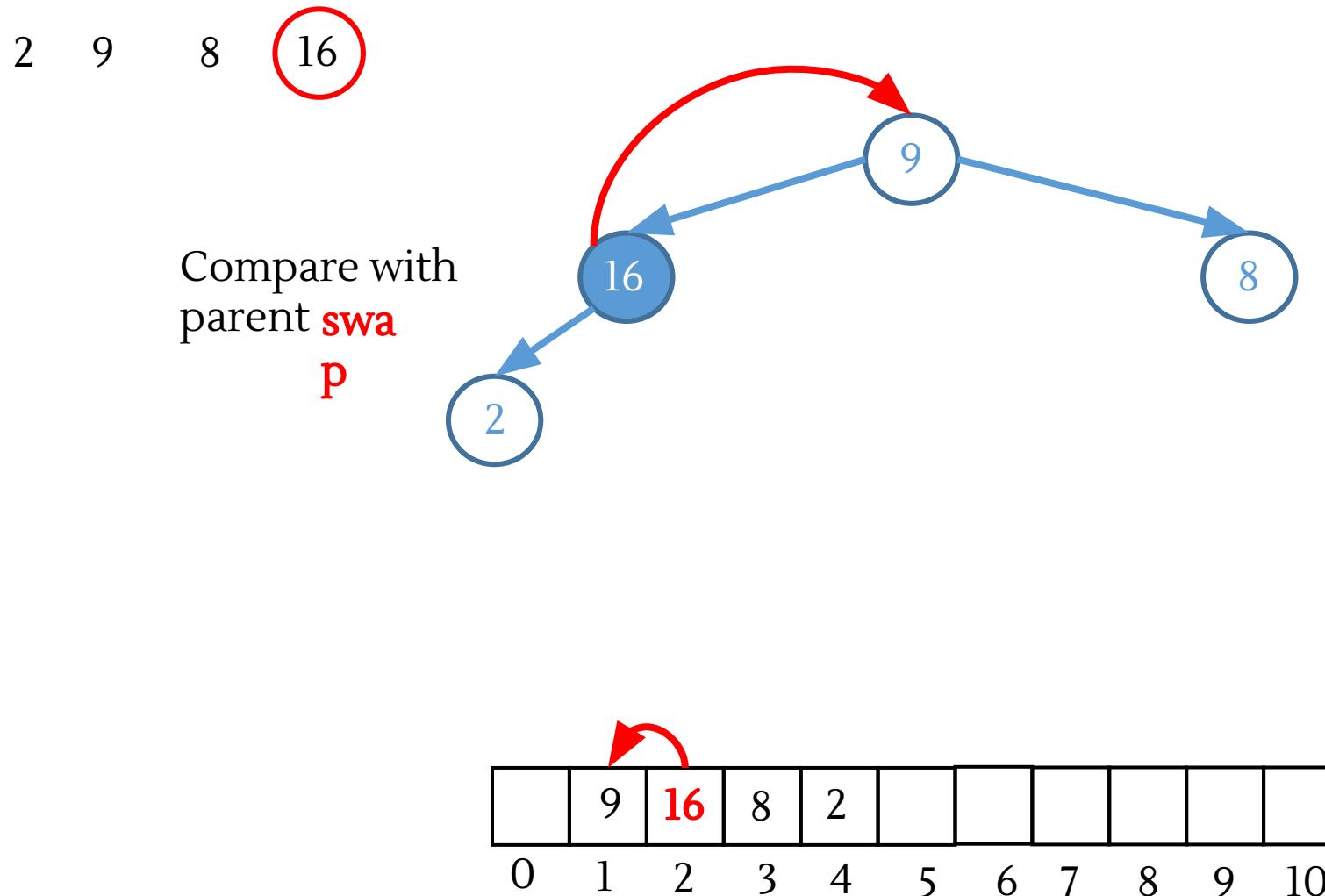
Insertion



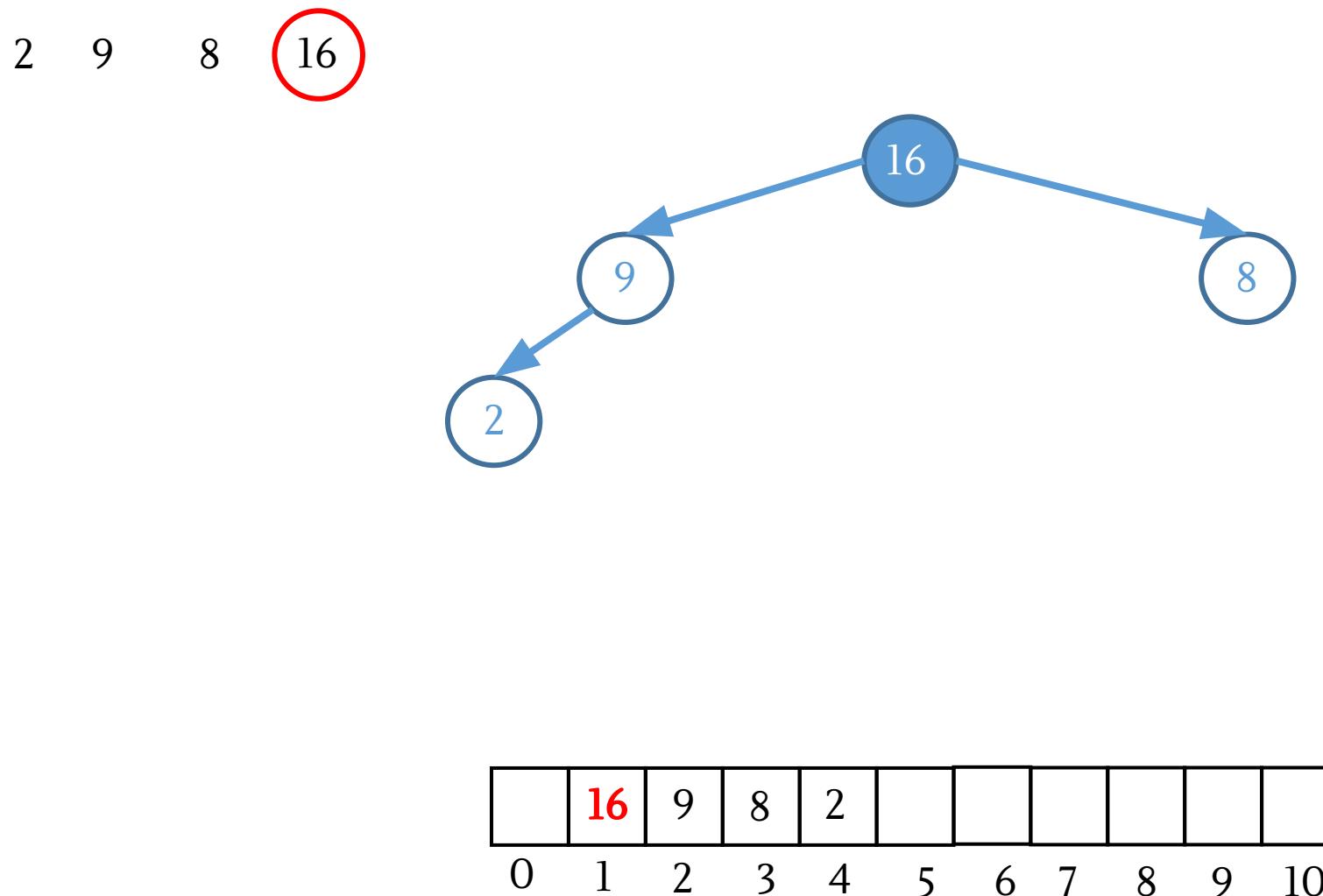
Insertion



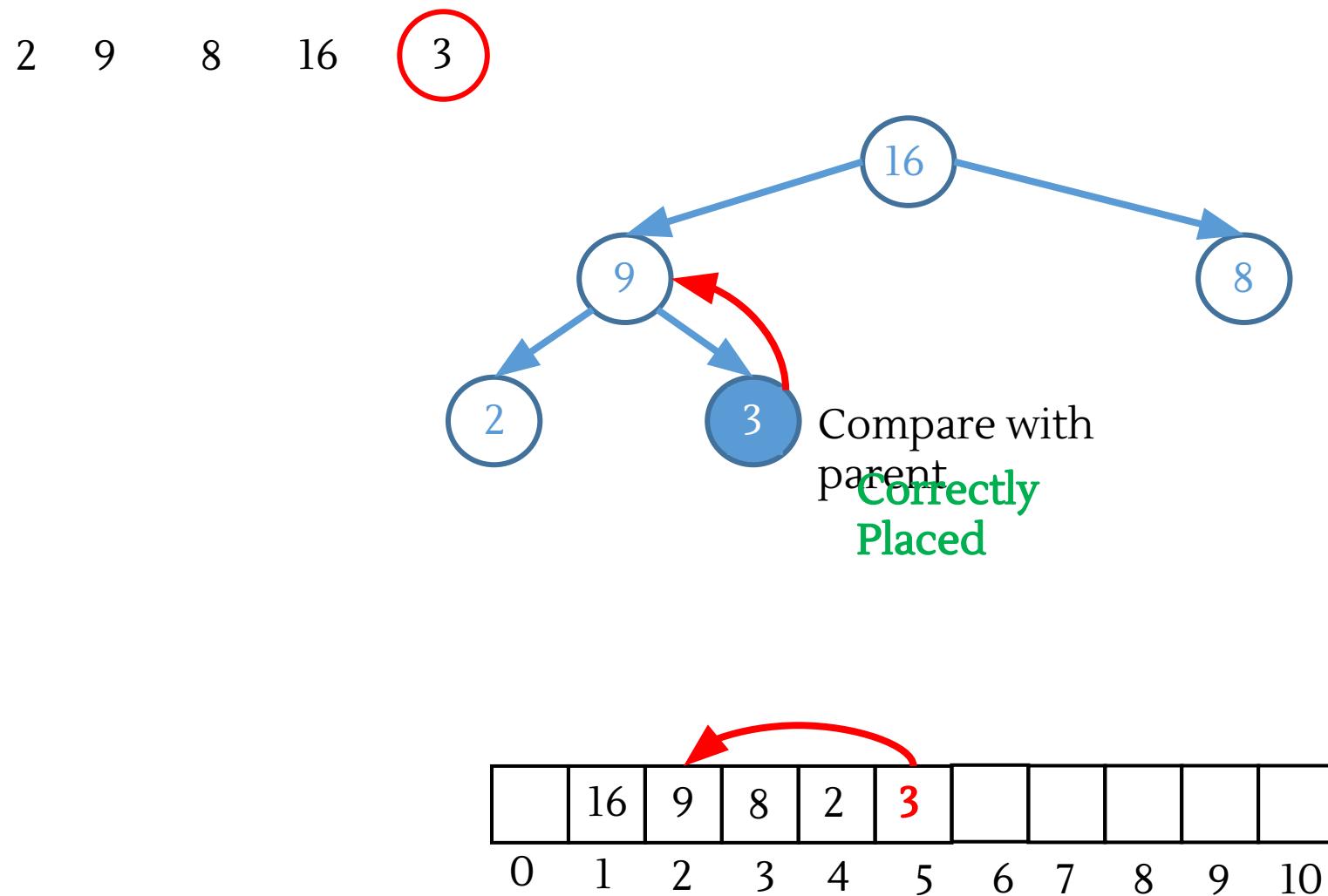
Insertion



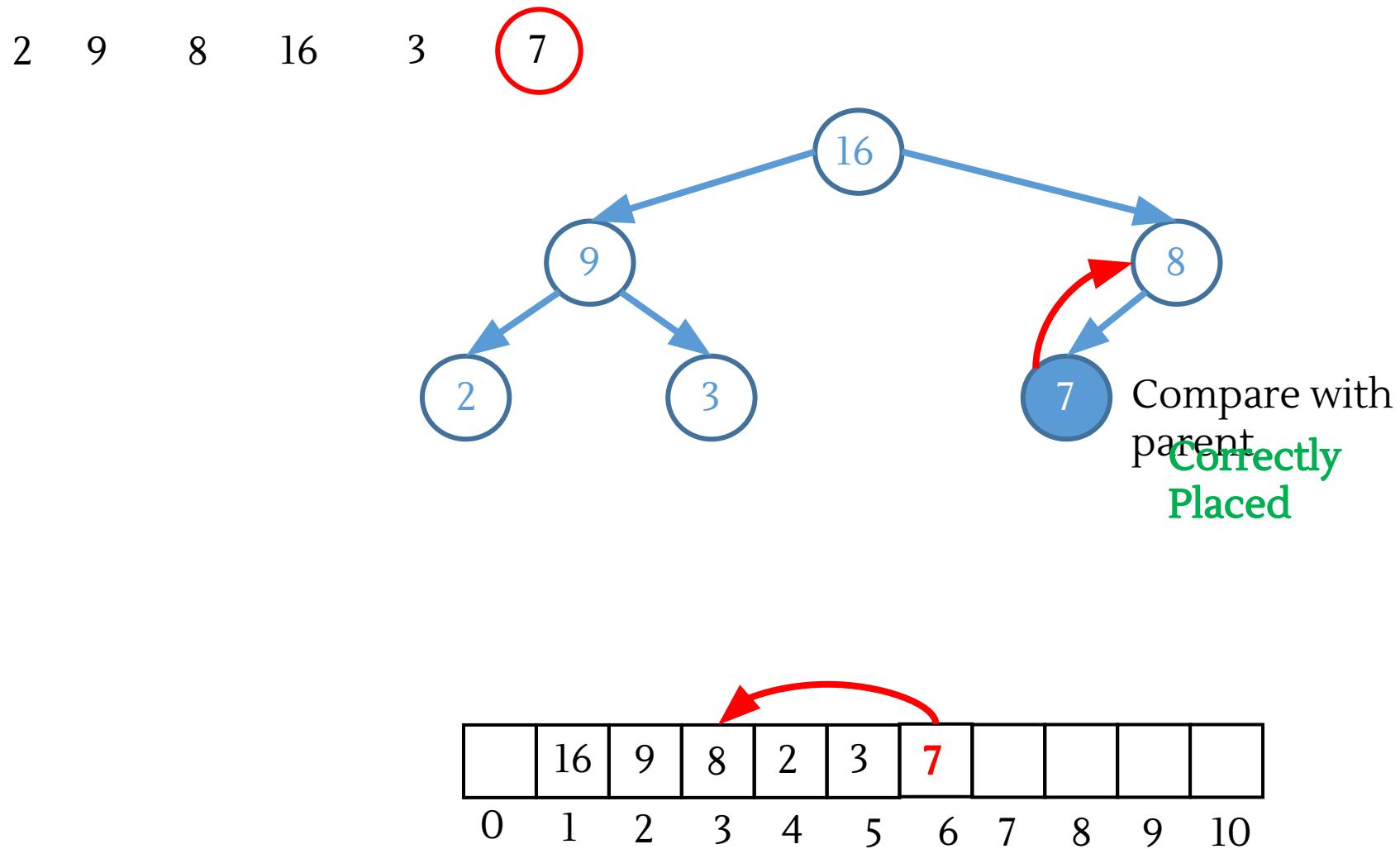
Insertion



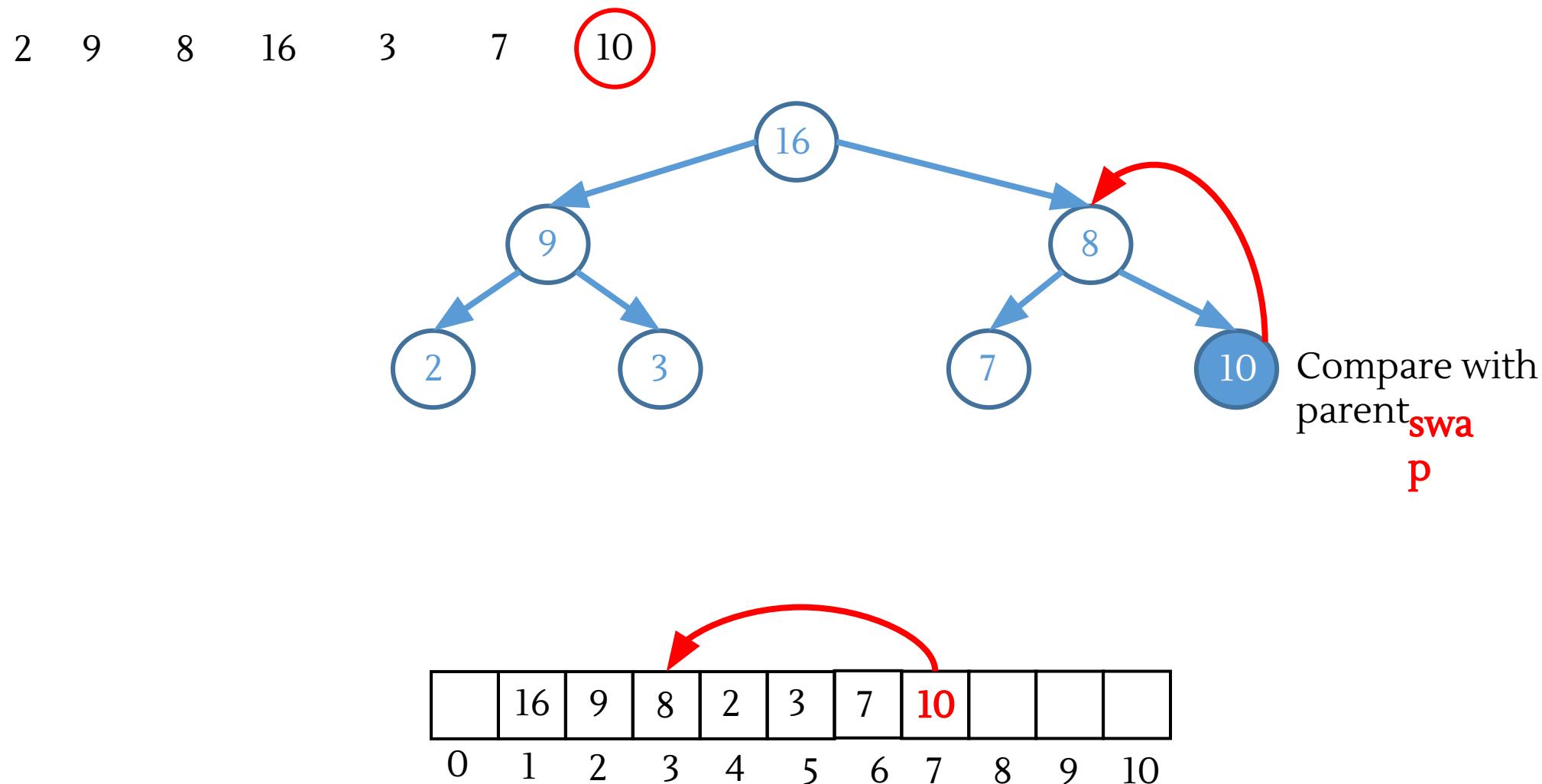
Insertion



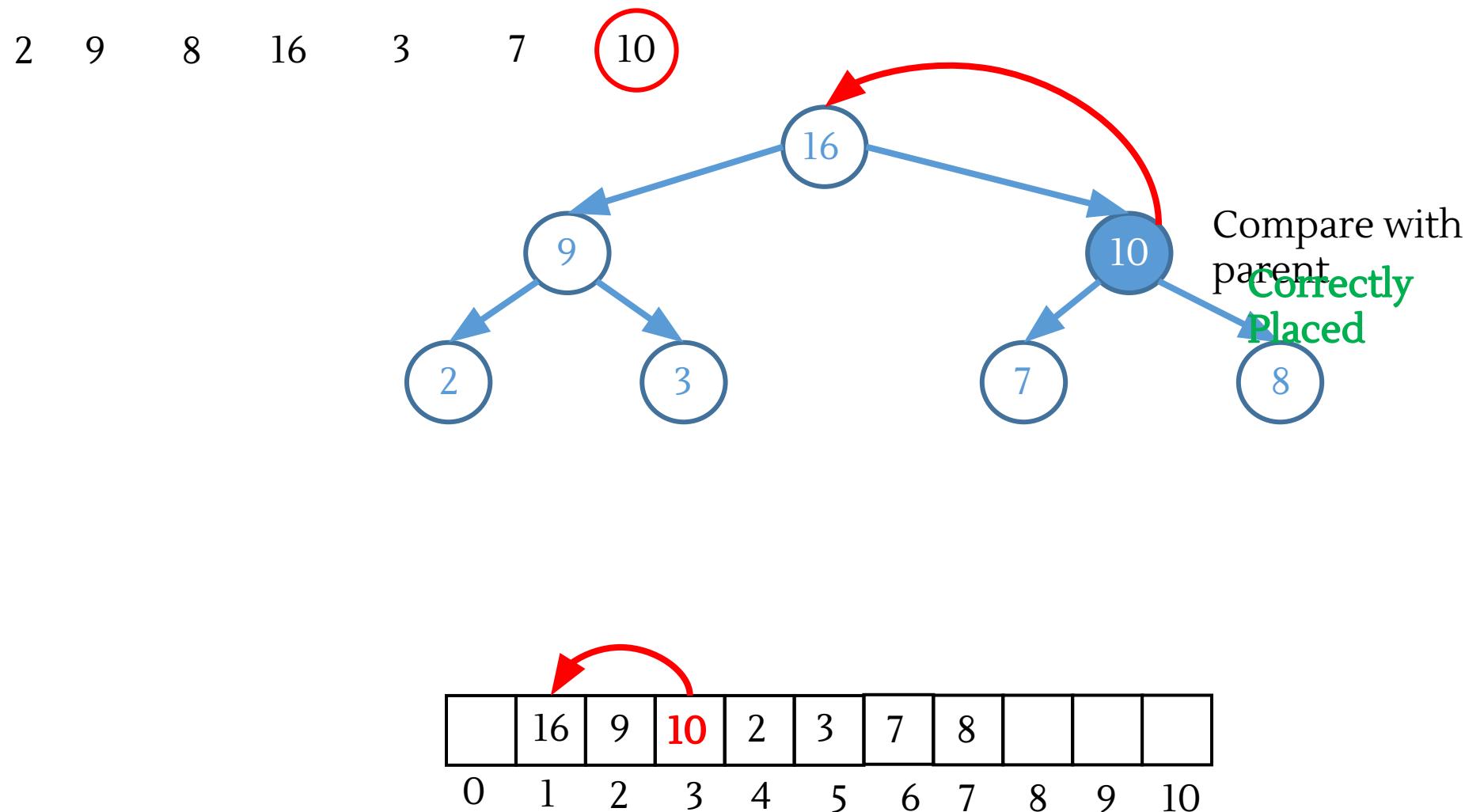
Insertion



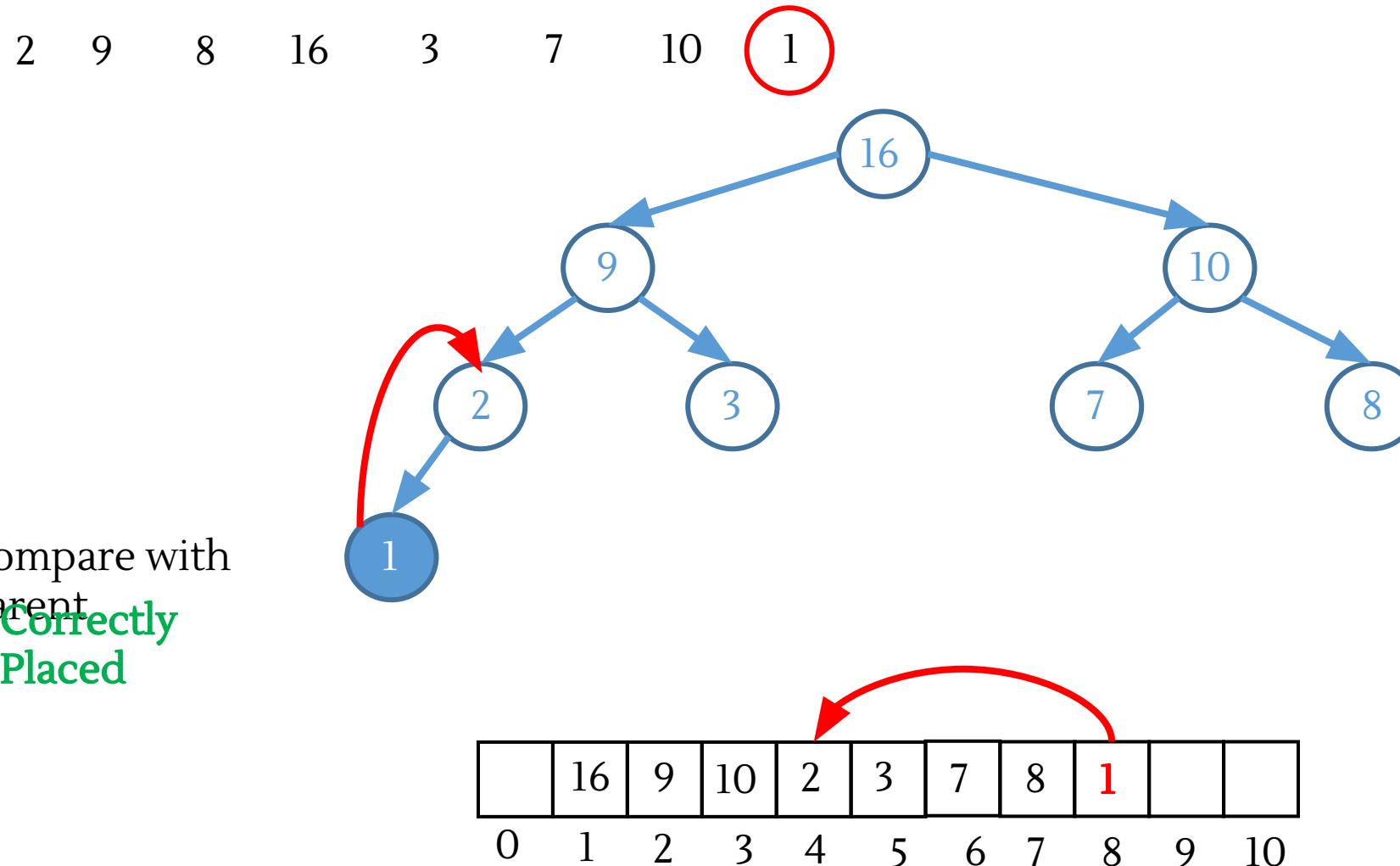
Insertion



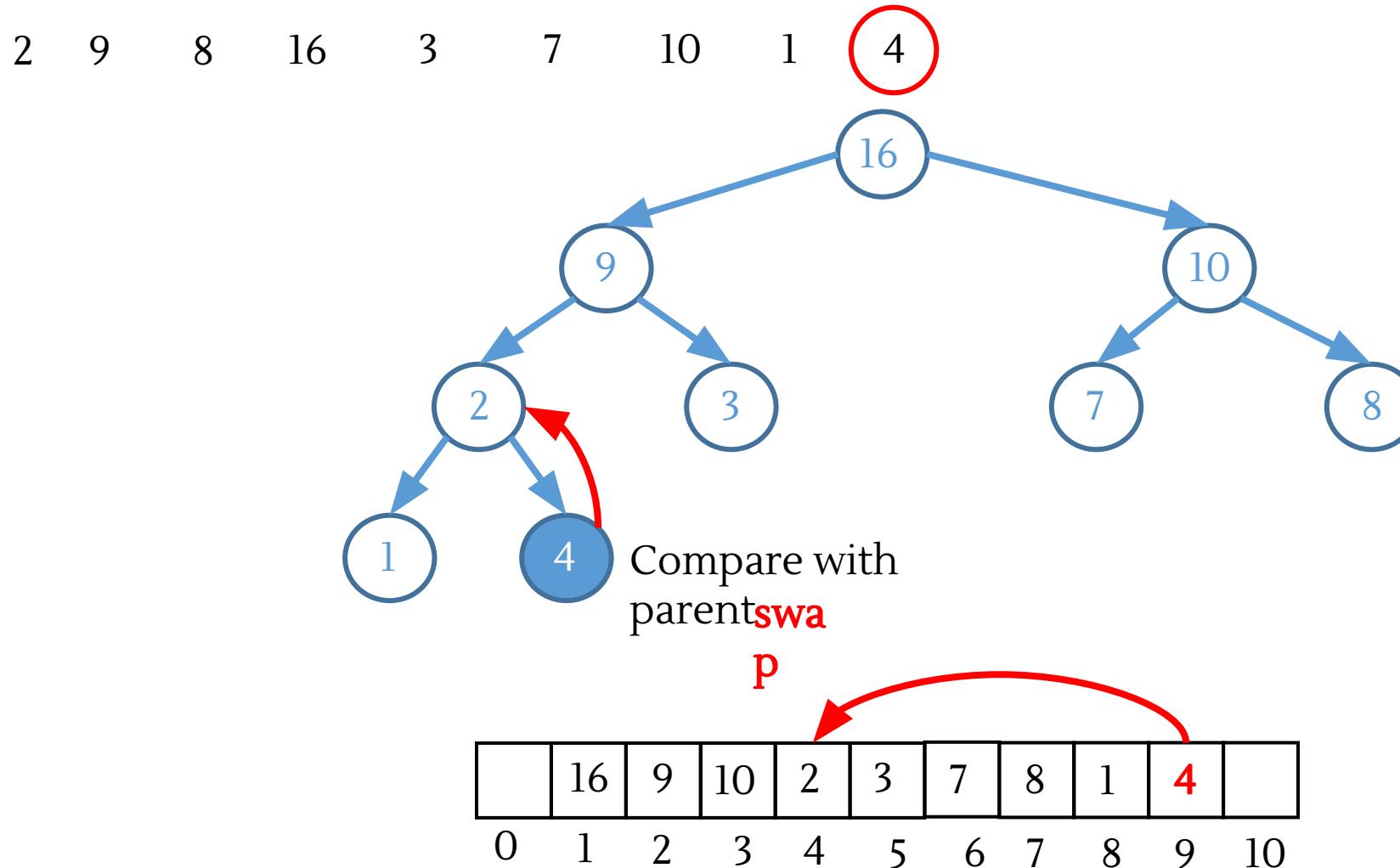
Insertion



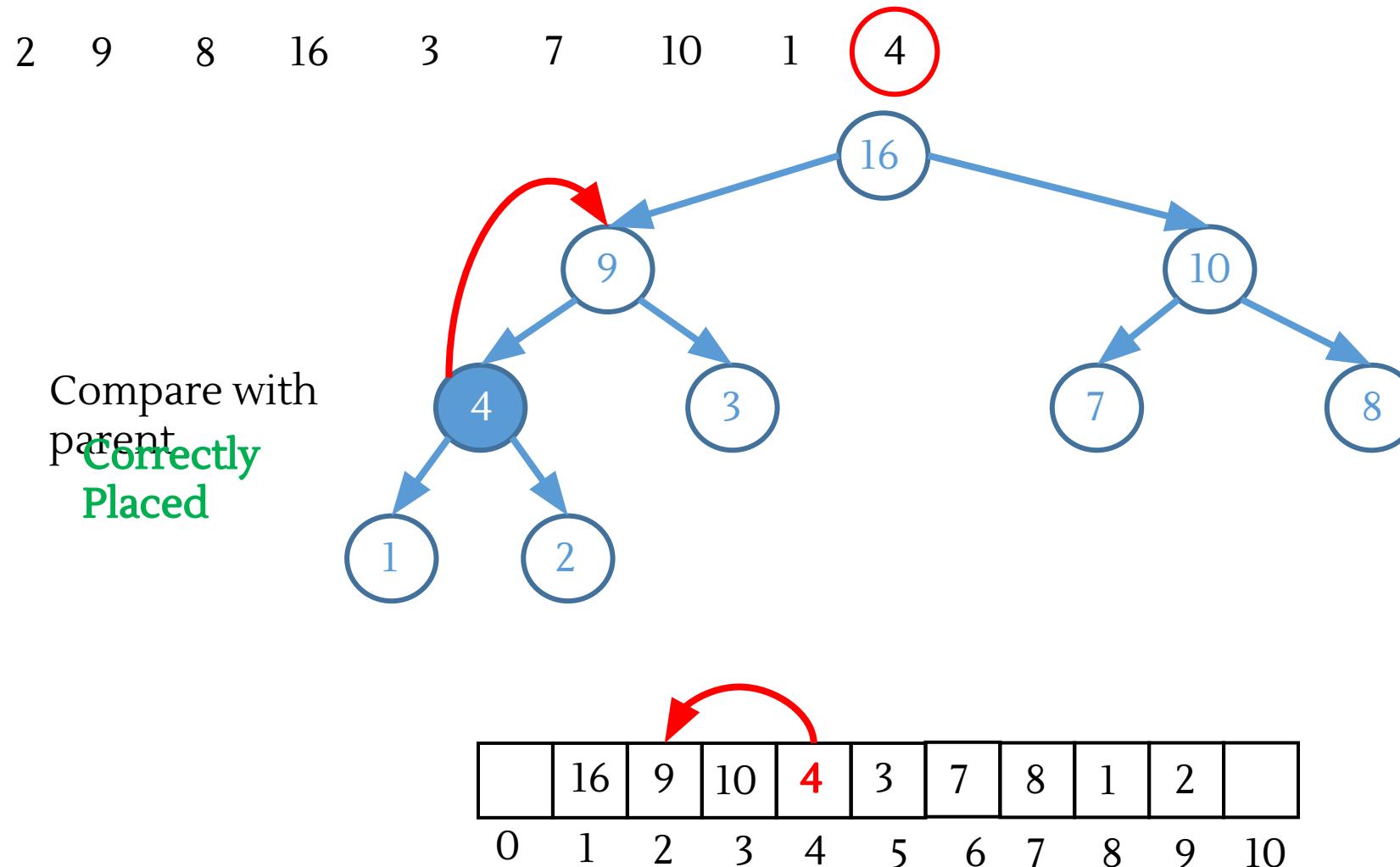
Insertion



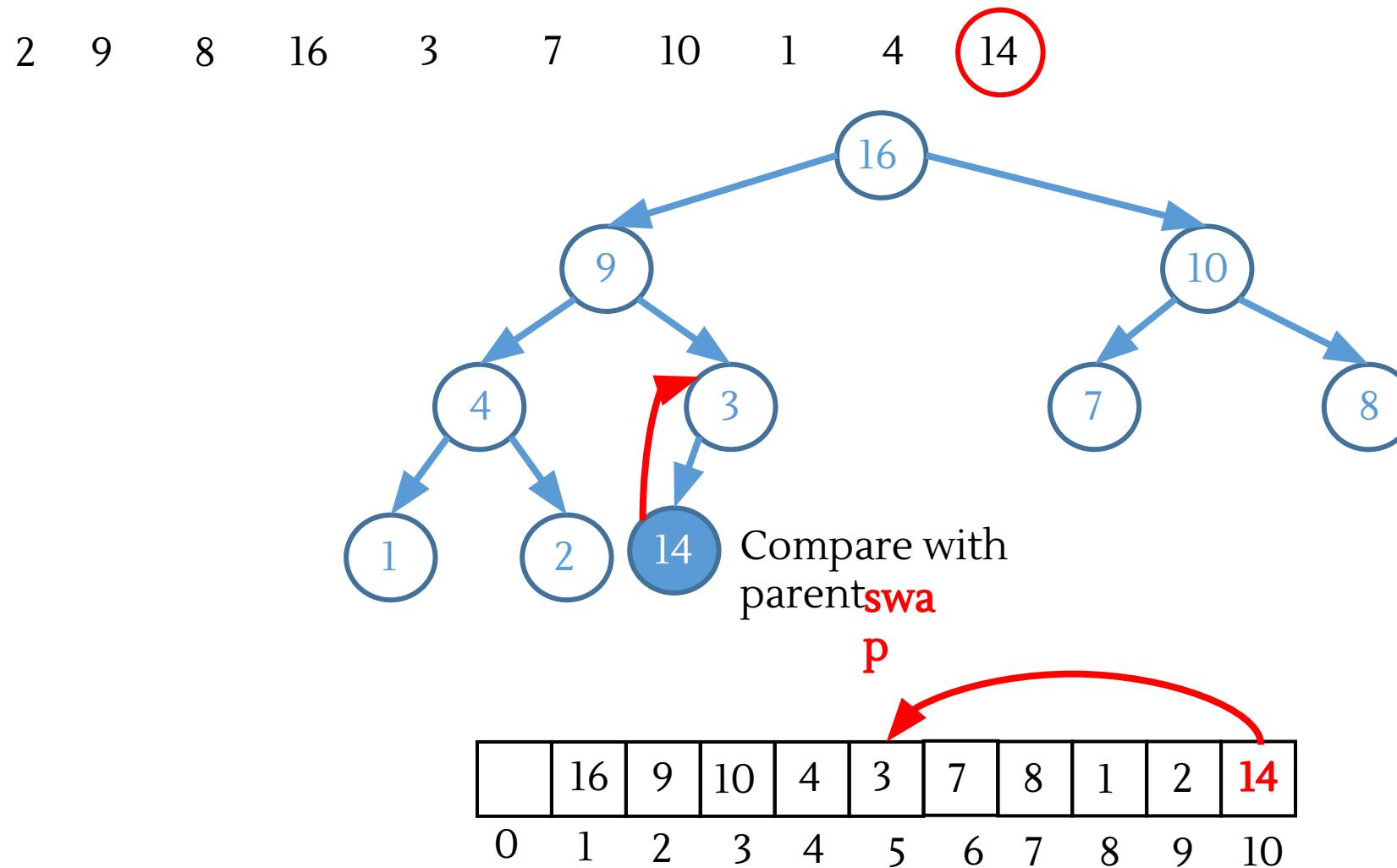
Insertion



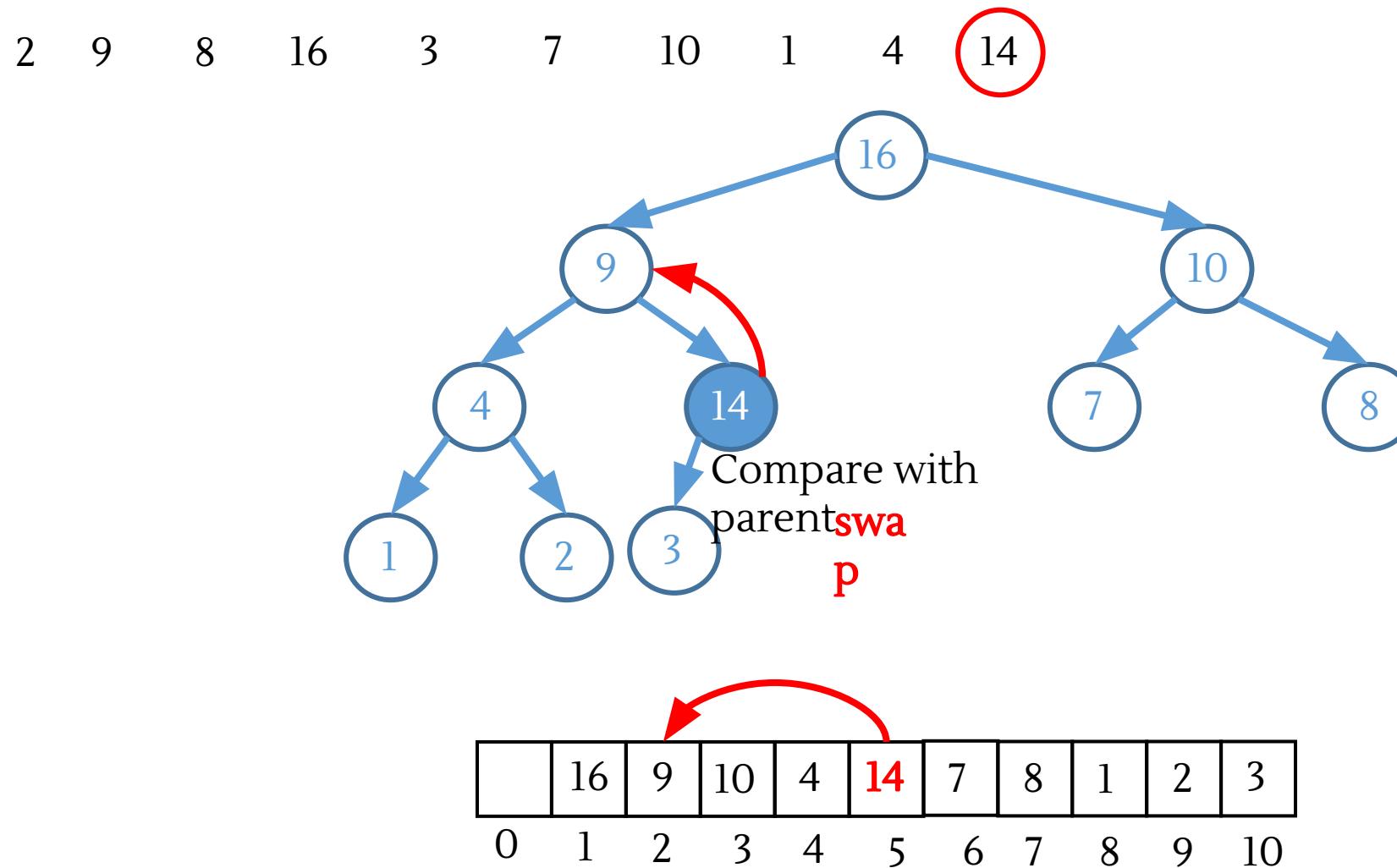
Insertion



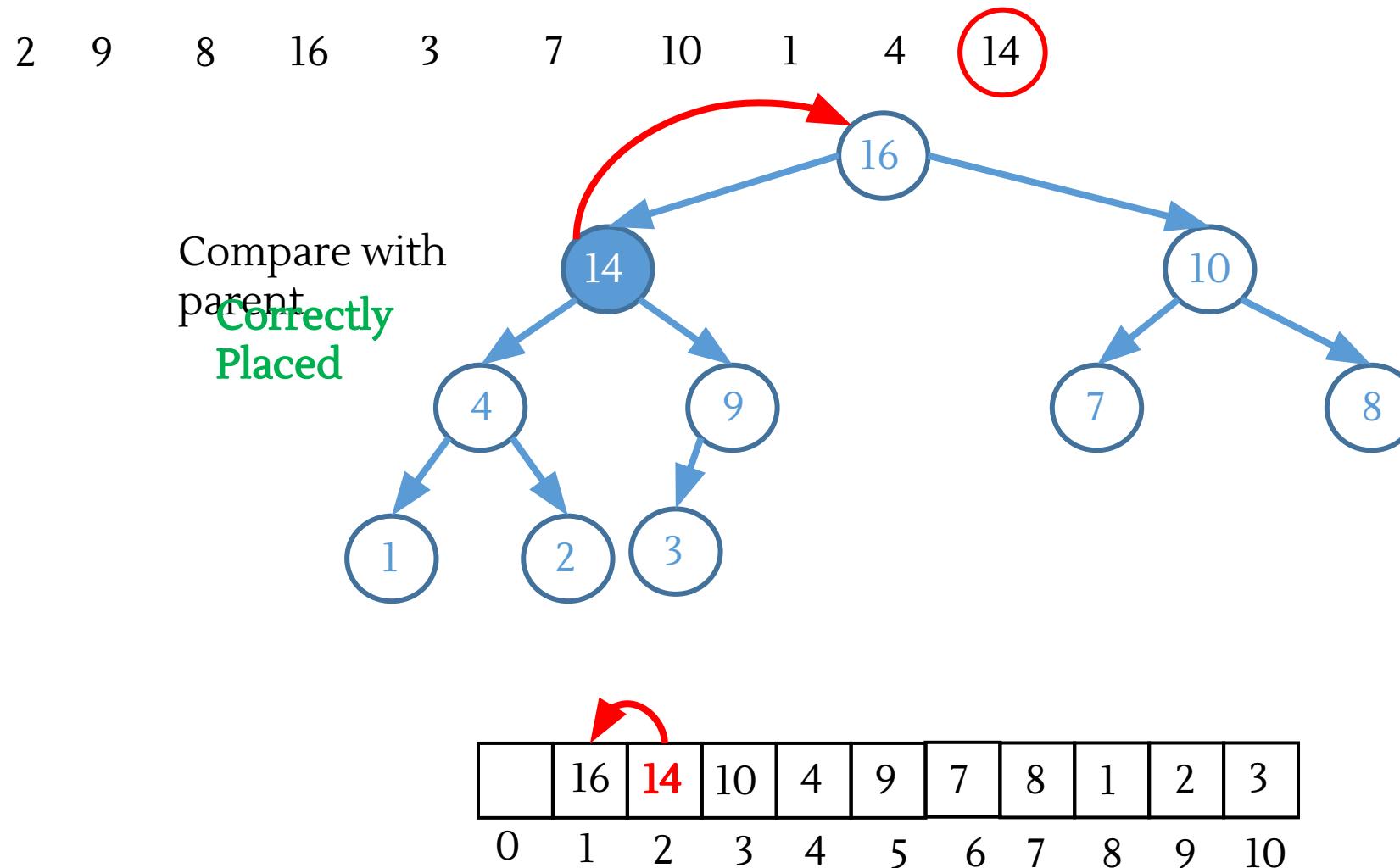
Insertion



Insertion

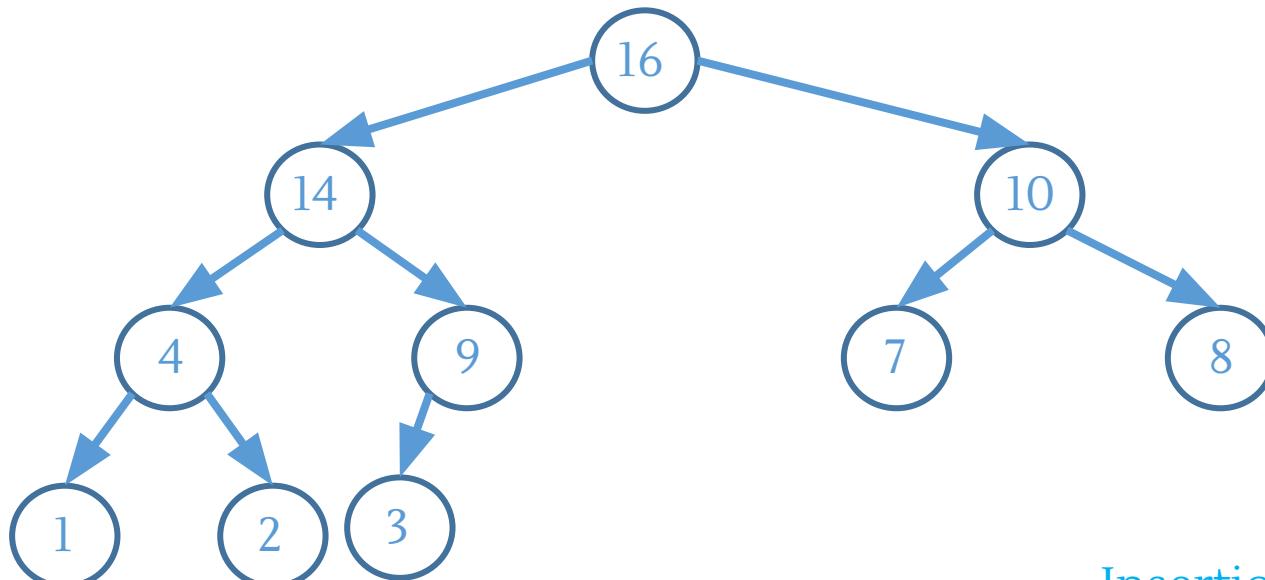


Insertion

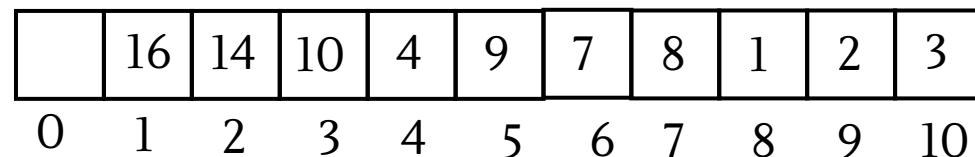


Insertion

2 9 8 16 3 7 10 1 4 14



Insertion is a **bottom to top** approach



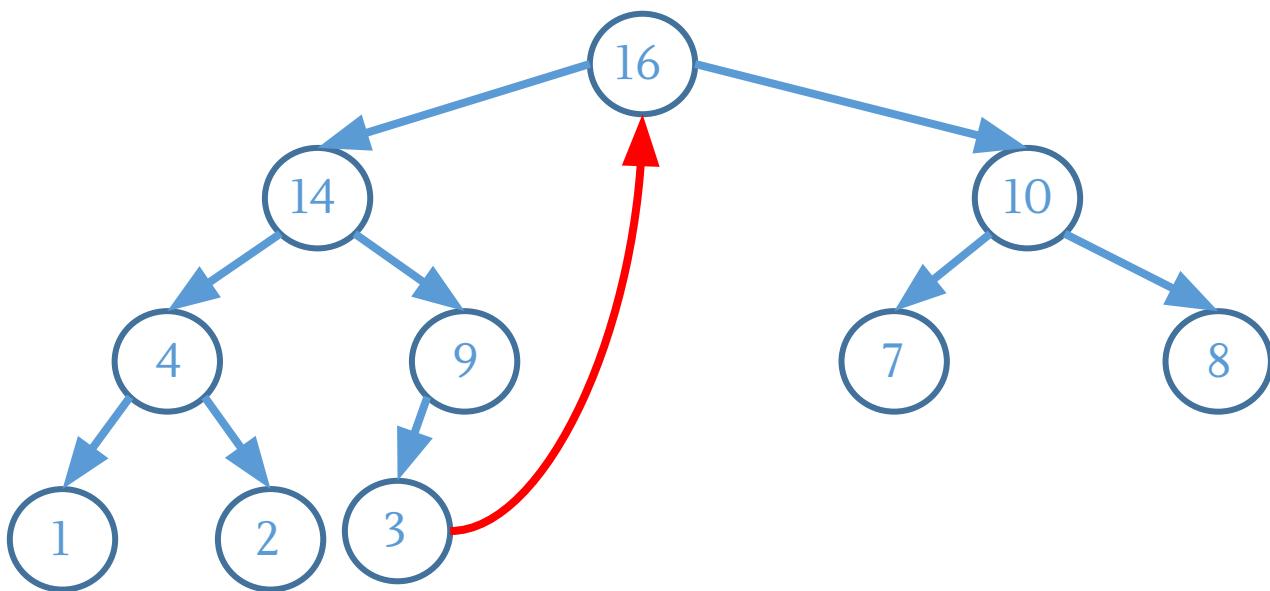
Effect of Insertion Order

- Order of children under a sub tree can be effected by insertion order
- But the height is always fixed for a fixed number of vertices

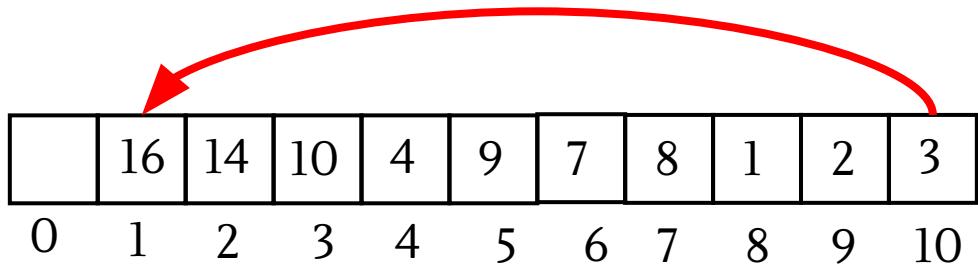
Deletion: ($O(n \log n)$ or $O(n)$)

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**) or **BottomToTop Adjustment**

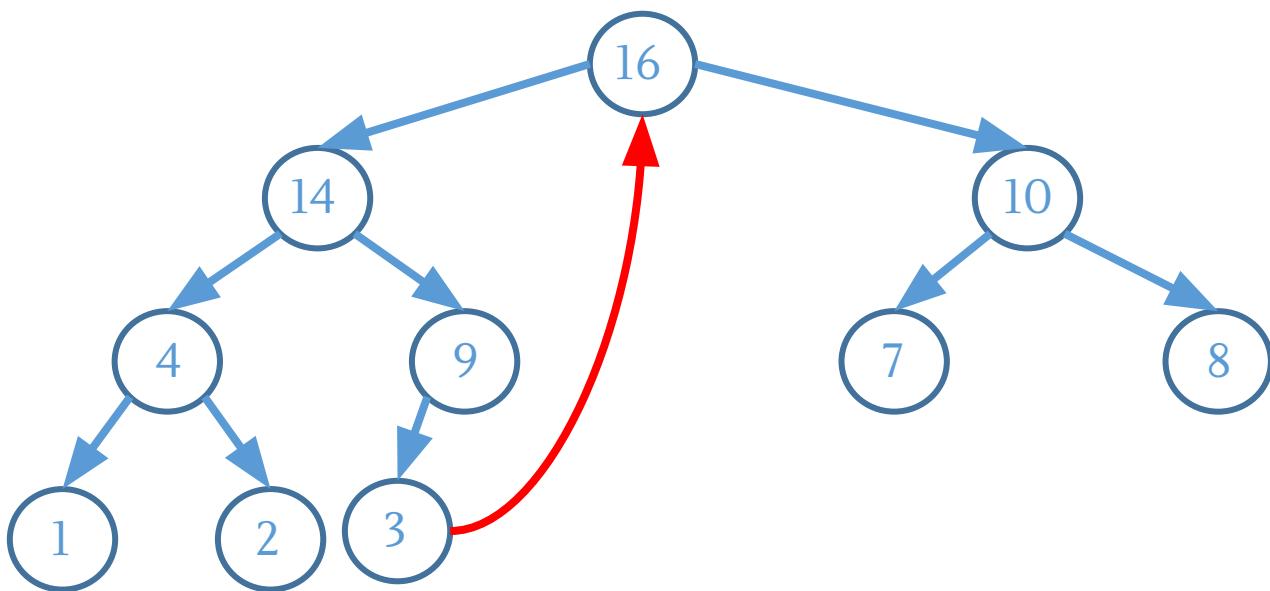
Deletion



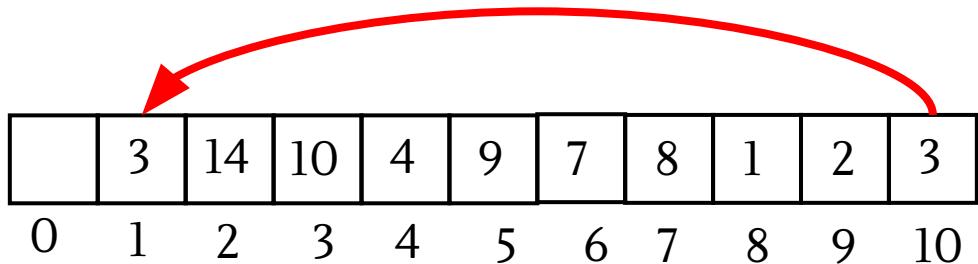
- ❑ Replace the first element by last element



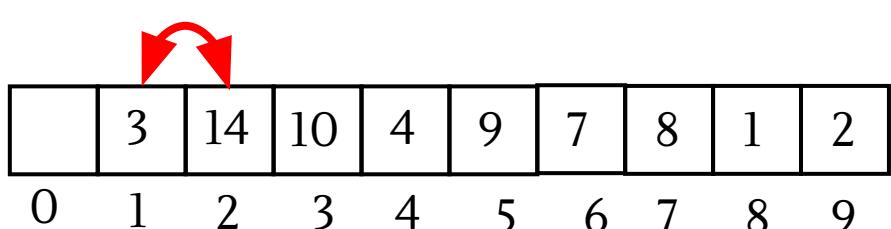
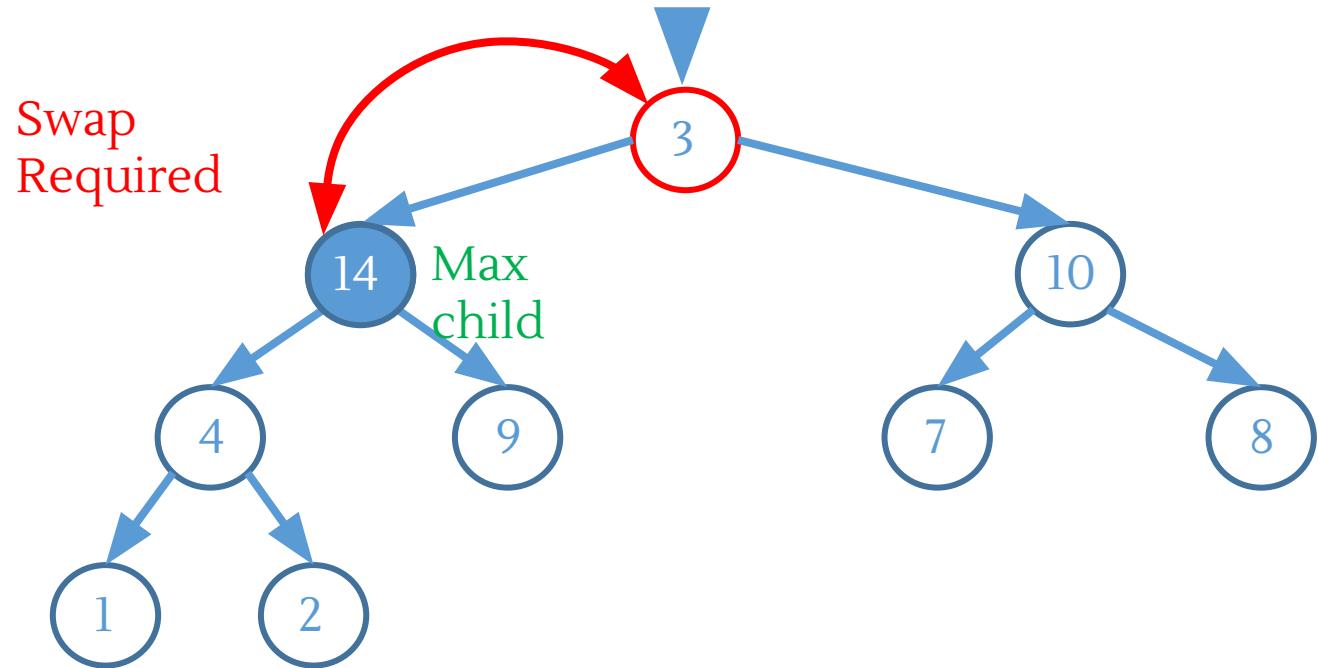
Deletion



- Replace the first element by last element
- Reduce the heap size by 1

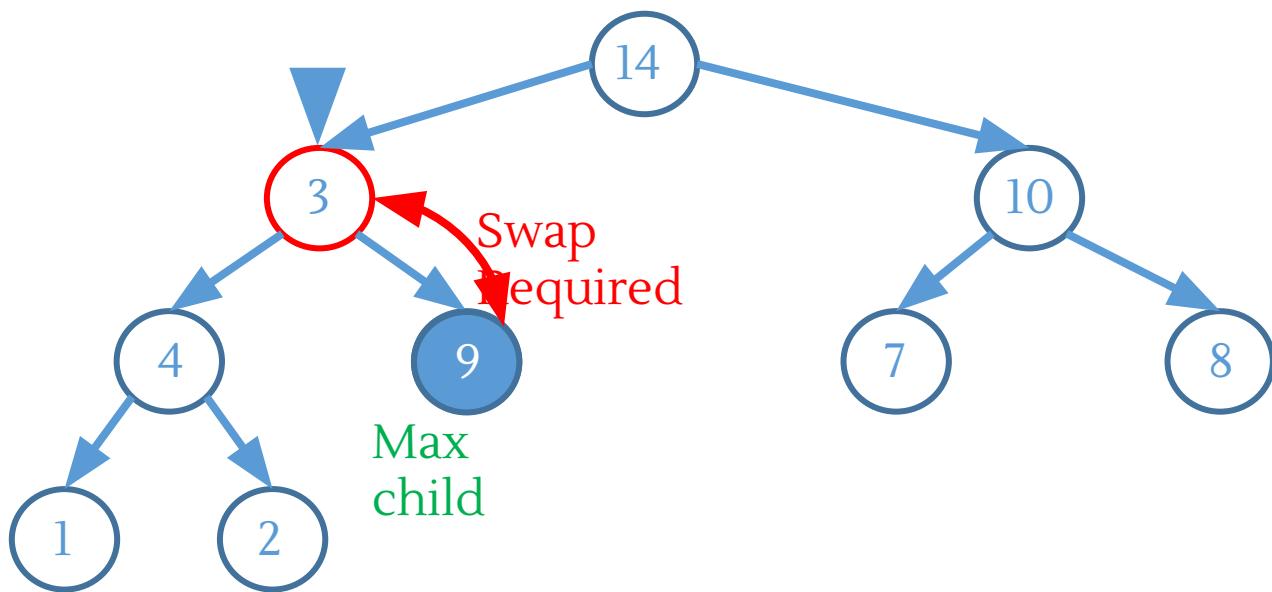


Deletion

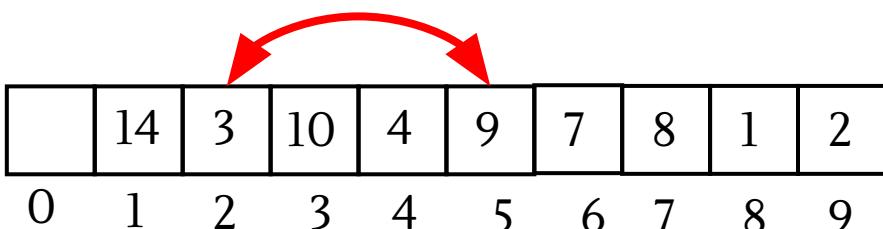


- Replace the first element by last element
- Reduce the heap size by 1
- Adjust from root to bottom
 - Find the max child
 - Swap parent and max child until correctly placed

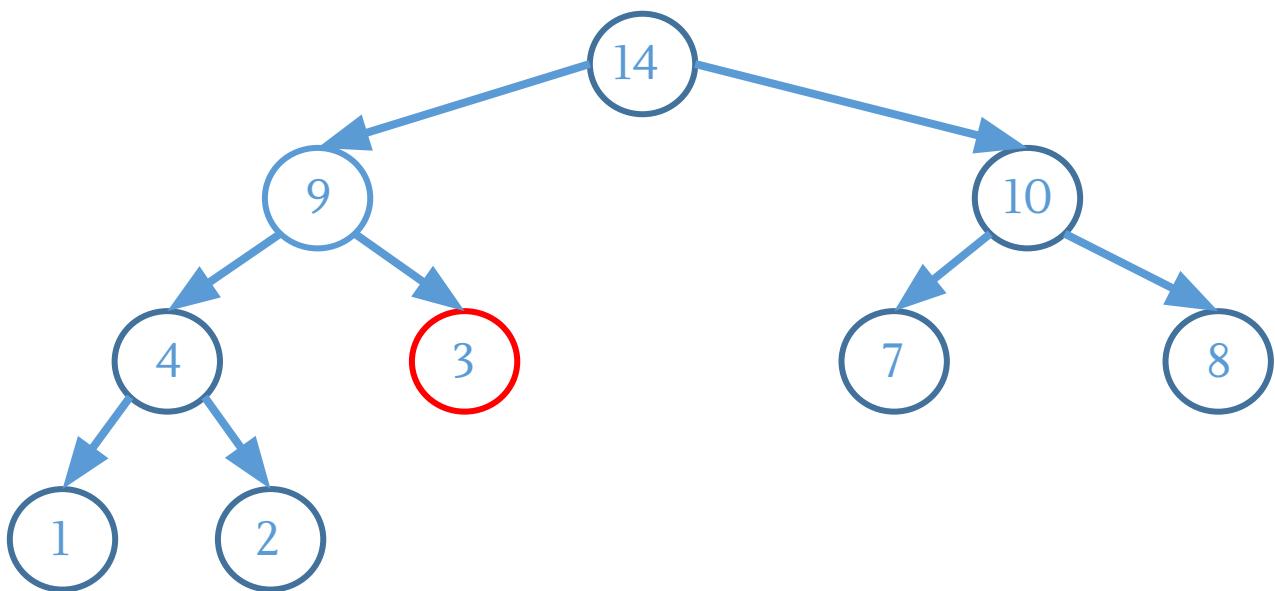
Deletion



- Replace the first element by last element
- Reduce the heap size by 1
- Adjust from root to bottom
 - Find the max child
 - Swap parent and max child until correctly placed



Deletion



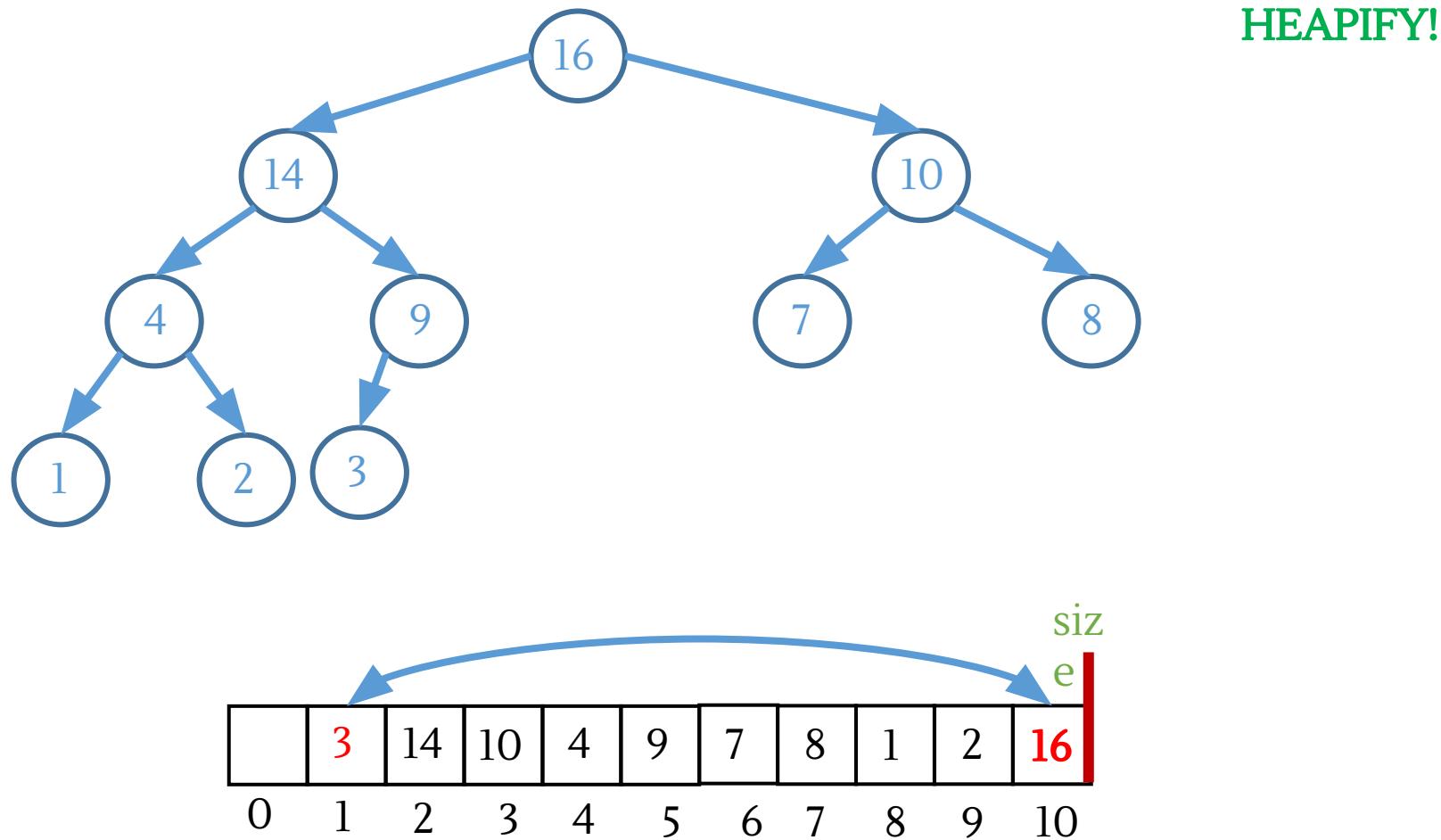
- Replace the first element by last element
- Reduce the heap size by 1
- Adjust from root to bottom
 - Find the max child
 - Swap parent and max child until correctly placed

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

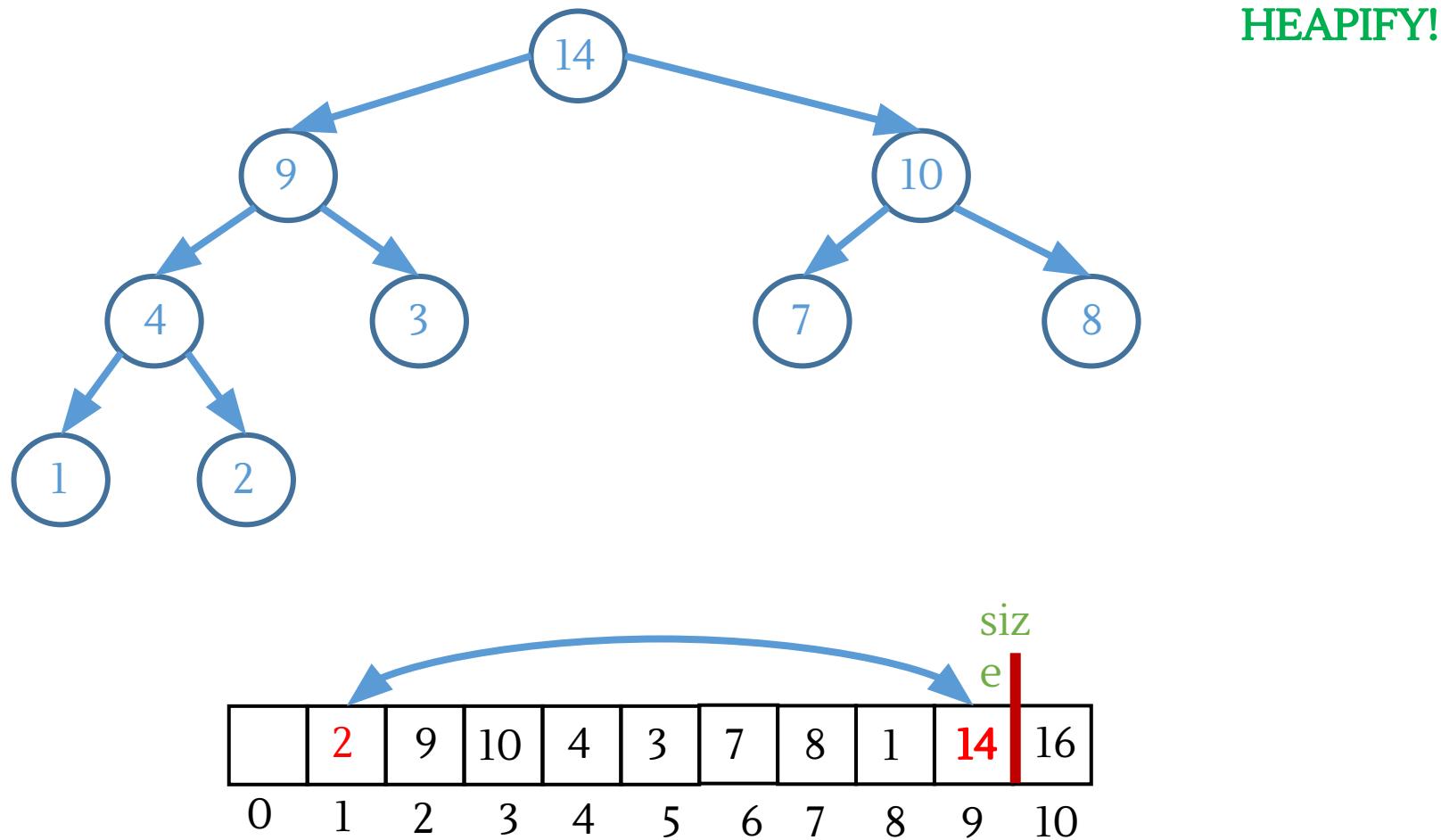
Any Idea About Heap Sort?

- ❑ Extracting all the roots sequentially produces the sorted
- ❑ ~~sequence~~ Storing the root element at the end of the array after resizing makes the array sorted

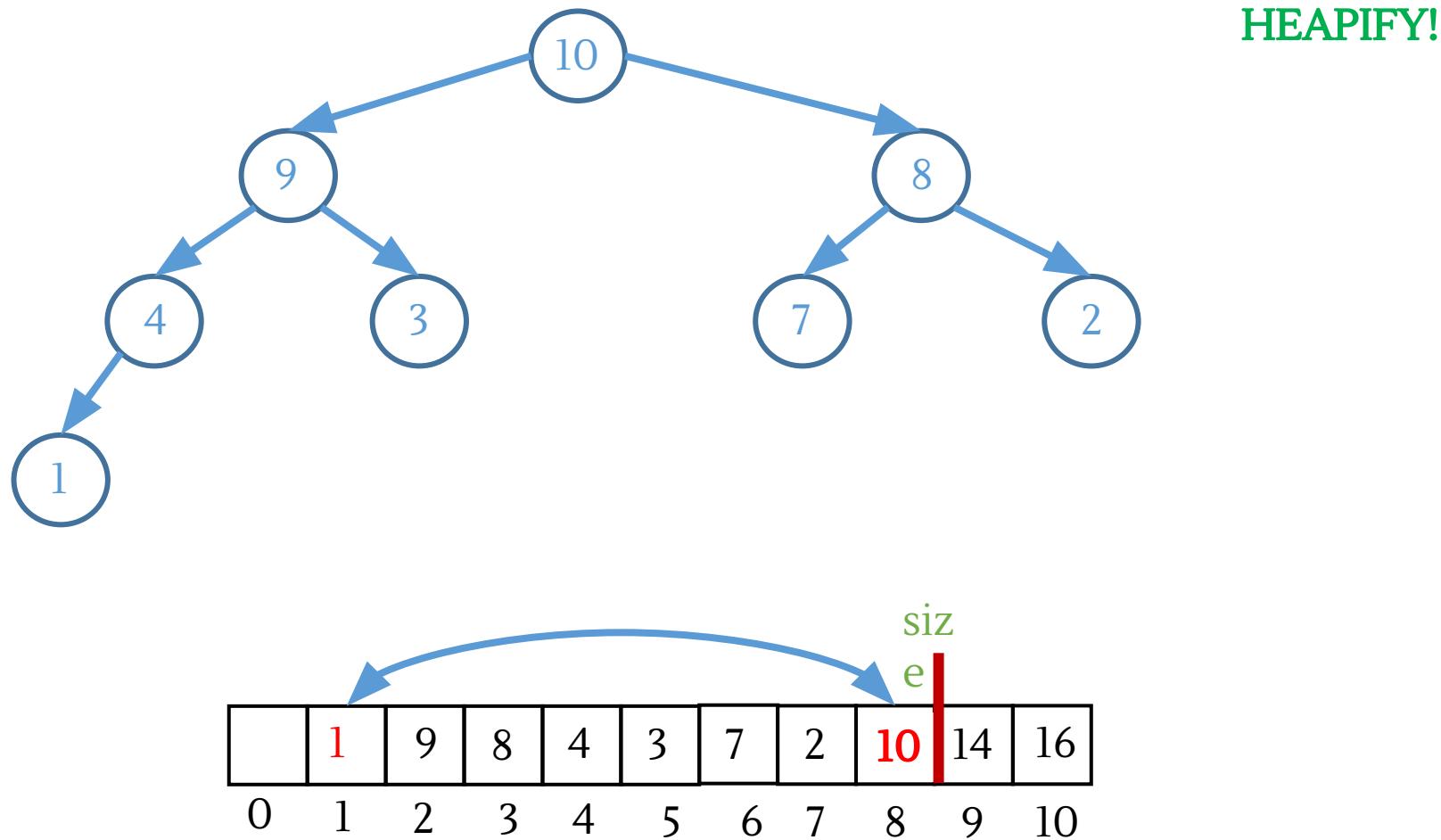
Heap Sort



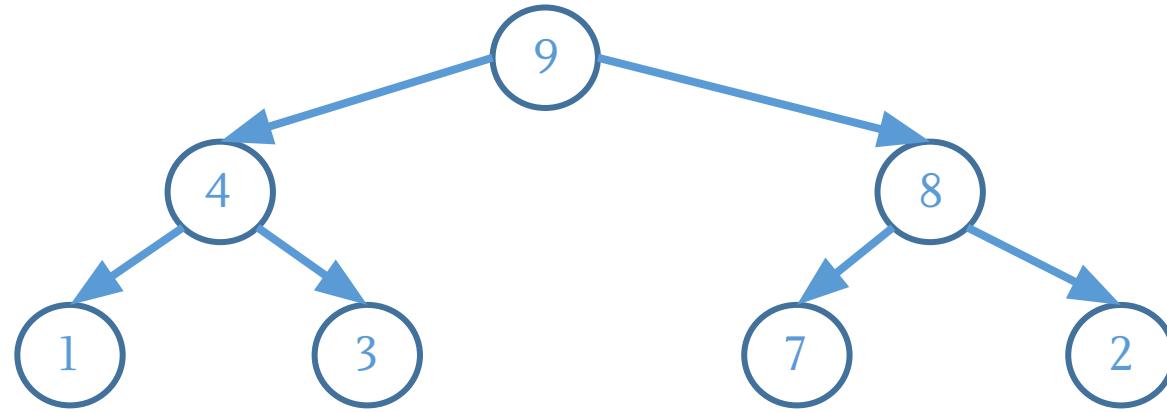
Heap Sort



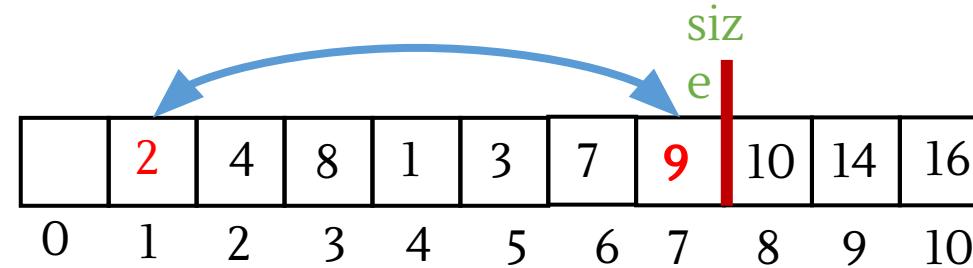
Heap Sort



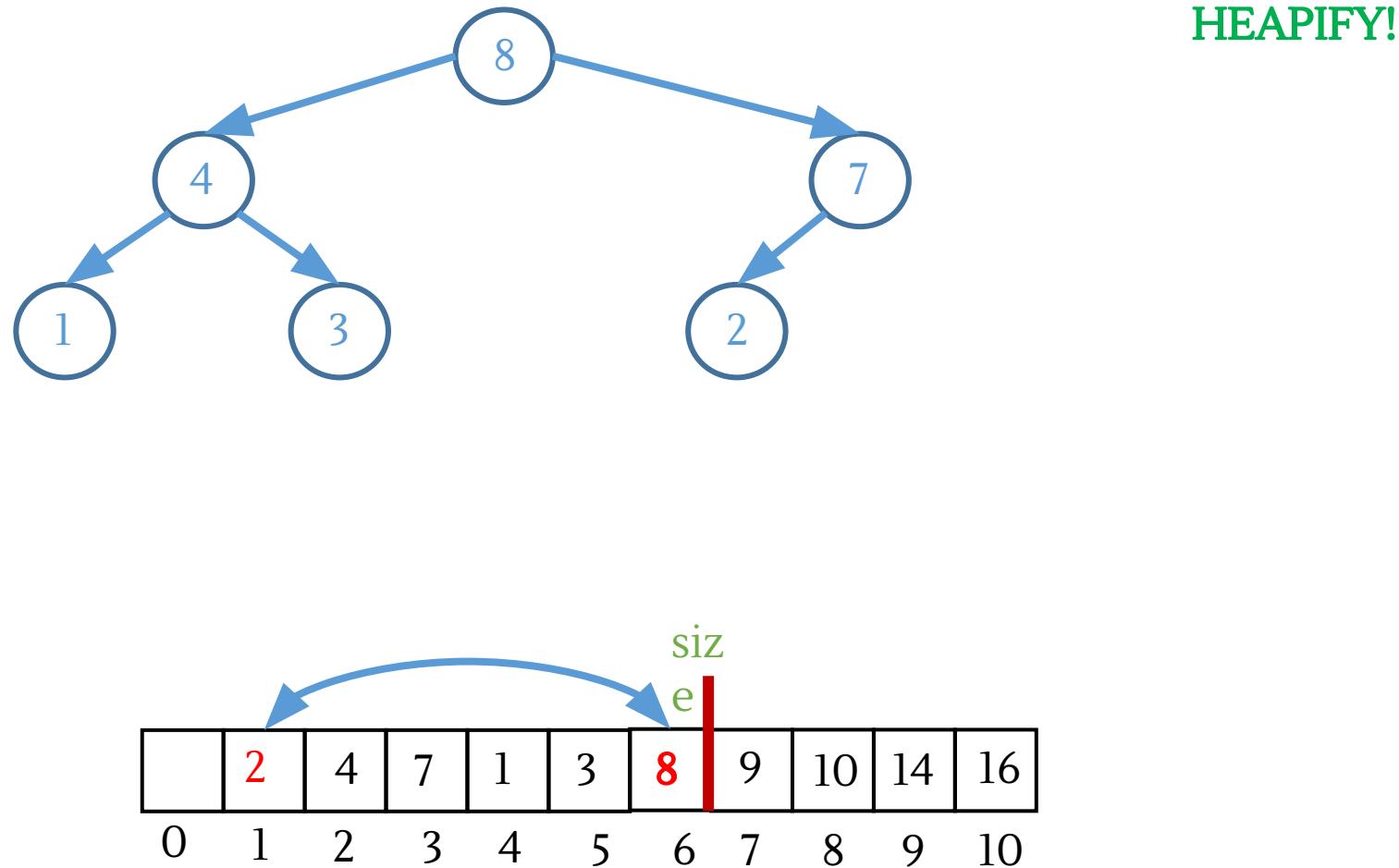
Heap Sort



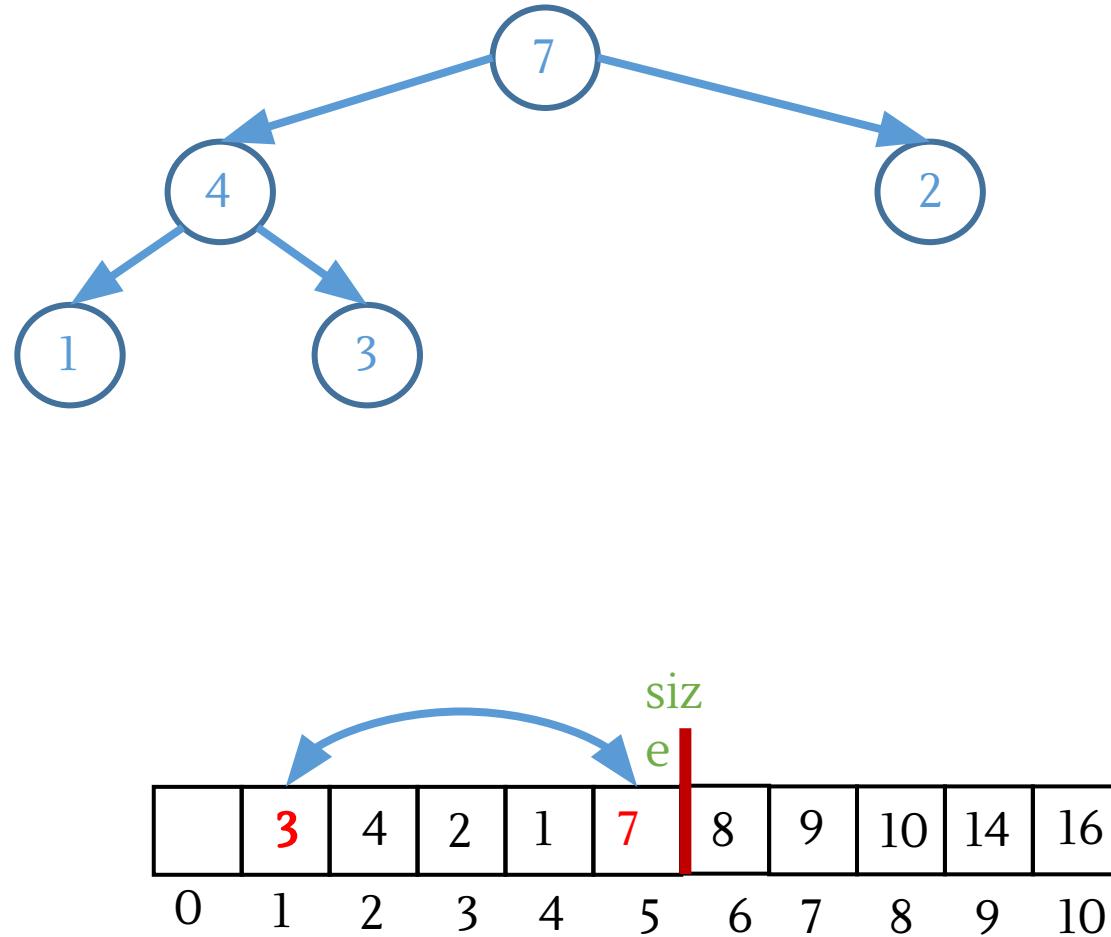
HEAPIFY!



Heap Sort

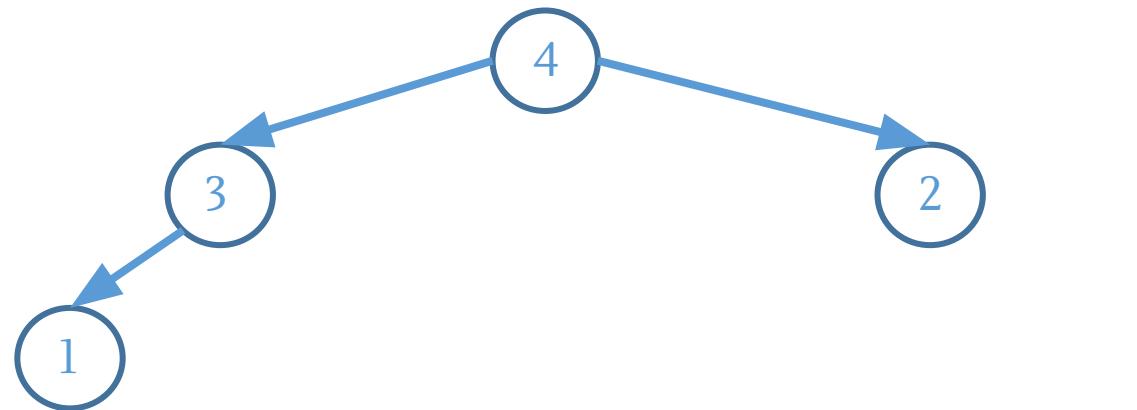


Heap Sort

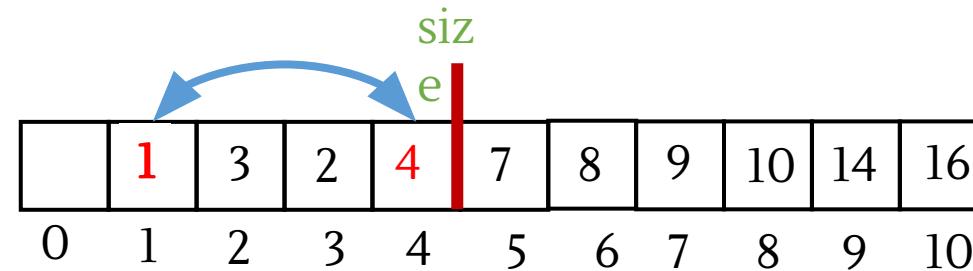


HEAPIFY!

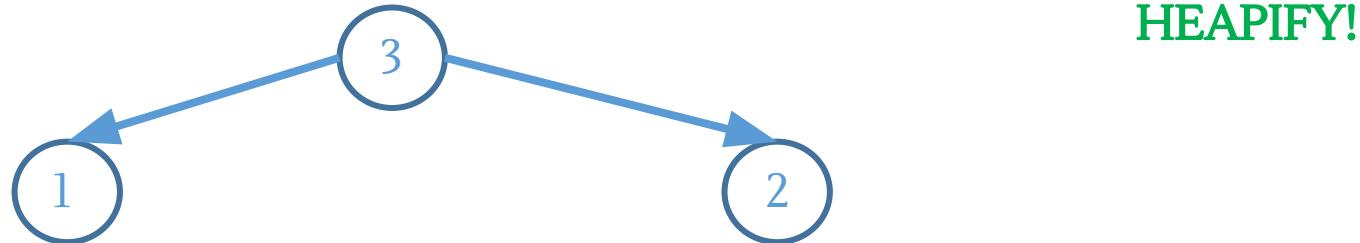
Heap Sort



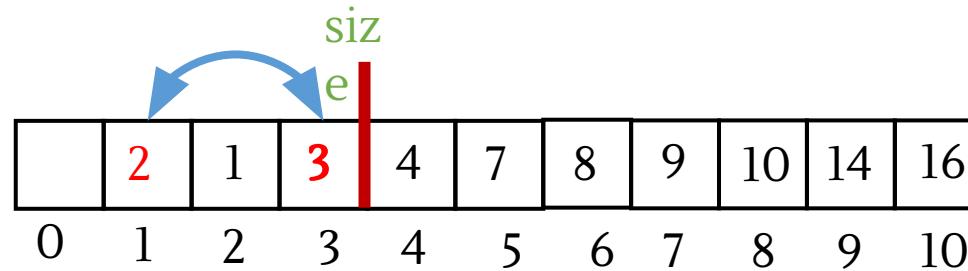
HEAPIFY!



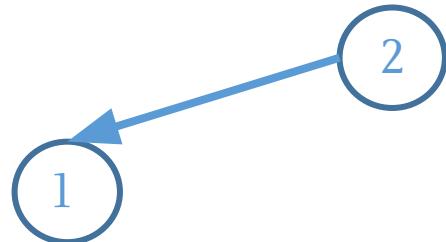
Heap Sort



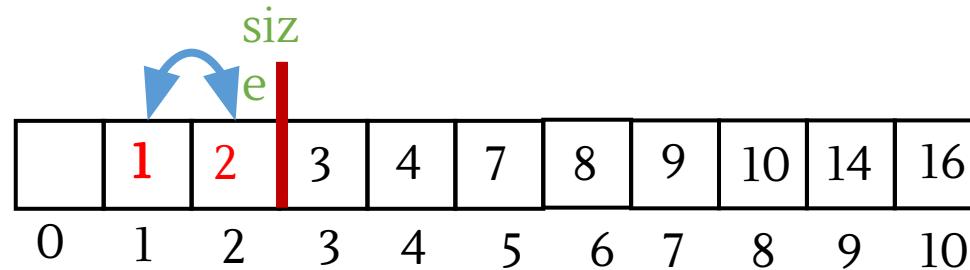
HEAPIFY!



Heap Sort



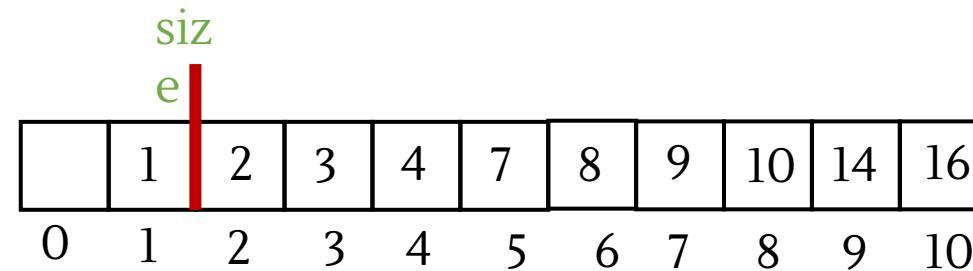
HEAPIFY!



Heap Sort

1

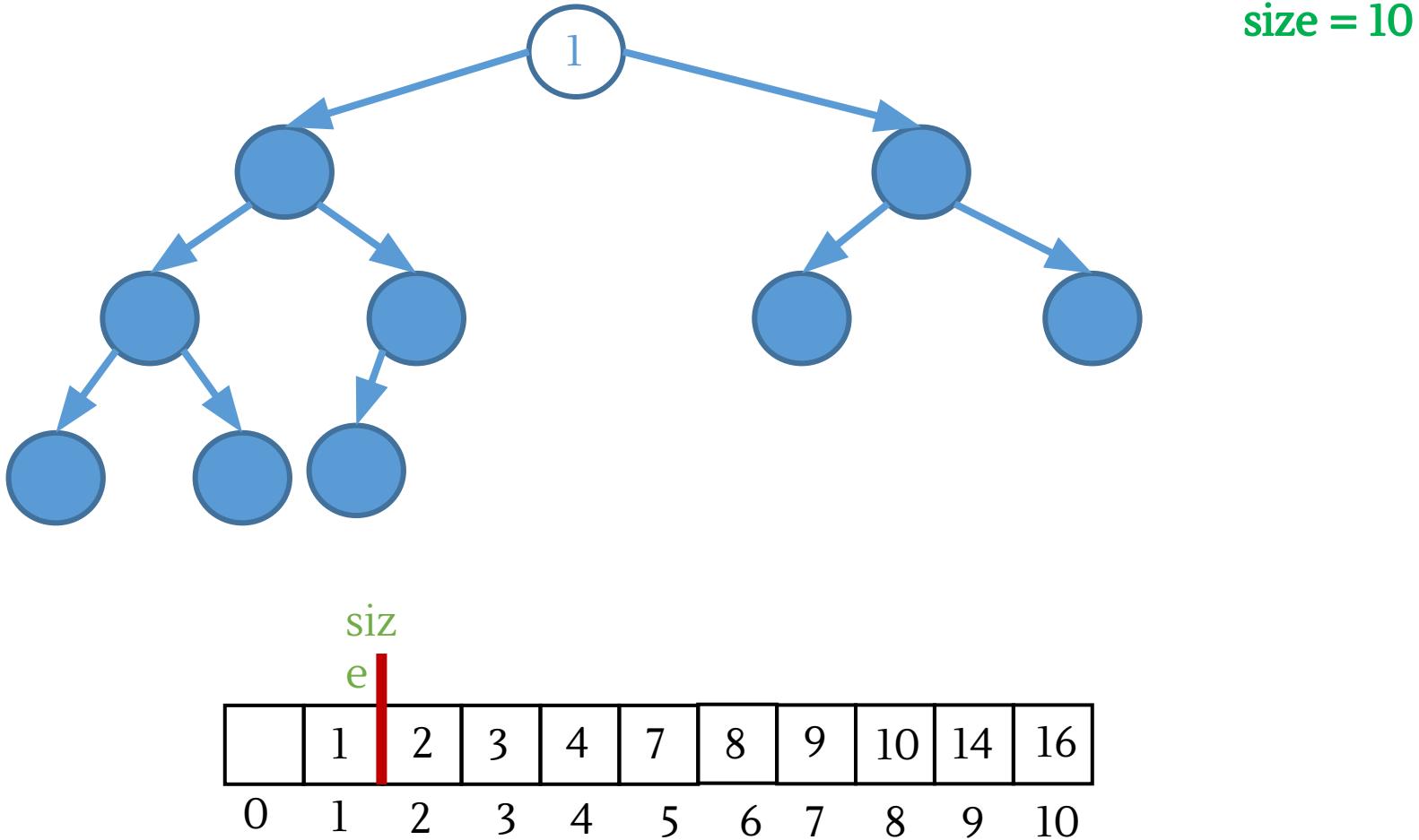
Sorted
!



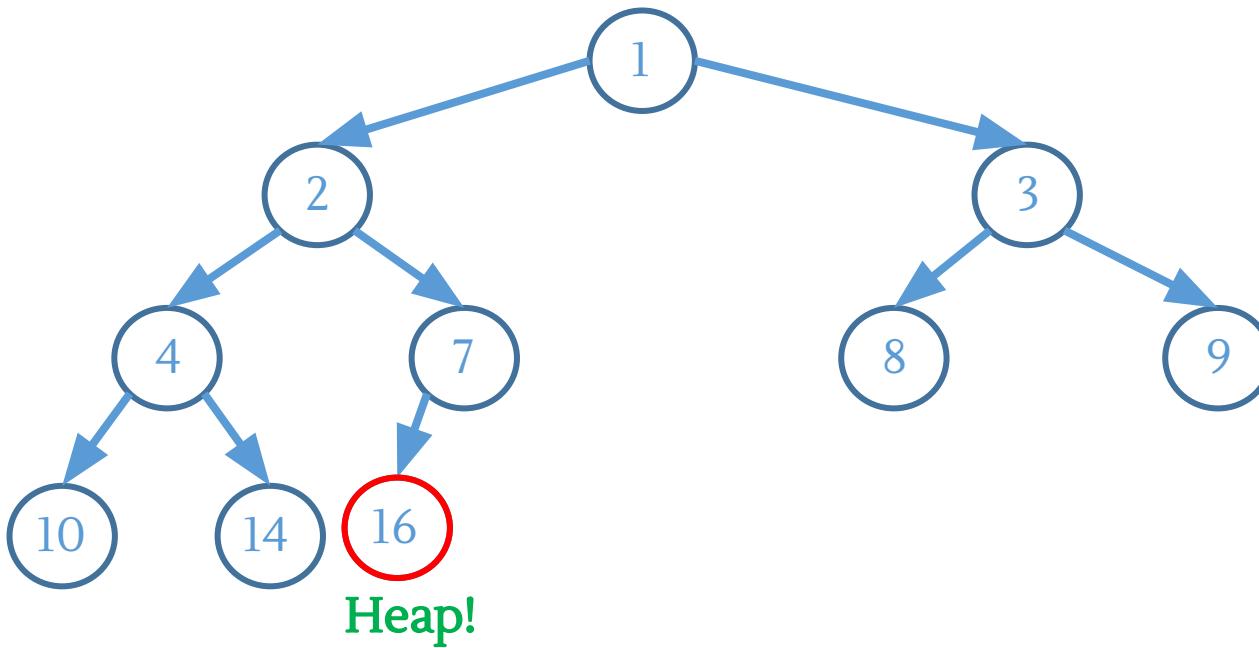
Time Complexity?

- $O(n \log n)$

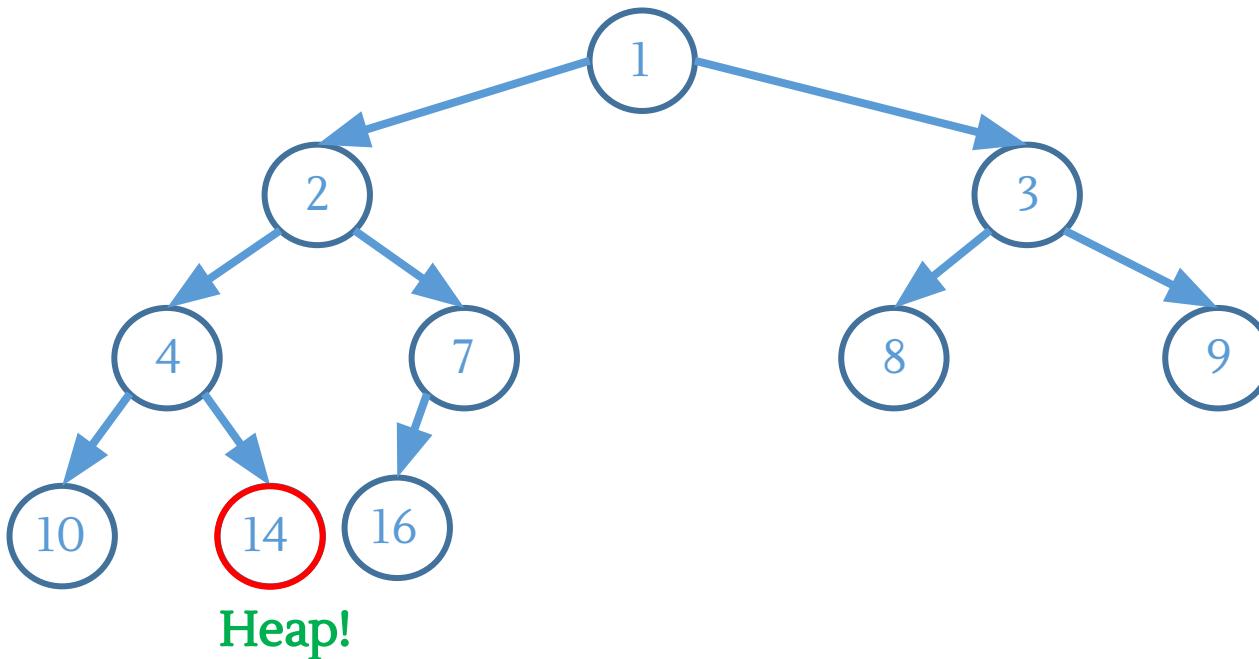
Restore The Array



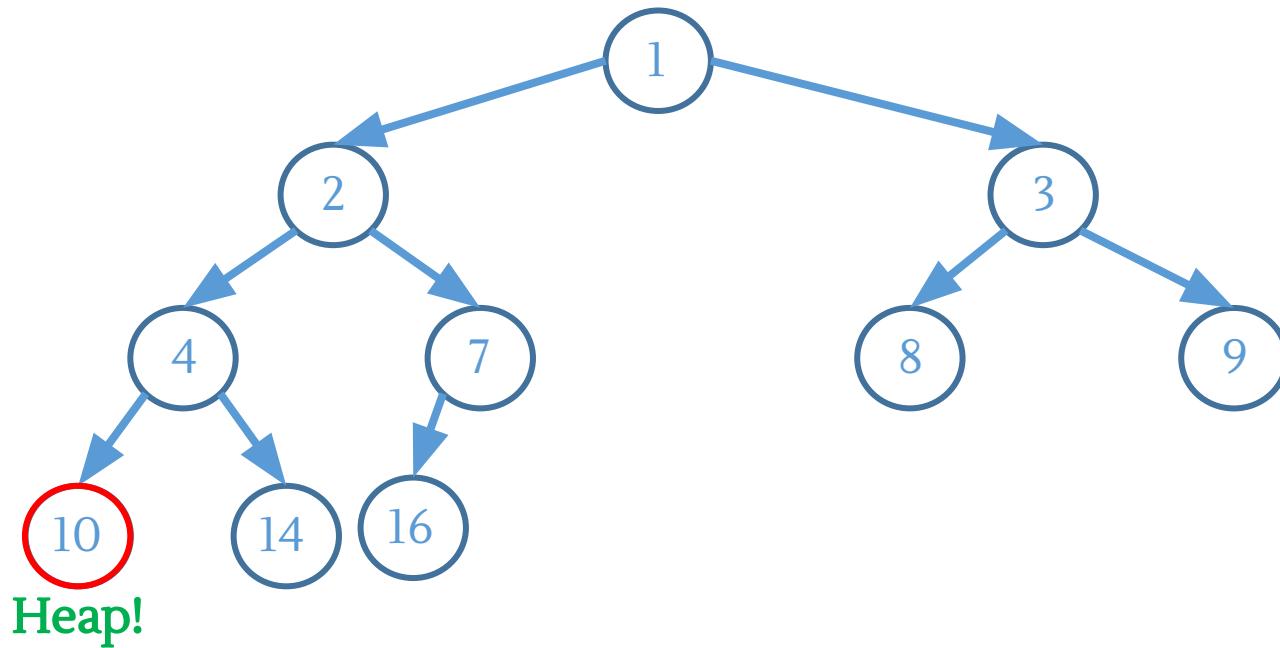
Build Heap



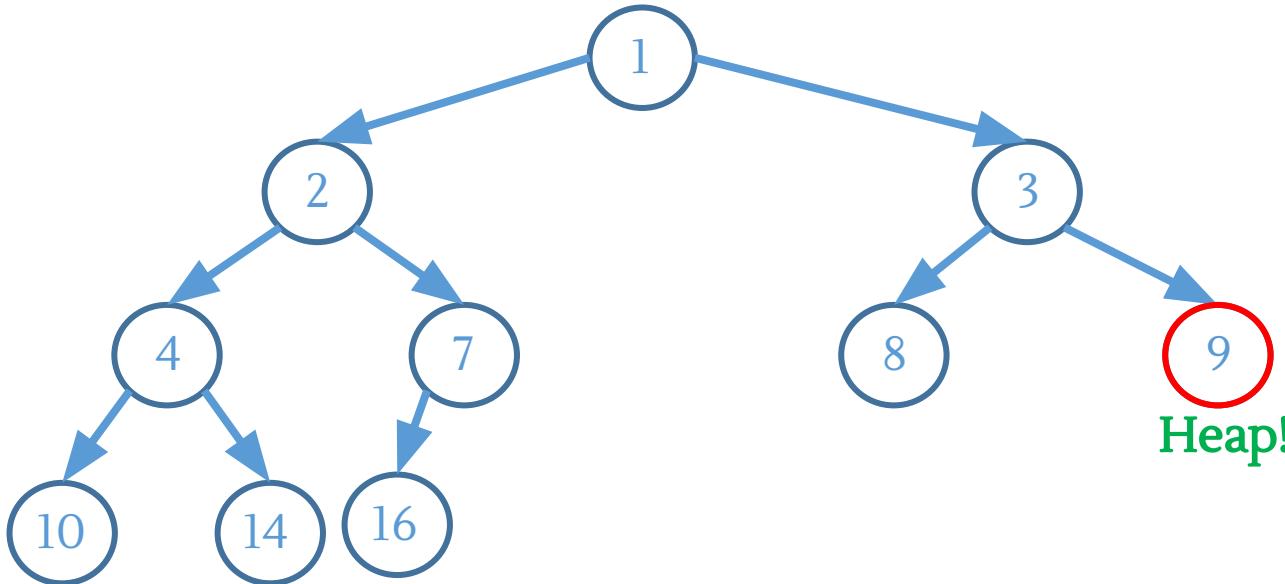
Build Heap



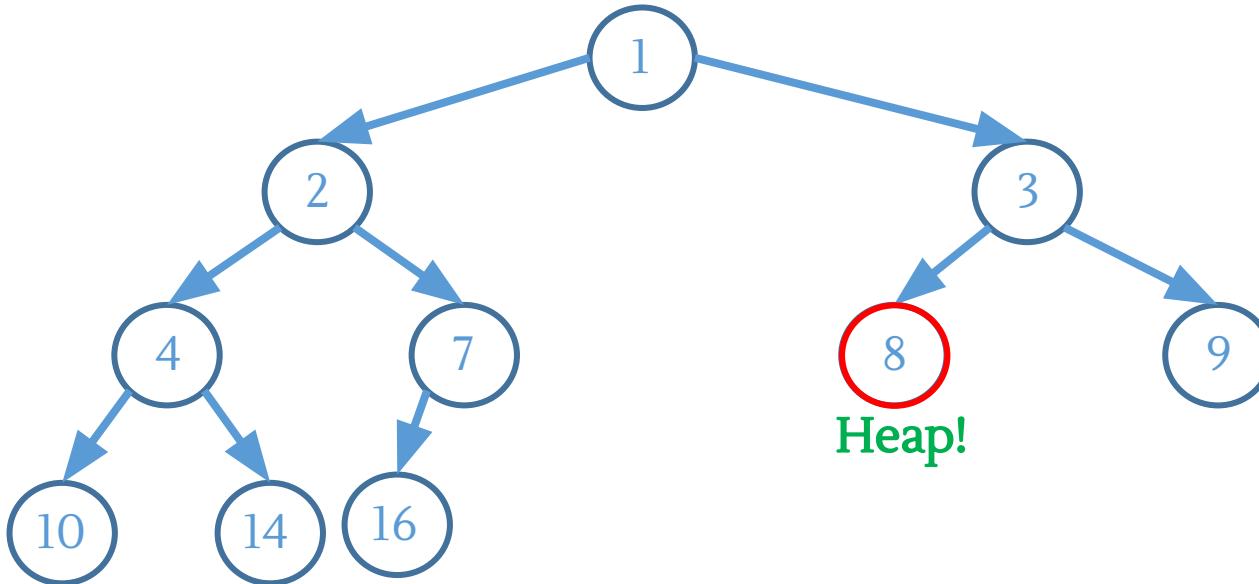
Build Heap



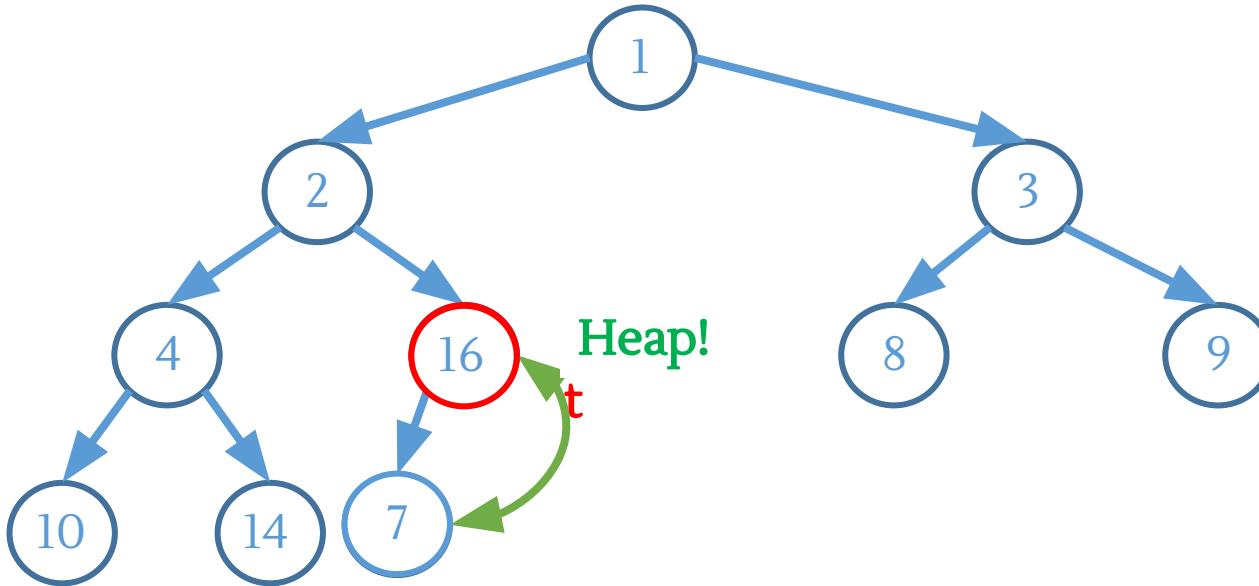
Build Heap



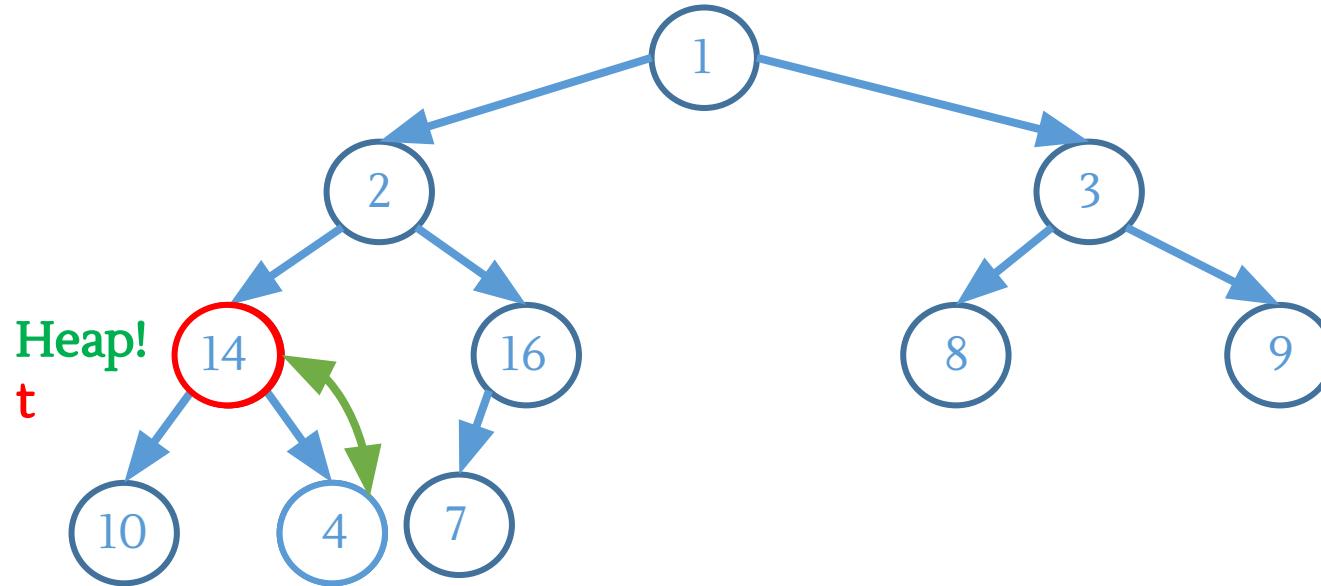
Build Heap



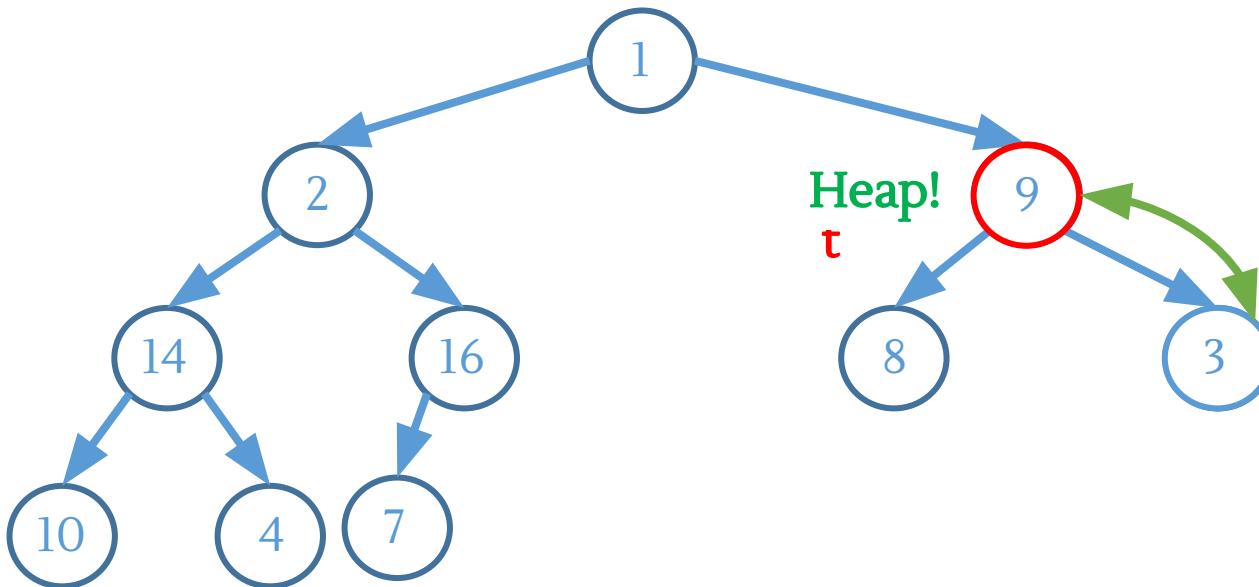
Build Heap



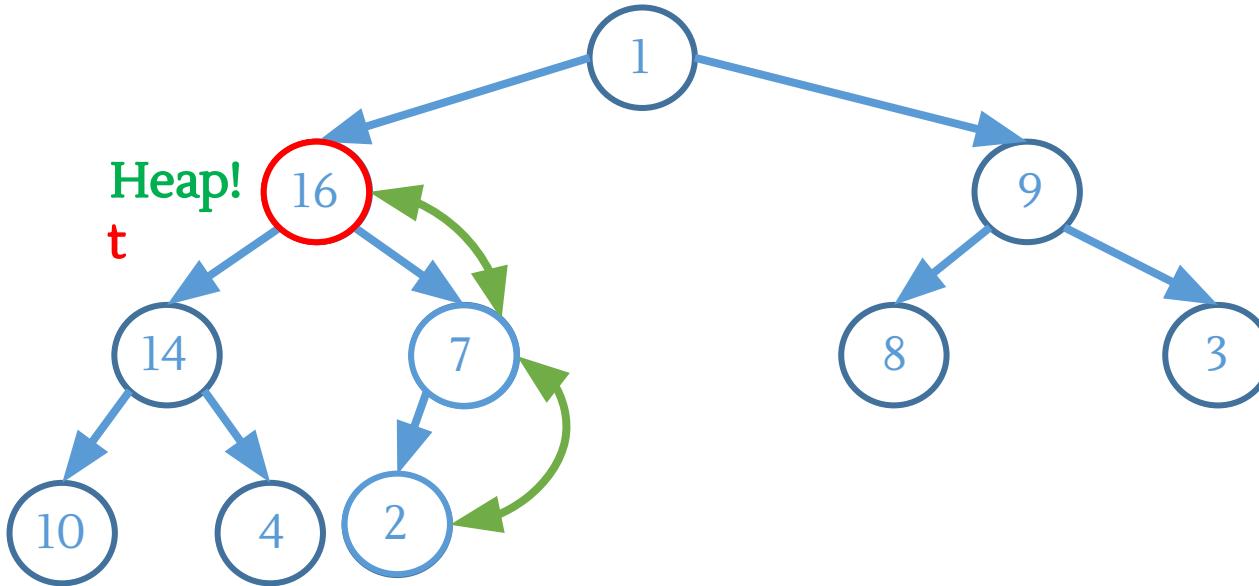
Build Heap



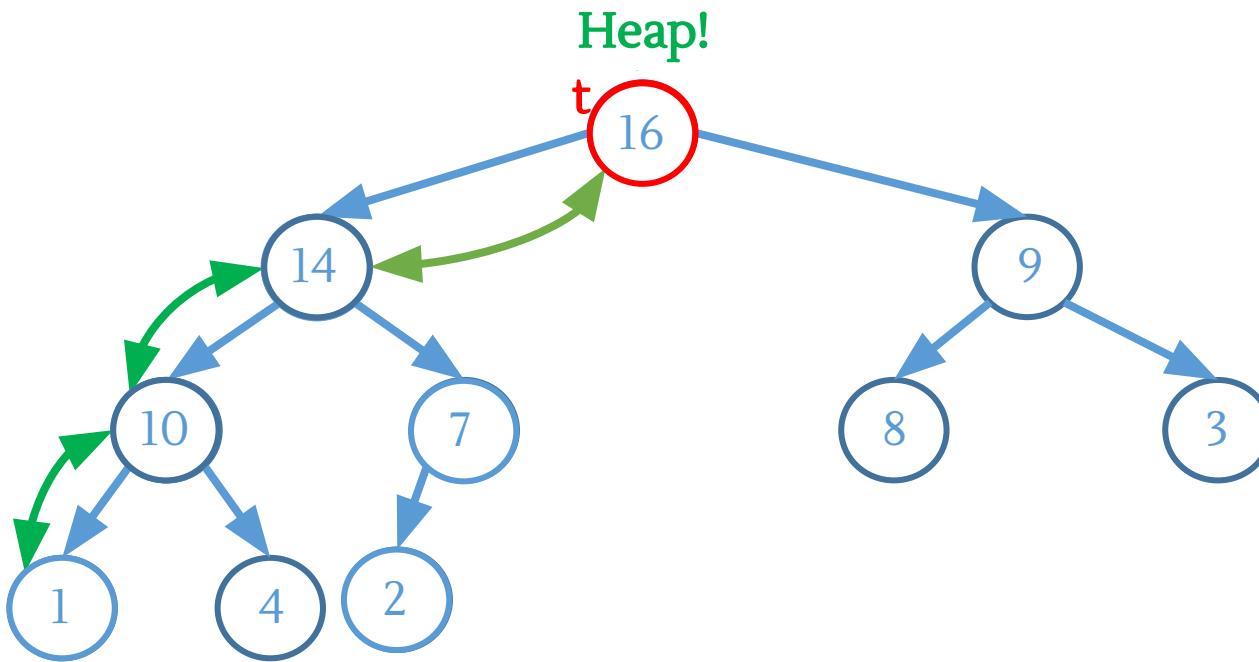
Build Heap



Build Heap



Build Heap



Build Heap

- ❑ For all internal vertices from $\text{floor}(i/2)$ to 1 perform top down adjustment (Heapify)

Queue Vs Priority Queue

- Enqueue Order: 10 15 4 8 9 20 17 18
- General Queue Dequeue Order: 10 15 4 8 9 20 17
18
- Priority Queue Dequeue Order: 20 18 17 15 10 9 8
4
- **Basic Functionalities of Priority Queue**
 - **Insert:** Inserts a value in the priority
 - **Top:** Returns the Maximum Element of the priority
 - **ExtractMax:** Removes and returns the Maximum Element of the priority
- Required Data Structure: **Max Heap**

Priority Queue

- **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)

Operations of Max Priority Queue:

1. **INSERT(S,x)**: inserts the element x into the set S.
2. **INCREASE-KEY(S,x,k)**: increases the value of element x's key to the new value k, which is assumed to be atleast as large as x's current key value.
3. **MAXIMUM(S)**: returns the element of S with the largest key.
4. **EXTRACT-MAX(S)**: removes and returns the element of S with the largest key.

Implementation of Max Priority Queue Using Max Heap:

1. **MAX-HEAP-INSERT**: Implements the INSERT operation and running time for an element heap is **O(logn)**.
2. **HEAP-INCREASE-KEY**: Implements the INCREASE-KEY operation and running time is **O(logn)**.
3. **HEAP-MAXIMUM(A)**: Implements MAXIMUM(S) operation in **$\Theta(1)$** time.
4. **HEAP-EXTRACT-MAX**: Implements EXTRACT-MAX(S) and running time is **O(logn)**.

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 [Retrieval](#)
- ❑ It refers to the quick retrieval of strings
- ❑ Used for storing strings, string matching, lexicographical sorting etc.

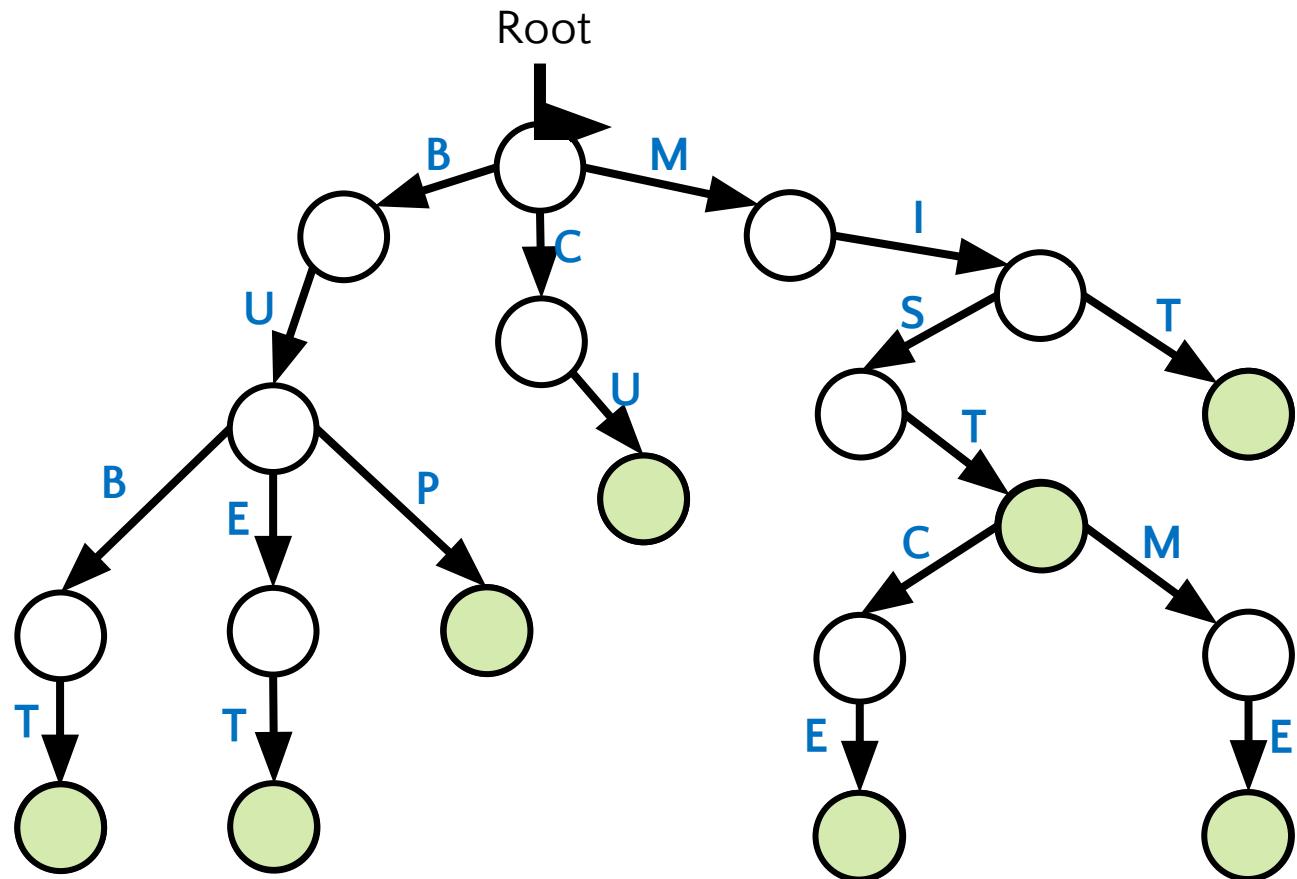


WHY TRIE?

- 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: $O(n \times m)$ 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 $O(\text{length}(x))$ complexity
- So doesn't matter how long the database is, time complexity of finding a string x will remain $\text{length}(x)$

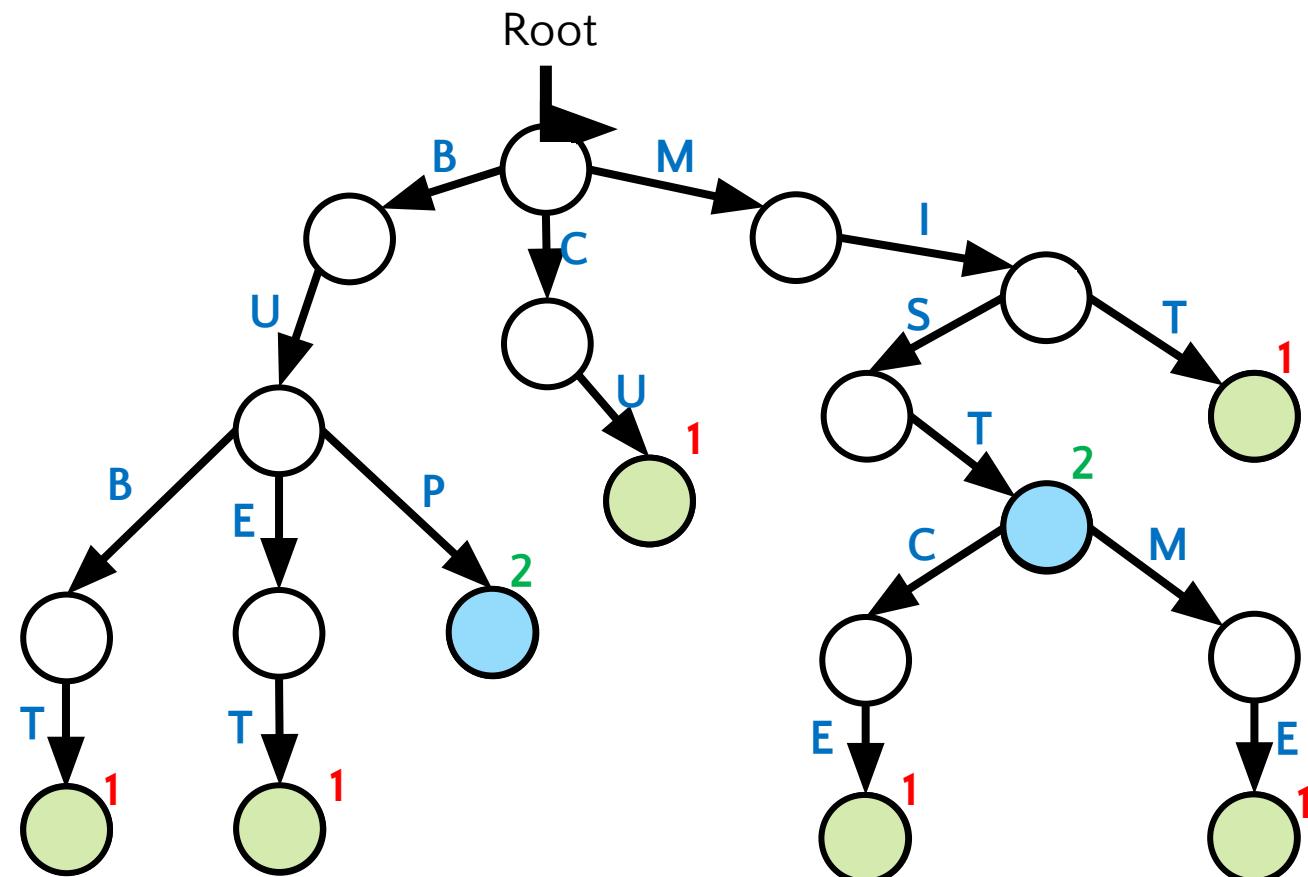
INSERT IN TRIE

- 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")





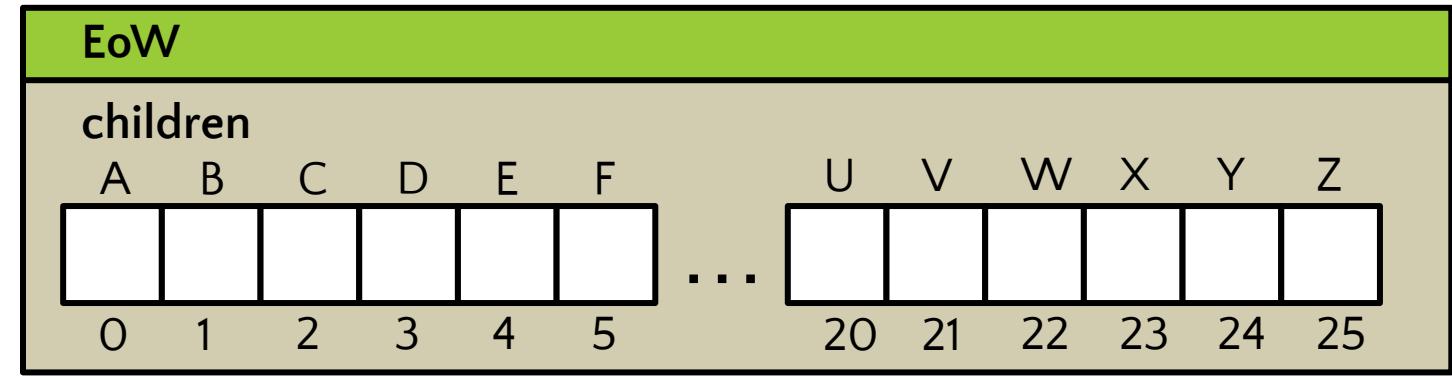
INSERT IN TRIE (WITH COUNTER)

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



NODE REPRESENTATION

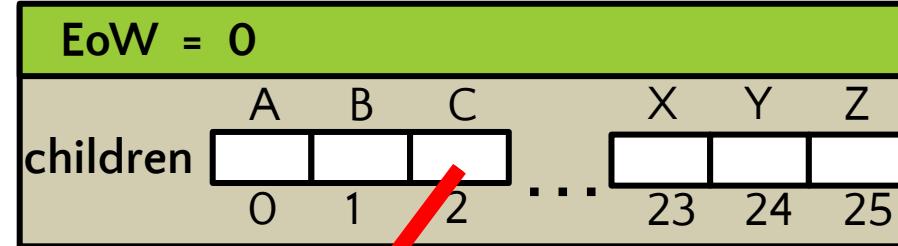
```
struct Node{  
    int EoW;  
    Node *children[26];  
}
```



NODE REPRESENTATION

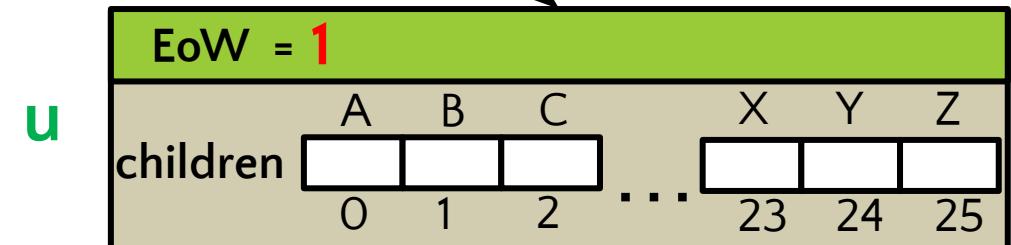
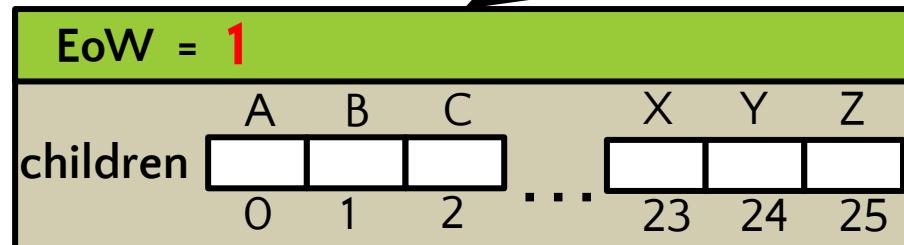
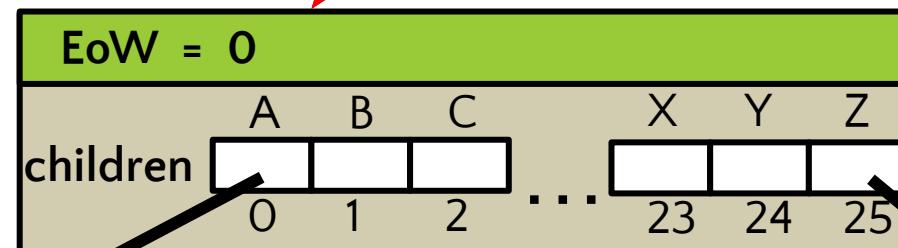
- insert("CA")
- insert("CZ")

root

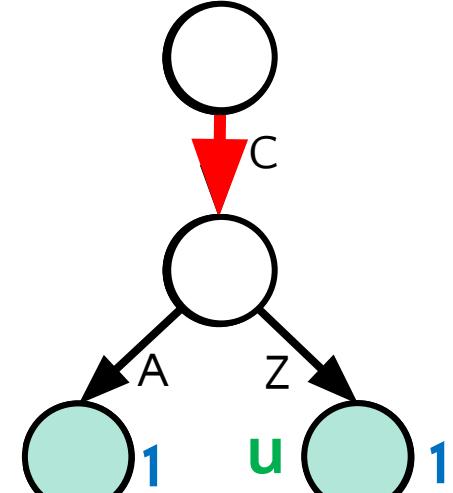


Iterations are completed

Increment EoW of u
 $u \rightarrow EoW = u \rightarrow EoW + 1$



root





INSERT IN TRIE

insert(*x*)

Node pointer *u* \leftarrow *root*

Initially pointing *u* at the *root*

for *k* \leftarrow 0 to *size(x)* - 1

Iterates for *size(x)* number of times

r \leftarrow *x*[*k*] - 65

r is the relative position of current char

O(|x|)

if *u->children[r]* is NULL

No children condition

u->children[r] \leftarrow new Node()

Creates new node under *children[r]*

u \leftarrow *u->children[r]*

Pushes *u* down for next iteration

u->EoW \leftarrow *u->EoW* + 1;

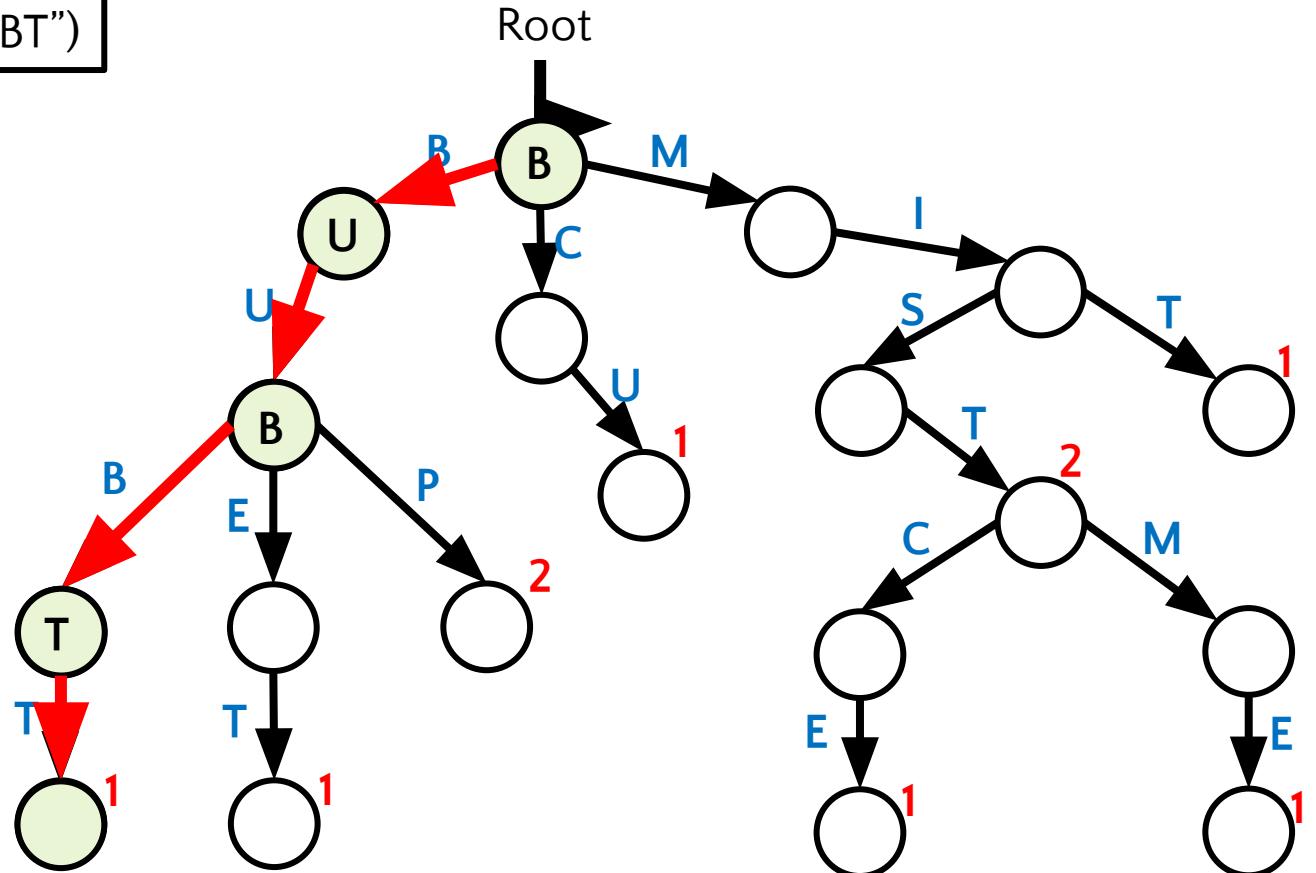
Increments *u->EoW* after completing iteration

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



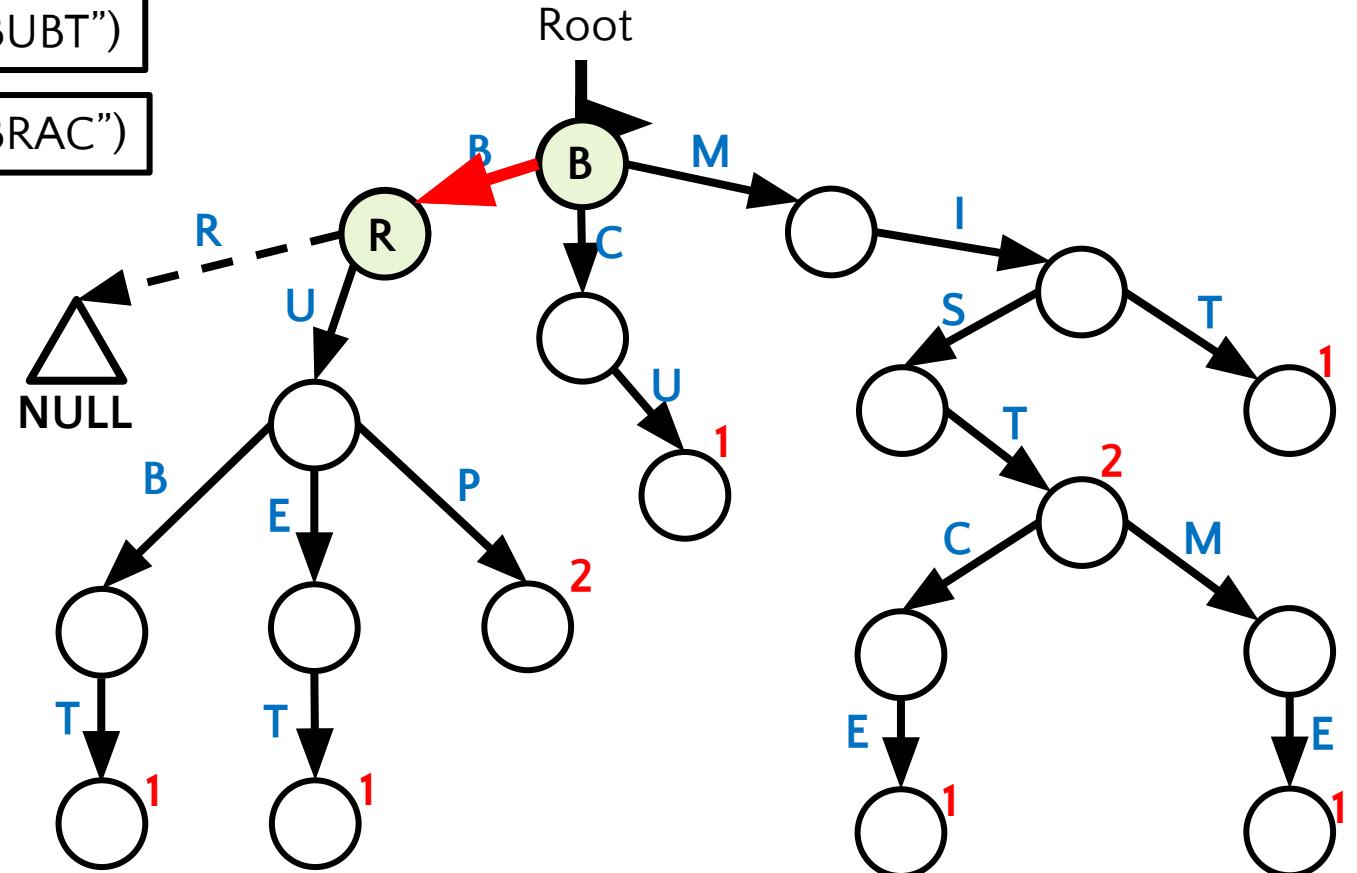
SEARCH IN TRIE

- 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

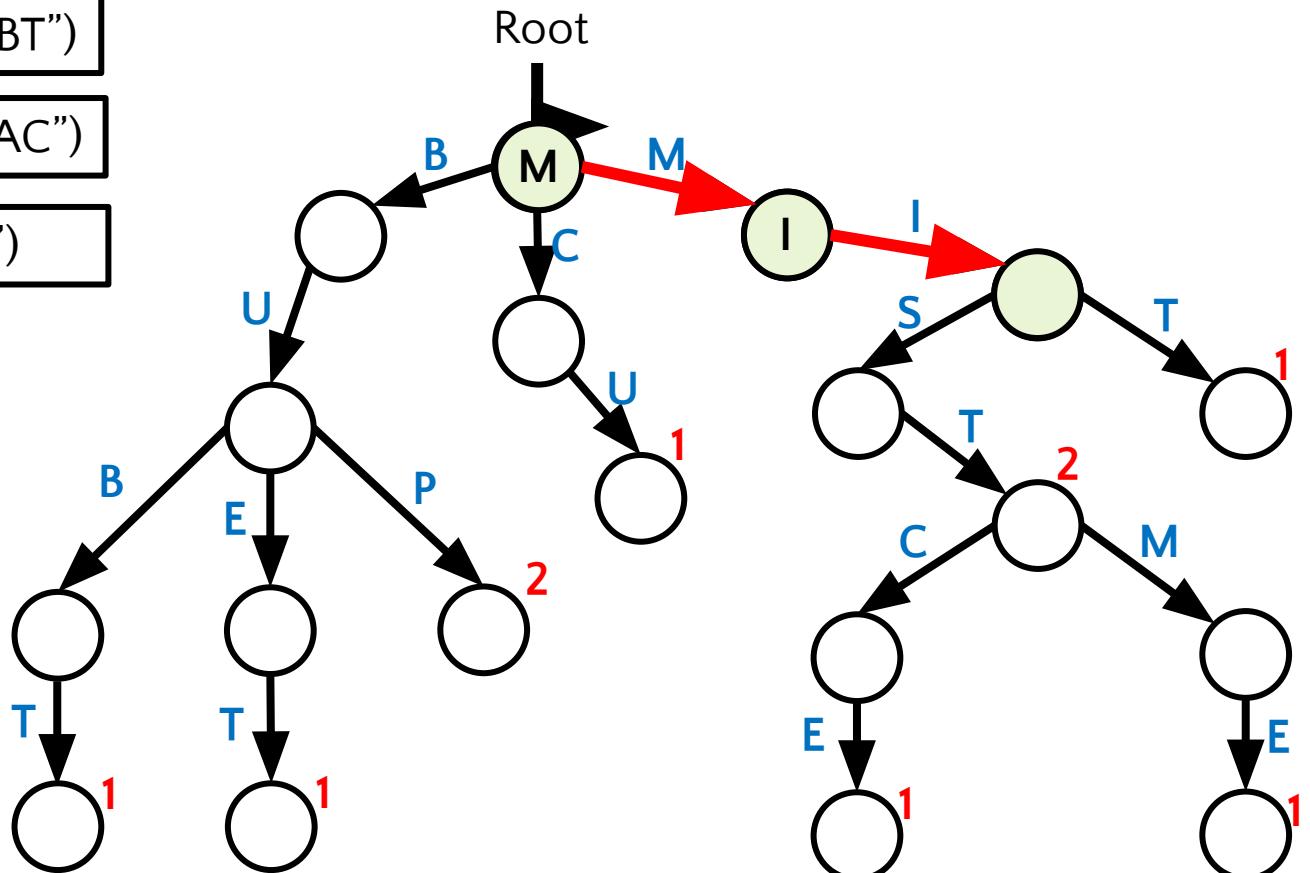


- insert("MIT")
 - insert("MIST")
 - insert("BUET")
 - insert("MISTCE")
 - insert("BUBT")
 - insert("MISTME")
 - insert("BUP")
 - insert("CU")
 - insert("MIST")
 - insert("BUP")

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

We can't reach a node with counter=0

Means “MI” doesn’t exist



SEARCH IN TRIE

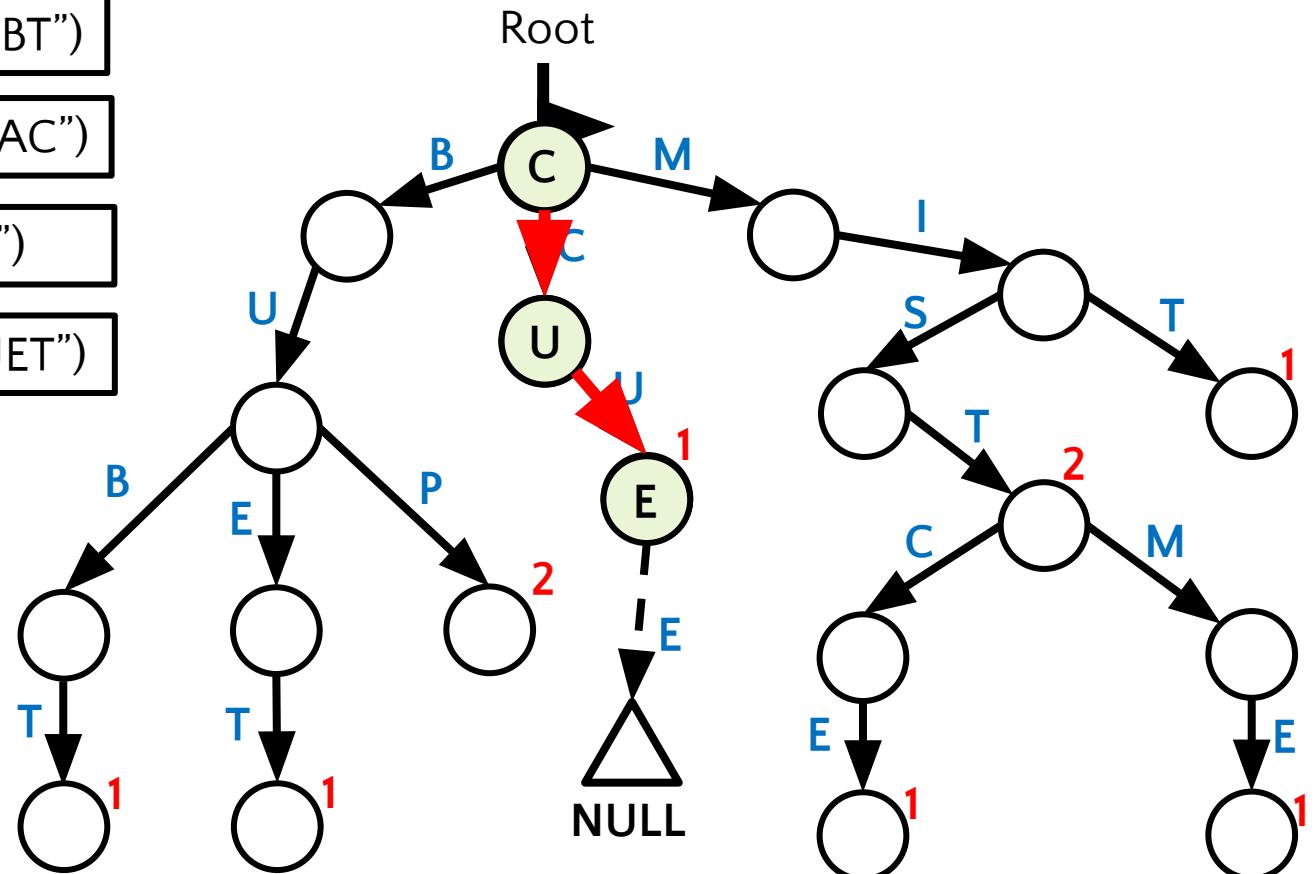


- insert("MIT")
 - insert("MIST")
 - insert("BUET")
 - insert("MISTCE")
 - insert("BUBT")
 - insert("MISTME")
 - insert("BUP")
 - insert("CU")
 - insert("MIST")
 - insert("BUP")

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

We reach to NULL

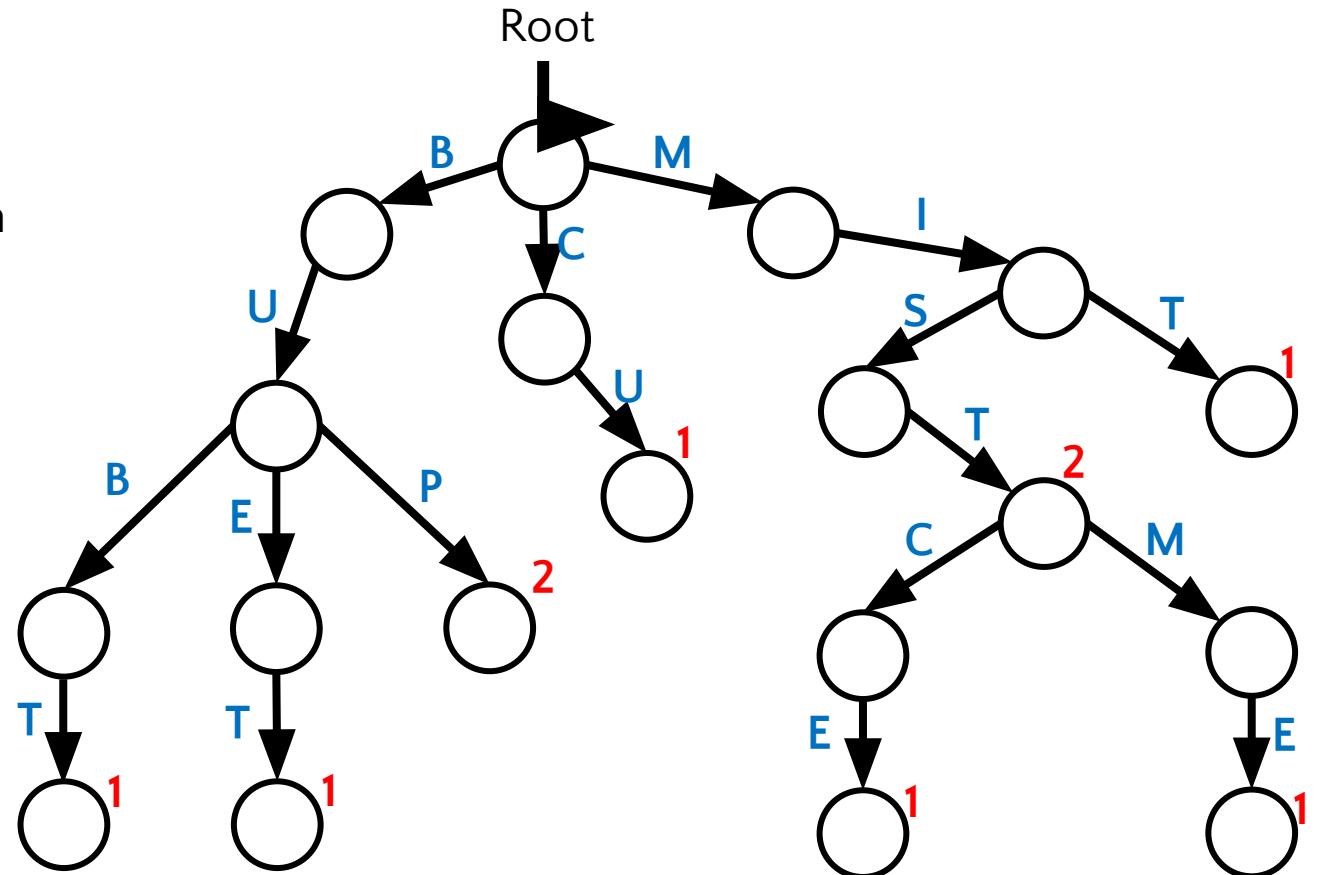
Means “CUET” doesn’t exist



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

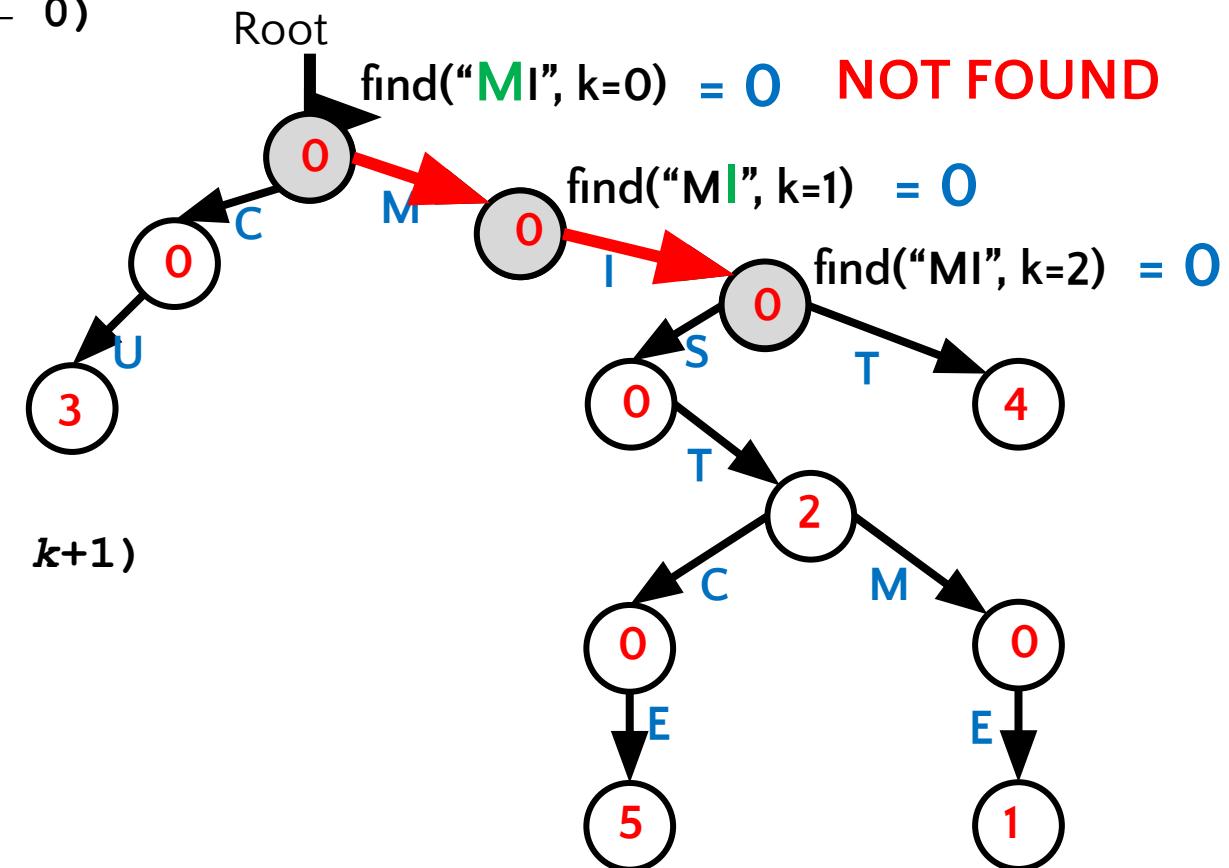
```
int relPos(char c){  
    int ascii = (int) c;  
    return ascii - 65;  
}
```

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")

```





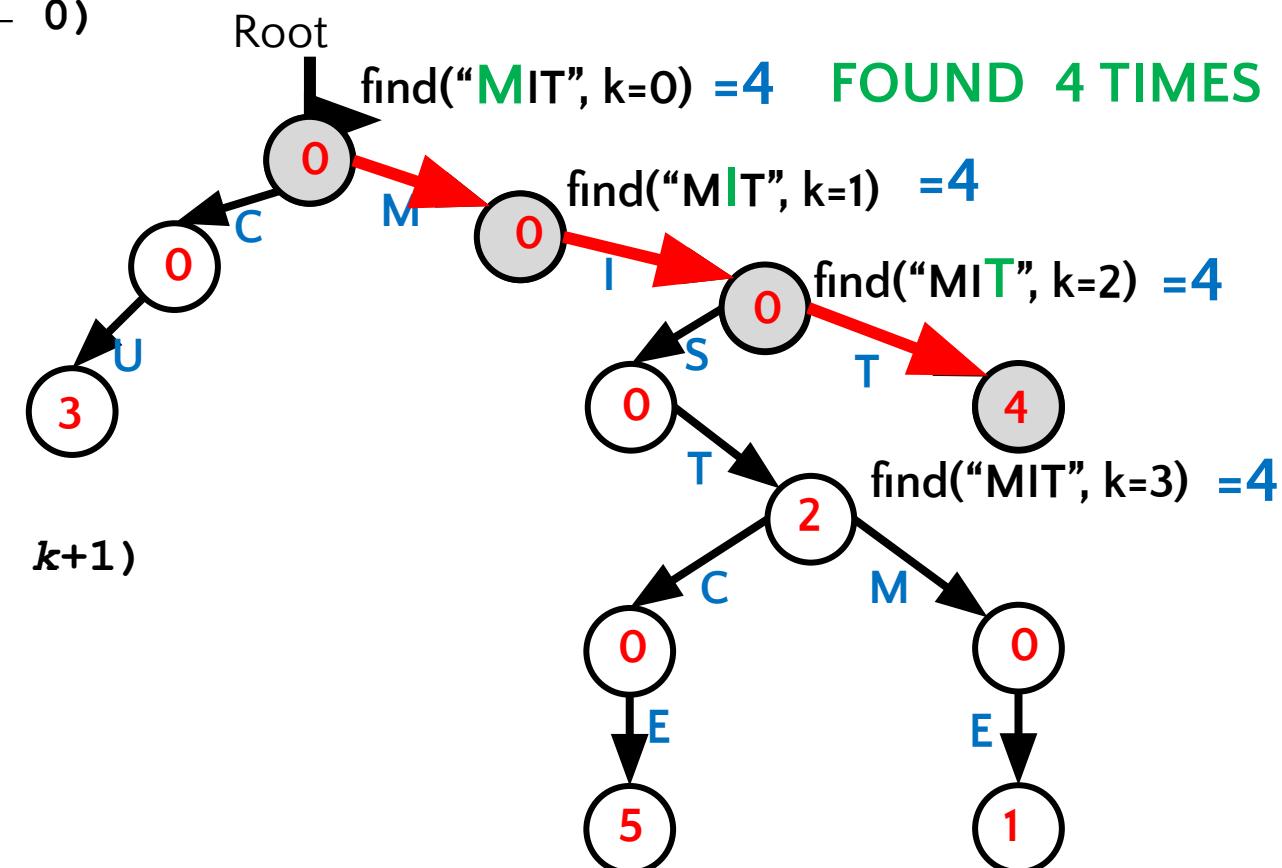
SEARCH IN TRIE

```

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")



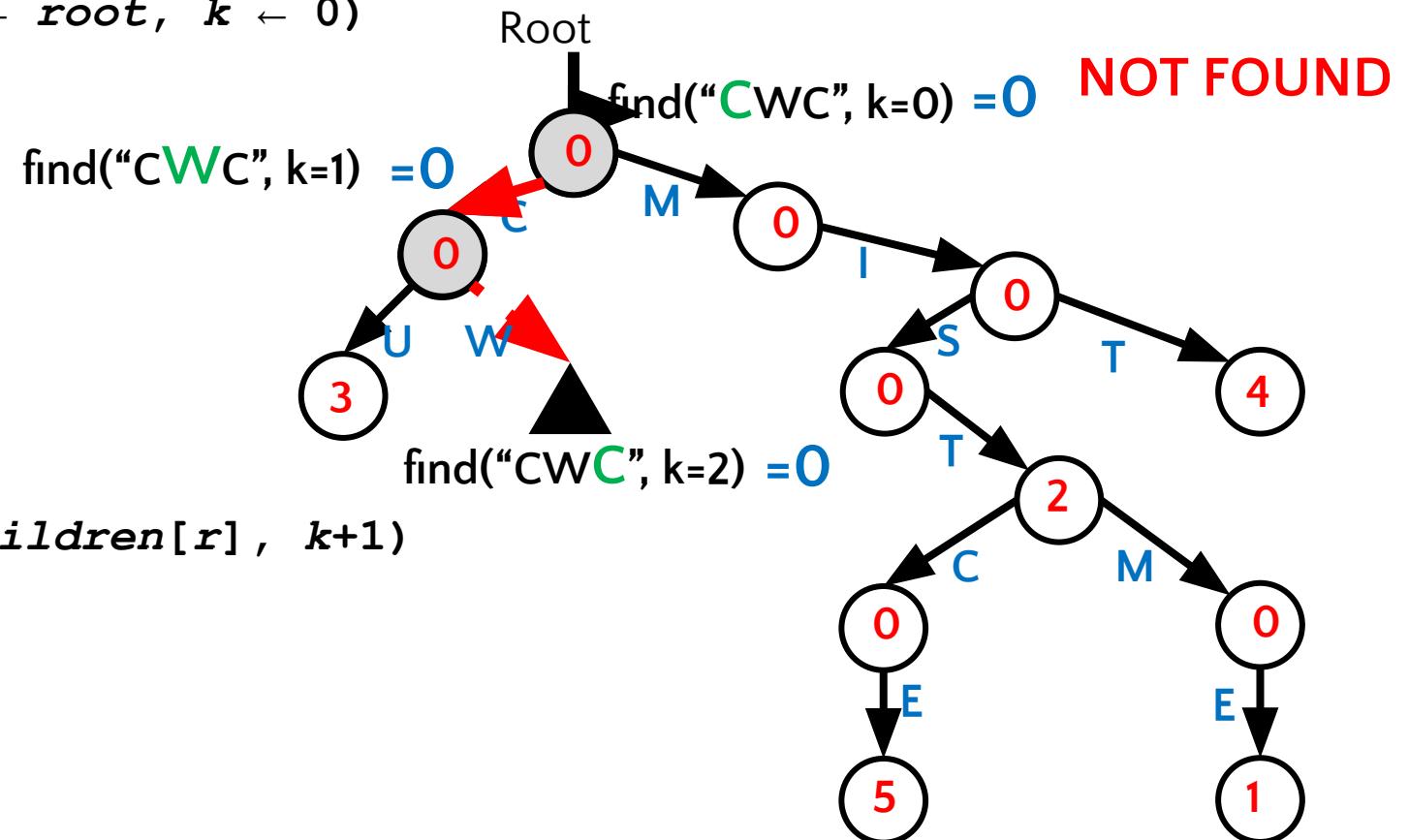
SEARCH IN TRIE

```

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

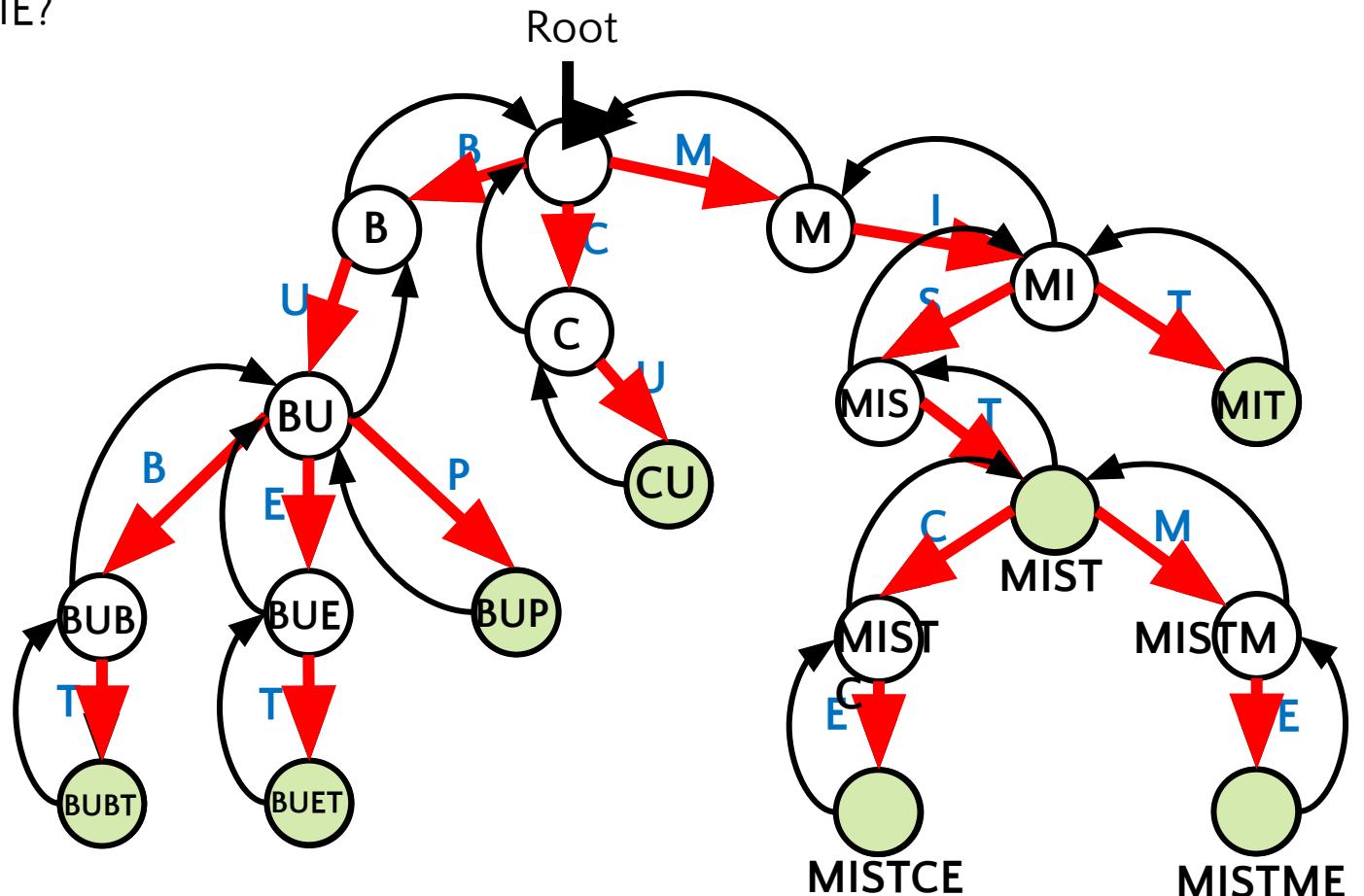
MISTCE

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);  
        }  
    }  
}
```

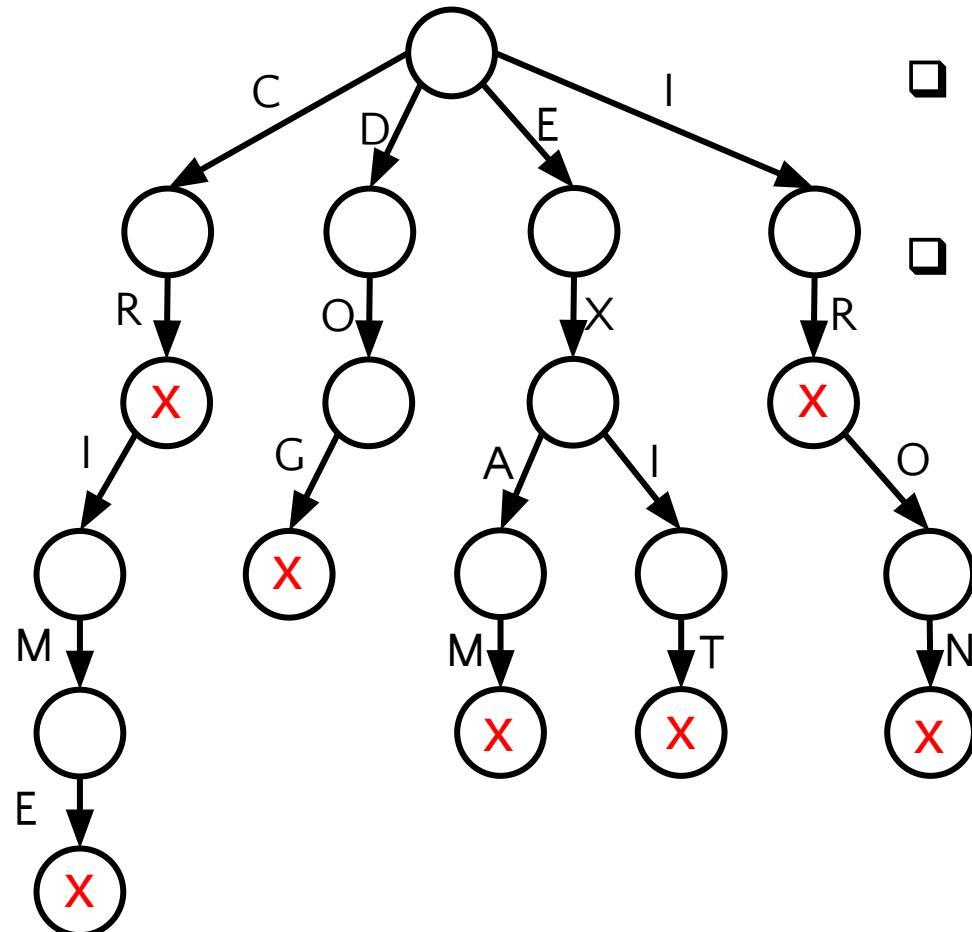
Base case:

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

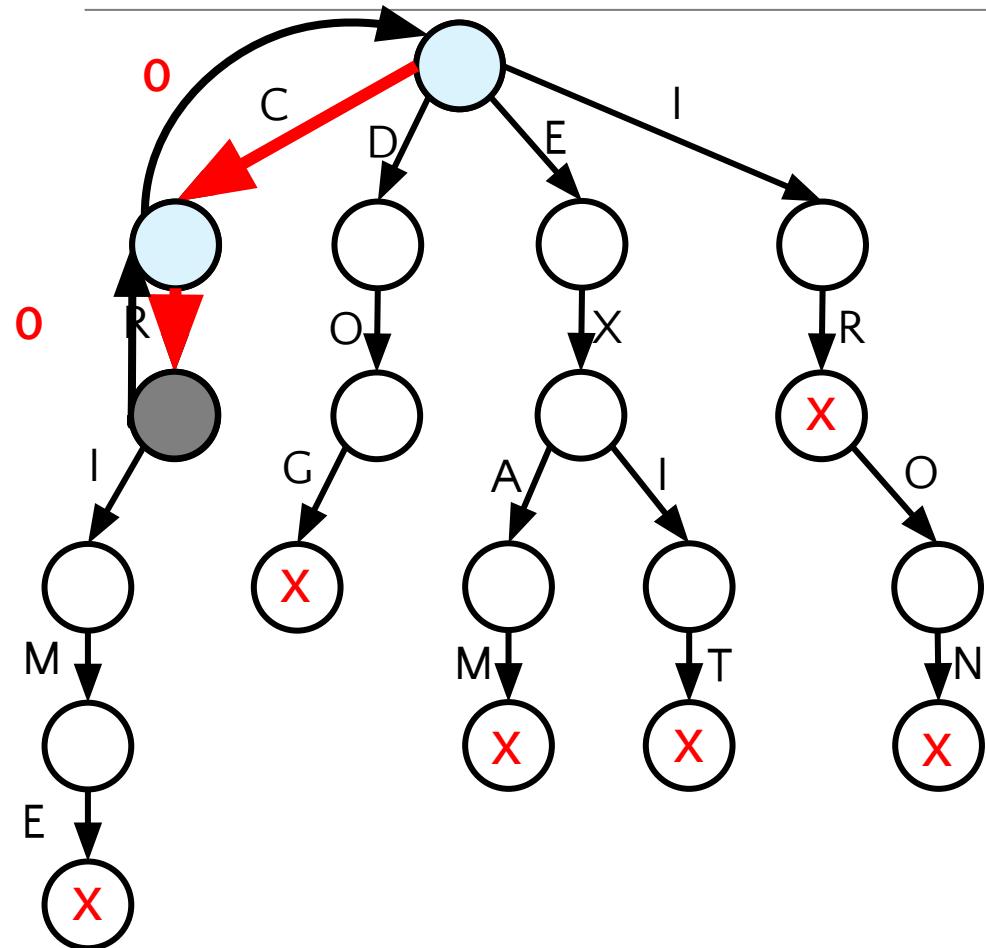
CR CRIME DOG EXAM EXIT IR IRON

□ 2 Cases for deletion

- The word is a prefix of other words
 - Ex CR IR : IRON
- 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)

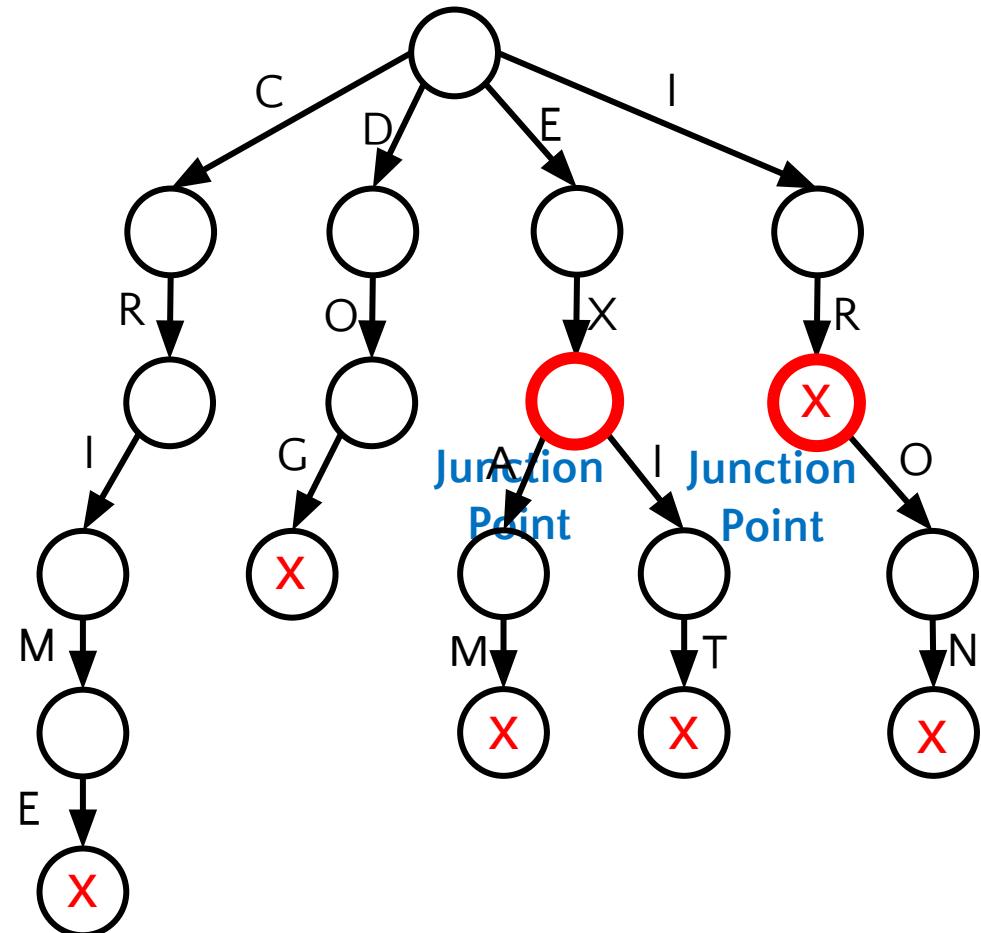


- ❑ The word is a prefix of other words
 - Simply remove the EoW mark from the final node of the string in TRIE
 - delete("CR")
 - ❑ How did we understand that “CR” is a prefix of other words?
 - Because the final node of CR in TRIE is not a leaf
 - ❑ How to check that whether a node is a leaf or not?
 - 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;
}

```

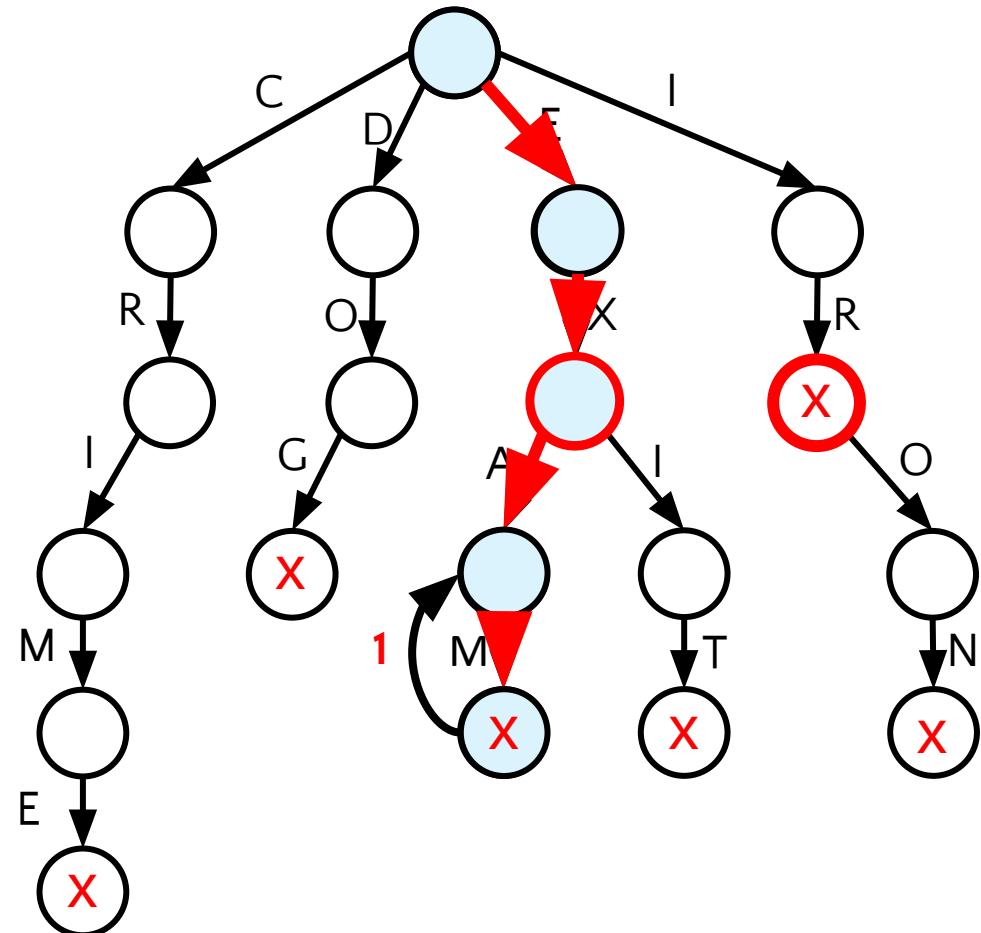
# DELETE FROM TRIE (CASE-2)



❑ 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")

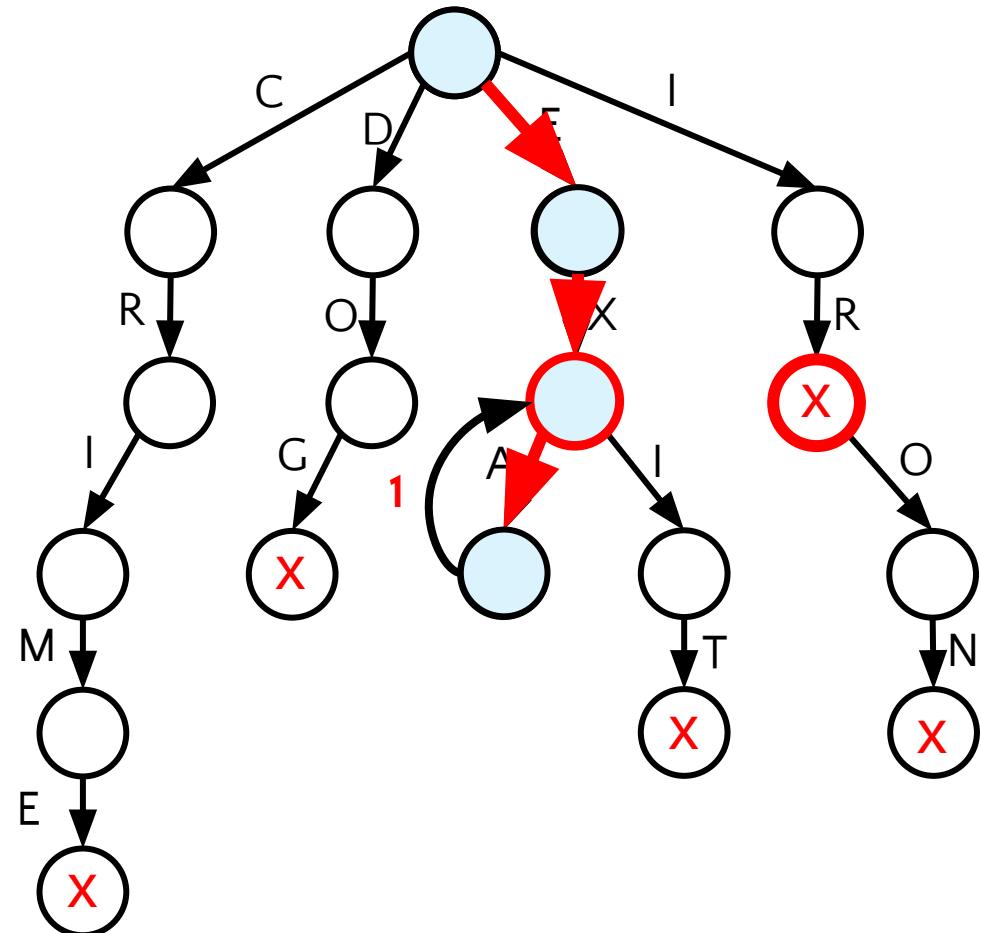
# DELETE FROM TRIE (CASE-2)



❑ 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")

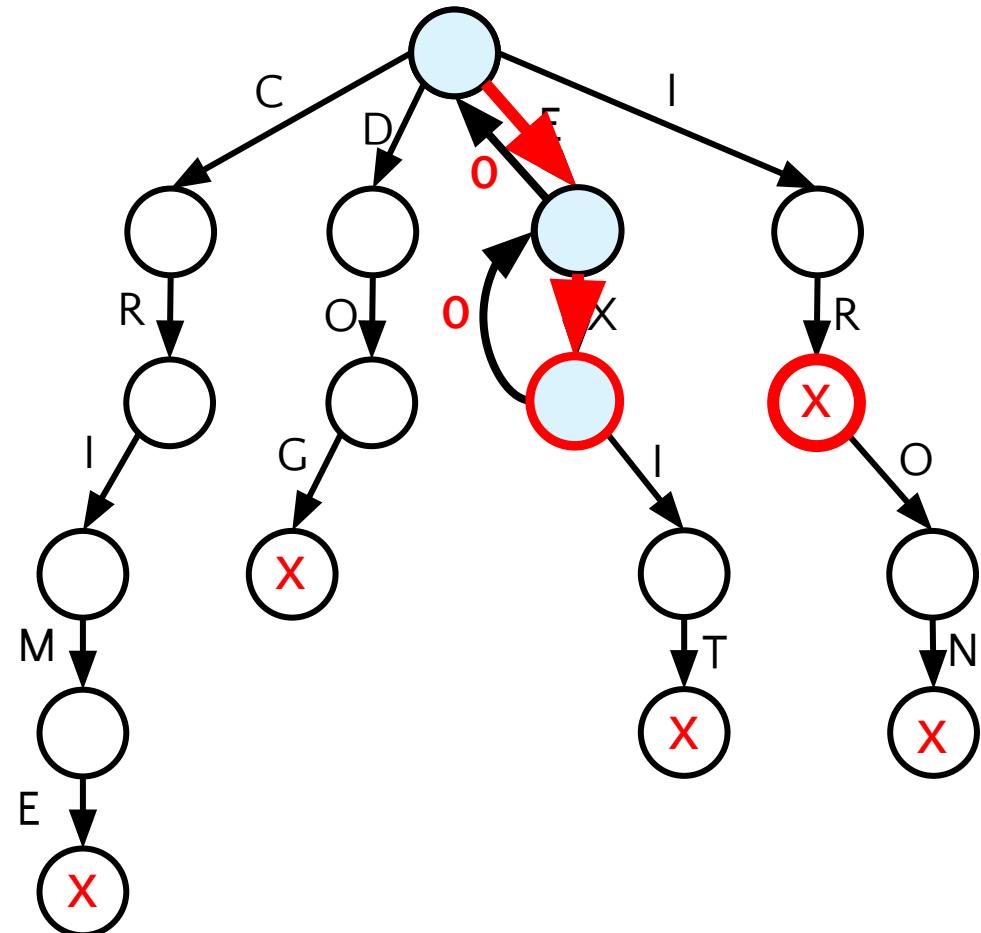
# DELETE FROM TRIE (CASE-2)



❑ 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")

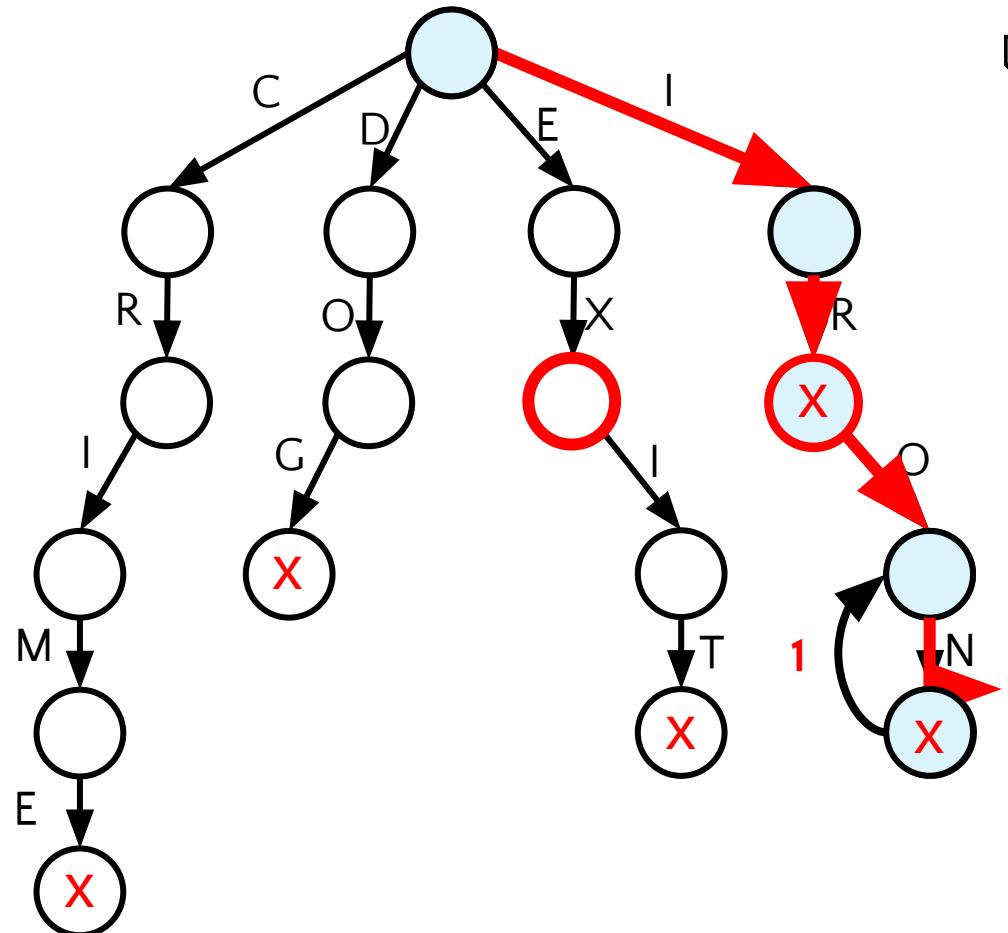
# DELETE FROM TRIE (CASE-2)



❑ 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")

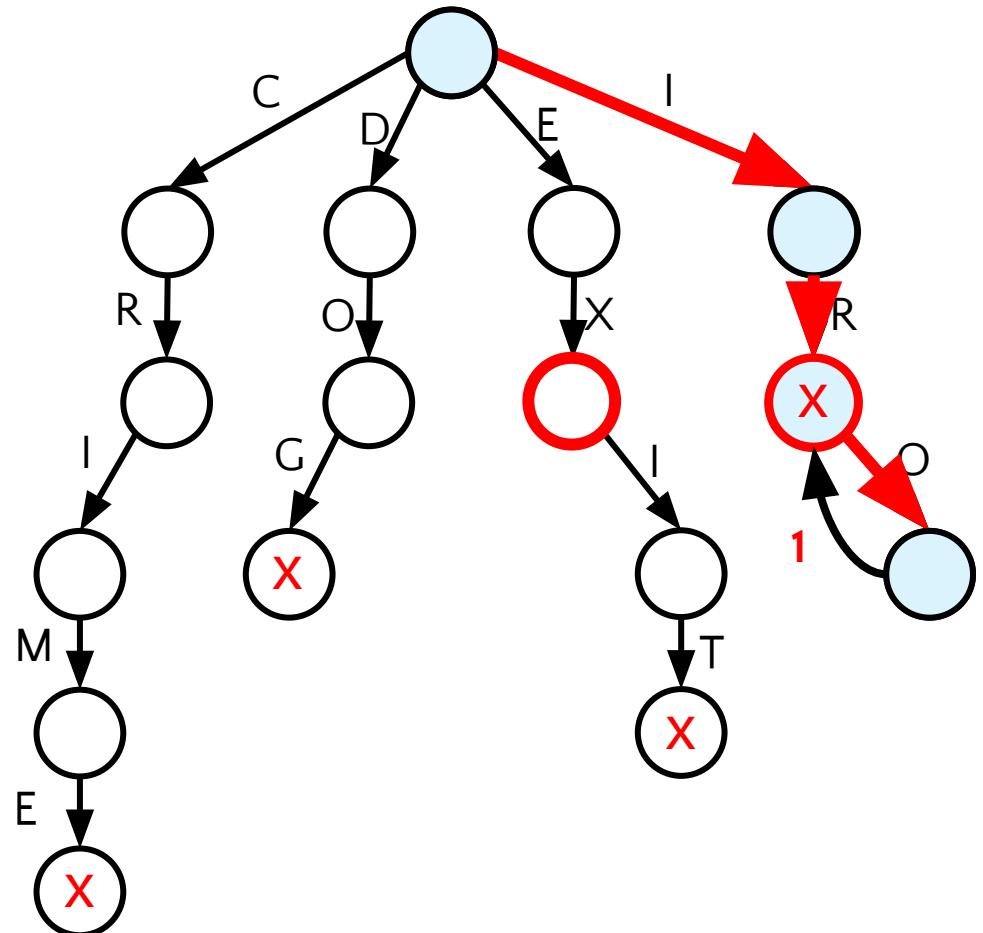
# DELETE FROM TRIE (CASE-2)



❑ 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")

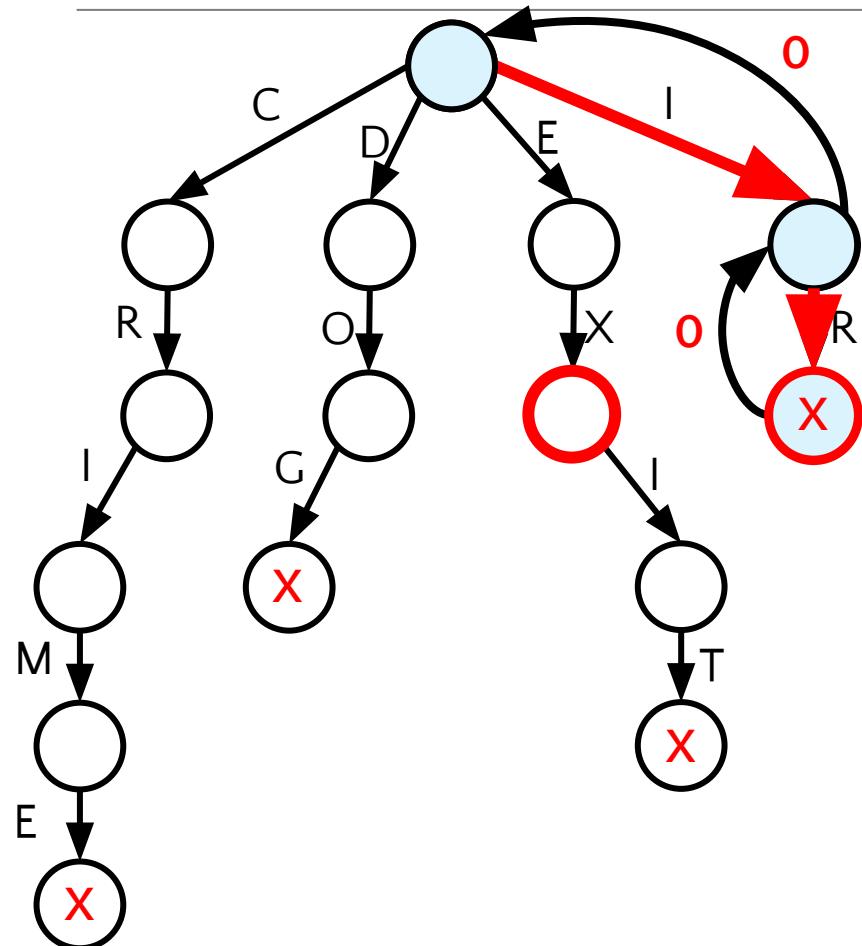
# DELETE FROM TRIE (CASE-2)



❑ 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)



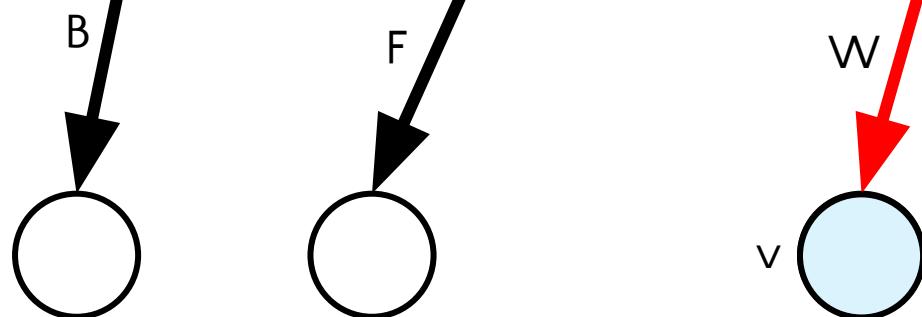
❑ The word is not a prefix of other words

- 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

U

| EoW      |   |   |   |   |   |     |   |    |    |    |    |
|----------|---|---|---|---|---|-----|---|----|----|----|----|
| children |   |   |   |   |   |     |   |    |    |    |    |
| A        | B | C | D | E | F | ... | U | V  | W  | X  | Y  |
| N        |   | N | N | N |   |     | N | N  |    | N  | Z  |
| 0        | 1 | 2 | 3 | 4 | 5 | .   | 2 | 21 | 22 | 23 | 24 |
|          |   |   |   |   |   | 0   |   |    |    |    | 25 |



DELETE THE RED MARKED EDGE

$r \leftarrow 22$

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

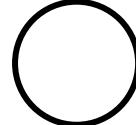
`u->children[r] = NULL`

# DELETION OF AN EDGE

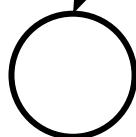
U

| EoW      |   |   |   |   |   |     |   |    |    |    |    |    |
|----------|---|---|---|---|---|-----|---|----|----|----|----|----|
| children |   |   |   |   |   |     |   |    |    |    |    |    |
| A        | B | C | D | E | F | ... | U | V  | W  | X  | Y  | Z  |
| N        |   | N | N | N |   | ... | N | N  | N  | N  | N  | N  |
| 0        | 1 | 2 | 3 | 4 | 5 | .   | 2 | 21 | 22 | 23 | 24 | 25 |

B



F



DELETE THE RED MARKED EDGE

$r \leftarrow 22$

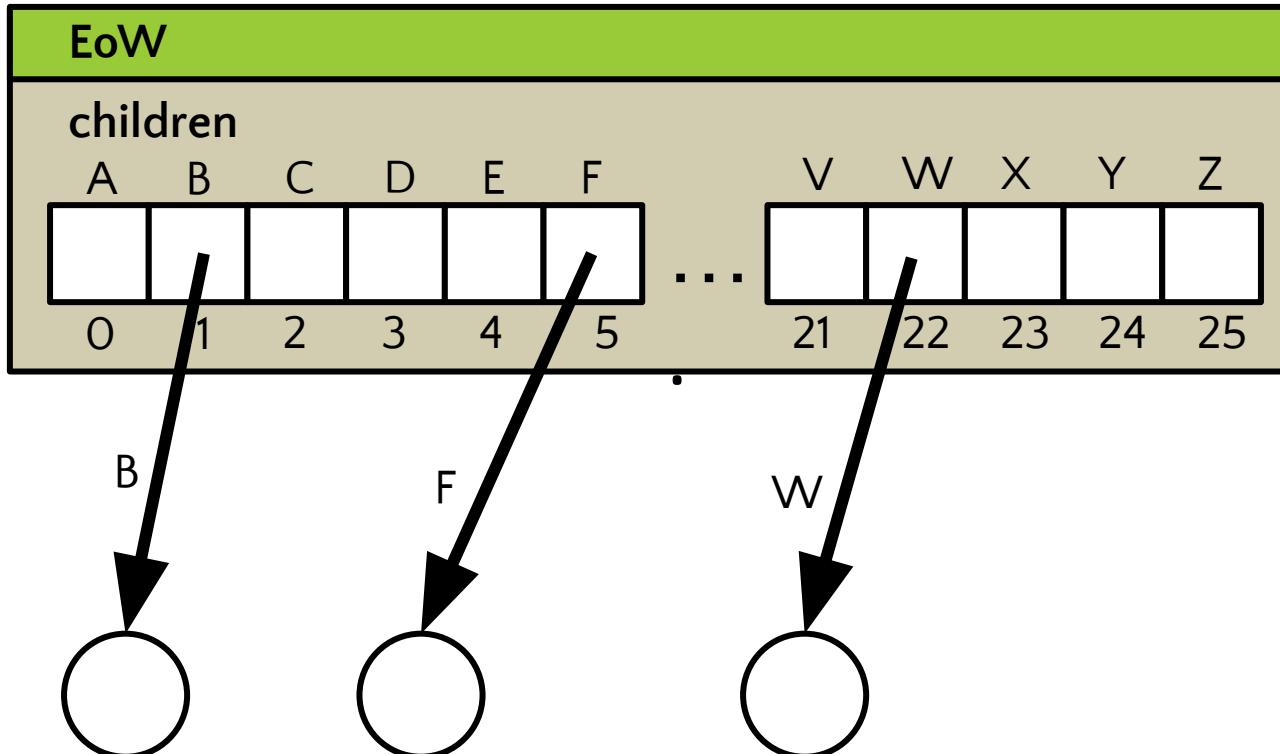
**Node \*v**  $\leftarrow u->\text{children}[r]$

$u->\text{children}[r] = \text{NULL}$

**delete v**

# DELETION OF AN EDGE

U



```

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

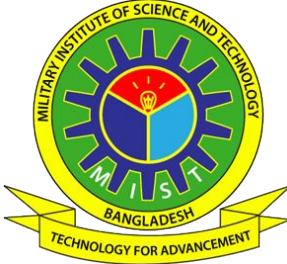


# JUNCTION POINT

---

- 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;
}
```



# Thank You!

---

# Dynamic Programming

0-1 Knapsack

# Contents

- What is dynamic Programming
- Remember the Fibonacci Series? : A basic intuition
- Dynamic Programming and Optimization Problems
  - 0/1 – Knapsack Problem

# What is Dynamic Programming?

Optimization problem is something that Maximizes or minimizes. For example Maximizing profit or minimizing travel cost.

Now, you can solve optimization problem using Greedy Programming. But the issue is, Greedy algorithm doesn't always provide correct solution or global optima. It might fall into local optima. Greedy algorithm doesn't ensure correct or optimized solution all the time.

So, what's the solution? **Dynamic Programming!**

# What is Dynamic Programming?

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have
  - **Overlapping sub problems**
  - **Optimal substructure property.**
- We will see what these properties mean soon.

# 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 sub problems, and the solution to the larger problem relies on the solutions to those sub problems, then the problem exhibits optimal substructure.
- **Example:**
- One classic example is the problem of finding the shortest path in a weighted directed graph. If you want to find the shortest path from node A to node B, and you know the shortest paths from A to intermediate nodes C and D, then the shortest path from A to B will be the minimum of (A to C + C to B) and (A to D + D to B).

# Overlapping sub problems

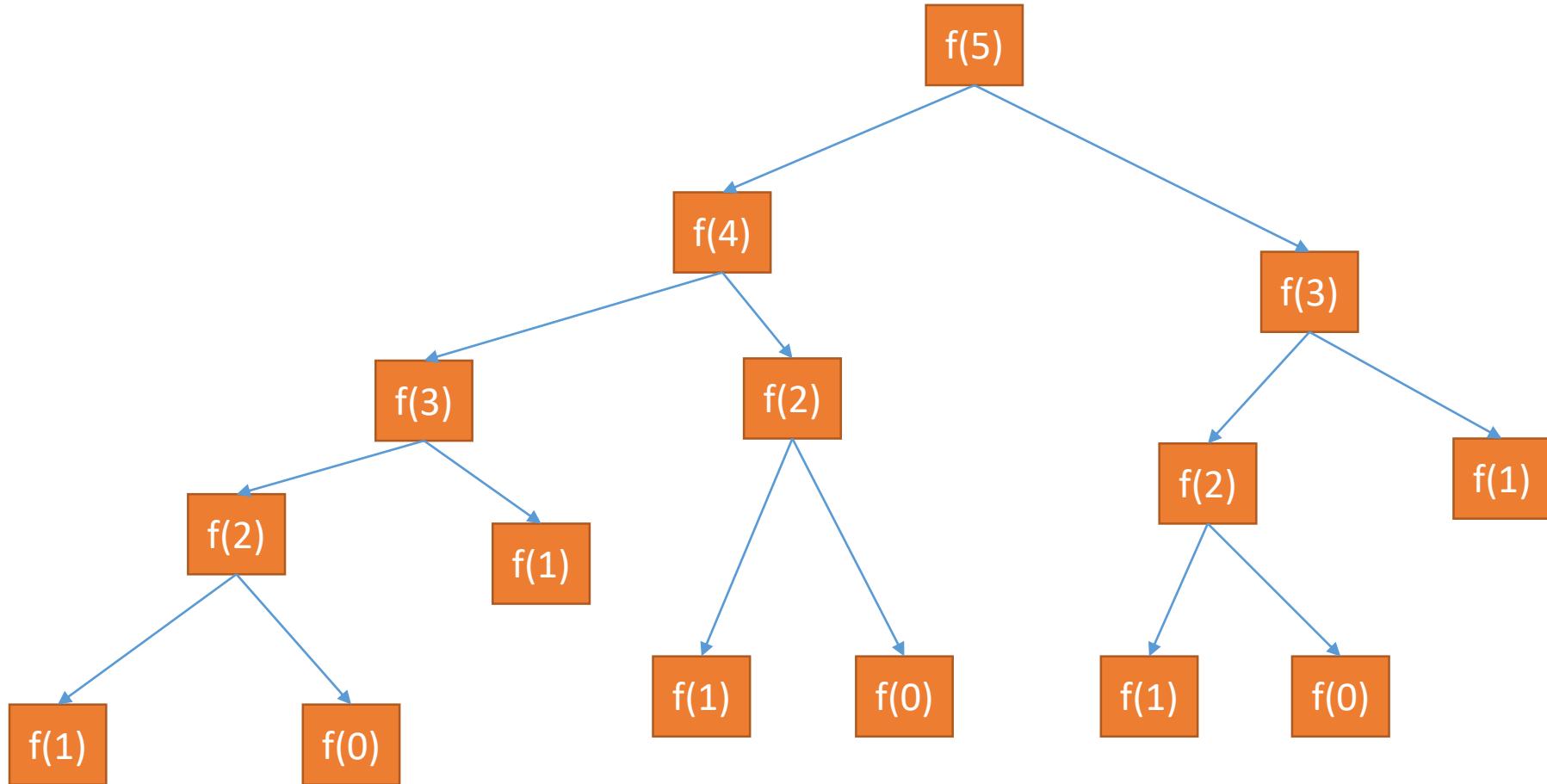
- Overlapping sub problems occur when a problem can be broken down into sub problems which are reused several times.
- This means that the same sub problem is solved multiple times in the process of solving the larger problem. Dynamic programming takes advantage of this property by solving each sub problem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

$$F(n) = F(n-1) + F(n-2)$$

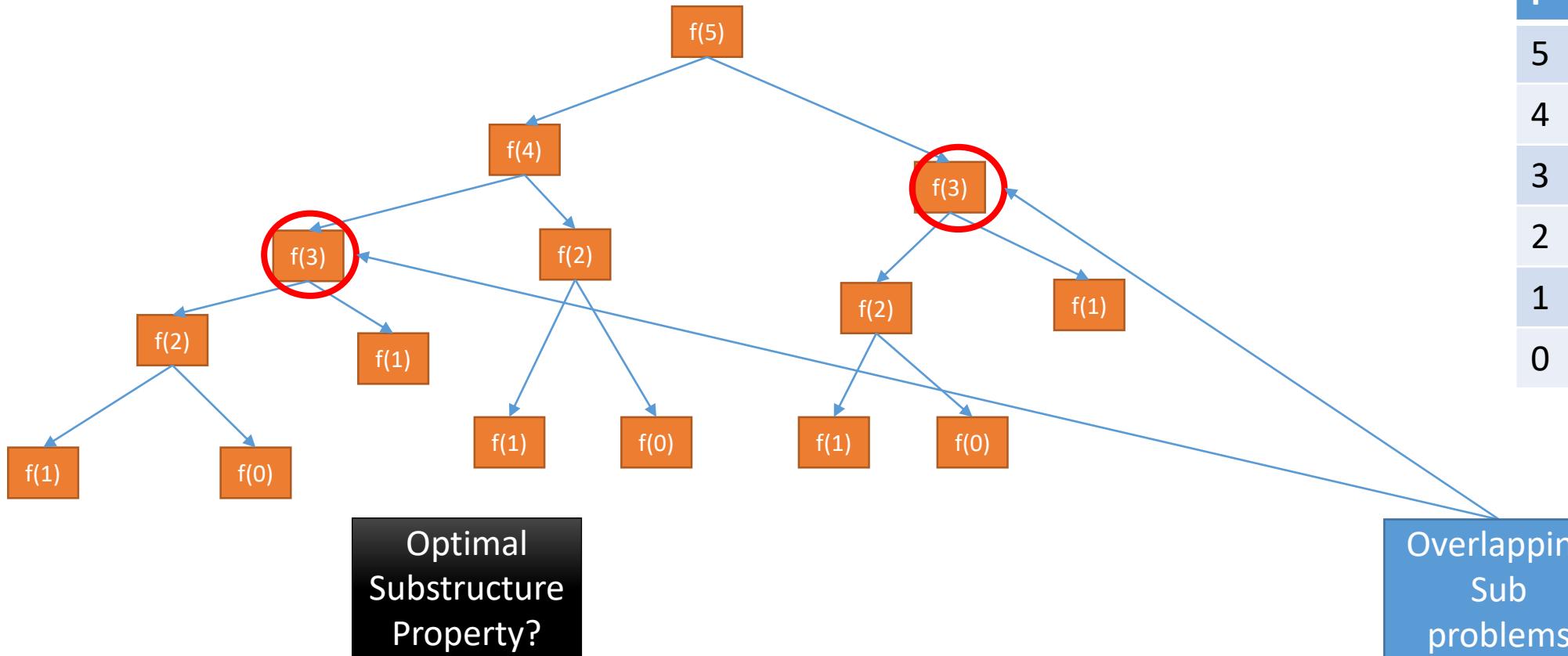
# Remember the Fibonacci Series?

A basic intuition about Dynamic Programming

# Fibonacci Series

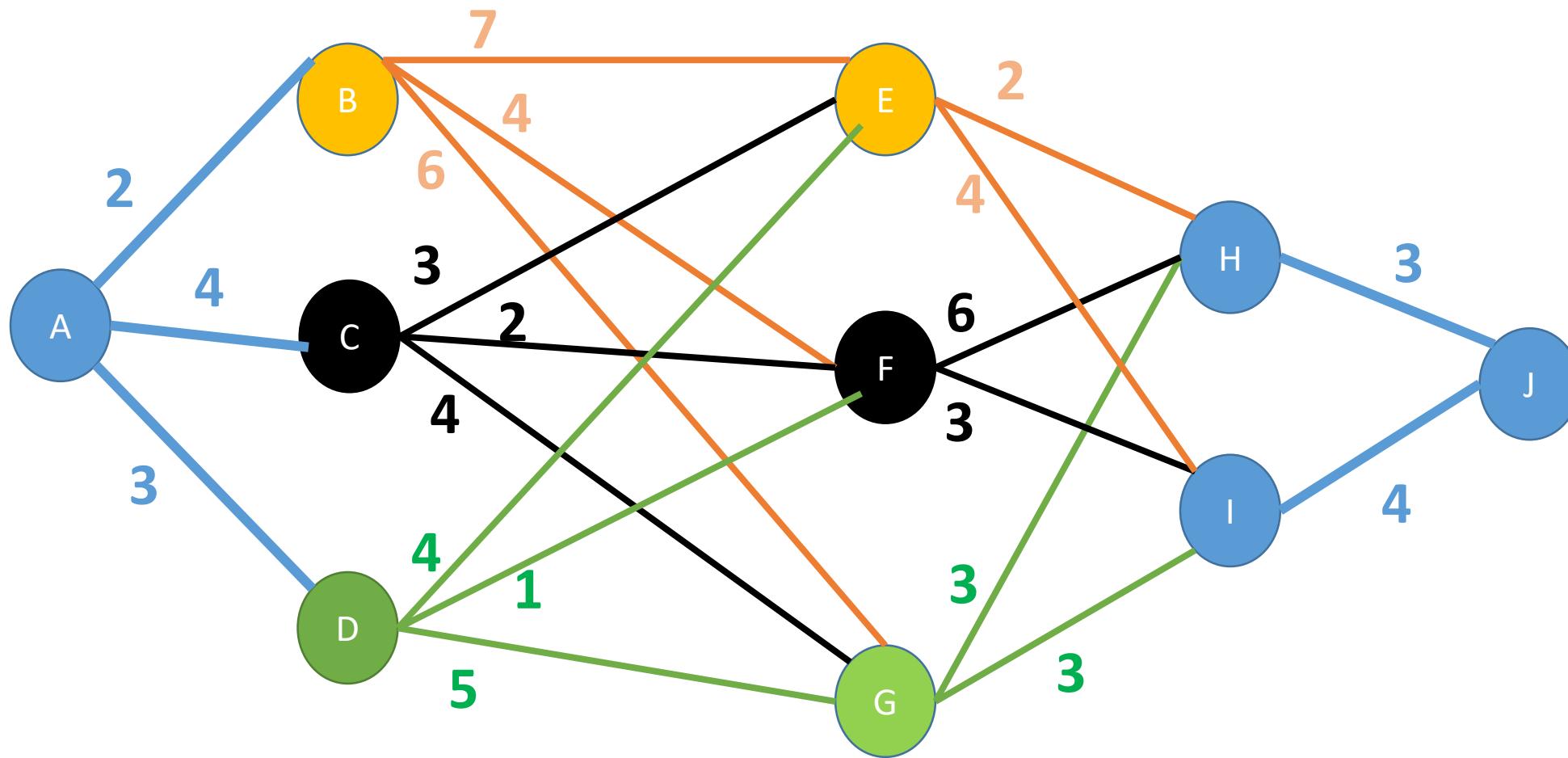


# Fibonacci Series with an extra table

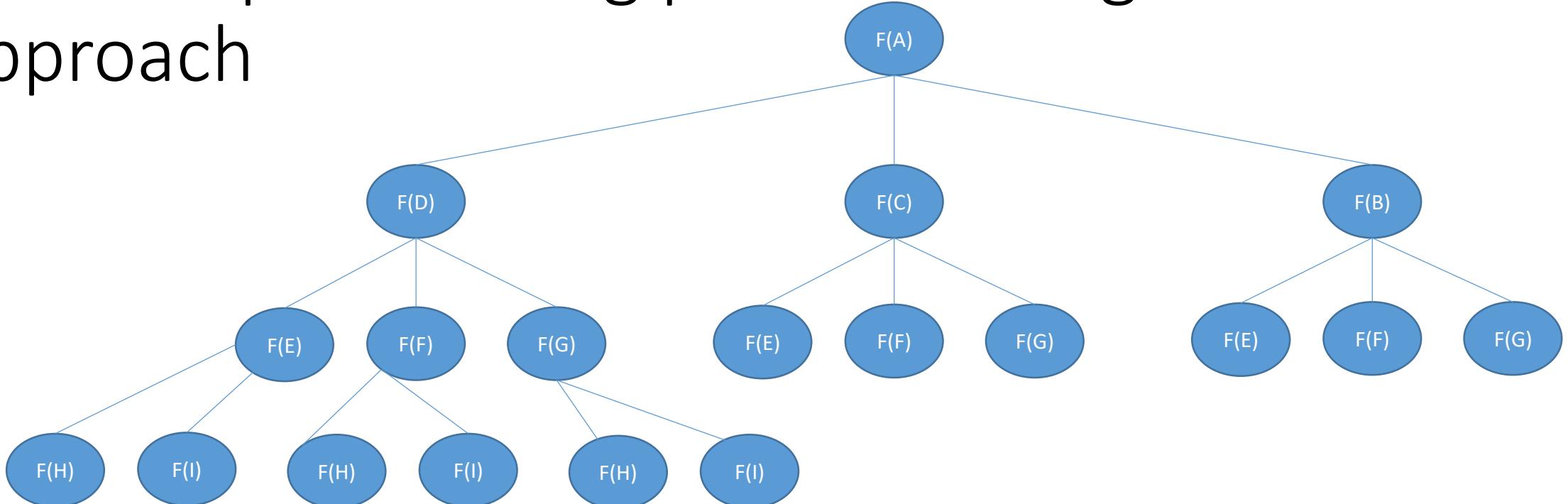


| F | value |
|---|-------|
| 5 | -1    |
| 4 | -1    |
| 3 | -1    |
| 2 | -1    |
| 1 | 1     |
| 0 | 1     |

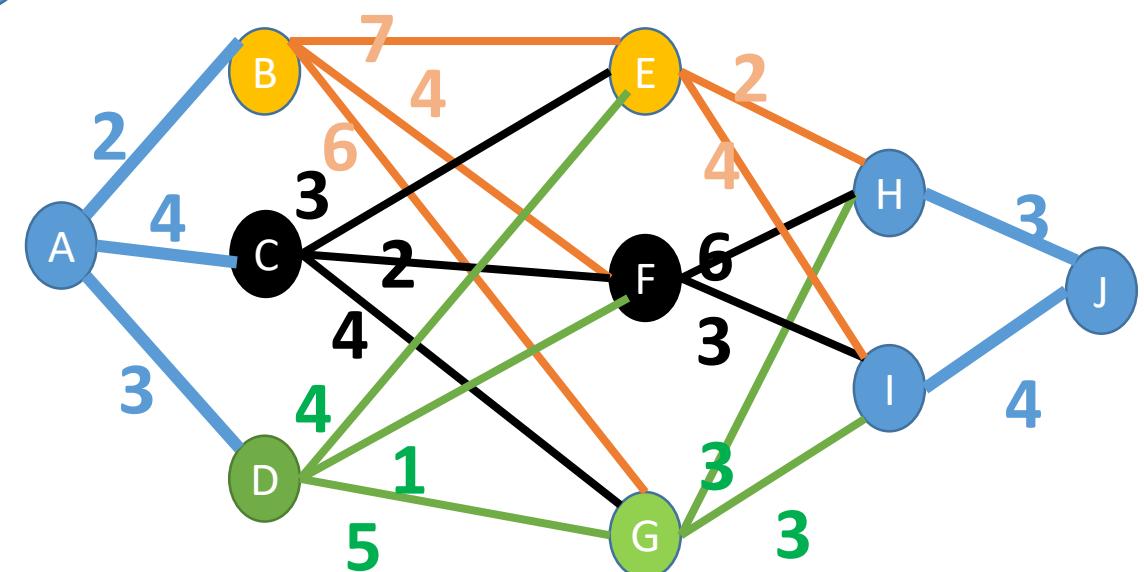
# Shortest path finding problem using recursive approach



# Shortest path finding problem using recursive approach



While solving each problem, from its Sub problems, we chose the one that gave The smallest value (optimal).  
This is an example of **optimal substructure property**



# Outcomes obtained so far

Till now we seem to have understood the following:

- We know about the **overlapping** sub problems property .
- We know about the **optimal** substructure property.
- We know that dynamic programming requires the use of a **table**, to avoid calling recursive functions

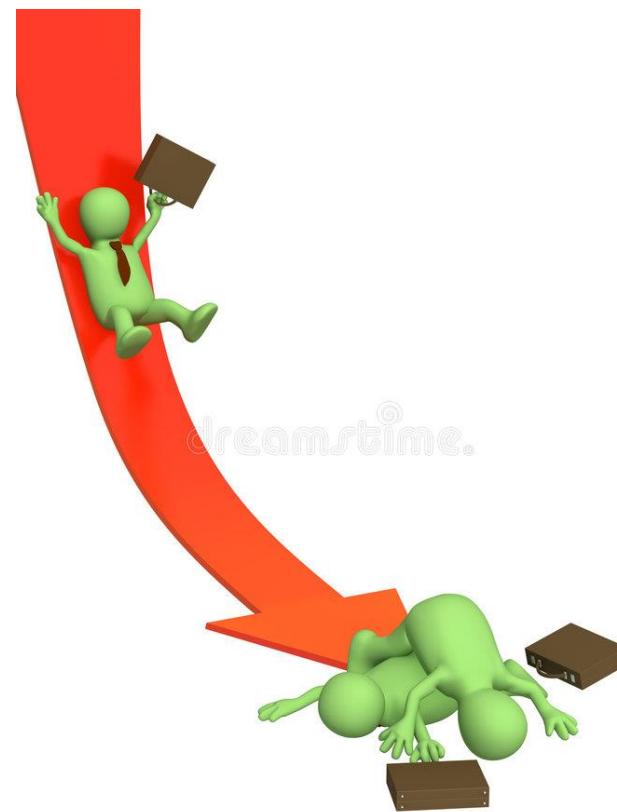
# Dynamic Programming and Optimization Problems

0/1 – Knapsack Problem

# Optimization Problems

For optimization problems 3 popular strategies are:

- Greedy Method
- Dynamic Programming Method
- Branch and Bound Method



# What is an optimization problem?

- You are studying Dynamic Programming. What is the **minimum** time you would require for finishing the chapter? (and how?)
- You are given a set of Dynamic Programming problems. What is the **maximum** score you can obtain from the test? (and how?)
- You are given an infinite number of coins with values 1, 2 and 5 taka. What is the **minimum** number of coins you can use to have 11 taka? (and how?)

# The 0/1 Knapsack Problem

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- Huge sentence ! Lets avoid reading this.

# 0/1 knapsack problem



\$ 3

2 Kg



\$ 4

3 Kg



\$ 5

4 Kg



\$ 6

5 Kg

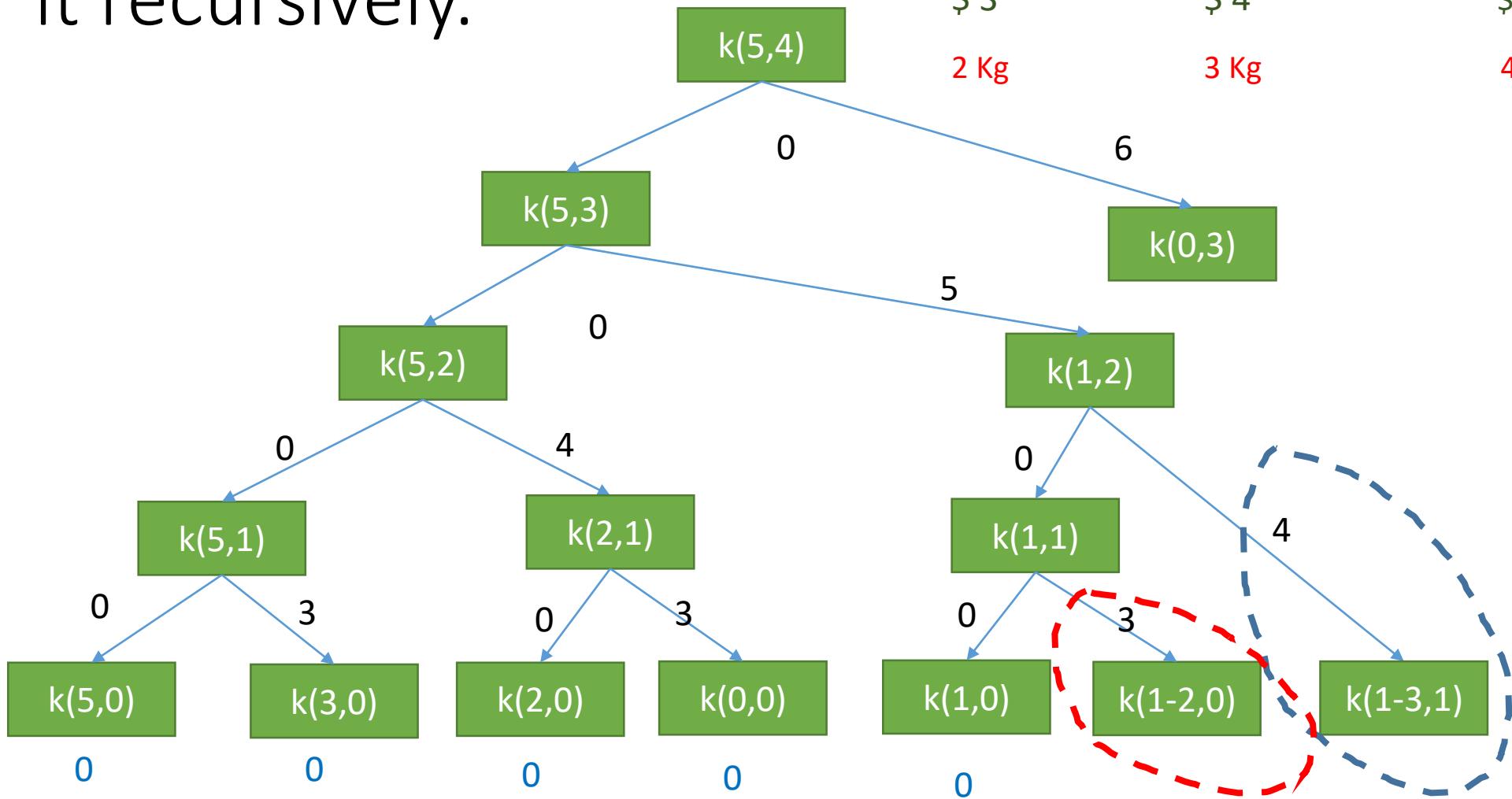


5 Kg

# Problem Formulation

- **How much maximum amount** can the thief with a bag with 5Kg of capacity can steal from the 4 items?
- K (5 kg, 4 items)
- K ( w, n )

# Yes, we can solve it recursively.



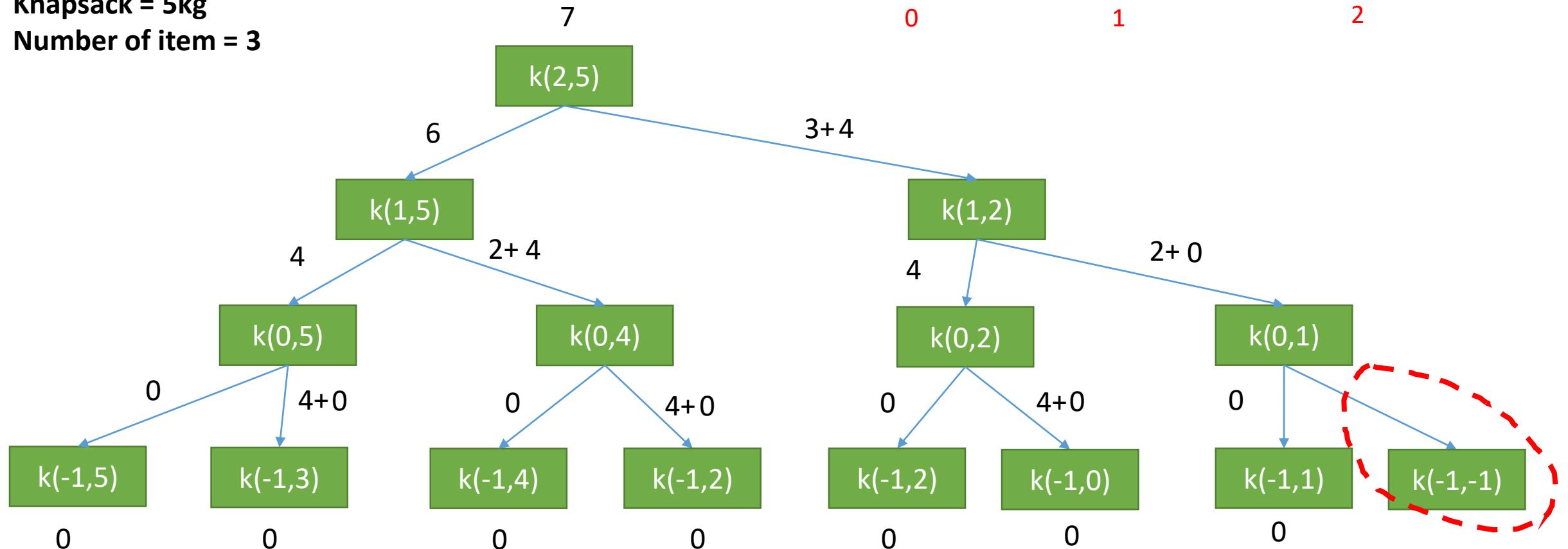
Problems with this approach:

1. Recursion takes **a lot of time** (does the same work again and again due to overlapping sub problems property).
2. No table, hence no way to trace the **sequence of actions**.

# Solve using Memoization

Knapsack = 5kg

Number of item = 3



\$ 4

2 Kg

0



\$ 2

1 Kg

1



\$ 3

3 Kg

2

# Let us formulate this solution using dynamic programming

The recursive equation:

- $K(w, i) = \max \{ k(w, i-1), \text{prices}[i] + k(w - \text{weights}[i], i-1) \}$
- Here:
  - $w$  = free space inside my bag
  - $i$  = the number of items left to steal/ the item I am about to steal
  - $\text{prices}$  = array containing the prices of the items we are thinking about stealing
  - $\text{weights}$  = array containing the weights of the items we are thinking about stealing
- There are two approaches for solving this issue using dynamic programming
  - Top Down (memorization) – starts from the root
  - Bottom Up – starts from the leaves

# Top Down Approach

- Using memoization
  - Start with declaration and initialization
  - Then call knapsack function passing (n-1, k)
  - Then print the output

```
int knapsack(int i, int j)
{
 if(i<0 || j<=0) return 0;
 if(dp[i][j]!=-1) return dp[i][j];
 int v1 = knapsack(i-1,j), v2=-1;
 if(w[i]<=j) v2 = p[i] + knapsack(i-1,j-w[i]);
 return dp[i][j] = max(v1, v2);
}
```

```
#include<bits/stdc++.h>
using namespace std;

int dp[2005][2005];
int c, n;
int p[2005],w[2005];
int main()
{
 cin>>c>>n;
 for(int i=0; i<n; i++) cin>>w[i]>>p[i];
 for(int i=0; i<2005; i++)
 for(int j=0; j<2005; j++)
 dp[i][j] = -1;

 cout<<knapsack(n-1,c)<<endl;
 for(int i=0; i<=n; i++)
 {
 for(int j=0; j<=c; j++)
 {
 cout<<dp[i][j]<<" ";
 }
 cout<<endl;
 }
}
```

# Top Down Approach

- Using memorization
  - Input and Output will be look like this

```
5 3
2 4
1 2
3 3
```

```
7
-1 0 4 -1 4 4
-1 -1 4 -1 -1 6
-1 -1 -1 -1 -1 7
-1 -1 -1 -1 -1 -1
```

# Bottom Up Approach

- We will determine the base cases first
- Then using the base cases we will build our solution.
- Unlike memoization we will work towards filling up the entire table

| prices  | 12 | 10 | 20 | 15 |   |
|---------|----|----|----|----|---|
| weights | 2  | 1  | 3  | 2  |   |
| n       | 0  | 1  | 2  | 3  | 4 |
| 0       |    |    |    |    |   |
| 1       |    |    |    |    |   |
| 2       |    |    |    |    |   |
| 3       |    |    |    |    |   |
| 4       |    |    |    |    |   |
| 5       |    |    |    |    |   |

The diagram illustrates a dynamic programming table for the knapsack problem. The columns represent prices (12, 10, 20, 15) and the rows represent weights (0, 1, 2, 3, 4, 5). A blue arrow labeled 'n' points from the top-left towards the bottom-right, indicating the direction of filling the table. The table cells are all filled with blue rectangles, representing the fact that every cell is being updated or considered in the bottom-up approach. The first row and column are also labeled with their respective values (0, 1, 2, 3, 4, 5).

# Bottom Up Approach

- We will at first fill up all the base cases

```
for (int i=0;i<=W;i++) {
 m[i][0]=0;
}

for (int j=0;j<=N;j++) {
 m[0][j]=0;
}
```

| prices  | 12 | 10 | 20 | 15 |   |
|---------|----|----|----|----|---|
| weights | 2  | 1  | 3  | 2  |   |
| n       | 0  | 1  | 2  | 3  | 4 |
| 0       | 0  | 0  | 0  | 0  | 0 |
| 1       | 0  |    |    |    |   |
| 2       | 0  |    |    |    |   |
| 3       | 0  |    |    |    |   |
| 4       | 0  |    |    |    |   |
| 5       | 0  |    |    |    |   |

The diagram illustrates the initial state of a dynamic programming table for the knapsack problem. The columns represent weights (n) from 0 to 4, and the rows represent values (w) from 0 to 5. A blue arrow labeled 'n' points to the column headers, and another blue arrow labeled 'w' points to the row headers. All cells in the table are orange and contain the value '0', indicating that no items have been selected for each weight and value combination.

# Bottom Up Approach

- Then we will traverse and fill up the cells of the table.
- Each cell will be based on the given formula
  - $m[w][i] = \max\{m[w][i-1], \text{prices}[i] + m[w - \text{weights}[i]][i-1]\}$
- Of course we have to check if  $w - \text{weights}[i] \geq 0$

$n$   
 $w$

| prices  | 12 | 10 | 20 | 15 |
|---------|----|----|----|----|
| weights | 2  | 1  | 3  | 2  |
| 0       | 0  | 1  | 2  | 3  |
| 0       | 0  | 0  | 0  | 0  |
| 1       | 0  | 0  | 10 | 10 |
| 2       | 0  | 12 | 12 | 12 |
| 3       | 0  | 12 | 22 | 22 |
| 4       | 0  | 12 | 22 | 30 |
| 5       | 0  | 12 | 22 | 32 |
|         |    |    |    | 37 |

# Time Complexity

- Naive Recursive Solution:
  - $O(2^n)$
- Bottom up approach:
  - $O(N*W)$
- Memoization
  - $O(N*W)$

Thank You

# CSE 216: Data Structures & Algorithms II

## Sessional



### — Single Source Shortest Paths —

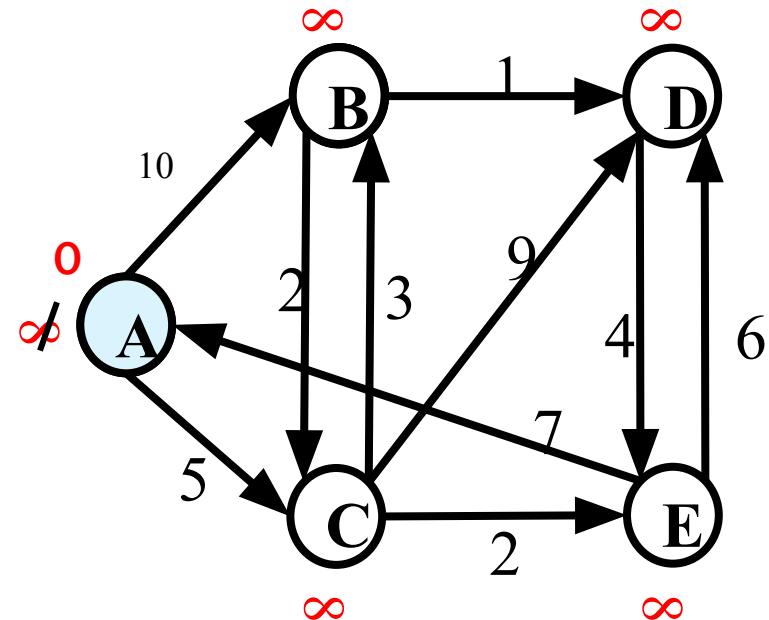
Lec Sumaiya Nuha Mustafina  
Dept Of CSE  
[sumaiyamustafina@cse.mist.ac.bd](mailto:sumaiyamustafina@cse.mist.ac.bd)



# Shortest Path Concepts

**INITIALIZE-SINGLE-SOURCE( $G, s$ )**

```
1 for each vertex $v \in G.V$
2 $v.d = \infty$
3 $v.\pi = \text{NIL}$
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  $v.\pi$

**RELAX( $u, v, w$ )**

- 1   **if**  $v.d > u.d + w(u, v)$
- 2        $v.d = u.d + w(u, v)$
- 3        $v.\pi = u$

# Dijkstra Algorithm

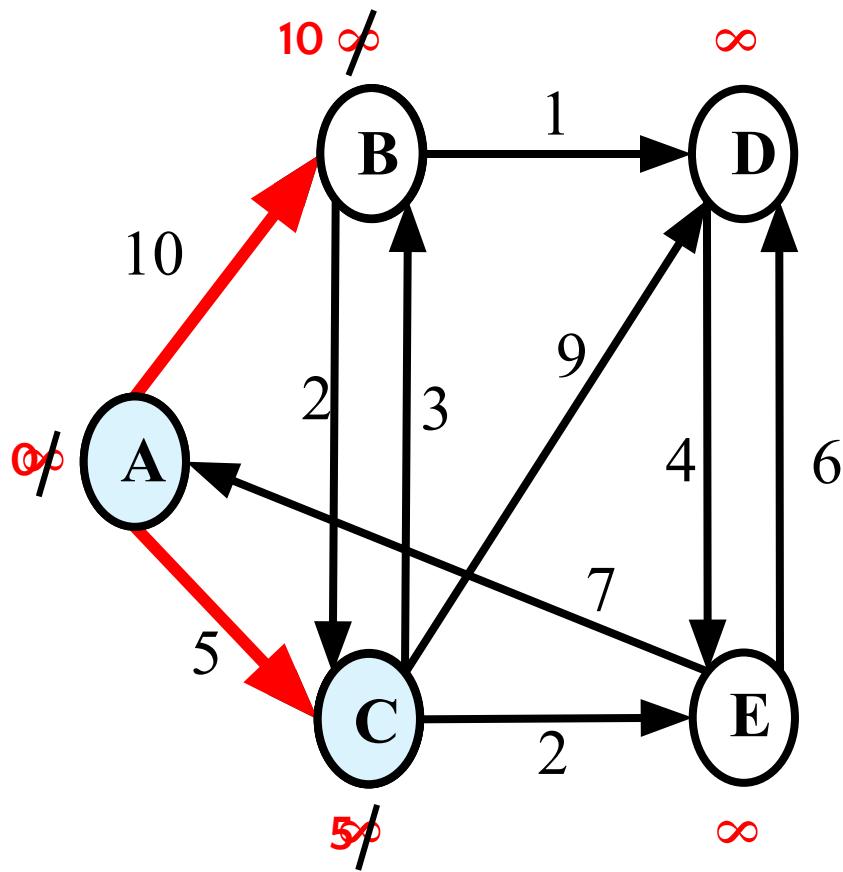
**DIJKSTRA**( $G, w, s$ )

```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S = \emptyset$
3 $Q = G.V$
4 while $Q \neq \emptyset$
5 $u = \text{EXTRACT-MIN}(Q)$
6 $S = S \cup \{u\}$
7 for each vertex $v \in G.Adj[u]$
8 RELAX(u, v, w)
```

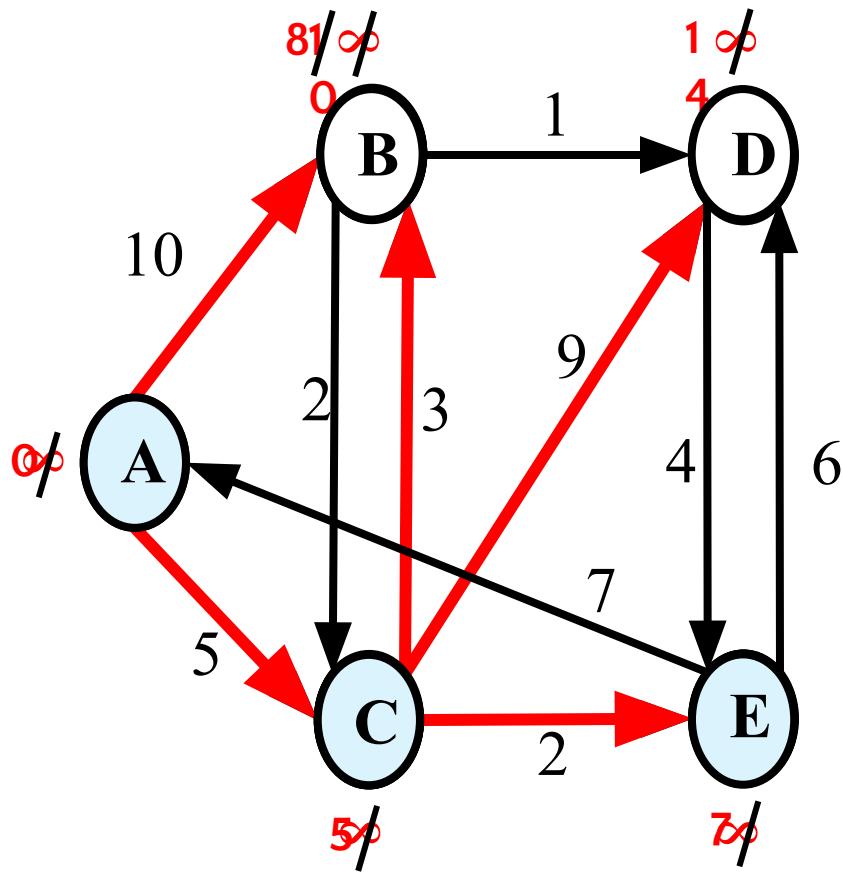
$Q$  : Min Priority Queue with priority value based on  $d$ .

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

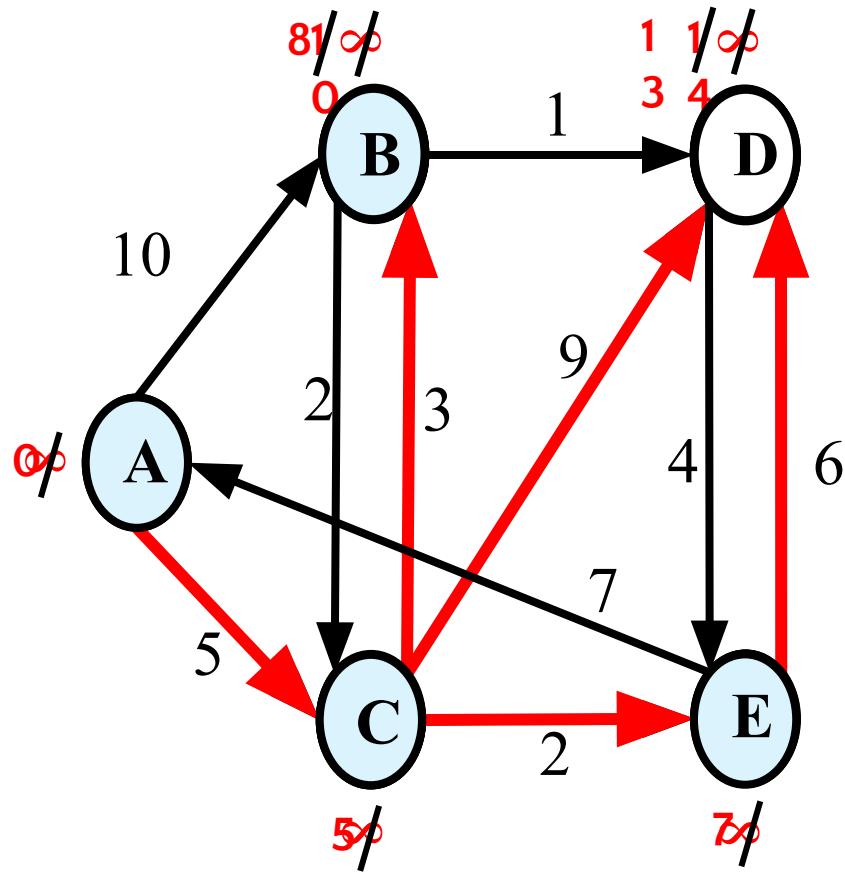
# DIJKSTRA's SIMULATION



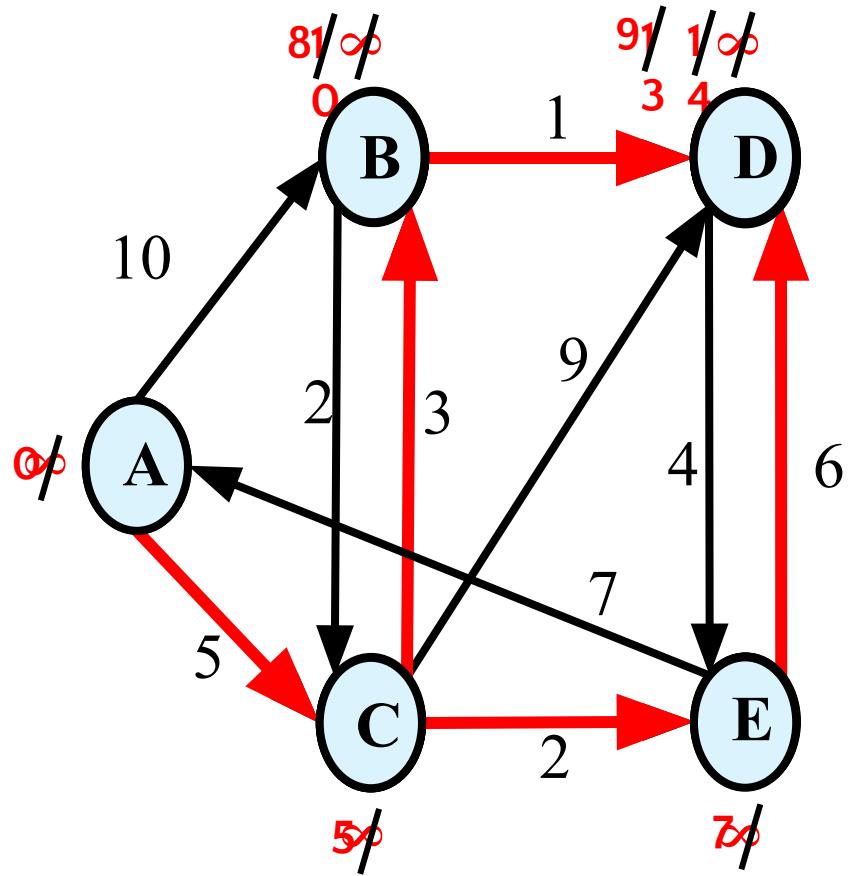
# DIJKSTRA's SIMULATION



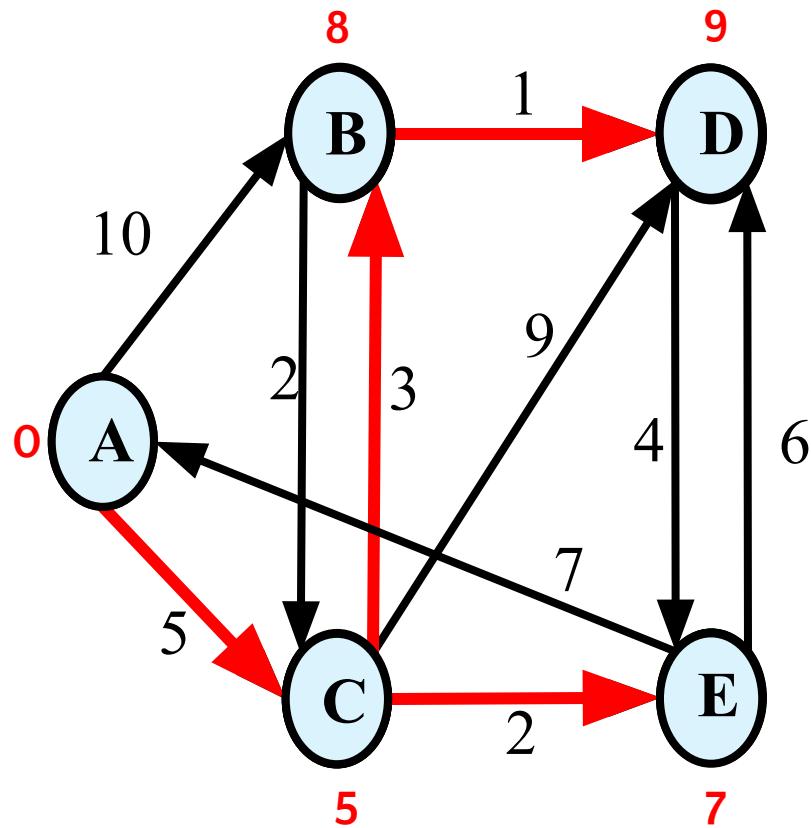
# DIJKSTRA's SIMULATION



# DIJKSTRA's SIMULATION

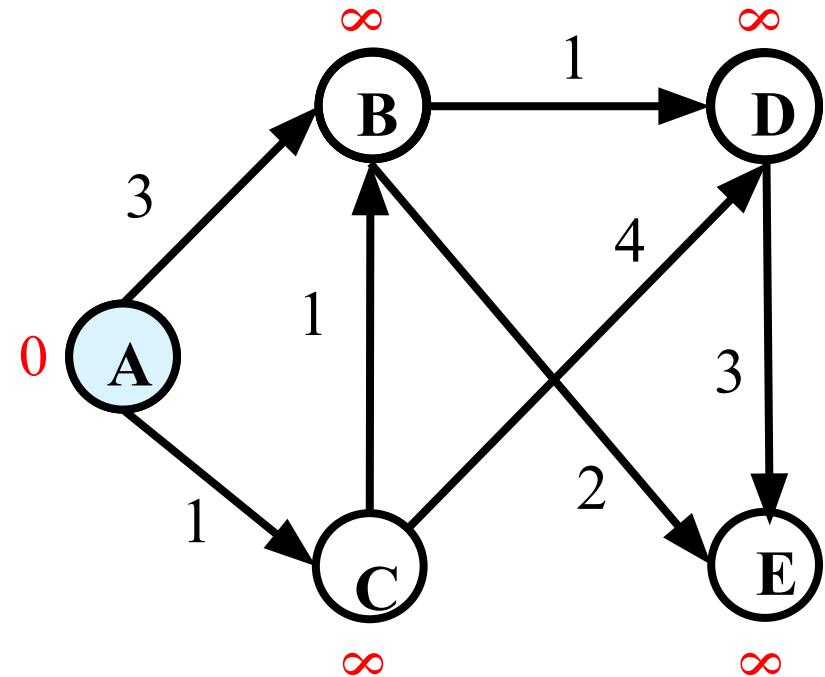


# DIJKSTRA's SIMULATION



# 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 $(u, v) \in G.E$
4 RELAX(u, v, w)
5 for each edge $(u, v) \in G.E$
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 $(u, v) \in G.E$
4 RELAX(u, v, w)
5 for each edge $(u, v) \in G.E$
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

thank  
you

# Dynamic Programming

---

MATRIX CHAIN MULTIPLICATION

# Matrix Chain Multiplication Problem

Multiplying non-square matrices:

- $A$  is  $p \times q$ ,  $B$  is  $q \times r$
- $AB$  is  $p \times r$  whose  $(i, j)$  entry is  $\sum a_{ik} b_{kj}$

Must be equal

Computing  $AB$  takes  $p \cdot q \cdot r$  scalar multiplications and  $p(q-1)r$  scalar additions (using basic algorithm).

Suppose we have a sequence of matrices to multiply. What is the best order?

# Matrix Chain Multiplication Problem

Given a sequence of matrices  $A_1, A_2, \dots, A_n$ , then

Compute  $C = A_1 \cdot A_2 \cdot \dots \cdot A_n$

Different ways to compute  $C$

$$C = (A_1 A_2)((A_3 A_4)(A_5 A_6))$$

$$C = (A_1 (A_2 A_3)(A_4 A_5)) A_6$$

- Matrix multiplication is associative
  - So output will be the same
- However, time cost can be very different
  - Example

# Why Order Matters

---

Suppose we have 4 matrices:

- $A, 30 \times 01$
- $B, 01 \times 40$
- $C, 40 \times 10$
- $D, 10 \times 25$

$((AB)(CD))$  : requires 41,200 multiplications

$$[ (30 \times 1 \times 40) + (40 \times 10 \times 25) + (30 \times 40 \times 25) = 41,200 ]$$

$(A((BC)D))$  : requires 1400 multiplications

$$[ (1 \times 40 \times 10) + (1 \times 10 \times 25) + (30 \times 1 \times 25) = 1,400 ]$$



# Matrix Chain Multiplication Problem

Given a sequence of matrices  $A_1, A_2, \dots, A_n$ , where  $A_i$  is  $p_{i-1} \times p_i$ :

- What is minimum number of scalar multiplications required to compute  $A_1 \cdot A_2 \cdot \dots \cdot A_n$ ?
- What order of matrix multiplications achieves this minimum?
- Fully parenthesize the product in a way that minimizes the number of scalar multiplications

(( )( ))(( )( ( )( )( )))

No. of parenthesizations: ???

# A Possible Solution

---

Try all possibilities and choose the best one.

Drawback is there are too many of them (exponential in the number of matrices to be multiplied)

The number of parenthesizations is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is  $\Omega(2^n)$

No. of parenthesizations: **Exponential**

Need to be more clever - try dynamic programming !

# Step 1: Optimal Substructure Property

---

A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.

Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We must also take care to ensure that the total number of distinct subproblems is a polynomial in the input size.

# Step 1: Optimal Substructure Property

---

Define  $A_{i..j}$ ,  $i \leq j$ , to be the matrix that results from evaluating the product  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ .

If the problem is nontrivial, i.e.,  $i < j$ , then to parenthesize  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ , split the product between  $A_k$  and  $A_{k+1..j}$  for some  $k$ , where  $i \leq k < j$ .

- The cost of parenthesizing this way is
  - The cost of computing the matrix  $A_{i..k}$  +
  - The cost of computing the matrix  $A_{k+1..j}$  +
  - The cost of multiplying them together

- The optimal substructure of this problem is:

An optimal parenthesization of  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  contains within it optimal parenthesizations of  $A_i \cdot A_{i+1} \cdot \dots \cdot A_k$  and  $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_j$

**Proof ?**

# Overlapping Subproblem Property

---

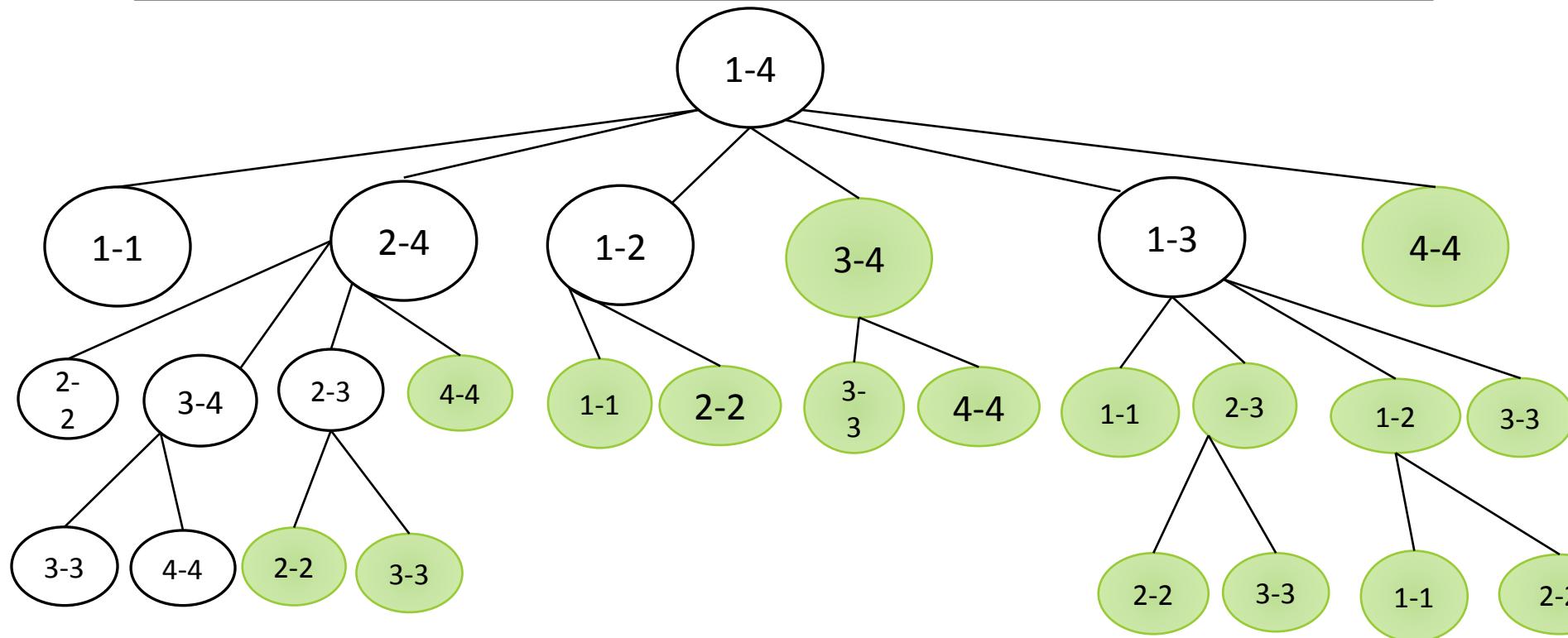
Two subproblems of the same problem are **independent** if they do not share resources.

Two subproblems are ***overlapping*** if they are really the same subproblem that occurs as a subproblem of different problems.

A problem exhibits ***overlapping subproblem*** if the number of subproblems is “small” in the sense that a recursive algorithm solves the same subproblems over and over, rather than always generating new subproblems.

Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.

# Overlapping Subproblem Property



# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

m represents cost of multiplication

$$A_1 \times A_1 \quad m[1,1] = 0;$$

$$A_2 \times A_2 \quad m[2,2] = 0;$$

$$A_3 \times A_3 \quad m[3,3] = 0;$$

$$A_4 \times A_4 \quad m[4,4] = 0;$$

| m | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 |   |   |   |
| 2 |   | 0 |   |   |
| 3 |   |   | 0 |   |
| 4 |   |   |   | 0 |

| s | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

$$5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$$

**m** represents cost of multiplication

$$A_1 \times A_2$$

$$m[1,2] = 5 \times 6 \times 4 = 120;$$

$$A_2 \times A_3$$

$$m[2,3] = 4 \times 2 \times 6 = 48;$$

$$A_3 \times A_4$$

$$m[3,4] = 6 \times 7 \times 2 = 84;$$

| m | 1 | 2   | 3  | 4  |
|---|---|-----|----|----|
| 1 | 0 | 120 |    |    |
| 2 |   | 0   | 48 |    |
| 3 |   |     | 0  | 84 |
| 4 |   |     |    | 0  |

| s | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   | 1 |   |   |
| 2 |   |   | 2 |   |
| 3 |   |   |   | 3 |
| 4 |   |   |   |   |

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

**m** represents cost of multiplication

$$A_1 \times A_2 \times A_3$$

$$A_1 (A_2 \cdot A_3)$$

$$\begin{aligned} & m[1,1] + m[2,3] + d_0 d_1 d_3 \\ & = 0 + 48 + 5 \times 4 \times 2 = 88; \end{aligned}$$

$$(A_1 \cdot A_2) \cdot A_3$$

$$\begin{aligned} & m[1,2] + m[3,3] + d_0 d_2 d_3 \\ & = 120 + 0 + 5 \times 6 \times 2 = 180; \end{aligned}$$

| m | 1 | 2   | 3  | 4  |
|---|---|-----|----|----|
| 1 | 0 | 120 | 88 |    |
| 2 |   | 0   | 48 |    |
| 3 |   |     | 0  | 84 |
| 4 |   |     |    | 0  |

| s | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   | 1 | 1 |   |
| 2 |   |   | 2 |   |
| 3 |   |   |   | 3 |
| 4 |   |   |   |   |

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

**m** represents cost of multiplication

$$A_2 \times A_3 \times A_4$$

$$A_2 (A_3 \cdot A_4)$$

$$m[2,2] + m[3,4] + d_1 d_2 d_4$$

$$= 0 + 84 + 4 \times 6 \times 7 = 252;$$

$$(A_2 \cdot A_3) \cdot A_4$$

$$m[2,3] + m[4,4] + d_1 d_3 d_4$$

$$= 48 + 0 + 4 \times 2 \times 7 = 104;$$

| m | 1 | 2   | 3  | 4   |
|---|---|-----|----|-----|
| 1 | 0 | 120 | 88 |     |
| 2 |   | 0   | 48 | 104 |
| 3 |   |     | 0  | 84  |
| 4 |   |     |    | 0   |

| s | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   | 1 | 1 |   |
| 2 |   |   | 2 | 3 |
| 3 |   |   |   | 3 |
| 4 |   |   |   |   |

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

$$5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$$

**m** represents cost of multiplication

$$A_1 \times A_2 \times A_3 \times A_4$$

$$A_1 (A_2 A_3 \cdot A_4)$$

$$m[1,1] + m[2,4] + d_0 d_1 d_4$$

$$= 0 + 104 + 5 \times 4 \times 7 = 244;$$

$$(A_1 \cdot A_2) (A_3 \cdot A_4)$$

$$m[1,2] + m[3,4] + d_0 d_2 d_4$$

$$= 120 + 84 + 5 \times 6 \times 7 = 414;$$

$$(A_1 \cdot A_2 \cdot A_3) \cdot A_4$$

$$m[1,3] + m[4,4] + d_0 d_3 d_4$$

$$= 88 + 0 + 5 \times 2 \times 7 = 158;$$

| m | 1 | 2   | 3  | 4   |
|---|---|-----|----|-----|
| 1 | 0 | 120 | 88 | 158 |
| 2 |   | 0   | 48 | 104 |
| 3 |   |     | 0  | 84  |
| 4 |   |     |    | 0   |

| s | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   | 1 | 1 | 3 |
| 2 |   |   | 2 | 3 |
| 3 |   |   |   | 3 |
| 4 |   |   |   |   |

# Now The Formula

---

- Define  $m[i, j]$  to be the minimum number of multiplications needed to compute  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ .
  - Goal: Find  $m[1, n]$
  - Basis:  $m[i, i] = 0$
  - Recursion: How to define  $m[i, j]$  recursively ?
- Consider all possible ways to split  $A_i$  through  $A_j$  into two pieces.
- Compare the costs of all these splits:
  - best case cost for computing the product of the two pieces
  - plus the cost of multiplying the two products
- Take the best one

# Now The Formula

---

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

# Step 2: Develop a Recursive Solution

Matrix\_Chain\_RecursiveWay (m, i, j)

1. for (i <- 0 to n-1)
2.     for (j <- i to n-1)  
           m[ i, j ] <-  $\infty$
3.     if(i=j)  
4.         then return 0;
5.     if(m[ i, j ]  $\neq \infty$ )  
6.         then return m[i, j];
7.     for (k<- i to j)  
8.         do cost= Matrix\_Chain\_RecursiveWay (m, i, k) +  
                  Matrix\_Chain\_RecursiveWay (m, k+1, j) + d<sub>i-1</sub>d<sub>k</sub>d<sub>j</sub>;
9.         if(cost < m[ i, j])  
10.             then m[ i, j ] <- cost;
11.     return m [ i, j];



Memorization

running time  
 $O(n^3)$

## Step 2: Develop a Recursive Solution

---

- Let  $T(n)$  be the time taken by Recursive-Matrix-Chain for  $n$  matrices.  $T(1) \geq 1$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1$$

For  $i = 1, 2, \dots, n-1$ , each term  $T(i)$  appears once as  $T(k)$  and once as  $T(n-k)$ . Thus, we have

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} T(i) + n \\ &\geq 2^{n-1} \end{aligned}$$

Then  $T(n) = \Omega(2^n)$

# Step 3: Compute the Optimal Costs

Find Dependencies among Subproblems

---

| $m:$ | 1   | 2   | 3   | 4   | 5 |
|------|-----|-----|-----|-----|---|
| 1    | 0   |     |     |     |   |
| 2    | n/a | 0   |     |     |   |
| 3    | n/a | n/a | 0   |     |   |
| 4    | n/a | n/a | n/a | 0   |   |
| 5    | n/a | n/a | n/a | n/a | 0 |

← GOAL

computing the red square requires the blue ones: to the left and below.

# Step 3: Compute the Optimal Costs

## Find Dependencies among Subproblems

| $m:$ | 1   | 2   | 3   | 4   | 5 |
|------|-----|-----|-----|-----|---|
| 1    | 0   |     |     |     |   |
| 2    | n/a | 0   |     |     |   |
| 3    | n/a | n/a | 0   |     |   |
| 4    | n/a | n/a | n/a | 0   |   |
| 5    | n/a | n/a | n/a | n/a | 0 |

- Computing  $m(i, j)$  uses
  - everything in same row to the left:  
 $m(i, i), m(i, i+1), \dots, m(i, j-1)$
  - and everything in same column below:  
 $m(i+1, j), m(i+2, j), \dots, m(j, j)$

# Step 3: Compute the Optimal Costs

## Identify Order for Solving Subproblems

- Solve the subproblems (i.e., fill in the table entries) this way:
  - go along the diagonal
  - start just above the main diagonal
  - end in the upper right corner (goal)

*m:*

|   |     |     |     |     |   |
|---|-----|-----|-----|-----|---|
|   | 1   | 2   | 3   | 4   | 5 |
| 1 |     |     |     |     |   |
| 2 | n/a |     |     |     |   |
| 3 | n/a | n/a |     |     |   |
| 4 | n/a | n/a | n/a |     |   |
| 5 | n/a | n/a | n/a | n/a | 0 |

GOAL

# Step 4: Construct an Optimal Solution

---

- It's fine to know the cost of the cheapest order, but what is that cheapest order?
- Keep another array  $s$  and update it when computing the minimum cost in the inner loop
- After  $m$  and  $s$  have been filled in, then call a recursive algorithm on  $s$  to print out the actual order

`Print_order(s, i, j)`

```
1. if (i=j)
 then print "A"i;
Else
 print "("
 Print_order (s, i, s[i,j]);
 Print_order (s, s[i,j] +1, j);
 print ")";
```

---

Thank You!

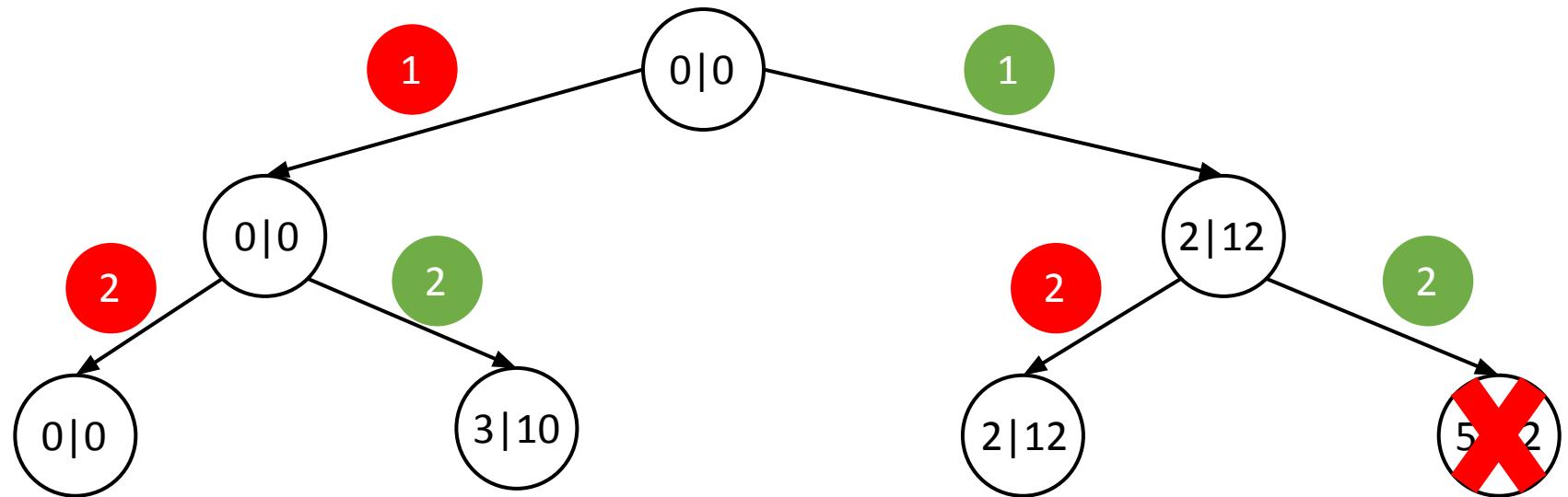
# Backtracking

*-Lec Ramisha Fariha Baki*

# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

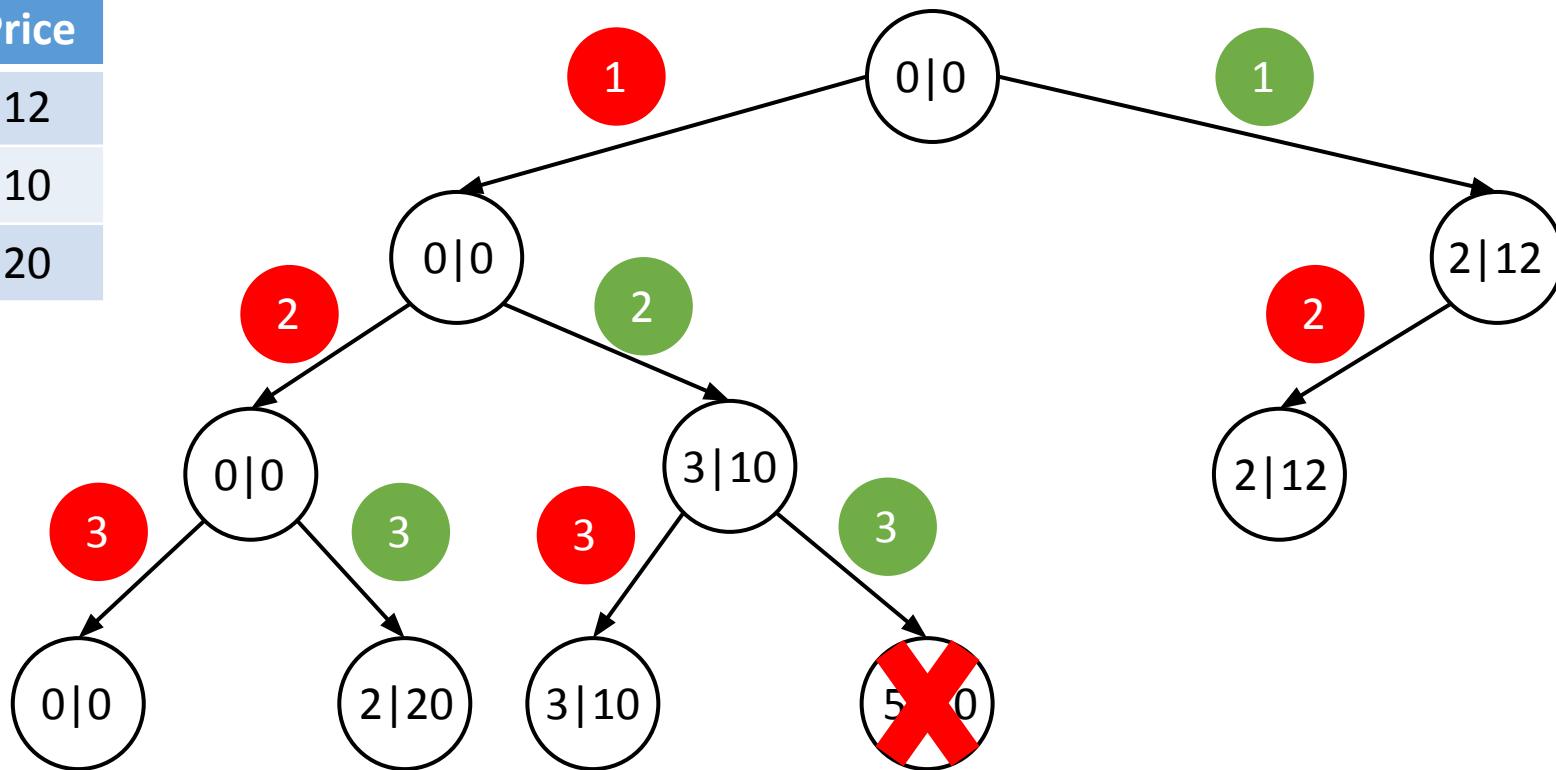
$W = 4$



# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

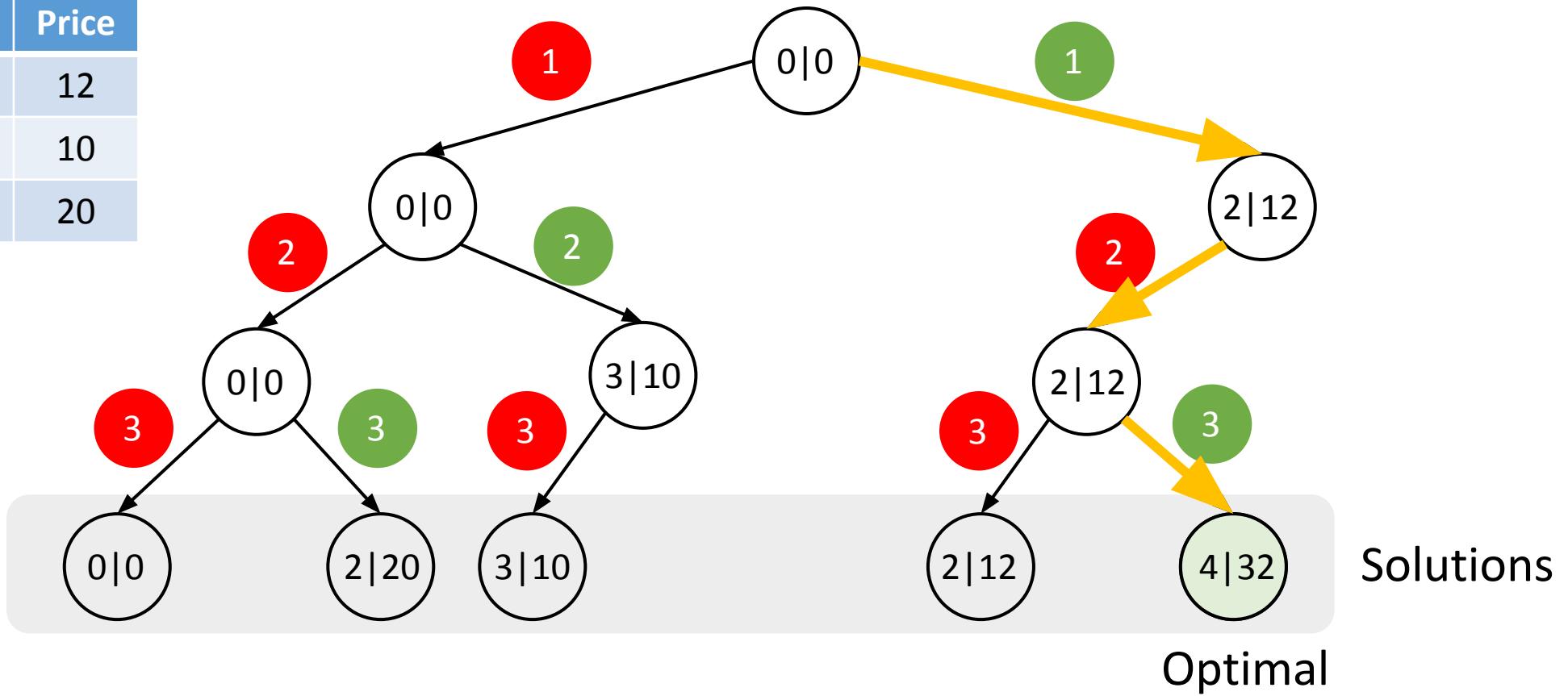
$W = 4$



# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

$W = 4$



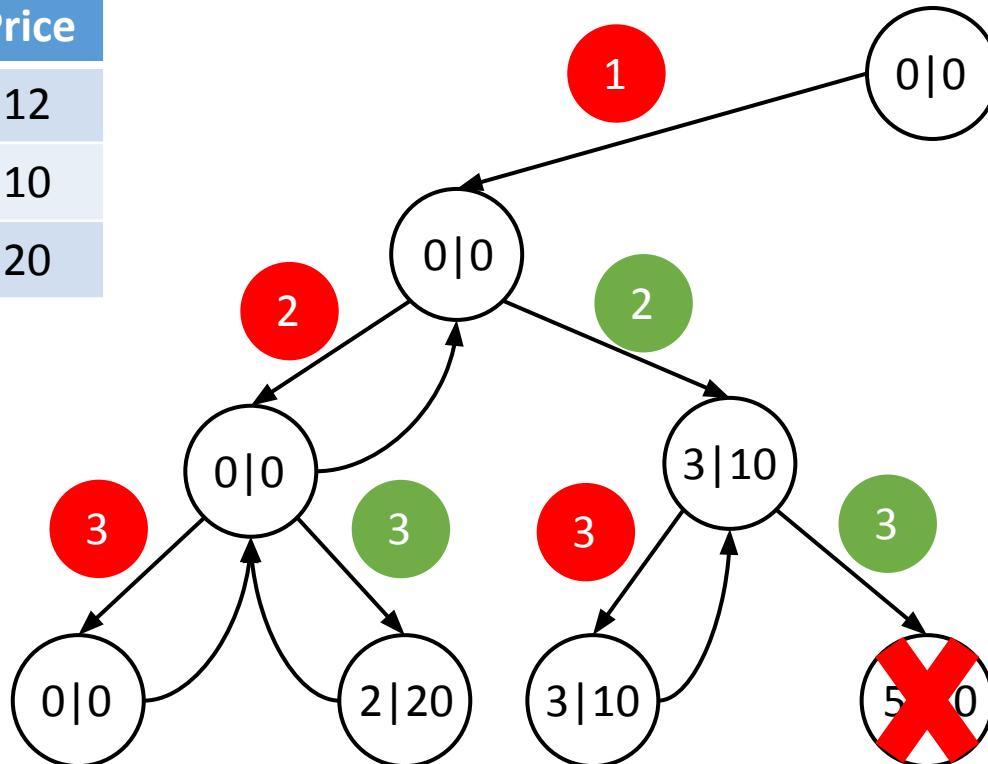
Solutions

Optimal

# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

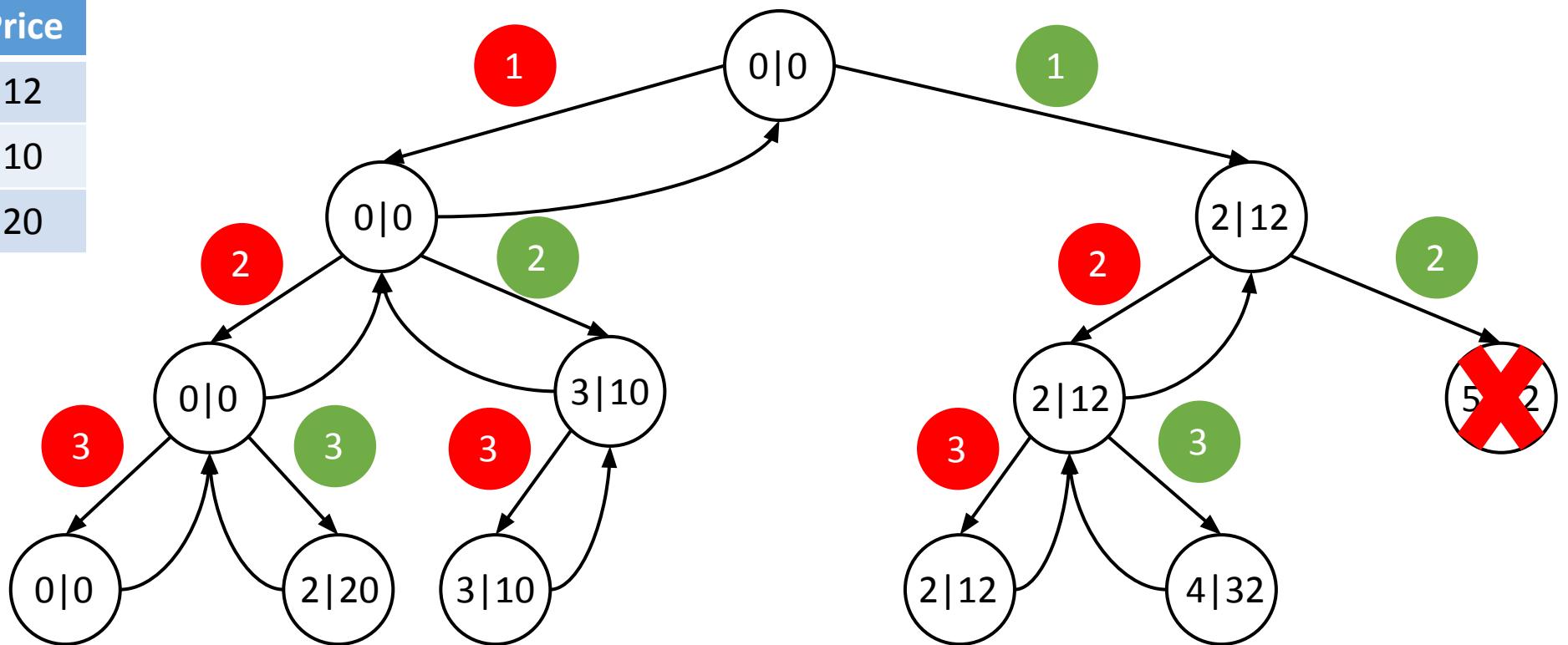
$W = 4$



# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

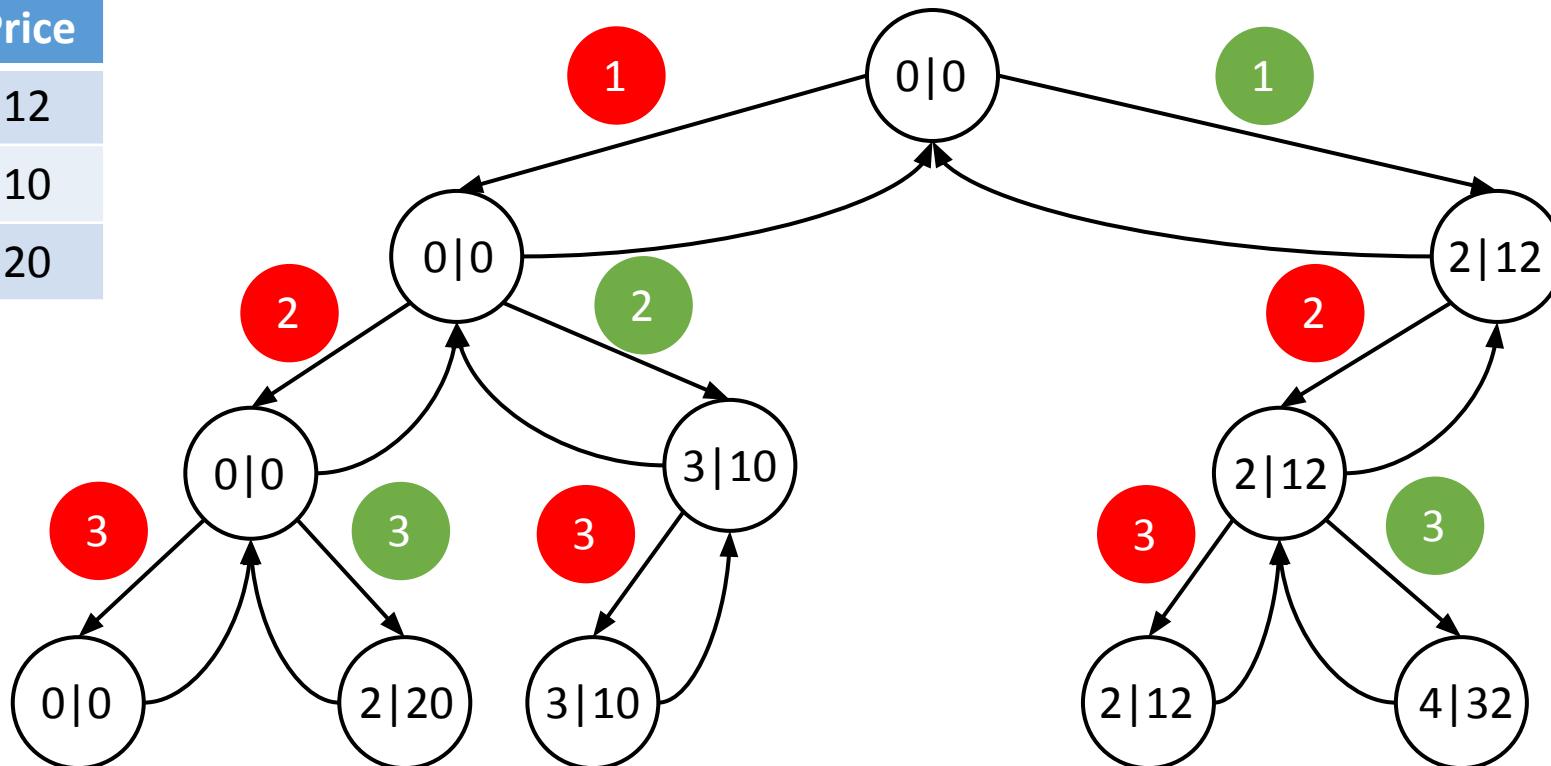
$W = 4$



# 0-1 Knapsack By Backtracking

| Item | Weight | Price |
|------|--------|-------|
| 1    | 2      | 12    |
| 2    | 3      | 10    |
| 3    | 2      | 20    |

$W = 4$



Item 1 having 2 probabilities: Can't be taken or can be taken

For each path of Item-1, Item 2 has 2 probabilities: Can't be taken or can be taken. So, For item 2 there will be  $2 \times 2$  nodes

Similarly for Item-3, there will be  $2 \times 2 \times 2$  nodes

So, for  $N$  items, there will be  $2^N$  nodes

Exponential

# Backtracking

- Generating all possible solutions
- Prune some nodes based on some constraints

# Backtracking Complexity

- ❑ If there are total **N** states
- ❑ If state having **P** number of probabilities
- ❑ Then total number of nodes generated:  $P^N + P^{N-1} + P^{N-2} + \dots + P^1$
- ❑  $O(P^N + P^{N-1} + P^{N-2} + \dots + P^1)$
- ❑  $O(P^N)$

# **Sum of Subsets Problem**

# Sum of Subsets

## Problem

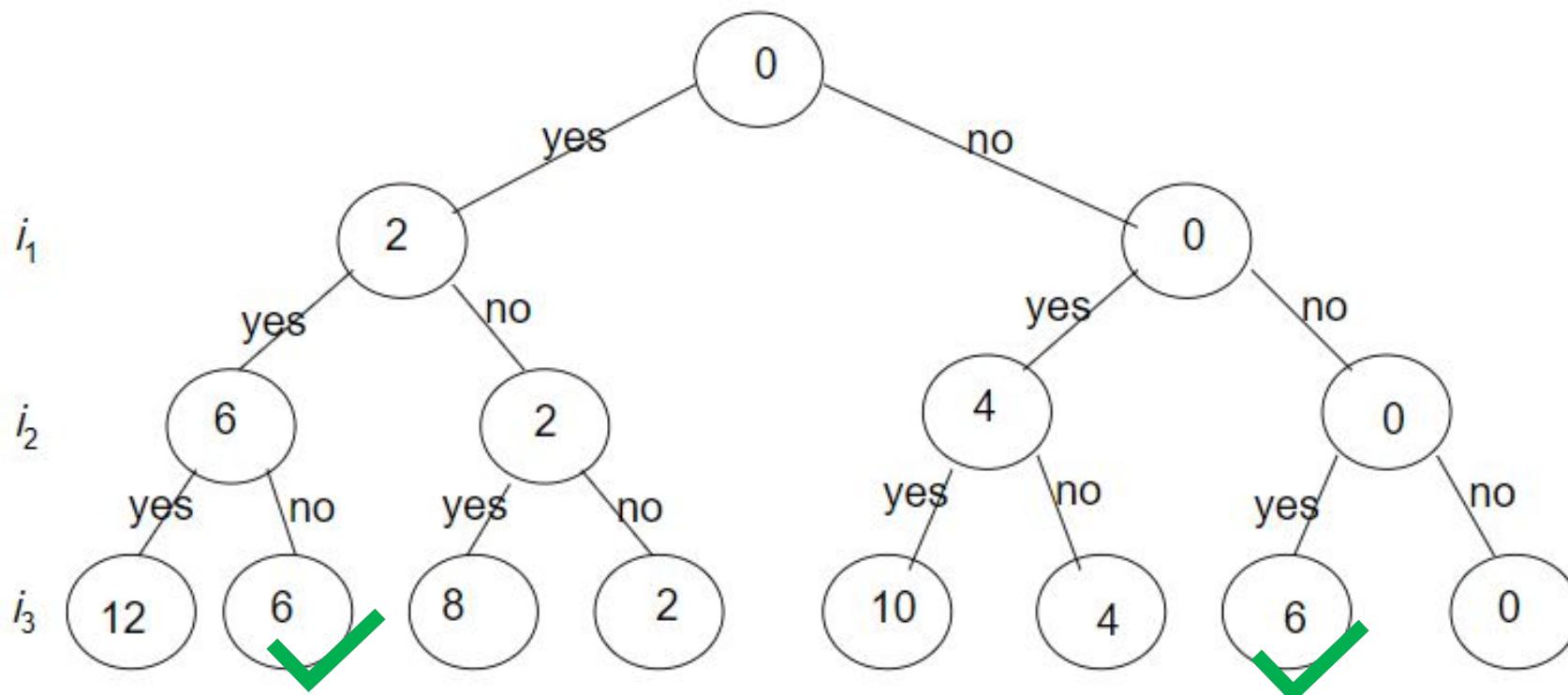
- Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number.
- A subset A of **n positive integers and a value sum** is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum

# Sum of Subsets Problem

- Given the following set of positive numbers: { 2, 4, 6}
- We need to find if there is a subset for a given sum say 6:
- { 2, 4 } { 6}

## Sum of subset Problem: State SpaceTree for 3 items

$$w_1 = 2, \quad w_2 = 4, \quad w_3 = 6 \text{ and } S = 6$$



The sum of the included integers is stored at the node.

## A DFS solution

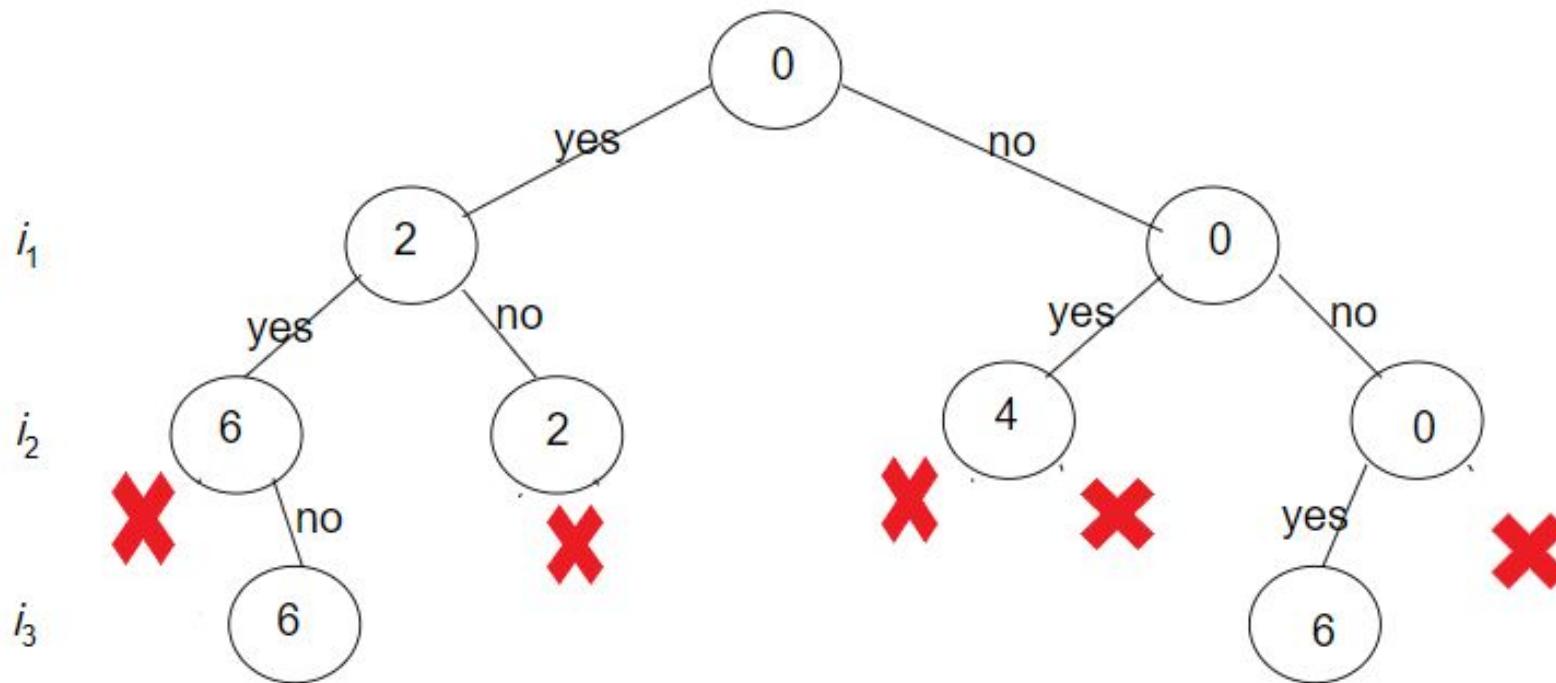
- Such a DFS algorithm will be very slow.
- It does not check for every **solution state (node)** whether a solution has been reached, or whether a *partial* solution can lead to a *feasible* solution
- Is there a more efficient solution?

# Backtracking

- **Definition:** We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising*
- **Main idea:** Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent

**Sum of subset Problem:  
State SpaceTree for 3 items**

$$w_1 = 2, \quad w_2 = 4, \quad w_3 = 6 \text{ and } S = 6$$



The sum of the included integers is stored at the node.

# Sum of Subsets Problem

- Given the following set of positive numbers: { 1, 2, 1}
- We need to find if there is a subset for a given sum say 2:
- { 1, 2 } { 2, 1}

# Sum of Subsets Problem

{1, 2, 1}  
↑

Index, targetSum, subset

0,        3,

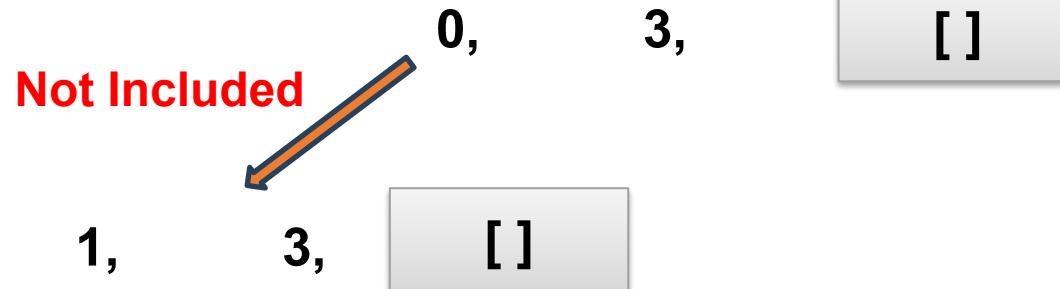
[ ]

# Sum of Subsets Problem

{1, 2, 1}



Index, targetSum, subset

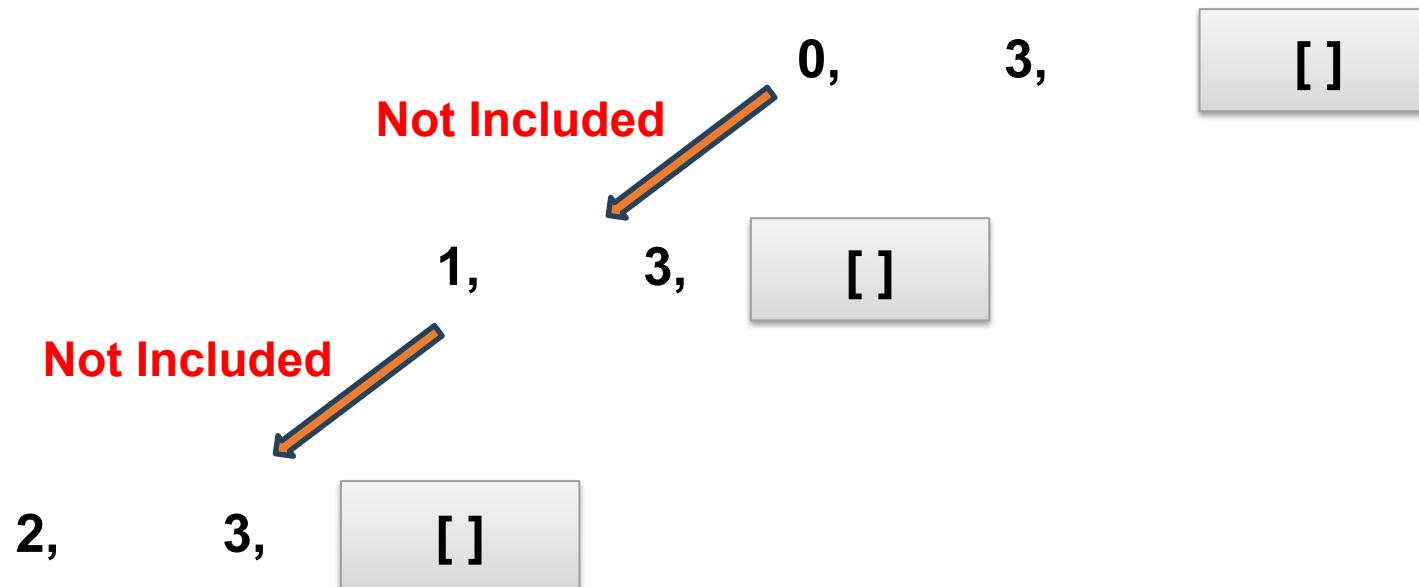


# Sum of Subsets Problem

{1, 2, 1}



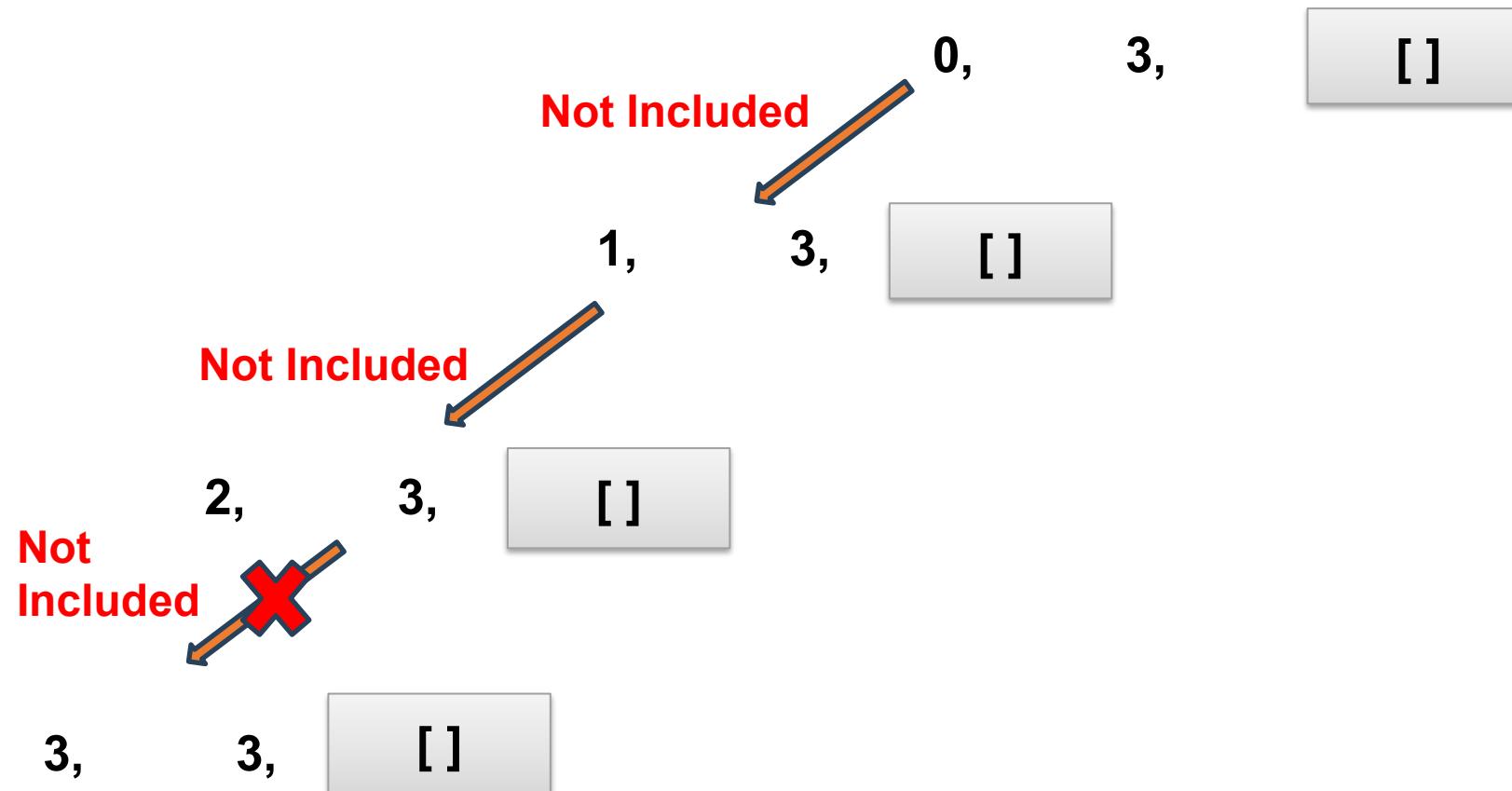
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1}  
↑

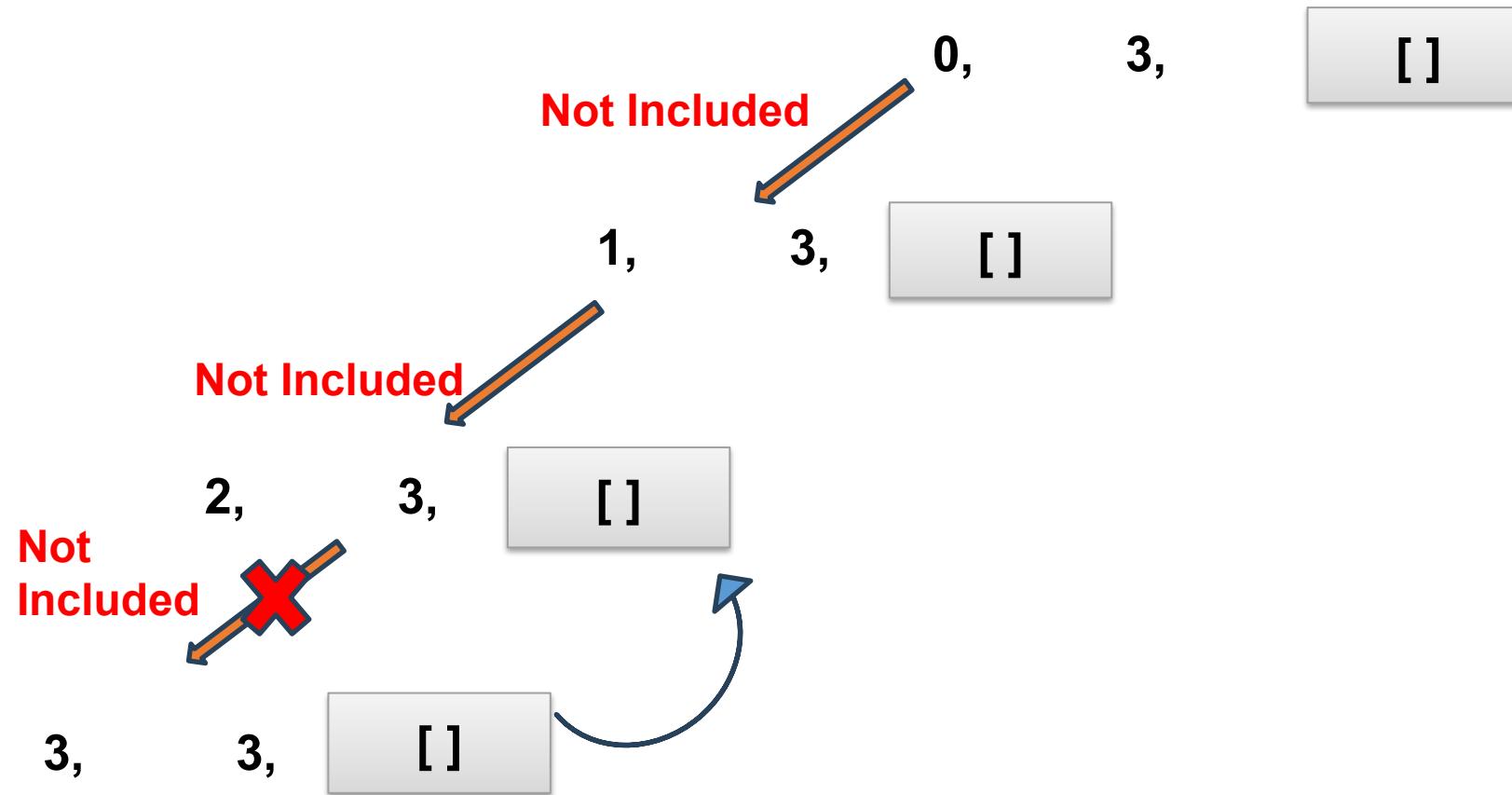
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1} ↑

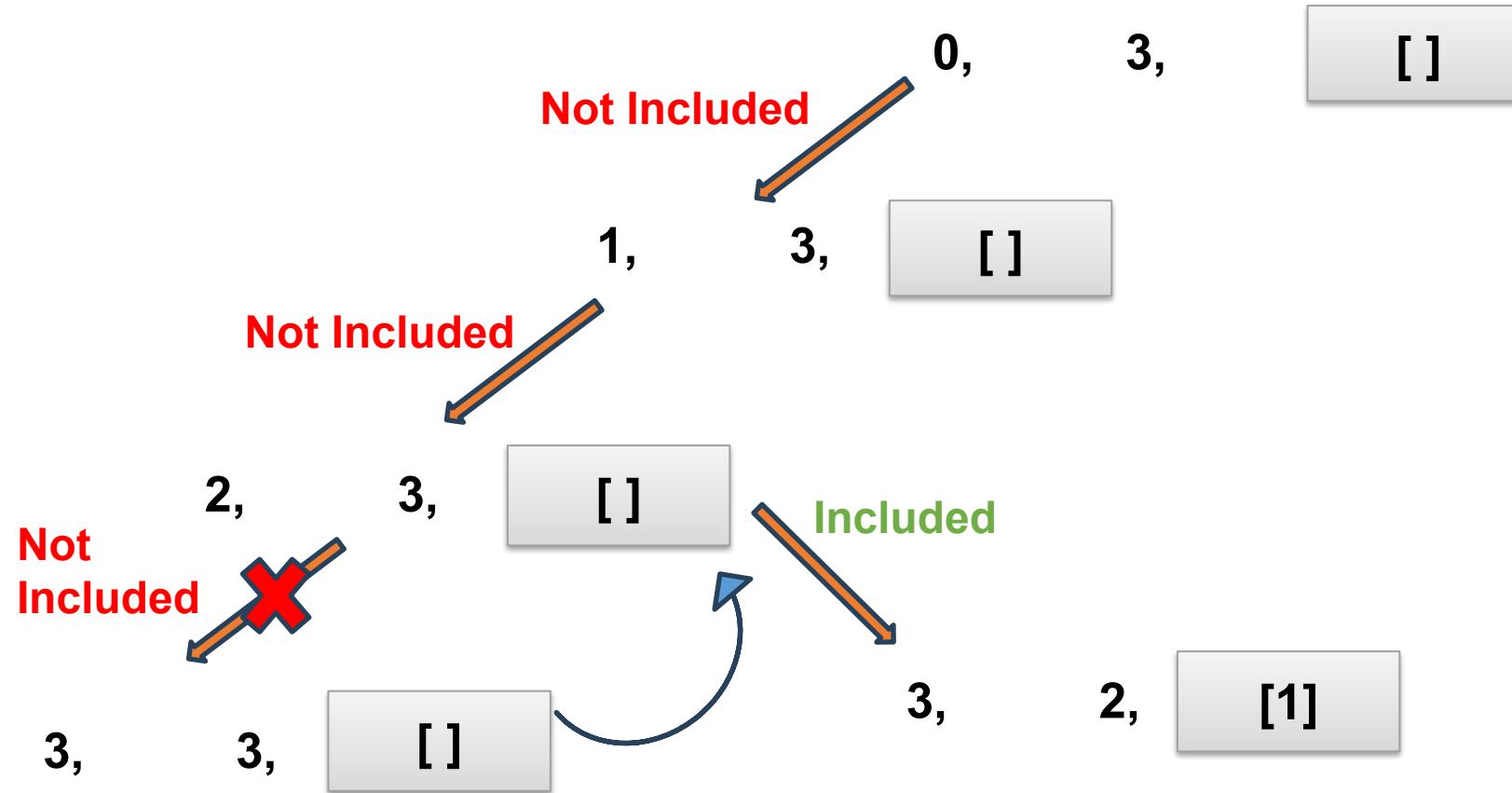
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1} ↑

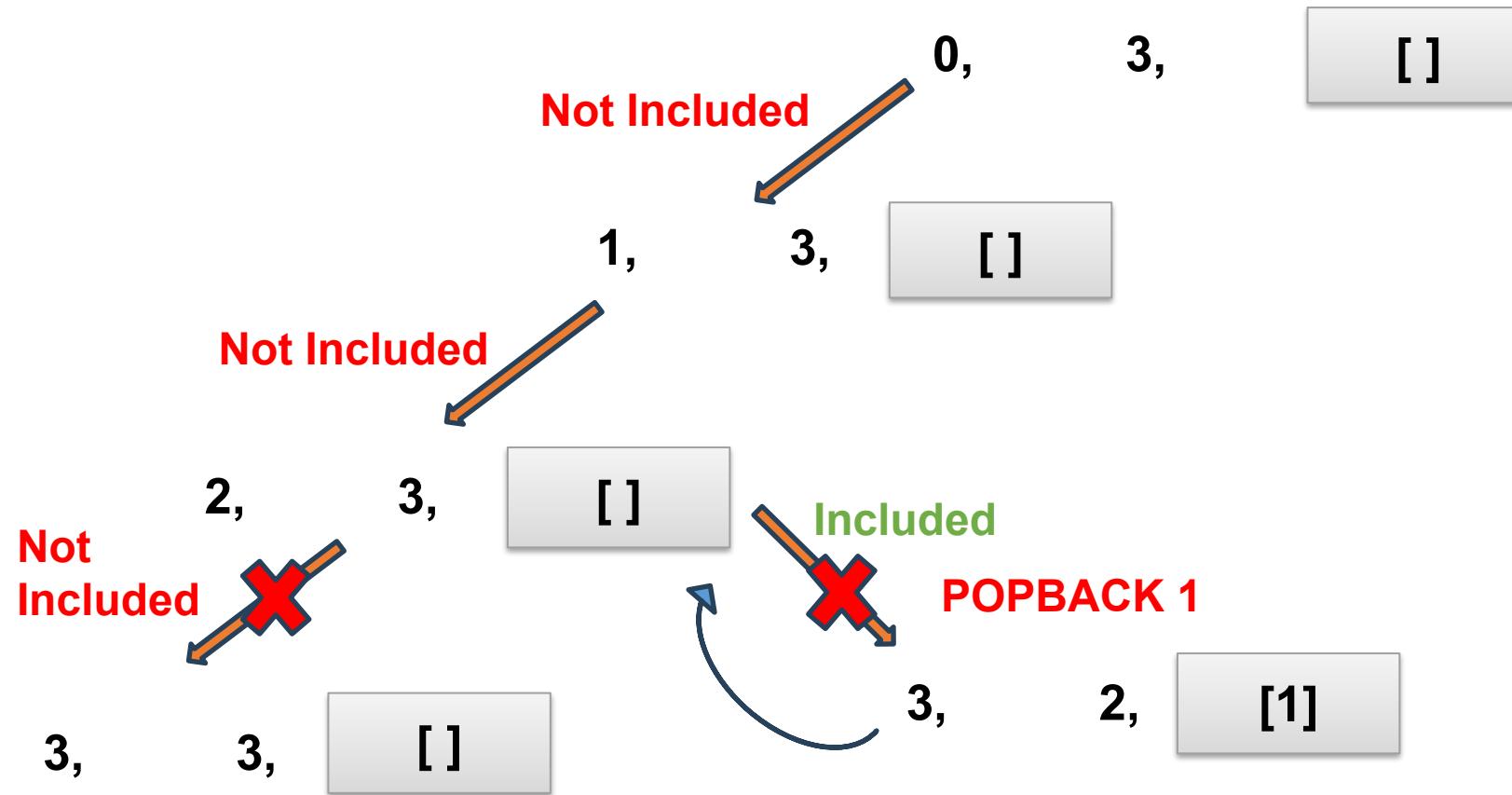
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1} ↑

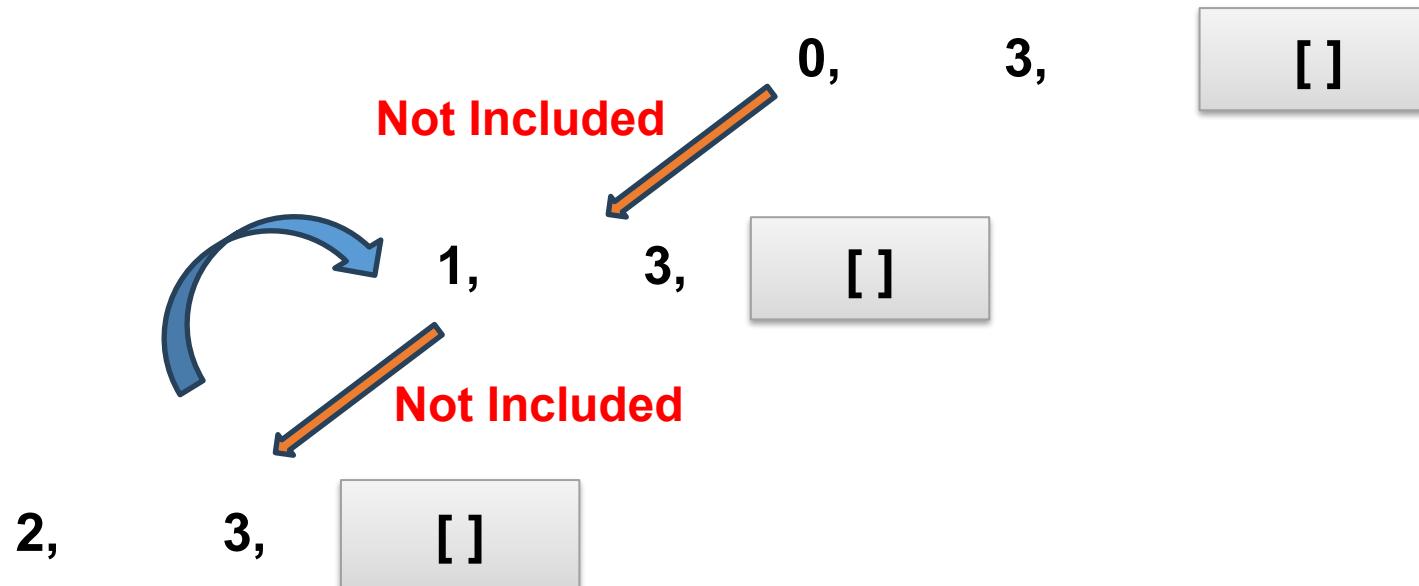
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1} ↑

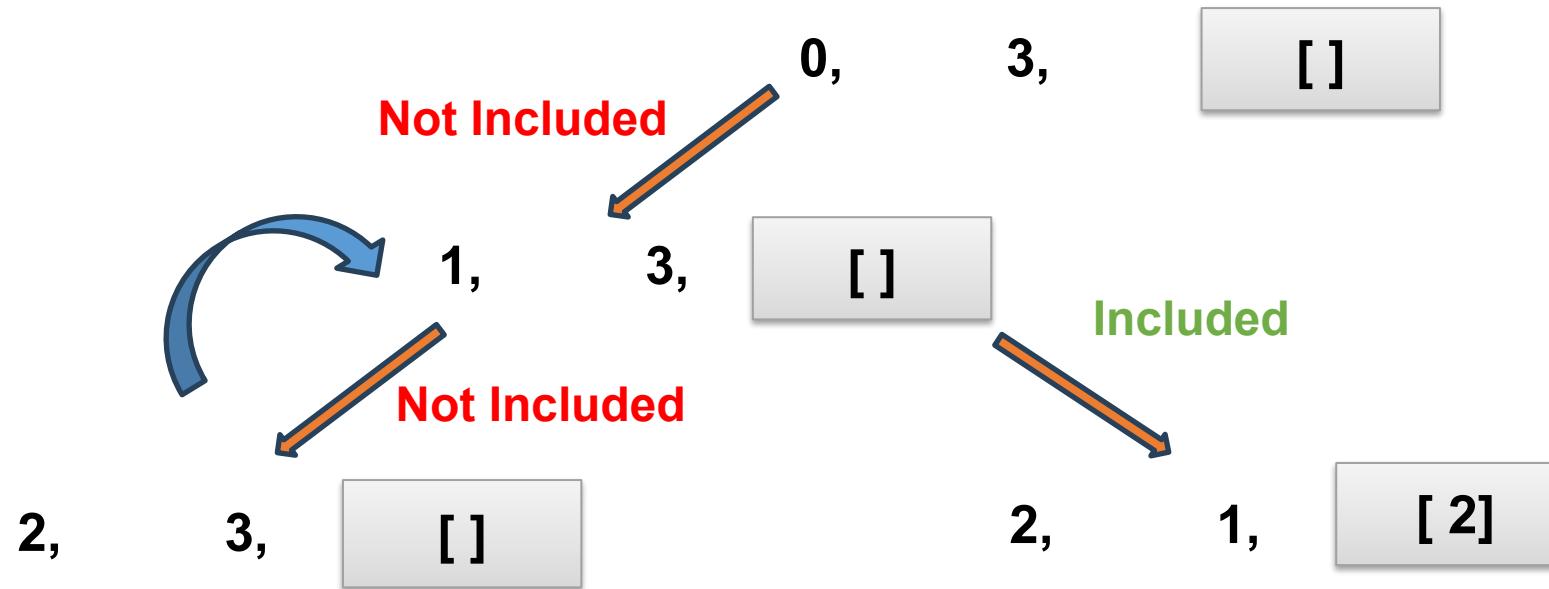
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1}

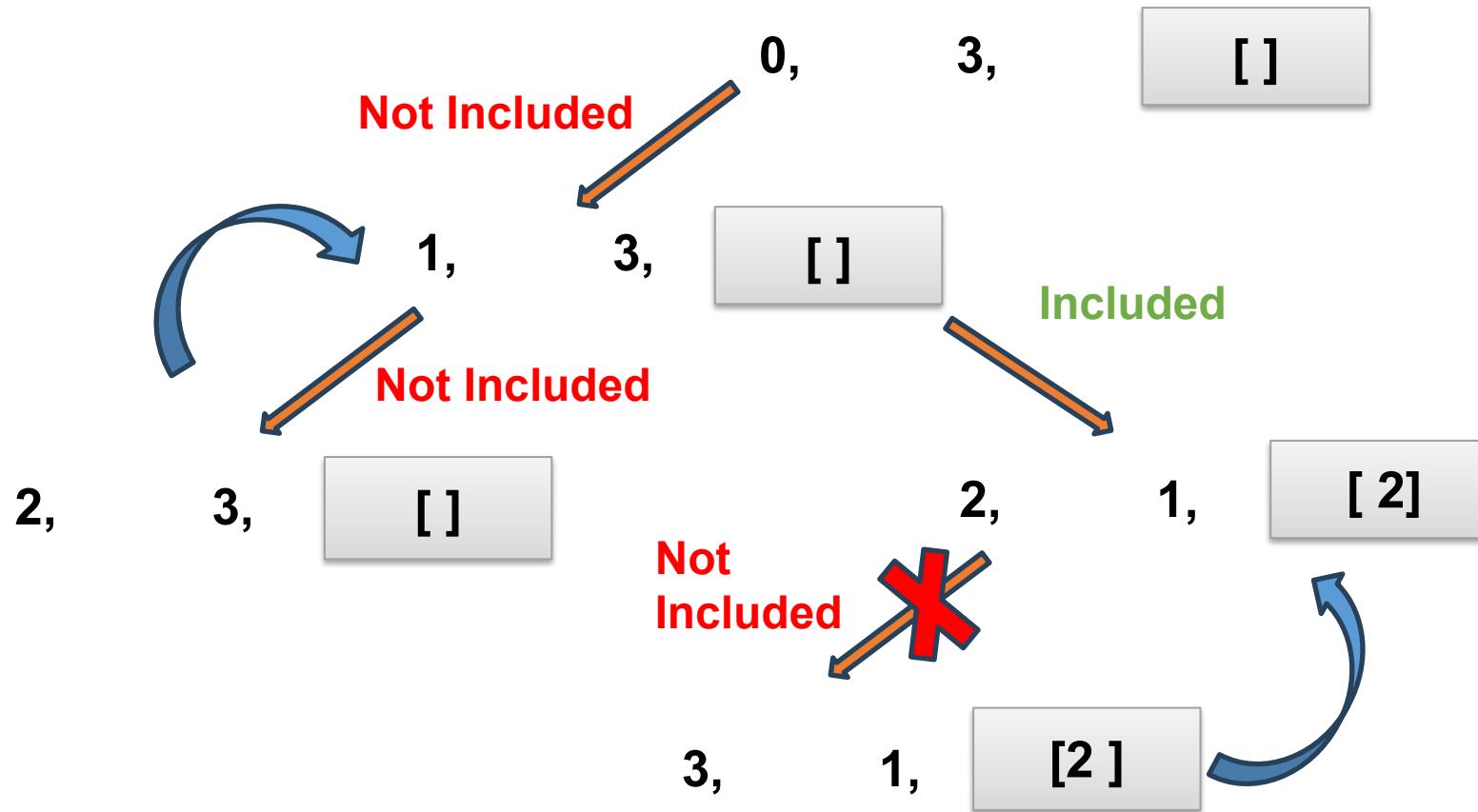
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1}

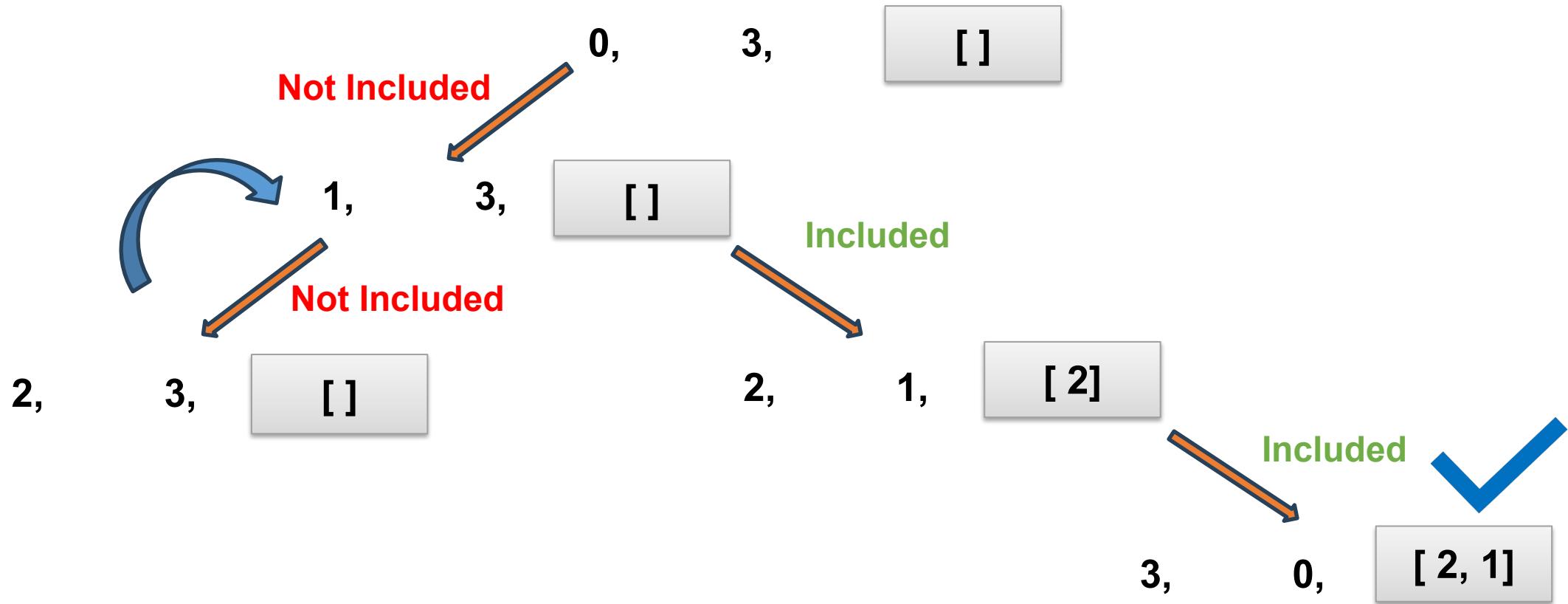
Index, targetSum, subset



# Sum of Subsets Problem

{1, 2, 1} ↑

Index, targetSum, subset



# Observe what we just did:

The entire system can be defined as recursion. Which includes:

- The no of array element -> Integer
  - The array elements? Array
  - The target sum? Integer
  - Possible answer (subset)-> Can be represented by a vector
- 
- We approached one index at a time.
  - We include or not include that index.
  - We keep the target sum as it is when we do not include the index
  - We reduce the target sum when we include the index value and pushback the index value

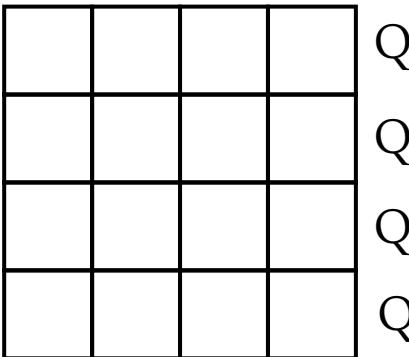
# The elements of backtracking

- Pass index, targetSum, subset
- Check if target Sum == 0 ↗ we have reached to an answer
- Check if it is the end of array
- Call the next index without including the present index
- When including the index value:
  - A. Reduce the value from targetSum
  - B. Pushback in the vector
  - C. Popback after the recursion call is ended

# N Queen Problem

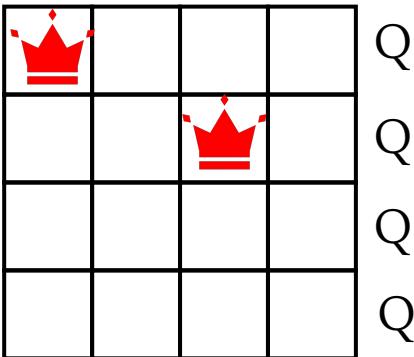
# N Queen Problem

- A chess board of **NxN** size
- **N** number of queens are given
- The queens need to be placed on the board in such a way that no 2 queens are in attacking position
- Let, N=4



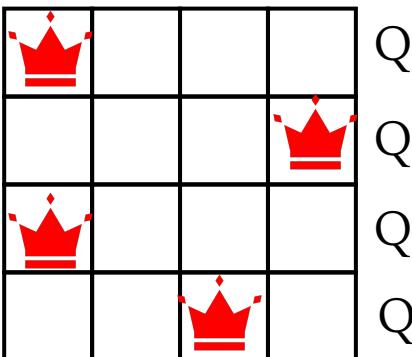
# N Queen Problem

- A chess board of **NxN** size
- **N** number of queens are given
- The queens need to be placed on the board in such a way that no 2 queens are in attacking position
- Let, N=4

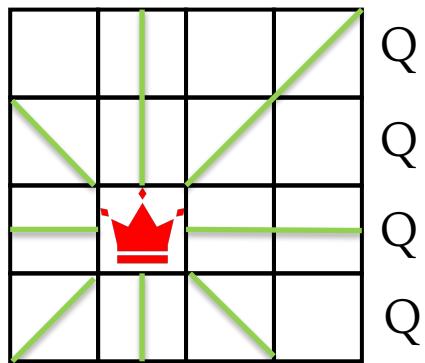


# N Queen Problem

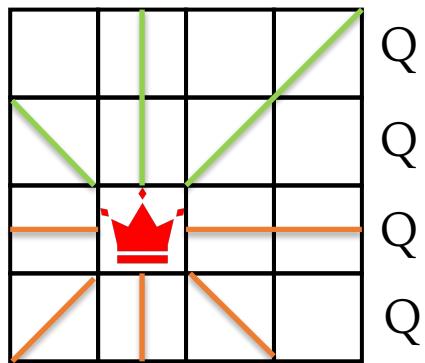
- A chess board of **NxN** size
- **N** number of queens are given
- The queens need to be placed on the board in such a way that no 2 queens are in attacking position
- Let, N=4



# Constraints to be checked



# Constraints to be checked

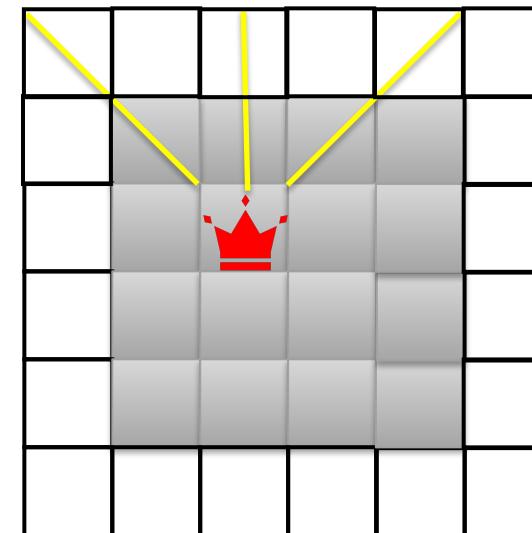
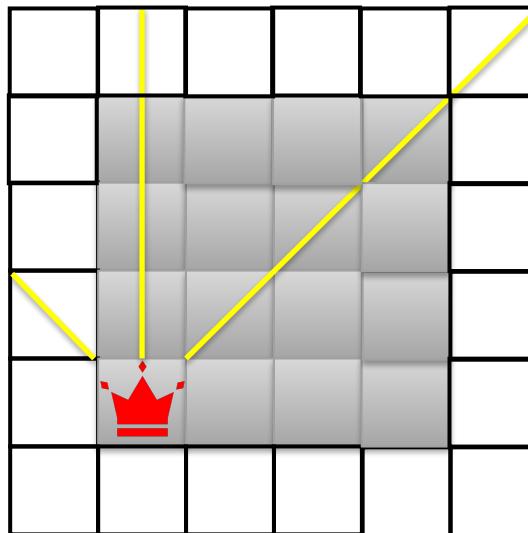


# Constraints to be checked

For Upper Side  Row=0

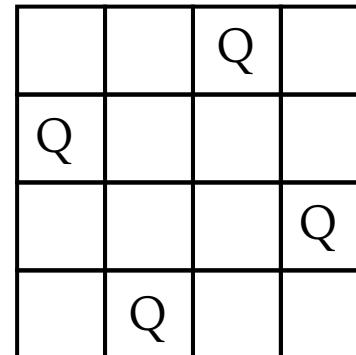
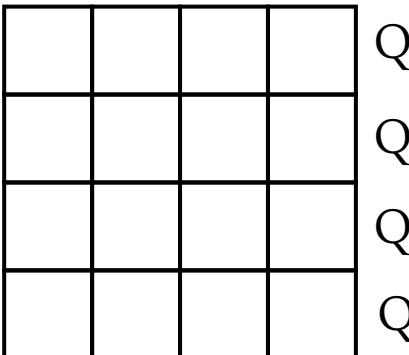
For Right Diagonal  Column=N+1 OR Row=0

For Left Diagonal  Column=0 OR Row=0



# N Queen Problem

- A chess board of **NxN** size
- **N** number of queens are given
- The queens need to be placed on the board in such a way that no 2 queens are in attacking position
- Let, N=4



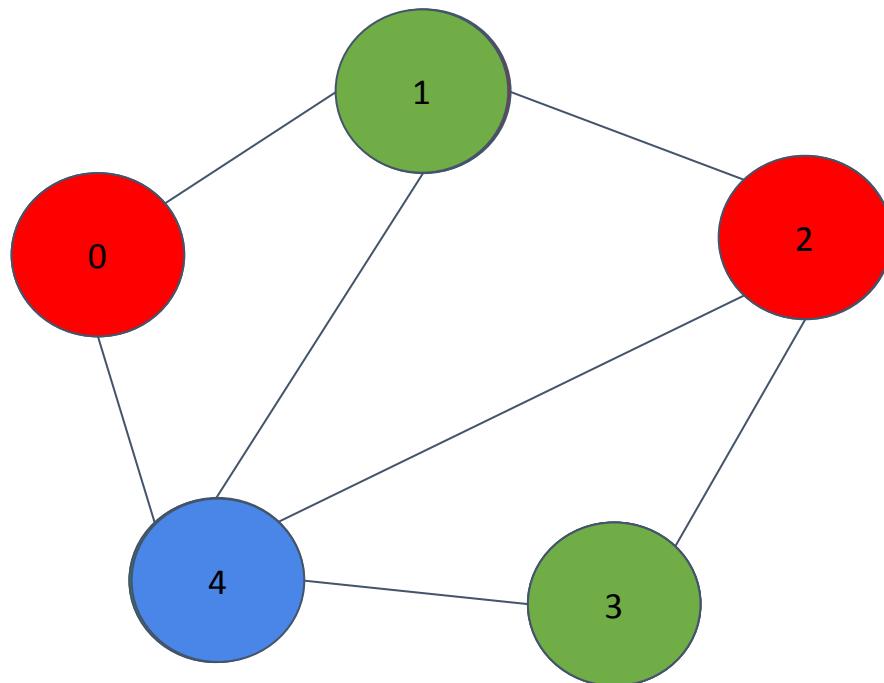
Another  
Solution

# The elements of backtracking

- Start with Row 1
- Check if the column in that row is Safe for the Queen to be Placed
- How to check if it is Safe? ↗ Call a (isSafe(row,col)) function and check the constraints
- Place the Queen if it isSafe and pass the next Row (Row+1).
- What will be the Goal State? ↗ Row==N+1

# The Graph Coloring Problem

- You have 3 colors : R,G,B
- Color all the vertices of the graph in such a way that.
- Each vertex has a color.
- The color of two adjacent vertices are different



# Observe what we just did:

The entire system can be regarded as a **state/node/world**. Which includes:

- The no of Vertices -> Integer
  - The no. of vertices colored so far-> Integer
  - The no of Candidate Colors? Integer
  - The graph-> Can be represented by adjacent matrix
  - Candidate Colors-> Can be represented as an array
  - The colors of the vertices-> Can be represented as an array
- 
- We approached one unassigned vertex at a time.
  - We tried to assign a color to each vertex
  - While assigning colors by ensuring different colors for adjacent matrices

# The elements of backtracking

- Pass one node
- Check if it is Valid with a particular color -> color it
- Check if it goal state? Print the entire vertices with color
- Pass the next node

## **Valid Condition:**

- Edge Connected && colors of the connected nodes are not same

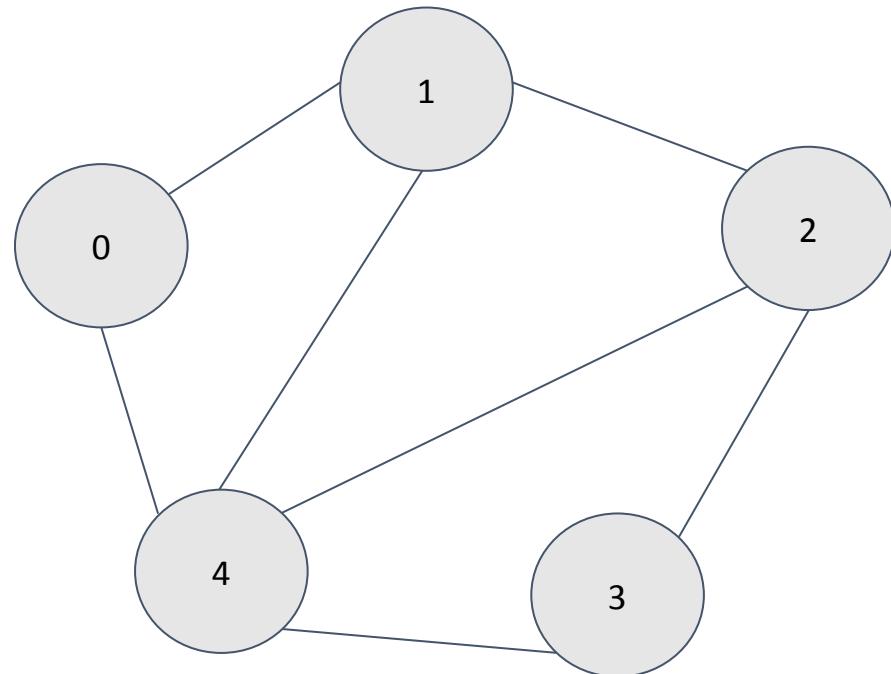
## **Goal State:**

- Number of colored vertices is equal to number of vertices

# The elements of backtracking

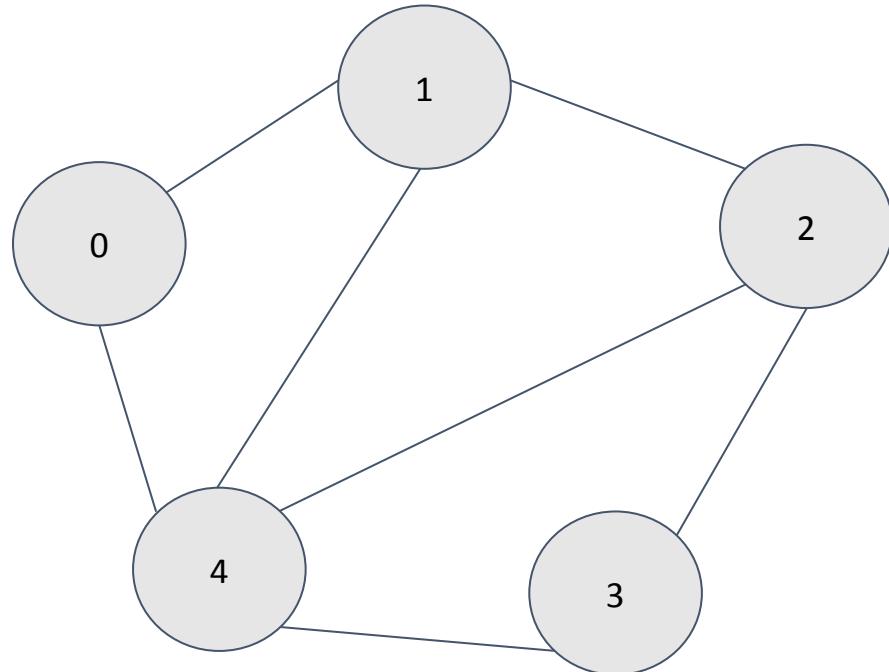
- The no of Vertices -> 5
- The no. of vertices colored so far-> 0
- The no of Candidate Colors ?3
- The graph->

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |
- Candidate Colors-> 0(R) 1(G) 2(B)
- The colors of the vertices-> -1 -1 -1 -1 -1



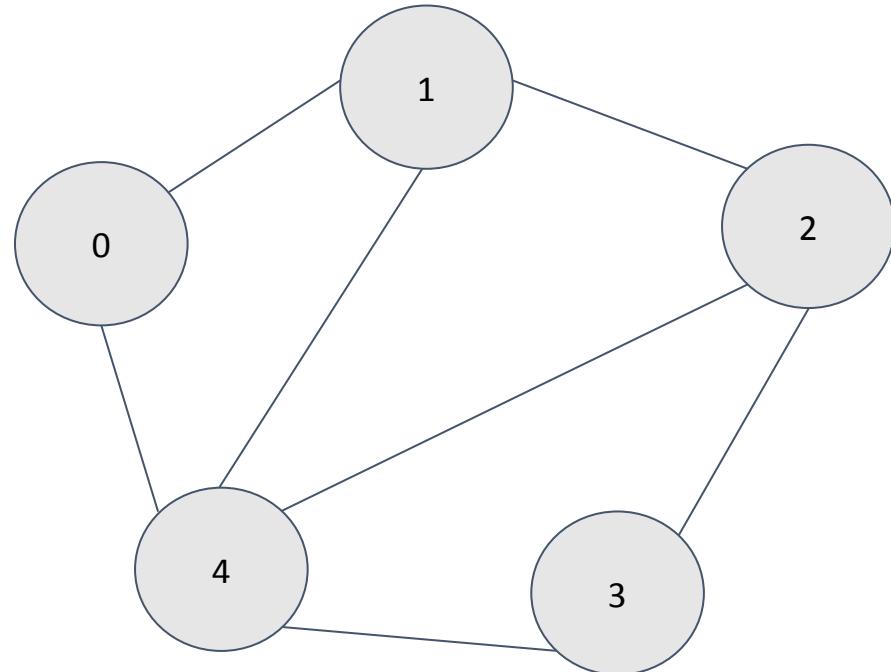
# The elements of backtracking

- Pass one node and one candidate color
- Check if it is Valid with a particular color -> color it
- Check if it goal state  Print the entire vertices with color
- Pass the next node



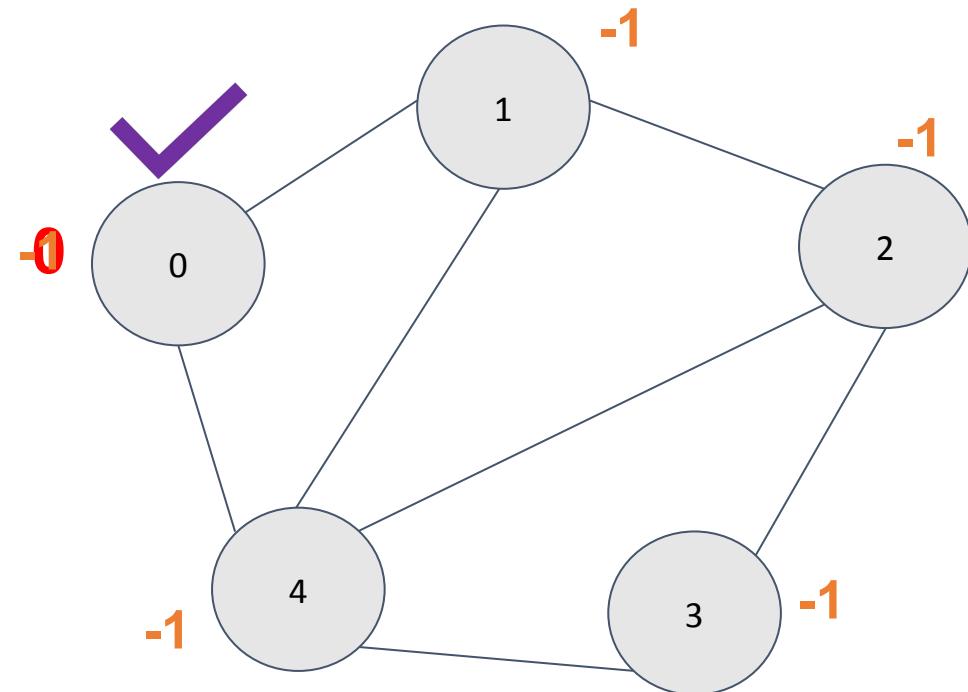
# The elements of backtracking

- **Pass one node and one candidate color**
- Check if it is Valid with a particular color -> color it
- Check if it goal state  Print the entire vertices with color
- Pass the next node



# The elements of backtracking

- **Pass one node and one candidate color**
- Check if it is Valid with a particular color -> color it
- Check if it goal state  Print the entire vertices with color
- Pass the next node

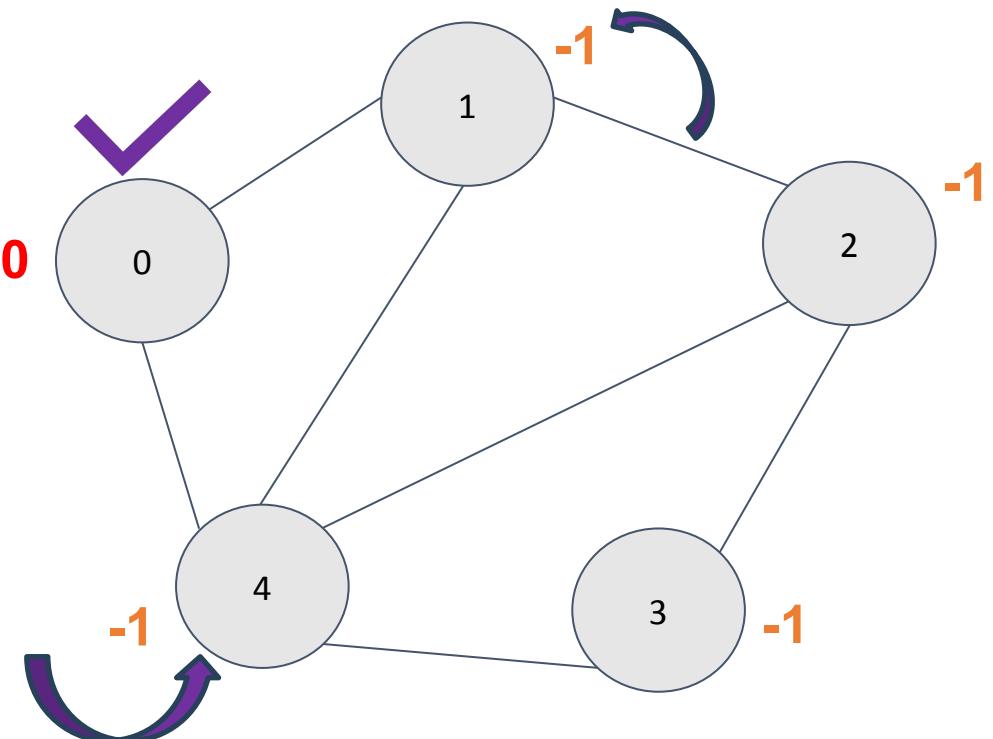


# The elements of backtracking

- Pass one node
- **Check if it is Valid with a particular color -> color it**

(Edge Connected && colors of the connected nodes are **not same**)

- Check if it goal state  Print the entire vertices with color
- Pass the next node

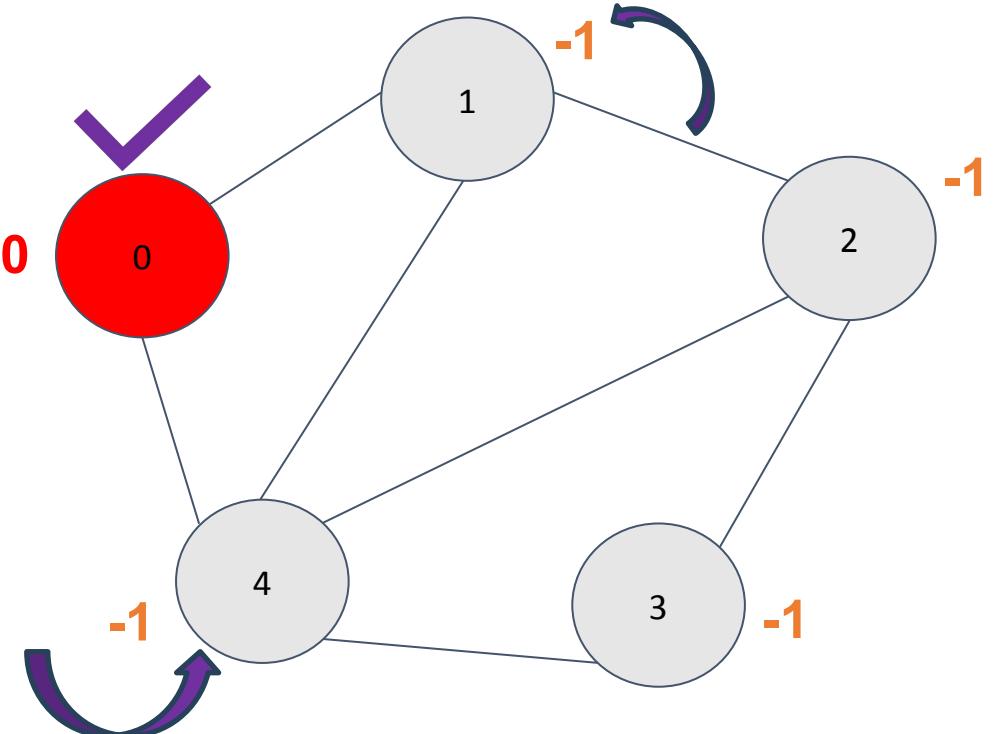


# The elements of backtracking

- Pass one node
- **Check if it is Valid with a particular color -> color it**

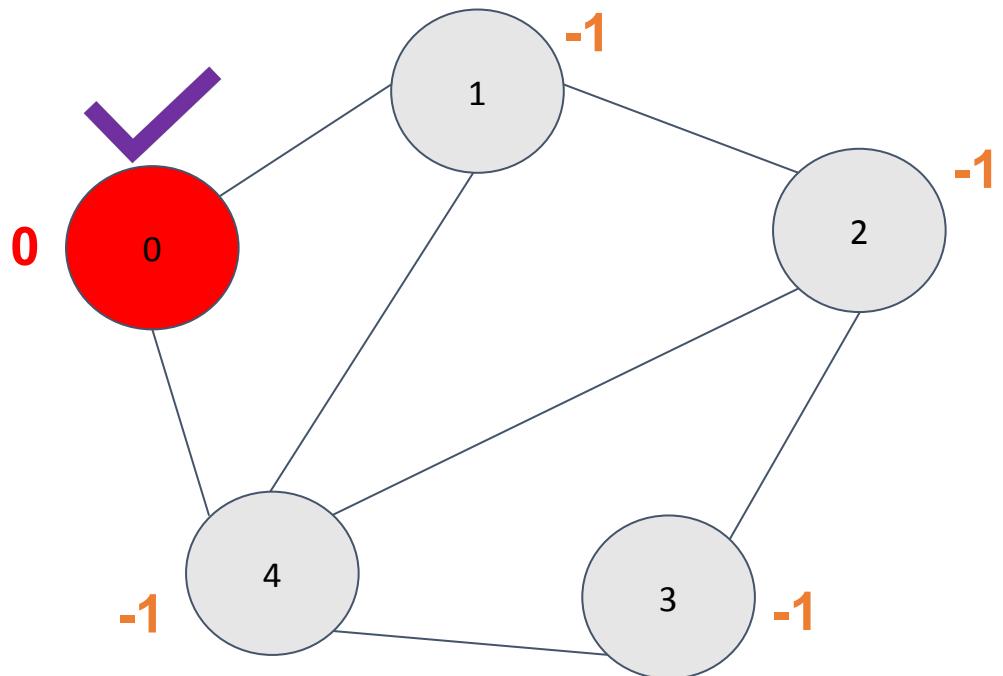
(Edge Connected && colors of the connected nodes are **not same**)

- Check if it goal state  Print the entire vertices with color
- Pass the next node



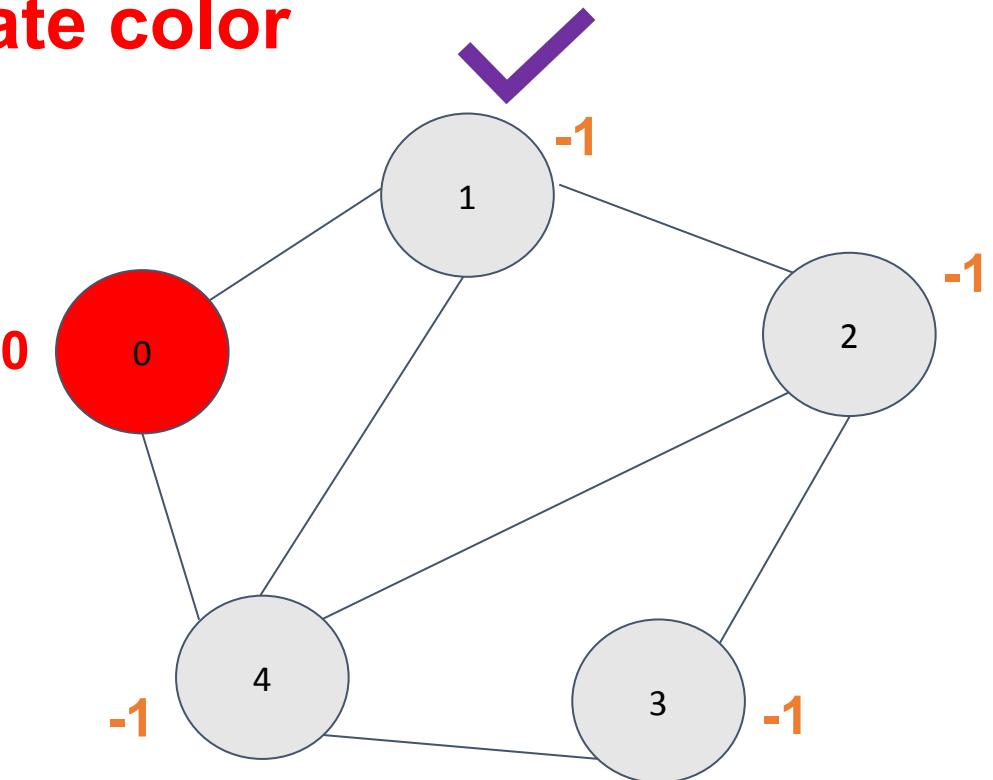
# The elements of backtracking

- Pass one node
- Check if it is Valid with a particular color -> color it
- **Check if it goal state**  Print the entire vertices with color
- Pass the next node



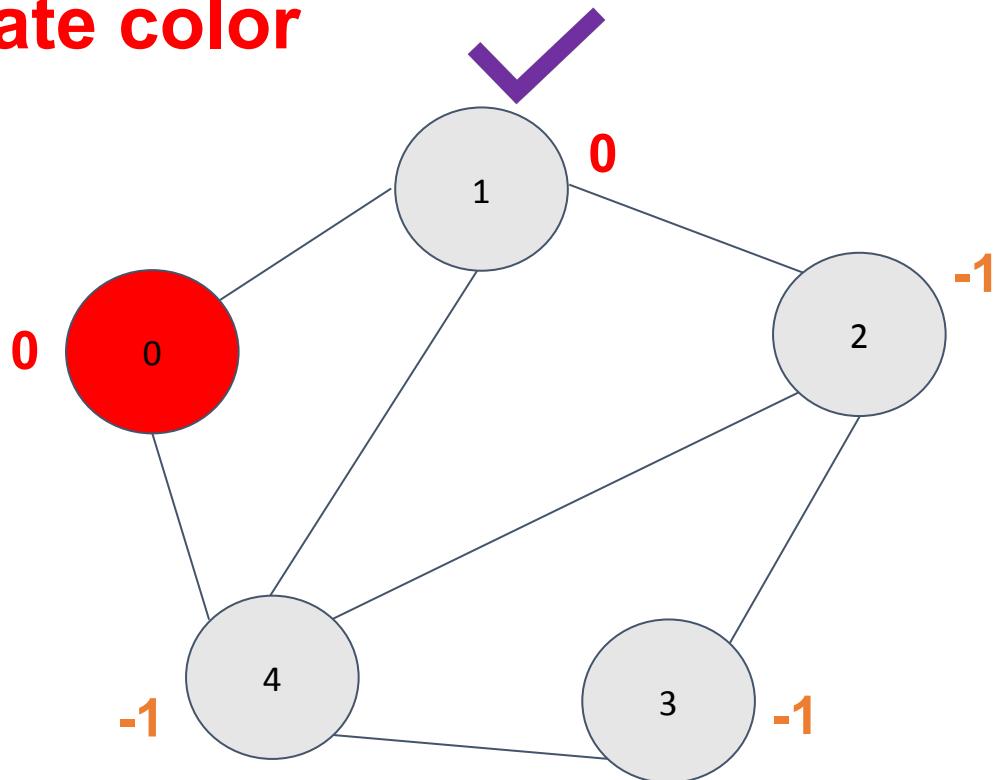
# The elements of backtracking

- Pass one node
- Check if it is Valid with a particular color -> color it
- Check if it goal state  Print the entire vertices with color
- **Pass the next node with a candidate color**



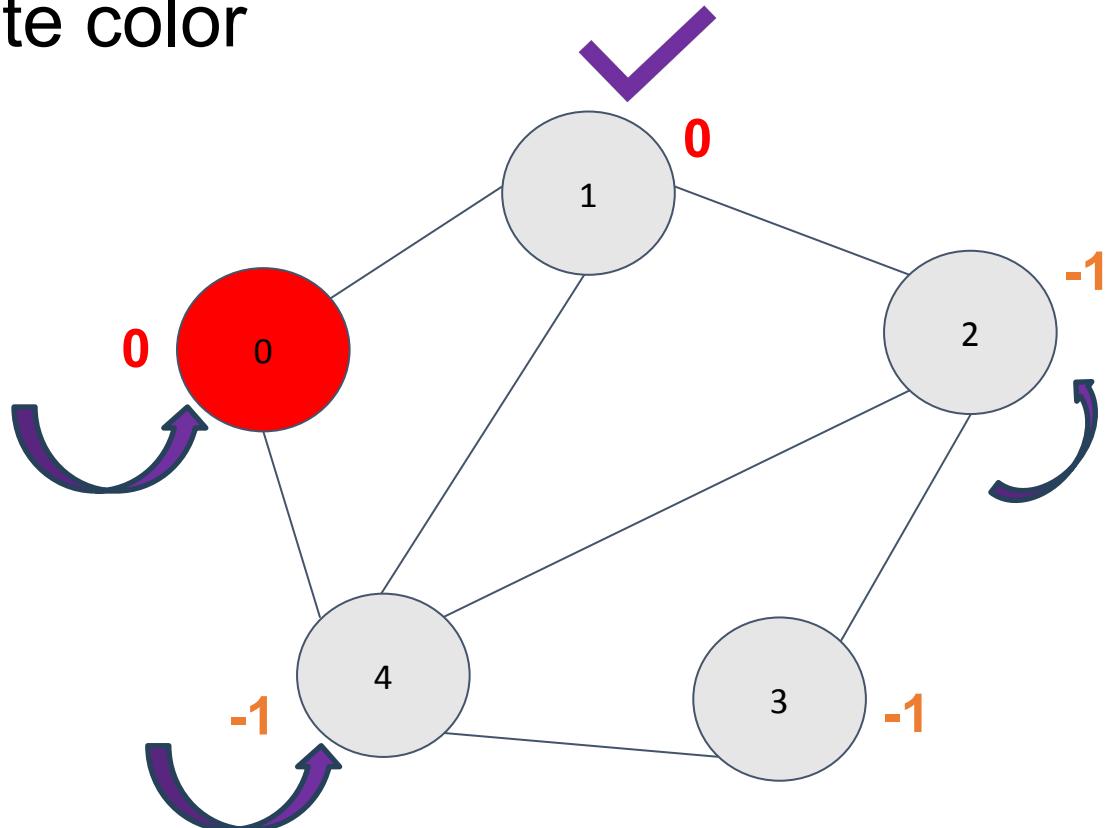
# The elements of backtracking

- Pass one node
- Check if it is Valid with a particular color -> color it
- Check if it goal state  Print the entire vertices with color
- **Pass the next node with a candidate color**



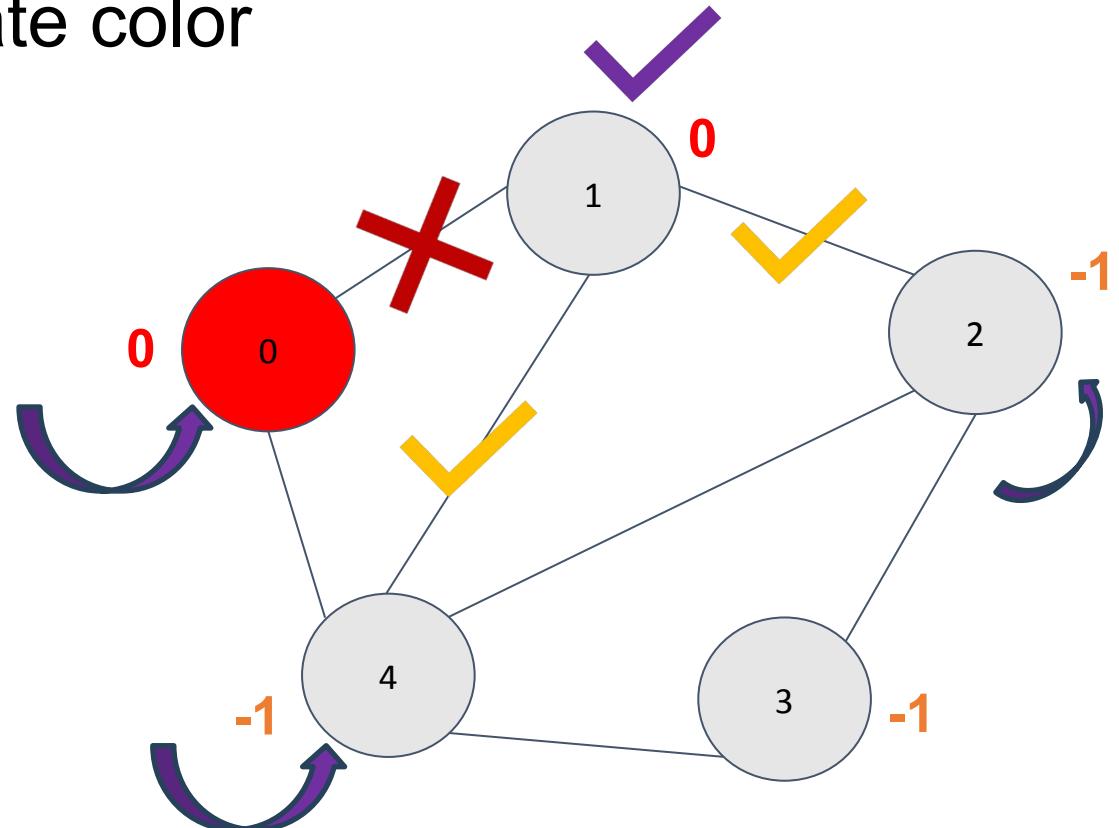
# The elements of backtracking

- Pass one node
- **Check if it is Valid with a particular color -> color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color



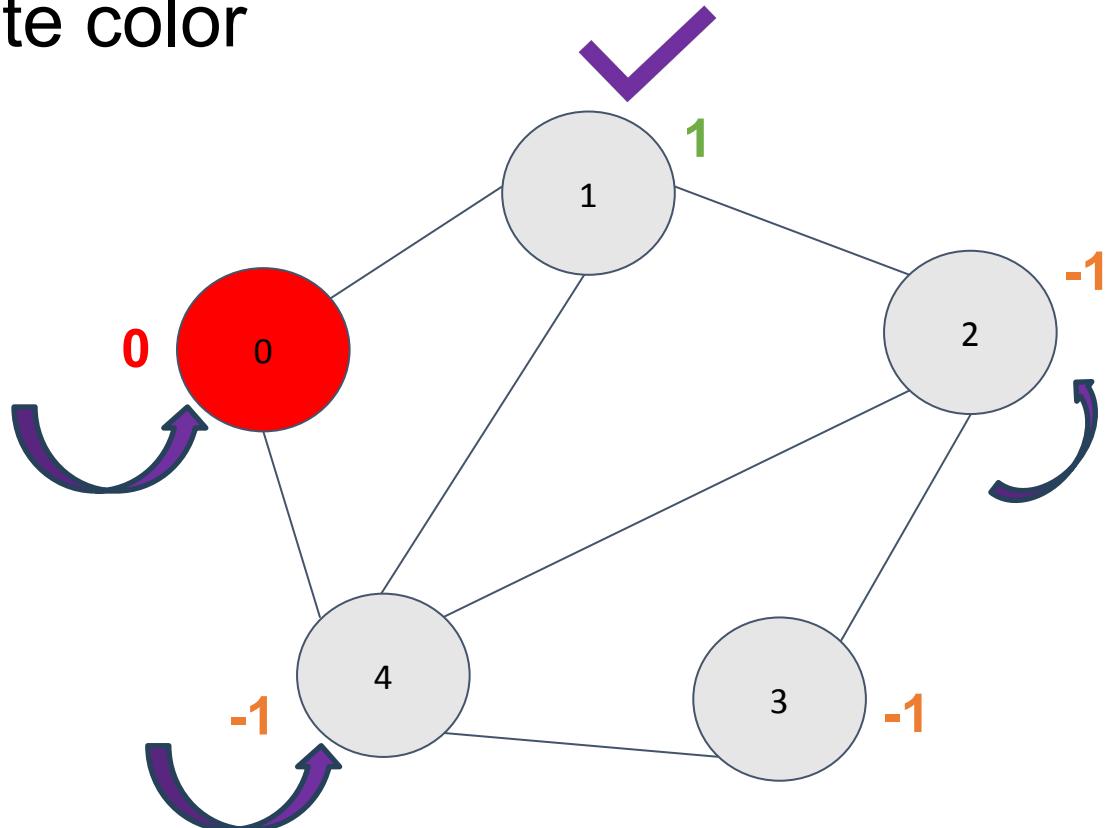
# The elements of backtracking

- Pass one node
- **Check if it is Valid with a particular color -> color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color



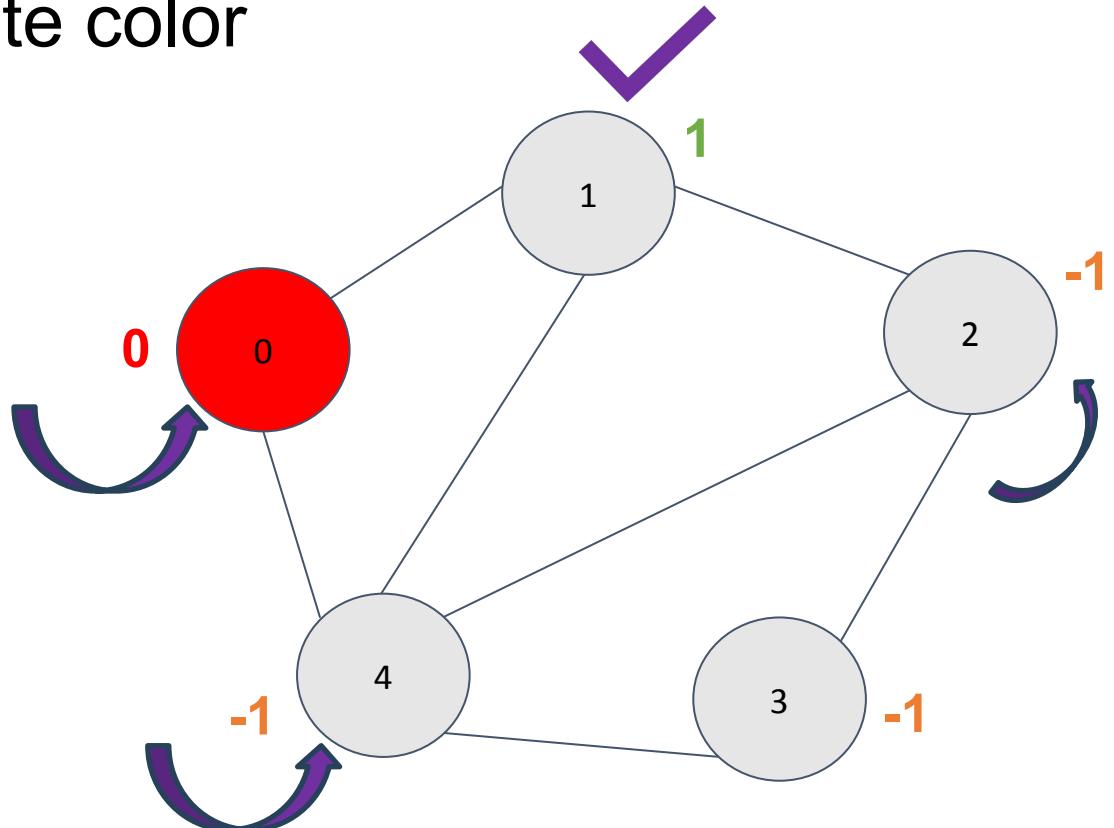
# The elements of backtracking

- Pass one node
- **Check if it is Valid with another color particular color  $\rightarrow$  color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color



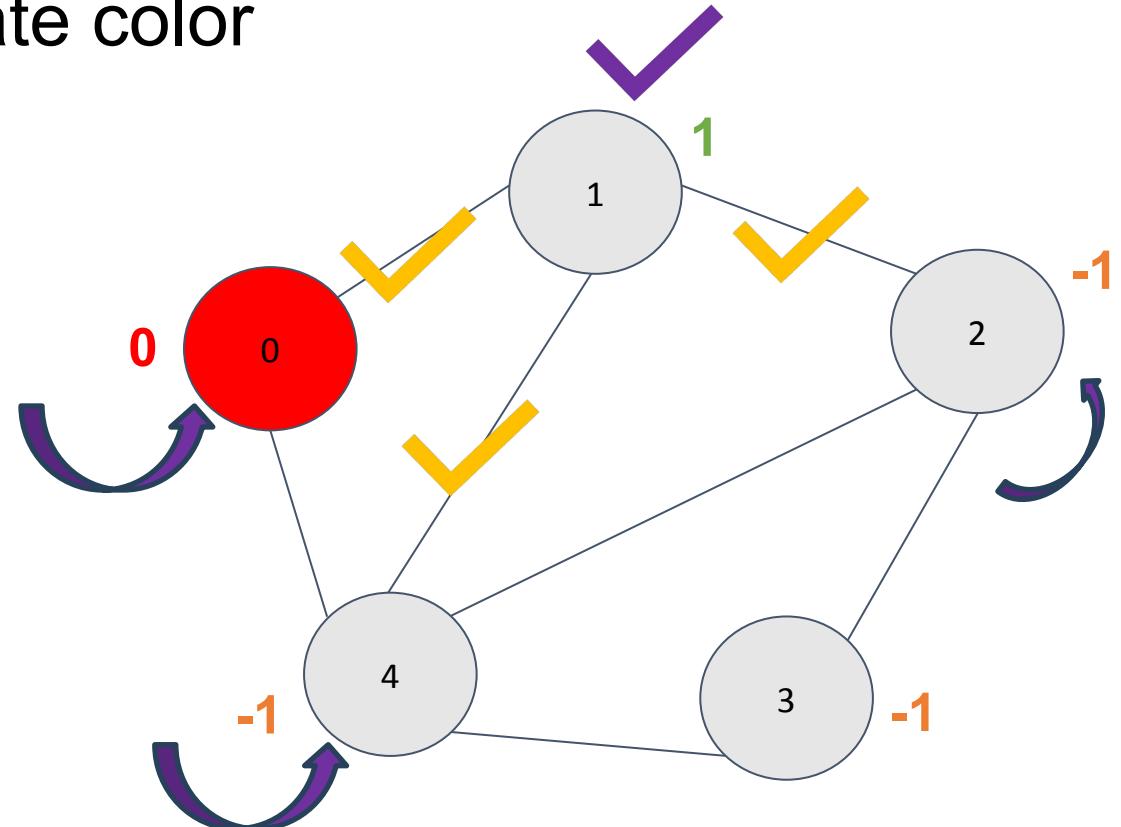
# The elements of backtracking

- Pass one node
- **Check if it is Valid with another color particular color -> color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color



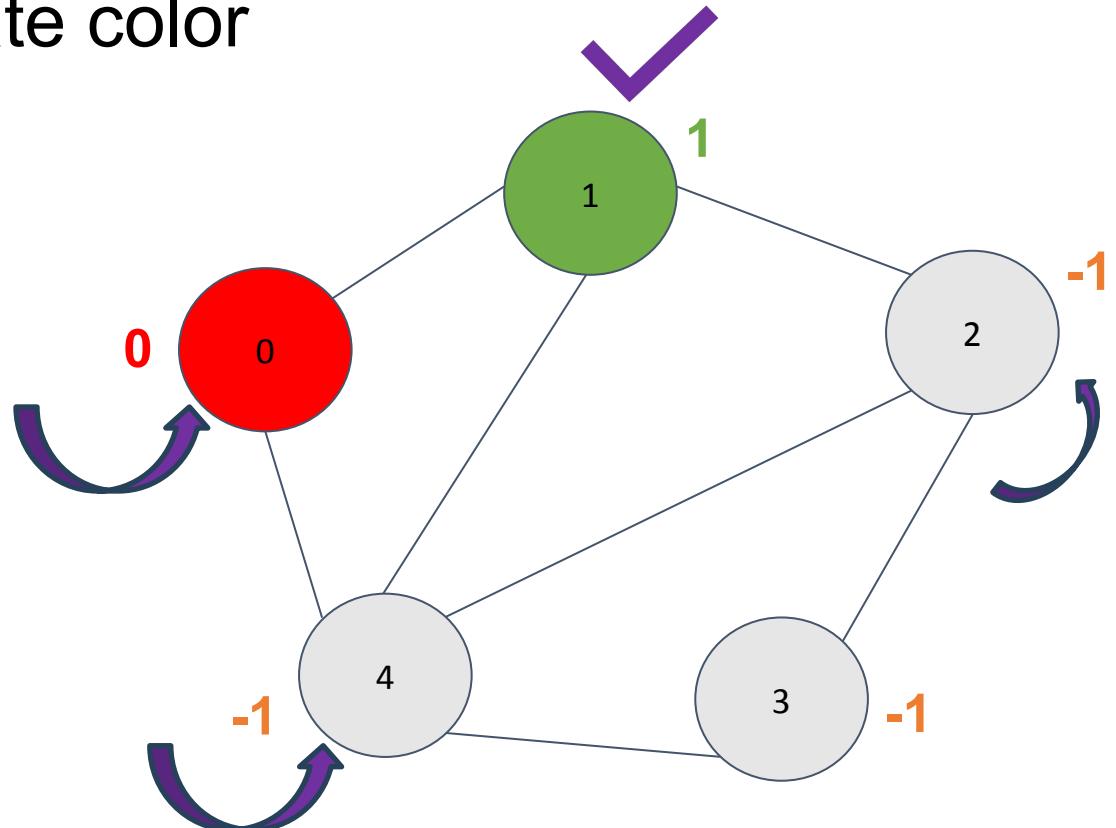
# The elements of backtracking

- Pass one node
- **Check if it is Valid with another color -> color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color



# The elements of backtracking

- Pass one node
- **Check if it is Valid with another color particular color -> color it**
- Check if it goal state  Print the entire vertices with color
- Pass the next node with candidate color

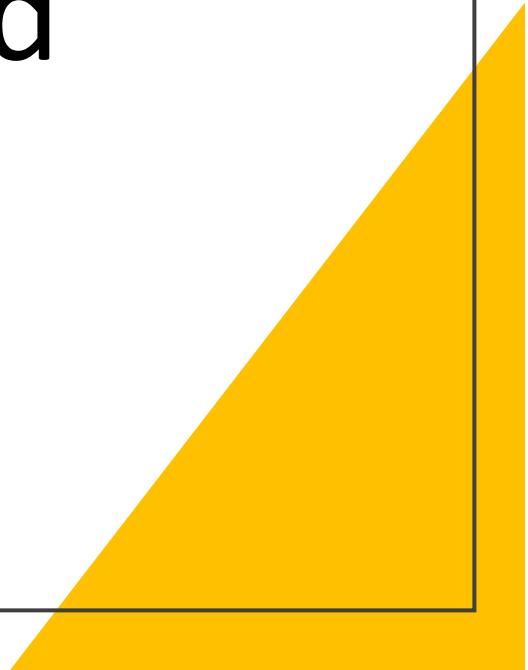


# CSE 215

# Data Structures and

# Algorithms-II

Branch and Bound



# Branch N Bound



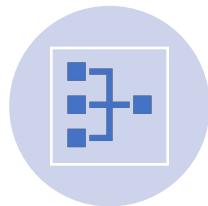
Branch and bound is an algorithm design paradigm for discrete and combinatoric optimisation problems, as well as mathematical optimisation.



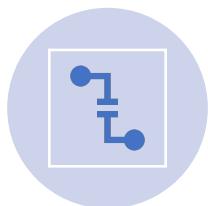
A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions.



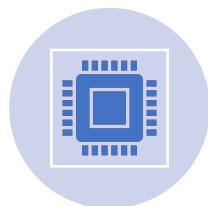
That is, the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.



The algorithm explores branches of this tree, which represent the subsets of the solution set.



Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution.



And is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

# Backtracking vs Branch N Bound



| Parameter    | Backtracking                                                                                                                                                                                                                                            | Branch and Bound                                                                                                                                                                                                                                            |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Approach     | Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem. | Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution than the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution. |
| Traversal    | Backtracking traverses the state space tree by <a href="#">DFS(Depth First Search)</a> manner.                                                                                                                                                          | Branch-and-Bound traverse the tree in any manner, <a href="#">DFS</a> or <a href="#">BFS</a> .                                                                                                                                                              |
| Function     | Backtracking involves feasibility function.                                                                                                                                                                                                             | Branch-and-Bound involves a bounding function.                                                                                                                                                                                                              |
| Problems     | Backtracking is used for solving Decision Problem.                                                                                                                                                                                                      | Branch-and-Bound is used for solving Optimisation Problem.                                                                                                                                                                                                  |
| Searching    | In backtracking, the state space tree is searched until the solution is obtained.                                                                                                                                                                       | In Branch-and-Bound as the optimum solution may be present anywhere in the state space tree, so the tree need to be searched completely.                                                                                                                    |
| Efficiency   | Backtracking is more efficient.                                                                                                                                                                                                                         | Branch-and-Bound is less efficient.                                                                                                                                                                                                                         |
| Applications | Useful in solving <a href="#">N-Queen Problem</a> , <a href="#">Sum of subset</a> , <a href="#">Hamilton cycle problem</a> , <a href="#">graph coloring problem</a>                                                                                     | Useful in solving <a href="#">Knapsack Problem</a> , <a href="#">Travelling Salesman Problem</a> .                                                                                                                                                          |
| Solve        | Backtracking can solve almost any problem. (chess, sudoku, etc ).                                                                                                                                                                                       | Branch-and-Bound can not solve almost any problem.                                                                                                                                                                                                          |
| Used for     | Typically backtracking is used to solve decision problems.                                                                                                                                                                                              | Branch and bound is used to solve optimization problems.                                                                                                                                                                                                    |
| Nodes        | Nodes in state space tree are explored in depth first tree.                                                                                                                                                                                             | Nodes in tree may be explored in depth-first or breadth-first order.                                                                                                                                                                                        |
| Next move    | Next move from current state can lead to bad choice.                                                                                                                                                                                                    | Next move is always towards better solution.                                                                                                                                                                                                                |
| Solution     | On successful search of solution in state space tree, search stops.                                                                                                                                                                                     | Entire state space tree is search in order to find optimal solution.                                                                                                                                                                                        |

# Branch and Bound (B & B)

---

- An enhancement of backtracking
  - **Similarity** : A state space tree is used to solve a problem.
  - **Difference** : Used only for optimization problems.
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many branches can be terminated

# Branch and Bound

---

- Search the tree using a breadth-first search (FIFO branch and bound).
- Search the tree as in a BFS, but replace the FIFO queue with a stack (LIFO branch and bound).
- Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound).

The priority of a node  $p$  in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is  $p$ .

# Branch and Bound

---

- FIFO branch and bound finds solution closest to the root.
- Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated).
- Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.

# Branch and Bound

---

- The idea:

Set up a **bounding function**, which is used to compute a **bound** (for the value of the objective function) **at a node** on a state-space tree and determine **if it is promising**

- **Promising** (if the bound is better than the value of the best solution so far): expand beyond the node.
- **Nonpromising** (if the bound is no better than the value of the best solution so far): do not expand beyond the node (pruning the state-space tree).
- The search proceeds until all nodes have been solved or pruned.

# Bounding

---

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or, less than some number (upper bound)
- If we are looking for a maximal optimal (knapsack), then we need an upper bound
  - For example, if the best solution we have found so far has a profit of 12 and the upper bound on a node is 10 then there is no point in expanding the node
    - The node cannot lead to anything better than a 10

# Bounding

---

- We prune (via bounding) when:  
 $(currentBestSolutionCost \geq nodeBound)$
- This tells us that we get more pruning if:
  - The currentBestSolution is high
  - And the nodeBound is low
- So we want to find a high solution quickly and we want the highest possible upper bound
  - One has to factor in the extra computation cost of computing higher upper bounds vs. the expected pruning savings

# Branch and Bound

---

- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

# 0-1 Knapsack

---

- Capacity W is 10
- Upper bound is \$100  
(use fractional value)

| Item | Weight | Value | Value/weight |
|------|--------|-------|--------------|
| 1    | 4      | \$40  | 10           |
| 2    | 7      | \$42  | 6            |
| 3    | 5      | \$25  | 5            |
| 4    | 3      | \$12  | 4            |

# Computing Upper Bound

---

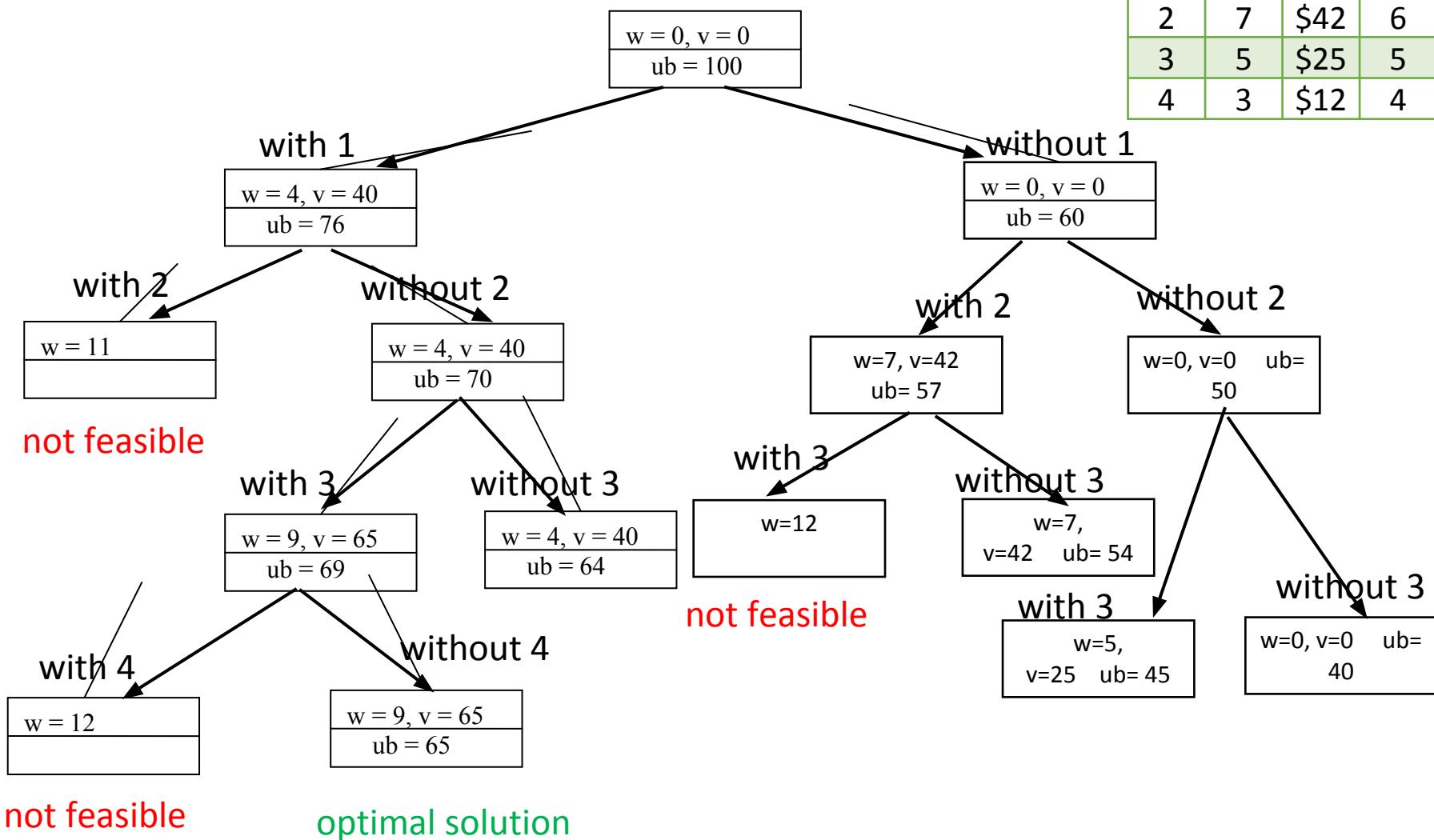
| Item | Weight | Value | $V_i / W_i$ |
|------|--------|-------|-------------|
| 1    | 4      | \$40  | 10          |
| 2    | 7      | \$42  | 6           |
| 3    | 5      | \$25  | 5           |
| 4    | 3      | \$12  | 4           |

- To compute the upper bound, we use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - $ub = (10 - 0) * (\$10) = \$100$
- After we pick item 1, we calculate the upper bound as
  - all of item 1 (4, \$40) + partial of item 2 (6, \$36)
  - $\$40 + (10-4) * (\$6) = \$76$
- If we don't pick item 1:
  - $ub = (10 - 0) * (\$6) = \$60$

# State Space Tree

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

| Item | Weig<br>ht | Valu<br>e | Vi /<br>Wi |
|------|------------|-----------|------------|
| 1    | 4          | \$40      | 10         |
| 2    | 7          | \$42      | 6          |
| 3    | 5          | \$25      | 5          |
| 4    | 3          | \$12      | 4          |



# Pseudocode

---

- Sort the items using Value/Weight in descending order
- Initialize Maximum Profit, **maxProfit=0**
- Create an empty queue, Q ([You can use LIFO or Priority Queue also](#))
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.

While Q is not empty

Do

- Extract an item from Q. Let the extracted item be u.
  - Compute profit of next level node (with the item and without the item). If the profit is more than maxProfit, then update maxProfit.
  - Compute bound( $ub = v_i + (W - w_i) * (v_{i+1} / w_{i+1})$ ) of next level node. If bound is more than maxProfit, then add next level node to Q.
- return **maxProfit**

# Travelling Salesman Problem

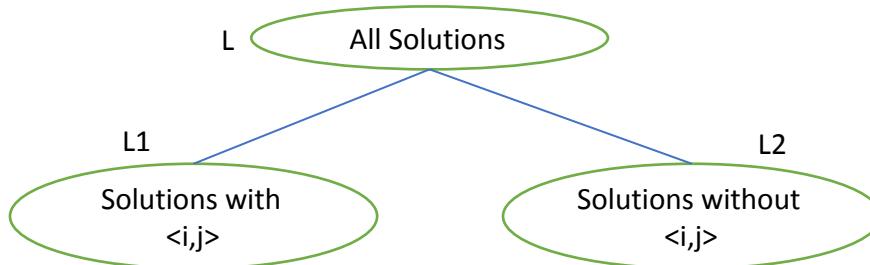
---

- Definition: Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.
- Definitions:
  - A row or column is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.
  - A matrix is reduced iff every row and column is reduced.

# Branching

- **Branching:**

- Each node splits the remaining solutions into two groups: those that include a particular edge and those that exclude that edge.
- Each node has a lower bound.
- Example: Given a graph  $G=(V,E)$ , let  $\langle i,j \rangle \in E$ ,



# Bounding

How to compute the cost of each node?

- Subtract of a constant from any row and any column does not change the optimal solution (The path).
- The cost of the path changes but the path itself.
- Let A be the cost matrix of a  $G=(V,E)$ .
- The cost of each node in the search tree is computed as follows:
  - Let R be a node in the tree and  $A^R$  its reduced matrix
  - The cost of the child  $R^S$ :
    - Set row i and column j to infinity
    - Set  $A(j,i)$  to infinity
    - Reduced S and let RCL be the reduced cost.
    - $C(S) = C^R + RCL + A(i,j)$

# Bounding cont..



**Get the reduced matrix  $A'$  of  $A$  and let  $L$  be value subtracted from  $A$**



**$L$ : represents the lower bound of the path solution**



**The cost of the path is exactly reduced by  $L$ .**



**What to determine the branching edge?**

The rule favors a solution through left subtree rather than right subtree, i.e. the matrix is reduced by a dimension.

Note that the right subtree only sets the branching edge to infinity.

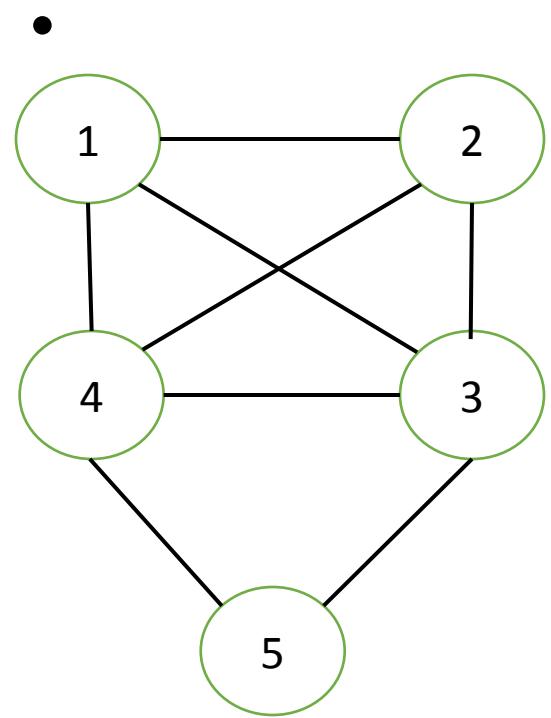
Pick the edge that causes the greatest increase in the lower bound of the right subtree, i.e. the lower bound of the root of the right subtree is greater.

# Cost Determine

Example:

- The reduced cost matrix is done as follows:
  - Change all entries of row  $i$  and column  $j$  to infinity
  - Set  $A(j,1)$  to infinity (assuming the start node is 1)
  - Reduce all rows first and then column of the resulting matrix

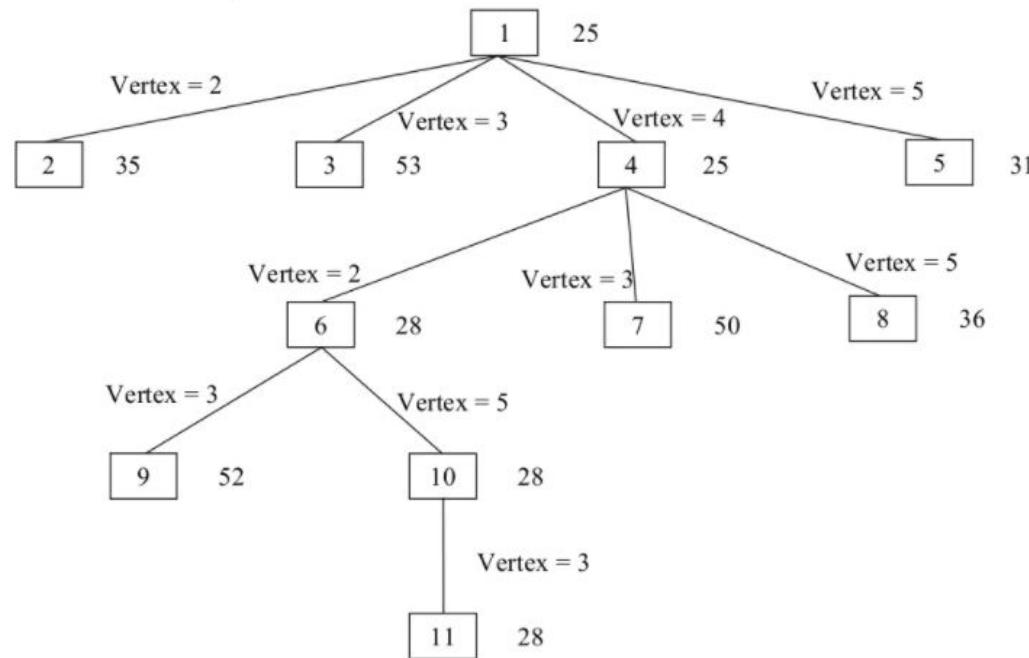
# TSP



Cost matrix

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 20  | 30  | 10  | 11  |
| 15  | inf | 16  | 4   | 2   |
| 3   | 5   | inf | 2   | 4   |
| 19  | 6   | 18  | inf | 3   |
| 16  | 4   | 7   | 16  | inf |

# State Space Tree



- The TSP starts from node 1: **Node 1**

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #1: reduce by 10

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 20  | 30  | 10  | 11  |
| 15  | inf | 16  | 4   | 2   |
| 3   | 5   | inf | 2   | 4   |
| 19  | 6   | 18  | inf | 3   |
| 16  | 4   | 7   | 16  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 15  | inf | 16  | 4   | 2   |
| 3   | 5   | inf | 2   | 4   |
| 19  | 6   | 18  | inf | 3   |
| 16  | 4   | 7   | 16  | inf |

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #2: reduce by 2

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 15 & \text{inf} & 16 & 4 & 2 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #3: reduce by 2

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 1 & 3 & \text{inf} & 0 & 2 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1

- Row #4: reduce by 3

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 19  | 6   | 18  | inf | 3   |
| 16  | 4   | 7   | 16  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 16  | 3   | 15  | inf | 0   |
| 16  | 4   | 7   | 16  | inf |

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #5: reduce by 4

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 16  | 3   | 15  | inf | 0   |
| 16  | 4   | 7   | 16  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 16  | 3   | 15  | inf | 0   |
| 12  | 0   | 3   | 12  | inf |

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#1: reduce by 1

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 16  | 3   | 15  | inf | 0   |
| 12  | 0   | 3   | 12  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 12  | inf | 14  | 2   | 0   |
| 0   | 3   | inf | 0   | 2   |
| 15  | 3   | 15  | inf | 0   |
| 11  | 0   | 3   | 12  | inf |

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#2: It is reduced (**no change**)

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 13  | inf | 14  | 2   | 0   |
| 1   | 3   | inf | 0   | 2   |
| 16  | 3   | 15  | inf | 0   |
| 12  | 0   | 3   | 12  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | 10  | 20  | 0   | 1   |
| 12  | inf | 14  | 2   | 0   |
| 0   | 3   | inf | 0   | 2   |
| 15  | 3   | 15  | inf | 0   |
| 11  | 0   | 3   | 12  | inf |

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#3: Reduce by 3

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#4: It is reduced (**no change**)

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#5: It is reduced (no change)

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 2**
  - Cost of edge  $\langle 1,2 \rangle$  is:  $A(1,2) = 10$
  - Set row#1 = **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set column #2= **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set  $A(2,1) = \text{inf}$

**The resulting cost matrix is:**

The matrix is reduced:

RCL=0 (As min is 0 for all row

And columns)

The cost of node 2 (Considering Vertex 2 from vertex 1) is:

Cost(2)

$$= \text{cost}(1) + \text{RCL} + A(1,2) = 25 + 0 + 10$$

$$= 35$$

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | 11  | 2   | 0   |
| 0   | inf | inf | 0   | 2   |
| 15  | inf | 12  | inf | 0   |
| 11  | inf | 0   | 12  | inf |

- Choose to go to vertex 3: **Node 3**
  - Cost of edge  $<1,3>$  is:  $A(1,2) = 17$  (in the reduced matrix)
  - Set row#1 = **inf** since we are staring from node 1
  - Set column #3= **inf** since we are choosing edge  $<1,3>$
  - Set  $A(3,1) = \text{inf}$

**The resulting cost matrix is:**

Reduce the matrix:

- Rows are reduced
- The colum are reduced except for column # 1:
  - Reduce column 1 by 11

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 12  | inf | inf | 2   | 0   |
| inf | 3   | inf | 0   | 2   |
| 15  | 3   | inf | inf | 0   |
| 11  | 0   | inf | 12  | inf |

- The lower bound is:

- RCL=11

- The cost of going through node 3 is:

- $$\begin{aligned} \text{Cost}(3) &= \text{cost}(1) + \text{RCL} + A(1,3) = 25+11+17 \\ &= 53 \end{aligned}$$

Before

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 12  | inf | inf | 2   | 0   |
| inf | 3   | inf | 0   | 2   |
| 15  | 3   | inf | inf | 0   |
| 11  | 0   | inf | 12  | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 1   | inf | inf | 2   | 0   |
| inf | 3   | inf | 0   | 2   |
| 4   | 3   | inf | inf | 0   |
| 0   | 0   | inf | 12  | inf |

After

- Choose to go to vertex 4: **Node 4**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,4 \rangle$  is:  $A(1,4) = 0$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #4= **inf** since we are choosing edge  $\langle 1,4 \rangle$
    - Set  $A(4,1) = \text{inf}$
- The resulting cost matrix is:**

Reduce the matrix:

- Rows are reduced
- Columns are reduced

- The cost of going through node 4 is:
  - $\text{Cost}(4) = \text{cost}(1) + \text{RCL} + A(1,4)$
  - $= 25+0+0$
  - $= 25$

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 12  | inf | 11  | inf | 0   |
| 0   | 3   | inf | inf | 2   |
| inf | 3   | 12  | inf | 0   |
| 11  | 0   | 0   | inf | inf |

- Choose to go to vertex 5: **Node 5**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,5 \rangle$  is:  $A(1,5) = 1$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #5= **inf** since we are choosing edge  $\langle 1,5 \rangle$
    - Set  $A(5,1) = \text{inf}$
- The resulting cost matrix is:**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & 2 & \text{inf} \\ 0 & 3 & \text{inf} & 0 & \text{inf} \\ 15 & 3 & 12 & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2: by 2
    - Reduce Row# 4: by 3
  - Columns are reduced
- The lower bound is:
  - $RCL = 2+3 = 5$
- The cost of going through node 5 is:
  - $\text{Cost}(5) = \text{cost}(1) + RCL + A(1,5)$   
 $= 25+5+1$   
 $= 31$

Reduce by 2

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 10  | inf | 9   | 0   | inf |
| 0   | 3   | inf | 0   | inf |
| 15  | 3   | 12  | inf | inf |
| inf | 0   | 0   | 12  | inf |

Reduce by 3

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 10  | inf | 9   | 0   | inf |
| 0   | 3   | inf | 0   | inf |
| 12  | 0   | 9   | inf | inf |
| inf | 0   | 0   | 12  | inf |

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 → 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 → 3
  - 4:  $\text{cost}(4) = 25$ , path: 1 → 4
  - 5:  $\text{cost}(5) = 31$ , path: 1 → 5
- Explore the node with the lowest cost: Node 4 has a cost of 25
- Vertices to be explored from node 4: 2, 3, and 5

- Now we are starting from the cost matrix at node 4 is:

$$\text{Cost}(4) = 25$$
$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & 0 \\ 0 & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 6** (path 1→4→2)
  - Cost of edge <4,2> is:  $A(4,2) = 3$
  - Set row#4 = **inf** since we are considering edge <4,2>
  - Set column # 2= **inf** since we are considering edge <4,2>
  - Set  $A(2,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced
- The lower bound is  $RCL = 0$
- The cost of going through node 2 is:

$$\text{Cost}(6) = \text{cost}(4) + RCL + A(4,2)$$

$$= 25 + 0 + 3$$

$$= 28$$

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | 11  | inf | 0   |
| 0   | inf | inf | inf | 2   |
| inf | inf | inf | inf | inf |
| 11  | inf | 0   | inf | inf |

- Choose to go to vertex 3: **Node 7** (path 1→4→3)
  - Cost of edge <4,3> is:  $A(4,3) = 12$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,3>
  - Set  $A(3,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & \text{inf} & \text{inf} & 0 \\ \text{inf} & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & 0 & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3:
  - Reduce column#1:
- The lower bound is:
  - $RCL = 2+11 = 13$
- Considering vertex 3 from vertex 4 is:
  - $\text{Cost}(7) = \text{cost}(4) + RCL + A(4,3)$   
 $= 25+13+12$   
 $= 50$

Reduce by 2

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 12  | inf | inf | inf | 0   |
| inf | 1   | inf | inf | 0   |
| inf | inf | inf | inf | inf |
| 11  | 0   | inf | inf | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| 1   | inf | inf | inf | 0   |
| inf | 1   | inf | inf | 0   |
| inf | inf | inf | inf | inf |
| 0   | 0   | inf | inf | inf |

Reduce by 11

- Choose to go to vertex 5: **Node 8** (path 1→4→5)
  - Cost of edge <4,5> is:  $A(4,5) = 0$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,5>
  - Set  $A(5,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & \text{inf} \\ 0 & 3 & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2
  - Columns are reduced
- The lower bound is:
  - $RCL = 11 = 11$
- Considering vertex 5 from vertex 4 is:
  - $\text{Cost}(8) = \text{cost}(4) + RCL + A(4,5)$   
 $= 25 + 11 + 0$   
 $= 36$

Reduce by 11

|      |     |     |     |     |
|------|-----|-----|-----|-----|
| inf  | inf | inf | inf | inf |
| 1    | inf | 0   | inf | inf |
| 0    | 3   | inf | inf | inf |
| :inf | inf | inf | inf | inf |
| inf  | 0   | 0   | inf | inf |

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ↗ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ↗ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ↗ 5
  - 6:  $\text{cost}(6) = 28$ , path: 1 ↗ 4 ↗ 2
  - 7:  $\text{cost}(7) = 50$ , path: 1 ↗ 4 ↗ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ↗ 4 ↗ 5
- Explore the node with the lowest cost: Node 6 has a cost of 28
- Vertices to be explored from node 6: 3, and 5

- Now we are starting from the cost matrix at node 6 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 11 & \text{inf} & 0 \\ 0 & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 9** (path 1→4→2→3)

- Cost of edge <2,3> is:  $A(2,3) = 11$
- Set row#2 = **inf** since we are considering edge <2,3>
- Set column # 3= **inf** since we are considering edge <2,3>
- Set  $A(3,1) = \text{inf}$

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3
  - Reduce column# 1

• The lower bound is:

- $RCL = 2 + 11 = 13$

• Considering vertex 3 from vertex

2 is:

- $\text{Cost}(9) = \text{cost}(6) + RCL + A(2,3)$   
 $= 28 + 13 + 11$   
 $= 52$

By 2

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | 0   |
| inf | inf | inf | inf | inf |
| 11  | inf | inf | inf | inf |

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | 0   |
| inf | inf | inf | inf | inf |
| 0   | inf | inf | inf | inf |

- Choose to go to vertex 5: **Node 10** (path 1→4→2→5)
  - Cost of edge <2,5> is:  $A(2,5) = 0$
  - Set row#2 = **inf** since we are considering edge <2,3>
  - Set column # 3= **inf** since we are considering edge <2,3>
  - Set  $A(5,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 3 from vertex 2 is:

$$\text{Cost}(10) = \text{cost}(6) + RCL + A(2,3)$$

$$= 28 + 0 + 0 \\ = 28$$

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| 0   | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | 0   | inf | inf |

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ⊕ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ⊕ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ⊕ 5
  - 7:  $\text{cost}(7) = 50$ , path: 1 ⊕ 4 ⊕ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ⊕ 4 ⊕ 5
  - 9:  $\text{cost}(9) = 52$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 3
  - 10:  $\text{cost}(2) = 28$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 5
- Explore the node with the lowest cost: Node 10 has a cost of 28
- Vertices to be explored from node 10: 3

- Now we are starting from the cost matrix at node 10 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 0 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 11** (path 1?4?2?5?3)

- Cost of edge  $\langle 5,3 \rangle$  is:  $A(5,3) = 0$
- Set row#5= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set column # 3= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set  $A(3,1) = \text{inf}$

- **Reduce the matrix:**

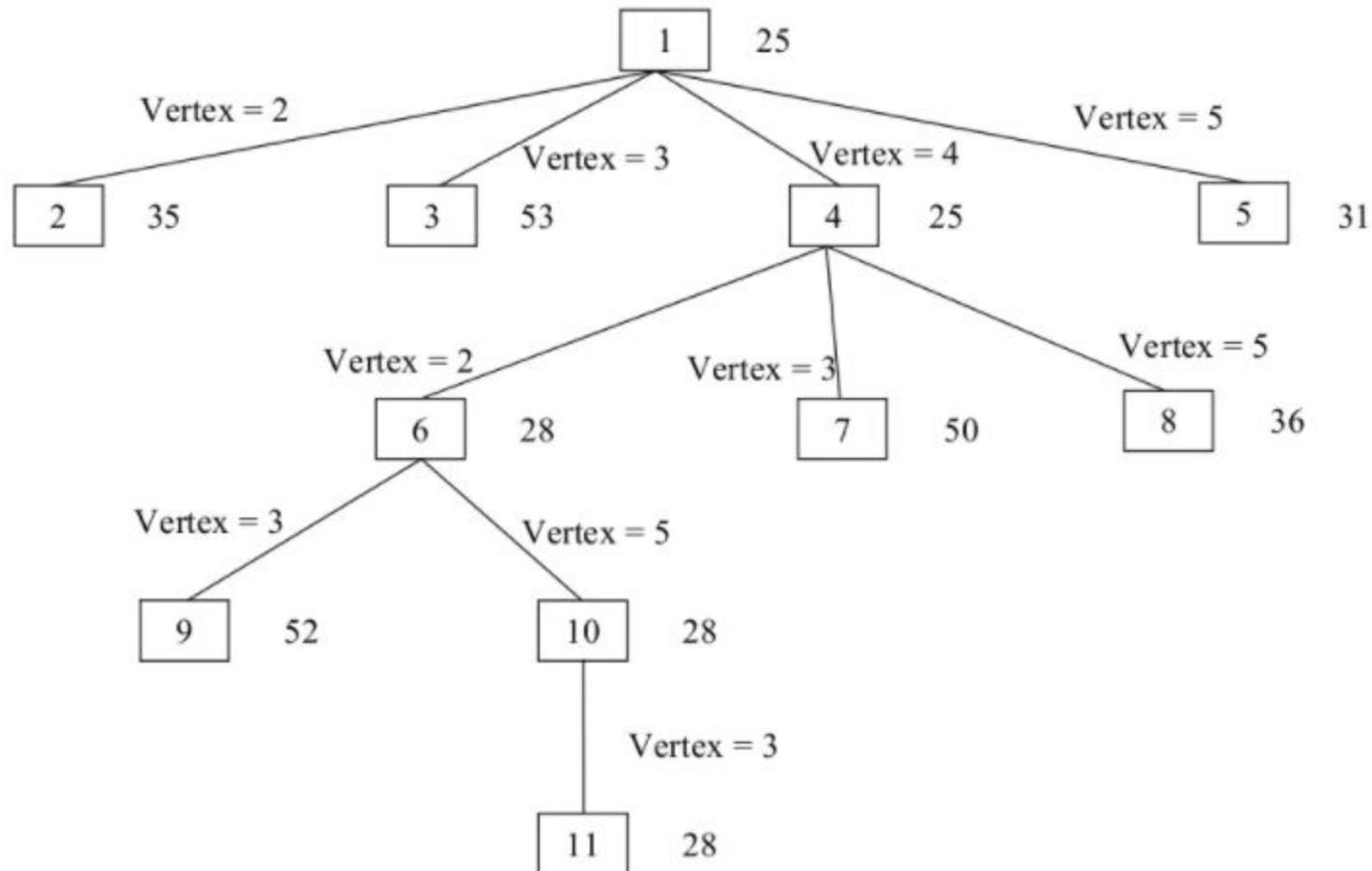
- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 5 from vertex 3 is:

$$\begin{aligned} \text{Cost}(11) &= \text{cost}(10) + RCL + A(5,3) \\ &= 28 + 0 + 0 \\ &= 28 \end{aligned}$$

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |
| inf | inf | inf | inf | inf |

# State Space Tree



The short tour: path 1 → 4 → 2 → 5 → 3 → 1)

*Thank You*

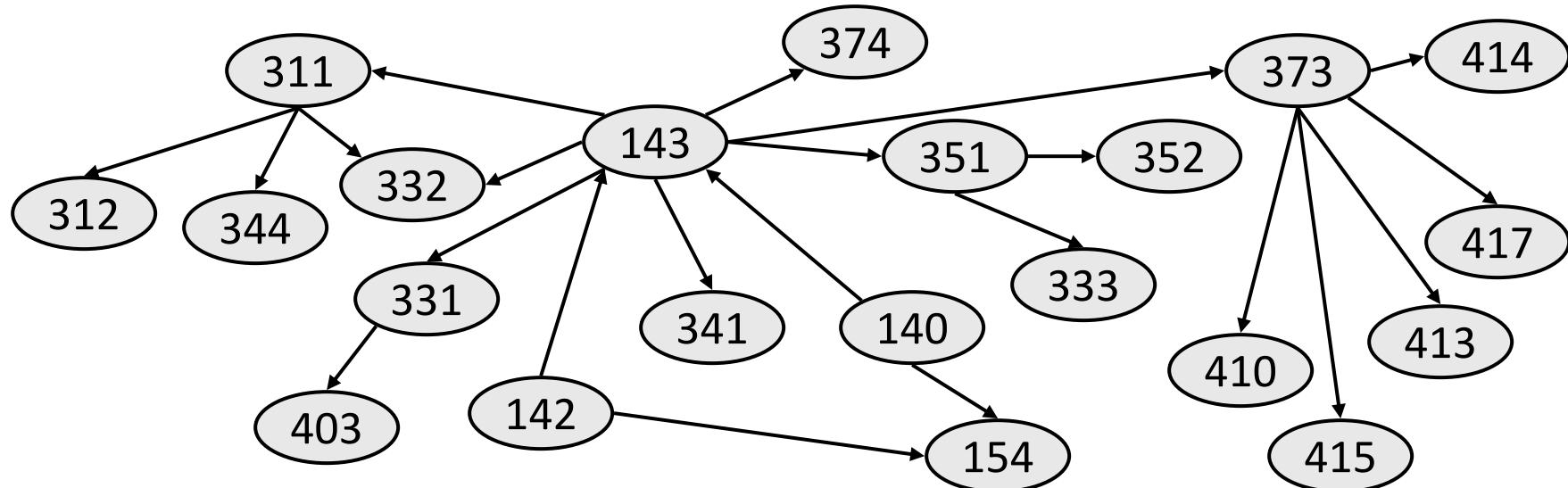
---

# CSE 216

Topological Sort

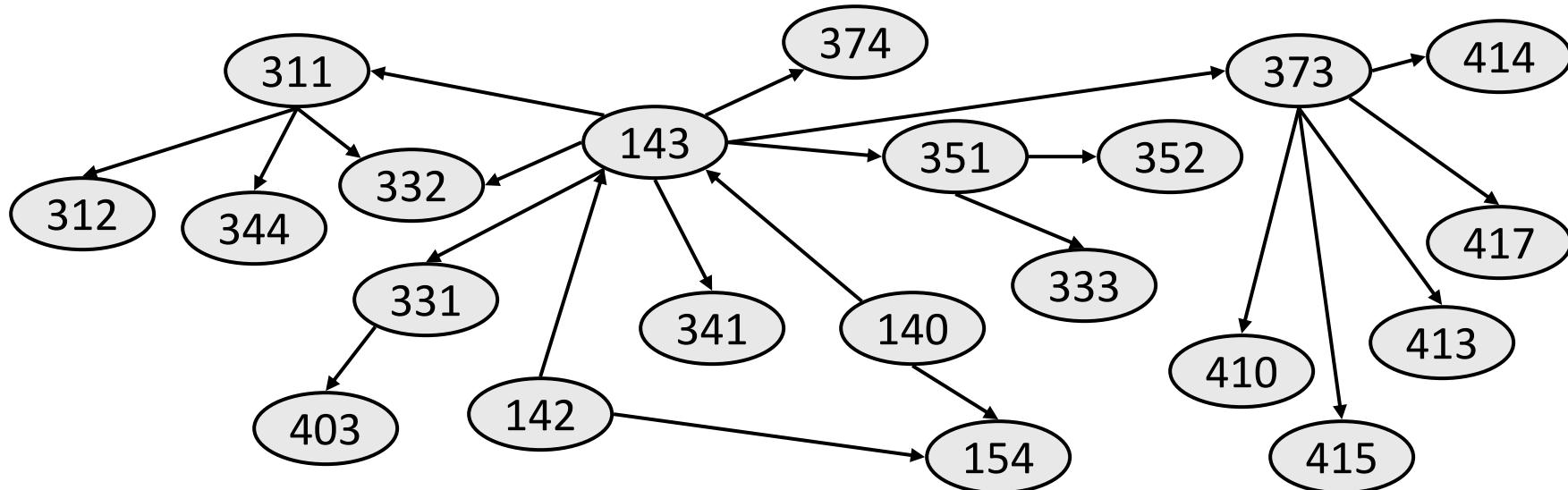
# Ordering a graph

- Suppose we have a directed acyclic graph (DAG) of courses, and we want to find an order in which the courses can be taken.



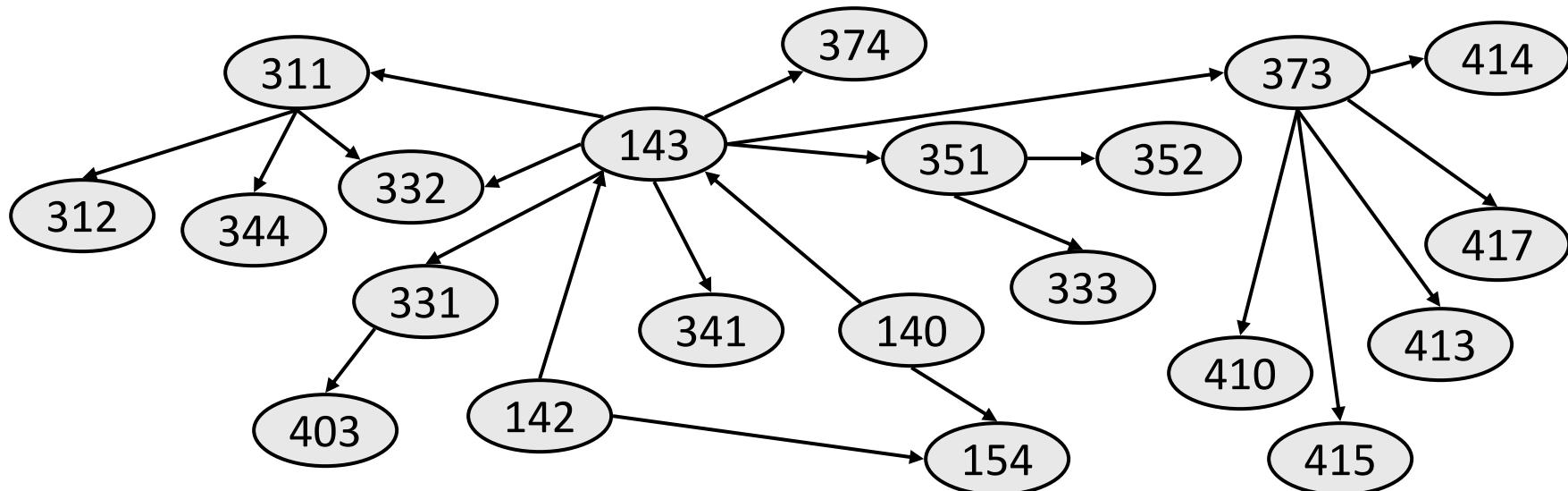
# Ordering a graph

- Suppose we have a directed acyclic graph (DAG) of courses, and we want to find an order in which the courses can be taken.
  - Must take all prereqs before you can take a given course. Example:
    - [142, 143, 140, 154, 341, 374, 331, 403, 311, 332, 344, 312, 351, 333, 352, 373, 414, 410, 417, 413, 415]
    - There might be more than one allowable ordering.
  - How can we find a valid ordering of the vertices?



# Topological Sort

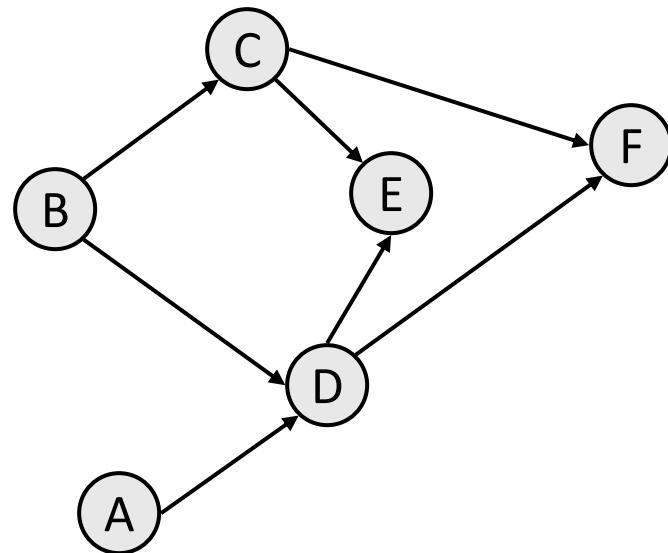
- **topological sort:** Given a graph  $G = (V, E)$ , a total ordering of  $G$ 's vertices such that for every edge  $(v, w)$  in  $E$ , vertex  $v$  precedes  $w$  in the ordering. Examples:
  - determining the order to recalculate updated cells in a spreadsheet
  - finding an order to recompile files that have dependencies
    - (any problem of finding an order to perform tasks with dependencies)



# Topo sort example

---

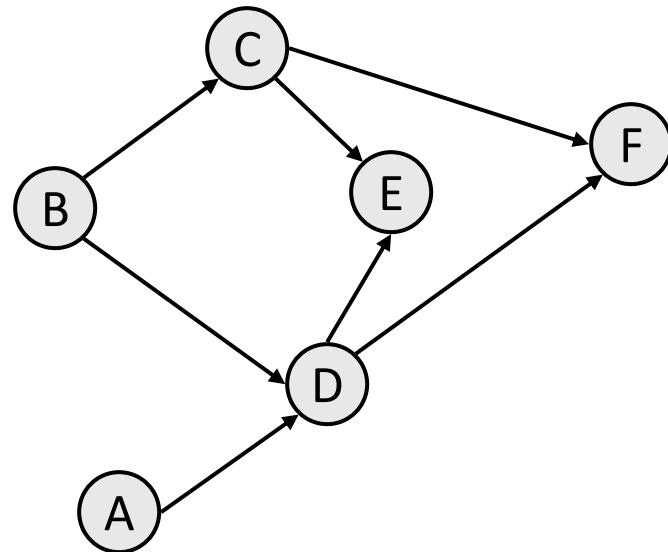
- How many valid topological sort orderings can you find for the vertices in the graph below?



# Topo sort example

---

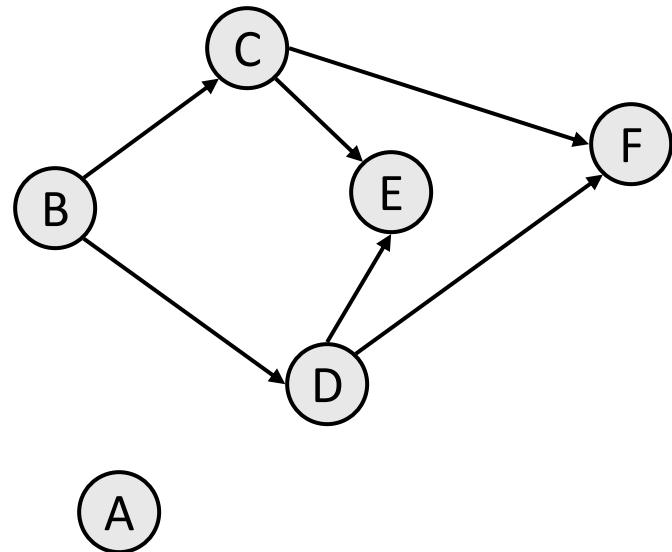
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A]



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A]

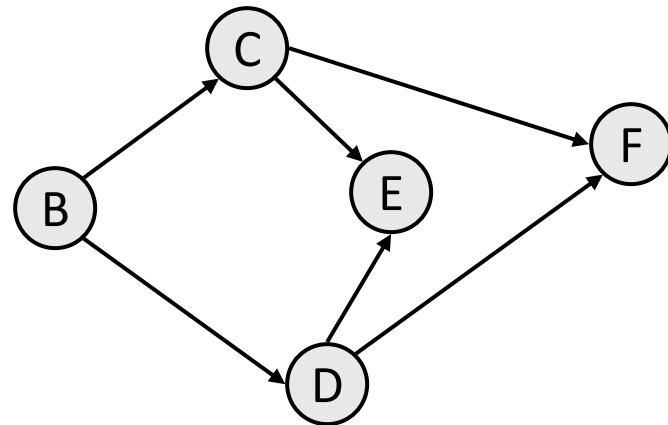


# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?

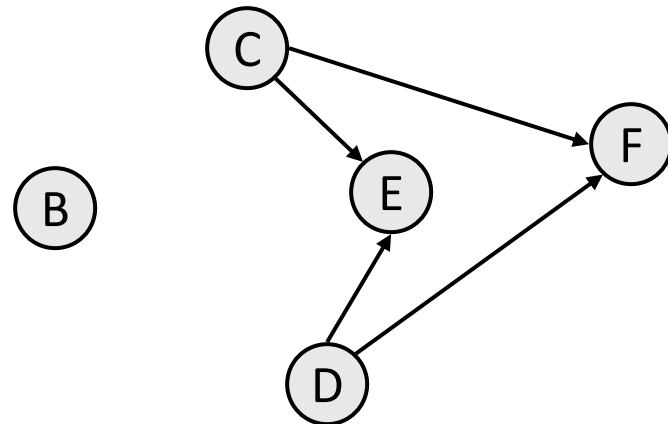
- [A]



# Topo sort example

---

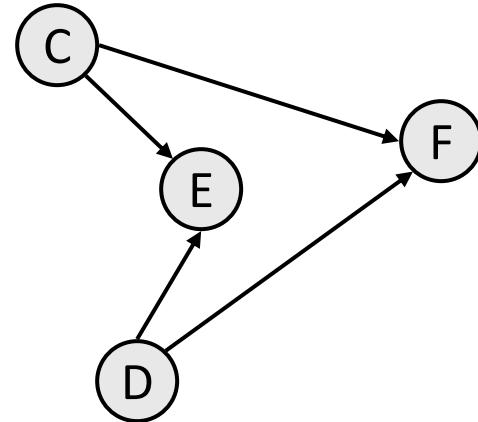
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B]



# Topo sort example

---

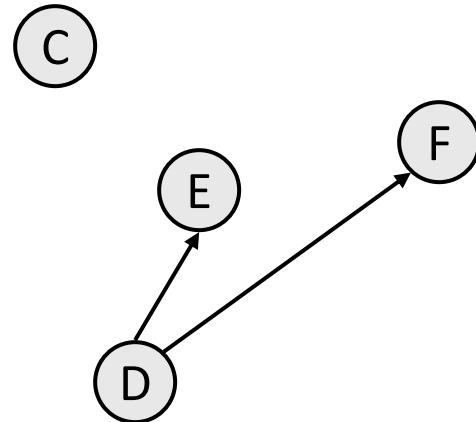
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B]



# Topo sort example

---

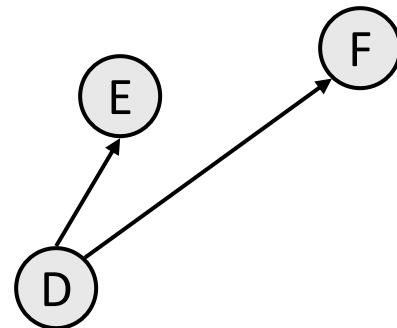
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C]



# Topo sort example

---

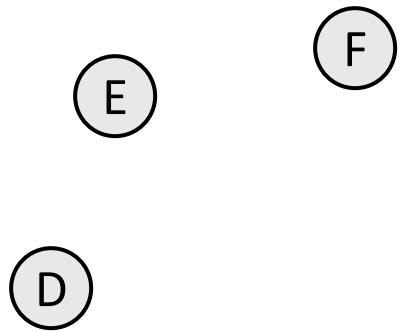
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C]



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D]



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D]



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D, E]



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D, E]

F

# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D, E, F]

F

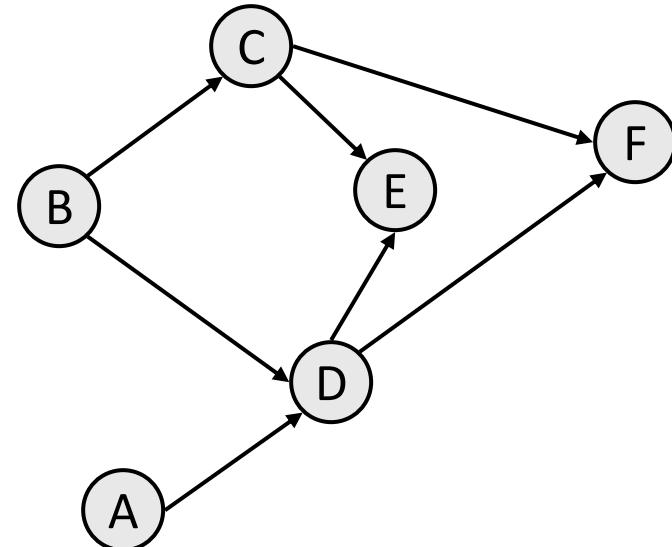
# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D, E, F]

# Topo sort example

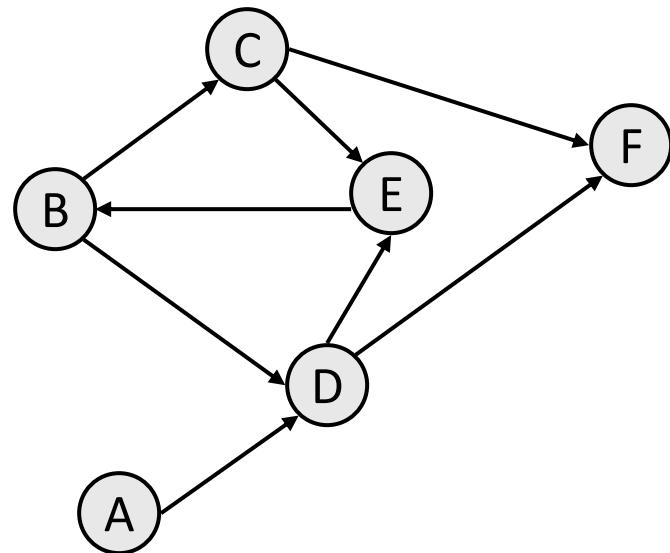
- How many valid topological sort orderings can you find for the vertices in the graph below?
  - [A, B, C, D, E, F], [A, B, C, D, F, E],
  - [A, B, D, C, E, F], [A, B, D, C, F, E],
  - [B, A, C, D, E, F], [B, A, C, D, F, E],
  - [B, A, D, C, E, F], [B, A, D, C, F, E],
  - [B, C, A, D, E, F], [B, C, A, D, F, E],
  - ...



# Topo sort example

---

- How many valid topological sort orderings can you find for the vertices in the graph below?

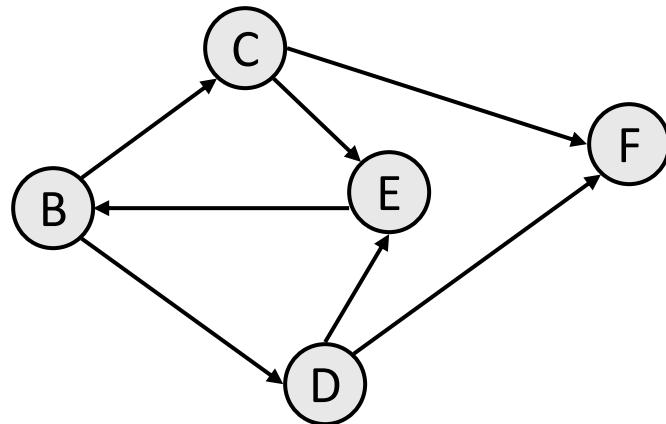


# Topo sort example

---

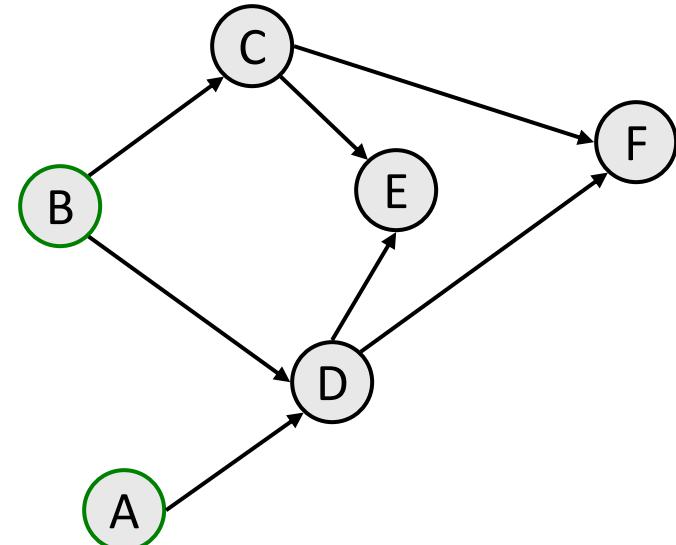
- How many valid topological sort orderings can you find for the vertices in the graph below?

- [A]



# Topo sort: Algorithm 1

- function topologicalSort():
  - *ordering* := { }.
  - Repeat until graph is empty:
    - Find a vertex  $v$  with in-degree of 0 (no incoming edges).
      - (If there is no such vertex, the graph cannot be sorted; stop.)
    - Delete  $v$  and all of its outgoing edges from the graph.
    - *ordering* +=  $v$  .



# Revised algorithm

---

- We don't want to literally delete vertices and edges from the graph while trying to topological sort it; so let's revise the algorithm:
  - $map := \{\text{each vertex} \rightarrow \text{its in-degree}\}$ .
  - $queue := \{\text{all vertices with in-degree} = 0\}$ .
  - $ordering := \{\}$ .
  - Repeat until queue is empty:
    - Dequeue the first vertex  $v$  from the queue.
    - $ordering += v$ .
    - Decrease the in-degree of all  $v$ 's neighbors by 1 in the  $map$ .
    - $queue += \{\text{any neighbors whose in-degree is now } 0\}$ .
  - If all vertices are processed, success.  
Otherwise, there is a cycle.

# Topo sort example 2

- function topologicalSort():

- *map* := {each vertex → its in-degree}.

- *queue* := {all vertices with in-degree = 0}.

- *ordering* := { }.

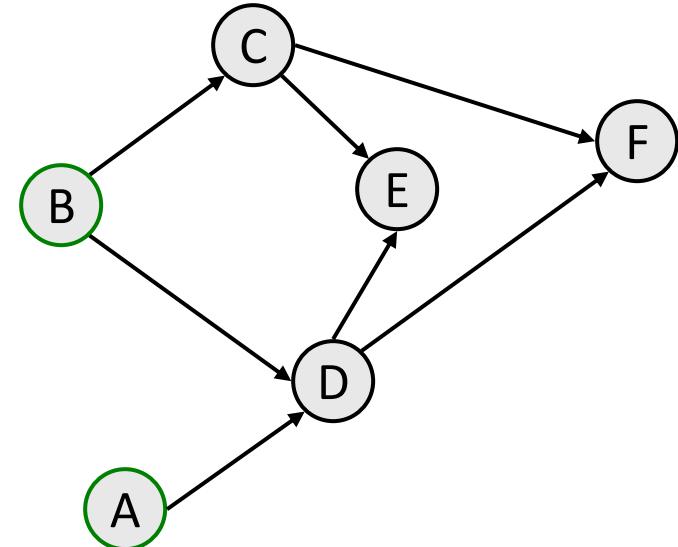
- Repeat until queue is empty:

- Dequeue the first vertex *v* from the queue.

- *ordering* += *v*.

- Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

- *queue* += {any neighbors whose in-degree is now 0}.



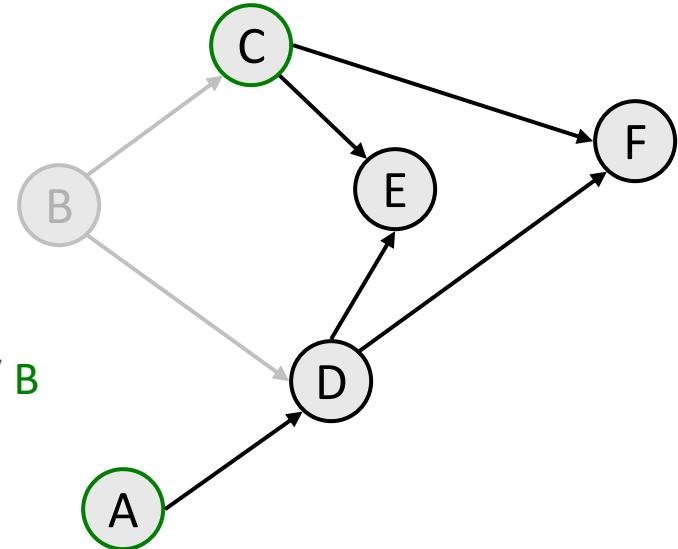
- map := { A=0, B=0, C=1, D=2, E=2, F=2 }

- queue := { B, A }

- ordering := { }

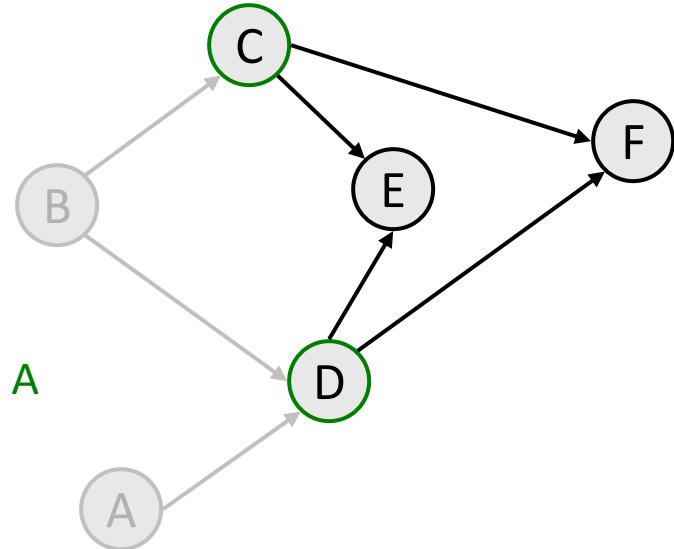
# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // *B*
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // *C, D* neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
- *map* := { A=0, B=0, **C=0**, **D=1**, E=2, F=2 }
- *queue* := { A, **C** }
- *ordering* := { **B** }



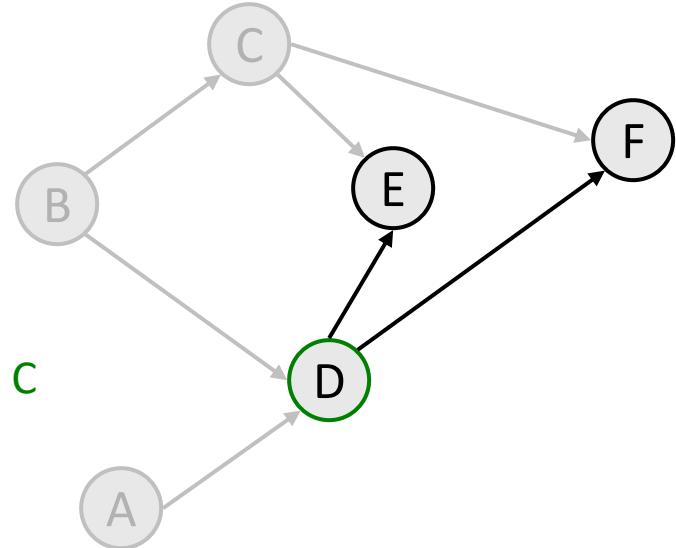
# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // A
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // D neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
- map := { A=0, B=0, C=0, **D=0**, E=2, F=2 }
- queue := { C, **D** }
- ordering := { B, **A** }



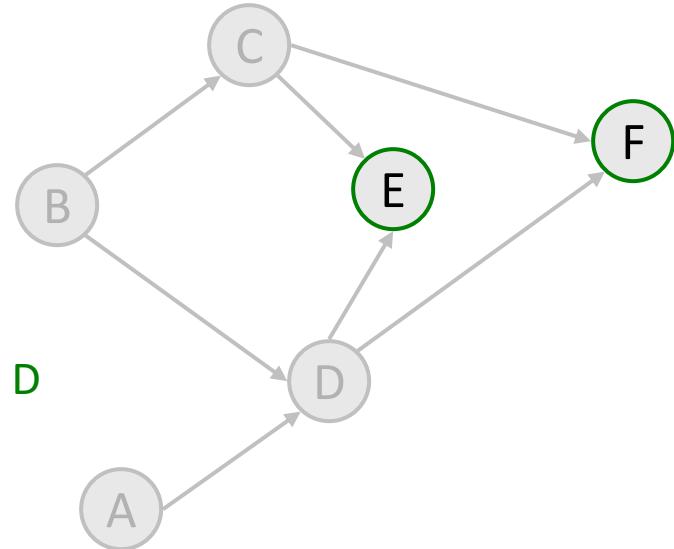
# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // C
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // E, F neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
  - map := { A=0, B=0, C=0, D=0, E=1, F=1 }
  - queue := { D }
  - ordering := { B, A, C }



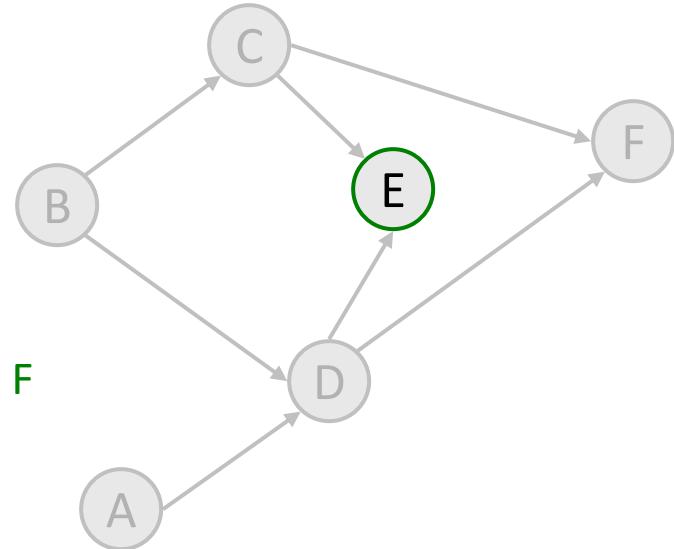
# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // D
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // F, E neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
- map := { A=0, B=0, C=0, D=0, E=0, F=0 }
- queue := { F, E }
- ordering := { B, A, C, D }



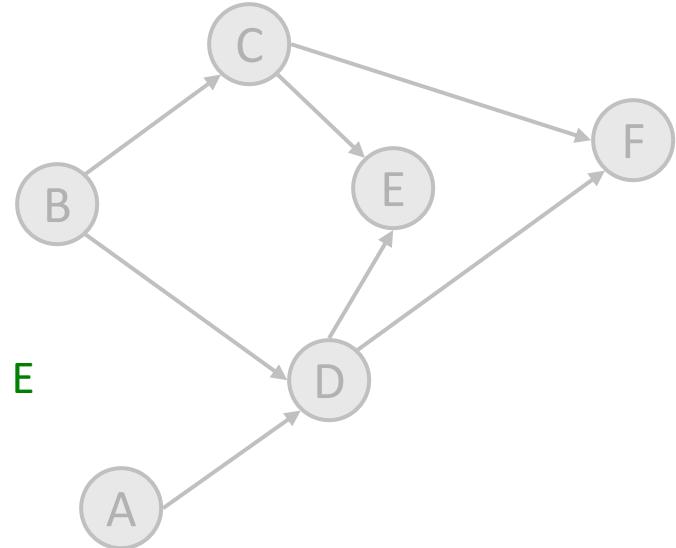
# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // F
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // none neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
  - *map* := { A=0, B=0, C=0, D=0, E=0, F=0 }
  - *queue* := { E }
  - *ordering* := { B, A, C, D, F }



# Topo sort example 2

- function topologicalSort():
  - *map* := {each vertex → its in-degree}.
  - *queue* := {all vertices with in-degree = 0}.
  - *ordering* := { }.
  - Repeat until queue is empty:
    - Dequeue the first vertex *v* from the queue. // E
    - *ordering* += *v*.
    - Decrease the in-degree of all *v*'s // none neighbors by 1 in the *map*.
    - *queue* += {any neighbors whose in-degree is now 0}.
  - map := { A=0, B=0, C=0, D=0, E=0, F=0 }
  - queue := { }
  - ordering := { B, A, C, D, F, E }



# Topo sort runtime

---

- What is the runtime of our topological sort algorithm?
  - (with an "adjacency map" graph internal representation)
  - function `topologicalSort()`:
    - $map := \{\text{each vertex} \rightarrow \text{its in-degree}\}$ . //  $O(V)$
    - $queue := \{\text{all vertices with in-degree } = 0\}$ .
    - $ordering := \{ \}$ .
    - Repeat until queue is empty: //  $O(V)$ 
      - Dequeue the first vertex  $v$  from the queue. //  $O(1)$
      - $ordering += v$ . //  $O(1)$
      - Decrease the in-degree of all  $v$ 's neighbors by 1 in the  $map$ . //  $O(E)$  for all passes
      - $queue += \{\text{any neighbors whose in-degree is now } 0\}$ .
  - Overall:  **$O(V + E)$**  ; essentially  $O(V)$  time on a sparse graph (fast!)

---

**Thank You**