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	100

Pick Pivot Element

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	100

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 >= pivot

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

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	**20**	10	80	60	50	7	30	**100**

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

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 data[too_big_index] <= data[pivot]

   ++too_big_index

pivot_index = 0

40	20	**10**	80	60	50	7	30	**100**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

pivot_index = 0

40	20	10	**80**	60	50	7	30	**100**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

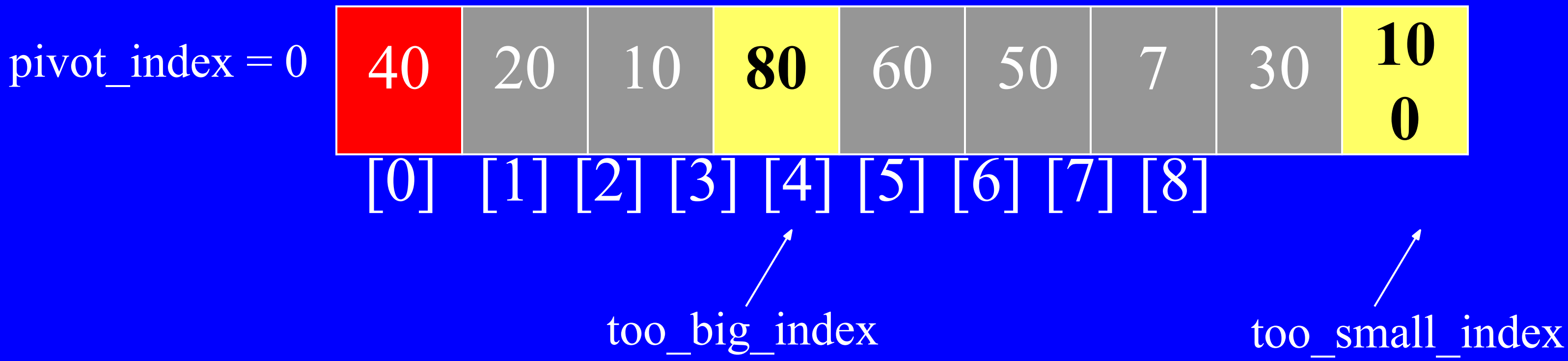
too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index



1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

pivot_index = 0

40	20	10	**80**	60	50	7	**30**	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

pivot_index = 0

40	20	10	**80**	60	50	7	**30**	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

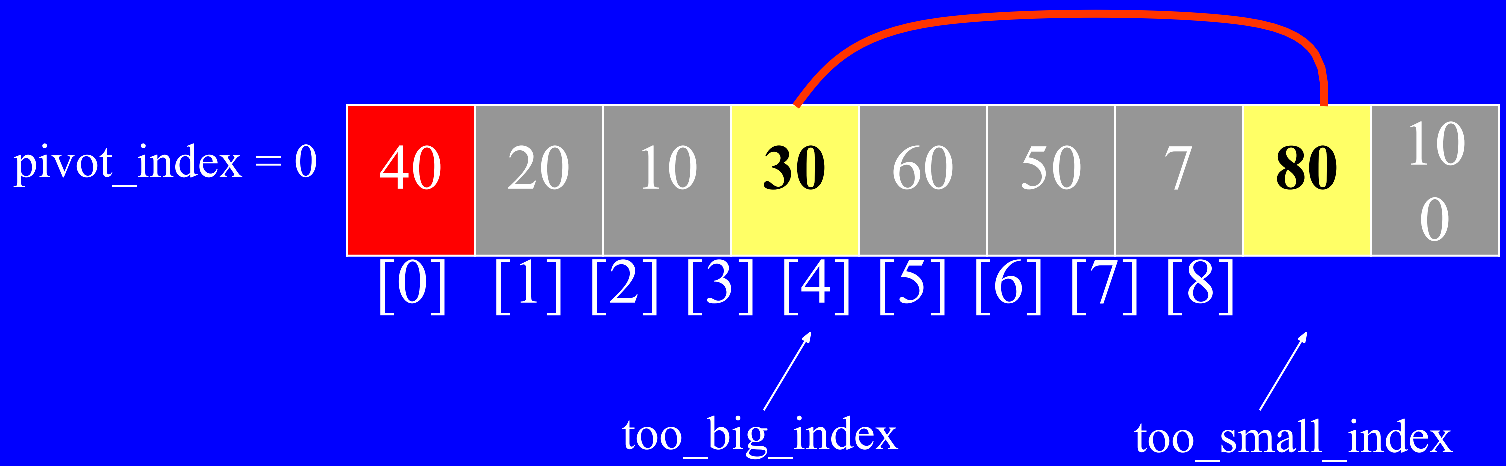
    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]



1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	**30**	60	50	7	**80**	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > 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 data[too_big_index] <= data[pivot]

   ++too_big_index
2. While data[too_small_index] > data[pivot]

   --too_small_index
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	**60**	50	7	**80**	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > 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 data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	**60**	50	**7**	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

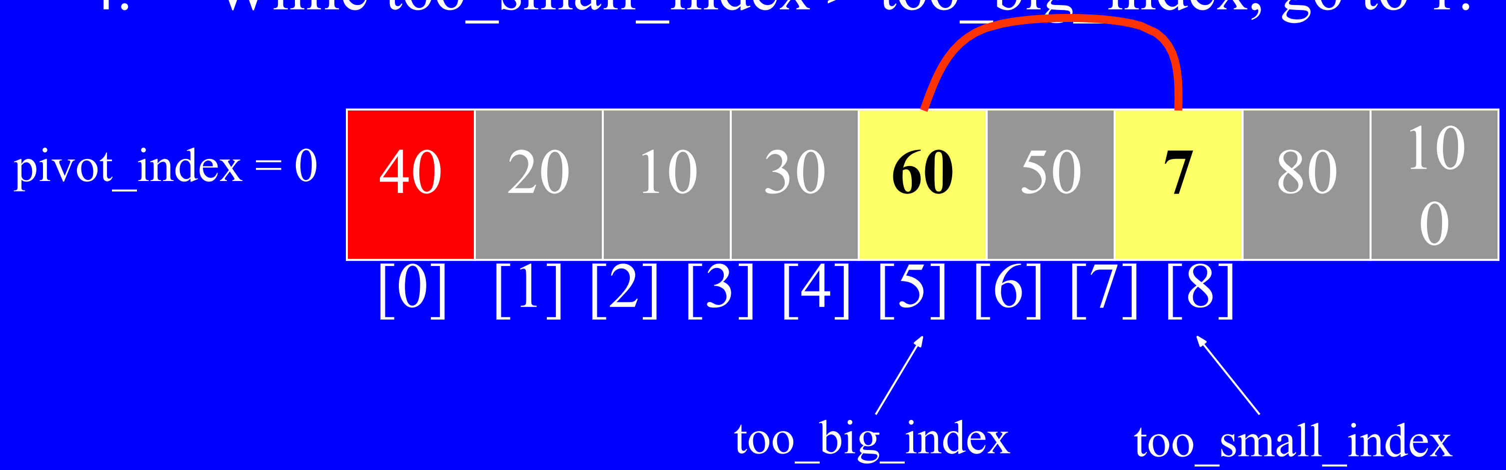
2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	**7**	50	**60**	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	**7**	50	**60**	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	**7**	50	**60**	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	**50**	**60**	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > 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

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	7	**50**	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > 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

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > 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

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

pivot_index = 0	40	20	10	30	**7**	**50**	60	80	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	**7**	**50**	60	80	100

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index
2. While data[too_small_index] > data[pivot]

   --too_small_index
3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 4

7	20	10	30	40	**50**	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

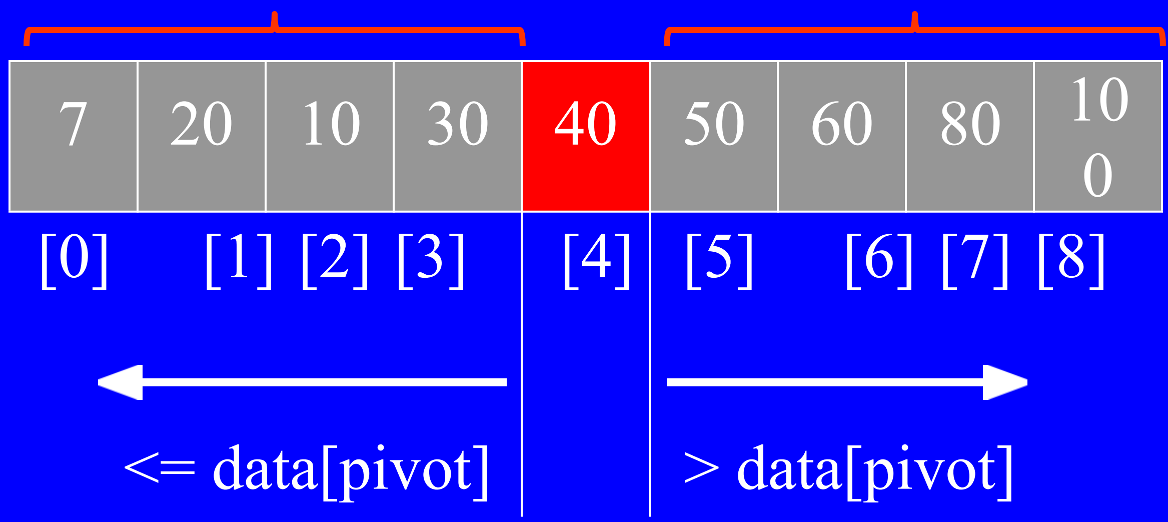
Partition Result

7	20	10	30	40	50	60	80	100
[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:

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	**4**	10	12	13	50	57	63	**100**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

2	4	10	12	13	50	57	63	10 0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

pivot_index = 0

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

    ++too_big_index

2. While data[too_small_index] > data[pivot]

    --too_small_index

3. If too_big_index < too_small_index

    swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

2	4	10	12	13	50	57	63	**100**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

pivot_index = 0

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

 ++too_big_index

2. While data[too_small_index] > data[pivot]

 --too_small_index

3. If too_big_index < too_small_index

 swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

5. Swap data[too_small_index] and data[pivot_index]

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

pivot_index = 0

too_big_index

too_small_index

1. While data[too_big_index] <= data[pivot]

   ++too_big_index

2. While data[too_small_index] > data[pivot]

   --too_small_index

3. If too_big_index < too_small_index

   swap data[too_big_index] and data[too_small_index]

4. While too_small_index > too_big_index, go to 1.

5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	4	10	12	13	50	57	63	**100**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]

> data[pivot]

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: data[n/2].

Use this mid value as pivot.

Improved Pivot Selection

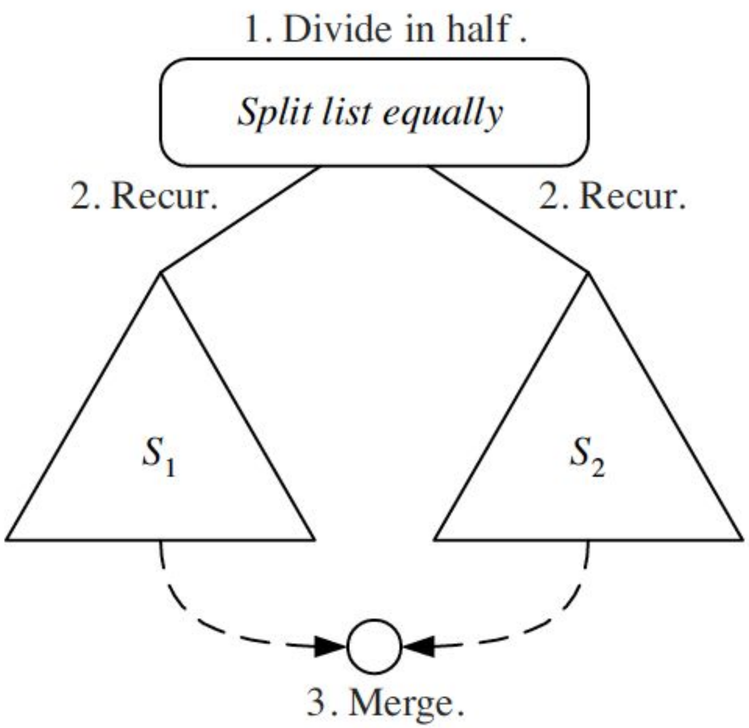
Pick median value of three elements from data array:
data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Merge Sort

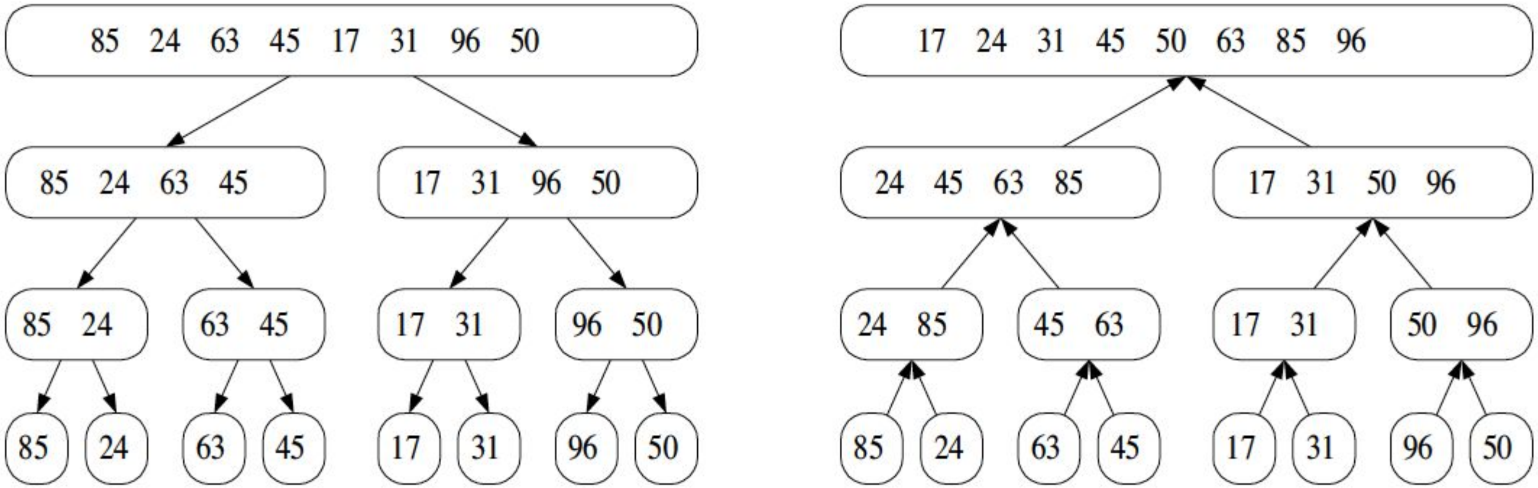
Divide-and-Conquer

- 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

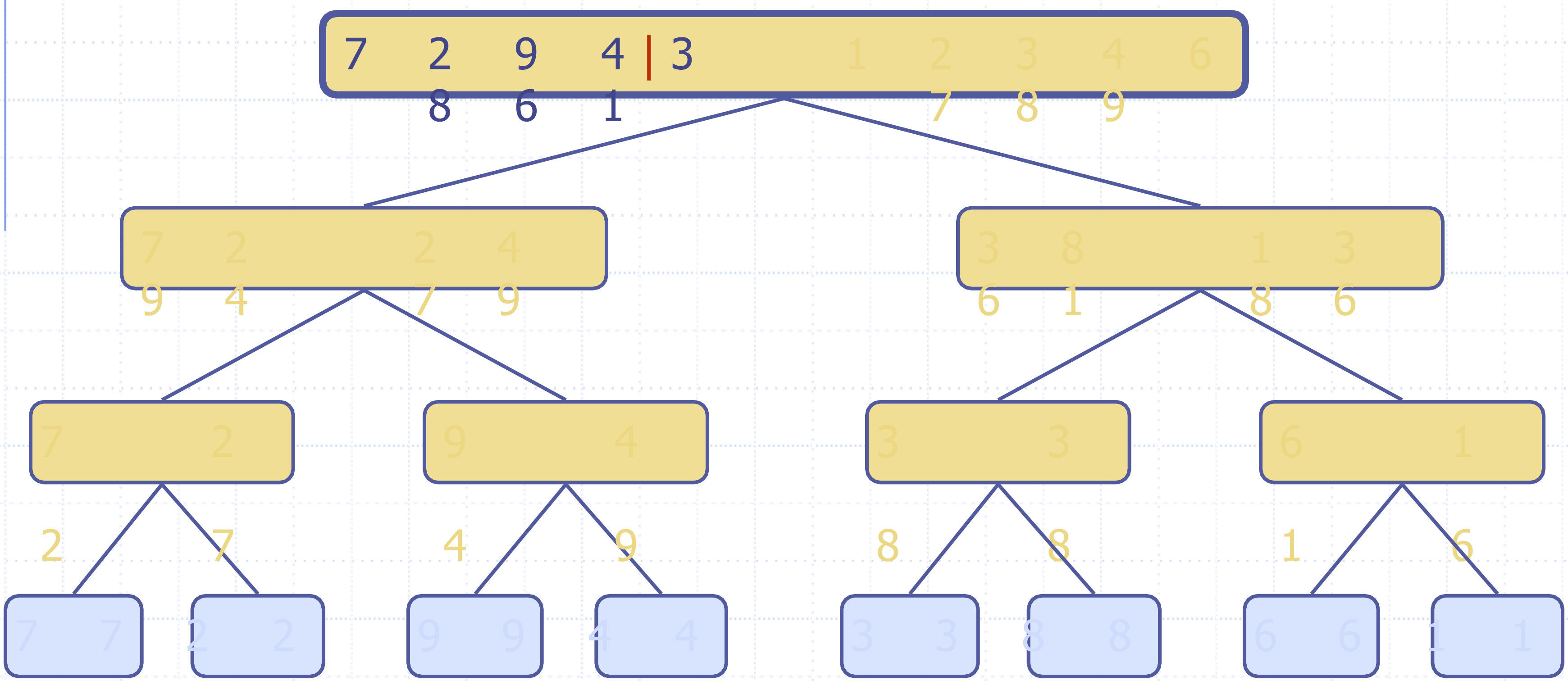
```
MERGE_SORT(arr, start, end)

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
```

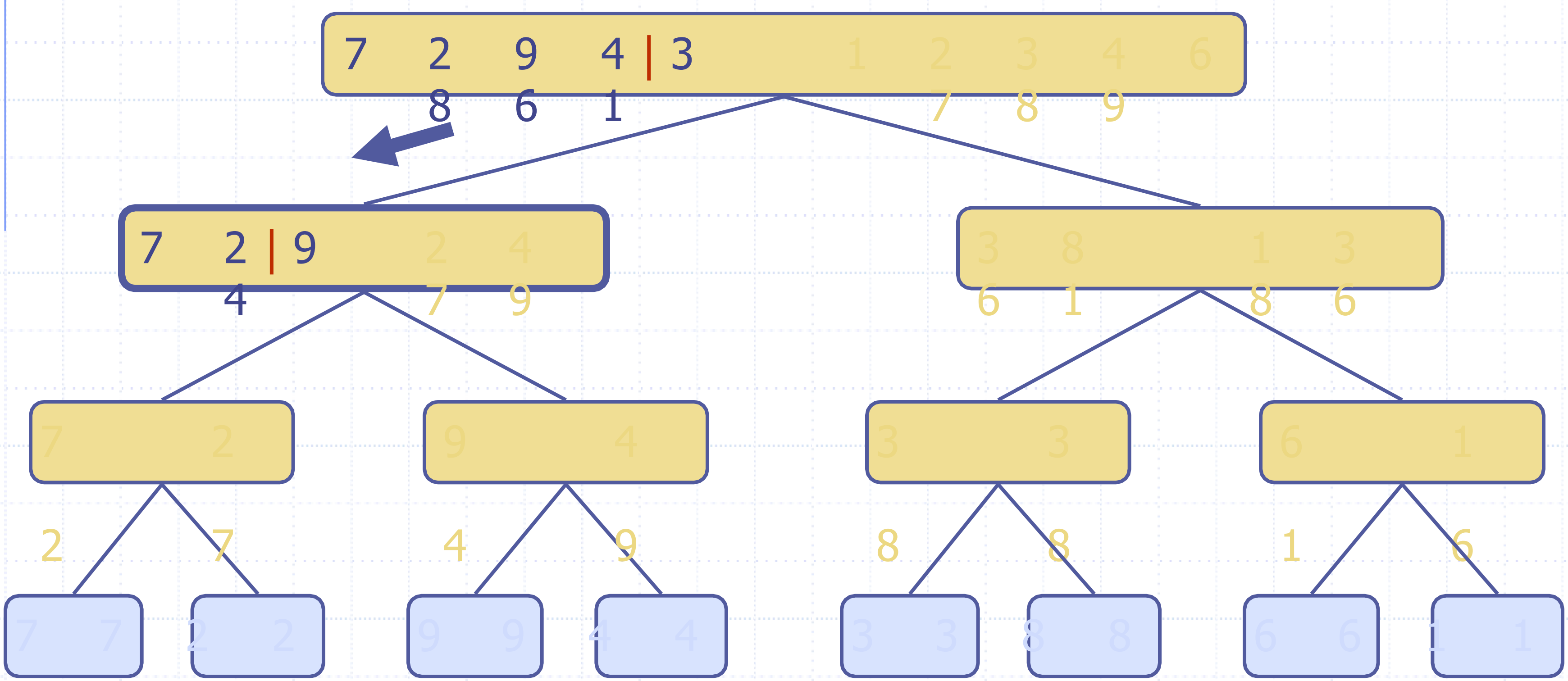
Execution Example

- Partition



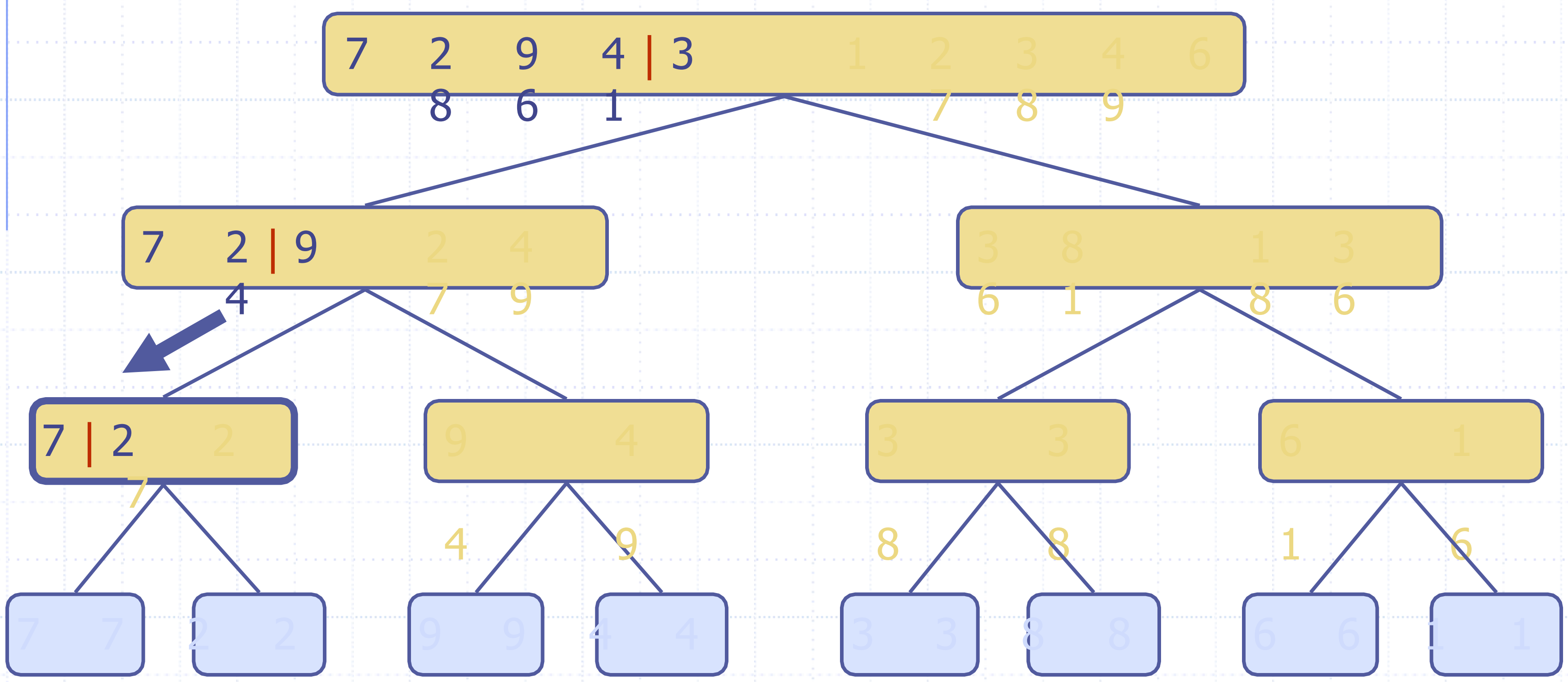
Execution Example (cont.)

- Recursive call, partition



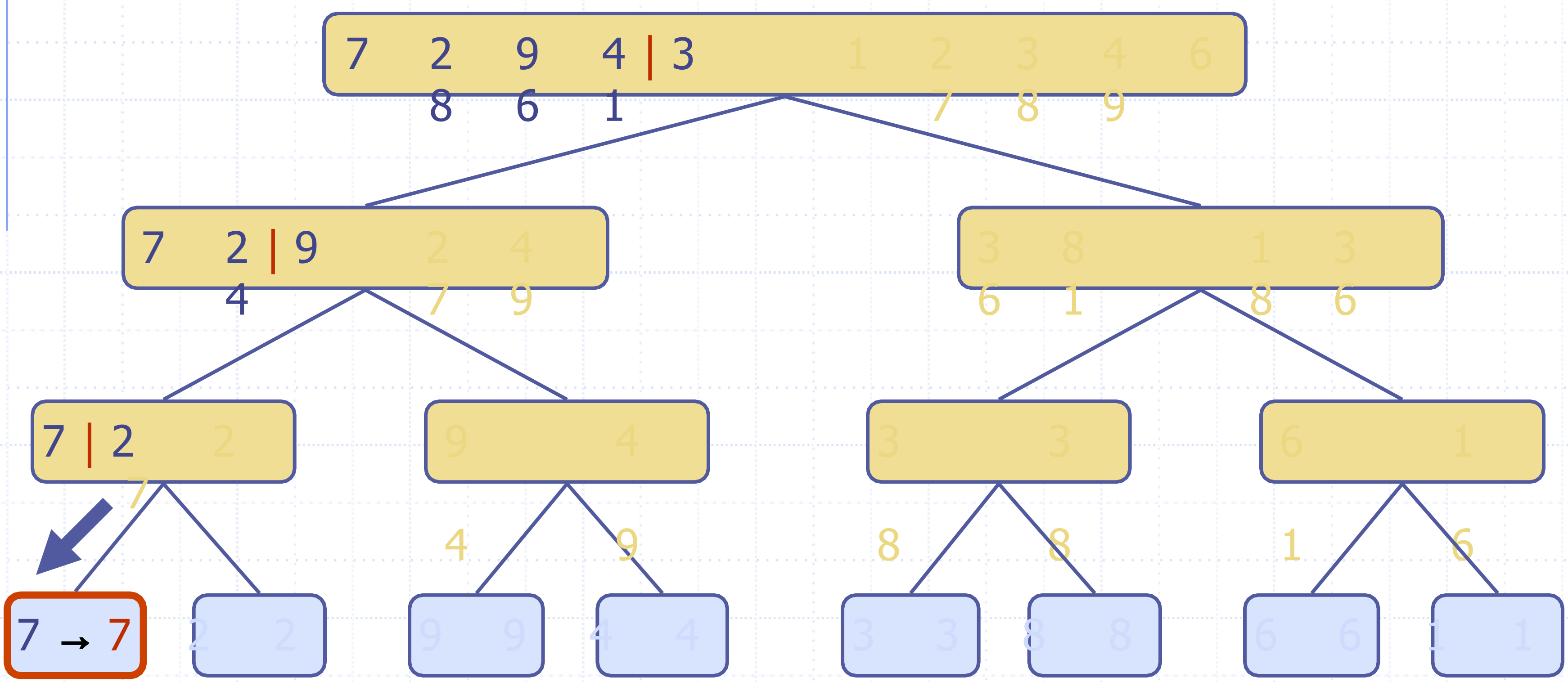
Execution Example (cont.)

- Recursive call, partition



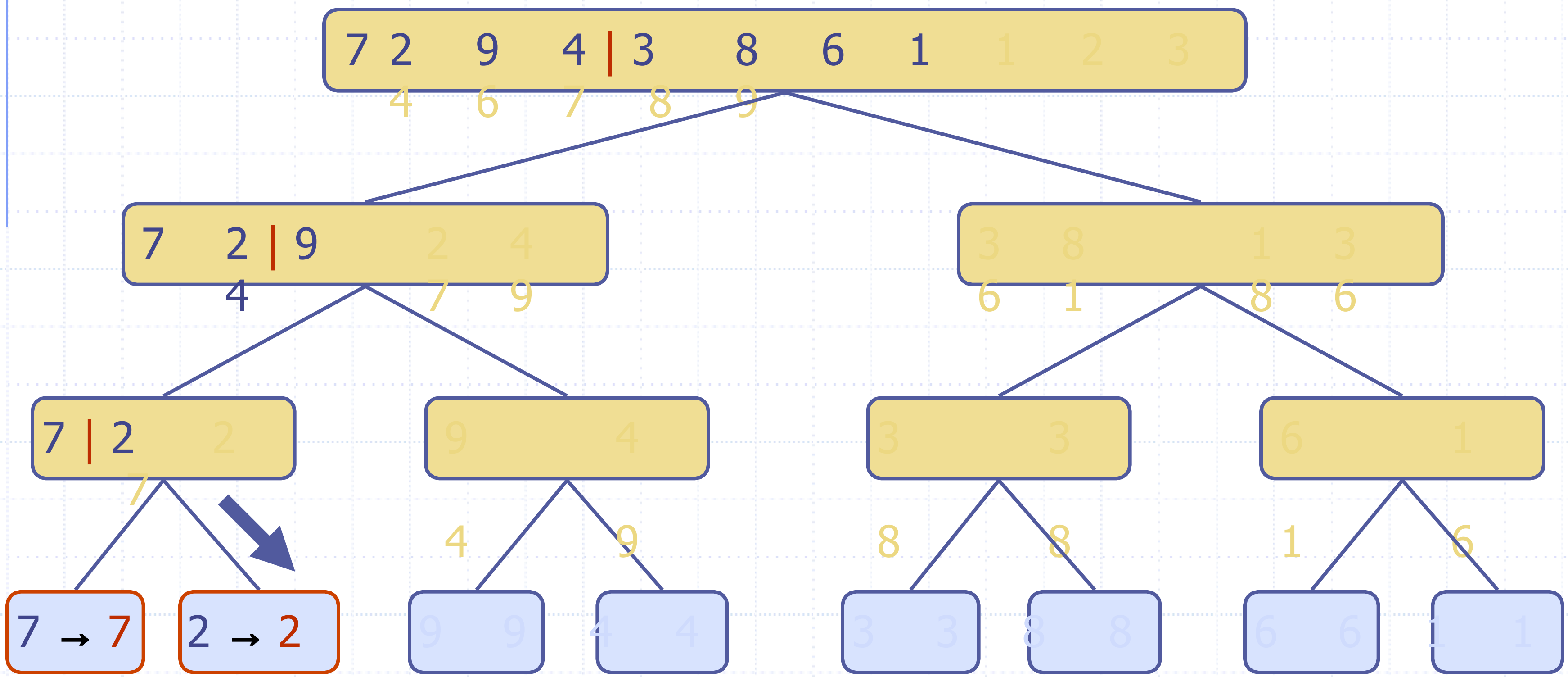
Execution Example (cont.)

- Recursive call, base case



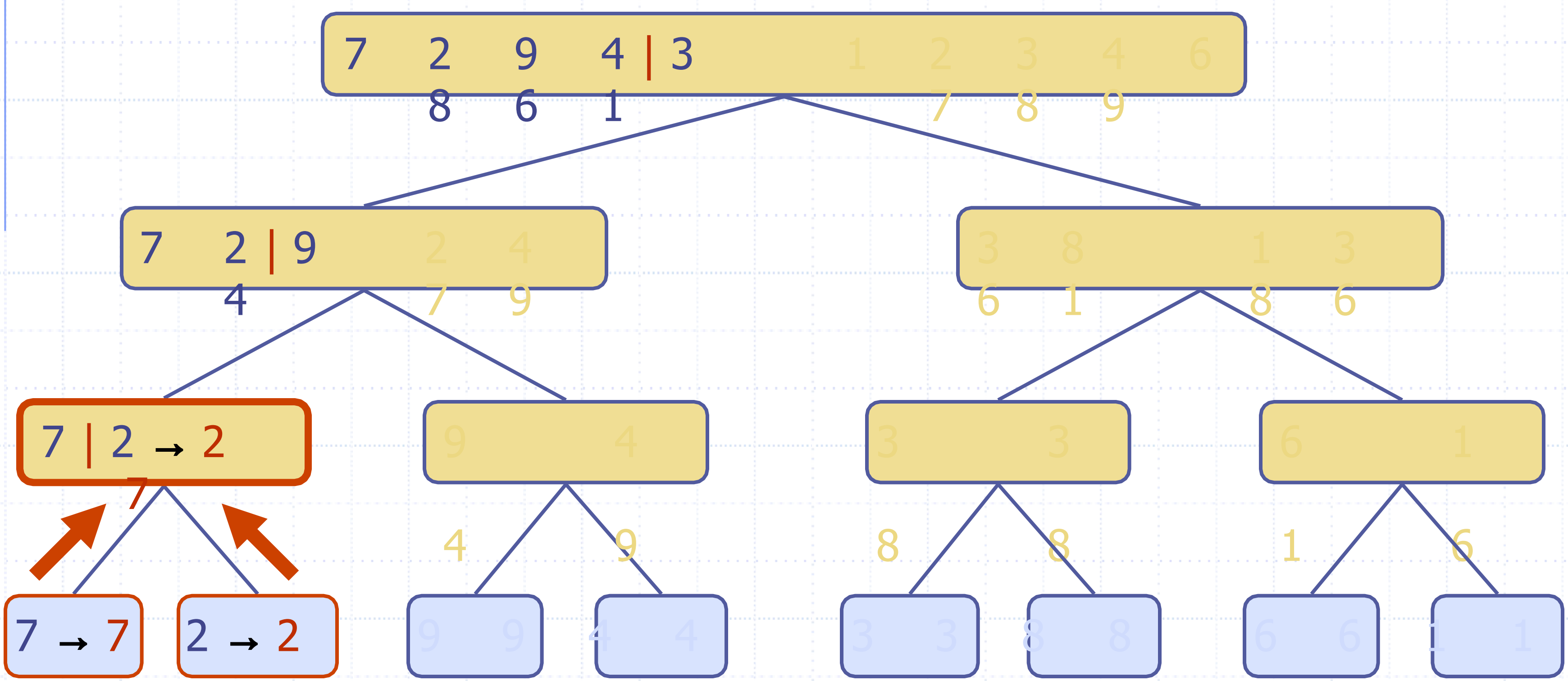
Execution Example (cont.)

- Recursive call, base case



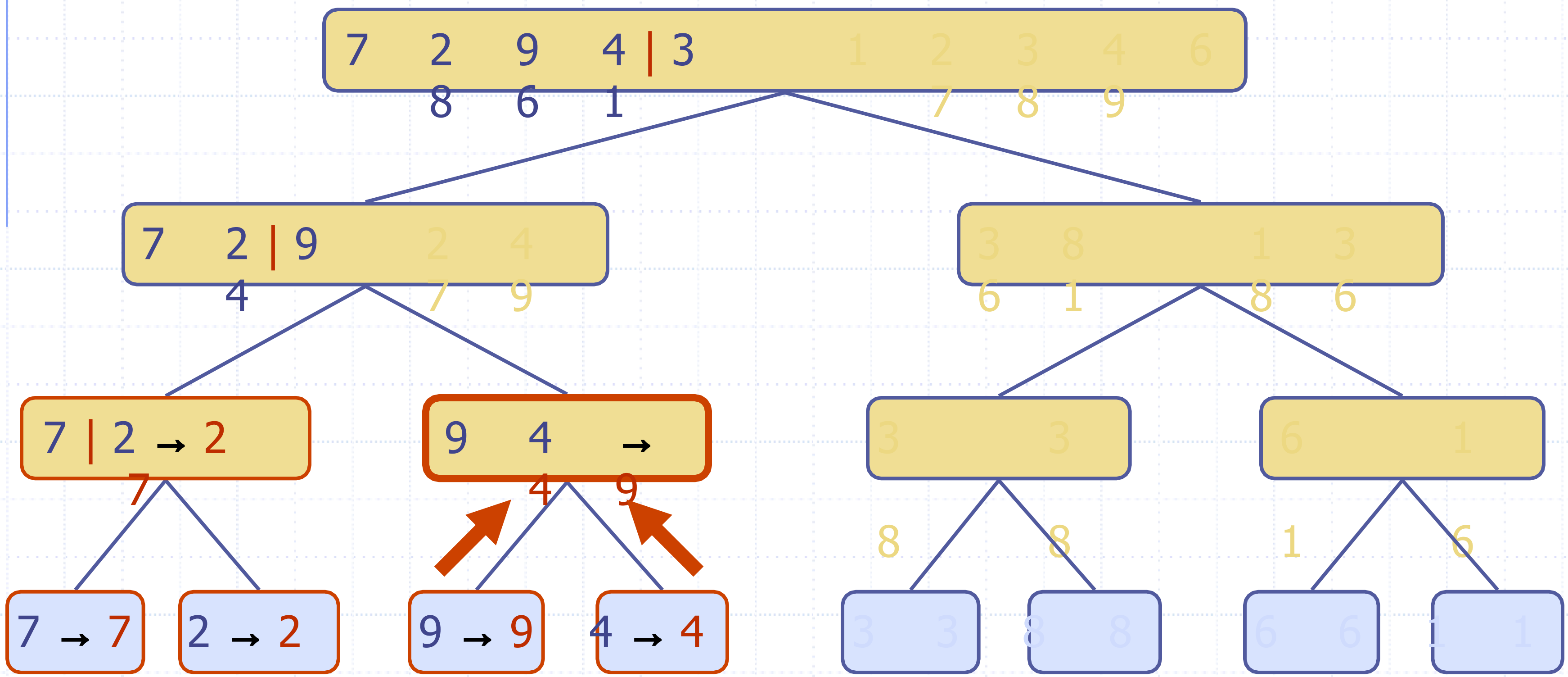
Execution Example (cont.)

- Merge



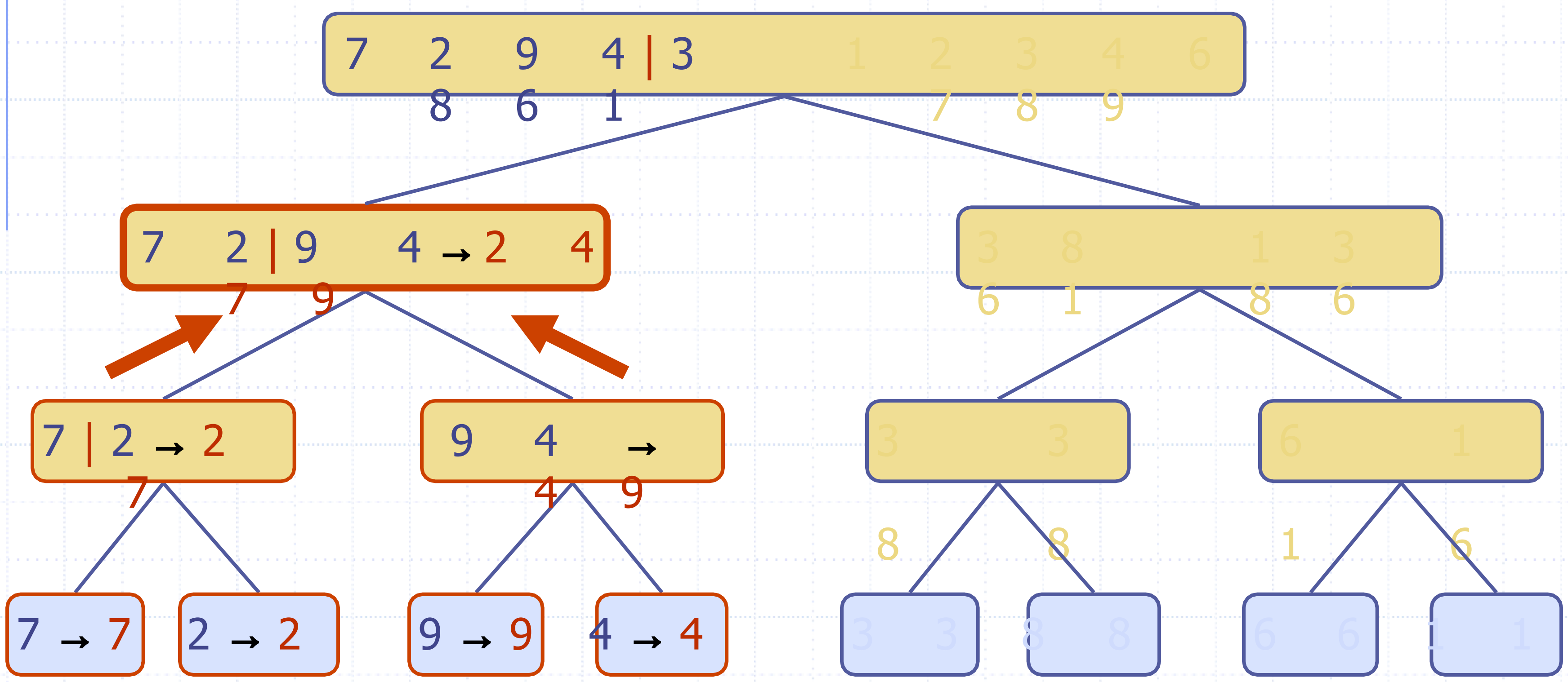
Execution Example (cont.)

- Recursive call, …, base case, merge



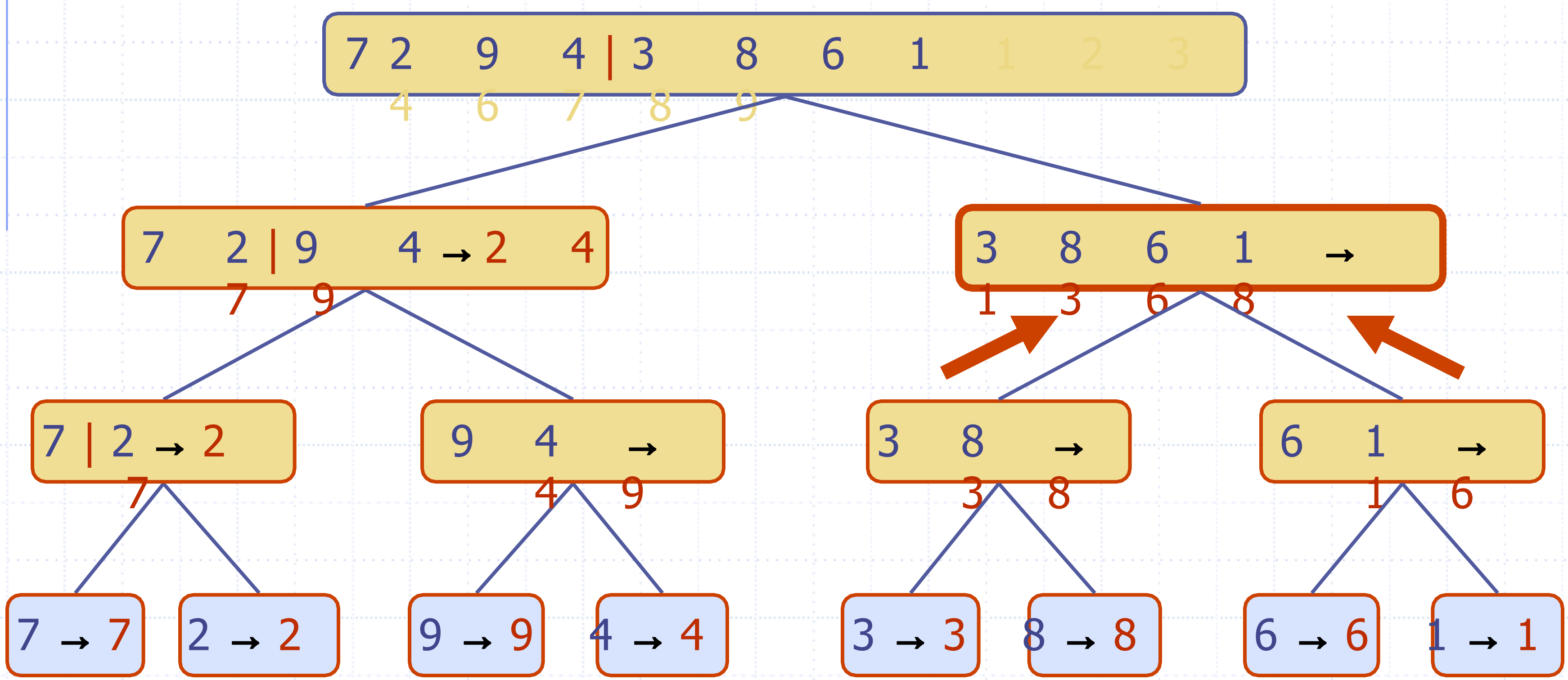
Execution Example (cont.)

- Merge



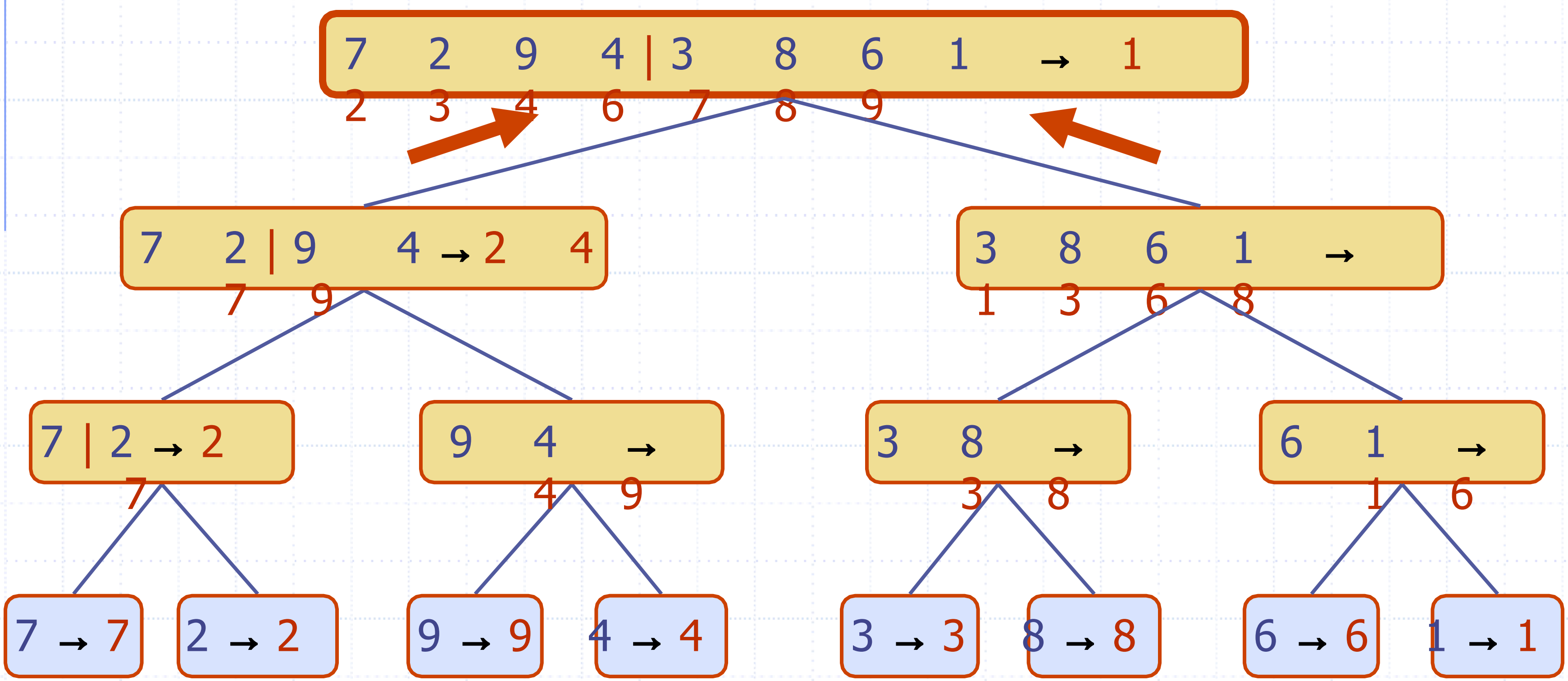
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

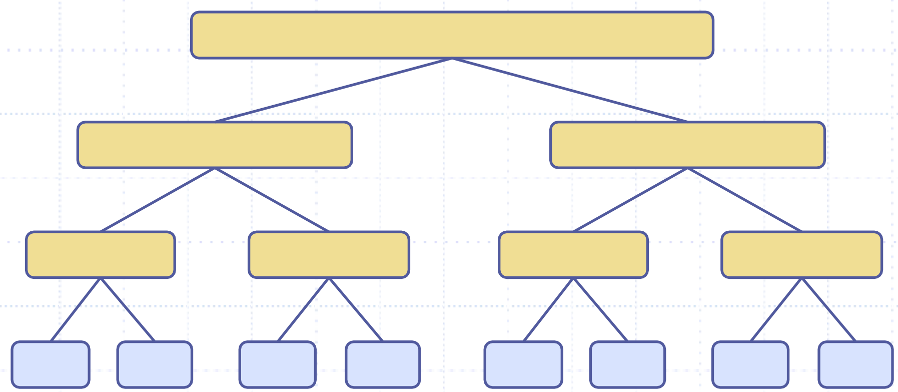
- Merge



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	2^i	$n/2^i$
…	…	…

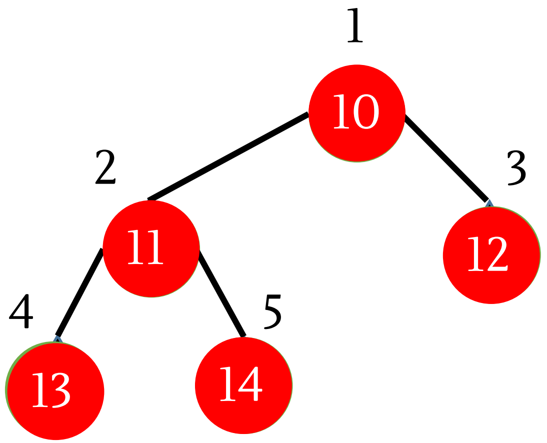


Heap Sort & Priority Queue

Properties of a Complete Binary Tree

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

Representation of Complete Binary Tree



5 vertices

X	10	11	12	13	14
0	1	2	3	4	5

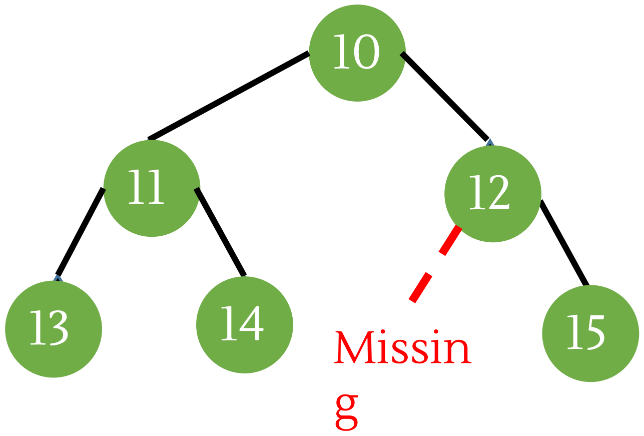
Left child of $\mathbf{i^{th}}$ node is: **2*i**

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

Parent of $\mathbf{i^{th}}$ node is: **floor(i/2)**

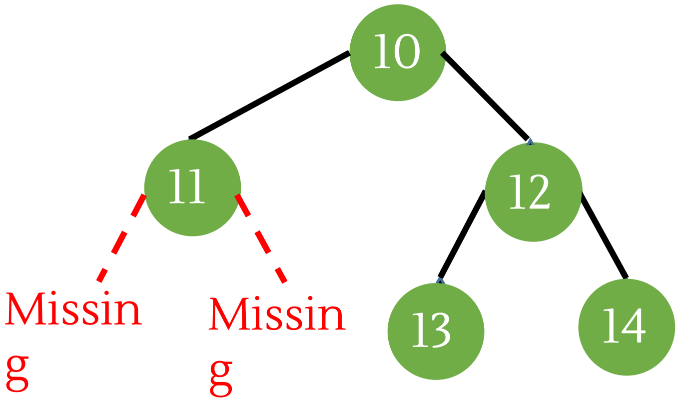
What is a Complete Binary Tree?

- Filled from left to right



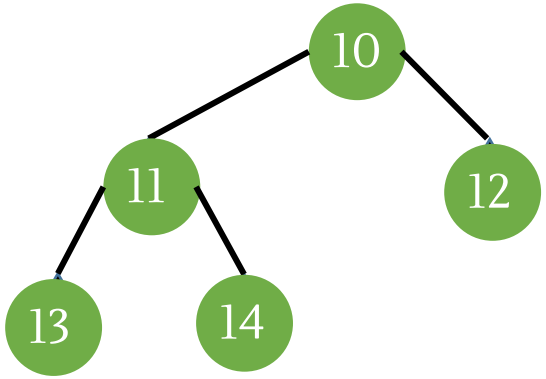
Not a Complete Binary Tree

10	11	12	13	14	--	15



Not a Complete Binary Tree

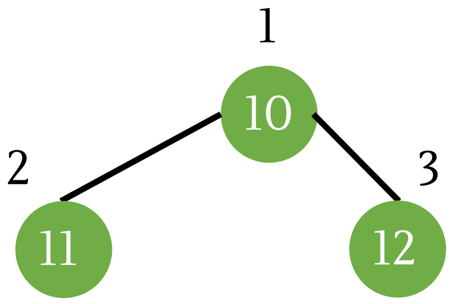
10	11	12	--	--	13	14

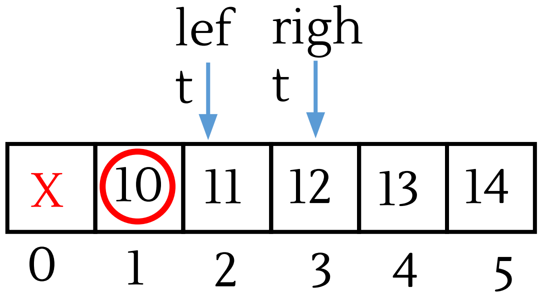


Complete Binary Tree

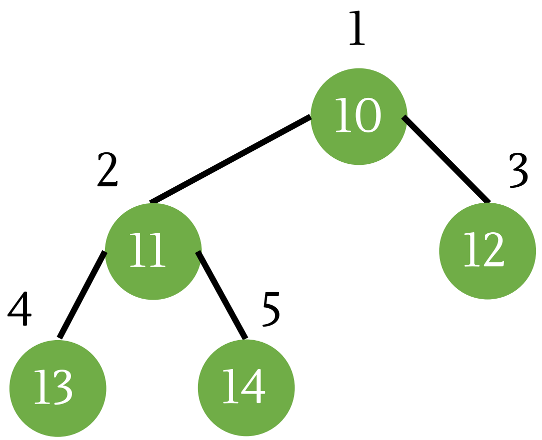
10	11	12	13	14

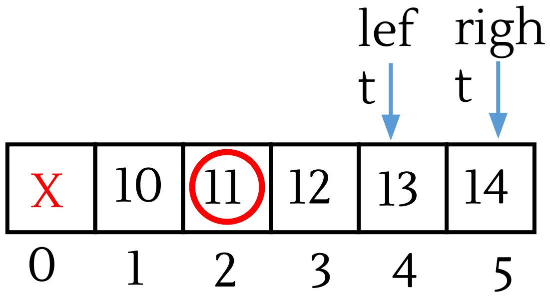
Representation of Complete Binary Tree From Array





Representation of Complete Binary Tree From Array

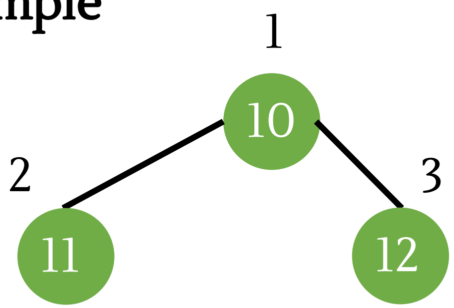


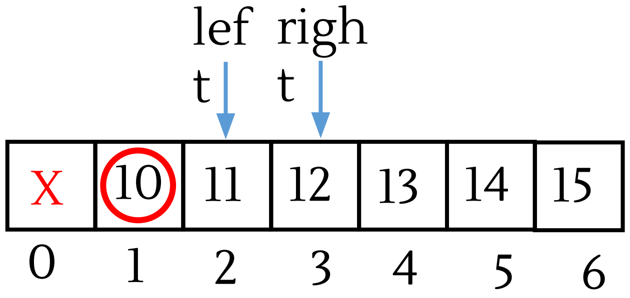


- ❑ How many vertices need to be explored to generate the complete binary tree?
  - ✔ 2
- ❑ Because only 2 vertices have child (internal vertices)
- ❑ Rest (5-2) = 3 vertices have no child (external vertices)
- ❑ Summary: If a complete binary tree having **n** vertices, it's **first floor(n/2)** vertices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

Representation of Complete Binary Tree From Array

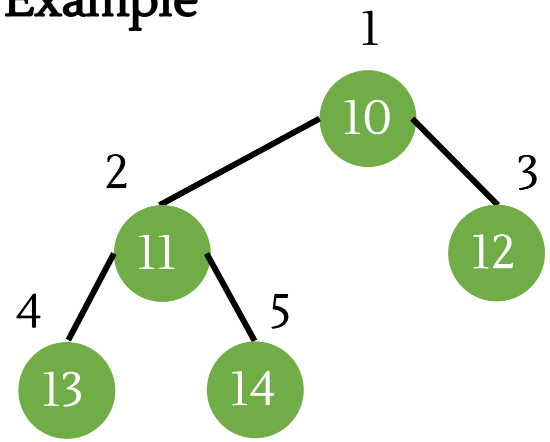
- Another Example

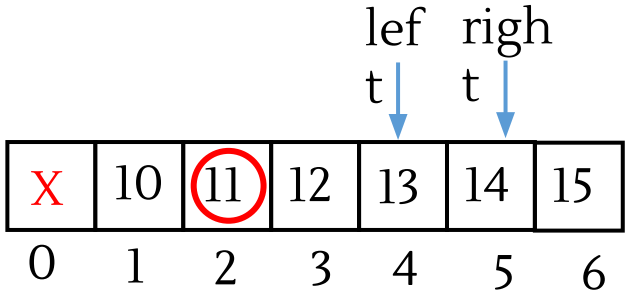




Representation of Complete Binary Tree From Array

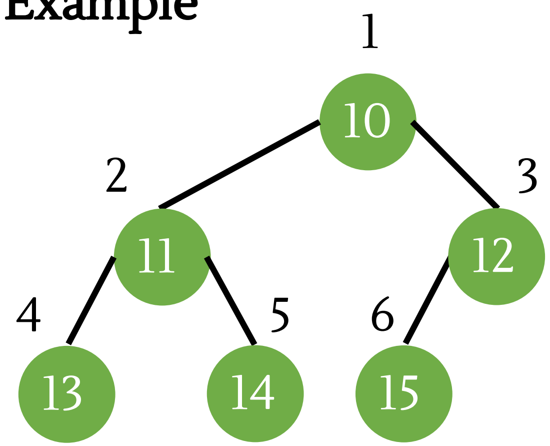
- Another Example

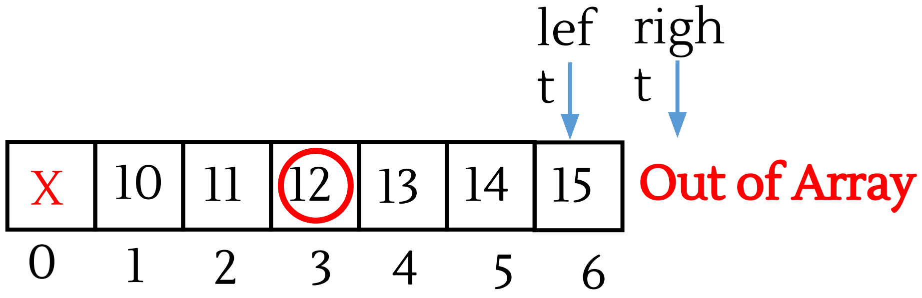




Representation of Complete Binary Tree From Array

- **Another Example**





- How many vertices need to be explored to generate the complete binary tree?
  - ✓ 3
- Maintains the same formula for **even** value of **n**
- First **floor(n/2)** vertices are ***internal*** and rest vertices are ***external****(leaf)*

Heap

Max Heap

- parent>=left child && parent>=right child
- All the sub trees maintain the same
- Is this a Max Heap?

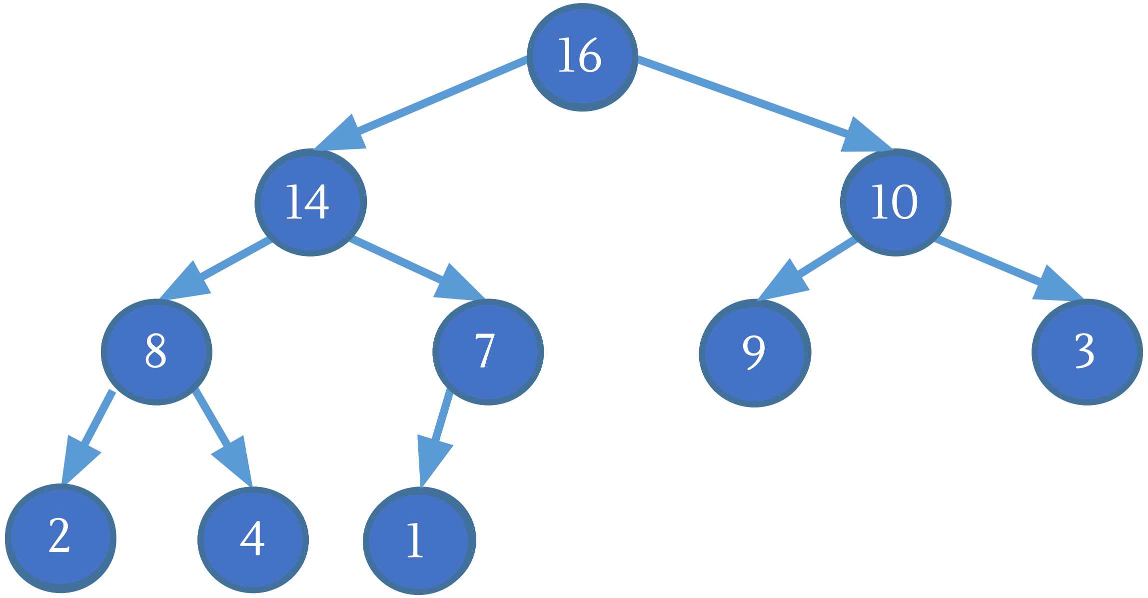
21	20	18	19	17	13	15

Min Heap

- parent<=left child && parent<=right child
- 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 Θ(log n).**
- The total number of comparisons required in heap is according to the height of the tree.
- Thus the **time complexity of basic operation** would also be **O(logn)**.

Basic Operations of a Max-Heap

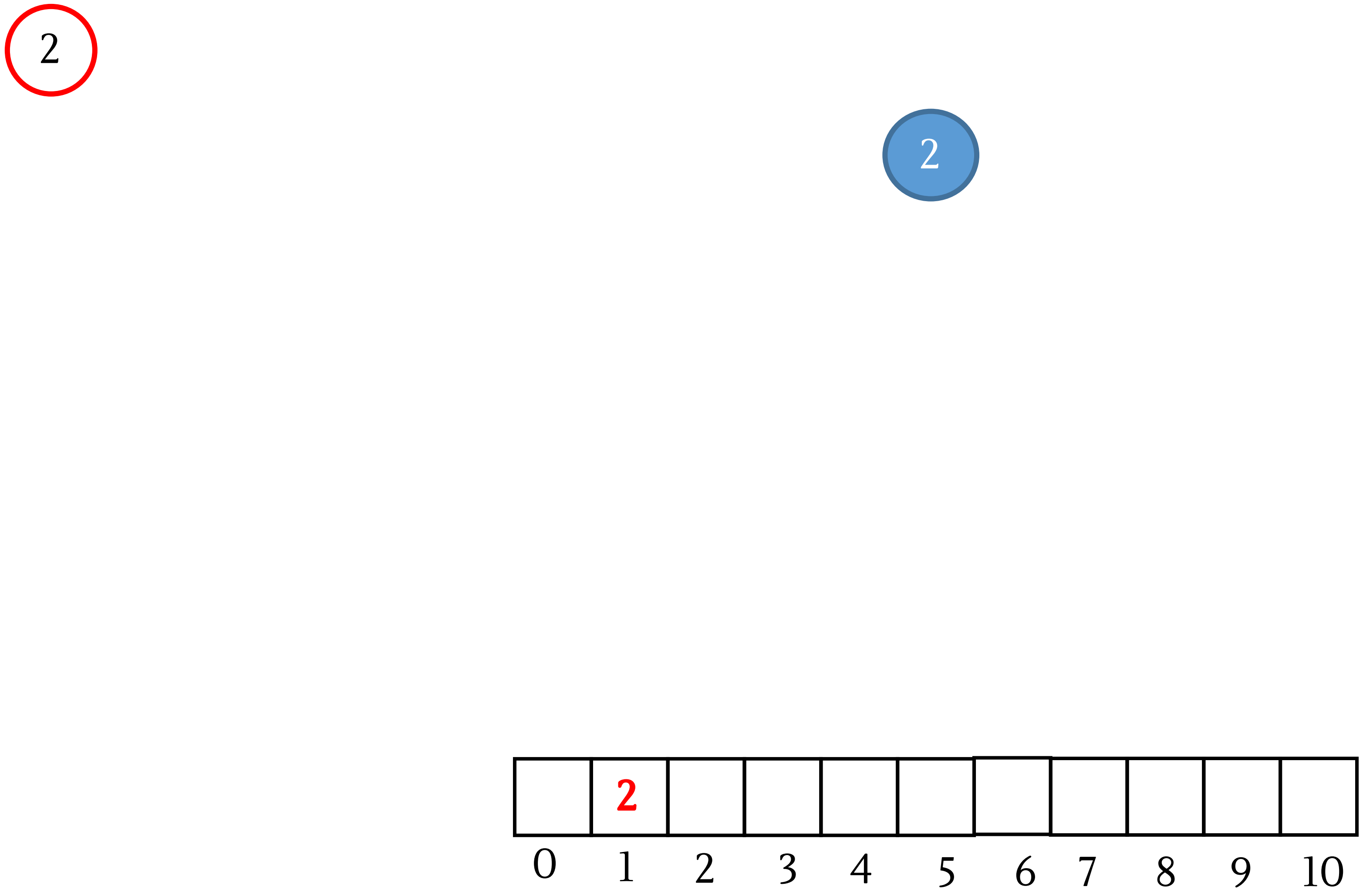
1. Max-Heap-Insert (Insertion)
2. Heap-Increase-Key (Increase the value of a current node)
3. Heap-Extract-Max (Remove the root element)
4. Heap-Maximum (Show the maximum element of the heap ⍰ root)
5. Heap Sort

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(nlogn)or O(n))

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

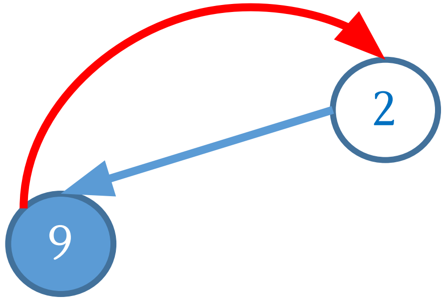
Insertion



Insertion

2 9

Compare with parent swap

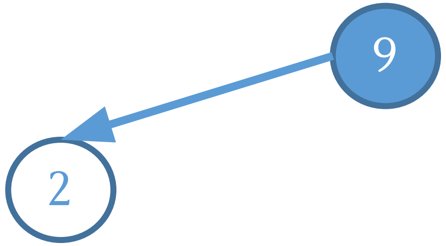


0	1	2	3	4	5	6	7	8	9	10
	2	9								

Insertion

2 9

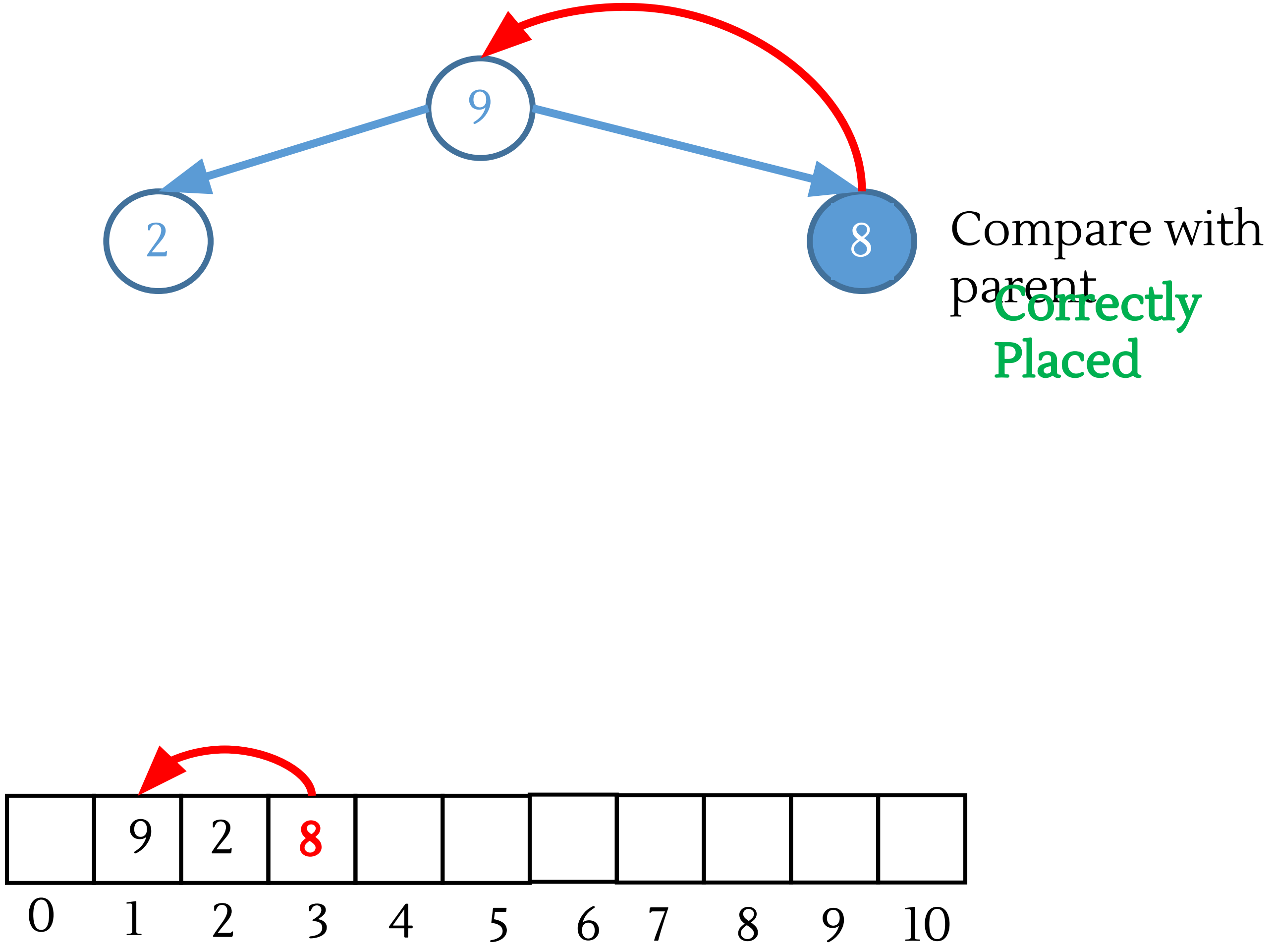
Compare with parent swa
p



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

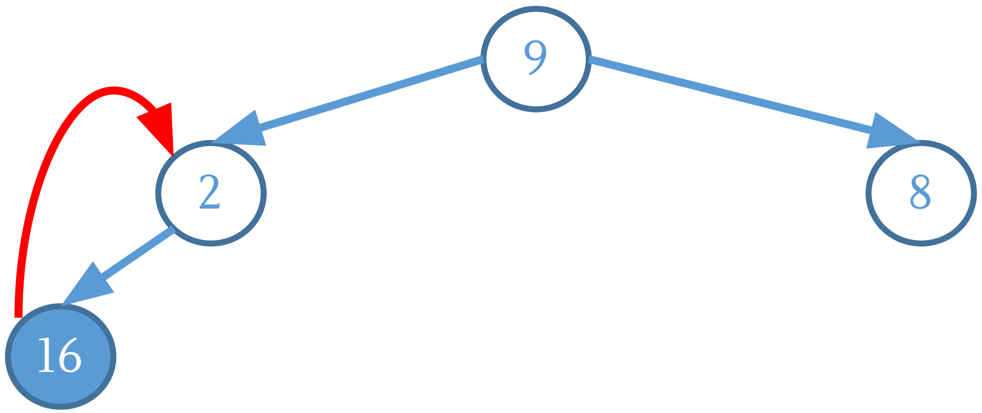
Insertion

2 9 8

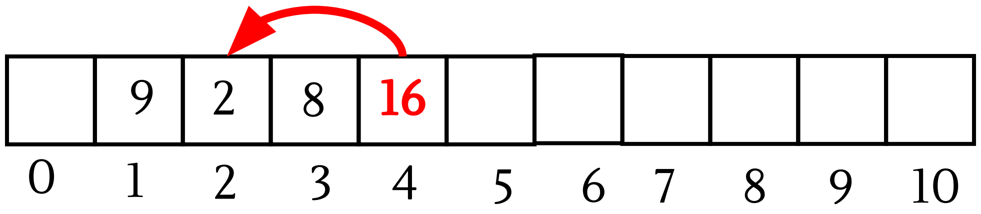


Insertion

2 9 8 16



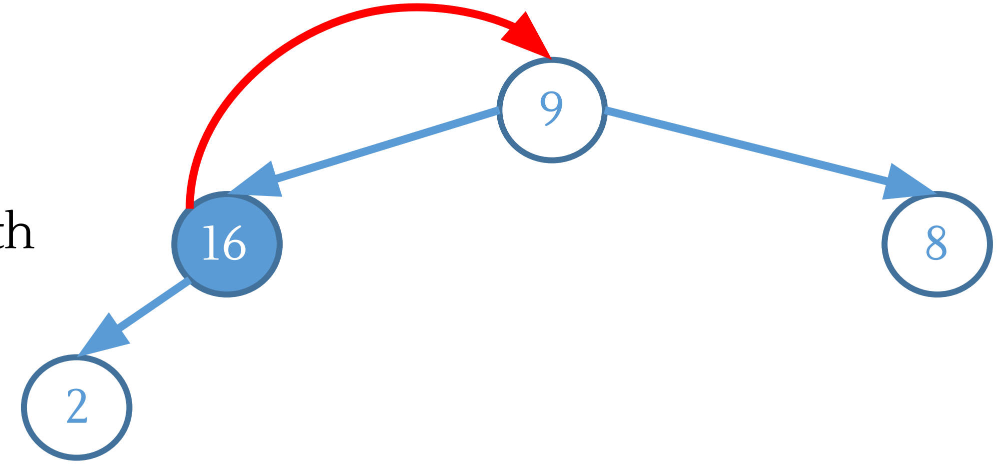
Compare with parent swa p

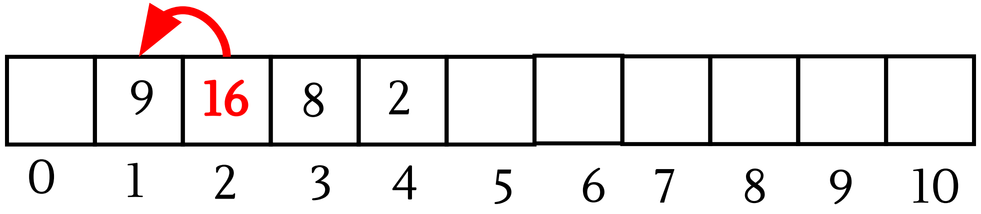


Insertion

2 9 8 16

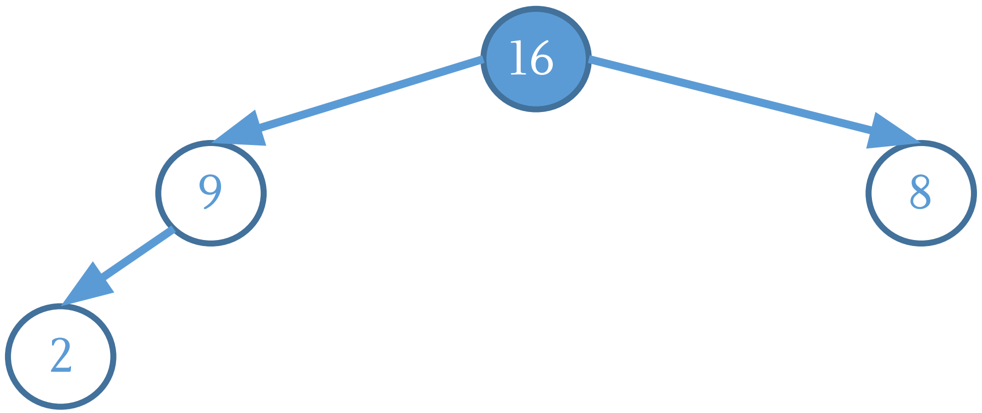
Compare with
parent swa
p





Insertion

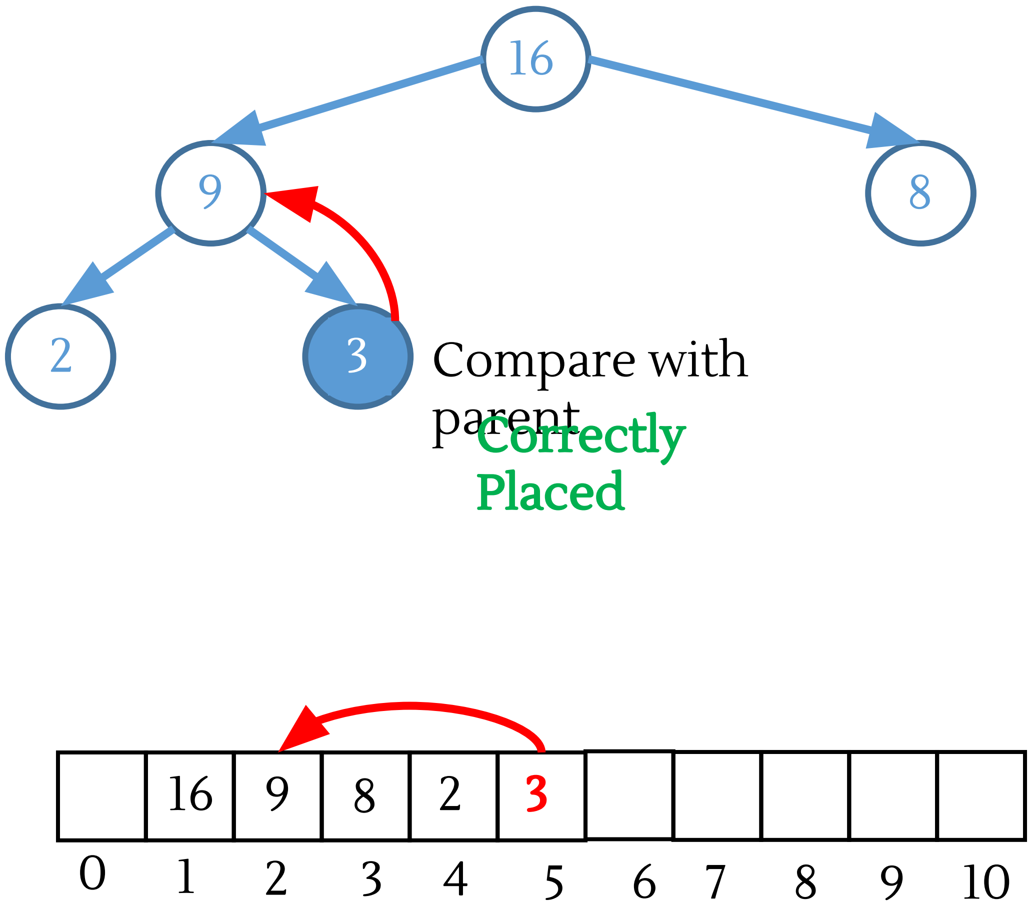
2 9 8 16



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

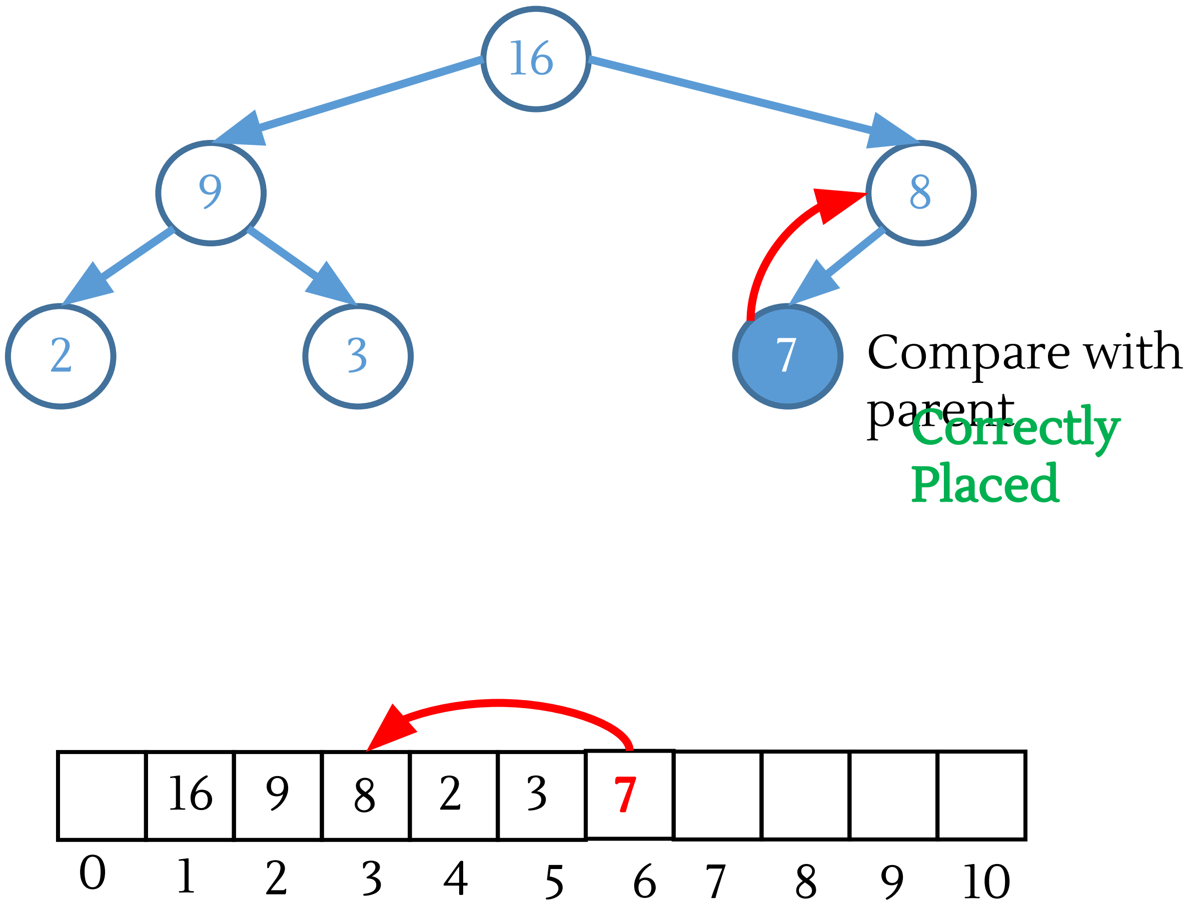
Insertion

2 9 8 16 3



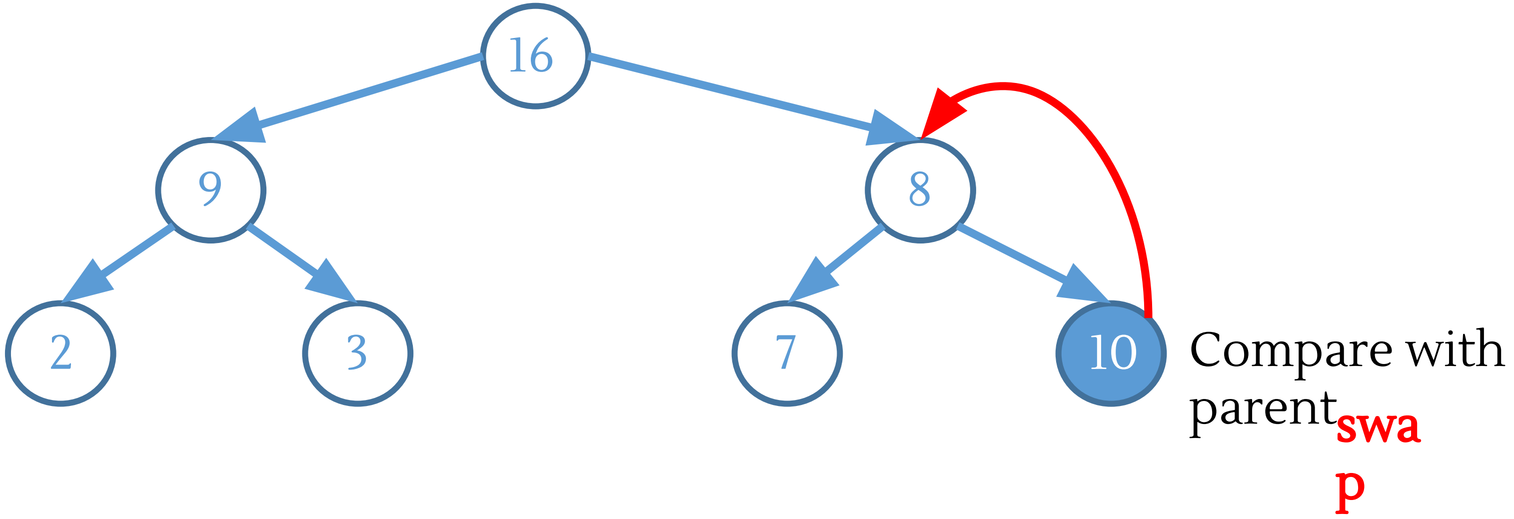
Insertion

2 9 8 16 3 7



Insertion

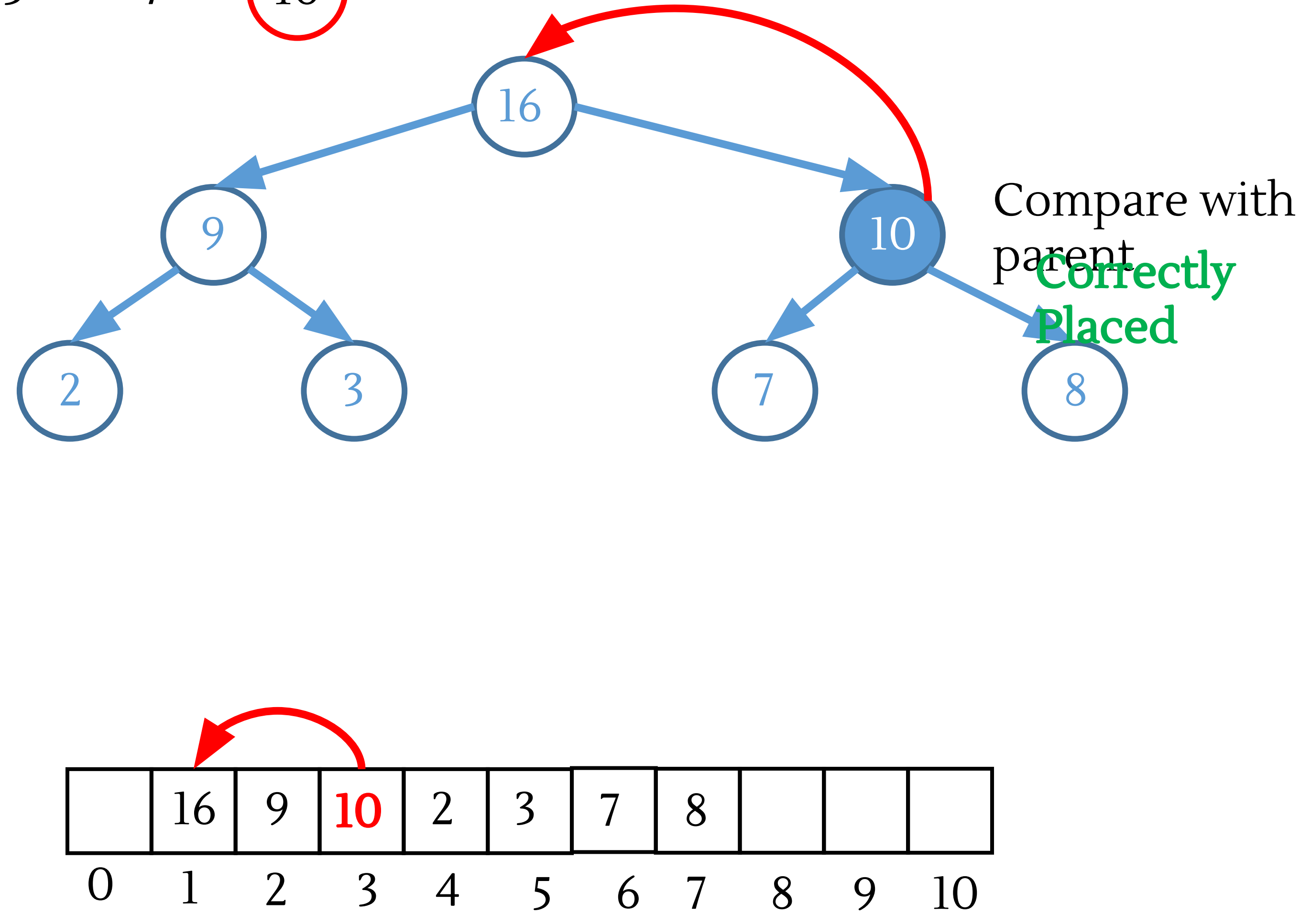
2 9 8 16 3 7 10



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

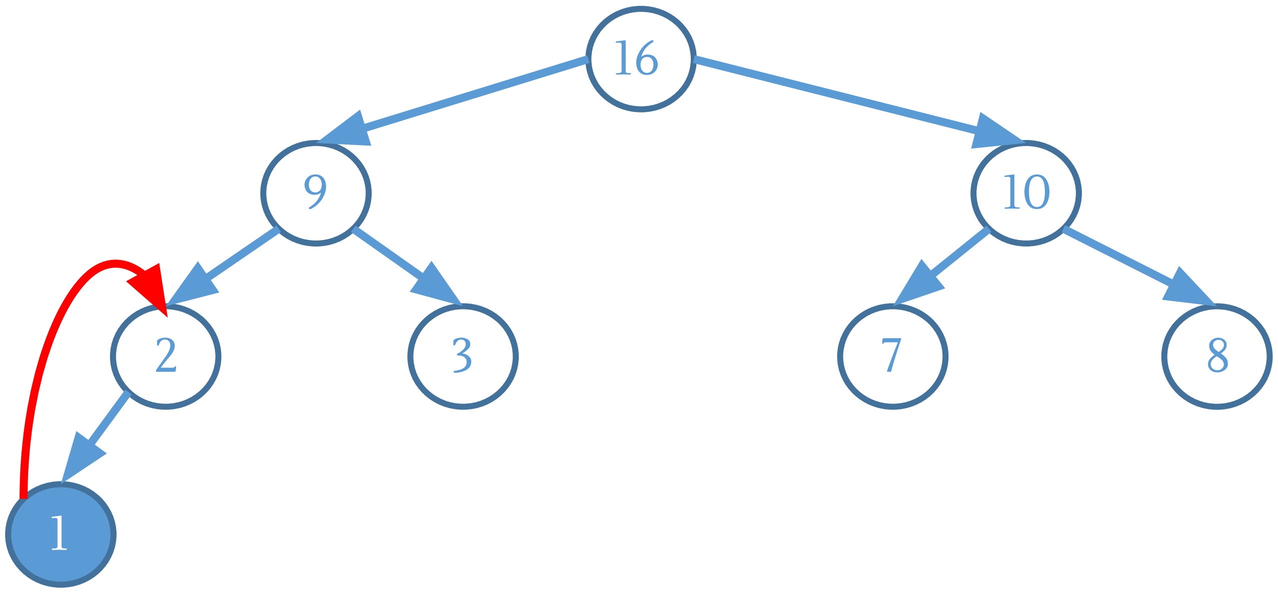
Insertion

2 9 8 16 3 7 10



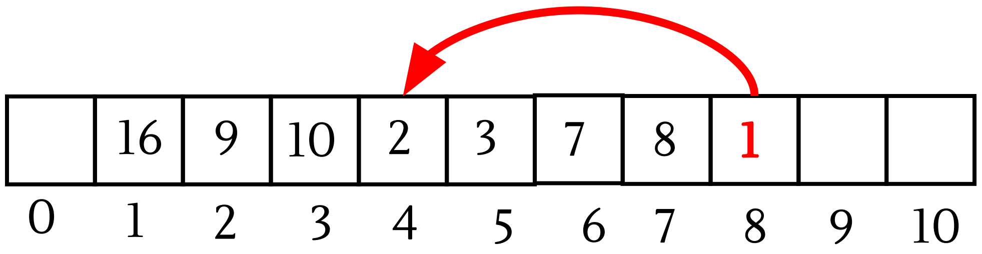
Insertion

2 9 8 16 3 7 10 1



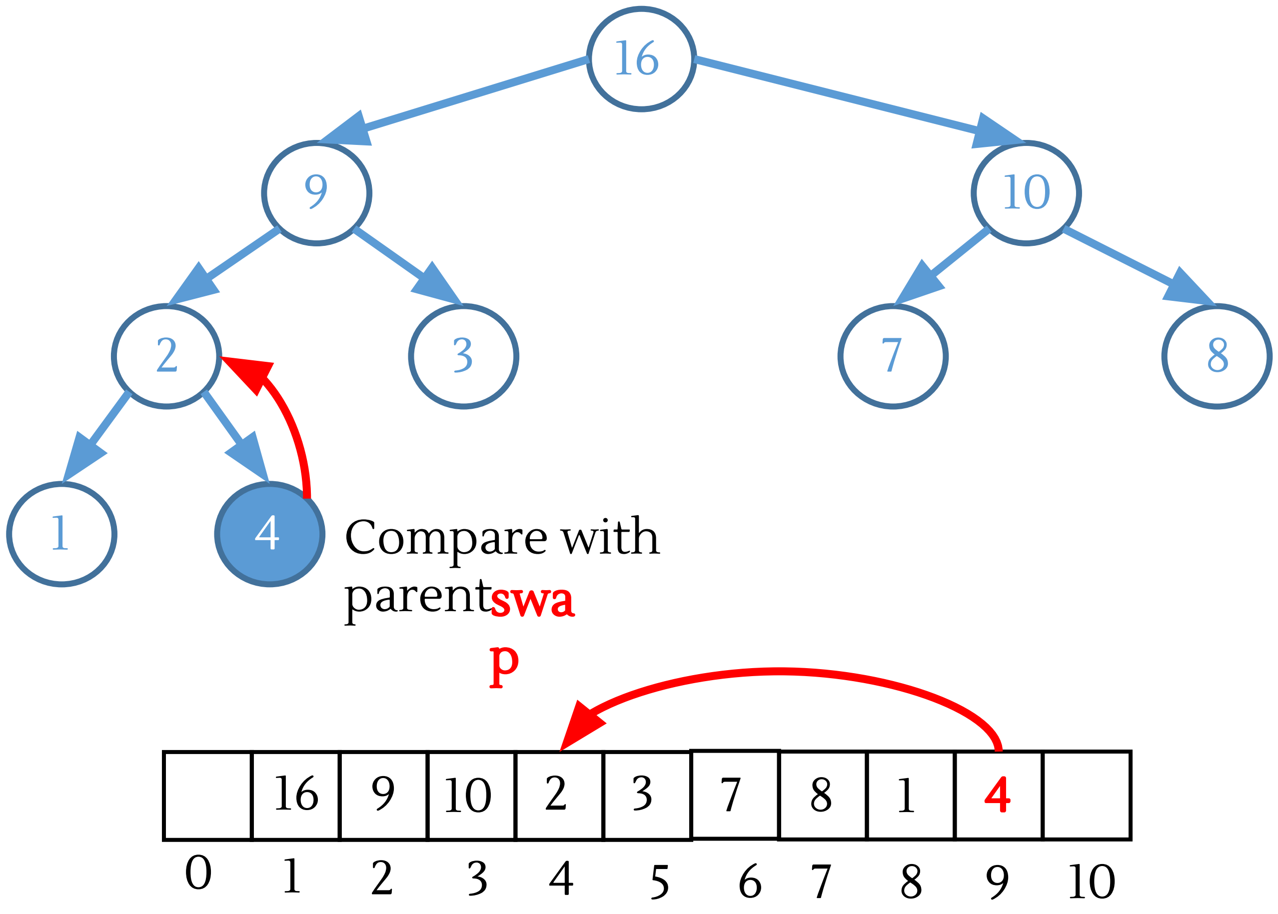
Compare with parent

Correctly Placed



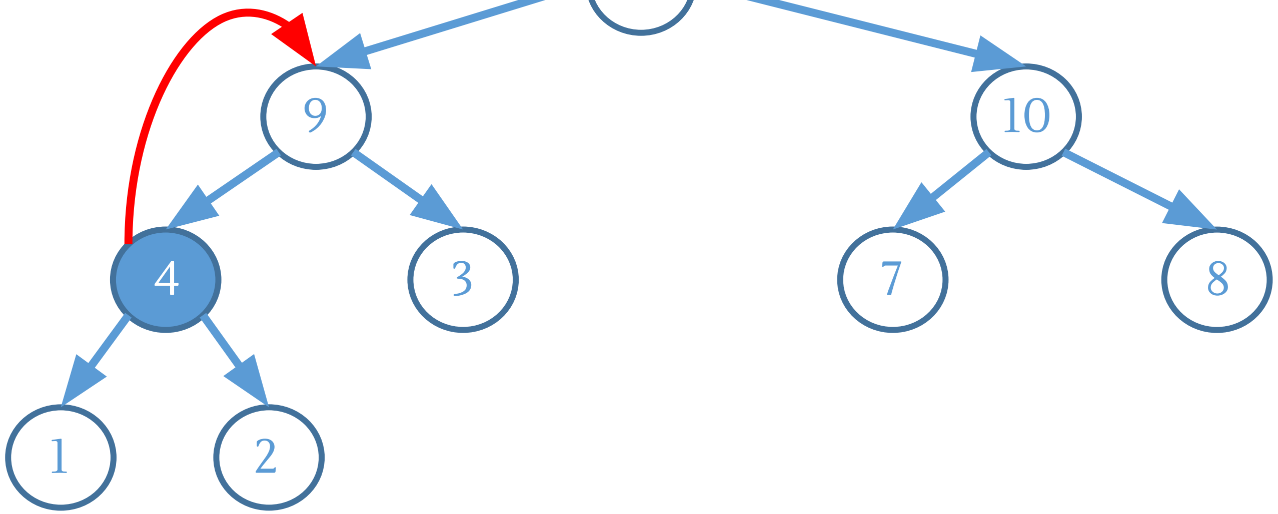
Insertion

2 9 8 16 3 7 10 1 4



Insertion

2 9 8 16 3 7 10 1 4



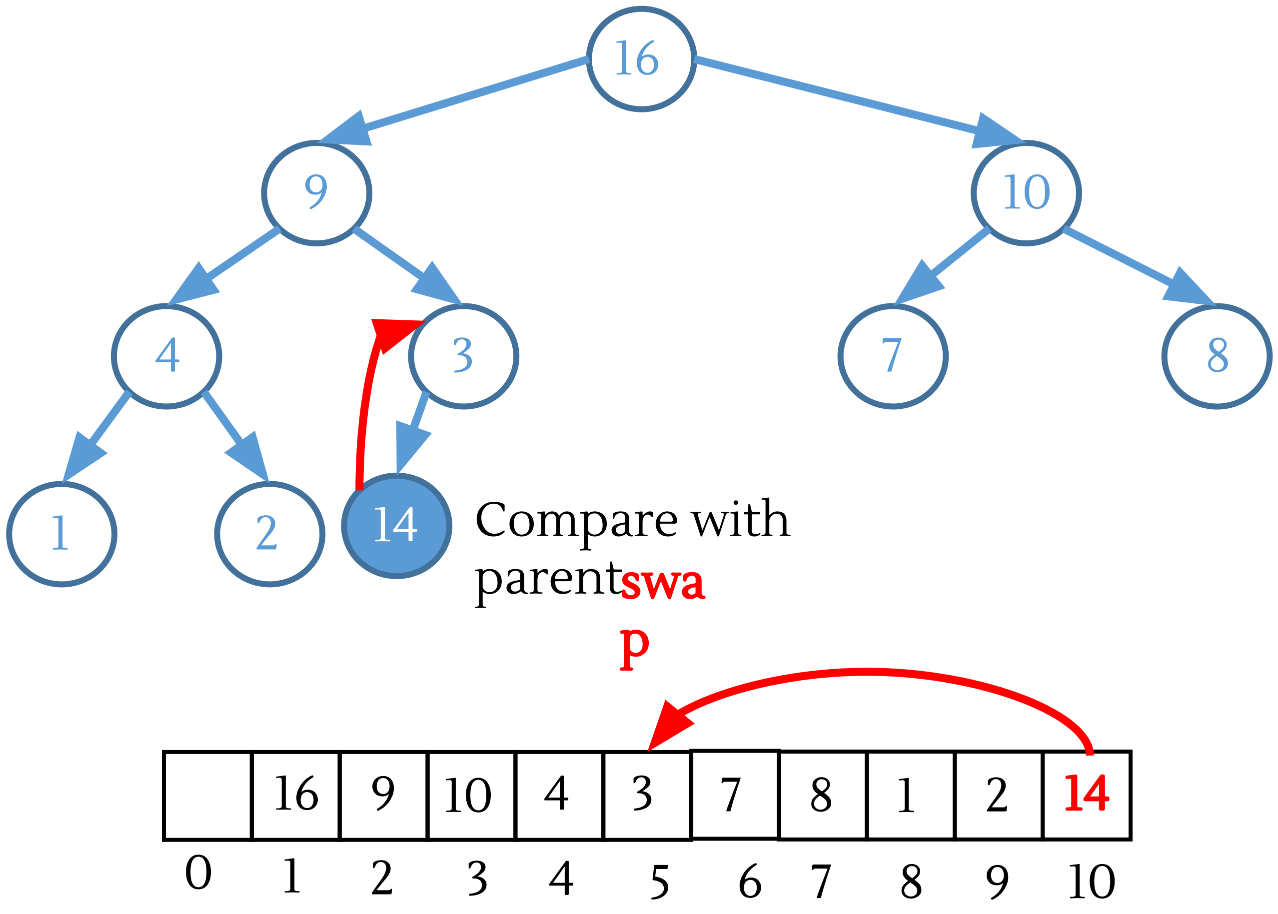
Compare with parent

Correctly Placed

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

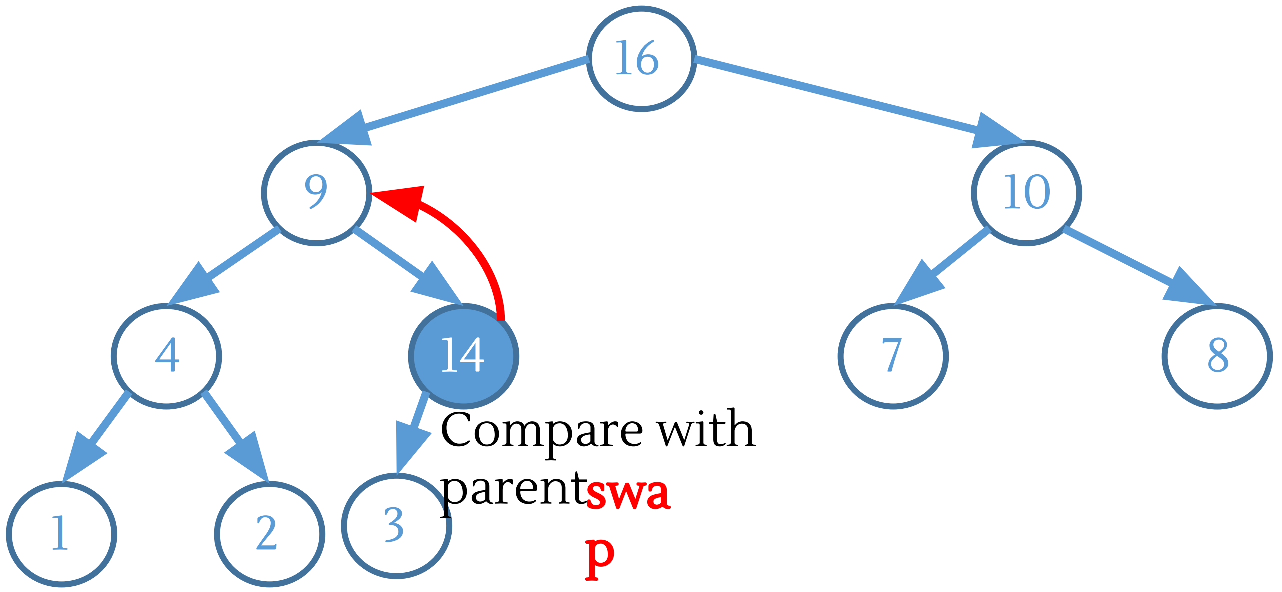
Insertion

2 9 8 16 3 7 10 1 4 14



Insertion

2 9 8 16 3 7 10 1 4 14



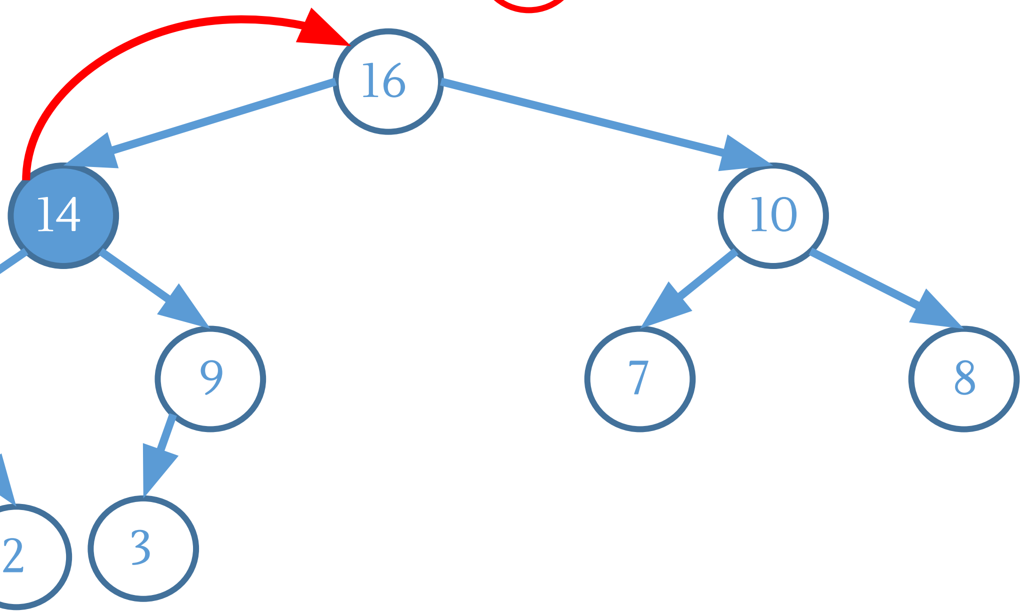
	16	9	10	4	14	7	8	1	2	3
0	1	2	3	4	5	6	7	8	9	10

Insertion

2 9 8 16 3 7 10 1 4 14

Compare with parent

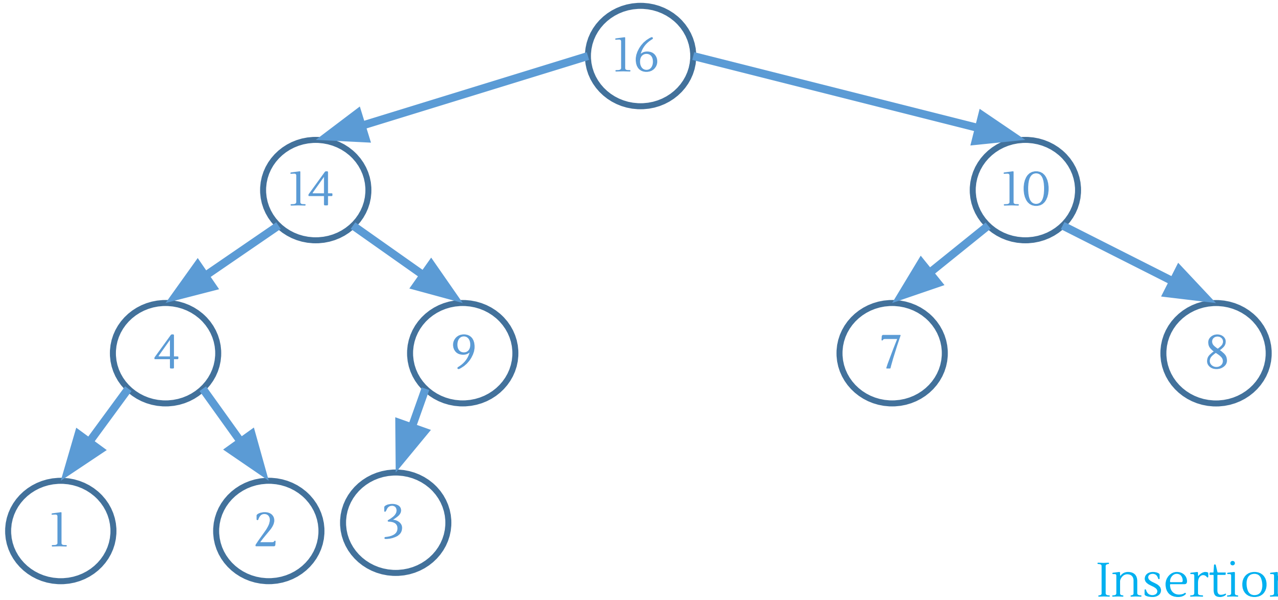
Correctly Placed



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

Insertion

2 9 8 16 3 7 10 1 4 14



Insertion is a ***bottom to top*** approach

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

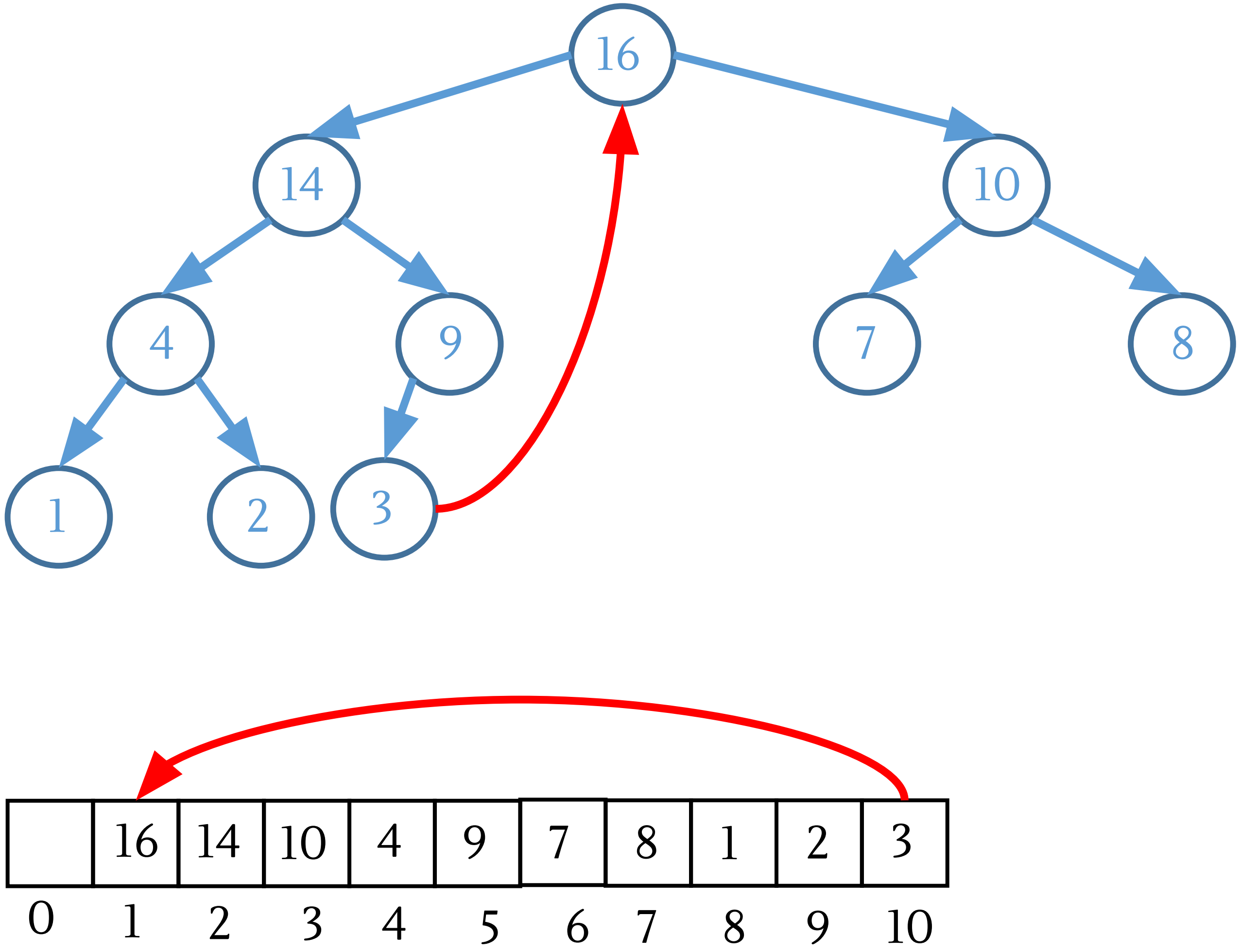
Effect of Insertion Order

- 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(nlogn) or O(n))

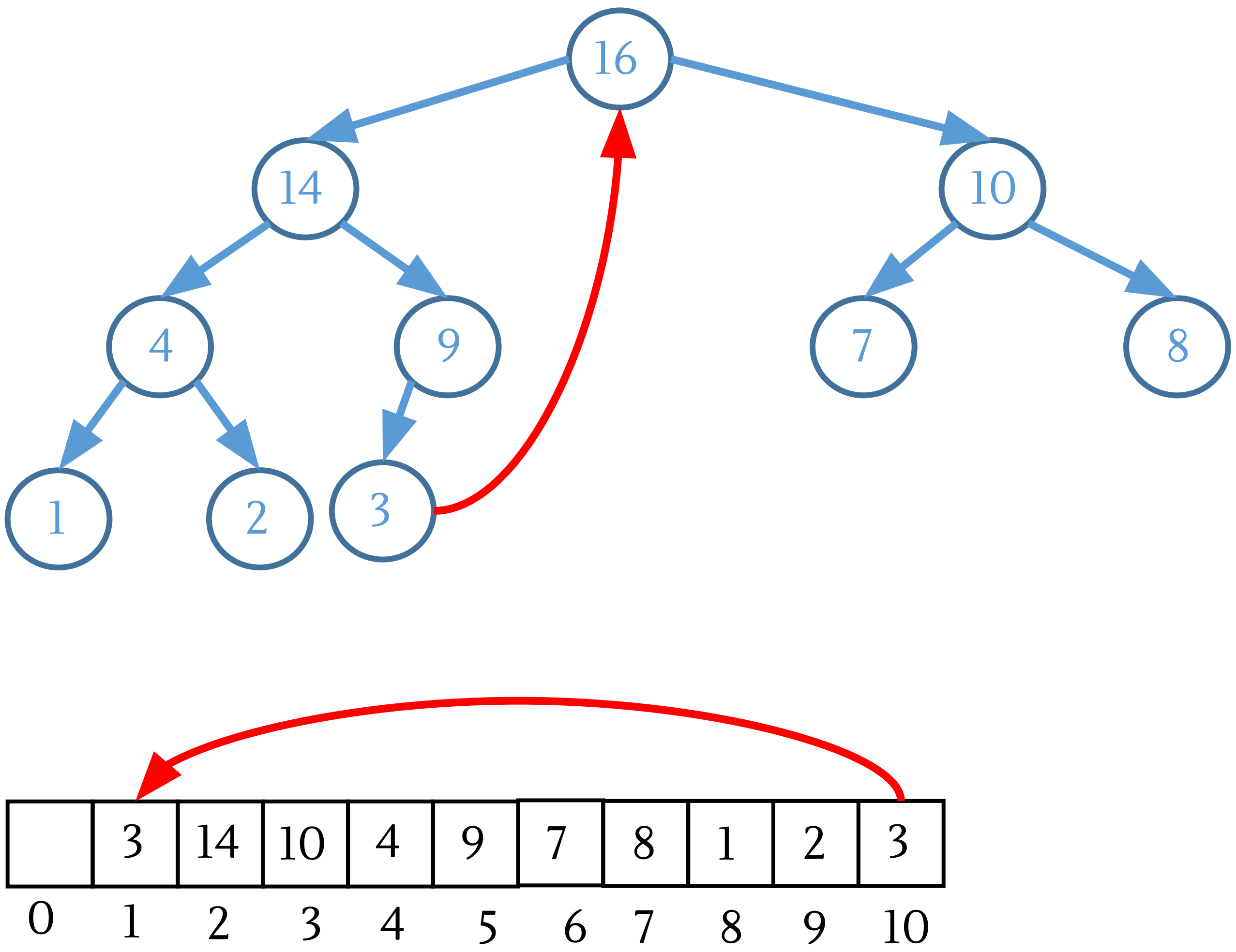
1. Swap the node to be deleted with the last leaf node.
2. Remove the last leaf node.
3. Heapify the whole tree (BUILD-MAX-HEAP) or BottomToTop Adjustment

Deletion



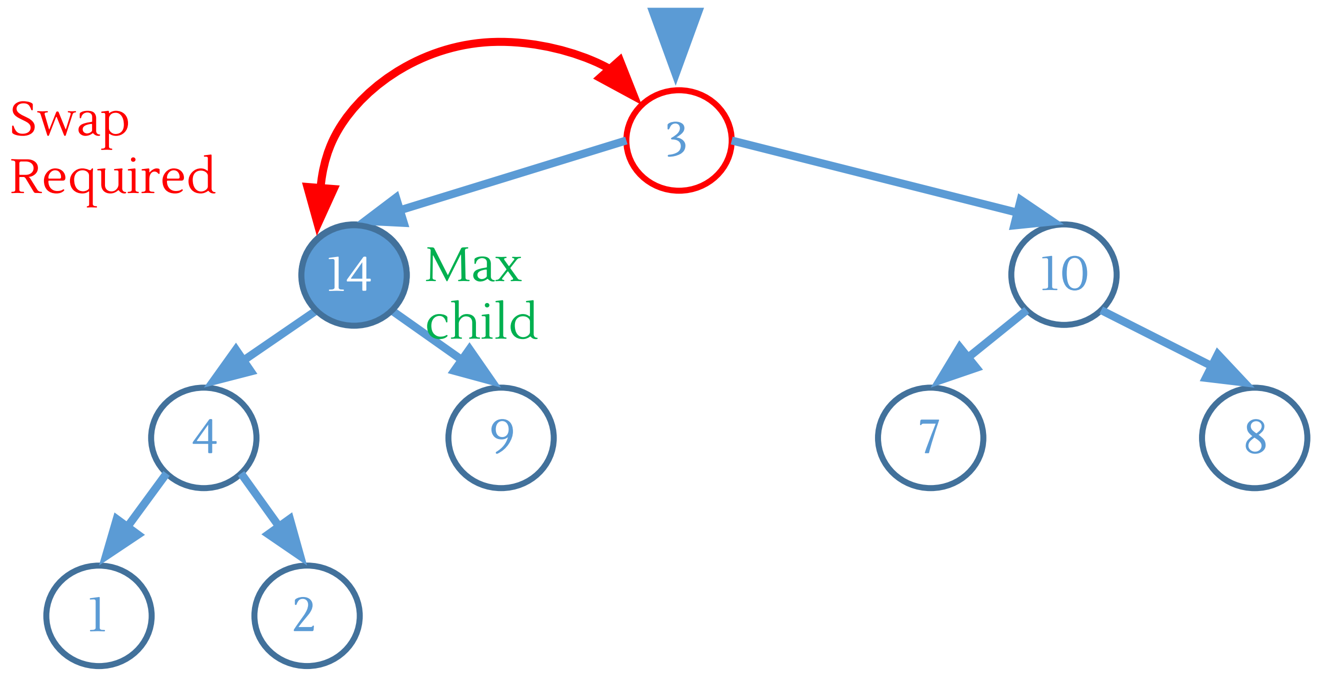
- ❑ Replace the first element by last element

Deletion



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

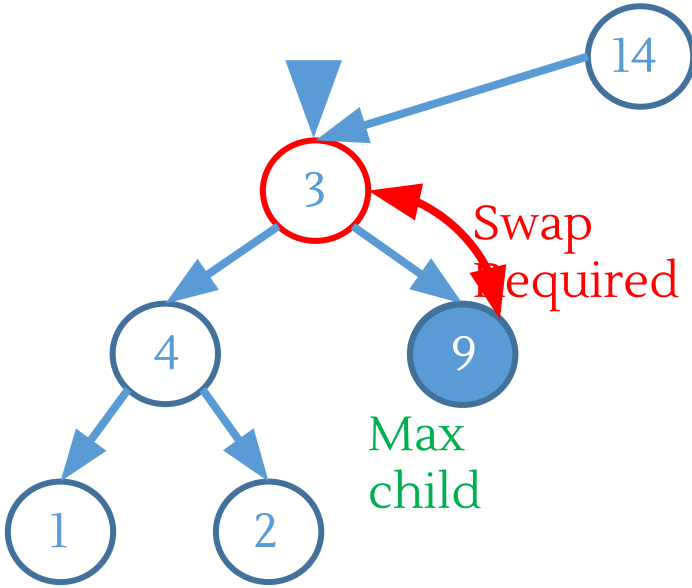
Deletion



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

- 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

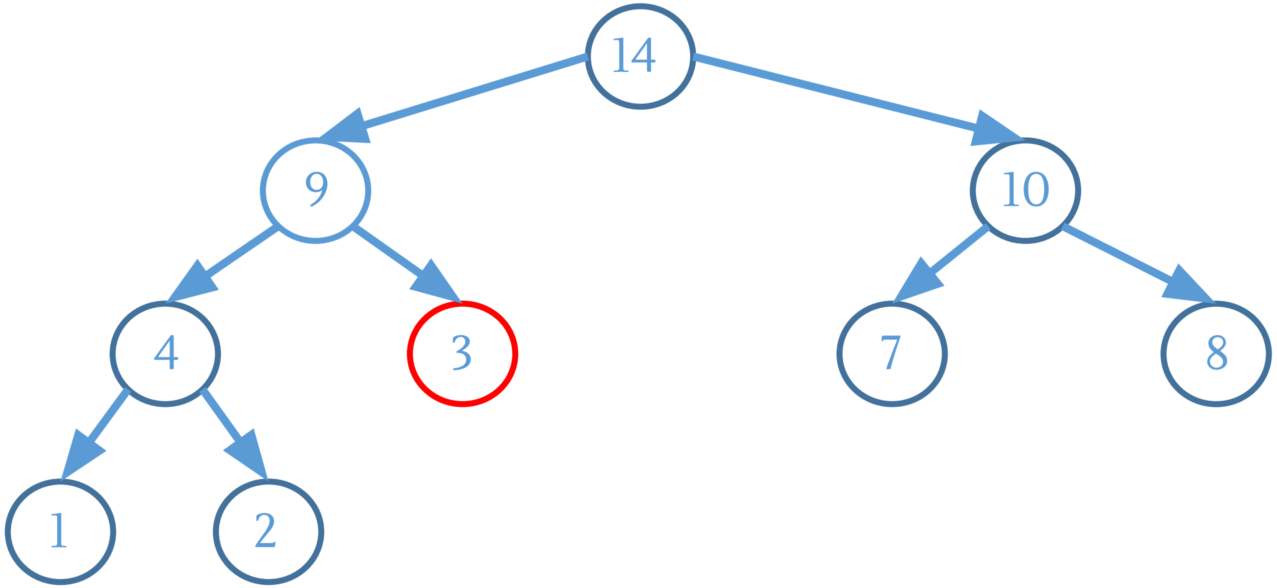


	14	3	10	4	9	7	8	1	2
0	1	2	3	4	5	6	7	8	9

- 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

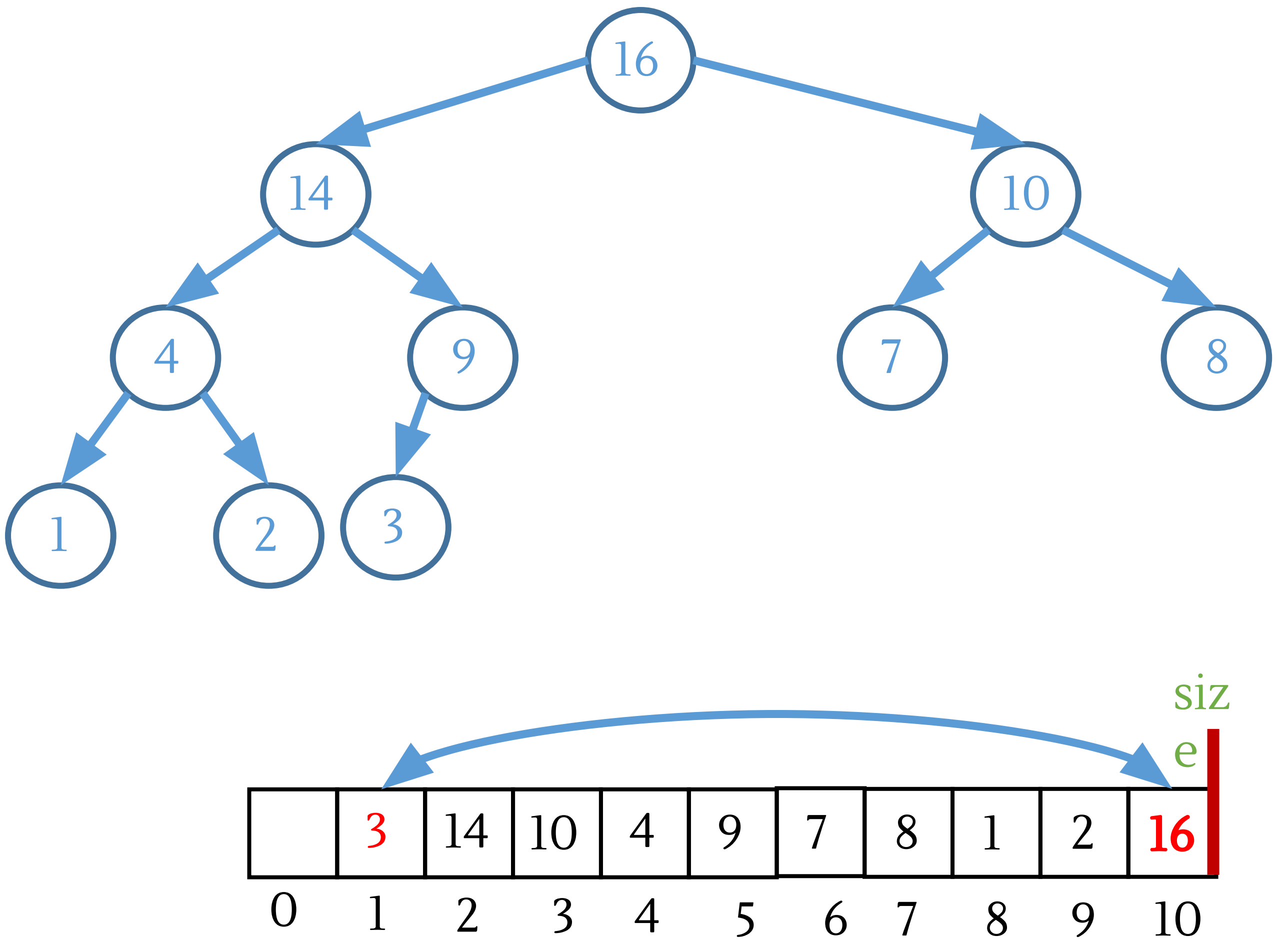
	14	9	10	4	3	7	8	1	2
0	1	2	3	4	5	6	7	8	9

Any Idea About Heap Sort?

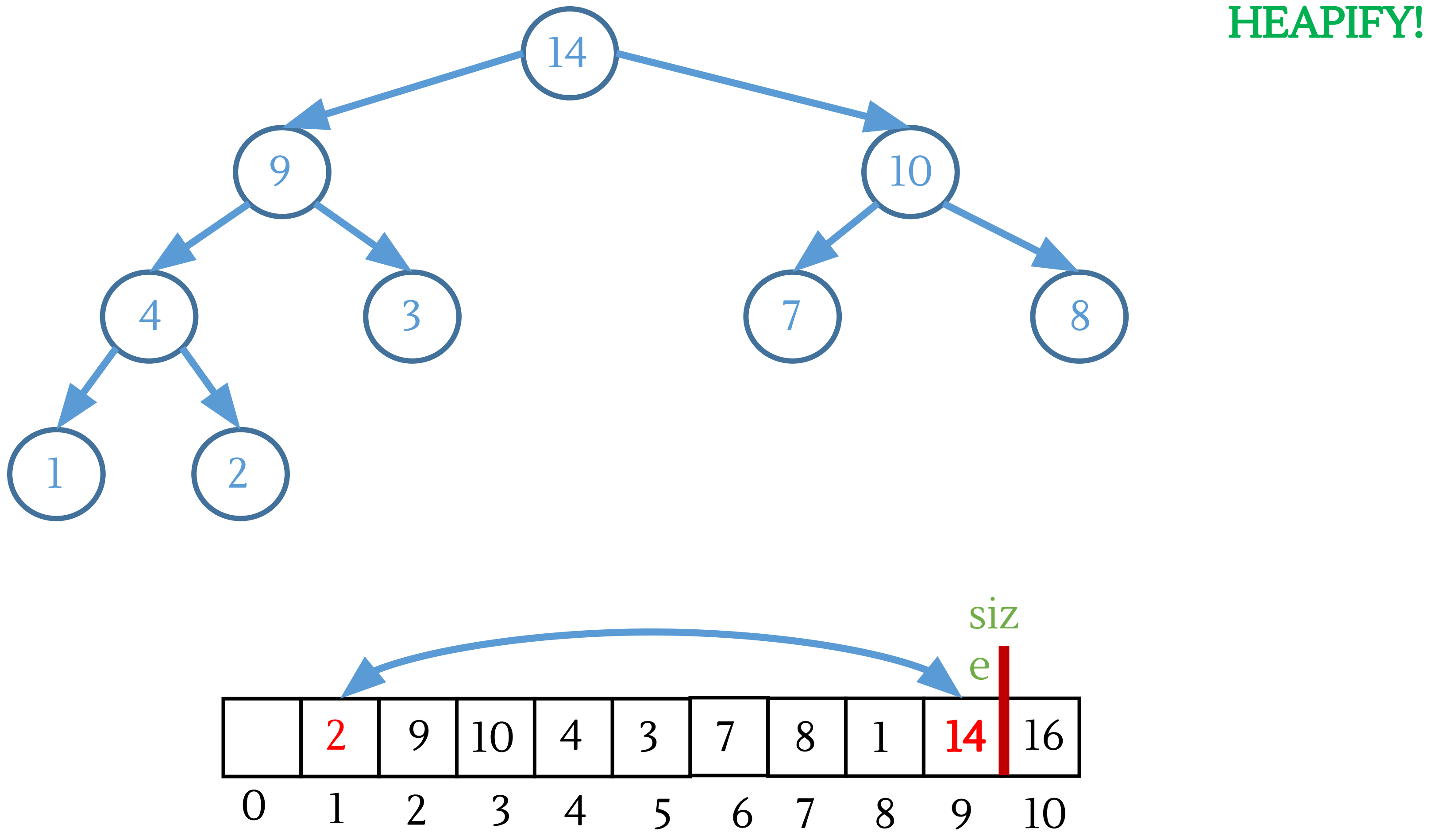
- Extracting all the roots sequentially produces the sorted sequence
- Storing the root element at the end of the array after resizing makes the array sorted

Heap Sort

HEAPIFY!

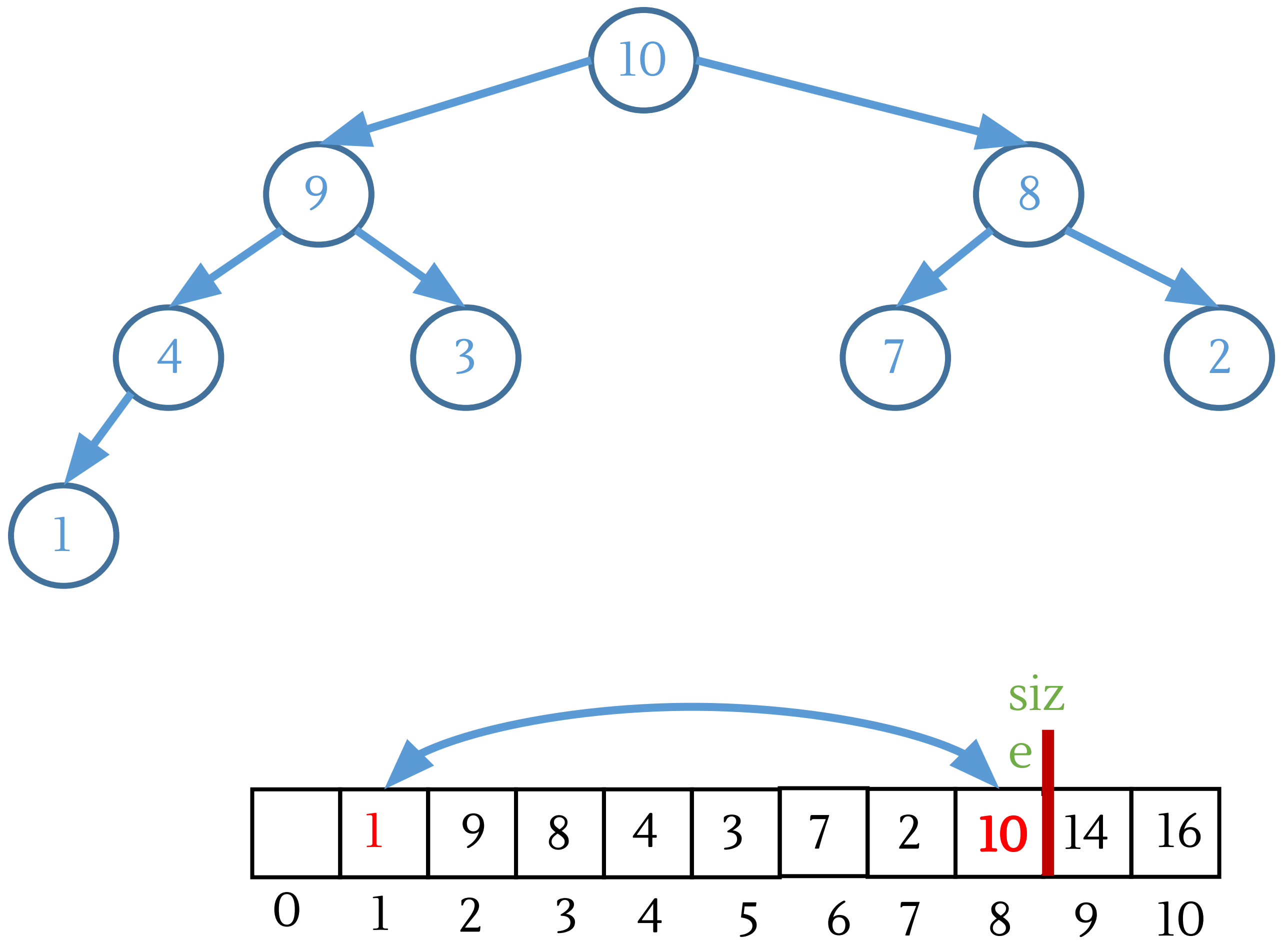


Heap Sort

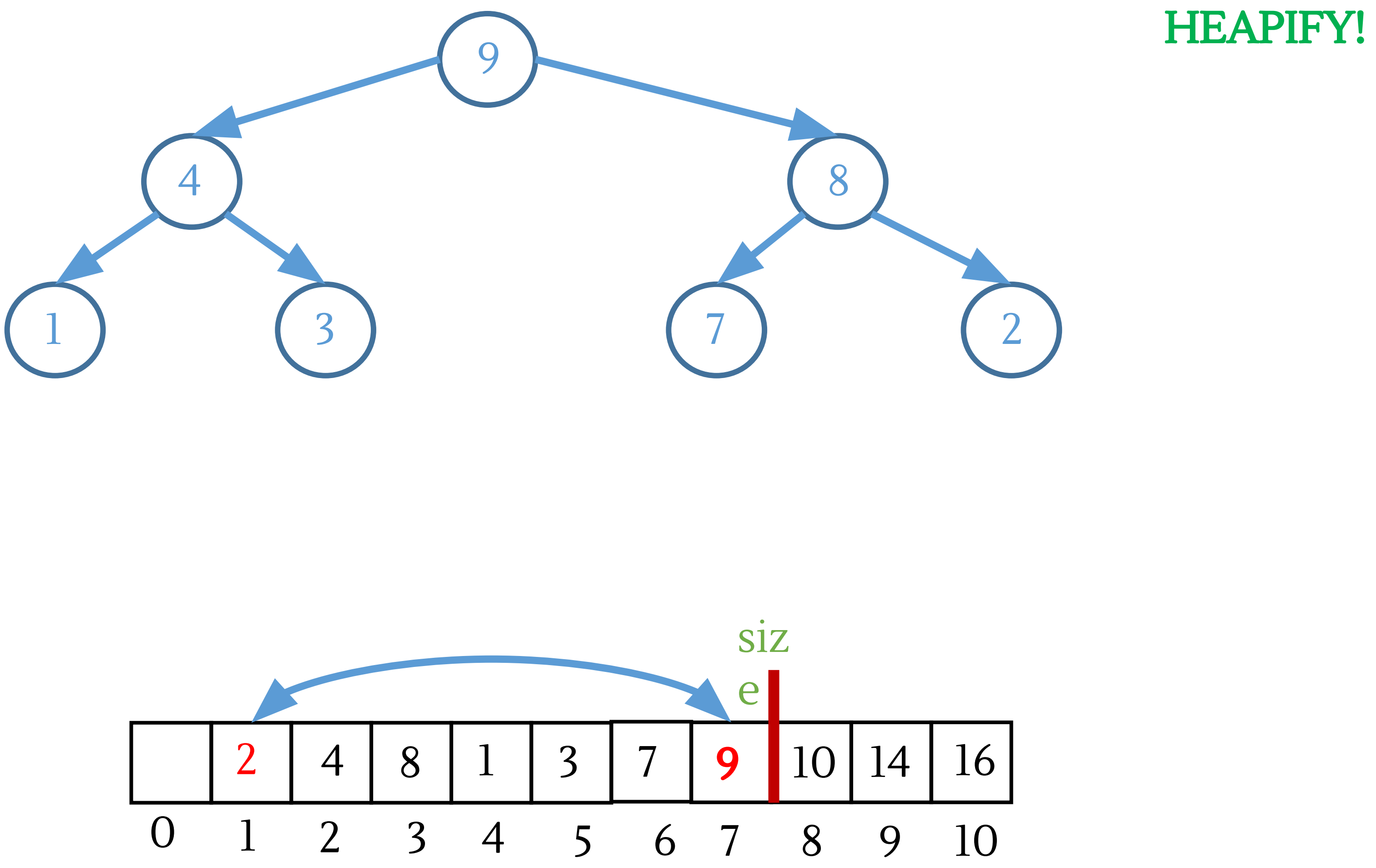


Heap Sort

HEAPIFY!

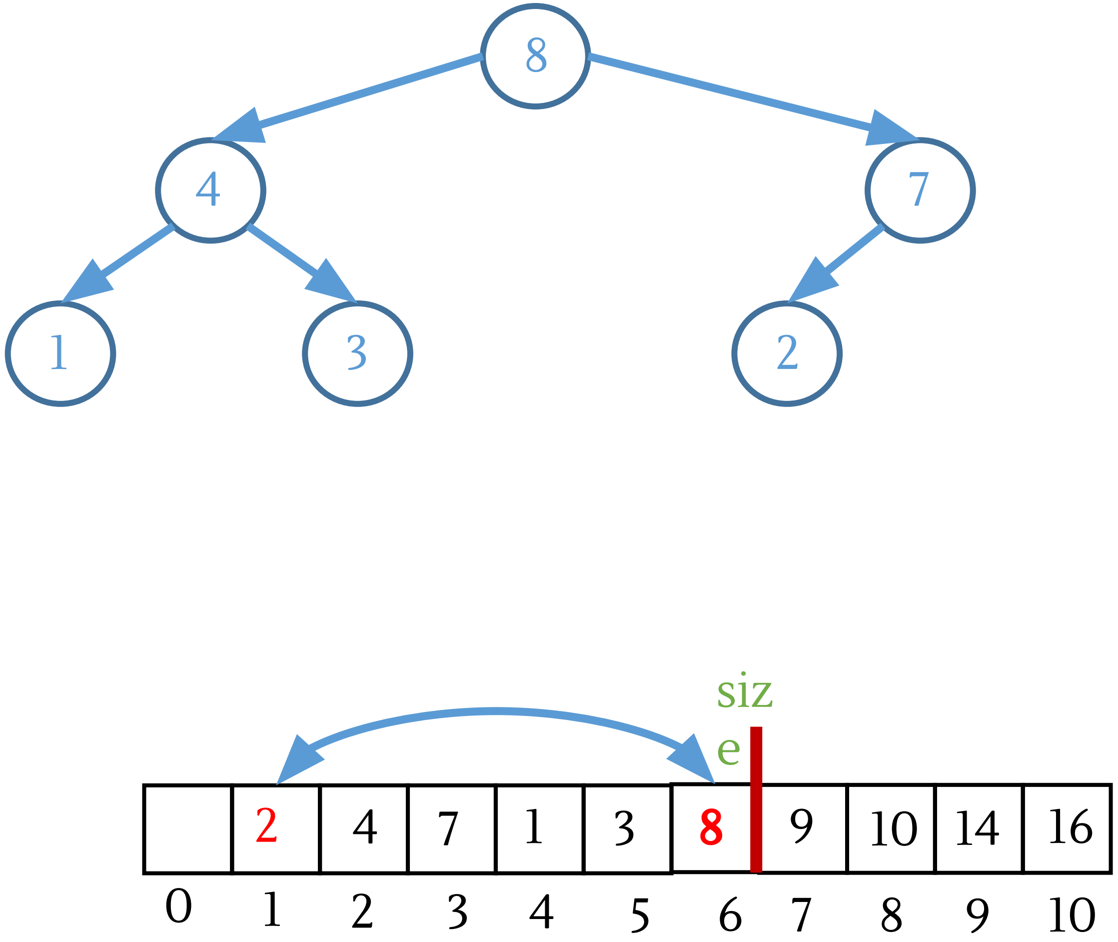


Heap Sort



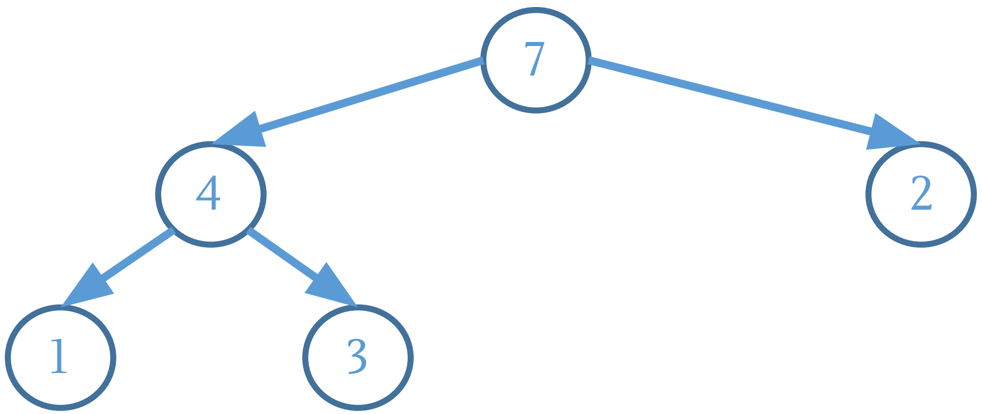
Heap Sort

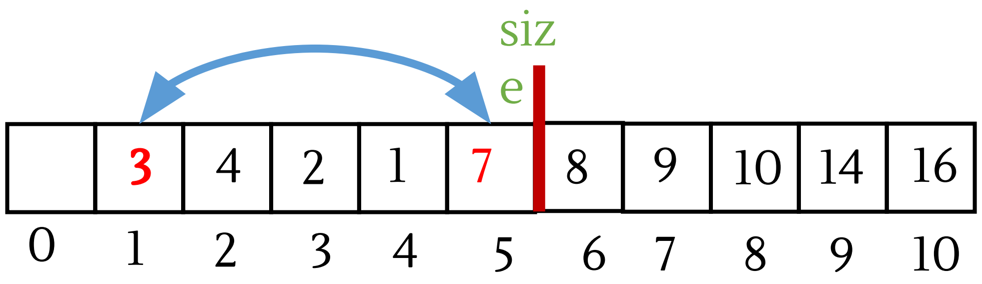
HEAPIFY!



Heap Sort

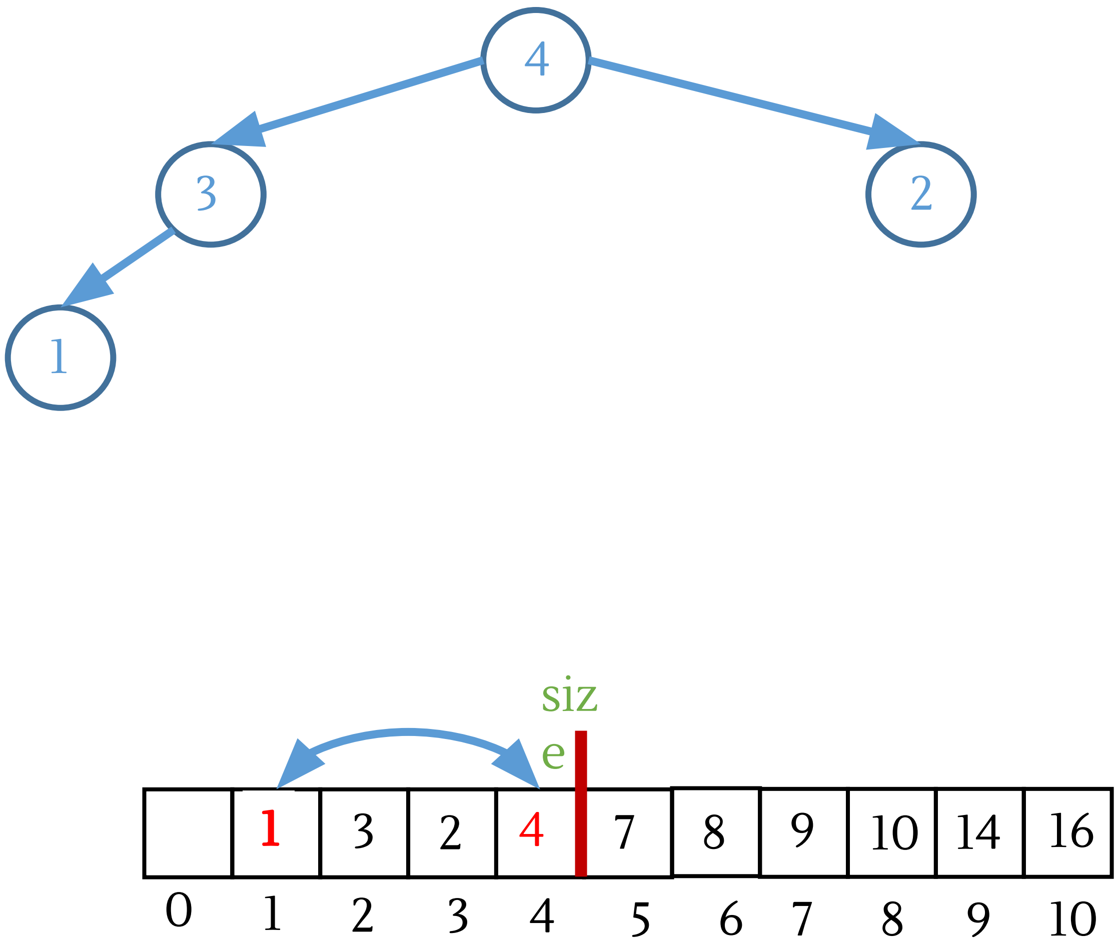
HEAPIFY!



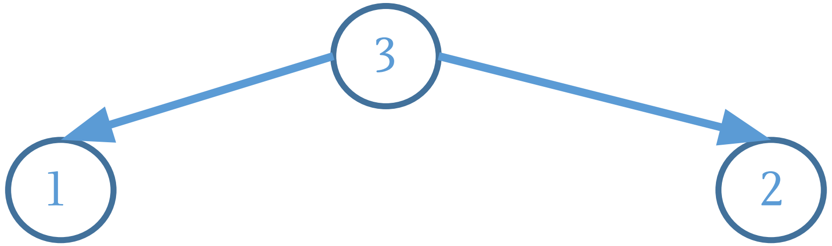


Heap Sort

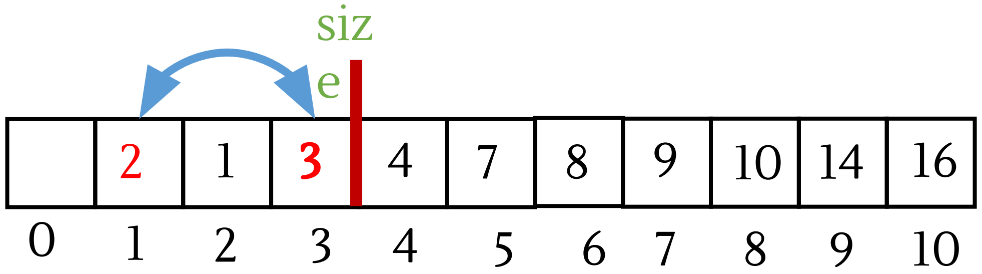
HEAPIFY!



Heap Sort

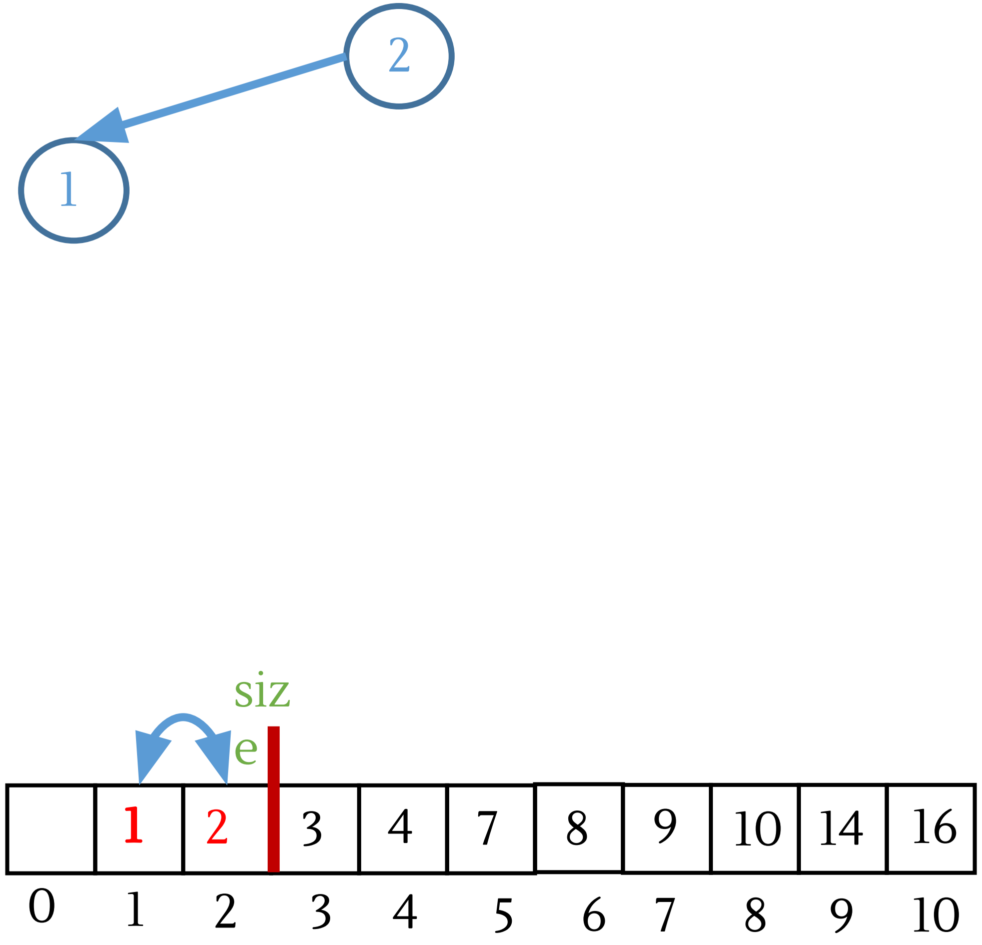


HEAPIFY!

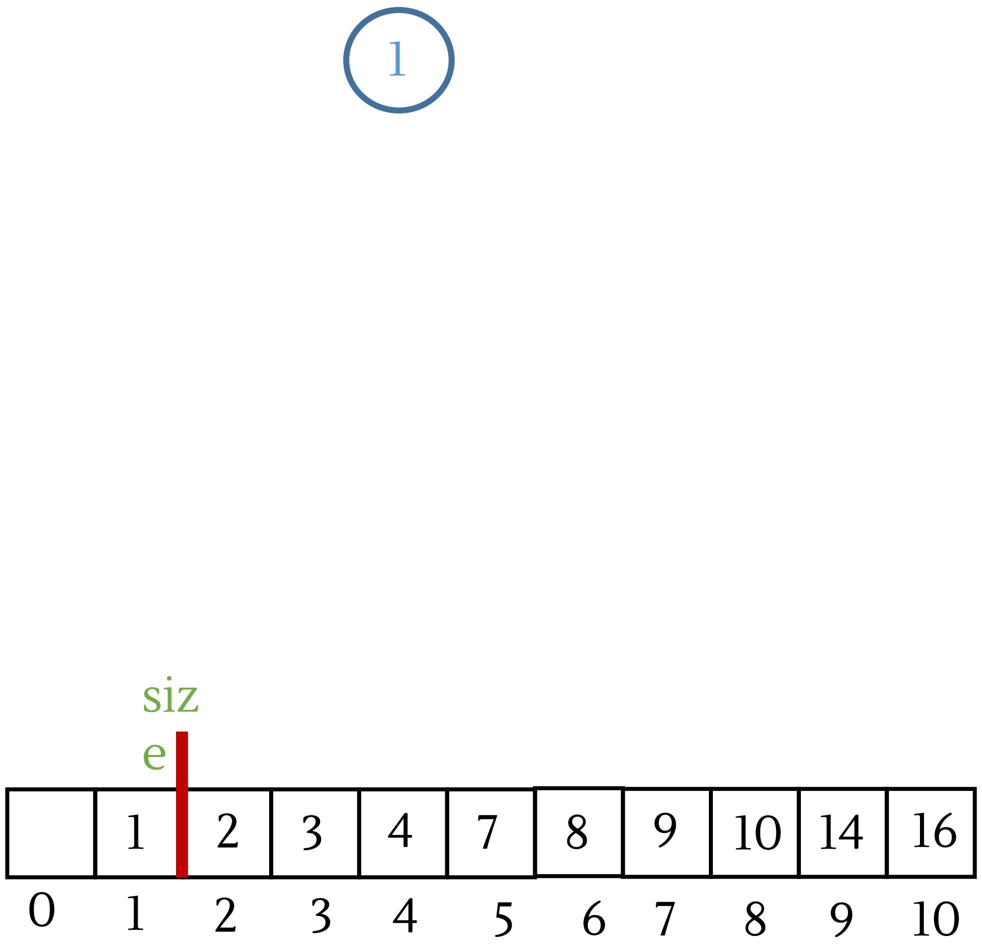


Heap Sort

HEAPIFY!



Heap Sort



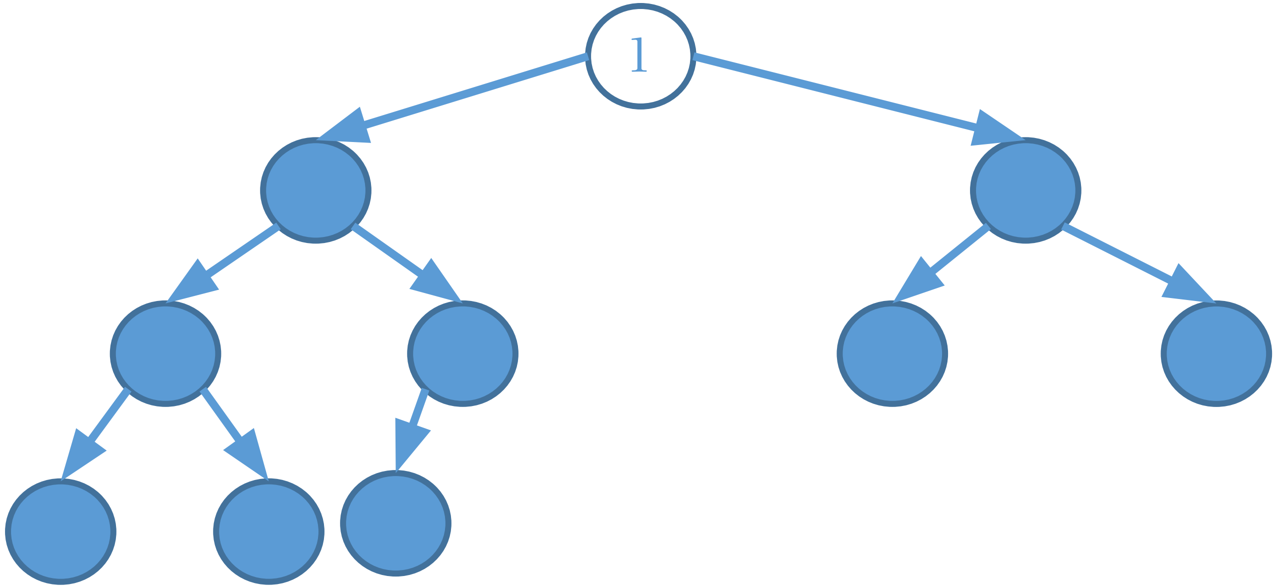
Sorted!

Time Complexity?

- **O(n logn)**

Restore The Array

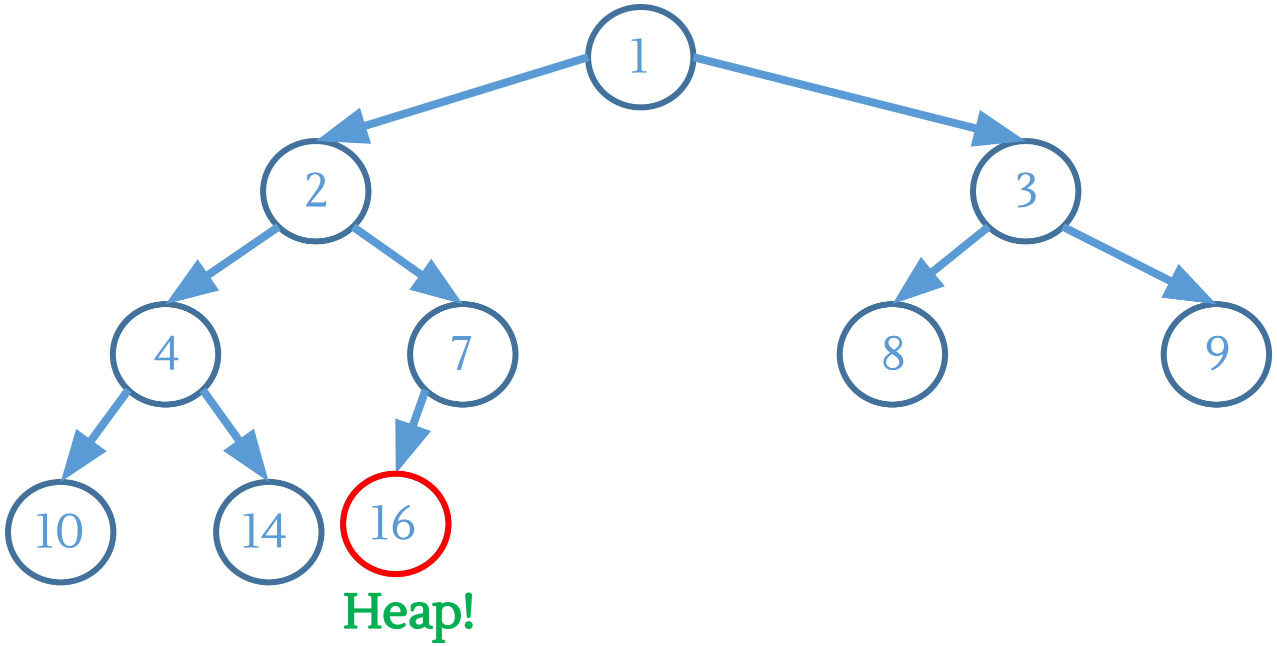
size = 10



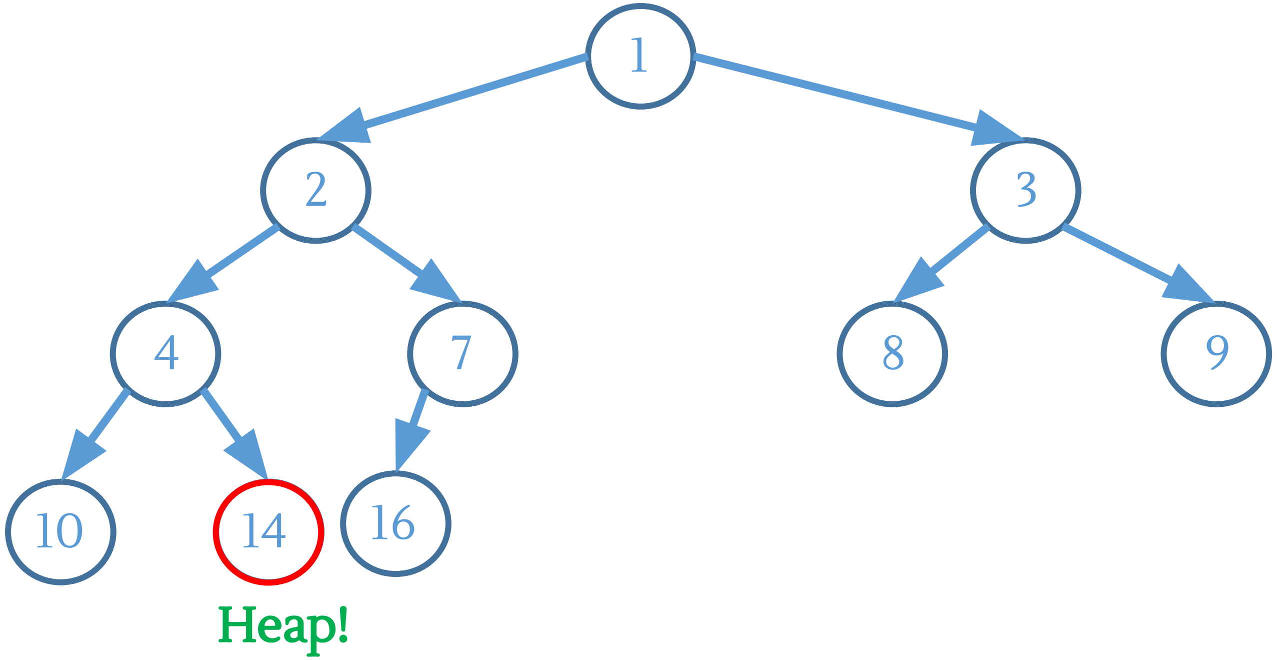
siz
e

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

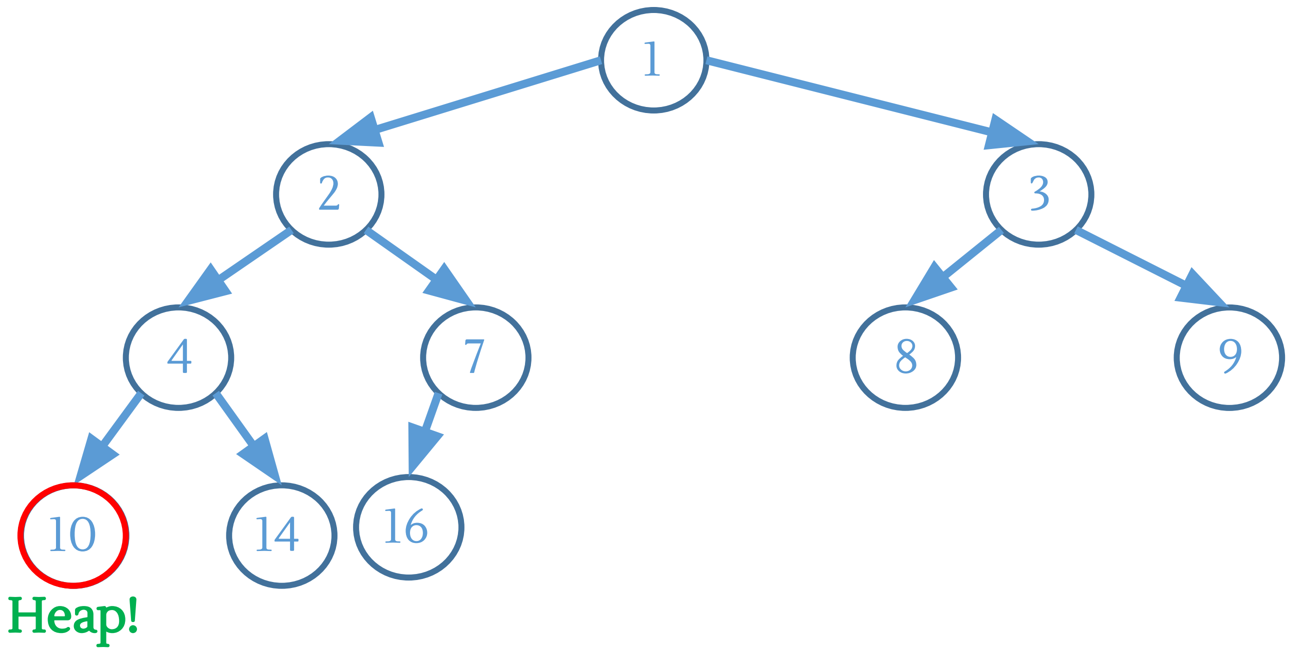
Build Heap



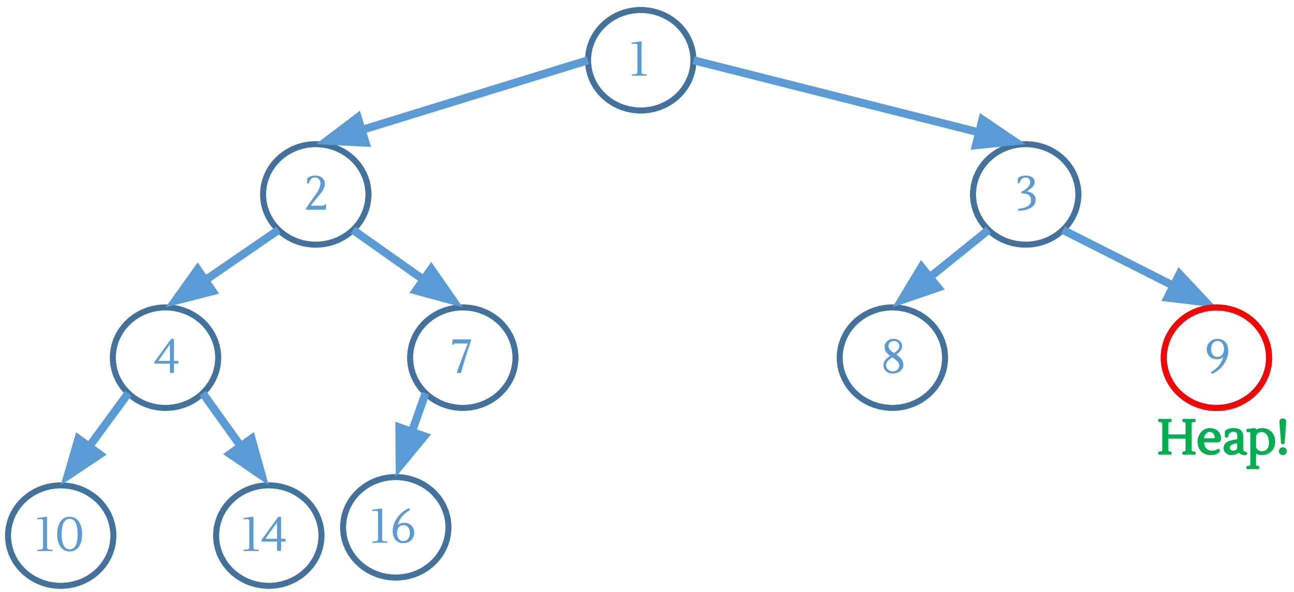
Build Heap



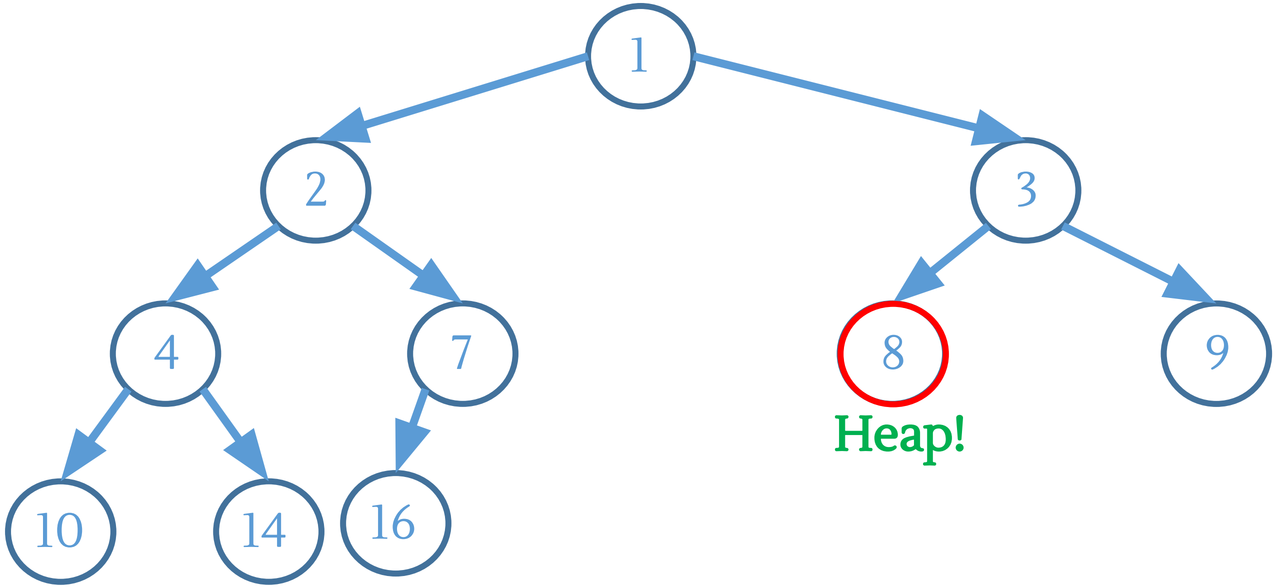
Build Heap



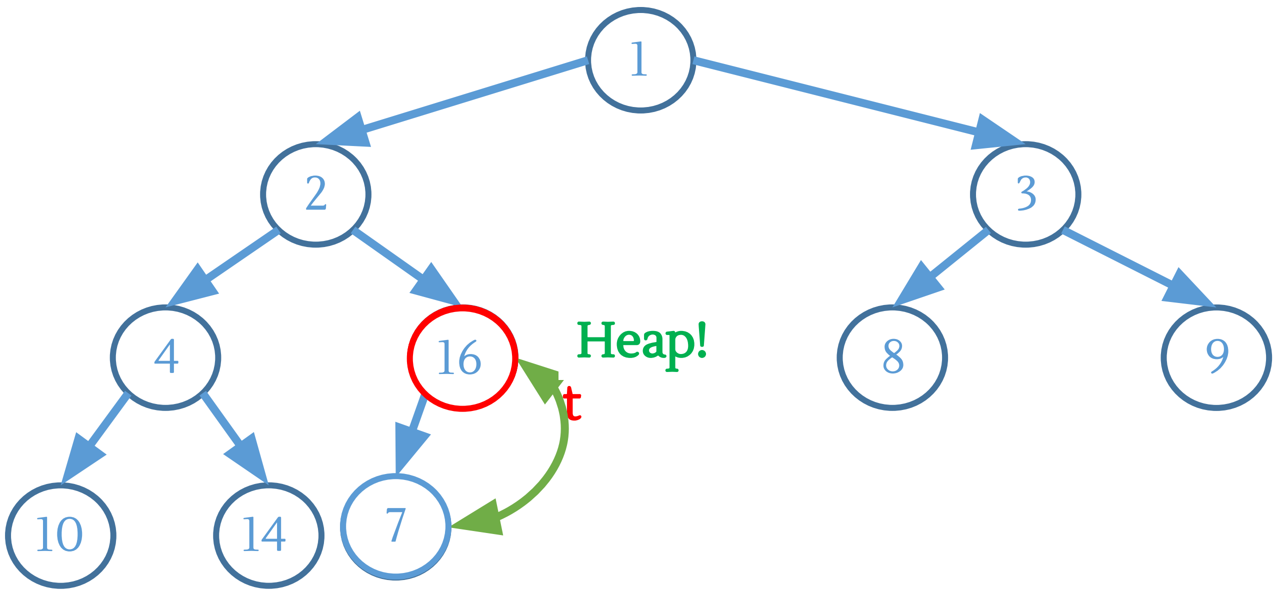
Build Heap



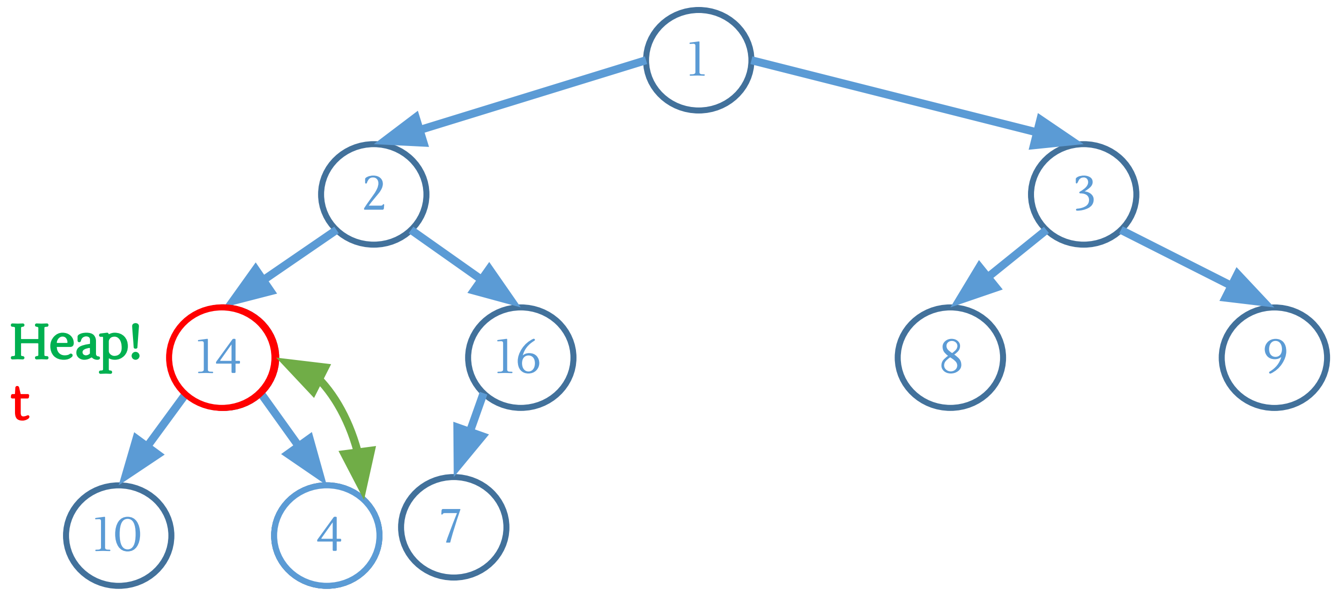
Build Heap



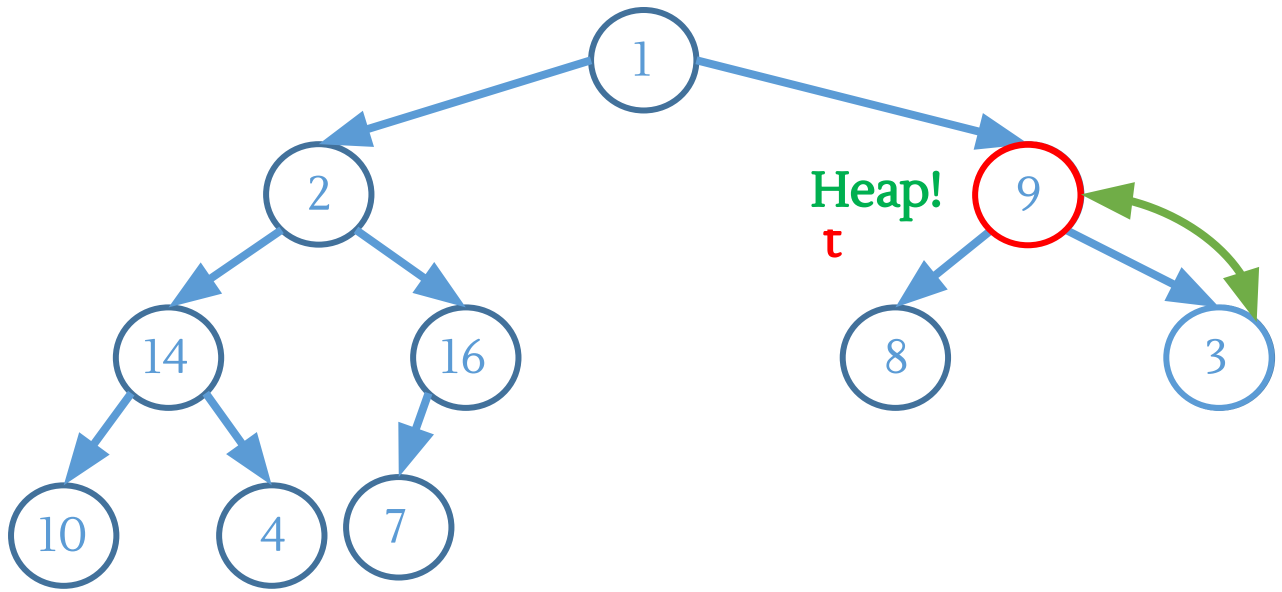
Build Heap



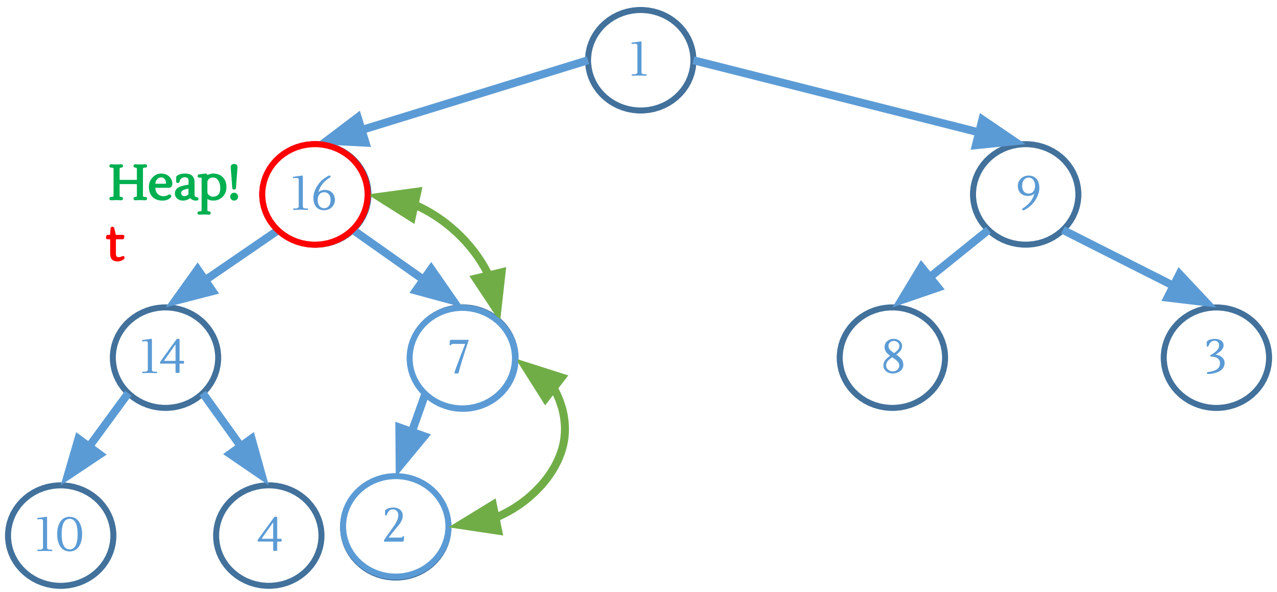
Build Heap



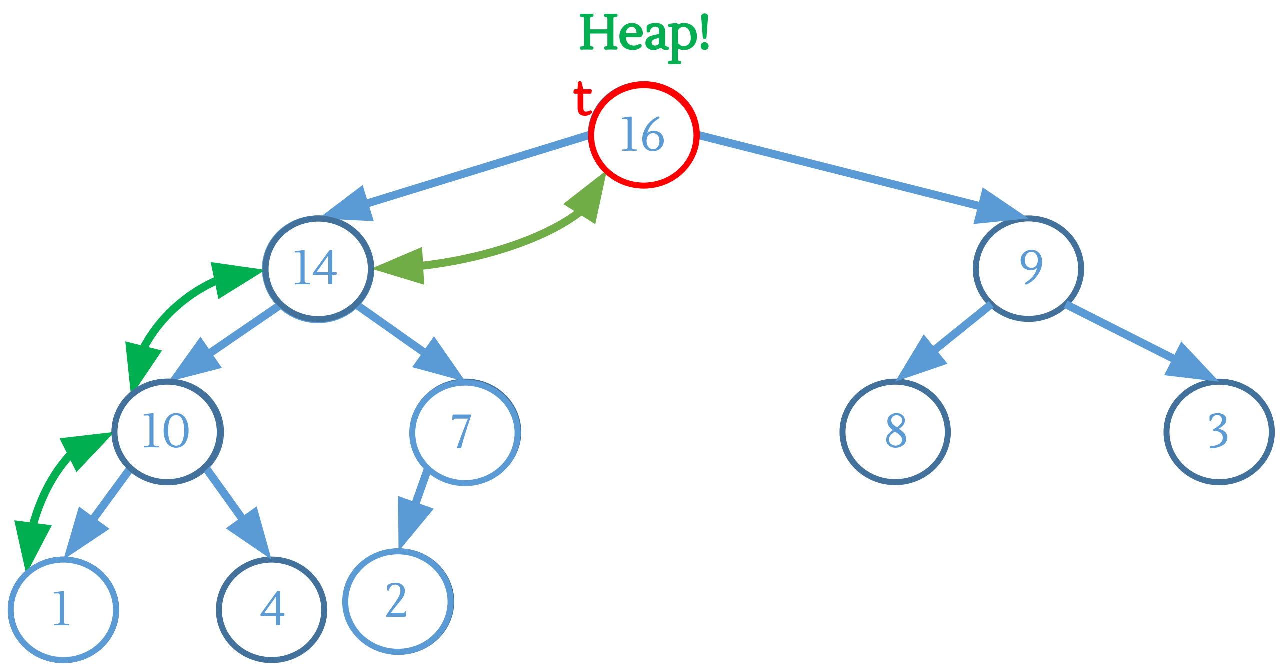
Build Heap



Build Heap



Build Heap



Build Heap

- For all internal vertices from **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 queue
  - **Top:** Returns the Maximum Element of the priority queue
  - **ExtractMax:** Removes and returns the Maximum Element of the priority queue
- 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 **$O(\log n)$.**
2. **HEAP-INCREASE-KEY:** Implements the INCREASE-KEY operation and running time is **$O(\log n)$.**
3. **HEAP-MAXIMUM(A):** Implements MAXIMUM(S) operation in **$\Theta(1)$** time.
4. **HEAP-EXTRACT-MAX:** Implements EXTRACT-MAX(S) and running time is **$O(\log n)$**.

TRIE

Prepared By

Lec Swapnil Biswas

TRIE

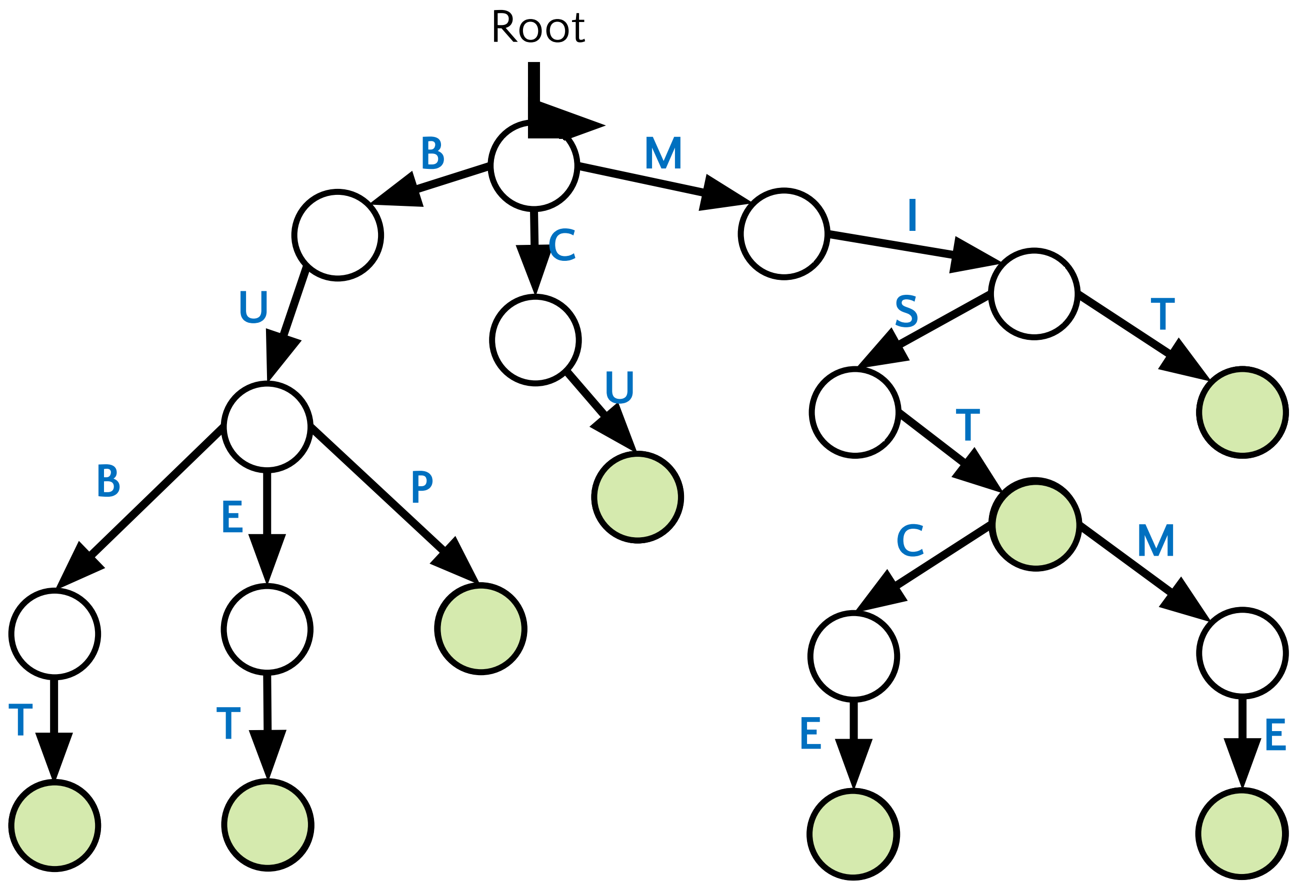
- A tree based data structure (k-ary tree)
- Root is an empty node.
- (k=26) Each node will have 26 children (Each child represents a alphabetic letter)
- Implemented by linked data structure
- It allows for very fast searching and insertion operations
- The word TRIE comes from the word Re**trie**val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

WHY TRIE?

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

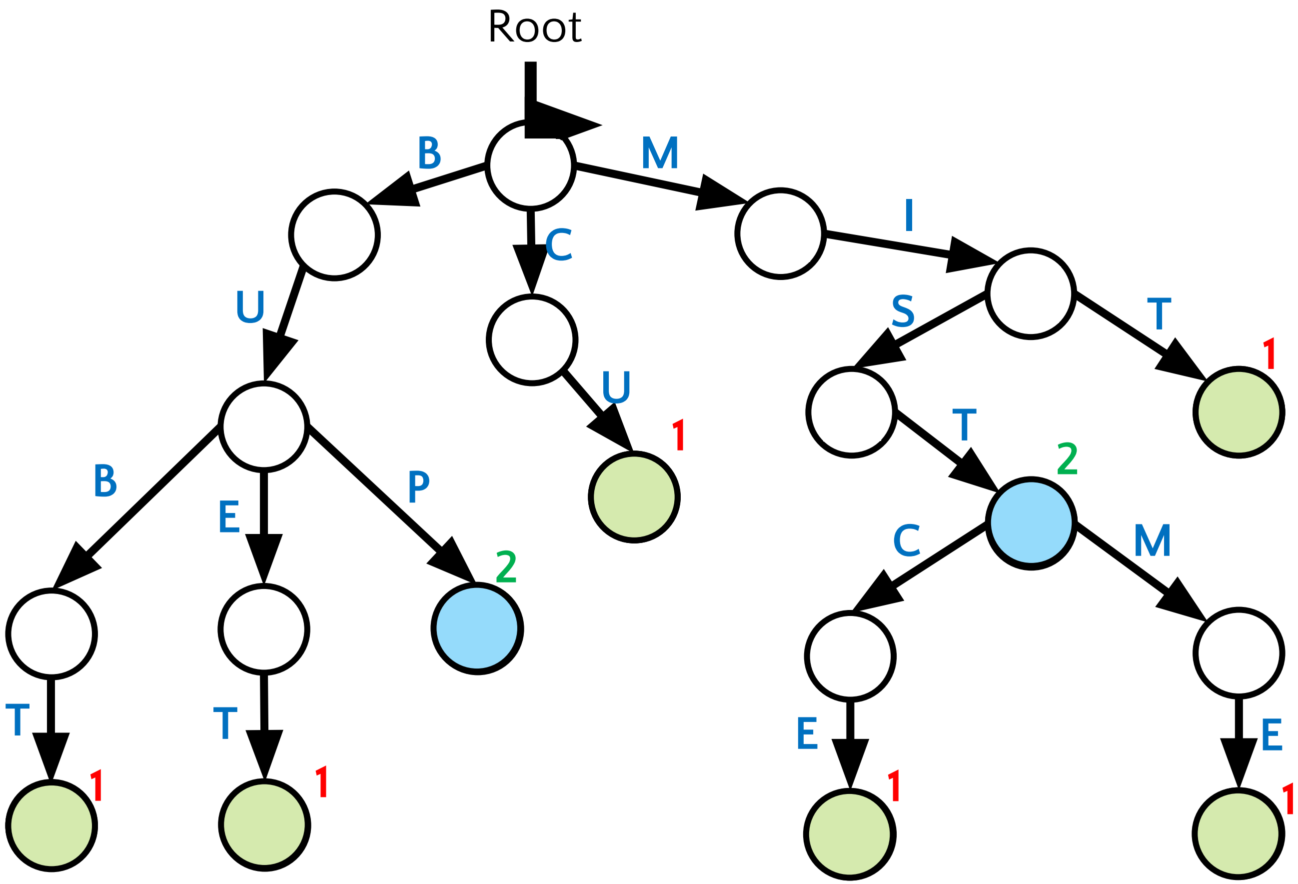
INSERT IN TRIE

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



INSERT IN TRIE (WITH COUNTER)

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

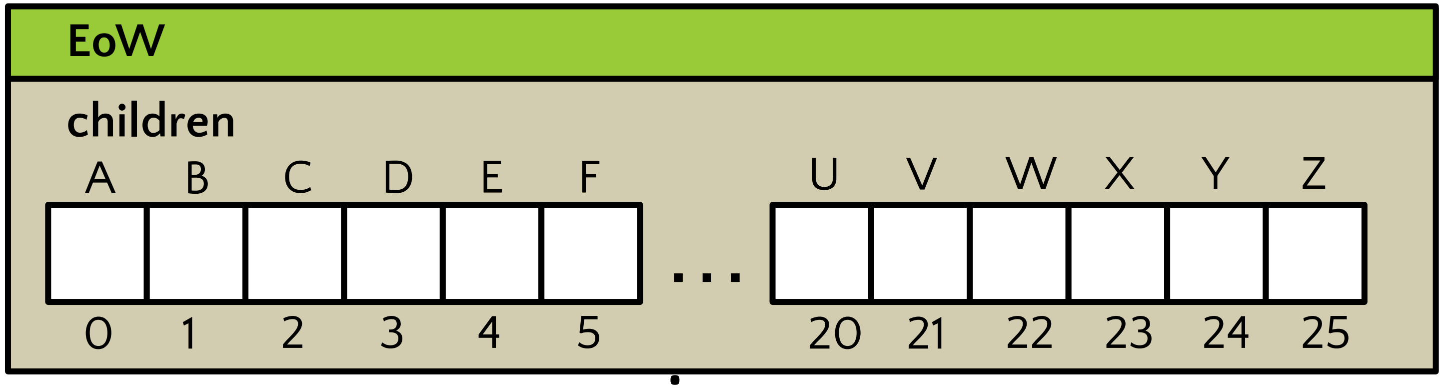


INSERT IN TRIE (WITH COUNTER)

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

NODE REPRESENTATION

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



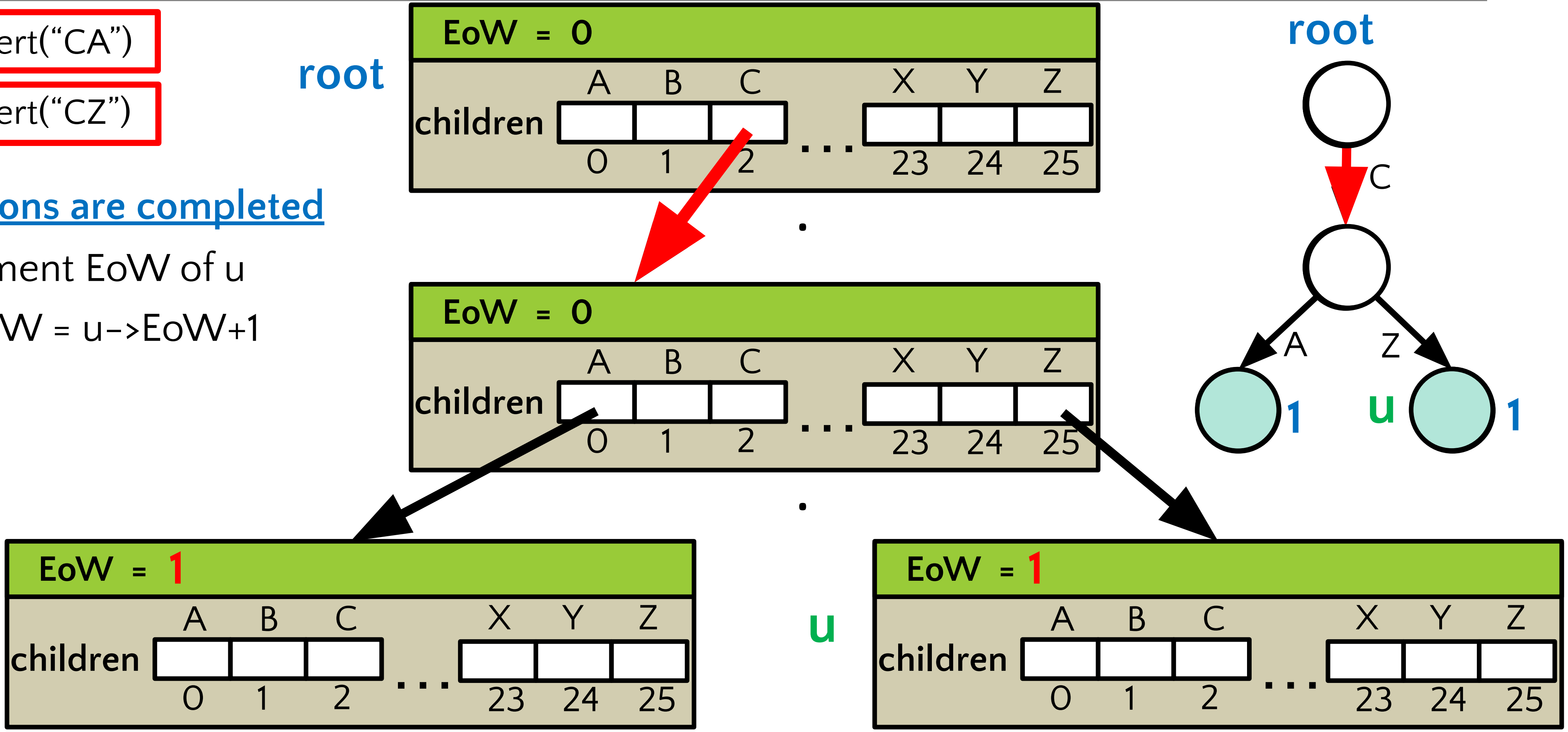
NODE REPRESENTATION

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

Iterations are completed

Increment EoW of u

u->EoW = u->EoW+1



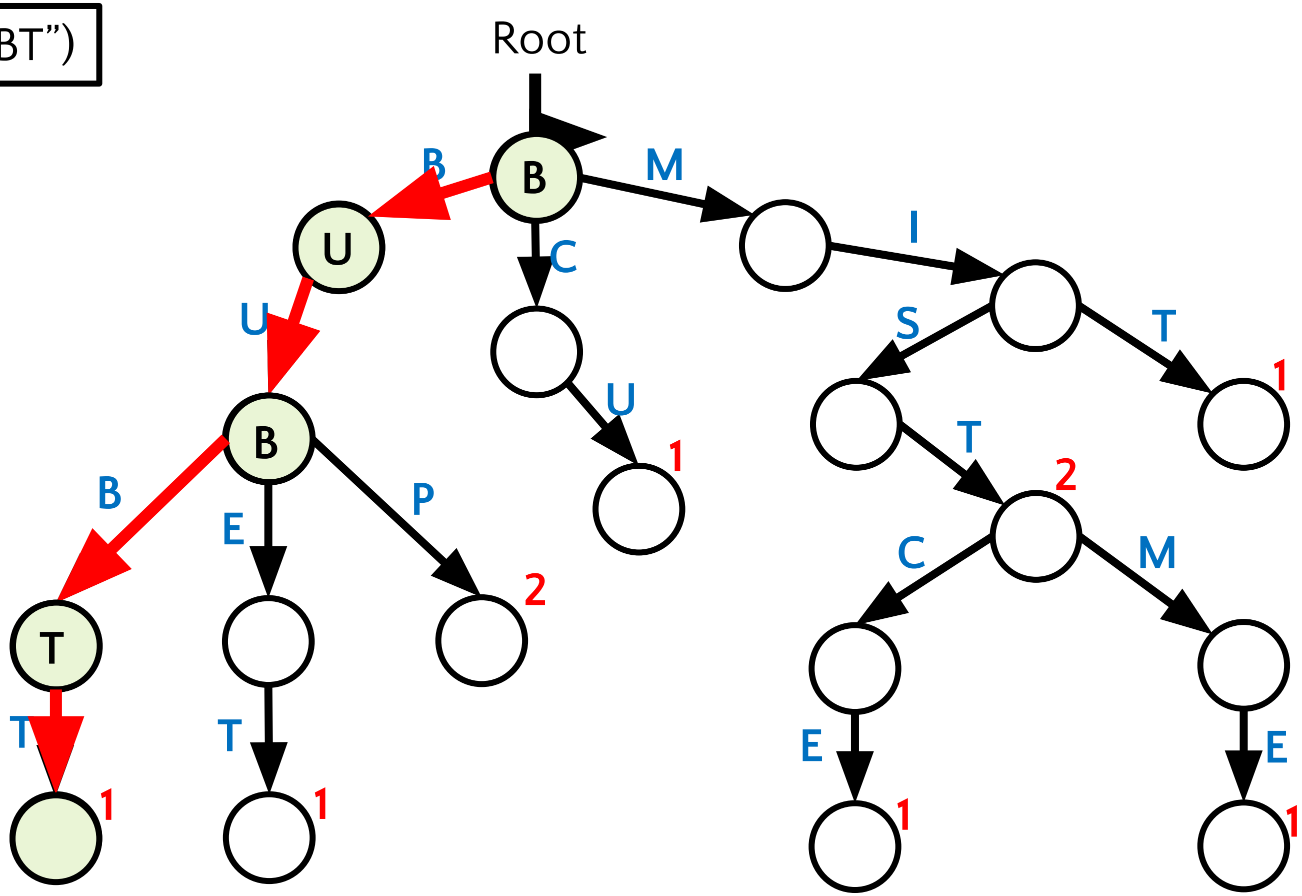
INSERT IN TRIE

```
insert(x)
    Node pointer u ← root                  Initially pointing u at the root
    for k ← 0 to size(x) - 1               Iterates for size(x) number of times
        r ← 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] ← new Node( )   Creates new node under children[r]
        u ← u->children[r]                 Pushes u down for next iteration
    u->EoW ← 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

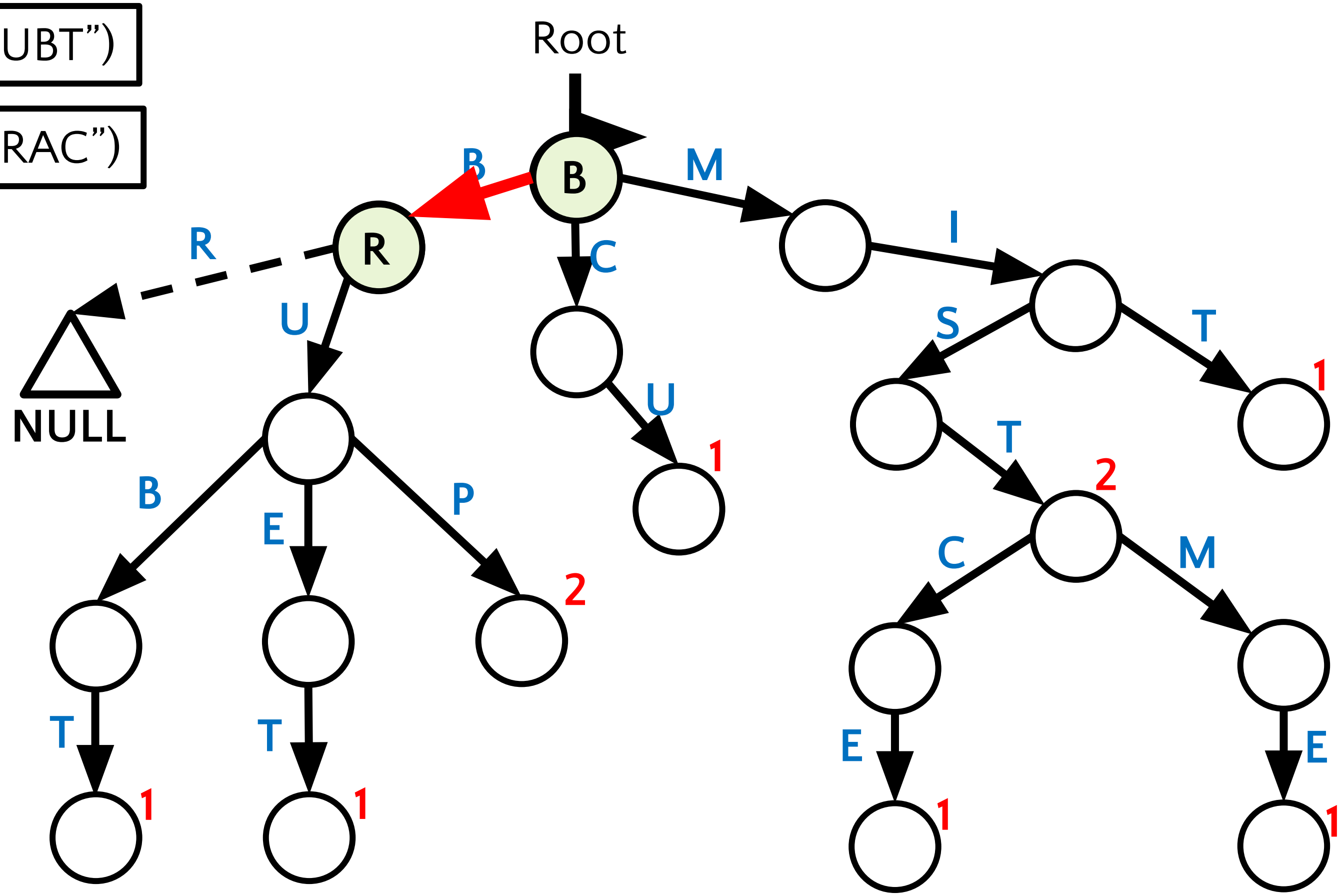
Means "BUBT" exists

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

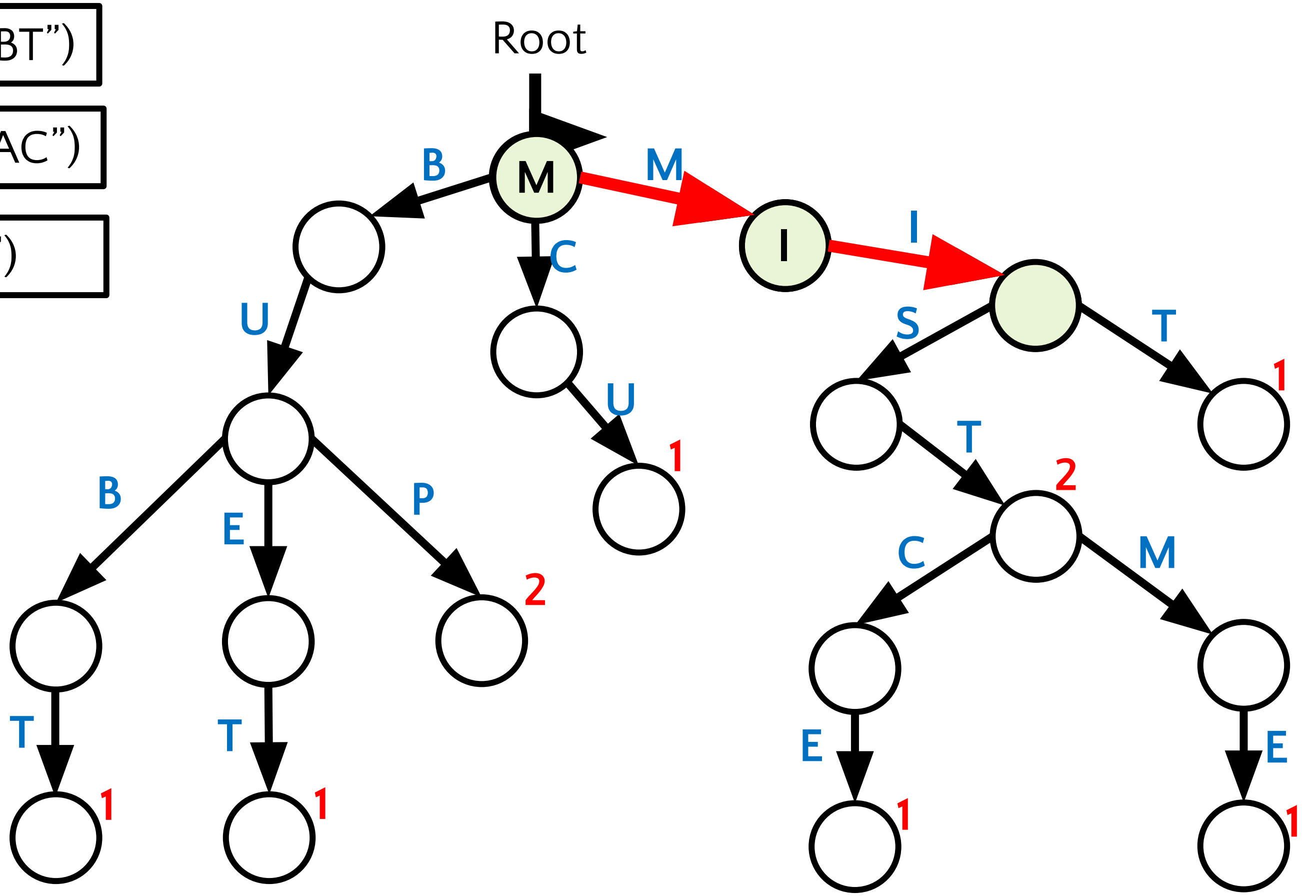
SEARCH IN TRIE

- ❑ 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")



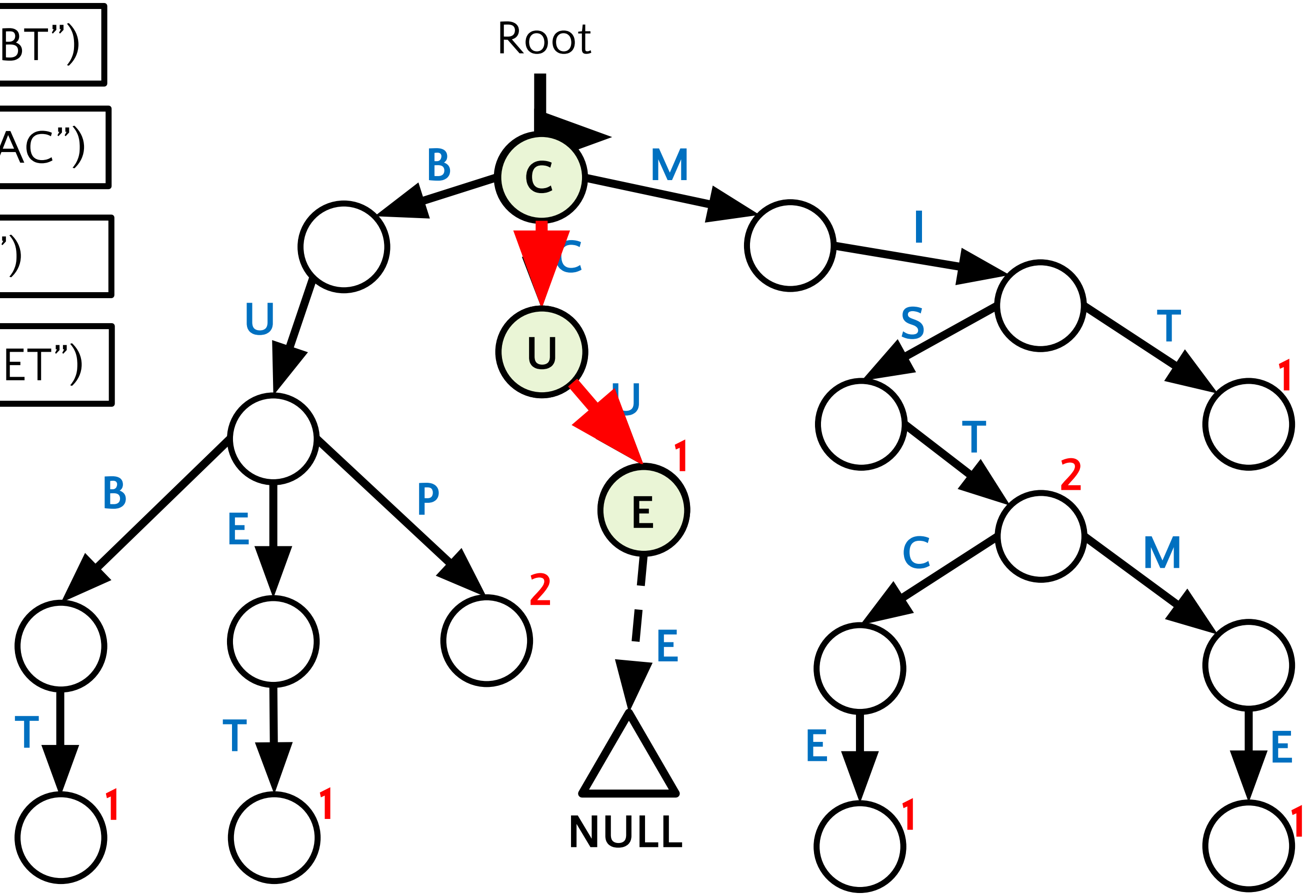
We can't reach a node with counter=0

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")

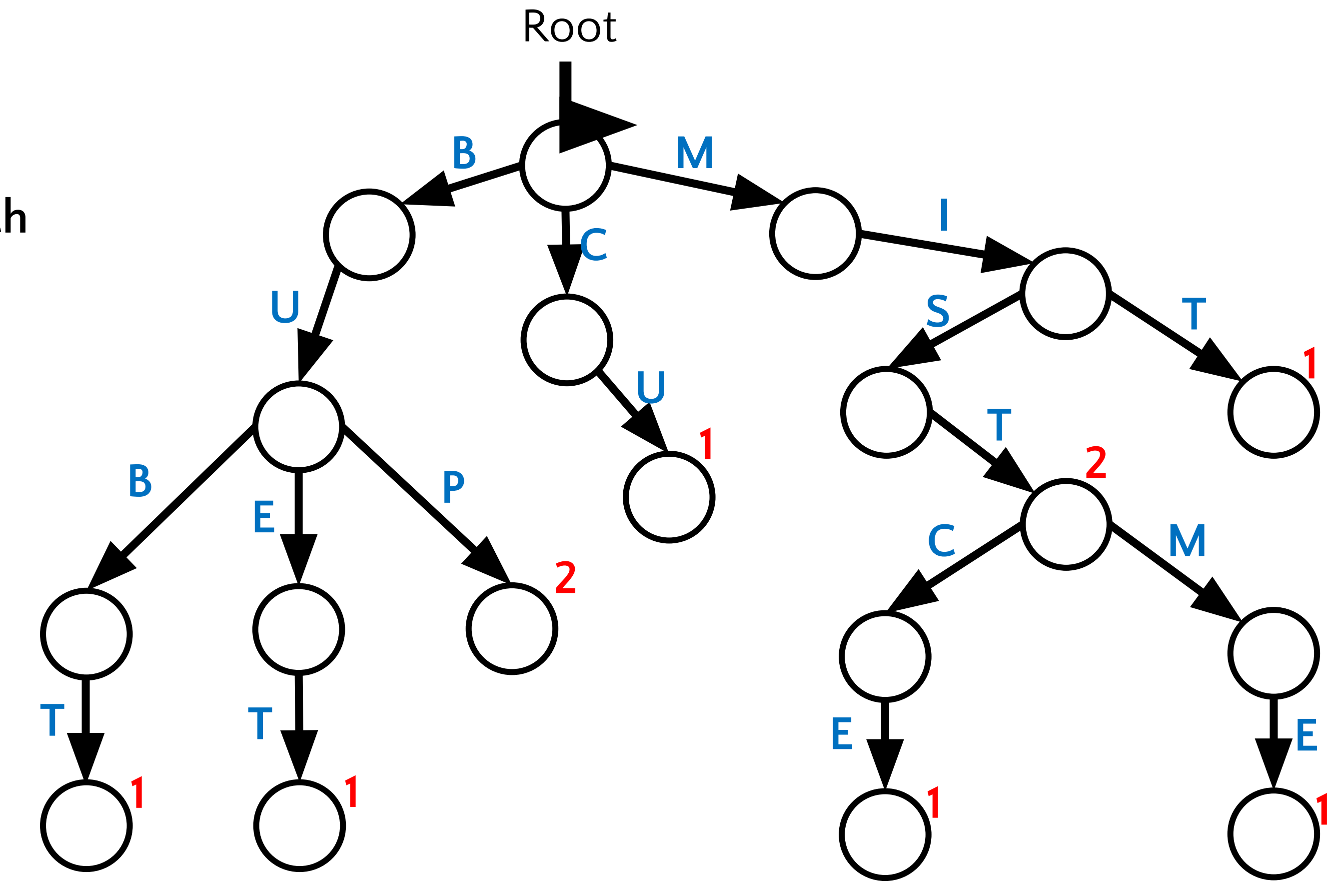


We reach to NULL

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

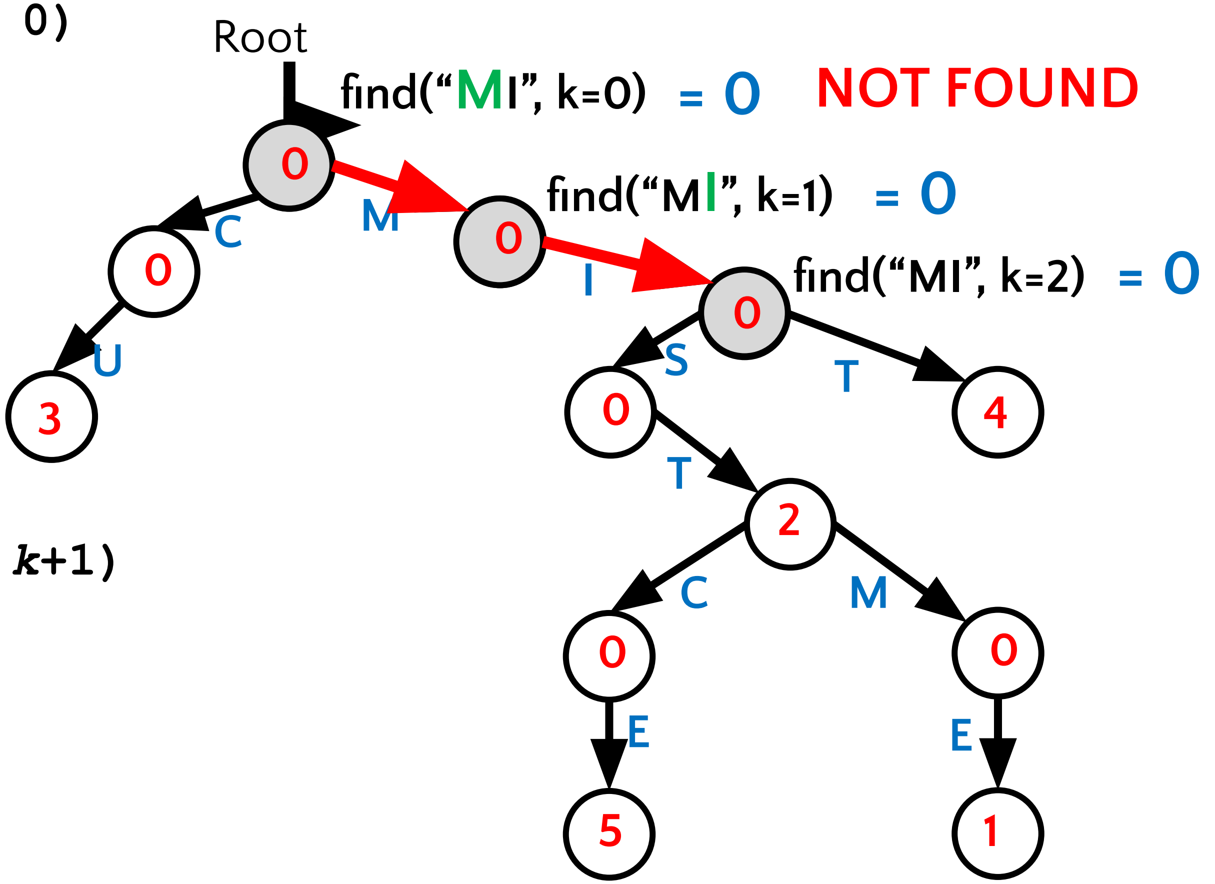
```
int relPos(char c){
    int ascii = (int) c;
    return ascii – 65;
}
```

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")

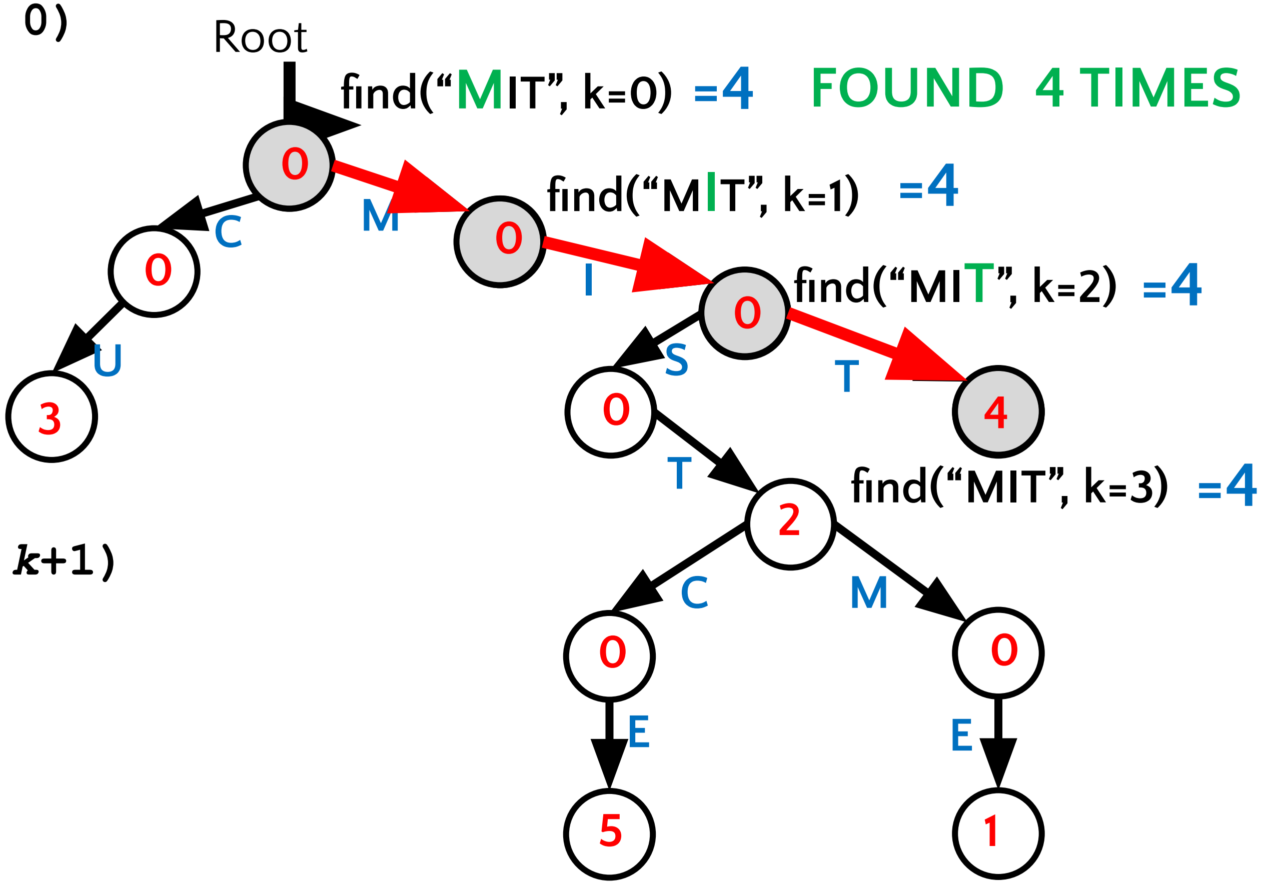


SEARCH IN TRIE

```
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")

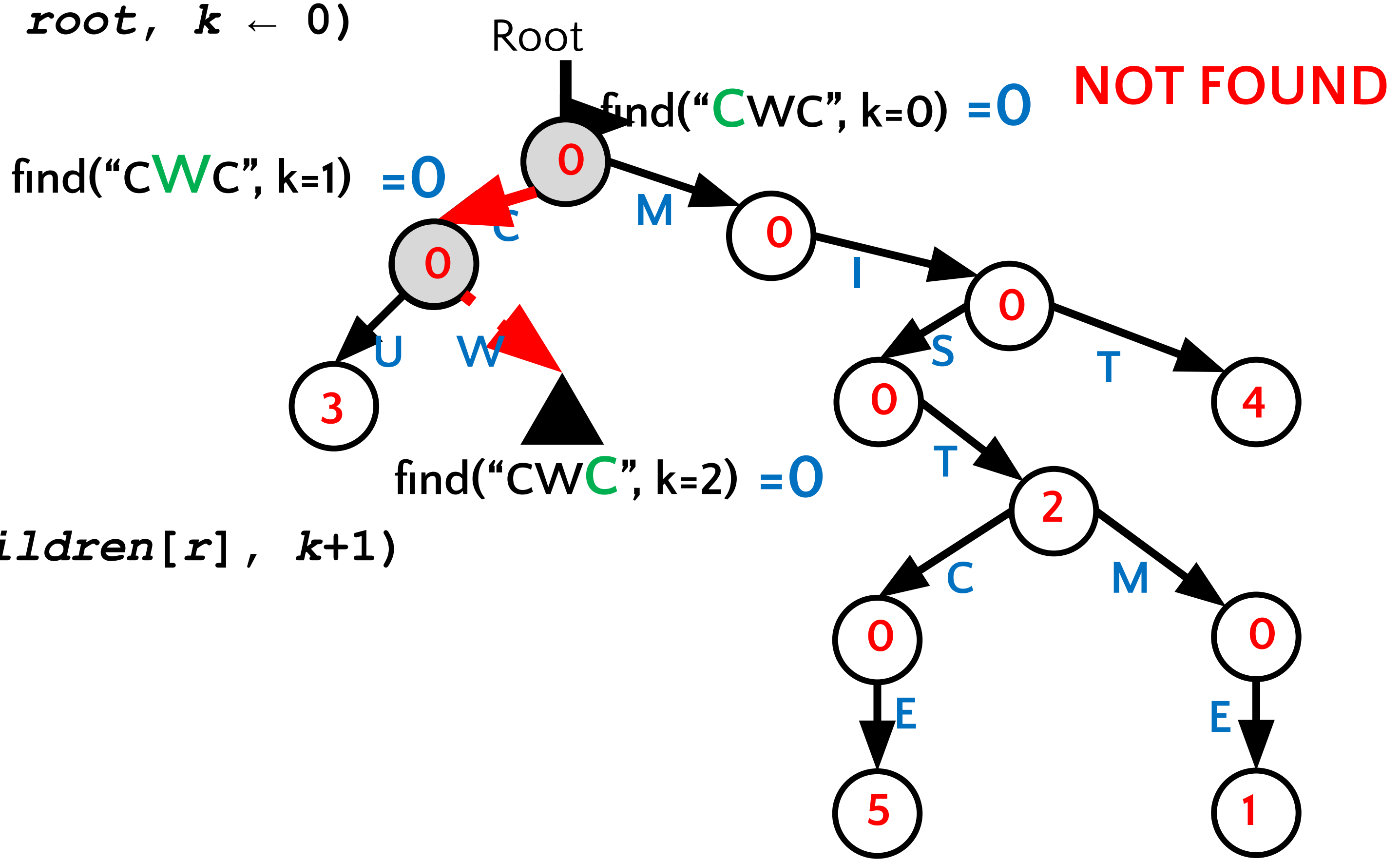


SEARCH IN TRIE

```
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

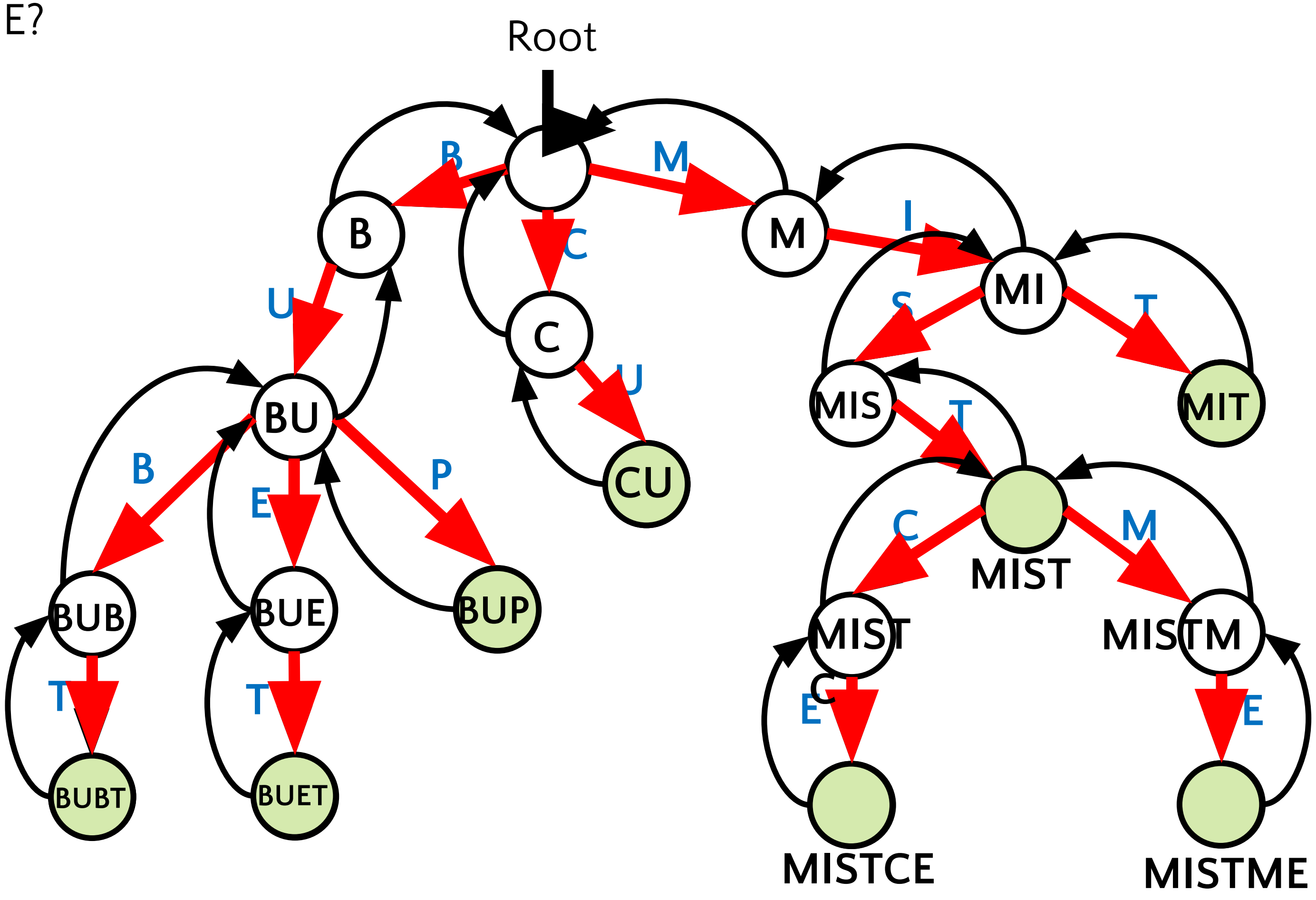
  CU

  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);
                }
        }
}
```

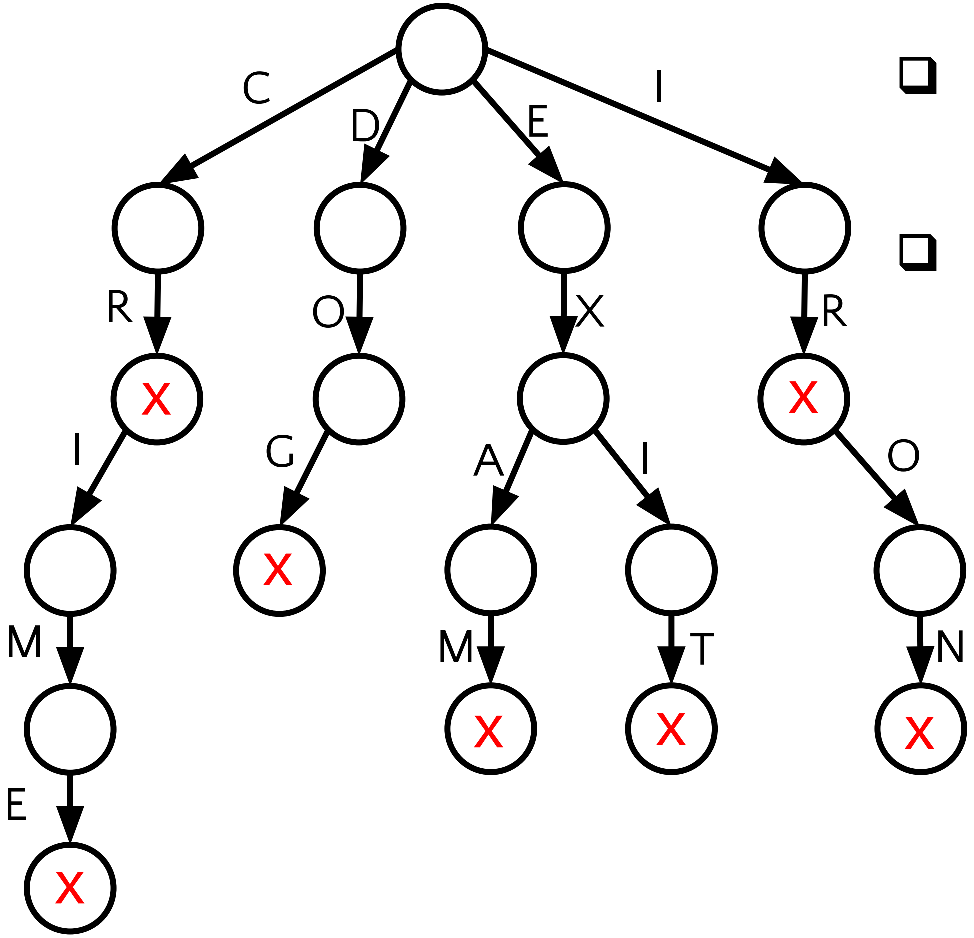
Base case:
If the pointer reaches to the end of a word
Then the word is printed

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

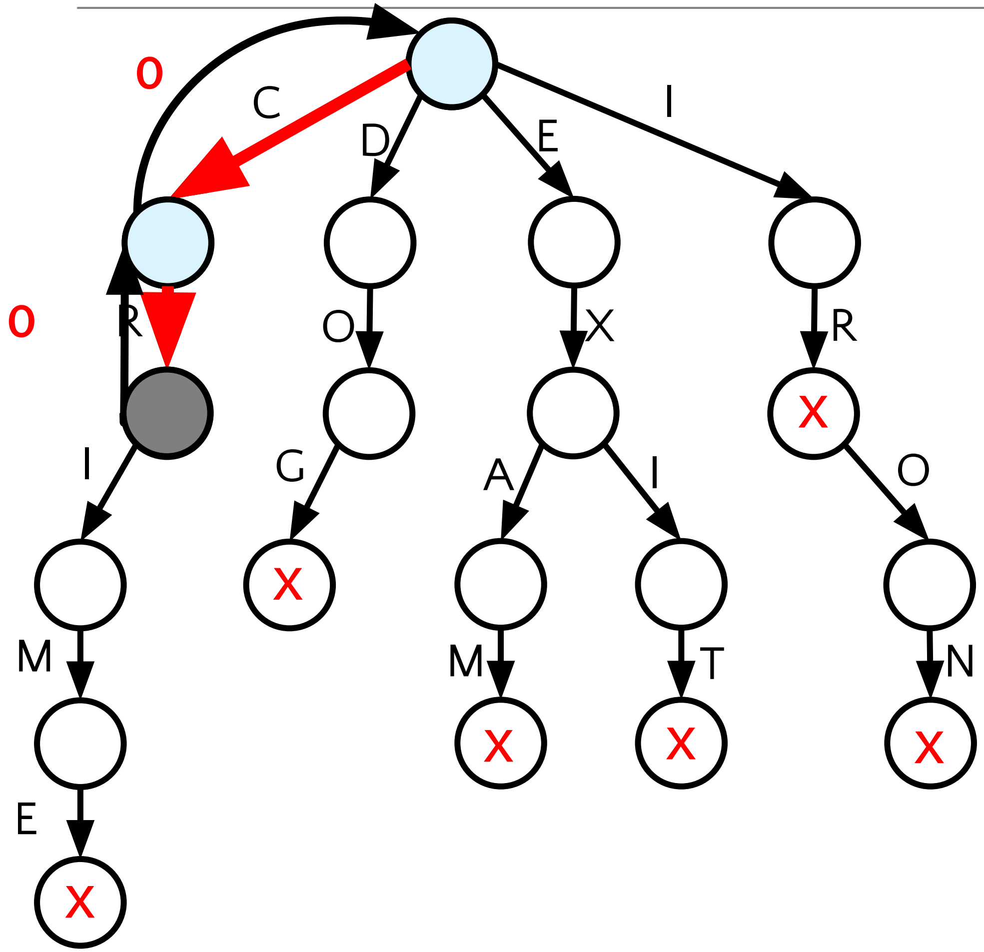
  CR CRIME DOG EXAM EXIT IR IRON

- 2 Cases for deletion

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

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

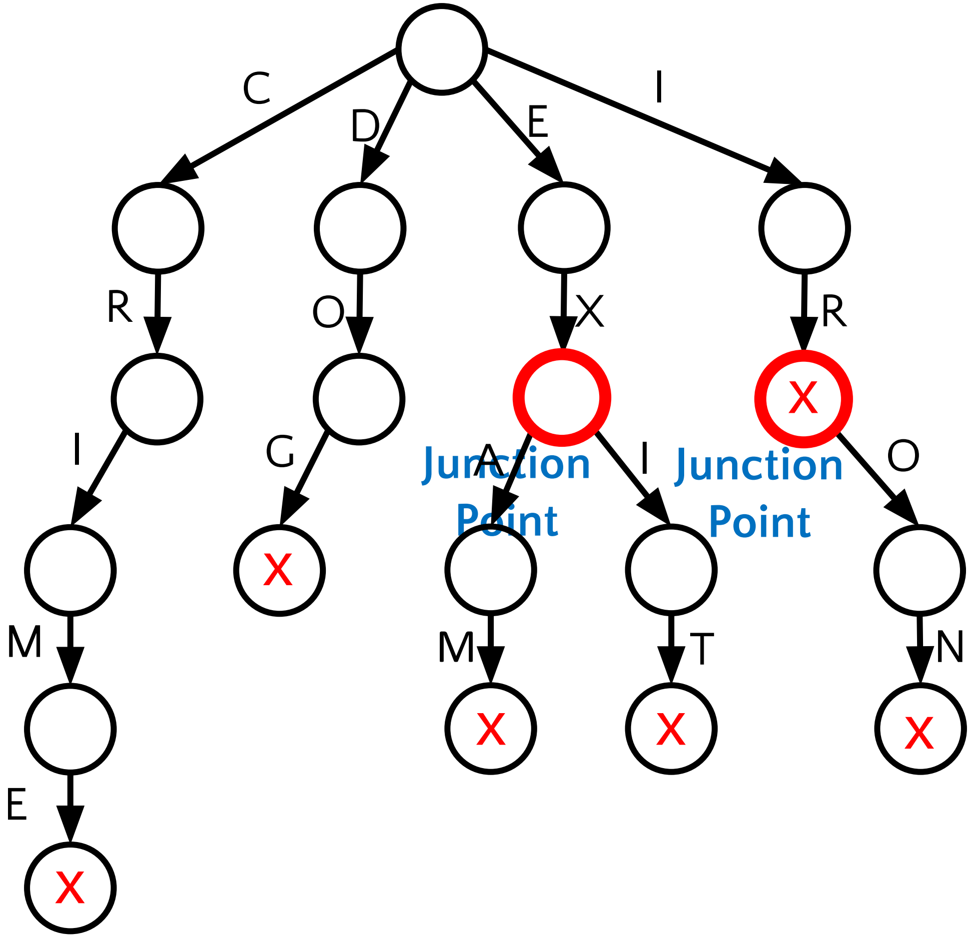
DELETE FROM TRIE (CASE-1)



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

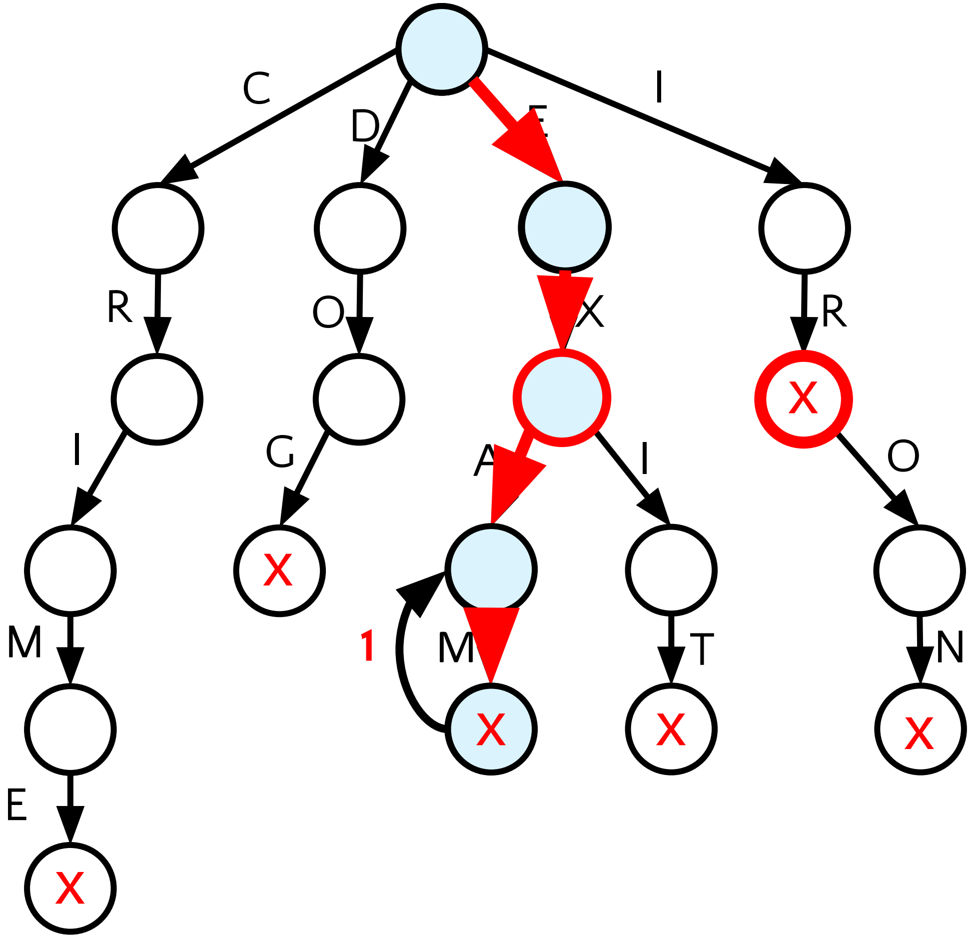
```
bool isLeaf(Node *u){
    for(int i=0; i<26; i++)    if(u->children[i]!=NULL)    return false;
    return true;
}
```

DELETE FROM TRIE (CASE-2)



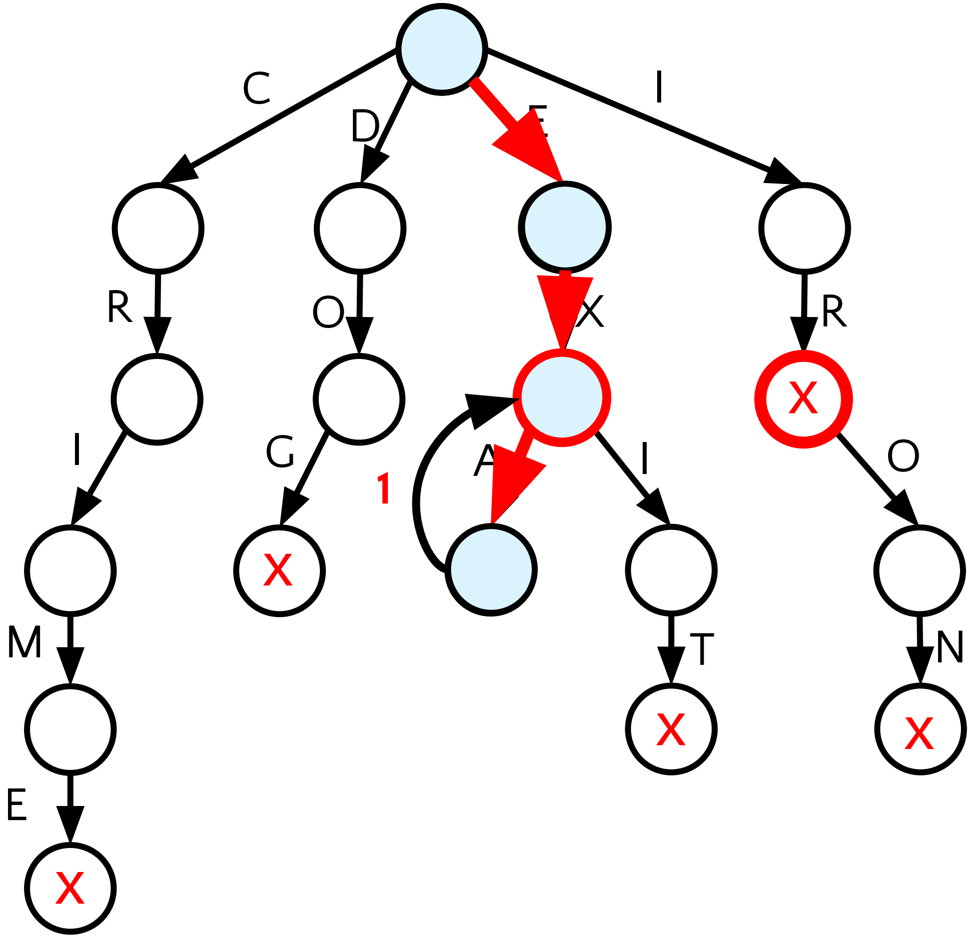
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



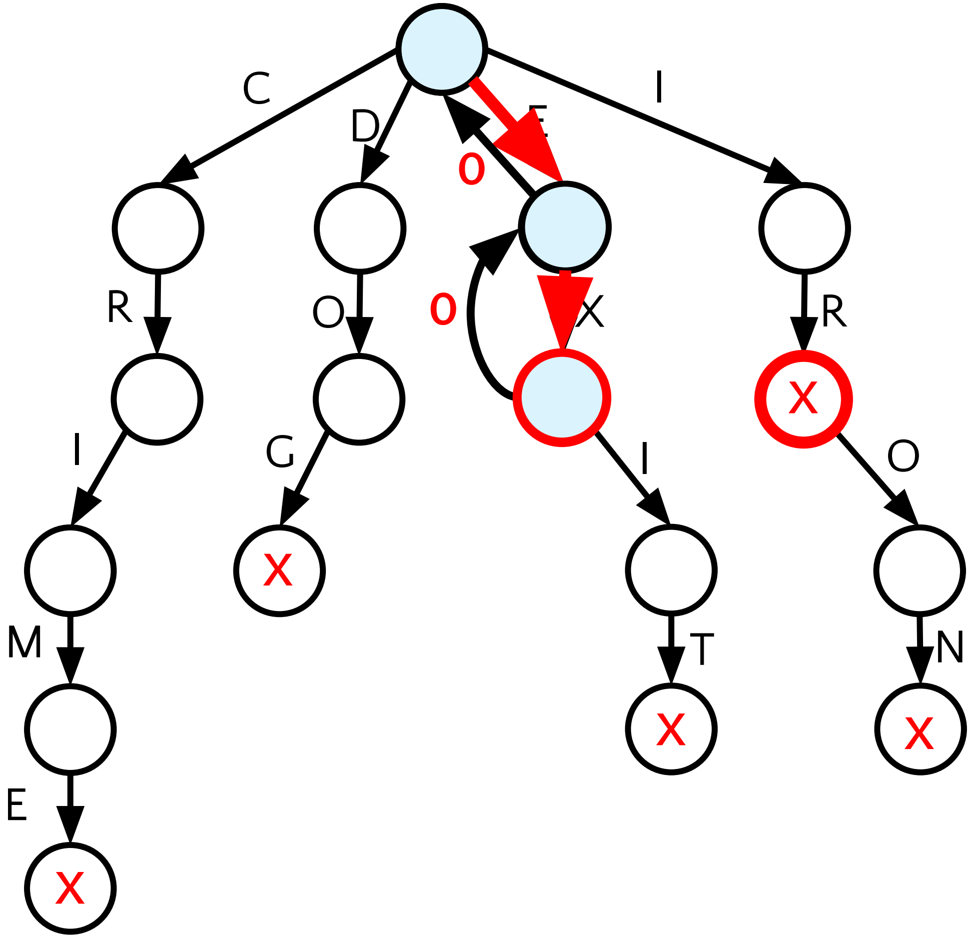
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



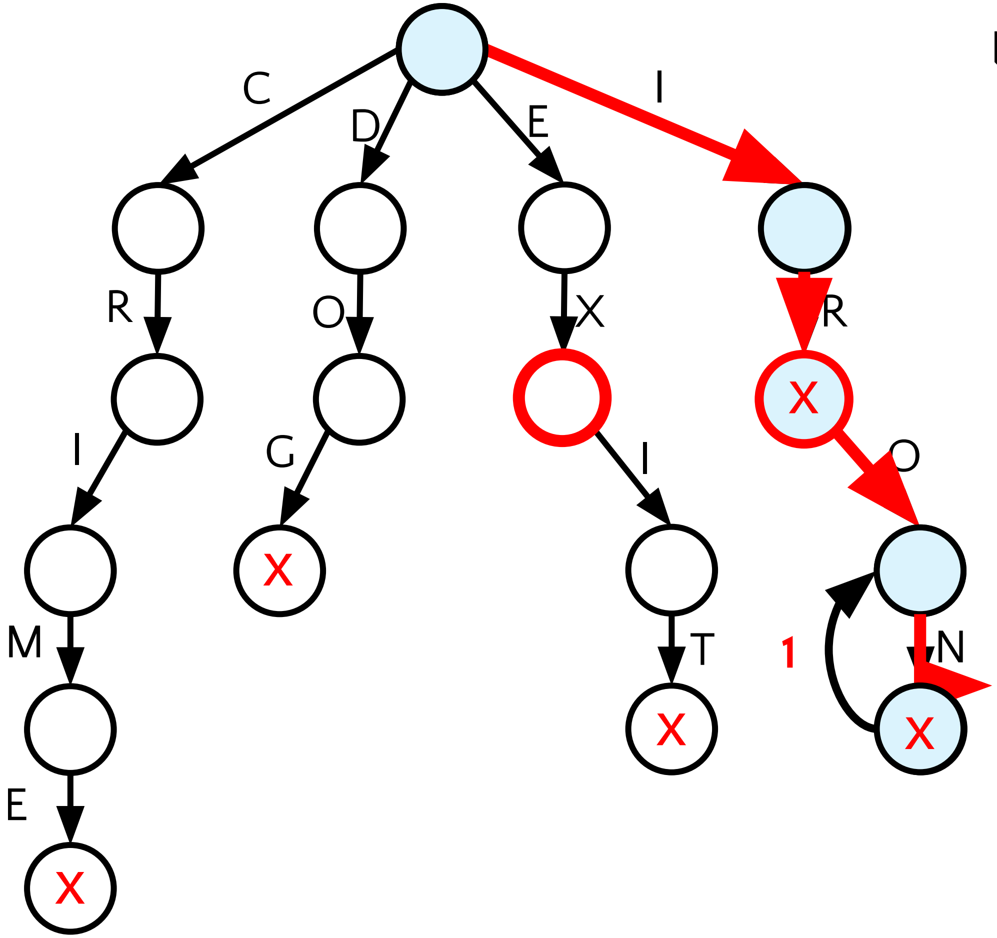
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete(“EXAM”)

DELETE FROM TRIE (CASE-2)



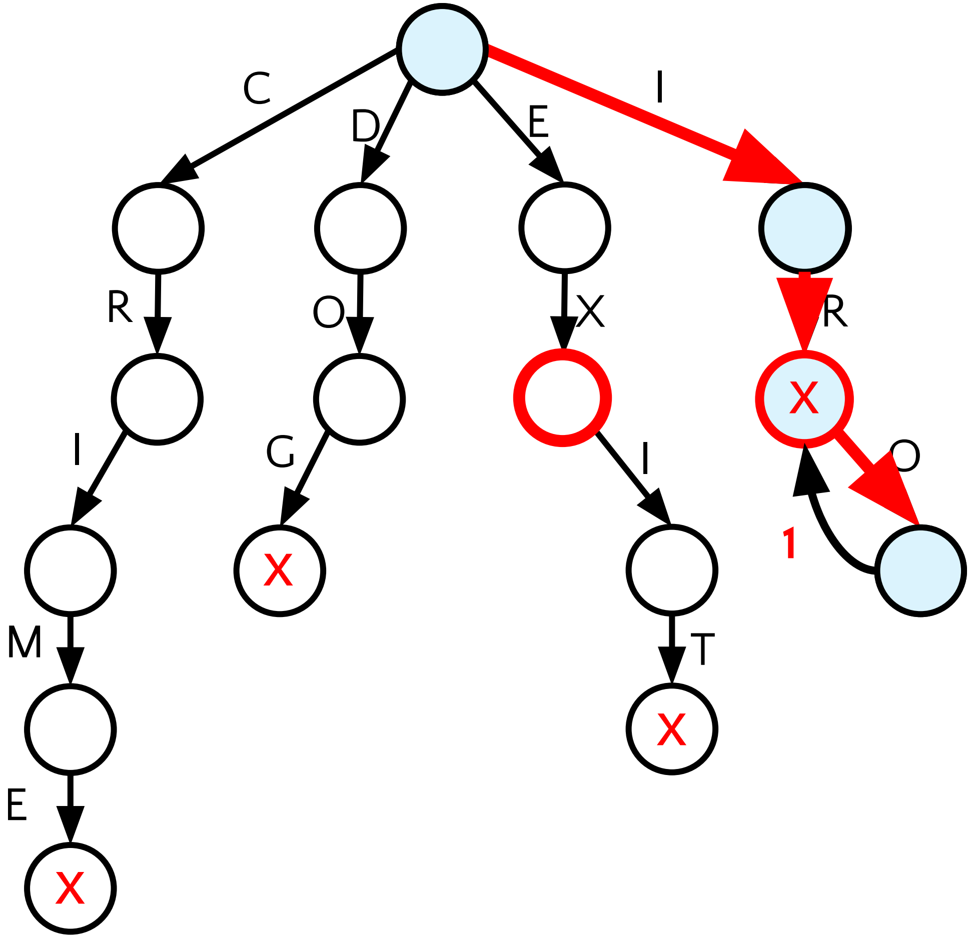
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



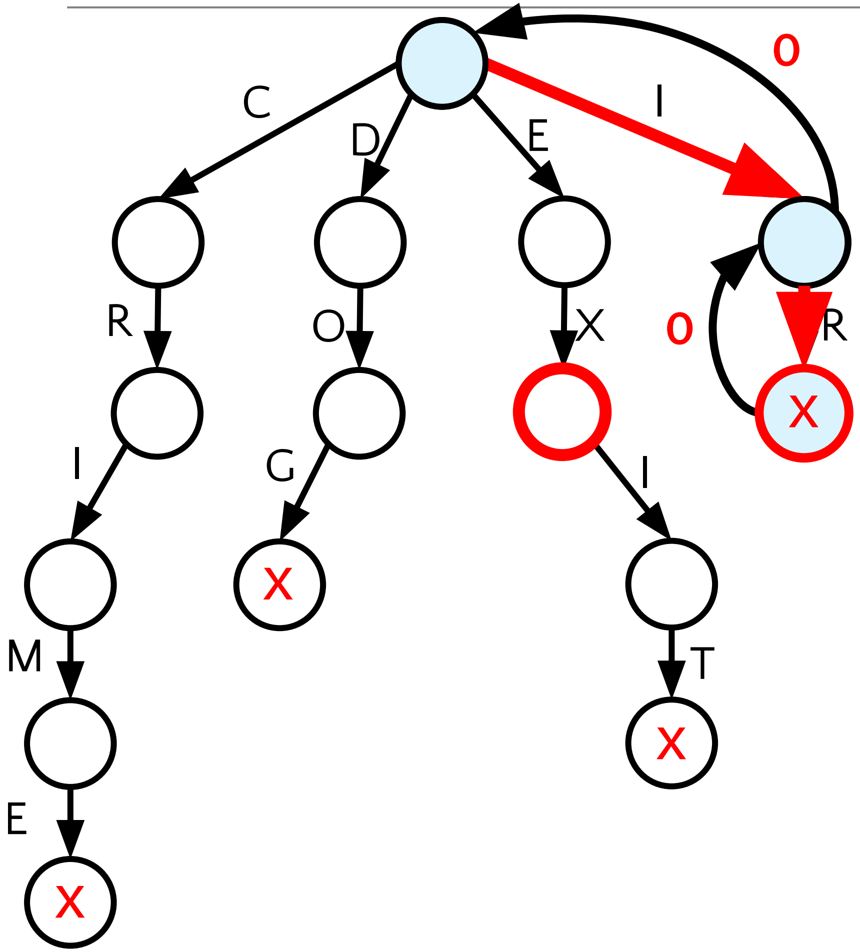
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



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

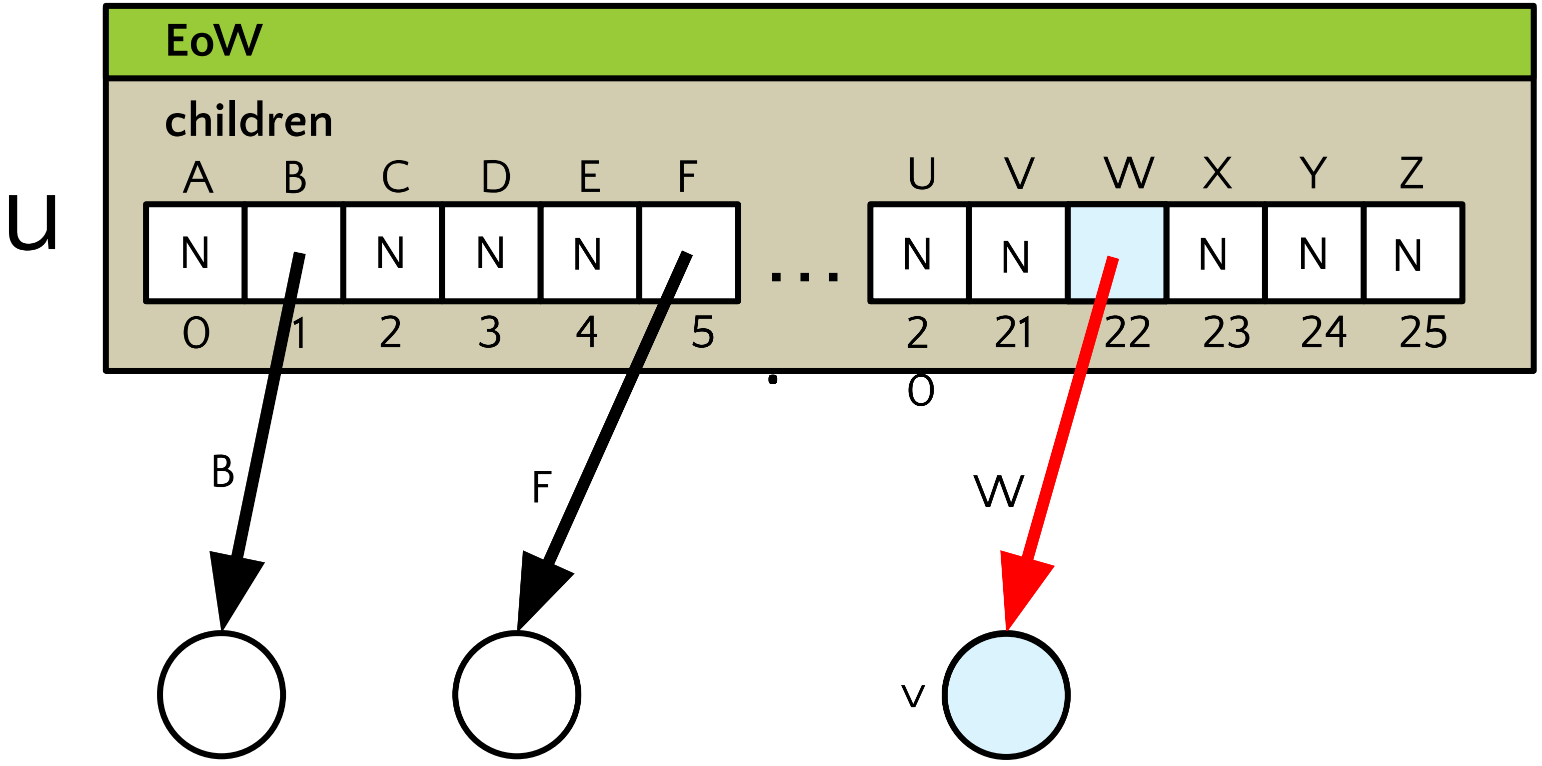
DELETE FROM TRIE (CASE-2)



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

  - delete("EXAM")
  - delete("IRON")

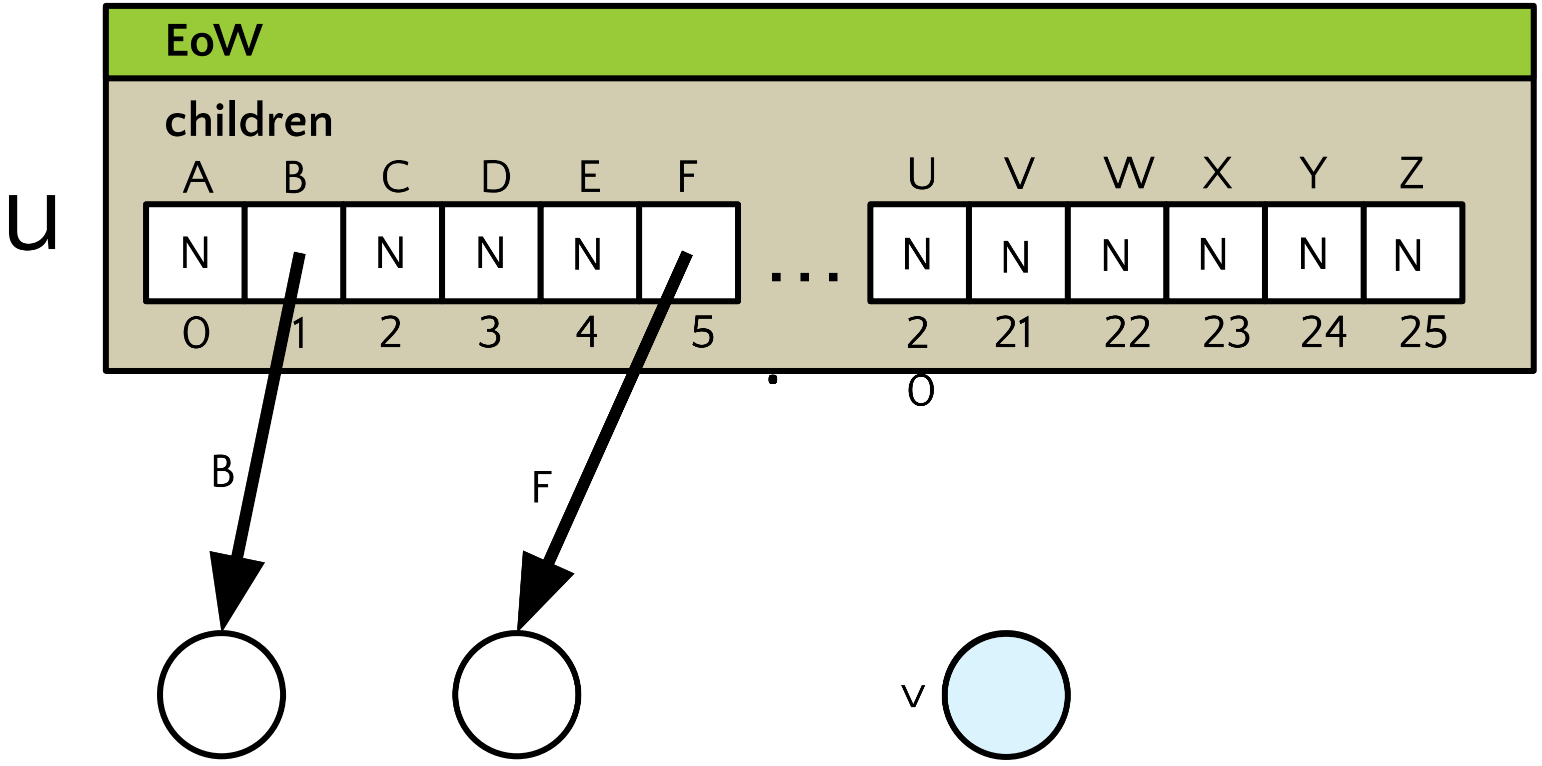
DELETION OF AN EDGE




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

DELETE THE RED MARKED EDGE

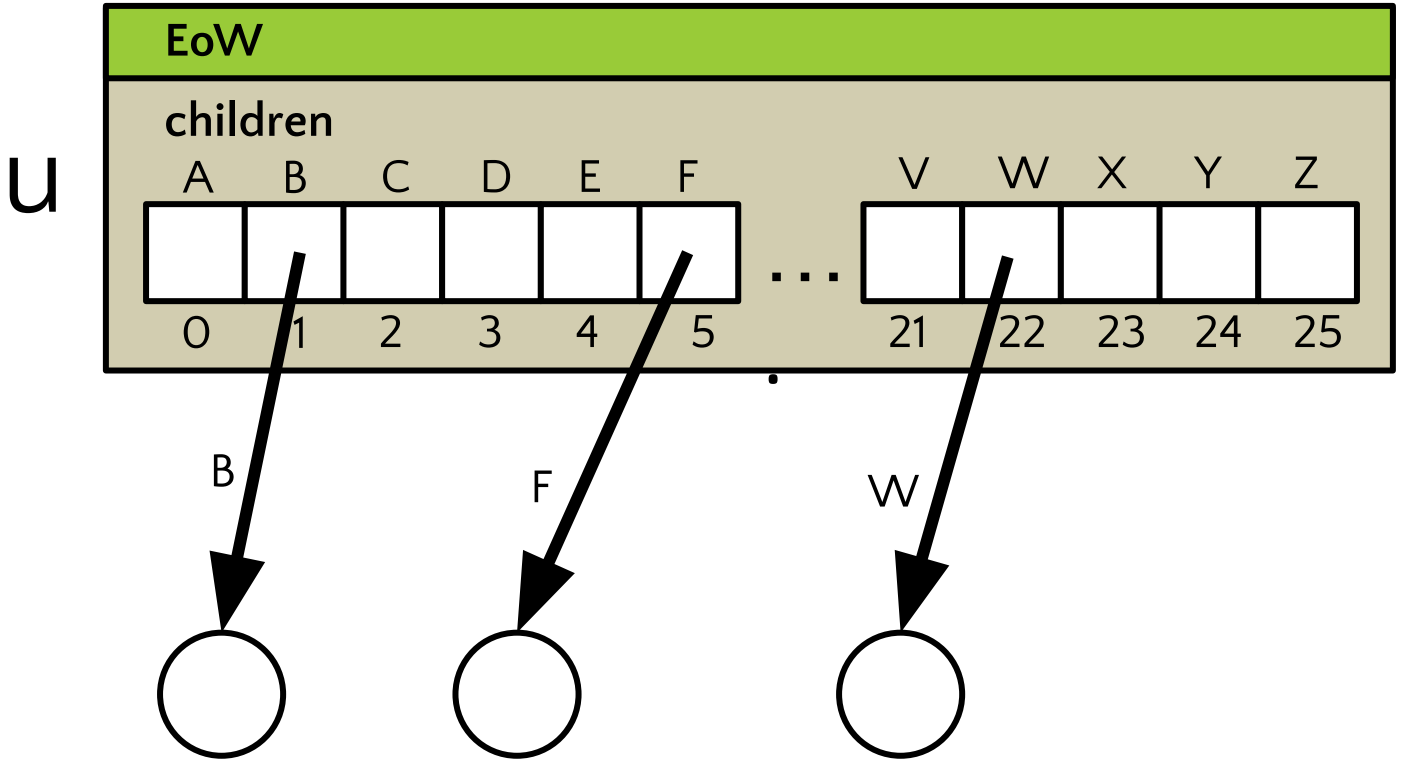
DELETION OF AN EDGE




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

DELETE THE RED MARKED EDGE

DELETION OF AN EDGE




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

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

DELETE IN TRIE

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

Traversing of x is not complete

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

d becomes 1 if the next node is removable

Otherwise d becomes 0

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

Removes the k-th edge of u if d permits

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

Then sends the permission to it's parent

JUNCTION POINT

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

```
bool isJunction(Node *u)

   count<-0
   for(int i=0;i<26;i++)
        if(u->children[i]!=NULL)    count++;

   if(u->EoW>0 or count>1)    return true;

   return false;
```

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

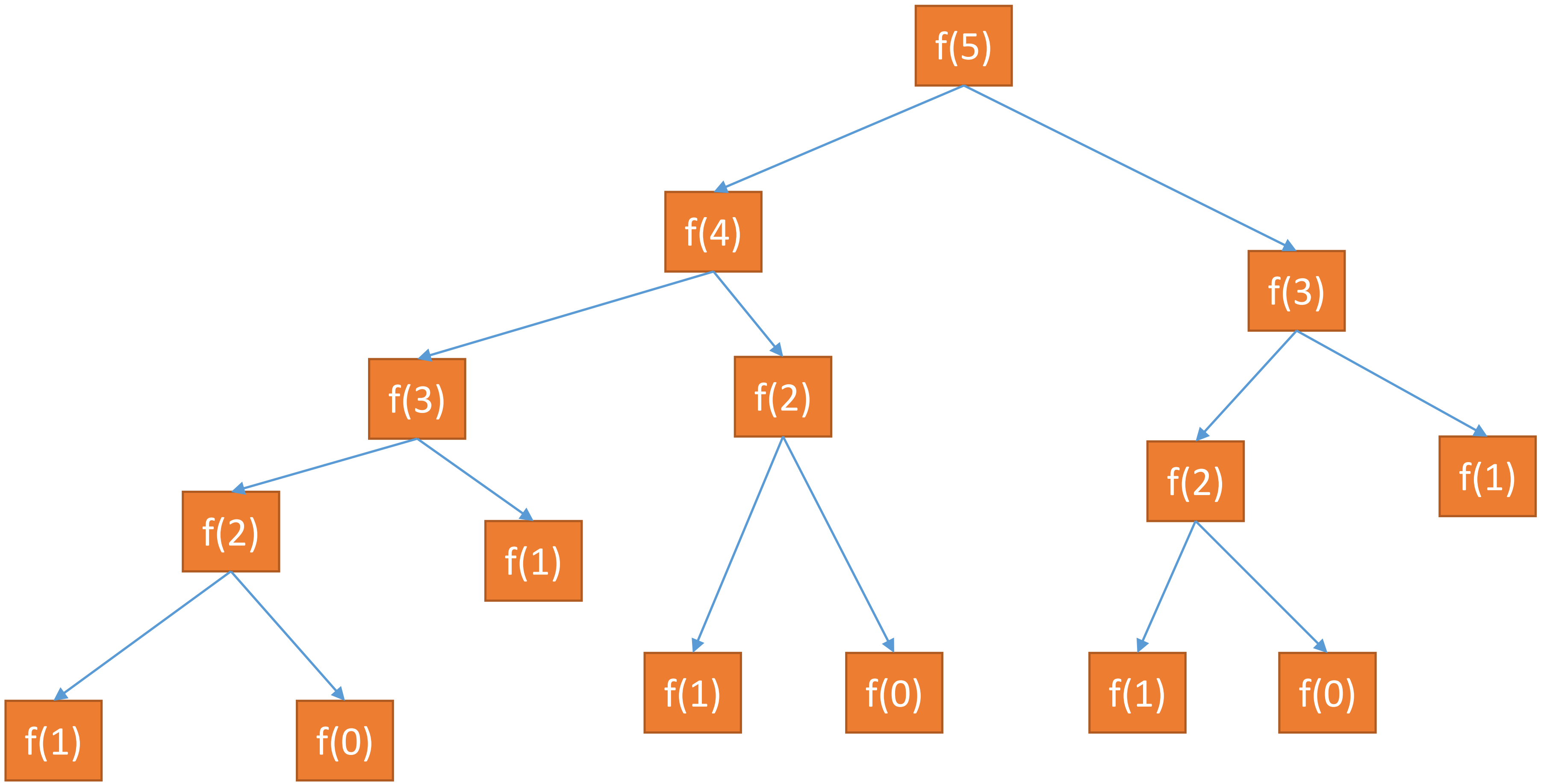
- Overlapping sub problems occur when a problem can be broken down into sub problems which are reused several times.
- This means that the same sub problem is solved multiple times in the process of solving the larger problem. Dynamic programming takes advantage of this property by solving each sub problem 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:

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

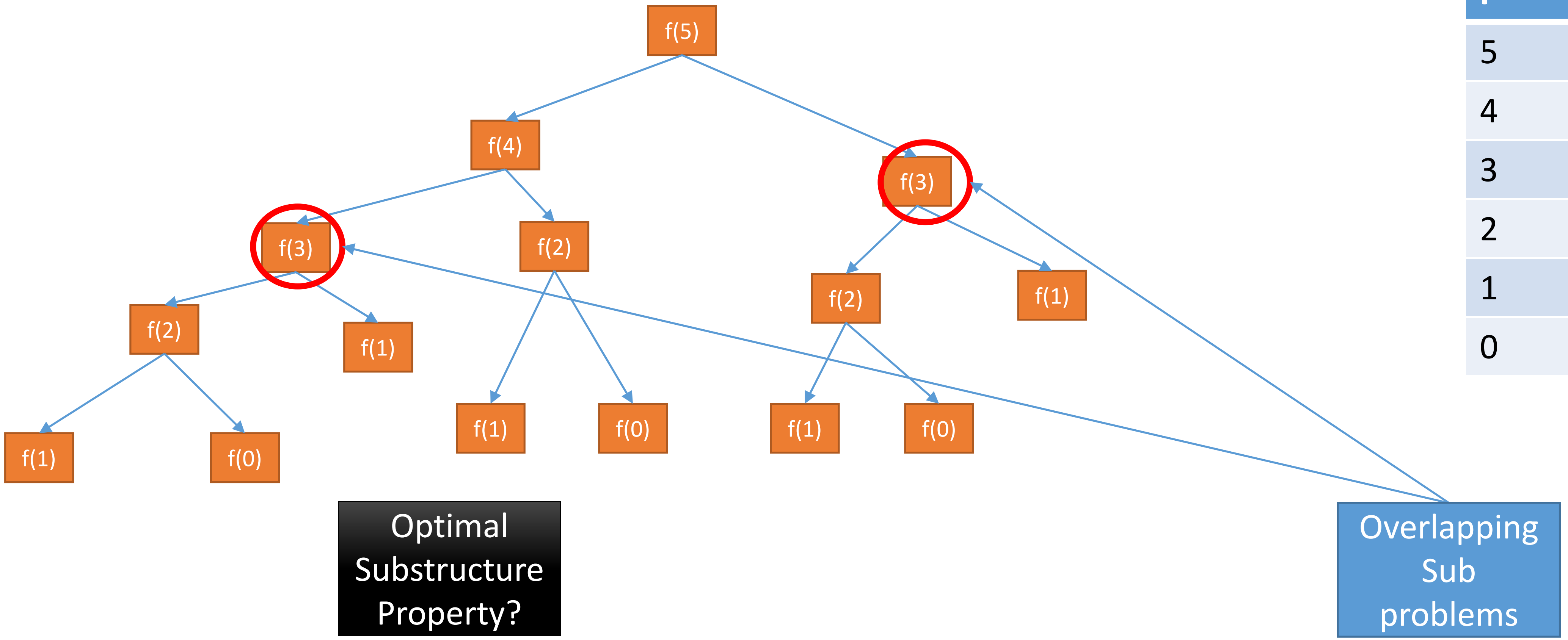
Remember the Fibonacci Series?

A basic intuition about Dynamic Programming

Fibonacci Series

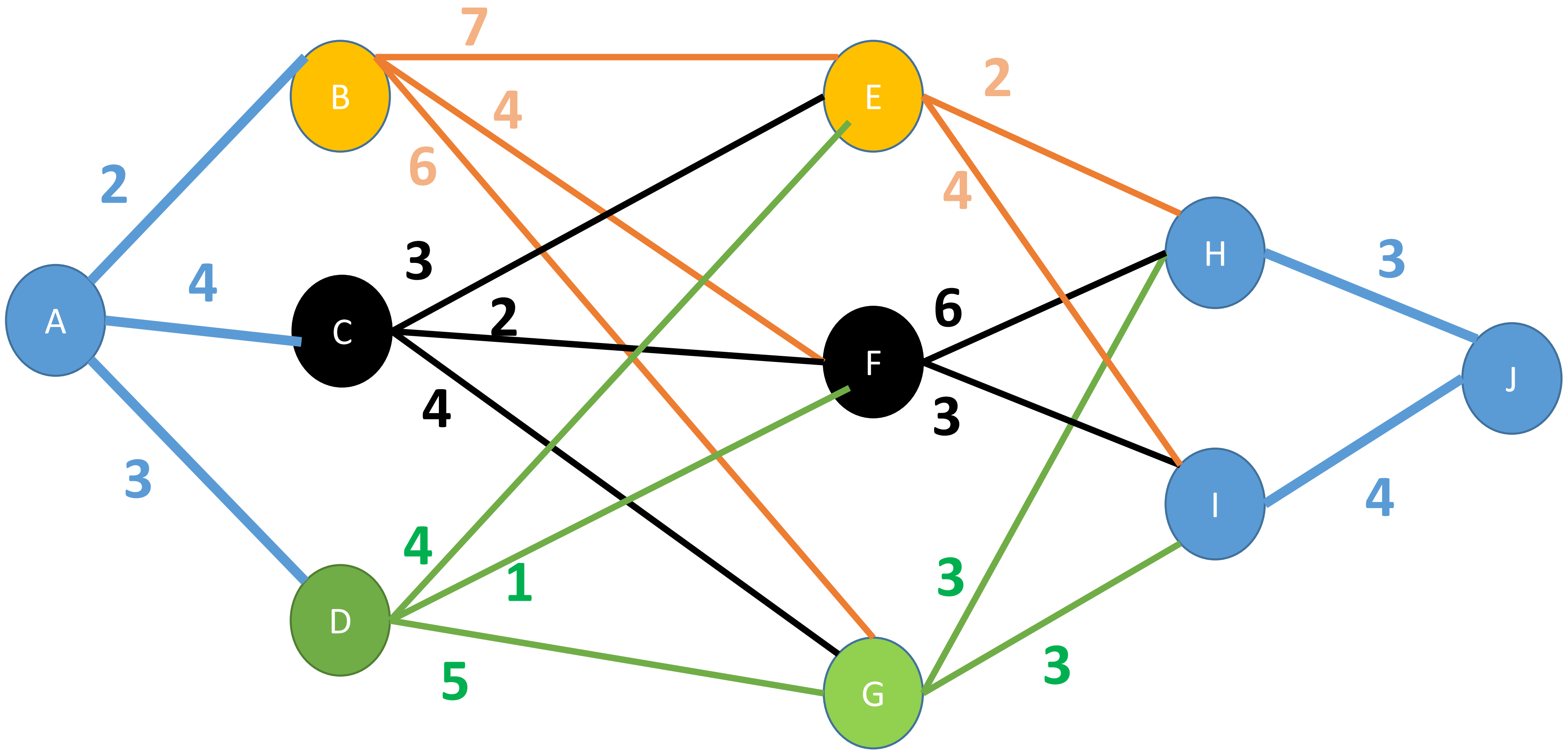


Fibonacci Series with an extra table

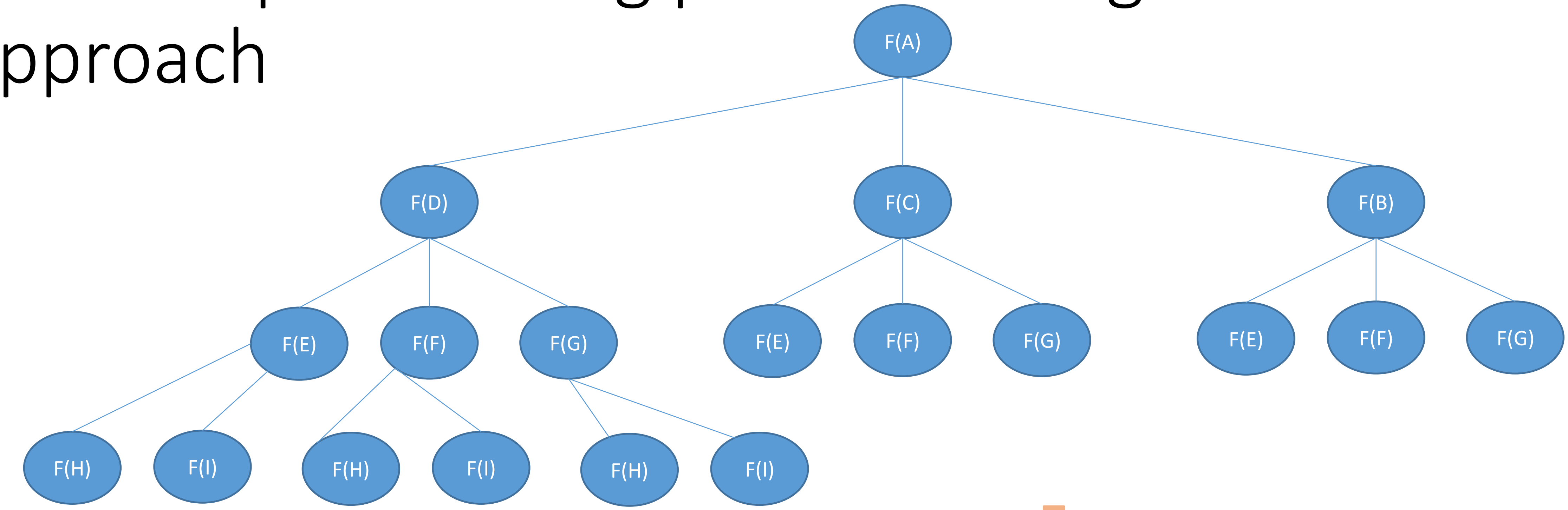


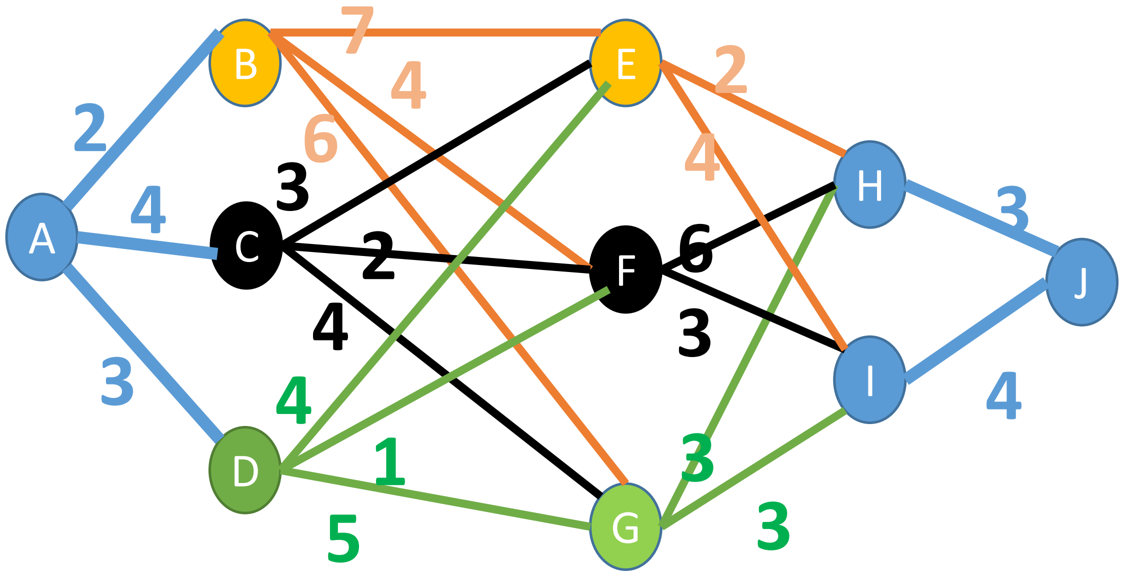
F	value
5	-1
4	-1
3	-1
2	-1
1	1
0	1

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

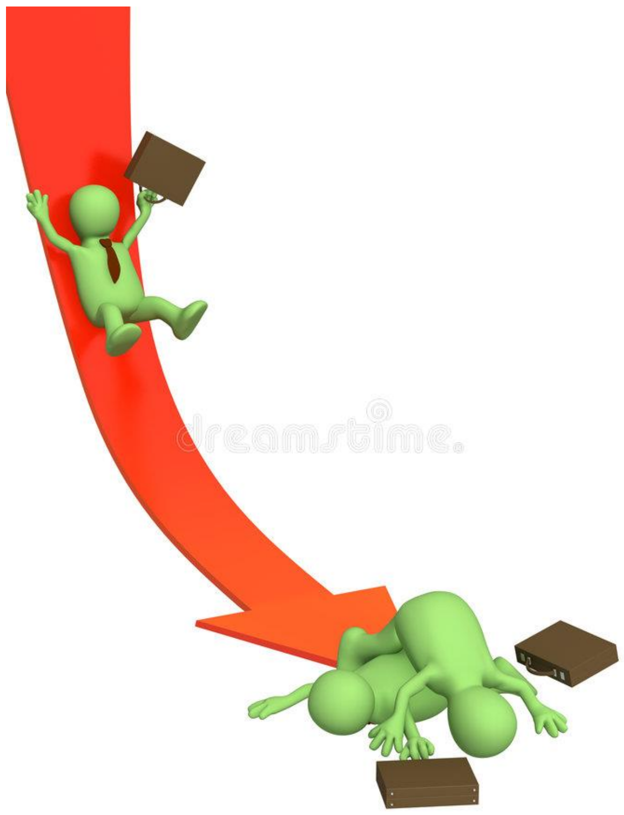
Dynamic Programming and Optimization Problems

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

Problem Formulation

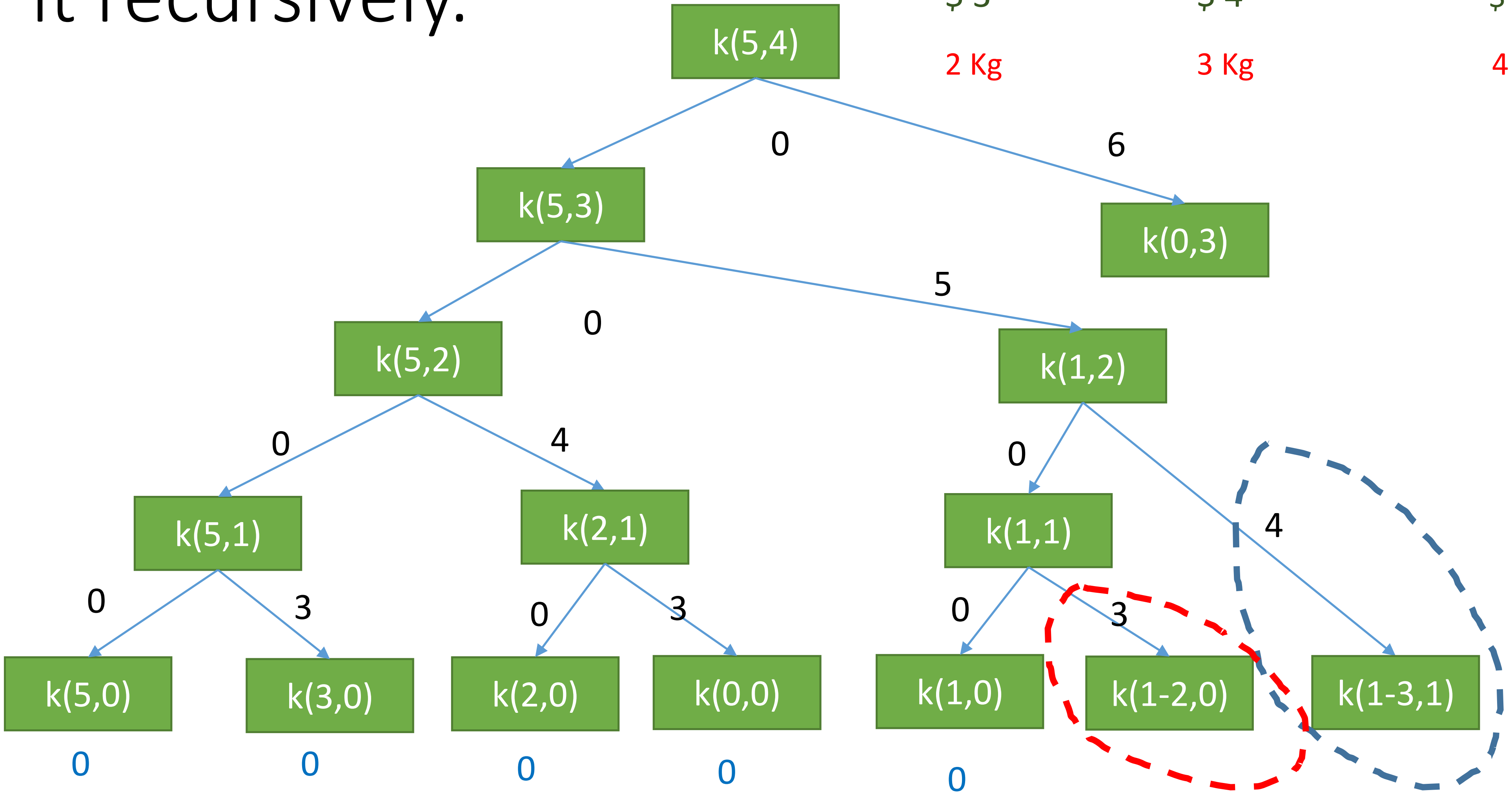
- **How much maximum amount** can the thief with a bag with 5Kg of capacity can steal from the 4 items?

- K (5 kg, 4 items)

- K (w, n)

Yes, we can solve it recursively.

$ 3	$ 4	$ 5	$ 6
2 Kg	3 Kg	4 Kg	5 Kg



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

$ 4

2 Kg

0

$ 2

1 Kg

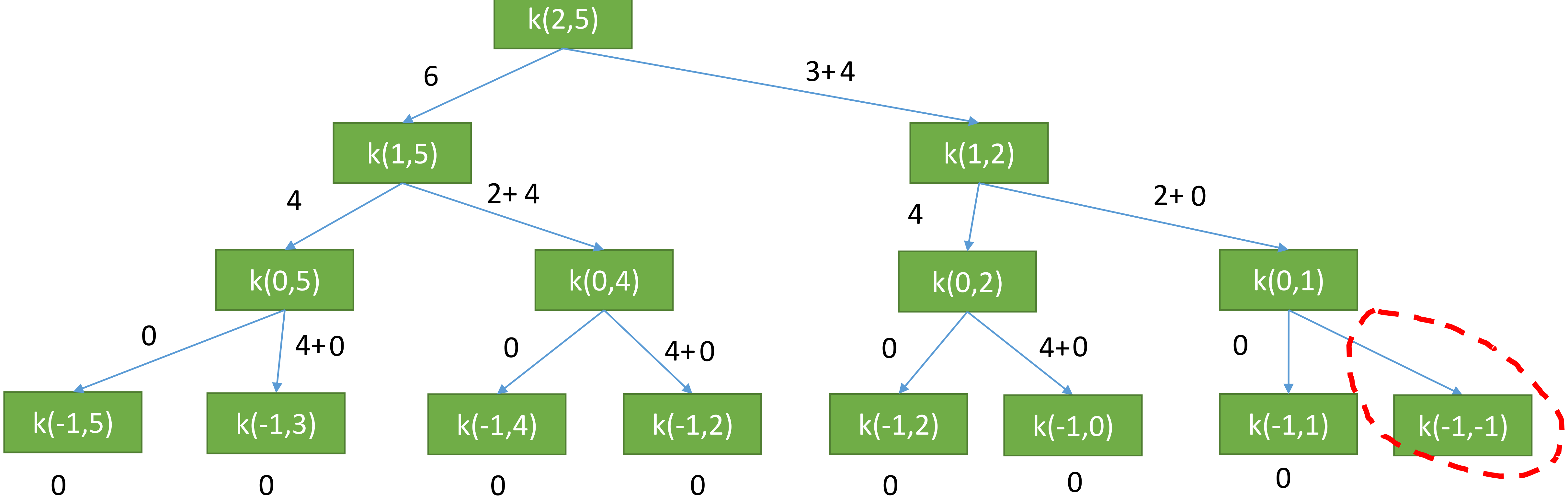
1

$ 3

3 Kg

2

Knapsack = 5kg
Number of item = 3



Let us formulate this solution using dynamic programming

The recursive equation:

- K(w, i) = max { k(w , i-1), prices[i]+k(w - 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
  - prices = array containing the prices of the items we are thinking about stealing
  - 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

w \ n	0	1	2	3	4
0					
1					
2					
3					
4					
5					

Bottom Up Approach

prices	12	10	20	15
weights	2	1	3	2

- 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;
}
```

w \ n	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

Bottom Up Approach

prices	12	10	20	15
weights	2	1	3	2

- 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], prices[i]+ m[w - weights[i]][i-1] }
- Of course we have to check if

  w-weights[i]>=0

w \ n	0	1	2	3	4
0	0	0	0	0	0
1	0	0	10	10	10
2	0	12	12	12	15
3	0	12	22	22	25
4	0	12	22	30	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

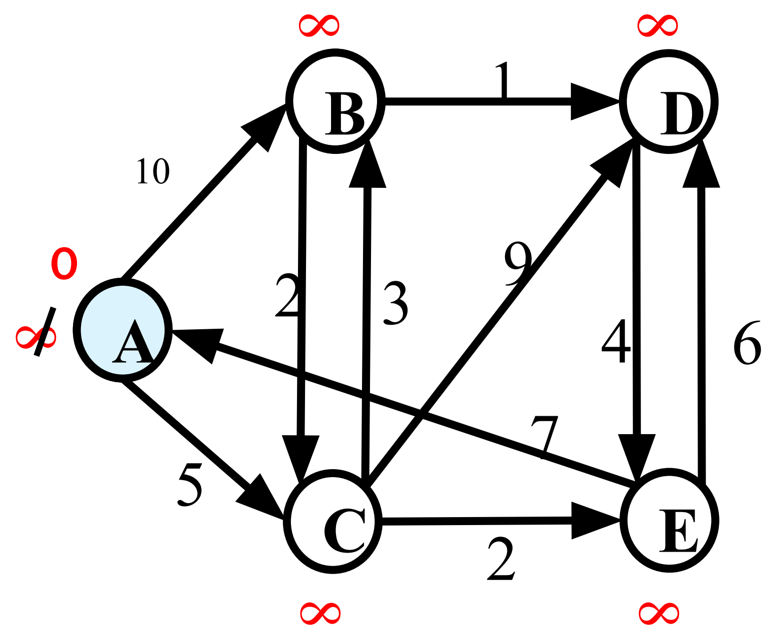
Single Source Shortest Paths

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

Shortest Path Concepts

INITIALIZE-SINGLE-SOURCE(G, s)

```
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```



Shortest Path Concepts

Relaxation:

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

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
  $v.d = u.d + w(u, v)$
  $v.\pi = u$

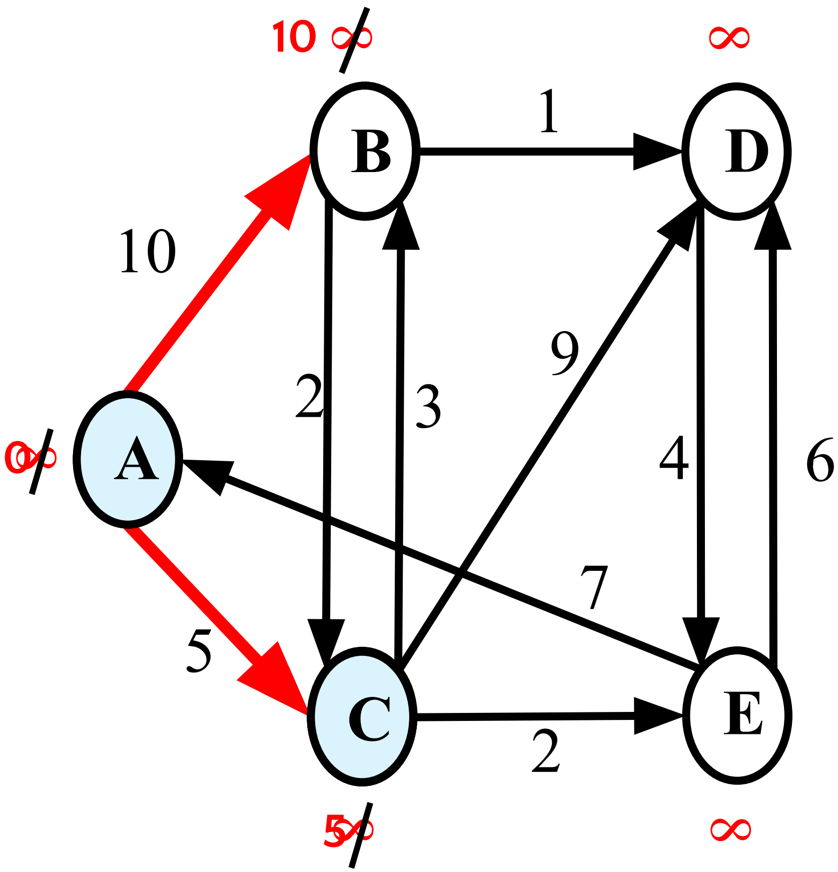
Dijkstra Algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

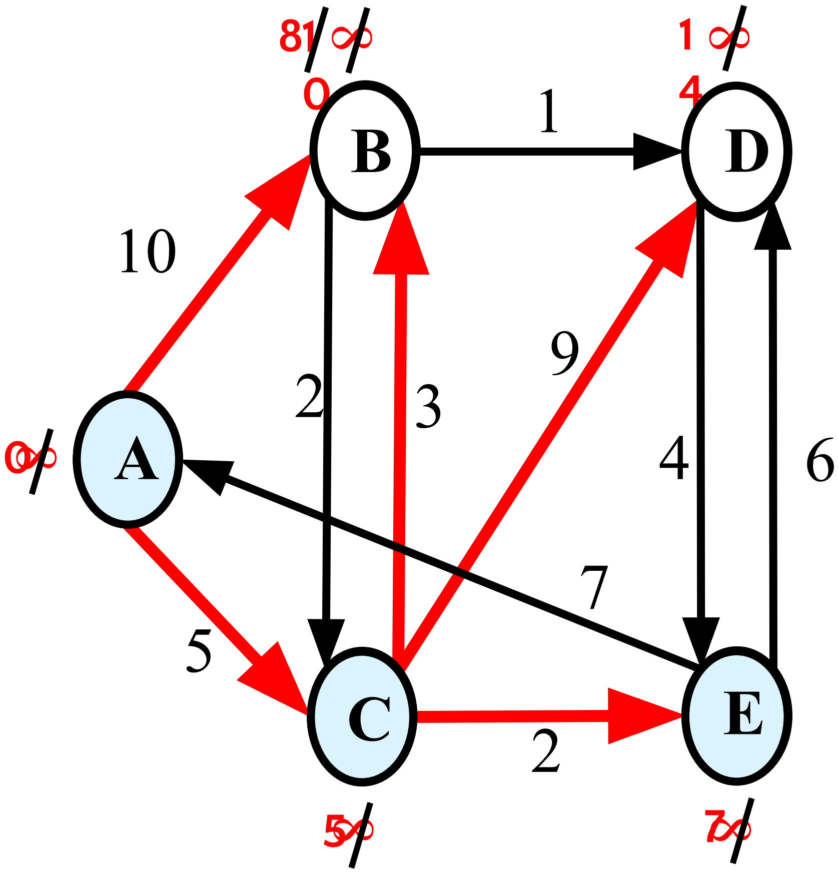
Q : Min Priority Queue with priority value based on d.

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

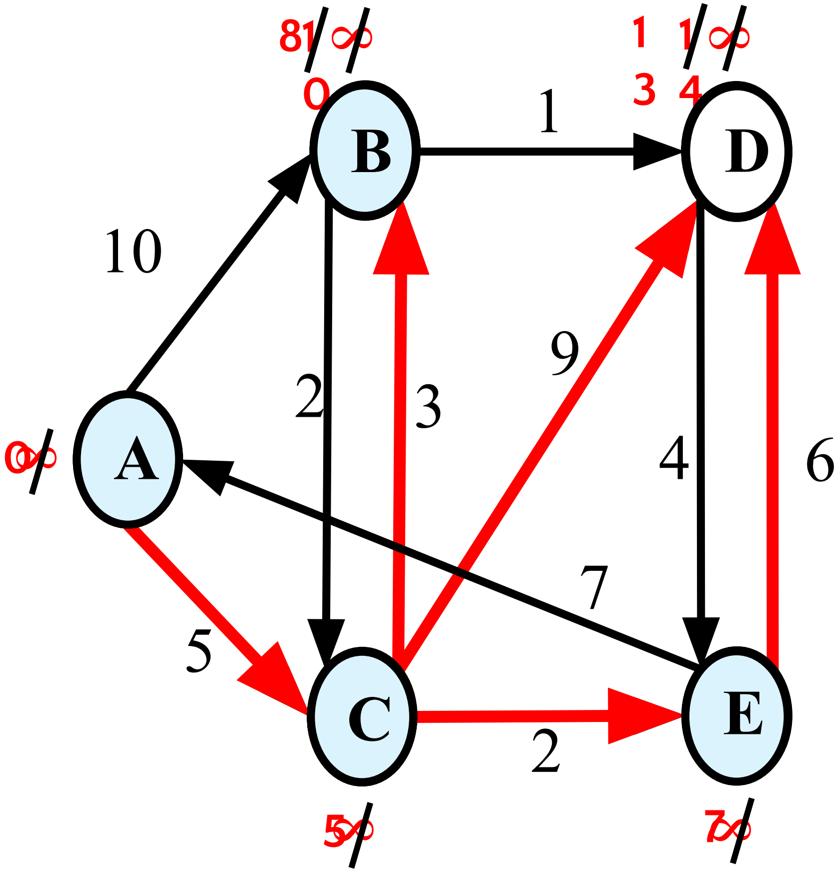
DIJSKTRA's SIMULATION



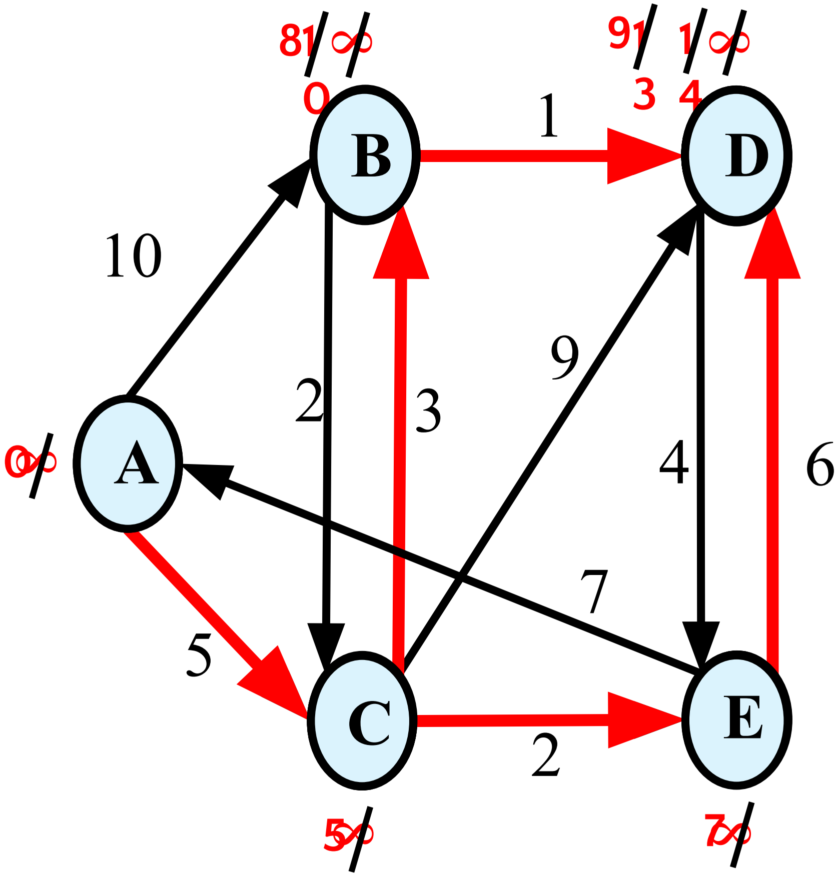
DIJSKTRA's SIMULATION



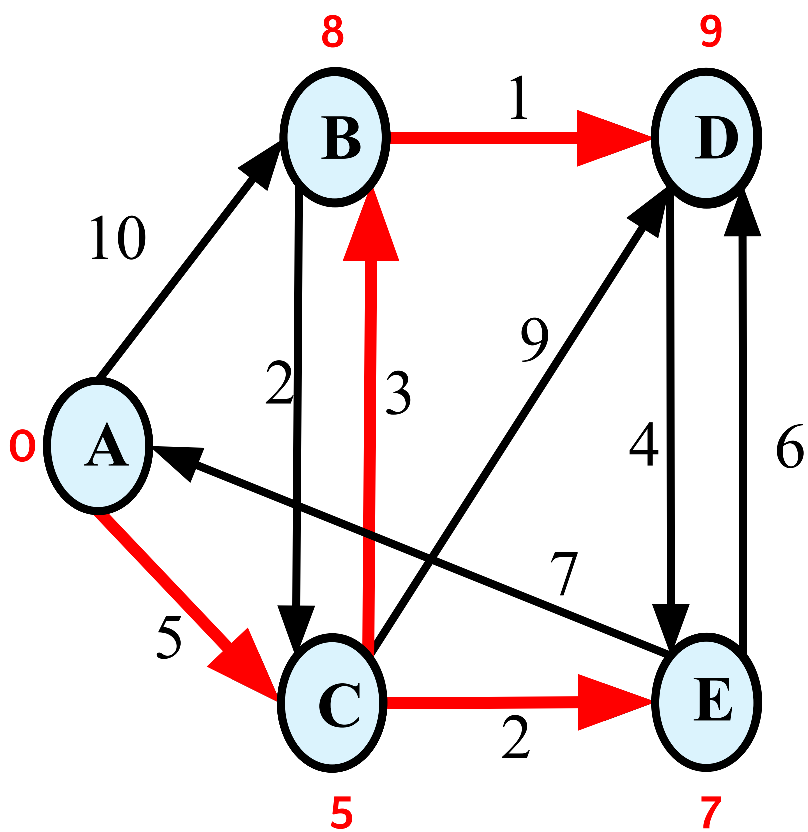
DIJSKTRA's SIMULATION



DIJSKTRA's SIMULATION

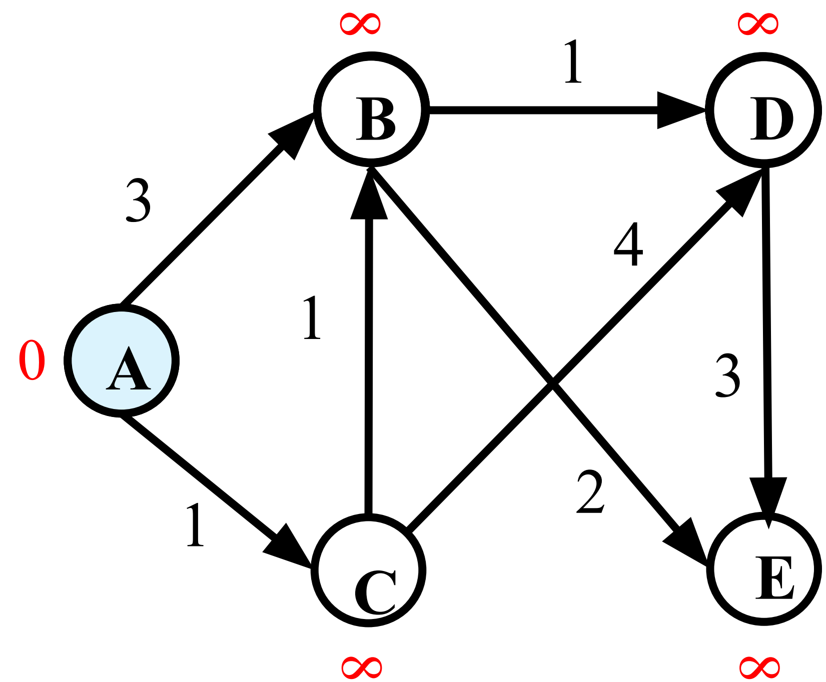


DIJSKTRA's SIMULATION



Bellman Ford Algorithm

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



Bellman Ford Algorithm

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

Finds the shortest path and distance from source

Detects Negative Weighted Cycle

Thank you

CSE 215: Data Structures & Algorithms II

Dynamic Programming : LCS

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

Dynamic Programming

Dynamic programming :

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

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

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

Dynamic Programming

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

- Optimal substructure property.
- Overlapping subproblems property.

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

Dynamic Programming

Optimal substructure property.

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

Overlapping subproblems property.

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

Dynamic Programming

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

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

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

Dynamic Programming

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

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

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

Dynamic Programming

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

1. Characterize the structure of an optimal solution.

1. Recursively define the value of an optimal solution.

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

1. Construct an optimal solution from computed information.

Longest common subsequence (LCS)

LCS:

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

Let's say, X = ABCD Y= JBAGHCED

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

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

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

Longest common subsequence (LCS)

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

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

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

Longest common subsequence (LCS)

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

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

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

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

Longest common subsequence (LCS)

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

The LCS problem has an optimal-substructure property.

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

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

Proof : Theorem 15.1 (3rd Edition)
Self Study

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

Longest common subsequence (LCS)

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

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

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

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

Longest common subsequence (LCS)

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

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

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

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS

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

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

Longest common subsequence (LCS)

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

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

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

X = **ABCBDAB**
Y= **BDCABA**
LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

O(mn)

Longest common subsequence (LCS)

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

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

LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
        c[i, 0] = 0
for j = 0 to n
        c[0, j] = 0
for i = 1 to m
        for j = 1 to n
                if x_i == y_j
                        c[i, j] = c[i − 1, j − 1] + 1
                        b[i, j] = "↖"
                elseif c[i − 1, j] ≥ c[i, j − 1]
                        c[i, j] = c[i − 1, j]
                        b[i, j] = "↑"
                else c[i, j] = c[i, j − 1]
                        b[i, j] = "←"
return c and b
```

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

Longest common subsequence (LCS)

Step 4: Constructing an Optimal Solution / Constructing an LCS

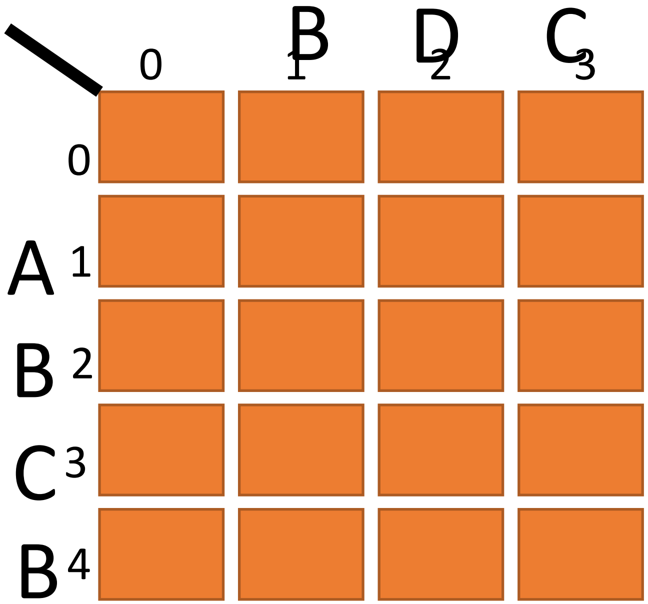
```
PRINT-LCS(b, X, i, j)
    if i == 0 or j == 0
        return
    if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
    elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
    else PRINT-LCS(b, X, i, j − 1)
```

O(m+n)

Longest common subsequence (LCS)

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

Recursive code + Memoization code



X= ABCB
Y=BDC

Memo

Longest common subsequence (LCS)

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

		0	B 1	D 2	C 3
	0	-1	-1	-1	-1
A	1	-1	-1	-1	-1
B	2	-1	-1	-1	-1
C	3	-1	-1	-1	-1
B	4	-1	-1	-1	-1

X= ABCB
Y=BDC

Longest common subsequence (LCS)

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

	0	B 1	D 2	C 3
0	-1	-1	-1	-1
A 1	-1	-1	-1	-1
B 2	-1	-1	-1	-1
C 3	-1	-1	-1	-1
B 4	-1	-1	-1	-1

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

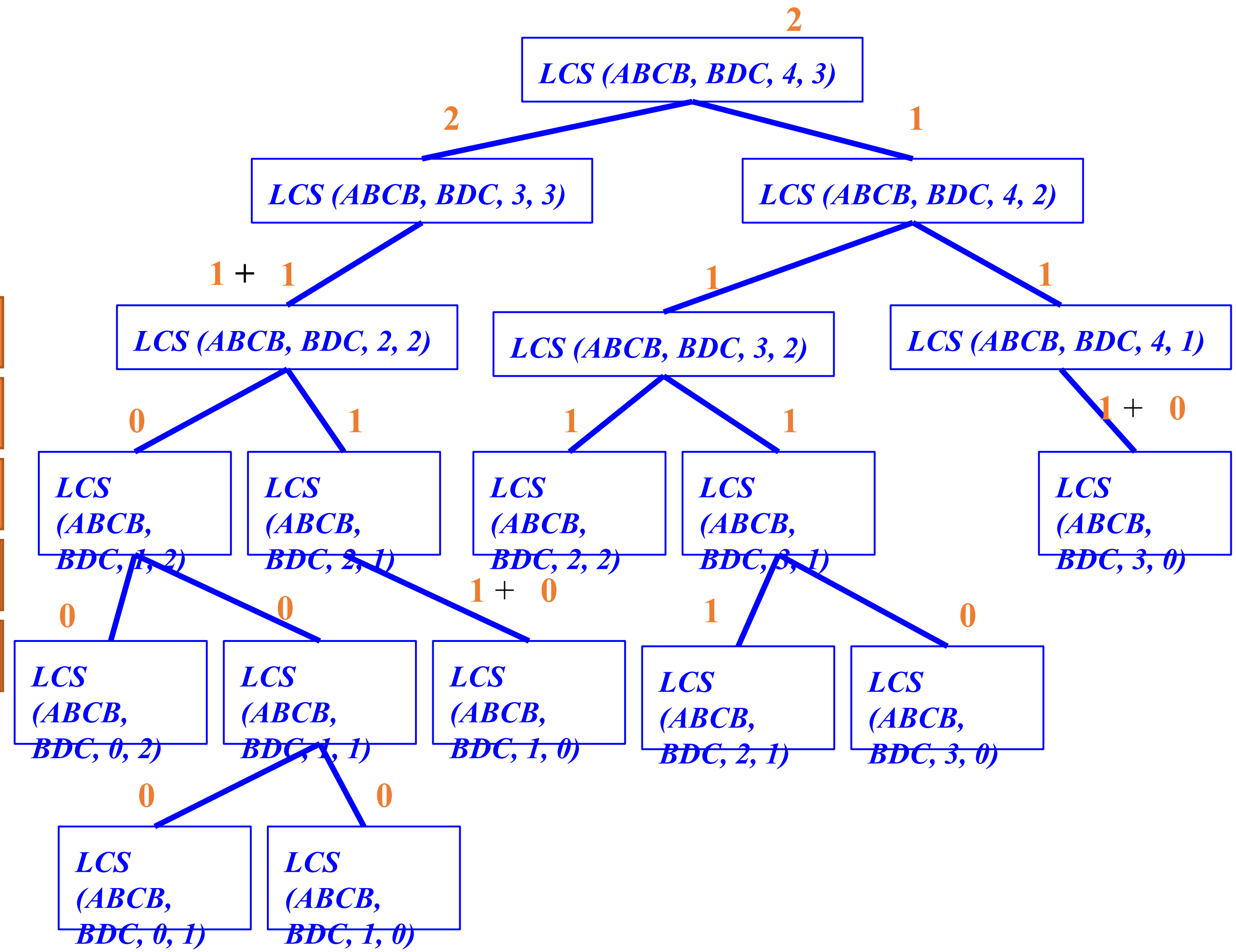
X= ABCB , Y=BDC

LCS (X, Y, 4, 3)

Longest common subsequence (LCS)

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

			B	D	C
		0	1	2	3
	0	-1	0	0	-1
A	1	0	0	0	-1
B	2	-1	1	1	-1
C	3	0	1	1	2
B	4	-1	1	1	2



Longest common subsequence (LCS)

Time Complexity of LCS:

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

Bottom-Up Approach : O(mn)

Top-Down Approach : O(mn)

Brute Force Approach : Exponential time

Longest common subsequence (LCS)

Exercises:

15.4-2

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

15.4-3

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


Thank You