Complete Binary Trees

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- Introducing complete binary trees
    - Background
    - Definitions
    - Examples
    - Logarithmic height
    - Array storage

Background

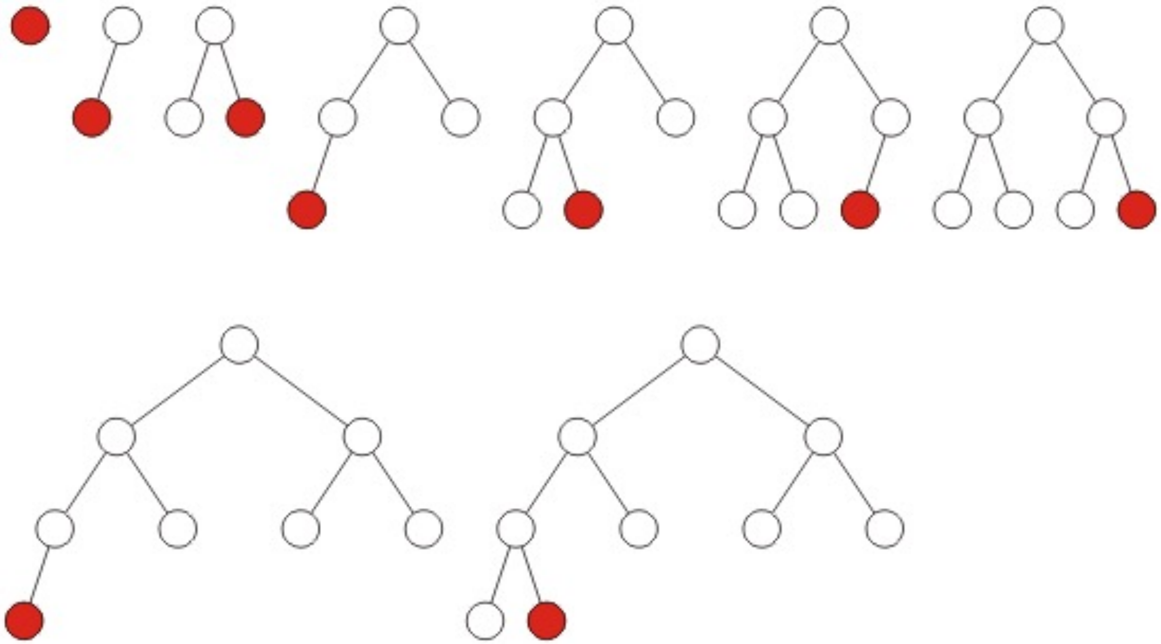
- A perfect binary tree has ideal properties but restricted in the number of nodes: $n = 2^{h+1} - 1$ for $h = 0, 1, \ldots$

  1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ...

- We require binary trees which are
  - Similar to perfect binary trees, but
  - Defined for all n

Definition

- A complete binary tree filled at each depth from left to right:
  - The order is identical to that of a breadth-first traversal



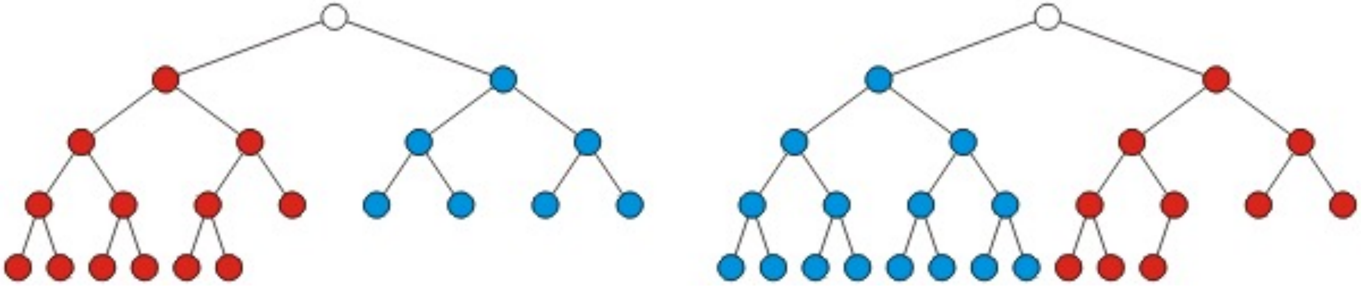
Recursive Definition

- ☐ Recursive definition:

  i) a binary tree with a single node is a complete binary tree of height $h = 0$

  ii) a complete binary tree of height h is a tree where either:

  - The left sub-tree is a complete tree of height $h - 1$ and the right sub-tree is a perfect tree of height $h - 2$, or
  - The left sub-tree is perfect tree with height $h - 1$ and the right sub-tree is complete tree with height $h - 1$



Height

- Theorem

  The height of a complete binary tree with n nodes is $h = \lfloor \lg(n) \rfloor$

  Proof:

  - Base case:
    - When $n = 1$ then $\lfloor \lg(1) \rfloor = 0$ and a tree with one node is a complete tree with height $h = 0$
  - Inductive step:
    - Assume that a complete tree with n nodes has height $\lfloor \lg(n) \rfloor$
    - Must show that $\lfloor \lg(n + 1) \rfloor$ gives the height of a complete tree with $n + 1$ nodes
    - Two cases:
      - ✓ If the tree with n nodes is perfect, and
      - ✓ If the tree with n nodes is complete but not perfect

Height

- Case 1 (the tree is perfect):
  - If it is a perfect tree then
    - Adding one more node must increase the height
  - Before the insertion, it had $n = 2^{h+1} - 1$ nodes:

$$2^h < 2^{h+1} - 1 < 2^{h+1}$$

$$h = \lg\left(2^h\right) < \lg\left(2^{h+1} - 1\right) < \lg\left(2^{h+1}\right) = h + 1$$

$$\underbrace{h \le \left\lfloor \lg\left(2^{h+1} - 1\right) \right\rfloor}_{} < h + 1$$

  - Thus, $\left\lfloor \lg(n) \right\rfloor = h$
  - However, $\left\lfloor \lg(n+1) \right\rfloor = \left\lfloor \lg\left(2^{h+1} - 1 + 1\right) \right\rfloor = \left\lfloor \lg\left(2^{h+1}\right) \right\rfloor = h + 1$

Correct for a perfect tree

Height

- Case 2 (the tree is complete but not perfect):
  - If it is not a perfect tree of height *h* then

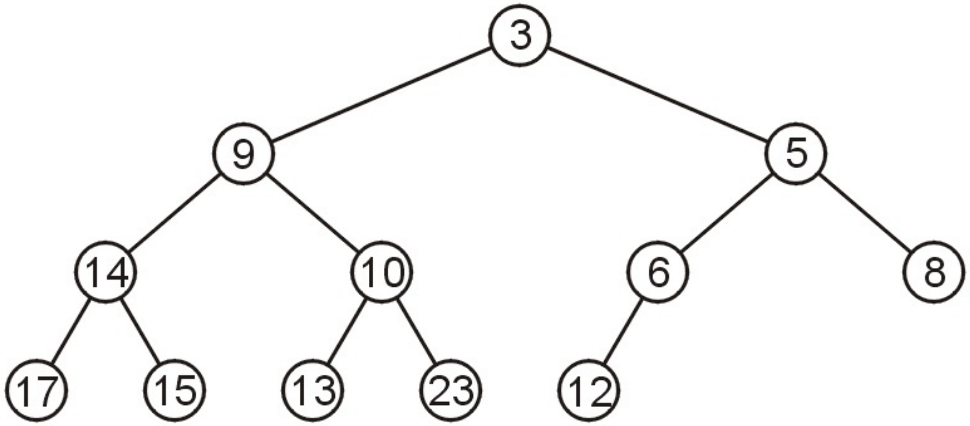
$$\begin{aligned} 2^h \le \quad & n & < 2^{h+1} - 1 \\ 2^h + 1 \le \quad & n+1 & < 2^{h+1} \\ h = \lg\left(2^h\right) < \lg\left(2^h+1\right) \le \quad & \lg\left(n+1\right) & < \lg\left(2^{h+1}\right) = h+1 \\ h \le \left\lfloor \lg\left(2^h+1\right)\right\rfloor \le & \left\lfloor \lg\left(n+1\right)\right\rfloor & < h+1 \end{aligned}$$

  - Consequently, the height is unchanged: $\lfloor \lg(n + 1)\rfloor = h$

- By mathematical induction, the statement must be true for all $n \ge 1$

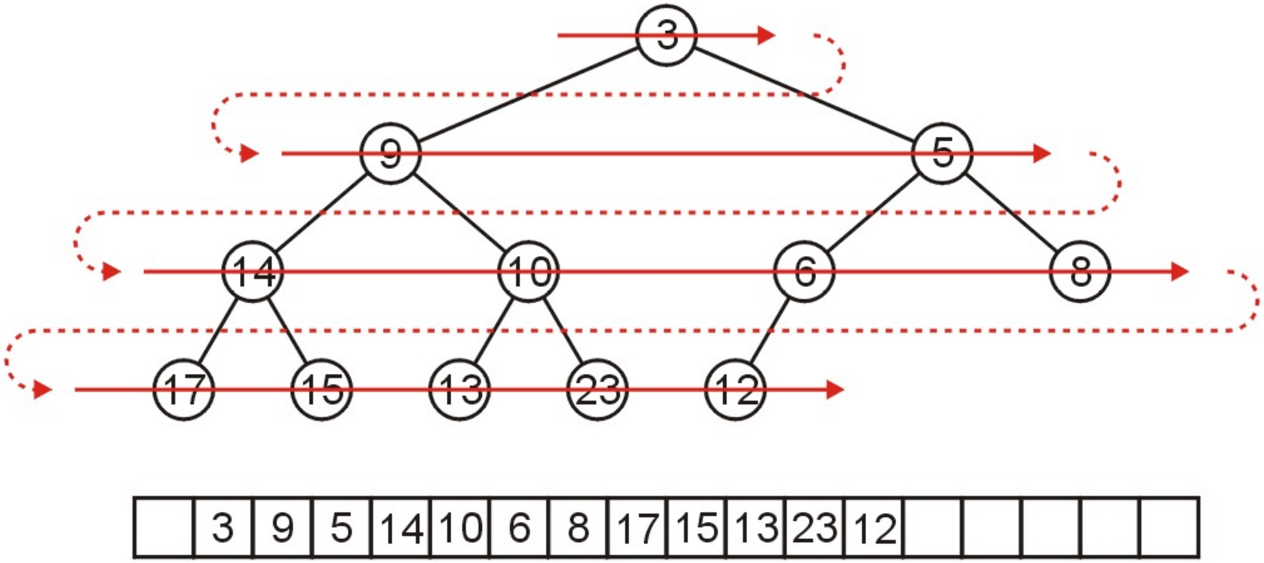
Array storage

- We are able to store a complete tree as an array
  - Traverse the tree in breadth-first order, placing the entries into the array
  - What if it is not a complete tree?



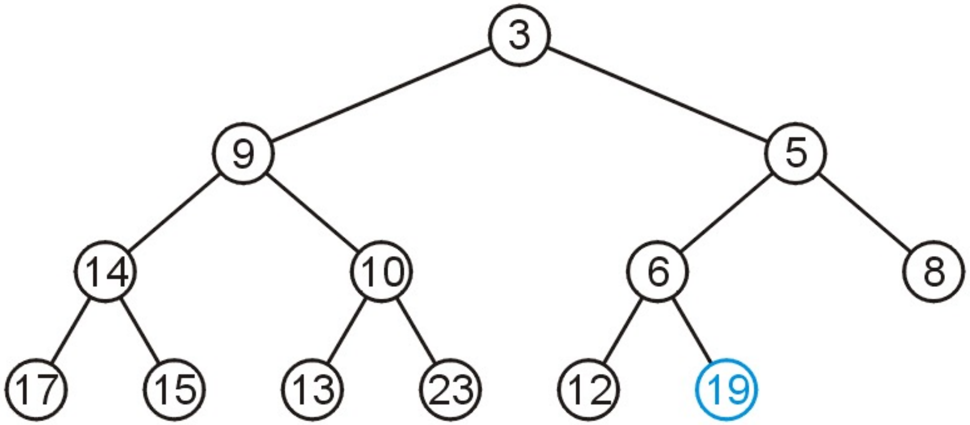
Array storage

- □ We can store this in an array after a quick traversal:



Array storage

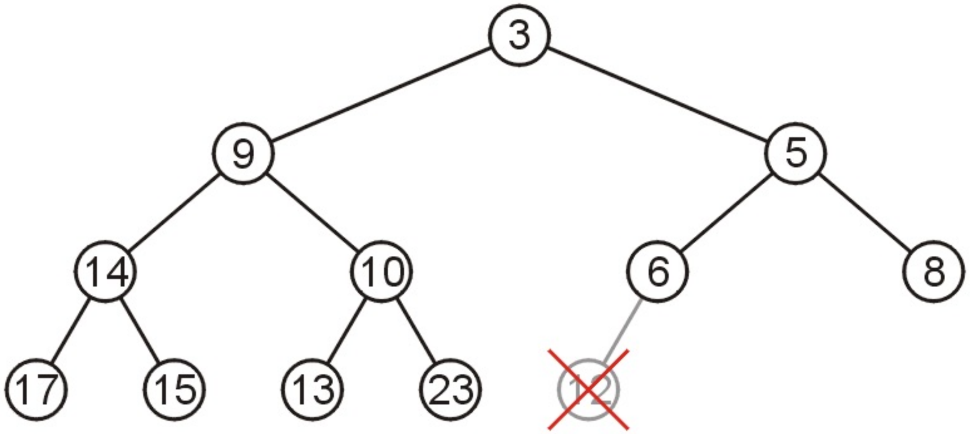
- To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location



	3	9	5	14	10	6	8	17	15	13	23	12	19			

Array storage

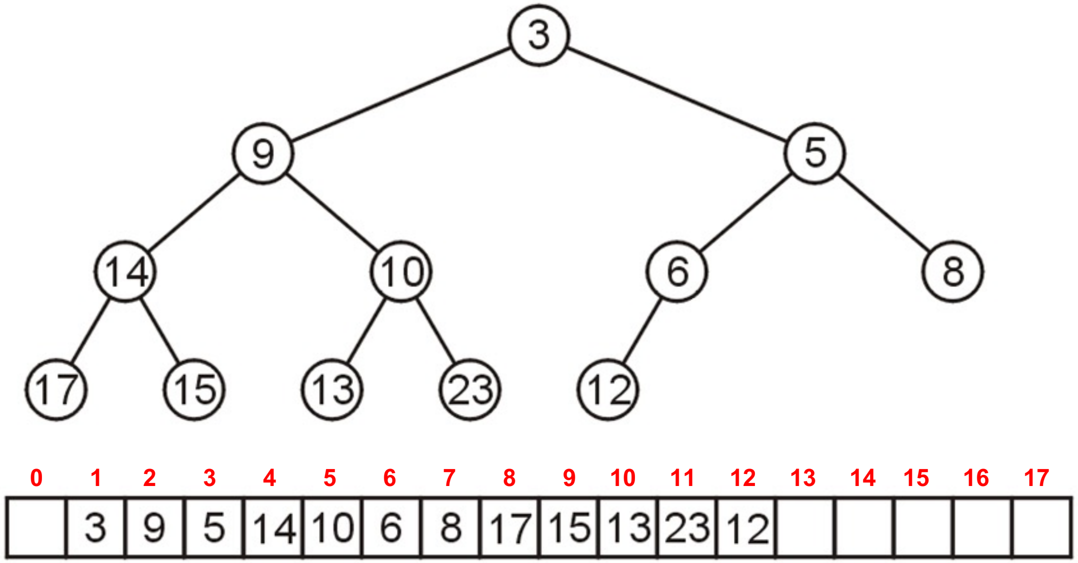
- To remove a node while keeping the complete-tree structure, we must remove the last element in the array



	3	9	5	14	10	6	8	17	15	13	23	~~12~~					

Array storage

- Leaving the first entry blank yields a bonus:
  - The children of the node with index k are in 2k and 2k + 1
  - The parent of node with index k is in k ÷ 2
    - Note that index is always an integer

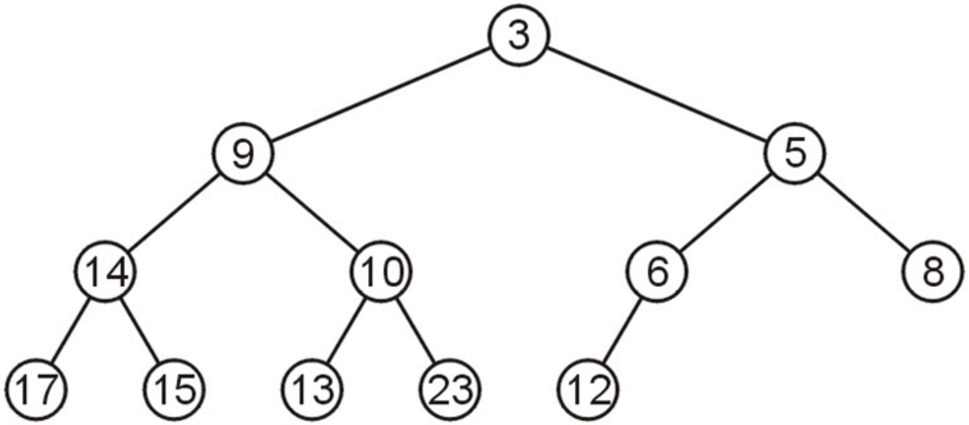


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	3	9	5	14	10	6	8	17	15	13	23	12					

Array storage

- Leaving the first entry blank yields a bonus:
  - In C++, this simplifies the calculations:

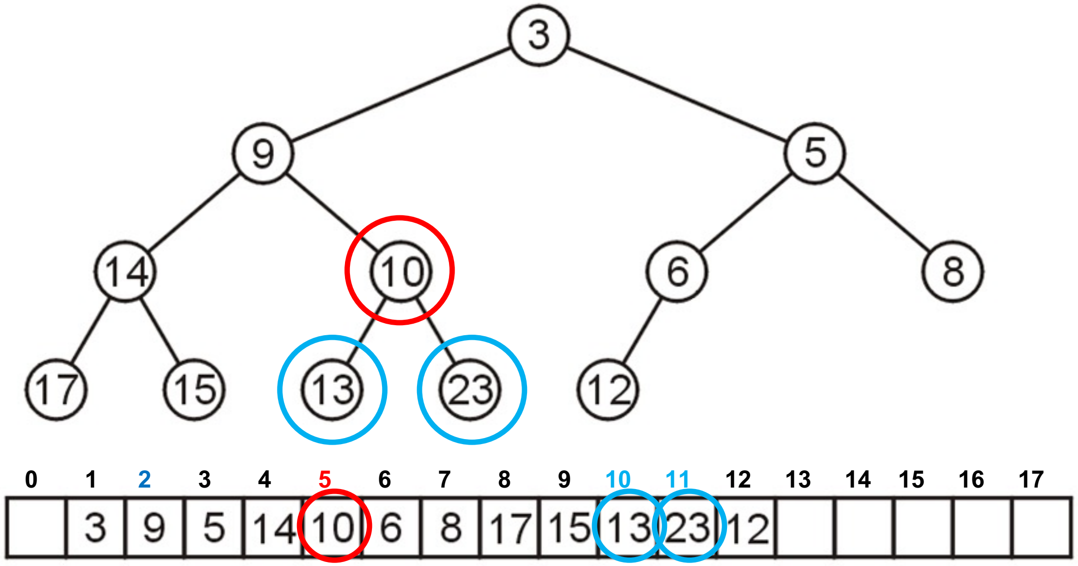
```
parent = k >> 1;
left_child = k << 1;
right_child = left_child | 1;
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	3	9	5	14	10	6	8	17	15	13	23	12					

Array storage

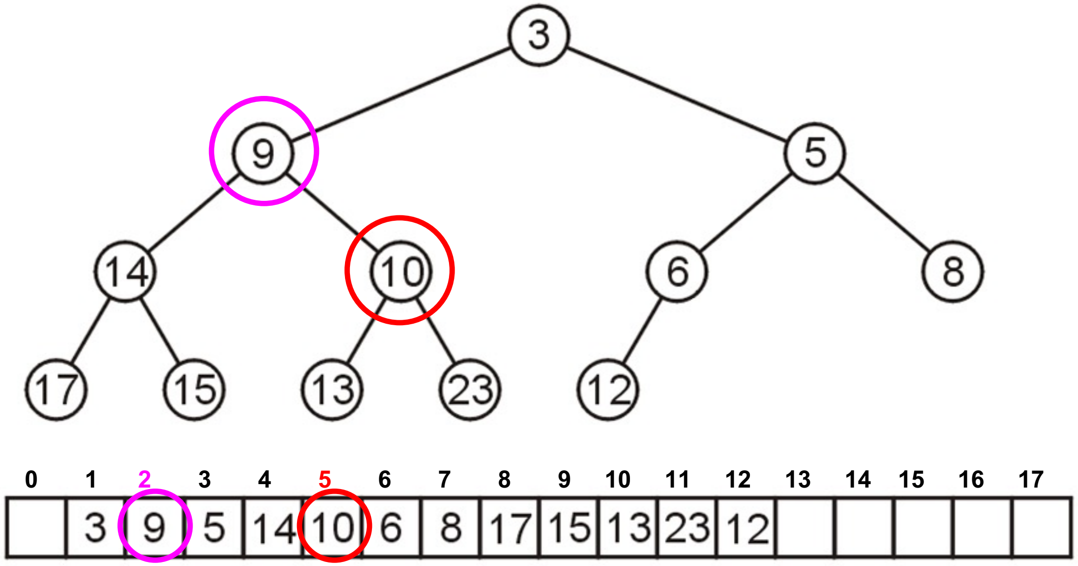
- ☐ For example, node 10 has index 5:
  - Its children 13 and 23 have indices 10 and 11, respectively



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	3	9	5	14	10	6	8	17	15	13	23	12					

Array storage

- For example, node 10 has index 5:
  - Its children 13 and 23 have indices 10 and 11, respectively
  - Its parent is node 9 with index 5/2 = 2

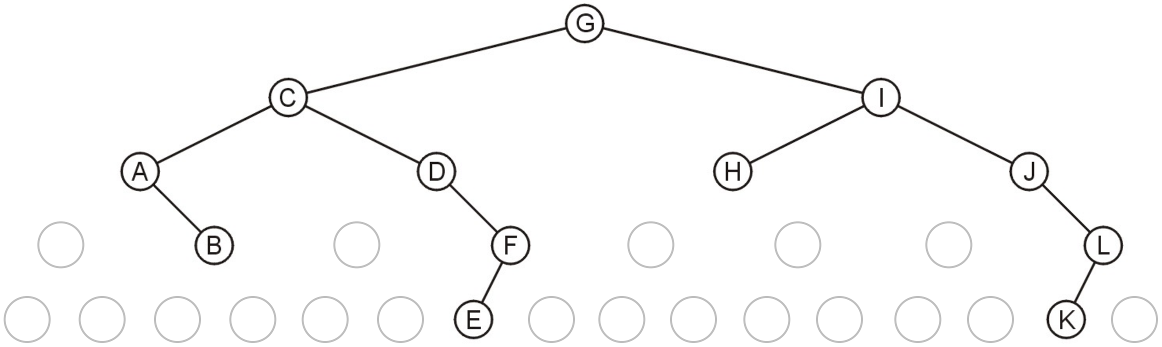


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	3	9	5	14	10	6	8	17	15	13	23	12					

Array storage

- Question: why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

- Consider this tree with 12 nodes would require an array of size 32
  - Adding a child to node K doubles the required memory

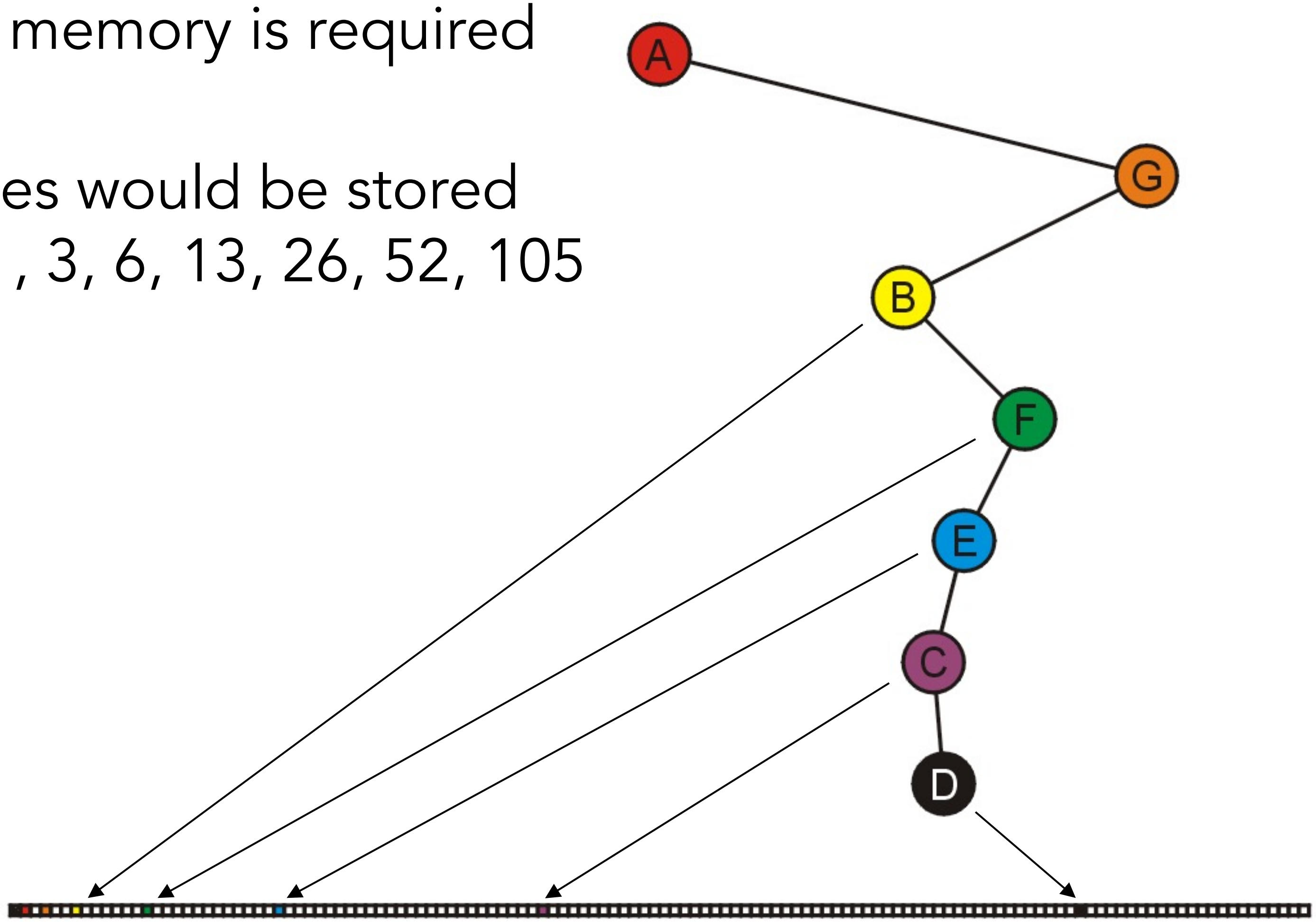


☒	G	C	I	A	D	H	J		B		F				L							E								K	

Array storage

- □ In the worst case, an exponential amount of memory is required

- □ These nodes would be stored in entries 1, 3, 6, 13, 26, 52, 105



Summary

- In this topic, we have covered the concept of a complete binary tree:
  - A useful relaxation of the concept of a perfect binary tree
  - It has a compact array representation

Balanced Trees

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- In this topic, we will:
  - Introduce the idea of *balance*
  - We will introduce a few examples

Background

- ☐ Run times depend on the height of the trees

- ☐ As was noted in the previous section:
  - The best case height is $\Theta(\ln(n))$
  - The worst case height is $\Theta(n)$

- ☐ The average height of a randomly generated binary search tree is actually $\Theta(\ln(n))$
  - However, following random insertions and erases, the average height $\Theta\ (\sqrt{n})$ tends to increase to
    - This is yet to be proven. Check more in Textbook (Weiss) $4.3.6

Requirement for Balance

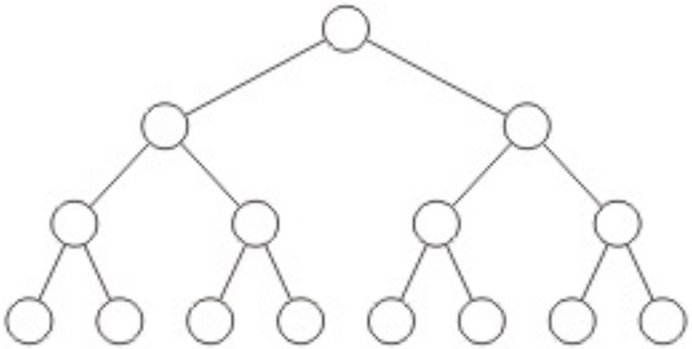
- We want to ensure that the run times never fall into $\omega(\ln(n))$

- Requirement:
  - We must maintain a height which is $\Theta(\ln(n))$

- To do this, we will define an idea of balance

Examples

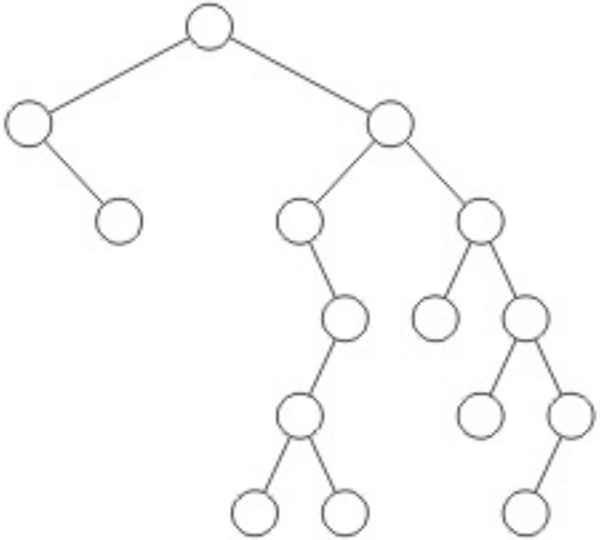
- For a perfect tree, all nodes have the same number of descendants on each side



- Perfect binary trees are balanced while linked lists are not

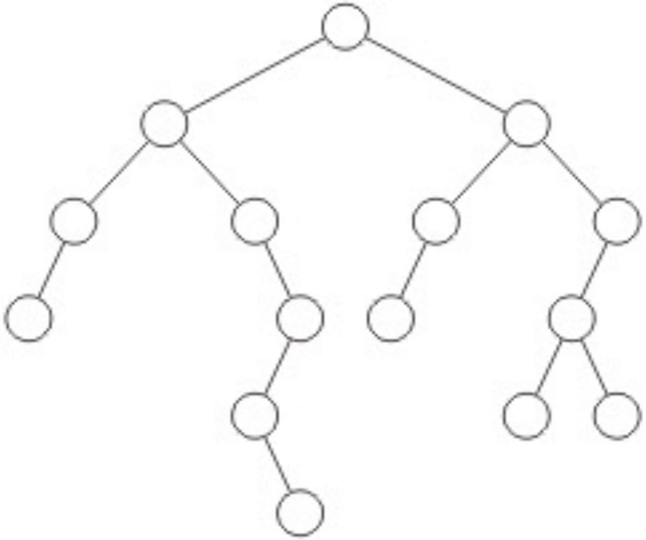
Examples

- This binary tree would also probably not be considered to be “balanced” at the root node



Examples

- How about this example?
  - The root seems balanced, but what about the left sub-tree?



Definition for Balance

- We need a quantitative definition of *balance*

- "Balanced" may be defined by:
  - *Height balancing*: comparing the heights of the two sub trees
  - *Null-path-length balancing*: comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree/empty node)
  - *Weight balancing*: comparing the number of null sub-trees in each of the two sub trees

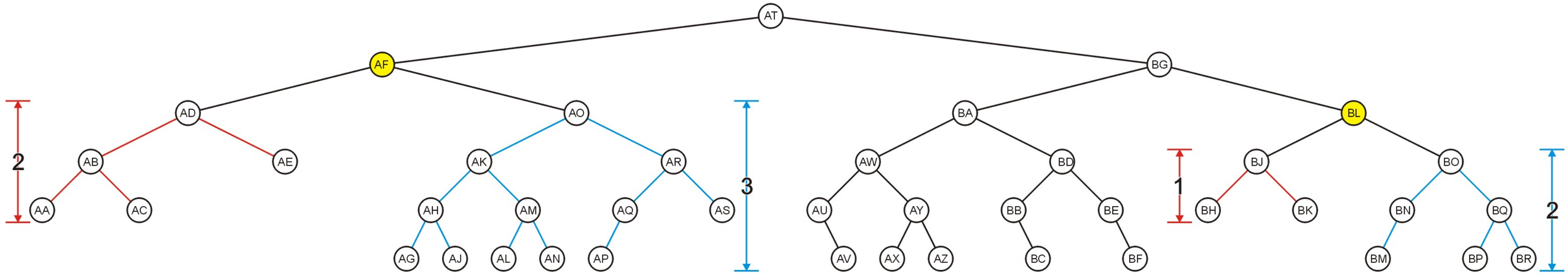
- We will have to mathematically prove that if a tree satisfies the definition of balance, its height is $\Theta(\ln(n))$

Balanced Trees

- Height balancing:
  - AVL trees
    - AVL: named after inventors **A**delson-**V**elsky and **L**andis
- Null-path-length balancing
  - Red-Black Trees
- Weight-Balanced Trees
  - BB Trees

AVL Trees

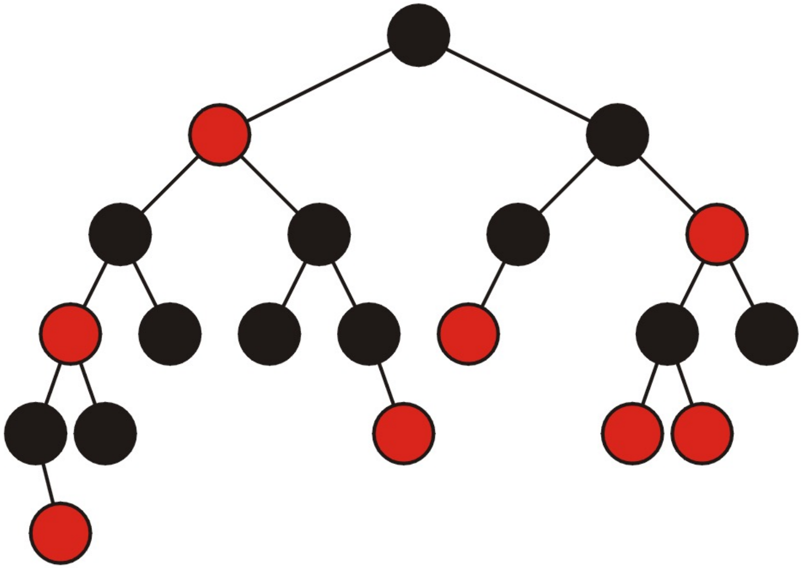
- A node is AVL balanced
  - if two sub-trees differ in height by at most one



Red-Black Trees

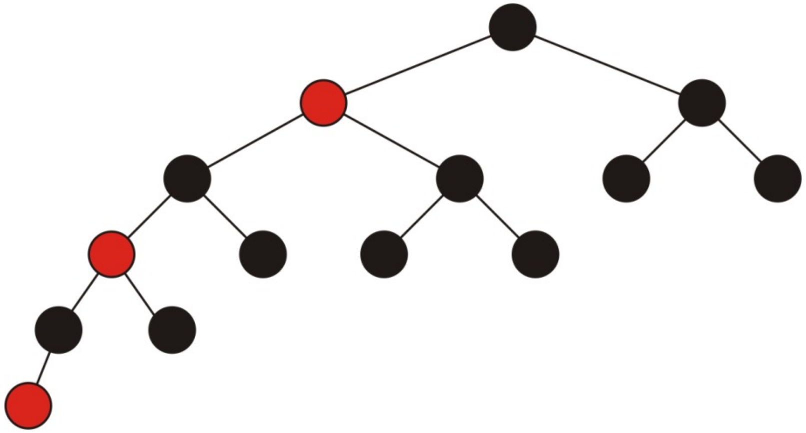
- Red-black trees maintain balance by
    - All nodes are *colored* red or black (0 or 1)

- Requirements:
    - The root must be black
    - All children of a red node must be black
    - Any path from the root to an empty node must have the same number of black nodes



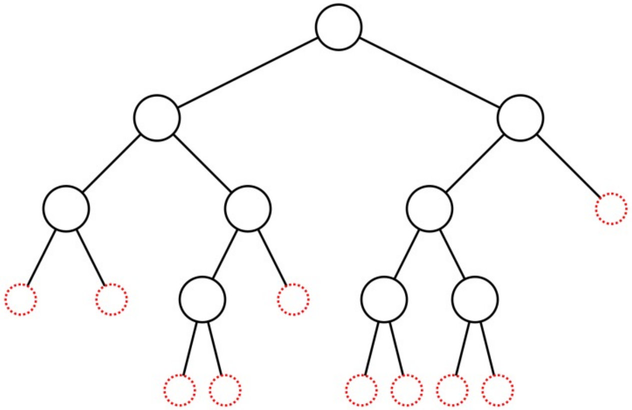
Red-Black Trees

- ☐ **Red-black trees** are **null-path-length balanced** in that the null-path length going through one sub-tree must not be greater than twice the null-path length going through the other
  - For all path
    - # of black nodes >= # of red nodes



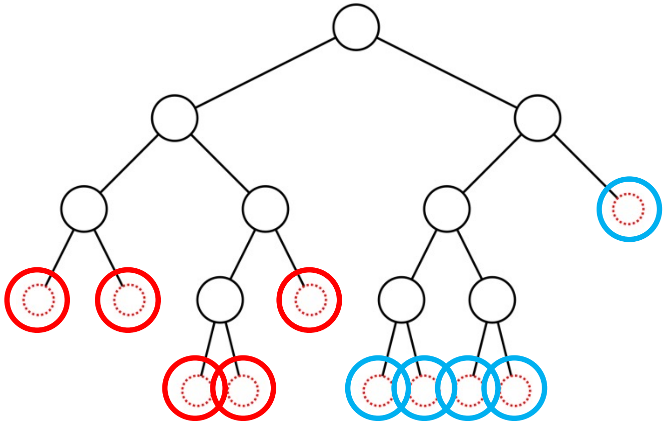
Weight-Balanced Trees

- ☐ Recall: an empty node/null subtree is any position within a binary tree that could be filled with the next insertion:
  - This tree has 9 nodes and 10 empty nodes:



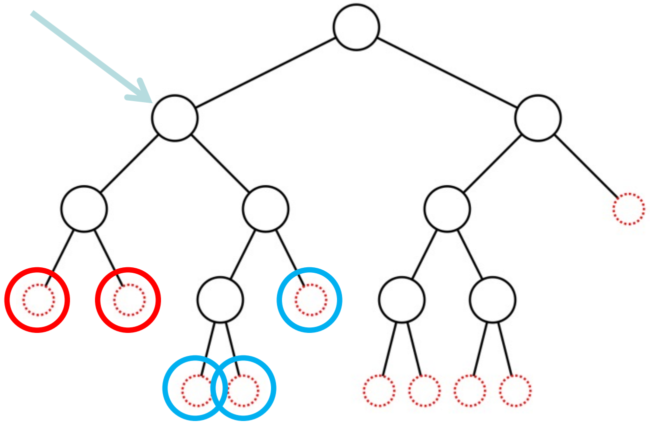
Weight-Balanced Trees

- The ratios of the empty nodes at the root node are 5/10 and 5/10



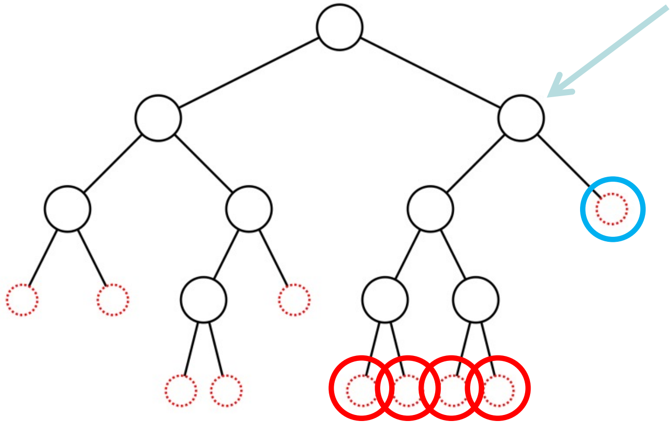
Weight-Balanced Trees

- The ratios of the empty nodes at this node are 2/5 and 3/5



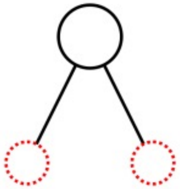
Weight-Balanced Trees

- The ratios of the empty nodes at this node, however, are 4/5 and 1/5

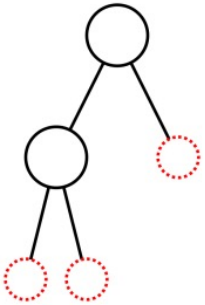


Weight-Balanced Trees

- Bounded balance trees (BB(α) trees) maintain weight balance requiring that neither side has less than a proportion of the empty nodes, *i.e.*, both proportions fall in $[\alpha, 1 - \alpha]$
  - With one node, both are 0.5



  - With two, the proportions are 1/3 and 2/3



Summary

- In this talk, we introduced the idea of *balance*
  - We require O(ln(*n*)) run times
  - Balance will ensure the height is Θ(ln(*n*))

- There are numerous definitions:
  - AVL trees use height balancing
  - Red-black trees use null-path-length balancing
  - BB(a) trees use weight balancing

References

- Blieberger, J., *Discrete Loops and Worst Case Performance*, Computer Languages, Vol. 20, No. 3, pp.193-212, 1994.