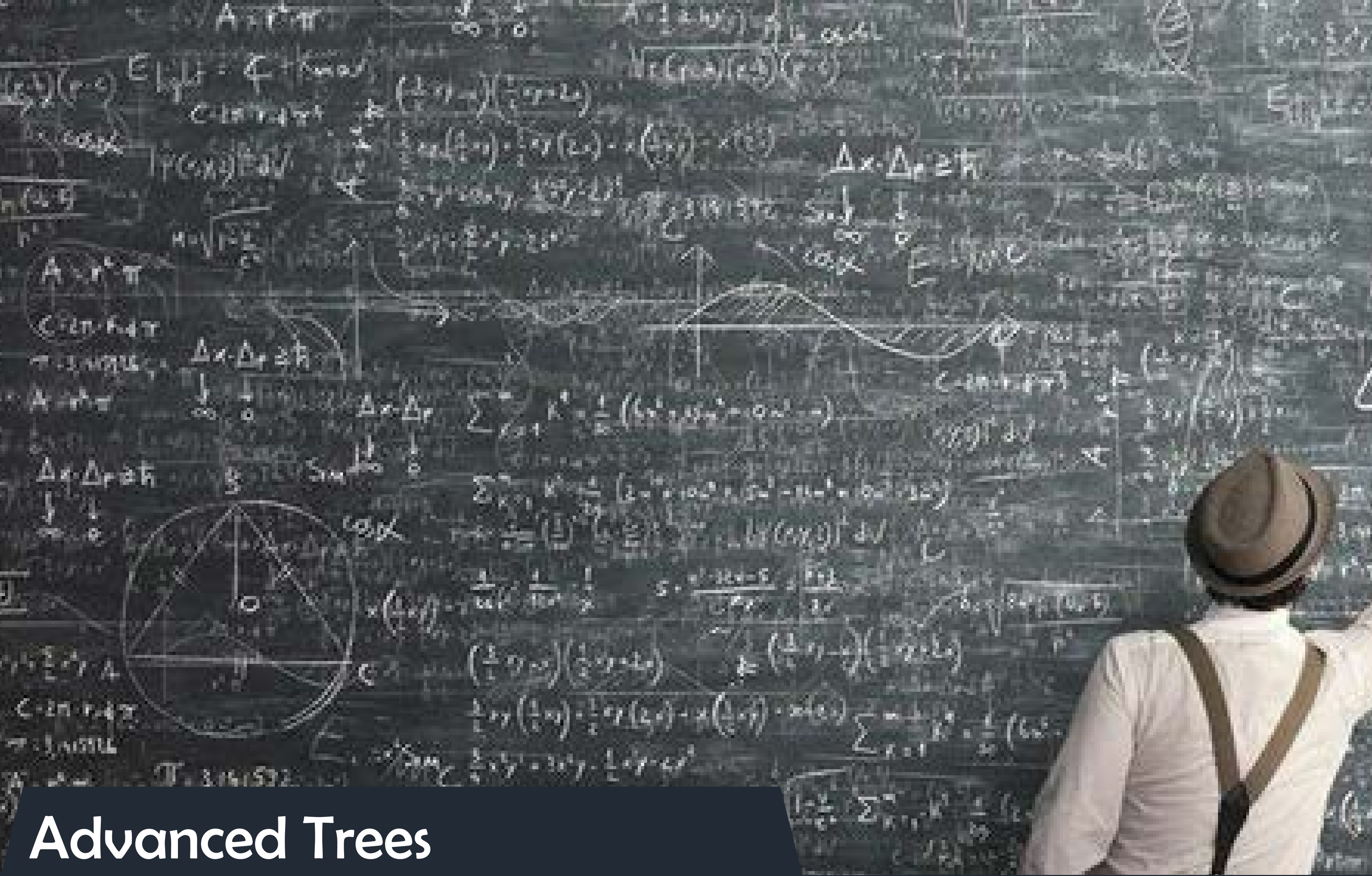
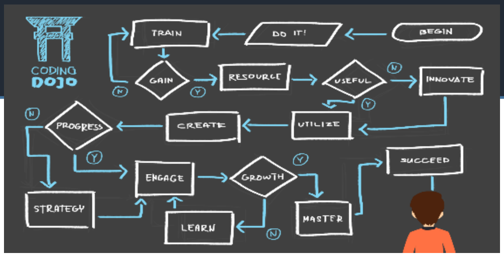


Advanced Trees

Ja-Hee Kim

Agenda



Review

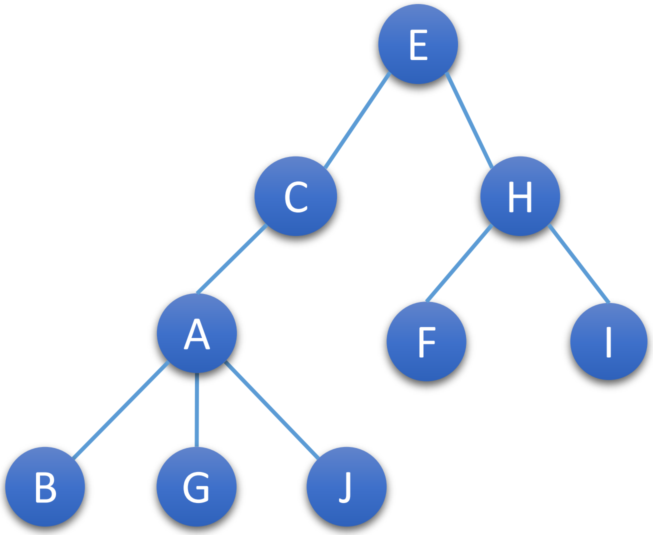
Algorithms

Group activity

Review

Tree

- Tree is a non-linear structure based on nodes and links.
- Rooted tree
  - Empty tree is a tree.
  - If S is a set of trees
    - any trees of S do not share a node.
    - T = (r, S) is a tree
      - r is a root
      - a tree in S is a sub-tree of T



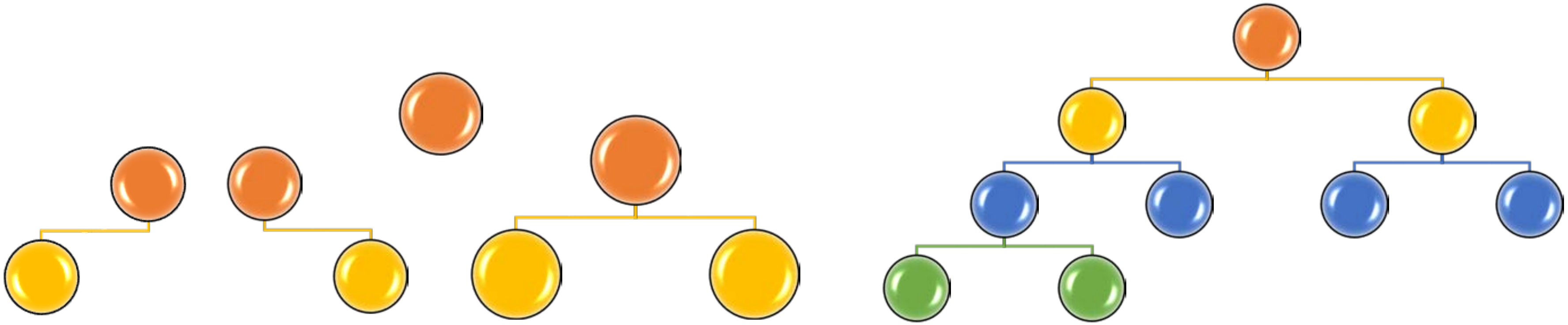
- Terminologies
  - Node, Edge, Root, Parent, Children, Ancestor, Descendant , Subtree, Degree, Leaf, Interior node, Path, Level, Depth, Height

Binary Tree

- A tree of which the maximum degree is two

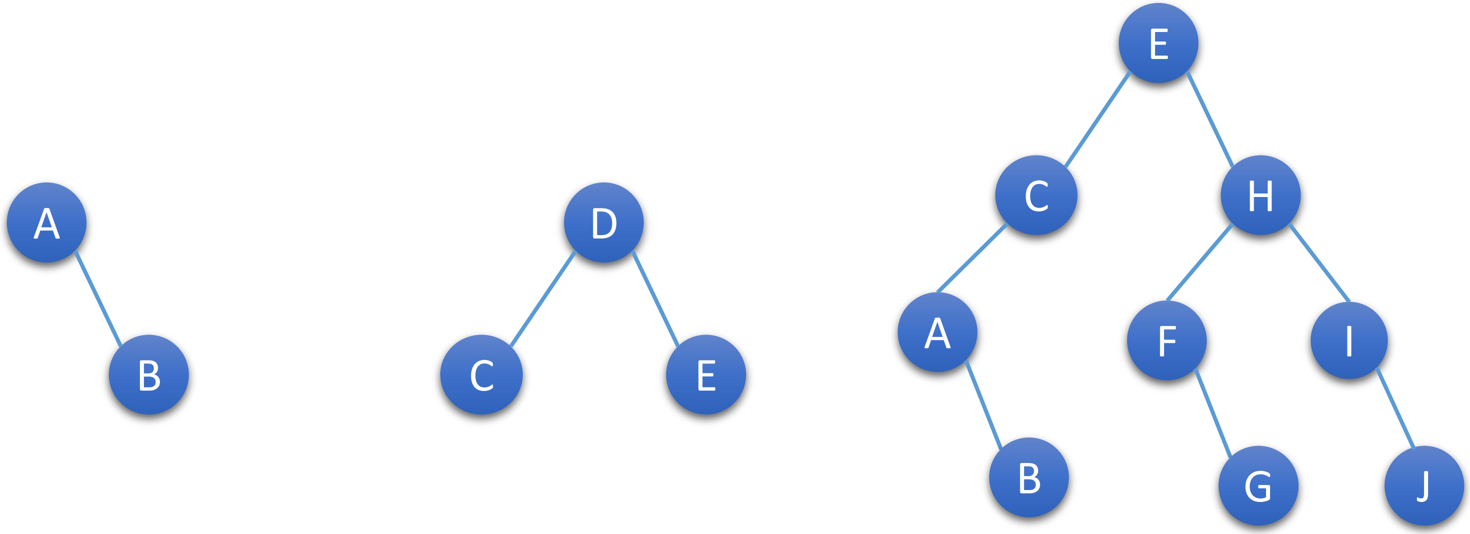
  ≠ a tree of 2 degree
- Recursive definition of a binary tree
  - Empty tree is a binary tree.
  - Each tree has two subtrees whose root nodes are the nodes pointed to by the leftLink and rightLink of the root node.
- Terminologies
  - Ordered tree, full tree, complete tree, skewed tree, expression tree
- The number of node of which tree (n) where the height is h

$$h + 1 \leq n \leq 2^{h+1} - 1$$



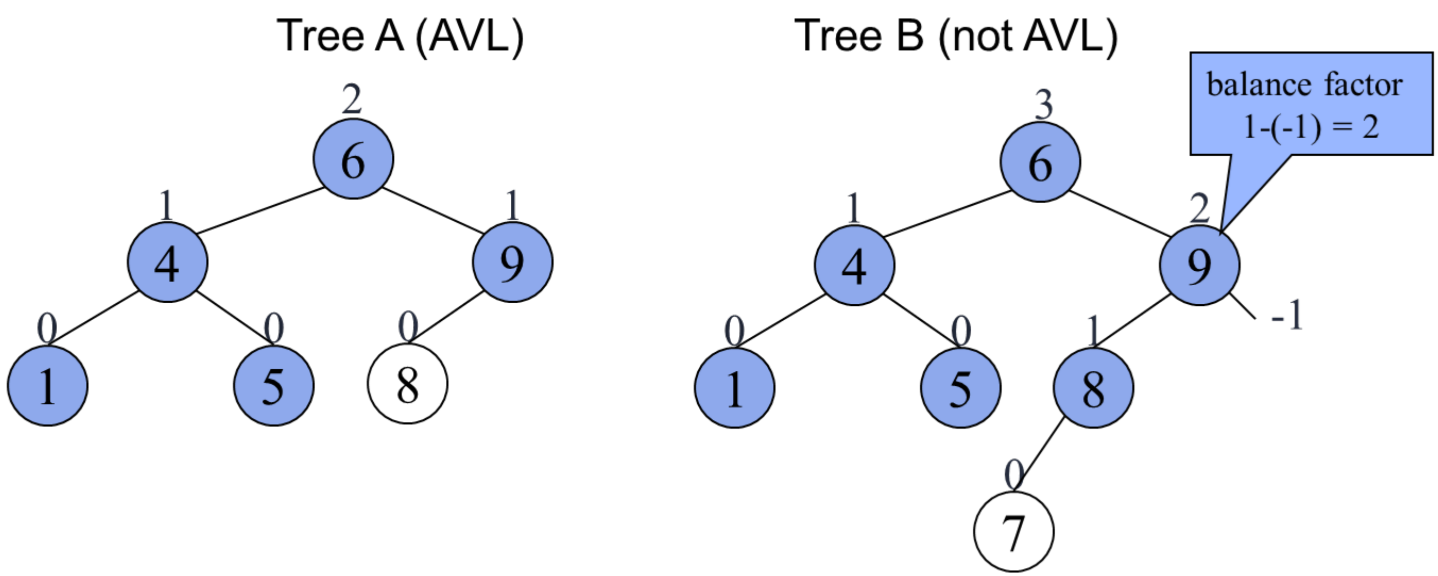
Binary Search Tree

- Binary search Tree
- Each node can have up to two child nodes.
- All keys should be different.
- Each node has a key
  - All keys of the left subtree is less than the root's
  - All keys of the right subtree is greater than the root's



AVL tree

- Motivation
  - Performance of a BST depends on the height of the tree.
- AVL tree
  - a self-balancing binary search tree
  - the heights of the two child subtrees of any node differ by at most one



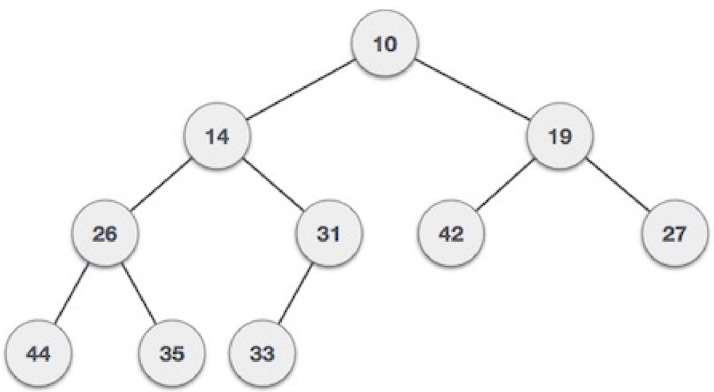
Performance Comparison

- Is there any data structure whose expectation time complexities for looking up, adding, and removing are constant?

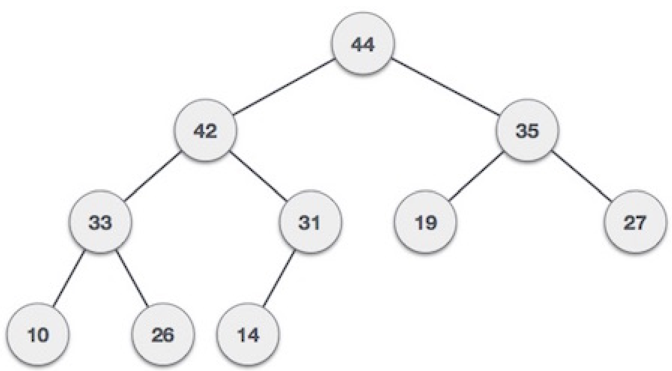
	Binary search tree	Balance BST	Sorted List	Hash table
Look up	Expected O(log n) Worst case O(n)	O(log n)	O(log n)	Worst: O(n) Average: O(1)
add			O(n)	
delete				

Heap

- Complete binary tree with keys
- It satisfies two properties:
  - MinHeap: key(parent) <= key(child)
  - [OR MaxHeap: key(parent) >= key(child)]



Min-Heap – Where the value of the root node is less than or equal to either of its children.



Max-Heap – Where the value of the root node is less than or equal to either of its children.

Homework

- Solve Quiz!

Visualization: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
Source code: https://junboom.tistory.com/18

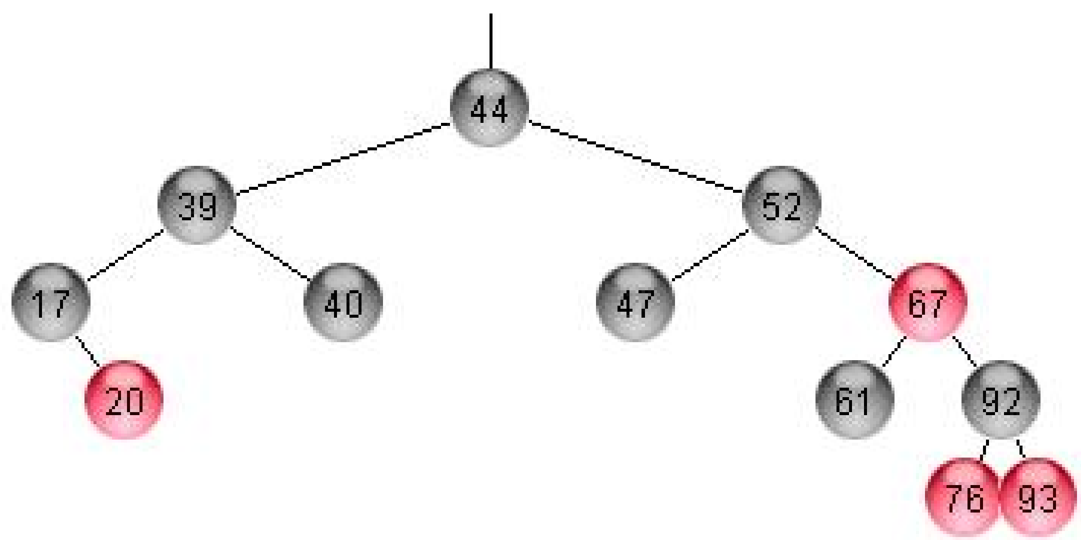
Red-Black tree

Balanced BST

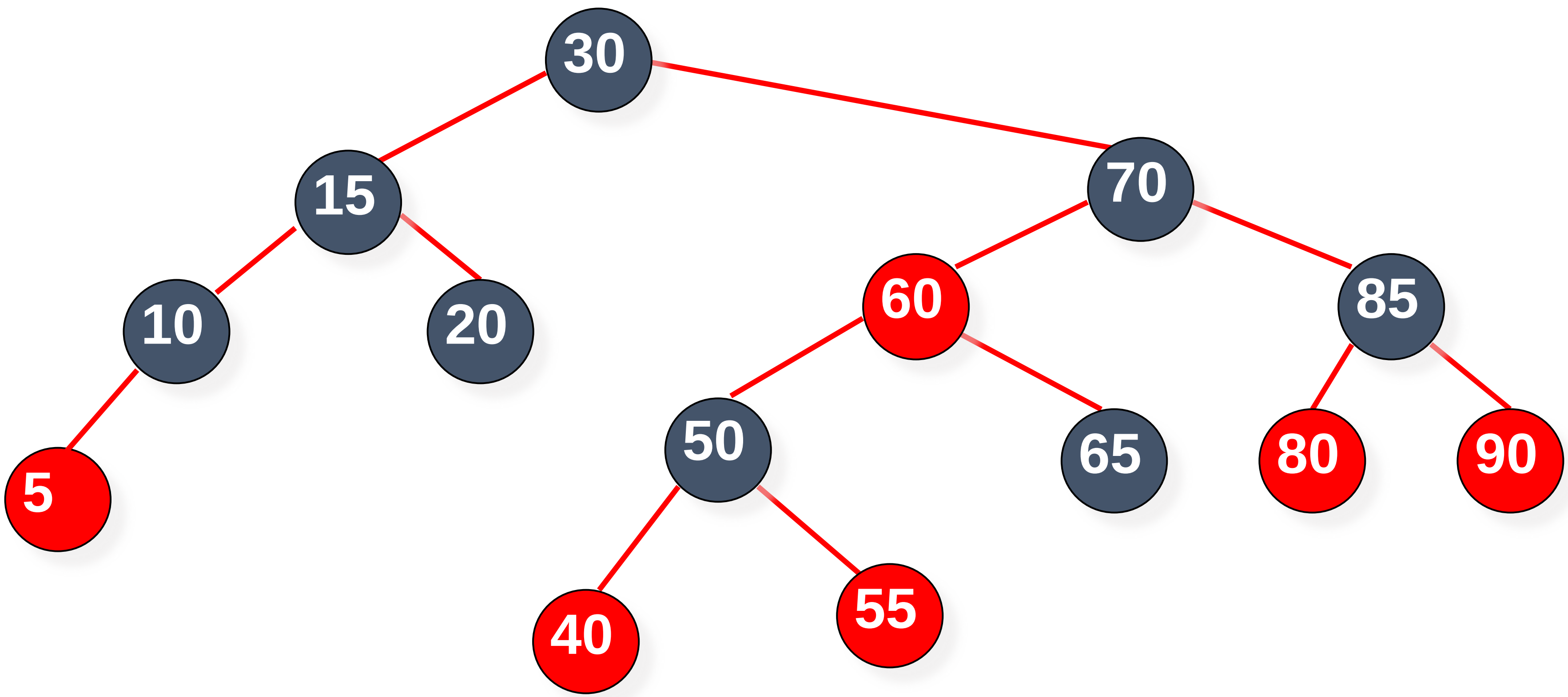
- |depth(leftChild) – depth(rightChild)| <= 1
- Example
  - AVL Trees – Maintain a three-way flag at each node (-1,0,1) determining whether the left sub-tree is longer, shorter or the same length. Restructure the tree when the flag would go to –2 or +2.
  - Red-black trees – Restructure the tree when rules among nodes of the tree are violated as we follow the path from root to the insertion point.

A Red-Black Tree

- Red-black trees are tress that conform to the following rules:
  1. Every node is colored (either red or black)
  2. The root is always black
  3. If a node is red, its parent and children must be black
  4. Every path from the root to leaf, or to a null child, must contain the same number of black nodes.

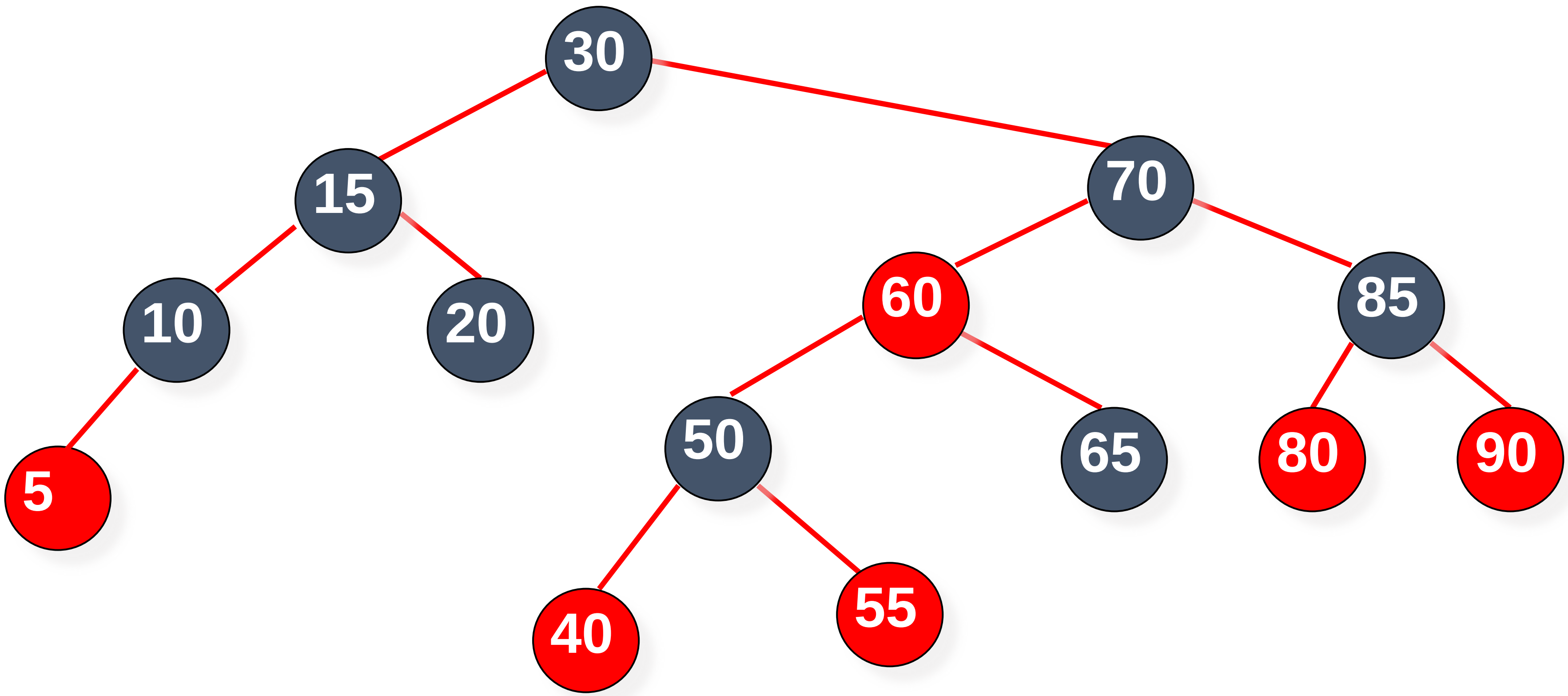


A Red-Black Tree



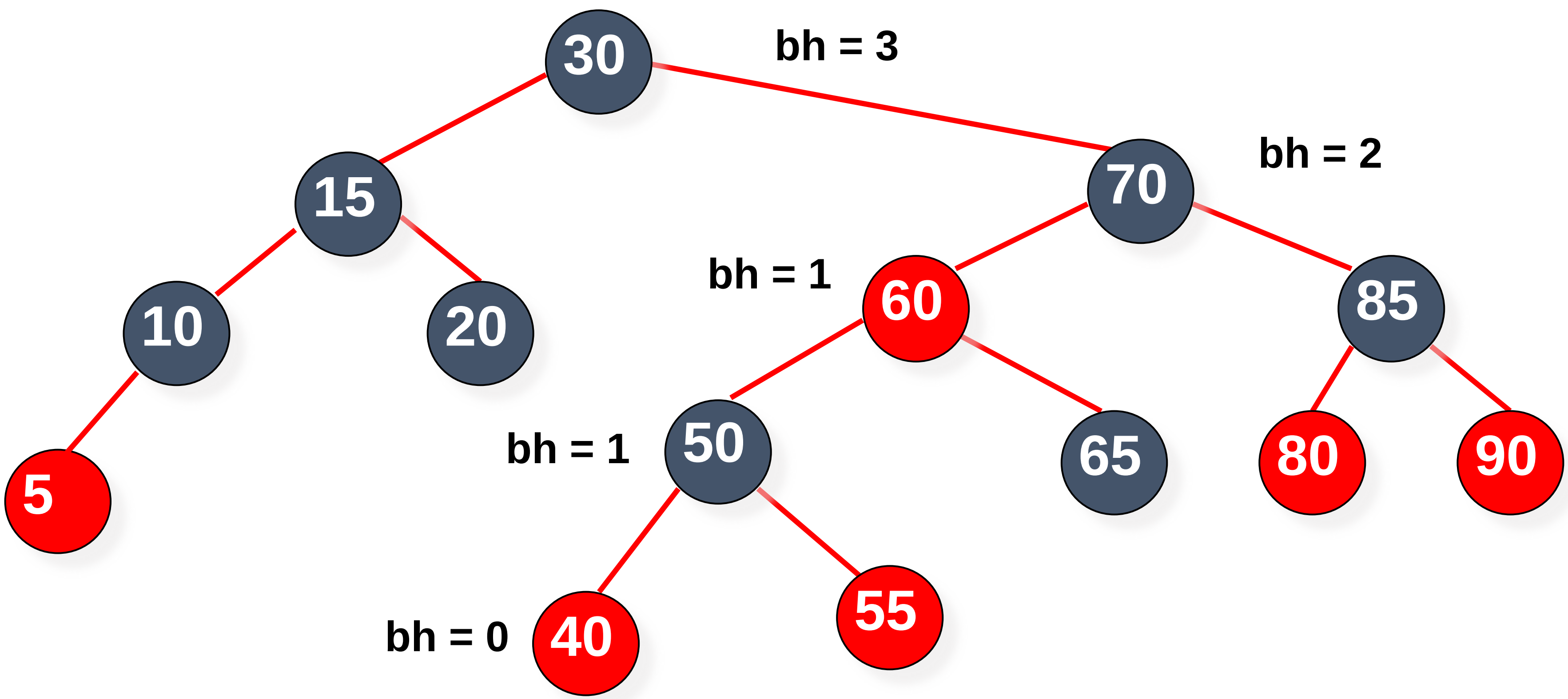
1. Every node is colored either red or black
2. The root is black

A Red-Black Tree



3. If a node is red, its children must be black

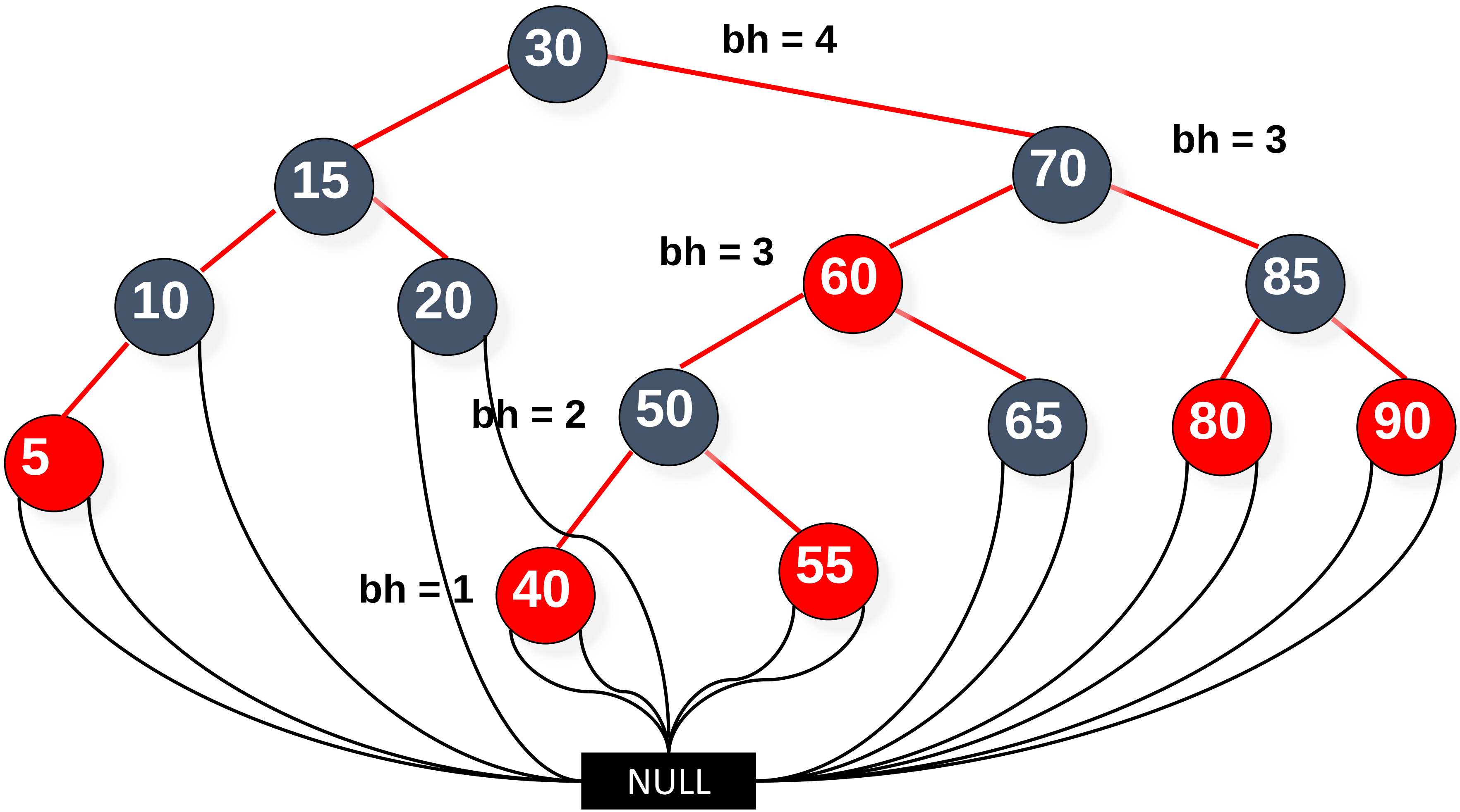
A Red-Black Tree



4. All simple paths from any node x to a descendent leaf must contain the same number of black nodes

= *black-height(x)*

A Red-Black Tree



5. We assume a null point is black.

Insertion Algorithm

- Where
  - New node: *x*
    - *x* is always red
  - The parent of *x*: *p*
  - The parent of *p*: p^2
  - The sibling of *p(uncle)*: *u*
- Case 1: empty tree – insert black node
- Case 2: *p* is black – insert red node
- Case 3: *p* is red
  - Case 3-1: *u* is red
  - Case 3-2: *u* is black
    - Case 3-2-1: *x* is the right child of *p*
    - Case 3-2-2: *x* is the left child of *p*

We consider only when p is the left child of p^2.

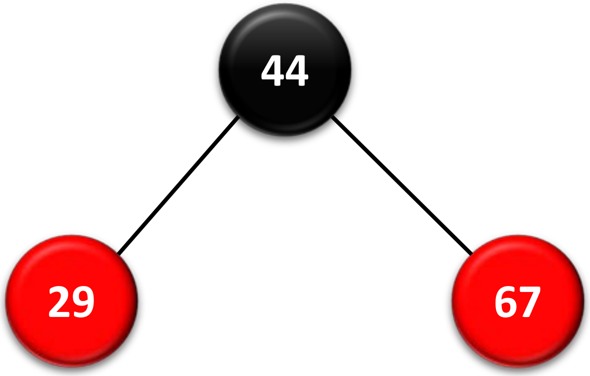
If p is the right child of p^2, left ←→ right

Case 1

- Empty tree: insert black node
- Insert 44

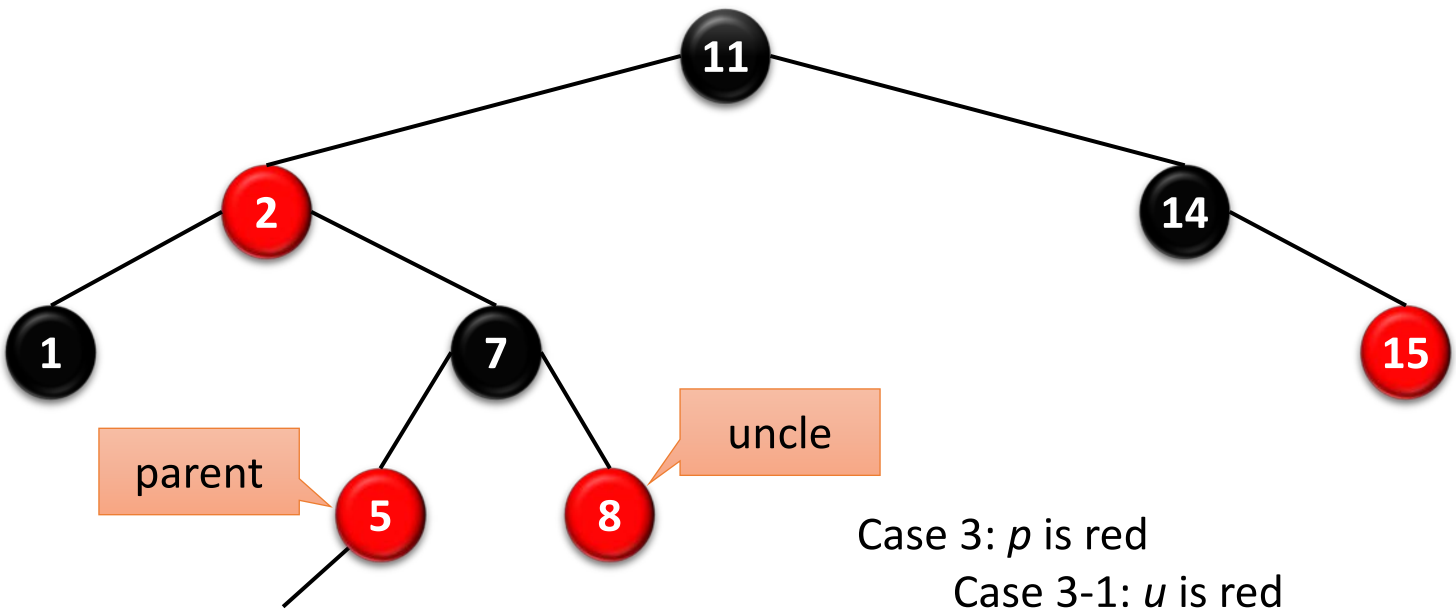
Case 2

- Parent node is black: Insert red node
- Insert 29 and 67



Case 3

- Insert 4



Case 3: *p* is red

Case 3-1: *u* is red

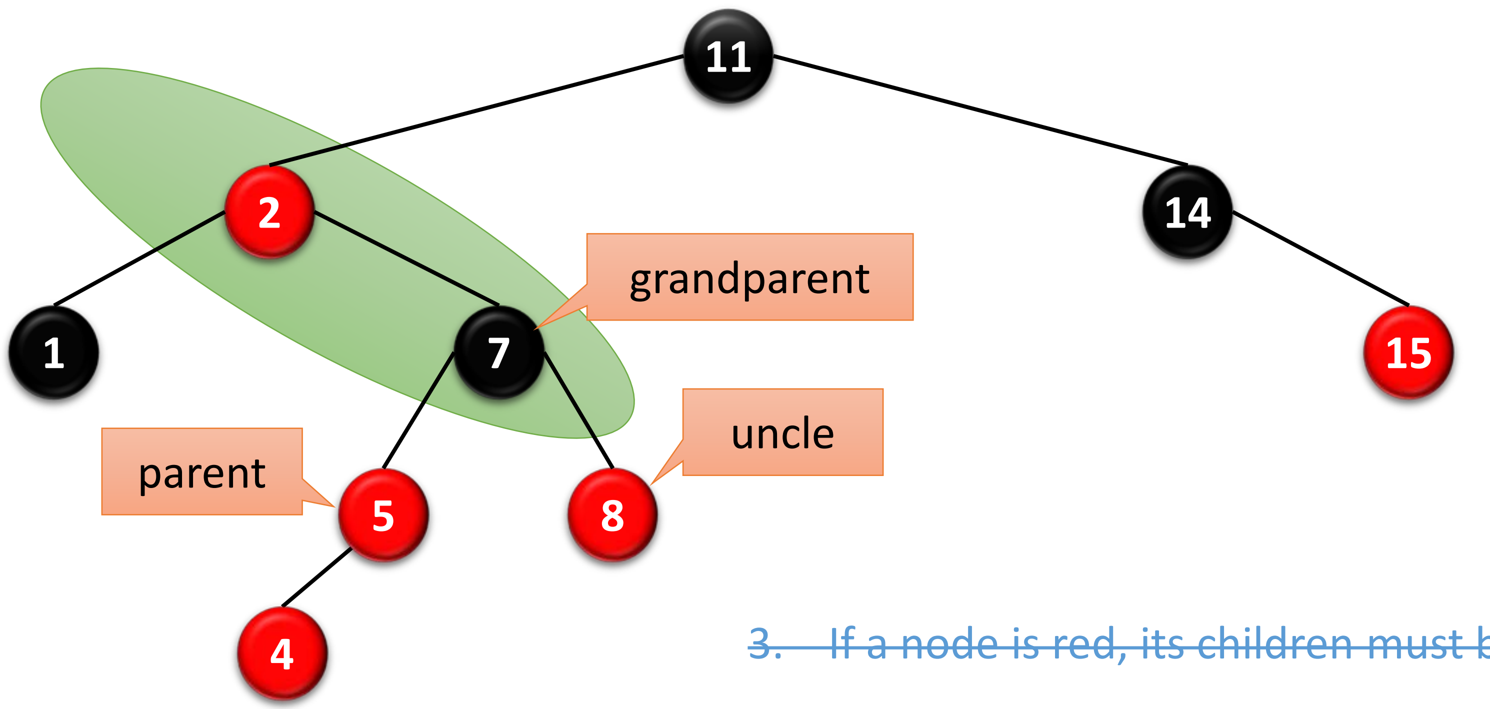
Case 3-2: u is black

Case 3-2-1: x is the right child of p

Case 3-2-2: x is the left child of p

Case 3-1

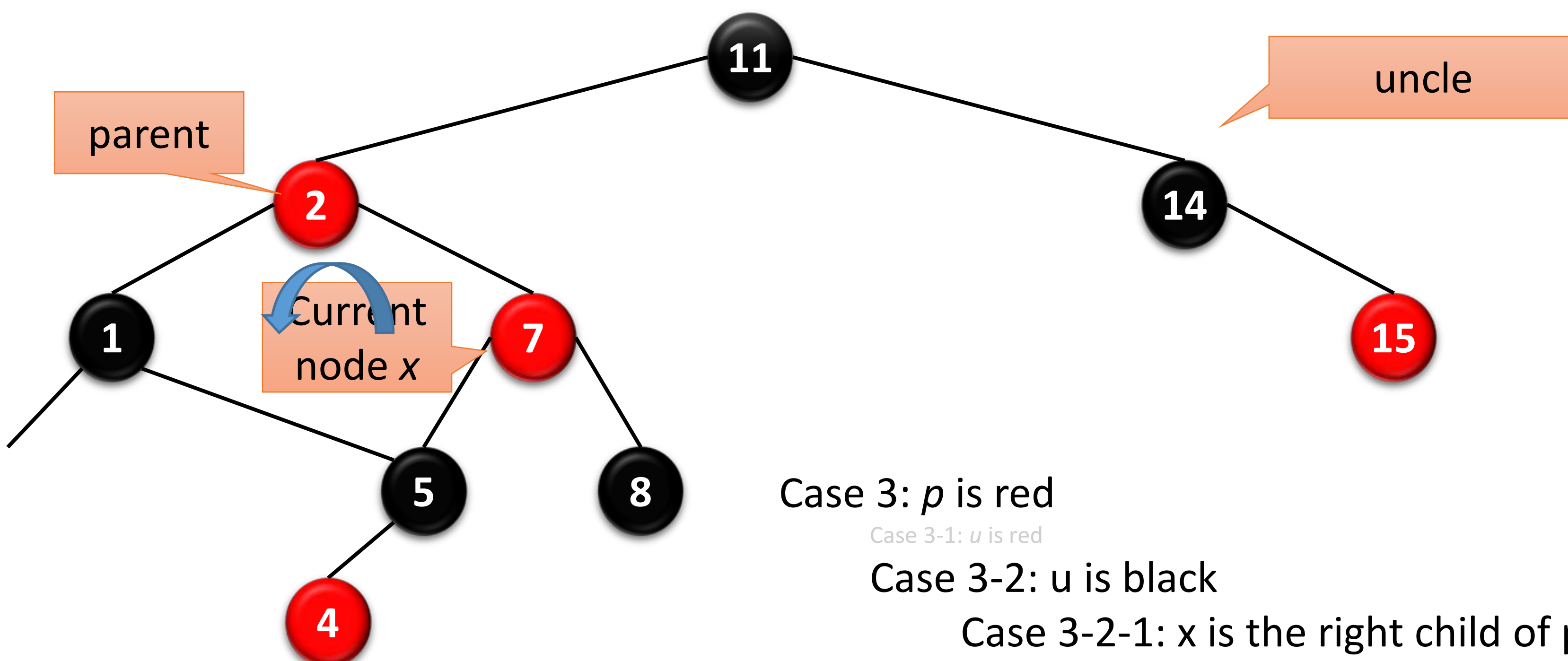
- Change colors
  - Parent & uncle : black
  - Grandparent: red



~~3. If a node is red, its children must be black~~

Case 3-2-1

- Left-rotation on parent



Case 3: *p* is red

Case 3-1: *u* is red

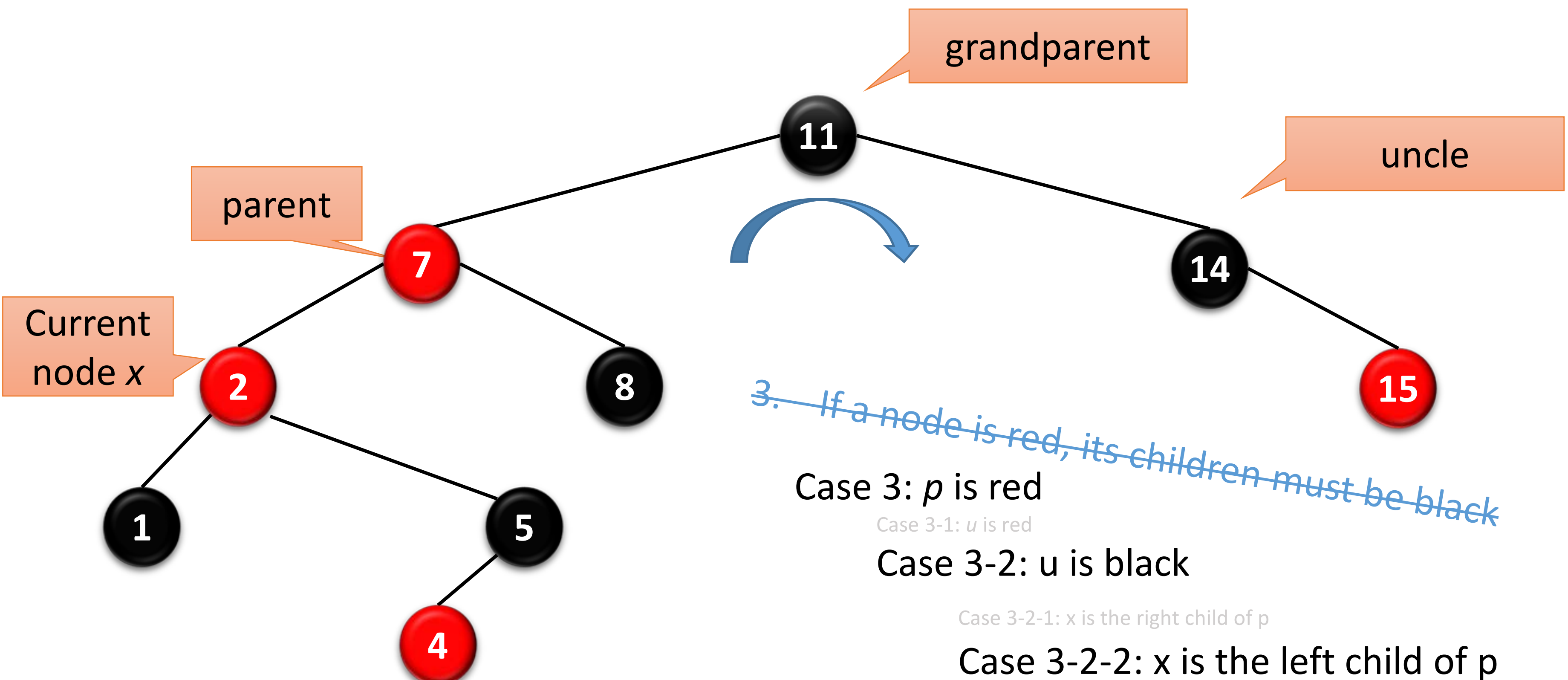
Case 3-2: u is black

Case 3-2-1: x is the right child of p

Case 3-2-2: x is the left child of p

Case 3-2-2

- Right-rotation on grandparent
- Change colors
  - Parent(7) & grandparent(11)



~~3. If a node is red, its children must be black~~

Case 3: *p* is red

Case 3-1: *u* is red

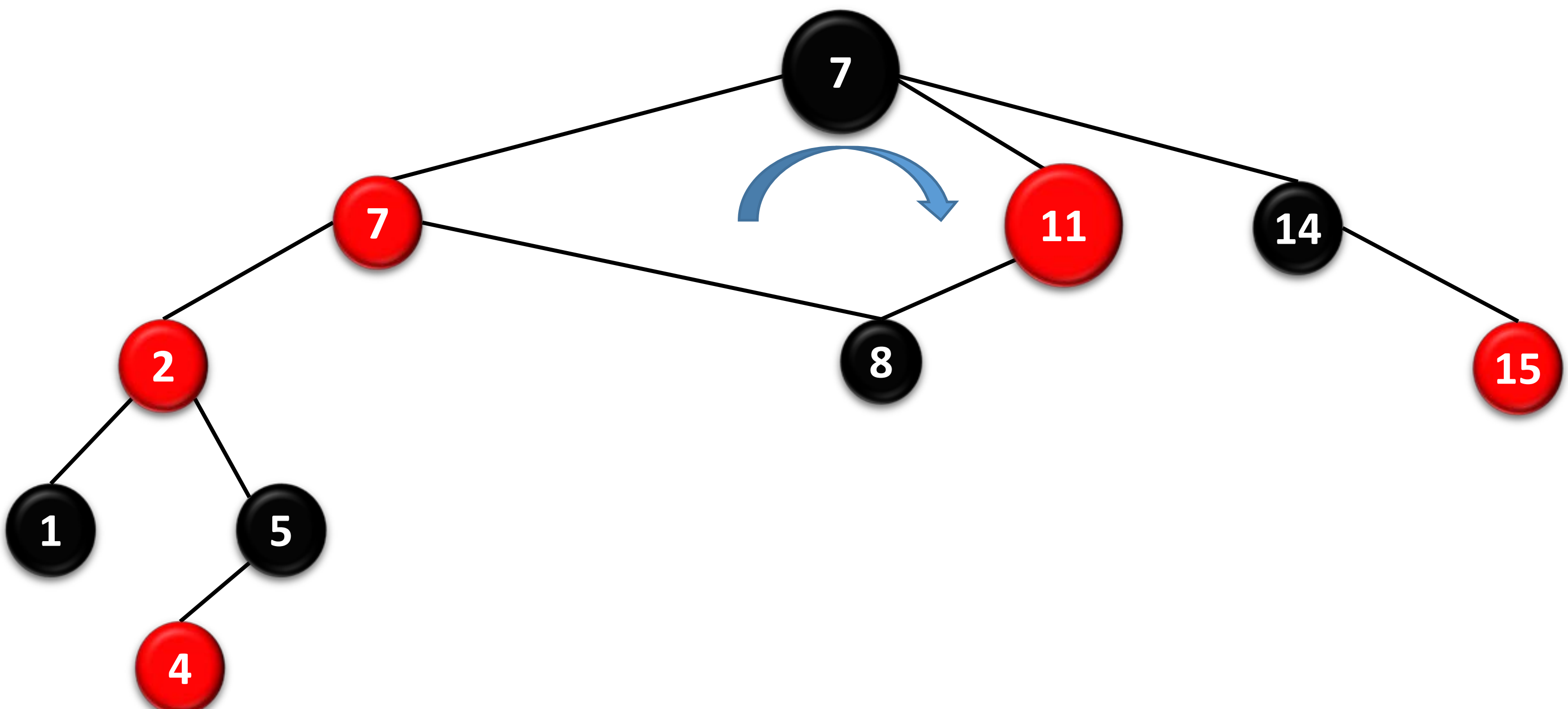
Case 3-2: u is black

Case 3-2-1: x is the right child of p

Case 3-2-2: x is the left child of p

Case 3-2-2

- Right-rotation on grandparent
- Change colors
  - Parent(7) & grandparent(11)



Time complexity

- Insert: $O(\log n)$: maximum height of tree
- Color red: $O(1)$
- Fix violations: $O(\log n)$
  - Const # of:
    - Recolor: O(1)
    - Rotation: O(1)

- Total: $O(\log n)$

Pro and Con of Red-black Trees

- Advantages
  - AVL: relatively easy to program. More balanced. Insert requires only one rotation.
  - Red-Black: Fastest in practice, no traversal back up the tree on insert
- Disadvantages
  - AVL: Repeated rotations are needed on deletion, must traverse back up the tree.
  - Red-Black: Multiple rotates on insertion, delete algorithm difficult to understand and program

Visualization: https://www.cs.usfca.edu/~galles/visualization/BTree.html
Source code: https://www.geeksforgeeks.org/introduction-of-b-tree-2/

B tree

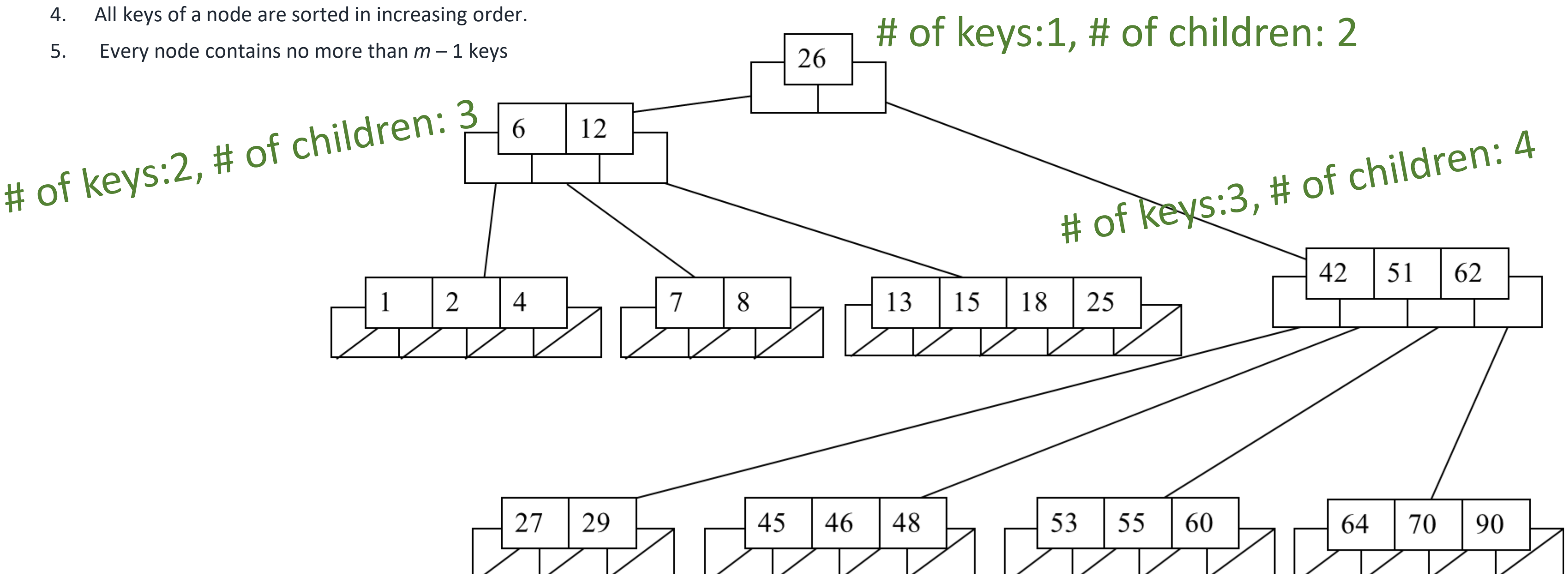
Motivation

- Index structures for large datasets cannot be stored in main memory
- We end up with a very deep binary tree with lots of different disk accesses;
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition

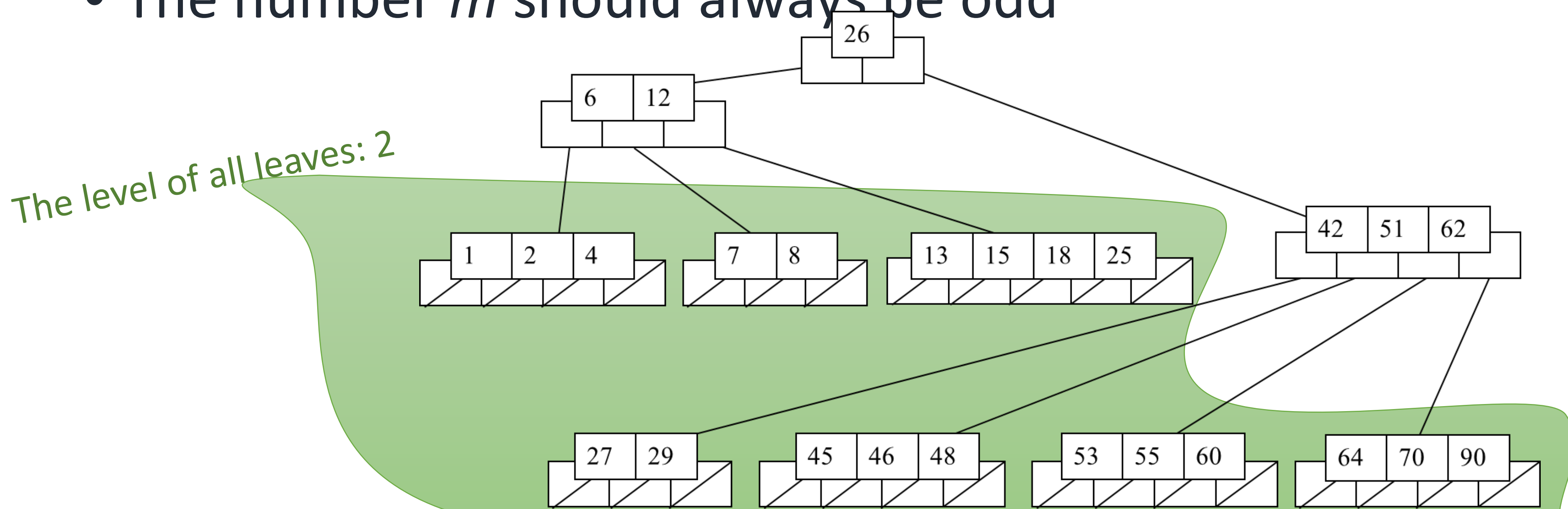
- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

    1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
    2. all leaves are on the same level
    3. all non-leaf nodes except the root have at least $\lceil (m-1)/2 \rceil$ keys.
    4. All keys of a node are sorted in increasing order.
    5. Every node contains no more than $m - 1$ keys



Definition

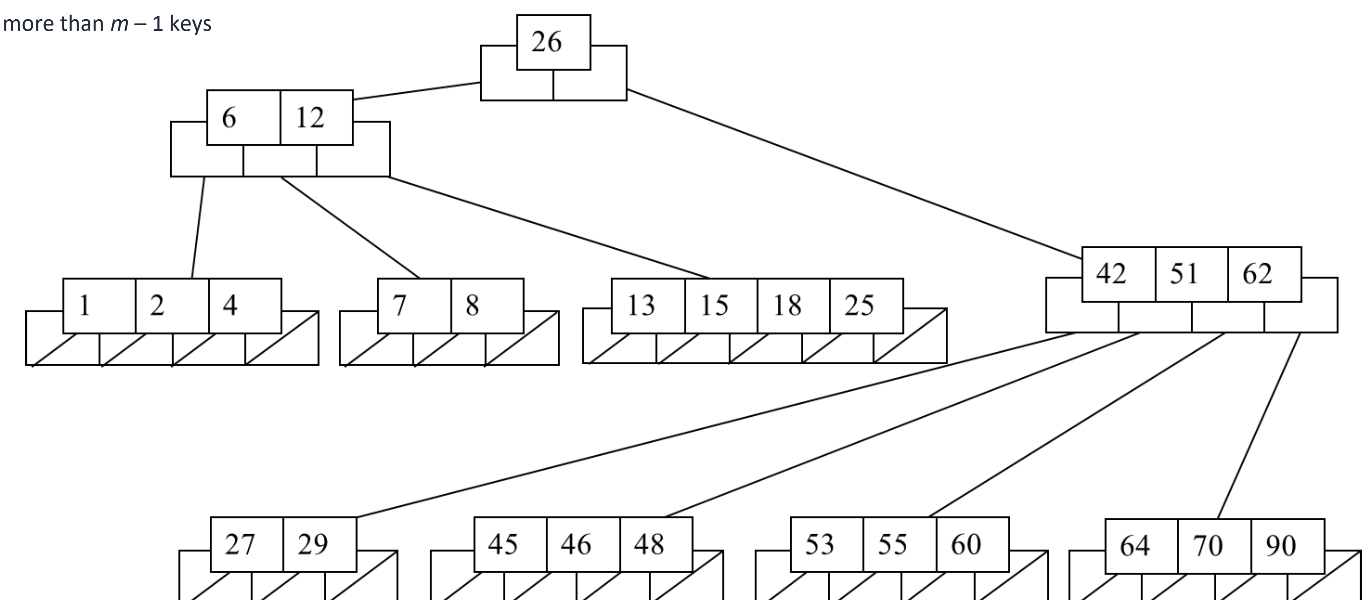
- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil (m-1)/2 \rceil$ keys
  4. All keys of a node are sorted in increasing order.
  5. Every node contains no more than $m - 1$ keys
- The number *m* should always be odd



Definition

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

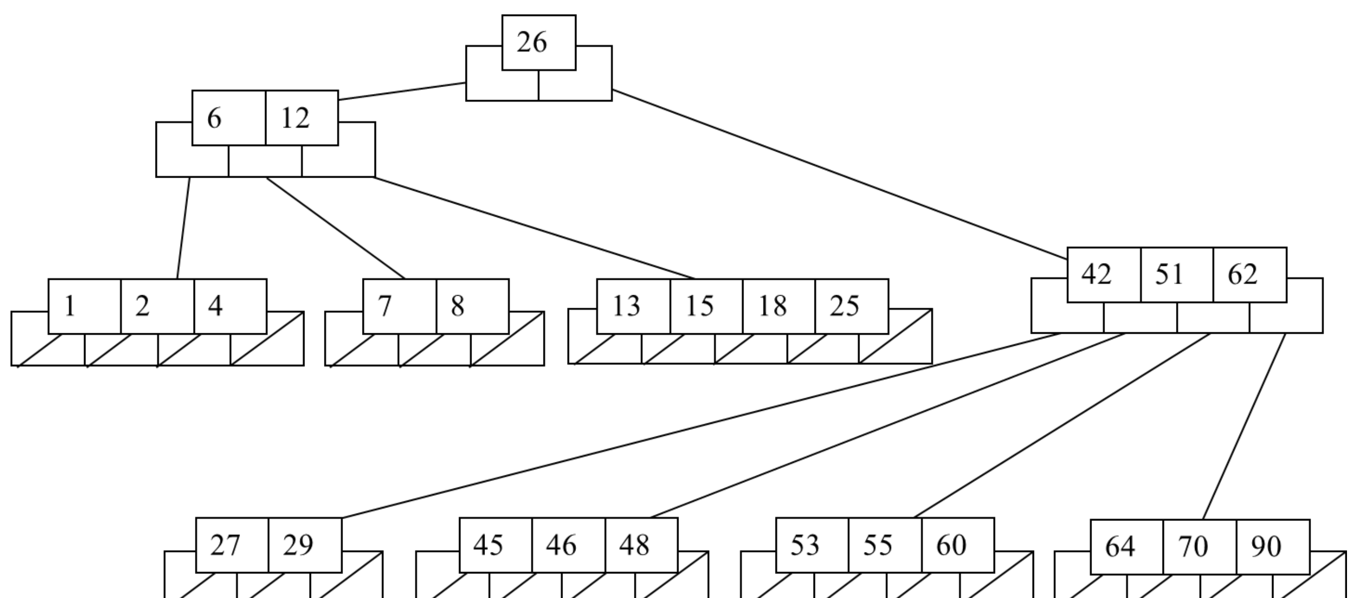
    1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
    2. all leaves are on the same level
    3. all non-leaf nodes except the root have at least $\left\lceil \frac{m-1}{2} \right\rceil$ key.
    4. All keys of a node are sorted in increasing order.
    5. Every node contains no more than $m-1$ keys



Definition

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

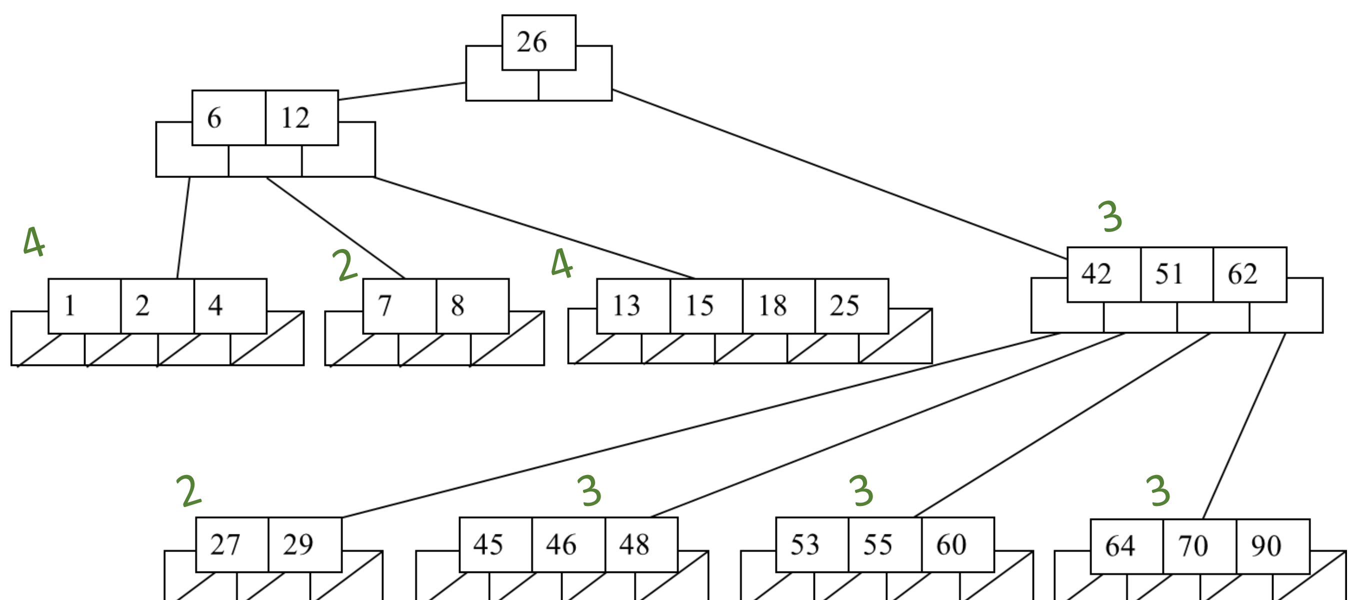
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil (m-1)/2 \rceil$ keys
  4. All keys of a node are sorted in increasing order.
  5. Every node de contains no more than $m - 1$ keys



Definition

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil (m-1)/2 \rceil$ keys
  4. All keys of a node are sorted in increasing order.
  5. Every node contains no more than *m* – 1 keys

The number of keys of each leaf <= 5-1

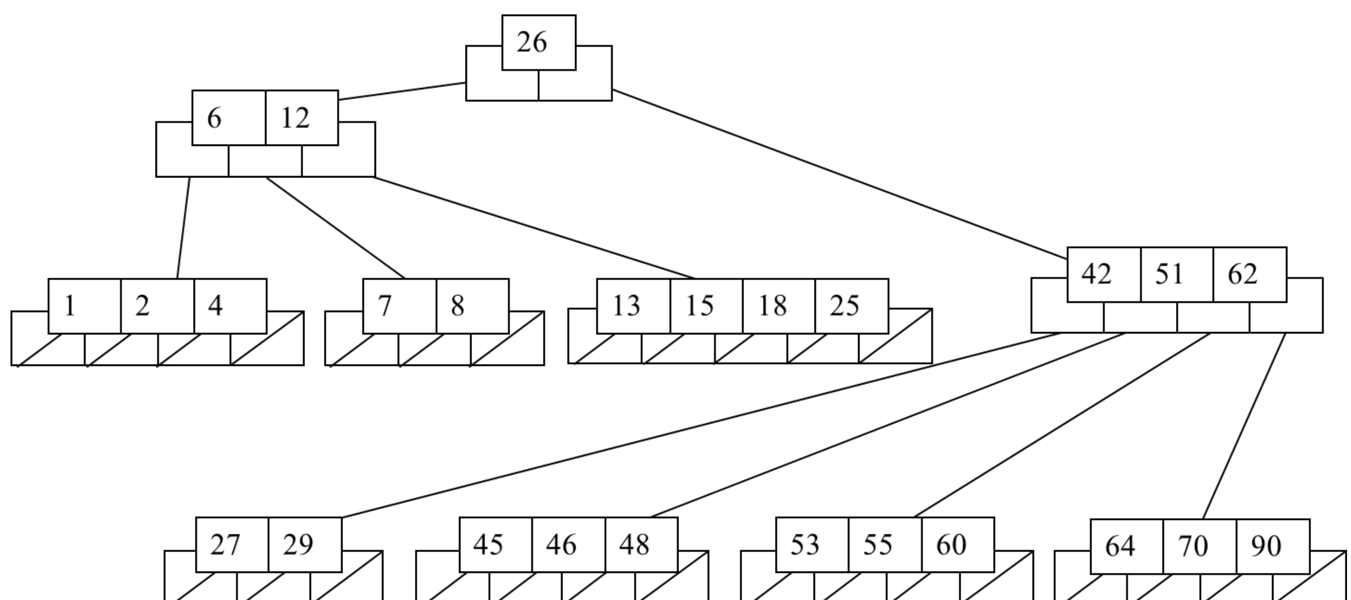


Definition

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
    1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
    2. all leaves are on the same level
    3. all non-leaf nodes except the root have at least $\lceil (m-1)/2 \rceil$ keys
    4. All keys of a node are sorted in increasing order.
    5. Every node contains no more than $m - 1$ keys

- The number *m* should always be odd

m = 5: odd

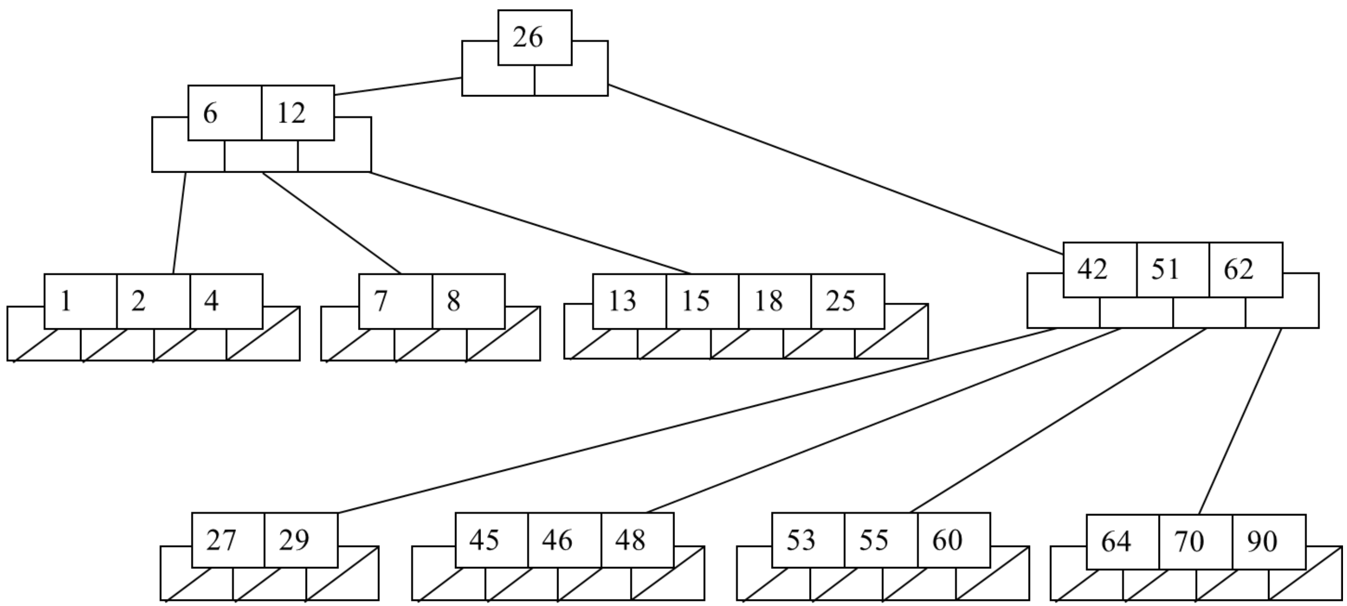


Definition

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\left\lceil \frac{m-1}{2} \right\rceil$ keys.
  4. All keys of a node are sorted in increasing order.
  5. Every node contains no more than $m - 1$ keys
- The number *m* should always be odd

Searching

- Search will start with the root.
- Check in which range the key is.
- Example: Finding 60.

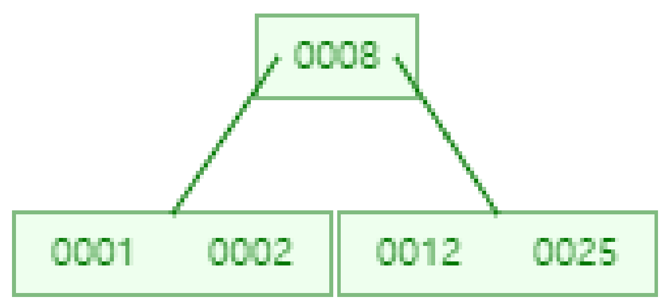


Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

1	2	8	12

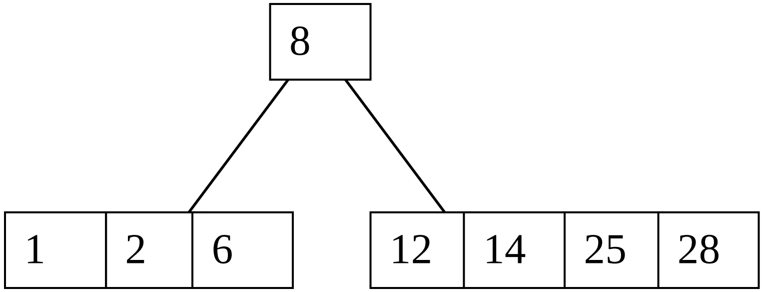
- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root



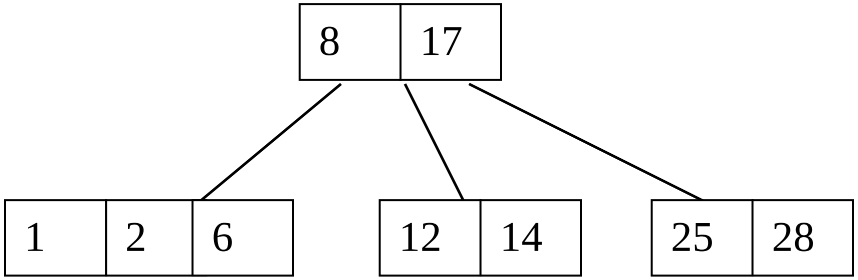
Constructing a B-tree (contd.)

~~1 12 8 2 25~~ 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

6, 14, 28 get added to the leaf nodes:



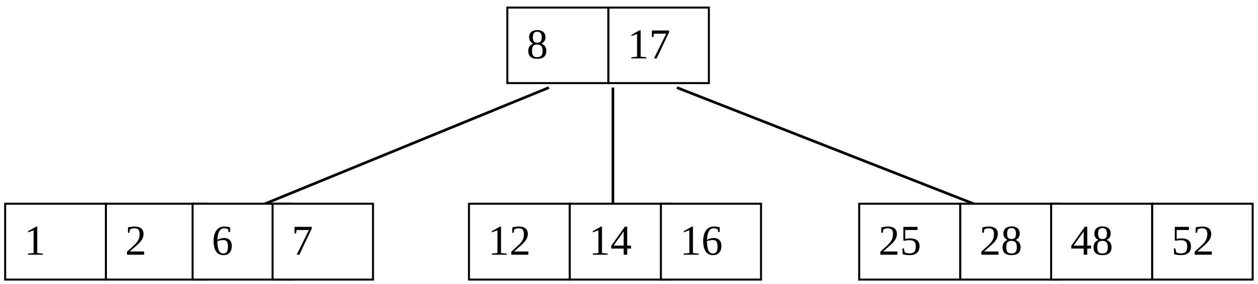
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



Constructing a B-tree (contd.)

~~1 12 8 2 25 6 14 28 17~~ 7 52 16 48 68 3 26 29 53 55 45

7, 52, 16, 48 get added to the leaf nodes

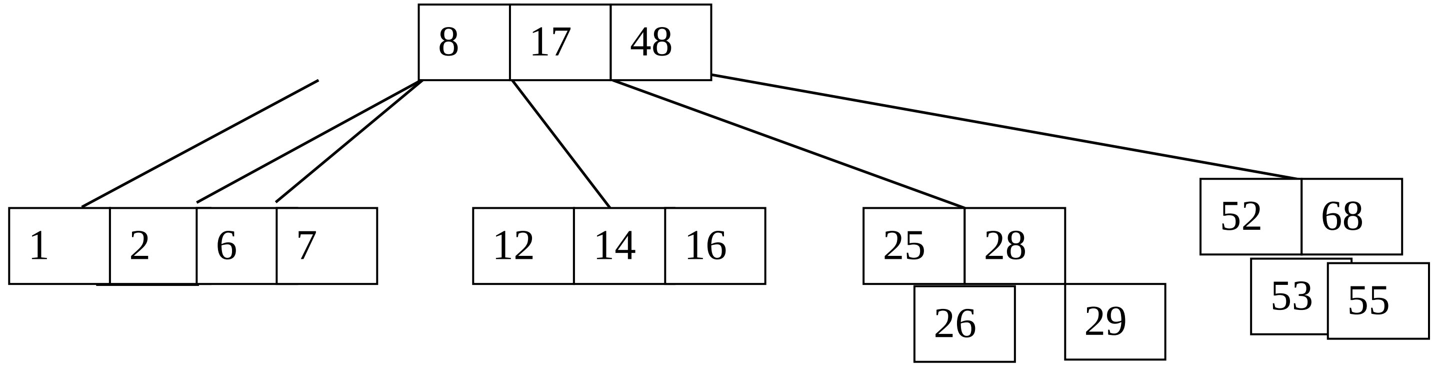


Adding 68 causes us to split the right most leaf, promoting 48 to the root

Constructing a B-tree (contd.)

~~1 12 8 2 25 6 14 28 17 7 52 16 48 68~~ 3 26 29 53 55 45

adding 3 causes us to split the left most leaf, promoting 3 to the root;

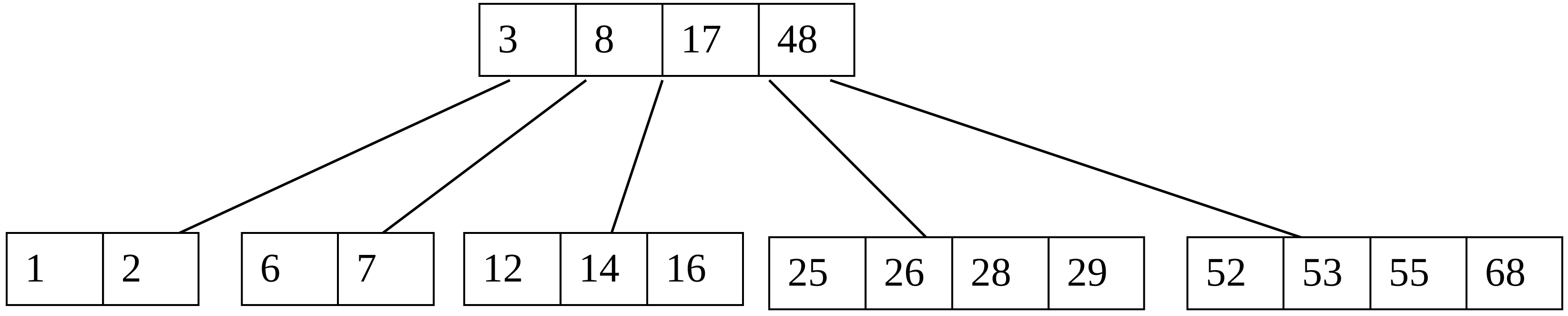


26, 29, 53, 55 then go into the leaves

Constructing a B-tree (contd.)

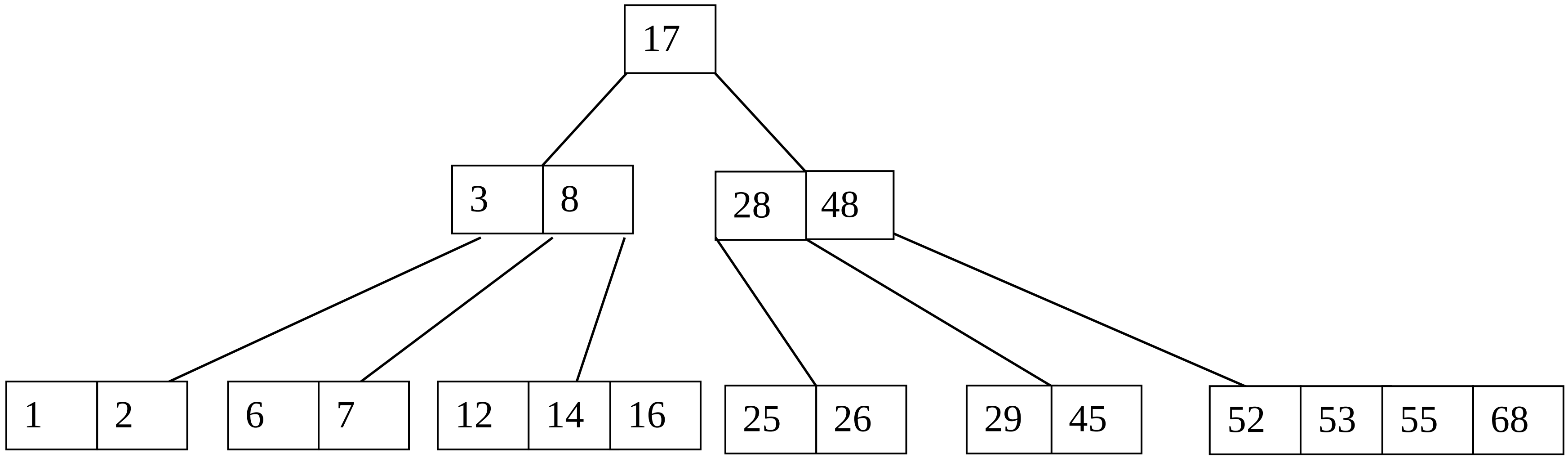
~~1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55~~ 45

Adding 45 causes a split of a leaf and promoting 28 to the root then causes the root to split



Final B-tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

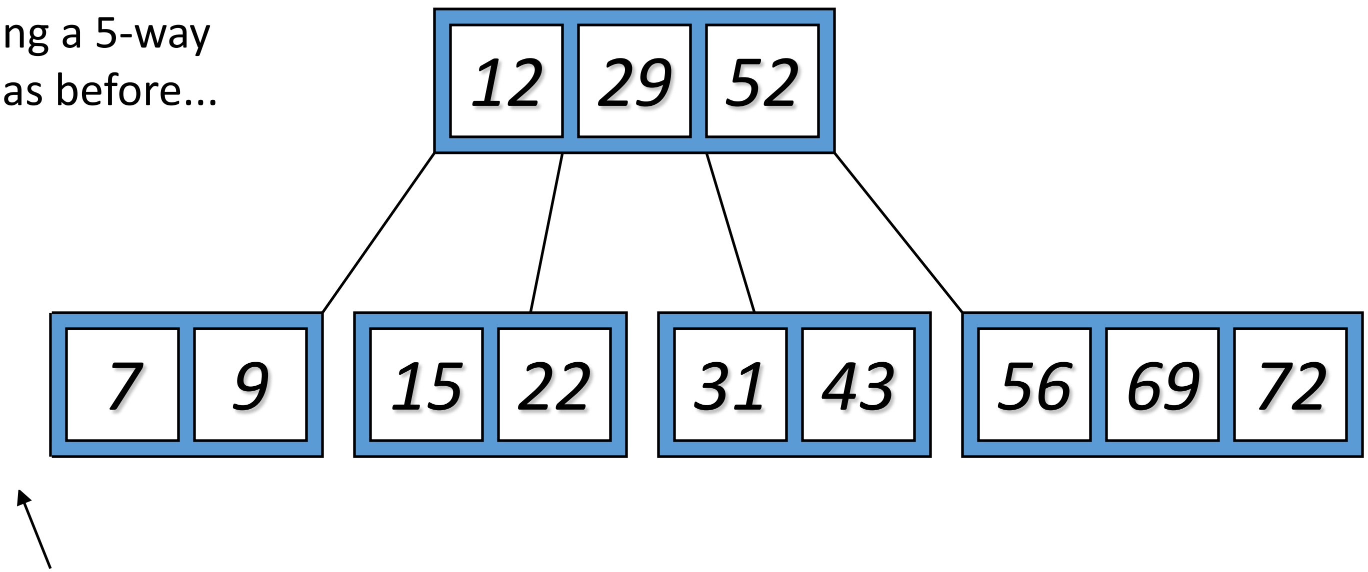


Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

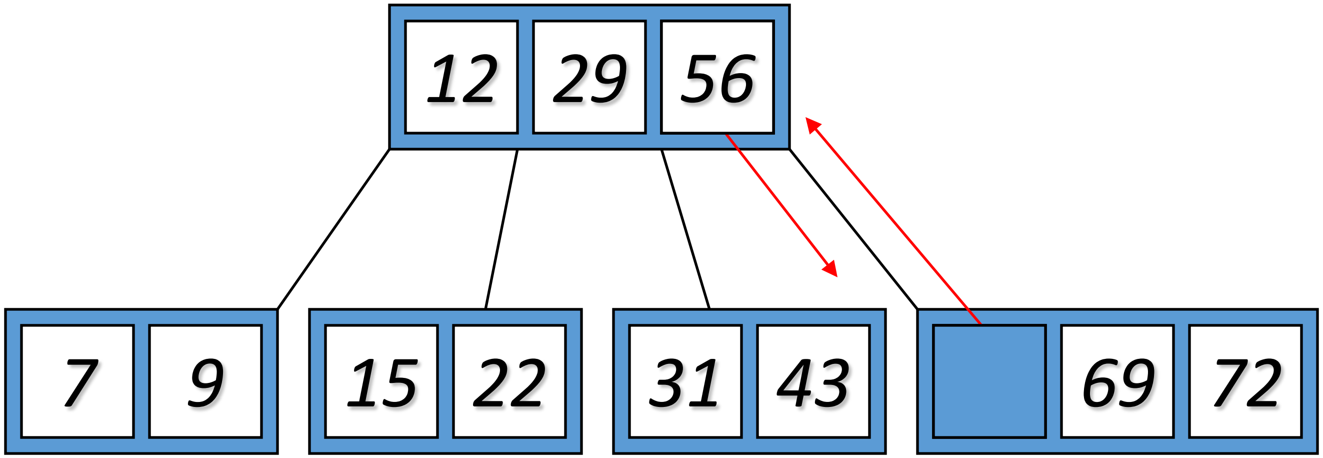
Case 1: Simple leaf deletion

Assuming a 5-way B-Tree, as before...

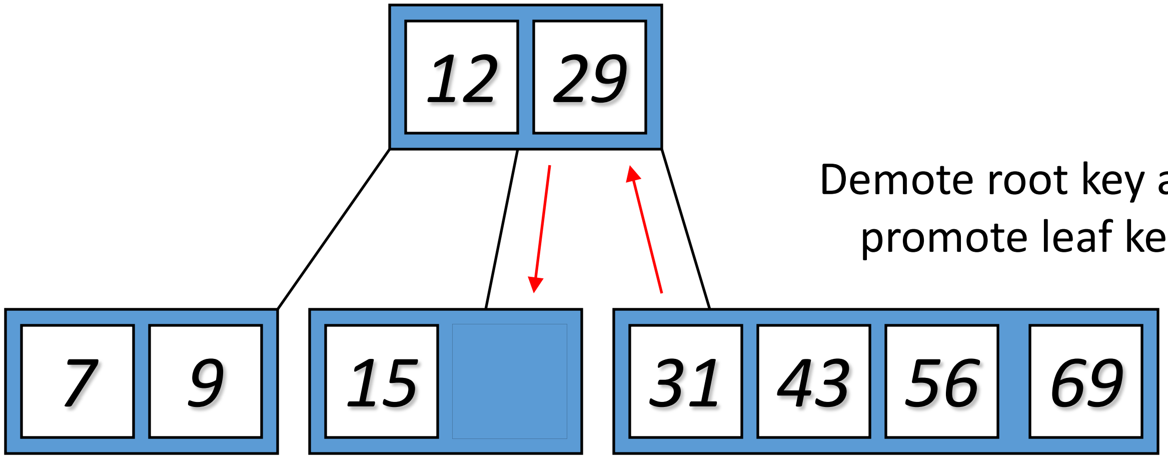


Delete 2: Since there are enough keys in the node, just delete it

Case 2: Simple non-leaf deletion

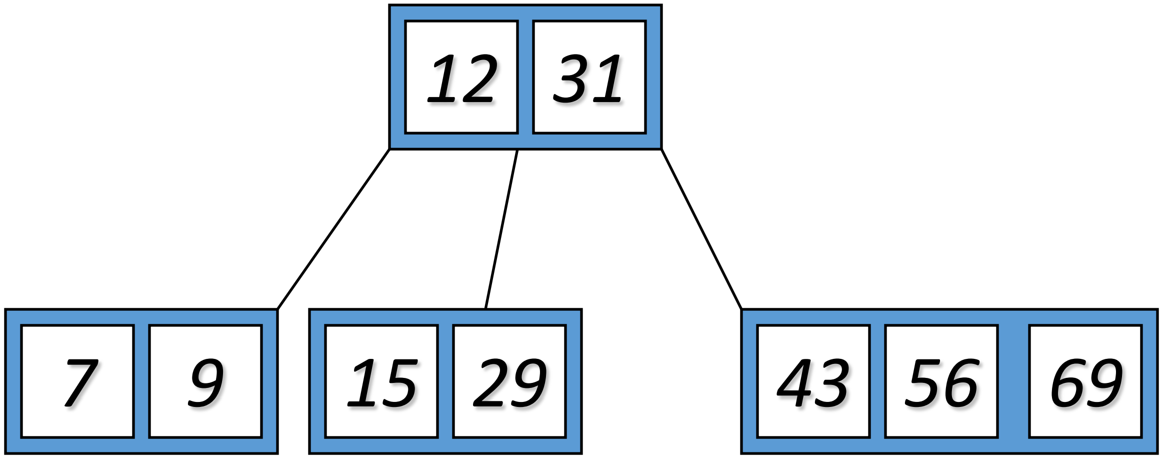


Case 3: Enough siblings

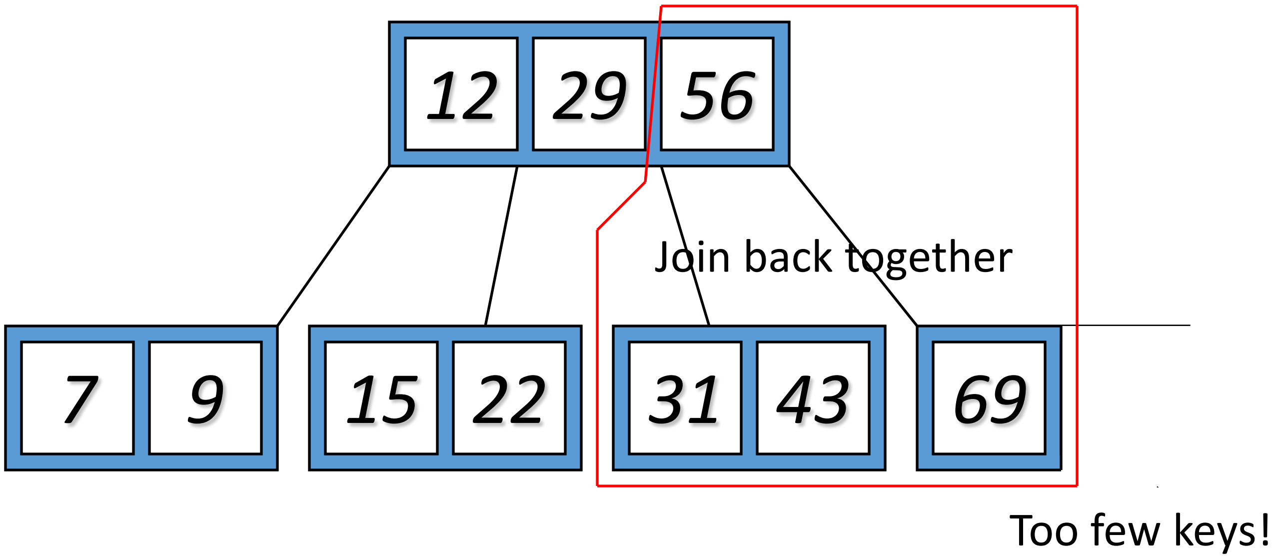


Demote root key and
promote leaf key

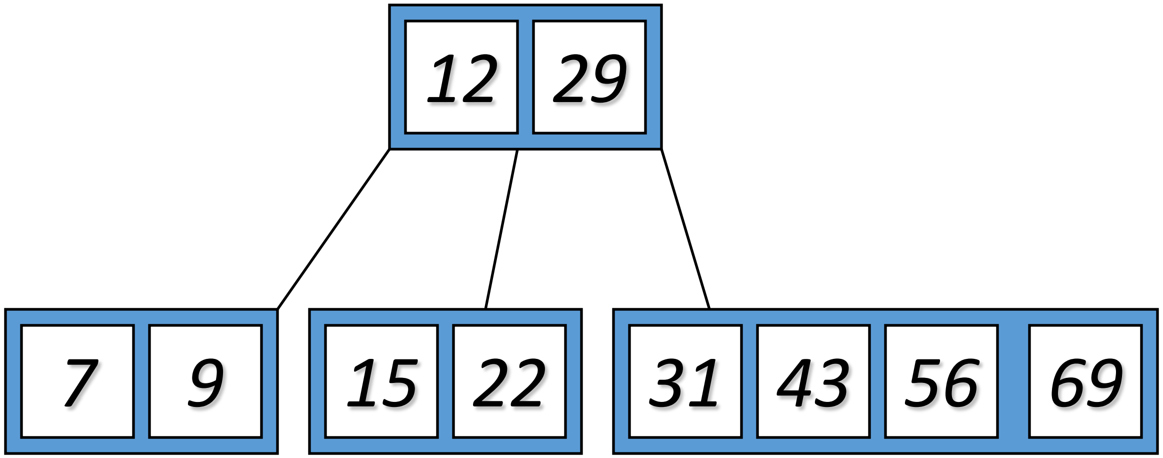
Case 3: Enough siblings



Case 4: Too few keys



Case 4: Too few keys



Removal from a B-tree

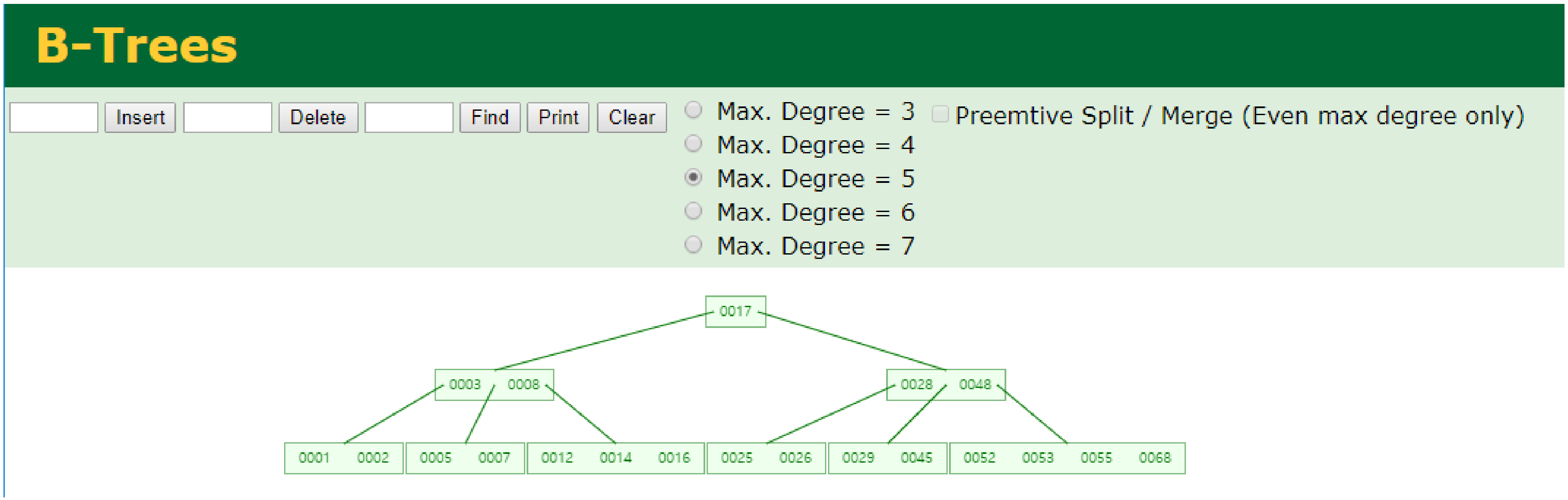
- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- Case 1: the key is already in a leaf node
  - removing it
  - Case 1-1 leaf node to have too few keys
    - simply remove the key to be deleted.
- Case 2:the key is *not* in a leaf
  - it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf
  - we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - Case 3: one of them has more than the min. number of keys
    - we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - Case 4: neither of them has more than the min. number of keys
    - the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key)
    - the new leaf will have the correct number of keys;
    - if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

Constructing a B-tree (contd.)

https://www.cs.usfca.edu/~galles/visualization/BTree.html



Analysis of B-Trees

- The maximum number of items in a B-tree of order *m* and height *h*:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)]\ (m - 1) = \mathbf{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take *m* = 3, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only *prioritise* (not order) the keys

- Multi-way trees
  - B-Trees can be *m*-way, they can have any (odd) number of children
  - One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

Huffman coding

Huffman encoding

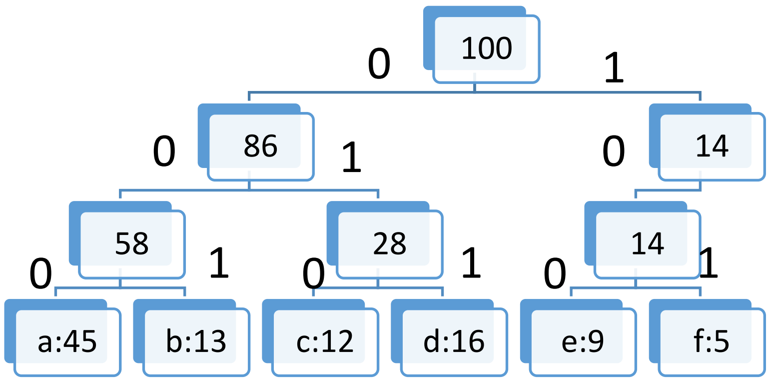
- The Huffman encoding algorithm is a greedy algorithm
- Given the percentage each character appears in a corpus, determine a v ariable-bit pattern for each char.
- You always pick the two smallest percentages to combine.

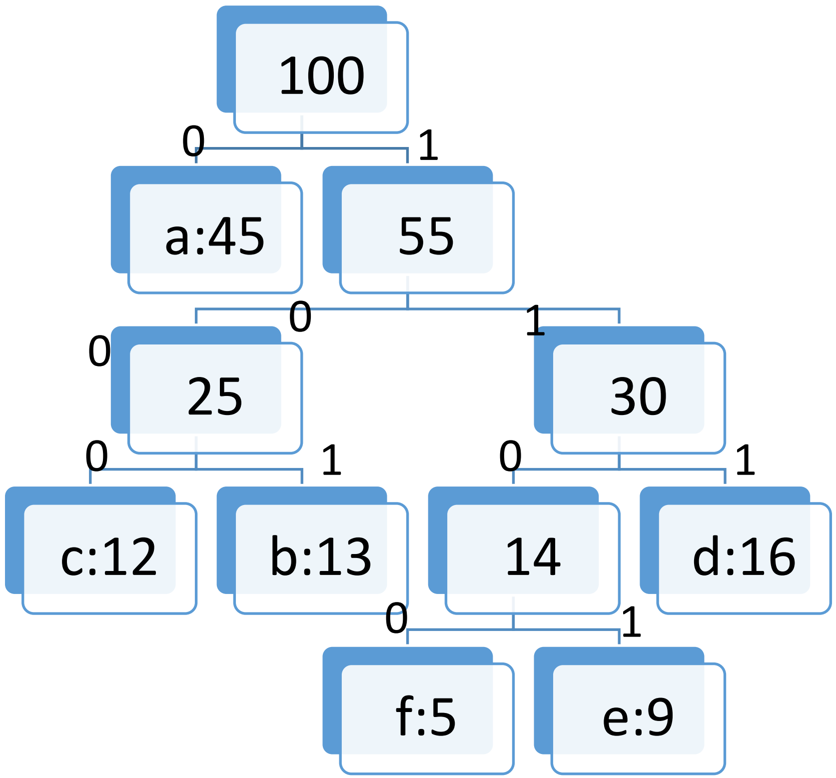
- Example

	a	b	c	d	e	f
frequency	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
Variable-length code	0	101	100	111	1101	1100

- Fixed: (45+13+12+16+9+5)*3=300 bits
- Various: $\sum freq \times code\_length$
  $= 45 \times 1 + (13 + 12 + 16) \times 3 + (9 + 5) \times 4 = 224 \text{ bits}$

Prefix code





- Prefix code:
- Example: 001011101
- The solution found doing this is an optimal solution.
- The resulting binary tree is a **full tree**.
- Cost of Tree T $B(T) = \sum_{c \in C} c.freq \times d_T(c)$

Algorithm

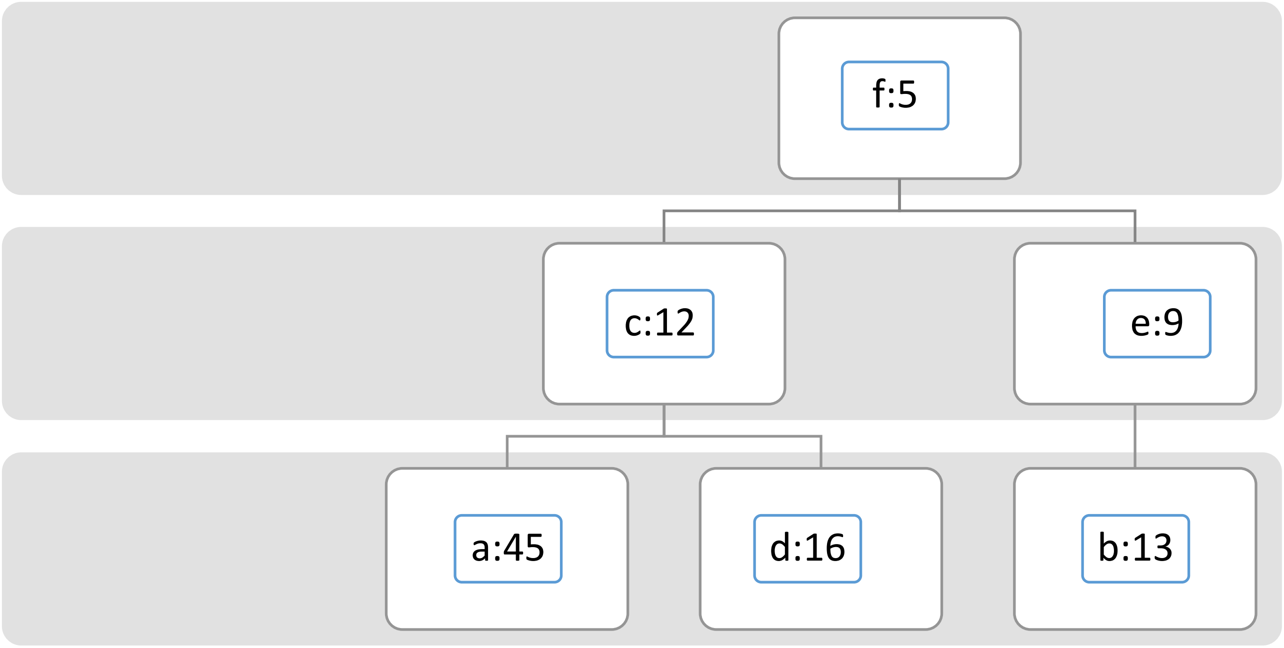
1. Create a leaf node for each unique character and build a min heap of all leaf nodes.
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies.
4. Repeat steps#2 and #3 until the heap contains only one node.

Example

- Given array of character and frequencies

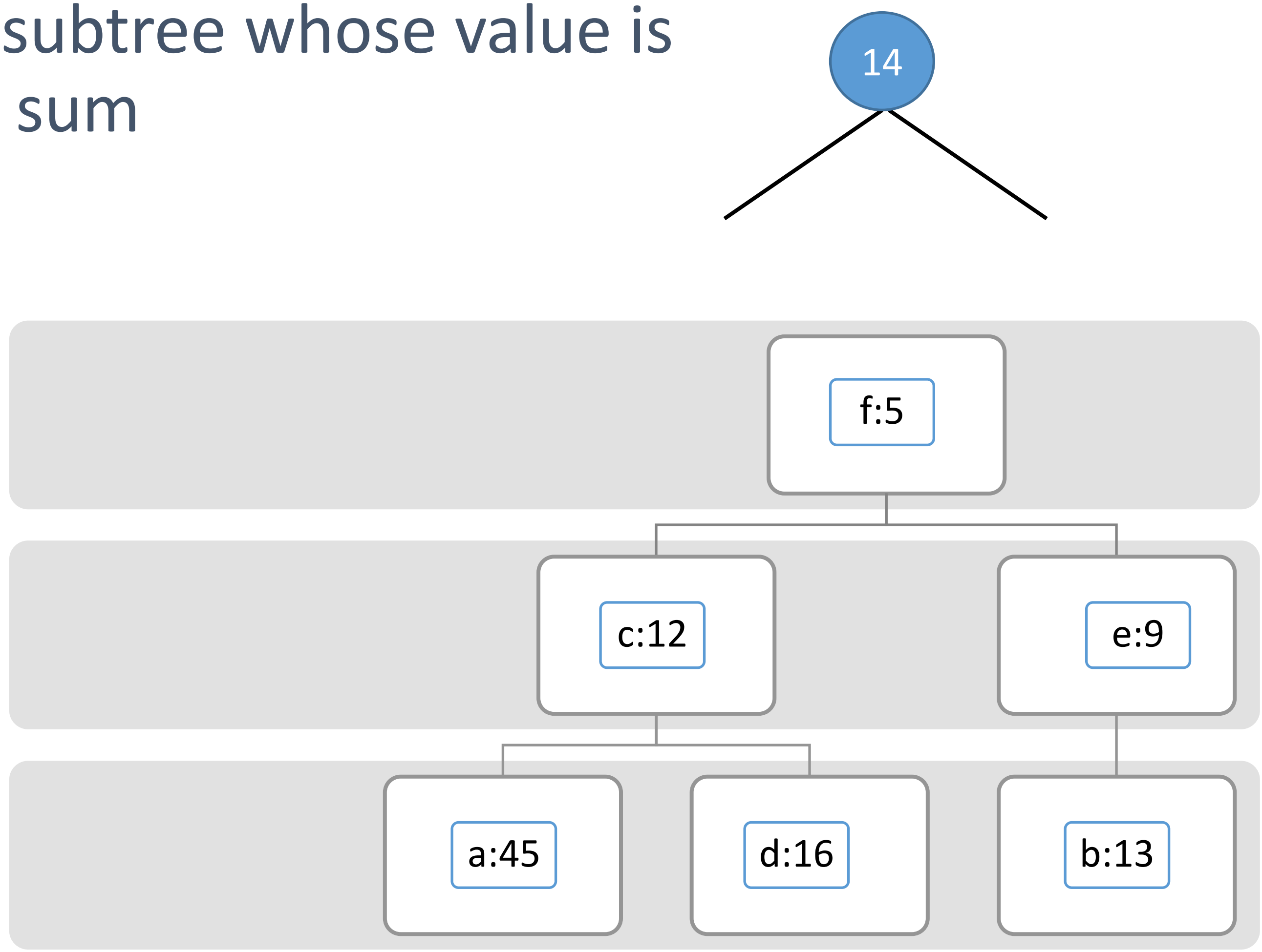
a:45	b:13	c:12	d:16	e:9	f:5

- Create a min Heap



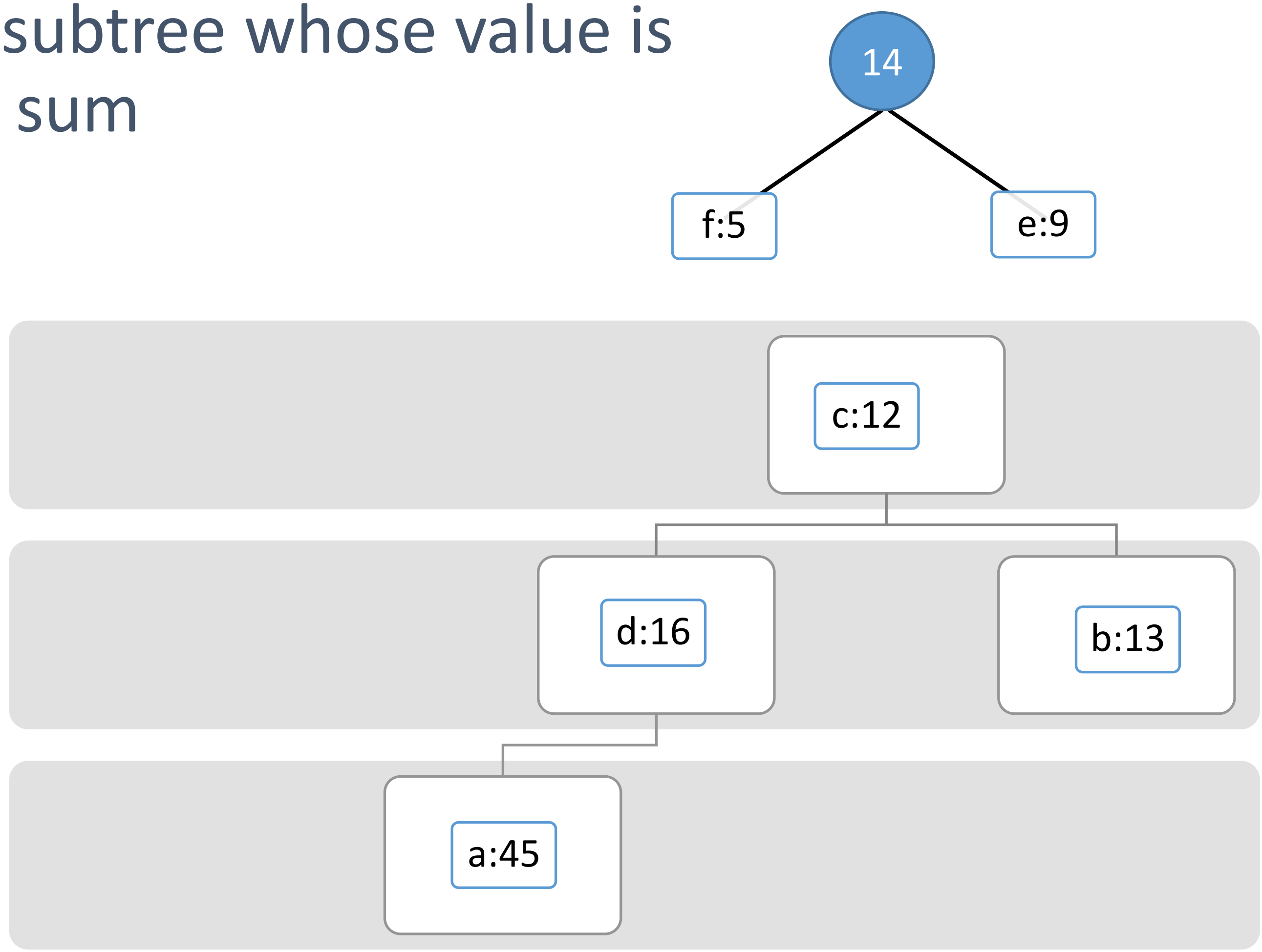
Example

- Extract two nodes and create new subtree whose value is their sum



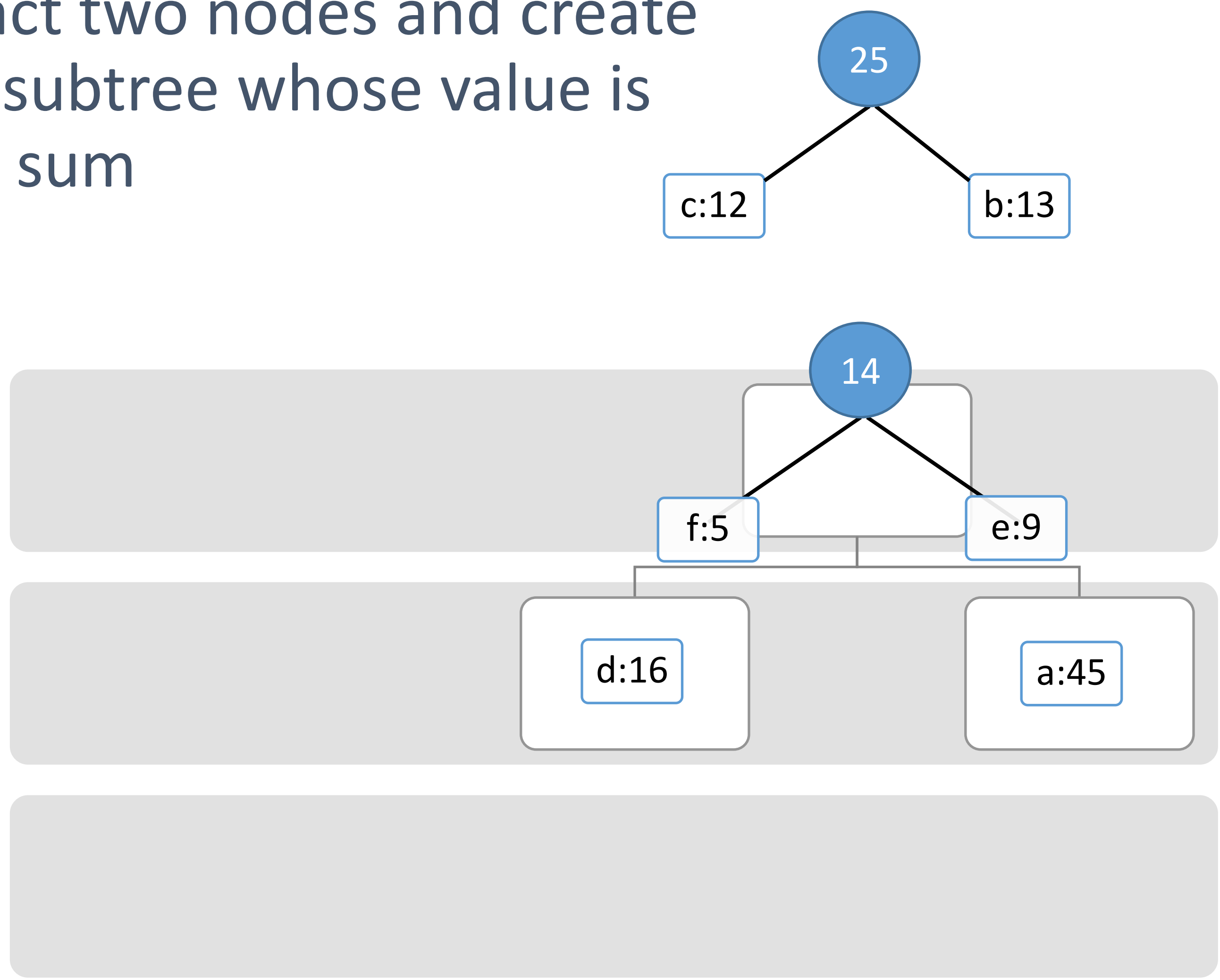
Example

- Extract two nodes and create new subtree whose value is their sum



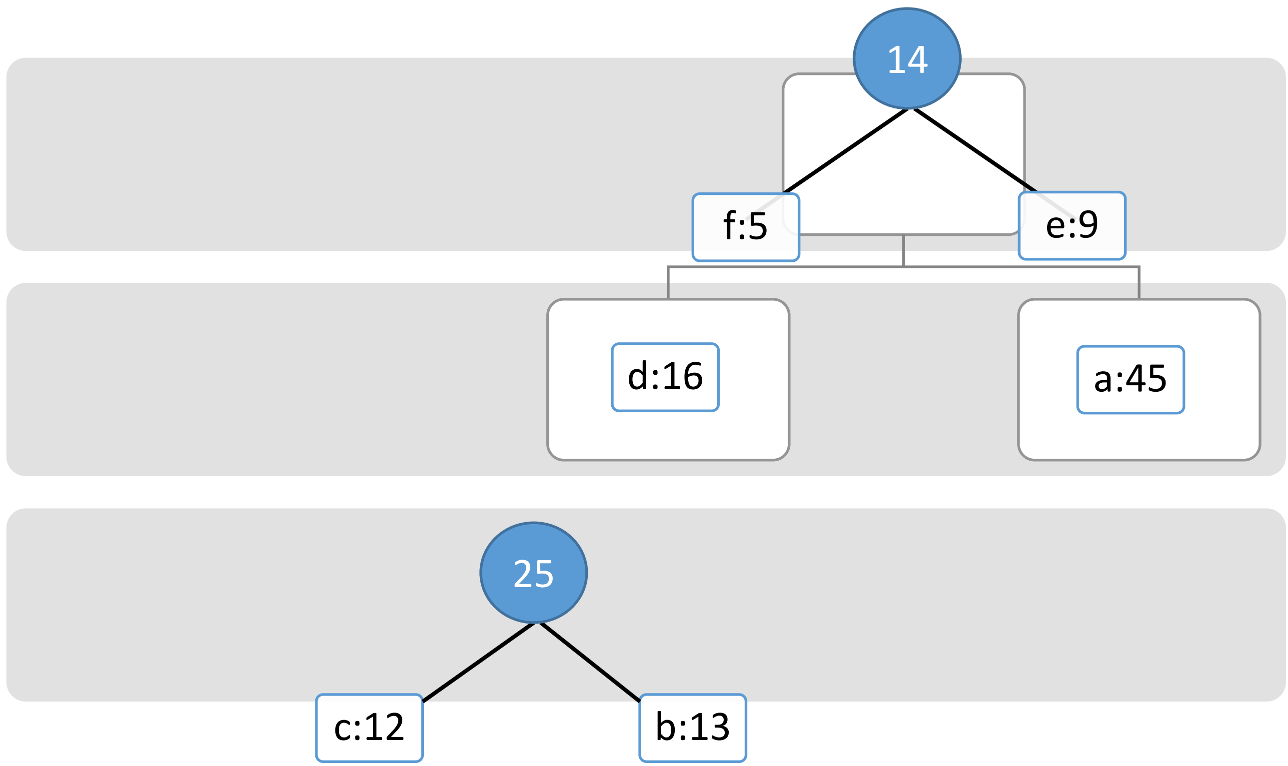
Example

- Extract two nodes and create new subtree whose value is their sum



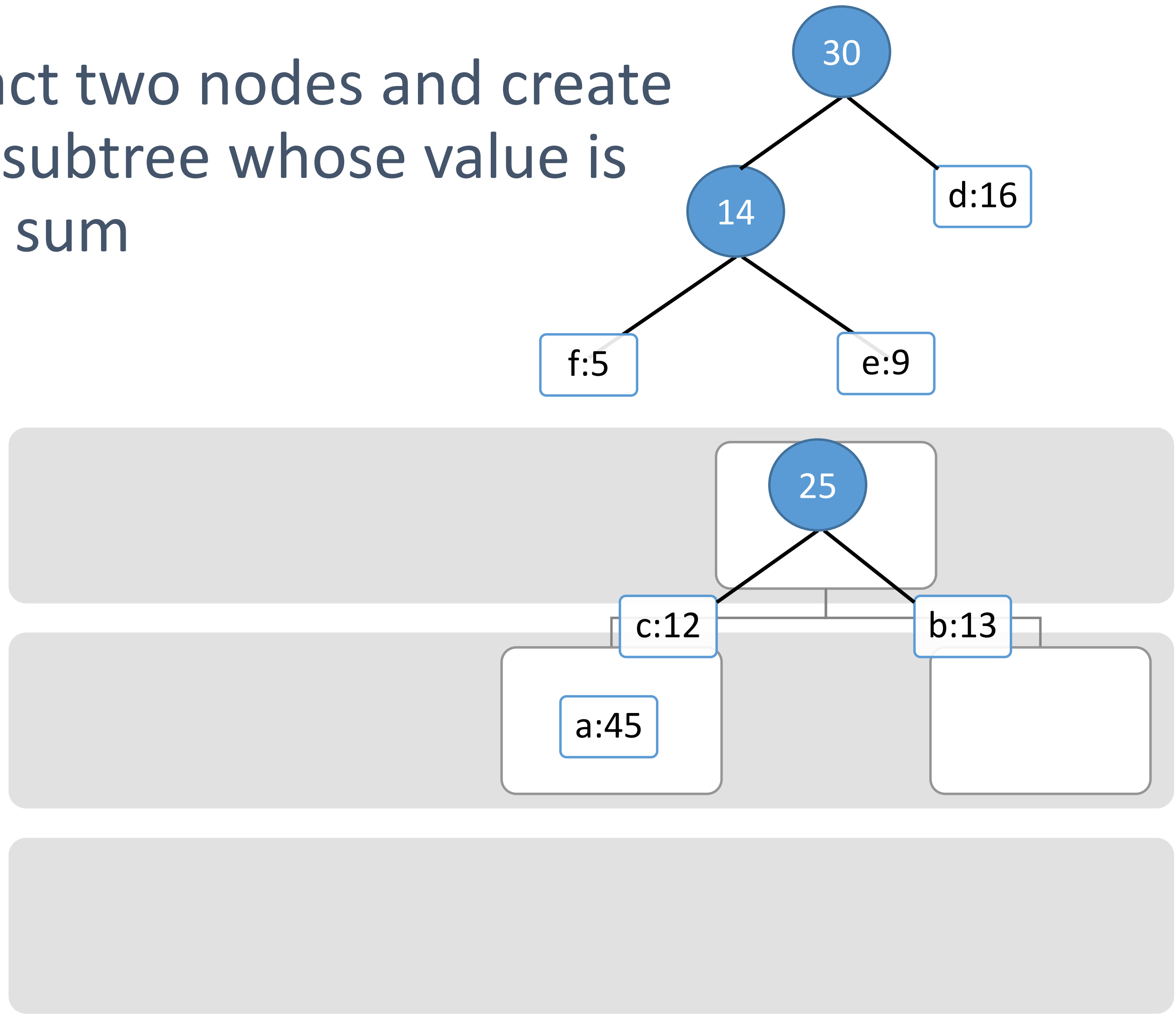
Example

- Extract two nodes and create new subtree whose value is their sum



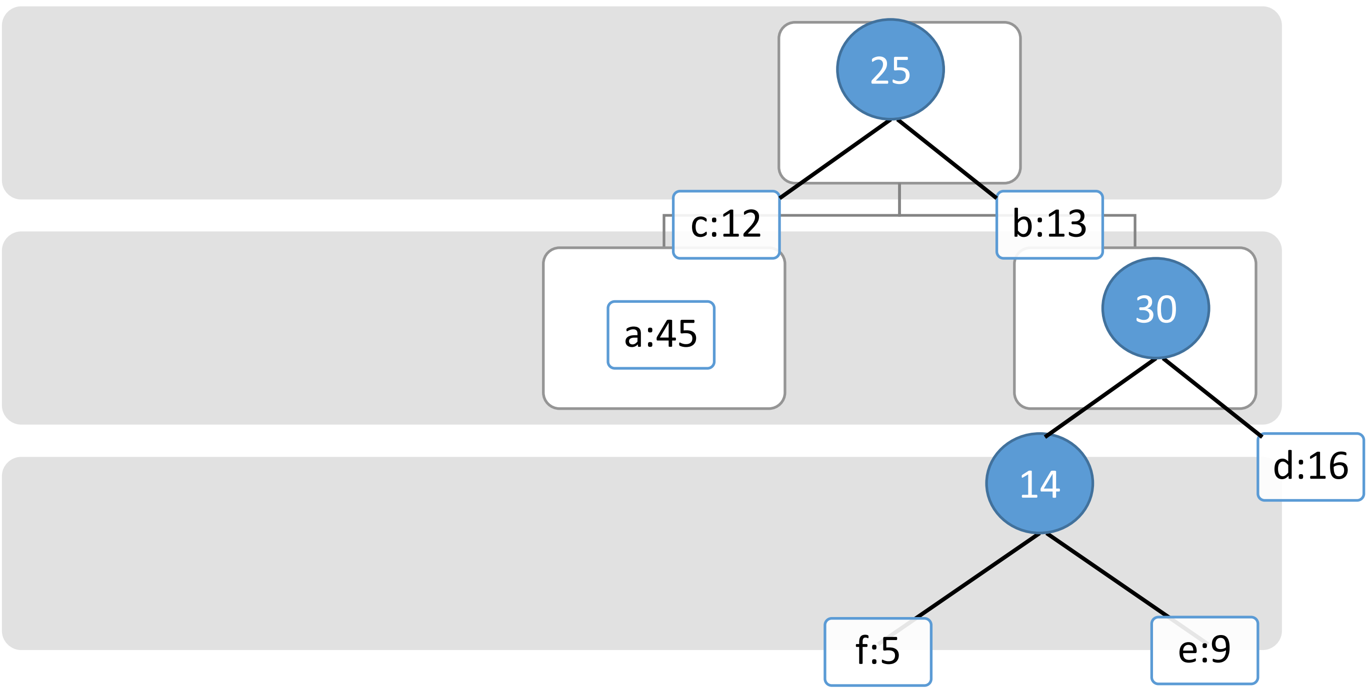
Example

- Extract two nodes and create new subtree whose value is their sum



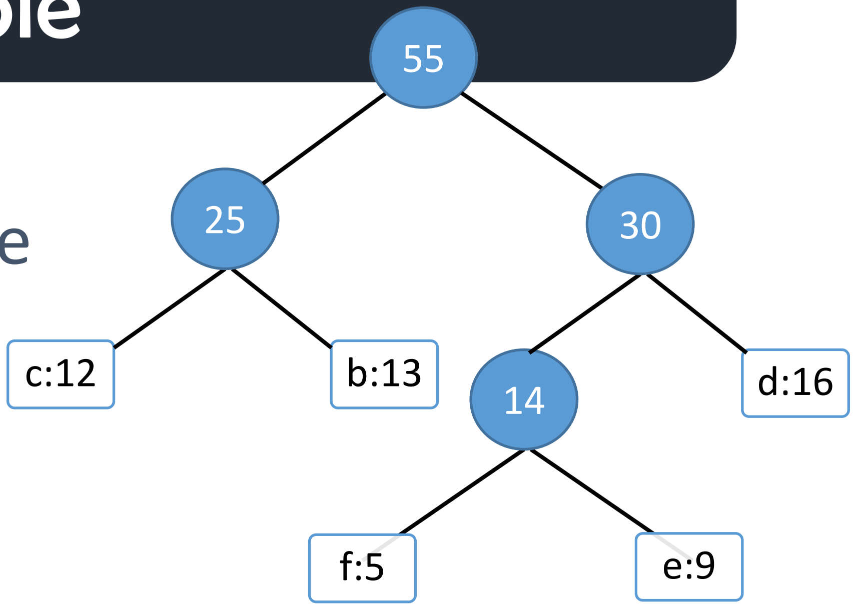
Example

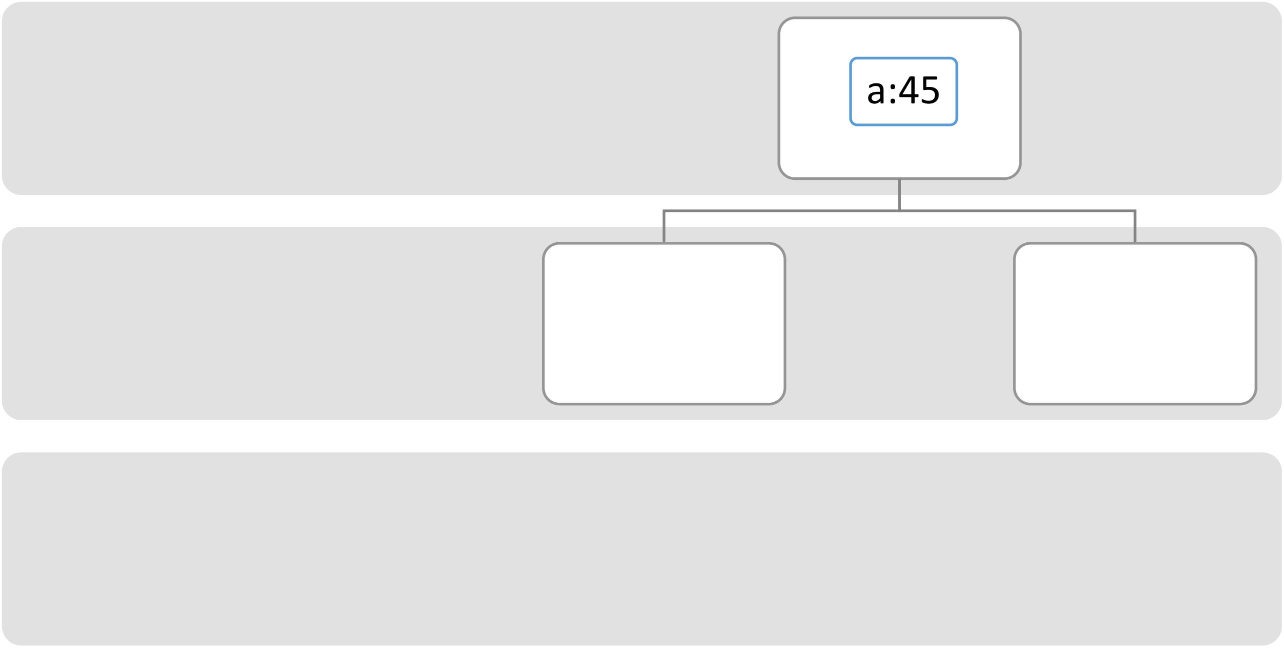
- Extract two nodes and create new subtree whose value is their sum



Example

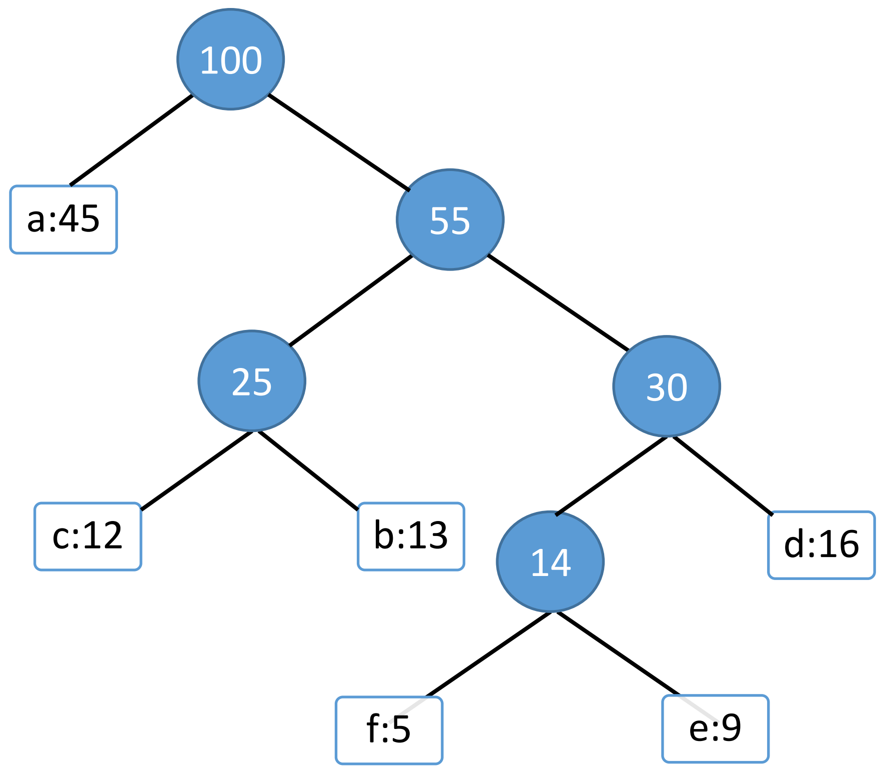
- Extract two nodes and create new subtree whose value is their sum





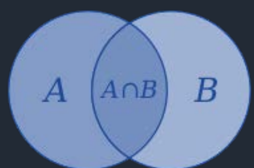
Example

- Repeat the previous steps until the heap contains only one node.



Analysis

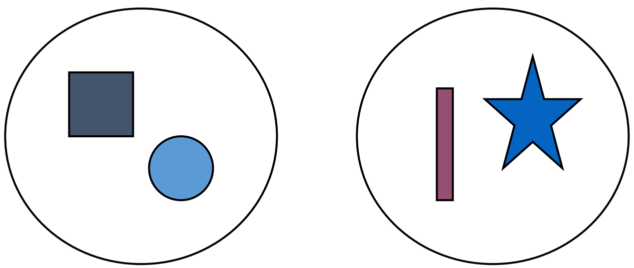
- The algorithm typically makes (approximately) n choices for a problem of size n
  - (The first or last choice may be forced)
- Hence the expected running time is:
  O(n * O(choice(n))), where choice(n) is making a choice a mong n objects
  - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
    - Therefore, coin counting is (n)
  - Huffman: Must sort n values before making n choices
    - Therefore, Huffman is O(n log n) + O(n) = O(n log n)



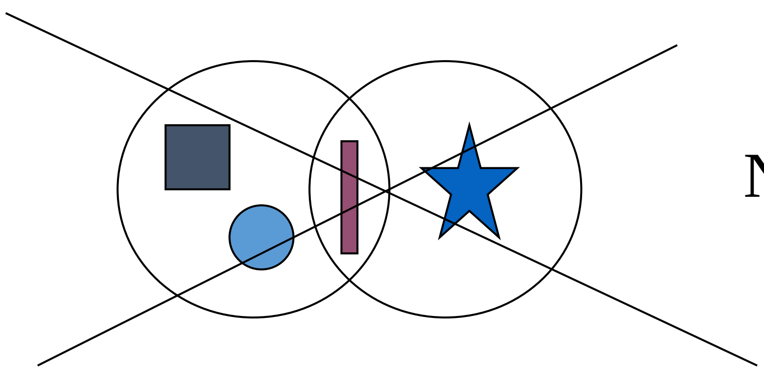
Sets

Disjoint sets

- Two sets A and B are disjoint if they have NO eleme nts in common. $(A \cap B = \emptyset)$



Disjoint Sets



NOT Disjoint Sets

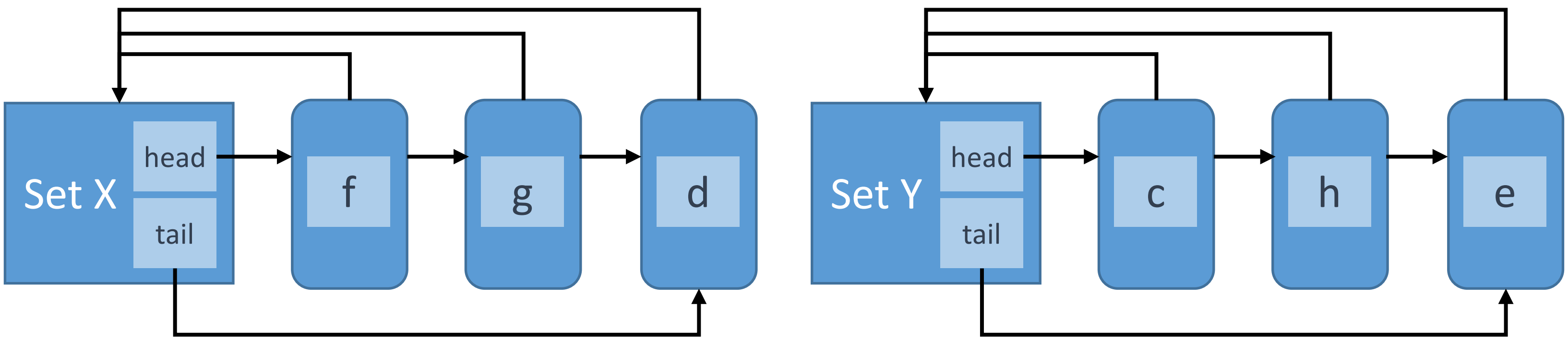
Data structures for disjoint sets

- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic (changing) sets.
  - Each set has a representative (member of the set).
  - Each element of a set is represented by an object (x).

- Operations
  - MAKE-SET(x): creates a new set with a single member p ointed to by x.
  - UNION(x,y): unites the sets that contain x, y into a single set.
  - FIND-SET(x): returns a pointer to the representative of th e set containing x.

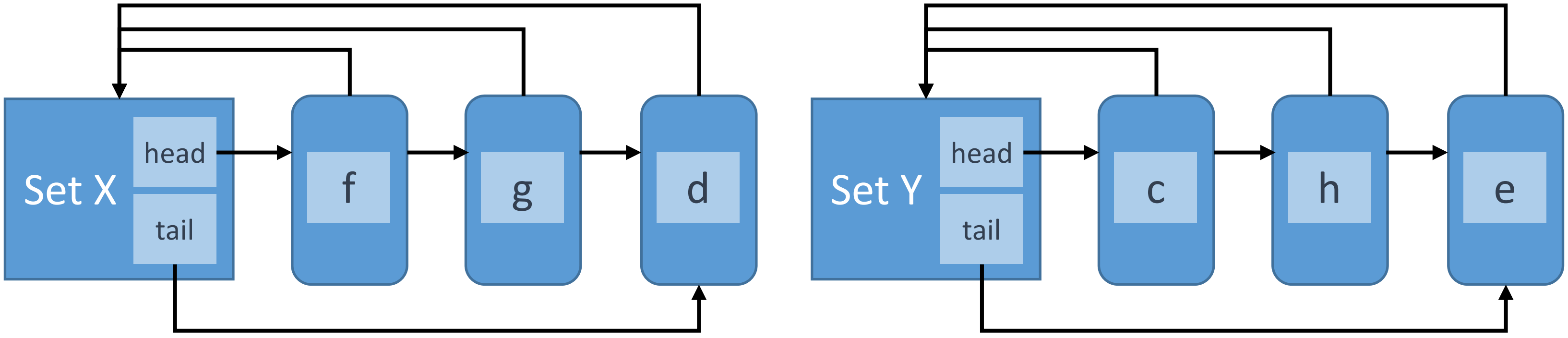
Linked list representation

- Make-Set: O(n)
- Find-Set: O(1)
- Union: O(n)



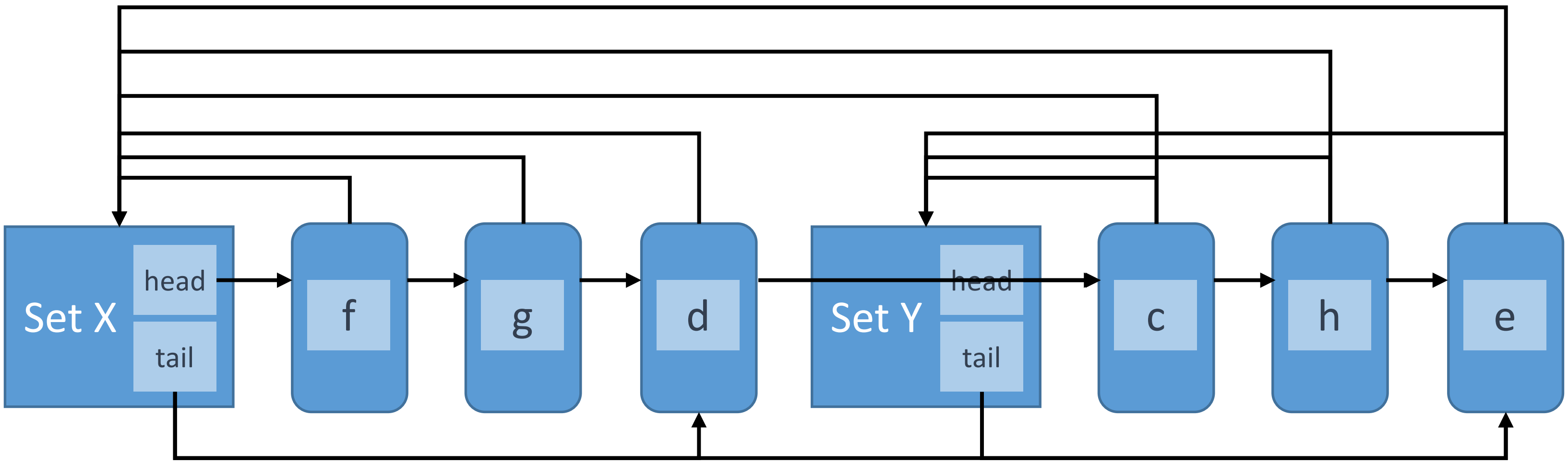
Linked list representation

- Make-Set: O(1)
- Find-Set: O(1)
  - Example: Find-Set(g)
- Union: O(n)



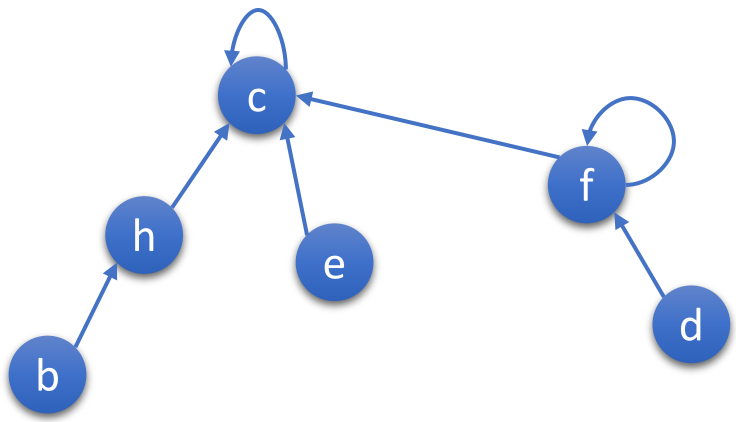
Linked list representation

- Make-Set: O(n)
- Find-Set: O(1)
- Union: O(n)
  - Example: Union(g,e)



Forest representation

- Disjoint-set forest
- Union set S1 and S2, where $b \in S1$ and $d \in S2$
  - Example(b,d)

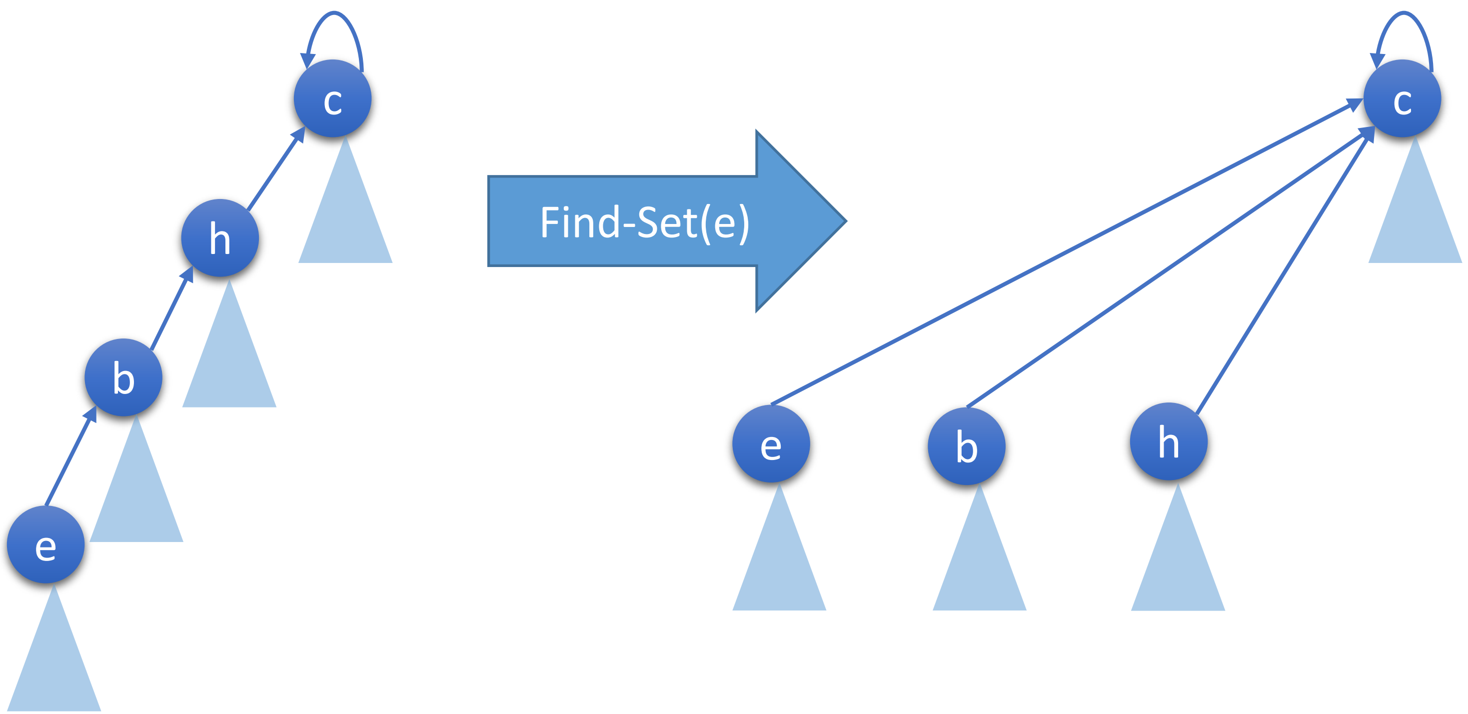


Find(b) = c
b → h → c

Find(d) = f
d → f

Heuristics to improve(Cont)

- Union by rank
- Path compression



Algorithm

- Make-set(X)
  - X.p=x
  - X.rank=0

- Union(x,y)
  - Link(find-set(x), find-set(y))

- Link(x,y)
  - If x.rank > y.rank
    - y.p = x
  - Else x.p = y
    - If x.rank == y.rank
      - y.rank = y.rank +1
- Find-set(x)
  - If x != x.p
    - X.p = find-set(x.p)
  - Return x.p

Integer array

- Array for data

b	**c**	**d**	**e**	**f**	**g**	**h**

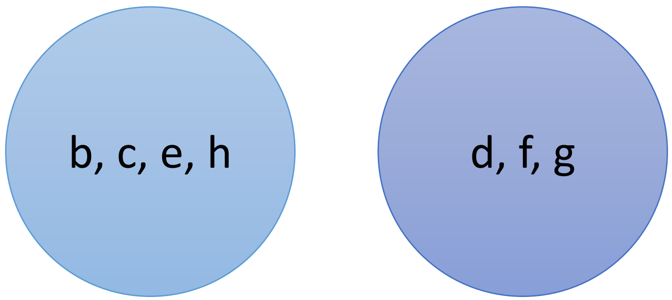
- Array for sets

1	**1**	**0**	**1**	**0**	**0**	**1**

0	**0**	**1**	**0**	**1**	**1**	**0**

  - After the union

1	**1**	**1**	**1**	**1**	**1**	**1**



Thanks