

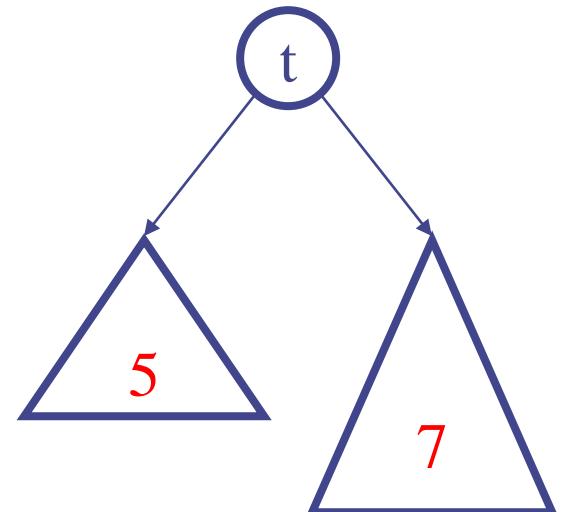
# AVL Trees

# Balance

Balance == height(left subtree) - height(right subtree)

- zero everywhere  $\Rightarrow$  perfectly balanced
- small everywhere  $\Rightarrow$  balanced enough

Balance between -1 and 1 everywhere  
 $\Rightarrow$  maximum height of  $\sim 1.44 \log n$



# AVL Tree

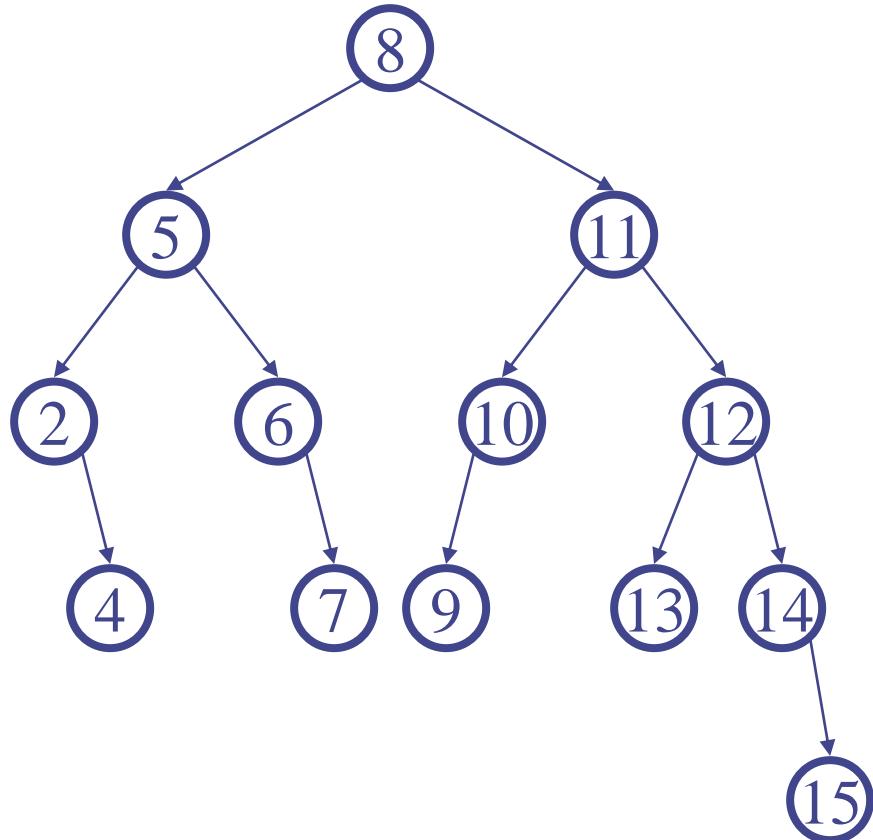
## Dictionary Data Structure

Binary search tree properties

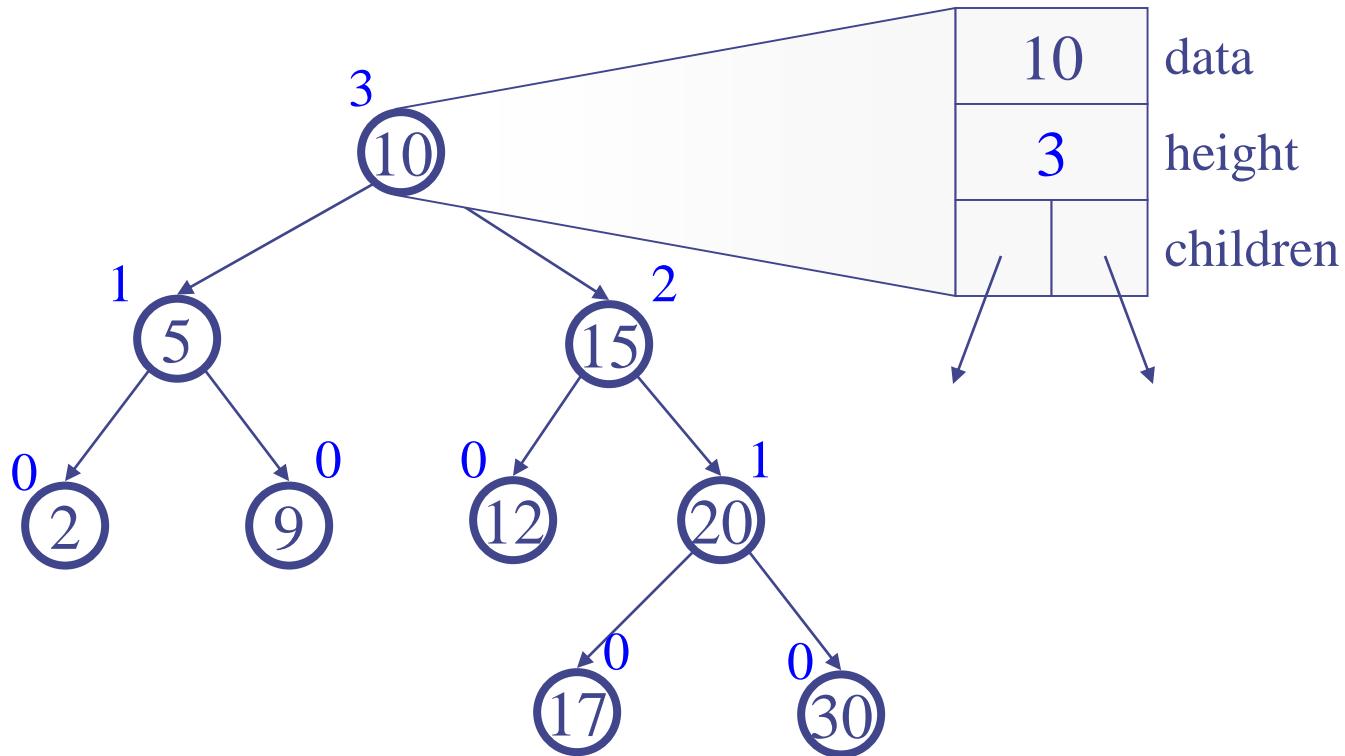
- binary tree property
- search tree property

Balance property

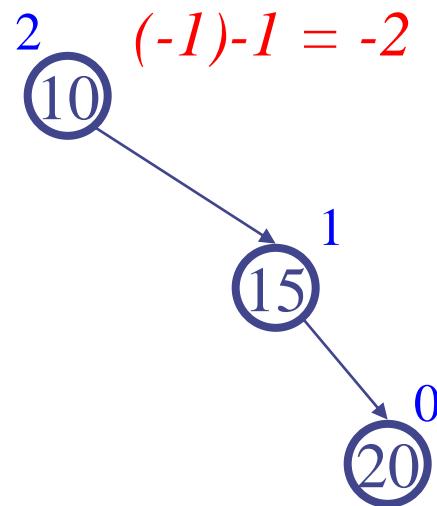
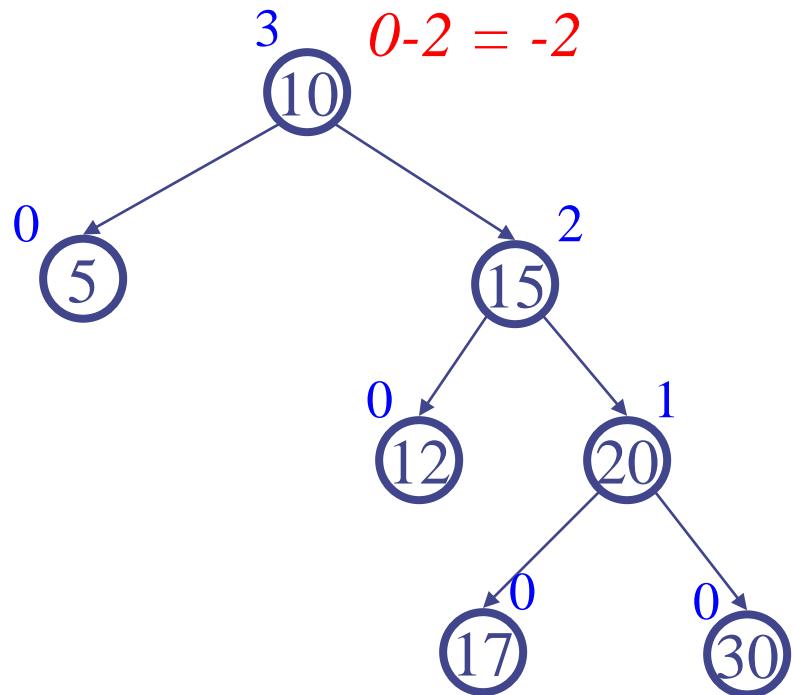
- balance of every node is:  
 $-1 \leq b \leq 1$
- result:
  - ◆ depth is  $\Theta(\log n)$



# An AVL Tree



# Not AVL Trees



Note:  $\text{height}(\text{empty tree}) == -1$

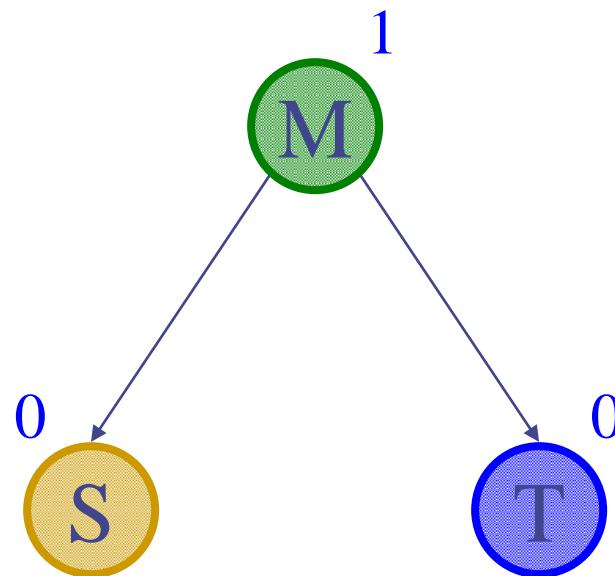
# Good Insert Case: Balance Preserved

Good case: insert **middle**, then **small**, then **tall**

Insert(**middle**)

Insert(**small**)

Insert(**tall**)



# Bad Insert Case #1: Left-Left or Right-Right Imbalance

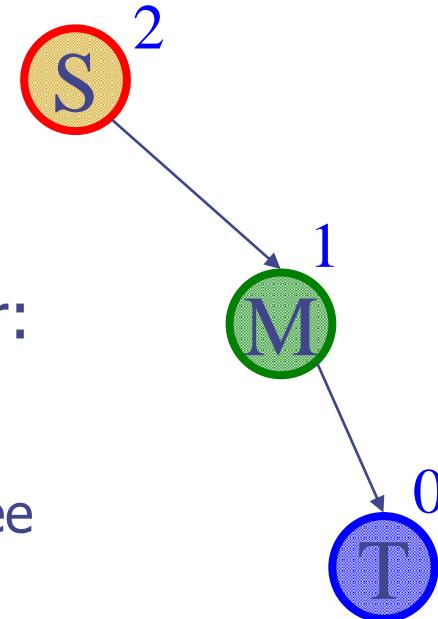
Insert(**small**)

Insert(**middle**)

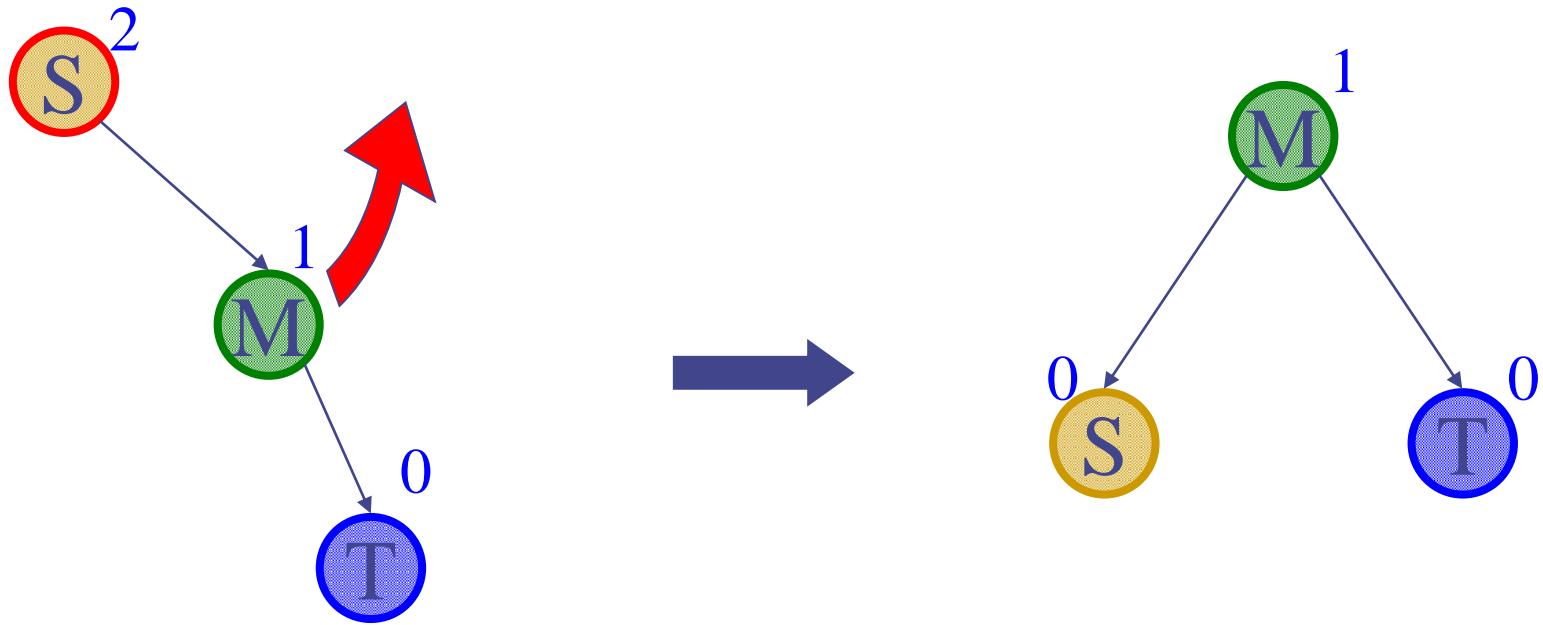
Insert(**tall**)

BC#1 Imbalance caused by either:

- Insert into **left** child's **left** subtree
- Insert into **right** child's **right** subtree



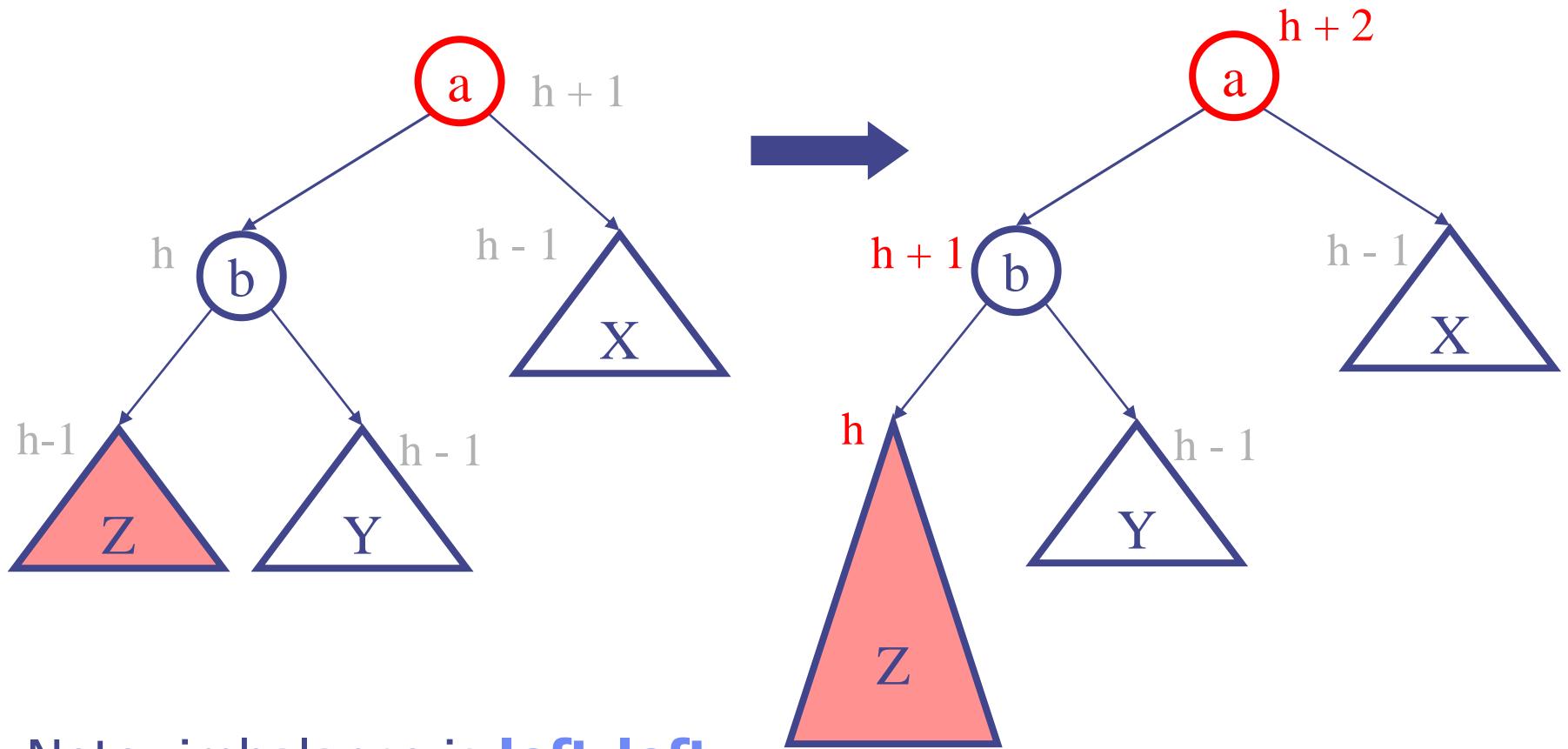
# Single Rotation



Basic operation used in AVL trees:

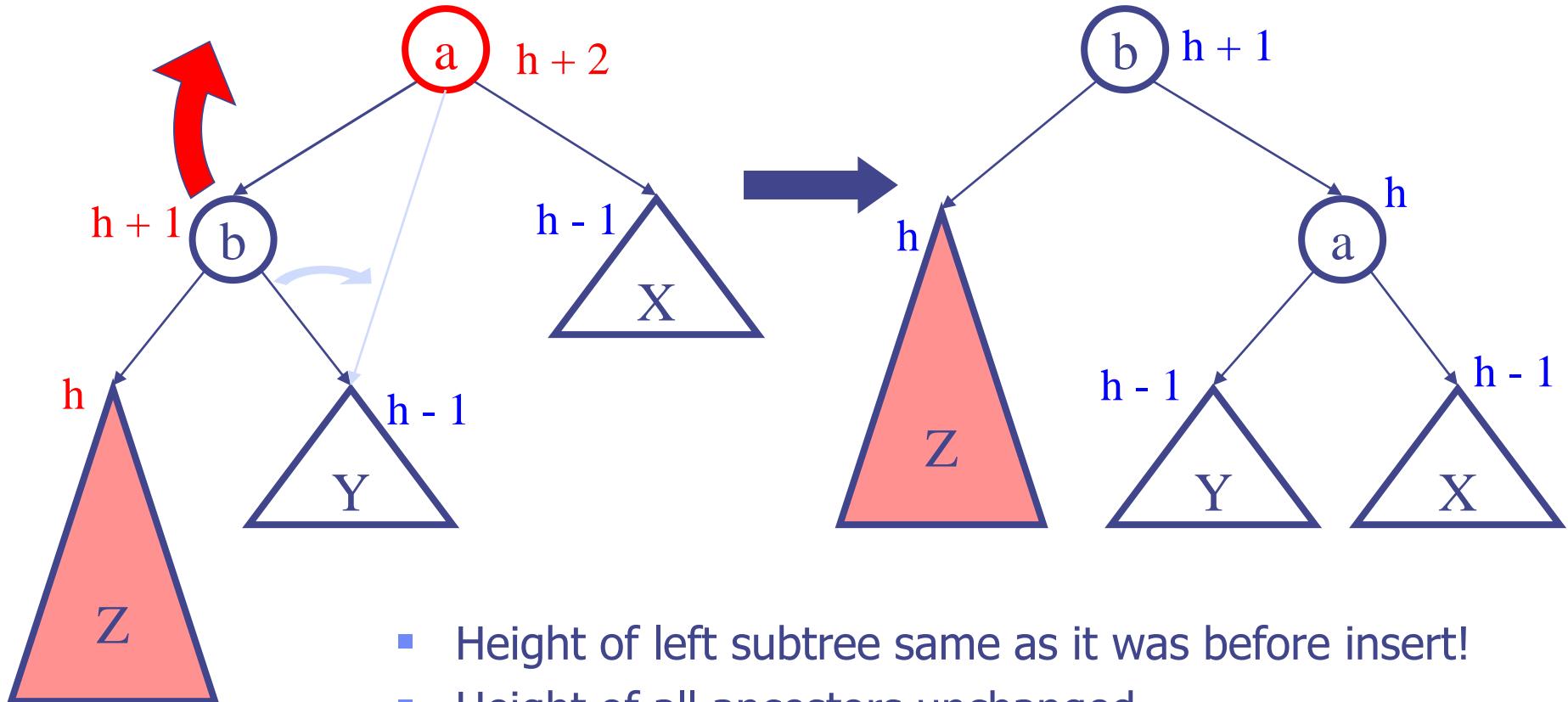
A right child could legally have its parent as its left child.

# General Bad Case #1



# Single Rotation

## Fixes Case #1 Imbalance



- Height of left subtree same as it was before insert!
- Height of all ancestors unchanged
  - *We can stop here!*

# Bad Insert Case #2: Left-Right or Right-Left Imbalance

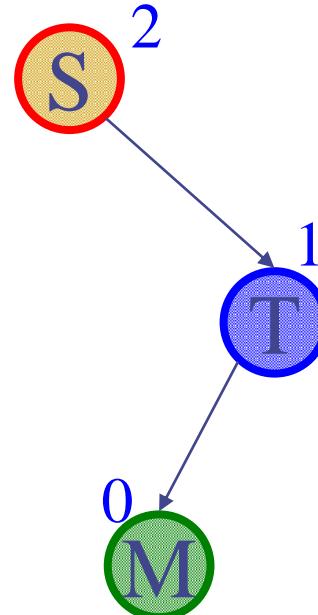
Insert(**small**)

Insert(**tall**)

Insert(**middle**)

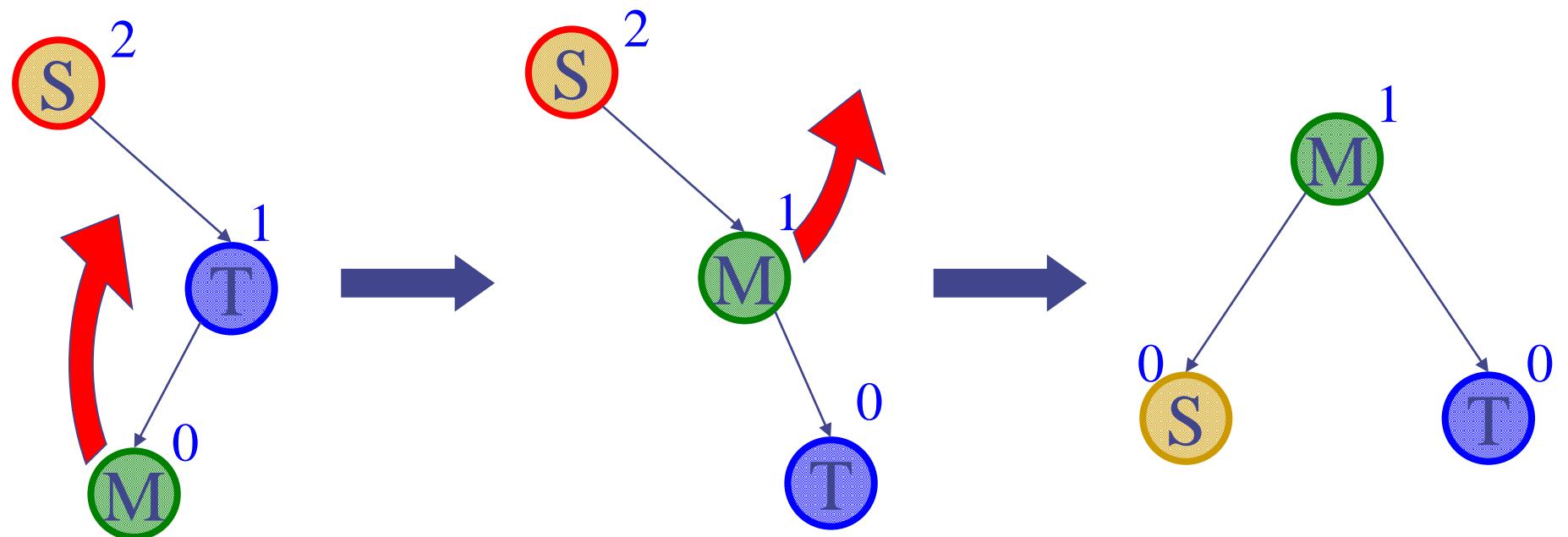
BC#2 Imbalance caused by either:

- Insert into **left** child's **right** subtree
- Insert into **right** child's **left** subtree

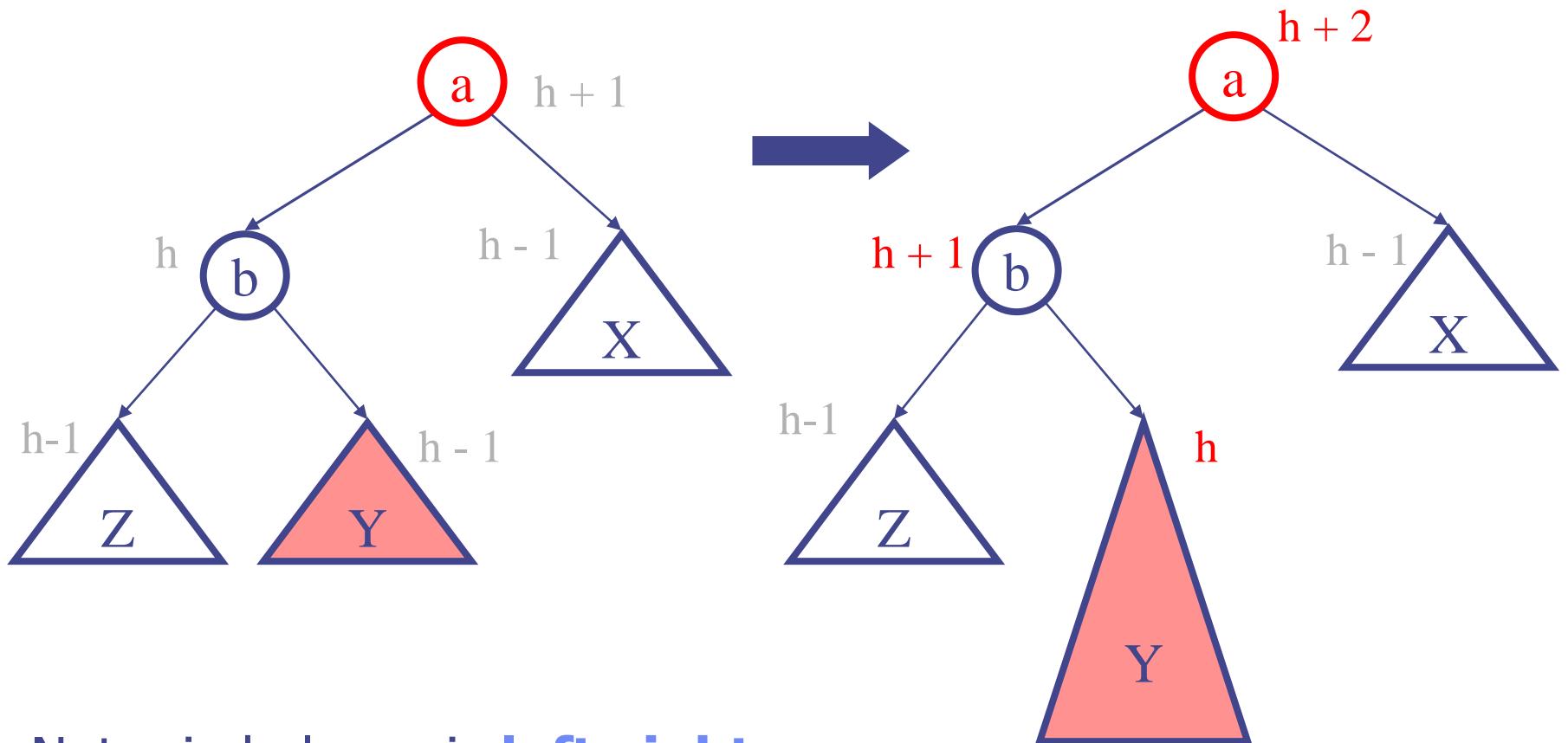


*Will a single rotation fix this?*

# Double Rotation



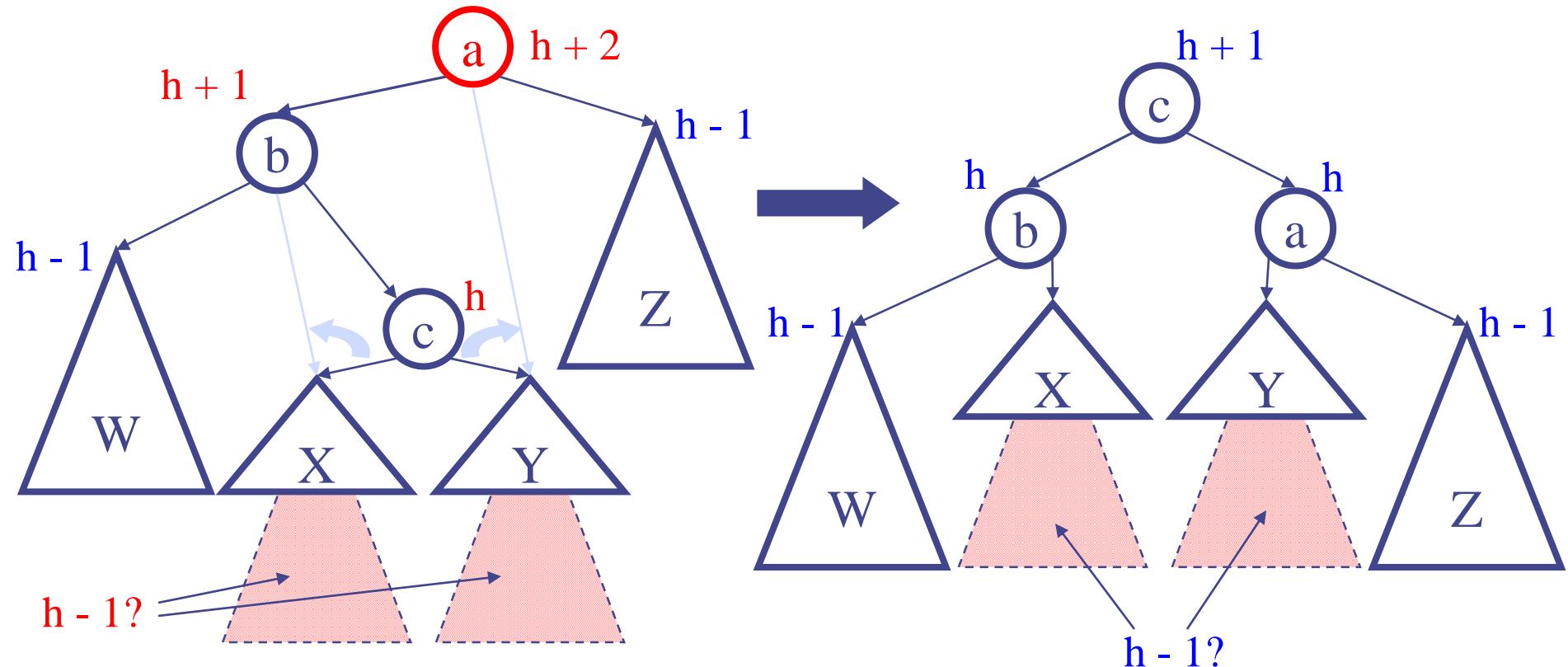
# General Bad Case #2



Note: imbalance is **left-right**

# Double Rotation

## Fixes Case #2 Imbalance



Initially: insert into either X or Y unbalances tree (root balance goes to 2 or -2)  
“Zig zag” to pull up c – restores root height to  $h+1$ , left subtree height to  $h$

# AVL Insert Algorithm

- Find spot for value
- Hang new node
- Search back up looking for imbalance
- If there is an imbalance:

case #1: Perform single rotation

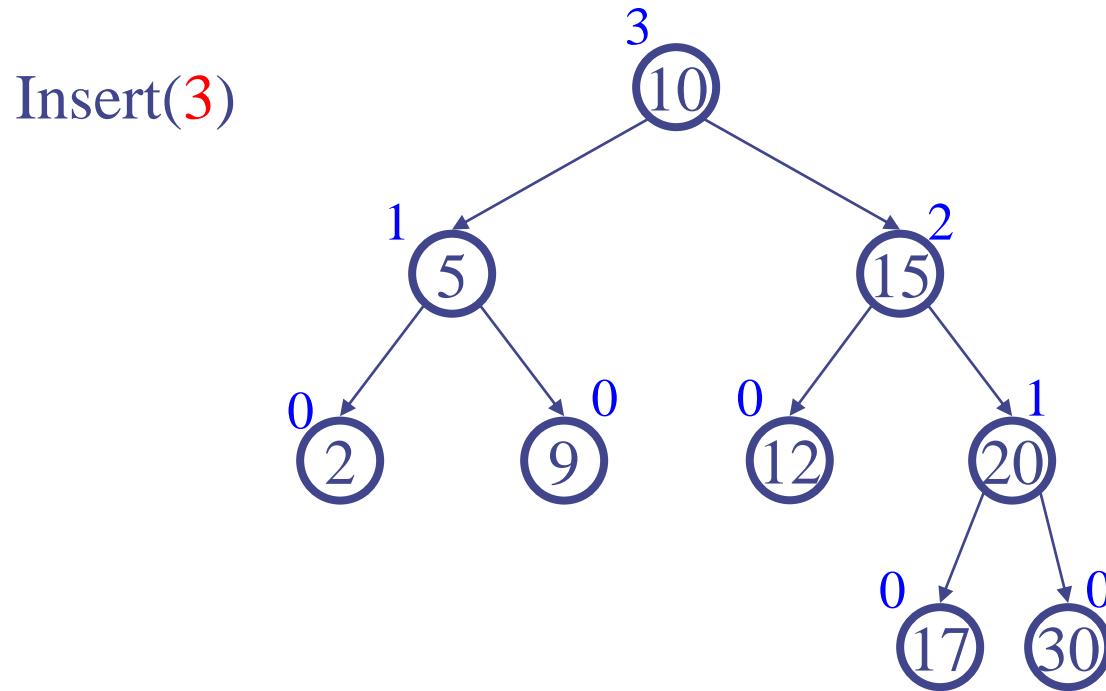


case #2: Perform double rotation

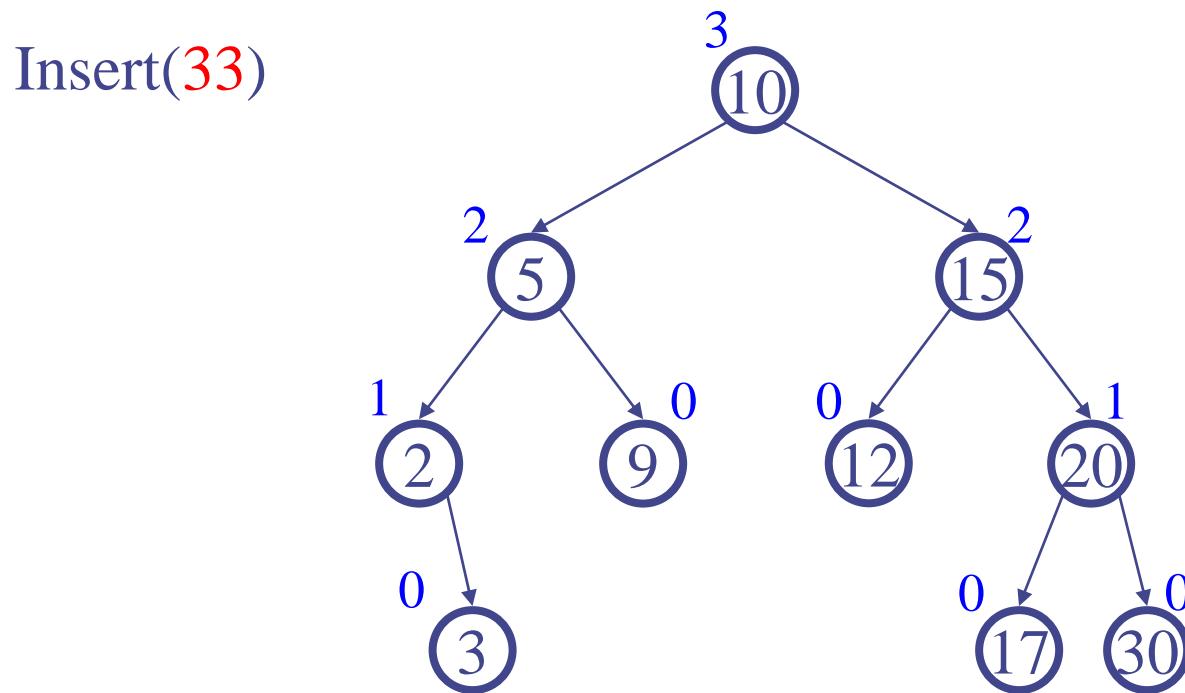


- *Done!*  
(There can only be one imbalance!)

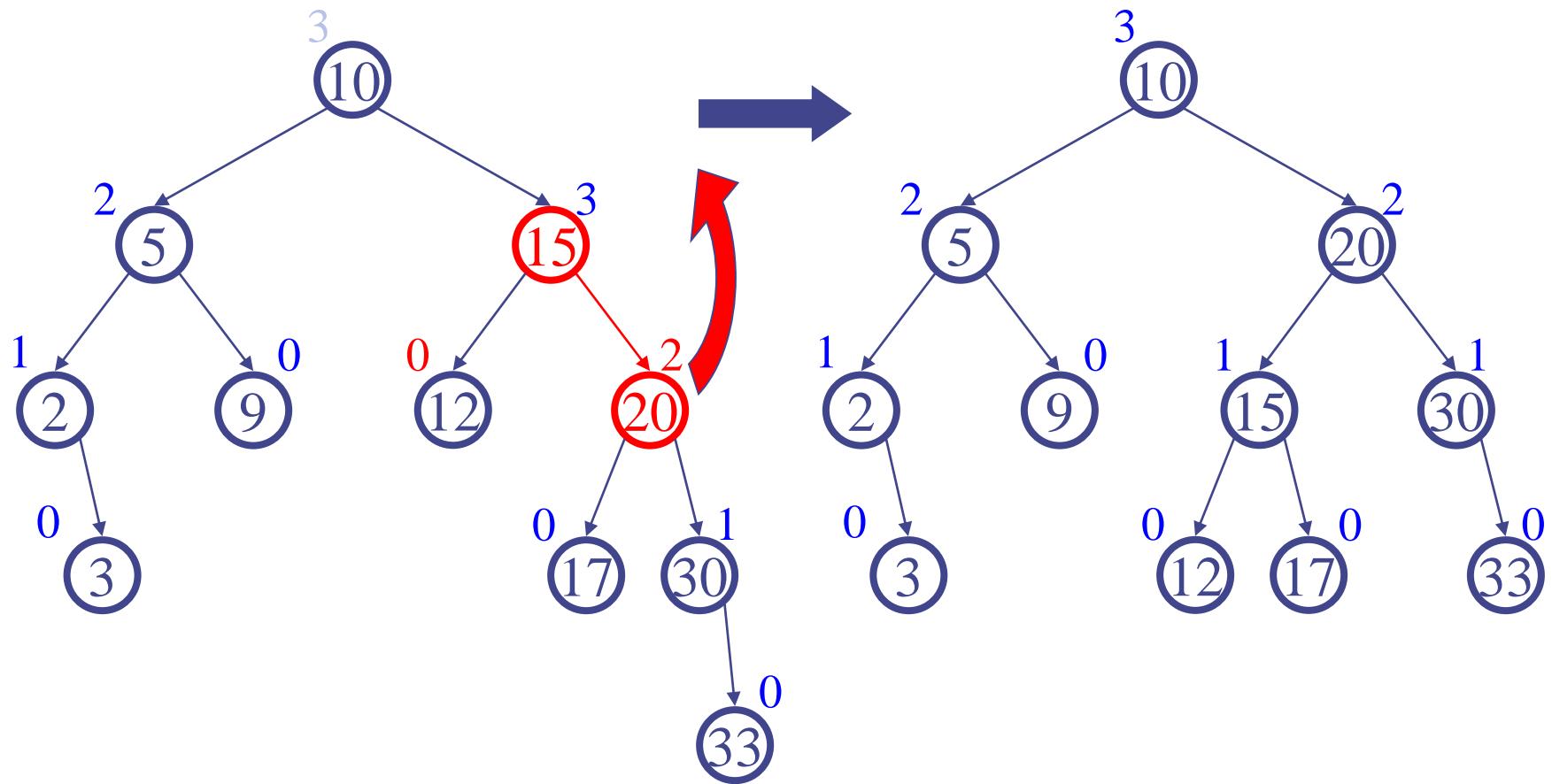
# Easy Insert



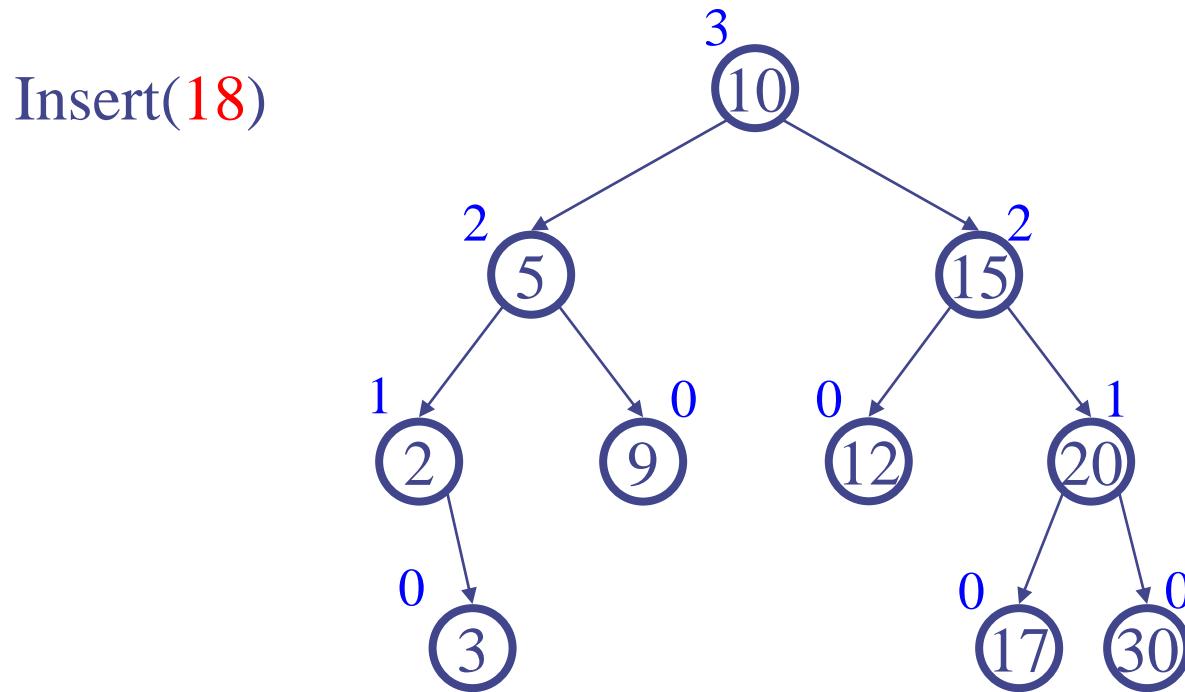
# Hard Insert (Bad Case #1)



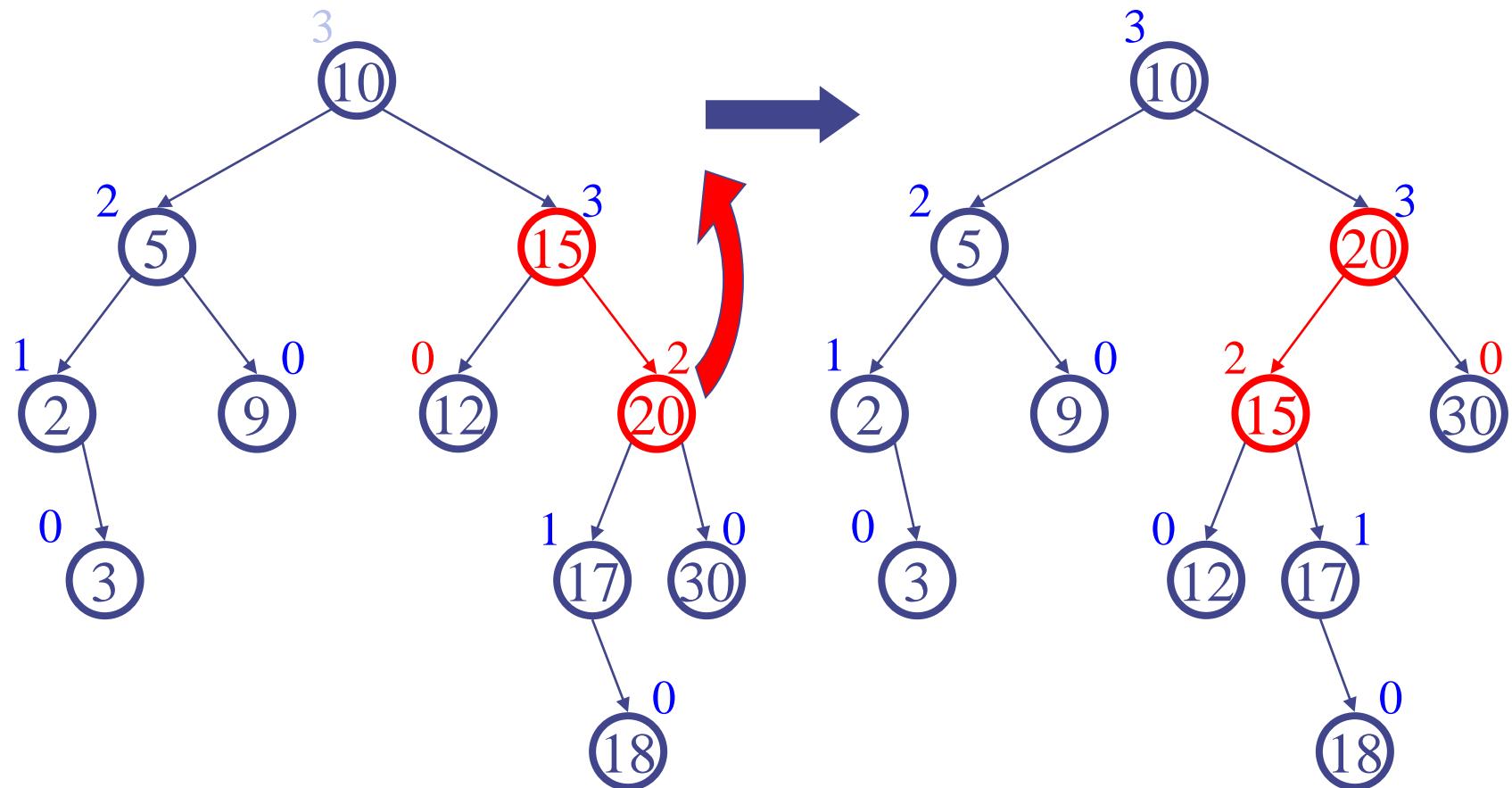
# Single Rotation



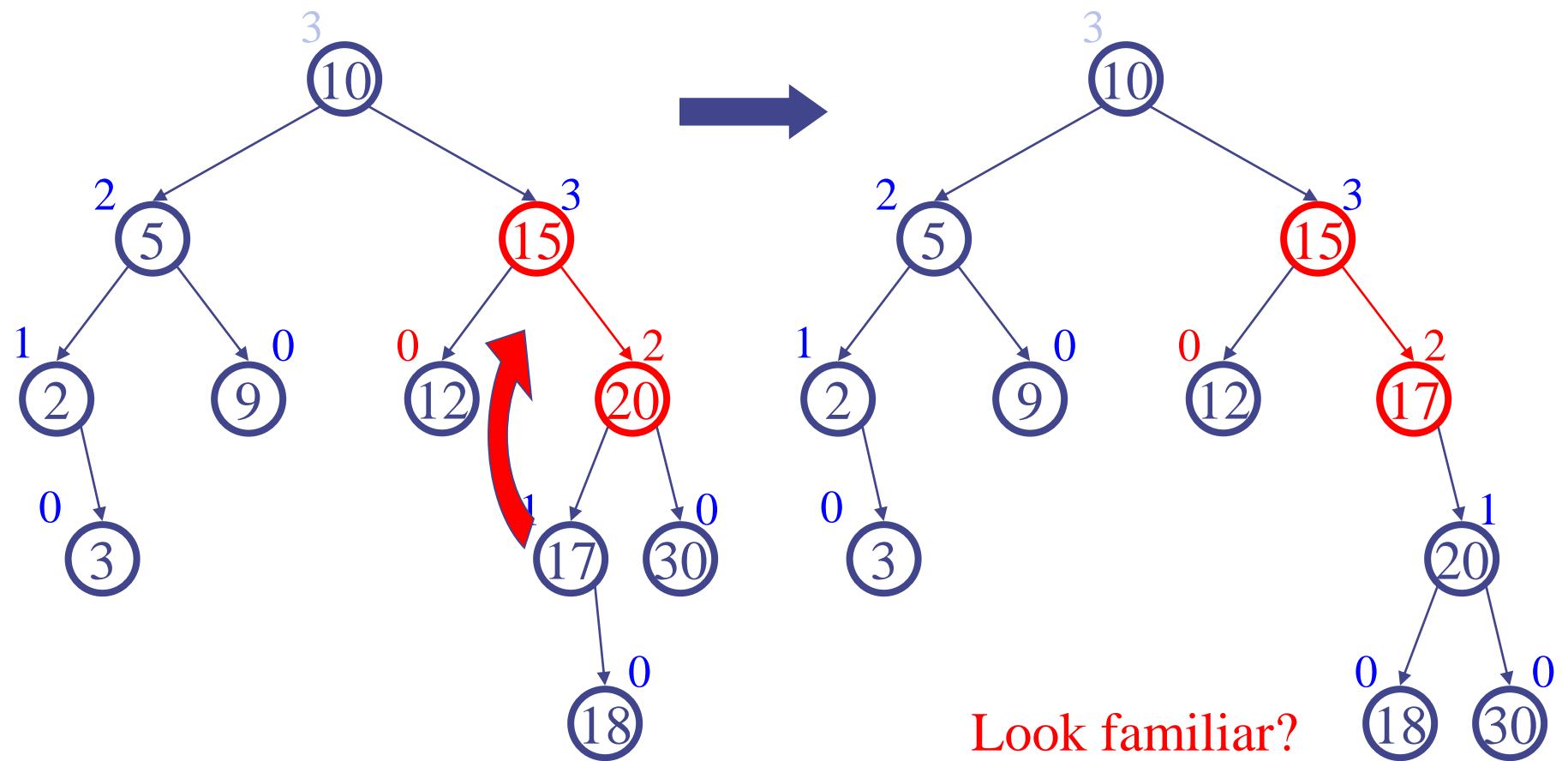
# Hard Insert (Bad Case #2)



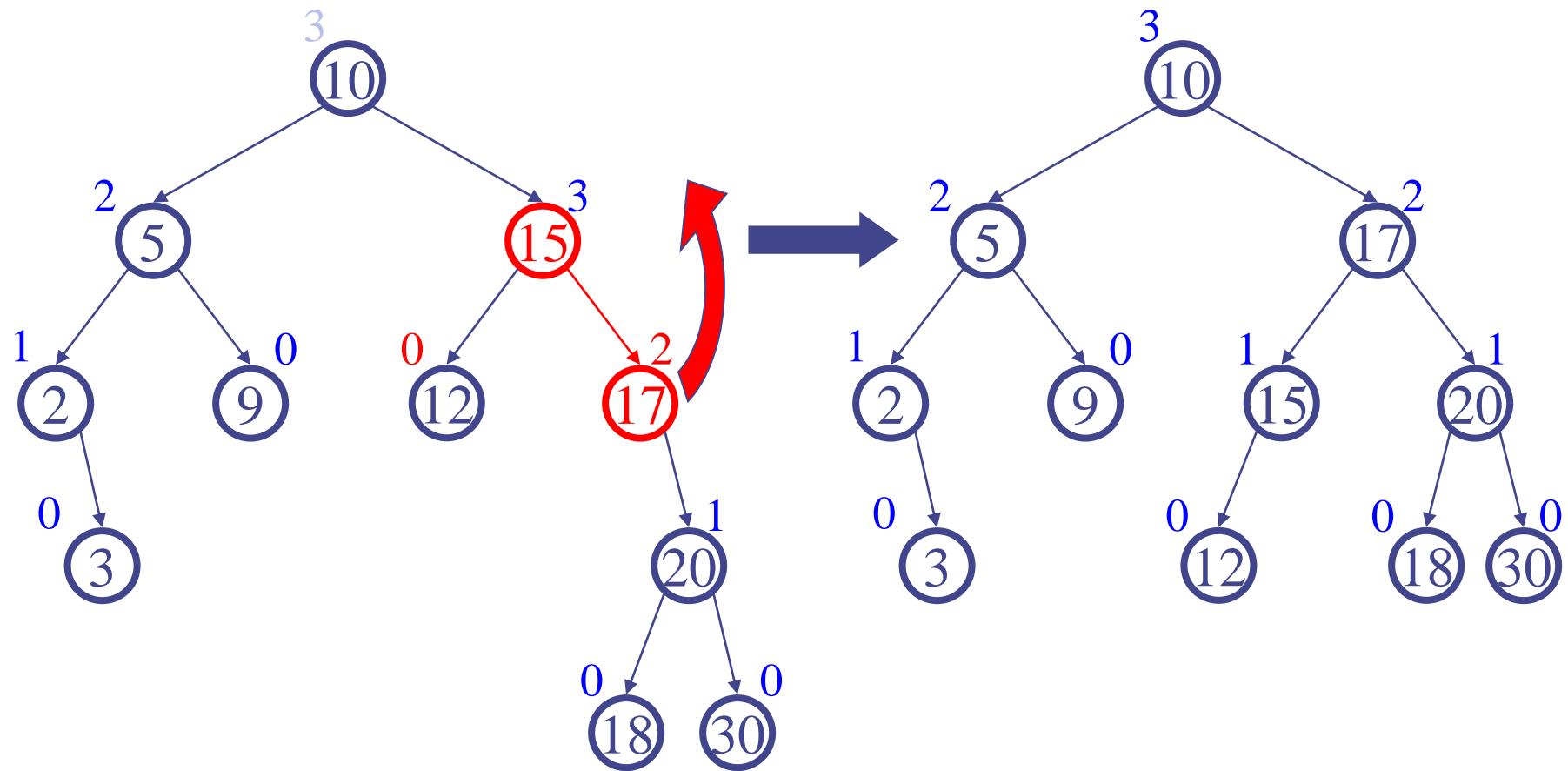
# Single Rotation (oops!)



# Double Rotation (Step #1)



# Double Rotation (Step #2)

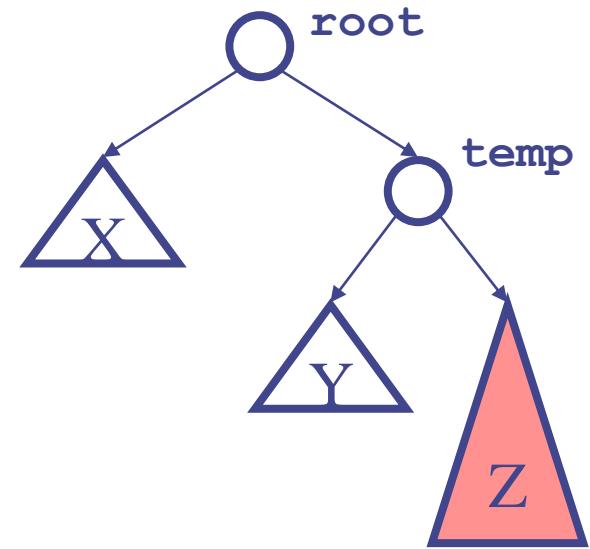


# AVL Insert Algorithm Revisited

- Recursive
  - 1. Search downward for spot
  - 2. Insert node
  - 3. Unwind stack, correcting heights
    - a. If imbalance #1, single rotate
    - b. If imbalance #2, double rotate
- Iterative
  - 1. Search downward for spot, **stacking parent nodes**
  - 2. Insert node
  - 3. Unwind stack, correcting heights
    - a. If imbalance #1, single rotate **and exit**
    - b. If imbalance #2, double rotate **and exit**

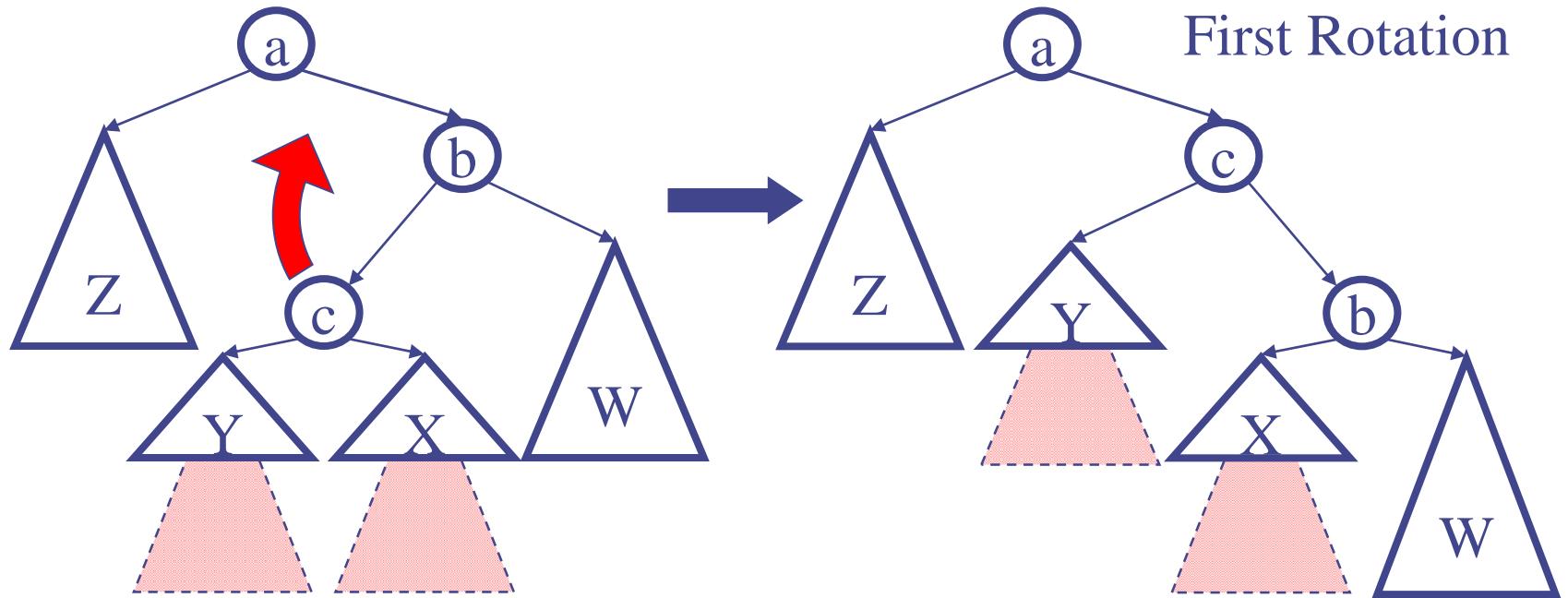
# Single Rotation Code

```
void RotateRight(Node *& root) {  
    Node * temp = root->right;  
    root->right = temp->left;  
    temp->left = root;  
    root->height =  
        max(root->right->height,  
             root->left->height) + 1;  
    temp->height =  
        max(temp->right->height,  
             temp->left->height) + 1;  
    root = temp;  
}
```



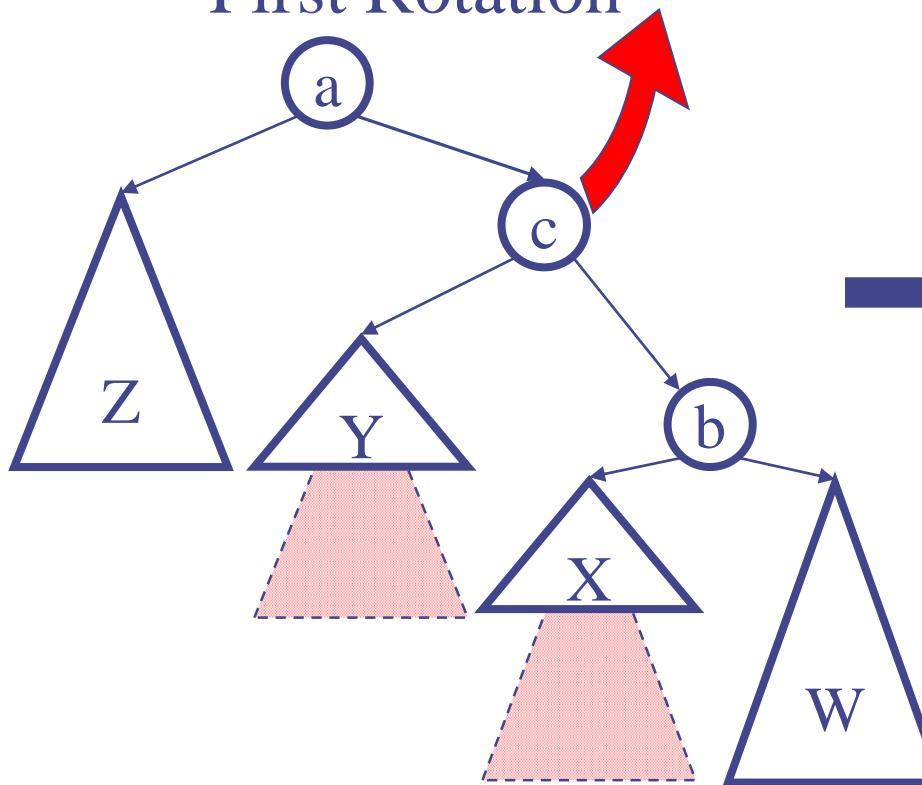
# Double Rotation Code

```
void DoubleRotateRight(Node *& root) {  
    RotateLeft(root->right);  
    RotateRight(root);  
}
```

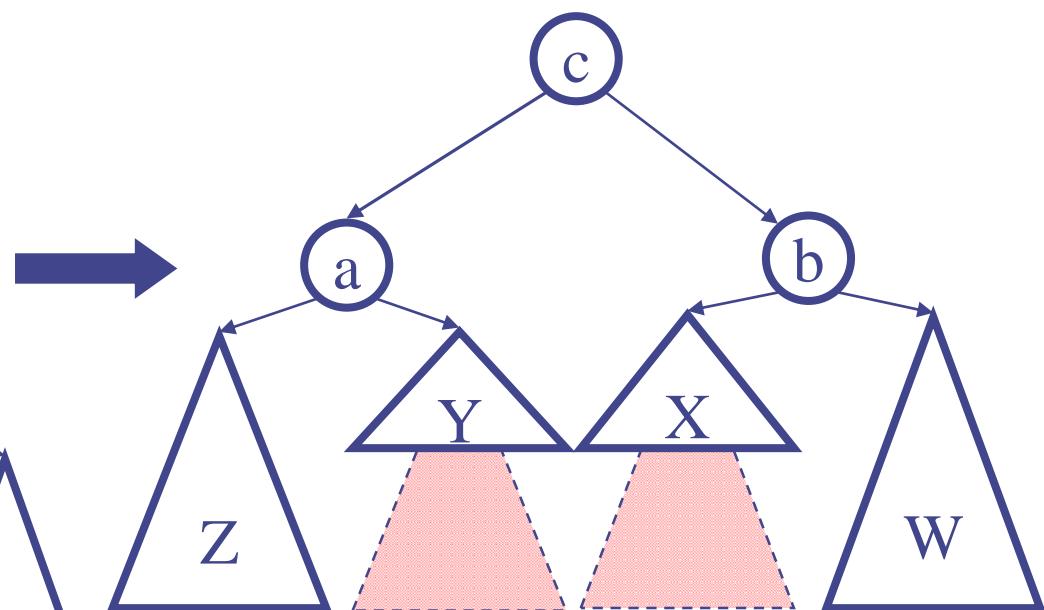


# Double Rotation Completed

First Rotation

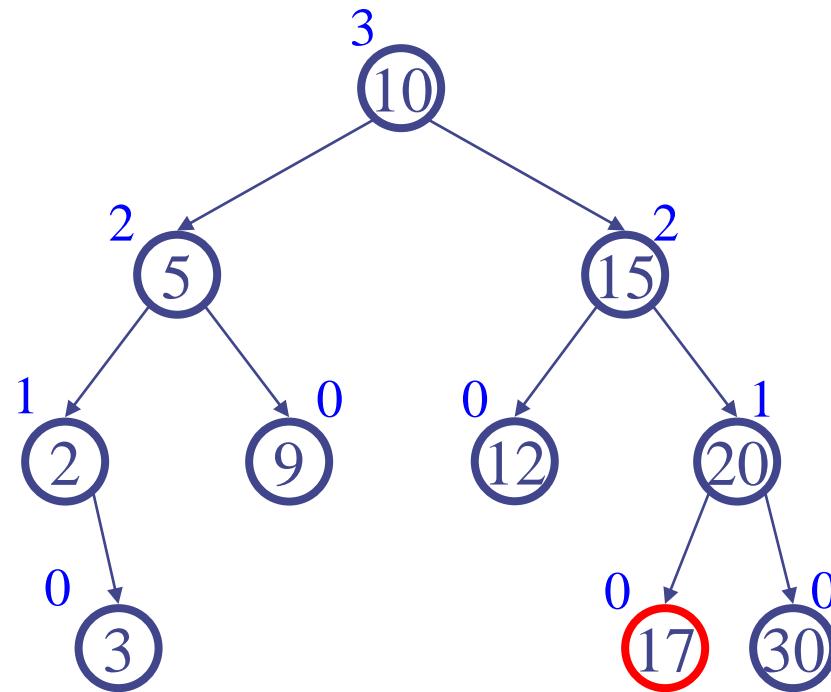


Second Rotation



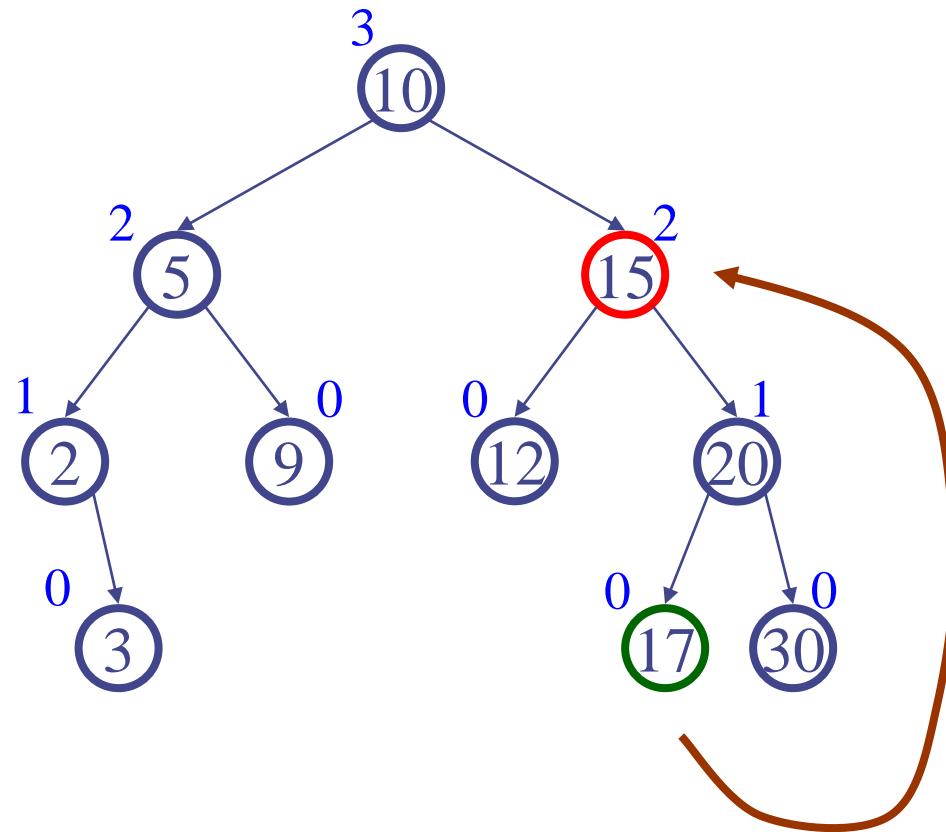
# Deletion: Really Easy Case

Delete(17)



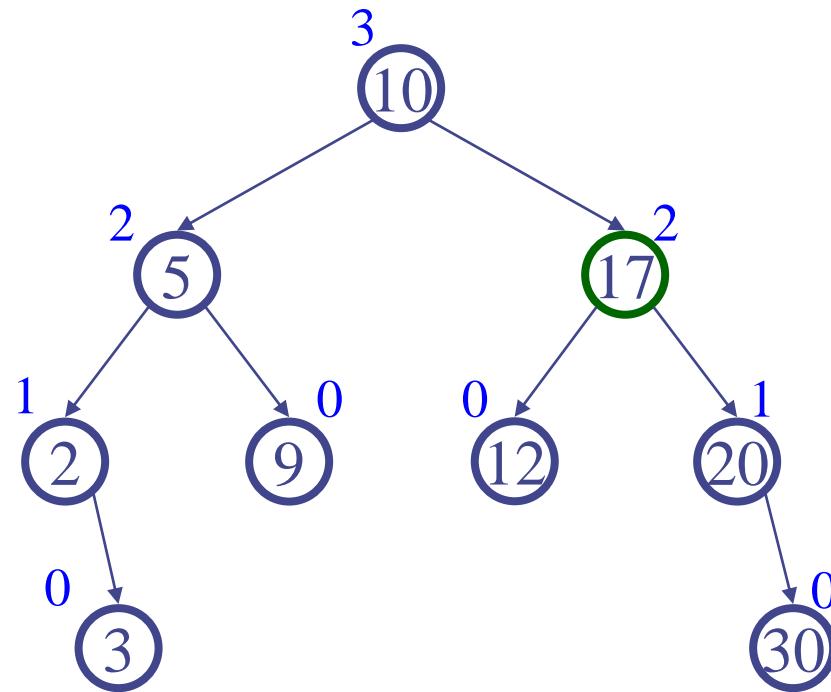
# Deletion: Pretty Easy Case

Delete(15)



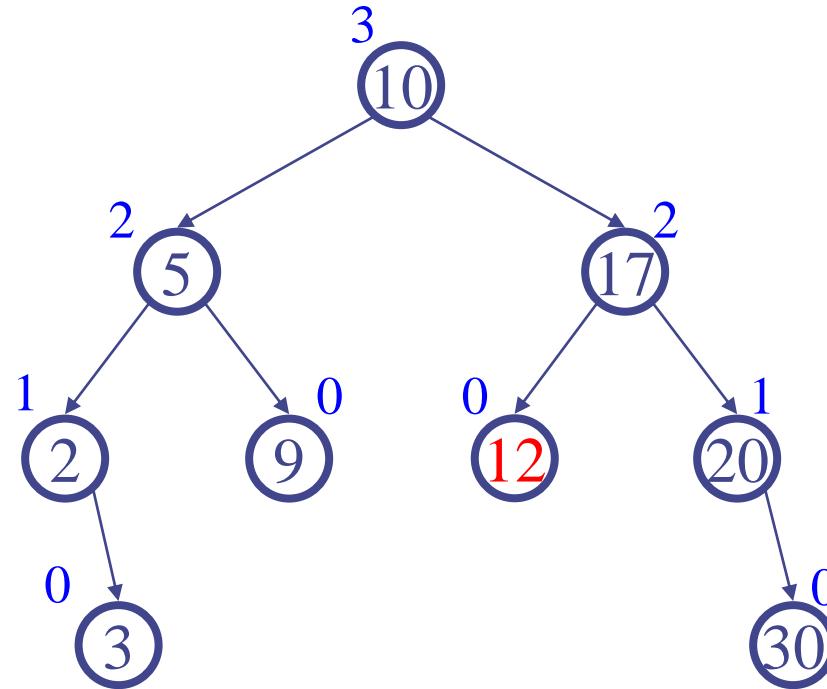
# Deletion: Pretty Easy Case (*cont.*)

Delete(15)

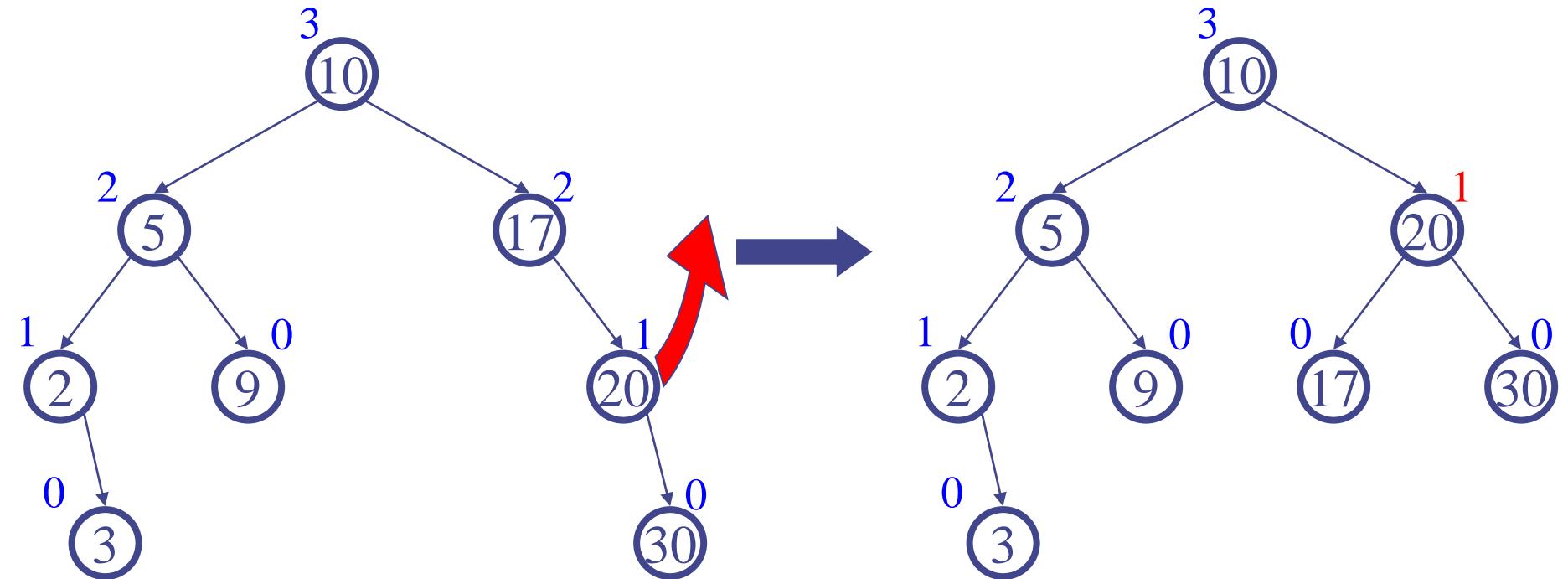


# Deletion (Hard Case #1)

Delete(12)



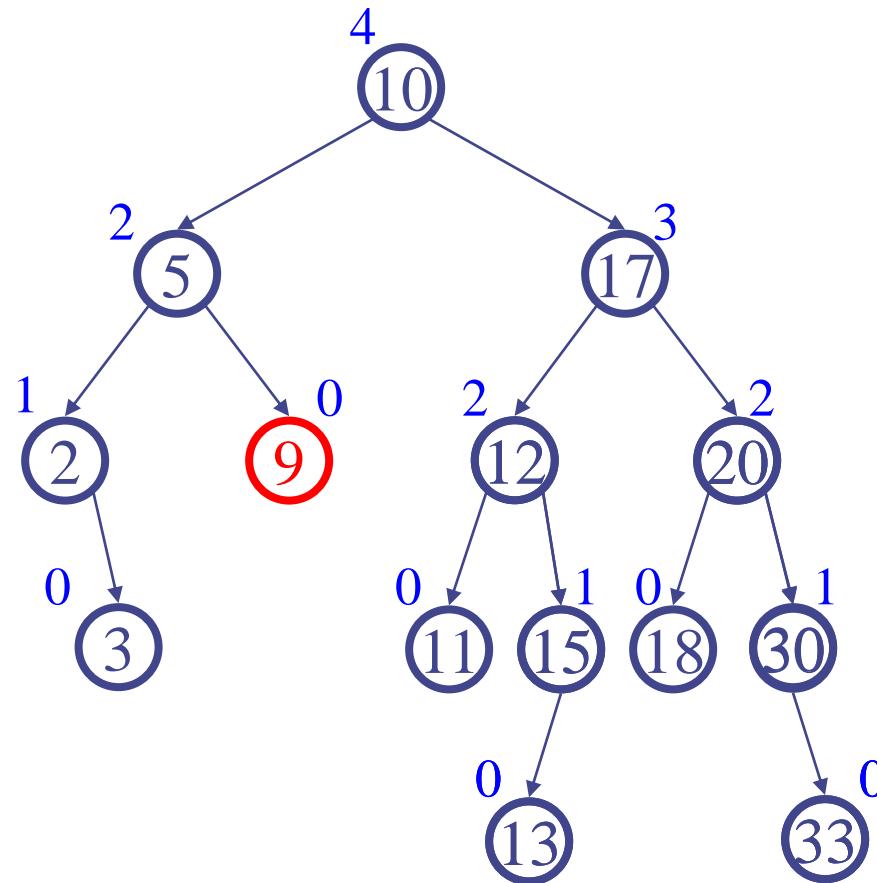
# Single Rotation on Deletion



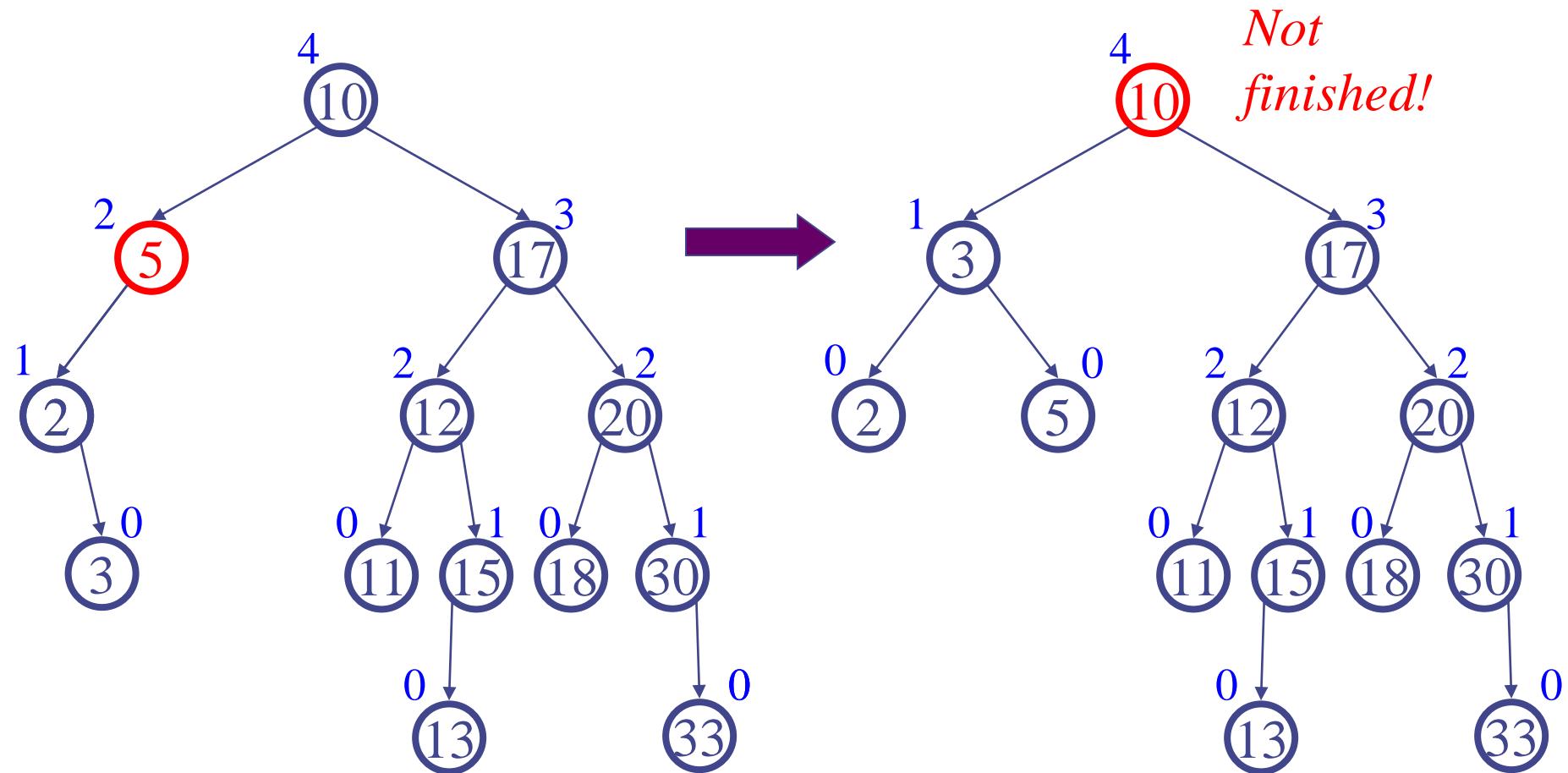
Deletion can differ from insertion – ***How?***

# Deletion (Hard Case)

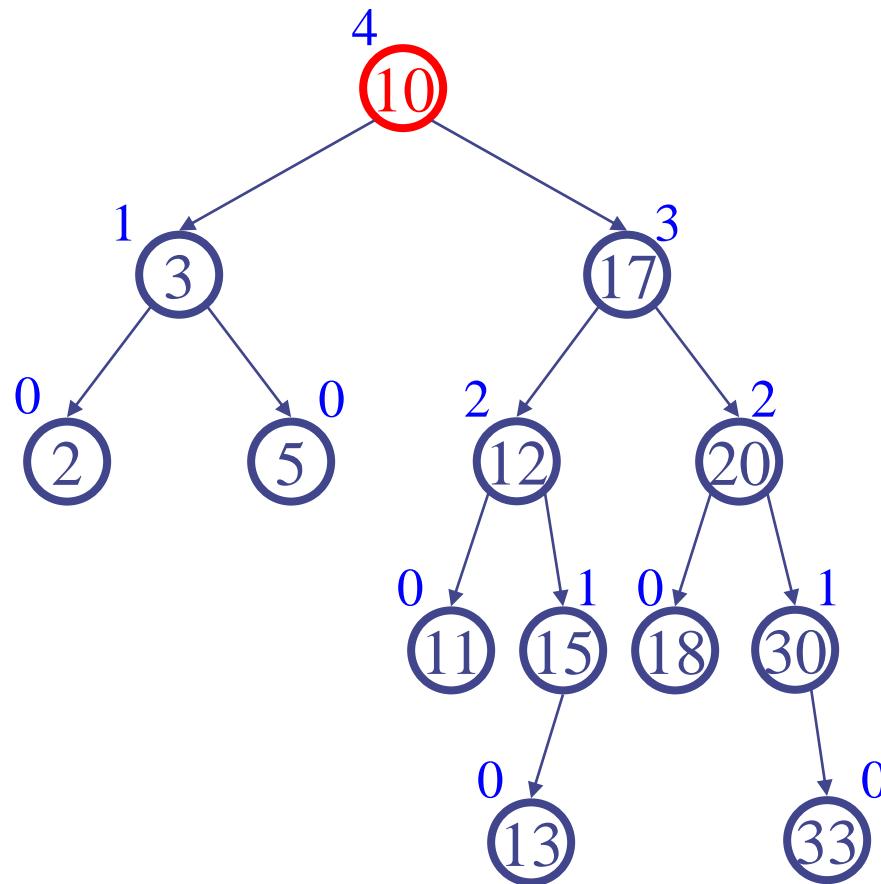
Delete(9)



# Double Rotation on Deletion



# Deletion with Propagation

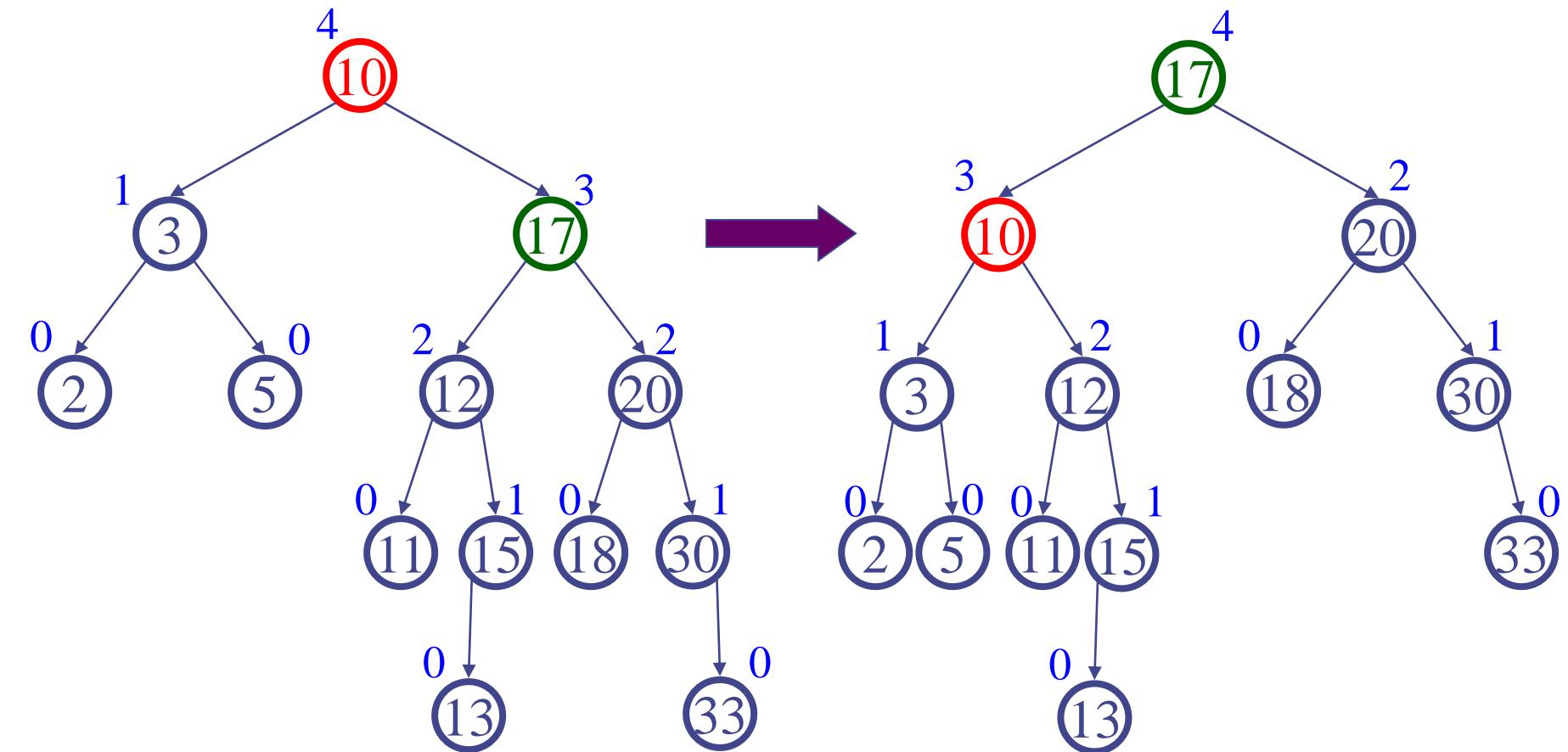


What different about this case?

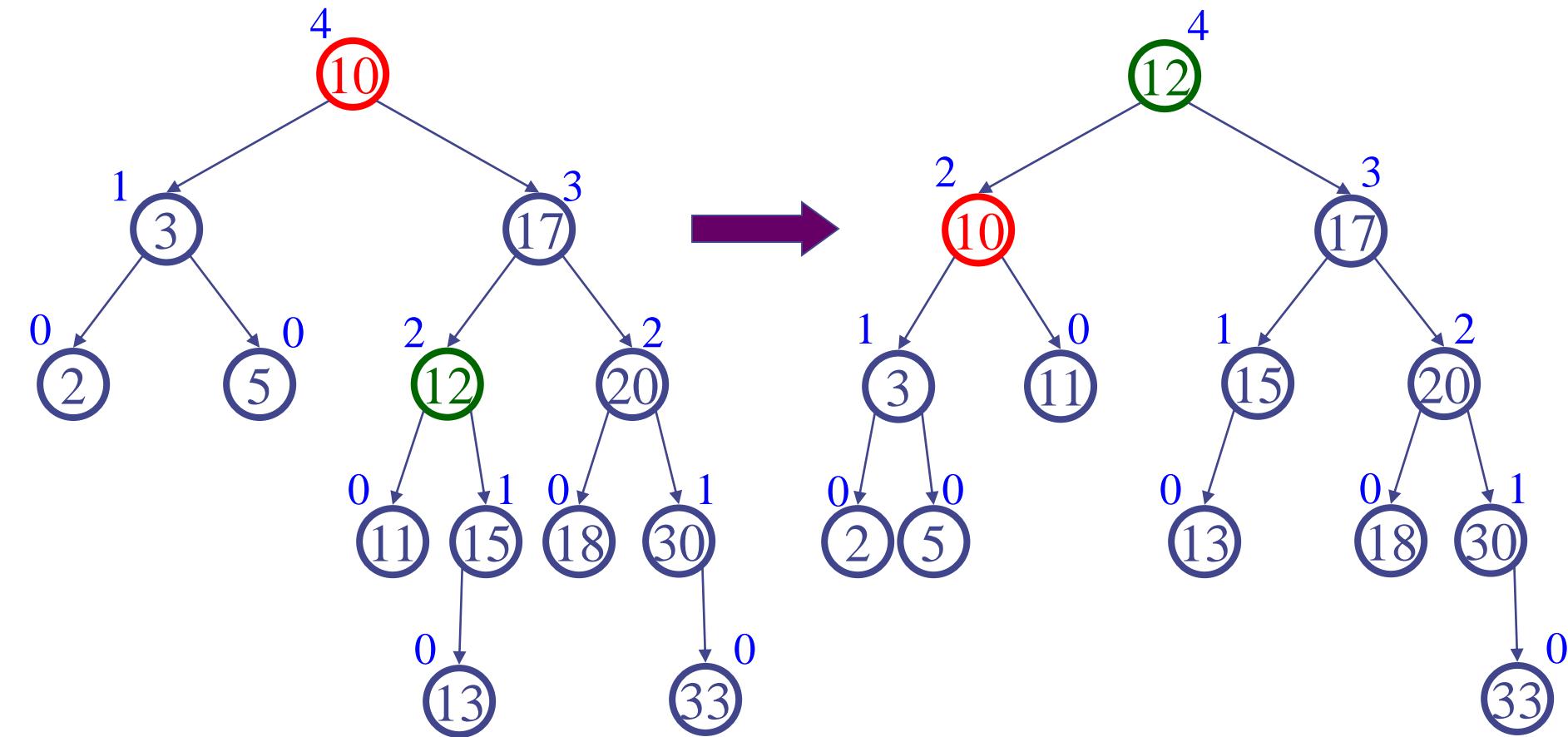
We get to choose whether to single or double rotate!



# Propagated Single Rotation



# Propagated Double Rotation



# AVL Deletion Algorithm

- Recursive
  - 1. Search downward for node
  - 2. Delete node
  - 3. Unwind, correcting heights as we go
    - a. If imbalance #1, single rotate
    - b. If imbalance #2 (or don't care), double rotate
- Iterative
  - 1. Search downward for node, **stacking parent nodes**
  - 2. Delete node
  - 3. Unwind stack, correcting heights
    - a. If imbalance #1, single rotate
    - b. If imbalance #2 (or don't care) double rotate

# Thinking About AVL

## Observations

- + Worst case height of an AVL tree is about  $1.44 \log n$
- + Insert, Find, Delete in worst case  $O(\log n)$
- + Only one (single or double) rotation needed on insertion
- + Compatible with lazy deletion
  - $O(\log n)$  rotations needed on deletion
  - Height fields must be maintained (or 2-bit balance)

## Coding complexity?

# Alternatives to AVL Trees

- Weight balanced trees
  - keep about the same number of nodes in each subtree
  - not nearly as nice
- Splay trees
  - “blind” adjusting version of AVL trees
    - ◆ no height information maintained!
  - insert/find always rotates node *to the root!*
  - worst case time is O(n)
  - amortized time for all operations is O(log n)
  - mysterious, but often faster than AVL trees in practice  
(better low-order terms)

# *Red-black Trees, Rotations, Insertions, Deletions*

# Balanced search trees

***Balanced search tree:*** A search-tree data structure for which a height of  $O(\lg n)$  is guaranteed when implementing a dynamic set of  $n$  items.

**Examples:**

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

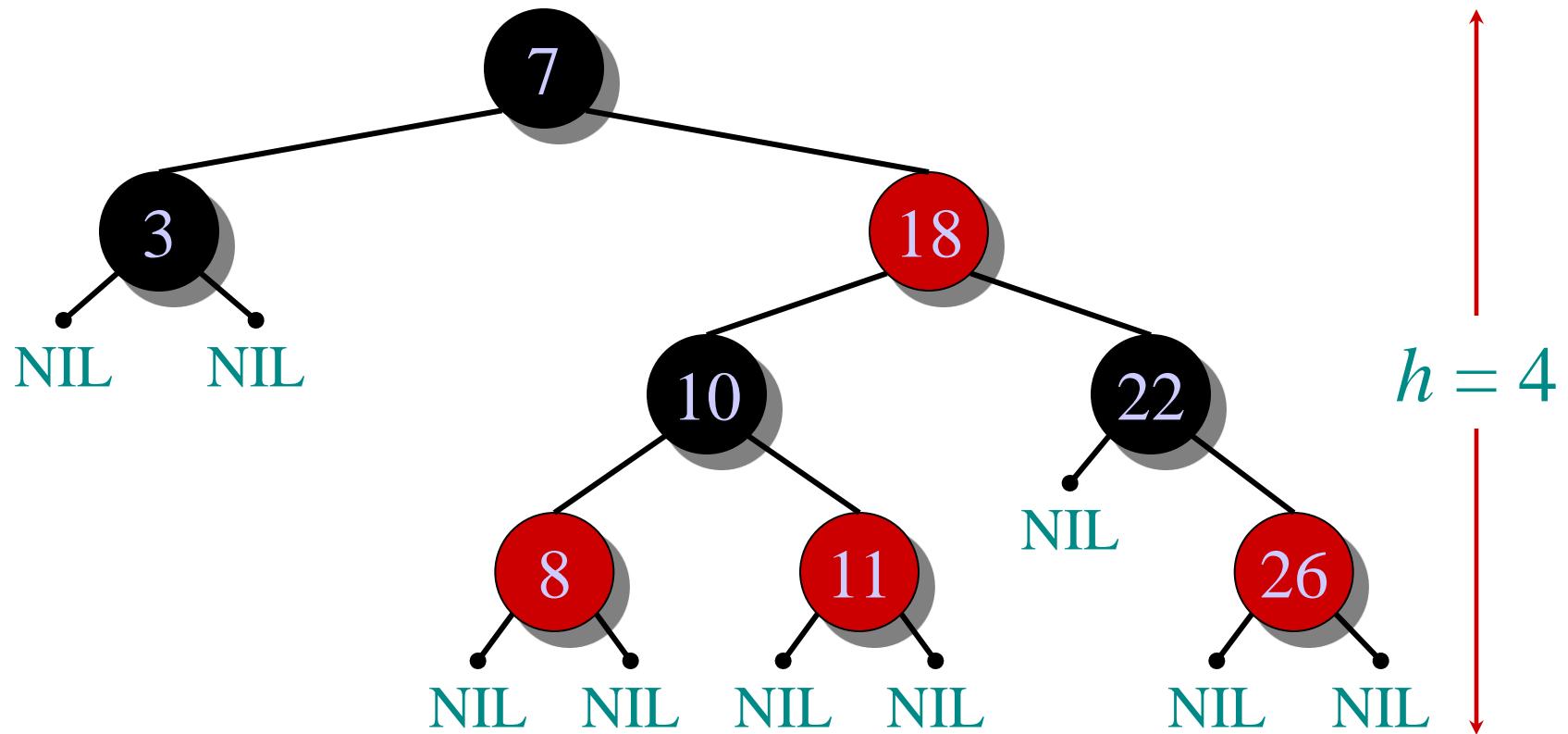
# Red-black trees

This data structure requires an extra one-bit **color** field in each node.

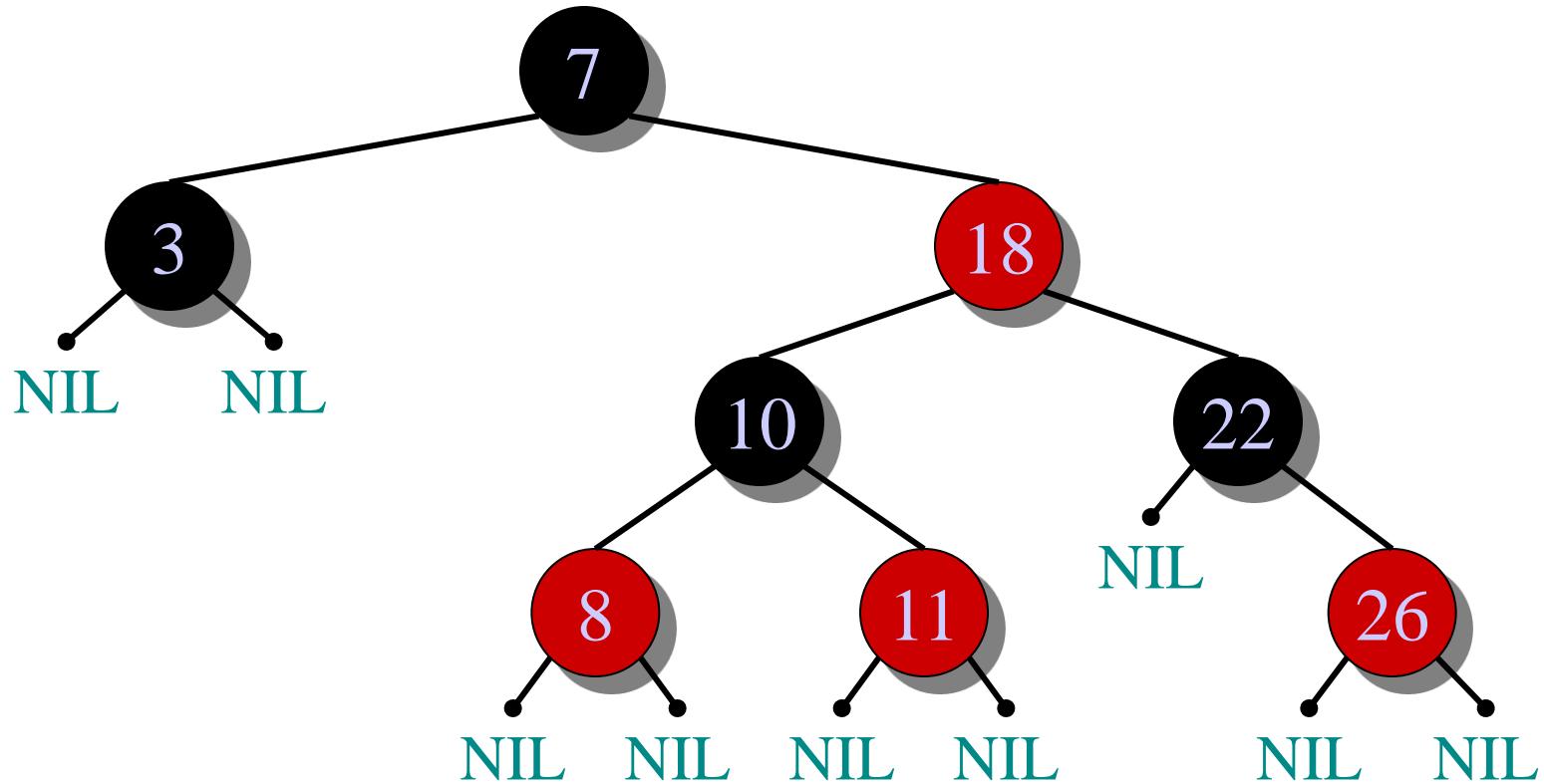
## *Red-black properties:*

1. Every node is either red or black.
2. The root and leaves (**NIL**'s) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes = **black-height( $x$ )**.

# Example of a red-black tree

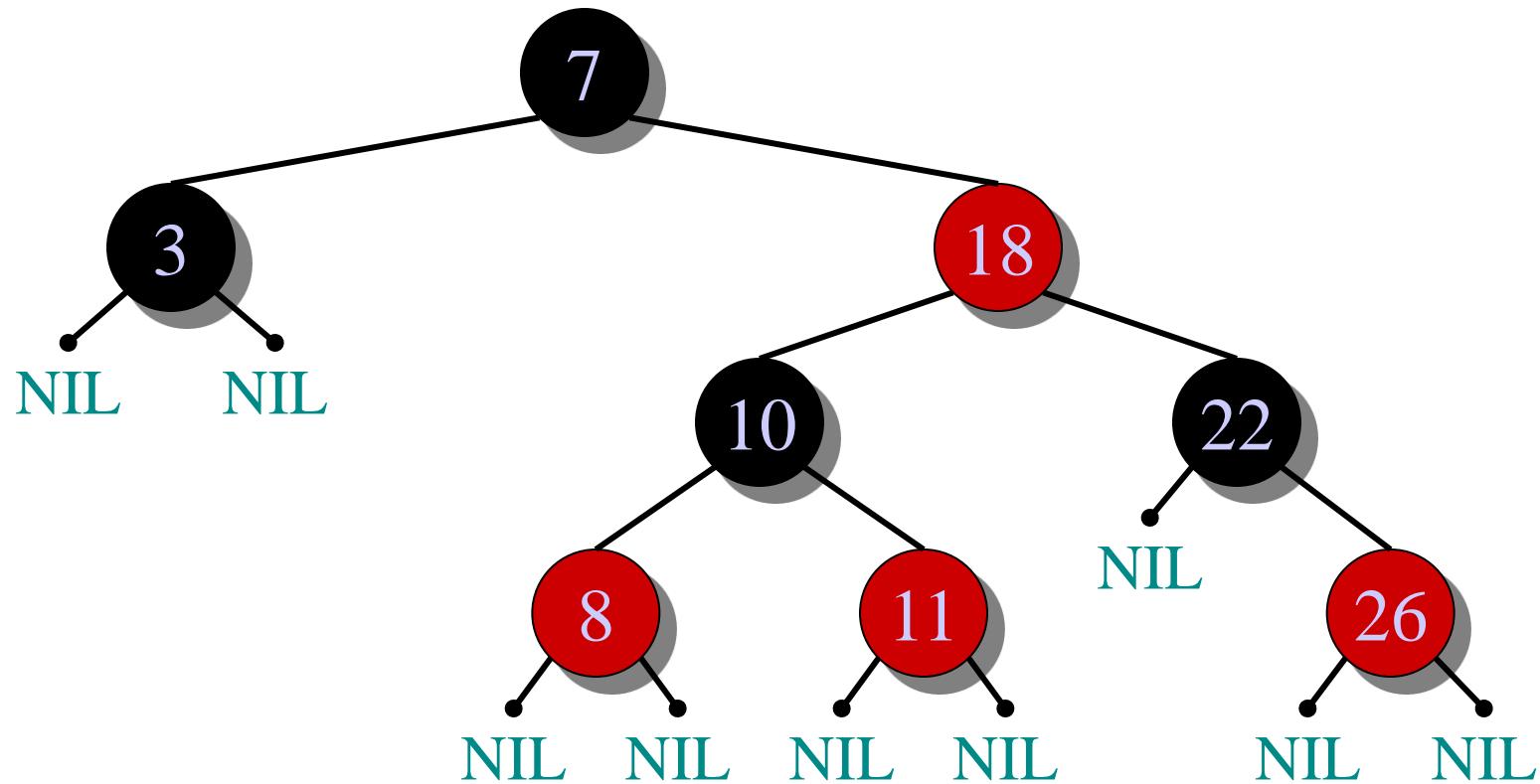


# Example of a red-black tree



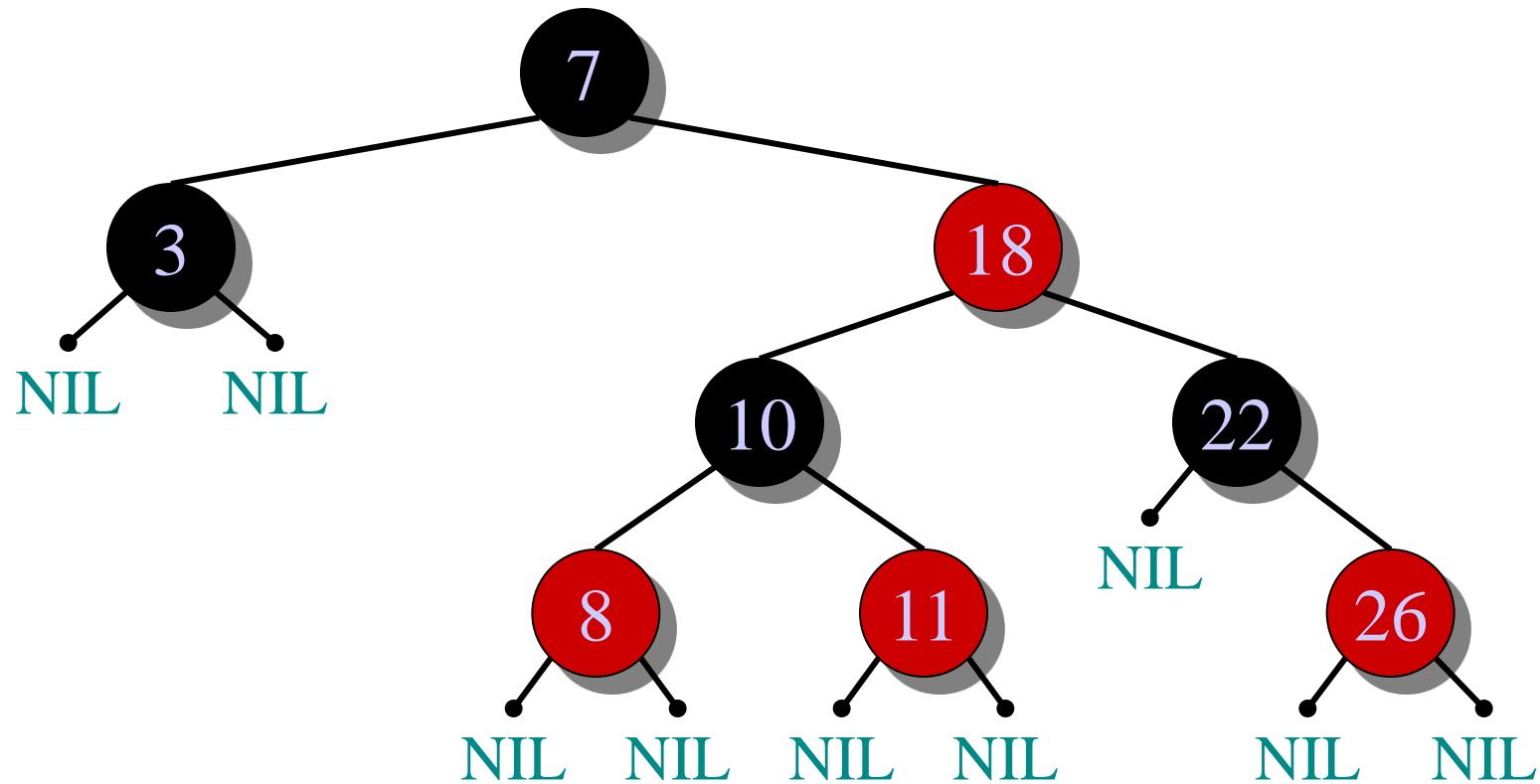
1. Every node is either red or black.

# Example of a red-black tree



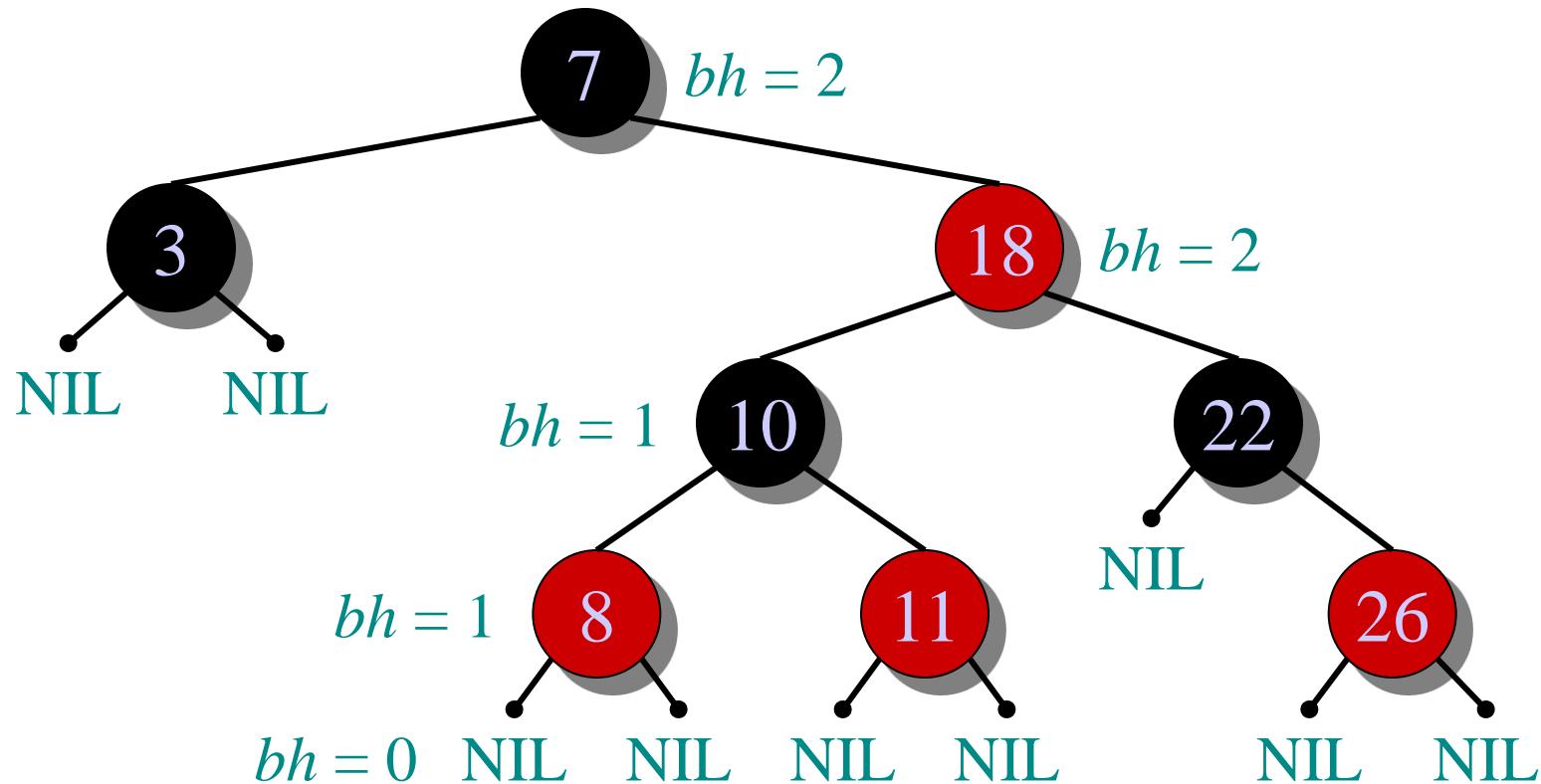
2. The root and leaves (NIL's) are black.

# Example of a red-black tree



3. If a node is red, then its parent is black.

# Example of a red-black tree



4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes =  $black-height(x)$ .

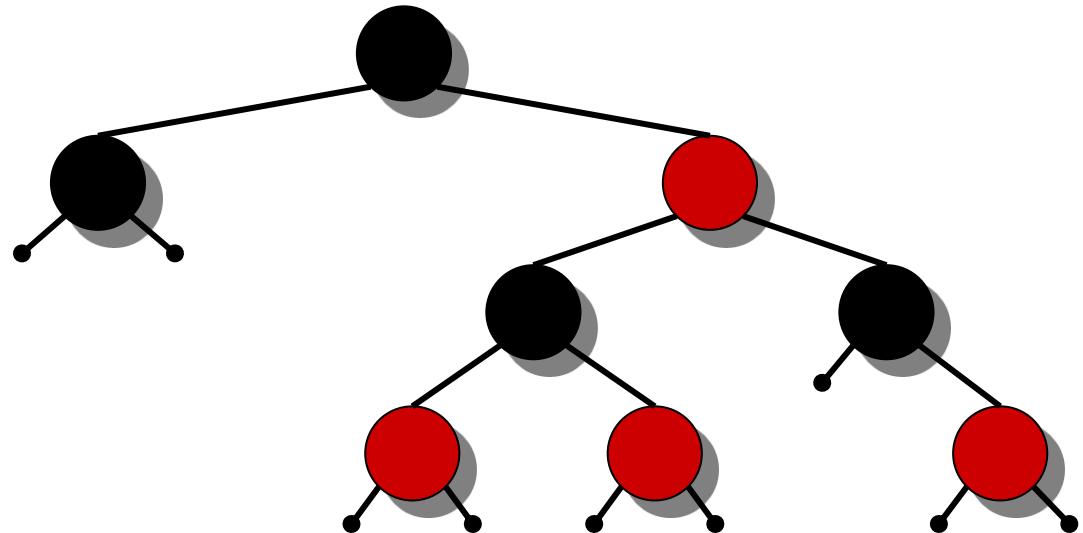
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.



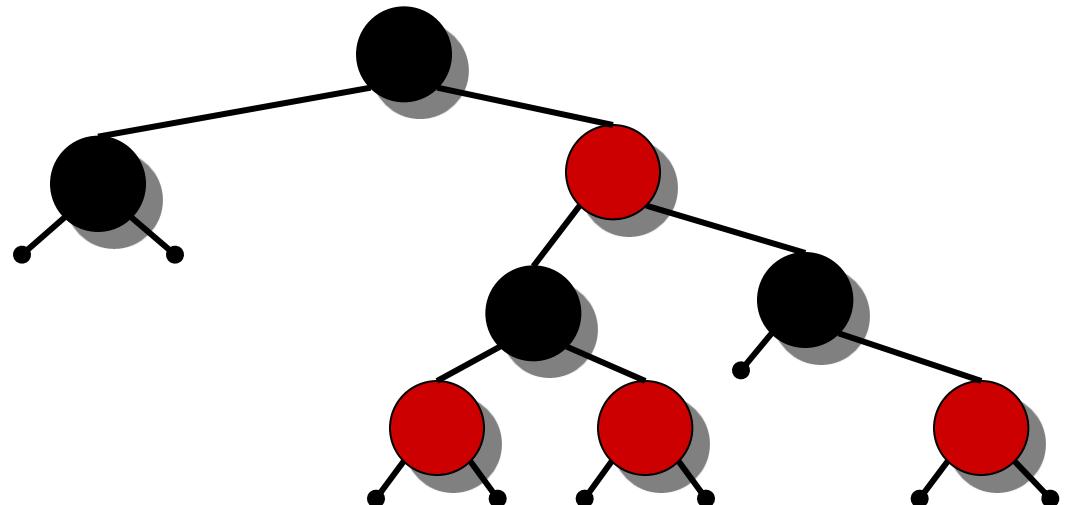
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.



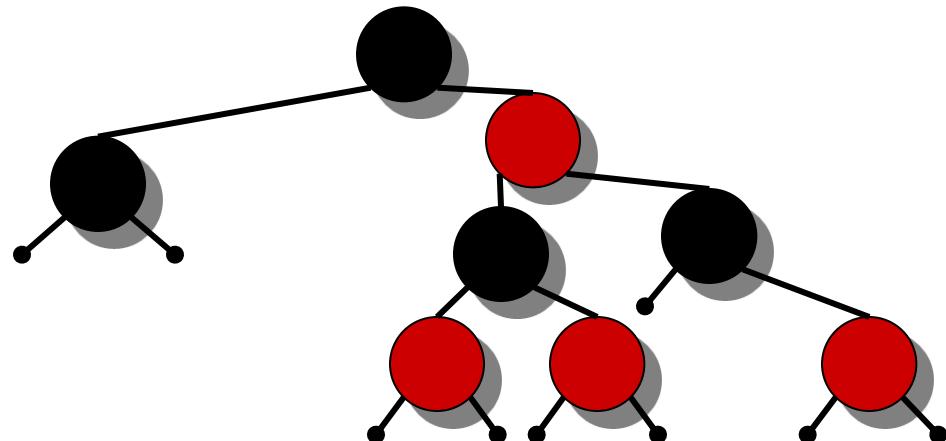
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.



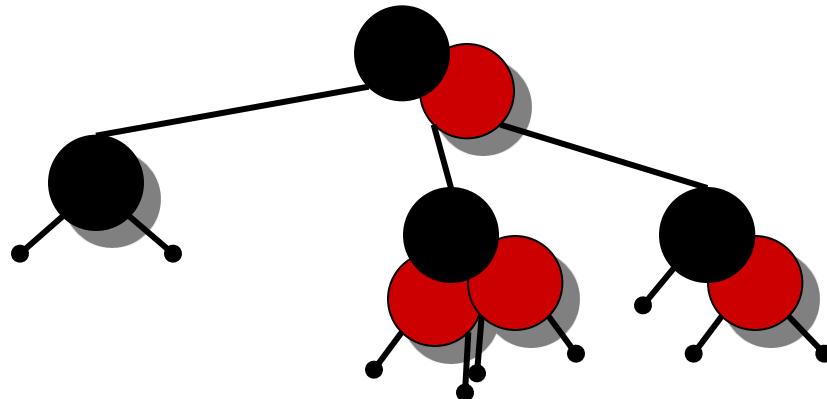
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.



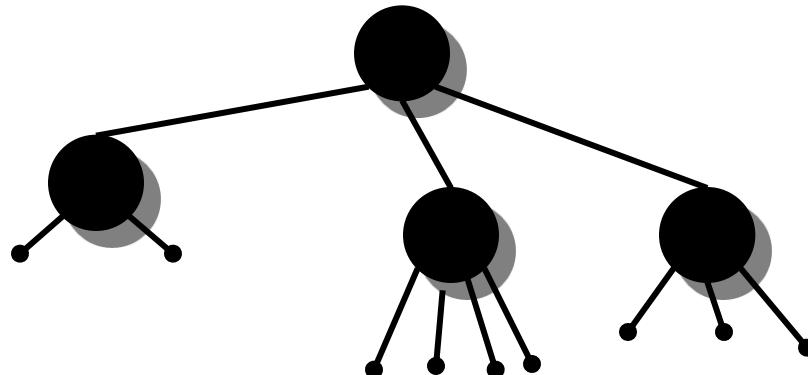
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.



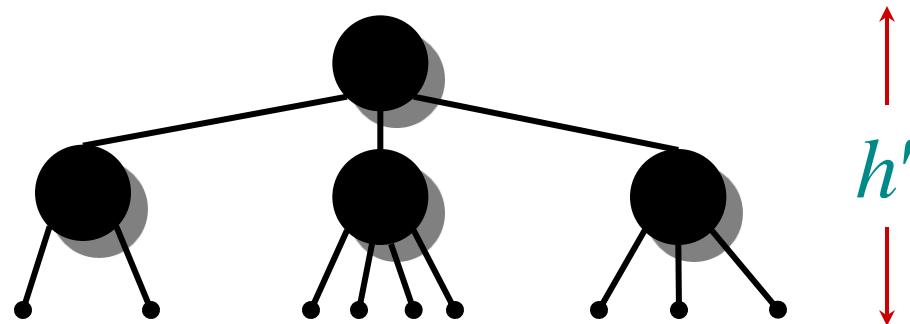
# Height of a red-black tree

**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

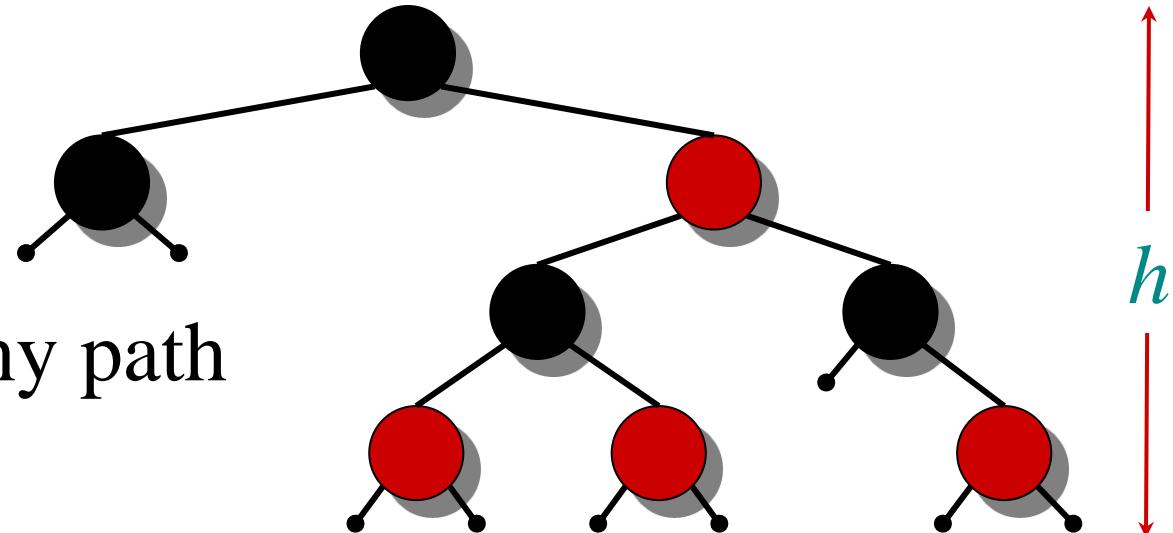
## INTUITION:

- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth  $h'$  of leaves.



# Proof (continued)

- We have  $h' \geq h/2$ , since at most half the leaves on any path are red.

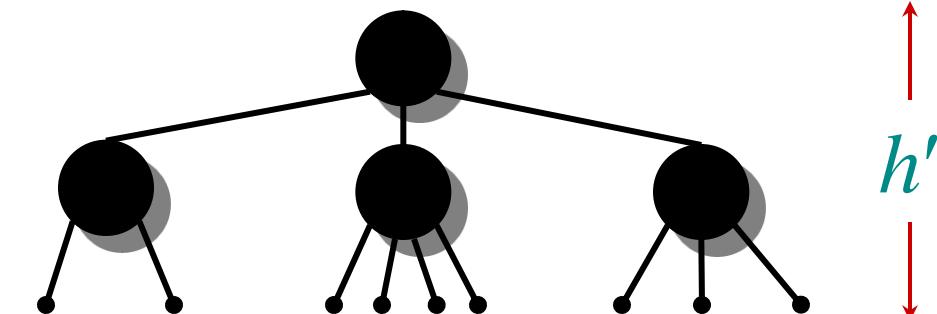


- The number of leaves in each tree is  $n + 1$

$$\Rightarrow n + 1 \geq 2^{h'}$$

$$\Rightarrow \lg(n + 1) \geq h' \geq h/2$$

$$\Rightarrow h \leq 2 \lg(n + 1).$$



# Query operations

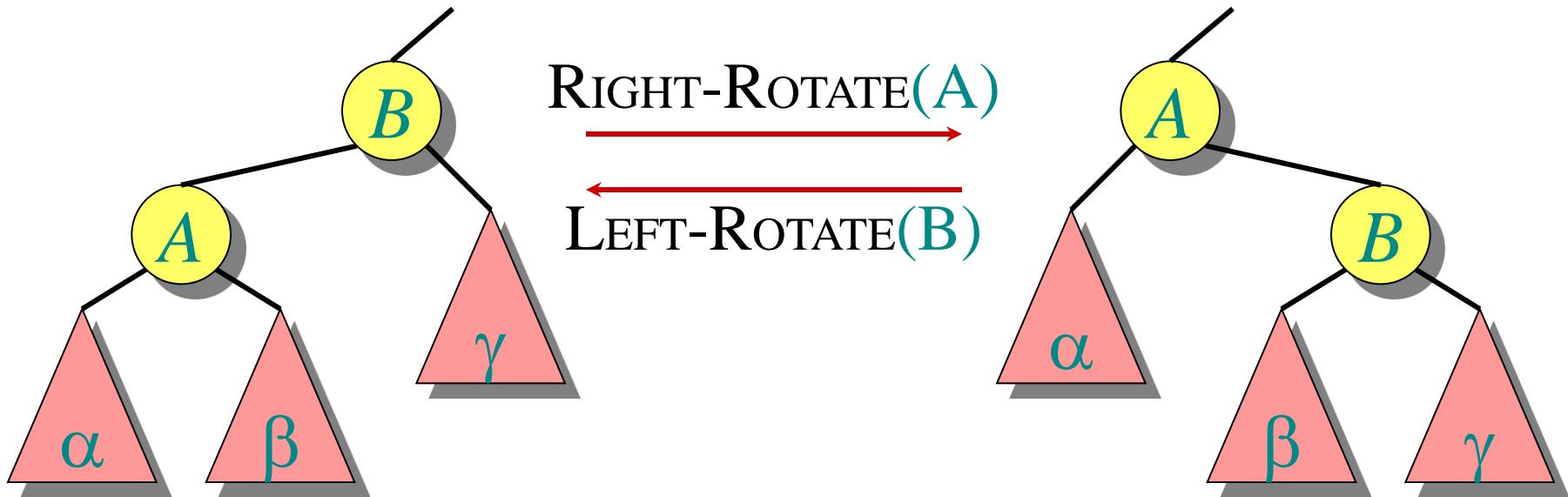
**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in  $O(\lg n)$  time on a red-black tree with  $n$  nodes.

# Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree:  
***“rotations”.***

# Rotations



Rotations maintain the inorder ordering of keys:

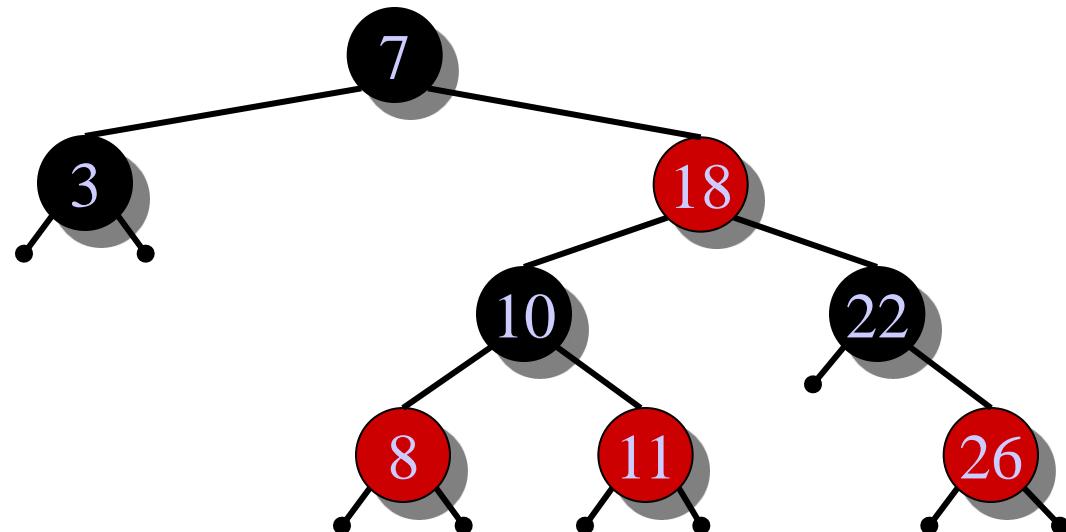
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$

A rotation can be performed in  $O(1)$  time.

# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

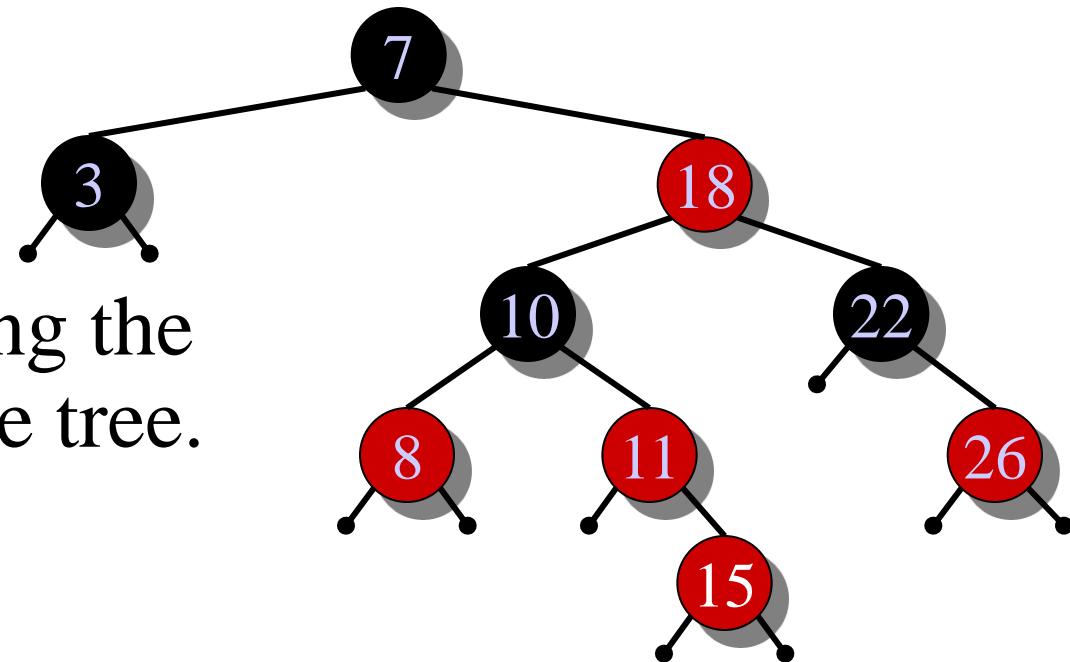


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.

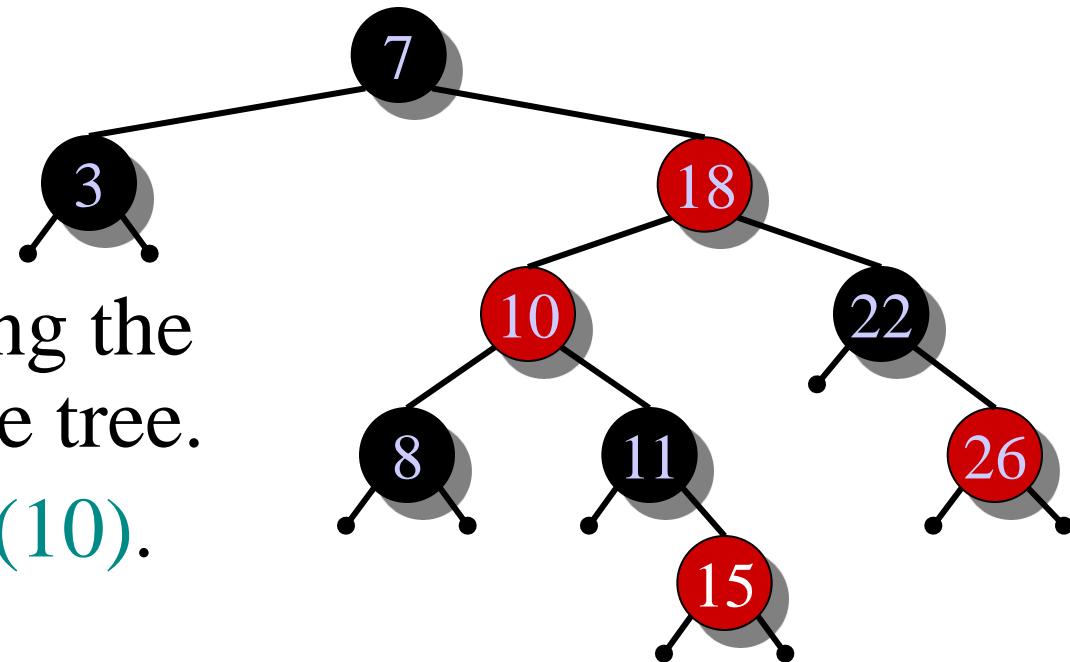


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(10).

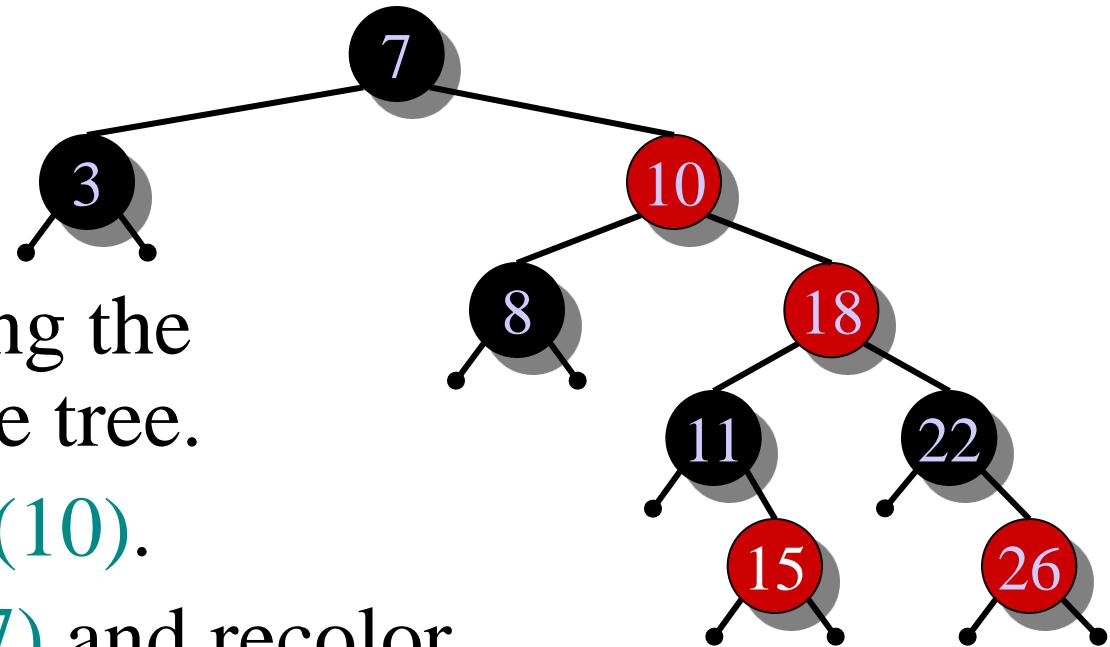


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(10).
- LEFT-ROTATE(7) and recolor.

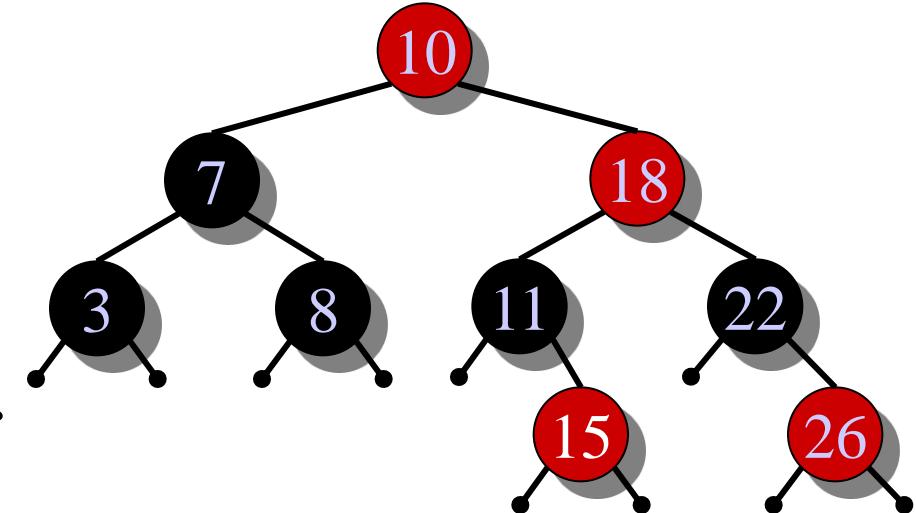


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(10).
- LEFT-ROTATE(7) and recolor.

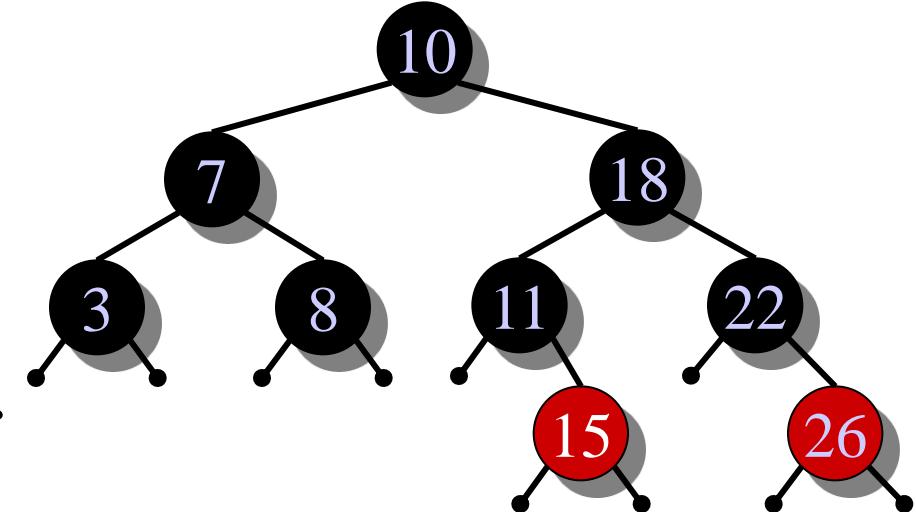


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(10).
- LEFT-ROTATE(7) and recolor.



# Pseudocode

RB-INSERT( $T, x$ )

  TREE-INSERT( $T, x$ )

$\text{color}[x] \leftarrow \text{RED}$    ▷ only RB property 3 can be violated

**while**  $x \neq \text{root}[T]$  and  $\text{color}[p[x]] = \text{RED}$

**do if**  $p[x] = \text{left}[p[p[x]]]$

**then**  $y \leftarrow \text{right}[p[p[x]]]$                           ▷  $y = \text{aunt/uncle of } x$

**if**  $\text{color}[y] = \text{RED}$

**then** ⟨Case 1⟩

**else if**  $x = \text{right}[p[x]]$

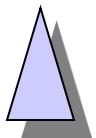
**then** ⟨Case 2⟩   ▷ Case 2 falls into Case 3

          ⟨Case 3⟩

**else** ⟨“then” clause with “left” and “right” swapped⟩

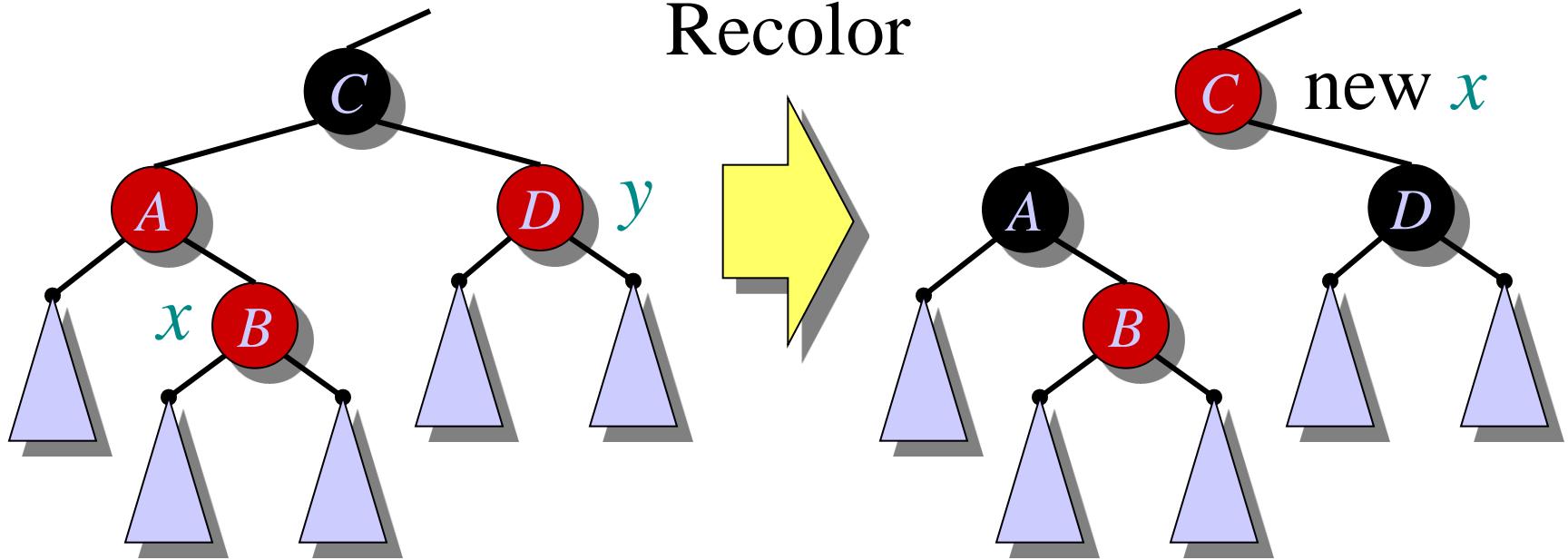
$\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$

# Graphical notation

Let  denote a subtree with a black root.

All 's have the same black-height.

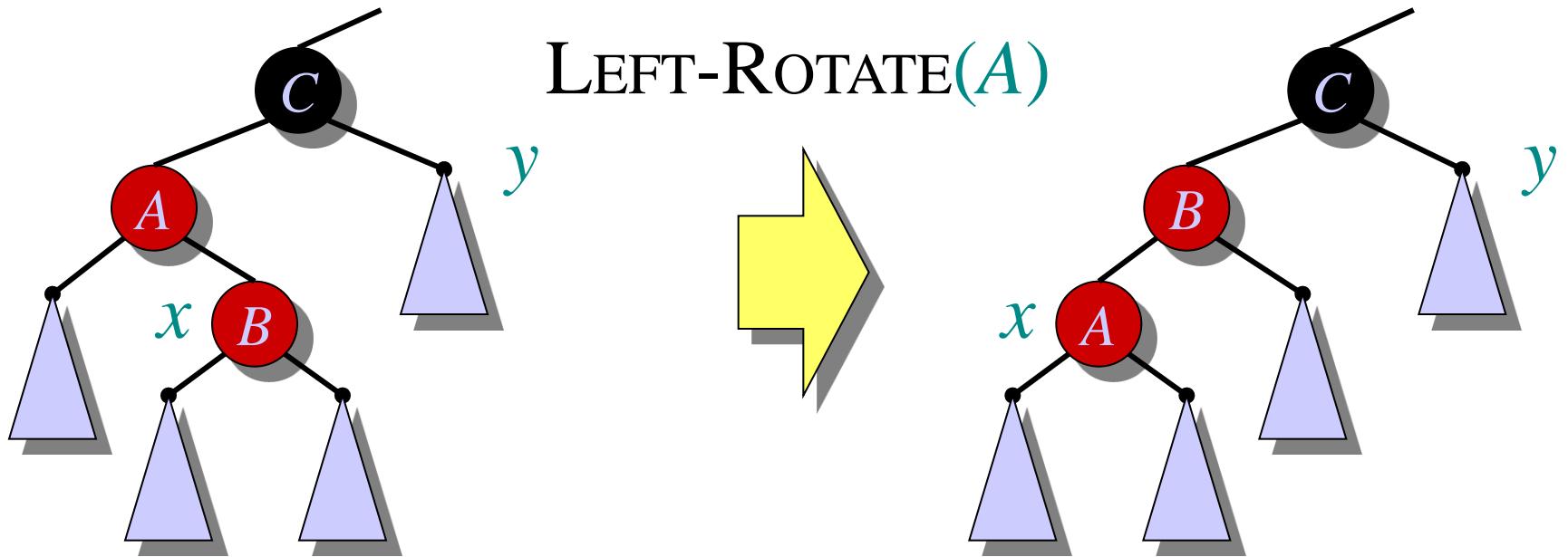
# Case 1



(Or, children of  $A$  are swapped.)

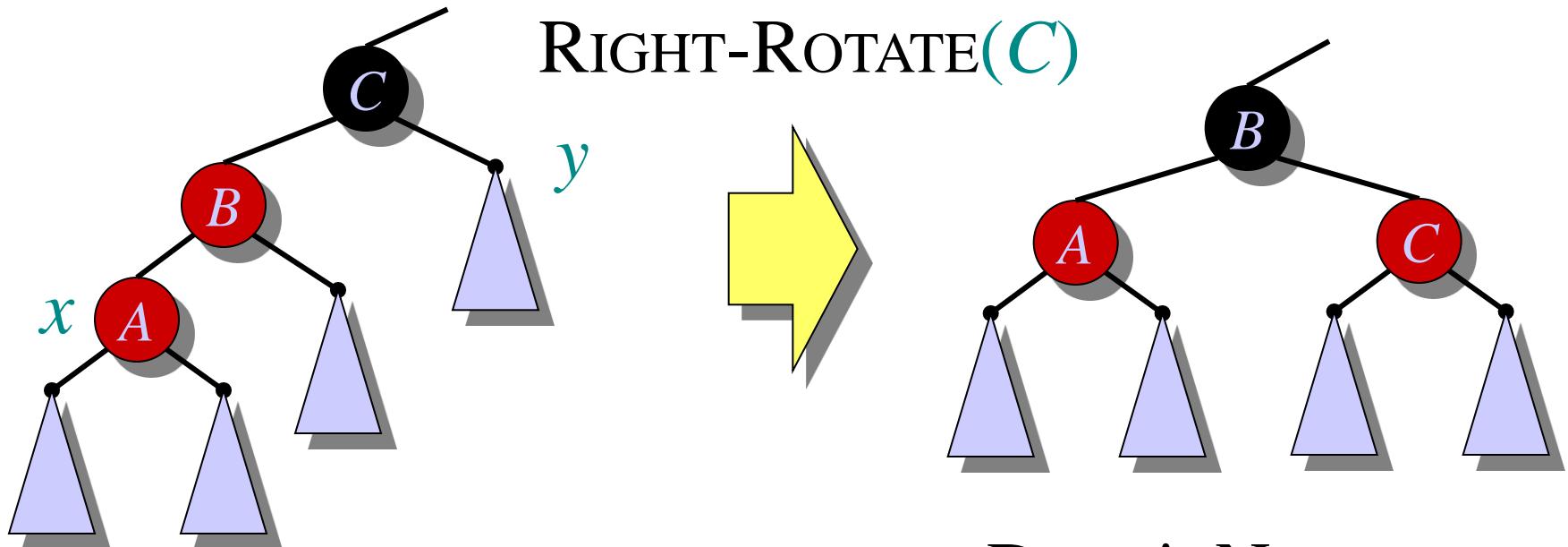
Push  $C$ 's black onto  $A$  and  $D$ , and recurse, since  $C$ 's parent may be red.

# Case 2



Transform to Case 3.

# Case 3



Done! No more violations of RB property 3 are possible.

# Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:**  $O(\lg n)$  with  $O(1)$  rotations.

RB-DELETE — same asymptotic running time (see textbook).

# Red–black tree

---

Red–black tree		
Type	Tree	
Invented	1972	
Invented by	Rudolf Bayer	
Time complexity in big O notation		
	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

A **red–black tree** is a type of self-balancing binary search tree, a data structure used in computer science, typically to implement associative arrays. The original structure was invented in 1972 by Rudolf Bayer<sup>[1]</sup> and named "symmetric binary B-tree," but acquired its modern name in a paper in 1978 by Leonidas J. Guibas and Robert Sedgewick.<sup>[2]</sup>

Since it is a balanced tree, it guarantees insertion, search and delete to be  $O(\log n)$  in time, where  $n$  is the total number of elements in the tree.<sup>[3]</sup>

A red–black tree is a binary search tree that inserts and deletes in such a way that the tree is always reasonably balanced.

## Terminology

A red–black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as text fragments or numbers.

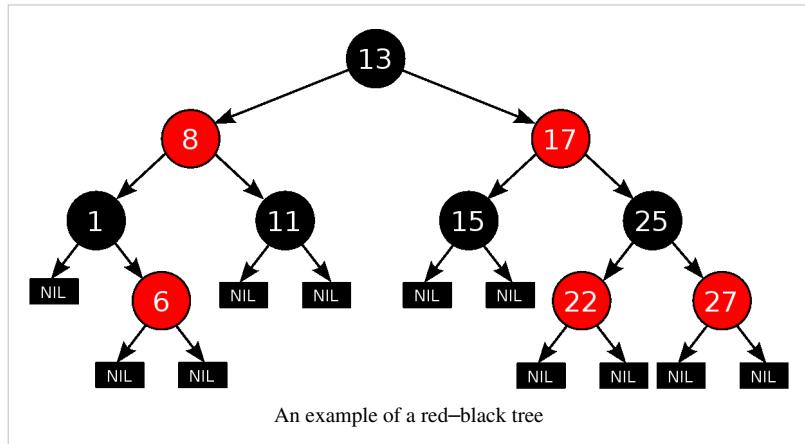
The leaf nodes of red–black trees do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.

Red–black trees, like all binary search trees, allow efficient in-order traversal (that is: in the order Left–Root–Right) of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree, having the least possible tree height, results in  $O(\log n)$  search time.

## Properties

A red–black tree is a binary search tree where each node has a *color* attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following requirements apply to red–black trees:

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted from other definitions. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves are the same color as the root.
4. Both children of every red node are black.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.



These constraints enforce a critical property of red–black trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst-case, unlike ordinary binary search trees.

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red–black tree  $T$ , let  $B$  be the number of black nodes in property 5. Therefore the shortest possible path from the root of  $T$  to any leaf consists of  $B$  black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore the longest possible path consists of  $2B$  nodes, alternating black and red.

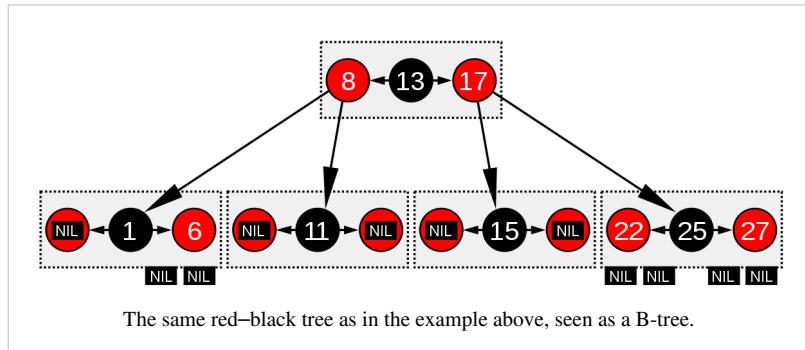
The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

In many of the presentations of tree data structures, it is possible for a node to have only one child, and leaf nodes contain data. It is possible to present red–black trees in this paradigm, but it changes several of the properties and complicates the algorithms. For this reason, this article uses "null leaves", which contain no data and merely serve to indicate where the tree ends, as shown above. These nodes are often omitted in drawings, resulting in a tree that seems to contradict the above principles, but in fact does not. A consequence of this is that all internal (non-leaf) nodes have two children, although one or both of those children may be null leaves. Property 5 ensures that a red node must have either two black null leaves or two black non-leaves as children. For a black node with one null leaf child and one non-null-leaf child, properties 3, 4 and 5 ensure that the non-null-leaf child must be a red node with two black null leaves as children.

Some explain a red–black tree as a binary search tree whose edges, instead of nodes, are colored in red or black, but this does not make any difference. The color of a node in this article's terminology corresponds to the color of the edge connecting the node to its parent, except that the root node is always black (property 2) whereas the corresponding edge does not exist.

## Analogy to B-trees of order 4

A red-black tree is similar in structure to a B-tree of order 4, where each node can contain between 1 to 3 values and (accordingly) between 2 to 4 child pointers. In such B-tree, each node will contain only one value matching the value in a black node of the red-black tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the red-black tree.



One way to see this equivalence is to "move up" the red nodes in a graphical representation of the red-black tree, so that they align horizontally with their parent black node, by creating together a horizontal cluster. In the B-tree, or in the modified graphical representation of the red-black tree, all leaf nodes are at the same depth.

The red-black tree is then structurally equivalent to a B-tree of order 4, with a minimum fill factor of 33% of values per cluster with a maximum capacity of 3 values.

This B-tree type is still more general than a red-black tree though, as it allows ambiguity in a red-black tree conversion—multiple red-black trees can be produced from an equivalent B-tree of order 4. If a B-tree cluster contains only 1 value, it is the minimum, black, and has two child pointers. If a cluster contains 3 values, then the central value will be black and each value stored on its sides will be red. If the cluster contains two values, however, either one can become the black node in the red-black tree (and the other one will be red).

So the order-4 B-tree does not maintain which of the values contained in each cluster is the root black tree for the whole cluster and the parent of the other values in the same cluster. Despite this, the operations on red-black trees are more economical in time because you don't have to maintain the vector of values. It may be costly if values are stored directly in each node rather than being stored by reference. B-tree nodes, however, are more economical in space because you don't need to store the color attribute for each node. Instead, you have to know which slot in the cluster vector is used. If values are stored by reference, e.g. objects, null references can be used and so the cluster can be represented by a vector containing 3 slots for value pointers plus 4 slots for child references in the tree. In that case, the B-tree can be more compact in memory, improving data locality.

The same analogy can be made with B-trees with larger orders that can be structurally equivalent to a colored binary tree: you just need more colors. Suppose that you add blue, then the blue-red-black tree defined like red-black trees but with the additional constraint that no two successive nodes in the hierarchy will be blue and all blue nodes will be children of a red node, then it becomes equivalent to a B-tree whose clusters will have at most 7 values in the following colors: blue, red, blue, black, blue, red, blue (For each cluster, there will be at most 1 black node, 2 red nodes, and 4 blue nodes).

For moderate volumes of values, insertions and deletions in a colored binary tree are faster compared to B-trees because colored trees don't attempt to maximize the fill factor of each horizontal cluster of nodes (only the minimum fill factor is guaranteed in colored binary trees, limiting the number of splits or junctions of clusters). B-trees will be faster for performing rotations (because rotations will frequently occur within the same cluster rather than with multiple separate nodes in a colored binary tree). However for storing large volumes, B-trees will be much faster as they will be more compact by grouping several children in the same cluster where they can be accessed locally.

All optimizations possible in B-trees to increase the average fill factors of clusters are possible in the equivalent multicolored binary tree. Notably, maximizing the average fill factor in a structurally equivalent B-tree is the same as reducing the total height of the multicolored tree, by increasing the number of non-black nodes. The worst case

occurs when all nodes in a colored binary tree are black, the best case occurs when only a third of them are black (and the other two thirds are red nodes).

## Applications and related data structures

Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry can be based on red–black trees, and the Completely Fair Scheduler used in current Linux kernels uses red–black trees.

The AVL tree is another structure supporting  $O(\log n)$  search, insertion, and removal. It is more rigidly balanced than red–black trees, leading to slower insertion and removal but faster retrieval. This makes it attractive for data structures that may be built once and loaded without reconstruction, such as language dictionaries (or program dictionaries, such as the opcodes of an assembler or interpreter).

Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red–black trees requires  $O(\log n)$  space for each insertion or deletion, in addition to time.

For every 2-4 tree, there are corresponding red–black trees with data elements in the same order. The insertion and deletion operations on 2-4 trees are also equivalent to color-flipping and rotations in red–black trees. This makes 2-4 trees an important tool for understanding the logic behind red–black trees, and this is why many introductory algorithm texts introduce 2-4 trees just before red–black trees, even though 2-4 trees are not often used in practice.

In 2008, Sedgewick introduced a simpler version of the red–black tree called the left-leaning red-black tree<sup>[4]</sup> by eliminating a previously unspecified degree of freedom in the implementation. The LLRB maintains an additional invariant that all red links must lean left except during inserts and deletes. Red–black trees can be made isometric to either 2-3 trees,<sup>[5]</sup> or 2-4 trees,<sup>[4]</sup> for any sequence of operations. The 2-4 tree isometry was described in 1978 by Sedgewick. With 2-4 trees, the isometry is resolved by a "color flip," corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node. The tango tree, a type of tree optimized for fast searches, usually uses red–black trees as part of its data structure.

## Operations

Read-only operations on a red–black tree require no modification from those used for binary search trees, because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number ( $O(\log n)$  or amortized  $O(1)$ ) of color changes (which are very quick in practice) and no more than three tree rotations (two for insertion). Although insert and delete operations are complicated, their times remain  $O(\log n)$ .

### Insertion

Insertion begins by adding the node much as binary search tree insertion does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red–black tree leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

What happens next depends on the color of other nearby nodes. The term *uncle node* will be used to refer to the sibling of a node's parent, as in human family trees. Note that:

- property 3 (all leaves are black) always holds.
- property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.

- property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black (or vice versa), or a rotation.

*Note:* The label **N** will be used to denote the current node (colored red). At the beginning, this is the new node being inserted, but the entire procedure may also be applied recursively to other nodes (see case 3). **P** will denote **N**'s parent node, **G** will denote **N**'s grandparent, and **U** will denote **N**'s uncle. Note that in between some cases, the roles and labels of the nodes are exchanged, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions.

Each case will be demonstrated with example C code. The uncle and grandparent nodes can be found by these functions:

```
struct node *grandparent(struct node *n)
{
    if ((n != NULL) && (n->parent != NULL))
        return n->parent->parent;
    else
        return NULL;
}

struct node *uncle(struct node *n)
{
    struct node *g = grandparent(n);
    if (g == NULL)
        return NULL; // No grandparent means no uncle
    if (n->parent == g->left)
        return g->right;
    else
        return g->left;
}
```

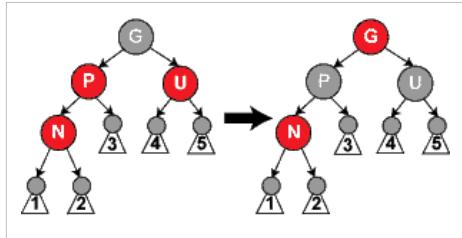
**Case 1:** The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

```
void insert_case1(struct node *n)
{
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(n);
}
```

**Case 2:** The current node's parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

```
void insert_case2(struct node *n)
{
    if (n->parent->color == BLACK)
        return; /* Tree is still valid */
    else
        insert_case3(n);
}
```

*Note:* In the following cases it can be assumed that **N** has a grandparent node **G**, because its parent **P** is red, and if it were the root, it would be black. Thus, **N** also has an uncle node **U**, although it may be a leaf in cases 4 and 5.

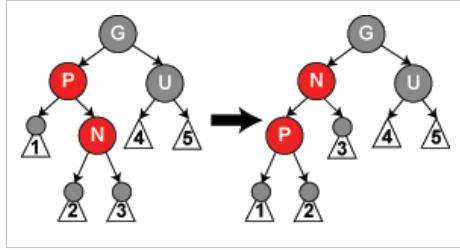


**Case 3:** If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent **G** becomes red (to maintain property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the current red node **N** has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate properties 2 (The root is black) or 4 (Both children of every red node are black) (property 4 possibly being violated since **G** may have a red parent). To fix this, the entire procedure is recursively performed on **G** from case 1. Note that this is a tail-recursive call, so it could be rewritten as a loop; since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur.

```
void insert_case3(struct node *n)
{
    struct node *u = uncle(n), *g;

    if ((u != NULL) && (u->color == RED)) {
        n->parent->color = BLACK;
        u->color = BLACK;
        g = grandparent(n);
        g->color = RED;
        insert_case1(g);
    } else {
        insert_case4(n);
    }
}
```

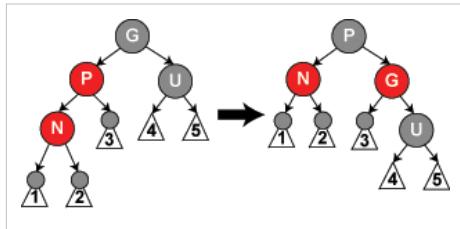
*Note:* In the remaining cases, it is assumed that the parent node **P** is the left child of its parent. If it is the right child, *left* and *right* should be reversed throughout cases 4 and 5. The code samples take care of this.



**Case 4:** The parent **P** is red but the uncle **U** is black; also, the current node **N** is the right child of **P**, and **P** in turn is the left child of its parent **G**. In this case, a left rotation that switches the roles of the current node **N** and its parent **P** can be performed; then, the former parent node **P** is dealt with using case 5 (relabeling **N** and **P**) because property 4 (both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node **N** where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node **P** where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to case 5.

```
void insert_case4(struct node *n)
{
    struct node *g = grandparent(n);

    if ((n == n->parent->right) && (n->parent == g->left)) {
        rotate_left(n->parent);
        n = n->left;
    } else if ((n == n->parent->left) && (n->parent == g->right)) {
        rotate_right(n->parent);
        n = n->right;
    }
    insert_case5(n);
}
```



**Case 5:** The parent **P** is red but the uncle **U** is black, the current node **N** is the left child of **P**, and **P** is the left child of its parent **G**. In this case, a right rotation on **G** is performed; the result is a tree where the former parent **P** is now the parent of both the current node **N** and the former grandparent **G**. **G** is known to be black, since its former child **P** could not have been red otherwise (without violating property 4). Then, the colors of **P** and **G** are switched, and the resulting tree satisfies property 4 (both children of every red node are black). Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through **G** before, and now they all go through **P**. In each case, this is the only black node of the three.

```
void insert_case5(struct node *n)
{
    struct node *g = grandparent(n);

    n->parent->color = BLACK;
    g->color = RED;
```

```

if (n == n->parent->left)
    rotate_right(g);
else
    rotate_left(g);
}

```

Note that inserting is actually in-place, since all the calls above use tail recursion.

## Removal

In a regular binary search tree when deleting a node with two non-leaf children, we find either the maximum element in its left subtree (which is the in-order predecessor) or the minimum element in its right subtree (which is the in-order successor) and move its value into the node being deleted (as shown here). We then delete the node we copied the value from, which must have fewer than two non-leaf children. (Non-leaf children, rather than all children, are specified here because unlike normal binary search trees, red-black trees have leaf nodes anywhere they can have them, so that all nodes are either internal nodes with two children or leaf nodes with, by definition, zero children. In effect, internal nodes having two leaf children in a red-black tree are like the leaf nodes in a regular binary search tree.) Because merely copying a value does not violate any red-black properties, this reduces to the problem of deleting a node with at most one non-leaf child. Once we have solved that problem, the solution applies equally to the case where the node we originally want to delete has at most one non-leaf child as to the case just considered where it has two non-leaf children.

Therefore, for the remainder of this discussion we address the deletion of a node with at most one non-leaf child. We use the label **M** to denote the node to be deleted; **C** will denote a selected child of **M**, which we will also call "its child". If **M** does have a non-leaf child, call that its child, **C**; otherwise, choose either leaf as its child, **C**.

If **M** is a red node, we simply replace it with its child **C**, which must be black by property 4. (This can only occur when **M** has two leaf children, because if the red node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would violate property 5.) All paths through the deleted node will simply pass through one less red node, and both the deleted node's parent and child must be black, so property 3 (all leaves are black) and property 4 (both children of every red node are black) still hold.

Another simple case is when **M** is black and **C** is red. Simply removing a black node could break Properties 4 ("Both children of every red node are black") and 5 ("All paths from any given node to its leaf nodes contain the same number of black nodes"), but if we repaint **C** black, both of these properties are preserved.

The complex case is when both **M** and **C** are black. (This can only occur when deleting a black node which has two leaf children, because if the black node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would have been an invalid red-black tree by violation of property 5.) We begin by replacing **M** with its child **C**. We will call (or *label*—that is, *relabel*) this child (in its new position) **N**, and its sibling (its new parent's other child) **S**. (**S** was previously the sibling of **M**.) In the diagrams below, we will also use **P** for **N**'s new parent (**M**'s old parent), **S<sub>L</sub>** for **S**'s left child, and **S<sub>R</sub>** for **S**'s right child (**S** cannot be a leaf because if **N** is black, which we presumed, then **P**'s one subtree which includes **N** counts two black-height and thus **P**'s other subtree which includes **S** must also count two black-height, which cannot be the case if **S** is a leaf node).

*Note:* In between some cases, we exchange the roles and labels of the nodes, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. White represents an unknown color (either red or black).

We will find the sibling using this function:

```
struct node *sibling(struct node *n)
{
    if (n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
}
```

*Note:* In order that the tree remains well-defined, we need that every null leaf remains a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child **N**, it is easy to see that the property is satisfied. If, on the other hand, **N** would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

We can perform the steps outlined above with the following code, where the function `replace_node` substitutes `child` into `n`'s place in the tree. For convenience, code in this section will assume that null leaves are represented by actual node objects rather than `NULL` (the code in the *Insertion* section works with either representation).

```
void delete_one_child(struct node *n)
{
    /*
     * Precondition: n has at most one non-null child.
     */
    struct node *child = is_leaf(n->right) ? n->left : n->right;

    replace_node(n, child);
    if (n->color == BLACK) {
        if (child->color == RED)
            child->color = BLACK;
        else
            delete_case1(child);
    }
    free(n);
}
```

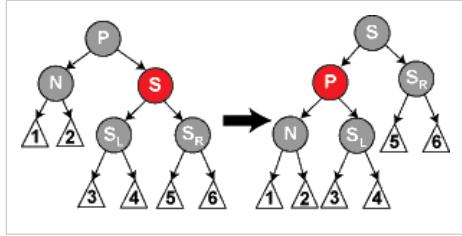
*Note:* If **N** is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling `delete_case1()` on its parent (the node that we delete, `n` in the code above) and deleting it afterwards. We can do this because the parent is black, so it behaves in the same way as a null leaf (and is sometimes called a 'phantom' leaf). And we can safely delete it at the end as `n` will remain a leaf after all operations, as shown above.

If both **N** and its original parent are black, then deleting this original parent causes paths which proceed through **N** to have one fewer black node than paths that do not. As this violates property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

**Case 1:** **N** is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

```
void delete_case1(struct node *n)
{
    if (n->parent != NULL)
        delete_case2(n);
}
```

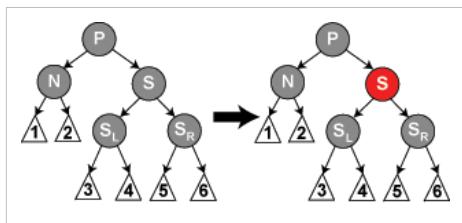
*Note:* In cases 2, 5, and 6, we assume **N** is the left child of its parent **P**. If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.



**Case 2:** **S** is red. In this case we reverse the colors of **P** and **S**, and then rotate left at **P**, turning **S** into **N**'s grandparent. Note that **P** has to be black as it had a red child. Although all paths still have the same number of black nodes, now **N** has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red **S**.) In later cases, we will relabel **N**'s new sibling as **S**.

```
void delete_case2(struct node *n)
{
    struct node *s = sibling(n);

    if (s->color == RED) {
        n->parent->color = RED;
        s->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    delete_case3(n);
}
```



**Case 3:** **P**, **S**, and **S**'s children are black. In this case, we simply repaint **S** red. The result is that all paths passing through **S**, which are precisely those paths *not* passing through **N**, have one less black node. Because deleting **N**'s original parent made all paths passing through **N** have one less black node, this evens things up. However, all paths through **P** now have one fewer black node than paths that do not pass through **P**, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is still violated. To correct this, we perform the rebalancing procedure on **P**, starting at case 1.

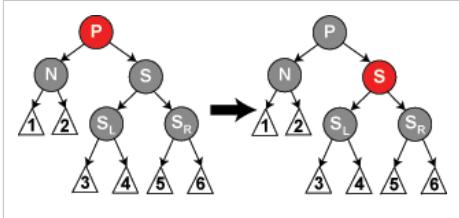
```
void delete_case3(struct node *n)
{
    struct node *s = sibling(n);

    if ((n->parent->color == BLACK) &&
        (s->color == BLACK) &&
        (s->left->color == BLACK) &&
        (s->right->color == BLACK)) {
```

```

        s->color = RED;
        delete_case1(n->parent);
    } else
        delete_case4(n);
}

```



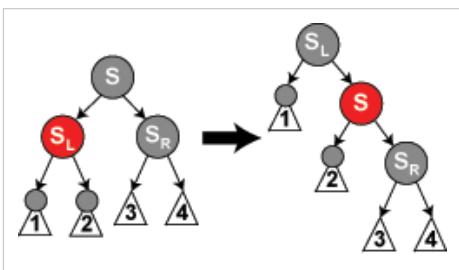
**Case 4:** S and S's children are black, but P is red. In this case, we simply exchange the colors of S and P. This does not affect the number of black nodes on paths going through S, but it does add one to the number of black nodes on paths going through N, making up for the deleted black node on those paths.

```

void delete_case4(struct node *n)
{
    struct node *s = sibling(n);

    if ((n->parent->color == RED) &&
        (s->color == BLACK) &&
        (s->left->color == BLACK) &&
        (s->right->color == BLACK)) {
        s->color = RED;
        n->parent->color = BLACK;
    } else
        delete_case5(n);
}

```



**Case 5:** S is black, S's left child is red, S's right child is black, and N is the left child of its parent. In this case we rotate right at S, so that S's left child becomes S's parent and N's new sibling. We then exchange the colors of S and its new parent. All paths still have the same number of black nodes, but now N has a black sibling whose right child is red, so we fall into case 6. Neither N nor its parent are affected by this transformation. (Again, for case 6, we relabel N's new sibling as S.)

```

void delete_case5(struct node *n)
{
    struct node *s = sibling(n);

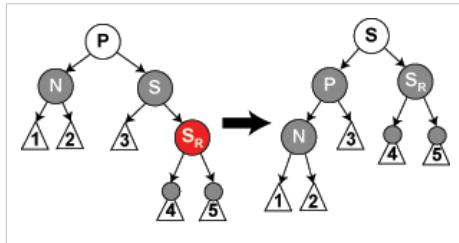
    if (s->color == BLACK) { /* this if statement is trivial,

```

```

due to case 2 (even though case 2 changed the sibling to a sibling's
child,
the sibling's child can't be red, since no red parent can have a red
child). */
/* the following statements just force the red to be on the left of the
left of the parent,
or right of the right, so case six will rotate correctly. */
if ((n == n->parent->left) &&
      (s->right->color == BLACK) &&
      (s->left->color == RED)) { /* this last test is trivial
too due to cases 2-4. */
    s->color = RED;
    s->left->color = BLACK;
    rotate_right(s);
} else if ((n == n->parent->right) &&
            (s->left->color == BLACK) &&
            (s->right->color == RED)) { /* this last test is
trivial too due to cases 2-4. */
    s->color = RED;
    s->right->color = BLACK;
    rotate_left(s);
}
delete_case6(n);
}

```



**Case 6:** **S** is black, **S**'s right child is red, and **N** is the left child of its parent **P**. In this case we rotate left at **P**, so that **S** becomes the parent of **P** and **S**'s right child. We then exchange the colors of **P** and **S**, and make **S**'s right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, **N** now has one additional black ancestor: either **P** has become black, or it was black and **S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node.

Meanwhile, if a path does not go through **N**, then there are two possibilities:

- It goes through **N**'s new sibling. Then, it must go through **S** and **P**, both formerly and currently, as they have only exchanged colors and places. Thus the path contains the same number of black nodes.
- It goes through **N**'s new uncle, **S**'s right child. Then, it formerly went through **S**, **S**'s parent, and **S**'s right child (which was red), but now only goes through **S**, which has assumed the color of its former parent, and **S**'s right child, which has changed from red to black (assuming **S**'s color: black). The net effect is that this path goes through the same number of black nodes.

Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation.

```

void delete_case6(struct node *n)
{
    struct node *s = sibling(n);

    s->color = n->parent->color;
    n->parent->color = BLACK;

    if (n == n->parent->left) {
        s->right->color = BLACK;
        rotate_left(n->parent);
    } else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}

```

Again, the function calls all use tail recursion, so the algorithm is in-place. In the algorithm above, all cases are chained in order, except in delete case 3 where it can recurse to case 1 back to the parent node: this is the only case where an in-place implementation will effectively loop (after only one rotation in case 3).

Additionally, no tail recursion ever occurs on a child node, so the tail recursion loop can only move from a child back to its successive ancestors. No more than  $O(\log n)$  loops back to case 1 will occur (where  $n$  is the total number of nodes in the tree before deletion). If a rotation occurs in case 2 (which is the only possibility of rotation within the loop of cases 1–3), then the parent of the node  $N$  becomes red after the rotation and we will exit the loop. Therefore at most one rotation will occur within this loop. Since no more than two additional rotations will occur after exiting the loop, at most three rotations occur in total.

## Insertion complexity

In the tree code there is only one loop where the node of the root of the red–black property that we wish to restore,  $x$ , can be moved up the tree by one level at each iteration.

Since the original height of the tree is  $O(\log n)$ , there are  $O(\log n)$  iterations. So overall the insert routine has  $O(\log n)$  complexity.

# Splay Tree

<b>SPLAY-SEARCH(key)</b> $x = \text{TREE-SEARCH}(key)$ if $x = \text{NIL}$ $\text{SPLAY}(x)$	<b>SPLAY-INSERT(x)</b> $\text{TREE-INSERT}(x)$ $\text{SPLAY}(x)$	<b>SPLAY-DELETE(x)</b> $T, S = \text{SPLIT}(x)$ if $S.\text{left} \neq \text{NIL}$ $S.\text{left.parent} = \text{NIL}$ $\text{JOIN}(S.\text{left}, T)$
---	--	--

```

SPLAY(x)
while x.parent ≠ NIL
    if x.parent.parent == NIL
        if x == x.parent.left
            // zig rotation
            RIGHT-ROTATE(x.parent)
        else
            // zag rotation
            LEFT-ROTATE(x.parent)
    else if x==x.parent.left and x.parent == x.parent.parent.left
        // zig-zig rotation
        RIGHT-ROTATE(x.parent.parent)
        RIGHT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.right      // zag-zag
rotation
        LEFT-ROTATE(x.parent.parent)
        LEFT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.left
        // zig-zag rotation
        LEFT-ROTATE(x.parent)
        RIGHT-ROTATE(x.parent)
    else
        // zag-zig rotation
        RIGHT-ROTATE(x.parent)
        LEFT-ROTATE(x.parent)

JOIN(S, T)
if S == NIL
    return T
if T == NIL
    return S
x = TREE-MAXIMUM(S)
SPLAY(x)
x.right = t
t.parent = x
return x

SPLIT(x)
SPLAY(x)
if x.right ≠ NIL
    T = x.right
    T.parent = NIL
else
    T = NIL
S = x
S.right = NIL
x = NIL
return S, T

```

# Splay Tree

Splay trees are self branching binary search tree which has the property of reaccessing the elements quickly that which are recently accessed. Splay trees have basic operations such as Insertion, Search, Deletion. The performance of the splay trees are much efficient than other search trees.

Splaying is the basic operation for the splay trees which rearranges the tree so that element is placed at the root of the tree. The performance of the splay trees depends on the self balancing and self optimizing. The nodes of the tree are moved closer to the root so that they can be accessed quickly. In real time splay trees are used for implementing caches. The worst case with this splay tree algorithm is that this will sequentially access all the elements of the tree which makes the tree unbalanced. The reaccessing of the first element takes more time as it has to access all the remaining elements and then it starts accessing the first. If the access pattern is nonuniform, it needs an extra space for storage of balance information.

The real time applications of the splay trees are

- It is used to implement caches. Cache keeps track of the contents of memory locations that were recently requested by processor. It can be made to deliver requests much faster than main memory. Similarly, splay trees uses this concept of accessing the elements quickly that which were recently accessed. Splay trees are used to implement Cache algorithms.
- It has the ability of not to store any data, which results in minimization of memory requirements.
- It can also be used for data compression, e.g. dynamic huffman coding.

The main ideology behind selecting the splay tree is to reduce the consumption of time while reaccessing of the elements and to improve the performance and also to reduce the storage space.

## HISTORY

Splay trees are invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985. Splay trees are self adjusting binary search tree, these trees perform better than other search trees. It performs basic operations such as insertion, search and deletion in amortized time.

## ADVANTAGES

- It is easy to implement than other self balancing binary search trees, such as Red black trees or AVL trees. Much simpler to code than AVL, Red Black trees.
- Require less space as no balance information is required.
- Can be much more efficient if usage pattern is skewed. Good heuristics can have amortized running time similar to the running time of the best structure.

## DISADVANTAGES

- More local adjustments during Search operations. Searching the key starts from the root node. If key is not found in the left node which has nodes less than the given key, then it searches in the right node which has nodes greater than the key. In order to splay the searched key to root of the tree more local adjustments has to be made.
- Individual operations can be expensive . drawback for real-time applications.
- The main disadvantage of splay trees is that the height of a splay tree can be linear. After accessing all  $n$  elements in non-decreasing order, Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be slow. However the amortized access cost of this worse case is logarithmic,  $O(\log n)$ . Splay trees can change even when they are accessed in a 'read-only' manner. This complicates the use of splay trees in a multi-threaded environment. Specifically, extra management is needed if multiple threads are allowed to perform search operations.

The disadvantage of the splay tree is the height. When we access all the elements of the tree in order which corresponds to the worst case access time, which means the cost of each operation can be slow. I would like to solve this issue regarding the height of the tree which causes the tree to consume more time to access. Amortized cost for locating an item in the tree would be  $O(\log n)$ , where

$n$  is the number of items in the tree. The cost of an individual operation is  $O(\log n)$  on average, but an individual operation can be more expensive.

## Operations of Splay Tree

Splay trees are used to implement caches and also used for the minimization of memory requirements. Splay trees are self adjusting binary search trees which performs basic operations such as

<ul style="list-style-type: none"> <li>• Search</li> <li>• Insertion</li> <li>• Deletion</li> </ul>	<pre> SPLAY-SEARCH(key)   x = TREE-SEARCH(key)   if x = NIL     SPLAY(x) </pre>	<pre> SPLAY-INSERT(x)   TREE-INSERT(x)   SPLAY(x) </pre>	<pre> SPLAY-DELETE(x)   T, S = SPLIT(x)   if S.left ≠ NIL     S.left.parent = NIL   JOIN(S.left, T) </pre>
---	---	--	--

Search,insert,delete operations are like in Binary Search trees, except at the end of each operation, a special step called Splaying is done. Splaying the tree rearranges the tree so that element is placed at the root of the tree. It uses tree rotations to bring the element to the top.

The main basic opeations of the Splay trees are

- Search
- Insert
- Delete
- Splaying

### When to Splay

- **Search:** Splay node where key was found.
- **Insert:** When an item is inserted , a Splay is performed. As a result, the newly inserted node becomes the root of the tree.
- **Delete:** Splay parent of removed node which is either the node with the deleted key or its successor.

## Splaying

To perform a splay operation , there is need to carry out sequence of Splay steps, which moves the node closer to the root. The recently accessed nodes are kept closer to the root so that tree remains balanced.

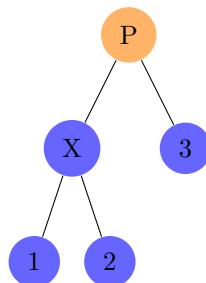
Each Splay step depends on three factors

- X is left or right child of its parent node,P.
- Check P is root node or not,if not
- P is left or right child of its parent G.

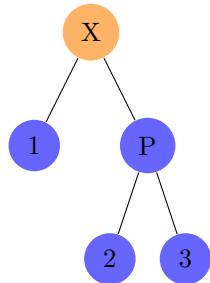
The three types of Splay steps are

### Zig Step

- Let X be a non-root node on the access path on which we are rotating.
- If the parent of X is the root of the tree, we merely rotate X and the root.
- This is the same as a normal single rotation.

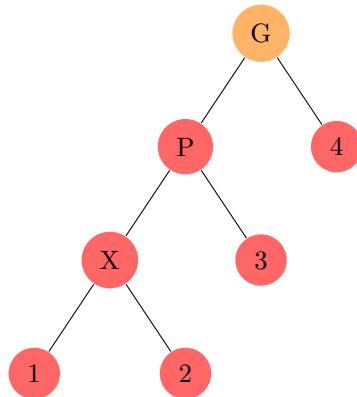


After performing the **Zig step** the tree is transformed with X as root and the tree is rotated as edge between X and P.

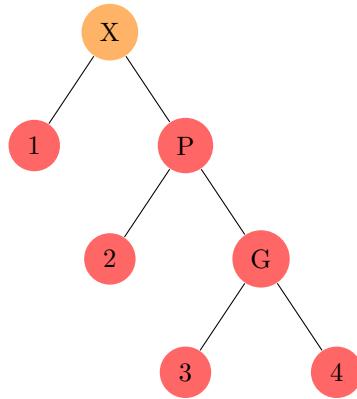


### Zig-Zig step

- This is a different rotation from the previous one.
- Here X and P are either both left children or both right children.
- This is different from the rotate-to-root. Rotate-to-root rotates between X and P and then between X and G. The zig-zig splay rotates between P and G and X and P.

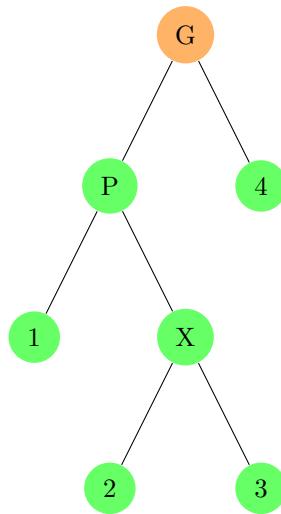


After **Zig-Zig step** is performed, X becomes the root for the transformed tree. Tree is rotated on the edge joining P and its parent G.

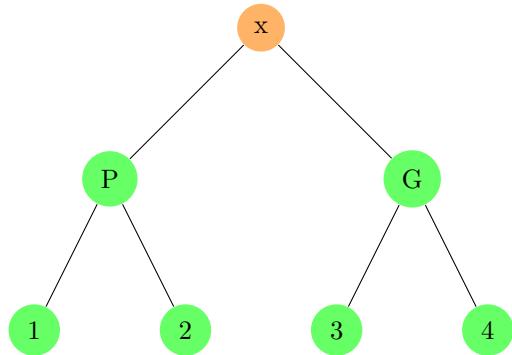


### Zig-Zag step

- In this case, X and both a parent P and a grandparent G. X is a right child and P is a left child or vice versa.
- This is same as the double rotation.



After **zig-zag** step is performed, X becomes the root for the transformed tree. The tree is rotated on the edge between X and P , then rotated between X and G.



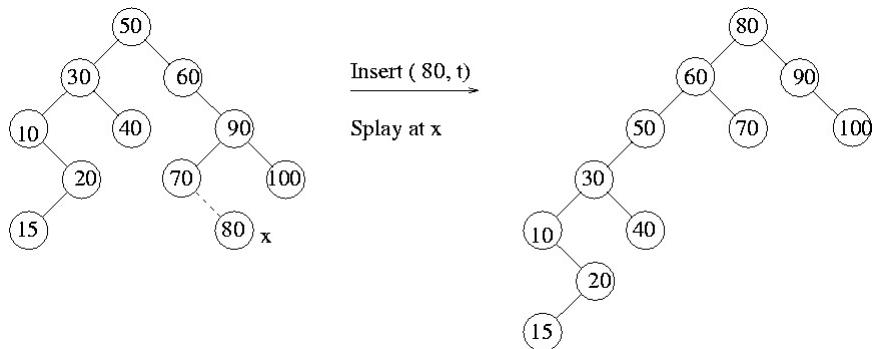
### Results of Splaying

- The result is a binary tree, with the left subtree having all keys less than the root, and the right subtree having keys greater than the root.
- The resulted tree is more balanced than the original tree.
- If an operation near the root is done, the tree can become less balanced.

### Insertion

To insert a node in to the tree

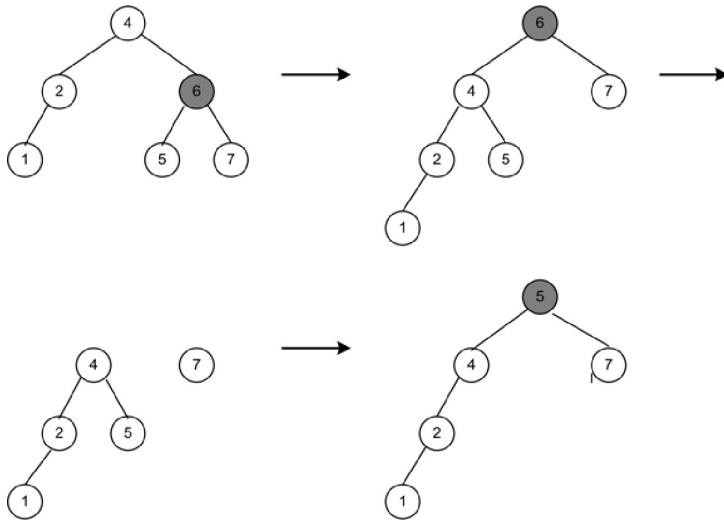
- Insert a node normally in to the tree.
- Splay the newly inserted node to the top of the tree.



## Deletion

- Access the node to be deleted bringing it to the root.
- Delete the root leaving two subtrees L left and R right.
- Find the largest element in L, thus the root of L will have no right child.
- Make R the right child of L 's root.

Delete element 6

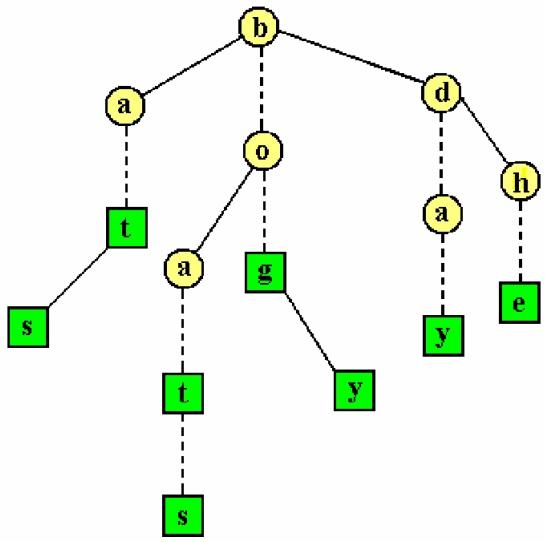


## Splay Tree Application

In this section, an application for splay tree will be introduced. The application is **Lexicographic Search Tree**.

In a LST, all the circles and squares are the nodes of the tree. Each node will contain a character and the square nodes are the terminal nodes of the strings. Solid line edges form the splay tree, and the nodes connected by a solid line represents different strings. The dashed line edges used to represent a single string. From the figure, it stores different words "at", "as", "bat", "bats", "bog", "boy", "day" and "he". When a string is requested, we just splay the character in the string one by one. In LST, after the string is accessed, the

first character of the string will become the root, as a result the most frequent accessed string will be near the root.

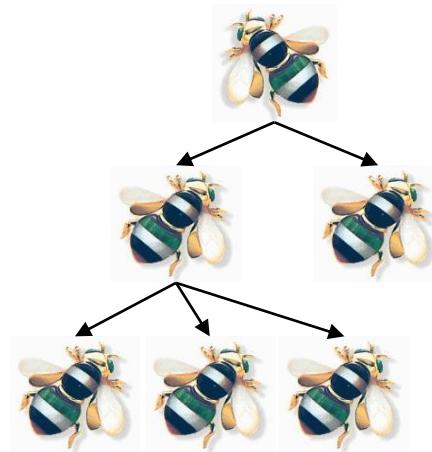


## TIME COMPLEXITY

For  $m$  operations to be performed on splay tree it requires  $O(m \log n)$  steps where  $n$  is the size of the tree. Therefore the amortized complexity of the splay tree is  $O(\log n)$ . The time complexity of maintaining a splay trees is analyzed using an Amortized Analysis. Each tree operation has actual costs proportional to its running time. Each splay operation has amortized cost  $O(\log n)$  thereby on an average the time complexity of each operation on the splay tree is

- **Search:** Searching for a node in the tree would take  $O(\log n)$ .
- **Insert:** Inserting a node in to the tree takes  $O(\log n)$ .
- **Delete:** Deleting a node from the tree takes  $O(\log n)$ .

# B- Trees



# Motivation

- Large differences between time access to disk, cash memory and core memory
- Minimize expensive access  
(e.g., disk access)
- B-tree: Dynamic sets that is optimized for disks

# B-Trees

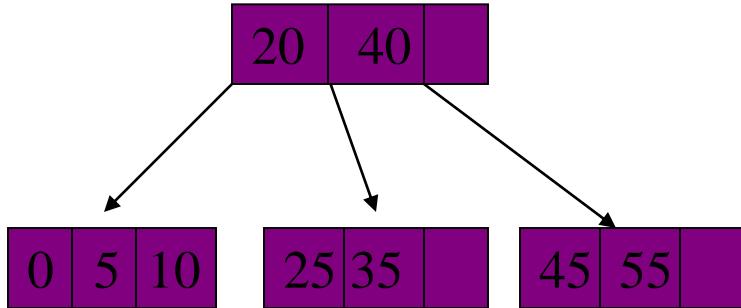
A **B-tree** is an  $M$ -way search tree with two properties :

1. It is perfectly balanced: every leaf node is at the same depth
2. Every internal node other than the root, is at least half-full, i.e.  $M/2-1 \leq \#keys \leq M-1$
3. Every internal node with  $k$  keys has  $k+1$  non-null children

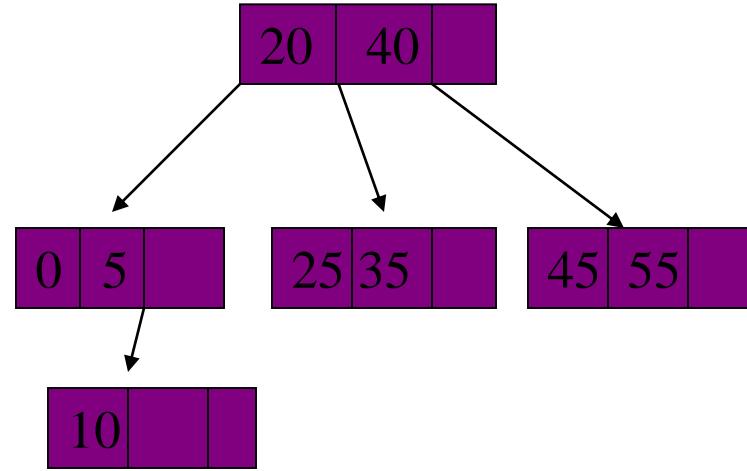
For simplicity we consider  $M$  even and we use  $t=M/2$ :

- 2.\* Every internal node other than the root is at least half-full, i.e.  $t-1 \leq \#keys \leq 2t-1$ ,  $t \leq \#children \leq 2t$

# Example: a 4-way B-tree



*B-tree*



*4-way tree*

## B-tree

1. It is perfectly balanced: every leaf node is at the same depth.
2. Every node, except maybe the root, is at least half-full  
 $t-1 \leq \#keys \leq 2t-1$
3. Every internal node with  $k$  keys has  $k+1$  non-null children

# B-tree Height

**Claim:** any B-tree with  $n$  keys, height  $h$  and minimum degree  $t$  satisfies:

$$h \leq \log_t \frac{n+1}{2}$$

**Proof:**

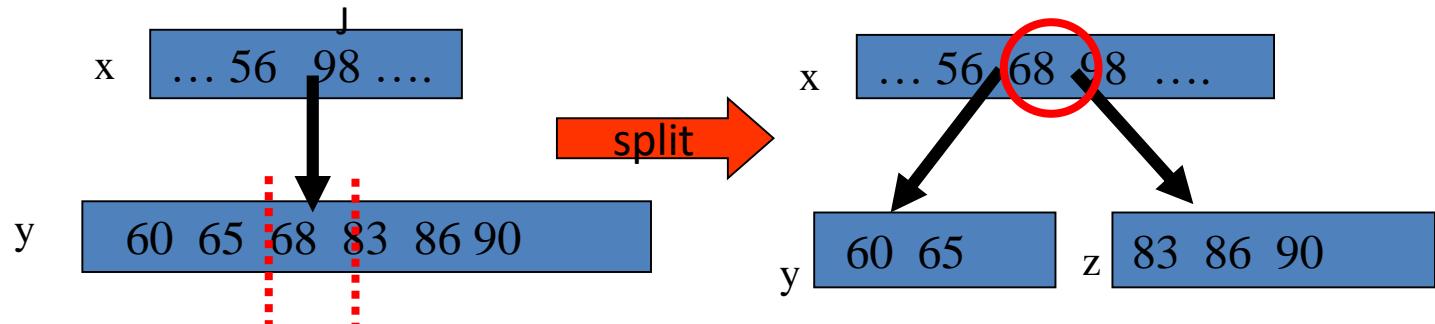
- The minimum number of KEYS for a tree with height  $h$  is obtained when:
  - The root contains one key
  - All other nodes contain  $t-1$  keys

# B-Tree: Insert X

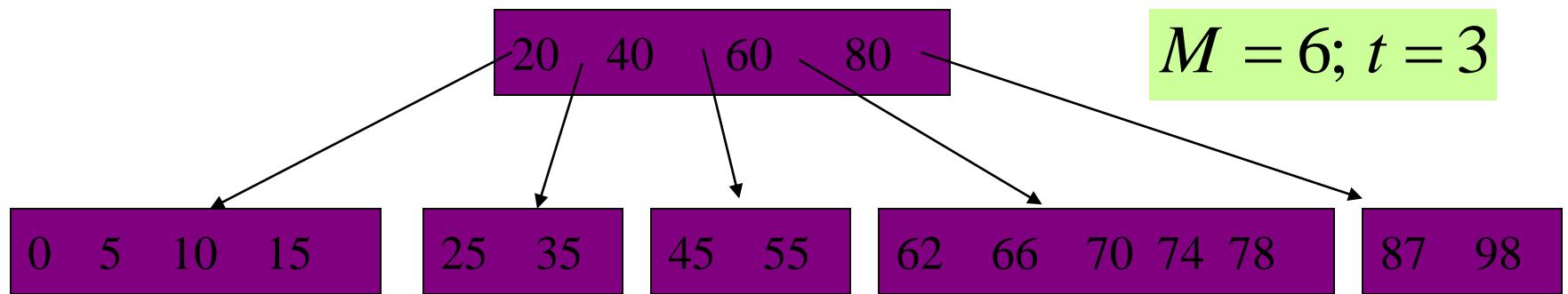
1. As in  $M$ -way tree find the leaf node to which  $X$  should be added
2. Add  $X$  to this node in the appropriate place among the values already there  
*(there are no subtrees to worry about)*
3. Number of values in the node after adding the key:
  - Fewer than  $2t-1$ : done
  - Equal to  $2t$ : *overflowed*
4. Fix overflowed node

# Fix an Overflowed

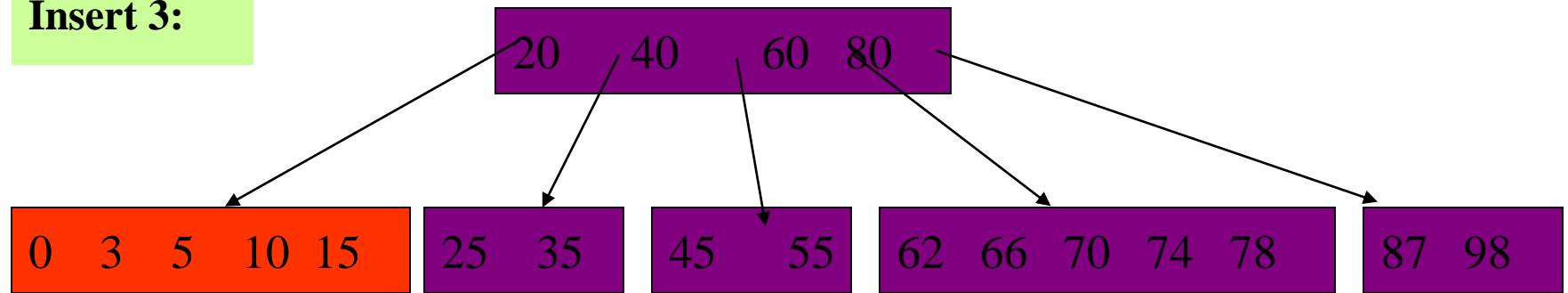
1. Split the node into three parts,  $M=2t$ :
  - **Left**: the first  $t$  values, become a left child node
  - **Middle**: the middle value at position  $t$ , goes up to parent
  - **Right**: the last  $t-1$  values, become a right child node
2. Continue with the parent:
  1. Until no overflow occurs in the parent
  2. If the root overflows, split it too, and create a new root node

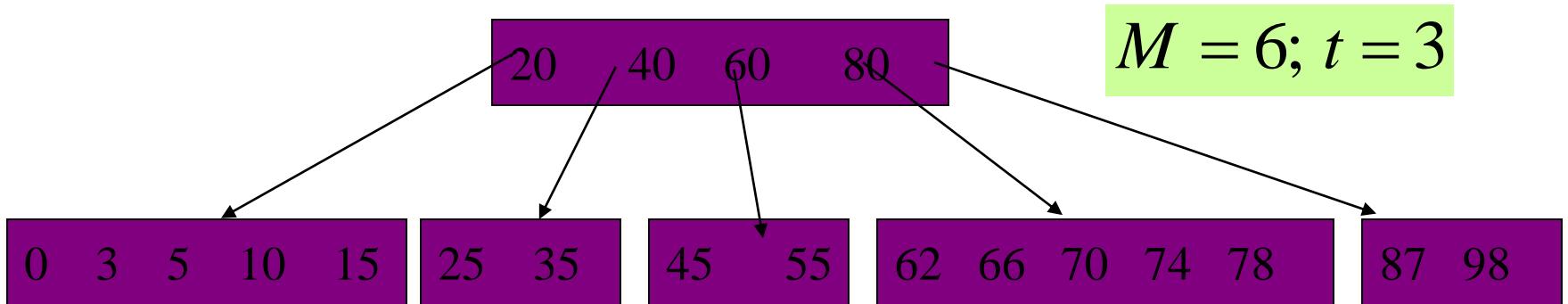


# Insert example

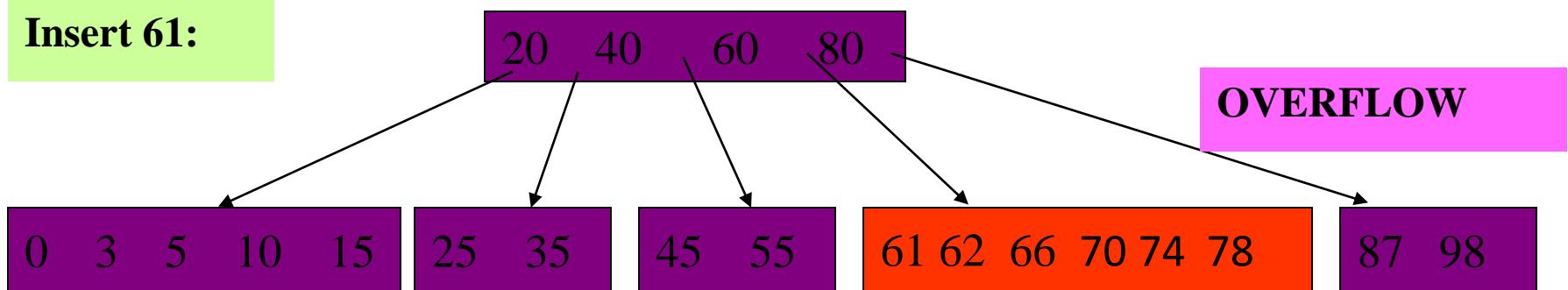


Insert 3:

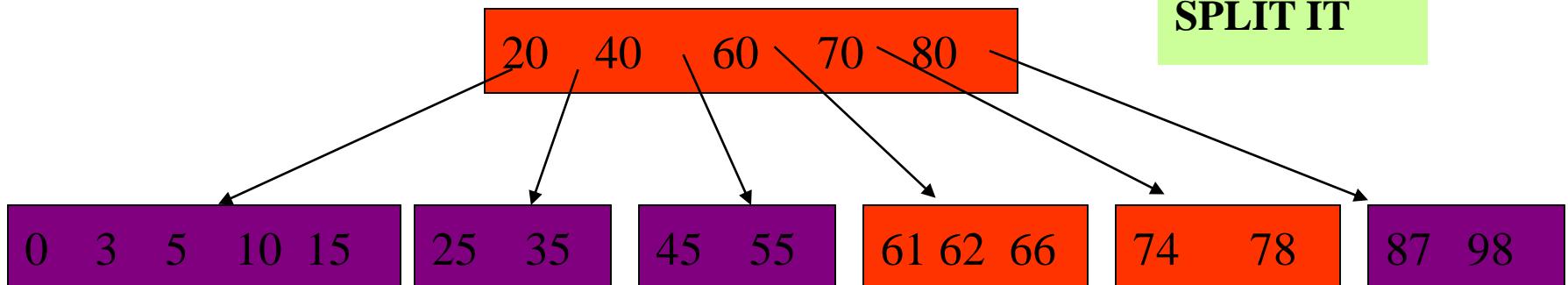




**Insert 61:**

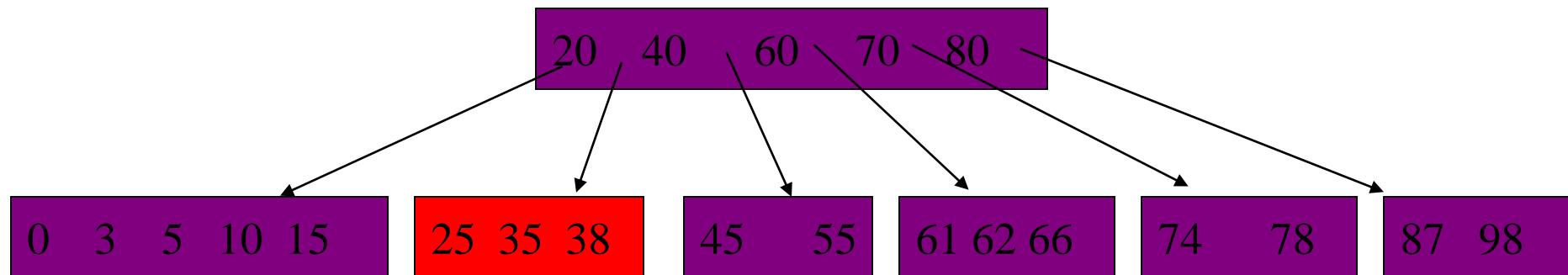
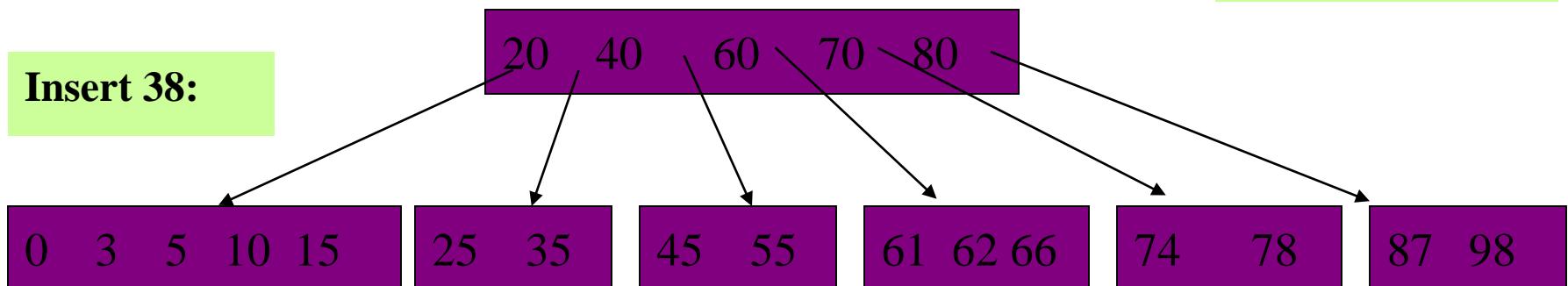


**SPLIT IT**



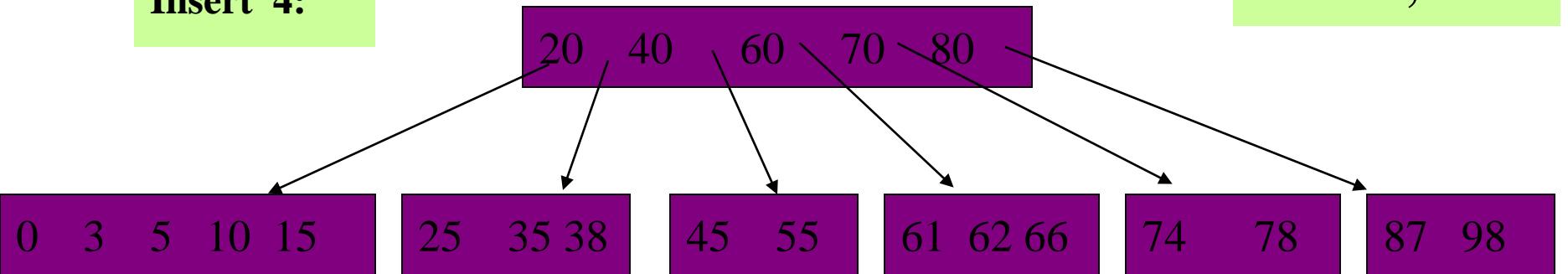
$$M = 6; t = 3$$

Insert 38:

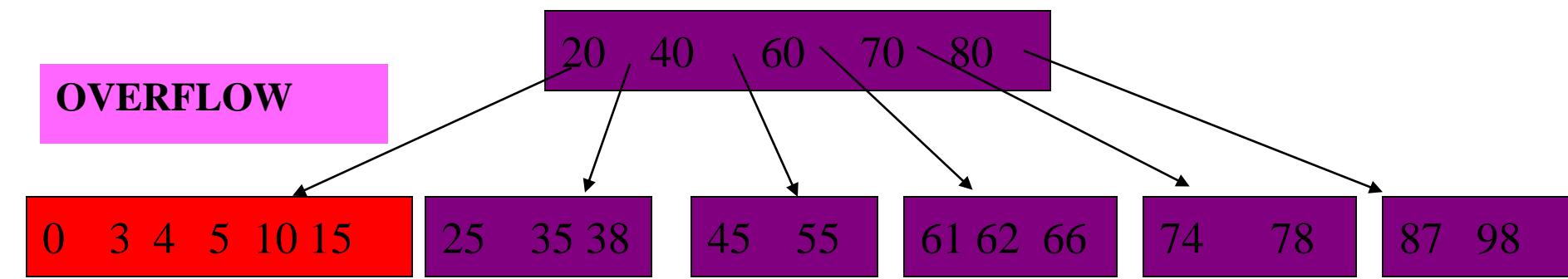


**Insert 4:**

$M = 6; t = 3$



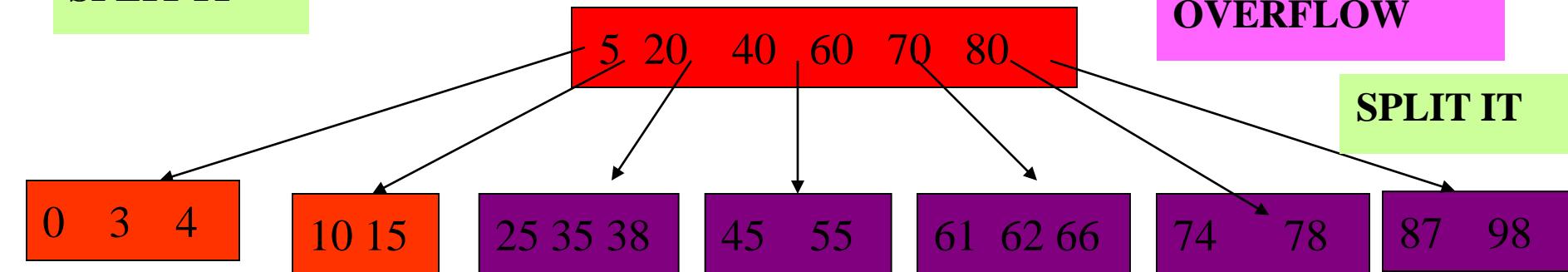
**OVERFLOW**



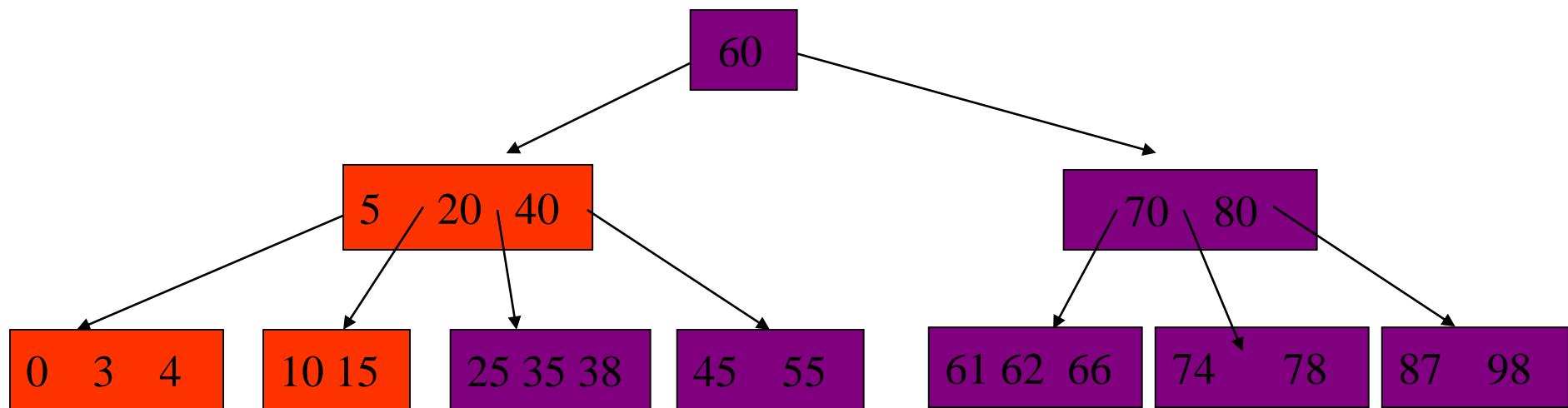
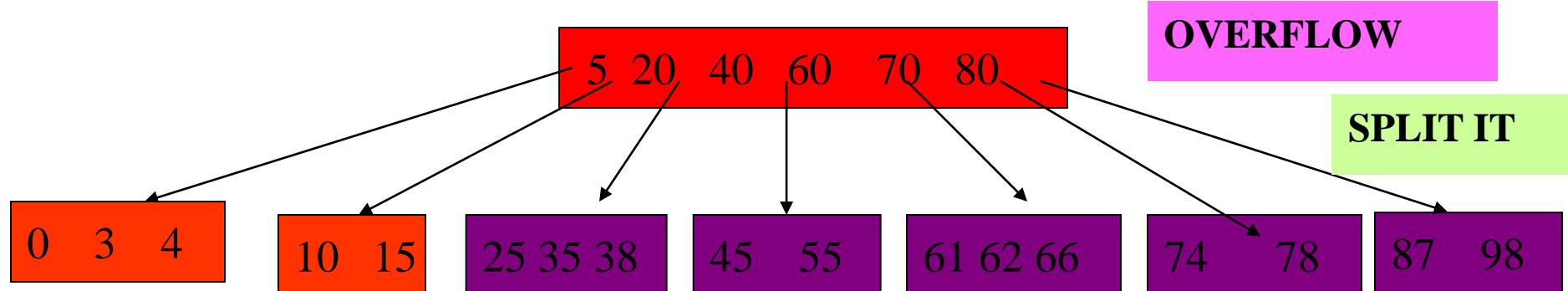
**SPLIT IT**

**OVERFLOW**

**SPLIT IT**



$M = 6; t = 3$



# Complexity Insert

- Inserting a key into a B-tree of height  $h$  is done in a single pass down the tree and a single pass up the tree

Complexity:  $O(h) = O(\log_t n)$

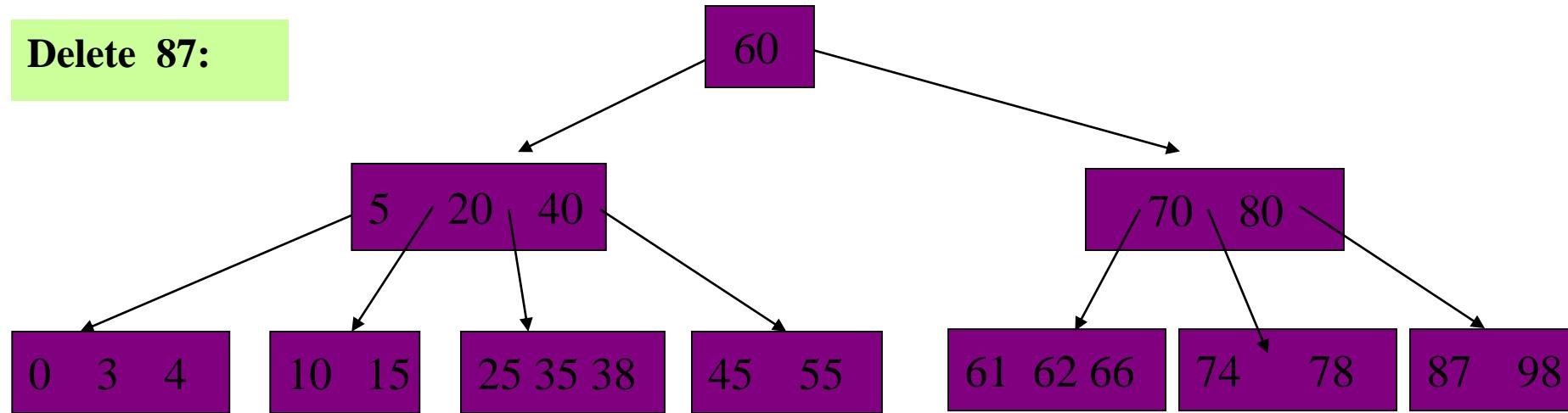
# *B-Tree: Delete X*

- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node <  $t-1$

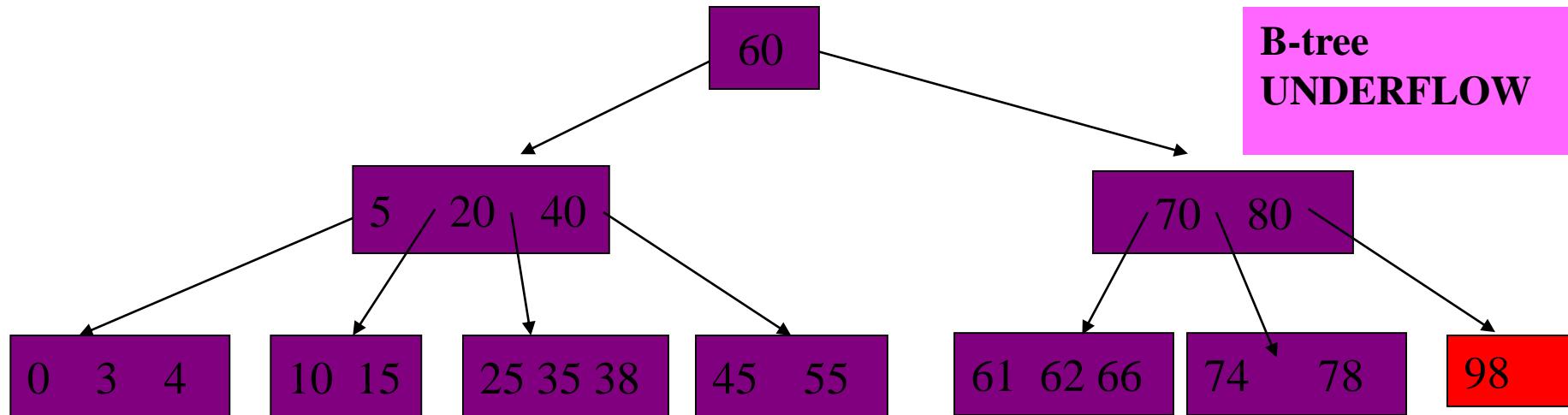
Recall: The root should have at least 1 value in it, and all other nodes should have at least  $t-1$  values in them

# Underflow Example

Delete 87:



B-tree  
UNDERFLOW



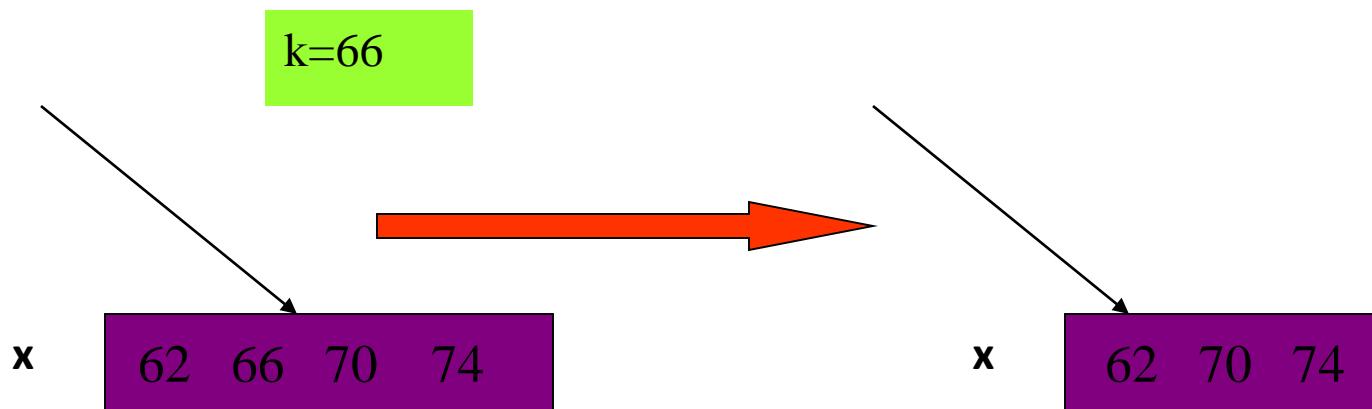
# *B-Tree: Delete X,k*

- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node  $< t-1$
- *Solution:*
  - make sure a node that is visited has at least  $t$  instead of  $t-1$  keys.
  - If it doesn't have  $k$ 
    - (1) either take from sibling via a rotate, or
    - (2) merge with the parent
  - If it does have  $k$ 
    - See next slides

Recall: The root should have at least 1 value in it, and all other nodes should have at least  $t-1$  (at most  $2t-1$ ) values in them

# B-Tree-Delete ( $x, k$ )

**1st case:**  $k$  is in  $x$  and  $x$  is a *leaf*  $\rightarrow$  delete  $k$



How many keys are left?

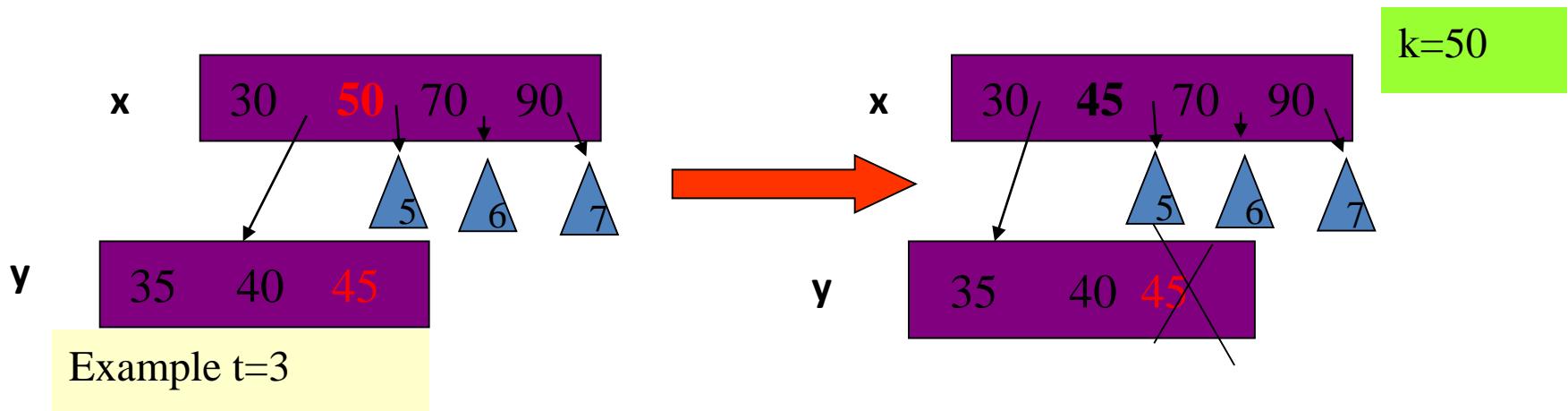
Example  $t=3$

**2nd case:**  $k$  in the internal node  $x$ ,  $y$  and  $z$  are the preceding and succeeding nodes of the key  $k \in x$

a. If  $y$  has at least  $t$  keys:

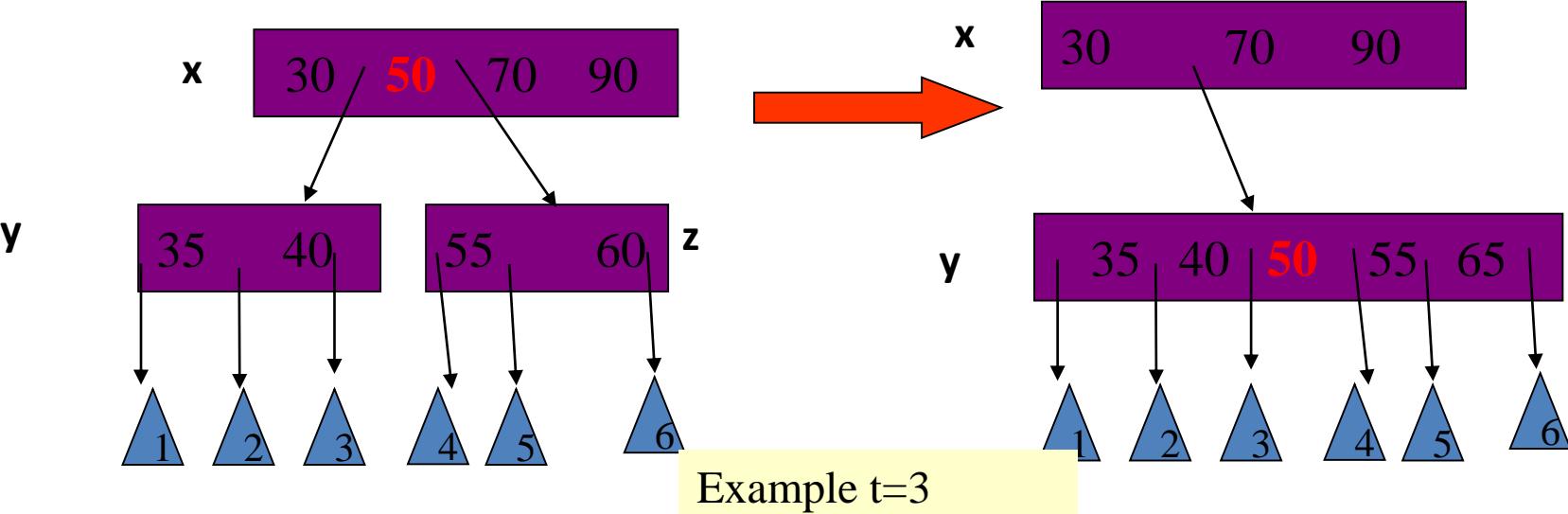
- ▷ Replace  $k$  in  $x$  by  $k' \in y$ , where  $k'$  is the predecessor of  $k$  in  $y$
- ▷ Delete  $k'$  recursively

b. Similar check for successor case



## 2nd case cont.:

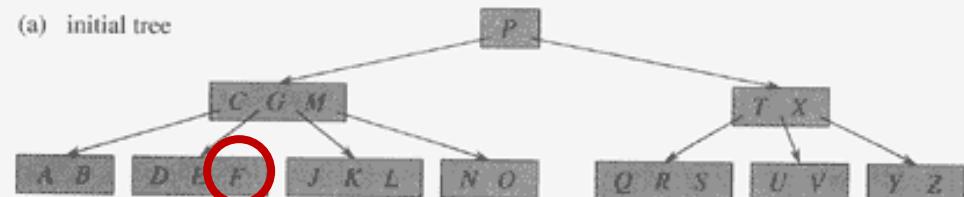
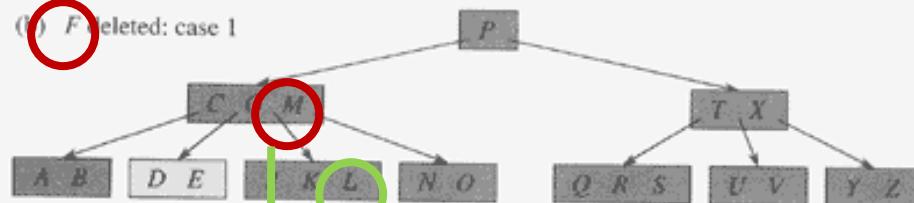
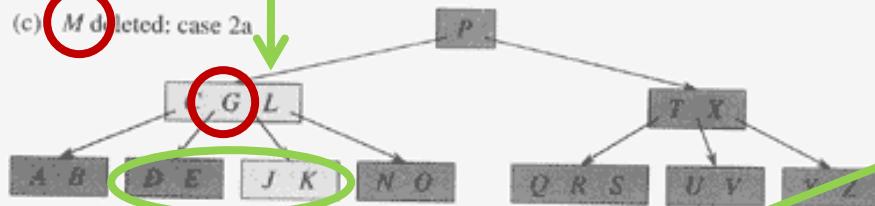
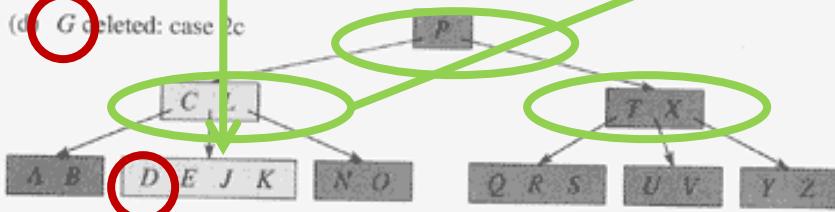
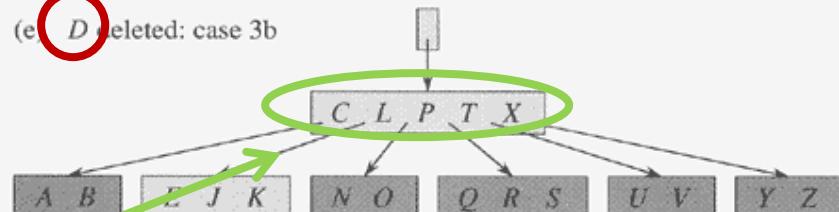
- c. Both **a** and **b** are not satisfied:  $y$  and  $z$  have  $t-1$  keys
- Merge the two children,  $y$  and  $z$
  - Recursively delete **k** from the merged cell



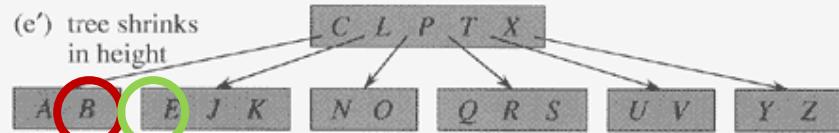
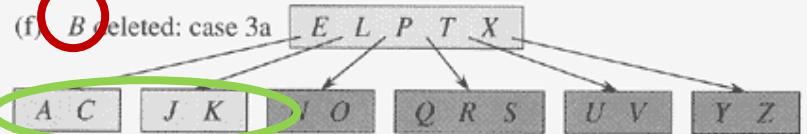
# Questions

- When does the height of the tree shrink?
- Why do we need the number of keys to be at least  $t$  and not  $t-1$  when we proceed down in the tree?

(a) initial tree

(b) *F* deleted: case 1(c) *M* deleted: case 2a(d) *G* deleted: case 2c(e) *D* deleted: case 3b

(e') tree shrinks in height

(f) *B* deleted: case 3a

**Figure 18.8** Deleting keys from a B-tree. The minimum degree for this B-tree is  $t = 3$ , so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded.

(a) The B-tree of Figure 18.7(e). (b) Deletion of *F*. This is case 1: simple deletion from a leaf. (c) Deletion of *M*. This is case 2a: the predecessor *L* of *M* is moved up to take *M*'s position. (d) Deletion of *G*. This is case 2c: *G* is pushed down to make node *DEGJK*, and then *G* is deleted from this leaf (case 1). (e) Deletion of *D*. This is case 3b: the recursion can't descend to node *CL* because it has only 2 keys, so *P* is pushed down and merged with *CL* and *TX* to form *CLPTX*; then, *D* is deleted from a leaf (case 1). (e') After (d), the root is deleted and the tree shrinks in height by one. (f) Deletion of *B*. This is case 3a: *C* is moved to fill *B*'s position and *E* is moved to fill *C*'s position.

# Delete Complexity

- Basically downward pass:
  - Most of the keys are in the leaves – one downward pass
  - When deleting a key in internal node – may have to go one step up to replace the key with its predecessor or successor

**Complexity**

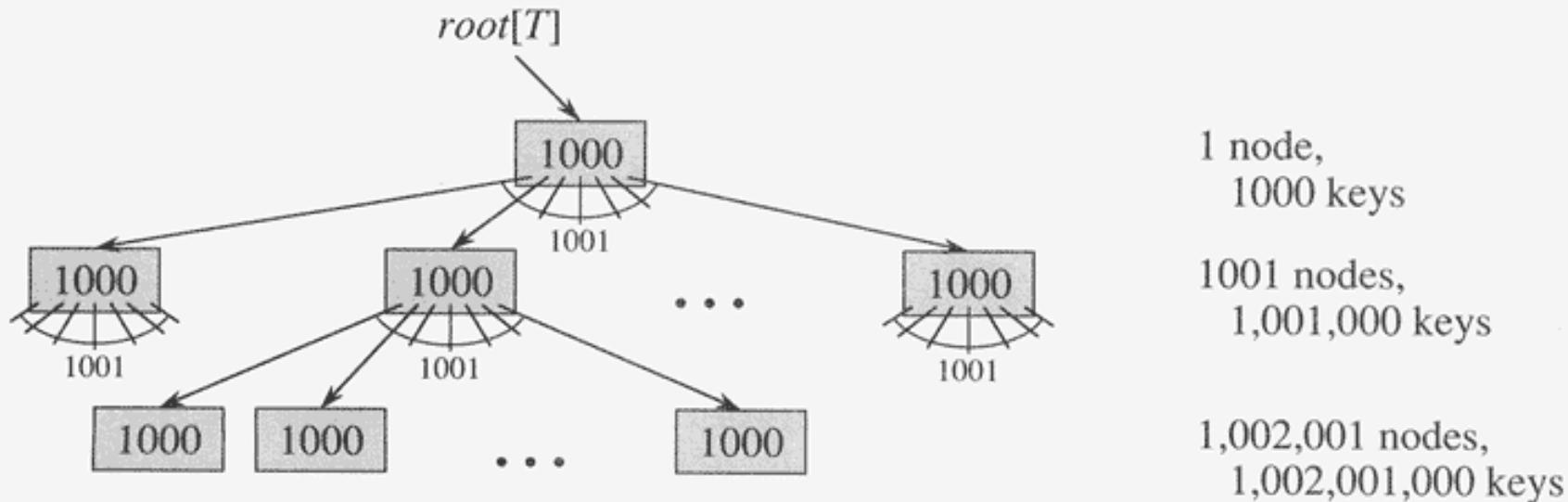
$$O(h) = O(\log_t n)$$

# Run Time Analysis of B-Tree Operations

- For a B-Tree of order  $M=2t$ 
  - #keys in internal node:  $M-1$
  - #children of internal node: between  $M/2$  and  $M$
  - → Depth of B-Tree storing  $n$  items is  $O(\log_{M/2} N)$
- Find run time is:
  - $O(\log M)$  to binary search which branch to take at each node, since  $M$  is constant it is  $O(1)$ .
  - Total time to find an item is  $O(h * \log M) = O(\log n)$
- Insert & Delete
  - Similar to find but update a node may take :  $O(M)=O(1)$

Note: if  $M$  is  $>32$  it worth using binary search at each node

# A typical B-Tree



**Figure 18.3** A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node  $x$  is  $n[x]$ , the number of keys in  $x$ .

# Why B-Tree?

- B-trees is an implementation of dynamic sets that is optimized for disks
  - The memory has an hierarchy and there is a tradeoff between size of units/blocks and access time
  - The goal is to optimize the number of times needed to access an “expensive access time memory”
  - The size of a node is determined by characteristics of the disk – block size – page size
  - The number of access is proportional to the tree depth

# B+ Trees

---

- What are B+ Trees used for
- What is a B Tree
- What is a B+ Tree
- Searching
- Insertion
- Deletion

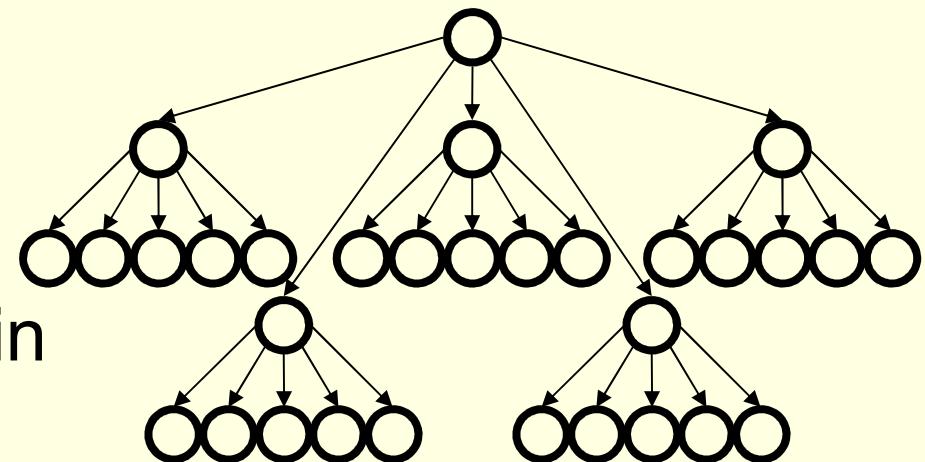
# What are B+ Trees Used For?

---

- When we store data in a table in a DBMS we want
  - Fast lookup by primary key
    - Just this – hashtable  $O(c)$
  - Ability to add/remove records on the fly
    - Some kind of dynamic tree **on disk**
  - Sequential access to records (physically sorted by primary key **on disk**)
    - Tree structured keys (hierarchical index for searching)
    - Records all at leaves in sorted order

# What is an M-ary Search Tree?

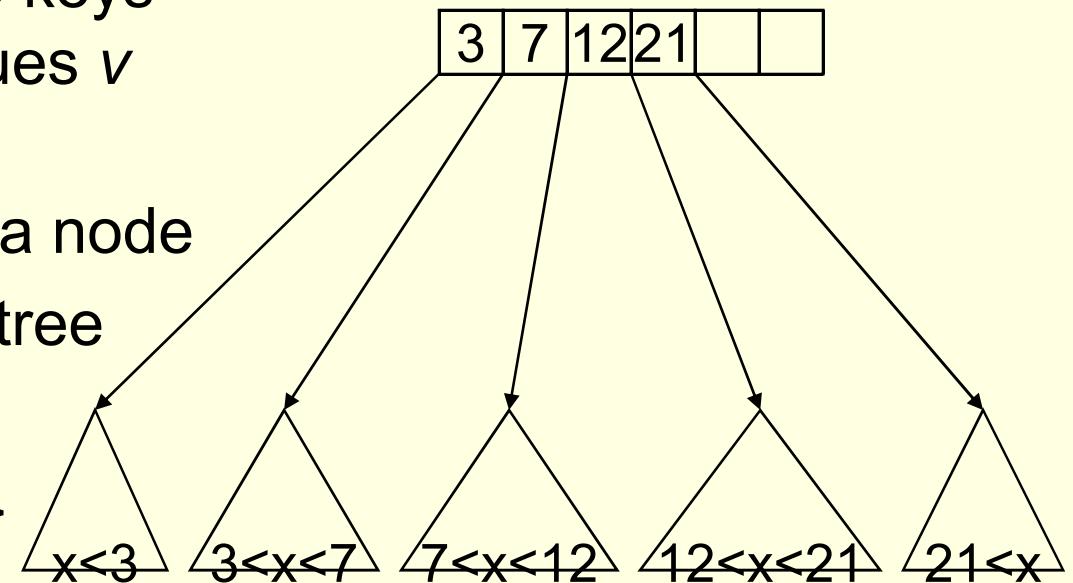
- Maximum branching factor of  $M$
- Complete tree has depth =  $\log_M N$
- Each internal node in a complete tree has  $M - 1$  keys



Binary search tree is a B Tree where M is 2

# B Trees

- B-Trees are specialized  $M$ -ary search trees
- Each node has many keys
  - subtree between two keys  $x$  and  $y$  contains values  $v$  such that  $x \leq v < y$
  - binary search within a node
  - to find correct subtree
- Each node takes one full  $\{page, block, line\}$  of memory (disk)



# B-Tree Properties

---

- **Properties**

- maximum branching factor of  $M$
- the root has between 2 and  $M$  children or at most  $M-1$  keys
- All other nodes have between  $\lceil M/2 \rceil$  and  $M$  records
  - Keys+data

- **Result**

- tree is  $O(\log M)$  deep
- all operations run in  $O(\log M)$  time
- operations pull in about  $M$  items at a time

# What is a B+ Tree?

---

- A variation of B trees in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth

# Definition of a B+Tree

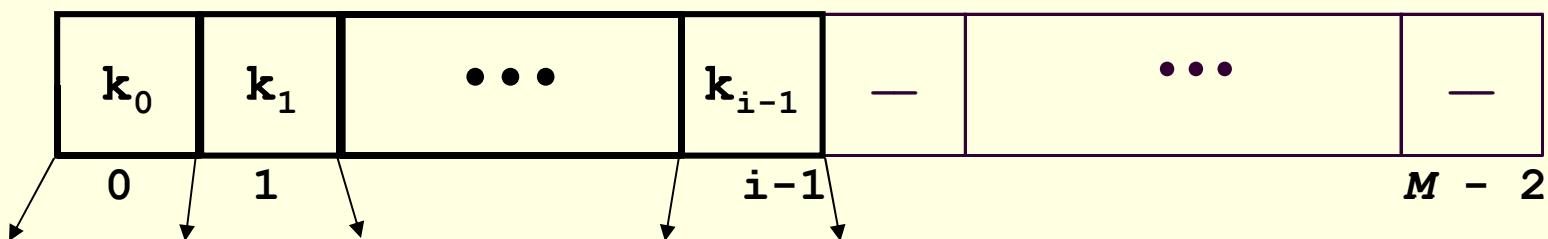
---

A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between  $[M/2]$  and  $[M]$  children, where  $n$  is fixed for a particular tree.

# B+ Tree Nodes

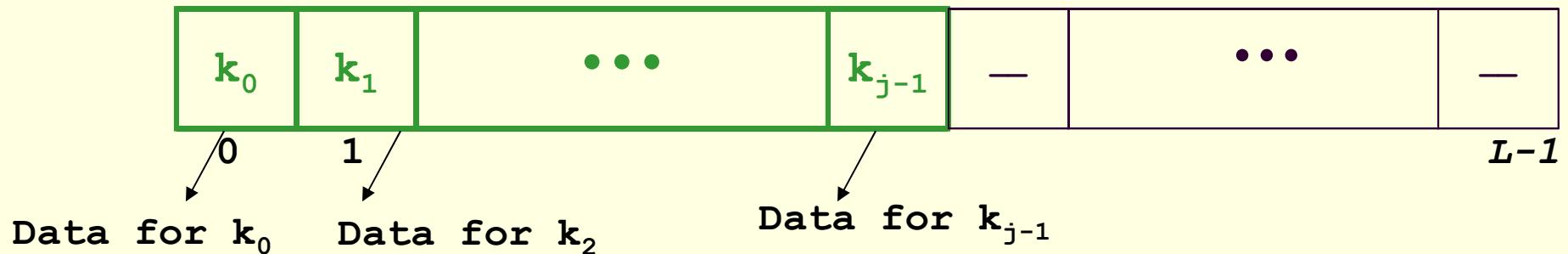
## ■ Internal node

- Pointer (Key, NodePointer)\*M-1 in each node
- First i keys are currently in use



## • Leaf

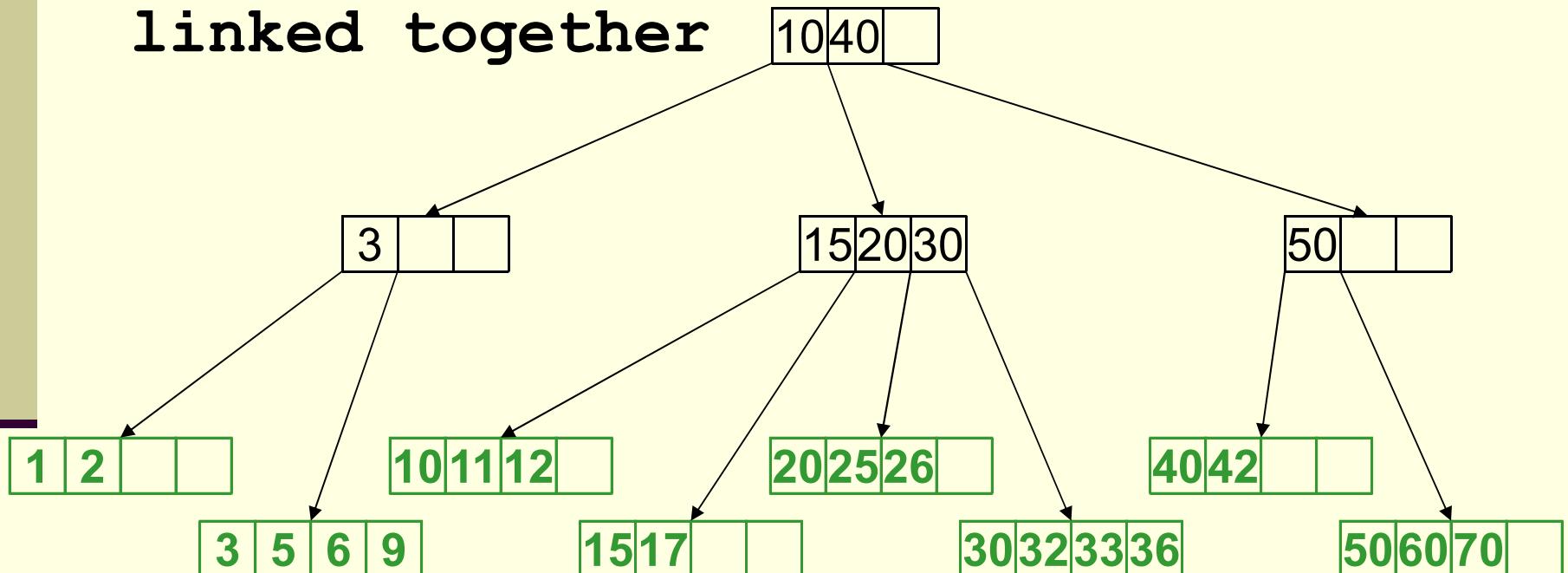
- (Key, DataPointer)\* L in each node
- first j Keys currently in use



# Example

B+ Tree with  $M = 4$

Often, leaf nodes  
linked together



# Advantages of B+ tree usage for databases

---

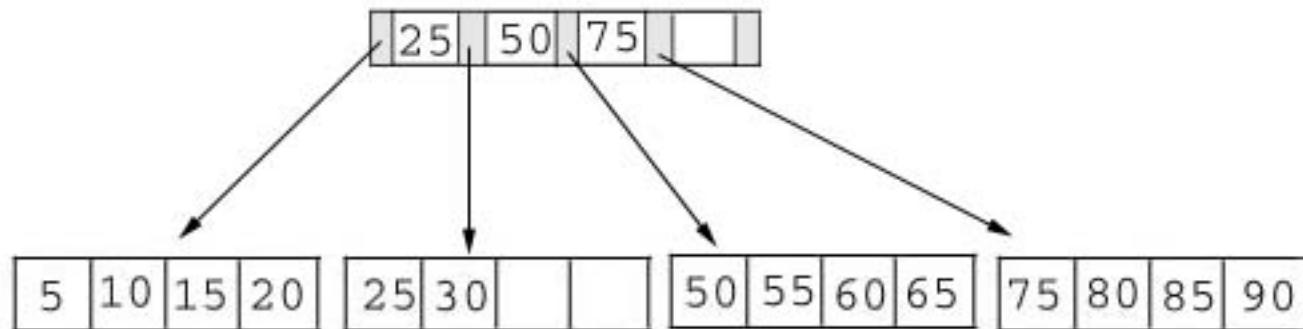
- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm
- In addition, a B+ tree minimizes waste by making sure the interior nodes are at least half full. A B+ tree can handle an arbitrary number of insertions and deletions.

# Searching

---

- Just compare the key value with the data in the tree, then return the result.

For example: find the value **45**, and **15** in below tree.



# Searching

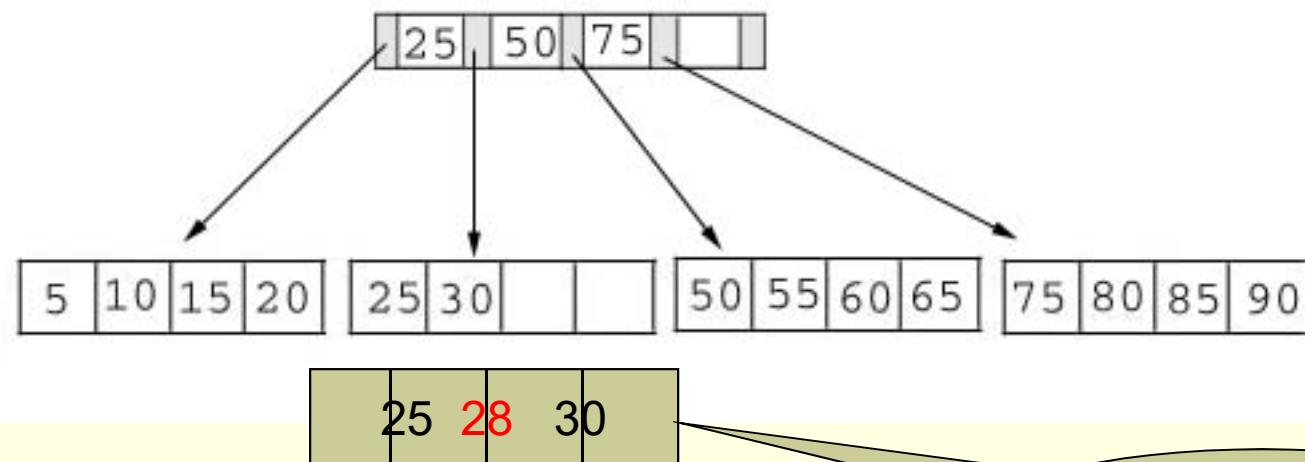
---

- Result:

1. For the value of 45, not found.
2. For the value of 15, return the position where the pointer located.

# Insertion

- inserting a value into a B+ tree may unbalance the tree, so rearrange the tree if needed.
- Example #1: insert **28** into the below tree.

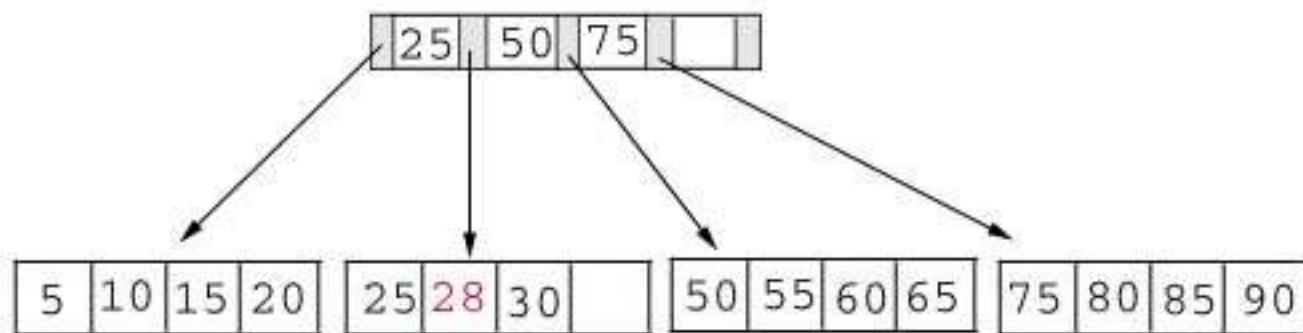


Fits inside the leaf

# Insertion

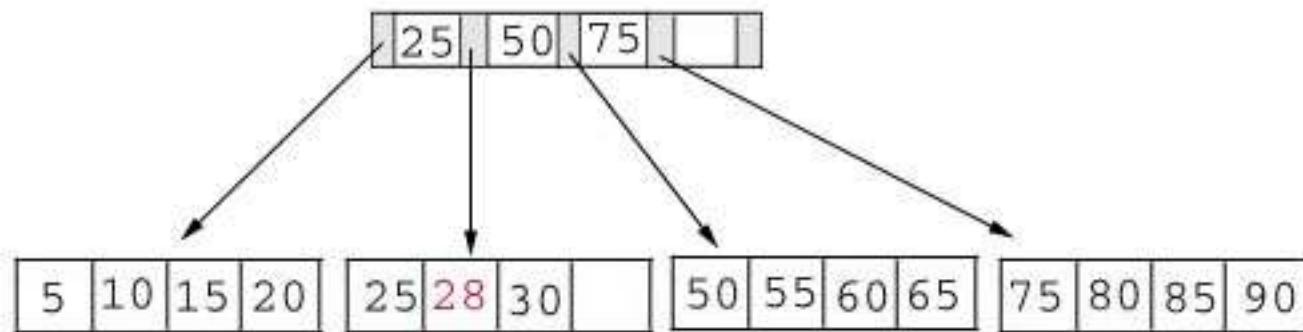
---

## Result:



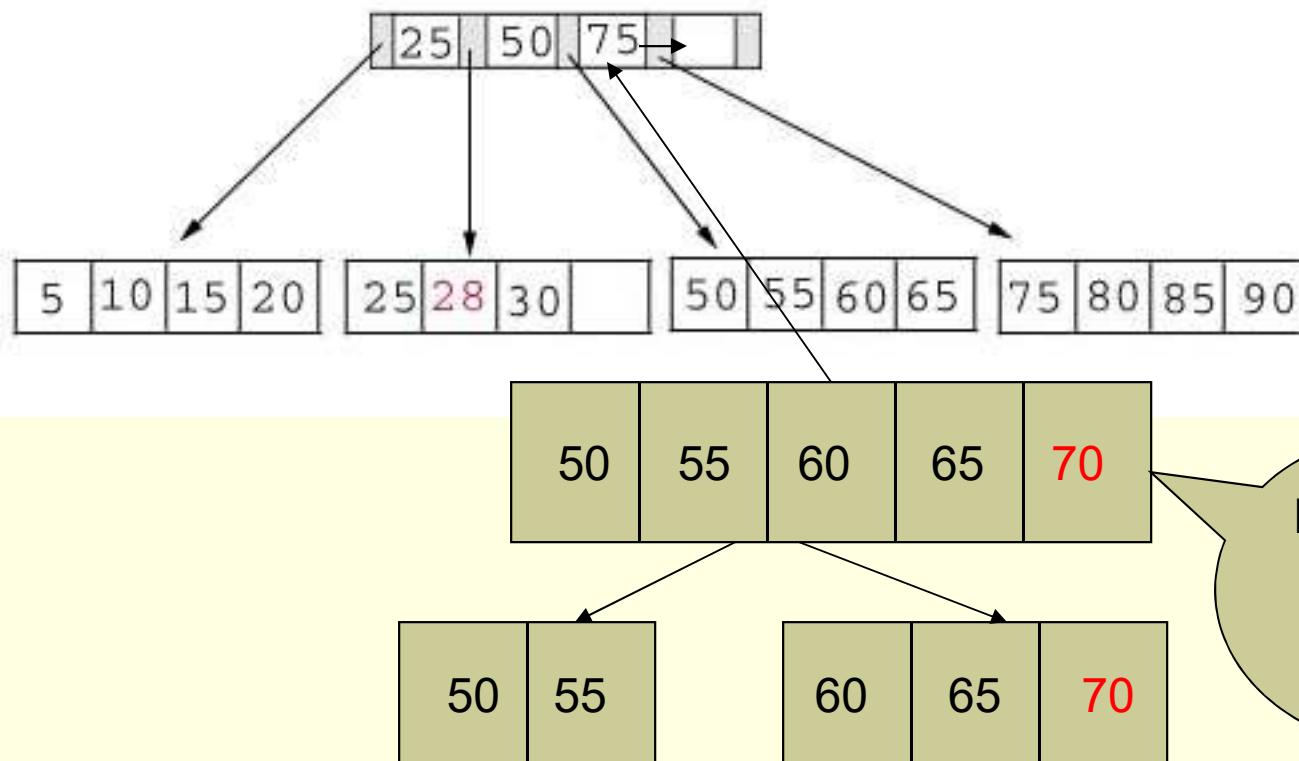
# Insertion

- Example #2: insert **70** into below tree



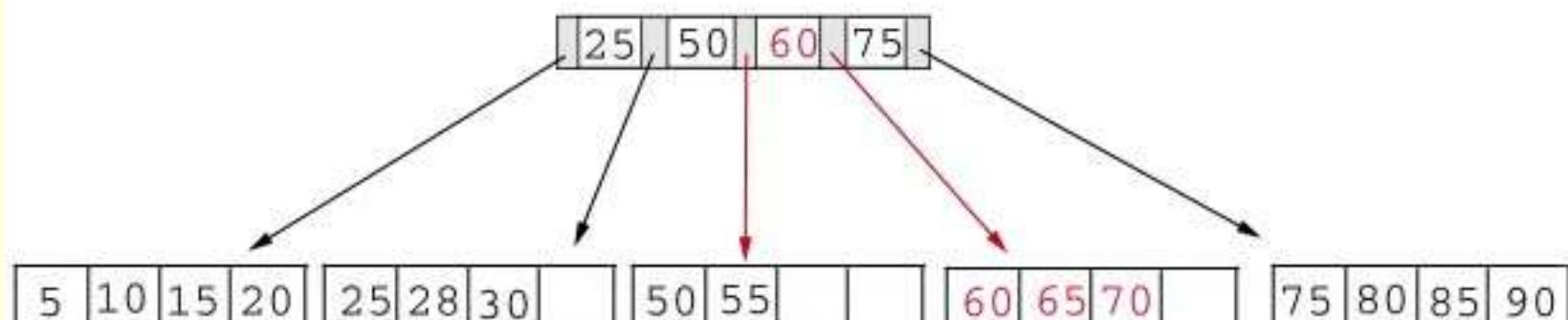
# Insertion

- Process: split the leaf and propagate middle key up the tree



# Insertion

- Result: chose the middle key 60, and place it in the index page between 50 and 75.



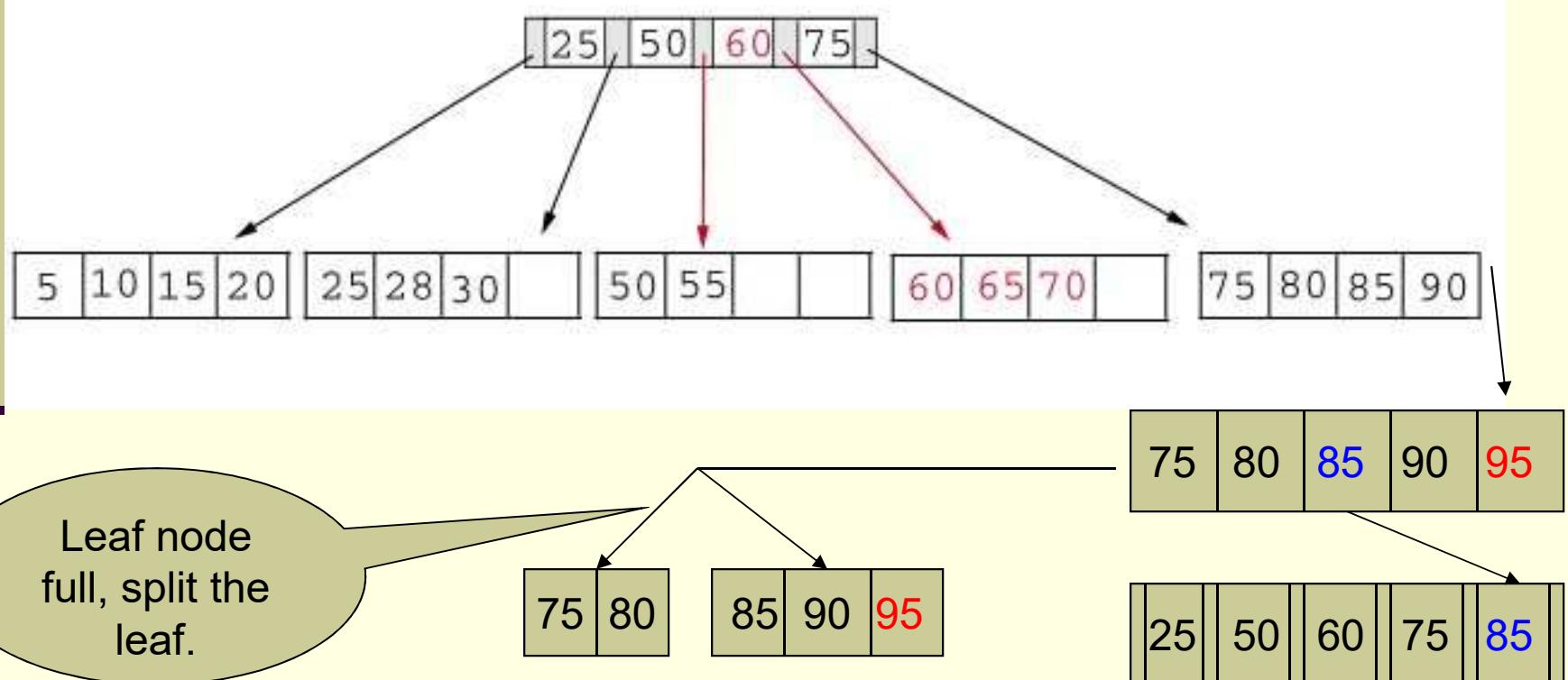
# Insertion

## The insert algorithm for B+ Tree

Leaf Node Full	Index Node Full	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	<ol style="list-style-type: none"><li>1. Split the leaf node</li><li>2. Place Middle Key in the index node in sorted order.</li><li>3. Left leaf node contains records with keys below the middle key.</li><li>4. Right leaf node contains records with keys equal to or greater than the middle key.</li></ol>
YES	YES	<ol style="list-style-type: none"><li>1. Split the leaf node.</li><li>2. Records with keys &lt; middle key go to the left leaf node.</li><li>3. Records with keys <math>\geq</math> middle key go to the right leaf node.</li><li>4. Split the index node.</li><li>5. Keys &lt; middle key go to the left index node.</li><li>6. Keys <math>&gt;</math> middle key go to the right index node.</li><li>The middle key goes to the next (higher level) index node.</li></ol> <p>IF the next level index node is full, continue splitting the index nodes.</p>

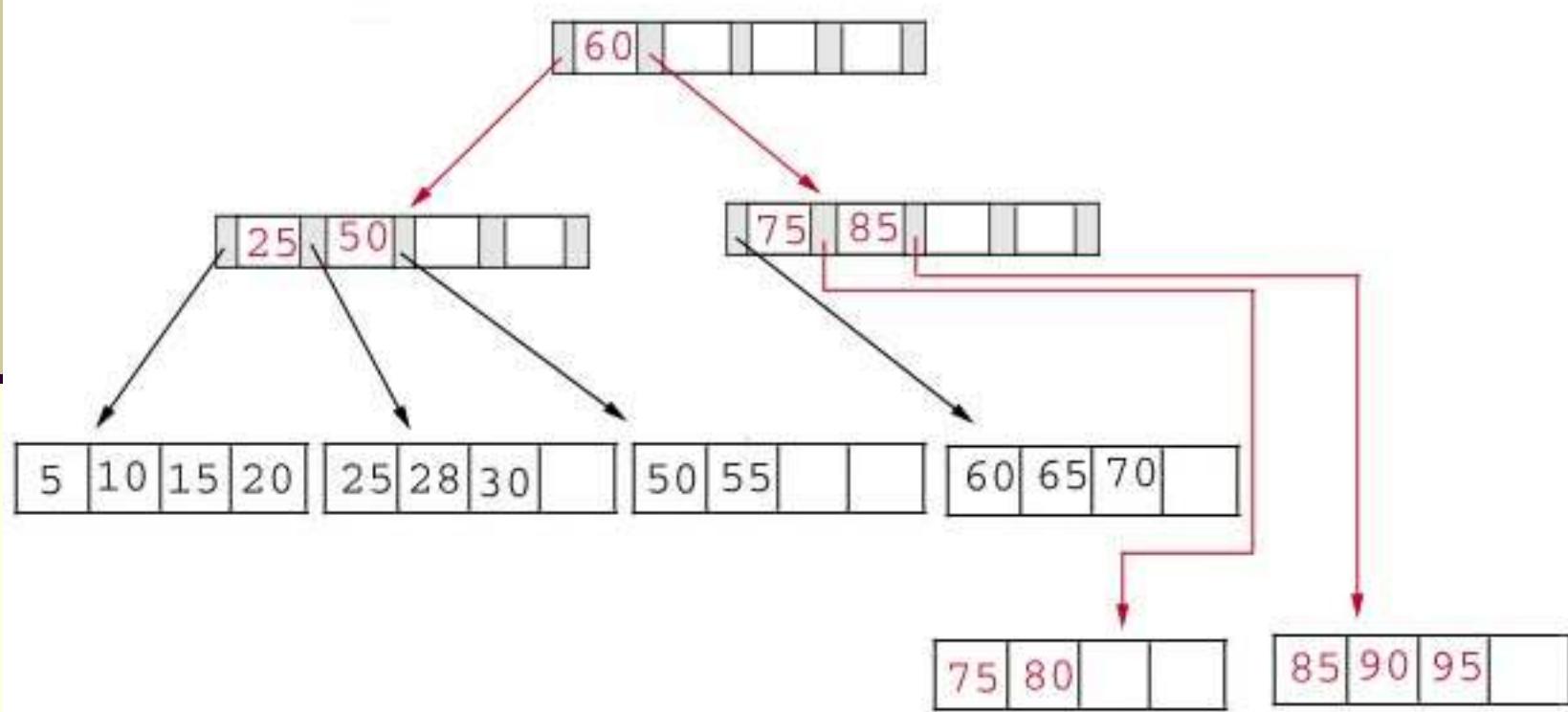
# Insertion

- Exercise: add a key value **95** to the below tree.



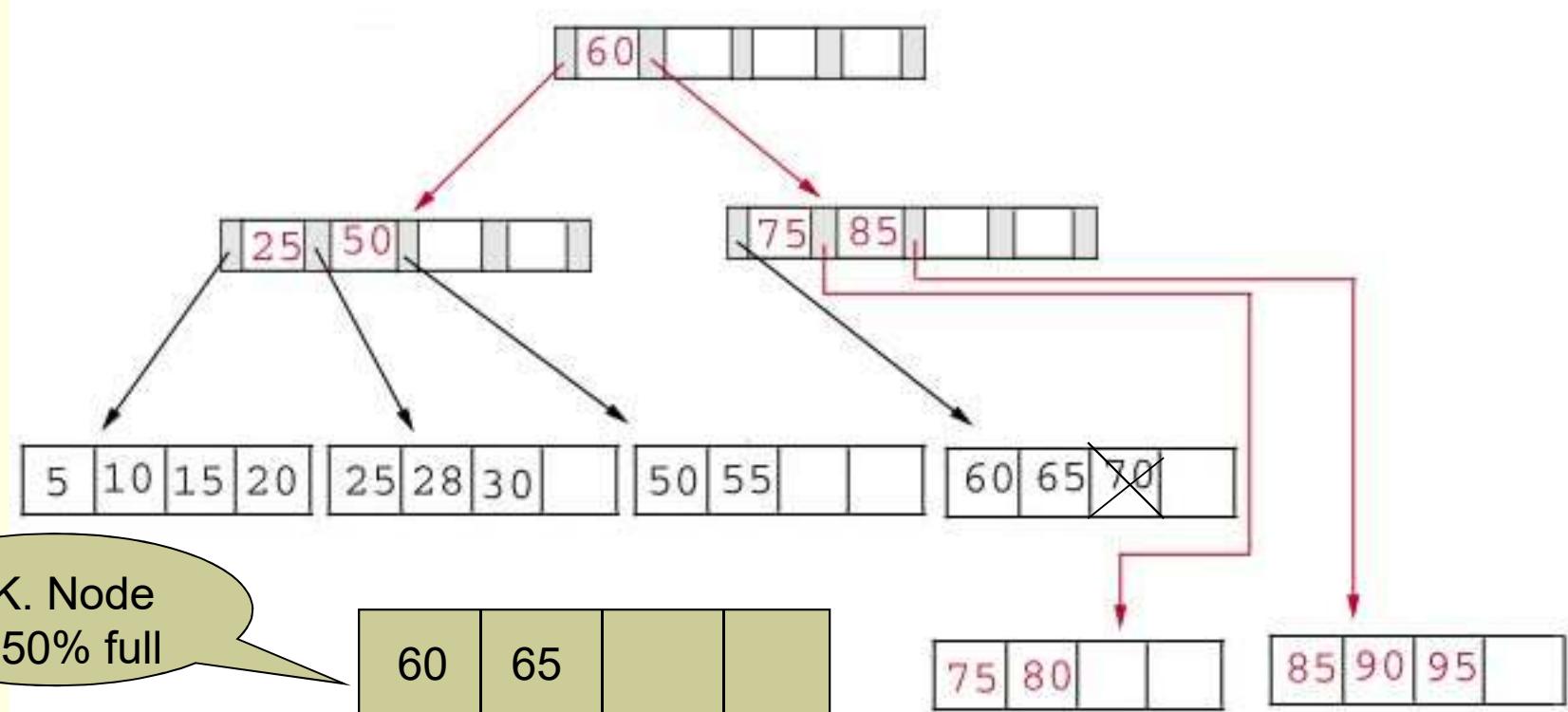
# Insertion

- Result: again put the middle key 60 to the index page and rearrange the tree.



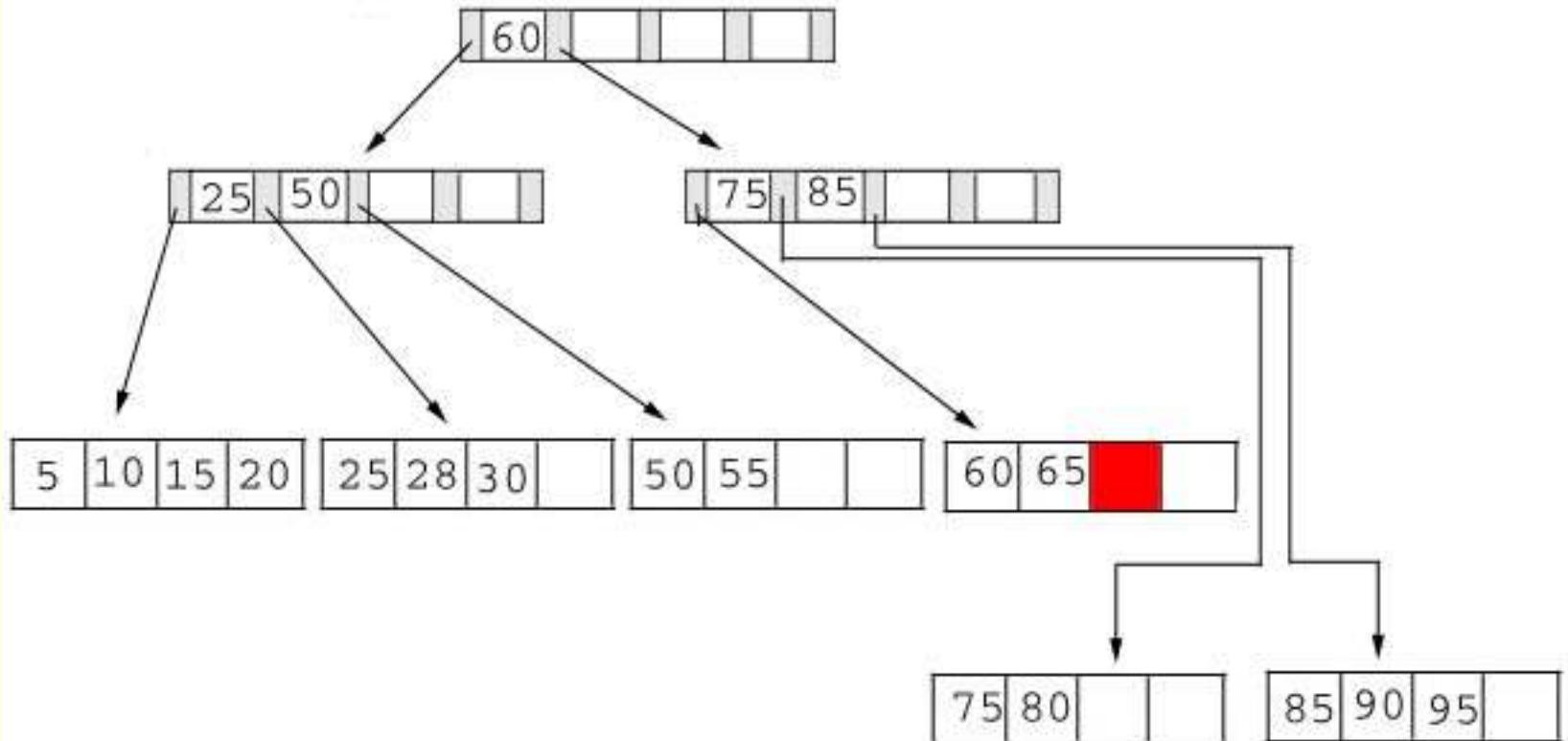
# Deletion

- Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.
- Example #1: delete **70** from the tree



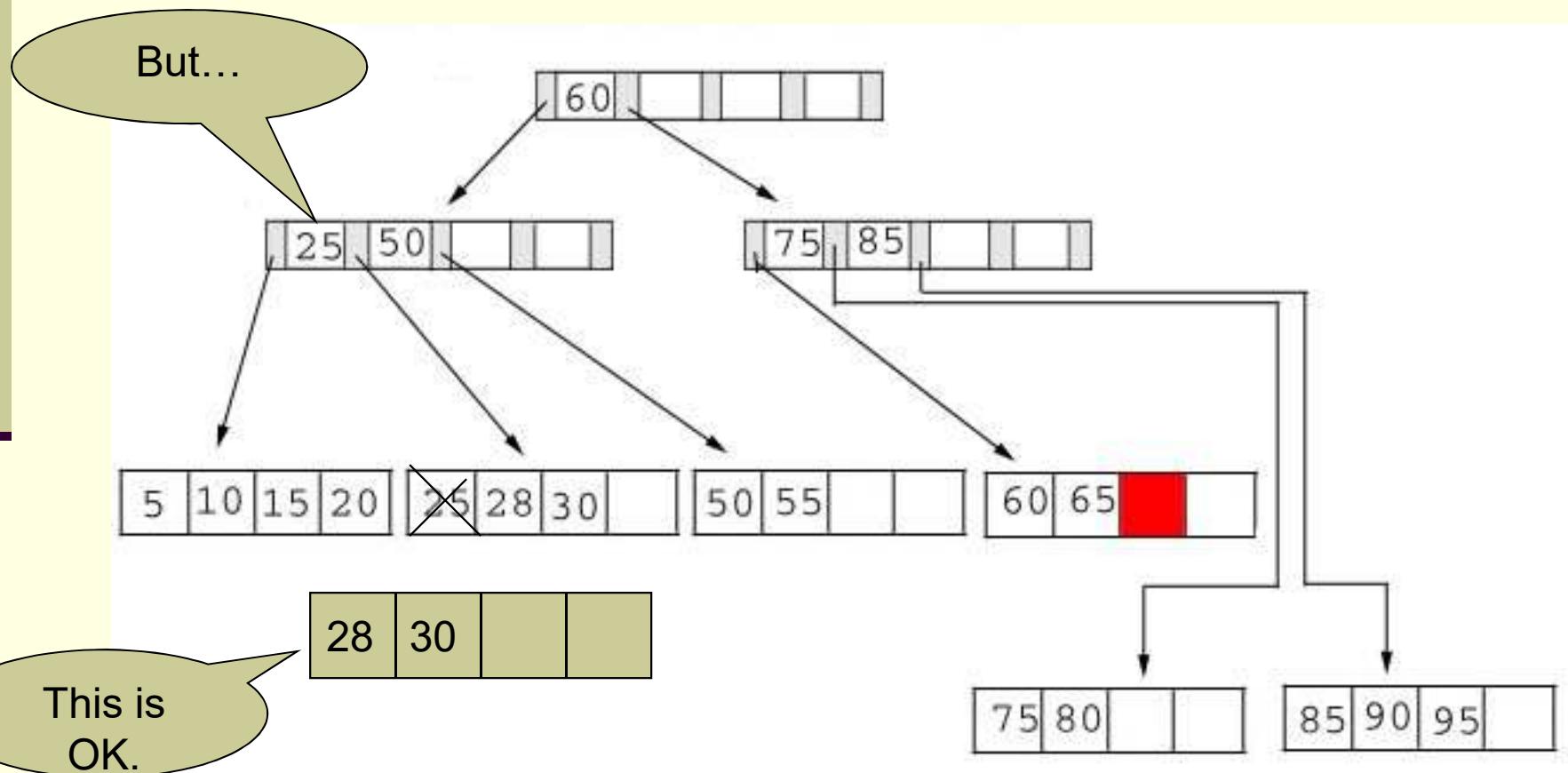
# Deletion

- Result:



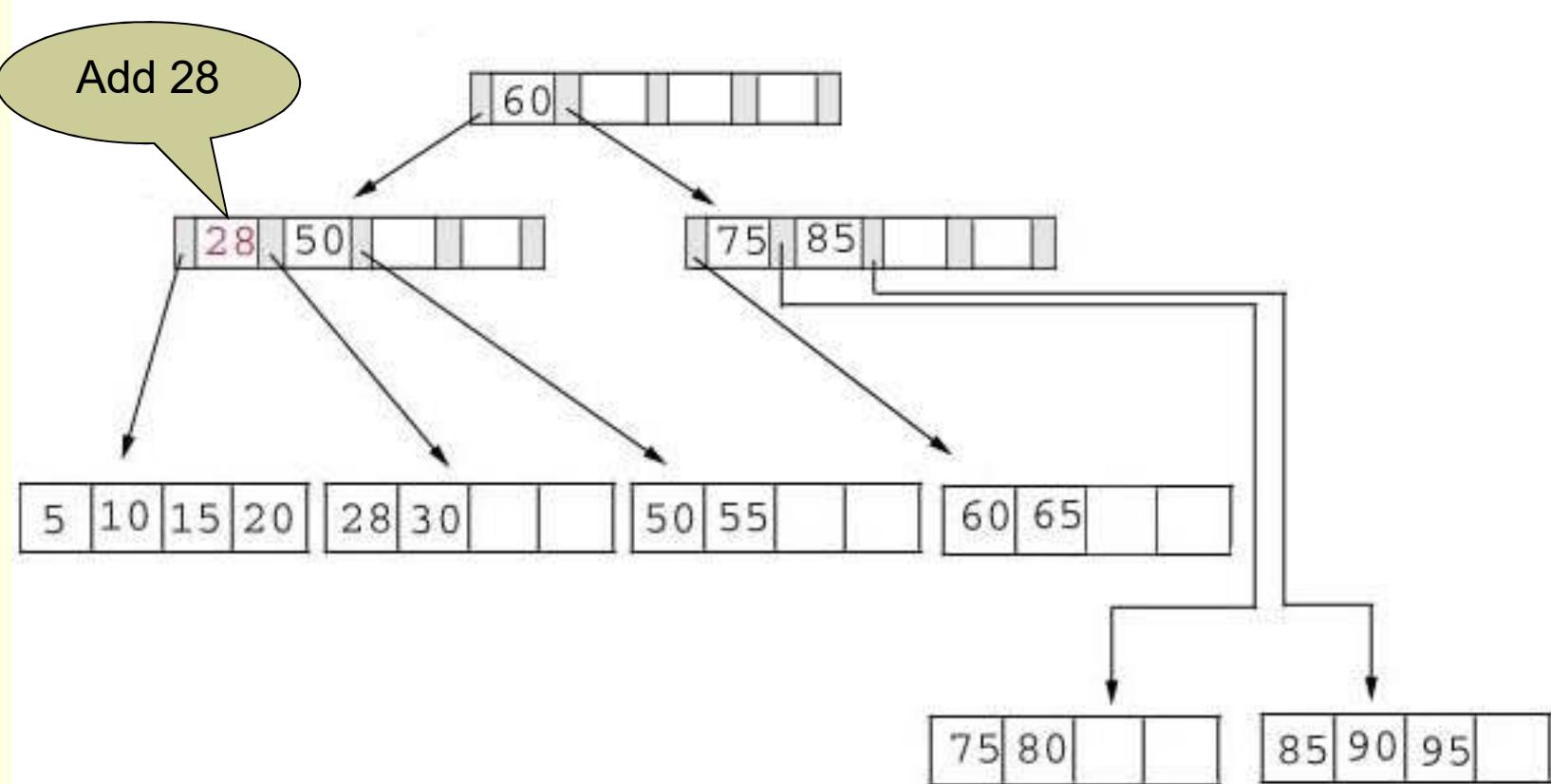
# Deletion

Example #2: delete **25** from below tree, but **25** appears in the index page.



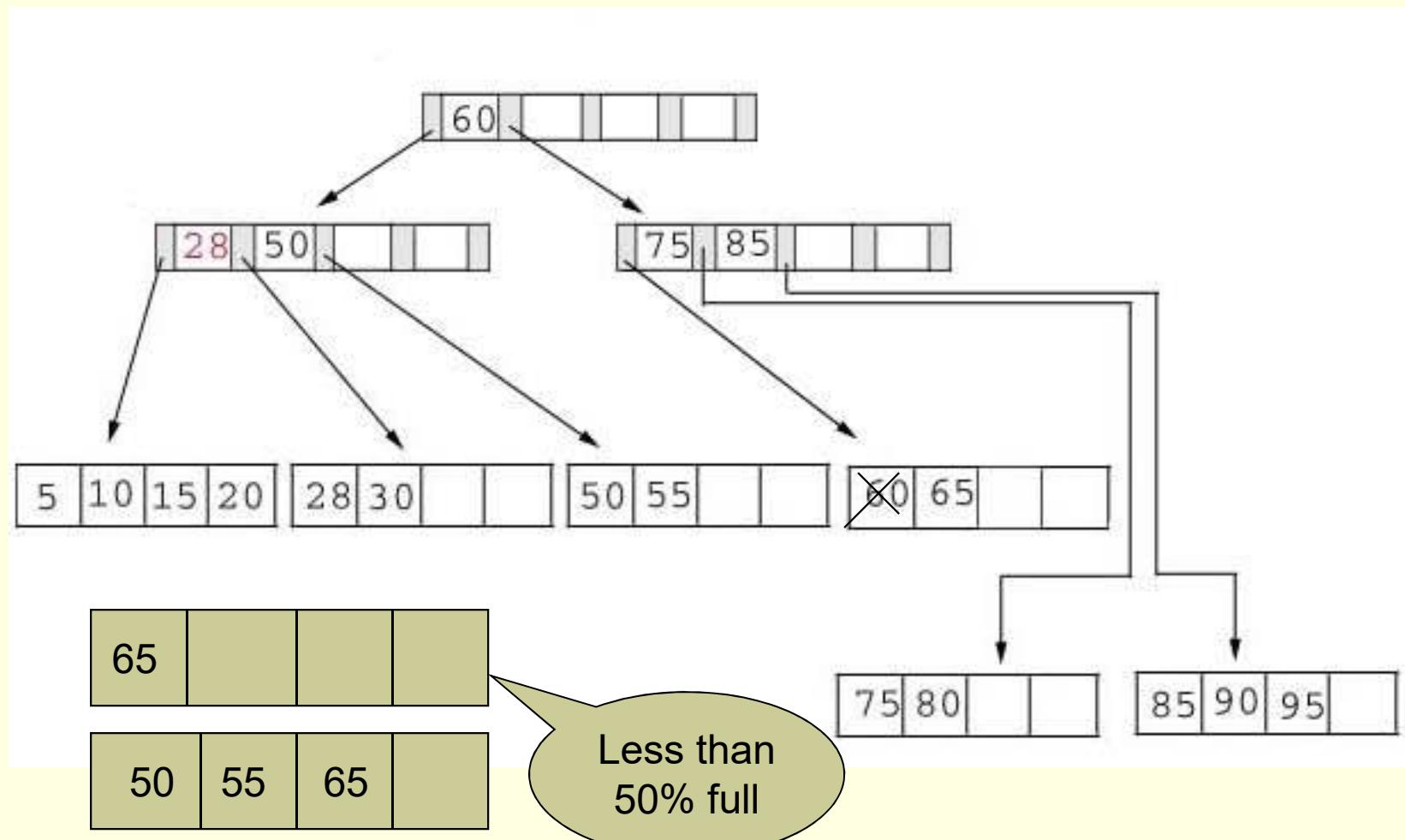
# Deletion

- Result: replace 28 in the index page.



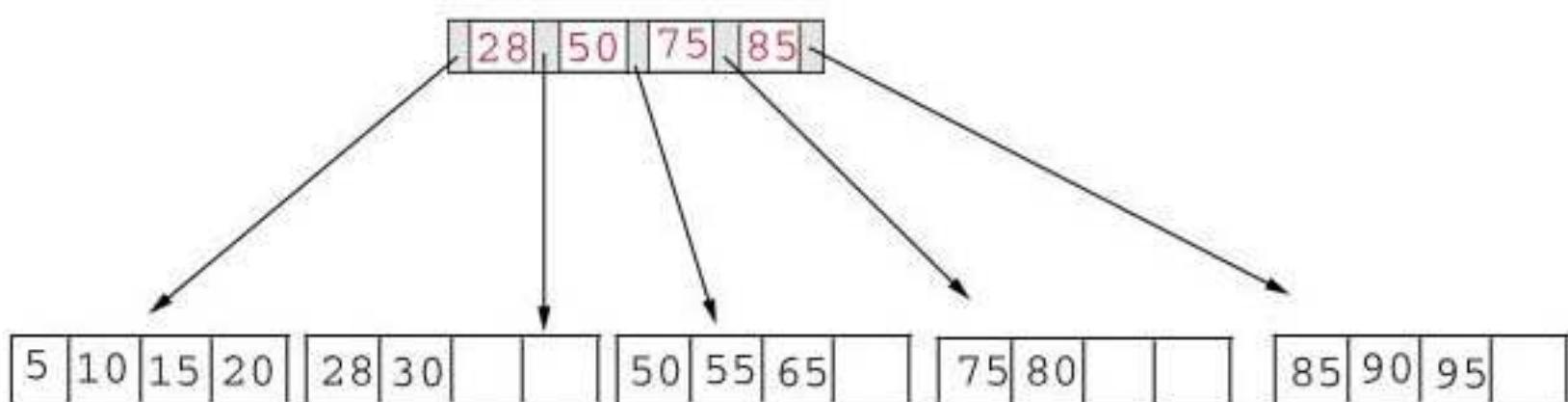
# Deletion

- Example #3: delete **60** from the below tree



# Deletion

- Result: delete 60 from the index page and combine the rest of index pages.



# Deletion

## Delete algorithm for B+ trees

Data Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	<ol style="list-style-type: none"><li>1. Combine the leaf page and its sibling.</li><li>2. Adjust the index page to reflect the change.</li><li>3. Combine the index page with its sibling.</li></ol> <p>Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.</p>

# Conclusion

---

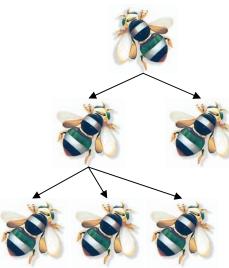
- For a B+ Tree:
- It is “easy” to maintain its balance
  - Insert/Deletion complexity  $O(\log_{M/2})$
- The searching time is shorter than most of other types of trees because branching factor is high

# B+Trees and DBMS

---

- Used to index primary keys
- Can access records in  $O(\log_{M/2})$  traversals (height of the tree)
- Interior nodes contain Keys only
  - Set node sizes so that the  $M-1$  keys and  $M$  pointers fits inside a single block on disk
    - E.g., block size 4096B, keys 10B, pointers 8 bytes
    - $(8 + (10+8)*M-1) = 4096$
    - $M = 228$ ; 2.7 billion nodes in 4 levels
  - One block read per node visited

## B-Trees and B+ Trees



2

## Traversing very large datasets

Suppose we had very many pieces of data (as in a database), e.g.,  $n = 2^{30} \approx 10^9$ .

How many (worst case) hops through the tree to find a node?

- BST
- AVL
- Splay

3

## B-Trees and B+ Trees

## Memory considerations

What is in a tree node? In an object?

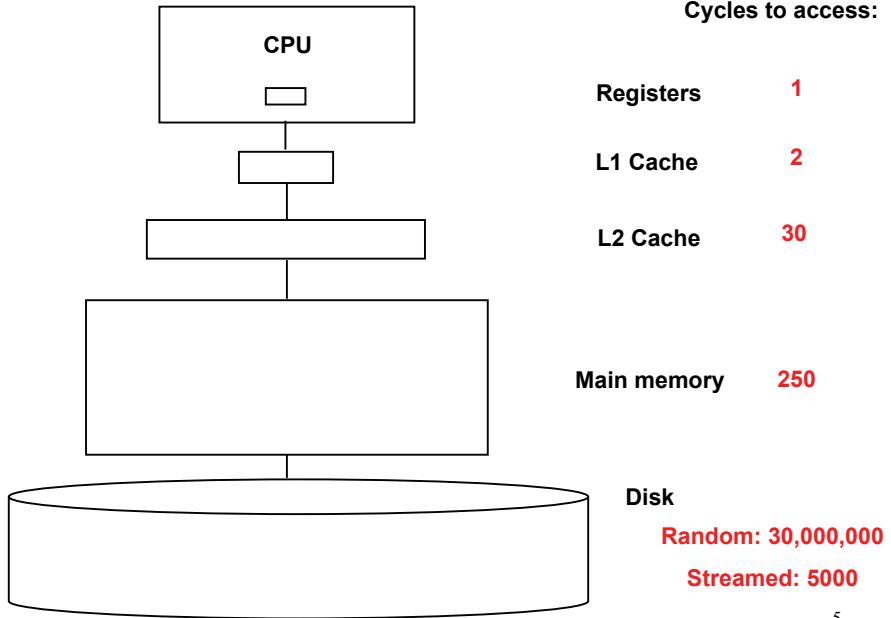
**Node:**  
Object obj;  
Node left;  
Node right;  
Node parent;

**Object:**  
Key key;  
...data...

Suppose the data is 1KB.

How much space does the tree take?  
How much of the data can live in 1GB of RAM?

4



## Minimizing random disk access

In our example, almost all of our data structure is on disk.

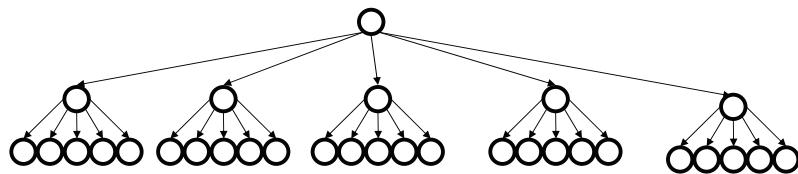
Thus, hopping through a tree amounts to random accesses to disk. Ouch!

How can we address this problem?

6

## *M*-ary Search Tree

Suppose, *somewhat*, we devised a search tree with maximum branching factor  $M$ :



Complete tree has height:

# hops for *find*:

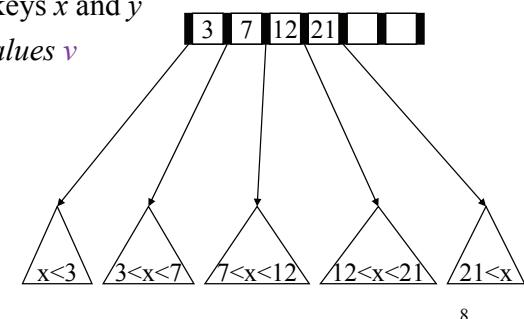
Runtime of *find*:

7

## B-Trees

How do we make an  $M$ -ary search tree work?

- Each **node** has (up to)  $M-1$  keys.
- Order property:
  - subtree between two keys  $x$  and  $y$  contain leaves with values  $v$  such that  $x < v < y$



8

# B-Tree Structure Properties

## Root (special case)

- has between 2 and  $M$  children (or root could be a leaf)

## Internal nodes

- store up to  $M-1$  keys
- have between  $\lceil M/2 \rceil$  and  $M$  children

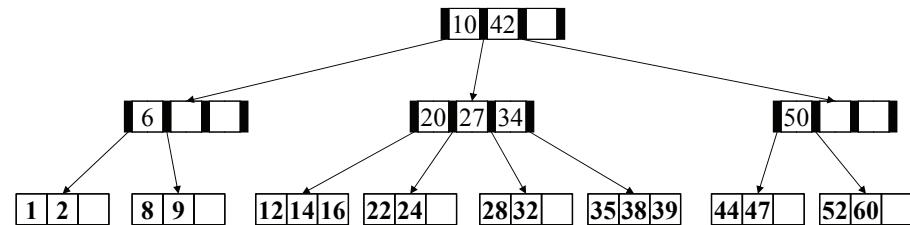
## Leaf nodes

- store between  $\lceil (M-1)/2 \rceil$  and  $M-1$  sorted keys
- all at the same depth

9

# B-Tree: Example

## B-Tree with $M = 4$

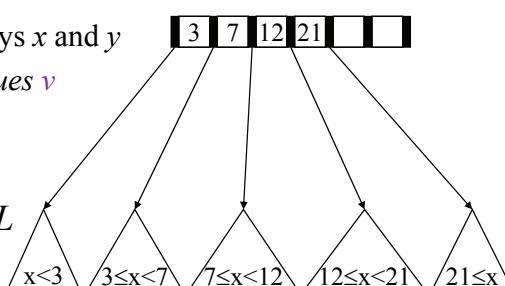


10

# B+ Trees

In a B+ tree, the internal nodes have no data – only the leaves do!

- Each internal node still has (up to)  $M-1$  keys:
- Order property:
  - subtree between two keys  $x$  and  $y$  contain leaves with values  $v$  such that  $x \leq v < y$
  - Note the “ $\leq$ ”
- Leaf nodes have up to  $L$  sorted keys.



11

# B+ Tree Structure Properties

## Root (special case)

- has between 2 and  $M$  children (or root could be a leaf)

## Internal nodes

- store up to  $M-1$  keys
- have between  $\lceil M/2 \rceil$  and  $M$  children

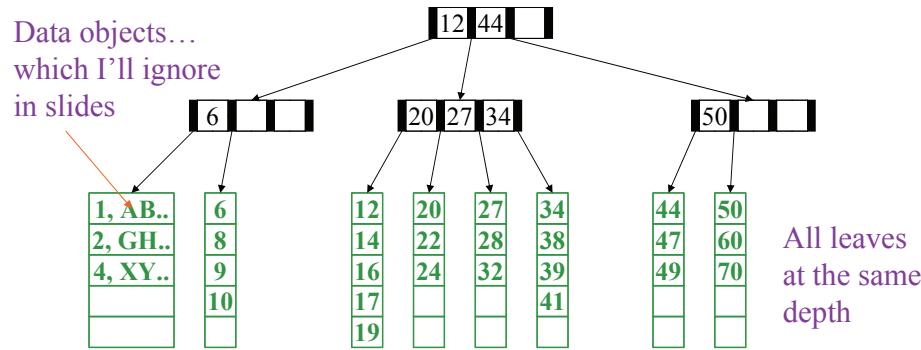
## Leaf nodes

- where data is stored
- all at the same depth
- contain between  $\lceil L/2 \rceil$  and  $L$  data items

12

## B+ Tree: Example

B+ Tree with  $M = 4$  (# pointers in internal node)  
and  $L = 5$  (# data items in leaf)



Definition for later: “neighbor” is the next sibling to the left or right.<sup>13</sup>

## B+ trees vs. AVL trees

Suppose again we have  $n = 2^{30} \approx 10^9$  items:

- Depth of AVL Tree
- Depth of B+ Tree with  $M = 256$ ,  $L = 256$

Great, but how do we actually make a B+ tree and keep it balanced...?

## Disk Friendliness

What makes B+ trees disk-friendly?

### 1. Many keys stored in a node

- All brought to memory/cache in one disk access.

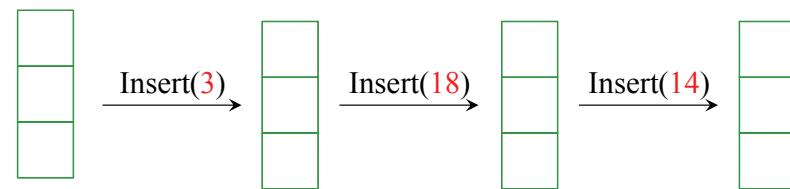
### 2. Internal nodes contain *only* keys;

**Only leaf nodes contain keys and actual data**

- Much of tree structure can be loaded into memory irrespective of data object size
- Data actually resides in disk

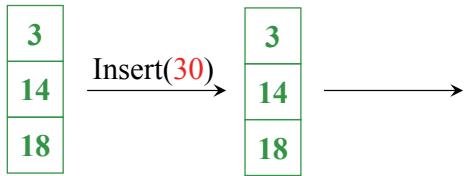
14

## Building a B+ Tree with Insertions

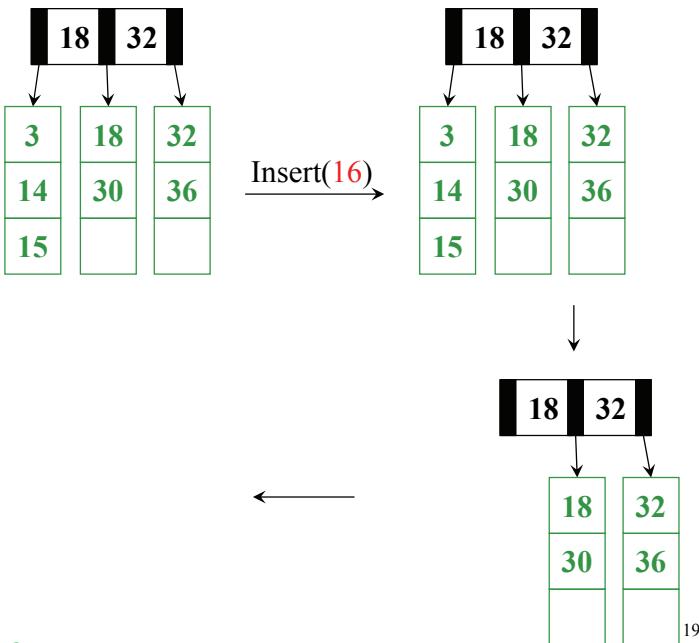


15

16

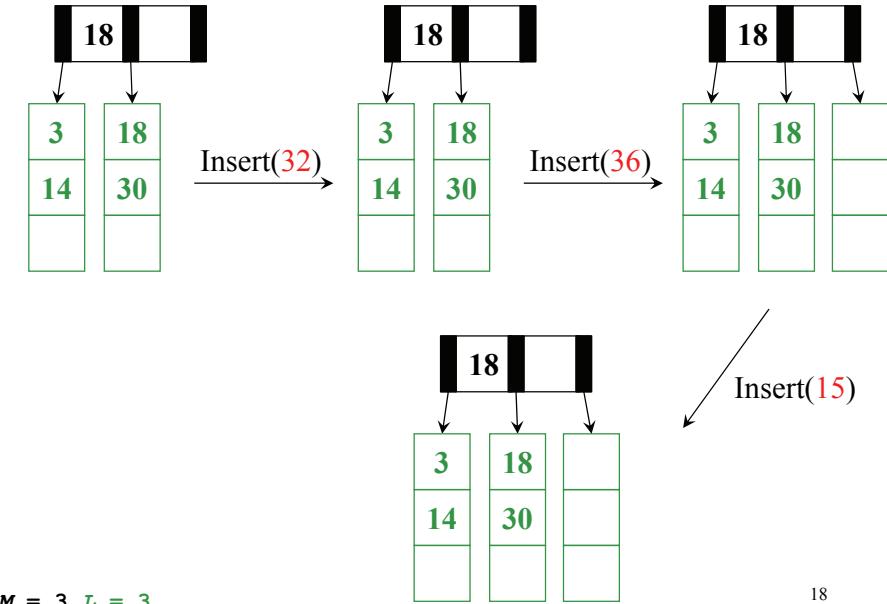


**M = 3 L = 3**



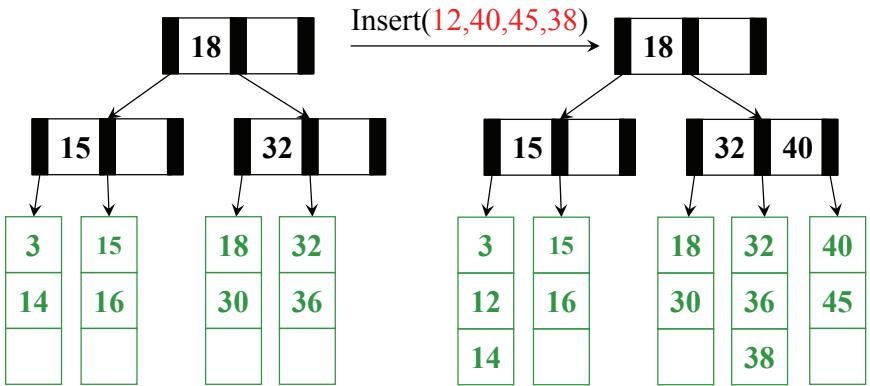
**M = 3 L = 3**

17



**M = 3 L = 3**

18



20

19

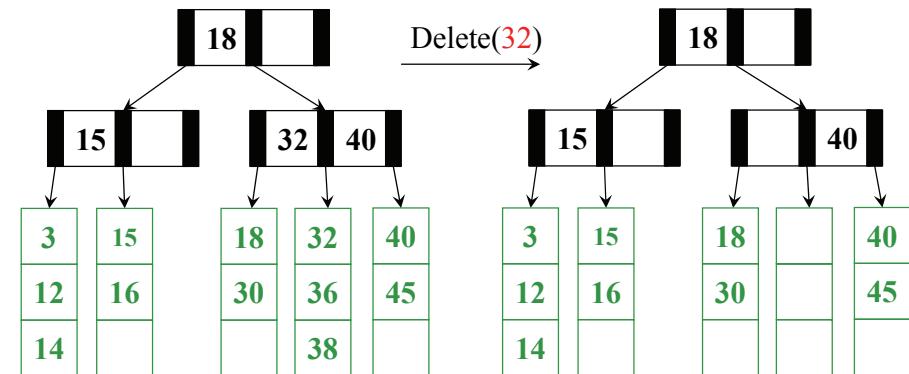
## Insertion Algorithm

1. Insert the key in its leaf in sorted order
2. If the leaf ends up with  $L+1$  items, **overflow!**
  - Split the leaf into two nodes:
    - original with  $\lceil (L+1)/2 \rceil$  items
    - new one with  $\lfloor (L+1)/2 \rfloor$  items
  - Add the new child to the parent
  - If the parent ends up with  $M+1$  children, **overflow!**
3. If an internal node ends up with  $M+1$  children, **overflow!**
  - Split the node into two nodes:
    - original with  $\lceil (M+1)/2 \rceil$  children
    - new one with  $\lfloor (M+1)/2 \rfloor$  children
  - Add the new child to the parent
  - If the parent ends up with  $M+1$  items, **overflow!**
4. Split an overflowed root in two and hang the new nodes under a new root
5. Propagate keys up tree.

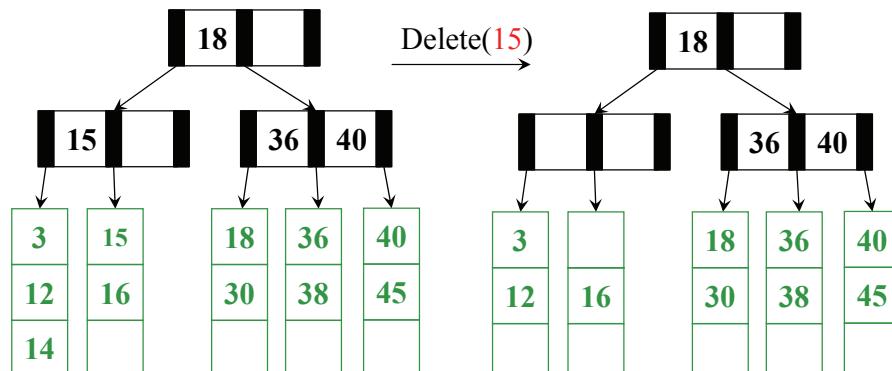
This makes the tree deeper!

21

## And Now for Deletion...

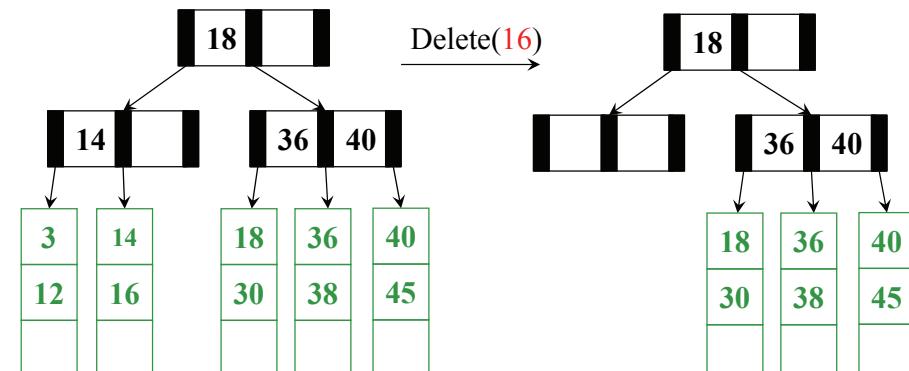


22



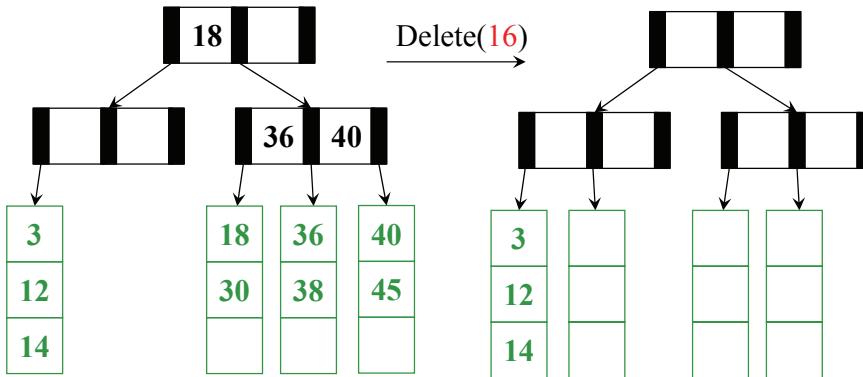
$M = 3$   $L = 3$

23



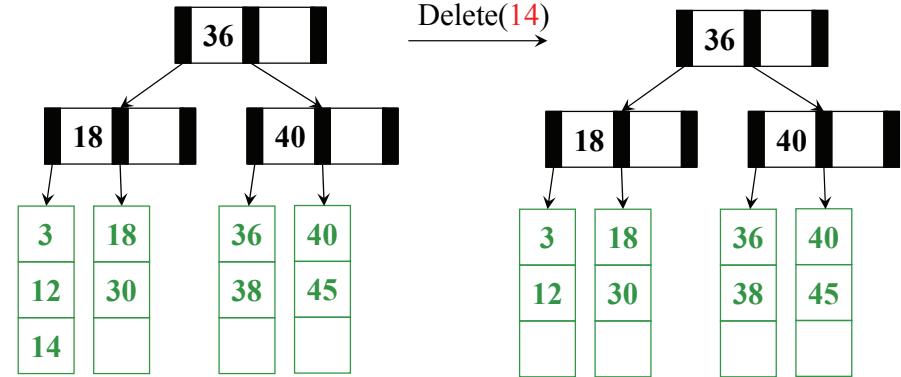
$M = 3$   $L = 3$

24



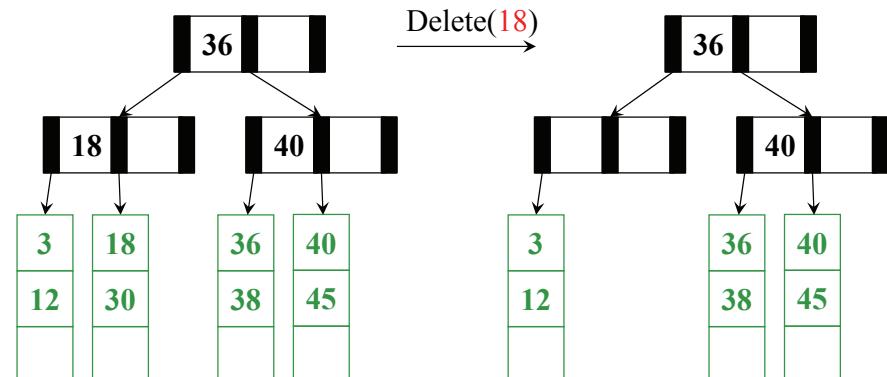
$M = 3 \ L = 3$

25



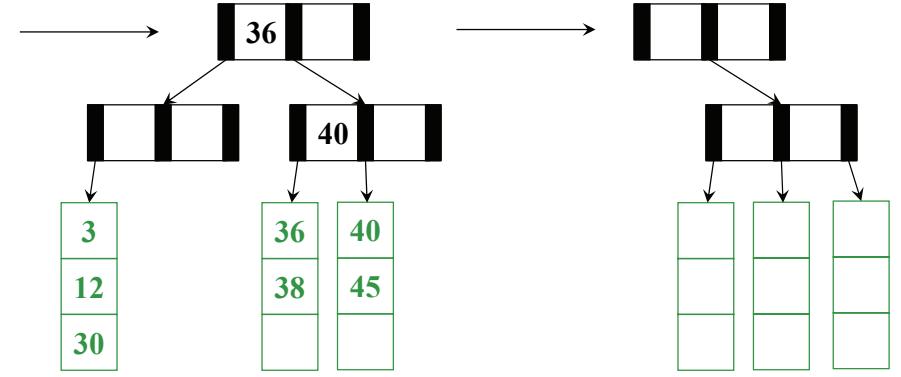
$M = 3 \ L = 3$

26



$M = 3 \ L = 3$

27



$M = 3 \ L = 3$

28

## Deletion Algorithm

1. Remove the key from its leaf
2. If the leaf ends up with fewer than  $\lceil L/2 \rceil$  items, **underflow!**
  - Adopt data from a neighbor; update the parent
  - If adopting won't work, delete node and merge with neighbor
  - If the parent ends up with fewer than  $\lceil M/2 \rceil$  children, **underflow!**

29

## Deletion Slide Two

3. If an internal node ends up with fewer than  $\lceil M/2 \rceil$  children, **underflow!**
  - Adopt from a neighbor; update the parent
  - If adoption won't work, merge with neighbor
  - If the parent ends up with fewer than  $\lceil M/2 \rceil$  children, **underflow!**
4. If the root ends up with only one child, make the child the new root of the tree
5. Propagate keys up through tree.

This reduces the height of the tree!

30

## Thinking about B+ Trees

- B+ Tree insertion can cause (expensive) splitting and propagation
- B+ Tree deletion can cause (cheap) adoption or (expensive) deletion, merging and propagation
- Propagation is rare if  $M$  and  $L$  are large (*Why?*)
- Pick branching factor  $M$  and data items/leaf  $L$  such that each node takes one full page/block of memory/disk.

31

## Tree Names You Might Encounter

### FYI:

- B-Trees with  $M = 3$ ,  $L = x$  are called **2-3 trees**
  - Nodes can have 2 or 3 keys
- B-Trees with  $M = 4$ ,  $L = x$  are called **2-3-4 trees**
  - Nodes can have 2, 3, or 4 keys

32

This chapter and Chapter 20 present data structures known as *mergeable heaps*, which support the following five operations.

`MAKE-HEAP()` creates and returns a new heap containing no elements.

`INSERT( $H, x$ )` inserts node  $x$ , whose *key* field has already been filled in, into heap  $H$ .

`MINIMUM( $H$ )` returns a pointer to the node in heap  $H$  whose key is minimum.

`EXTRACT-MIN( $H$ )` deletes the node from heap  $H$  whose key is minimum, returning a pointer to the node.

`UNION( $H_1, H_2$ )` creates and returns a new heap that contains all the nodes of heaps  $H_1$  and  $H_2$ . Heaps  $H_1$  and  $H_2$  are “destroyed” by this operation.

In addition, the data structures in these chapters also support the following two operations.

`DECREASE-KEY( $H, x, k$ )` assigns to node  $x$  within heap  $H$  the new key value  $k$ , which is assumed to be no greater than its current key value.<sup>1</sup>

`DELETE( $H, x$ )` deletes node  $x$  from heap  $H$ .

As the table in Figure 19.1 shows, if we don’t need the UNION operation, ordinary binary heaps, as used in heapsort (Chapter 6), work well. Operations other than UNION run in worst-case time  $O(\lg n)$  (or better) on a binary heap. If the UNION operation must be supported, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running MIN-HEAPIFY (see Exercise 6.2-2), the UNION operation takes  $\Theta(n)$  time in the worst case.

---

<sup>1</sup>As mentioned in the introduction to Part V, our default mergeable heaps are mergeable min-heaps, and so the operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY apply. Alternatively, we could define a *mergeable max-heap* with the operations MAXIMUM, EXTRACT-MAX, and INCREASE-KEY.

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

**Figure 19.1** Running times for operations on three implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by  $n$ .

In this chapter, we examine “binomial heaps,” whose worst-case time bounds are also shown in Figure 19.1. In particular, the UNION operation takes only  $O(\lg n)$  time to merge two binomial heaps with a total of  $n$  elements.

In Chapter 20, we shall explore Fibonacci heaps, which have even better time bounds for some operations. Note, however, that the running times for Fibonacci heaps in Figure 19.1 are amortized time bounds, not worst-case per-operation time bounds.

This chapter ignores issues of allocating nodes prior to insertion and freeing nodes following deletion. We assume that the code that calls the heap procedures deals with these details.

Binary heaps, binomial heaps, and Fibonacci heaps are all inefficient in their support of the operation SEARCH; it can take a while to find a node with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given node require a pointer to that node as part of their input. As in our discussion of priority queues in Section 6.5, when we use a mergeable heap in an application, we often store a handle to the corresponding application object in each mergeable-heap element, as well as a handle to corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Section 19.1 defines binomial heaps after first defining their constituent binomial trees. It also introduces a particular representation of binomial heaps. Section 19.2 shows how we can implement operations on binomial heaps in the time bounds given in Figure 19.1.

## 19.1 Binomial trees and binomial heaps

A binomial heap is a collection of binomial trees, so this section starts by defining binomial trees and proving some key properties. We then define binomial heaps and show how they can be represented.

### 19.1.1 Binomial trees

The **binomial tree**  $B_k$  is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.2(a), the binomial tree  $B_0$  consists of a single node. The binomial tree  $B_k$  consists of two binomial trees  $B_{k-1}$  that are **linked** together: the root of one is the leftmost child of the root of the other. Figure 19.2(b) shows the binomial trees  $B_0$  through  $B_4$ .

Some properties of binomial trees are given by the following lemma.

**Lemma 19.1 (Properties of binomial trees)**

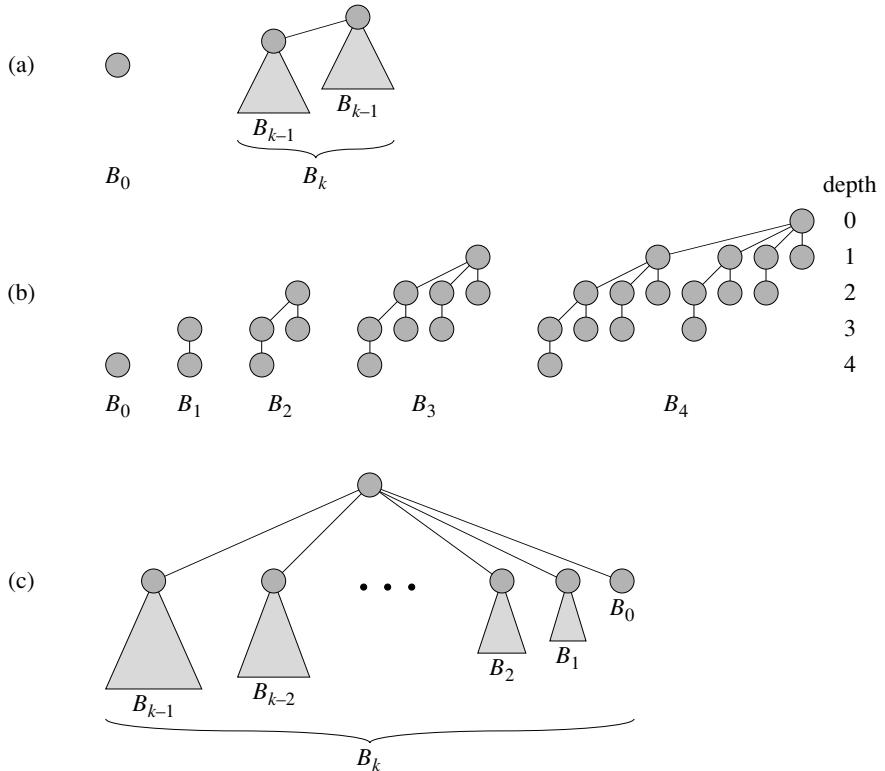
For the binomial tree  $B_k$ ,

1. there are  $2^k$  nodes,
2. the height of the tree is  $k$ ,
3. there are exactly  $\binom{k}{i}$  nodes at depth  $i$  for  $i = 0, 1, \dots, k$ , and
4. the root has degree  $k$ , which is greater than that of any other node; moreover if the children of the root are numbered from left to right by  $k - 1, k - 2, \dots, 0$ , child  $i$  is the root of a subtree  $B_i$ .

**Proof** The proof is by induction on  $k$ . For each property, the basis is the binomial tree  $B_0$ . Verifying that each property holds for  $B_0$  is trivial.

For the inductive step, we assume that the lemma holds for  $B_{k-1}$ .

1. Binomial tree  $B_k$  consists of two copies of  $B_{k-1}$ , and so  $B_k$  has  $2^{k-1} + 2^{k-1} = 2^k$  nodes.
2. Because of the way in which the two copies of  $B_{k-1}$  are linked to form  $B_k$ , the maximum depth of a node in  $B_k$  is one greater than the maximum depth in  $B_{k-1}$ . By the inductive hypothesis, this maximum depth is  $(k - 1) + 1 = k$ .
3. Let  $D(k, i)$  be the number of nodes at depth  $i$  of binomial tree  $B_k$ . Since  $B_k$  is composed of two copies of  $B_{k-1}$  linked together, a node at depth  $i$  in  $B_{k-1}$  appears in  $B_k$  once at depth  $i$  and once at depth  $i + 1$ . In other words, the number of nodes at depth  $i$  in  $B_k$  is the number of nodes at depth  $i$  in  $B_{k-1}$  plus



**Figure 19.2** (a) The recursive definition of the binomial tree  $B_k$ . Triangles represent rooted subtrees. (b) The binomial trees  $B_0$  through  $B_4$ . Node depths in  $B_4$  are shown. (c) Another way of looking at the binomial tree  $B_k$ .

the number of nodes at depth  $i - 1$  in  $B_{k-1}$ . Thus,

$$\begin{aligned}
 D(k, i) &= D(k-1, i) + D(k-1, i-1) \quad (\text{by the inductive hypothesis}) \\
 &= \binom{k-1}{i} + \binom{k-1}{i-1} \quad (\text{by Exercise C.1-7}) \\
 &= \binom{k}{i}.
 \end{aligned}$$

4. The only node with greater degree in  $B_k$  than in  $B_{k-1}$  is the root, which has one more child than in  $B_{k-1}$ . Since the root of  $B_{k-1}$  has degree  $k-1$ , the root of  $B_k$  has degree  $k$ . Now, by the inductive hypothesis, and as Figure 19.2(c) shows, from left to right, the children of the root of  $B_{k-1}$  are roots of  $B_{k-2}, B_{k-3}, \dots, B_0$ . When  $B_{k-1}$  is linked to  $B_{k-1}$ , therefore, the children of the resulting root are roots of  $B_{k-1}, B_{k-2}, \dots, B_0$ . ■

**Corollary 19.2**

The maximum degree of any node in an  $n$ -node binomial tree is  $\lg n$ .

**Proof** Immediate from properties 1 and 4 of Lemma 19.1. ■

The term “binomial tree” comes from property 3 of Lemma 19.1, since the terms  $\binom{k}{i}$  are the binomial coefficients. Exercise 19.1-3 gives further justification for the term.

### 19.1.2 Binomial heaps

A **binomial heap**  $H$  is a set of binomial trees that satisfies the following **binomial-heap properties**.

1. Each binomial tree in  $H$  obeys the **min-heap property**: the key of a node is greater than or equal to the key of its parent. We say that each such tree is **min-heap-ordered**.
2. For any nonnegative integer  $k$ , there is at most one binomial tree in  $H$  whose root has degree  $k$ .

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

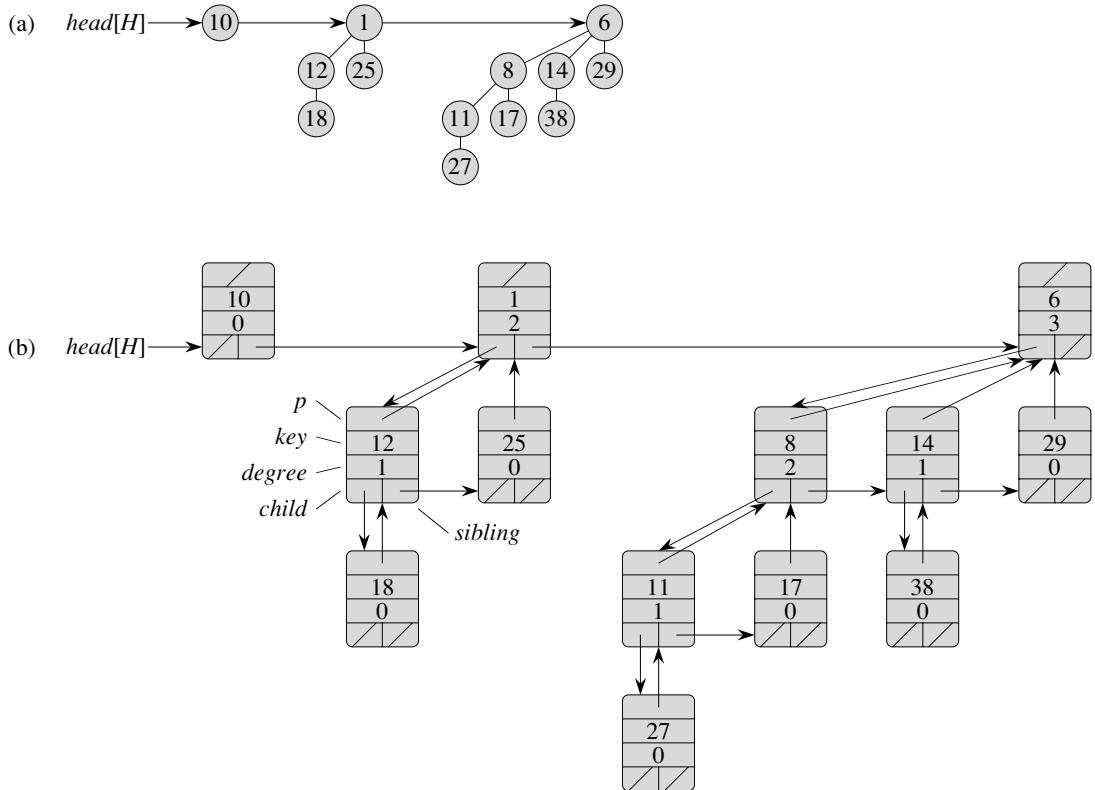
The second property implies that an  $n$ -node binomial heap  $H$  consists of at most  $\lfloor \lg n \rfloor + 1$  binomial trees. To see why, observe that the binary representation of  $n$  has  $\lfloor \lg n \rfloor + 1$  bits, say  $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \dots, b_0 \rangle$ , so that  $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$ . By property 1 of Lemma 19.1, therefore, binomial tree  $B_i$  appears in  $H$  if and only if bit  $b_i = 1$ . Thus, binomial heap  $H$  contains at most  $\lfloor \lg n \rfloor + 1$  binomial trees.

Figure 19.3(a) shows a binomial heap  $H$  with 13 nodes. The binary representation of 13 is  $\langle 1101 \rangle$ , and  $H$  consists of min-heap-ordered binomial trees  $B_3$ ,  $B_2$ , and  $B_0$ , having 8, 4, and 1 nodes respectively, for a total of 13 nodes.

### Representing binomial heaps

As shown in Figure 19.3(b), each binomial tree within a binomial heap is stored in the left-child, right-sibling representation of Section 10.4. Each node has a *key* field and any other satellite information required by the application. In addition, each node  $x$  contains pointers  $p[x]$  to its parent,  $child[x]$  to its leftmost child, and  $sibling[x]$  to the sibling of  $x$  immediately to its right. If node  $x$  is a root, then  $p[x] = \text{NIL}$ . If node  $x$  has no children, then  $child[x] = \text{NIL}$ , and if  $x$  is the rightmost child of its parent, then  $sibling[x] = \text{NIL}$ . Each node  $x$  also contains the field  $degree[x]$ , which is the number of children of  $x$ .

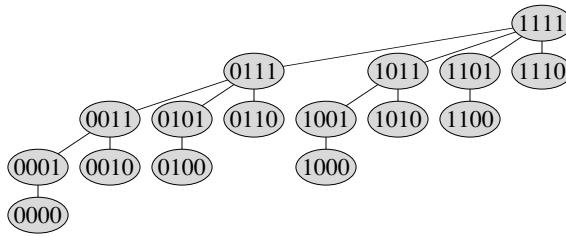
As Figure 19.3 also shows, the roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the **root list**. The degrees



**Figure 19.3** A binomial heap  $H$  with  $n = 13$  nodes. (a) The heap consists of binomial trees  $B_0, B_2$ , and  $B_3$ , which have 1, 4, and 8 nodes respectively, totaling  $n = 13$  nodes. Since each binomial tree is min-heap-ordered, the key of any node is no less than the key of its parent. Also shown is the root list, which is a linked list of roots in order of increasing degree. (b) A more detailed representation of binomial heap  $H$ . Each binomial tree is stored in the left-child, right-sibling representation, and each node stores its degree.

of the roots strictly increase as we traverse the root list. By the second binomial heap property, in an  $n$ -node binomial heap the degrees of the roots are a subset of  $\{0, 1, \dots, \lfloor \lg n \rfloor\}$ . The *sibling* field has a different meaning for roots than for nonroots. If  $x$  is a root, then  $sibling[x]$  points to the next root in the root list. (As usual,  $sibling[x] = \text{NIL}$  if  $x$  is the last root in the root list.)

A given binomial heap  $H$  is accessed by the field  $head[H]$ , which is simply a pointer to the first root in the root list of  $H$ . If binomial heap  $H$  has no elements, then  $head[H] = \text{NIL}$ .



**Figure 19.4** The binomial tree  $B_4$  with nodes labeled in binary by a postorder walk.

## Exercises

### 19.1-1

Suppose that  $x$  is a node in a binomial tree within a binomial heap, and assume that  $sibling[x] \neq \text{NIL}$ . If  $x$  is not a root, how does  $\text{degree}[sibling[x]]$  compare to  $\text{degree}[x]$ ? How about if  $x$  is a root?

### 19.1-2

If  $x$  is a nonroot node in a binomial tree within a binomial heap, how does  $\text{degree}[x]$  compare to  $\text{degree}[p[x]]$ ?

### 19.1-3

Suppose we label the nodes of binomial tree  $B_k$  in binary by a postorder walk, as in Figure 19.4. Consider a node  $x$  labeled  $l$  at depth  $i$ , and let  $j = k - i$ . Show that  $x$  has  $j$  1's in its binary representation. How many binary  $k$ -strings are there that contain exactly  $j$  1's? Show that the degree of  $x$  is equal to the number of 1's to the right of the rightmost 0 in the binary representation of  $l$ .

## 19.2 Operations on binomial heaps

In this section, we show how to perform operations on binomial heaps in the time bounds shown in Figure 19.1. We shall only show the upper bounds; the lower bounds are left as Exercise 19.2-10.

### Creating a new binomial heap

To make an empty binomial heap, the `MAKE-BINOMIAL-HEAP` procedure simply allocates and returns an object  $H$ , where  $\text{head}[H] = \text{NIL}$ . The running time is  $\Theta(1)$ .

### Finding the minimum key

The procedure `BINOMIAL-HEAP-MINIMUM` returns a pointer to the node with the minimum key in an  $n$ -node binomial heap  $H$ . This implementation assumes that there are no keys with value  $\infty$ . (See Exercise 19.2-5.)

`BINOMIAL-HEAP-MINIMUM( $H$ )`

```

1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{head}[H]$ 
3   $min \leftarrow \infty$ 
4  while  $x \neq \text{NIL}$ 
5    do if  $\text{key}[x] < min$ 
6      then  $min \leftarrow \text{key}[x]$ 
7       $y \leftarrow x$ 
8       $x \leftarrow \text{sibling}[x]$ 
9  return  $y$ 
```

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The `BINOMIAL-HEAP-MINIMUM` procedure checks all roots, which number at most  $\lfloor \lg n \rfloor + 1$ , saving the current minimum in  $min$  and a pointer to the current minimum in  $y$ . When called on the binomial heap of Figure 19.3, `BINOMIAL-HEAP-MINIMUM` returns a pointer to the node with key 1.

Because there are at most  $\lfloor \lg n \rfloor + 1$  roots to check, the running time of `BINOMIAL-HEAP-MINIMUM` is  $O(\lg n)$ .

### Uniting two binomial heaps

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The `BINOMIAL-HEAP-UNION` procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the  $B_{k-1}$  tree rooted at node  $y$  to the  $B_{k-1}$  tree rooted at node  $z$ ; that is, it makes  $z$  the parent of  $y$ . Node  $z$  thus becomes the root of a  $B_k$  tree.

`BINOMIAL-LINK( $y, z$ )`

```

1   $p[y] \leftarrow z$ 
2   $sibling[y] \leftarrow child[z]$ 
3   $child[z] \leftarrow y$ 
4   $degree[z] \leftarrow degree[z] + 1$ 
```

The `BINOMIAL-LINK` procedure makes node  $y$  the new head of the linked list of node  $z$ 's children in  $O(1)$  time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a  $B_k$  tree, the leftmost child of the root is the root of a  $B_{k-1}$  tree.

The following procedure unites binomial heaps  $H_1$  and  $H_2$ , returning the resulting heap. It destroys the representations of  $H_1$  and  $H_2$  in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIAL-HEAP-MERGE that merges the root lists of  $H_1$  and  $H_2$  into a single linked list that is sorted by degree into monotonically increasing order. The BINOMIAL-HEAP-MERGE procedure, whose pseudocode we leave as Exercise 19.2-1, is similar to the MERGE procedure in Section 2.3.1.

**BINOMIAL-HEAP-UNION( $H_1, H_2$ )**

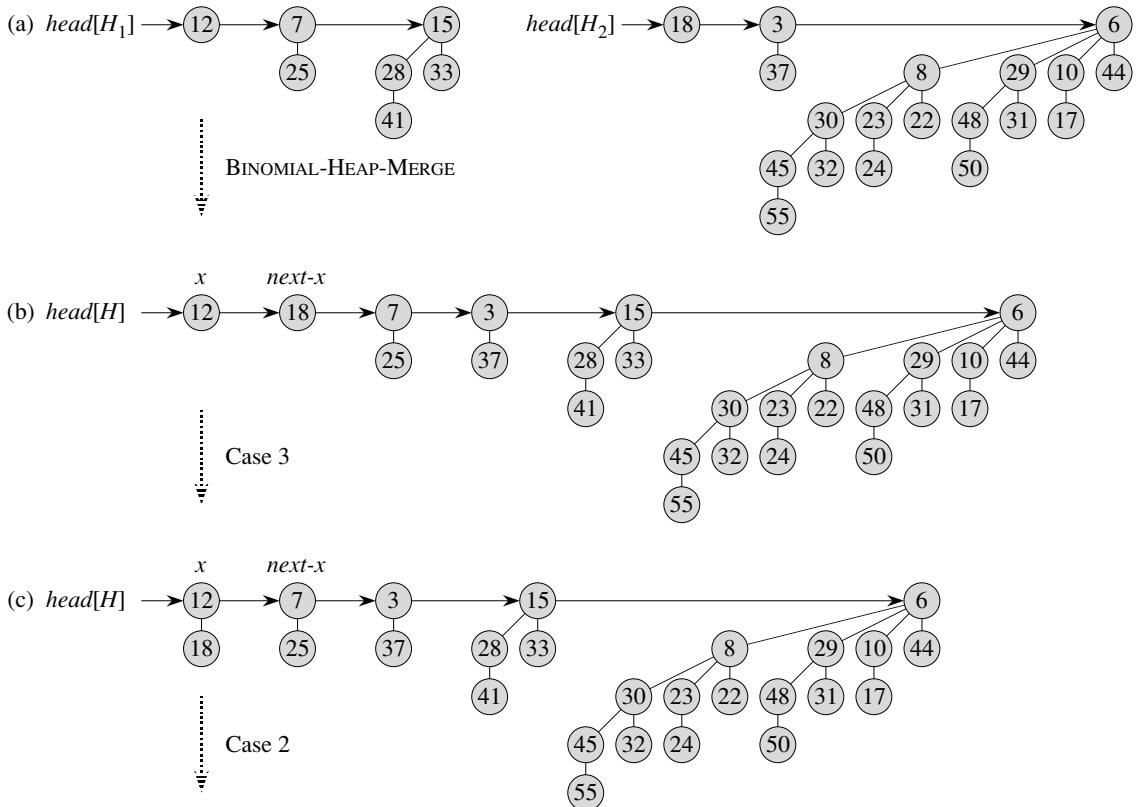
```

1   $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
2   $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$ 
3  free the objects  $H_1$  and  $H_2$  but not the lists they point to
4  if  $\text{head}[H] = \text{NIL}$ 
5    then return  $H$ 
6   $\text{prev-}x \leftarrow \text{NIL}$ 
7   $x \leftarrow \text{head}[H]$ 
8   $\text{next-}x \leftarrow \text{sibling}[x]$ 
9  while  $\text{next-}x \neq \text{NIL}$ 
10   do if ( $\text{degree}[x] \neq \text{degree}[\text{next-}x]$ ) or
        ( $\text{sibling}[\text{next-}x] \neq \text{NIL}$  and  $\text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x]$ )
11   then  $\text{prev-}x \leftarrow x$                                 ▷ Cases 1 and 2
12    $x \leftarrow \text{next-}x$                             ▷ Cases 1 and 2
13   else if  $\text{key}[x] \leq \text{key}[\text{next-}x]$ 
14     then  $\text{sibling}[x] \leftarrow \text{sibling}[\text{next-}x]$       ▷ Case 3
15      $\text{BINOMIAL-LINK}(\text{next-}x, x)$                   ▷ Case 3
16     else if  $\text{prev-}x = \text{NIL}$                       ▷ Case 4
17       then  $\text{head}[H] \leftarrow \text{next-}x$             ▷ Case 4
18       else  $\text{sibling}[\text{prev-}x] \leftarrow \text{next-}x$  ▷ Case 4
19      $\text{BINOMIAL-LINK}(x, \text{next-}x)$                   ▷ Case 4
20      $x \leftarrow \text{next-}x$                             ▷ Case 4
21    $\text{next-}x \leftarrow \text{sibling}[x]$ 
22 return  $H$ 

```

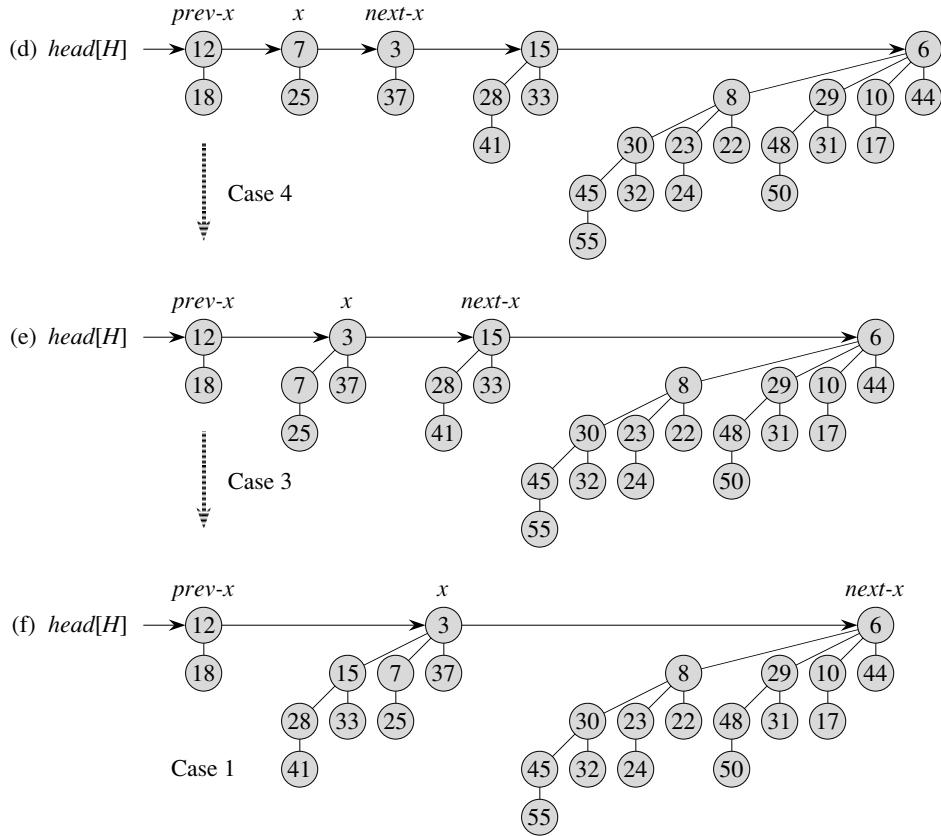
Figure 19.5 shows an example of BINOMIAL-HEAP-UNION in which all four cases given in the pseudocode occur.

The BINOMIAL-HEAP-UNION procedure has two phases. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps  $H_1$  and  $H_2$  into a single linked list  $H$  that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Because the linked list  $H$  is sorted by degree, we can perform all the link operations quickly.



**Figure 19.5** The execution of `BINOMIAL-HEAP-UNION`. **(a)** Binomial heaps  $H_1$  and  $H_2$ . **(b)** Binomial heap  $H$  is the output of `BINOMIAL-HEAP-MERGE`( $H_1, H_2$ ). Initially,  $x$  is the first root on the root list of  $H$ . Because both  $x$  and  $next-x$  have degree 0 and  $key[x] < key[next-x]$ , case 3 applies. **(c)** After the link occurs,  $x$  is the first of three roots with the same degree, so case 2 applies. **(d)** After all the pointers move down one position in the root list, case 4 applies, since  $x$  is the first of two roots of equal degree. **(e)** After the link occurs, case 3 applies. **(f)** After another link, case 1 applies, because  $x$  has degree 3 and  $next-x$  has degree 4. This iteration of the **while** loop is the last, because after the pointers move down one position in the root list,  $next-x = \text{NIL}$ .

In detail, the procedure works as follows. Lines 1–3 start by merging the root lists of binomial heaps  $H_1$  and  $H_2$  into a single root list  $H$ . The root lists of  $H_1$  and  $H_2$  are sorted by strictly increasing degree, and `BINOMIAL-HEAP-MERGE` returns a root list  $H$  that is sorted by monotonically increasing degree. If the root lists of  $H_1$  and  $H_2$  have  $m$  roots altogether, `BINOMIAL-HEAP-MERGE` runs in  $O(m)$  time by repeatedly examining the roots at the heads of the two root lists and appending the root with the lower degree to the output root list, removing it from its input root list in the process.



The BINOMIAL-HEAP-UNION procedure next initializes some pointers into the root list of  $H$ . First, it simply returns in lines 4–5 if it happens to be uniting two empty binomial heaps. From line 6 on, therefore, we know that  $H$  has at least one root. Throughout the procedure, we maintain three pointers into the root list:

- $x$  points to the root currently being examined,
- $prev-x$  points to the root preceding  $x$  on the root list:  $sibling[prev-x] = x$  (since initially  $x$  has no predecessor, we start with  $prev-x$  set to NIL), and
- $next-x$  points to the root following  $x$  on the root list:  $sibling[x] = next-x$ .

Initially, there are at most two roots on the root list  $H$  of a given degree: because  $H_1$  and  $H_2$  were binomial heaps, they each had at most one root of a given degree. Moreover, BINOMIAL-HEAP-MERGE guarantees us that if two roots in  $H$  have the same degree, they are adjacent in the root list.

In fact, during the execution of BINOMIAL-HEAP-UNION, there may be three roots of a given degree appearing on the root list  $H$  at some time. We shall see

in a moment how this situation could occur. At each iteration of the **while** loop of lines 9–21, therefore, we decide whether to link  $x$  and  $next-x$  based on their degrees and possibly the degree of  $sibling[next-x]$ . An invariant of the loop is that each time we start the body of the loop, both  $x$  and  $next-x$  are non-NIL. (See Exercise 19.2-4 for a precise loop invariant.)

Case 1, shown in Figure 19.6(a), occurs when  $degree[x] \neq degree[next-x]$ , that is, when  $x$  is the root of a  $B_k$ -tree and  $next-x$  is the root of a  $B_l$ -tree for some  $l > k$ . Lines 11–12 handle this case. We don't link  $x$  and  $next-x$ , so we simply march the pointers one position farther down the list. Updating  $next-x$  to point to the node following the new node  $x$  is handled in line 21, which is common to every case.

Case 2, shown in Figure 19.6(b), occurs when  $x$  is the first of three roots of equal degree, that is, when

$$degree[x] = degree[next-x] = degree[sibling[next-x]].$$

We handle this case in the same manner as case 1: we just march the pointers one position farther down the list. The next iteration will execute either case 3 or case 4 to combine the second and third of the three equal-degree roots. Line 10 tests for both cases 1 and 2, and lines 11–12 handle both cases.

Cases 3 and 4 occur when  $x$  is the first of two roots of equal degree, that is, when  $degree[x] = degree[next-x] \neq degree[sibling[next-x]]$ .

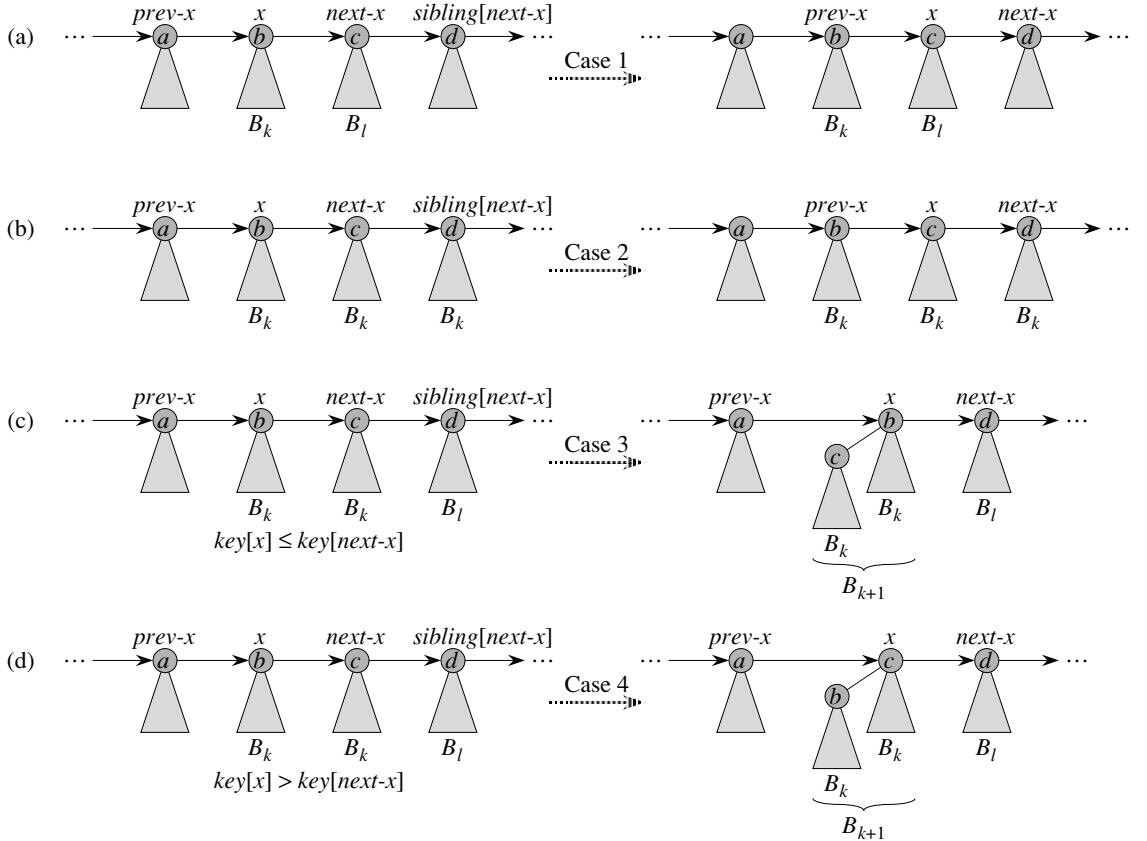
These cases may occur in any iteration, but one of them always occurs immediately following case 2. In cases 3 and 4, we link  $x$  and  $next-x$ . The two cases are distinguished by whether  $x$  or  $next-x$  has the smaller key, which determines the node that will be the root after the two are linked.

In case 3, shown in Figure 19.6(c),  $key[x] \leq key[next-x]$ , so  $next-x$  is linked to  $x$ . Line 14 removes  $next-x$  from the root list, and line 15 makes  $next-x$  the leftmost child of  $x$ .

In case 4, shown in Figure 19.6(d),  $next-x$  has the smaller key, so  $x$  is linked to  $next-x$ . Lines 16–18 remove  $x$  from the root list; there are two cases depending on whether  $x$  is the first root on the list (line 17) or is not (line 18). Line 19 then makes  $x$  the leftmost child of  $next-x$ , and line 20 updates  $x$  for the next iteration.

Following either case 3 or case 4, the setup for the next iteration of the **while** loop is the same. We have just linked two  $B_k$ -trees to form a  $B_{k+1}$ -tree, which  $x$  now points to. There were already zero, one, or two other  $B_{k+1}$ -trees on the root list resulting from BINOMIAL-HEAP-MERGE, so  $x$  is now the first of either one, two, or three  $B_{k+1}$ -trees on the root list. If  $x$  is the only one, then we enter case 1 in the next iteration:  $degree[x] \neq degree[next-x]$ . If  $x$  is the first of two, then we enter either case 3 or case 4 in the next iteration. It is when  $x$  is the first of three that we enter case 2 in the next iteration.

The running time of BINOMIAL-HEAP-UNION is  $O(\lg n)$ , where  $n$  is the total number of nodes in binomial heaps  $H_1$  and  $H_2$ . We can see this as follows. Let  $H_1$



**Figure 19.6** The four cases that occur in `BINOMIAL-HEAP-UNION`. Labels  $a, b, c$ , and  $d$  serve only to identify the roots involved; they do not indicate the degrees or keys of these roots. In each case,  $x$  is the root of a  $B_k$ -tree and  $l > k$ . **(a)** Case 1:  $degree[x] \neq degree[next-x]$ . The pointers move one position farther down the root list. **(b)** Case 2:  $degree[x] = degree[next-x] = degree[sibling[next-x]]$ . Again, the pointers move one position farther down the list, and the next iteration executes either case 3 or case 4. **(c)** Case 3:  $degree[x] = degree[next-x] \neq degree[sibling[next-x]]$  and  $key[x] \leq key[next-x]$ . We remove  $next-x$  from the root list and link it to  $x$ , creating a  $B_{k+1}$ -tree. **(d)** Case 4:  $degree[x] = degree[next-x] \neq degree[sibling[next-x]]$  and  $key[next-x] \leq key[x]$ . We remove  $x$  from the root list and link it to  $next-x$ , again creating a  $B_{k+1}$ -tree.

contain  $n_1$  nodes and  $H_2$  contain  $n_2$  nodes, so that  $n = n_1 + n_2$ . Then  $H_1$  contains at most  $\lfloor \lg n_1 \rfloor + 1$  roots and  $H_2$  contains at most  $\lfloor \lg n_2 \rfloor + 1$  roots, and so  $H$  contains at most  $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2 \lfloor \lg n \rfloor + 2 = O(\lg n)$  roots immediately after the call of `BINOMIAL-HEAP-MERGE`. The time to perform `BINOMIAL-HEAP-MERGE` is thus  $O(\lg n)$ . Each iteration of the `while` loop takes  $O(1)$  time, and there are at most  $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$  iterations because each iteration either advances the

pointers one position down the root list of  $H$  or removes a root from the root list. The total time is thus  $O(\lg n)$ .

### Inserting a node

The following procedure inserts node  $x$  into binomial heap  $H$ , assuming that  $x$  has already been allocated and  $\text{key}[x]$  has already been filled in.

**BINOMIAL-HEAP-INSERT( $H, x$ )**

- 1  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$
- 2  $p[x] \leftarrow \text{NIL}$
- 3  $\text{child}[x] \leftarrow \text{NIL}$
- 4  $\text{sibling}[x] \leftarrow \text{NIL}$
- 5  $\text{degree}[x] \leftarrow 0$
- 6  $\text{head}[H'] \leftarrow x$
- 7  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$

The procedure simply makes a one-node binomial heap  $H'$  in  $O(1)$  time and unites it with the  $n$ -node binomial heap  $H$  in  $O(\lg n)$  time. The call to BINOMIAL-HEAP-UNION takes care of freeing the temporary binomial heap  $H'$ . (A direct implementation that does not call BINOMIAL-HEAP-UNION is given as Exercise 19.2-8.)

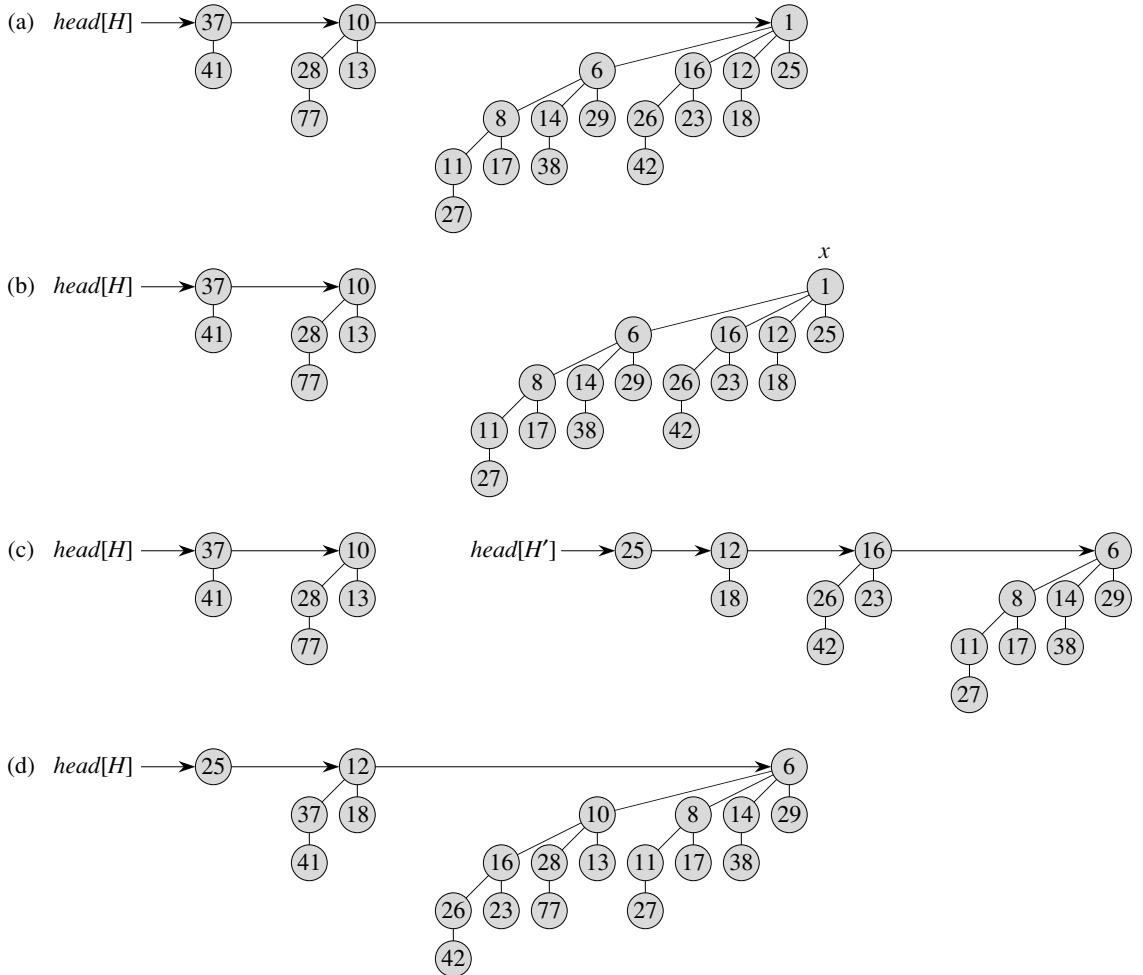
### Extracting the node with minimum key

The following procedure extracts the node with the minimum key from binomial heap  $H$  and returns a pointer to the extracted node.

**BINOMIAL-HEAP-EXTRACT-MIN( $H$ )**

- 1 find the root  $x$  with the minimum key in the root list of  $H$ ,  
and remove  $x$  from the root list of  $H$
- 2  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$
- 3 reverse the order of the linked list of  $x$ 's children,  
and set  $\text{head}[H']$  to point to the head of the resulting list
- 4  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$
- 5 **return**  $x$

This procedure works as shown in Figure 19.7. The input binomial heap  $H$  is shown in Figure 19.7(a). Figure 19.7(b) shows the situation after line 1: the root  $x$  with the minimum key has been removed from the root list of  $H$ . If  $x$  is the root of a  $B_k$ -tree, then by property 4 of Lemma 19.1,  $x$ 's children, from left to right, are roots of  $B_{k-1}$ ,  $B_{k-2}$ , ...,  $B_0$ -trees. Figure 19.7(c) shows that by reversing the list of  $x$ 's children in line 3, we have a binomial heap  $H'$  that contains every node



**Figure 19.7** The action of BINOMIAL-HEAP-EXTRACT-MIN. (a) A binomial heap  $H$ . (b) The root  $x$  with minimum key is removed from the root list of  $H$ . (c) The linked list of  $x$ 's children is reversed, giving another binomial heap  $H'$ . (d) The result of uniting  $H$  and  $H'$ .

in  $x$ 's tree except for  $x$  itself. Because  $x$ 's tree was removed from  $H$  in line 1, the binomial heap that results from uniting  $H$  and  $H'$  in line 4, shown in Figure 19.7(d), contains all the nodes originally in  $H$  except for  $x$ . Finally, line 5 returns  $x$ .

Since each of lines 1–4 takes  $O(\lg n)$  time if  $H$  has  $n$  nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in  $O(\lg n)$  time.

## Decreasing a key

The following procedure decreases the key of a node  $x$  in a binomial heap  $H$  to a new value  $k$ . It signals an error if  $k$  is greater than  $x$ 's current key.

**BINOMIAL-HEAP-DECREASE-KEY( $H, x, k$ )**

```

1  if  $k > \text{key}[x]$ 
2    then error “new key is greater than current key”
3   $\text{key}[x] \leftarrow k$ 
4   $y \leftarrow x$ 
5   $z \leftarrow p[y]$ 
6  while  $z \neq \text{NIL}$  and  $\text{key}[y] < \text{key}[z]$ 
7    do exchange  $\text{key}[y] \leftrightarrow \text{key}[z]$ 
8      ▷ If  $y$  and  $z$  have satellite fields, exchange them, too.
9       $y \leftarrow z$ 
10      $z \leftarrow p[y]$ 
```

As shown in Figure 19.8, this procedure decreases a key in the same manner as in a binary min-heap: by “bubbling up” the key in the heap. After ensuring that the new key is in fact no greater than the current key and then assigning the new key to  $x$ , the procedure goes up the tree, with  $y$  initially pointing to node  $x$ . In each iteration of the **while** loop of lines 6–10,  $\text{key}[y]$  is checked against the key of  $y$ 's parent  $z$ . If  $y$  is the root or  $\text{key}[y] \geq \text{key}[z]$ , the binomial tree is now min-heap-ordered. Otherwise, node  $y$  violates min-heap ordering, and so its key is exchanged with the key of its parent  $z$ , along with any other satellite information. The procedure then sets  $y$  to  $z$ , going up one level in the tree, and continues with the next iteration.

The **BINOMIAL-HEAP-DECREASE-KEY** procedure takes  $O(\lg n)$  time. By property 2 of Lemma 19.1, the maximum depth of  $x$  is  $\lfloor \lg n \rfloor$ , so the **while** loop of lines 6–10 iterates at most  $\lfloor \lg n \rfloor$  times.

## Deleting a key

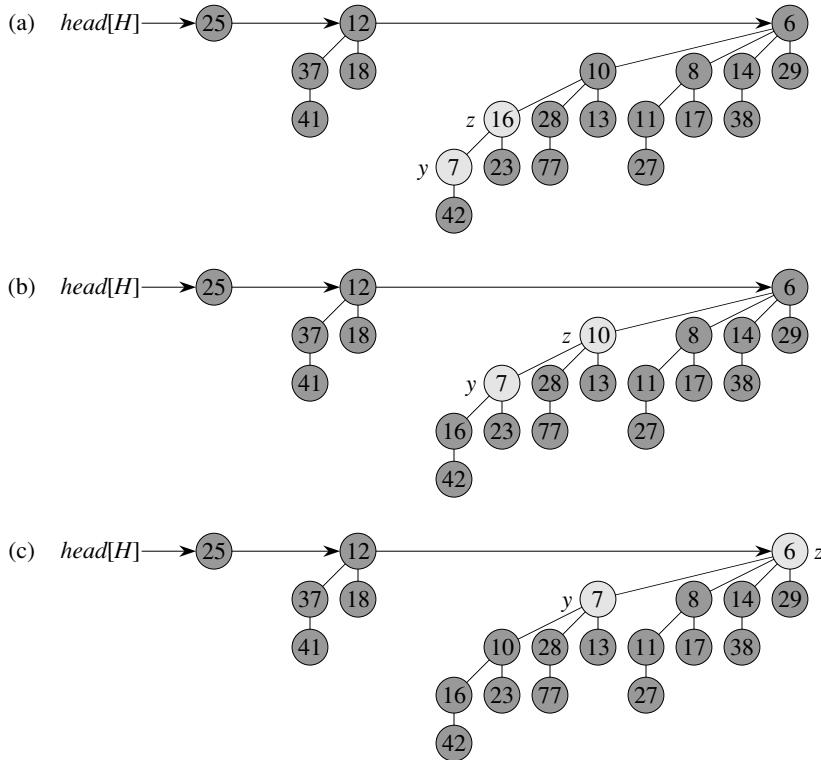
It is easy to delete a node  $x$ 's key and satellite information from binomial heap  $H$  in  $O(\lg n)$  time. The following implementation assumes that no node currently in the binomial heap has a key of  $-\infty$ .

**BINOMIAL-HEAP-DELETE( $H, x$ )**

```

1  BINOMIAL-HEAP-DECREASE-KEY( $H, x, -\infty$ )
2  BINOMIAL-HEAP-EXTRACT-MIN( $H$ )
```

The **BINOMIAL-HEAP-DELETE** procedure makes node  $x$  have the unique minimum key in the entire binomial heap by giving it a key of  $-\infty$ . (Exercise 19.2-6



**Figure 19.8** The action of BINOMIAL-HEAP-DECREASE-KEY. **(a)** The situation just before line 6 of the first iteration of the **while** loop. Node  $y$  has had its key decreased to 7, which is less than the key of  $y$ 's parent  $z$ . **(b)** The keys of the two nodes are exchanged, and the situation just before line 6 of the second iteration is shown. Pointers  $y$  and  $z$  have moved up one level in the tree, but min-heap order is still violated. **(c)** After another exchange and moving pointers  $y$  and  $z$  up one more level, we find that min-heap order is satisfied, so the **while** loop terminates.

deals with the situation in which  $-\infty$  cannot appear as a key, even temporarily.) It then bubbles this key and the associated satellite information up to a root by calling **BINOMIAL-HEAP-DECREASE-KEY**. This root is then removed from  $H$  by a call of **BINOMIAL-HEAP-EXTRACT-MIN**.

The BINOMIAL-HEAP-DELETE procedure takes  $O(\lg n)$  time.

## Exercises

19.2-1

Write pseudocode for BINOMIAL-HEAP-MERGE.

**19.2-2**

Show the binomial heap that results when a node with key 24 is inserted into the binomial heap shown in Figure 19.7(d).

**19.2-3**

Show the binomial heap that results when the node with key 28 is deleted from the binomial heap shown in Figure 19.8(c).

**19.2-4**

Argue the correctness of BINOMIAL-HEAP-UNION using the following loop invariant:

At the start of each iteration of the **while** loop of lines 9–21,  $x$  points to a root that is one of the following:

- the only root of its degree,
- the first of the only two roots of its degree, or
- the first or second of the only three roots of its degree.

Moreover, all roots preceding  $x$ 's predecessor on the root list have unique degrees on the root list, and if  $x$ 's predecessor has a degree different from that of  $x$ , its degree on the root list is unique, too. Finally, node degrees monotonically increase as we traverse the root list.

**19.2-5**

Explain why the BINOMIAL-HEAP-MINIMUM procedure might not work correctly if keys can have the value  $\infty$ . Rewrite the pseudocode to make it work correctly in such cases.

**19.2-6**

Suppose there is no way to represent the key  $-\infty$ . Rewrite the BINOMIAL-HEAP-DELETE procedure to work correctly in this situation. It should still take  $O(\lg n)$  time.

**19.2-7**

Discuss the relationship between inserting into a binomial heap and incrementing a binary number and the relationship between uniting two binomial heaps and adding two binary numbers.

**19.2-8**

In light of Exercise 19.2-7, rewrite BINOMIAL-HEAP-INSERT to insert a node directly into a binomial heap without calling BINOMIAL-HEAP-UNION.

**19.2-9**

Show that if root lists are kept in strictly decreasing order by degree (instead of strictly increasing order), each of the binomial heap operations can be implemented without changing its asymptotic running time.

**19.2-10**

Find inputs that cause BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY, and BINOMIAL-HEAP-DELETE to run in  $\Omega(\lg n)$  time. Explain why the worst-case running times of BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM, and BINOMIAL-HEAP-UNION are  $\tilde{\Omega}(\lg n)$  but not  $\Omega(\lg n)$ . (See Problem 3-5.)

## Problems

**19-1 2-3-4 heaps**

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement **2-3-4 heaps**, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf  $x$  stores exactly one key in the field  $key[x]$ . There is no particular ordering of the keys in the leaves; that is, from left to right, the keys may be in any order. Each internal node  $x$  contains a value  $small[x]$  that is equal to the smallest key stored in any leaf in the subtree rooted at  $x$ . The root  $r$  contains a field  $height[r]$  that is the height of the tree. Finally, 2-3-4 heaps are intended to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. Each of the operations in parts (a)–(e) should run in  $O(\lg n)$  time on a 2-3-4 heap with  $n$  elements. The UNION operation in part (f) should run in  $O(\lg n)$  time, where  $n$  is the number of elements in the two input heaps.

- a. MINIMUM, which returns a pointer to the leaf with the smallest key.
- b. DECREASE-KEY, which decreases the key of a given leaf  $x$  to a given value  $k \leq key[x]$ .
- c. INSERT, which inserts leaf  $x$  with key  $k$ .
- d. DELETE, which deletes a given leaf  $x$ .
- e. EXTRACT-MIN, which extracts the leaf with the smallest key.

- f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

### 19-2 Minimum-spanning-tree algorithm using binomial heaps

Chapter 23 presents two algorithms to solve the problem of finding a minimum spanning tree of an undirected graph. Here, we shall see how binomial heaps can be used to devise a different minimum-spanning-tree algorithm.

We are given a connected, undirected graph  $G = (V, E)$  with a weight function  $w : E \rightarrow \mathbf{R}$ . We call  $w(u, v)$  the weight of edge  $(u, v)$ . We wish to find a minimum spanning tree for  $G$ : an acyclic subset  $T \subseteq E$  that connects all the vertices in  $V$  and whose total weight

$$w(T) = \sum_{(u, v) \in T} w(u, v)$$

is minimized.

The following pseudocode, which can be proven correct using techniques from Section 23.1, constructs a minimum spanning tree  $T$ . It maintains a partition  $\{V_i\}$  of the vertices of  $V$  and, with each set  $V_i$ , a set

$$E_i \subseteq \{(u, v) : u \in V_i \text{ or } v \in V_i\}$$

of edges incident on vertices in  $V_i$ .

MST( $G$ )

```

1   $T \leftarrow \emptyset$ 
2  for each vertex  $v_i \in V[G]$ 
3      do  $V_i \leftarrow \{v_i\}$ 
4           $E_i \leftarrow \{(v_i, v) \in E[G]\}$ 
5  while there is more than one set  $V_i$ 
6      do choose any set  $V_i$ 
7          extract the minimum-weight edge  $(u, v)$  from  $E_i$ 
8          assume without loss of generality that  $u \in V_i$  and  $v \in V_j$ 
9          if  $i \neq j$ 
10             then  $T \leftarrow T \cup \{(u, v)\}$ 
11              $V_i \leftarrow V_i \cup V_j$ , destroying  $V_j$ 
12              $E_i \leftarrow E_i \cup E_j$ 
```

Describe how to implement this algorithm using binomial heaps to manage the vertex and edge sets. Do you need to change the representation of a binomial heap? Do you need to add operations beyond the mergeable-heap operations given in Figure 19.1? Give the running time of your implementation.

---

**Chapter notes**

Binomial heaps were introduced in 1978 by Vuillemin [307]. Brown [49, 50] studied their properties in detail.

# Fibonacci Heap

# Heaps as Priority Queues

- You have seen binary min-heaps/max-heaps
- Can support creating a heap, insert, finding/extracting the min (max) efficiently
- Can also support decrease-key operations efficiently
- However, not good for merging two heaps
  - $O(n)$  where  $n$  is the total no. of elements in the two heaps
- Variations of heaps exist that can merge heaps efficiently
  - May also improve the complexity of the other operations
  - Ex. Binomial heaps, Fibonacci heaps
- We will study Fibonacci heaps, an amortized data structure

# A Comparison

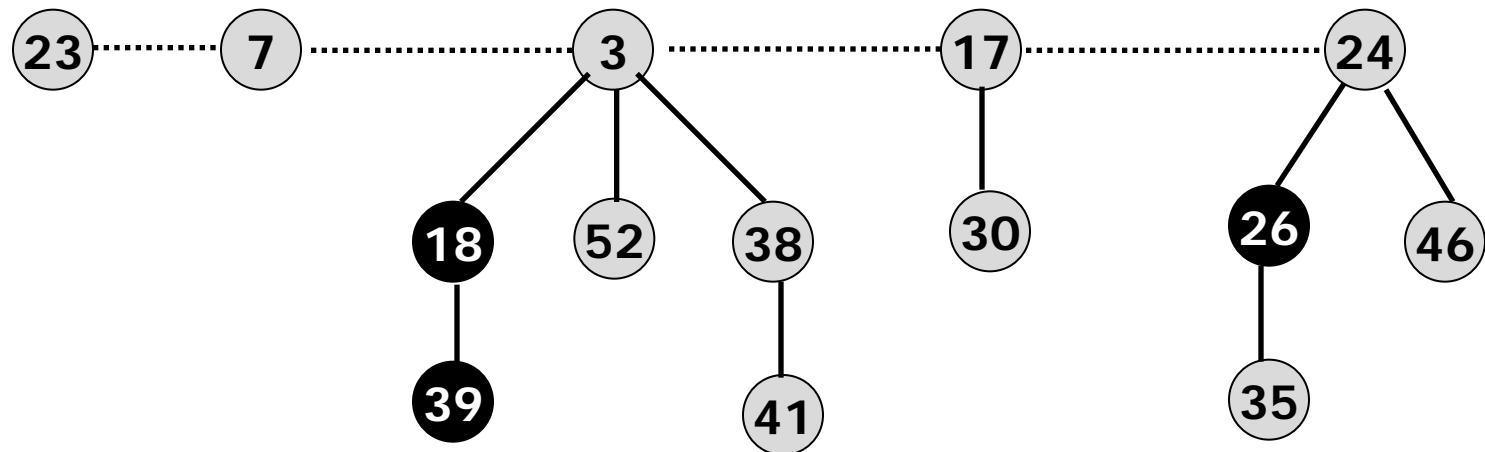
Operation	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
MERGE/UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

# Fibonacci Heap

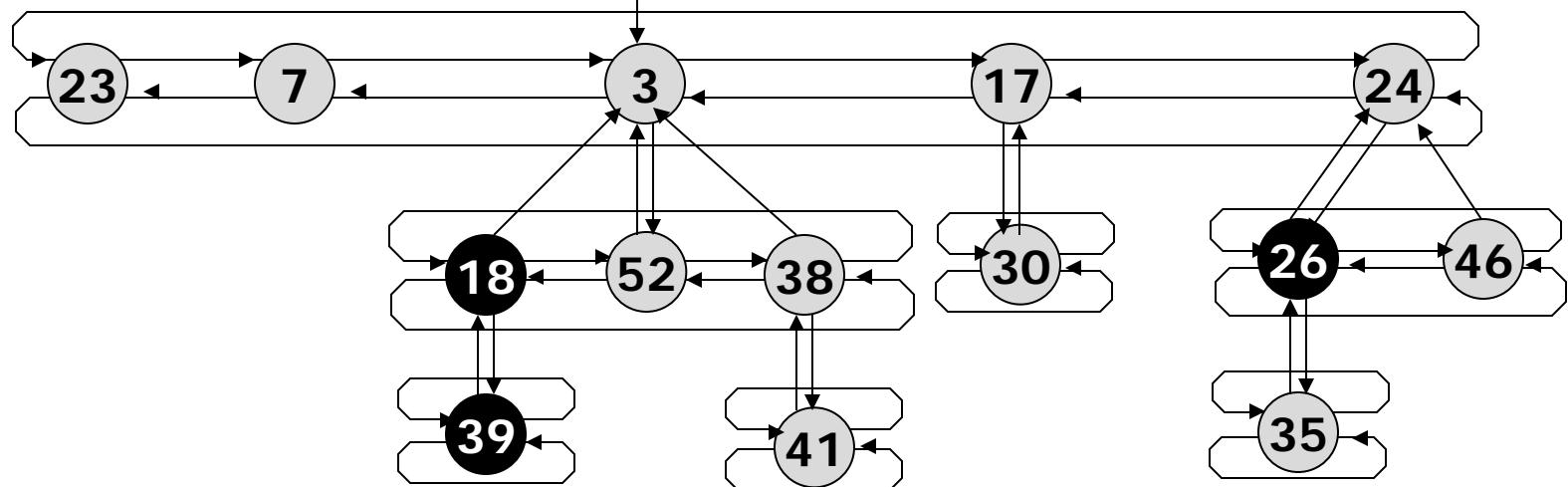
- A collection of min-heap ordered trees
  - Each tree is rooted but “unordered”, meaning there is no order between the child nodes of a node (unlike, for ex., left child and right child in a rooted, ordered binary tree)
  - Each node  $x$  has
    - One parent pointer  $p[x]$
    - One child pointer  $\text{child}[x]$  which points to an arbitrary child of  $x$
    - The children of  $x$  are linked together in a circular, doubly linked list
      - Each node  $y$  has pointers  $\text{left}[y]$  and  $\text{right}[y]$  to its left and right node in the list
      - So  $x$  basically stores a pointer to start in this list of its children

- The root of the trees are again connected with a circular, doubly linked list using their left and right pointers
- A Fibonacci heap  $H$  is defined by
  - A pointer  $\text{min}[H]$  which points to the root of a tree containing the minimum element (minimum node of the heap)
  - A variable  $n[H]$  storing the number of elements in the heap

$\min[H]$



$\min[H]$



# Additional Variables

- Each node  $x$  also has two other fields
  - $\text{degree}[x]$  – stores the number of children of  $x$
  - $\text{mark}[x]$  – indicates whether  $x$  has lost a child since the last time  $x$  was made the child of another node
    - We will denote marked nodes by color black, and unmarked ones by color grey
    - A newly created node is unmarked
    - A marked node also becomes unmarked whenever it is made the child of another node

# Amortized Analysis

- We mentioned Fibonacci heap is an amortized data structure
- We will use the potential method to analyse
- Let  $t(H)$  = no. of trees in a Fibonacci heap  $H$
- Let  $m(H)$  = number of marked nodes in  $H$
- Potential function used

$$\Phi(H) = t(H) + 2m(H)$$

# Operations

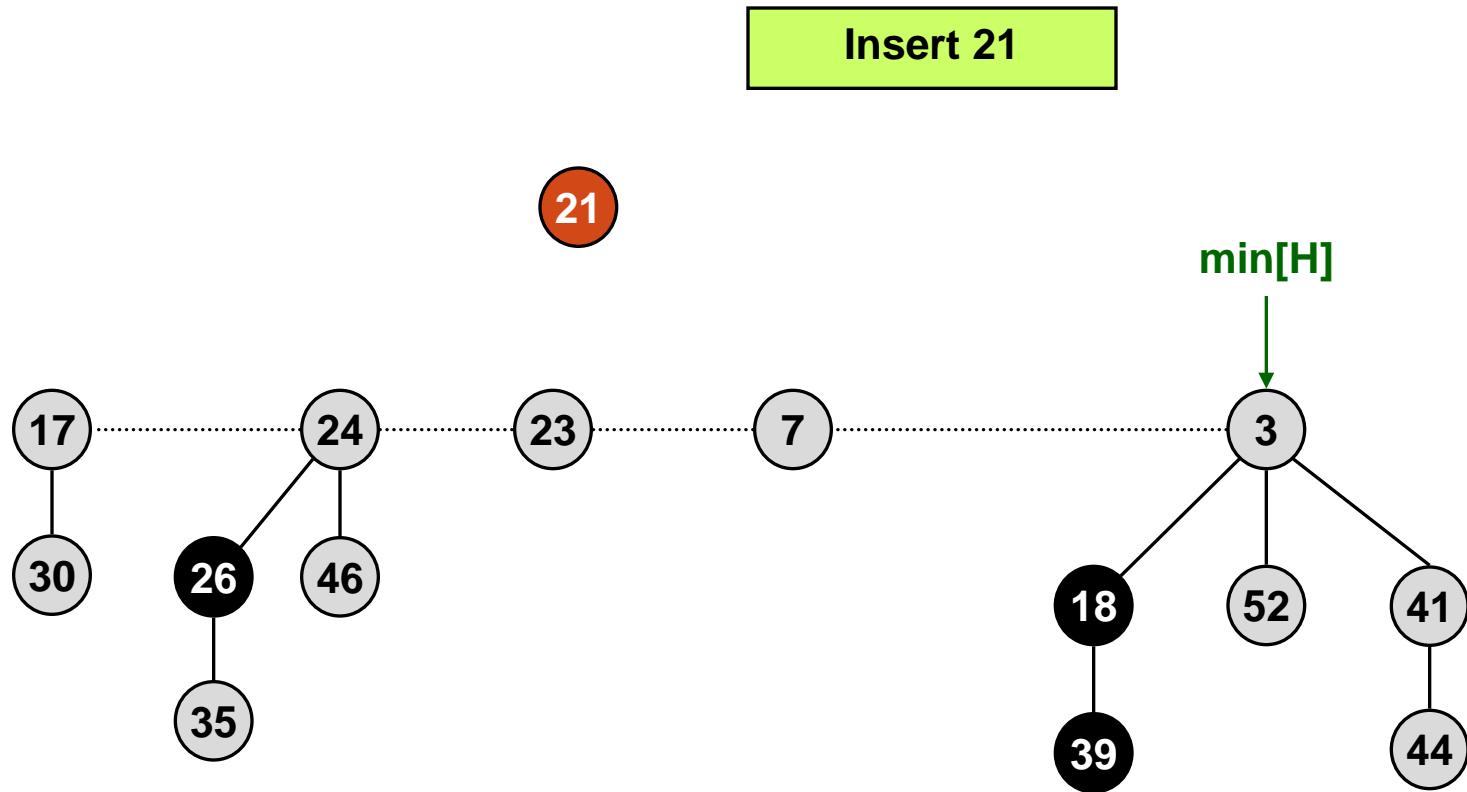
- Create an empty Fibonacci heap
- Insert an element in a Fibonacci heap
- Merge two Fibonacci heaps (Union)
- Extract the minimum element from a Fibonacci heap
- Decrease the value of an element in a Fibonacci heap
- Delete an element from a Fibonacci heap

# Creating a Fibonacci Heap

- This creates an empty Fibonacci heap
- Create an object to store  $\min[H]$  and  $n[H]$
- Initialize  $\min[H] = \text{NIL}$  and  $n[H] = 0$
- Potential of the newly created heap  $\Phi(H) = 0$
- Amortized cost = actual cost =  $O(1)$

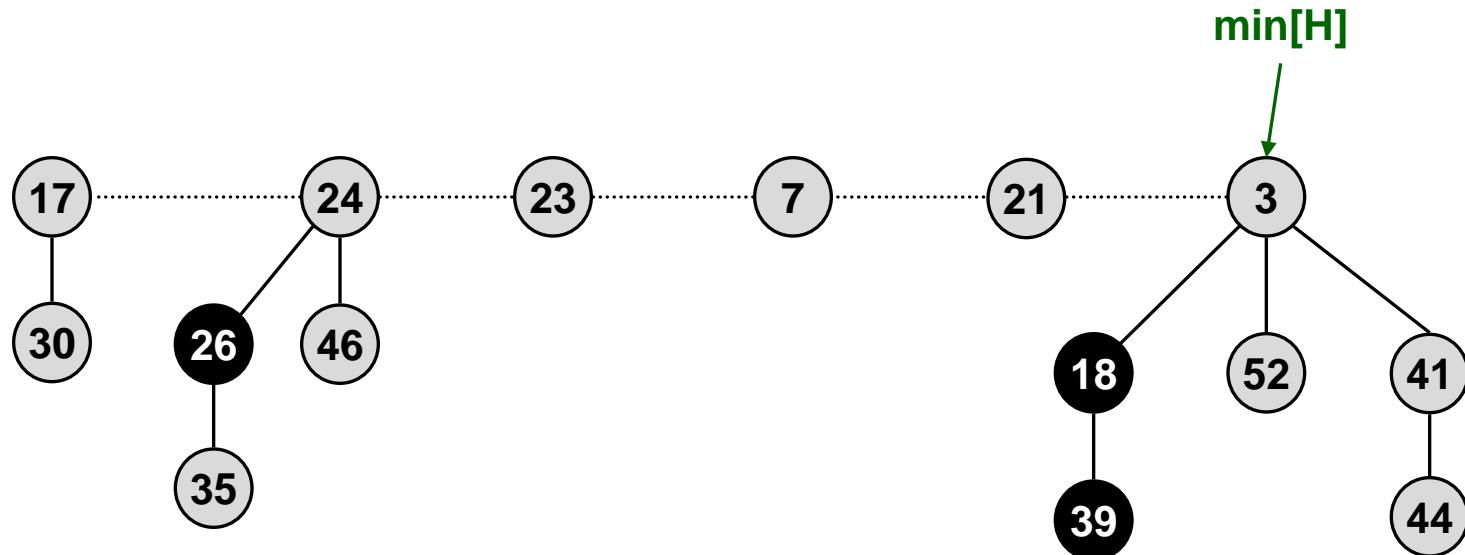
# Inserting an Element

- Add the element to the left of  $\text{min}[\text{H}]$
- Update  $\text{min}[\text{H}]$  if needed



# Inserting an Element (contd.)

- Add the element to the left of node pointed to by  $\text{min}[\text{H}]$
- Update  $\text{min}[\text{H}]$  if needed

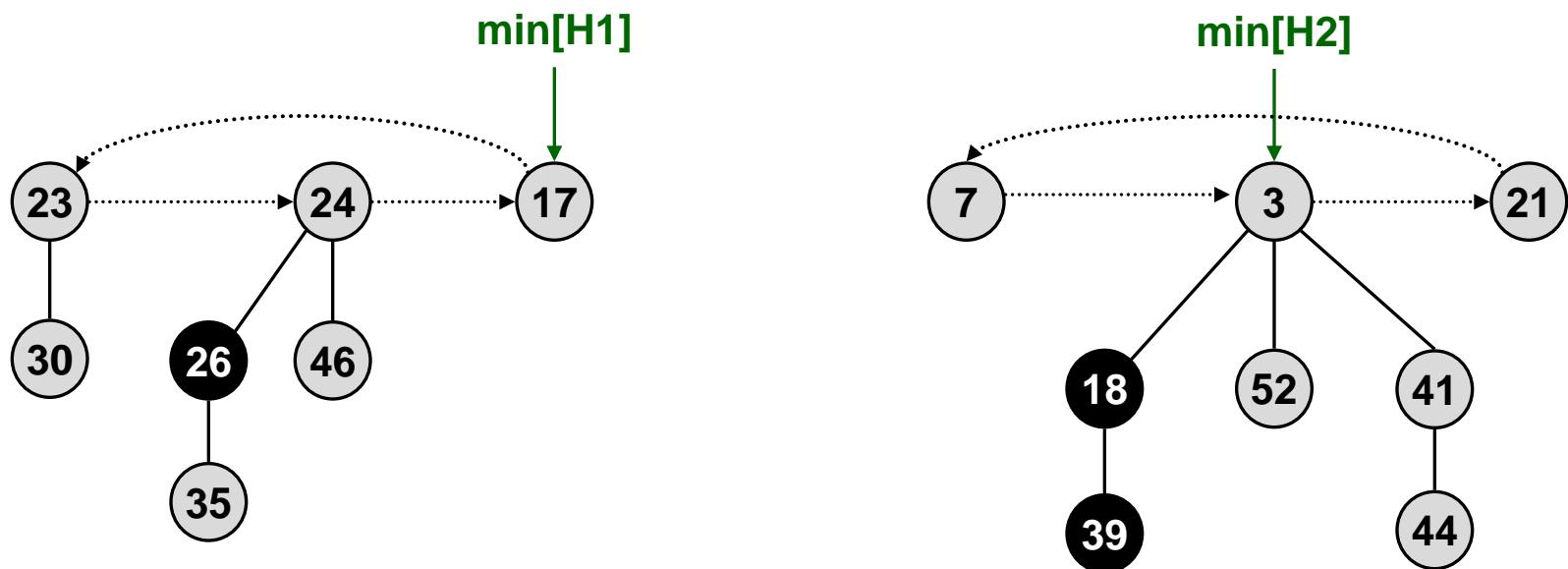


# Amortized Cost of Insert

- Actual Cost  $O(1)$
- Change in potential +1
  - One new tree, no new marked node
- Amortized cost  $O(1)$

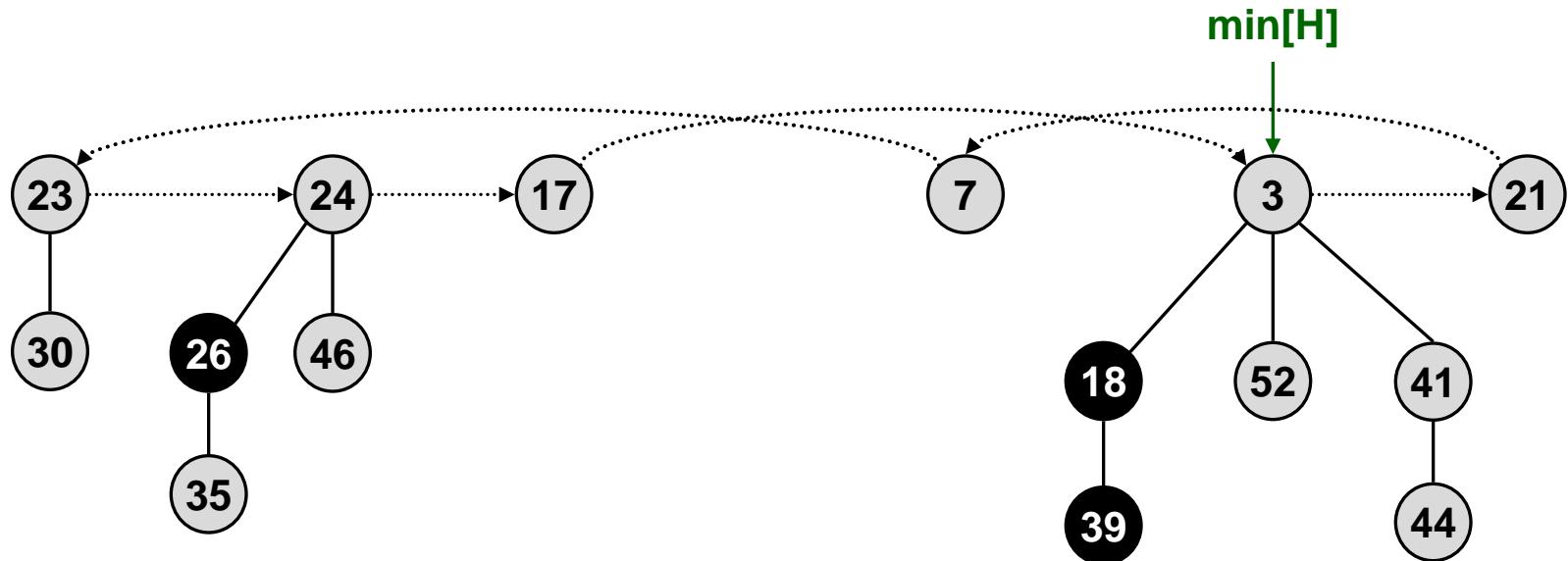
# Merging Two Heaps (Union)

- Concatenate the root lists of the two Fibonacci heaps
- Root lists are circular, doubly linked lists, so can be easily concatenated



# Merging Two Heaps (contd.)

- Concatenate the root lists of the two Fibonacci heaps
- Root lists are circular, doubly linked lists, so can be easily concatenated

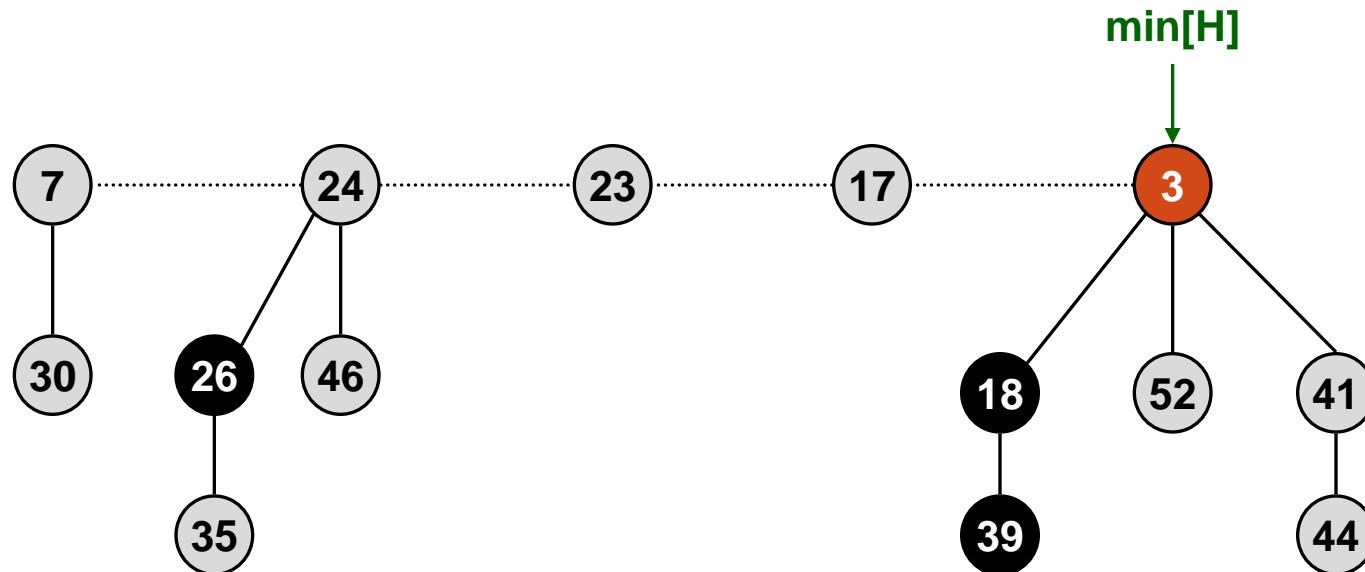


# Amortized Cost of Merge/Union

- Actual cost =  $O(1)$
- Change in potential = 0
- Amortized cost =  $O(1)$

# Extracting the Minimum Element

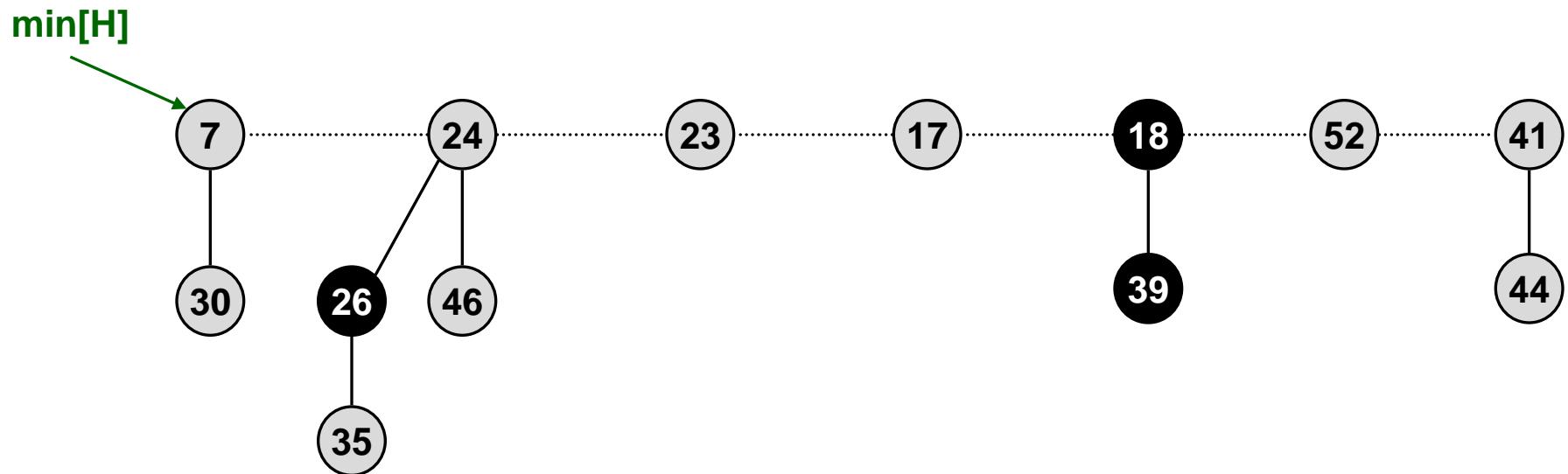
- Step 1:
  - Delete the node pointed to by  $\text{min}[H]$
  - Concatenate the deleted node's children into root list



# Extracting the Minimum (contd.)

- **Step 1:**

- Delete the node pointed to by  $\text{min}[H]$
- Concatenate the deleted node's children into root list

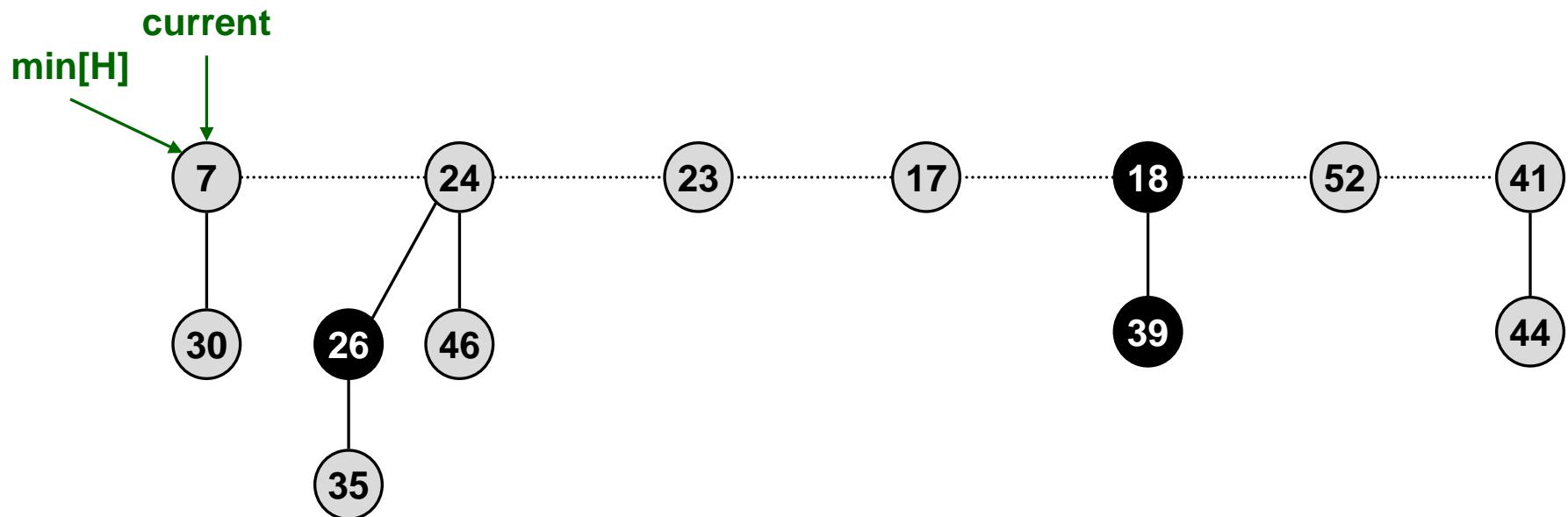


# Extracting the Minimum (contd.)

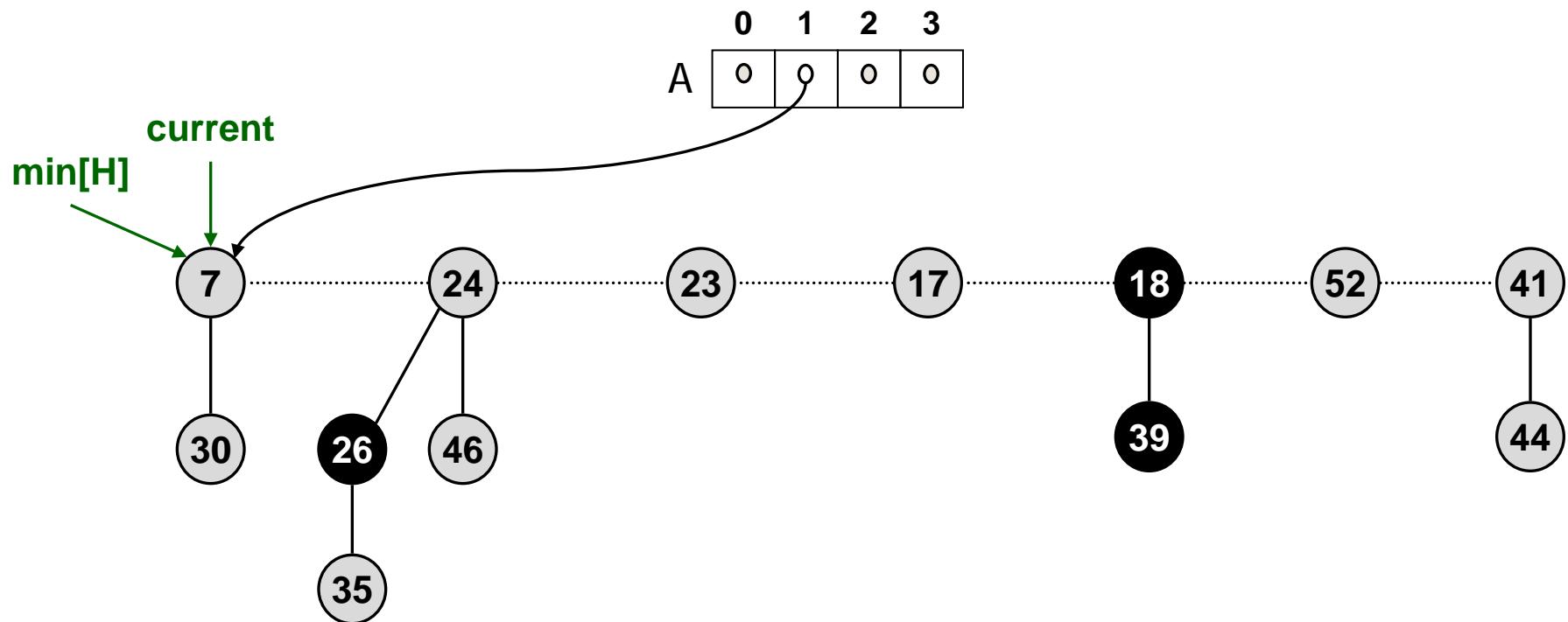
- **Step 2:** Consolidate trees so that no two roots have same degree
  - Traverse the roots from min towards right
  - Find two roots  $x$  and  $y$  with the same degree, with  $\text{key}[x] \leq \text{key}[y]$
  - Remove  $y$  from root list and make  $y$  a child of  $x$
  - Increment  $\text{degree}[x]$
  - Unmark  $y$  if marked
- We use an array  $A[0..D(n)]$  where  $D(n)$  is the maximum degree of any node in the heap with  $n$  nodes, initially all NIL
  - If  $A[k] = y$  at any time, then  $\text{degree}[y] = k$

# Extracting the Minimum (contd.)

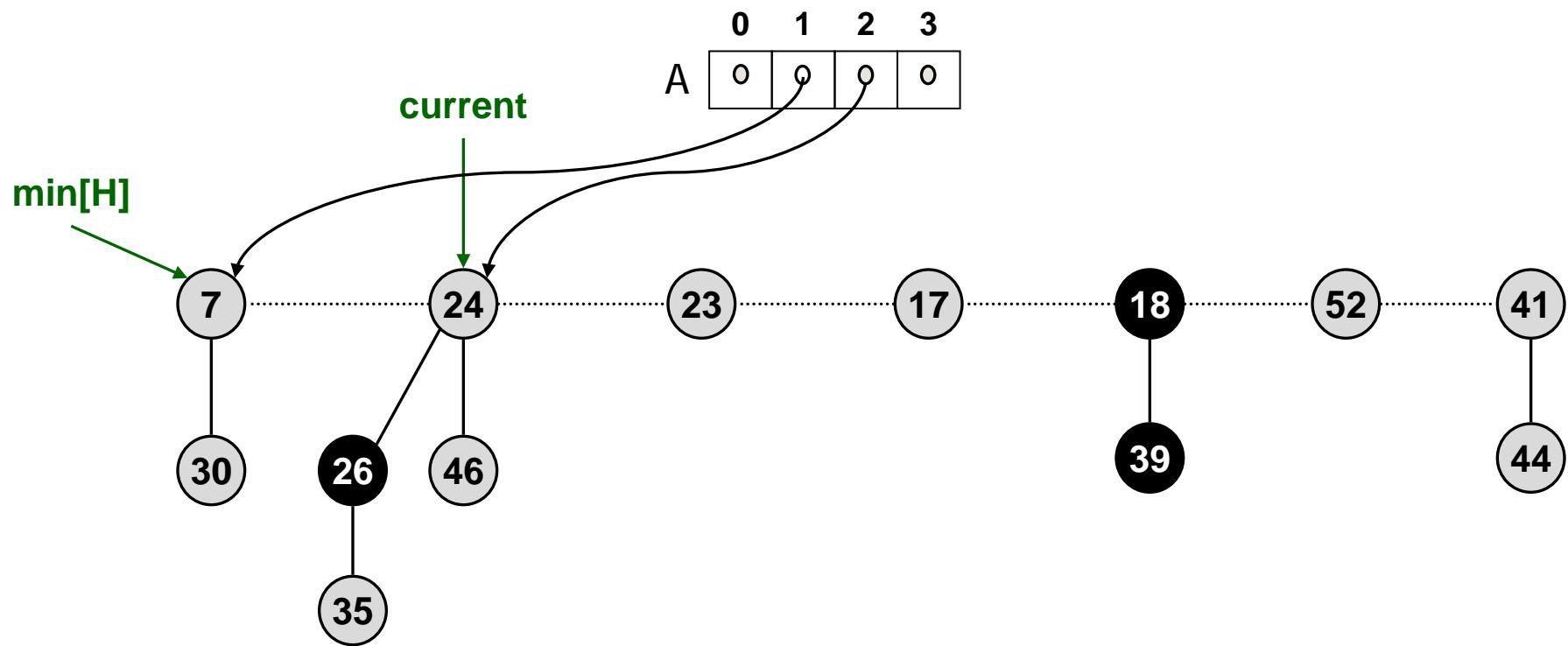
- **Step 2:** Consolidate trees so that no two roots have same degree. Update  $\text{min}[H]$  with the new min after consolidation.



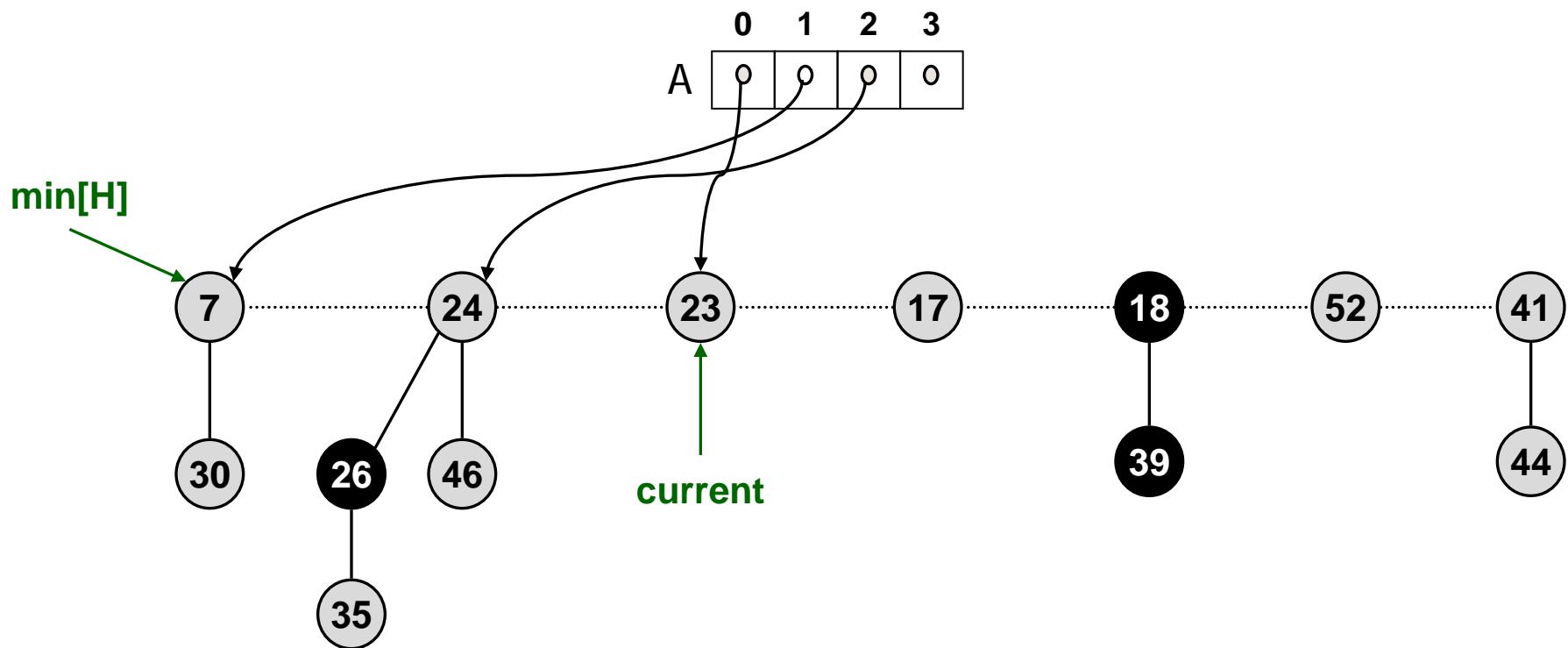
# Extracting the Minimum (contd.)



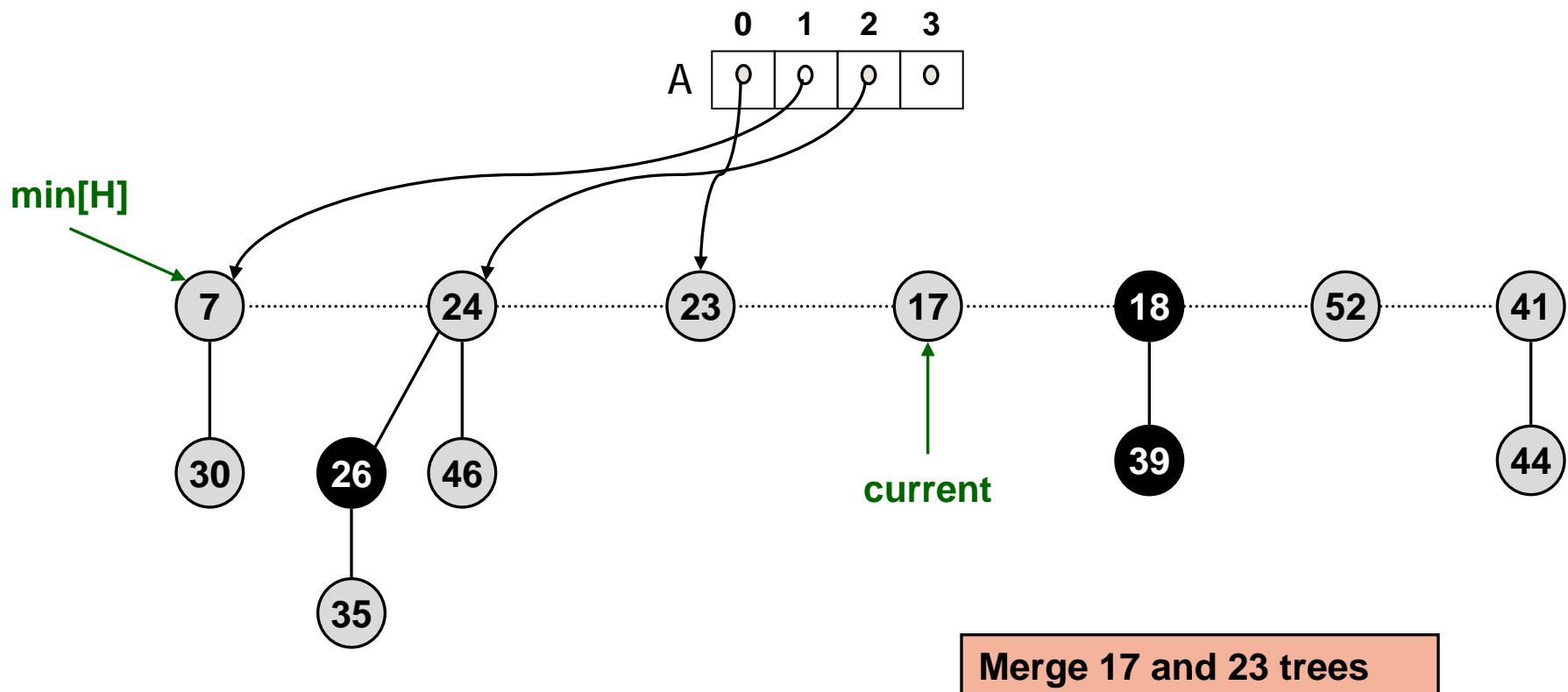
# Extracting the Minimum (contd.)



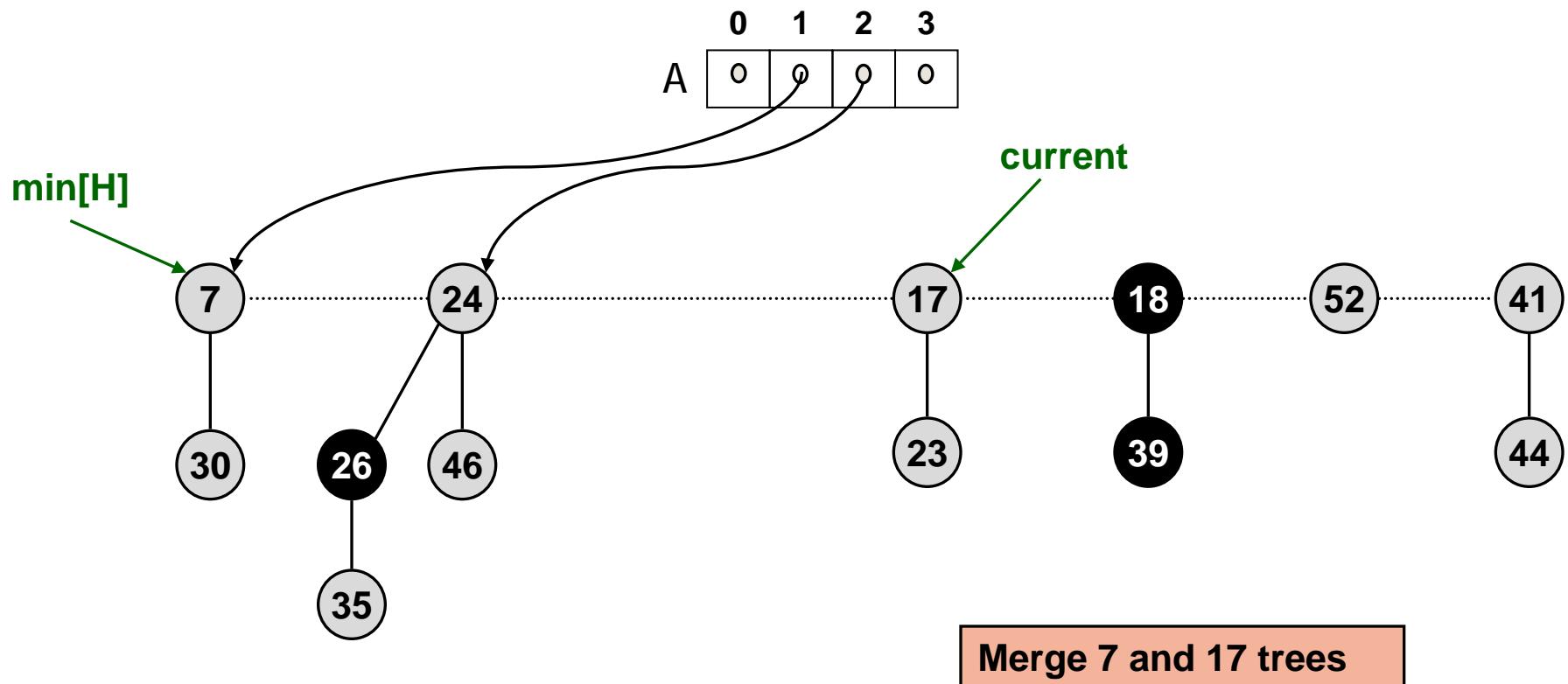
# Extracting the Minimum (contd.)



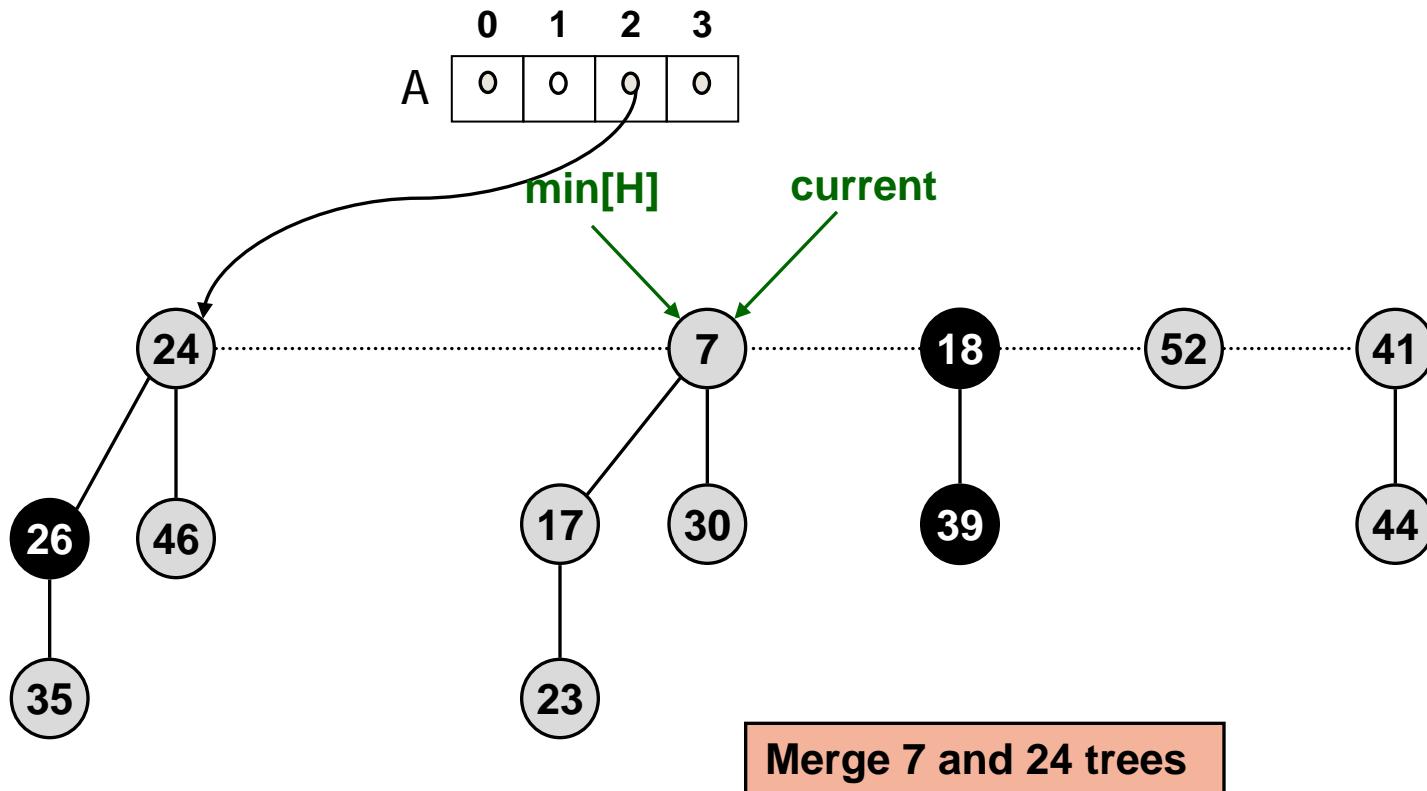
# Extracting the Minimum (contd.)



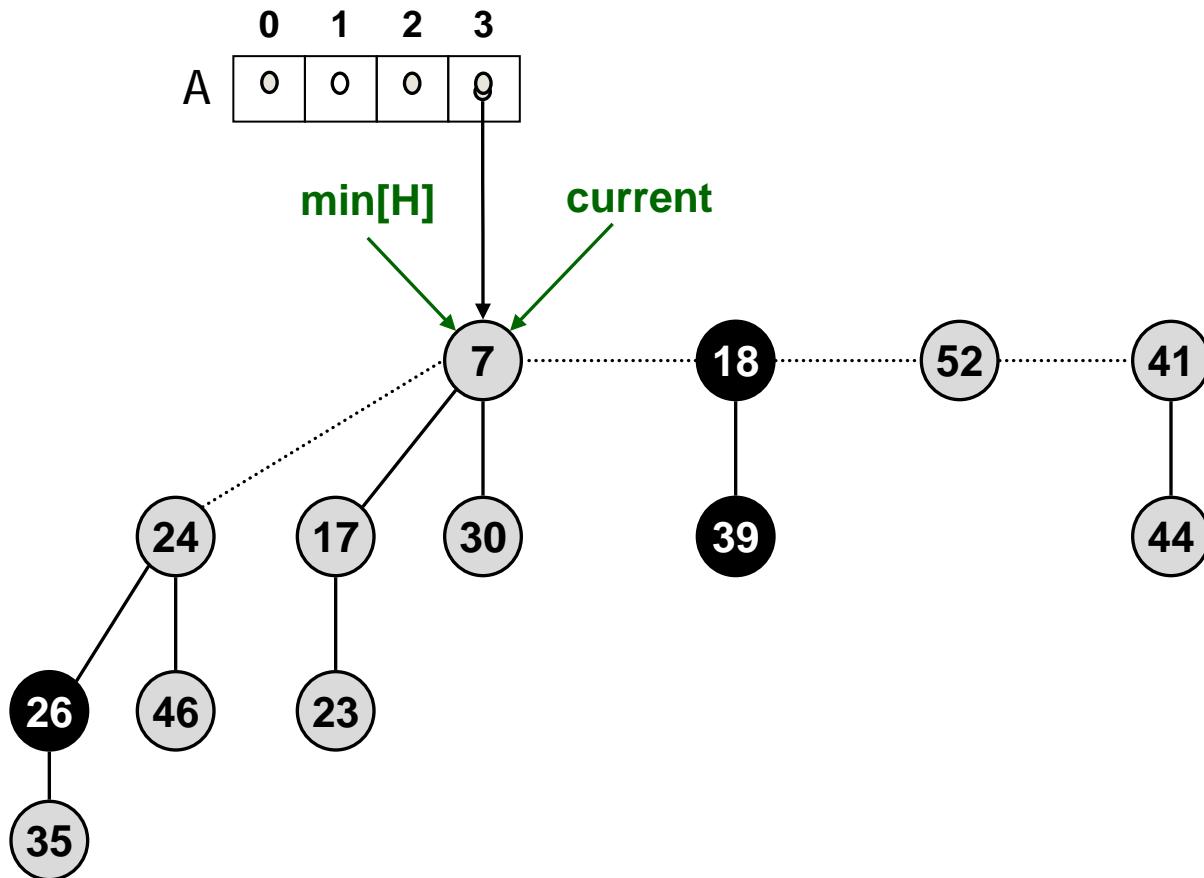
# Extracting the Minimum (contd.)



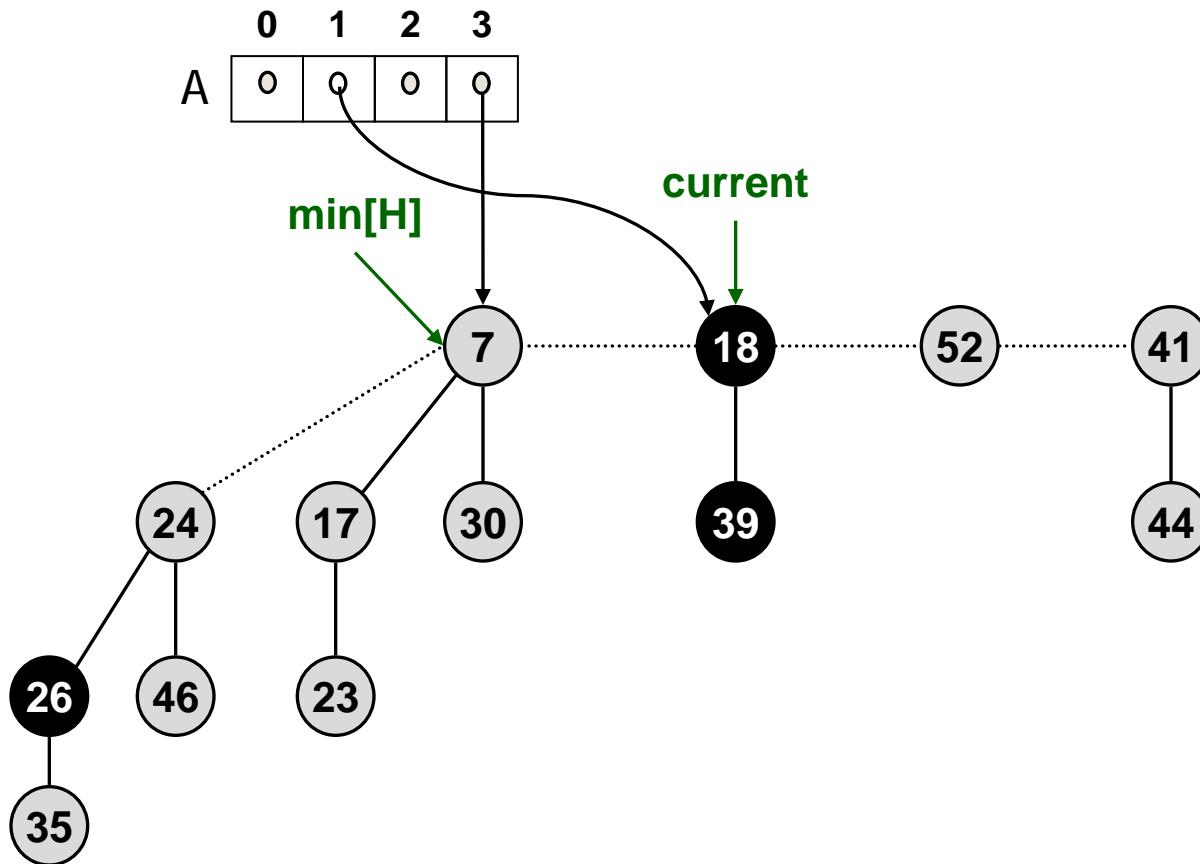
# Extracting the Minimum (contd.)



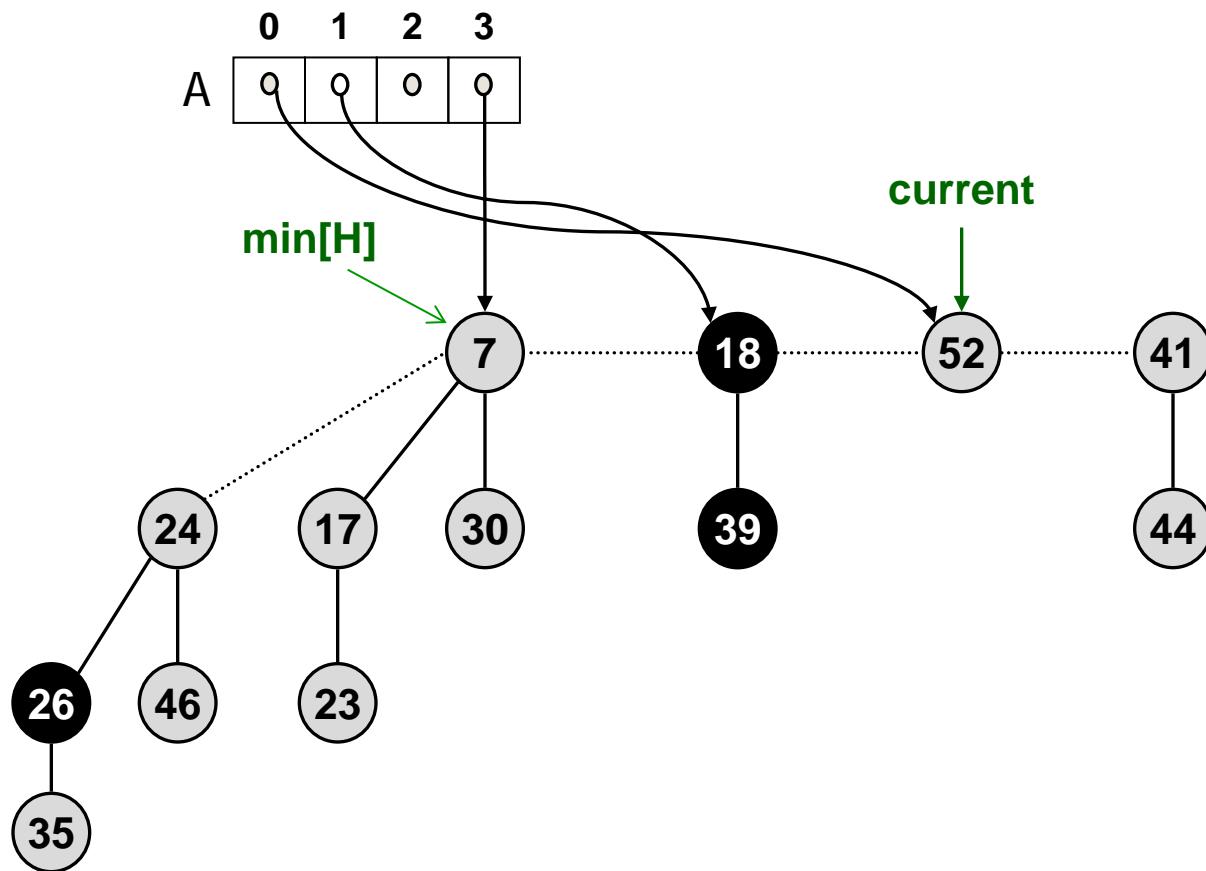
# Extracting the Minimum (contd.)



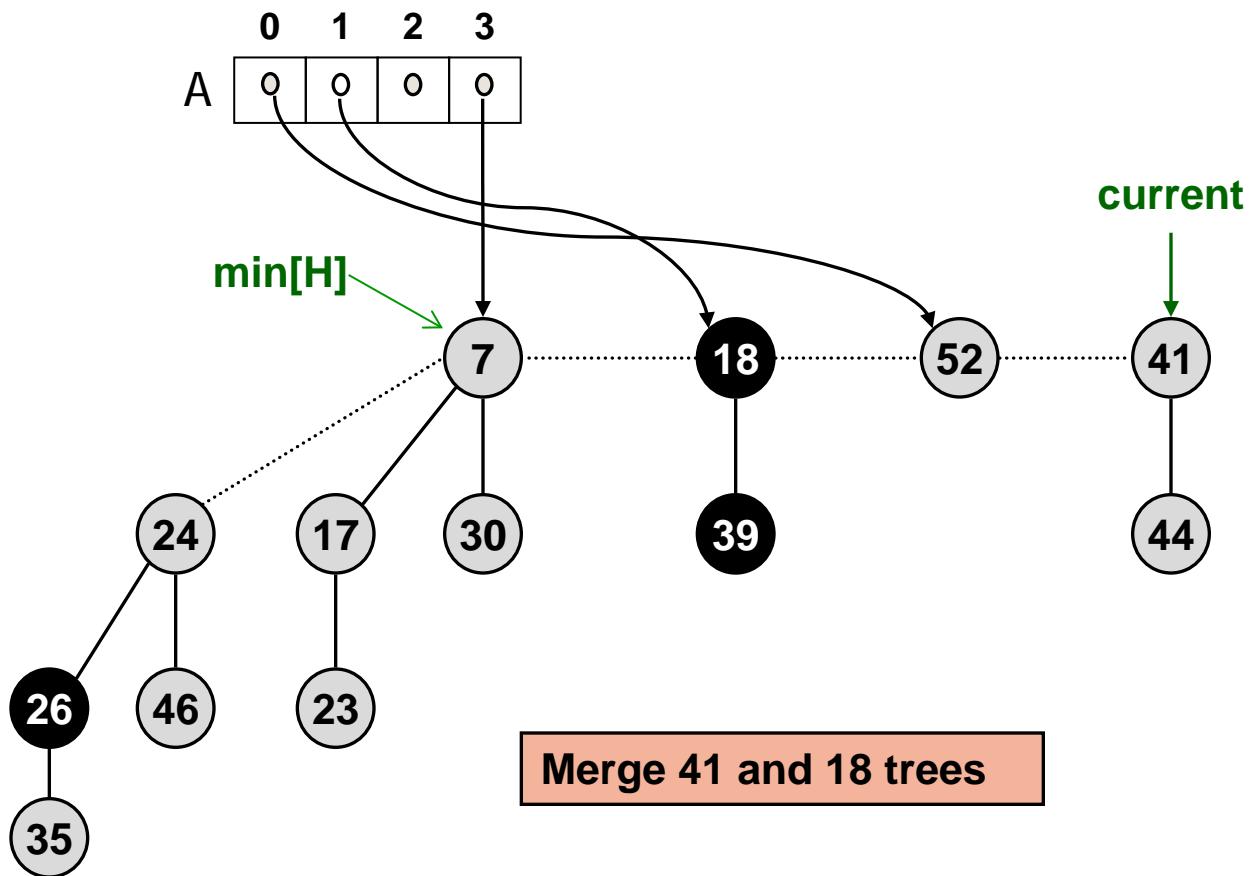
# Extracting the Minimum (contd.)



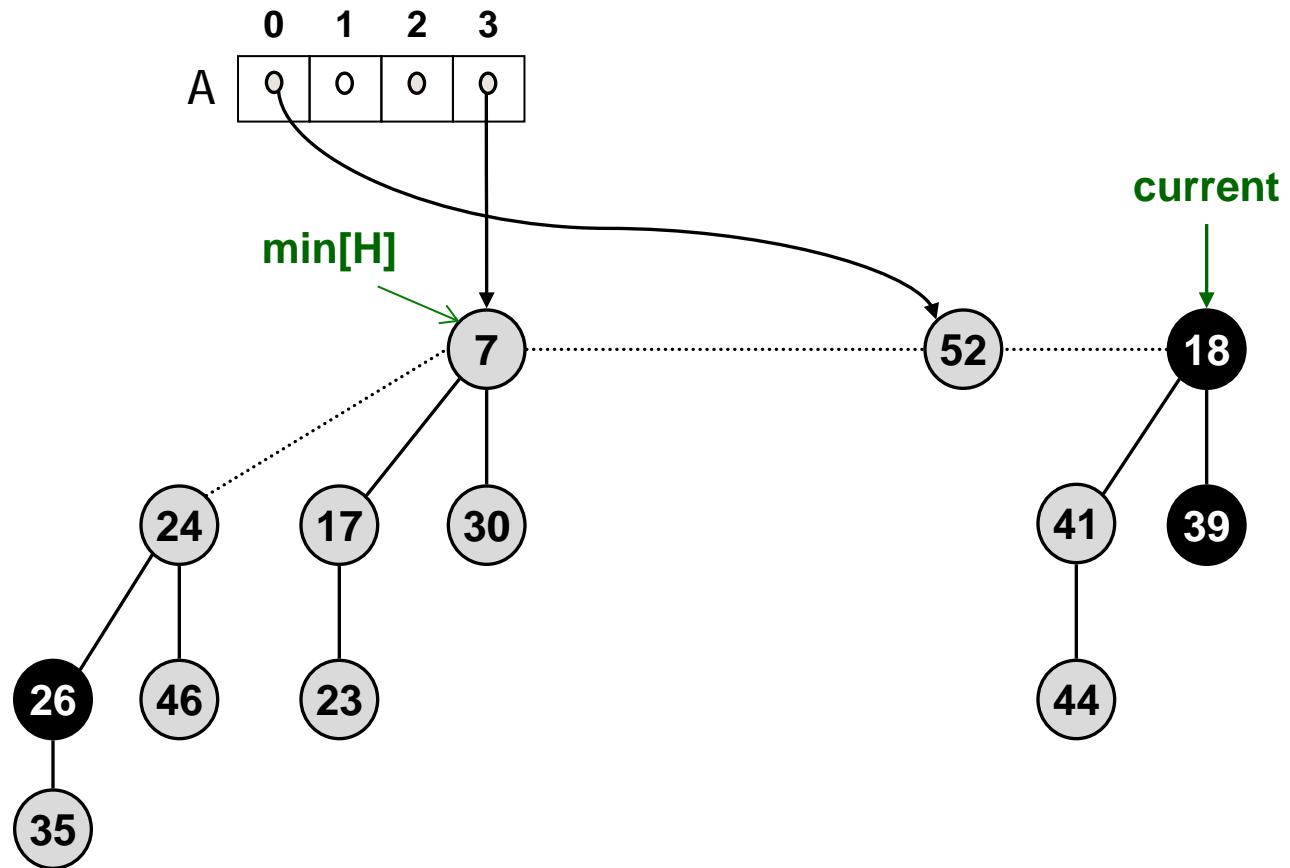
# Extracting the Minimum (contd.)



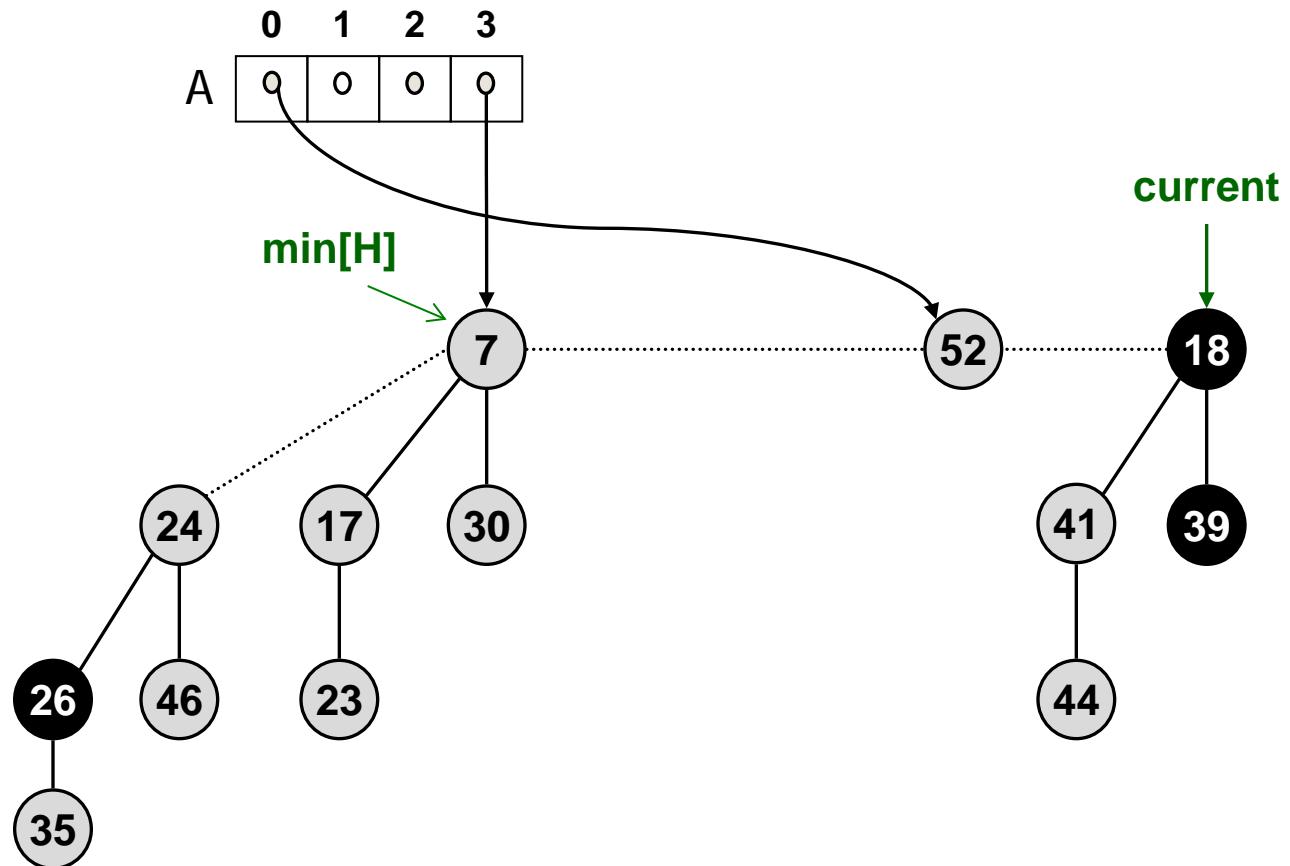
# Extracting the Minimum (contd.)



# Extracting the Minimum (contd.)

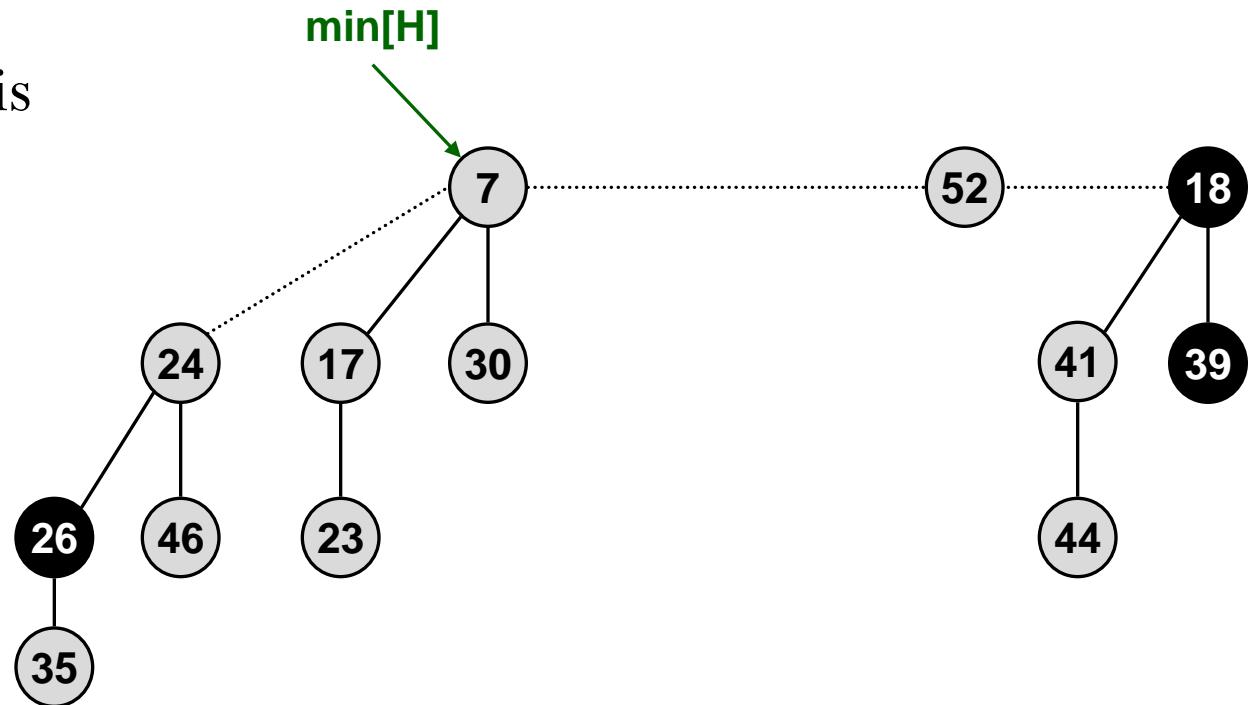


# Extracting the Minimum (contd.)



# Extracting the Minimum (contd.)

- All roots covered by current pointer, so done
- Now find the minimum among the roots and make  $\text{min[H]}$  point to it (already pointing to minimum in this example)
- Final heap is



# Amortized Cost of Extracting Min

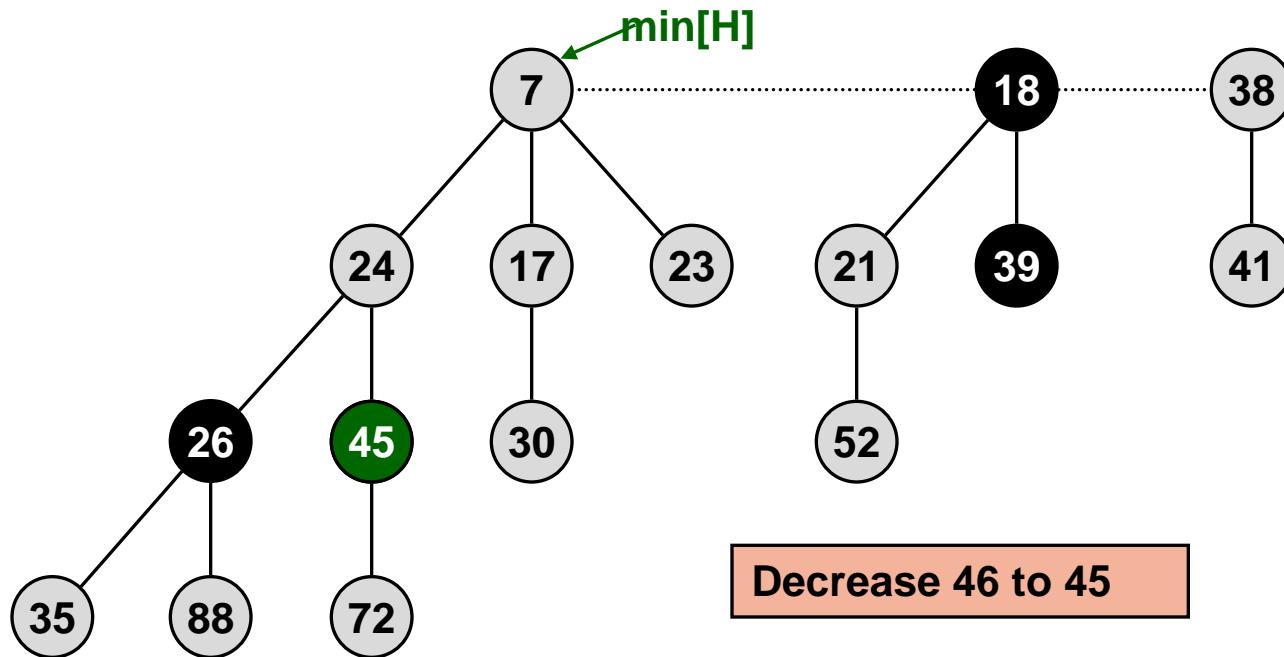
- Recall that
  - $D(n)$  = max degree of any node in the heap with  $n$  nodes
  - $t(H)$  = number of trees in heap  $H$
  - $m(H)$  = number of marked nodes in heap  $H$
  - Potential function  $\Phi(H) = t(H) + 2m(H)$
- Actual Cost
  - Time for Step 1:
    - $O(D(n))$  work adding min's children into root list

- Time for Step 2 (consolidating trees)
  - Size of root list just before Step 2 is  $\leq D(n) + t(H) - 1$ 
    - $t(H)$  original roots before deletion minus the one deleted plus the number of children of the deleted node
  - The maximum number of merges possible is the no. of nodes in the root list
  - Each merge takes  $O(1)$  time
  - So total  $O(D(n) + t(H))$  time for consolidation
  - $O(D(n))$  time to find the new min and updating  $\min[H]$  after consolidation, since at most  $D(n) + 1$  nodes in root list
- Total actual cost = time for Step 1 + time for Step 2
 
$$= O(D(n) + t(H))$$

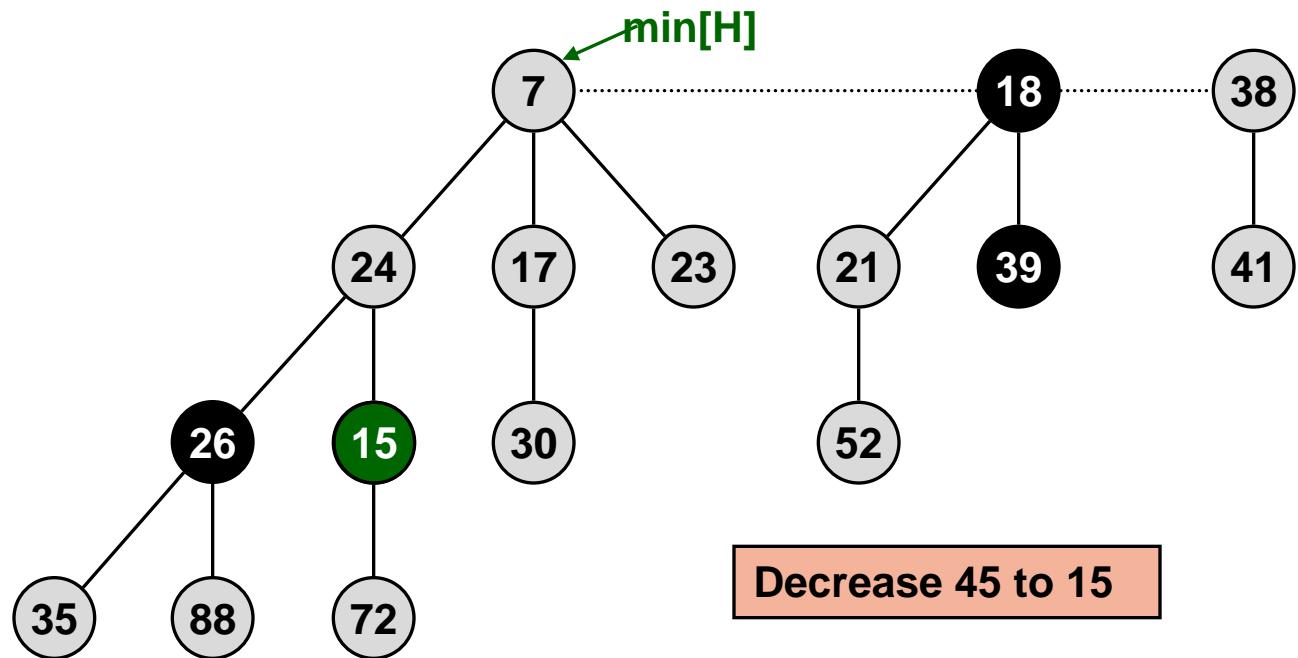
- Potential before extracting minimum =  $t(H) + 2m(H)$
- Potential after extracting minimum  $\leq (D(n) + 1) + 2m(H)$ 
  - At most  $D(n) + 1$  roots are there after deletion
  - No new node is marked during deletion
    - Can be unmarked, but not marked
- Amortized cost = actual cost + potential change
 
$$\begin{aligned} &= O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) \end{aligned}$$
- But  $D(n)$  can be  $O(n)$ , right? That seems too costly! So is  $O(D(n))$  any good?
  - Can show that  $D(n) = O(\lg n)$  (proof omitted)
  - So amortized cost =  $O(\lg n)$

# Decrease Key

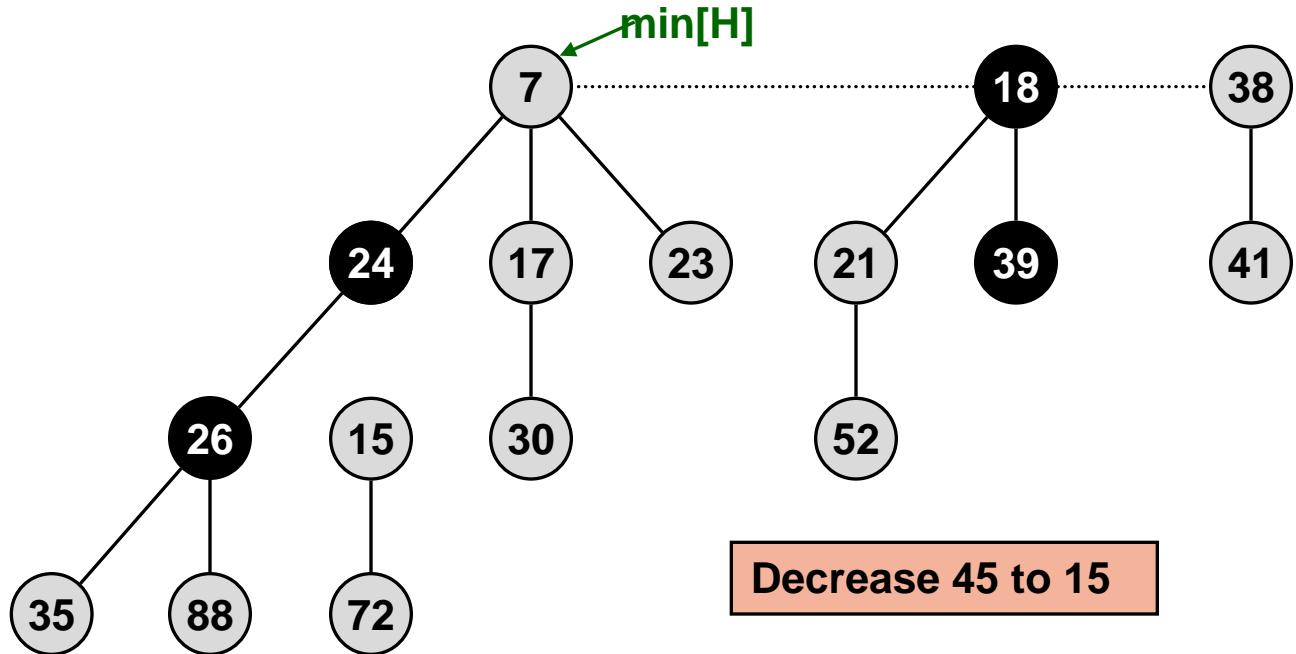
- Decrease key of element x to k
- Case 0: min-heap property not violated
  - decrease key of x to k
  - change heap min pointer if necessary



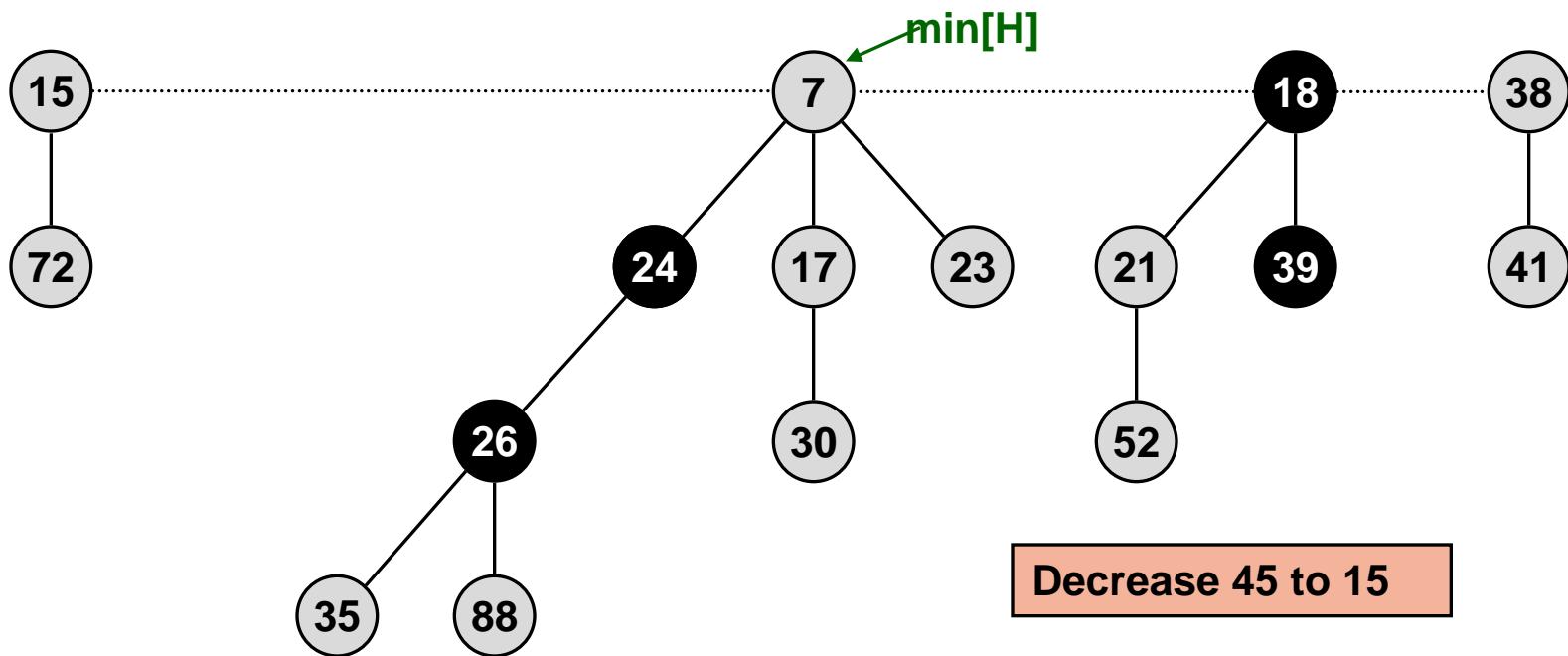
- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer



- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer

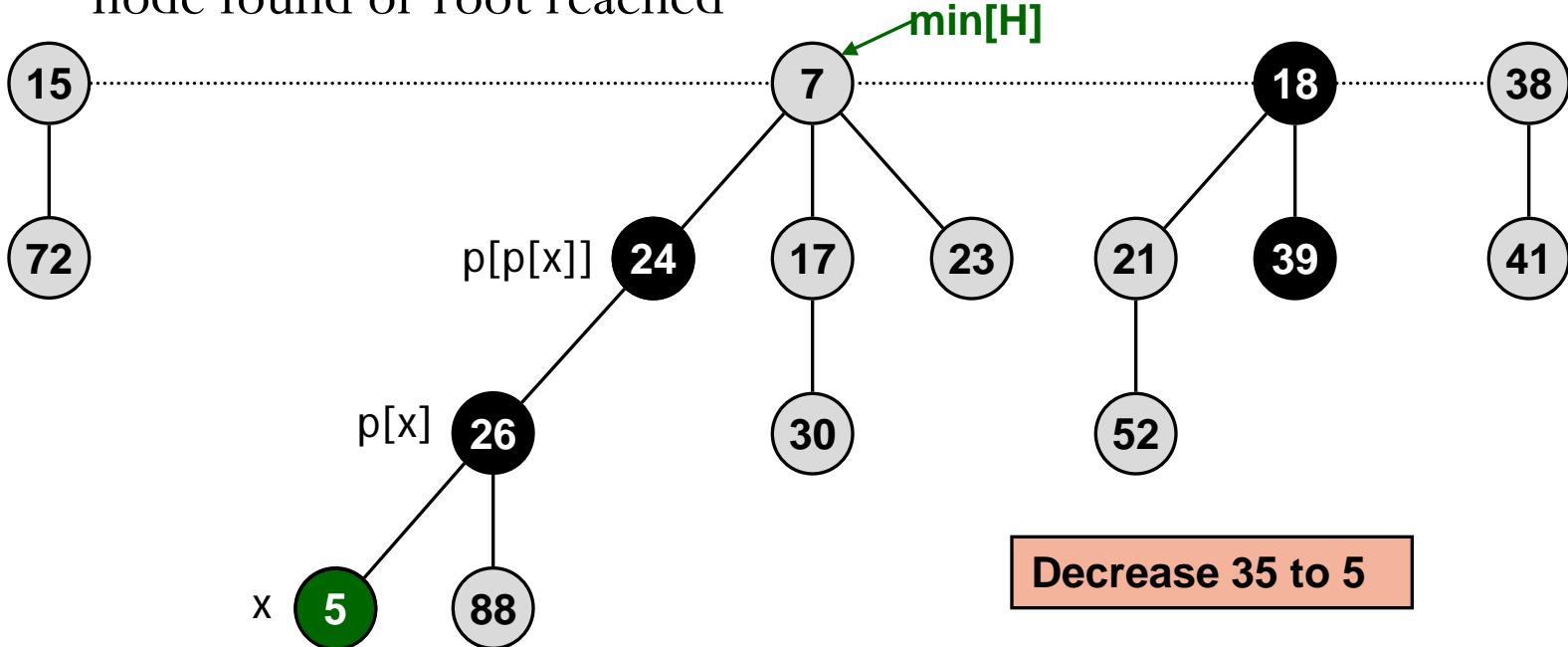


- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer



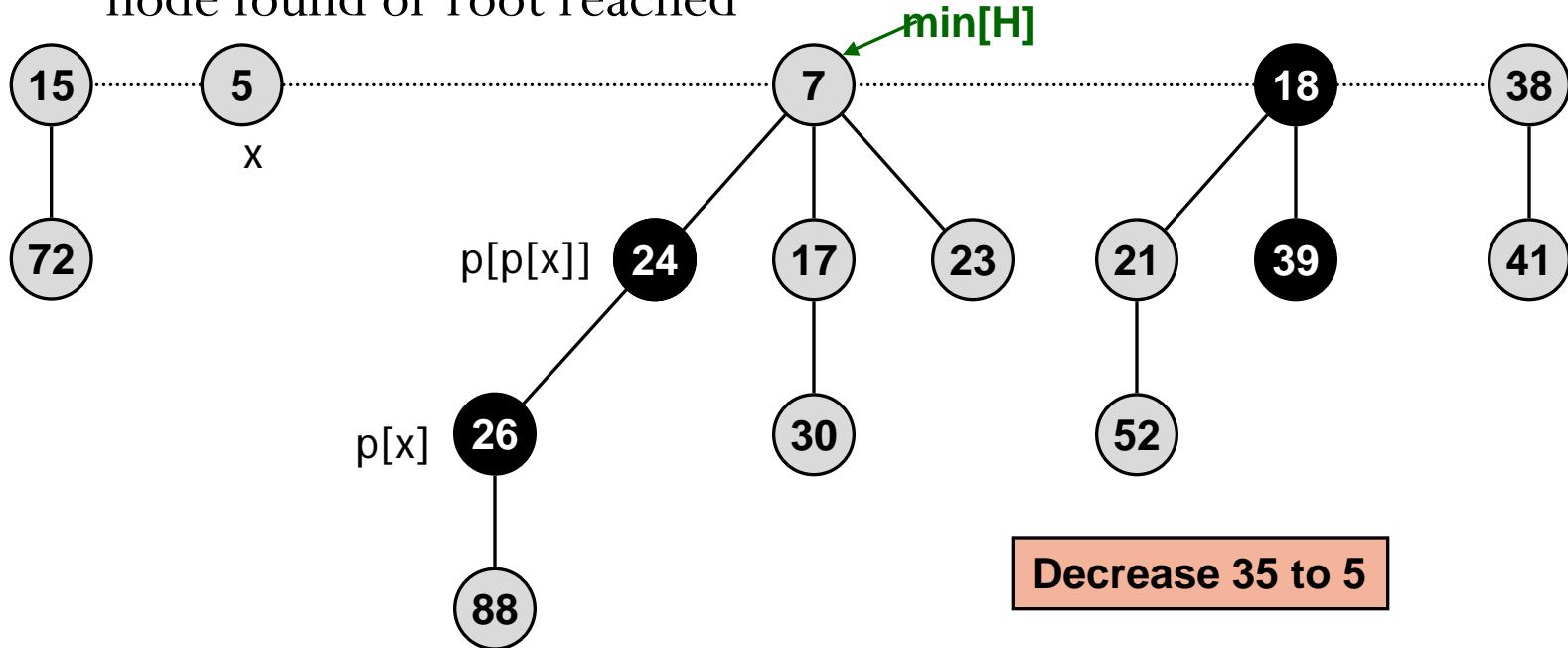
- Case 2: parent of  $x$  is marked

- decrease key of  $x$  to  $k$
- cut off link between  $x$  and its parent  $p[x]$ , add  $x$  to root list, unmark  $x$  if marked
- cut off link between  $p[x]$  and  $p[p[x]]$ , add  $p[x]$  to root list, unmark  $p[x]$  if marked
  - If  $p[p[x]]$  unmarked, then mark it and stop
  - If  $p[p[x]]$  marked, cut off  $p[p[x]]$ , unmark, and repeat until unmarked node found or root reached



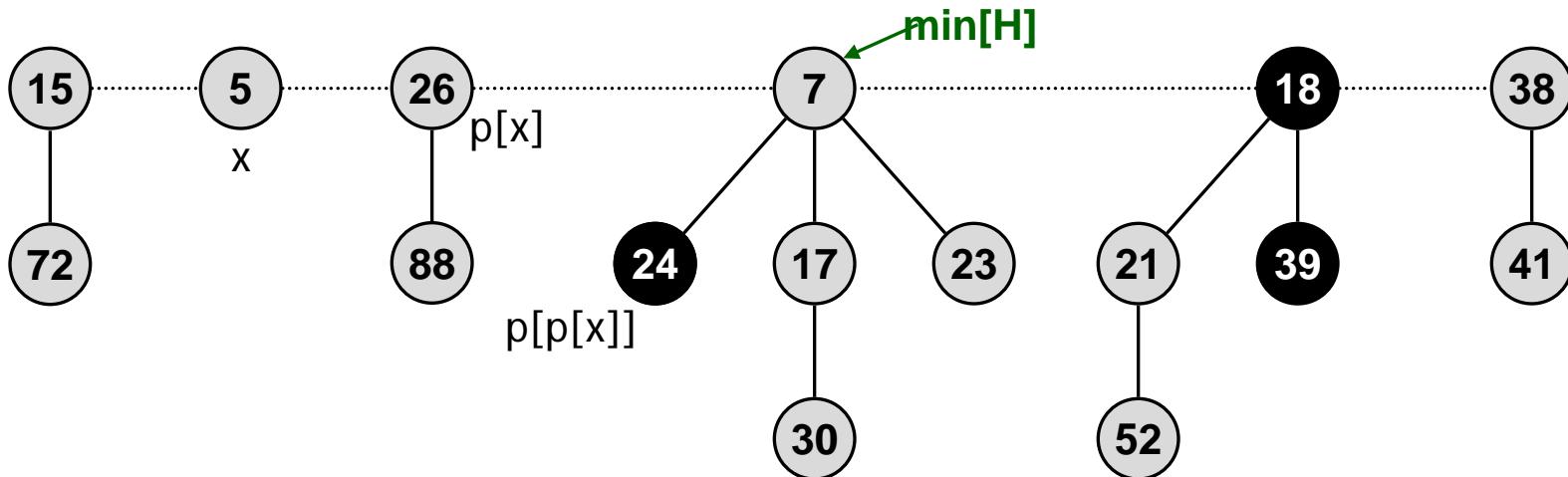
- Case 2: parent of x is marked

- decrease key of x to k
- cut off link between x and its parent  $p[x]$ , add x to root list, unmark x if marked
- cut off link between  $p[x]$  and  $p[p[x]]$ , add  $p[x]$  to root list, unmark  $p[x]$  if marked
  - If  $p[p[x]]$  unmarked, then mark it and stop
  - If  $p[p[x]]$  marked, cut off  $p[p[x]]$ , unmark, and repeat until unmarked node found or root reached



- Case 2: parent of x is marked

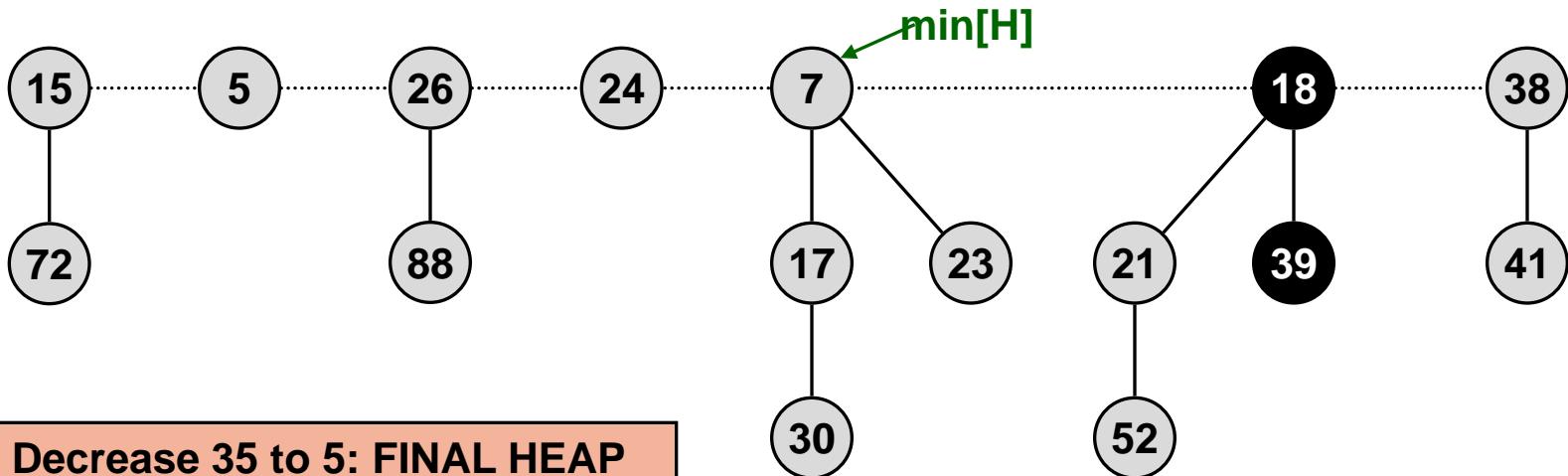
- decrease key of x to k
- cut off link between x and its parent  $p[x]$ , add x to root list, unmark x if marked
- cut off link between  $p[x]$  and  $p[p[x]]$ , add  $p[x]$  to root list, unmark  $p[x]$  if marked
  - If  $p[p[x]]$  unmarked, then mark it and stop
  - If  $p[p[x]]$  marked, cut off  $p[p[x]]$ , unmark, and repeat until unmarked node found or root reached



Decrease 35 to 5

- Case 2: parent of x is marked

- decrease key of x to k
- cut off link between x and its parent  $p[x]$ , add x to root list, unmark x if marked
- cut off link between  $p[x]$  and  $p[p[x]]$ , add  $p[x]$  to root list, unmark  $p[x]$  if marked
  - If  $p[p[x]]$  unmarked, then mark it and stop
  - If  $p[p[x]]$  marked, cut off  $p[p[x]]$ , unmark, and repeat until unmarked node found or root reached (cascading cut)



Fib-Heap-Decrease-key( $H$ ,  $x$ ,  $k$ )

1. if  $k > \text{key}[x]$
2.       error “new key is greater than current key”
3.  $\text{key}[x] = k$
4.  $y \leftarrow p[x]$
5. if  $y \neq \text{NIL}$  and  $\text{key}[x] < \text{key}[y]$
6.       { CUT( $H$ ,  $x$ ,  $y$ )
7.           CASCADING-CUT( $H$ ,  $y$ ) }
8. if  $\text{key}[x] < \text{key}[\text{min}[H]]$
9.        $\text{min}[H] = x$

$\text{CUT}(H, x, y)$

1. remove  $x$  from the child list of  $y$ , decrement  $\text{degree}[y]$
2. add  $x$  to the root list of  $H$
3.  $p[x] = \text{NIL}$
4.  $\text{mark}[x] = \text{FALSE}$

$\text{CASCADING-CUT}(H, y)$

1.  $z \leftarrow p[y]$
2. if  $z \neq \text{NIL}$ 
  3. if  $\text{mark}[y] = \text{FALSE}$ 
    4.  $\text{mark}[y] = \text{TRUE}$
    5. else  $\text{CUT}(H, y, z)$
    6.  $\text{CASCADING-CUT}(H, z)$

# Amortized Cost of Decrease Key

- Actual cost
  - $O(1)$  time for decreasing key value, and the first cut of  $x$
  - $O(1)$  time for each of  $c$  cascading cuts, plus reinserting in root list
  - Total  $O(c)$
- Change in Potential
  - $H$  = tree just before decreasing key,  $H'$  just after
  - $t(H') = t(H) + c$ 
    - $t(H) + (c-1)$  trees from the cascading cut + the tree rotted at  $x$
  - $m(H') \leq m(H) - c + 2$ 
    - Each cascading cut unmarks a node except the last one ( $-(c-1)$ )
    - Last cascading cut could potentially mark a node (+1)

- Change in potential

$$= (t(H') + 2m(H')) - (t(H) + 2m(H))$$

$$\leq c + 2(-c + 2) = 4 - c$$

- Amortized cost = actual cost + potential change

$$= O(c) + 4 - c = O(1)$$

# Deleting an Element

- Delete node  $x$ 
  - Decrease key of  $x$  to  $-\infty$
  - Delete min element in heap
- Amortized cost
  - $O(1)$  for decrease-key.
  - $O(D(n))$  for delete-min.
  - Total  $O(D(n))$ 
    - Again, can show that  $D(n) = O(\lg n)$
    - So amortized cost of delete =  $O(\lg n)$

## Data structures for Disjoint sets

MAIN IDEA: Group  $n$  distinct elements into a collection of disjoint sets; the following operations should be efficient:

- Finding the set to which a given element belongs.
- Uniting two sets.

## CONTENT OF THIS LECTURE

- ① The disjoint-set data structure + specific operations
- ② A simple application
- ③ Concrete implementations based on
  - linked lists
  - rooted trees
- ④ Discussion: the Ackermann function

# Disjoint-set data structure

## Main features

Container for a collection  $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$  of disjoint dynamic sets. ( $A, B$  are disjoint sets if  $A \cap B = \emptyset$ .)

- Each set is identified by some member of the set, called its **representative**
  - ▷ REQUIREMENT: If we ask for the representative of a dynamic set twice without modifying the set, we should get the same answer.

## DESIRABLE OPERATIONS

- ▷ **MAKESET( $x$ )**: creates a new set consisting of  $x$  only.  
(Requirement:  $x$  is not already in another set.)
- ▷ **UNION( $x, y$ )**: unites the sets that contain  $x$  and  $y$ , say  $S_x$  and  $S_y$ , into a new set that is their union. The sets  $S_x$  and  $S_y$  can be destroyed.
- ▷ **FINDSET( $x$ )**: returns a pointer to the representative of the unique set containing element  $x$ .

# Disjoint-set data structure

Application: Determining the connected components of an undirected graph

ASSUMPTION:  $G = (V, E)$  is an undirected graph.

- ➊ Computing the connected components of  $G$ :

**CONNECTEDCOMPONENTS( $G$ )**

- 1 for each node  $v \in V$
- 2 **MAKESET( $v$ )**
- 3 for each edge  $(u, v) \in E$
- 4 if **FINDSET( $u$ ) ≠ FINDSET( $v$ )**
- 5      **UNION( $u, v$ )**

- ➋ Determine if two elements are in the same component:

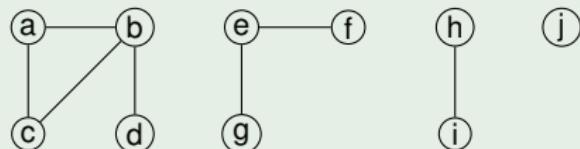
**SAMECOMPONENT( $u, v$ )**

- 1 if **FINDSET( $u$ ) = FINDSET( $v$ )**
- 2 return **TRUE**
- 3 return **FALSE**

# Disjoint-set data structure

Application: Determining the connected components of an undirected graph

Example (A graph with 4 connected components)



Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)		{a, b, c, d}			{e, g}	{f}		{h, i}		{j}
(e, f)		{a, b, c, d}			{e, f, g}			{h, i}		{j}
(b, c)		{a, b, c, d}			{e, f, g}			{h, i}		{j}

# Disjoint sets

A linked-list representation

## MAIN IDEAS

- Each set is represented by a linked list.
- The first element in each linked list is the representative of the set.
- Each object in the linked list contains
  - A pointer to the next set element
  - A pointer back to the set representative
- $\text{MAKESET}(x)$  and  $\text{FINDSET}(x)$  are straightforward to implement
  - They require  $O(1)$  time.

**Q1:** How to implement  $\text{UNION}(x, y)$ ?

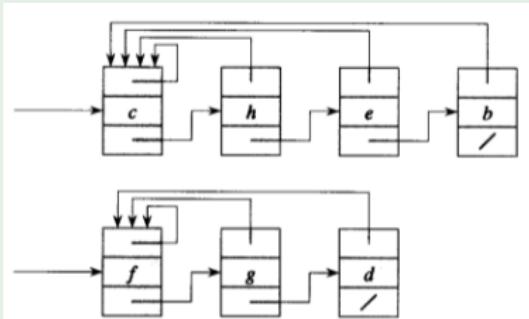
**Q2:** What is the time complexity of  $\text{UNION}(x, y)$ ?

# Disjoint sets

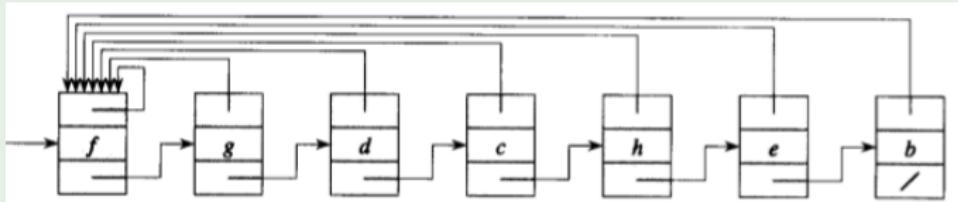
A linked-list representation

## Example

- 1 Linked-list representations of sets  $\{b, c, h, e\}$  and  $\{d, f, g\}$



- 2 Linked-list representation of their union



# Disjoint sets

A linked-list representation

## Implementation of UNION( $x, y$ )

- Append  $x$ -s list onto the end of  $y$ -s list and update all elements from  $x$ -s list to point to the representative of the set containing  $y$   
⇒ time linear in the length of  $x$ -s list.

# Disjoint sets

A linked-list representation

## Implementation of UNION( $x, y$ )

- Append  $x$ -s list onto the end of  $y$ -s list and update all elements from  $x$ -s list to point to the representative of the set containing  $y$   
⇒ time linear in the length of  $x$ -s list.

Some sequences of  $m$  operations may require  $\Theta(m^2)$  time (see next slide)

# Disjoint sets represented by linked lists

## Example

A sequence of  $m$  operations that takes  $\Theta(m^2)$  time

Operation	Number of objects updated
MAKESET( $x_1$ )	1
MAKESET( $x_2$ )	1
$\vdots$	$\vdots$
MAKESET( $x_q$ )	1
UNION( $x_1, x_2$ )	1
UNION( $x_2, x_3$ )	2
UNION( $x_3, x_4$ )	3
$\vdots$	$\vdots$
UNION( $x_{q-1}, x_q$ )	$q - 1$

The number of MAKESET ops. is  $n = \lceil m/2 \rceil + 1$ , and  $q = m - n$ .

Total time spent:  $\Theta(n + q^2) = \Theta(m^2)$  because  $n = \Theta(m)$  and  $q = \Theta(m) \Rightarrow$  amortized time of an operation is  $\Theta(m)$ .

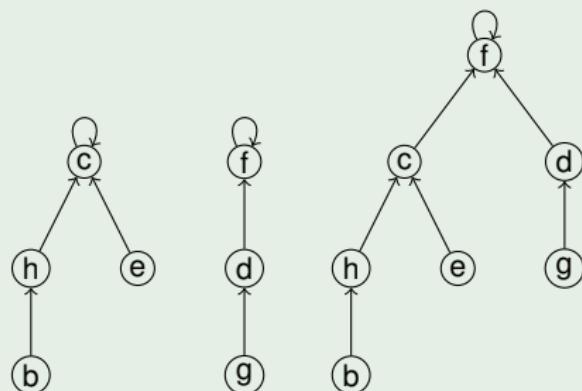
# Towards a faster implementation

## Disjoint-set forests

MAIN IDEA: Represent sets by rooted trees, with each node containing one member and each tree representing one set.

- A **disjoint-set forest** is a set of rooted trees, where each member points only to its parent.

### Example



# Towards a faster implementation

## Disjoint-set forests

Implementation of disjoint set operations:

- **MAKESET( $x$ )**: creates a tree with just one node.
- **FINDSET( $x$ )**: follows the parent pointers from a node until it reaches the root of the tree.
  - The nodes visited on the path towards the root constitute the **find path**.
- **UNION( $x, y$ )**: causes the root of one tree to point to the root of the other tree.

### Remarks

- ① A sequence of  $n$  UNION operations may create a tree which is just a linear chain of nodes
  - ⇒ Disjoint-set forests have not improved the linked list representation.
- ② We need 2 more heuristically improvements: **union by rank** and **path compression**.



# Disjoint-set forests

Heuristic 1: union by rank

## Implementation of $\text{UNION}(x, y)$

- MAIN IDEA: make the root of the tree with fewer nodes point to the root of the tree with more nodes.
  - Each node has a **rank** that approximates the logarithm of the size of the subtree rooted at each node and also an upper bound of the height of the node.
  - ⇒ perform union by rank: the root with smaller rank is made to point to the root with larger rank during the operation  $\text{UNION}(x, y)$ .

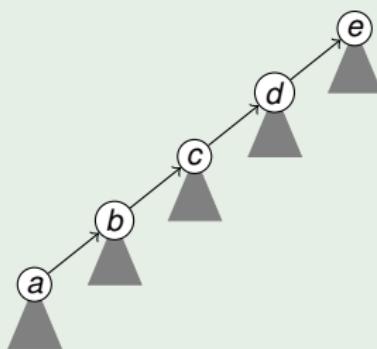
# Disjoint-set forests

Heuristic 2: path compression

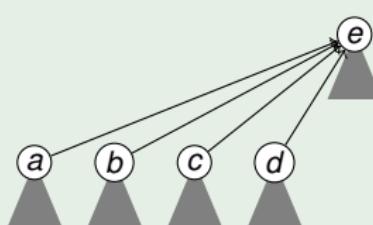
MAIN IDEA: During FINDSET operations, each node on the find path will be made to point directly to the root.

## Example

Path compression during the operation FINDSET( $a$ ).



(a) before executing FINDSET( $a$ )



(b) after executing FINDSET( $a$ )

# Disjoint-set forests

Pseudocode for main operations (1)

- With each node  $x$ , we maintain the `int` value  $x.rank$  which is an upper bound on the height of  $x$  (the number of edges on the longest path between  $x$  and a descendant leaf). The initial rank of a node in a newly created singleton tree is 0.

**MAKESET( $x$ )**

1.  $x.p = x$
2.  $x.rank = 0$

**UNION( $x, y$ )**

1. **LINK(FINDSET( $x$ ), FINDSET( $y$ ))**

**LINK( $x, y$ )**

- 1 if  $x.rank > y.rank$
- 2    $y.p = x$
- 3 else  $x.p = y$
- 4       if  $x.rank == y.rank$
- 5            $y.rank = y.rank + 1$

# Disjoint-set forests

Pseudocode for `FINDSET(x)`

`FINDSET` is a **two-pass method**:

- ➊ It makes one pass up the find path to find the root
- ➋ it makes a second pass back down the path to update each node so that it points directly to the root.

`FINDSET(x)`

```
1 if  $x \neq x.p$ 
2    $x.p = \text{FINDSET}(x.p)$ 
3 return  $x.p$ 
```

- ▷ Each call of `FINDSET(x)` returns  $x.p$  in line 3.
- ▷ If  $x$  is the root then line 2 is not executed and  $p[x] = x$  is returned.
  - This is the case when recursion bottoms out.
- ▷ Otherwise, line 2 is executed and the recursive call with parameter  $x.p$  returns (a pointer to) the root.
- ▷ Line 2 updates  $x$  to point directly to the root.

# Disjoint-set forests

Effect of heuristics on running time

## ASSUMPTIONS:

$n$  = number of MAKESET operations,

$m$  = total number of MAKESET, UNION and FINDSET operations.

- Union by rank has time complexity  $O(m \log n)$  [Cormen *et al.*, 2000]
- When we use both path compression and union by rank, the operations have worst-case time complexity  $O(m \cdot \alpha(m, n))$  where  $\alpha(m, n)$  is the *very slowly growing inverse of Ackermann's function* (see next slides.)
  - On all practical applications of a disjoint-set data structure,  $\alpha(m, n) \leq 4$ .
  - $\Rightarrow$  we can view the running time as linear in  $m$  in all practical situations.

# Ackermann's function and its inverse

## Preliminary notions

- Let  $g : \mathbb{N} \rightarrow \mathbb{N}$  be the function defined recursively by

$$g(i) = \begin{cases} 2^1 & \text{if } i = 0, \\ 2^2 & \text{if } i = 1, \\ 2^{g(i-1)} & \text{if } i > 1. \end{cases}$$

INTUITION:  $i$  gives the *height* of the stack of 2s that make up the exponent.

- For all  $i \in \mathbb{N}$  we define

$$\lg^{(i)}(n) = \begin{cases} n & \text{if } i = 0, \\ \lg(\lg^{(i-1)}(n)) & \text{if } i > 0 \text{ and } \lg^{(i-1)}(n) > 0, \\ \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)}(n) \leq 0 \text{ or } \lg^{(i-1)}(n) \text{ is undefined.} \end{cases}$$

where  $\lg$  stands for  $\log_2$

- $\lg^*(n) = \min\{i \geq 0 \mid \lg^{(i)}(n) \leq 1\}.$

REMARK:  $\lg^*(2^{g(n)}) = n + 1.$

# Skip Lists

# Linked Lists Benefits & Drawbacks

- Benefits:

- Easy to insert & delete in  $O(1)$  time
- Don't need to estimate total memory needed

- Drawbacks:

- Hard to search in less than  $O(n)$  time  
(binary search doesn't work, eg.)
- Hard to jump to the middle

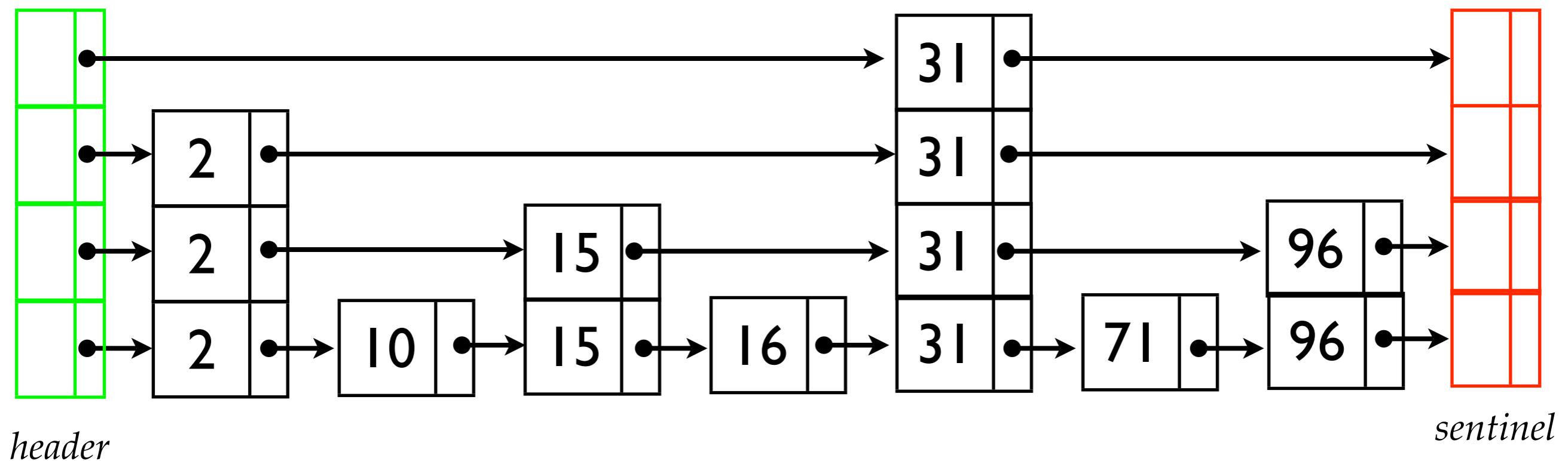
- Skip Lists:

- fix these drawbacks
- good data structure for a dictionary ADT

# Skip Lists

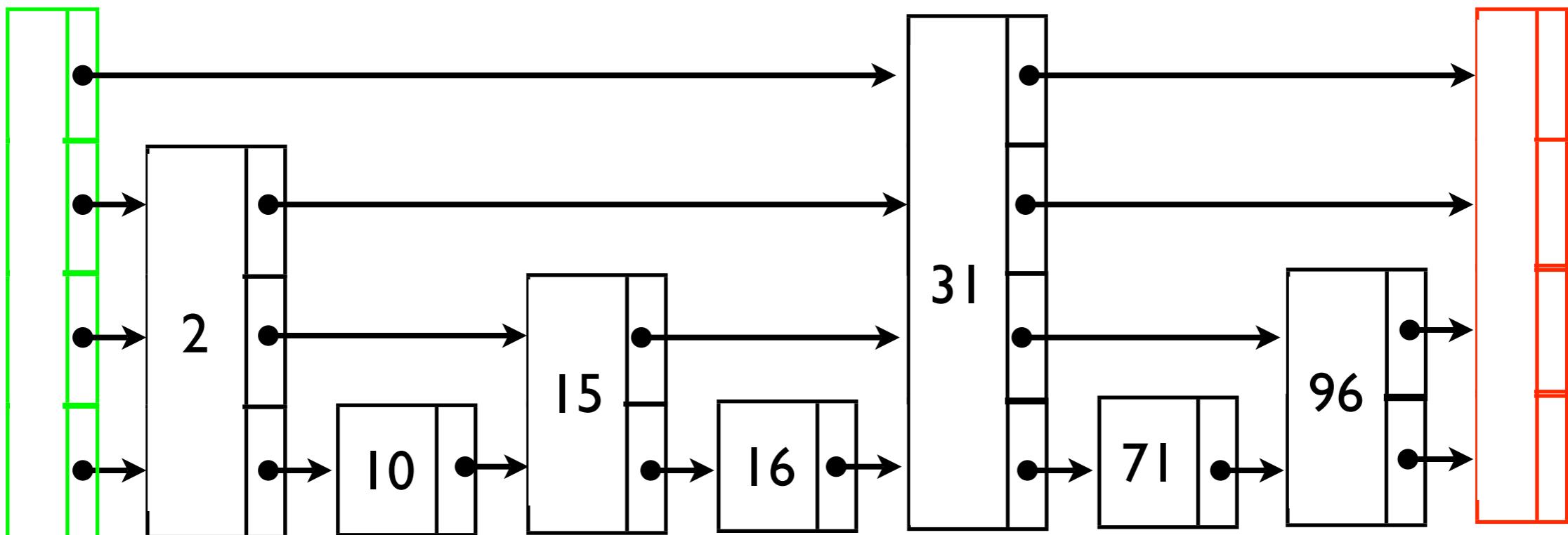
- Invented around 1990 by Bill Pugh
- Generalization of sorted linked lists – so simple to implement
- Expected search time is  $O(\log n)$
- *Randomized* data structure:
  - use random coin flips to build the data structure

# Perfect Skip Lists



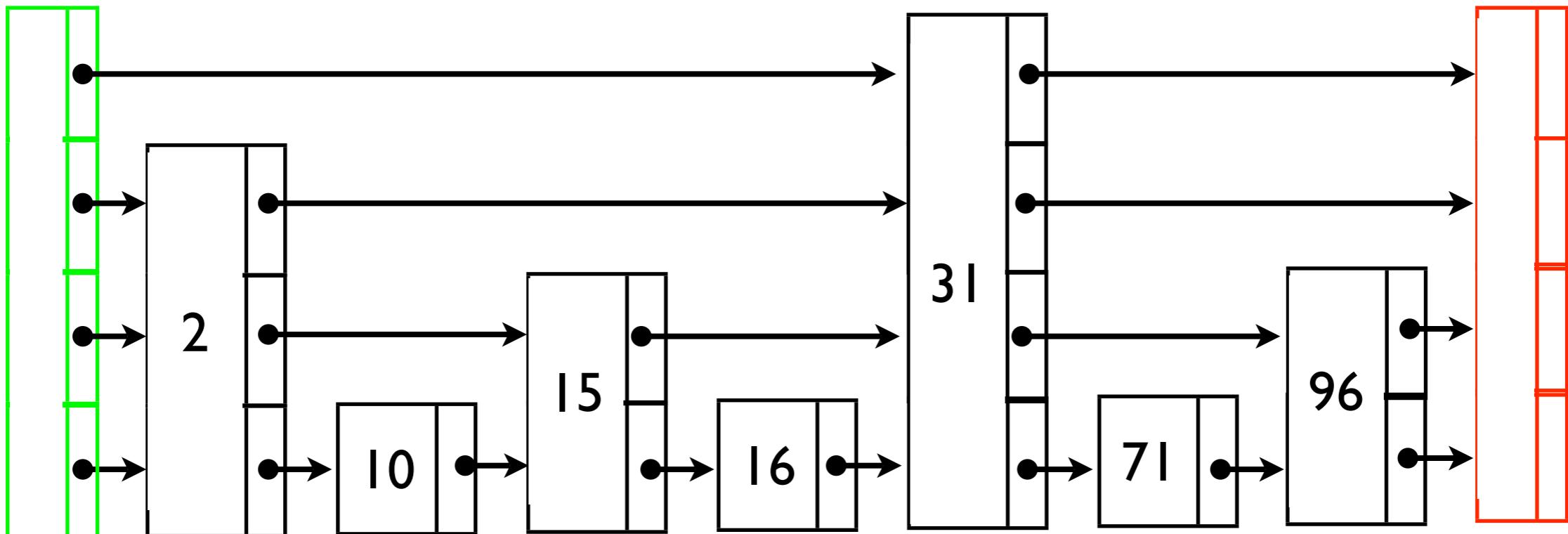
# Perfect Skip Lists

- Keys in sorted order.
- $O(\log n)$  levels
- Each higher level contains  $1/2$  the elements of the level below it.
- Header & sentinel nodes are in every level



# Perfect Skip Lists, continued

- Nodes are of variable size:
  - contain between 1 and  $O(\log n)$  pointers
- Pointers point to the start of each node  
(picture draws pointers horizontally for visual clarity)
- Called skip lists because higher level lists let you skip over many items

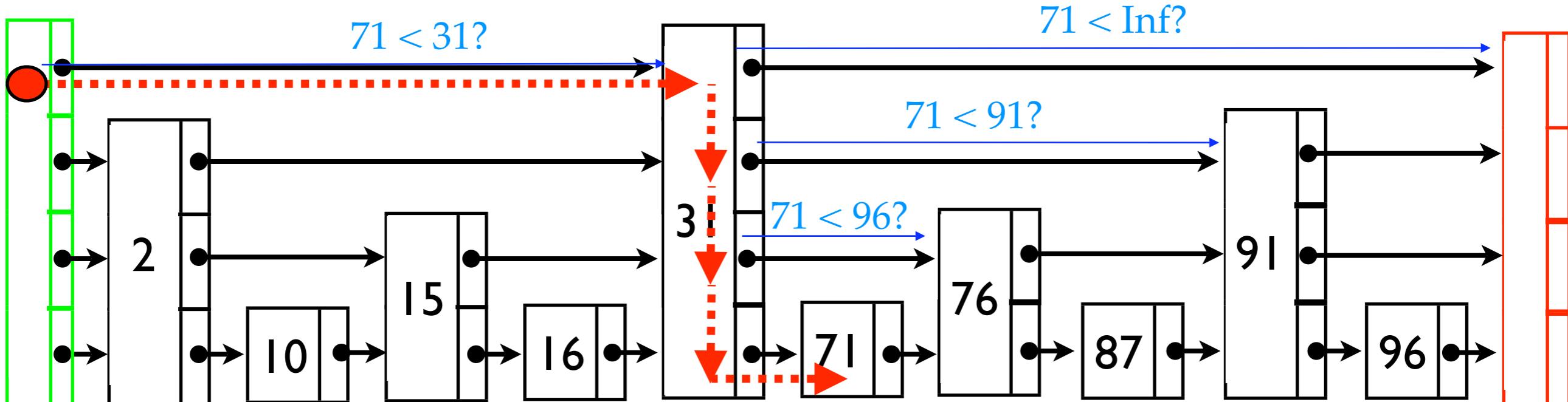


# Perfect Skip Lists, continued

Find 71

Comparison

Change  
current  
location



When search for  $k$ :

If  $k = \text{key}$ , done!

If  $k < \text{next key}$ , go down a level

If  $k \geq \text{next key}$ , go right

## In other words,

- To find an item, we scan along the shortest list until we would “pass” the desired item.
- At that point, we drop down to a slightly more complete list at one level lower.
- Remember: sorted sequential searching...

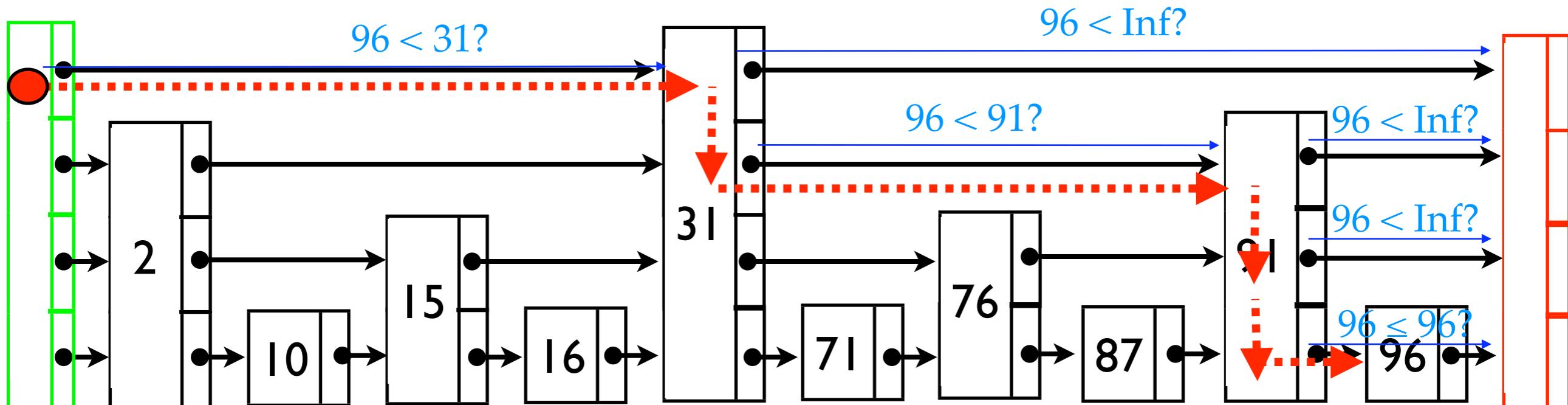
```
for(i = 0; i < n; i++)  
    if(X[i] >= K) break;  
if(X[i] != K) return FAIL;
```

# Perfect Skip Lists, continued

Find 96

Comparison

Change  
current  
location



When search for  $k$ :

If  $k = \text{key}$ , done!

If  $k < \text{next key}$ , go down a level

If  $k \geq \text{next key}$ , go right

## Search Time:

- $O(\log n)$  levels --- because you cut the # of items in half at each level
- Will visit at most 2 nodes per level:  
If you visit more, then you could have done it on one level higher up.
- Therefore, search time is  $O(\log n)$ .

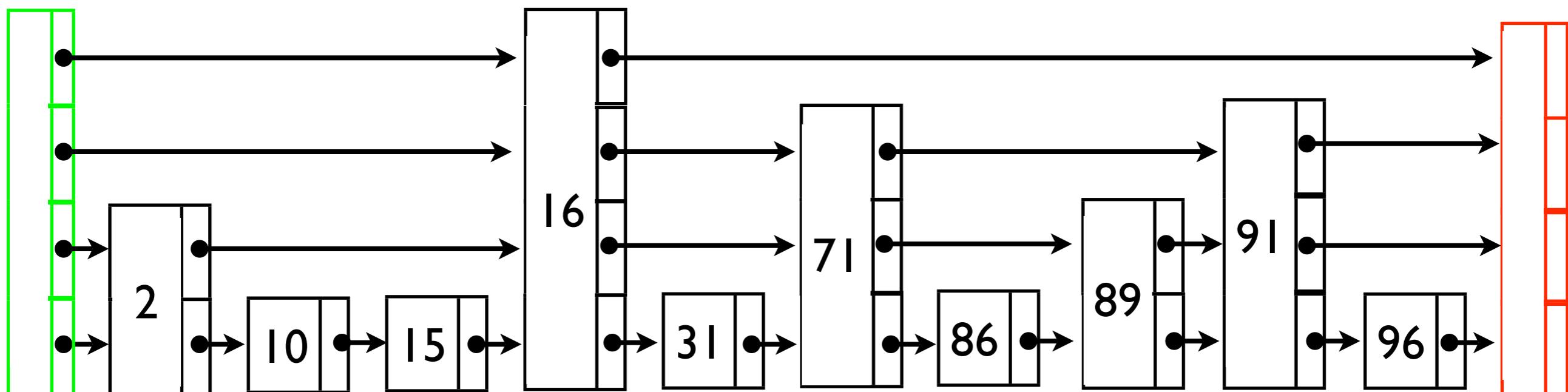
## Insert & Delete

- Insert & delete might need to rearrange the entire list
- Like Perfect Binary Search Trees, Perfect Skip Lists are too structured to support efficient updates.
- Idea:
  - Relax the requirement that each level have exactly half the items of the previous level
  - Instead: design structure so that we expect 1/2 the items to be carried up to the next level
  - Skip Lists are a randomized data structure: the same sequence of inserts / deletes may produce different structures depending on the outcome of random coin flips.

# Randomization

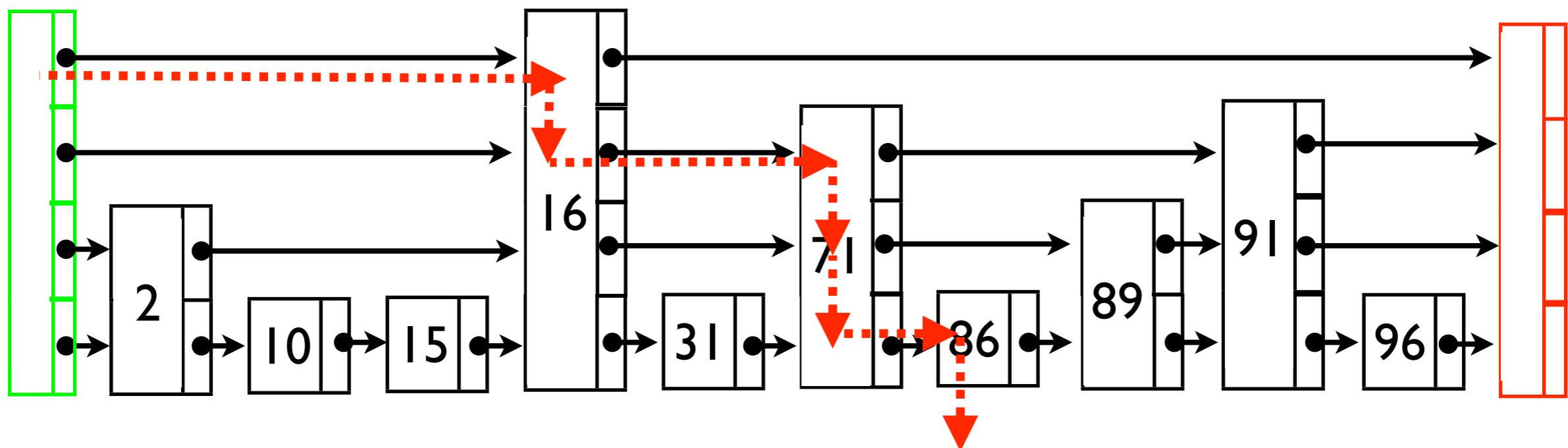
- Allows for some imbalance (like the +1 -1 in AVL trees)
- Expected behavior (over the random choices) remains the same as with perfect skip lists.
- Idea: Each node is promoted to the next higher level with probability 1/2
  - Expect 1/2 the nodes at level 1
  - Expect 1/4 the nodes at level 2
  - ...
- Therefore, expect # of nodes at each level is the same as with perfect skip lists.
- Also: expect the promoted nodes will be well distributed across the list

# Randomized Skip List:



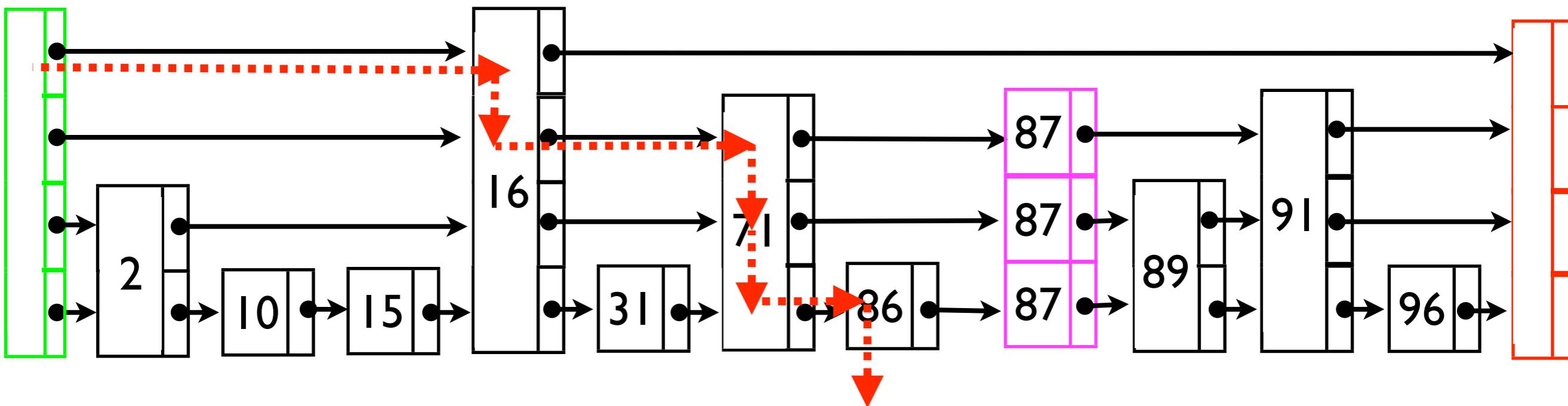
# Insertion:

Insert 87



# Insertion:

Insert 87



Find  $k$

Insert node in level 0

**let**  $i = 1$

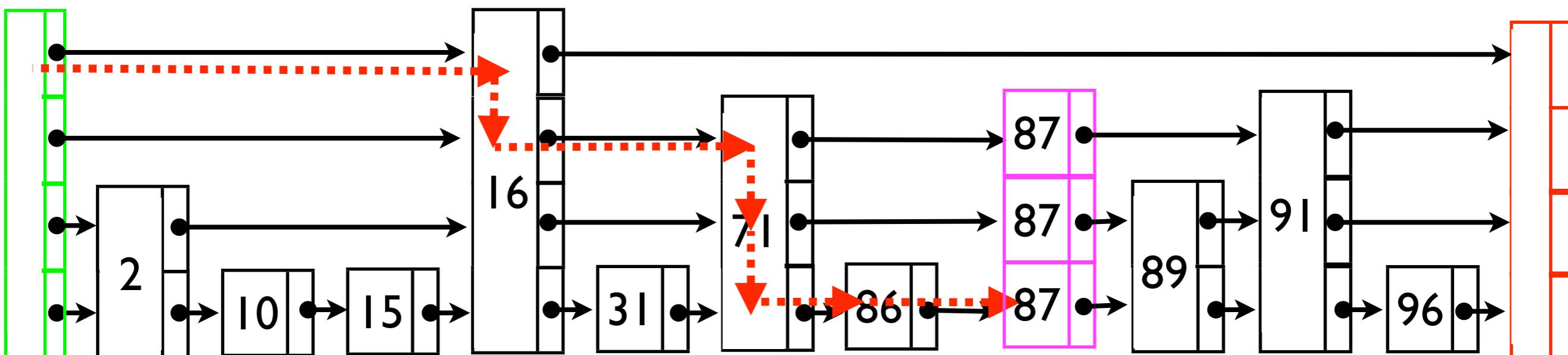
**while** FLIP() == "heads":

    insert node into level  $i \leftarrow i + 1$

Just insertion into  
a linked list after  
last visited node in  
level  $i$

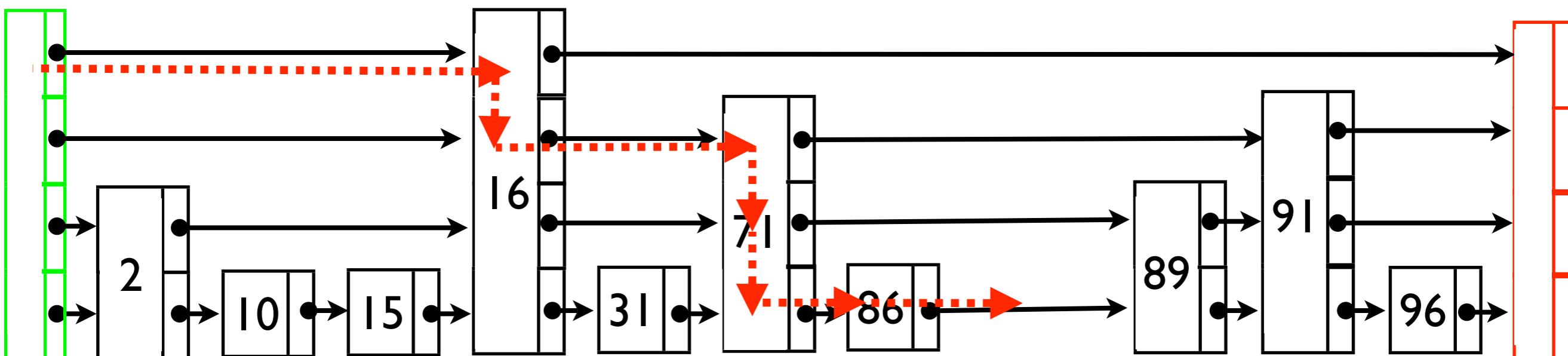
# Deletion:

Delete 87



# Deletion:

Delete 87



## There are no “bad” sequences:

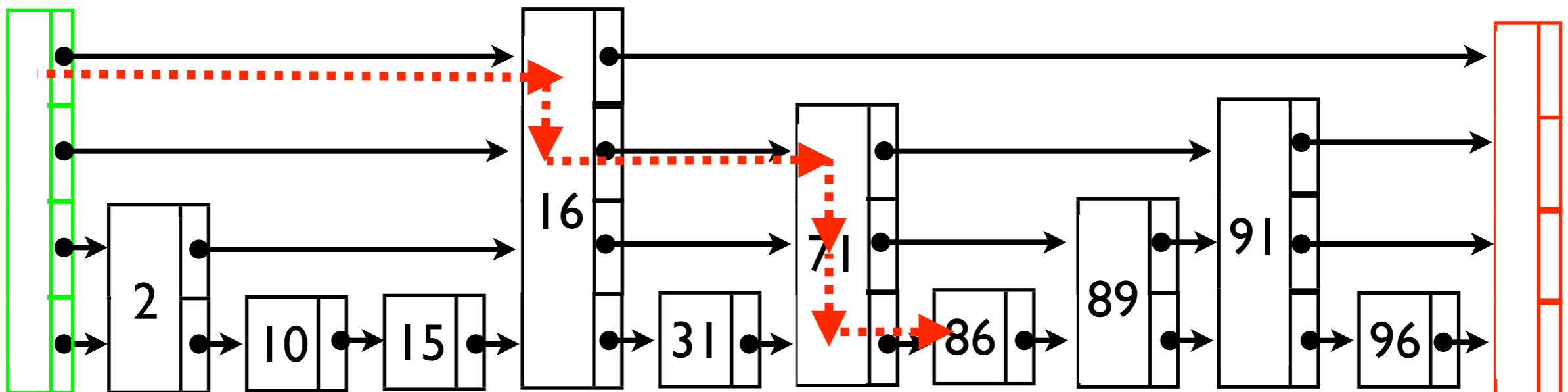
- We expect a randomized skip list to perform about as well as a perfect skip list.
- With some *very* small probability,
  - the skip list will just be a linked list, or
  - the skip list will have every node at every level
  - These *degenerate* skip lists are very unlikely!
- Level structure of a skip list is independent of the keys you insert.
- Therefore, there are no “bad” key sequences that will lead to degenerate skip lists

# Skip List Analysis

- Expected number of levels =  $O(\log n)$ 
  - $E[\# \text{ nodes at level 1}] = n/2$
  - $E[\# \text{ nodes at level 2}] = n/4$
  - ...
  - $E[\# \text{ nodes at level } \log n] = 1$
- Still need to prove that # of steps at each level is small.

# Backwards Analysis

Consider the reverse of the path you took to find  $k$ :



Note that you always move up if you can.  
(because you always enter a node from its topmost level when doing a find)

## Analysis, continued...

- What's the probability that you can move up at a give step of the reverse walk?

0.5

- Steps to go up  $j$  levels =  
Make one step, then make either  
 $C(j-1)$  steps if this step went up [Prob = 0.5]  
 $C(j)$  steps if this step went left [Prob = 0.5]
- Expected # of steps to walk up  $j$  levels is:  
$$C(j) = 1 + 0.5C(j-1) + 0.5C(j)$$

## Analysis Continue, 2

- Expected # of steps to walk up  $j$  levels is:

$$C(j) = 1 + 0.5C(j-1) + 0.5C(j)$$

So:

$$2C(j) = 2 + C(j-1) + C(j)$$

$$C(j) = 2 + C(j-1)$$



Expected # of steps at each level = 2

- Expanding  $C(j)$  above gives us:  $C(j) = 2j$
- Since  $O(\log n)$  levels, we have  $O(\log n)$  steps, expected

# Implementation Notes

- Node structures are of variable size
- But once a node is created, its size won't change
- It's often convenient to assume that you know the maximum number of levels in advance (but this is not a requirement).

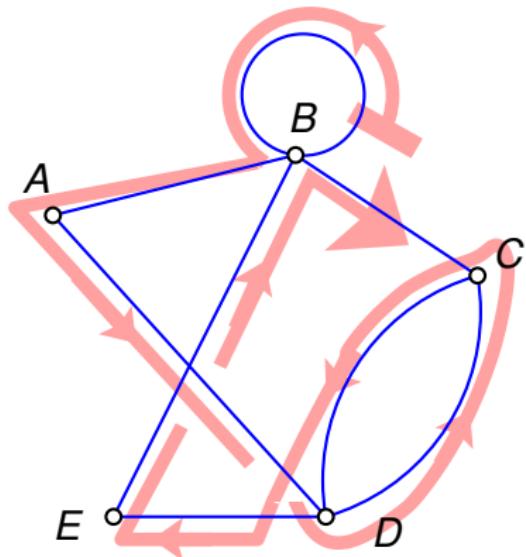
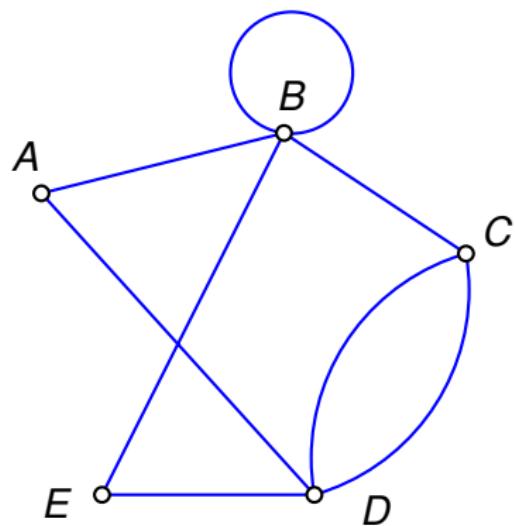
# Euler Paths and Euler Circuits

An **Euler path** is a path that uses every edge of a graph exactly once.

An **Euler circuit** is a circuit that uses every edge of a graph exactly once.

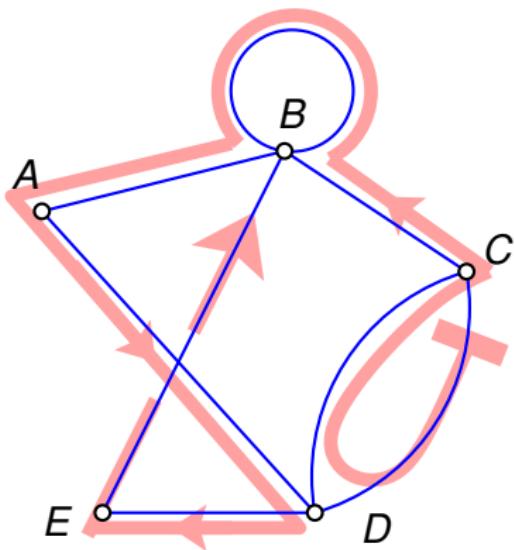
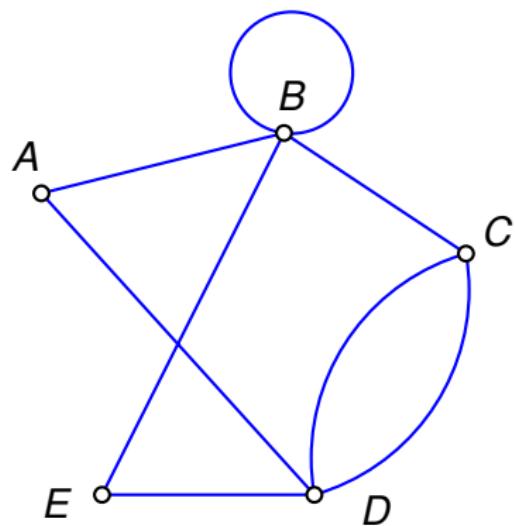
- ▶ An Euler path starts and ends at **different** vertices.
- ▶ An Euler circuit starts and ends at **the same** vertex.

# Euler Paths and Euler Circuits



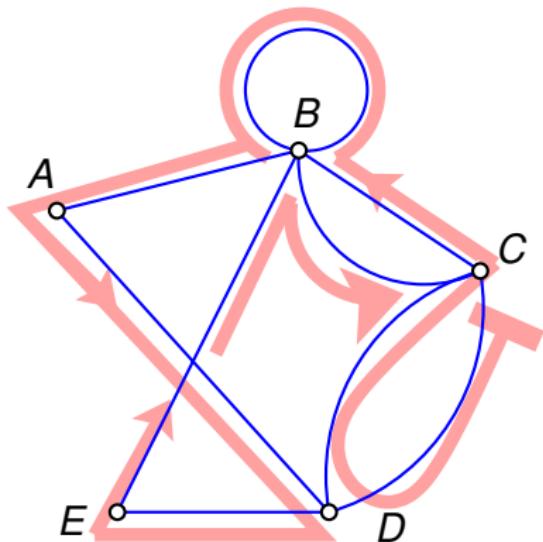
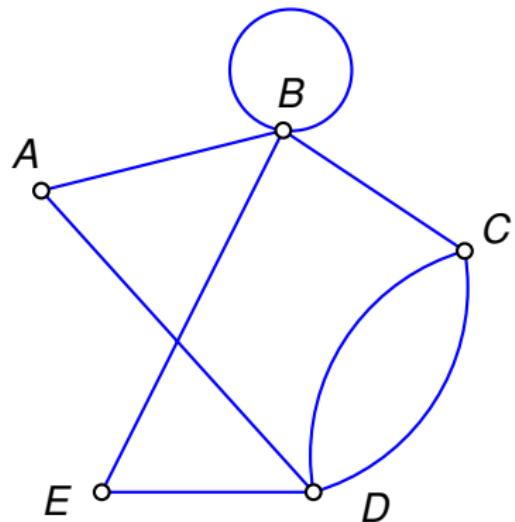
An Euler path: BBADCDEBC

# Euler Paths and Euler Circuits



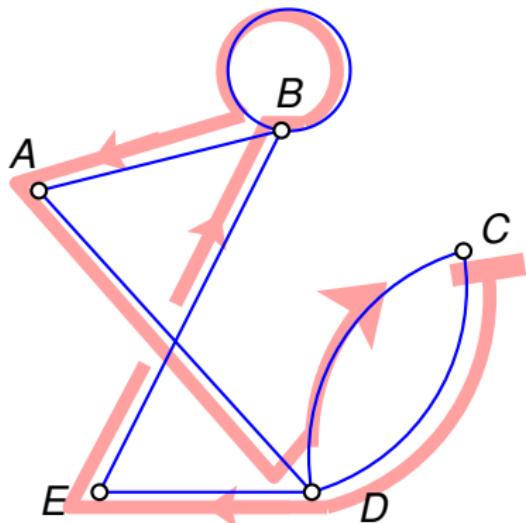
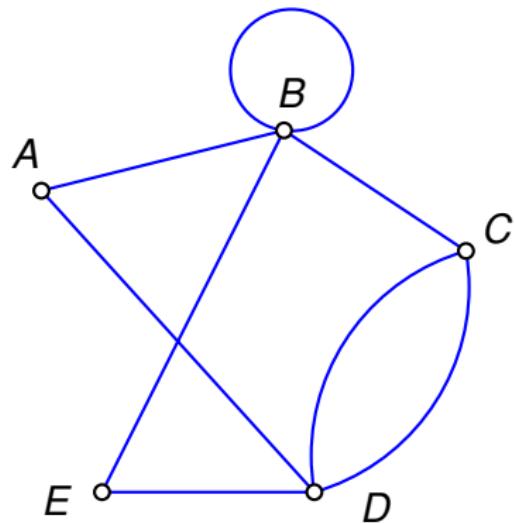
**Another Euler path: CDCBBADEB**

# Euler Paths and Euler Circuits



An Euler circuit: **CDCBBADEBC**

# Euler Paths and Euler Circuits



**Another Euler circuit: CDEBBADC**

# Euler Paths and Euler Circuits

**Is it possible to determine whether a graph has an Euler path or an Euler circuit, without necessarily having to find one explicitly?**



# The Criterion for Euler Paths

Suppose that a graph has an Euler path  $P$ .

# The Criterion for Euler Paths

Suppose that a graph has an Euler path  $P$ .

For every vertex  $v$  other than the starting and ending vertices,  
the path  $P$  enters  $v$  the same number of times that it leaves  $v$   
(say  $\boxed{s}$  times).

# The Criterion for Euler Paths

Suppose that a graph has an Euler path  $P$ .

For every vertex  $v$  other than the starting and ending vertices, the path  $P$  enters  $v$  the same number of times that it leaves  $v$  (say  $\boxed{s}$  times).

Therefore, there are  $\boxed{2s}$  edges having  $v$  as an endpoint.

# The Criterion for Euler Paths

Suppose that a graph has an Euler path  $P$ .

For every vertex  $v$  other than the starting and ending vertices, the path  $P$  enters  $v$  the same number of times that it leaves  $v$  (say  $\boxed{s}$  times).

Therefore, there are  $\boxed{2s}$  edges having  $v$  as an endpoint.

**Therefore, all vertices other than the two endpoints of  $P$  must be even vertices.**

# The Criterion for Euler Paths

Suppose the Euler path  $P$  starts at vertex  $x$  and ends at  $y$ .

# The Criterion for Euler Paths

Suppose the Euler path  $P$  starts at vertex  $x$  and ends at  $y$ .

Then  $P$  leaves  $x$  one more time than it enters, and leaves  $y$  one fewer time than it enters.

# The Criterion for Euler Paths

Suppose the Euler path  $P$  starts at vertex  $x$  and ends at  $y$ .

Then  $P$  leaves  $x$  one more time than it enters, and leaves  $y$  one fewer time than it enters.

Therefore, **the two endpoints of  $P$  must be odd vertices.**

# The Criterion for Euler Paths

The inescapable conclusion (“based on reason alone!”):

**If a graph  $G$  has an Euler path, then it must have exactly two odd vertices.**

Or, to put it another way,

**If the number of odd vertices in  $G$  is anything other than 2, then  $G$  cannot have an Euler path.**

# The Criterion for Euler Circuits

- ▶ Suppose that a graph  $G$  has an Euler circuit  $C$ .

# The Criterion for Euler Circuits

- ▶ Suppose that a graph  $G$  has an Euler circuit  $C$ .
- ▶ For every vertex  $v$  in  $G$ , each edge having  $v$  as an endpoint shows up *exactly once* in  $C$ .

# The Criterion for Euler Circuits

- ▶ Suppose that a graph  $G$  has an Euler circuit  $C$ .
- ▶ For every vertex  $v$  in  $G$ , each edge having  $v$  as an endpoint shows up *exactly once* in  $C$ .
- ▶ The circuit  $C$  **enters**  $v$  the same number of times that it **leaves**  $v$  (say  $s$  times), so  $v$  has degree  $2s$ .

# The Criterion for Euler Circuits

- ▶ Suppose that a graph  $G$  has an Euler circuit  $C$ .
- ▶ For every vertex  $v$  in  $G$ , each edge having  $v$  as an endpoint shows up *exactly once* in  $C$ .
- ▶ The circuit  $C$  **enters**  $v$  the same number of times that it **leaves**  $v$  (say  $s$  times), so  $v$  has degree  $2s$ .
- ▶ That is,  **$v$  must be an even vertex.**

# The Criterion for Euler Circuits

The inescapable conclusion (“based on reason alone”):

**If a graph  $G$  has an Euler circuit, then all of its vertices must be even vertices.**

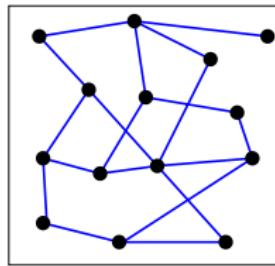
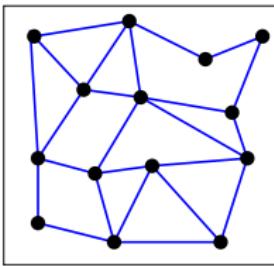
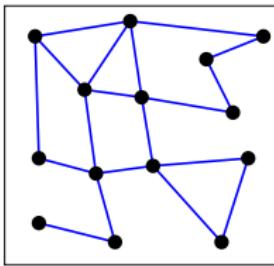
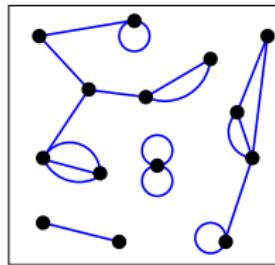
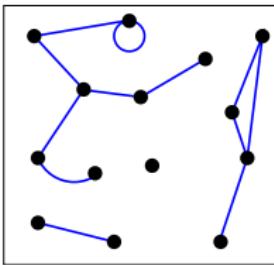
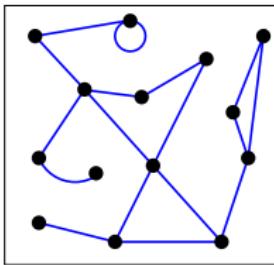
Or, to put it another way,

**If the number of odd vertices in  $G$  is anything other than 0, then  $G$  cannot have an Euler circuit.**

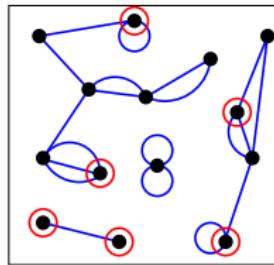
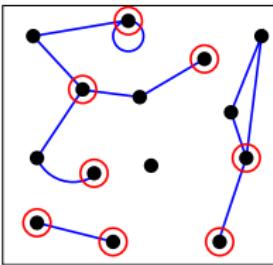
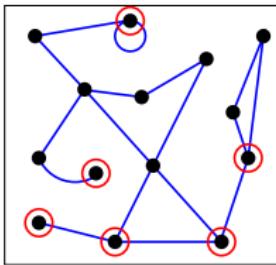
# Things You Should Be Wondering

- ▶ Does **every** graph with **zero** odd vertices have an Euler circuit?
- ▶ Does **every** graph with **two** odd vertices have an Euler path?
- ▶ Is it possible for a graph have just **one** odd vertex? 

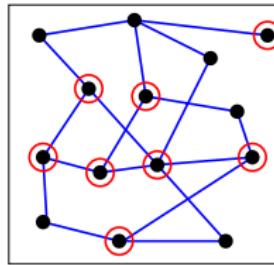
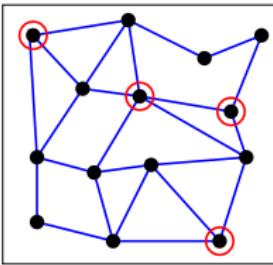
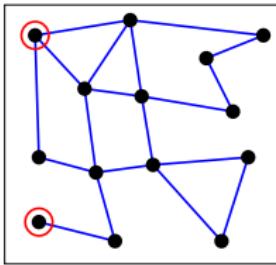
# How Many Odd Vertices?



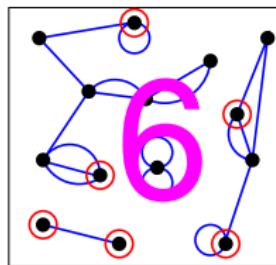
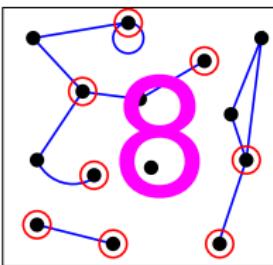
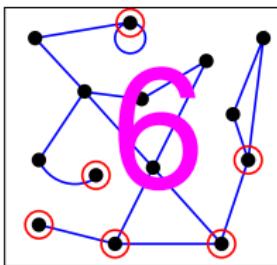
# How Many Odd Vertices?



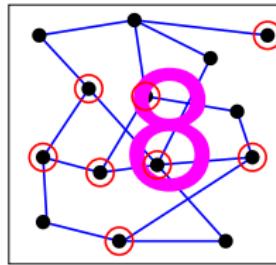
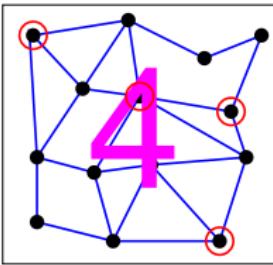
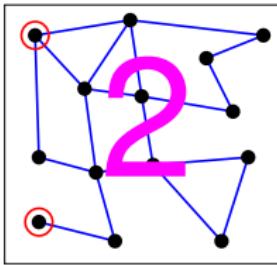
Odd vertices



# How Many Odd Vertices?



Number of odd vertices



# The Handshaking Theorem

The Handshaking Theorem says that

In every graph, the sum of the degrees of all vertices equals twice the number of edges.

If there are  $n$  vertices  $V_1, \dots, V_n$ , with degrees  $d_1, \dots, d_n$ , and there are  $e$  edges, then

$$d_1 + d_2 + \cdots + d_{n-1} + d_n = 2e$$

Or, equivalently,

$$e = \frac{d_1 + d_2 + \cdots + d_{n-1} + d_n}{2}$$

# The Handshaking Theorem

Why “Handshaking”?

If  $n$  people shake hands, and the  $i^{th}$  person shakes hands  $d_i$  times, then the total number of handshakes that take place is

$$\frac{d_1 + d_2 + \cdots + d_{n-1} + d_n}{2}.$$

(How come? Each handshake involves two people, so the number  $d_1 + d_2 + \cdots + d_{n-1} + d_n$  counts every handshake twice.)

# The Number of Odd Vertices

- ▶ The number of edges in a graph is

$$\frac{d_1 + d_2 + \cdots + d_n}{2}$$

which must be an **integer**.

# The Number of Odd Vertices

- ▶ The number of edges in a graph is

$$\frac{d_1 + d_2 + \cdots + d_n}{2}$$

which must be an **integer**.

- ▶ Therefore,  $d_1 + d_2 + \cdots + d_n$  must be an **even number**.

# The Number of Odd Vertices

- ▶ The number of edges in a graph is

$$\frac{d_1 + d_2 + \cdots + d_n}{2}$$

which must be an **integer**.

- ▶ Therefore,  $d_1 + d_2 + \cdots + d_n$  must be an **even number**.
- ▶ Therefore, the numbers  $d_1, d_2, \dots, d_n$  must include an **even number of odd numbers**.

# The Number of Odd Vertices

- ▶ The number of edges in a graph is

$$\frac{d_1 + d_2 + \cdots + d_n}{2}$$

which must be an **integer**.

- ▶ Therefore,  $d_1 + d_2 + \cdots + d_n$  must be an **even number**.
- ▶ Therefore, the numbers  $d_1, d_2, \dots, d_n$  must include an **even number of odd numbers**.
- ▶ **Every graph has an even number of odd vertices!**

# Back to Euler Paths and Circuits

Here's what we know so far:

# odd vertices	Euler path?	Euler circuit?
0	No	Maybe
2	Maybe	No
4, 6, 8, ...	No	No
1, 3, 5, ...	<i>No such graphs exist!</i>	

Can we give a better answer than “maybe”?

# Euler Paths and Circuits — The Last Word

Here is the answer Euler gave:

# odd vertices	Euler path?	Euler circuit?
0	No	Yes*
2	Yes*	No
4, 6, 8, ...	No	No
1, 3, 5,	No such graphs exist	

\* *Provided the graph is connected.*

# Euler Paths and Circuits — The Last Word

Here is the answer Euler gave:

# odd vertices	Euler path?	Euler circuit?
0	No	Yes*
2	Yes*	No
4, 6, 8, ...	No	No
1, 3, 5,	No such graphs exist	

\* Provided the graph is connected.

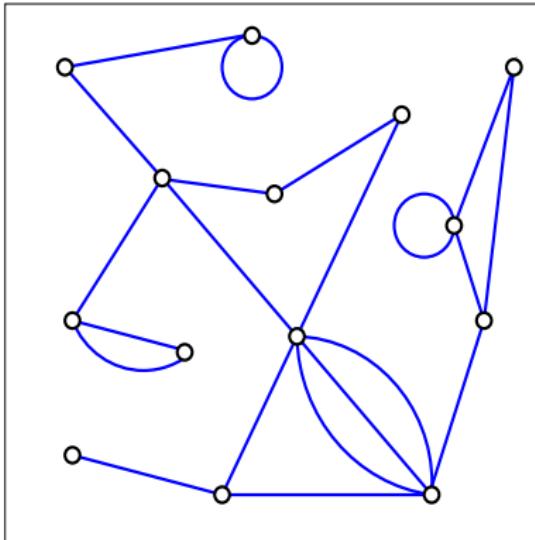
**Next question: If an Euler path or circuit exists, how do you find it?**

# Bridges

Removing a single edge from a connected graph can make it disconnected. Such an edge is called a **bridge**. 

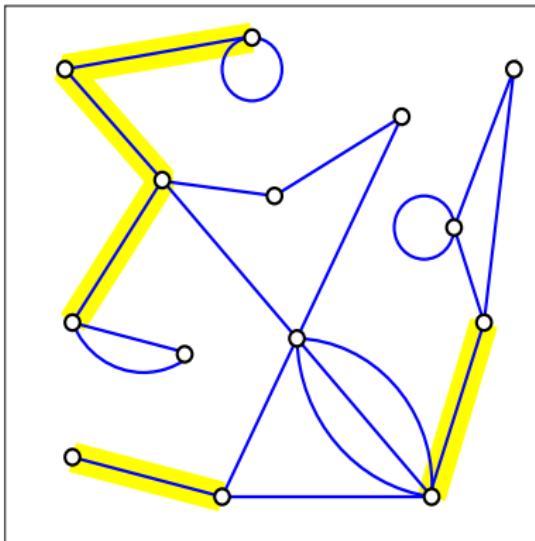
# Bridges

Removing a single edge from a connected graph can make it disconnected. Such an edge is called a **bridge**. 



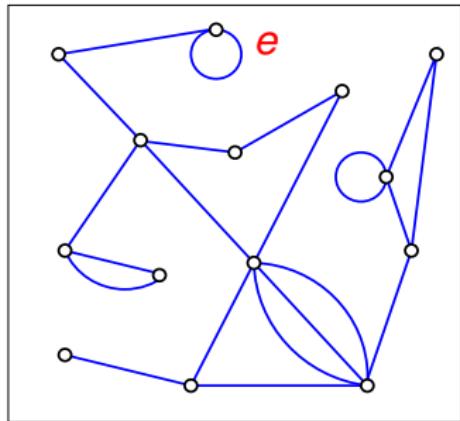
# Bridges

Removing a single edge from a connected graph can make it disconnected. Such an edge is called a **bridge**. 

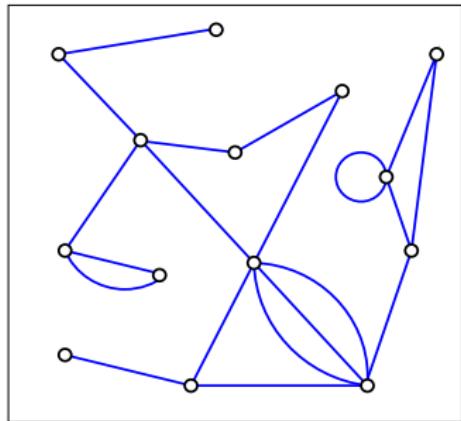


# Bridges

Loops cannot be bridges, because removing a loop from a graph cannot make it disconnected.

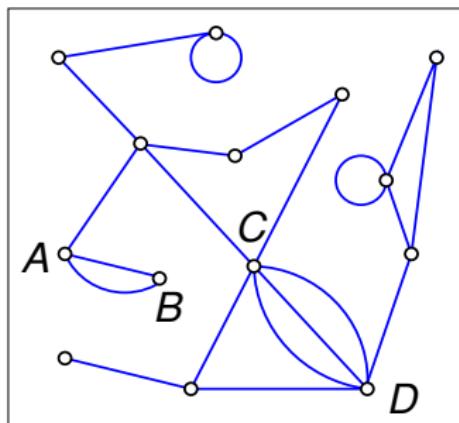


*delete  
loop e*

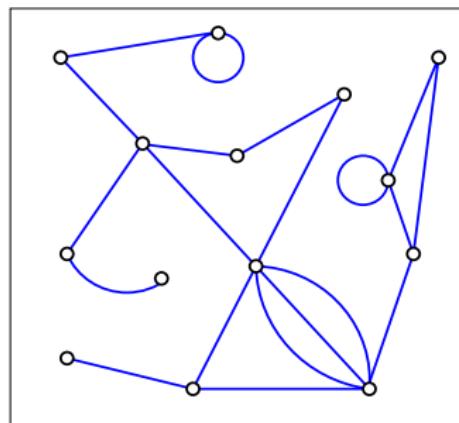


# Bridges

If two or more edges share both endpoints, then removing any one of them cannot make the graph disconnected. Therefore, none of those edges is a bridge.

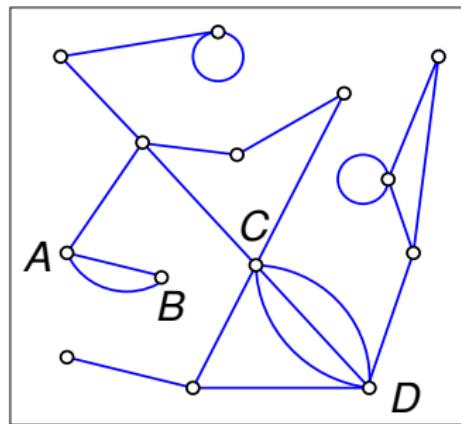


*delete  
multiple  
edges*

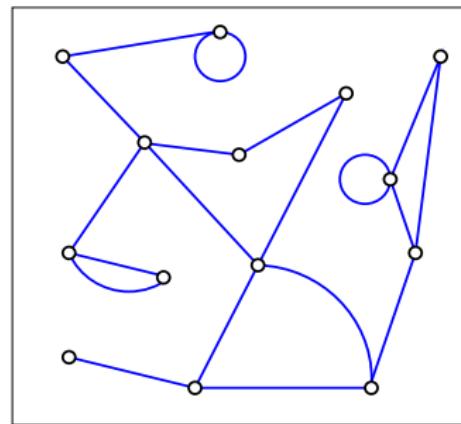


# Bridges

If two or more edges share both endpoints, then removing any one of them cannot make the graph disconnected. Therefore, none of those edges is a bridge.



*delete  
multiple  
edges*  
→

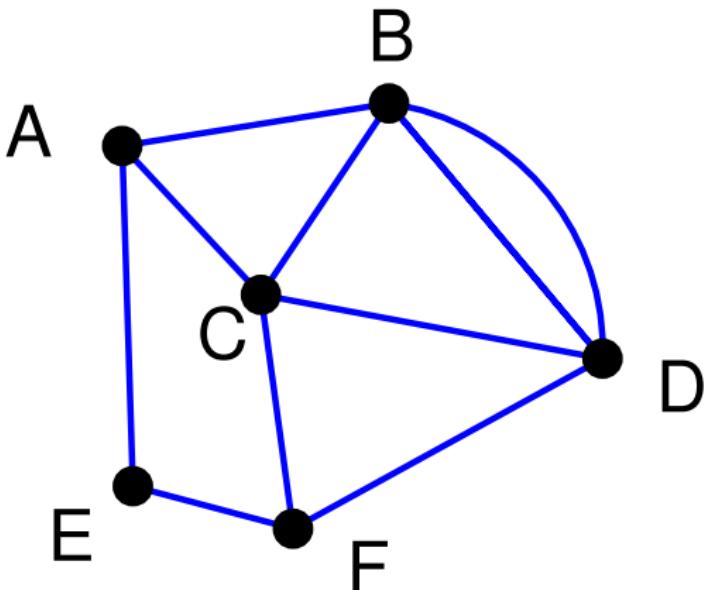


# Finding Euler Circuits and Paths

*“Don’t burn your bridges.”*

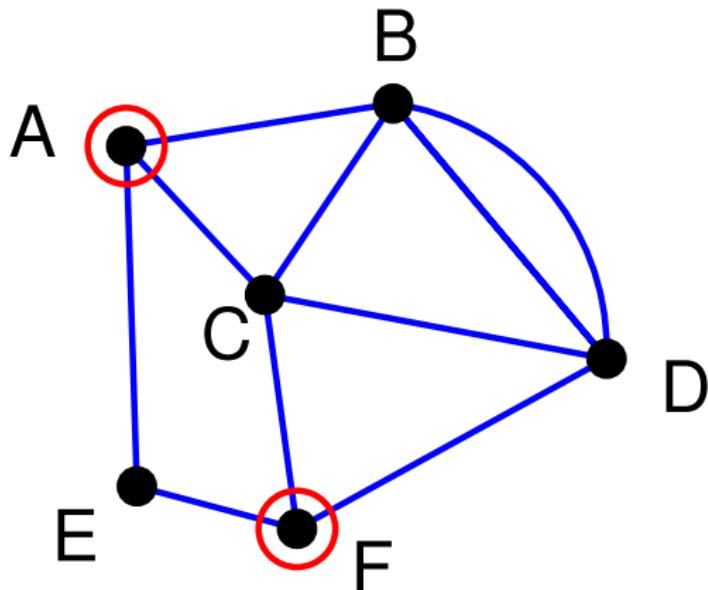
# Finding Euler Circuits and Paths

Problem: Find an Euler circuit in the graph below.



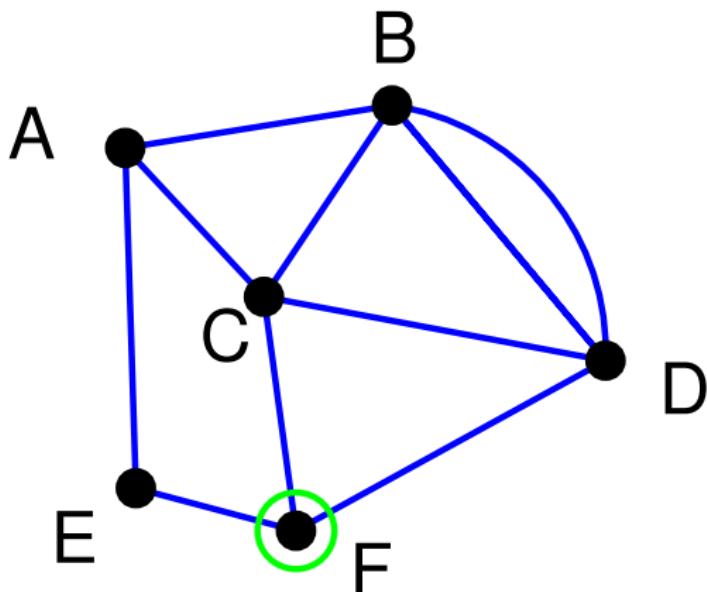
# Finding Euler Circuits and Paths

There are two odd vertices, A and F. Let's start at F.



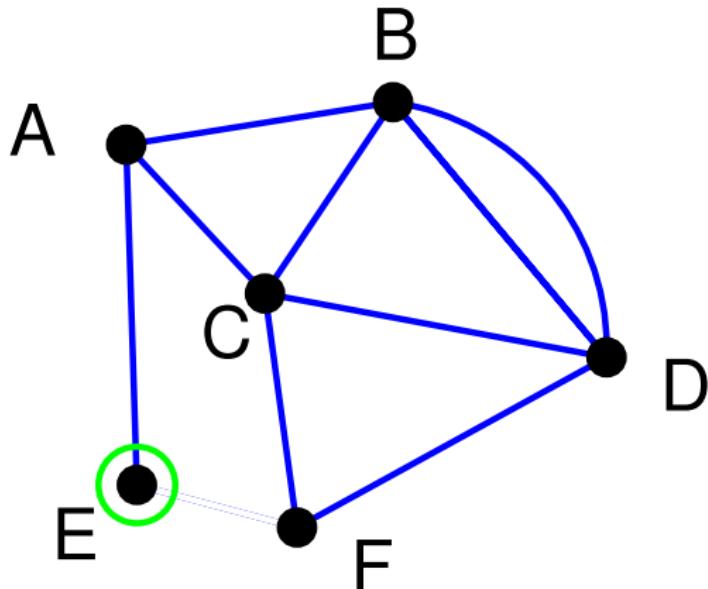
# Finding Euler Circuits and Paths

Start walking at F. When you use an edge, delete it.



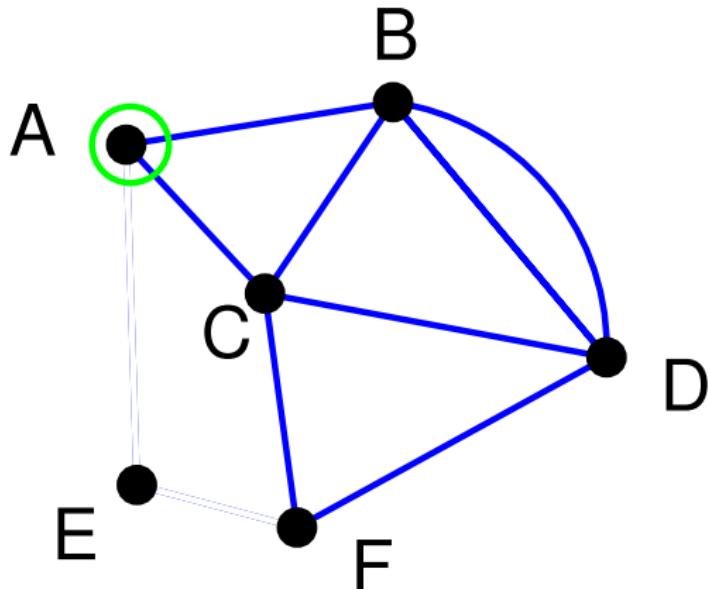
# Finding Euler Circuits and Paths

Path so far: FE



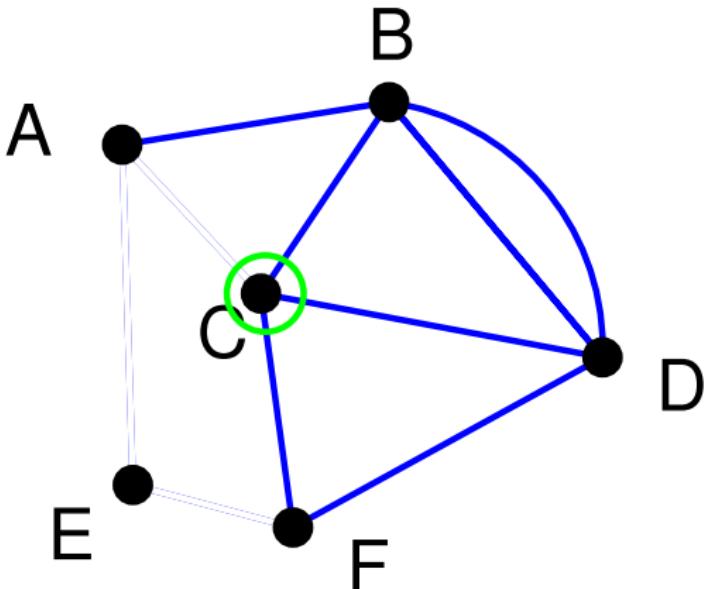
# Finding Euler Circuits and Paths

Path so far: FEA



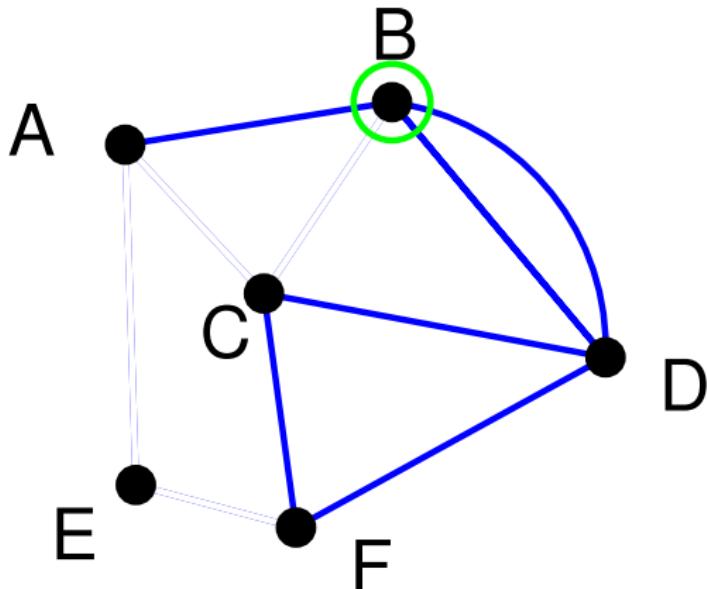
# Finding Euler Circuits and Paths

Path so far: FEAC



# Finding Euler Circuits and Paths

Path so far: FEACB



# Finding Euler Circuits and Paths

Up until this point, the choices didn't matter.

# Finding Euler Circuits and Paths

Up until this point, the choices didn't matter.

But now, crossing the edge BA would be a mistake, because we would be stuck there.

# Finding Euler Circuits and Paths

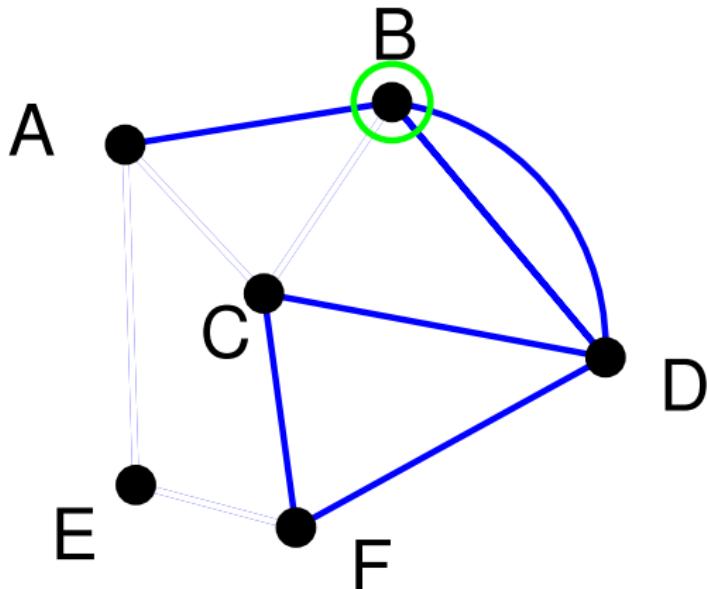
Up until this point, the choices didn't matter.

But now, crossing the edge BA would be a mistake, because we would be stuck there.

The reason is that BA is a **bridge**. We don't want to cross ("burn"?) a bridge unless it is the only edge available.

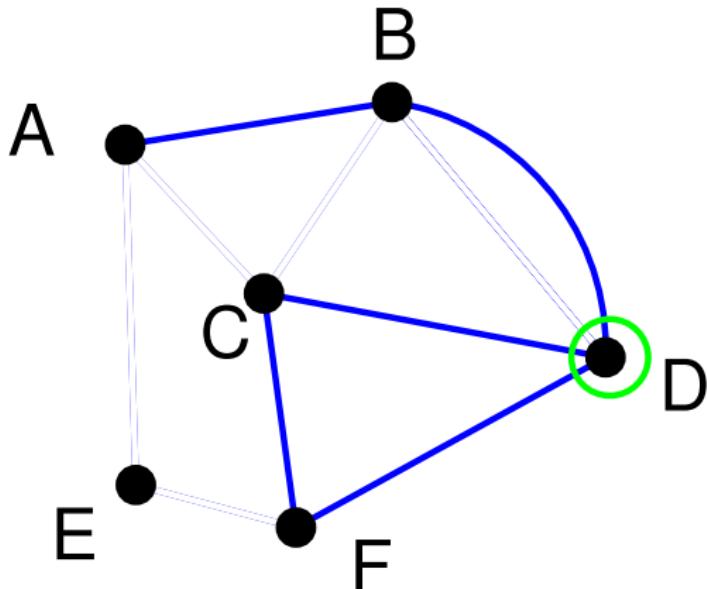
# Finding Euler Circuits and Paths

Path so far: FEACB



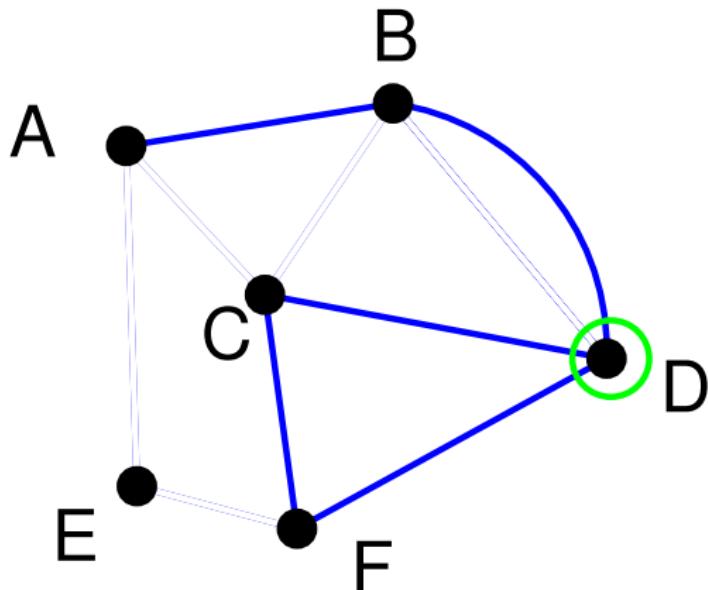
# Finding Euler Circuits and Paths

Path so far: FEACBD.



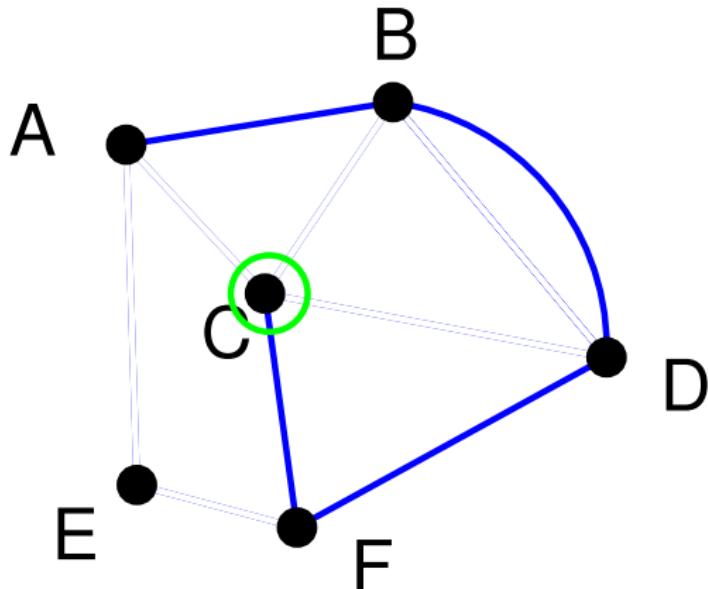
# Finding Euler Circuits and Paths

Path so far: FEACBD. **Don't cross the bridge!**



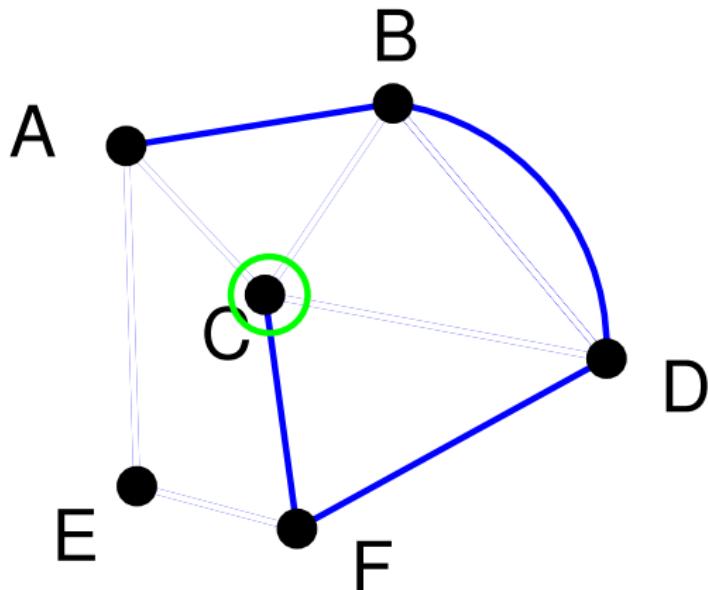
# Finding Euler Circuits and Paths

Path so far: FEACBDC



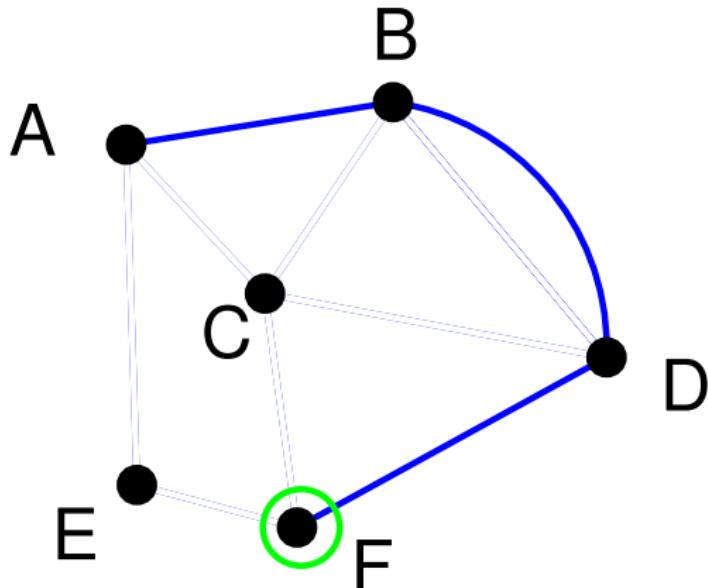
# Finding Euler Circuits and Paths

Path so far: FEACBDC   Now we have to cross the bridge CF.



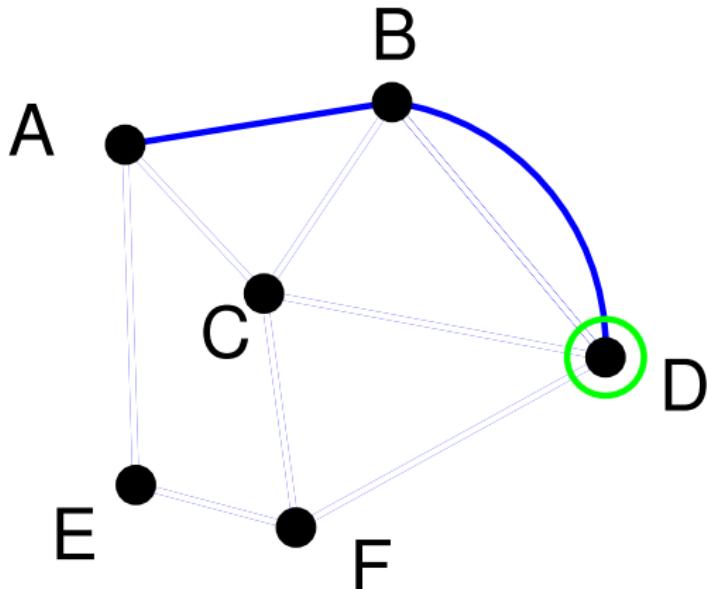
# Finding Euler Circuits and Paths

Path so far: FEACBDCF



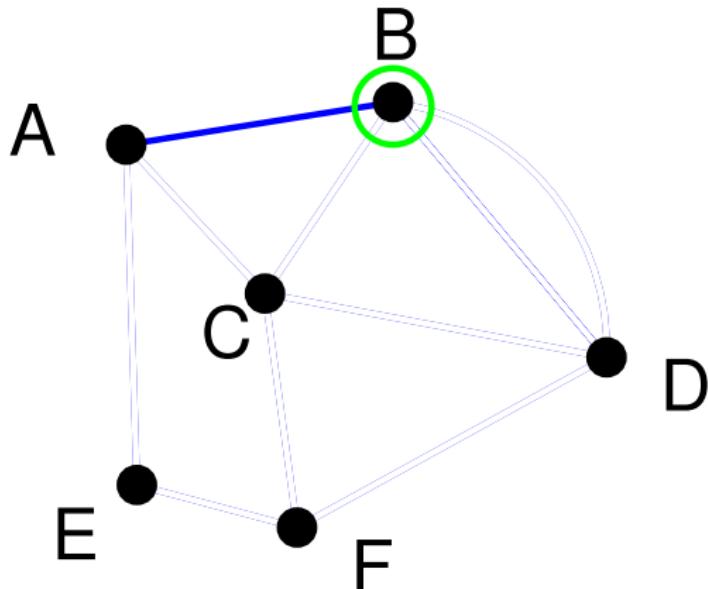
# Finding Euler Circuits and Paths

Path so far: FEACBDCFD



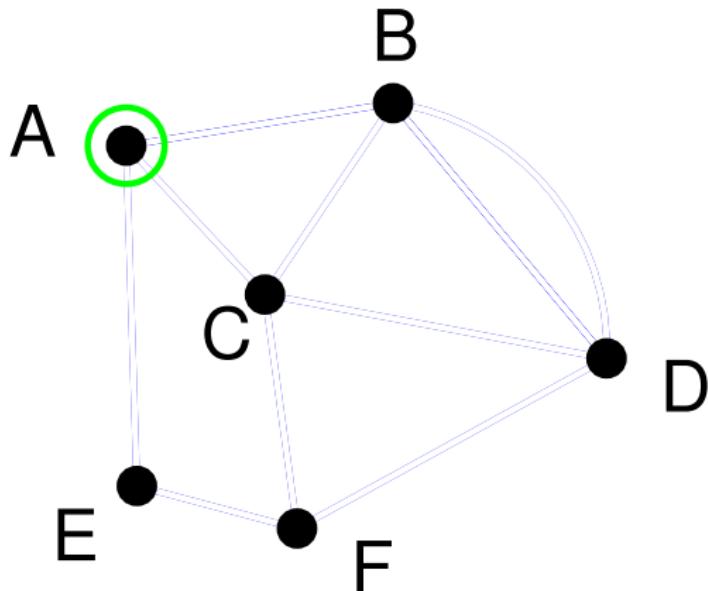
# Finding Euler Circuits and Paths

Path so far: FEACBDCFDB



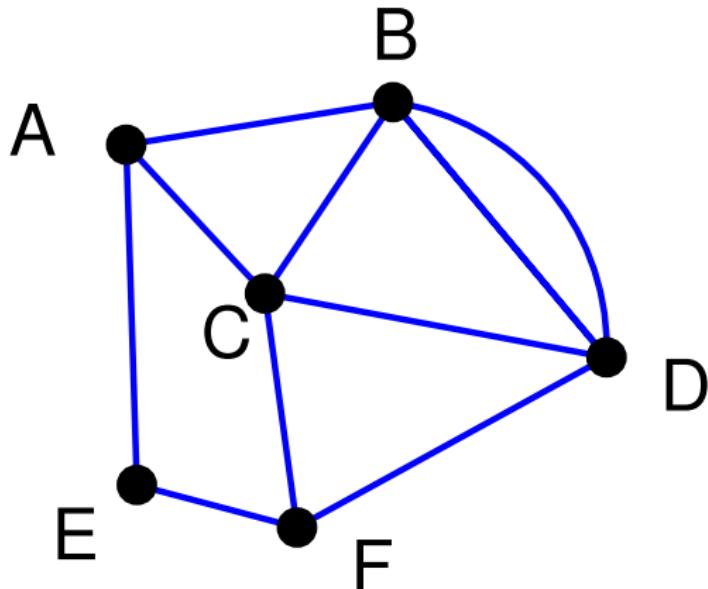
# Finding Euler Circuits and Paths

Euler Path: FEACBDCFDBA



# Finding Euler Circuits and Paths

Euler Path: FEACBDCFDBA



# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

1. Make sure the graph has either 0 or 2 odd vertices.

# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.

# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, **always choose the non-bridge**.

# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, **always choose the non-bridge**.
4. Stop when you run out of edges.

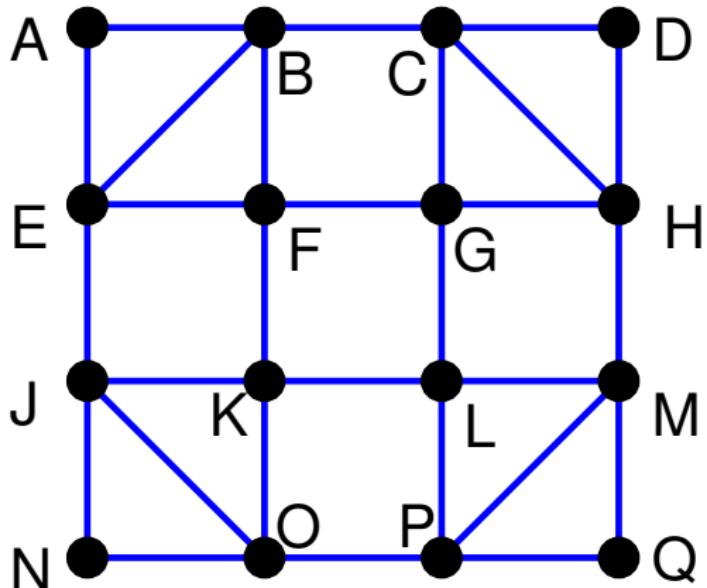
# Fleury's Algorithm

**To find an Euler path or an Euler circuit:**

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, **always choose the non-bridge**.
4. Stop when you run out of edges.

This is called **Fleury's algorithm**, and it always works!

# Fleury's Algorithm: Another Example



# Euler Graph | Euler Path | Euler Circuit

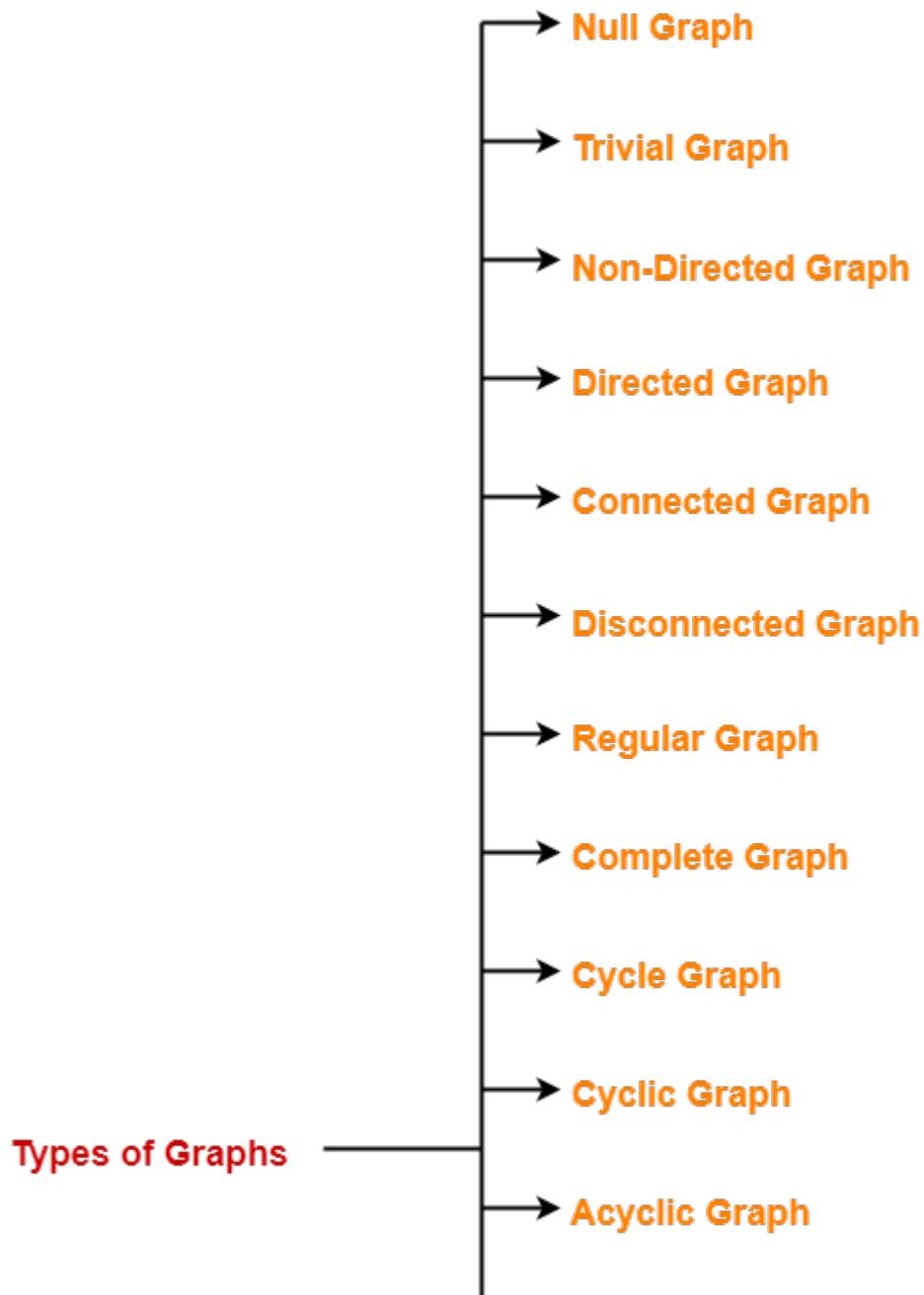
Graph Theory

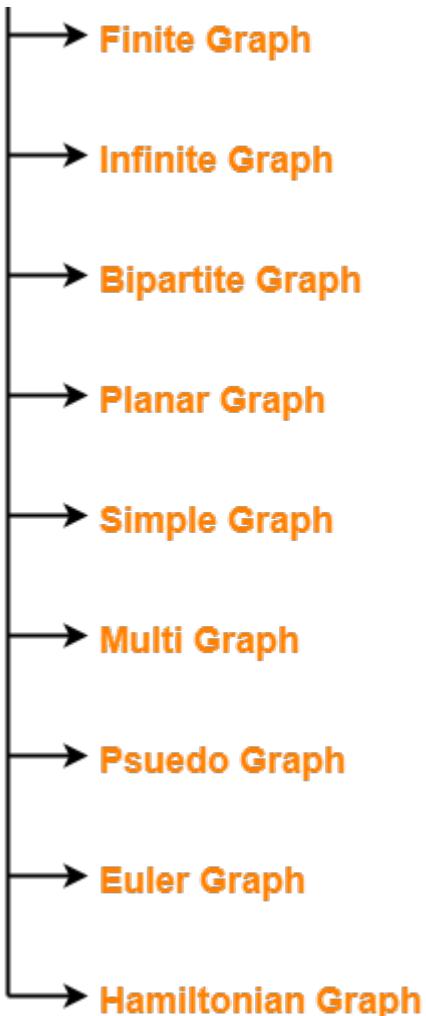
## Types of Graphs-

Before you go through this article, make sure that you have gone through the previous article on various [Types of Graphs](#) in Graph Theory.

We have discussed-

- A graph is a collection of vertices connected to each other through a set of edges.
- The study of graphs is known as Graph Theory.





In this article, we will discuss about Euler Graphs.

## Euler Graph-

An Euler graph may be defined as-

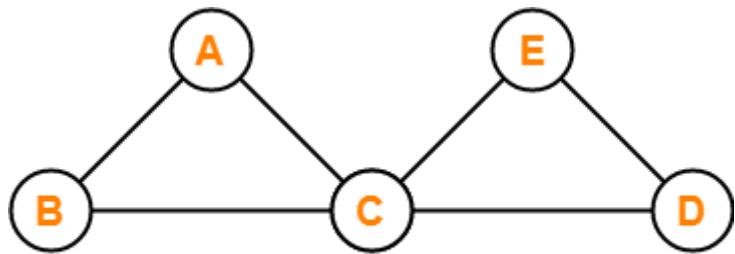
Any connected graph is called as an Euler Graph if and only if all its vertices are of even degree.

OR

An Euler Graph is a connected graph that contains an Euler Circuit.

## Euler Graph Example-

The following graph is an example of an Euler graph-



### Example of Euler Graph

Here,

- This graph is a connected graph and all its vertices are of even degree.
- Therefore, it is an Euler graph.

Alternatively, the above graph contains an Euler circuit BACEDCB, so it is an Euler graph.

Also Read- [Planar Graph](#)

## **Euler Path-**

Euler path is also known as Euler Trail or Euler Walk.

- If there exists a [Trail](#) in the connected graph that contains all the edges of the graph, then that

trail is called as an Euler trail.

OR

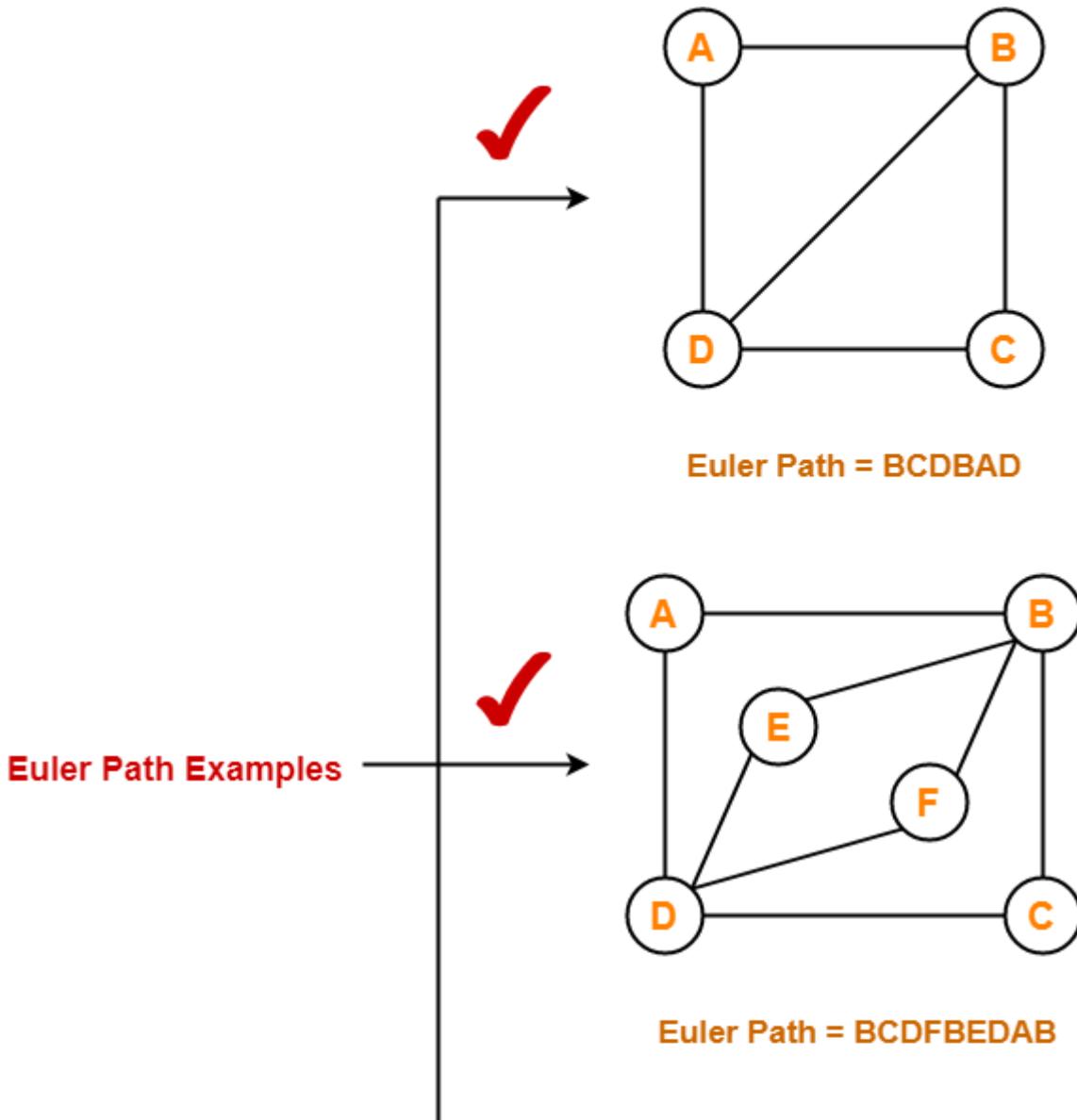
- If there exists a walk in the connected graph that visits every edge of the graph exactly once with or without repeating the vertices, then such a walk is called as an Euler walk.

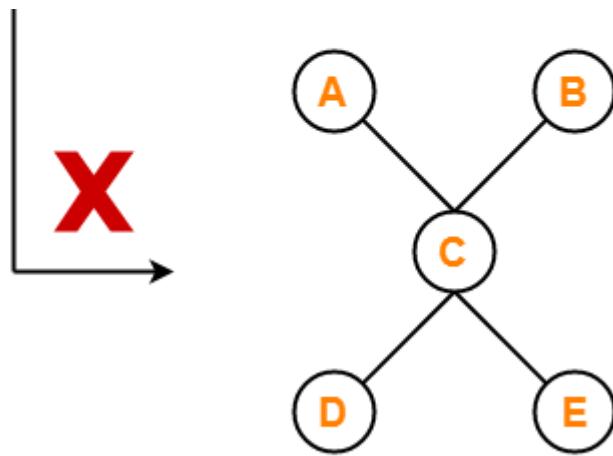
**NOTE**

A graph will contain an Euler path if and only if it contains at most two vertices of odd degree.

### Euler Path Examples-

Examples of Euler path are as follows-





**Euler Path Does Not Exist**

## Euler Circuit-

Euler circuit is also known as Euler Cycle or Euler Tour.

- If there exists a Circuit in the connected graph that contains all the edges of the graph, then that circuit is called as an Euler circuit.

OR

- If there exists a walk in the connected graph that starts and ends at the same vertex and visits every edge of the graph exactly once with or without repeating the vertices, then such a walk is called as an Euler circuit.

OR

- An Euler trail that starts and ends at the same vertex is called as an Euler circuit.

OR

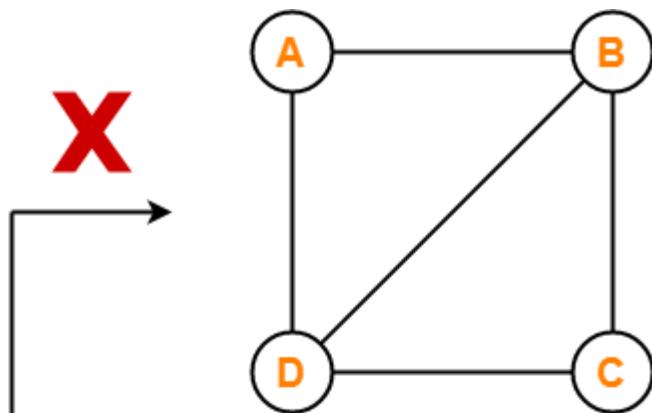
- A closed Euler trail is called as an Euler circuit.

### **NOTE**

A graph will contain an Euler circuit if and only if all its vertices are of even degree.

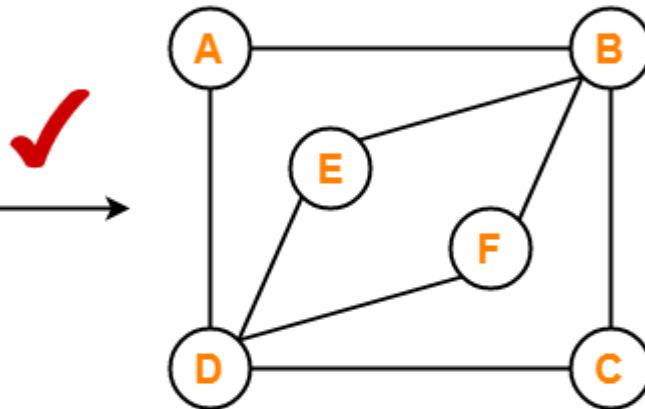
## Euler Circuit Examples-

Examples of Euler circuit are as follows-

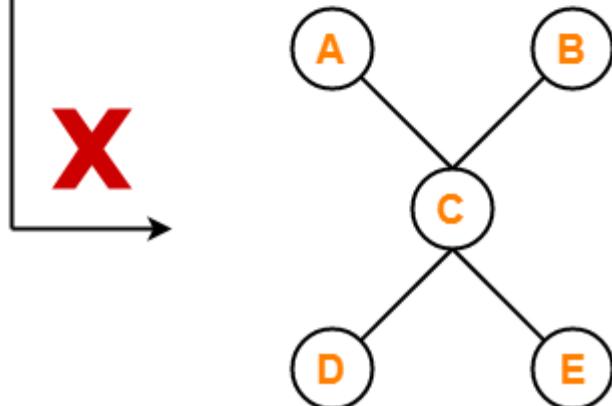


Euler Circuit Does Not Exist

Euler Circuit Examples



Euler Circuit = ABCDFBEDA



Euler Circuit Does Not Exist

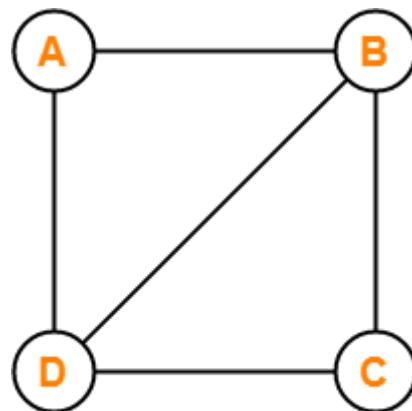
Semi-Euler Graph-

If a connected graph contains an Euler trail but does not contain an Euler circuit, then such a graph is called as a semi-Euler graph.

Thus, for a graph to be a semi-Euler graph, following two conditions must be satisfied-

- Graph must be connected.
- Graph must contain an Euler trail.

### **Example-**



**Semi-Euler Graph**

Here,

- This graph contains an Euler trail BCDBAD.
- But it does not contain an Euler circuit.
- Therefore, it is a semi-Euler graph.

Also Read- [Bipartite Graph](#)

### **Important Notes-**

#### **Note-01:**

To check whether any graph is an Euler graph or not, any one of the following two ways may be used-

- If the graph is connected and contains an Euler circuit, then it is an Euler graph.
- If all the vertices of the graph are of even degree, then it is an Euler graph.

### **Note-02:**

To check whether any graph contains an Euler circuit or not,

- Just make sure that all its vertices are of even degree.
- If all its vertices are of even degree, then graph contains an Euler circuit otherwise not.

### **Note-03:**

To check whether any graph is a semi-Euler graph or not,

- Just make sure that it is connected and contains an Euler trail.
- If the graph is connected and contains an Euler trail, then graph is a semi-Euler graph otherwise not.

### **Note-04:**

To check whether any graph contains an Euler trail or not,

- Just make sure that the number of vertices in the graph with odd degree are not more than 2.
- If the number of vertices with odd degree are at most 2, then graph contains an Euler trail otherwise not.

### **Note-05:**

- A graph will definitely contain an Euler trail if it contains an Euler circuit.
- A graph may or may not contain an Euler circuit if it contains an Euler trail.

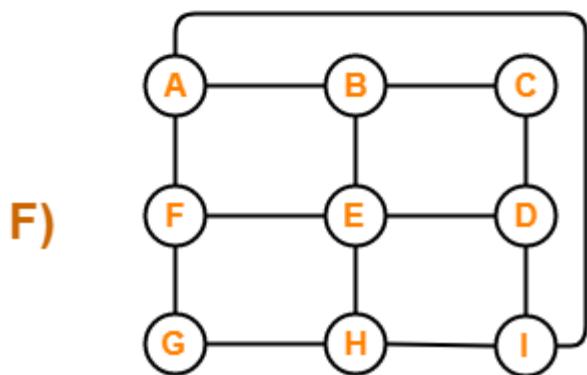
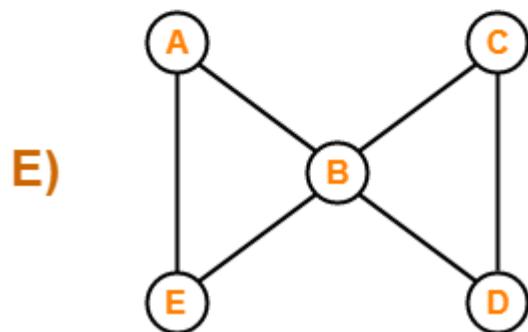
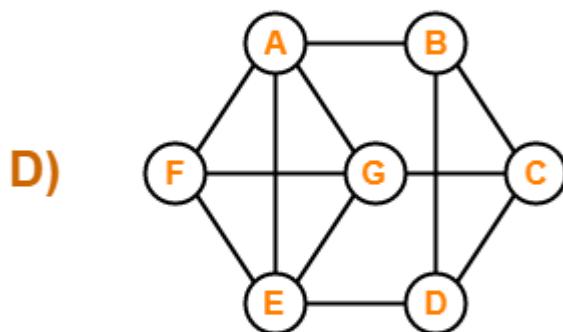
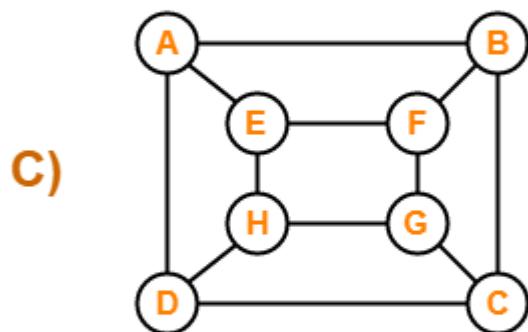
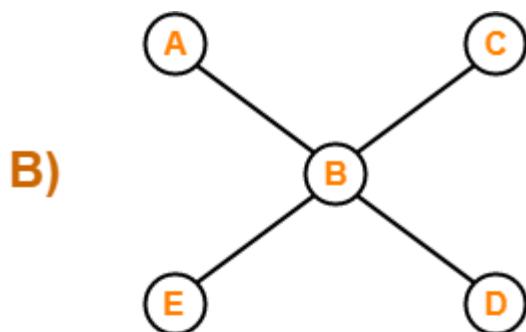
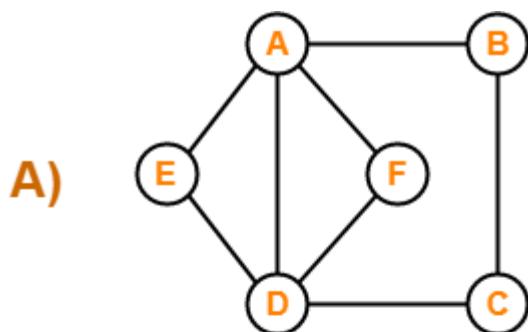
### **Note-06:**

- An Euler graph is definitely be a semi-Euler graph.
- But a semi-Euler graph may or may not be an Euler graph.

## **PRACTICE PROBLEMS BASED ON EULER GRAPHS IN GRAPH THEORY-**

### **Problems-**

Which of the following is / are Euler Graphs?



## **Solutions-**

If all the vertices of a graph are of even degree, then graph is an Euler Graph otherwise not.

Using the above rule, we have-

- A) It is an Euler graph.
- B) It is not an Euler graph.
- C) It is not an Euler graph.
- D) It is not an Euler graph.
- E) It is an Euler graph.
- F) It is not an Euler graph.

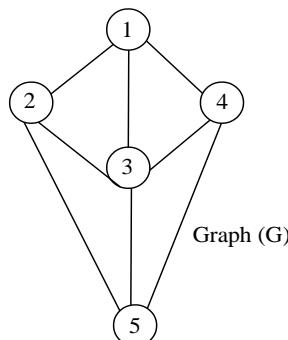
## Hamiltonian Cycle

### Hamiltonian Cycle

Let  $G = (V', E)$  be a connected graph with  $n$  vertices. A Hamiltonian cycle is a round-trip path along  $n$  edges of  $G$  that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex  $V_i \in G$  and the vertices of  $G$  are visited in the order ' $01, 112, \dots, V_{n+1}$ ', then the edges  $(V_i, V_{i+1})$  are in  $E$ ,  $1 \leq i \leq n$ , and the ' $V_i$ ' are distinct except for  $V_1$  and  $V_{n+1}$ , which are equal.

Given a graph  $G = (V, E)$  we have to find the Hamiltonian circuit using backtracking approach, we start our search from any arbitrary vertex, say  $x$ . This vertex ' $x$ ' becomes the root of our implicit tree. The next adjacent vertex is selected on the basis of alphabetical / or numerical order. If at any stage an arbitrary vertex, say ' $y$ ' makes a cycle with any vertex other than vertex ' $o$ ' then we say that dead end is reached. In this case we backtrack one step and again the search begins by selecting another vertex. It should be noted that, after backtracking the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

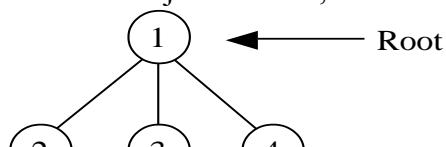
Example: Consider a graph  $G = (V, E)$ , we have to find the Hamiltonian circuit using backtracking method.



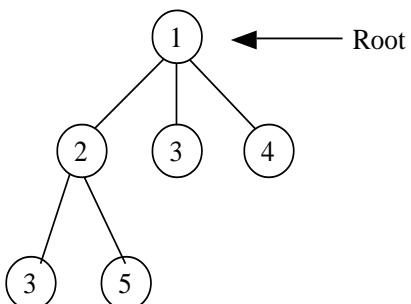
Solution: Initially we start our search with vertex '1' the vertex '1' becomes the root of our implicit tree.



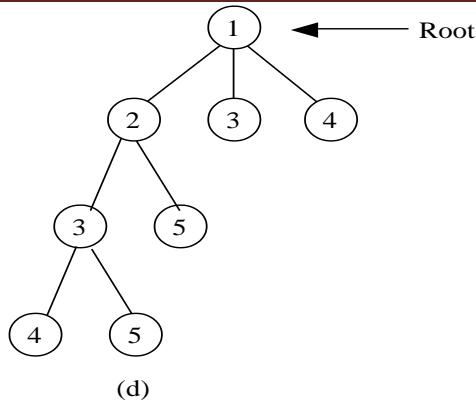
Next we choose vertex '2' adjacent to '1', as it comes first in numerical order (2, 3, 4).



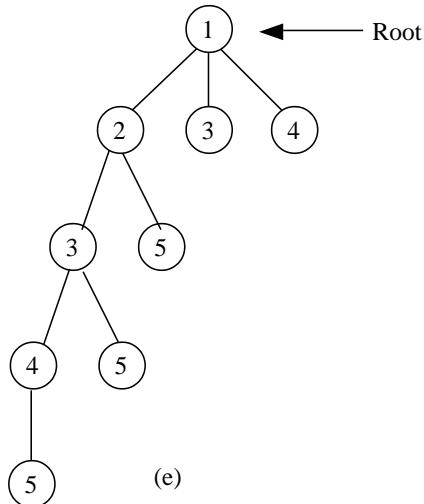
Next vertex '3' is selected which is adjacent to '2' and which comes first in numerical order (3, 5).



Next we select vertex '4' adjacent to '3' which comes first in numerical order (4, 5).



Next vertex '5' is selected. If we choose vertex '1' then we do not get the Hamiltonian cycle.

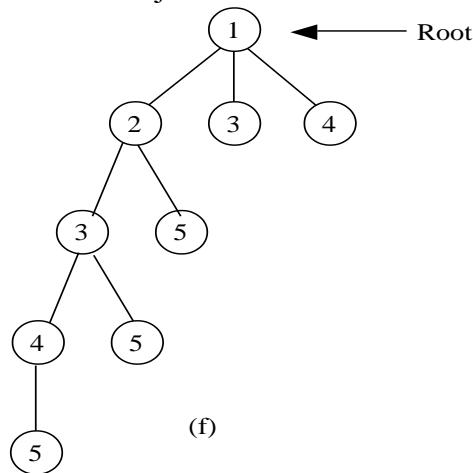


Dead end

The vertex adjacent to 5 is 2, 3, 4 but they are already visited. Thus, we get the dead end. So, we backtrack one step and remove the vertex '5' from our partial solution.

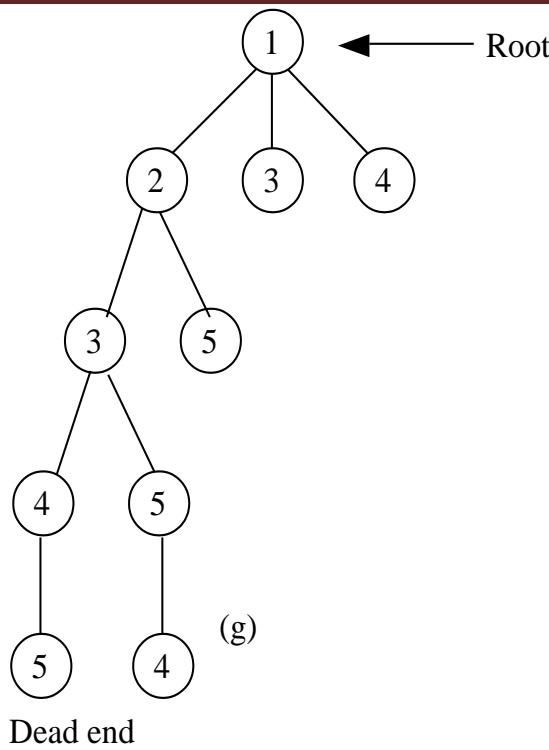
The vertex adjacent to '4' are 5, 3, 1 from which vertex '5' has already been checked and we are left with vertex '1' but by choosing vertex '1' we do not get the Hamiltonian cycle. So, we again backtrack one step.

Hence we select the vertex '5' adjacent to '3'.



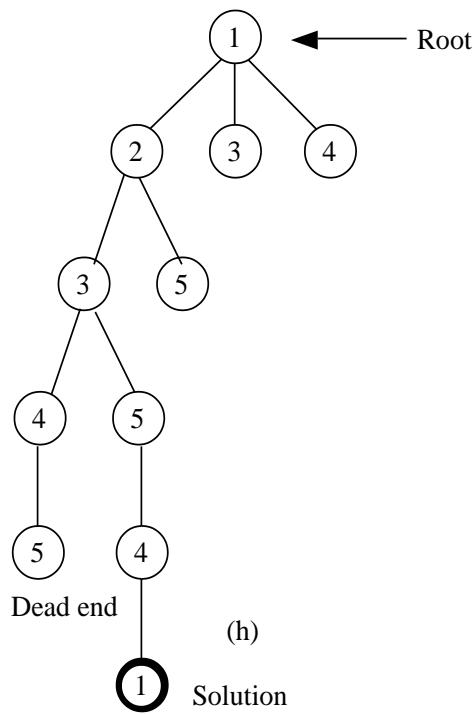
Dead end

The vertex adjacent to '5' are (2,3,4) so vertex 4 is selected.

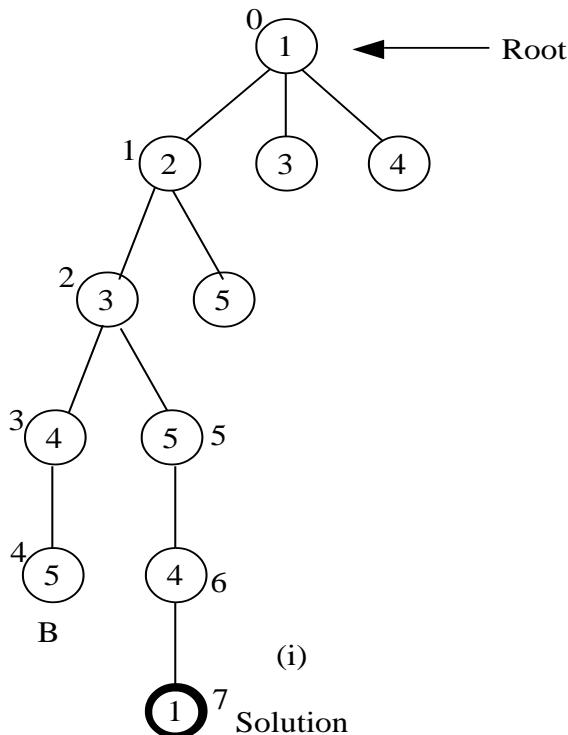


Dead end

The vertex adjacent to '4' are (1, 3, 5) so vertex '1' is selected. Hence we get the Hamiltonian cycle as all the vertex other than the start vertex '1' is visited only once, 1- 2- 3- 5- 4- 1.



The final implicit tree for the Hamiltonian circuit is shown below. The number above each node indicates the order in which these nodes are visited.



### **Fig Construction of Hamilton Cycle using Backtracking**

```

1 Algorithm Hamiltonian( $k$ )
2 // This algorithm uses the recursive formulation of
3 // backtracking to find all the Hamiltonian cycles
4 // of a graph. The graph is stored as an adjacency
5 // matrix  $G[1 : n, 1 : n]$ . All cycles begin at node 1.
6 {
7     repeat
8         { // Generate values for  $x[k]$ .
9             NextValue( $k$ ); // Assign a legal next value to  $x[k]$ .
10            if ( $x[k] = 0$ ) then return;
11            if ( $k = n$ ) then write ( $x[1 : n]$ );
12            else Hamiltonian( $k + 1$ );
13        } until (false);
14    }

```

### Algorithm: Finding all Hamiltonian cycles

---

```

1  Algorithm NextValue( $k$ )
2  //  $x[1 : k - 1]$  is a path of  $k - 1$  distinct vertices. If  $x[k] = 0$ , then
3  // no vertex has as yet been assigned to  $x[k]$ . After execution,
4  //  $x[k]$  is assigned to the next highest numbered vertex which
5  // does not already appear in  $x[1 : k - 1]$  and is connected by
6  // an edge to  $x[k - 1]$ . Otherwise  $x[k] = 0$ . If  $k = n$ , then
7  // in addition  $x[k]$  is connected to  $x[1]$ .
8  {
9      repeat
10     {
11          $x[k] := (x[k] + 1) \bmod (n + 1)$ ; // Next vertex.
12         if ( $x[k] = 0$ ) then return;
13         if ( $G[x[k - 1], x[k]] \neq 0$ ) then
14             { // Is there an edge?
15                 for  $j := 1$  to  $k - 1$  do if ( $x[j] = x[k]$ ) then break;
16                     // Check for distinctness.
17                 if ( $j = k$ ) then // If true, then the vertex is distinct.
18                     if (( $k < n$ ) or (( $k = n$ ) and  $G[x[n], x[1]] \neq 0$ ))
19                         then return;
20             }
21     } until (false);
22 }
```

Algorithm: Generating a next vertex

# ALL CUTSETS IN A GRAPH

## 4-2. SOME PROPERTIES OF A CUT-SET

Consider a spanning tree  $T$  in a connected graph  $G$  and an arbitrary cutset  $S$  in  $G$ . Is it possible for  $S$  not to have any edge in common with  $T$ ? The answer is *no*. Otherwise, removal of the cut-set  $S$  from  $G$  would not disconnect the graph. Therefore,

THEOREM 4-1

Every cut-set in a connected graph  $G$  must contain at least one branch of every spanning tree of  $G$ .

Will the converse also be true? In other words, will any minimal set of edges containing at least one branch of every spanning tree be a cut-set? The answer is *yes*.

THEOREM 4-2

In a connected graph  $G$ , any minimal set of edges containing at least one branch of every spanning tree of  $G$  is a cut-set.

THEOREM 4-3

Every circuit has an even number of edges in common with any cut-set.

## 4-3. ALL CUTSETS IN A GRAPH

In [Section 4-1](#) it was shown how cutsets are used to identify weak spots in a communication net. For this purpose we list all cutsets of the corresponding graph, and find which ones have the smallest number of edges. It must also have become apparent to you that even in a simple example, there is a large number of cutsets, and we must have a systematic method of generating all relevant cutsets.

In the case of circuits, we solved a similar problem by the simple technique of finding a set of **fundamental circuits** and then realizing that other circuits in a graph are just *combinations* of two or more fundamental circuits. We shall follow a similar strategy here. Just as a spanning tree is essential for defining a set of fundamental circuits, so is a spanning tree essential for a set of *fundamental cutsets*. It will be beneficial for us to look for the parallelism between circuits and cutsets.

## Fundamental CutSets:

Consider a spanning tree T of a connected graph G. Take any branch b in T. Since  $\{b\}$  is a cut-set in T,  $\{b\}$  partitions all vertices of T into two disjoint sets—one at each end of b.

Consider the same partition of vertices in G, and the cut set S in G that corresponds to this partition. Cutset S will contain only one branch b of T, and the rest (if any) of the edges in S are chords with respect to T. Such a cut-set S containing exactly one branch of a tree T is called a **fundamental cut-set** with respect to T. Sometimes a fundamental cut-set is also called a **basic cut-set**.

In Fig. 4-3, a spanning tree T (in heavy lines) and all five of the fundamental cutsets with respect to T are shown (broken lines “cutting” through each cut-set).

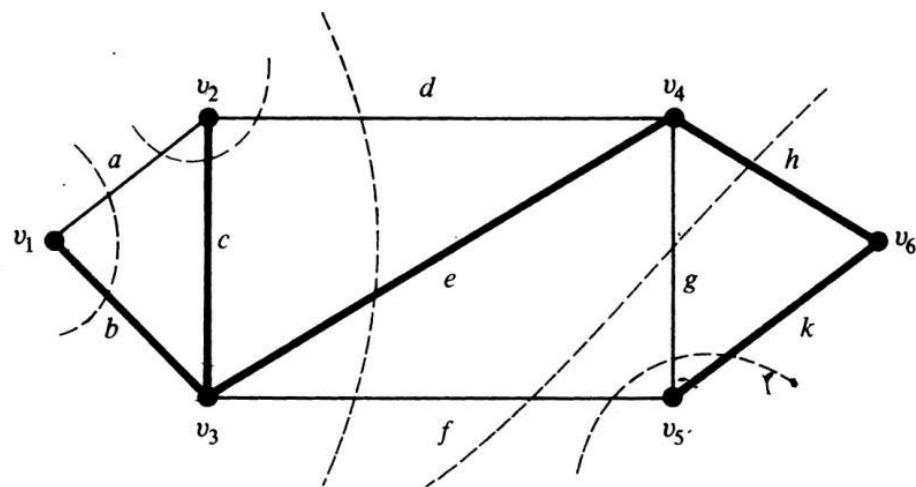


Fig. 4-3 Fundamental cutsets of a graph.

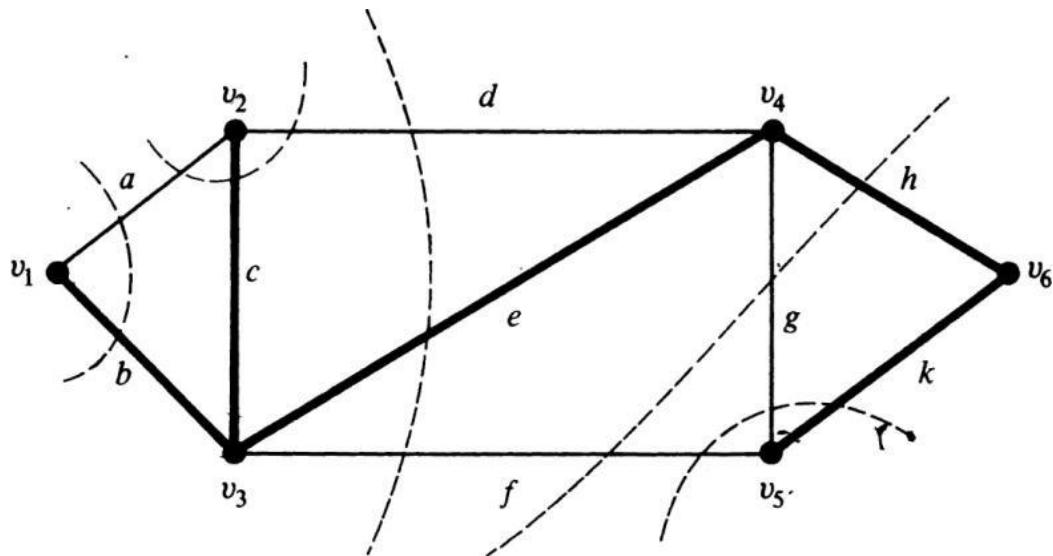
Just as every chord of a spanning tree defines a **unique fundamental circuit**, every branch of a spanning tree defines a **unique fundamental cut-set**. It must also be kept in mind that the term fundamental cut-set (like the term fundamental circuit) has meaning only with respect to a given spanning tree.

Now we shall show how other cutsets of a graph can be obtained from a given set of cutsets.

#### THEOREM 4-4

The ring sum of any two cutsets in a graph is either a third cut-set or an edge-disjoint union of cutsets.

*Example:* In Fig. 4-3 let us consider ring sums of the following three pairs of cutsets.



$$\{d, e, f\} \oplus \{f, g, h\} = \{d, e, g, h\}, \quad \text{another cut-set,}$$

$$\{a, b\} \oplus \{b, c, e, f\} = \{a, c, e, f\}, \quad \text{another cut-set,}$$

$$\begin{aligned} \{d, e, g, h\} \oplus \{f, g, k\} &= \{d, e, f, h, k\} \\ &= \{d, e, f\} \cup \{h, k\}, \text{ an edge-disjoint} \\ &\quad \text{union of cut-sets. } \blacksquare \end{aligned}$$

# Planar Graphs



# Scope of the lecture

- *Characterisation of Planar Graphs:* First we introduce planar graphs, and give its characterisation and some simple properties.



# Scope of the lecture

- *Characterisation of Planar Graphs:* First we introduce planar graphs, and give its characterisation and some simple properties.
- *Planarity Testing:* Next, we give an algorithm to test planarity.



# Scope of the lecture

- *Characterisation of Planar Graphs:* First we introduce planar graphs, and give its characterisation and some simple properties.
- *Planarity Testing:* Next, we give an algorithm to test planarity.
- *Planar Embedding:* Lastly we see how a given planar graph can be embedded in a plane.  
We also explore straight line embeddings.



# What is a Drawing?

## Definition (Drawing)

Given a graph  $G = (V, E)$ , a drawing maps each vertex  $v \in V$  to a distinct point  $\Gamma(v)$  in plane, and each edge  $e \in E$ ,  $e = (u, v)$  to a simple open jordan curve  $\Gamma(u, v)$  with end points  $\Gamma(u), \Gamma(v)$ .

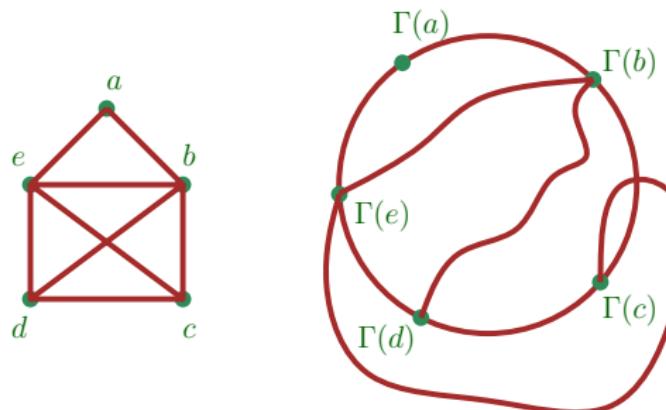


Figure: Drawing of a graph



# What is a Planar Graph?

## Definition (Planar Graphs)

Given a graph  $G = (V, E)$ ,  $G$  is planar if it admits a drawing such that no two distinct drawn edges intersect except at end points.

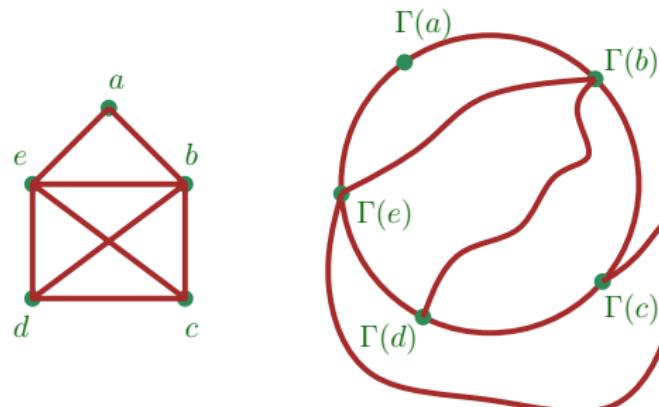


Figure: Planar drawing of a graph



# Motivation



# Properties of Planar Graphs

There are number of interesting properties of planar graphs.

- They are sparse. Their size including faces, edges and vertices is  $O(n)$ .
- They are 4-colourable.
- A number of operations can be performed on them very efficiently. Since there is a topological order to the incidences.
- They can be efficiently stored (A data structure called *SPQR*-trees even allows  $O(1)$  flipping of planar embeddings).



# Applications of Planar Graphs

Planar graphs are extensively used in Electrical and Civil engineering.

- Easy to visualize. In fact, crossings reduces comprehensibility.  
So all good graph drawing tools use planar graphs.
- VLSI design, circuit needs to be on surface: lesser the crossings, better is the design.
- Highspeed Highways/Railroads design, crossings are always problematic.
- Irrigation canals, crossings simply not admissible.
- Determination of *isomorphism* of chemical structures.
- Most of facility location problems on maps are actually problems of planar graphs.



# Problem Definition: Planarity Testing

Problem (Decision Problem)

*Given a graph  $G = (V, E)$ , **is  $G$  planar**, i.e., can  $G$  be drawn in the plane without edge crossings?*



# Problem Definition: Planarity Embedding

## Problem (Computation Problem)

*Given a graph  $G = (V, E)$ , if  $G$  is planar, how can  $G$  be drawn in the plane such that there are no edge crossings? I.e., compute a planar representation of the graph  $G$ .*



# Question: $K_4$ ?

Is the following graph ( $K_4$ ) planar?

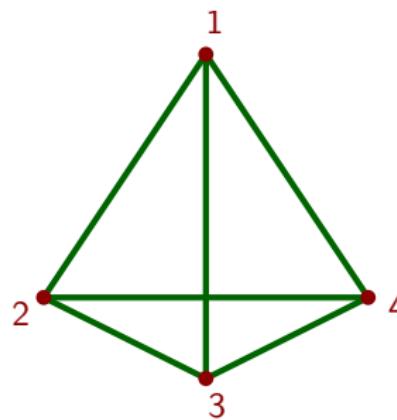


Figure: Graph  $K_4$

Answer:  $K_4$  is planar

Yes,  $K_4$  is planar!

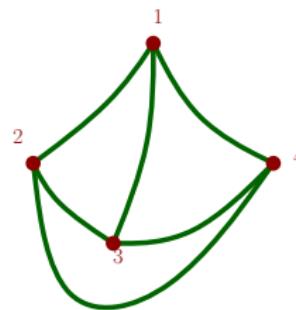


Figure: Planar  $K_4$

Question:  $K_5$  and  $K_{3,3}$ ?

Are the following graphs ( $K_5$  and  $K_{3,3}$ ) planar?

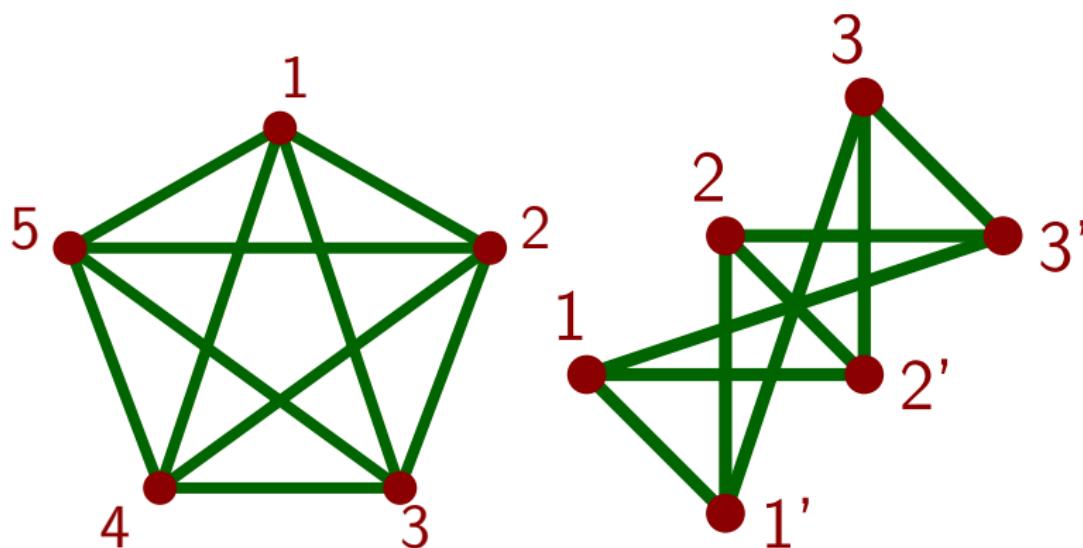


Figure: Graphs  $K_5$  and  $K_{3,3}$

Answer:  $K_4$  is planar

No!! They aren't. There always will be at least one crossing.

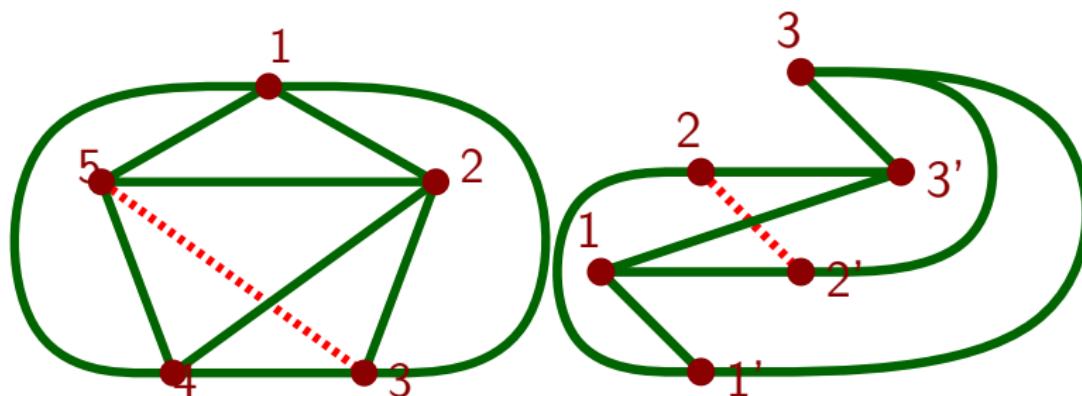


Figure: Non-planarity of  $K_5$  and  $K_{3,3}$

Full proofs by Euler's celebrated theorem.



# Question: Is a given graph planar?

Is the following graph planar?

There are a lot of crossings  $O(n^2)$ .

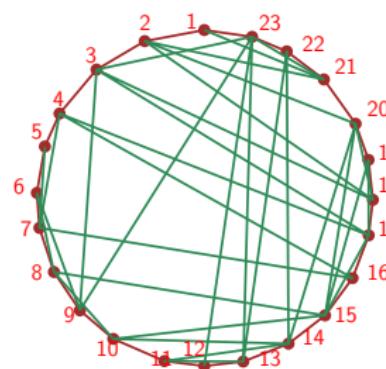


Figure: A Hamiltonian Graph



# Answer: Yes

Yes, it is.

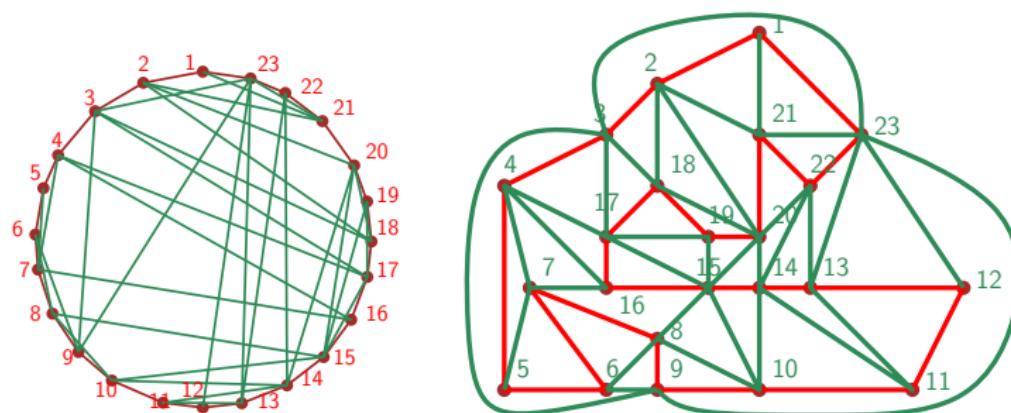


Figure: Planar embedding of the given graph

But, how to arrive at this answer? It is tough.

# Characterisation of Planar Graphs



# Basic Assumptions

Euler's formula gives the necessary condition for a graph to be planar[3].

We assume that our graphs are connected and there are no self loops and no multi-edges.

Disconnected graphs, 1-degree vertices, multi-edges can be easily dealt with.

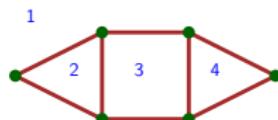


# Euler's Relation for nodes, arcs and faces<sup>1</sup>

## Theorem (Euler's Relation)

*Given a planar graph with  $n$  vertices,  $m$  edges and  $f$  faces:*

$$n - m + f = 2$$



A Planar Graph :

$$n = |V| = 6,$$

$$m = |E| = 8, \text{ and}$$

$$f = |F| = 4$$

$$6 - 8 + 4 = 2$$

---

<sup>1</sup>The exterior is also counted as a face. The above relation also applies to simple polyhedrons with no holes.

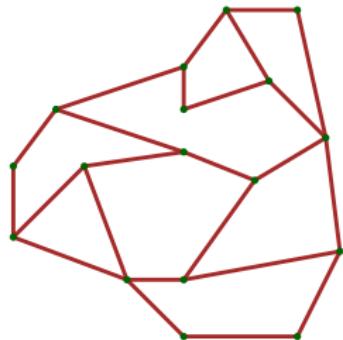


# Euler's Relation for nodes, arcs and faces

## Theorem (Euler's Relation)

Given a planar graph with  $n$  vertices,  $m$  edges and  $f$  faces:

$$n - m + f = 2$$



A Planar Graph :

$$n = |V| = 17,$$

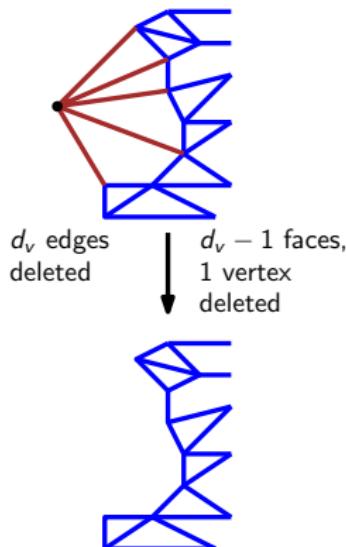
$$m = |E| = 24, \text{ and}$$

$$f = |F| = 9$$

$$17 - 24 + 9 = 2$$



# Euler's Relation for nodes, arcs and faces



*Proof:* We prove Euler's Relation by Mathematical Induction on vertices.

If we remove one vertex  $v$  on boundary (don't delete a cut vertex) with  $d_v$  edges, then we are removing  $d_v - 1$  faces.

Thus the invariant  $n - m + f = 2$  is maintained.

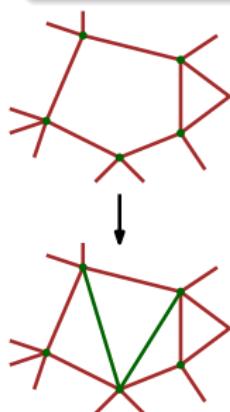
Basis is an isolated vertex,  
 $n = f = 1, m = 0. \square$



# Euler's Relation: Corollary 1

## Corollary

*For a maximal planar graph, where each face is a triangle,  $m = 3n - 6$ , and therefore, for any planar graph with at least three vertices, we should have:  $m \leq 3n - 6$ .*



*Proof:*  $\sum_{f \in F} e_f = 2m$  and therefore since

$$e_f \geq 3 \Rightarrow \sum_{f \in F} e_f \geq 3f \Rightarrow 2m \geq 3f.$$

Substituting in  $n - m + f = 2$  gives us  $n - m + 2m/3 \geq 2 \square$



# Euler's Relation: Corollary 1

Example to show  $m \leq 3n - 6$ .

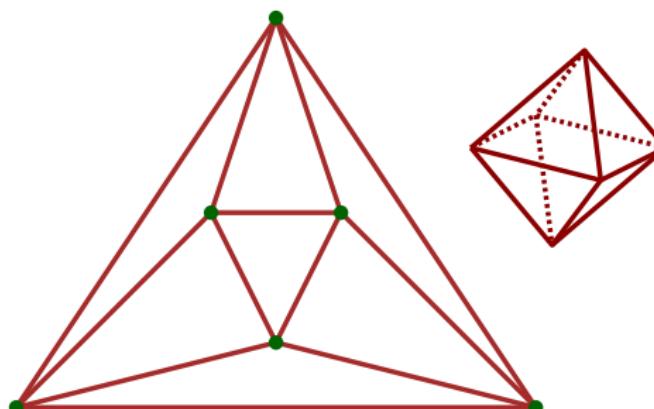


Figure: Octahedron:  $n = 6, m = 12, m \leq 3n - 6$



# Non-planarity of $K_5$

Earlier we said that  $K_5$  is non-planar.

## Lemma

$K_5$  is non-planar.

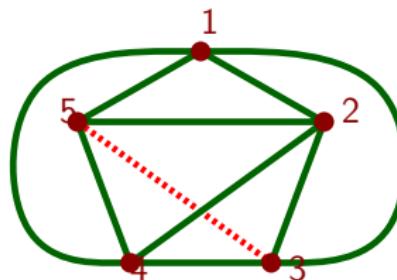


Figure: Proof:  $n = 5, m = 10, m > 3n - 6 (= 9)$   $\square$



## Euler's Relation: Corollary 2

### Corollary

For a planar graph, where no face is a triangle:  $m \leq 2n - 4$ .

*Proof:*  $\sum_{f \in F} e_f = 2m$  and therefore since

$$e_f \geq 4 \Rightarrow \sum_{f \in F} e_f \geq 4f \Rightarrow 2m \geq 4f.$$

Substituting in  $n - m + f = 2$  gives us  $n - m + m/2 \geq 2 \quad \square$

Above relation is true for bi-partite planar graphs and graphs with no 3-cycle.



## Euler's Relation: Corollary 2

Example to show  $m \leq 2n - 4$  for graphs without triangles.

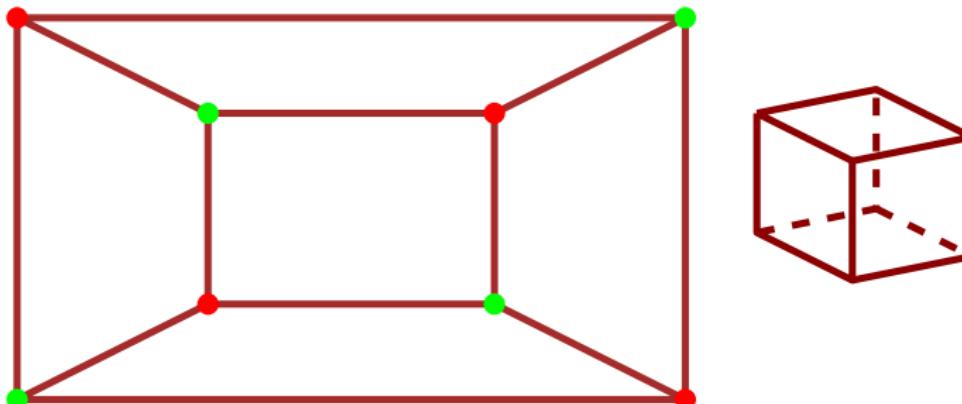


Figure: Cube :  $n = 8, m = 12, m \leq 2n - 4$



# Non-planarity of $K_{3,3}$

Earlier we said that  $K_{3,3}$  is non-planar.

## Lemma

$K_{3,3}$  is non-planar.

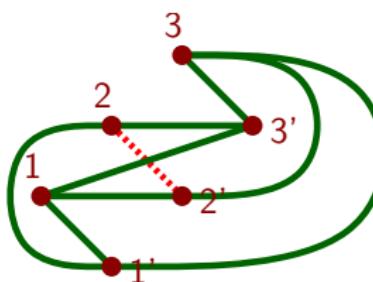


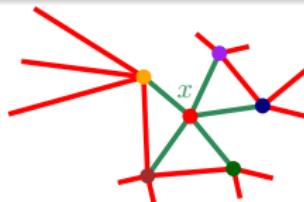
Figure: Proof:  $n = 6, m = 9, m > 2n - 4 (= 8)$   $\square$



# Euler's Relation: Corollary 3

## Corollary

*Any planar graph is 6 colourable.*



*Proof:*

- Since  $m \leq 3n - 6$ , there exists a vertex with degree less than 6 (otherwise  $\sum_v d_v = 2m \Rightarrow 2m \geq 6n$ ).
- By induction, if we remove this vertex, resulting graph is 6-colourable.
- Just give this vertex a colour other than the five colours of the neighbours.



# Euler's Relation: Corollary 4

## Corollary

*Any planar graph is 5 colourable.*

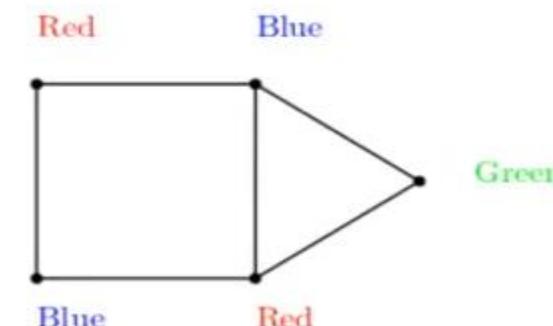
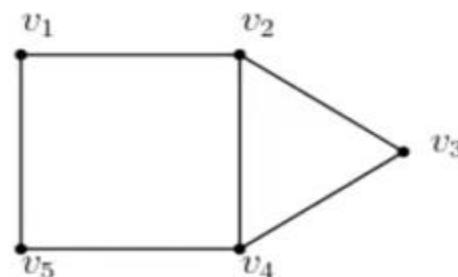


# Graph Coloring

## Definition

A *k-coloring* of a graph  $G$  is a function  $f : V(G) \rightarrow \{1, 2, \dots, k\}$  such that

$$u \sim v \implies f(u) \neq f(v).$$

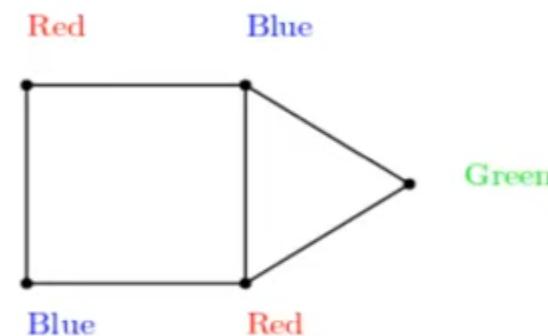
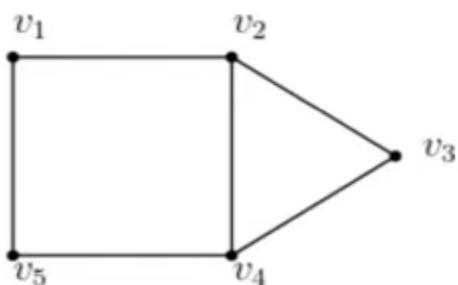


- This is a 3-coloring of  $G$ .
- This is also a 4-coloring and a 5-coloring of  $G$ . Why?
- $f : V(G) \rightarrow \{1, 2, \dots, k\}$  does not have to be onto.

# Chromatic Number

## Definition

The **chromatic number** of a graph  $G$ , denoted  $\chi(G)$ , is the *minimum  $k$*  such that  $G$  has a  $k$ -coloring.



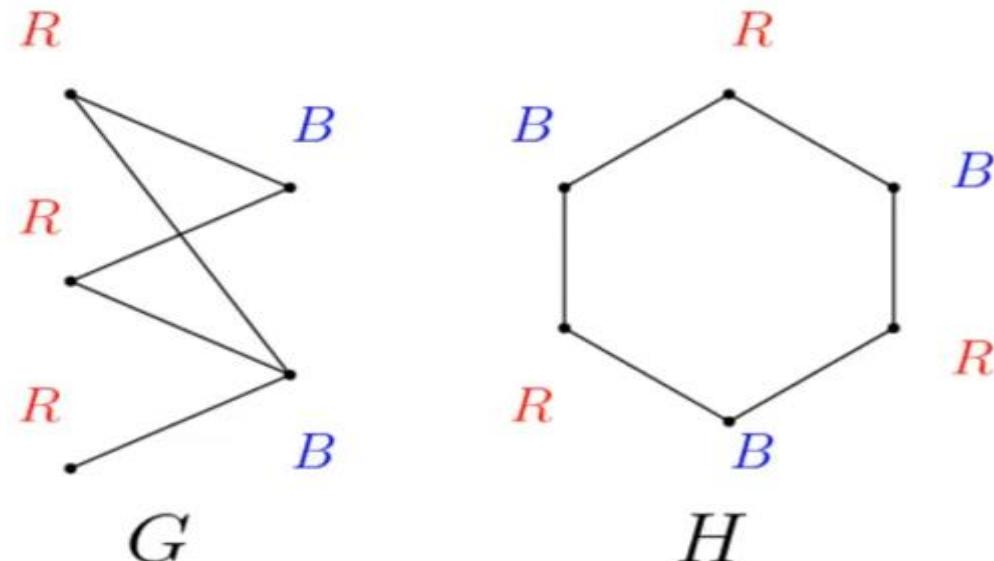
- $G$  has a 3-coloring:  $\chi(G) \leq 3$
- $G$  does not have a 2-coloring:  $v_2, v_3, v_4$  all must receive different colors
- $\chi(G) = 3$

# Chromatic Number

- Which graphs  $G$  have  $\chi(G) = 1$ ? If  $G$  has any edge then  $G$  needs at least two colors to color the vertices incident to the edge.

$\chi(G) = 1 \text{ if and only if } G \text{ has no edges}$

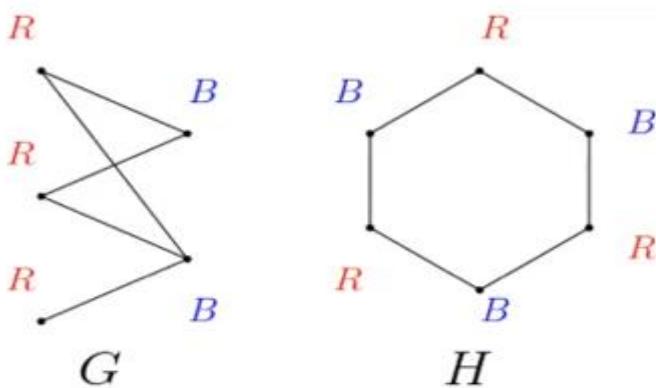
- Which graphs  $G$  have  $\chi(G) = 2$ ?



# Bipartite Graphs

## Definition

A graph  $G$  is **bipartite** if  $G$  is 2-colorable.



A few notes:

- Why the word *bipartite*? It comes from “bi” and “partite” which seems to mean two parts. The two parts are the vertices of one color and the vertices of the other color.
- If  $X$  is the set of vertices of one color and  $Y$  is the set of vertices of the other color, we usually say “ $G$  is a bipartite graph with bipartition  $V(G) = X \cup Y$ ”.
- If  $G$  is bipartite with bipartition  $V(G) = X \cup Y$  then  $X$  and  $Y$  are independent sets.

## Bipartite: Examples/Non-Examples

- Examples of Bipartite Graphs: Trees, even cycles
- **Complete Bipartite Graphs:**

### Definition

Let  $n, m$  be positive integers. The **complete bipartite graph**  $K_{m,n}$ , is a bipartite graph with bipartition  $X \cup Y$  where:

- $|X| = n, |Y| = m$
- $E(K_{m,n}) = \{(x, y) : x \in X, y \in Y\}$

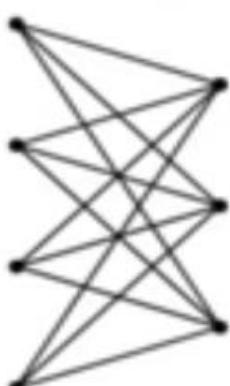
# Complete Bipartite Graph

## Definition

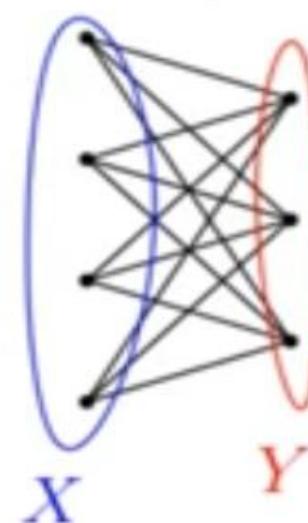
Let  $n, m$  be positive integers. The **complete bipartite graph**  $K_{m,n}$ , is a bipartite graph with bipartition  $X \cup Y$  where:

- $|X| = n, |Y| = m$
- $E(K_{m,n}) = \{(x, y) : x \in X, y \in Y\}$

$K_{4,3}$



$K_{4,3}$



# Bipartite Graphs & Odd Cycles

## Proposition

*If  $G$  is bipartite then  $G$  does not have an odd cycle.*



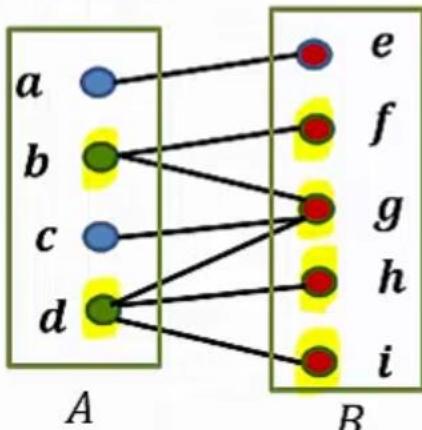
**Proof:** Suppose  $G$  has an odd cycle  $C$  in it. Then from last class  $\chi(C) = 3$  and  $\chi(C) \leq \chi(G)$  because  $C$  is a subgraph of  $G$ . Altogether this gives us

$$3 = \chi(C) \leq \chi(G)$$

so  $G$  is not 2-colorable, so  $G$  is not bipartite. Therefore if  $G$  is bipartite,  $G$  does not have an odd cycle.

## Matching in Bipartite Graphs

Given a bipartite graph  $G$  with  $A$  and  $B$  as its two groups of vertices. Let  $S \subseteq A$ .  $N(S)$  is the set of all the neighbors of vertices in  $S$ . That is  $N(S)$  contains all the vertices which are adjacent to at least one of the vertices in  $S$ .



$$S = \{a, b\}$$

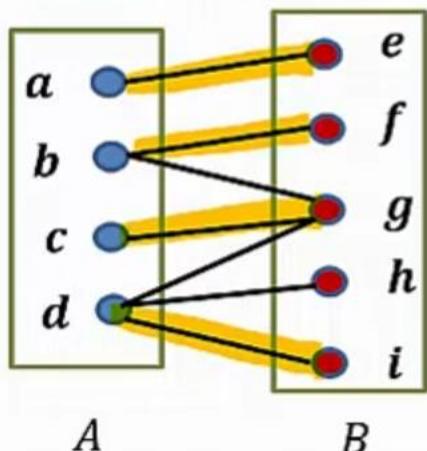
$$N(S) = \{e, f, g\}$$

$$S = \{b, d\}$$

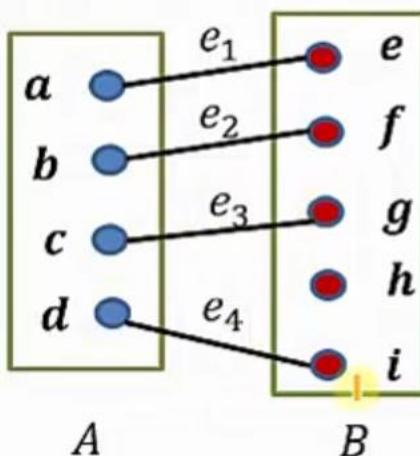
$$N(S) = \{f, g, h, i\}$$

A graph is **bipartite** if the vertices can be divided into two sets,  $A$  and  $B$ , with no two vertices in  $A$  adjacent and no two vertices in  $B$  adjacent. The vertices in  $A$  can be adjacent to some or all of the vertices in  $B$ . Every edge connects a vertex in  $A$  and  $B$ .

Given a bipartite graph  $G$ , with  $A$  and  $B$  as its two groups of vertices. A **matching** of the group  $A$  is a subset of the edges of  $G$  such that each edge connects all vertices of  $A$  with exactly one vertex of  $A$  to one vertex of  $B$ .



Here is a matching of group A.

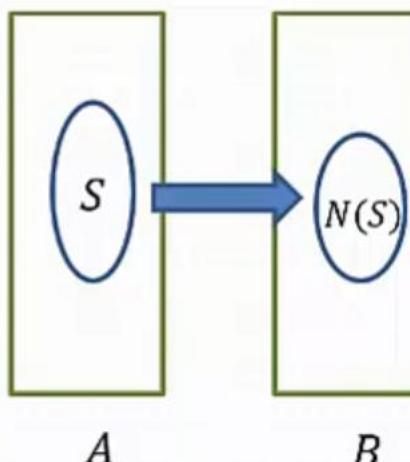


## Matching in Bipartite Graphs

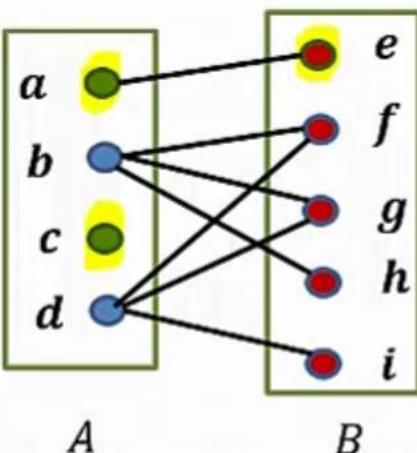
Given a bipartite graph  $G$ , with  $A$  and  $B$  as its two groups of vertices. A **matching** of the group  $A$  is a subset of the edges of  $G$  such that each edge connects all vertices of  $A$  with exactly one vertex of  $A$  to one vertex of  $B$ .

### Hall's Marriage Theorem

Let  $G$  be a bipartite graph with sets  $A$  and  $B$ . Then  $G$  has a matching of  $A$  if and only if  
 $|N(S)| \geq |S|$   
for all  $S \subseteq A$ .

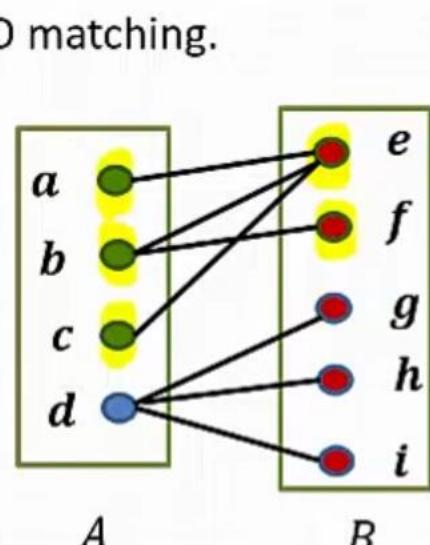


Examples of Bipartite graph with NO matching.



$$S = \{c\} \rightarrow |S| = 1 \\ N(S) = \{\quad\} \rightarrow |N(S)| = 0$$

$$S = \{a, c\} \rightarrow |S| = 2 \\ N(S) = \{e\} \rightarrow |N(S)| = 1$$



$$S = \{a, b, c\} \rightarrow |S| = 3 \\ N(S) = \{e, f\} \rightarrow |N(S)| = 2$$

$$|N(S)| \geq |S|$$

$$S \subseteq A$$

- $N(S)$  is the image of  $S$ .
- $N(S)$  is the set of all the neighbors of vertices in  $S$ .
- $N(S) \subseteq B$



# **String Matching Algorithms**



# Applications 1

- BioInformatics
- DNA sequencing

AGTAGGGTAGG	CAGTGATAGAT	AAATTTCGCG
TAGGTCA GTGA		TTTCGCGCTAT
	TAGATAGAAAT	GCGCTATCGAT
<hr/>		
AGTAGGGTAGGTCA GTGATAGATAGAAATTTCGCGCTATCGAT		

# Applications 2

## • Web pages search engine

Chrome File Edit View History Bookmarks People Window Help

string matching algorithms - Google

https://www.google.com/search?q=string+matching+algorithms&oq=string+matching+algorithms&aqs=chrome..69i57j69i60l3.8891j0j4&so...

string matching algorithms

string matching **algorithm with example**  
string matching **algorithm pdf**  
string matching **algorithm geeksforgeeks**  
**naive** string matching **algorithm**  
**kmp algorithm explained**  
**pattern matching algorithm in c**  
**boyer-moore string-search algorithm**  
string matching **algorithm ppt**  
Speeding up two **string-matching algorithms** - Crochemore - Cited by 259

Report inappropriate predictions

String-searching algorithm - Wikipedia  
[https://en.wikipedia.org/wiki/String-searching\\_algorithm](https://en.wikipedia.org/wiki/String-searching_algorithm)

String-searching algorithm. In computer science, **string-searching algorithms**, sometimes called **string-matching algorithms**, are an important class of **string algorithms** that try to find a place where one or several **strings** (also called **patterns**) are found within a larger **string** or **text**. Let  $\Sigma$  be an alphabet (finite set).

Boyer-Moore string-search · Knuth-Morris-Pratt algorithm · Rabin-Karp algorithm

KMP Algorithm for Pattern Searching - GeeksforGeeks  
<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

Searching **Algorithm**: Unlike Naive **algorithm**, where we slide the **pattern** by one and compare all characters at each shift, we use a value from `lps[]` to decide the next characters to be matched. The idea is to not **match** a character that we know will anyway **match**.

See string matching algorit... Sponsored

Vibia Algorithm Grid Multilight Pendant by Toan Nguyen | 60" H x 78" W | Metal | 105... \$12,600.00 WeLivv Free shipping

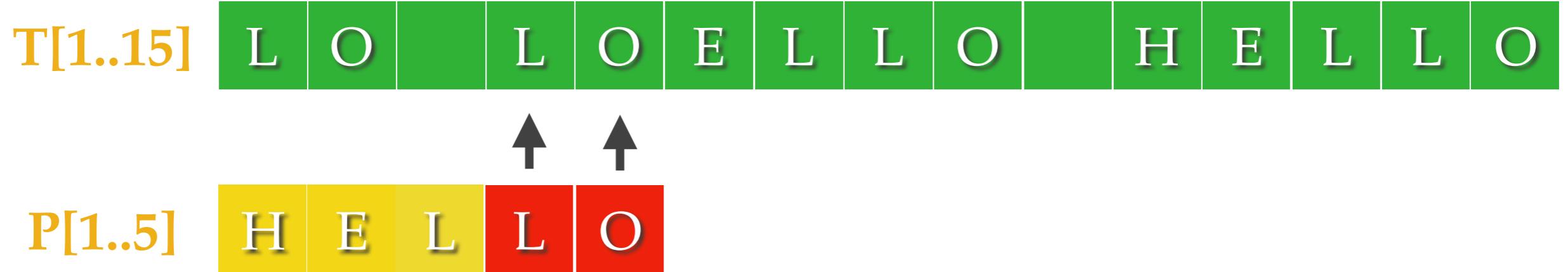
Vibia Algorithm XS Multilight Pendant by Toan Nguyen | 52" H x 78" W | 105% Price... \$4,989.00 WeLivv Free shipping

More on Google

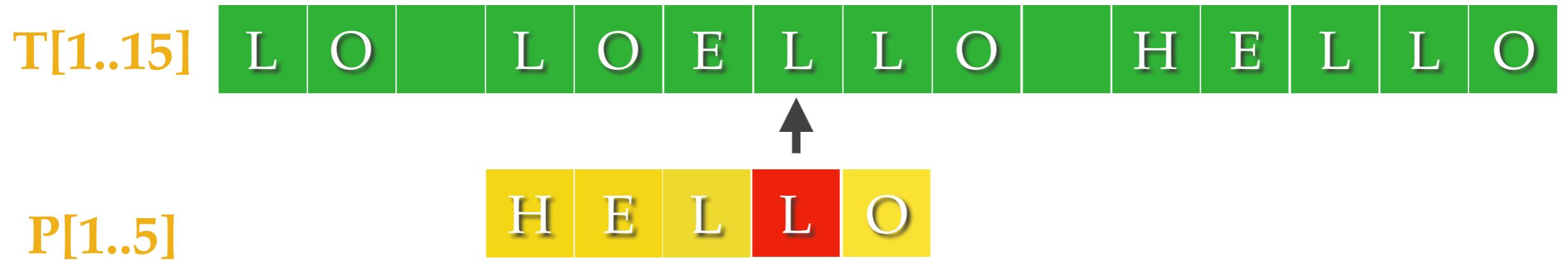
# Formalize String Matching Problem

- A text as an array of characters  $T[1..n]$
- A pattern as an array of characters  $P[1..m]$
- $m \leq n$
- The characters  $\Sigma = \{a, b, \dots, z\}$   $\Sigma = \{0, 1\}$

# Formalize String Matching Problem



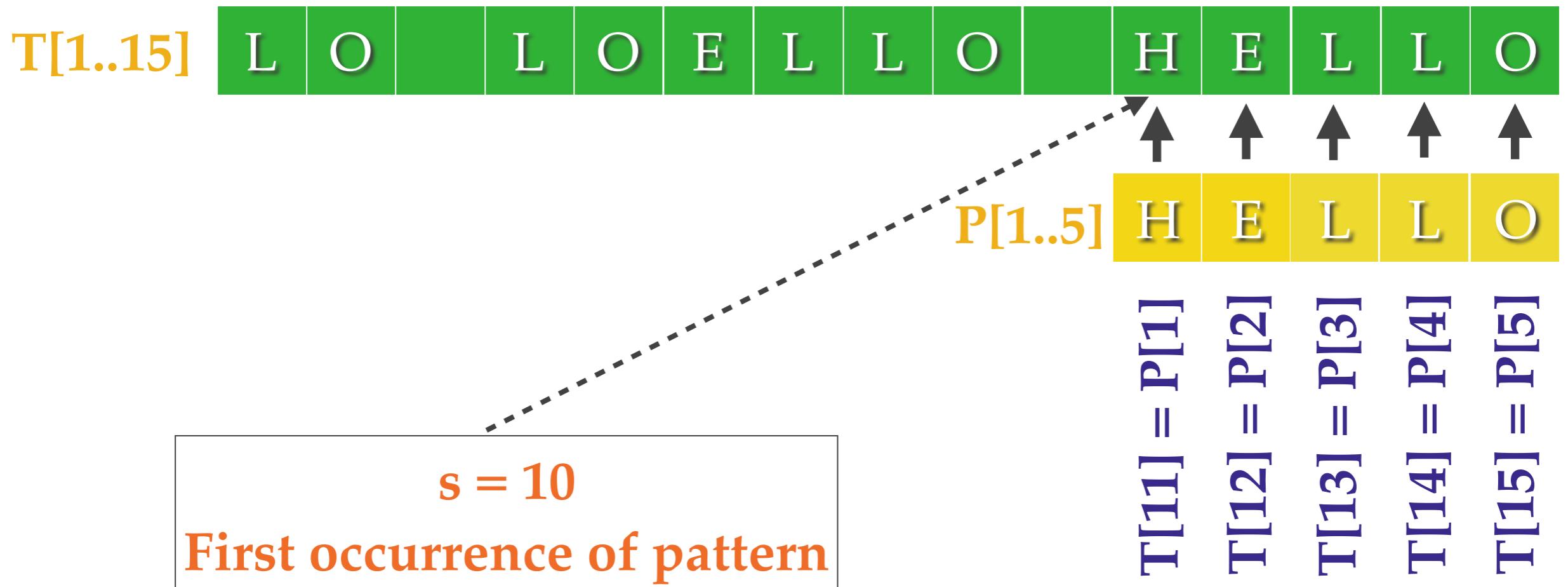
# Formalize String Matching Problem



# Formalize String Matching Problem



# String Matching Problem



# Naive String MatchingAlgorithm

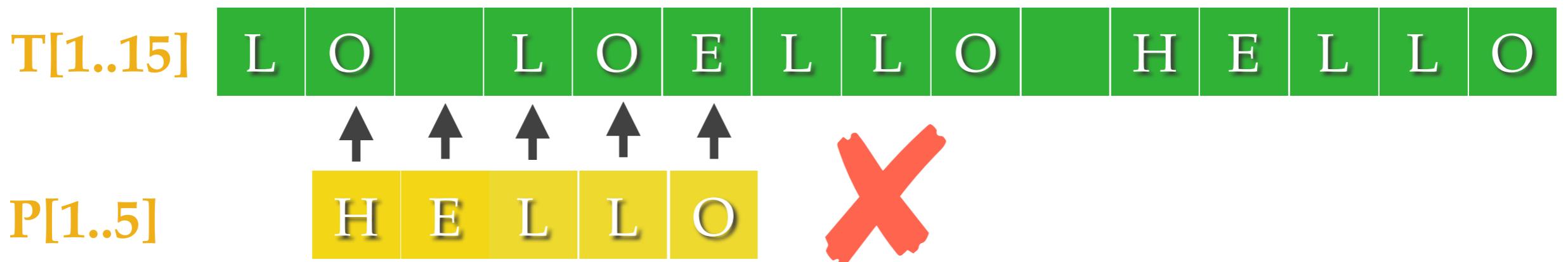
Check P with each substring of T for all possible shifts



for s=0 test  $T[1..5] = P[1..5]$

# Naive String MatchingAlgorithm

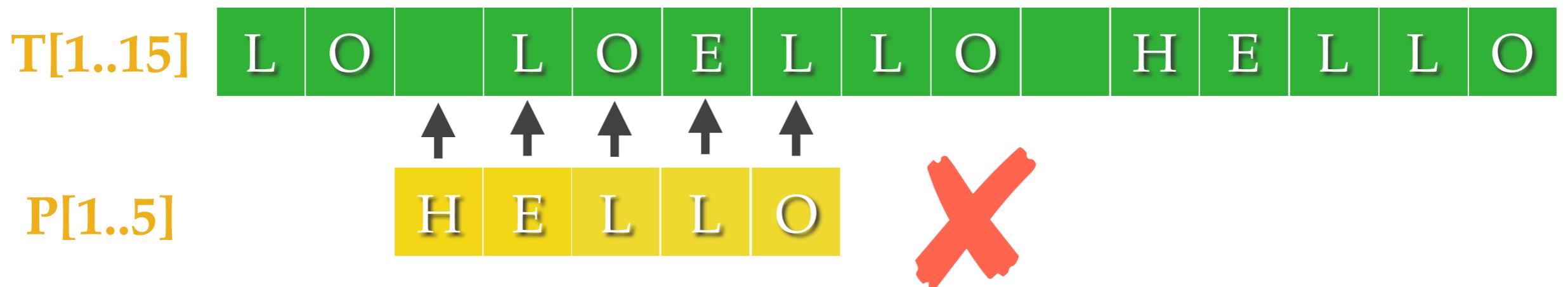
check P with each substring of T for all possible shifts



for s=1 test  $T[2..5+1] = P[1..5]$

# Naive String MatchingAlgorithm

check P with each substring of T for all possible shifts



for s=0 test  $T[3..5+2] = P[1..5]$

# Naive String MatchingAlgorithm

check P with each substring of T for all possible shifts



for s=0 test  $T[11..5+10] = P[1..5]$



# Naive String MatchingAlgorithm

**NAIVE-STRING-MATCHER( $T, P$ )**

```
1   $n = T.length$ 
2   $m = P.length$ 
3  for  $s = 0$  to  $n - m$ 
4      if  $P[1 .. m] == T[s + 1 .. s + m]$ 
5          print “Pattern occurs with shift”  $s$ 
```

# Naive String Matching Algorithm

## Time Complexity

Matching time in the worst case:  $O(m(n-m+1)) \sim O(n^2)$

Text =  $a^n$



Pattern =  $a^m$



# Naive String Matching Algorithm

## Time Complexity

Matching time in the worst case:  $O(m(n-m+1)) \sim O(n^2)$

Text =  $a^n$



Pattern =  $a^m$



# Naive String Matching Algorithm

## Time Complexity

Matching time in the worst case:  $O(m(n-m+1)) \sim O(n^2)$

Text =  $a^n$



Pattern =  $a^m$



# Naive String Matching Algorithm

## Time Complexity

Matching time in the worst case:  $O(m(n-m+1)) \sim O(n^2)$

Text =  $a^n$



Pattern =  $a^m$

# Rabin-Karp String Matching Algorithm

- The Rabin-Karp algorithm calculates a **hash value** for the **pattern**, and for each **M-character subsequence** of text to be compared.
- If the hash values are **unequal**, the algorithm will calculate the **hash value for next M-character** sequence.
- If the hash values are **equal**, the algorithm will **compare** the pattern and the M-character sequence.
- In this way, there is only **one comparison per text subsequence**, and character matching is only needed when hash values match.

# Some mathematics

- Consider an **M-character sequence** as an **M-digit number** in **base  $b$** , where  $b$  is the number of letters in the alphabet. The subsequent  $t[i..i+M-1]$  is mapped to the number:

$$x(i) = t[i]^*b^{(M-1)} + t[i+1]^*b^{(M-2)} + \dots + t[i+M-1]$$

- Furthermore, given  $x(i)$  we can compute  $x(i+1)$  for the **next subsequent  $t[i+1..i+M]$**  in constant time, as follows:

$$x(i+1) = t[i+1]^*b^{(M-1)} + t[i+2]^*b^{(M-2)} + \dots + t[i+M]$$

# Some mathematics

- $x(i+1) = x(i) * b - t[i] * b^M + t[i+M]$ 
  - → Shift left one digit
  - → Subtract leftmost digit
  - → Add new rightmost digit
- We adjust the **existing value** when we move over one character
- **Constant time** to compute M-digit numbers of each M-characters subsequence

# Some mathematics

- We hash the value by taking it *mod* a prime number *q*  
The *mod function* is useful in this case:
  1.  $[(x \text{ mod } q) + (y \text{ mod } q)] \text{ mod } q = (x+y) \text{ mod } q$
  2.  $(x \text{ mod } q) \text{ mod } q = x \text{ mod } q$
- For these reasons:  
$$\text{hash}(x(i)) = ((t[i]^* b^{\wedge} (M-1) \text{ mod } q) + (t[i+1]^* b^{\wedge} (M-2) \text{ mod } q) + \dots + (t[i+M-1] \text{ mod } q)) \text{ mod } q$$
- So:  
$$h(x(i+1)) = (h(x(i)^* b \text{ mod } q - t[i]^* b^{\wedge} M \text{ mod } q + t[i+M] \text{ mod } q) \text{ mod } q$$

# Rabin-Karp String Matching Algorithm

RABIN-KARP-MATCHER( $T, P, d, q$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $h = d^{m-1} \bmod q$ 
4   $p = 0$ 
5   $t_0 = 0$ 
6  for  $i = 1$  to  $m$            // preprocessing
7     $p = (dp + P[i]) \bmod q$ 
8     $t_0 = (dt_0 + T[i]) \bmod q$ 
9  for  $s = 0$  to  $n - m$       // matching
10   if  $p == t_s$ 
11     if  $P[1..m] == T[s + 1..s + m]$ 
12       print "Pattern occurs with shift"  $s$ 
13   if  $s < n - m$ 
14      $t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$ 
```

# Rabin-Karp Algorithm

## Example

$$\text{hash('aab')} = 3$$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$$\text{hash('cab')} = 0$$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$$\text{hash('cab')} = 0$$

# Rabin-Karp Algorithm Example

$$\text{hash('bbc')} = 3$$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$$\text{hash('cab')} = 0$$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$$\text{hash('cab')} = 0$$

# Rabin-Karp Algorithm

## Example

$\text{hash('cba')} = 0$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$\text{hash('cab')} = 0$

$\text{hash('aba')} = 0$

Text = 'aabbcaba'

a	a	b	b	c	a	b	a
---	---	---	---	---	---	---	---

Pattern = 'cab'

c	a	b
---	---	---

$\text{hash('cab')} = 0$

Collision happened  
in hashing  
But the algorithm  
handles it

# Time Complexity

- Matching time in the worst case

$$O(m(n-m+1)) \sim O(n^2)$$

- Performs better in average case  
preprocessing time

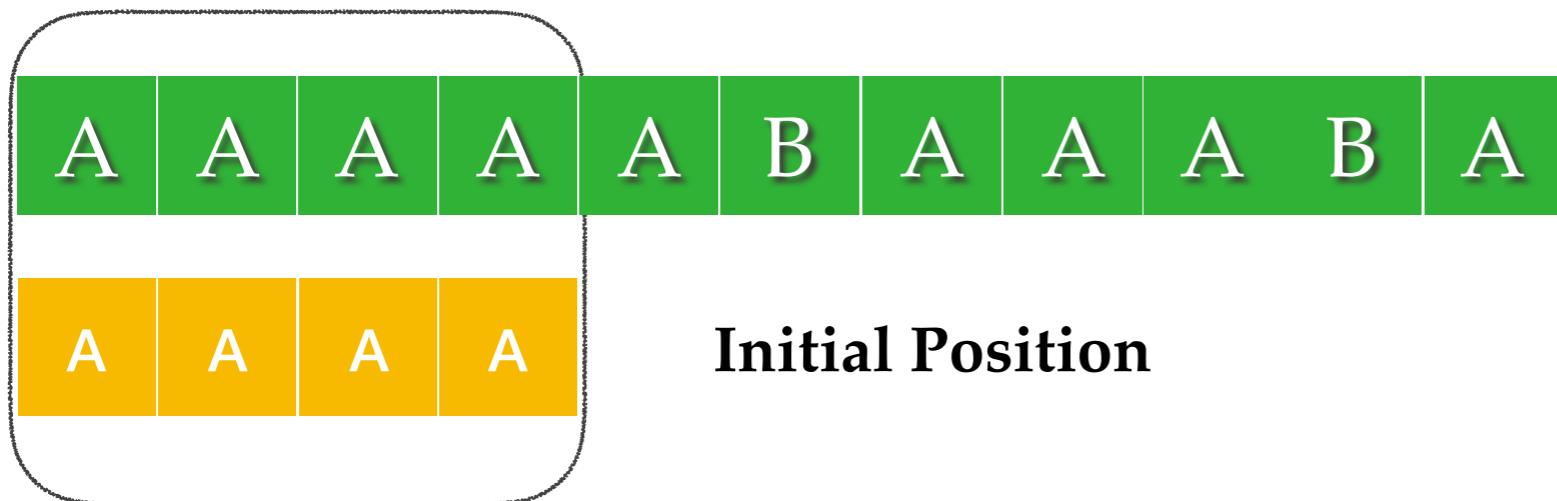
$$O(m)$$

# KMP String Matching Algorithm

- Knuth-Morris-Pratt Algorithm
- Improves the worst case time complexity to  $O(n)$
- Use degenerating property of the pattern

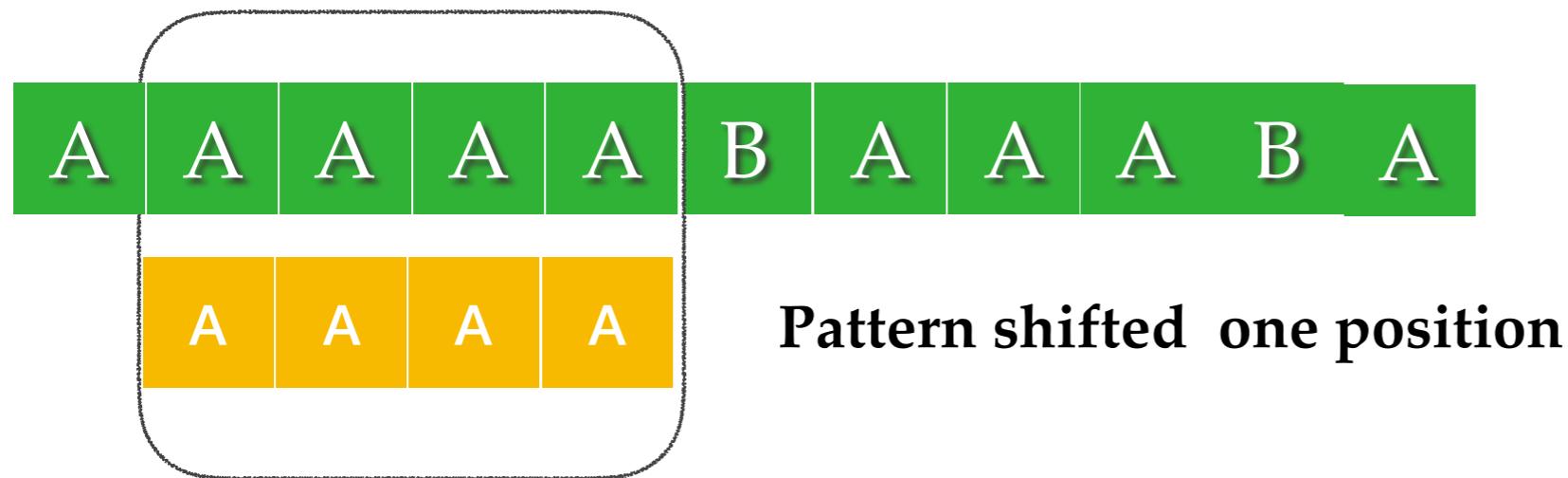
# KMP Algorithm

## Example

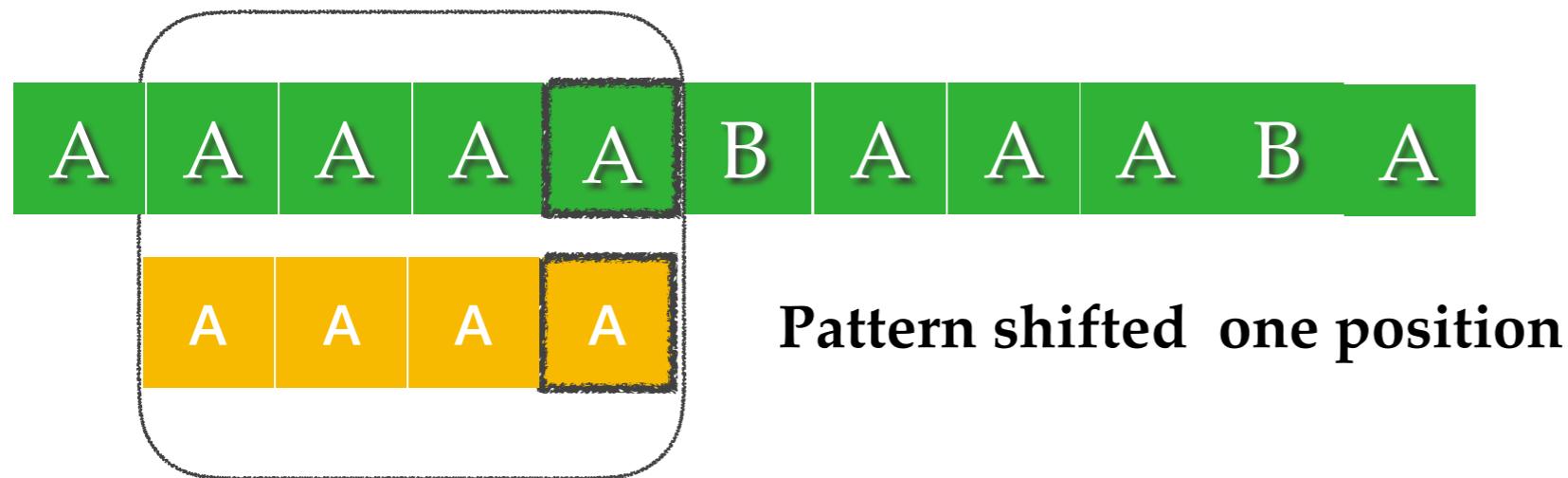


# KMP Algorithm

## Example



# KMP Algorithm Example



Need preprocessing of pattern

# KMP Algorithm

## Preprocessing

- $\text{text} = T[1..n]$
- $\text{pattern} = P[1..m]$
- $\text{LPS} = [1..m]$

# KMP Algorithm

## Preprocessing

- pattern[]

A	B	X	A	B
---	---	---	---	---

- LPS[]

--	--	--	--	--

0    1    2    3    4

LPS[i]

length of maximum matching  
prefix(suffix) of pattern[0..i]

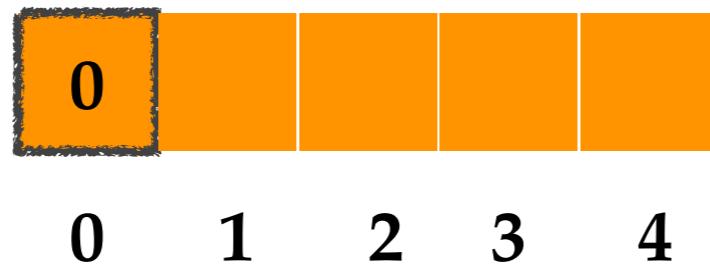
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$LPS[0] = 0$$

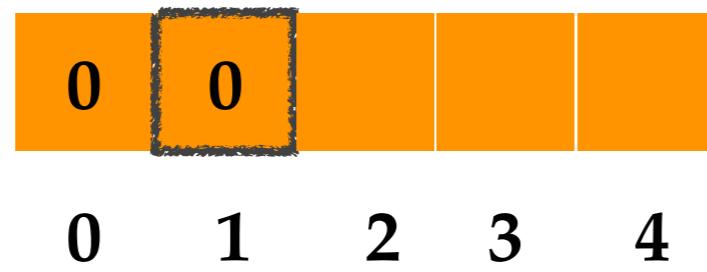
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$\text{LPS}[0] = 0$$

$$\text{LPS}[1] = 0$$

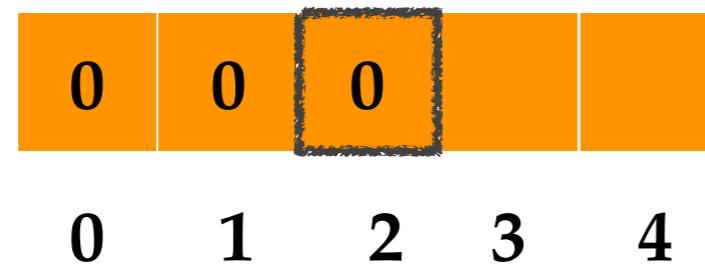
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$LPS[0] = 0$$

$$LPS[1] = 0$$

$$LPS[2] = 0$$

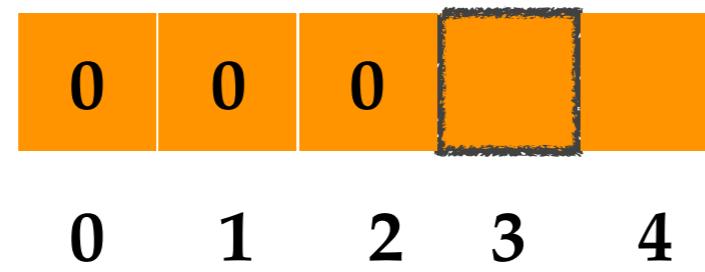
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$LPS[0] = 0$$

$$LPS[1] = 0$$

$$LPS[2] = 0$$

$$LPS[3] =$$

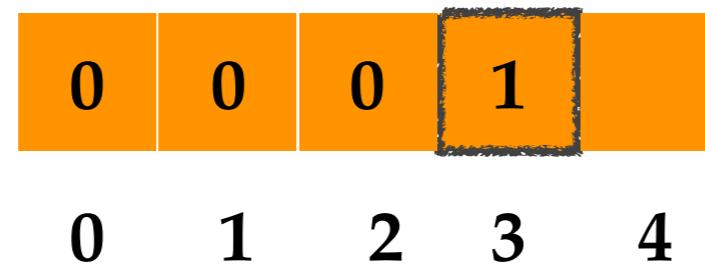
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$LPS[0] = 0$$

$$LPS[1] = 0$$

$$LPS[2] = 0$$

$$LPS[3] = 1$$

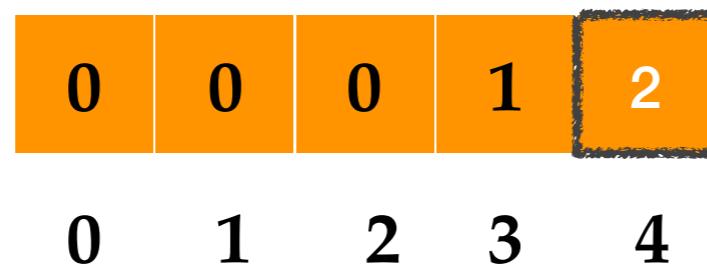
# KMP Algorithm

## Preprocessing

- pattern[]



- LPS[]



$$LPS[0] = 0$$

$$LPS[1] = 0$$

$$LPS[2] = 0$$

$$LPS[3] = 1$$

$$LPS[4] = 2$$

# KMP Algorithm

## Searching the Pattern

- To search pattern in the main text use the **LPS array**
- For each value of LPS we can decide which **next characters should be matched**
- The idea is not matching characters that we **already** know match anyway

# KMP Algorithm

## Searching the Pattern

- **Text[]**

A	B	X	A	B	A	B	X	A	B
---	---	---	---	---	---	---	---	---	---

- **pattern[]**

A	B	X	A	B
---	---	---	---	---

- **LPS[]**

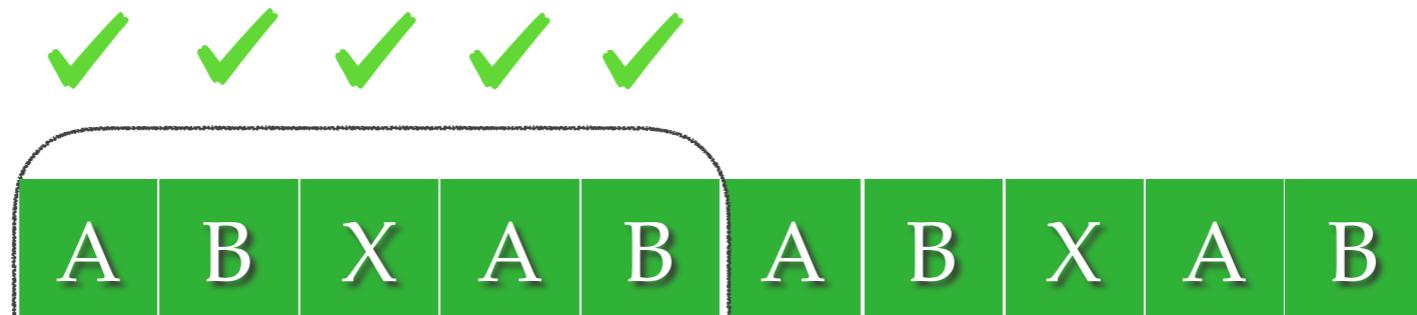
0	0	0	1	2
---	---	---	---	---

0    1    2    3    4

# KMP Algorithm

## Searching the Pattern

- **Text[]**



- **pattern[]**



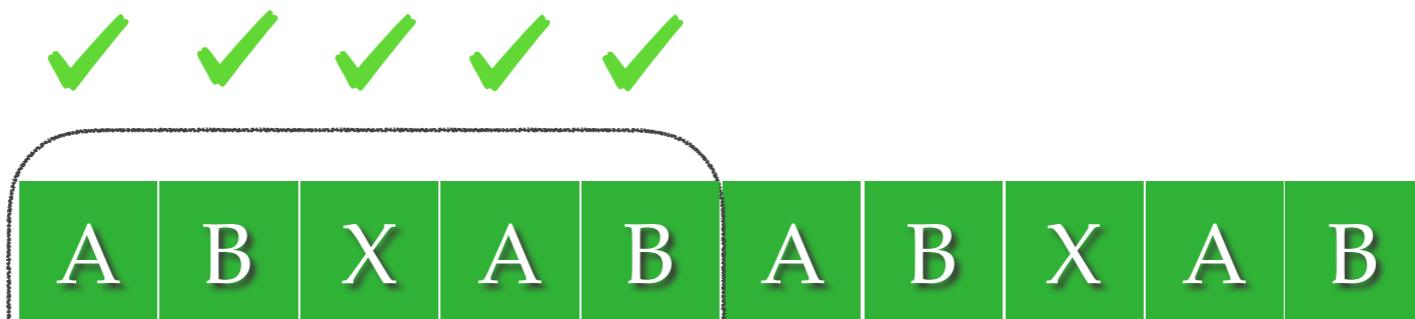
- **LPS[]**

0	0	0	1	2
0	1	2	3	4

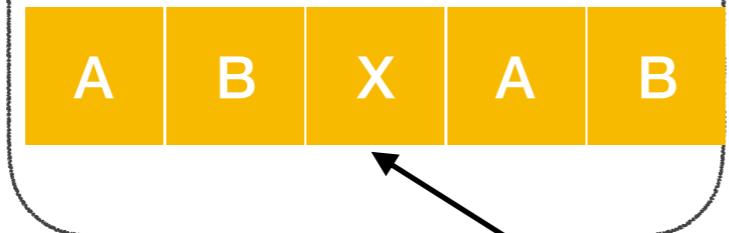
# KMP Algorithm

## Searching the Pattern

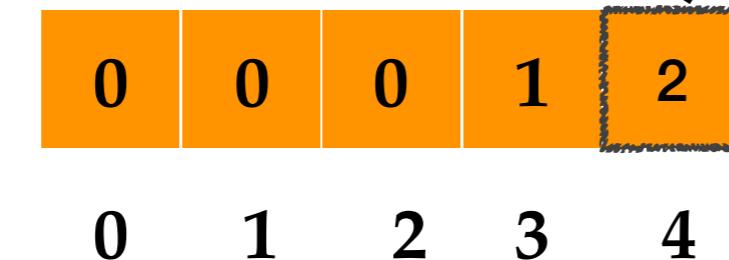
- **Text[]**



- **pattern[]**



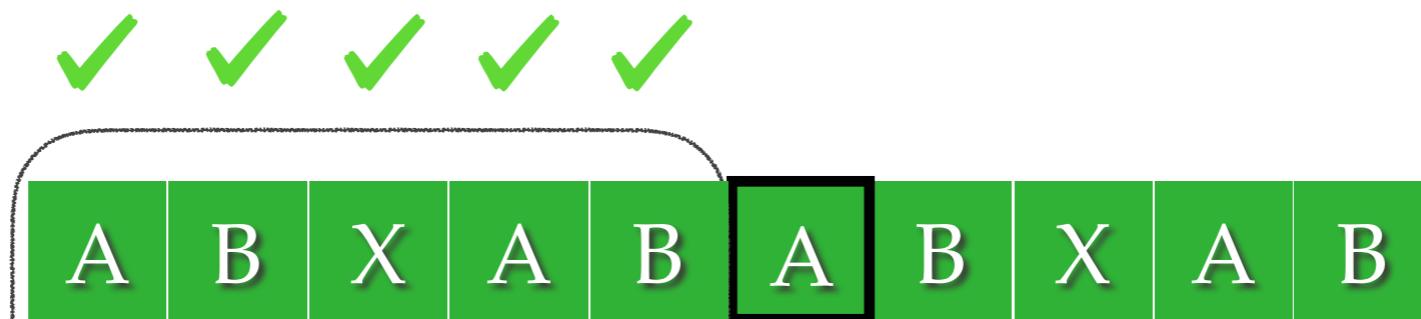
- **LPS[]**



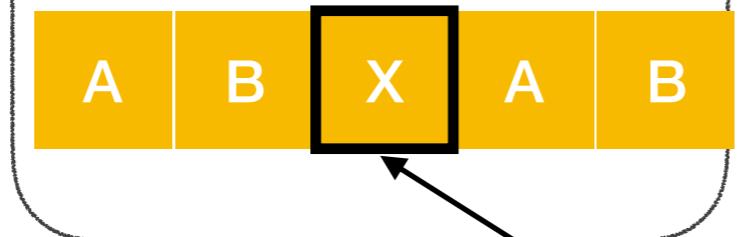
# KMP Algorithm

## Searching the Pattern

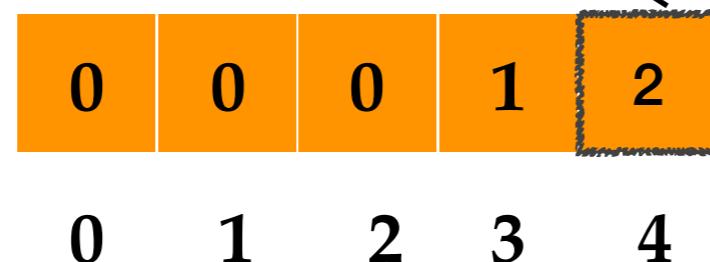
- **Text[]**



- **pattern[]**



- **LPS[]**



# KMP Algorithm

## Searching the Pattern

- **Text[]**



- **pattern[]**

- **LPS[]**

0	0	0	1	2
0	1	2	3	4

# KMP Algorithm

## Searching the Pattern

- **Text[]**



- **pattern[]**

- **LPS[]**

0	0	0	1	2
0	1	2	3	4

Current Character

# KMP Algorithm

## Searching the Pattern

- **Text[]**



- **pattern[]**

- **LPS[]**

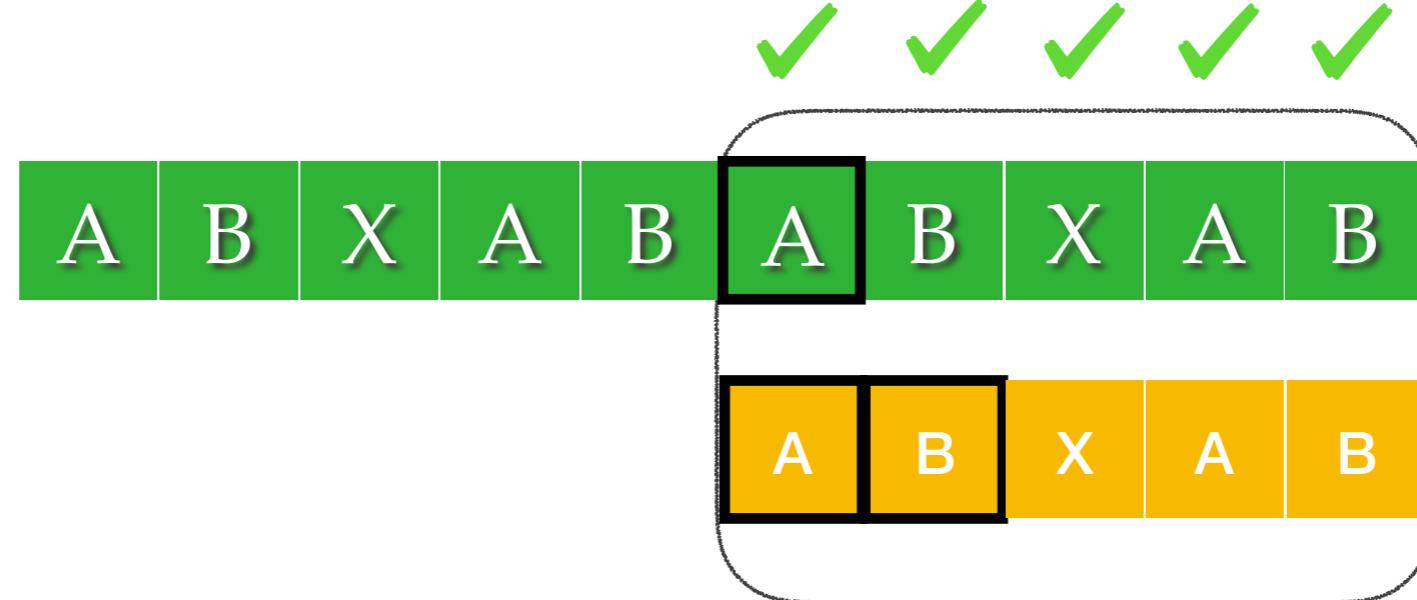
0	0	0	1	2
0	1	2	3	4

Substring behind the  
current character  
 $\text{pattern}[0..1] = \text{'AB'}$

# KMP Algorithm

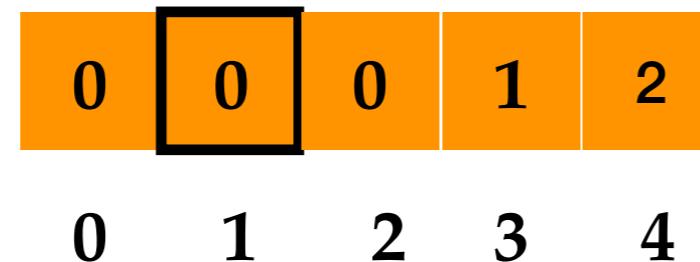
## Searching the Pattern

- **Text[]**



- **pattern[]**

- **LPS[]**



# String Matching Algorithms

## 1. Naïve String Matching

The naïve approach simply test all the possible placement of Pattern  $P[1 \dots m]$  relative to text  $T[1 \dots n]$ . Specifically, we try shift  $s = 0, 1, \dots, n - m$ , successively and for each shift,  $s$ . Compare  $T[s + 1 \dots s + m]$  to  $P[1 \dots m]$ .

```
NAÏVE_STRING_MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n - m do
4.   if P[1 .. m] = T[s + 1 .. s + m]
5.     then return valid shift s
```

The naïve string-matching procedure can be interpreted graphically as a sliding a pattern  $P[1 \dots m]$  over the text  $T[1 \dots n]$  and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

In order to analysis the time of naïve matching, we would like to implement above algorithm to understand the test involves in line 4.

Note that in this implementation, we use notation  $P[1 \dots j]$  to denote the substring of  $P$  from index  $i$  to index  $j$ . That is,  $P[1 \dots j] = P[i] P[i + 1] \dots P[j]$ .

```
NAÏVE_STRING_MATCHER (T, P)
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n-m do
4.   j ← 1
5.   while j ≤ m and T[s + j] = P[j] do
6.     j ← j +1
7.   If j > m then
8.     return valid shift s
9. return no valid shift exist // i.e., there is no substring of T matching P.
```

## Analysis

Referring to implementation of naïve matcher, we see that the for-loop in line 3 is executed at most  $n - m + 1$  times, and the while-loop in line 5 is executed at most  $m$  times. Therefore, the running time of the algorithm is  $O((n - m + 1)m)$ , which is clearly  $O(nm)$ . Hence, in the worst case, when the length of the pattern,  $m$  are roughly equal, this algorithm runs in the *quadratic* time.

One worst case is that text, T, has n number of A's and the pattern, P, has (m -1) number of A's followed by a single B.

## 2. Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naïve algorithm. Knuth-Morris-Pratt algorithm keeps the information that naïve approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of  $O(n + m)$ , which is optimal in the worst case sense. That is, in the worst case Knuth-Morris-Pratt algorithm we have to examine all the characters in the text and pattern at least once.

### The Failure Function

The KMP algorithm preprocesses the pattern P by computing a failure function  $f$  that indicates the largest possible shift s using previously performed comparisons. Specifically, the failure function  $f(j)$  is defined as the length of the longest prefix of P that is a suffix of  $P[i \dots j]$ .

```

Input: Pattern with m characters
Output: Failure function f for P[i .. j]
KNUTH-MORRIS-PRATT FAILURE (P)
1. i ← 1
2. j ← 0
3. f(0) ← 0
4. while i < m do
5.   if P[j] = P[i]
6.     f(i) ← j + 1
7.     i ← i + 1
8.     j ← j + 1
9.   else if j > 0
10.    j ← f(j - 1)
11.   else
12.     f(i) ← 0
13.   i ← i + 1

```

Note that the failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of  $P[1 \dots j]$ , encodes repeated substrings inside the pattern itself.

As an example, consider the pattern P = a b a c a b. The failure function,  $f(j)$ , using above algorithm is

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

By observing the above mapping we can see that the longest prefix of pattern, P, is "a b" which is also a suffix of pattern P.

Consider an attempt to match at position i, that is when the pattern  $P[0 \dots m -1]$  is aligned with text  $P[i \dots i + m -1]$ .

T: a b a c a a b a c c

P: a b a c a a b

Assume that the first mismatch occurs between characters  $T[i + j]$  and  $P[j]$  for  $0 < j < m$ . In the above example, the first mismatch is  $T[5] = a$  and  $P[5] = b$ .

Then,  $T[i \dots i + j - 1] = P[0 \dots j - 1] = u$

That is,  $T[0 \dots 4] = P[0 \dots 4] = u$ , in the example [ $u = a b a c a$ ] and

$T[i + j] \neq P[j]$  i.e.,  $T[5] \neq P[5]$ , In the example [ $T[5] = a \neq b = P[5]$ ].

When shifting, it is reasonable to expect that a prefix  $v$  of the pattern matches some suffix of the portion  $u$  of the text. In our example,  $u = a b a c a$  and  $v = a b a c a$ , therefore, 'a' a prefix of  $v$  matches with 'a' a suffix of  $u$ . Let  $l(j)$  be the length of the longest string  $P[0 \dots j - 1]$  of pattern that matches with text followed by a character  $c$  different from  $P[j]$ . Then after a shift, the comparisons can resume between characters  $T[i + j]$  and  $P[l(j)]$ , i.e.,  $T(5)$  and  $P(1)$ .

T : a b a c a a b a c c
P : a b a c a b

Note that no comparison between  $T[4]$  and  $P[1]$  needed here.

```

Input: Strings T[0 .. n] and P[0 .. m]
Output: Starting index of substring of T matching P
KNUTH-MORRIS-PRATT (T, P)
1. f ← compute failure function of Pattern P
2. i ← 0
3. j ← 0
4. while i < length[T] do
5.   if j ← m-1 then
6.     return i - m + 1 // we have a match
7.   i ← i + 1
8.   j ← j + 1
9.   else if j > 0
10.    j ← f(j - 1)
11.   else
12.    i ← i + 1

```

### Analysis

The running time of Knuth-Morris-Pratt algorithm is proportional to the time needed to read the characters in text and pattern. In other words, the worst-case running time of the algorithm is  $O(m + n)$  and it requires  $O(m)$  extra space. It is important to note that these quantities are independent of the size of the underlying alphabet.

### 3. Boyer-Moore Algorithm

The Boyer-Moore algorithm is consider the most efficient string-matching algorithm in usual applications, for example, in text editors and commands substitutions. The reason is that it works the fastest when the alphabet is moderately sized and the pattern is relatively long.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost

character. During the testing of a possible placement of pattern P against text T, a mismatch of text character  $T[i] = c$  with the corresponding pattern character  $P[j]$  is handled as follows: If c is not contained anywhere in P, then shift the pattern P completely past  $T[i]$ . Otherwise, shift P until an occurrence of character c in P gets aligned with  $T[i]$ .

This technique likely to avoid lots of needless comparisons by significantly shifting pattern relative to text.

### Last Function

We define a function  $\text{last}(c)$  that takes a character c from the alphabet and specifies how far may shift the pattern P if a character equal to c is found in the text that does not match the pattern.

$$\text{last}(c) = \begin{cases} \text{Index of the last occurrence of } c \text{ in pattern P} & ; \text{ if } c \text{ is in P} \\ -1 & ; \text{ otherwise} \end{cases}$$

For example consider :

0 1 2 3 4 5 6 7 8 9  
T: a b a c a a b a c c

0 1 2 3 4 5  
P: a b a c a b

$\text{last}(a)$  is the index of the last (rightmost) occurrence of 'a' in P, which is 4.

$\text{last}(c)$  is the index of the last occurrence of c in P, which is 3

'd' does not exist in the pattern there we have  $\text{last}(d) = -1$ .

c	a	b	c	d
$\text{last}(c)$	4		3	-1

Now, for 'b' notice

T: a b a c a a b a c c  
P: a b a c a b

Therefore,  $\text{last}(b)$  is the index of last occurrence of b in P, which is 5.

The complete  $\text{last}(c)$  function

c	a	b	c	d
---	---	---	---	---

last(c)	4	5	3	-1
---------	---	---	---	----

### Boyer-Moore algorithm

Input: Text with  $n$  characters and Pattern with  $m$  characters

Output: Index of the first substring of  $T$  matching  $P$

```
BOYER_MOORE_MATCHER ( $T, P$ )
1. Compute function last
2.  $i \leftarrow m - 1$ 
3.  $j \leftarrow m - 1$ 
4. Repeat
5.   If  $P[j] = T[i]$  then
6.     if  $j=0$  then
7.       return  $i$  // we have a match
10.    else
11.       $i \leftarrow i - 1$ 
12.       $j \leftarrow j - 1$ 
13.    else
14.       $i \leftarrow i + m - \text{Min}(j, 1 + \text{last}[T[i]])$ 
15.       $j \leftarrow m - 1$ 
16. until  $i > n - 1$ 
17. Return "no match"
```

### Analysis

The computation of the last function takes  $O(m+|\Sigma|)$  time and actual search takes  $O(mn)$  time.

Therefore the worst case running time of Boyer-Moore algorithm is  $O(nm + |\Sigma|)$ . Implies that the worst-case running time is quadratic, in case of  $n = m$ , the same as the naïve algorithm.

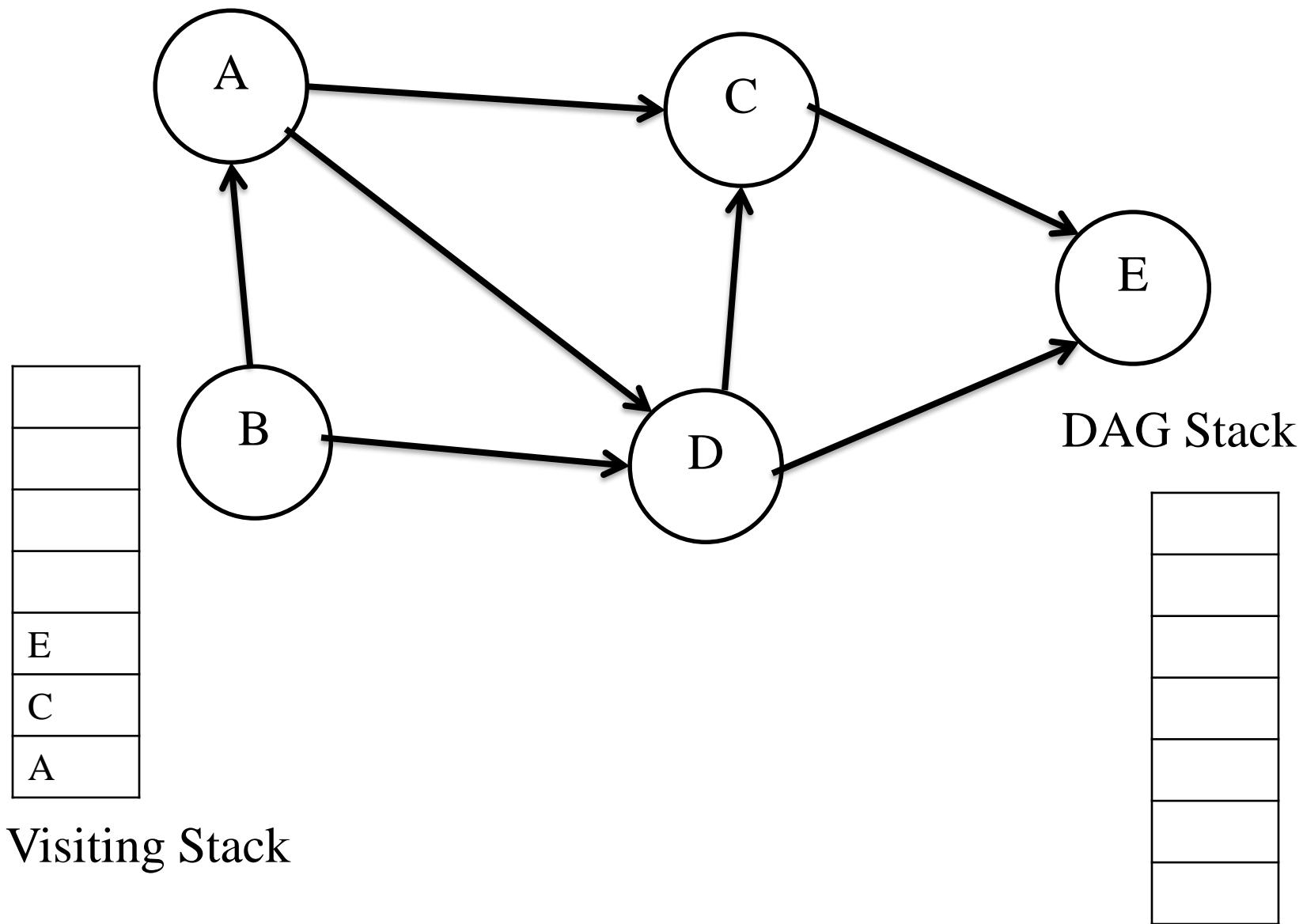
- (i) Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern).
- (ii) The payoff is not as for binary strings or for very short patterns.
- (iii) For binary strings Knuth-Morris-Pratt algorithm is recommended.
- (iv) For the very shortest patterns, the naïve algorithm may be better.

# Topological Sort

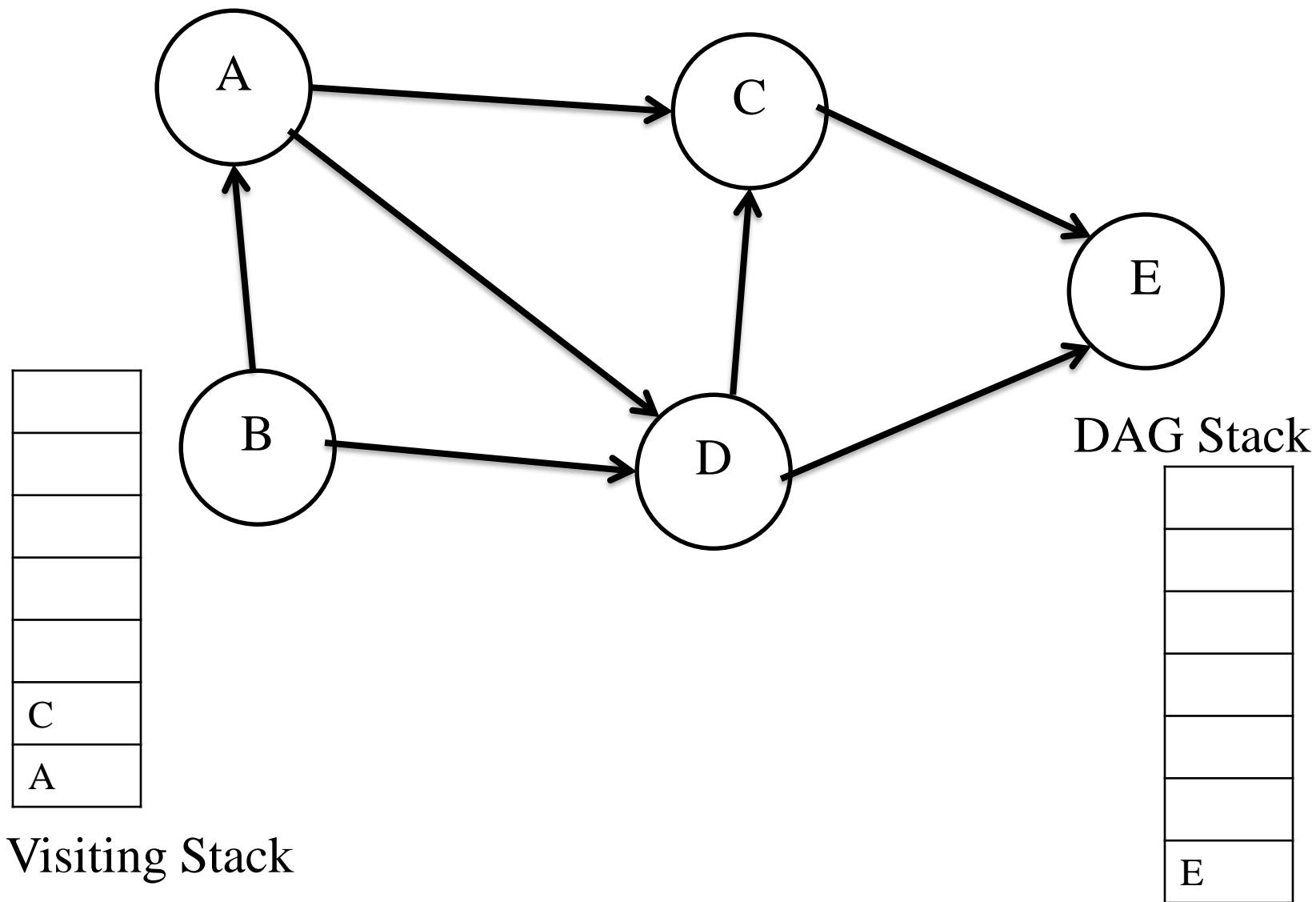
# Topological Sort

- **Sorting technique over DAGs (Directed Acyclic Graphs)**
- **It creates a linear sequence (ordering) for the nodes such that:**
  - If  $u$  has an outgoing edge to  $v \rightarrow$  then  $u$  must finish before  $v$  starts
- **Very common in ordering jobs or tasks**

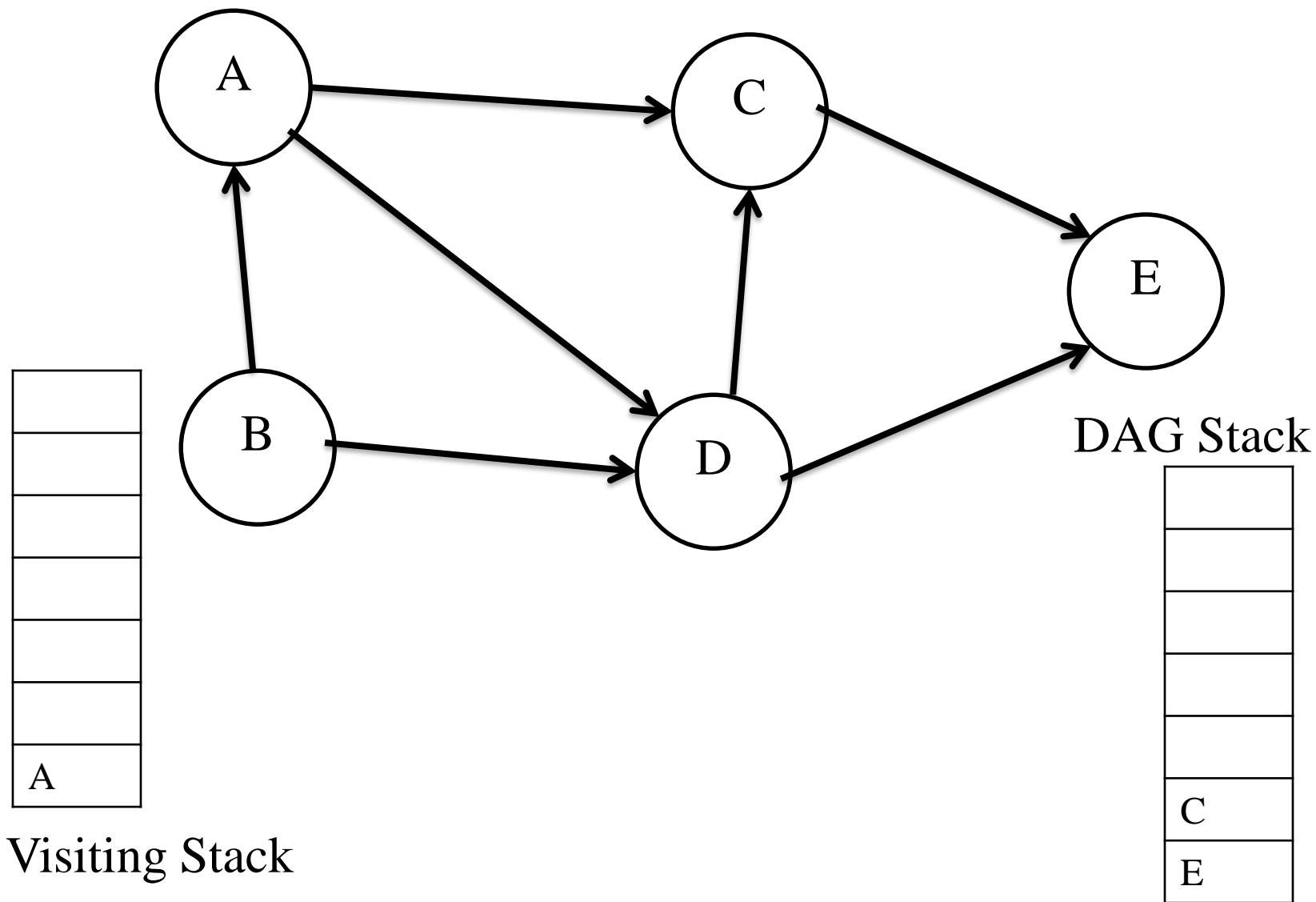
# Topological Sort- DAG



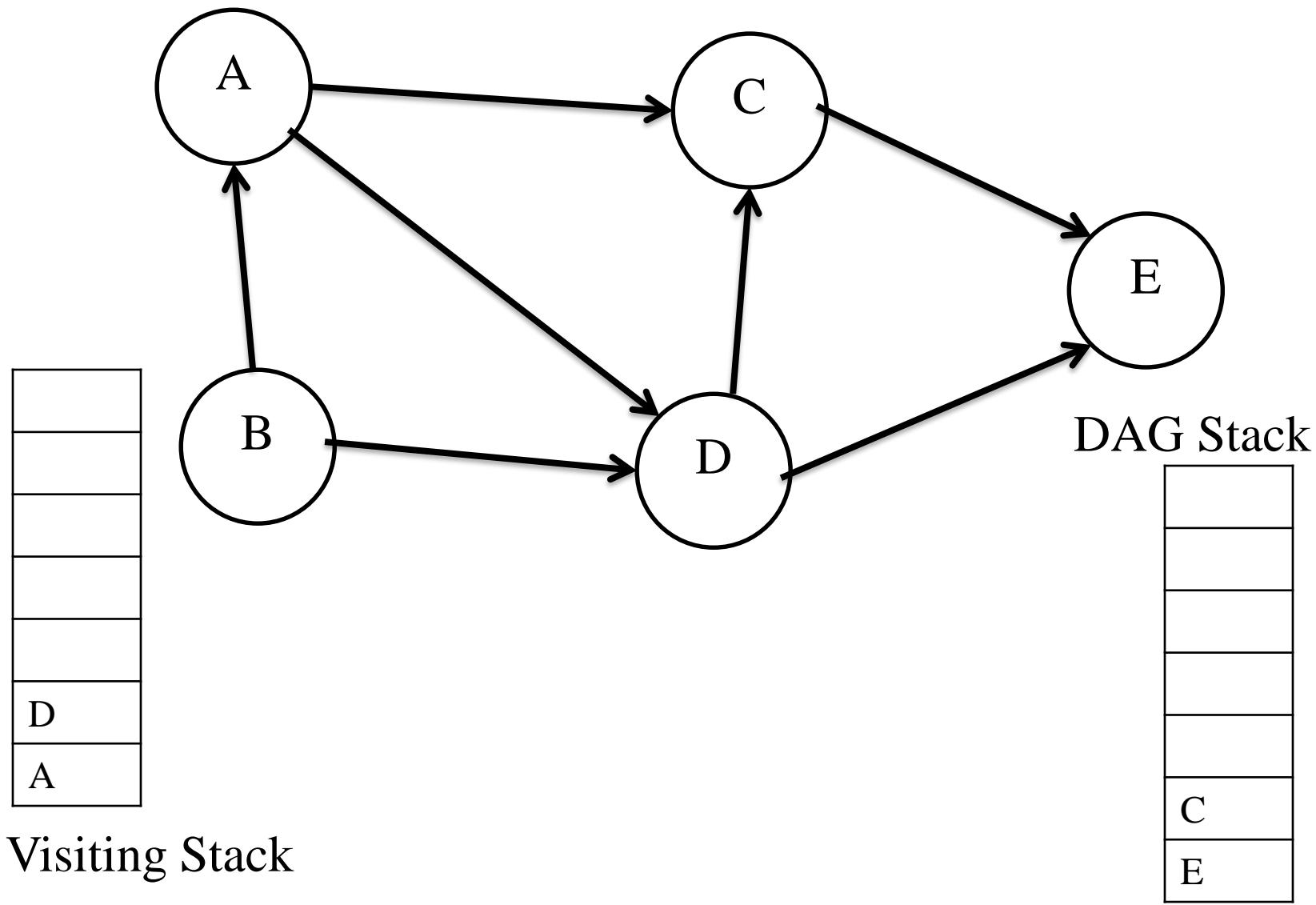
# Topological Sort- DAG



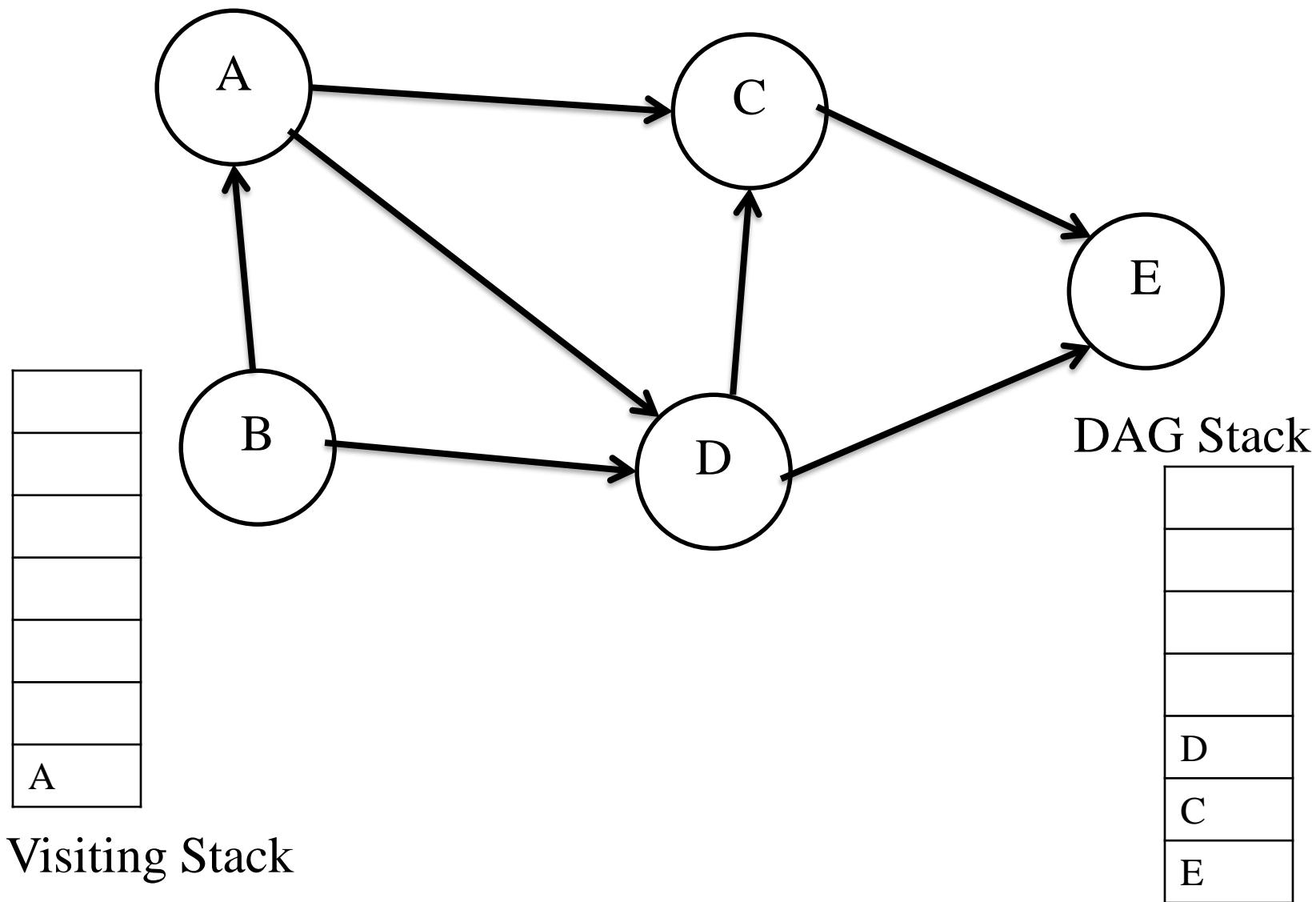
# Topological Sort- DAG



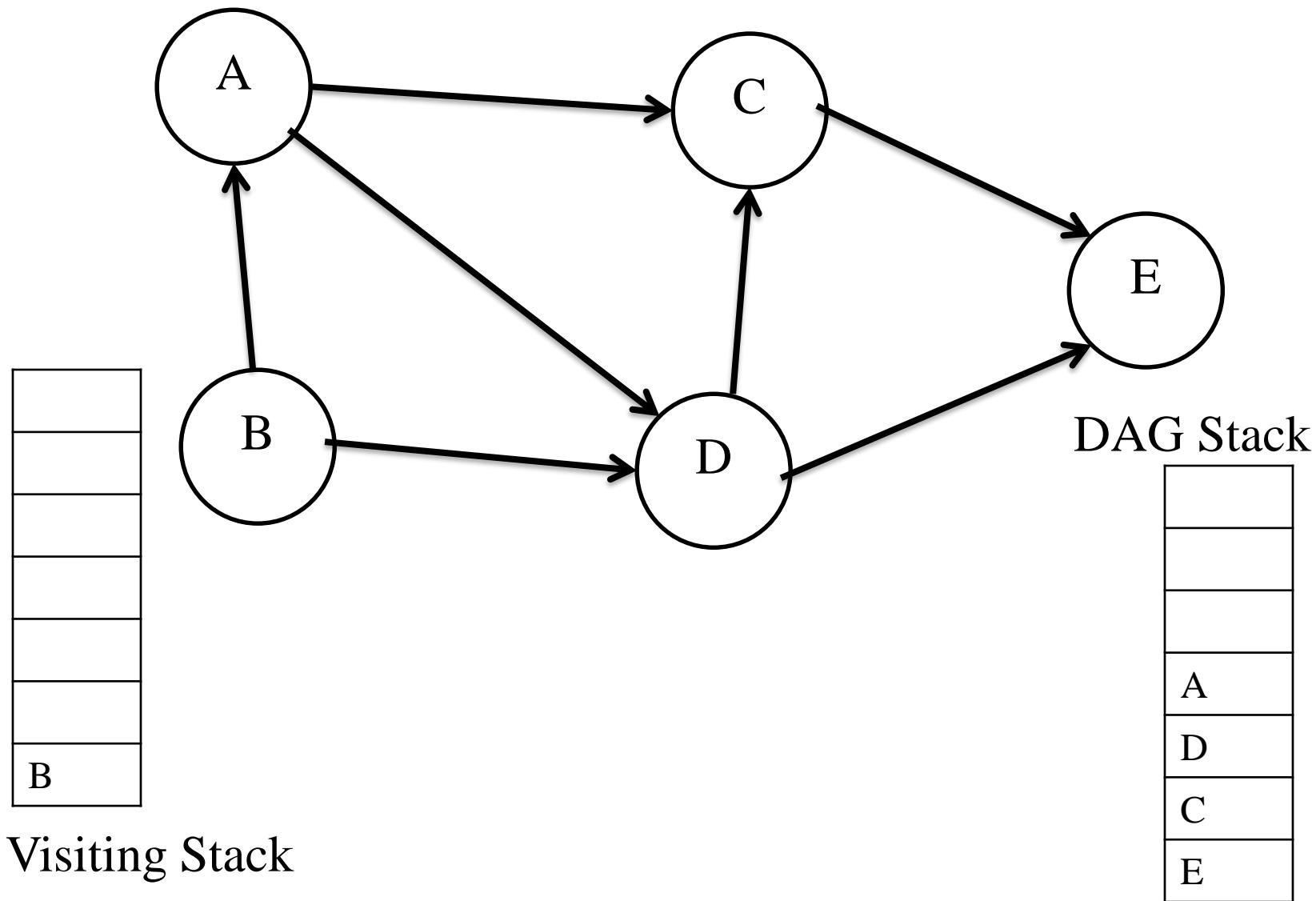
# Topological Sort- DAG



# Topological Sort- DAG

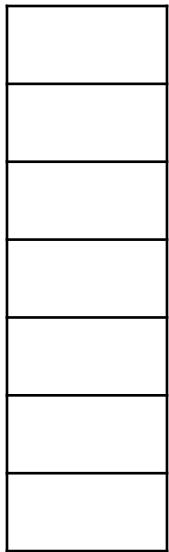
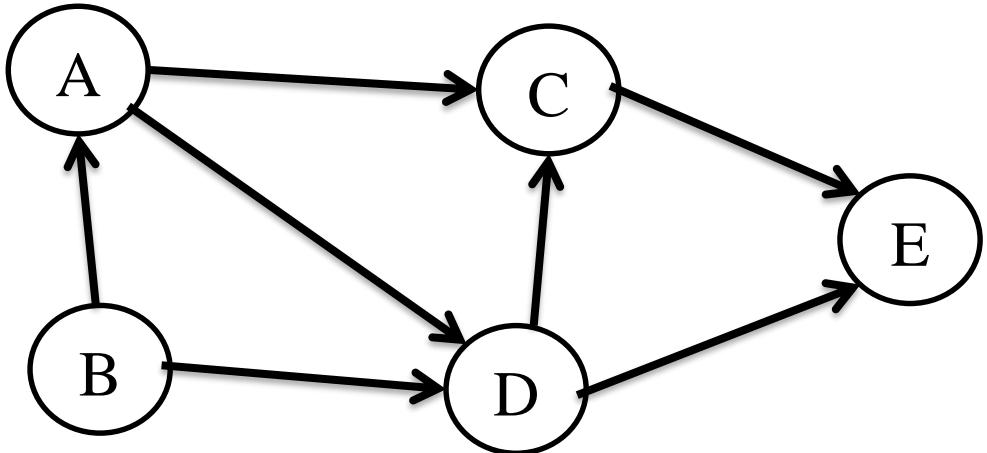


# Topological Sort- DAG



# Topological Sort- DAG

0	0	1	1	0
1	0	0	1	0
0	0	0	0	1
0	0	1	0	1
0	0	0	0	0



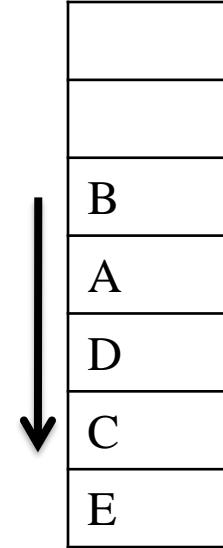
Visiting Stack

B → A → D → C → E

OR

V<sub>2</sub> → V<sub>1</sub> → V<sub>4</sub> → V<sub>3</sub> → V<sub>5</sub>

DAG Stack



# Topological Sort Algorithm using DFS

- To create a topological sort from a DAG

**1- Final linked list is empty**

**2- Run DFS**

**3- When a node becomes black (finishes) insert it to the top of a linked list**

Pseudocode:

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

    N = graph.numberOfNodes()
    V = [false,...,false] # Length N
    ordering = [0,...,0] # Length N
    i = N - 1 # Index for ordering array

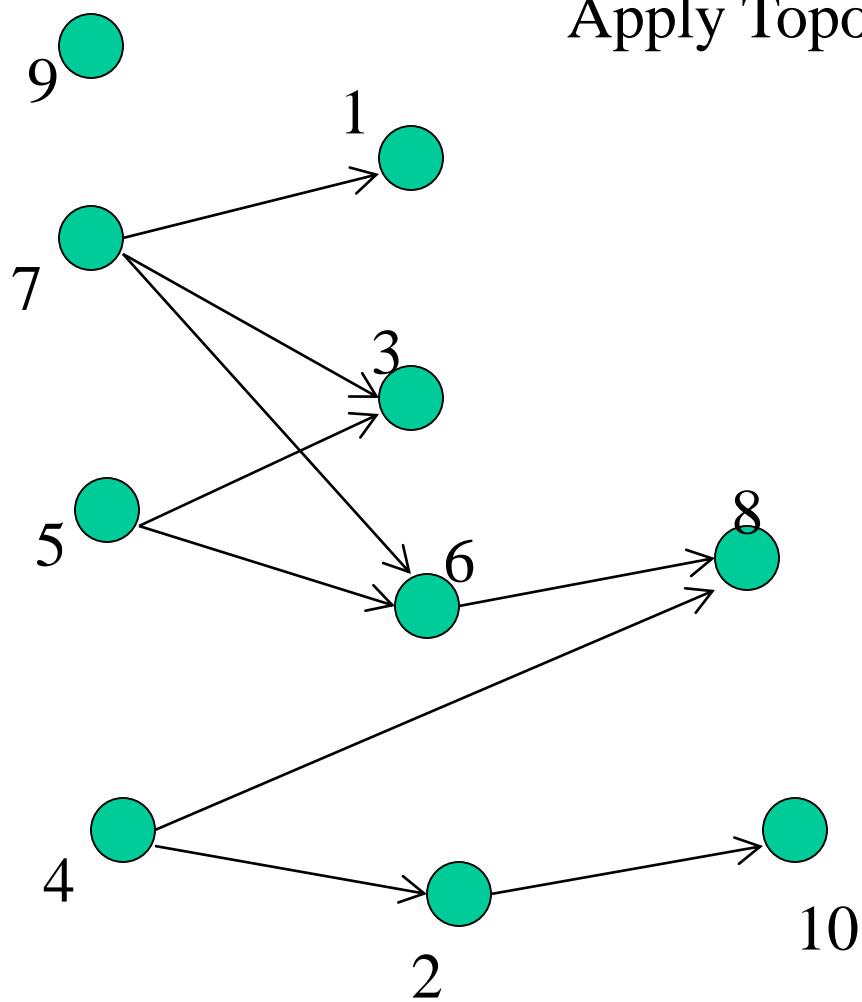
    for(at = 0; at < N; at++):
        if V[at] == false:
            visitedNodes = []
            dfs(at, V, visitedNodes, graph)
            for nodeId in visitedNodes:
                ordering[i] = nodeId
                i = i - 1
    return ordering
```

# Topological Sort Example

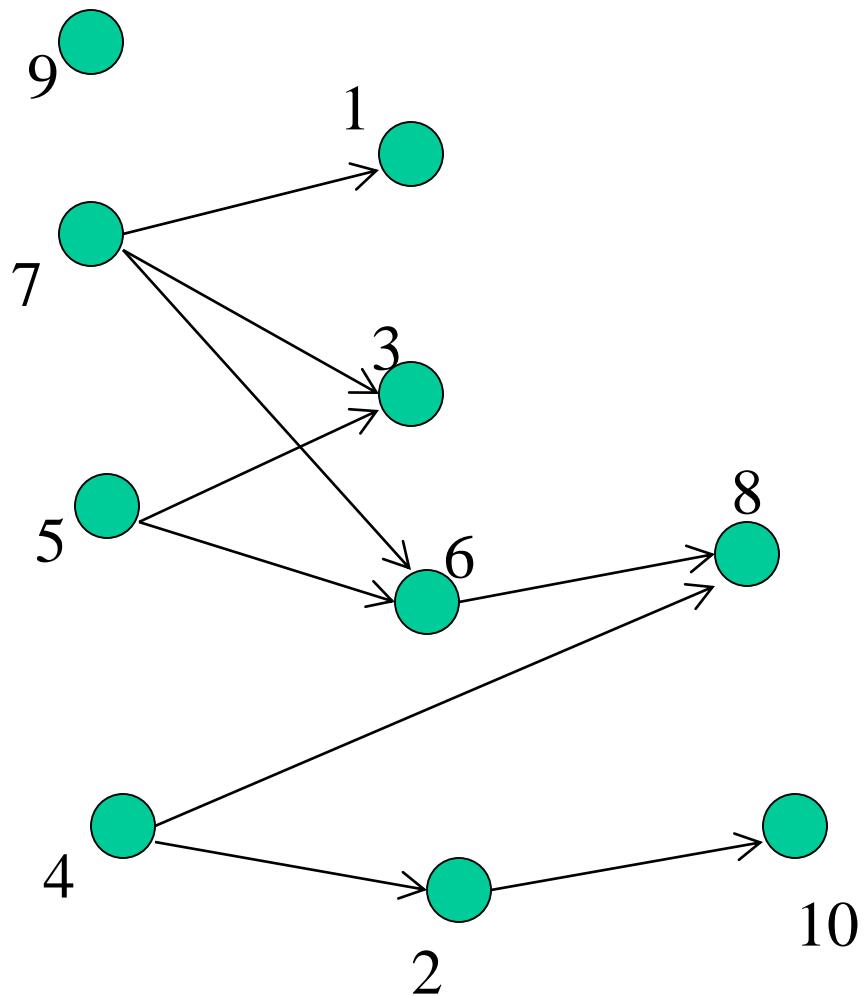
- A job consists of 10 tasks with the following precedence rules:
  - Must start with 7, 5, 4 or 9.
  - Task 1 must follow 7.
  - Tasks 3 & 6 must follow both 7 & 5.
  - 8 must follow 6 & 4.
  - 2 must follow 4.
  - 10 must follow 2.

**Make a directed graph and then do DFS.**

Apply Topological sort for the given DAG

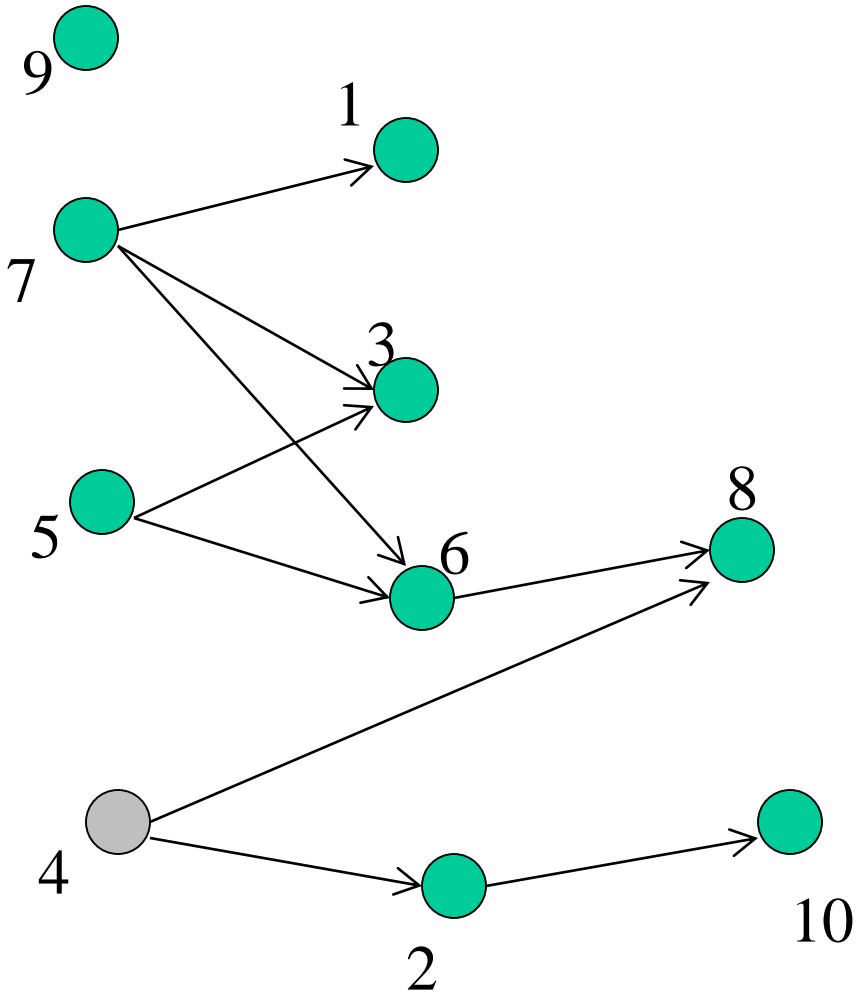


# Example

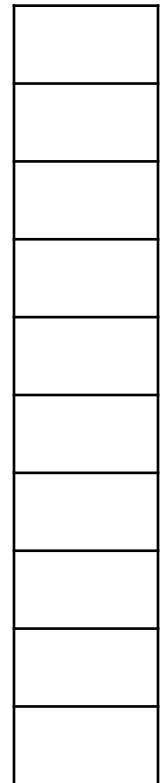


Select any vertex from the graph.

Here, Select vertex 4 and it marks as visited.



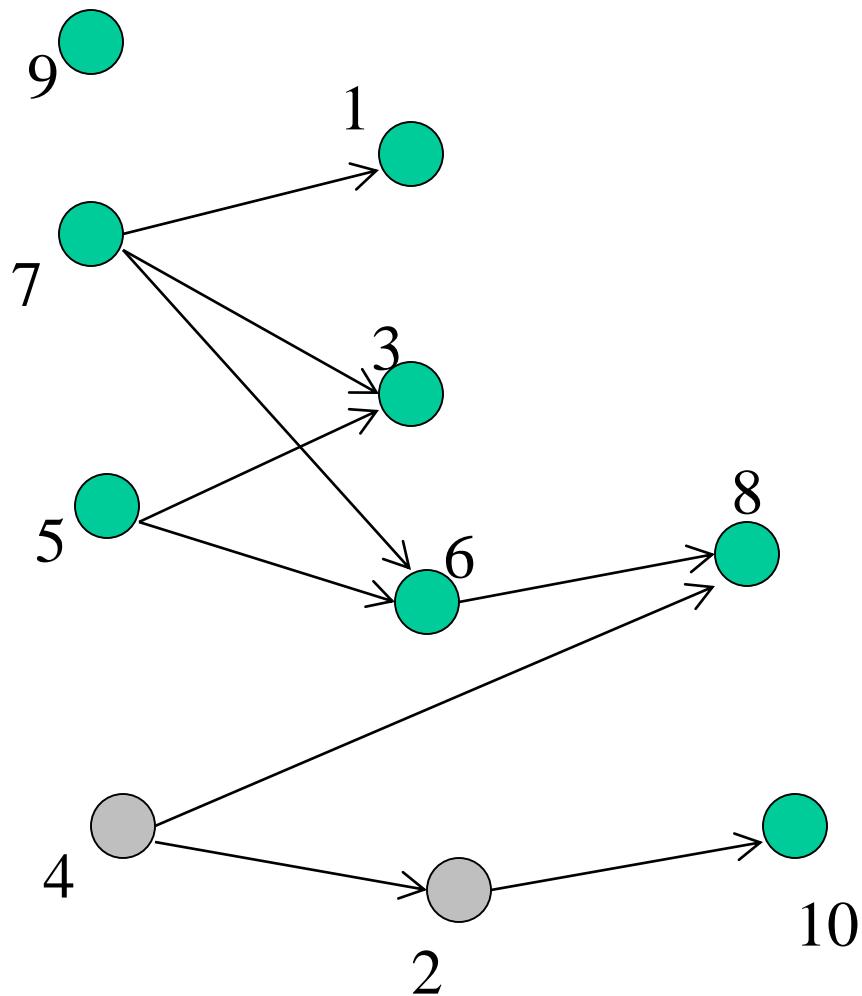
## Visited



T. Stack

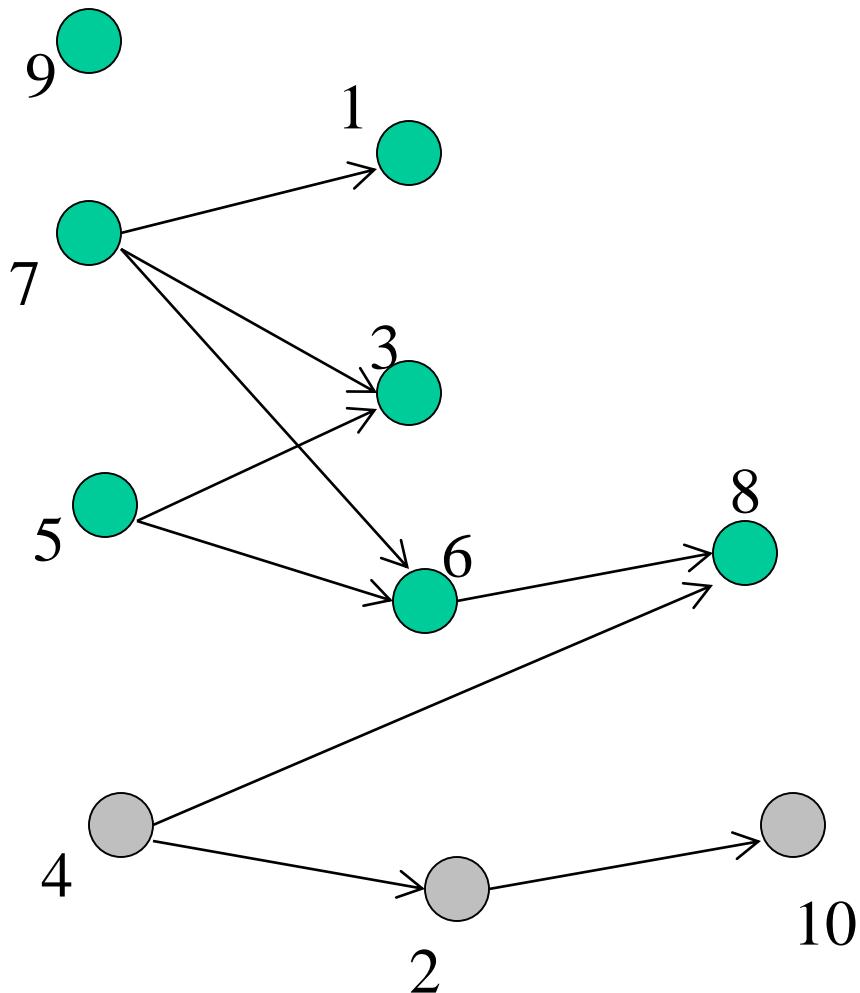
Find adjacent vertices of 4. There are 2 &8.

Select least value vertex 2 and mark as visited.

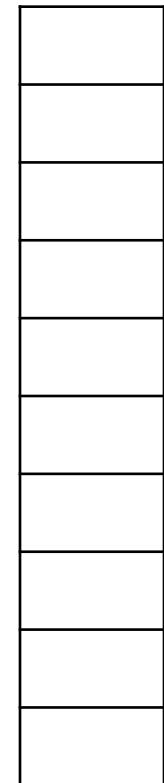


Find adjacent vertices of 2. There is 10.

Select vertex 10 and mark as visited.



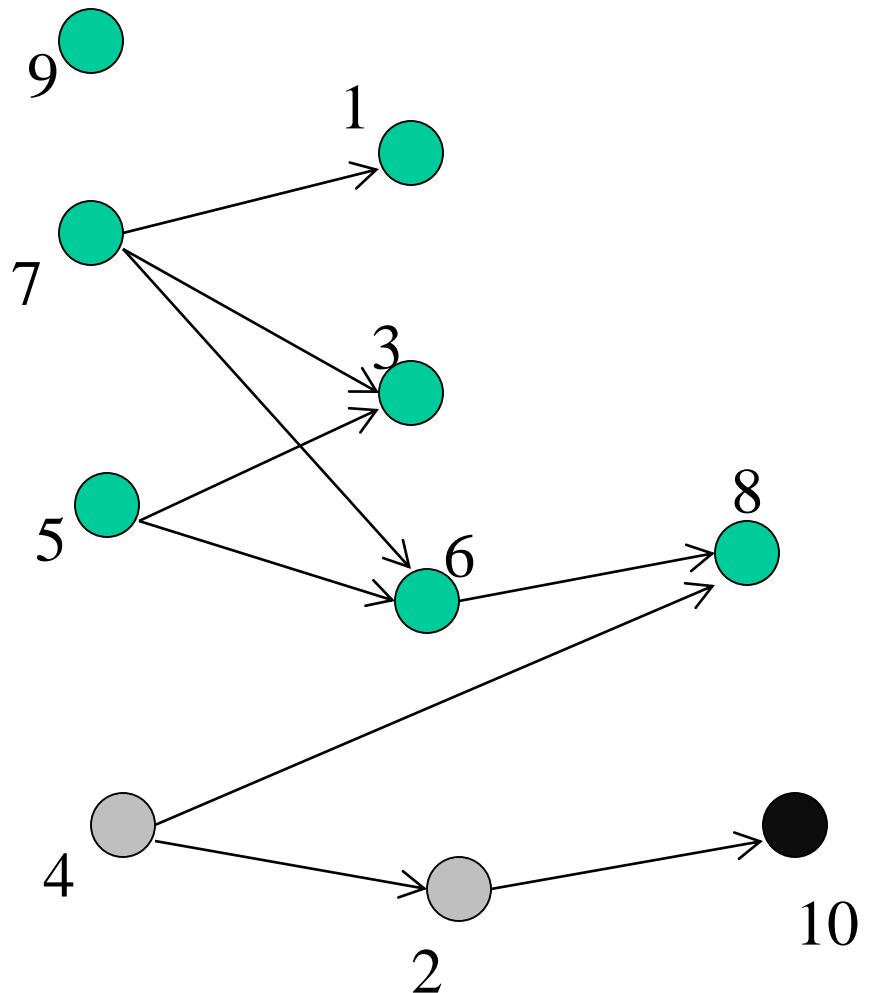
## Visited



## Stack

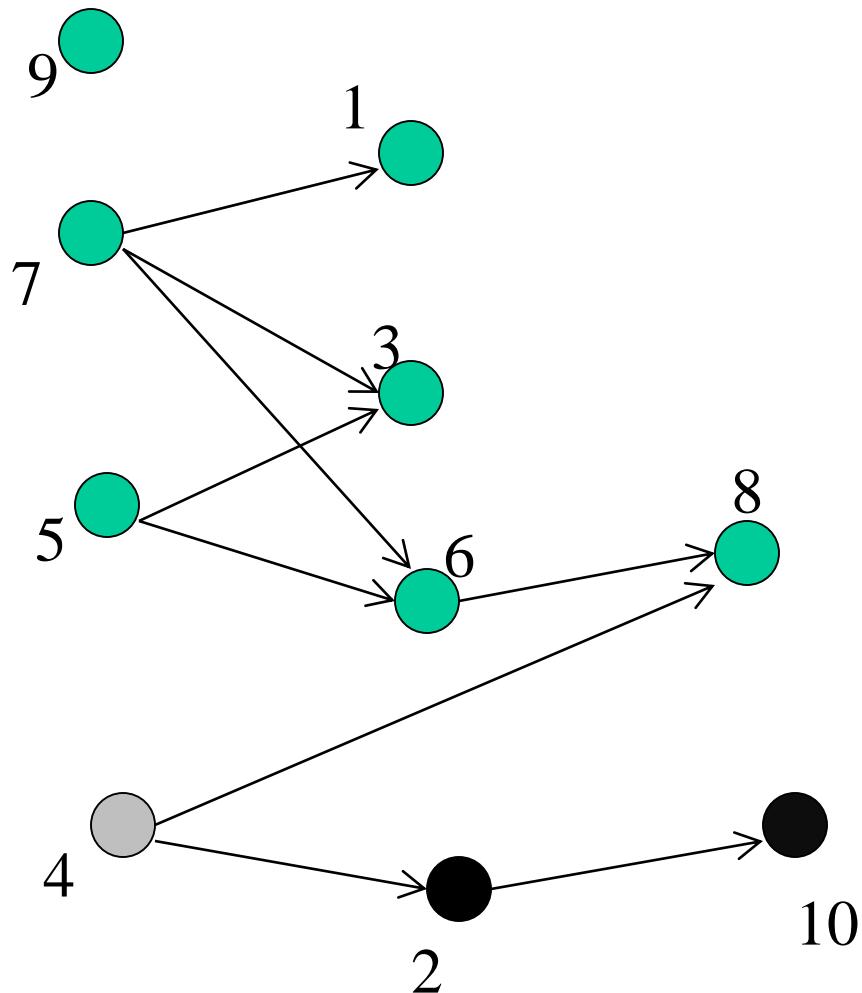
Find adjacent vertices of 10. There is no adjacent vertices for 10.

Hence insert vertex 10 into stack for scheduling. Then go back to parent vertex 2.

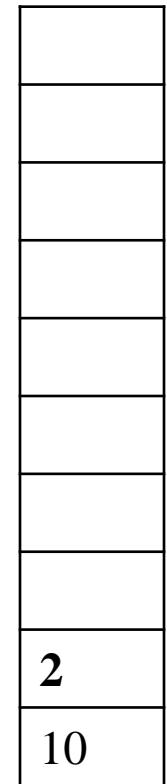


All adjacent vertices of 2 are visited. There is no more adjacent vertices for 2.

Hence insert vertex 2 into stack for scheduling. Then go back to parent vertex 4.



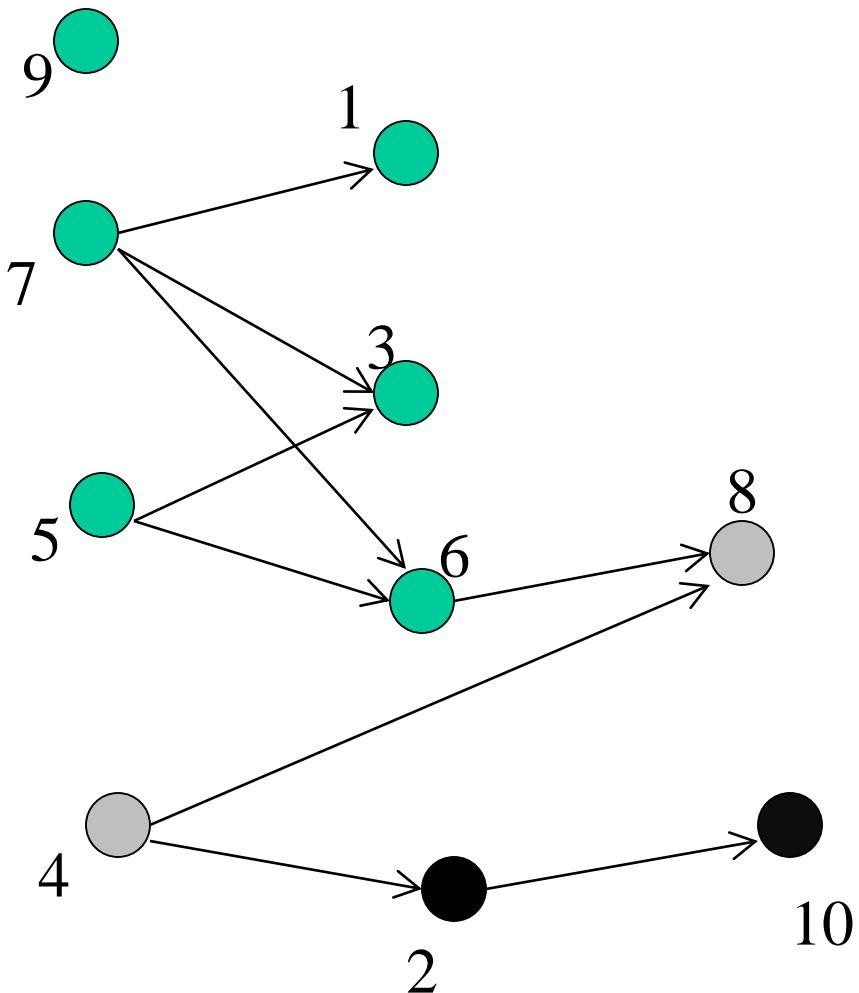
Visited



Stack

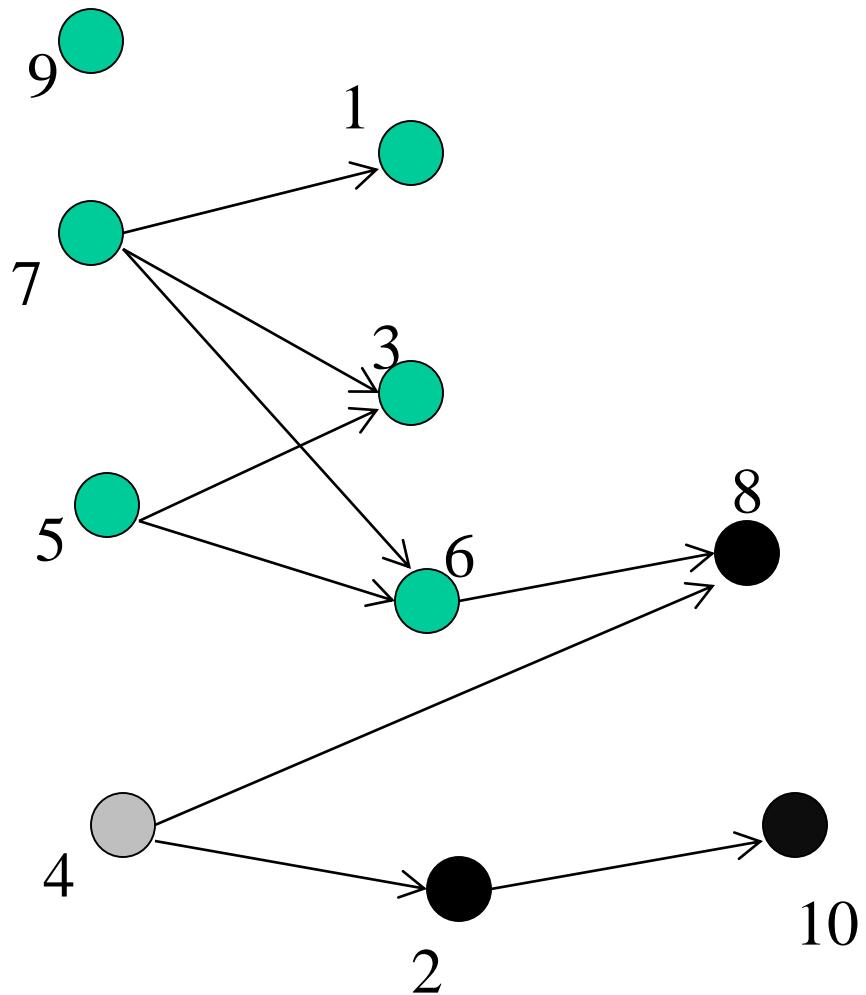
Find unvisited adjacent vertices of 4. There is 8.

Mark as visited vertex.



Find adjacent vertices of 8. There is no adjacent vertices for 8.

Hence insert vertex 8 into stack for scheduling. Then go back to parent vertex 4.



8
10
2
4

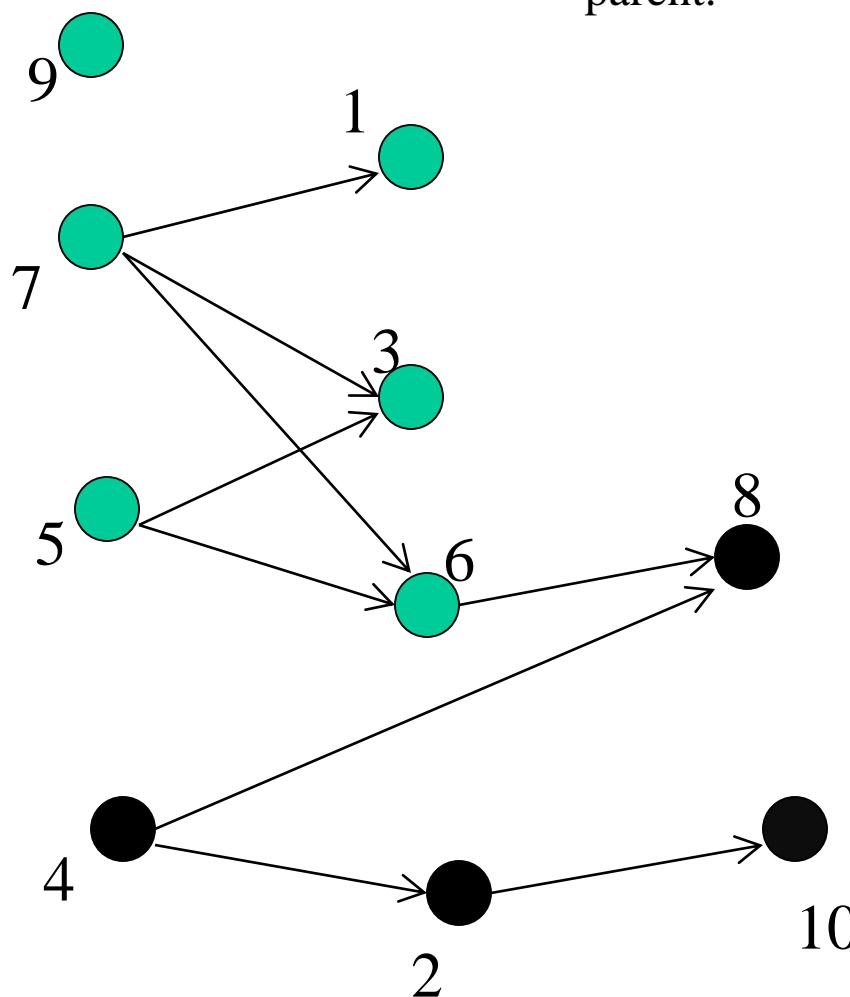
Visited

8
2
2
10

Stack

All adjacent vertices of 4 are visited. There is no more adjacent vertices for 4.

Hence insert vertex 4 into stack for scheduling. Then go back to parent vertex 4. but no parent.



8
10
2
4

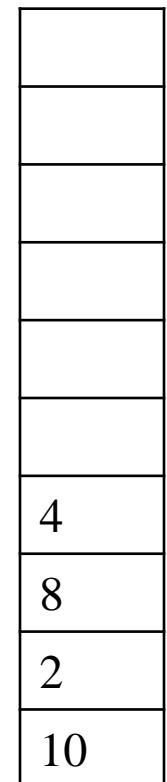
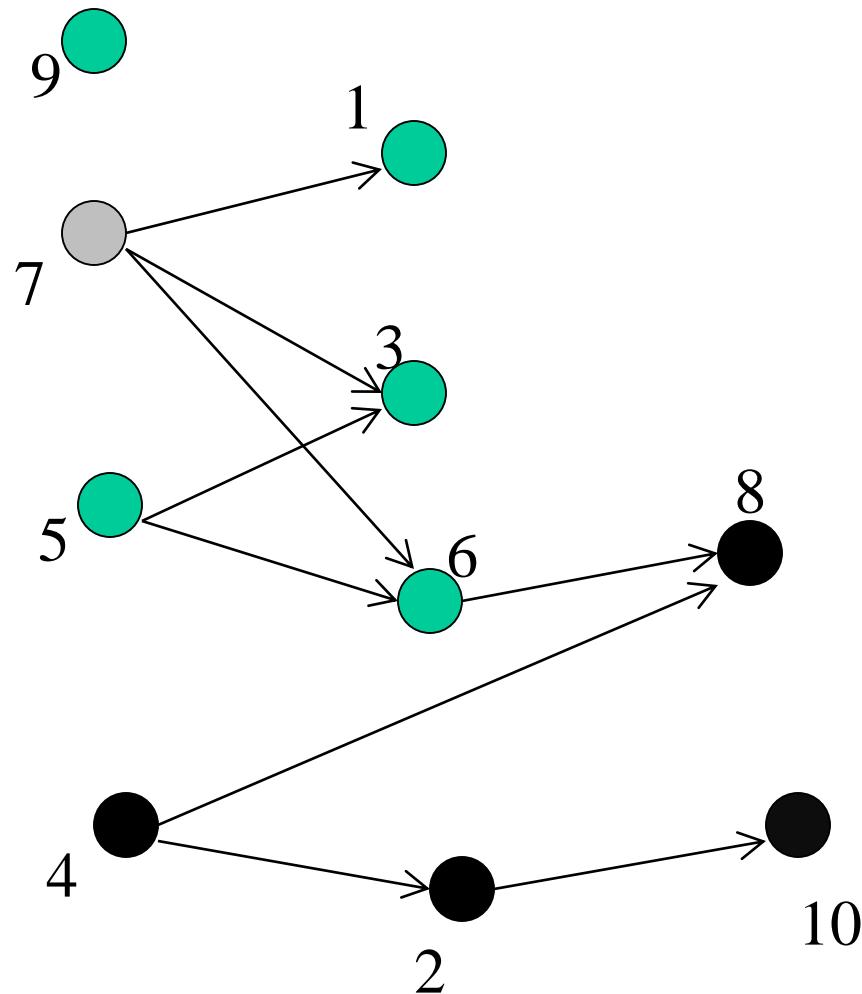
Visited

4
8
2
10

Stack

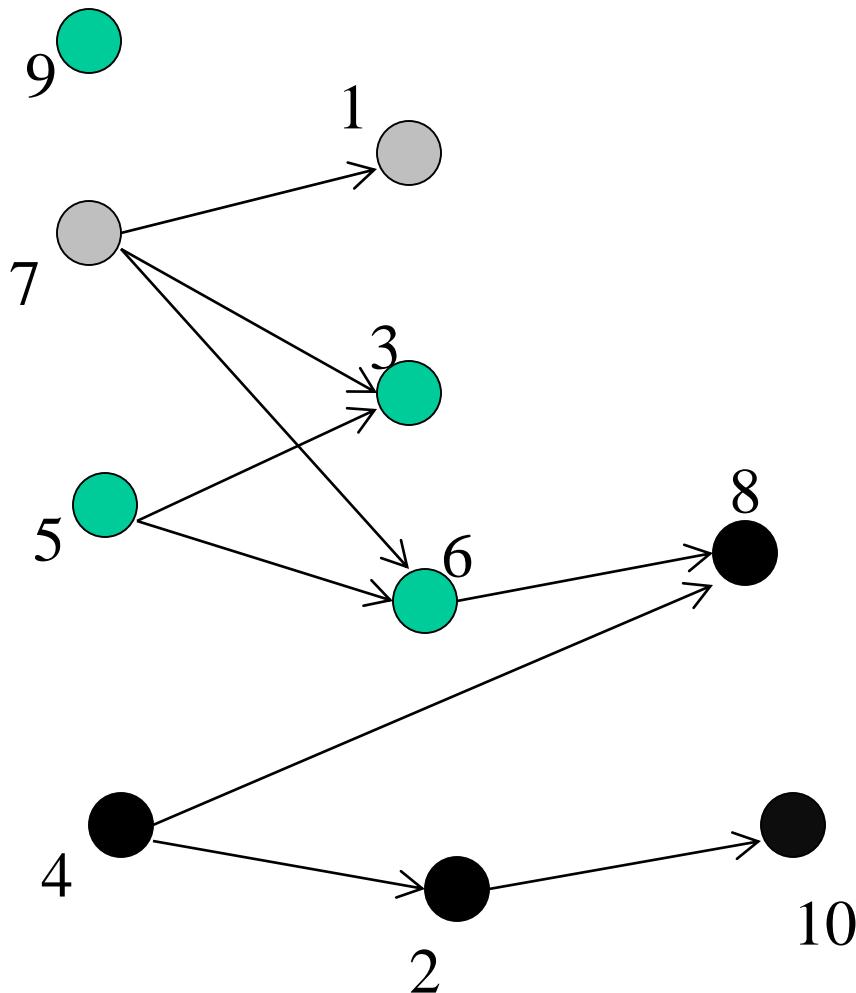
Select any unvisited vertex from the graph.

Here, Select vertex 7 and it marks as visited.



Find adjacent vertices of 7. There is 1, 3 & 6 .

Select Least vertex 1. Mark as visited.



<b>1</b>
7
8
10
2
4

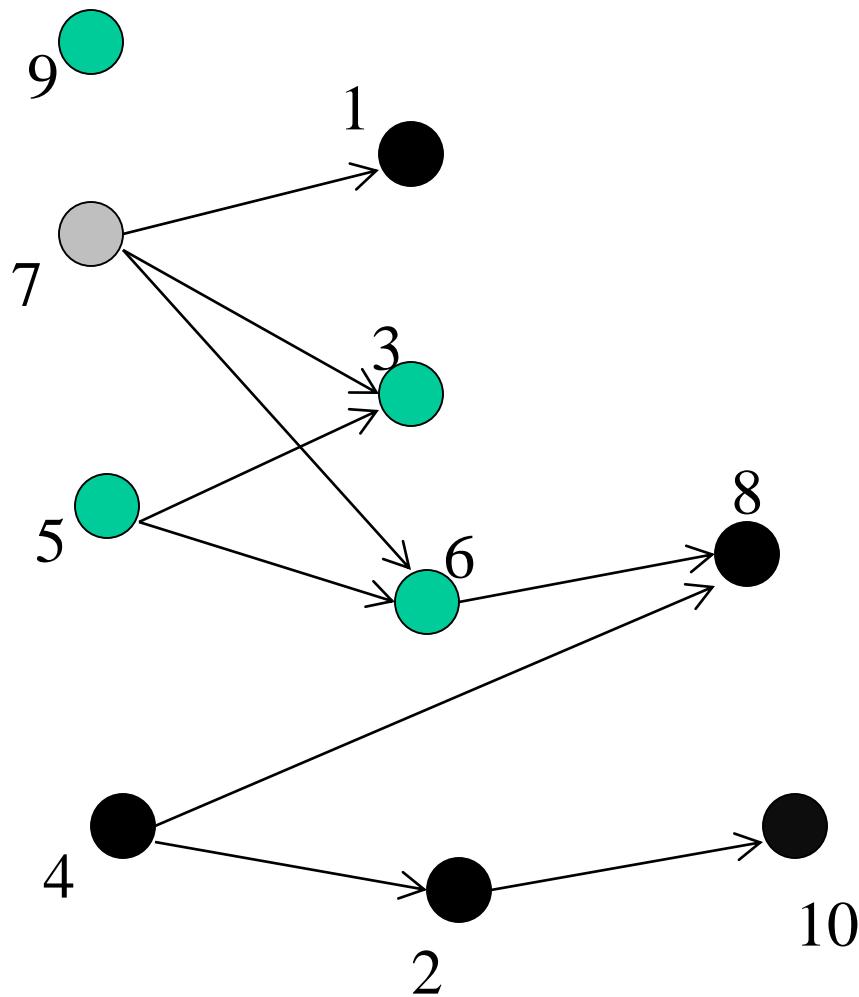
**Visited**

<b>4</b>
8
2
10

**Stack**

Find adjacent vertices of 1. There is no adjacent vertices for 1.

Hence insert vertex 1 into stack for scheduling. Then go back to parent vertex 7.



1
7
8
10
2
4

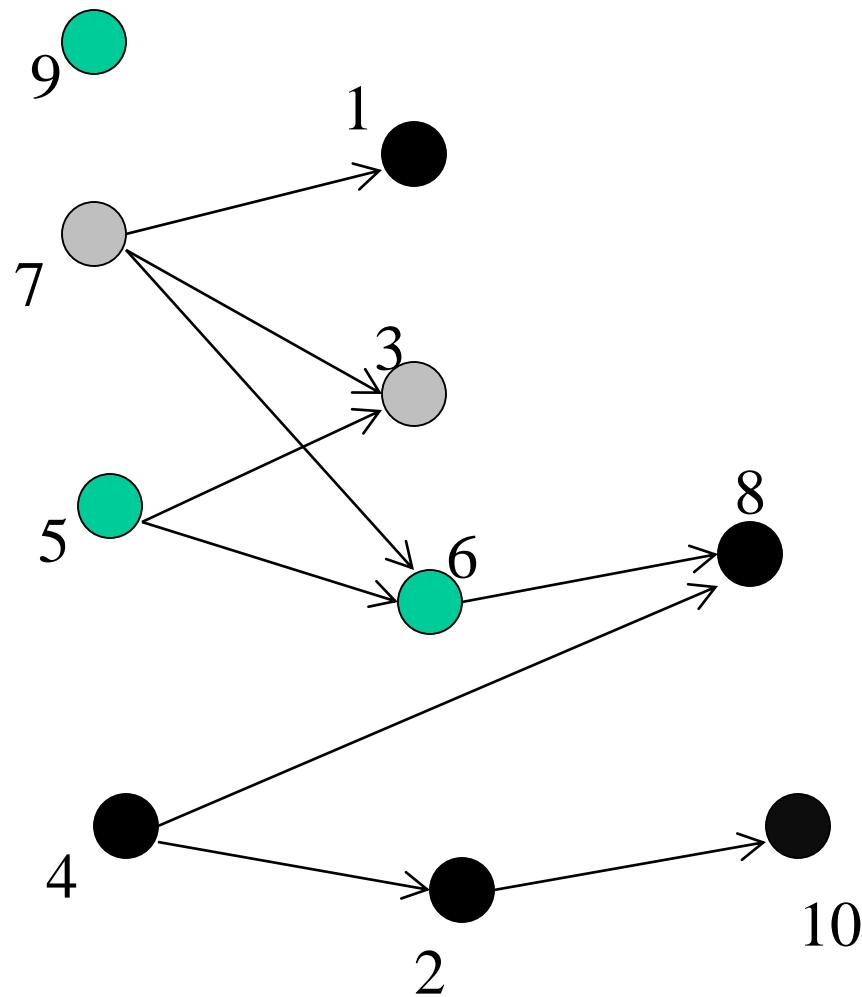
Visited

1
4
8
2
10

Stack

Find unvisited adjacent vertices of 7. There is 3 & 6.

Select Least vertex 3. Mark as visited.



<b>3</b>
1
7
8
10
2
4

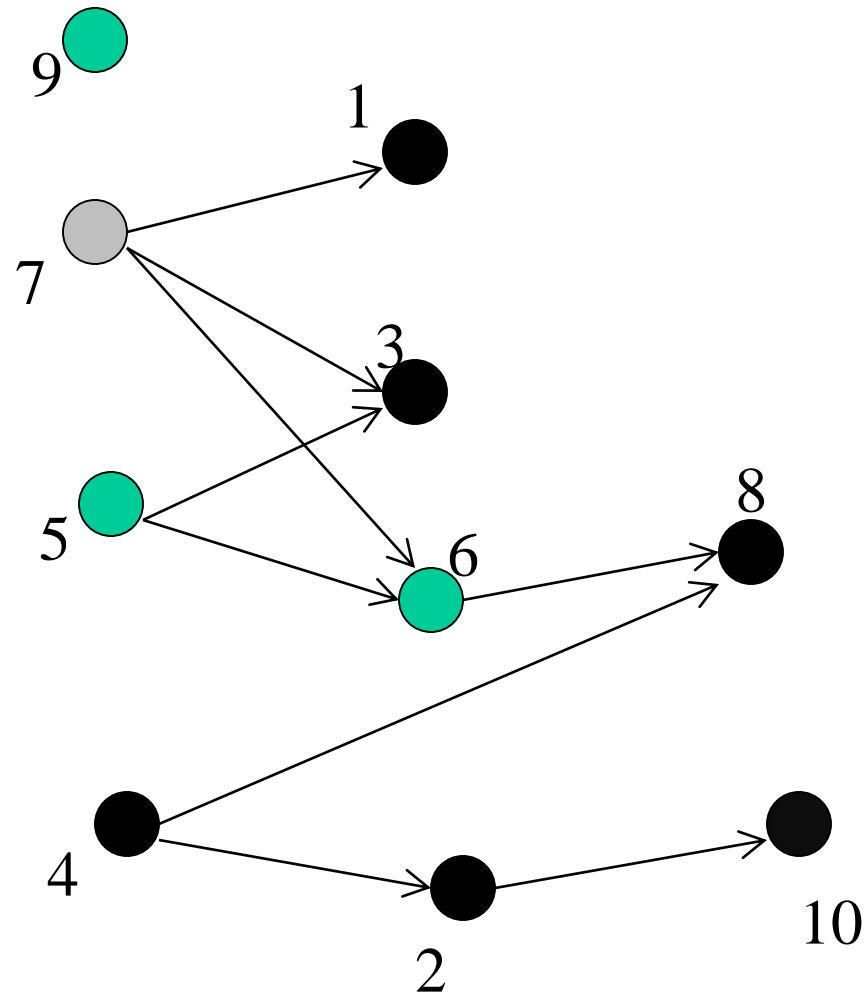
Visited

<b>1</b>
4
8
2
10

Stack

Find adjacent vertices of 3. There is no adjacent vertices for 3.

Hence insert vertex 3 into stack for scheduling. Then go back to parent vertex 7.



3
1
7
8
10
2
4

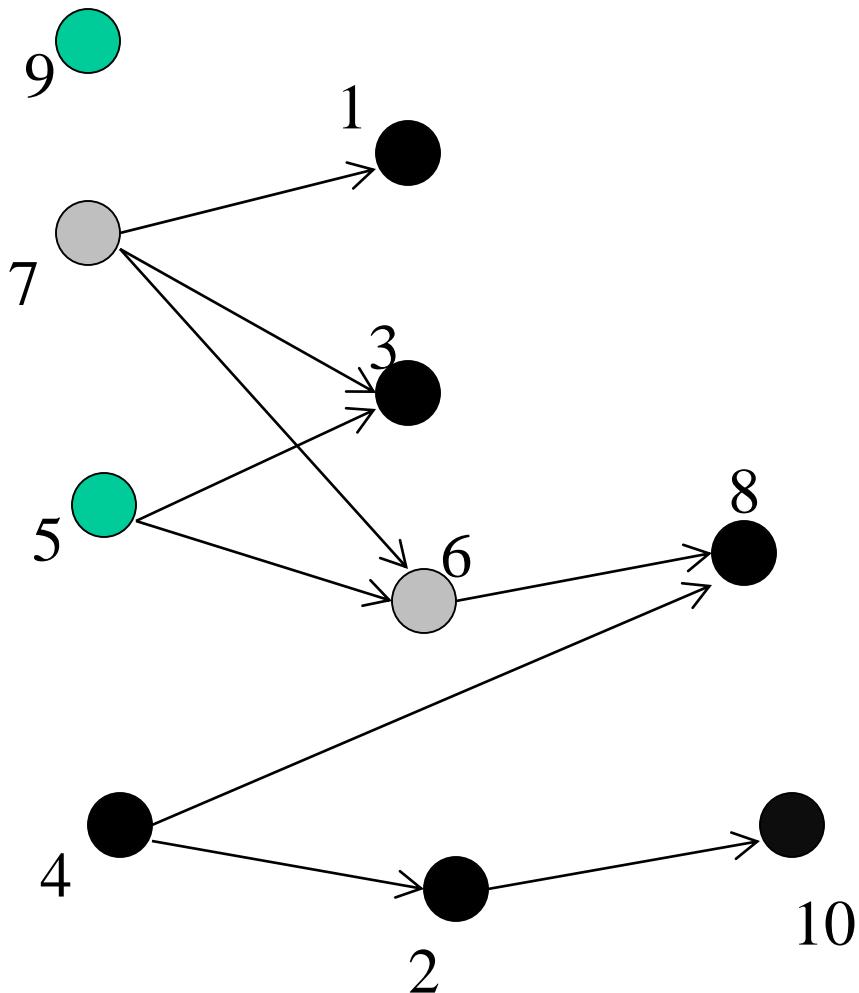
Visited

3
1
4
8
2
10

Stack

Find unvisited adjacent vertices of 7. There is 6.

Select vertex 6 and Mark as visited.



<b>6</b>
3
1
7
8
10
2
4

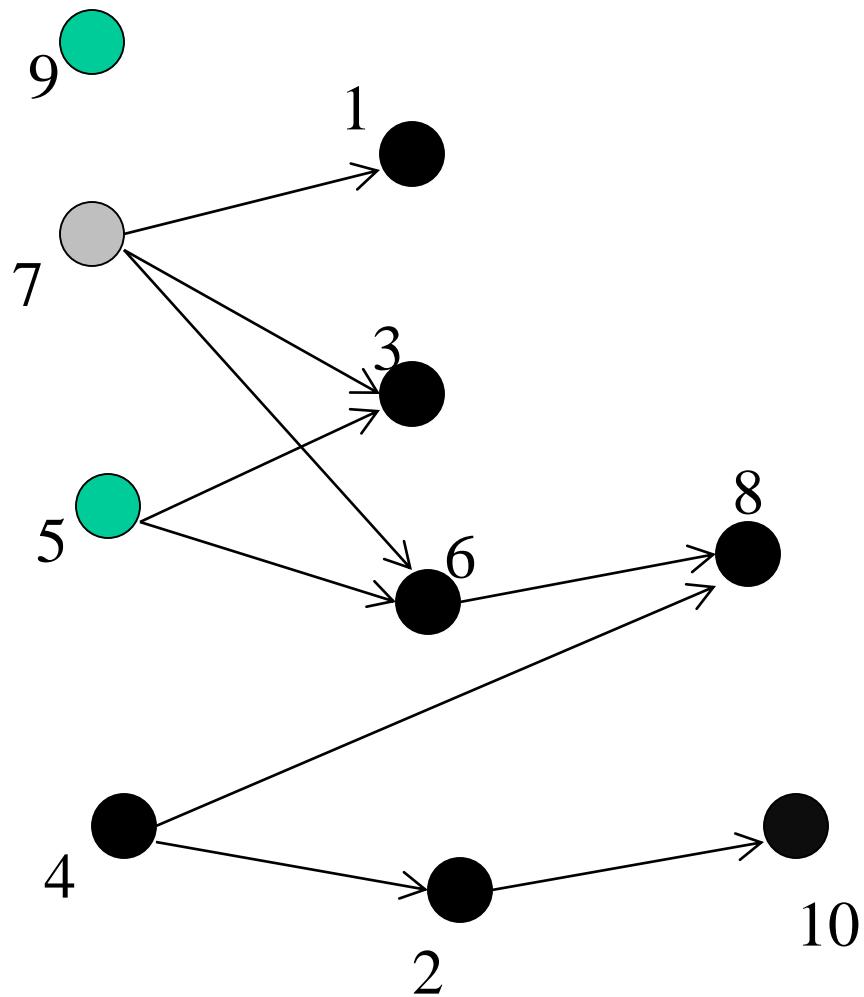
Visited

<b>3</b>
1
4
8
2
10

Stack

Find adjacent vertices of 6. There is no adjacent vertices for 6.

Hence insert vertex 6 into stack for scheduling. Then go back to parent vertex 7.



6
3
1
7
8
10
2
4

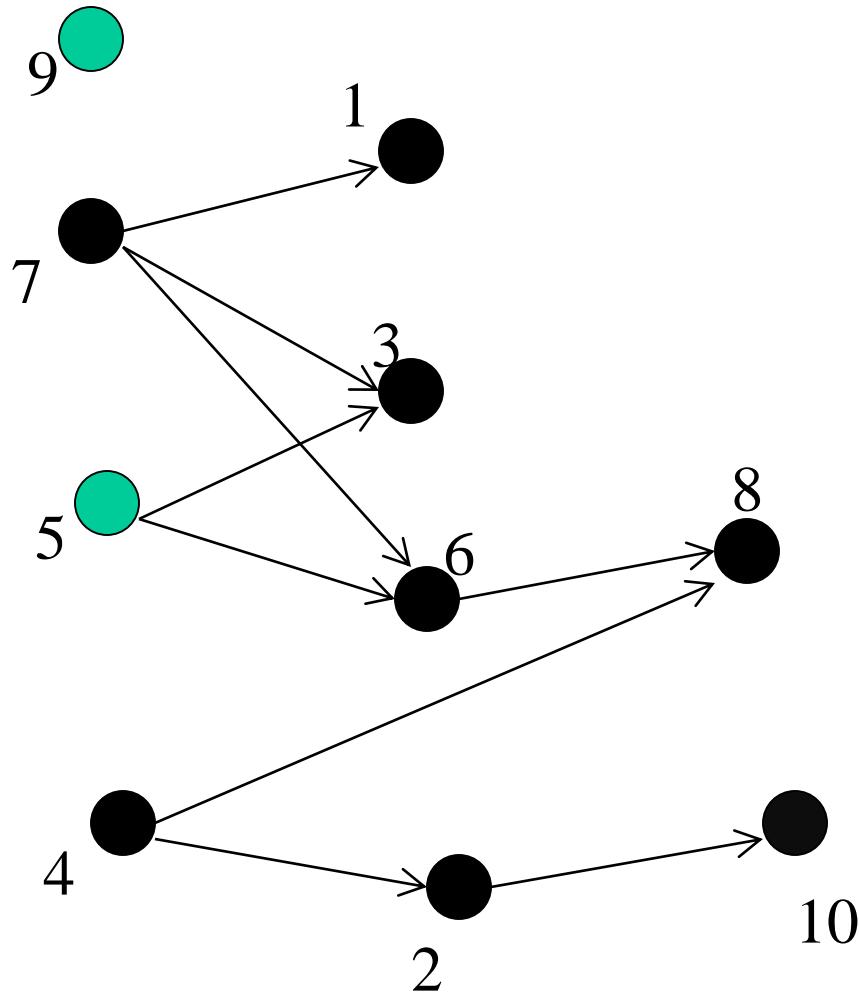
Visited

<b>6</b>
3
1
4
8
2
10

Stack

Find unvisited adjacent vertices of 7. There is no unvisited adjacent vertices for 7.

Hence insert vertex 7 into stack for scheduling. Then select any other unvisited vertex.



6
3
1
7
8
10
2
4

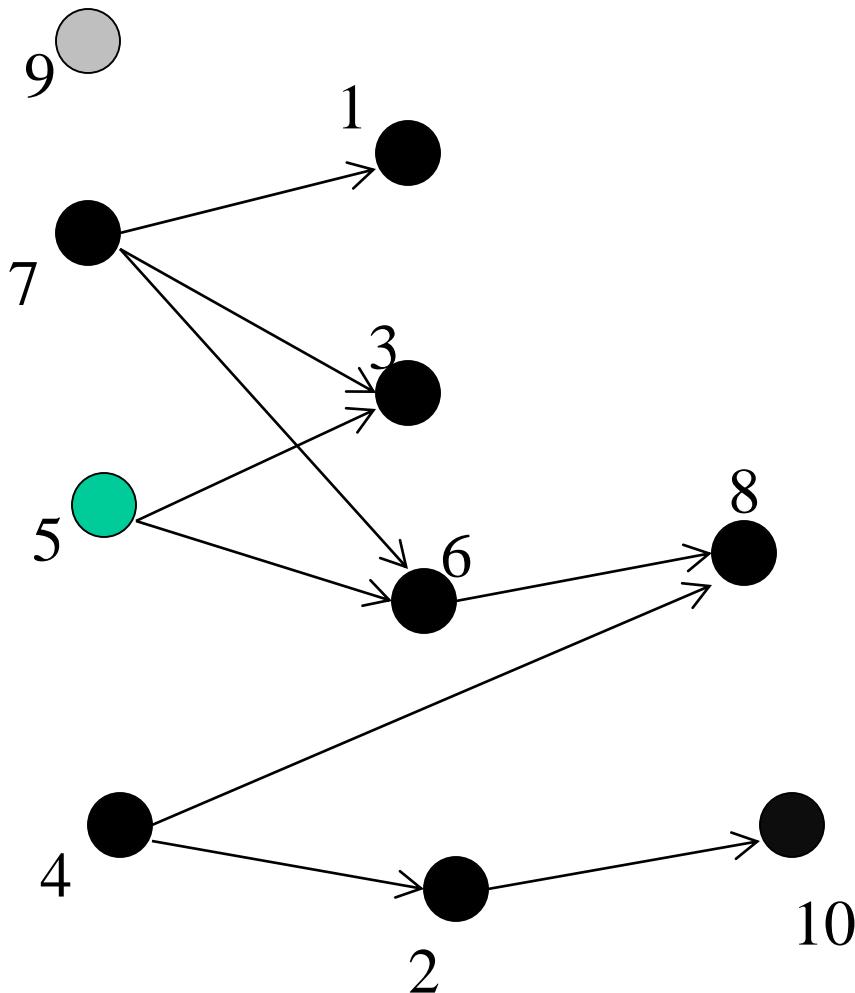
Visited

7
6
3
1
4
8
2
10

Stack

Select any unvisited vertex from the graph.

Here, Select vertex 9 and it marks as visited.



<b>9</b>
6
3
1
7
8
10
2
4

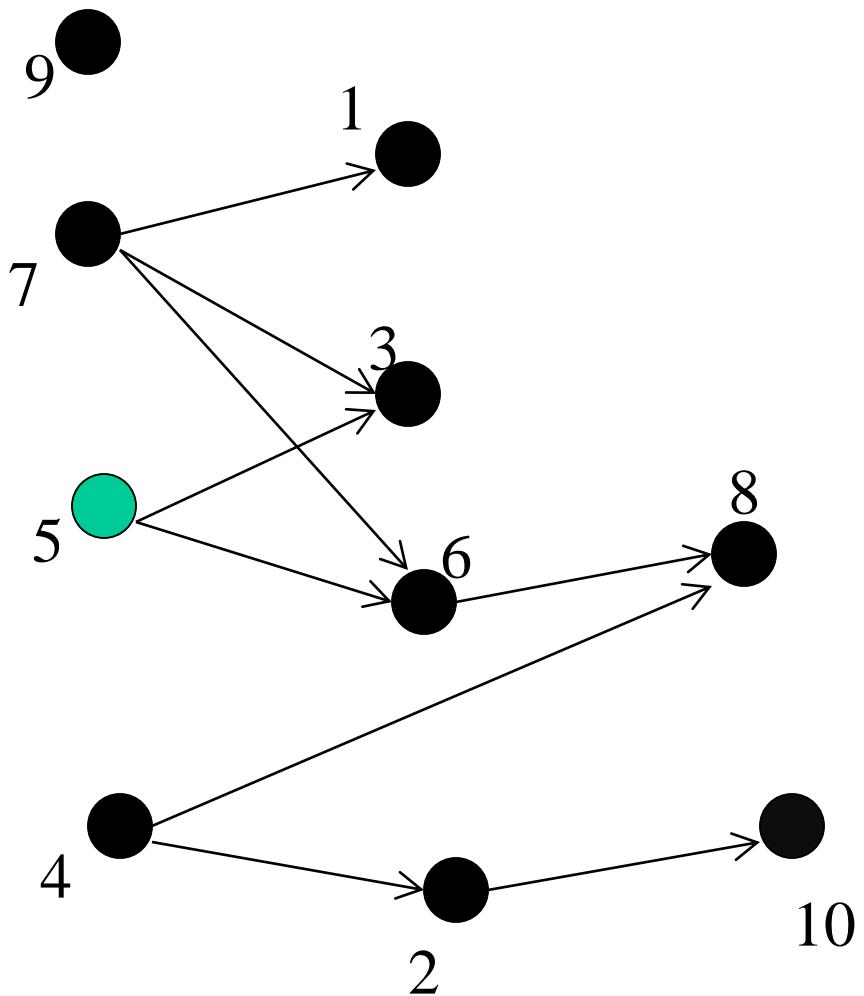
**Visited**

7
6
3
1
4
8
2
10

**Stack**

Find unvisited adjacent vertices of 9. There is no unvisited adjacent vertices for 9.

Hence insert vertex 9 into stack for scheduling. Then select any other unvisited vertex.



9
6
3
1
7
8
10
2
4

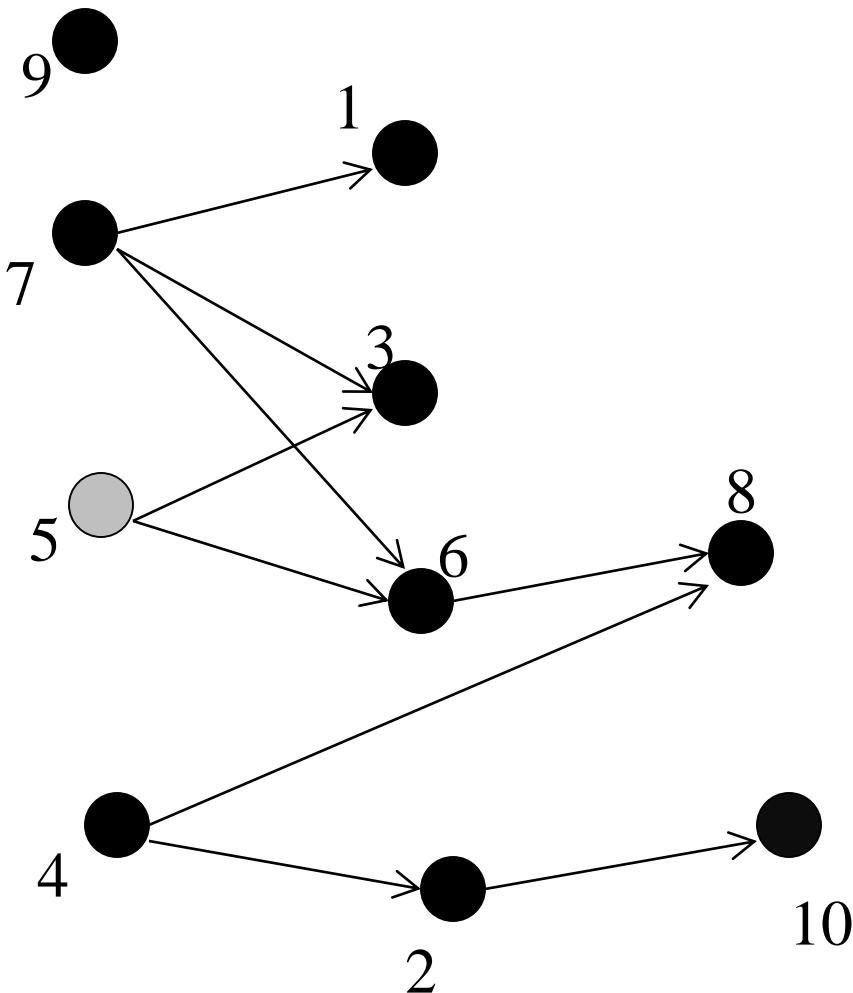
Visited

9
7
6
3
1
4
8
2
10

Stack

Select any unvisited vertex from the graph.

Here, Select vertex 5 and it marks as visited.



5
9
6
3
1
7
8
10
2
4

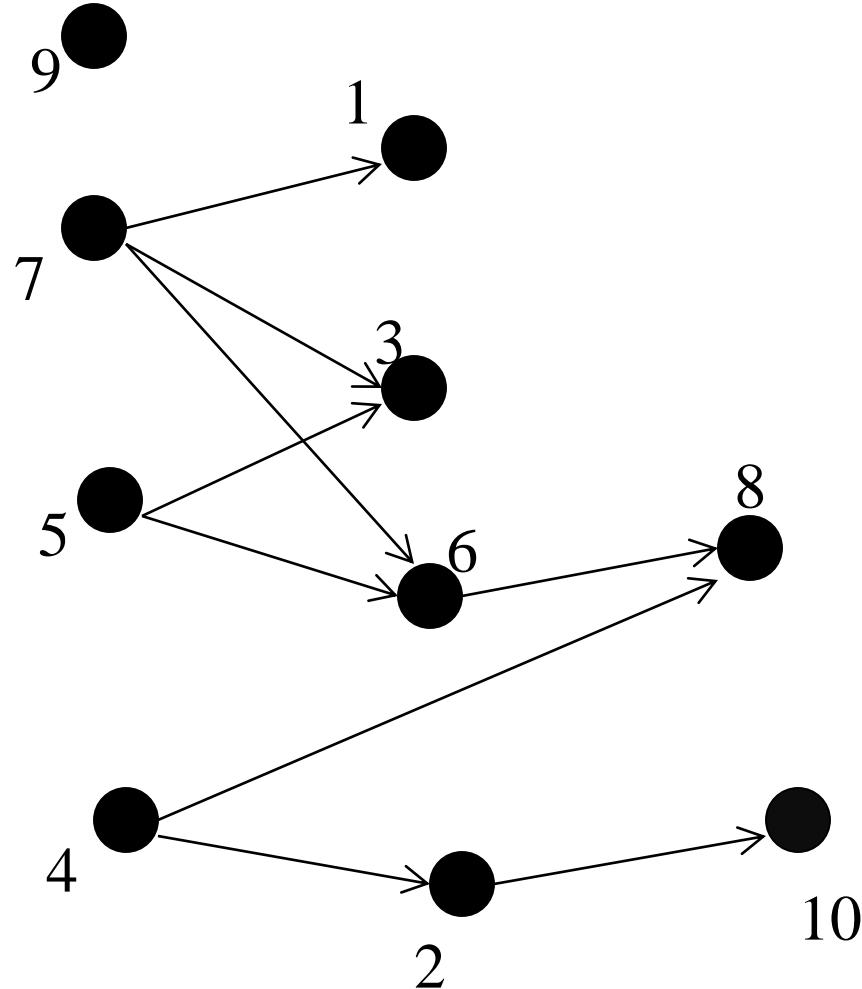
Visited

9
7
6
3
1
4
8
2
10

Stack

Find unvisited adjacent vertices of 5. There is no unvisited adjacent vertices for 5.

Hence insert vertex 5 into stack for scheduling. Now All vertices are visited.



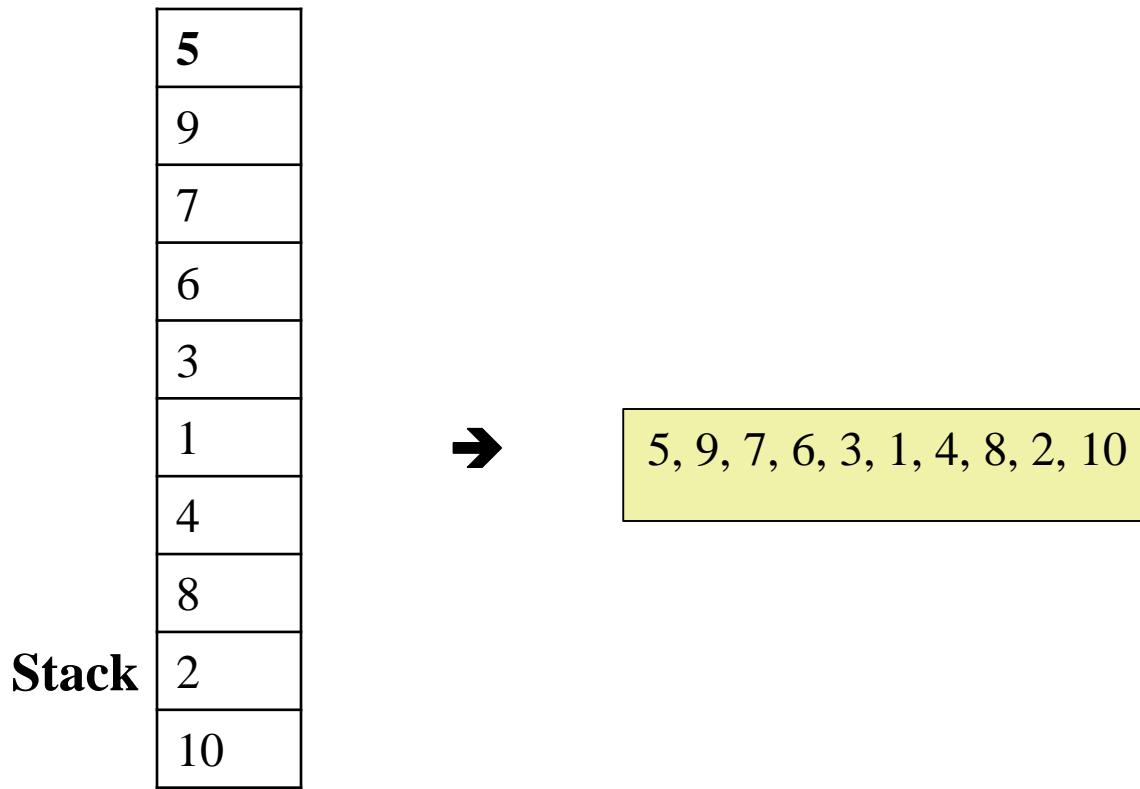
5
9
6
3
1
7
8
10
2
4

Visited

5
9
7
6
3
1
4
8
2
10

Stack

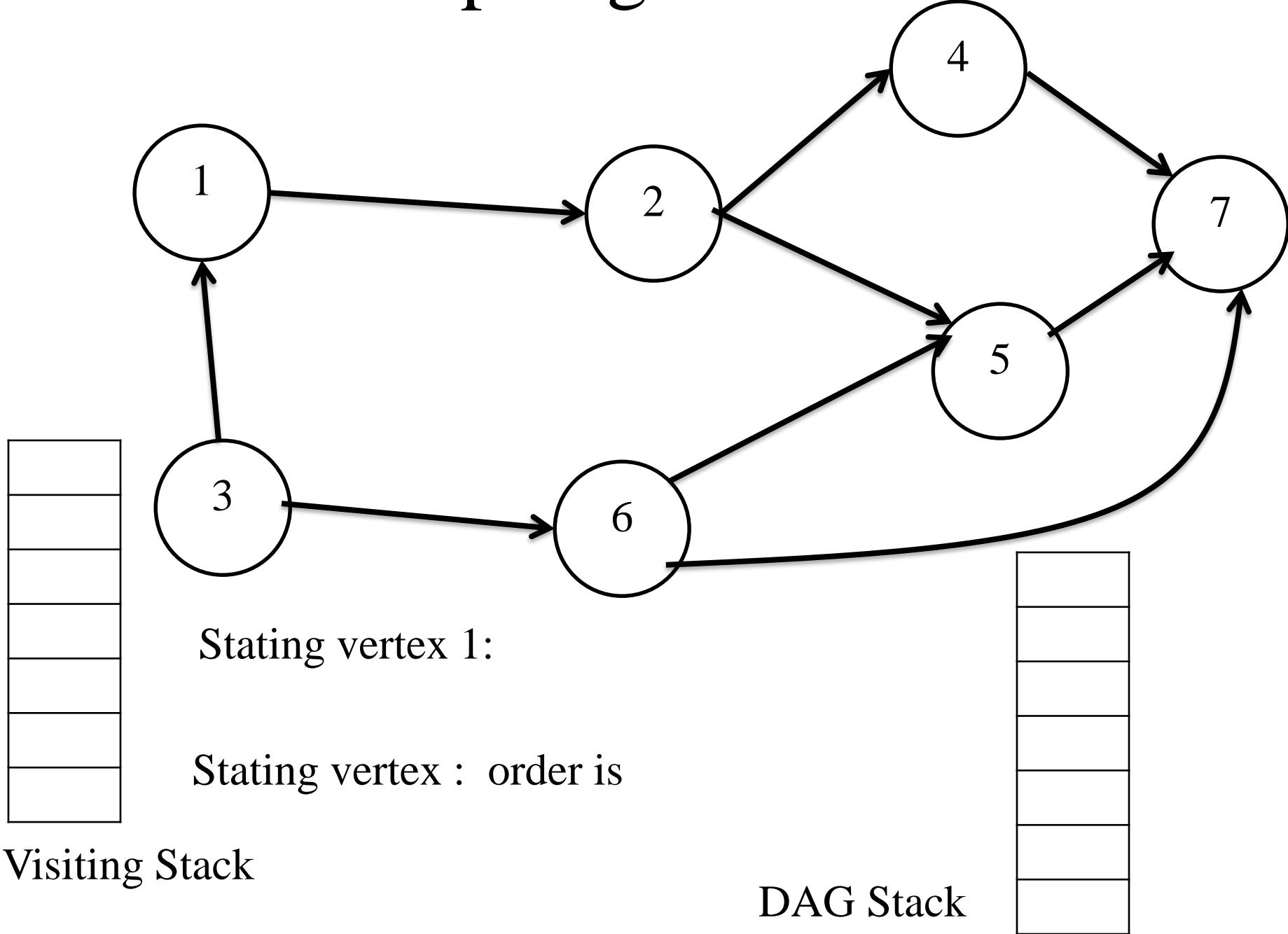
# The final order or jobs is



- There can be many orders that meet the requirements

- Time complexity = DFS complexity  $O(V + E)$

# Solve it: Topological Sort- DAG

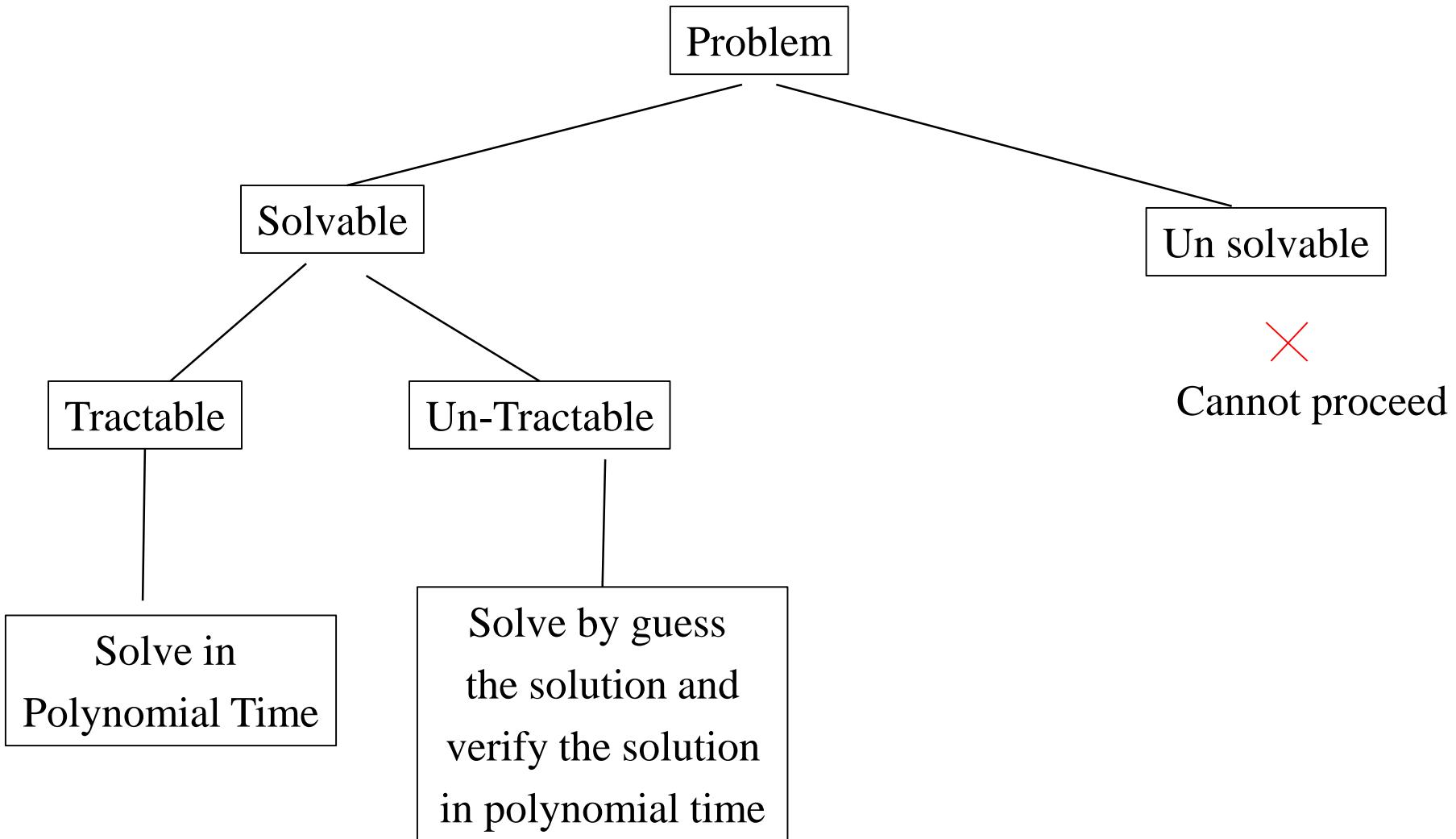


## Topological Sort: Applications

- instruction scheduling,
- ordering of formula cell evaluation when recomputing formula values in spreadsheets,
- determining the order of compilation tasks to perform in makefiles,
- data serialization, and
- resolving symbol dependencies in linkers

# **P, NP–HARD AND NP–COMPLETE PROBLEMS**

# Problem Structure



# The class P

- $P = \{ L \mid \text{Language } L\}$
- Problems solved by a **deterministic algorithms** (Turing Machine) in **Polynomial time** (i.e. User knows the working time of every statement in algorithm).
- More specifically, problems can be solved in time  **$O(n^c)$**  for some constant  $c$ , where  $n$  is the size of the input to the problem.

## Example Algorithms:

### Polynomial Time Complexity

Linear Search	- $O(n)$
Binary Search	- $O(\log n)$
Merge sort	- $O(n \log n)$
Quick sort	- $O(n \log n)$
Insertion sort	- $O(n^2)$
Matrix multiplication	- $O(n^3)$

### Exponential Time Complexity

• 0/1 Knapsack	- $O(2^n)$
• Travelling SP	- $O(2^n)$
• Sum of subsets	- $O(2^n)$
• Graph coloring	- $O(2^n)$
• Hamiltonian cycle	- $O(2^n)$
• Satisfiability	- $O(2^n)$

# The Class NP (Non deterministic Polynomial.)

- $NP = \{ L \mid \text{Language } L\}$
- Problems solved by a **Non-deterministic** algorithms (Turing Machine) (i.e. User not known execution time of few statements in algorithm) and **verified** in polynomial time.

**Algorithm:** Search an element from Array A with size n.

```
Nsearch(A,n,key)
{
    j=choice();// Non-Deterministic
    If(key==A[j])
    {
        write(j);
        Success();
    }
    Write(0);
    Failure();
}
```

## The Class NP (Non deterministic Polynomial.)

Nsearch(A,n,key)

{

J=choice() //time O(1), How to select (which algorithm) position in O(1)?

if(key==A[j])

{

write(j);

Success(); //time O(1)

}

Write(0);

Failure(); //time O(1)

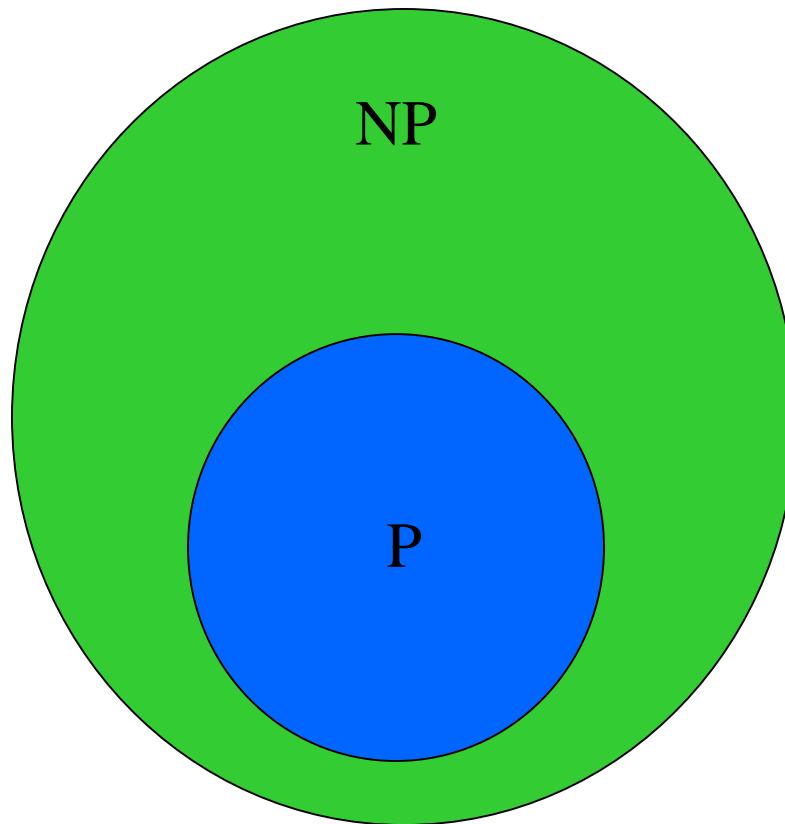
}

10	3	4	5
A[0]	A[1]	A[2]	A[3]

How to select choice 4 is in location (key) 2 in O(1). As of now it is not possible. But it may be possible in future. So it is Non deterministic.

# Two Classes P & NP

- Initially Problem belongs to NP, i.e. solved by **Non-deterministic** algorithms in Polynomial Time.
- Once algorithm found to solve in polynomial time it becomes P class problem.
- P is subset of NP.



# Why we need NP?

- Exponential time algorithms takes more time than polynomial time algorithms for very large values of n.

For Example:

$n^{10}$  Takes less time than  $2^n$  for large values.

$$n=100 \rightarrow 100^{10} = 100 e +18$$

$$2^{100} = 1.267 e +30$$

- So **Need of polynomial time** algorithms, but its Unknown (Non Deterministic) in this moment to reduce time of Exponential complexity problem.

## Two NP Classes:

1. NP Hard
2. NP Complete

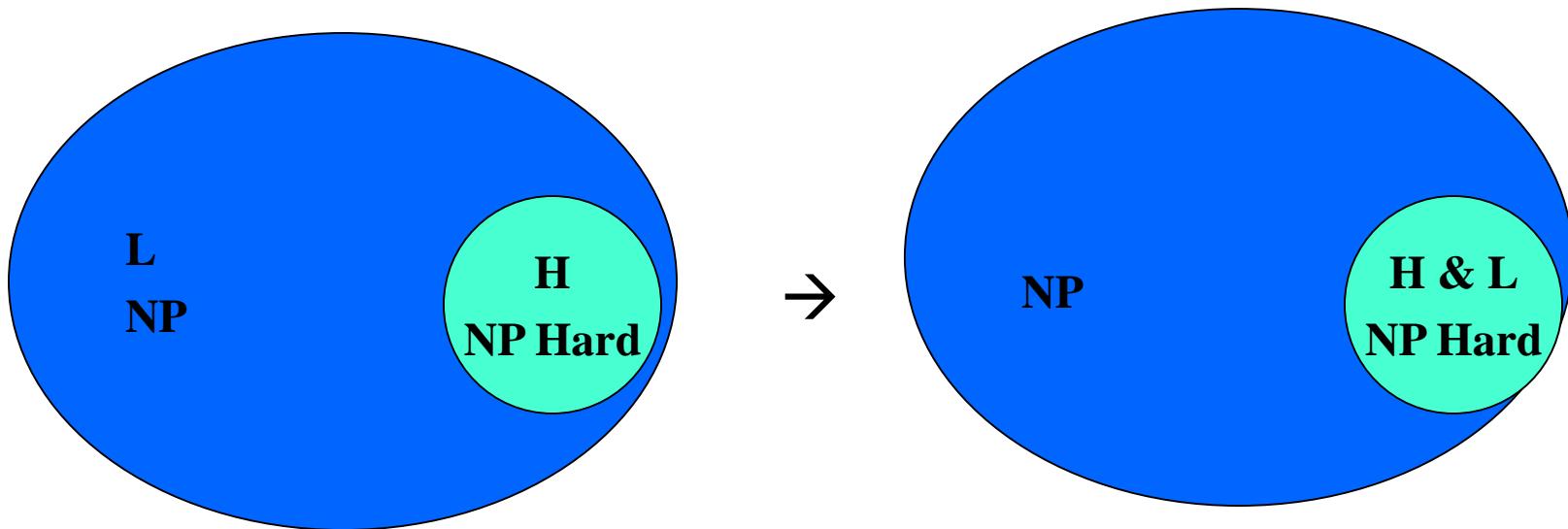
# The Class NP Hard

## NP Hard:

- A problem (a language) is said to NP-hard if every problem in NP can be reduced in to polynomial time.

For Example:

- Assume problem H is NP hard.  $\rightarrow$  Non-Det  $\rightarrow$  NP  $\rightarrow$  P
- When another problem L is NP and **if it is able to reduce into H**, then L becomes NP Hard.

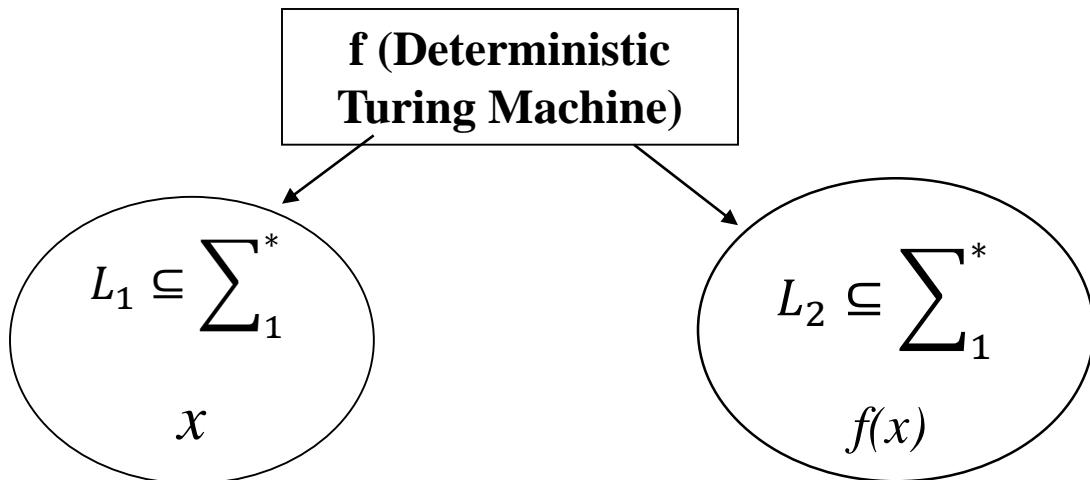


# NP - hard

$$L_1 \leq_p L_2$$

- That notation means that  $L_1$  is **reducible** (convert) to  $L_2$  in polynomial time.
- The less than symbol means that the time taken to solve  $L_1$  is no worse than a polynomial factor away from the time taken to solve  $L_2$ .

# Reducibility (Polynomial Transformation)



$$L_1 \subseteq \sum_1^* \quad L_2 \subseteq \sum_2^* \quad f: \sum_1^* \rightarrow \sum_2^*$$

Satisfy the following condition for transformation:

- Polynomial time DTM program that compute  $f$ .
- For all  $x \in \sum_1^*$ ,  $x \in L_1$ , if  $f(x) \in L_2$ , then  $L_1 \alpha L_2$

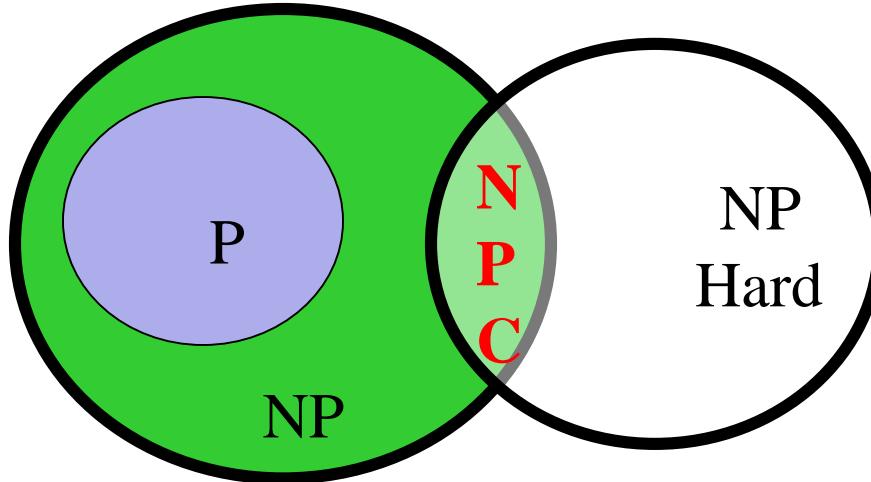
**Property:** 1. If  $L_1 \alpha L_2$  and  $L_2$  in P then  $L_1$  is in P.

2. If  $L_1 \alpha L_2$  and  $L_2$  is not in P then  $L_1$  is not in P.

**Equivalence:**  $L_1$  and  $L_2$  are polynomially equivalent if  $L_1 \alpha L_2$  and  $L_2 \alpha L_1$

# NP-Completeness

- If problem satisfies both NP and NP-Hard then it is an NP complete problem  
i.e. A problem Q is called NP complete iff Q is **NP Hard** and  $Q \in NP$

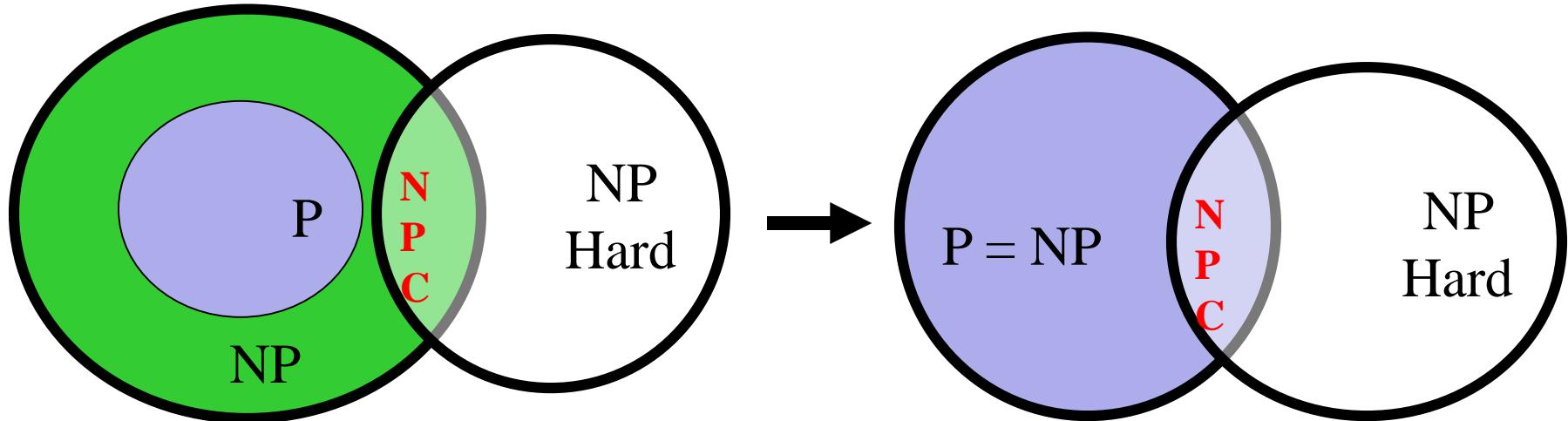


- If any NP Complete Q is in P then  $P=NP$ .
- If any NP Complete Q is not in P then  $P \neq NP$ .

# Proof P=NP

If any NP Complete Q is in P then P=NP.

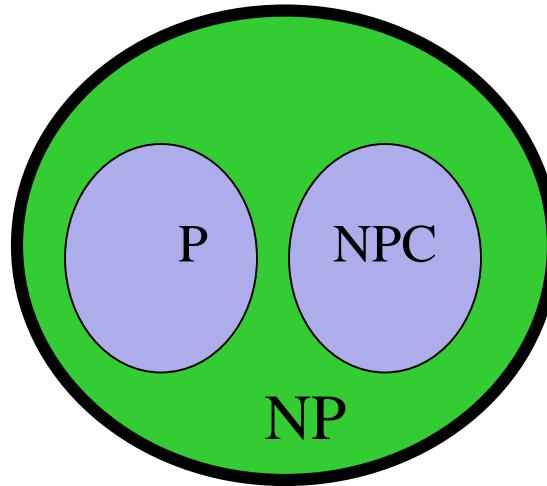
- Assume R is NP Complete.
  - Problem Q is NP Hard and it reduces to R. (i.e.  $Q \leq R$ )
  - Now Q is also NP Complete.
- 
- Thus any NP Complete problem R is in P then  $P = NP$ .



# Proof $P \neq NP$ .

If any NP Complete Q is not in P then  $P \neq NP$ .

- Assume R is **not NP Complete**.
- If no NP complete is there, then no possibility of reduction for Q.
- Now Q is also not NP Complete.
- Both P and NP are **separate**.
- Thus any NP Complete problem is not in P then  $P \neq NP$ .



# Satisfiability Problem - NP-Complete

- A boolean formula is *satisfiable* if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to **1 (TRUE)**.
- CNF – Conjunctive Normal Form. **ANDing** of clauses of **ORs**
- $3\text{-SAT} = \{ f \mid f \text{ is in Conjunctive Normal Form, each clause has exactly 3 literals } (x_0, x_1, x_2) \text{ and } f \text{ is satisfiable} \}$
- $\text{SAT } \emptyset = (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_2)$

**Clause 1**

**Clause 2**

**Clause 3**

- Answer of the above expression should be **TRUE**.
- Change the values of  $(x_0, x_1, x_2)$  to get True. 000, 001, 010, 011, 100, 101, 110, 111  $\rightarrow 2^3$
- $(X_1 \vee X_2 \vee X_3) \text{ AND } (X_1 \vee X_2 \vee X_3)$

# **NP-Completeness Proof Method**

To show that Q is NP-Complete:

- 1) Show that Q is in NP.
- 2) Pick an instance, R, of your favorite NP-Complete problem (ex:  $\Phi$  in 3-SAT)
- 3) Show a polynomial algorithm to transform R into an instance of Q.

**Example:**

- Assume R is SAT.
- Knapsack is Q.
- Then Transform SAT into instance of Q in polynomial time.

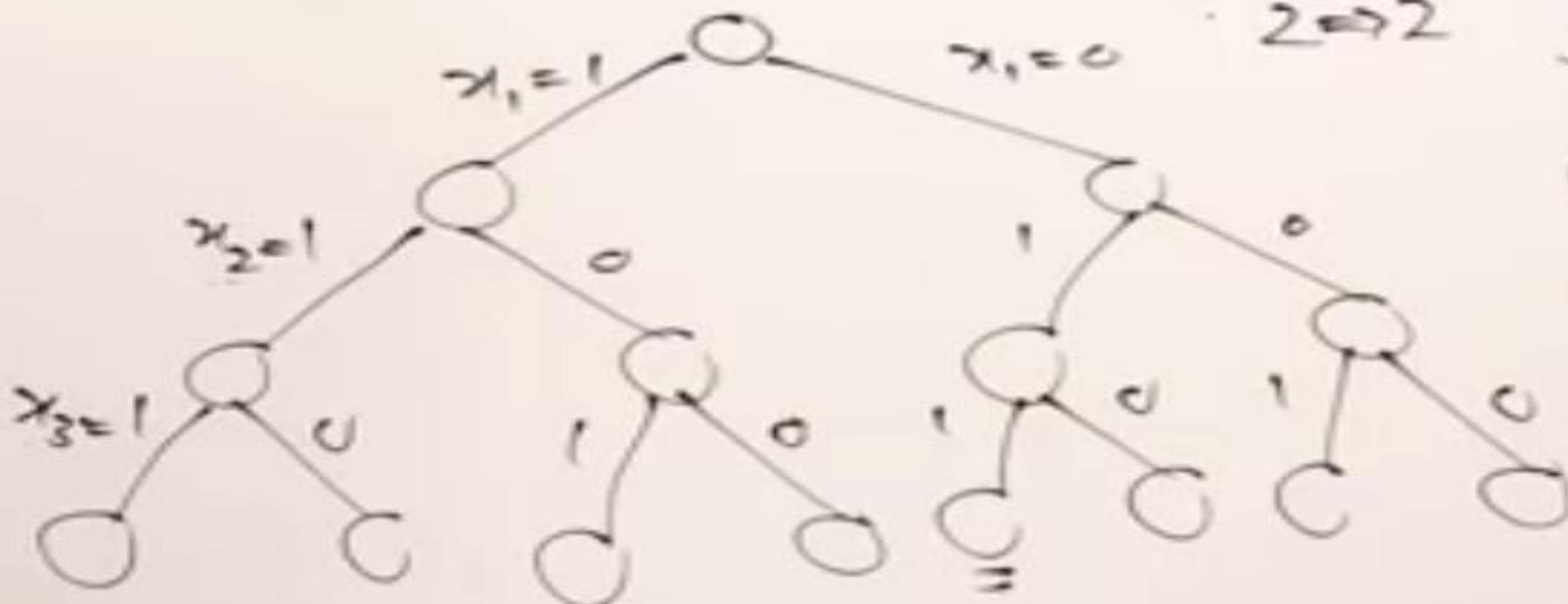
## Knapsack Problem

- Prove the Knapsack is NP Complete. Knapsack capacity is 10 with 3 items with weights 5, 4, 3 and corresponding profit is 10, 8, 12.

0/1 Knapsack

$$P = \{10, 8, 12\} \quad n=3$$
$$w = \{5, 4, 3\} \quad m=8$$
$$x_i = \{\underline{0/1}, \underline{0/1}, \underline{0/1}\}$$

$$\begin{matrix} x_1 & x_2 & x_3 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{matrix} \quad \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix}$$



## Reduce 3 CNF Satisfiability Problem into Knapsack

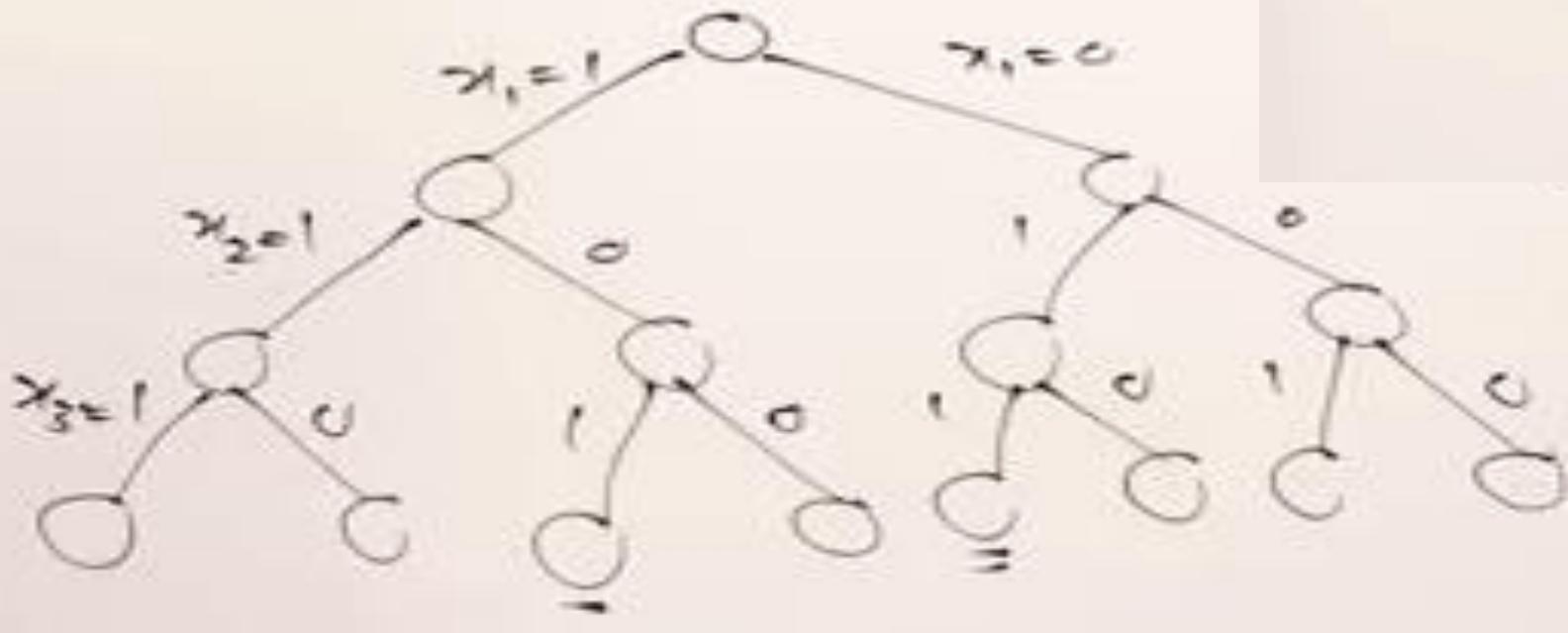
CNF-Satisfiability  $\longrightarrow 2^n$

$$x_i = \{x_1, x_2, x_3\}$$

$$\text{CNF} = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

$c_1 \qquad \qquad \qquad c_2$

$2^3$	$x_1$	$x_2$	$x_3$
2	0	0	0
3	0	0	1
4	0	1	0
5	0	1	1
6	1	0	0
7	1	0	1
8	1	1	0
9	1	1	1



- SAT  $\alpha$  Knapsack, SAT is already proved as NP Complete.
- Hence knapsack is also NP Complete.

# **NP-Complete**

- To prove a problem is NP-Complete show a polynomial time reduction from 3-SAT
- Other NP-Complete Problems:
  - PARTITION
  - SUBSET-SUM
  - CLIQUE
  - HAMILTONIAN PATH (TSP)
  - GRAPH COLORING
  - MINESWEEPER (and many more)

## The Class NP Complete

- A problem can be solved in polynomial time if and only if all other NP complete problems can also be solved in polynomial time.
- If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
- All NP-complete problems are NP-hard, but some NP-hard problems are not NP-complete.

**LINEAR PROGRAMMING  
&  
ADVANCED ALGORITHMS**

# Linear Programming (LP)

- LP is a mathematical modeling technique used to calculate optimal result in order to achieve an objective (Maximize profit or Minimize loss), subject to restrictions (constraints).
- A mathematical model consisting of linear relationships representing an (firm's) **objective** and **resource constraints**.
- LP allots resources in an optimal manner based on the criteria of optimality.

# LP Model Formulation

- ◆ **Decision variables**
  - Mathematical symbols representing things that can be adjusted or controlled.
  - Variable values are usually unknown when a problem stated.
  - **Ex:** Cost spend during product manufacturing.
  - **Goal:** Find the values of the variables that provide the best value of the objective function.

# LP Model Formulation

- ◆ **Objective function**
  - Mathematical expression that combines the variables to indicate the requirements. It represents to *maximize* / *minimization* of objective function.
  - Most frequent objective of business firms is to *maximize profit*
  - Most frequent objective of **individual operational units** (such as a production or packaging department) is to *minimize cost*
- ◆ **Constraint**
  - A linear relationship representing a restriction on decision making.
  - **Ex:** No. of workers available to operate a particular machine is limited.

## Steps for Formulation of LP Problems (LPP)

1. Write down the decision variables of the problem
2. Formulate the objective function (to be max / min) as a linear function with decision variables
3. Formulate the other constraints / conditions of the problem such as resource limitation, market constraints, interrelations between variables as linear equalities/ inequalities in terms of decision variable.
4. To add the non negativity constraint from the considerations so that the negative value of the decision variables do not have any physical interpretations.

Hence Objective function is combination of constraints and non negative restrictions.

# General LPP Formulation

**Max/min**      
$$z = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

**subject to:**

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n (\leq, =, \geq) b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n (\leq, =, \geq) b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n (\leq, =, \geq) b_m \end{array} \right.$$

$x_j$  = decision variables

$b_i$  = constraint levels

$c_j$  = objective function coefficients

$a_{ij}$  = constraint coefficients

# LPP Problem (Maximization)

1. A manufacturer produces two types of models M1 and M2. M1 requires 4 hours of grinding and 2 hours of polishing. M2 requires 2 hours of grinding and 5 hours of polishing. The manufacturer has 2 grinders and 3 polishers. Each grinder works 40 hours a week and each polisher works for 60 hours a week. Profit on M1 model is Rs. 3.00 and M2 is Rs. 4.00. Products are produced sold in a week in the market. How should the manufacturer allocate his production capacity to two models, so that he may make the maximum profit in a week?

# LPP Problem Formulation

- ◆ Find Decision variables.

Let  $x_1, x_2$  be the variables for models M1 and M2.

- ◆ Objective function

Since the profit on both models are given, Maximize the profit as

$$\text{Max } Z = 3x_1 + 4x_2$$

- ◆ Constraints

Two constraints. One is Grinding and Other is polishing.

No. of hours allot per grinder in one week = 40

No. of grinders = 2

So, Maximum grinding hours is  $2 \times 40 = 80$  hours

## LPP Problem

- ◆ M1 requires 4 hours of grinding and M2 requires 2 hours of grinding.
- ◆ The grinding constraints is given by

$$4x_1 + 2x_2 \leq 80$$

- ◆ No, of polishers = 3
- ◆ So, Maximum polishing hours is  $3 \times 60 = 180$  hours
- ◆ M1 requires 2 hours of polishing and M2 requires 5 hours of polishing.
- ◆ The polishing constraints is given by

$$2x_1 + 5x_2 \leq 180$$

- ◆ Finally,  $\text{Max } Z = 3x_1 + 4x_2$
- ◆ Subject to  $4x_1 + 2x_2 \leq 80$  and  $2x_1 + 5x_2 \leq 180$ , where  $x_1, x_2 \geq 0$

# LP Model: Example 2

RESOURCE REQUIREMENTS			
PRODUCT	<i>Labor (hr/unit)</i>	<i>Clay (lb/unit)</i>	<i>Revenue (\$/unit)</i>
Bowl	1	4	40
Mug	2	3	50

There are 40 hours of labor and 120 pounds of clay available each day

Decision variables

$x_1$  = number of bowls to produce

$x_2$  = number of mugs to produce

# LP Formulation: Example

Maximize  $Z = \$40 x_1 + 50 x_2$

Subject to

$$x_1 + 2x_2 \leq 40 \text{ hr} \quad (\text{labor constraint})$$

$$4x_1 + 3x_2 \leq 120 \text{ lb} \quad (\text{clay constraint})$$

$$x_1, x_2 \geq 0$$

Solution is  $x_1 = 24$  bowls       $x_2 = 8$  mugs

Revenue = \$1,360

# LPP - Minimization Problem

1. A chemist produces two types of fertilizers Gro-plus and Crop-fast. Gro-plus requires 2 units of nitrogen and 4 units of phosphate per bag. Crop-fast requires 4 units of nitrogen and 3 units of phosphate per bag. The chemist need minimum of 16 units nitrogen and 24 units phosphate. Cost of Gro-plus is Rs. 6.00 and Crop-fast is Rs. 3.00. How should the chemist minimize the cost of raw materials in production?

# LPP - Minimization Problem

CHEMICAL CONTRIBUTION		
<i>Brand</i>	<i>Nitrogen (lb/bag)</i>	<i>Phosphate (lb/bag)</i>
Gro-plus	2	4
Crop-fast	4	3

$$\text{Minimize } Z = \$6x_1 + \$3x_2$$

subject to

$$2x_1 + 4x_2 \geq 16 \text{ lb of nitrogen}$$

$$4x_1 + 3x_2 \geq 24 \text{ lb of phosphate}$$

$$x_1, x_2 \geq 0$$

# Linear programming

- ◆ Solution techniques:
  - (dual) simplex method
  - interior point methods (e.g. Karmarkar algorithm)
- ◆ Commercial solvers, for example:
  - CPLEX (ILOG)
  - XPRESS-MP (Dash optimization)
  - OSL (IBM)
- ◆ Modeling software, for example:
  - AIMMS
  - AMPL

# Simplex Method – General Method

1. Make sure that all  $b_i$  are positive values. If negative, multiply with -1 to make positive  $b_i$
2. Slack variables added to  $\leq$  constraints to represent unused resources. For Example
  - $x_1 + 2x_2 + s_1 = 40$  hours of labor
  - $4x_1 + 3x_2 + s_2 = 120$  lb of clay
3. Surplus variables subtracted from  $\geq$  constraints to represent excess above resource requirement. For example
  - $2x_1 + 4x_2 \geq 16$  is transformed into
  - $2x_1 + 4x_2 - s_1 = 16$
4. Slack/surplus variables have a 0 coefficient in the objective function. For Ex  
 $Z = \$40x_1 + \$50x_2 + 0s_1 + 0s_2$
5. Formulate the simplex table with above variables. Repeat upto find optimal solution.
6. Optimal Condition:
  - ♦ For Maximization: All  $C_j - Z_j \leq 0$  and For Min: All  $C_j - Z_j \geq 0$

## Simplex Method for LP Problem

- The given problem with Max  $Z = 5x_1 + 3x_2$

Subject to  $x_1 + x_2 \leq 2$

$$5x_1 + 2x_2 \leq 10,$$

$$3x_1 + 8x_2 \leq 12 \text{ where } x_1, x_2 \geq 0$$

- Make sure that all  $b_i$  are positive values. If negative, multiply with -1 to make positive  $b_i$

- No. of constraints = 3

- Convert LP into standard form by adding slack and surplus variables in constraints.

- Add 3 slack variables  $s_1, s_2, s_3$  since have three constraints to balance the constraints

- Max  $Z = 5x_1 + 3x_2 + 0s_1 + 0s_2 + 0s_3$

- Subject to  $x_1 + x_2 + s_1 = 2$

$$5x_1 + 2x_2 + s_2 = 10$$

$$3x_1 + 8x_2 + s_3 = 12 \text{ where } x_1, x_2, s_1, s_2, s_3 \geq 0$$

## Initial basic feasible solution:

Assume  $x_1, x_2 = 0$ , then  $s_1 = 2, s_2 = 10, s_3 = 12$

### Iteration I: Formulate simplex table

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub> of Max</b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution</b>	<b>Ratio</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>		
0	s <sub>1</sub>	1	1	1	0	0	2	
0	s <sub>2</sub>	5	2	0	1	0	10	
0	s <sub>3</sub>	3	8	0	0	1	12	
Z <sub>j</sub>		0	0	0	0	0		
Net Value C <sub>j</sub> – Z <sub>j</sub>		5	3	0	0	0		

$$Z_j = \sum_{i=1}^3 CB_i B_i$$

### Optimal Condition:

For Max: All C<sub>j</sub> – Z<sub>j</sub>  $\leq 0$  and For Min: All C<sub>j</sub> – Z<sub>j</sub>  $\geq 0$

But all C<sub>j</sub> – Z<sub>j</sub> is not  $\leq 0$ .

**Key Column:** Hence perform next step by choosing Maximum value of C<sub>j</sub> – Z<sub>j</sub> i.e. Key column (x<sub>1</sub>) 5 which is entering variable.

## Key Row:

- Select key row which is minimum value i.e. 2. But Rows  $s_1$  and  $s_2$  are same value 2.

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>	<b>Ratio I<sub>i</sub> / B<sub>i</sub></b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>		
0	s <sub>1</sub>	1	1	1	0	0	2	2/1 = 2
0	s <sub>2</sub>	5	2	0	1	0	10	10/5 = 2
0	s <sub>3</sub>	3	8	0	0	1	12	12/3 = 4
	Z <sub>j</sub>	0	0	0	0	0		
	Net Value C <sub>j</sub> – Z <sub>j</sub>	5	3	0	0	0		

- So choose arbitrarily any row as key row.
- Here Choose s<sub>1</sub> is leaving variable.

- ♦ Select key element (pivot) from intersection of key row and key column. Here key element is 1.

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>	<b>Ratio I<sub>i</sub> / B<sub>i</sub></b>
		x <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>		
0	s <sub>1</sub>	1	1	1	0	0	2	2/1 = 2
0	s <sub>2</sub>	5	2	0	1	0	10	10/5 = 2
0	s <sub>3</sub>	3	8	0	0	1	12	12/3 = 4
Z <sub>j</sub>		0	0	0	0	0		
Net Value C <sub>j</sub> – Z <sub>j</sub>		5	3	0	0	0		

## Iteration II

Key row elements divide by key element 1

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>
	Basic Variables (B <sub>i</sub> )	X <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
5	x <sub>1</sub> (Entering variable)	1/1 =1	1/1 =1	1/1=1	0/1 =0	0/1 =0	2/1=2
0	S <sub>2</sub>	5	2	0	1	0	10
0	S <sub>3</sub>	3	8	0	0	1	12
	Z <sub>j</sub>	0	0	0	0	0	
	Net Value C <sub>j</sub> – Z <sub>j</sub>	5	3	0	0	0	

## Other Rows calculation:

New value = Old value – [(corresponding key column value **X** corresponding key Row value) / key element]

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>
	Basic Variables (B <sub>i</sub> )	<b>X<sub>1</sub></b>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
5	x <sub>1</sub> (Entering variable)	1	1	1	0	0	2
0	S <sub>2</sub>	5	2	0	1	0	10
0	S <sub>3</sub>	3	8	0	0	1	12
	Z <sub>j</sub>	0	0	0	0	0	
	Net Value C <sub>j</sub> – Z <sub>j</sub>	5	3	0	0	0	

Row S<sub>2</sub>:

$$5 - [(5 \times 1) / 1] = 5 - 5 = 0;$$

$$0 - [(5 \times 1) / 1] = 0 - 5 = -5;$$

$$0 - [(5 \times 0) / 1] = 0 - 0 = 0;$$

$$2 - [(5 \times 1) / 1] = 2 - 5 = -3$$

$$1 - [(5 \times 0) / 1] = 1 - 0 = 1$$

$$10 - [(5 \times 2) / 1] = 10 - 10 = 0$$

Fill this value in table as in next slide.

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>
	Basic Variables (B <sub>i</sub> )	X <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
5	x <sub>1</sub> (Entering variable)	1	1	1	0	0	2
0	S <sub>2</sub>	0	-3	-5	1	0	0
0	S <sub>3</sub>	3	8	0	0	1	12
Z <sub>j</sub>		0	0	0	0	0	
Net Value C <sub>j</sub> - Z <sub>j</sub>		5	3	0	0	0	

**Row S<sub>3</sub>:**

$$3 - [(3 \times 1) / 1] = 3 - 3 = 0;$$

$$0 - [(3 \times 1) / 1] = 0 - 3 = -3;$$

$$1 - [(3 \times 0) / 1] = 1 - 0 = 1;$$

$$8 - [(3 \times 1) / 1] = 8 - 3 = 5$$

$$0 - [(3 \times 0) / 1] = 0 - 0 = 0$$

$$12 - [(3 \times 2) / 1] = 12 - 6 = 6$$

Fill this value in table as in next slide.

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>5</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Initial solution (I)</b>
	Basic Variables (B <sub>i</sub> )	X <sub>1</sub>	x <sub>2</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
5	x <sub>1</sub> (Entering variable)	1	1	1	0	0	2
0	S <sub>2</sub>	0	-3	-5	1	0	0
0	S <sub>3</sub>	0	5	-3	0	1	6
Z <sub>j</sub>		5	5	5	0	0	
Net Value C <sub>j</sub> - Z <sub>j</sub>		0	-2	0	0	0	

As per optimality Net Value C<sub>j</sub> - Z<sub>j</sub>  $\leq 0$ .

Hence Optimal solution is x<sub>1</sub> = 2. But x<sub>2</sub> is not included. So, x<sub>2</sub> = 0.

$$\text{Max } Z = 5(2) + 3(0) = 10$$

## Simplex Method for LP Problem 2

- The given problem with Max  $Z = 12x_1 + 16x_2$

Subject to  $10x_1 + 20x_2 \leq 120$

$$8x_1 + 8x_2 \leq 80, \text{ where } x_1, x_2 \geq 0$$

Solution:

$$\text{Max } Z = 12x_1 + 16x_2 + 0s_1 + 0s_2$$

Subject to  $10x_1 + 20x_2 + s_1 = 120$

$$8x_1 + 8x_2 + s_2 = 80 \quad \text{where } x_1, x_2, s_1, s_2 \geq 0$$

Optimal solution is  $x_1 = .$   $x_2 = .$

Max  $Z = 128$

## Dual Simplex Method

- ◆ **Optimal Condition:**
  - ◆ For Min: All  $C_j - Z_j \geq 0$  and
- ◆ **Feasible Solution condition**
  - ◆ solution  $\geq 0$
- ◆ Specialized form of simplex method in which optimality is maintained in all iterations.
- ◆ Initially, the solution may not feasible. But successive iterations will remove the infeasibility.
- ◆ If the problem is feasible in an iteration, then execution will be stopped. Because the calculated solution is feasible and optimal in that iteration.

## Dual Simplex Method for Minimization Problem

- The given problem with  $\text{Min } Z = x_1 + 2x_2 + 3x_3$

Subject to  $2x_1 - x_2 + x_3 \geq 4$

$$x_1 + x_2 + 2x_3 \leq 8$$

$$x_2 - x_3 \geq 2, \text{ where } x_1, x_2, x_3 \geq 0$$

Solution:

- $\text{Min } Z = x_1 + 2x_2 + 3x_3$

Subject to

Multiply the constraint which has  $\geq$  (min) to convert into  $\leq$  (max) inequality.

$$-2x_1 + x_2 - x_3 \leq -4$$

$$x_1 + x_2 + 2x_3 \leq 8$$

$$-x_2 + x_3 \leq -2, \text{ where } x_1, x_2, x_3 \geq 0$$

# Dual Simplex Method for Minimization Problem

- ♦ Add slack variables.

$$\text{Min} \quad Z = x_1 + 2x_2 + 3x_3 + 0s_1 + 0s_2 + 0s_3$$

Subject to

$$-2x_1 + x_2 - x_3 + s_1 = -4$$

$$x_1 + x_2 + 2x_3 + s_2 = 8$$

$$-x_2 + x_3 + s_3 = -2, \quad \text{where } x_1, x_2, x_3, s_1, s_2, s_3 \geq 0$$

- ♦ Initial basic feasible solution:

Assume  $x_1, x_2 = 0$ , then  $s_1 = -4$ ,  $s_2 = 8$ ,  $s_3 = -2$

- ♦ Iteration I: Formulate simplex table

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	1	2	3	0	0	0	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
(Row1) 0	<b>s<sub>1</sub></b>	-2	1	-1	1	0	0	<b>-4</b>
(Row2) 0	s <sub>2</sub>	1	1	2	0	1	0	8
(Row3) 0	s <sub>3</sub>	0	-1	1	0	0	1	-2
<b>Z<sub>j</sub></b>		0	0	0	0	0	0	
<b>Net Value C<sub>j</sub> – Z<sub>j</sub></b>		1	2	3	0	0	0	

## Optimal condition

For Min: All  $C_j - Z_j \geq 0$ , This solution is optimal.

## Feasible Solution condition

**solution  $\geq 0$ .** But it has two negative values -4, -2. So, it is not feasible solution.

Find out **Leaving variable** (key row) with most negative solution -4.

Determine Entering variable:

Variables	$x_1$	$x_2$	$x_3$	$s_1$	$s_2$	$s_3$
$-(C_j - Z_j)$	-1	-2	-3	0	0	0
$s_1$	-2	1	-1	1	0	0
Ratio $(-(C_j - Z_j) / s_1)$	1/2	-	3	-	-	-

- If  $s_1$  is 0 or any Negative values (i.e denominator), ignore that ratio.
- Choose minimum value as entering variable (key column)

- Key Row, Key column identified and key value is -2

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	1	2	3	0	0	0	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
0	s <sub>1</sub>	-2	1	-1	1	0	0	-4
0	s <sub>2</sub>	1	1	2	0	1	0	8
0	s <sub>3</sub>	0	-1	1	0	0	1	-2

## Iteration 2:

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	1	2	3	0	0	0	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
1 (R4)	x <sub>1</sub>	-2/-2 =1	1/ -2 = -1/2	-1/-2 = 1/2	1/-2 = -1/2	0/-2 = 0	0/-2 =0	-4/-2 = 2
0	s <sub>2</sub>	1	1	2	0	1	0	8
0	s <sub>3</sub>	0	-1	1	0	0	1	-2

- Compute the Row 5 and Row 6 using

**old value – (corresponding column value x corresponding new row (R4) value)**

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
1 (R4)	x <sub>1</sub>	1	-1/2	1/2	-1/2	0	0	2
0 (R5)	s <sub>2</sub>	1 - (1 * 1) = 0	1 - (1 * -1/2) = 3/2	3/2	1/2	1	0	6
0 (R6)	s <sub>3</sub>	0	-1	1	0	0	1	-2
Z <sub>j</sub>		1	-1/2	1/2	-1/2	0	0	
C <sub>j</sub> - Z <sub>j</sub>	0	5/2	5/2	1/2	0	0		

For Min: All C<sub>j</sub> - Z<sub>j</sub>  $\geq 0$ , This solution is optimal.

**Feasible Solution condition, solution  $\geq 0$ .** But it has negative value -2. So, it is not feasible solution. Again find leaving variable. **Here s<sub>3</sub> is leaving variable (min value).**

## Determine Entering variable:

Variables	$x_1$	$x_2$	$x_3$	$s_1$	$s_2$	$s_3$
$-(C_j - Z_j)$	0	-5/2	-5/2	-1/2	0	0
$s_3$	0	-1	1	0	0	1
Ratio $(-(C_j - Z_j) / s_3)$	-	5/2	-	-	-	-

- If  $s_1$  is 0 or any Negative values (i.e denominator), ignore that ratio.
- Choose minimum value as entering variable  $x_2$  (key column)

- Leaving variable  $s_3$  and Entering variable  $x_2$



<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	1	2	3	0	0	0	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
1 (R4)	x <sub>1</sub>	1	-1/2	1/2	-1/2	0	0	2
0 (R5)	s <sub>2</sub>	0	3/2	3/2	1/2	1/2	1	6
0 (R6)	s <sub>3</sub>	0	-1	1	0	0	1	-2
Z <sub>j</sub>		1	-1/2	1/2	-1/2	0	0	
C <sub>j</sub> - Z <sub>j</sub>	0	5/2	5/2	1/2	0	0	0	

## Iteration 3:

Row 8 = old value – (corresponding key column value x corresponding Row7 value)

<b>CB<sub>i</sub></b>	<b>C<sub>j</sub></b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>Solution</b>
	Basic Variables (B <sub>i</sub> )	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	
2 (R7)	x <sub>2</sub>	0	1	-1	0	0	-1	2
1 (R8)	x <sub>1</sub>	1	0	0	-1/2	0	-1/2	3
0 (R9)	s <sub>2</sub>	0	0	3	1/2	1	3/2	3
Z <sub>j</sub>		1	2	-2	-1/2	0	-5/2	7
C <sub>j</sub> – Z <sub>j</sub>		0	0	5	1/2	0	5/2	

All C<sub>j</sub> – Z<sub>j</sub>  $\geq 0$ , solution  $\geq 0$ . This solution is optimal and feasible.

Hence, x<sub>1</sub>=3, x<sub>2</sub>=2, x<sub>3</sub>=0 and Z=7.

$$Z(\text{op}) = x_1 + 2x_2 + 3x_3 = 3 + 2(2) + 3(0) = 7$$

## **Applications of LP:**

1. Production Management
2. Personnel Management
3. Inventory Management
4. Marketing Management
5. Finance management

## **Advantages of Linear Programming:**

- Utilized to analyze numerous economic, social, military and industrial problem.
- Linear programming is most suitable for solving complex problems.
- Helps in simplicity and productive management of an organization which gives better outcomes.
- Improves quality of decision: A better quality can be obtained with the system by making use of linear programming.
- Provides a way to unify results from disparate areas of mechanism design.

# Matrix Multiplication – Divide & Conquer

- Let A and B to be two  $n \times n$  matrices. Compute their product  $C=AB$ .

**Recursive version:**

- Assume  $n=2^k$ ,  $k \geq 0$ .
- Matrix size should be in power of 2.
- It should be square matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$T(n) = \begin{cases} m & \text{if } n < 2 \\ 8T(n/2) + 4(n/2)^2 a & \text{if } n \geq 2 \end{cases}$$

# Matrix Multiplication – Divide & Conquer

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$\begin{pmatrix} 5 & 3 \\ 2 & 6 \end{pmatrix} * \begin{pmatrix} 7 & 4 \\ 8 & 5 \end{pmatrix} = \begin{pmatrix} 5*7+3*8 & 5*4+3*5 \\ 2*7+6*8 & 2*4+6*5 \end{pmatrix}$$

$$\begin{pmatrix} 35+24 & 20+15 \\ 14+48 & 8+30 \end{pmatrix} = \begin{pmatrix} 59 & 35 \\ 62 & 38 \end{pmatrix}$$

## Strassen's matrix multiplication:

- Discover a way to compute the  $C_{ij}$ 's using 7 multiplications and 18 additions or subtractions

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$P = 11 * 12 \rightarrow 132$$

$$Q = 8 * 7 \rightarrow 56$$

$$R = 5(4-5) \rightarrow -5$$

$$S = 6(8-7) \rightarrow 6$$

$$T = (8)5 \rightarrow 40$$

$$U = (-3)(11) \rightarrow -33$$

$$V = (3-6)(8+5) \rightarrow (-3)(13) \rightarrow -39$$

$$C_{11} = 132 + 6 - 40 + (-39) \rightarrow 59$$

$$C_{12} = -5 + 40 \rightarrow 35$$

$$C_{21} = 56 + 6 \rightarrow 62$$

$$C_{22} = 132 + (-5) - 56 + (-33) \rightarrow 38$$

$$\left| \begin{array}{cccc|c} 1 & 2 & 3 & & 4 \\ 1 & 2 & 3 & & 4 \\ \hline 2 & 3 & 4 & 5 & \\ 2 & 3 & 5 & 1 & \end{array} \right|$$

## **Strassen's matrix multiplication:**

- Discover a way to compute the  $C_{ij}$ 's using **7 multiplications** and 18 additions or subtractions

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

## **Strassen's matrix multiplication Algorithm**

**Strassenmul( $n, A, B, C$ )** //  $n$  is size,  $A, B$  the input matrices,  $C$  output matrix

{

if  $n = 2$ ,

$$\begin{cases} C_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}; \\ C_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22}; \\ C_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21}; \\ C_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22}; \end{cases}$$

else

Partition  $A$  into 4 submatrices:  $A_{11}, A_{12}, A_{21}, A_{22}$ ;

Partition  $B$  into 4 submatrices:  $B_{11}, B_{12}, B_{21}, B_{22}$ ;

**Strassenmul** ( $\frac{n}{2}, A_{11} + A_{22}, B_{11} + B_{22}, P$ ) ;

**Strassenmul** ( $\frac{n}{2}, A_{21} + A_{22}, B_{11}, Q$ );

**Strassenmul** (  $\frac{n}{2}, A_{11}, B_{12} - B_{22}, R$  ) ;

**Strassenmul** (  $\frac{n}{2}, A_{22}, B_{21} - B_{11}, S$  ) ;

**Strassenmul** (  $\frac{n}{2}, A_{11} + A_{12}, B_{22}, T$  ) ;

**Strassenmul** (  $\frac{n}{2}, A_{21} - A_{11}, B_{11} + B_{12}, U$  ) ;

**Strassenmul** (  $\frac{n}{2}, A_{12} - A_{22}, B_{21} + B_{22}, V$  ) ;

$$C_{11} = P + S - T + V;$$

$$C_{12} = R + T;$$

$$C_{21} = Q + S;$$

$$C_{22} = P + R - Q + U;$$

}

## *Strassen's matrix multiplication Algorithm*

### Output Matrix:

$$C = \begin{pmatrix} P + S - T + V & R + T \\ Q + S & P + R - Q + U \end{pmatrix}$$

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

### Applications:

- Faster than the standard matrix multiplication algorithm and is useful in practice for large matrices,
- But would be slower than the fastest known algorithms for extremely large matrices.

## Time Complexity Analysis:

$$T(n) = \begin{cases} 7 \cdot T\left(\frac{n}{2}\right) + an^2 & n > 2 \\ b & n \leq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + an^2 \\ &= 7^2 \cdot T\left(\frac{n}{2^2}\right) + \left(\frac{7}{4}\right)an^2 + an^2 \\ &= 7^3 \cdot T\left(\frac{n}{2^3}\right) + \left(\frac{7}{4}\right)^2 \cdot an^2 + \left(\frac{7}{4}\right) \cdot an^2 + an^2 \\ &\quad \dots \\ T(n) &= \Theta(n^{\log 7}) = \Theta(n^{2.81}) \end{aligned}$$



# A Sample Proof of NP-Completeness

The following is the proof that the problem VERTEX COVER is NP-complete. This particular proof was chosen because it reduces 3SAT to VERTEX COVER and involves the transformation of a boolean formula to something geometrical. This is similar to what will be done for the two art gallery proofs.

This proof closely follows the one in "Computers and Intractability" by Garey and Johnson. A similar proof can be found in "Introduction to the Theory of Computation" by Sisper as this NP-completeness proof is quite basic.

---

Let us begin by defining the problems. First,

## 3-SATISFIABILITY (3SAT)

Instance: Set  $U$  of variables, a collection  $C$  of clauses over  $U$  such that each clause  $c$  in  $C$  has size exactly 3.

Question: Is there a truth assignment for  $U$  satisfying  $C$ ?

Next,

## VERTEX COVER

Instance: An undirected graph  $G$  and an integer  $K$

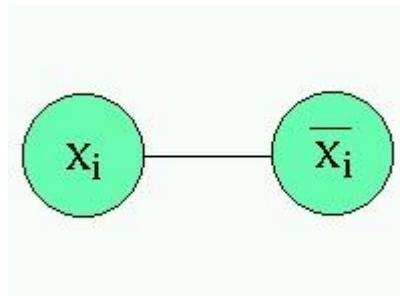
Question: Is there a vertex cover of size  $K$  or less for  $G$ , i.e., a subset  $V'$  of  $V$  with the size of  $V'$  less than  $K$  such that every edge has at least one endpoint in  $V'$ .

*Claim:* VERTEX COVER is NP-complete

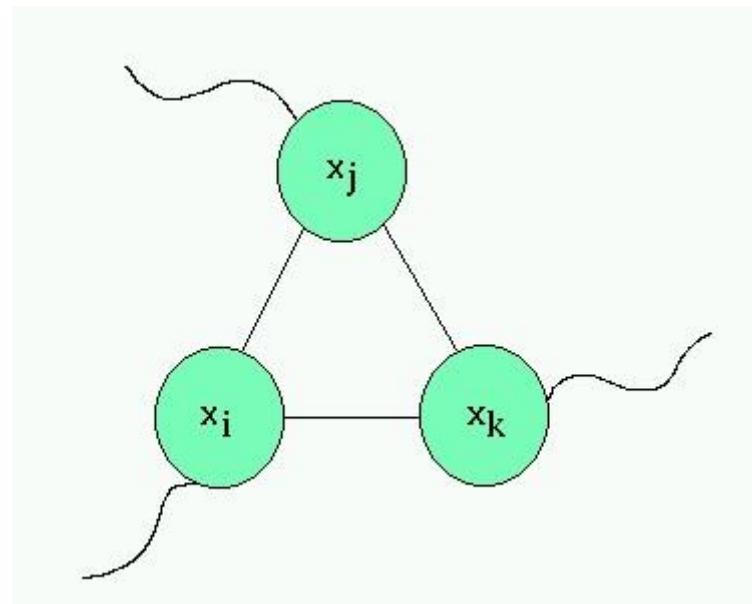
### Proof:

It was proved in 1971, by Cook, that 3SAT is NP-complete. Next, we know that VERTEX COVER is in NP because we could verify any solution in polytime with a simple  $n^2$  examination of all the edges for endpoint inclusion in the given vertex cover.

For the reduction, we are going to take an instance of 3SAT (a boolean formula) and reduce it to a vertex cover instance that has a cover if and only if the 3SAT formula has a satisfying assignment. The first thing we will do is force a choice for each variable to either True or False by having a pair of vertices for every literal and its negation. So an  $x_i$  in  $U$  will yield two vertices in  $G$ .



Next, we create a 'gadget' to represent the clauses. What we will do is for each clause  $(x_i, x_j, x_k)$  we'll create a three vertex triangle where each vertex is connected to the other two and to the corresponding literal from the section above.



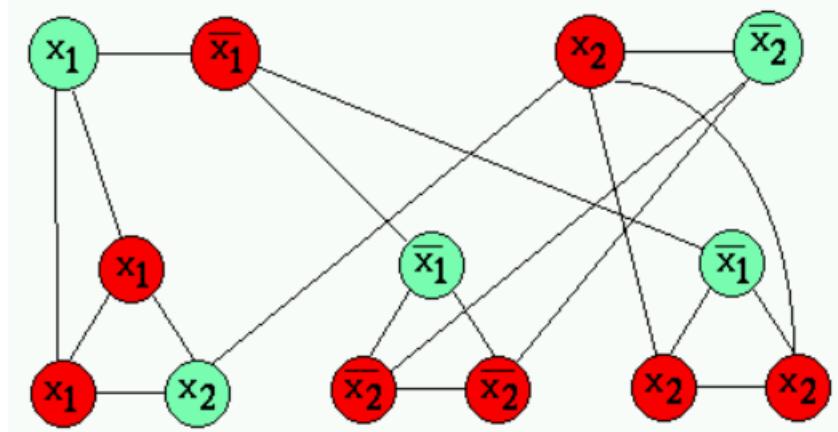
Finally, we must choose an appropriate  $K$  for the instance of VERTEX COVER. We'll choose  $l + 2m$  where  $l$  is the number of literals and  $m$  is the number of clauses.

So, if there is a satisfying assignment of the boolean 3SAT formula, then we can make the True literals part of our vertex cover in  $G$ . That's  $l$  nodes of our cover. Also, by choosing one node for every literal, we've covered the edge between them. Next, notice that since we've satisfied the 3SAT formula, we have to have chosen at least one of the literals in each clause to be True indicating that in each triangle gadget we have at least one node whose outgoing edge (the one going outside the gadget) is covered. Now, we need only two vertices (or less) from each triangle to cover the rest of the edges. This yields at most  $2m$  more vertices, for a maximum total of  $l + 2m$  vertices in our cover.

Next, assume there's a vertex cover of  $G$  with  $l + 2m$  vertices or less. We know at least  $l$  vertices are covering the literal pairs since the edge between them can't be covered in any other way. The other  $2m$  vertices (or less) must be covering the triangles since each triangle requires two vertices to be covered. We take the literal nodes in the vertex cover to be our assignment of the formula. This assignment has to satisfy the formula since every clause is satisfied by our gadget's construction.

Finally, this construction takes as input the  $m$  clauses and  $l$  literals and creates a graph with exactly  $2l + 3m$  nodes. This can easily be done in polynomial time. So it must be that VERTEX COVER is NP-complete.

Below is a sample construction of  $(x_1 \vee x_1 \vee x_2)(\neg x_1 \vee \neg x_2 \vee \neg x_2)(\neg x_1 \vee x_2 \vee x_2)$  and its cover in the constructed graph which will yield values of  $x_1=\text{FALSE}$  and  $x_2=\text{TRUE}$ :



## Set Cover

**SET COVER:** Given a set  $U$  of elements, a collection  $S_1, S_2, \dots, S_m$  of subsets of  $U$ , and an integer  $k$ , does there exist a collection of at most  $k$  of these sets whose union is equal of  $U$ ?

Sample application.

- $n$  available pieces of software.
- Set  $U$  of  $n$  capabilities that we would like our system to have.
- The  $i$ th piece of software provides the set  $S_i \subseteq U$  of capabilities.
- Goal: achieve all  $n$  capabilities using small number of pieces of software.

Ex.  $U = \{1, 2, 3, \dots, 12\}$ ,  $k = 3$ .

- $S_1 = \{1, 2, 3, 4, 5, 6\} \quad S_2 = \{5, 6, 8, 9\}$
- $S_3 = \{1, 4, 7, 10\} \quad S_4 = \{2, 5, 7, 8, 11\}$
- $S_5 = \{3, 6, 9, 12\} \quad S_6 = \{10, 11\}$

YES:  $S_3, S_4, S_5$ .

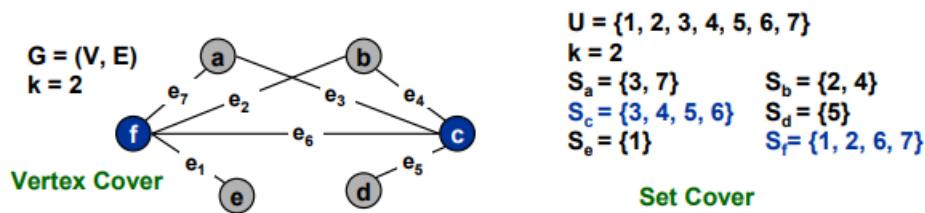
## Vertex Cover Reduces to Set Cover

**SET COVER:** Given a set  $U$  of elements, a collection  $S_1, S_2, \dots, S_n$  of subsets of  $U$ , and an integer  $k$ , does there exist a collection of at most  $k$  of these sets whose union is equal to  $U$ ?

**VERTEX COVER:** Given an undirected graph  $G = (V, E)$  and an integer  $k$ , is there a subset of vertices  $S \subseteq V$  such that  $|S| \leq k$ , and if  $(v, w) \in E$  then either  $v \in S$ ,  $w \in S$  or both.

**Claim.** VERTEX-COVER  $\leq_p$  SET-COVER.

**Proof.** Given black box that solves instances of SET-COVER.



## Vertex Cover Reduces to Set Cover

**SET COVER:** Given a set  $U$  of elements, a collection  $S_1, S_2, \dots, S_{n'}$  of subsets of  $U$ , and an integer  $k$ , does there exist a collection of at most  $k$  of these sets whose union is equal to  $U$ ?

**VERTEX COVER:** Given an undirected graph  $G = (V, E)$  and an integer  $k$ , is there a subset of vertices  $S \subseteq V$  such that  $|S| \leq k$ , and if  $(v, w) \in E$  then either  $v \in S$ ,  $w \in S$  or both.

**Claim.** VERTEX-COVER  $\leq_p$  SET-COVER.

**Proof.** Given black box that solves instances of SET-COVER.

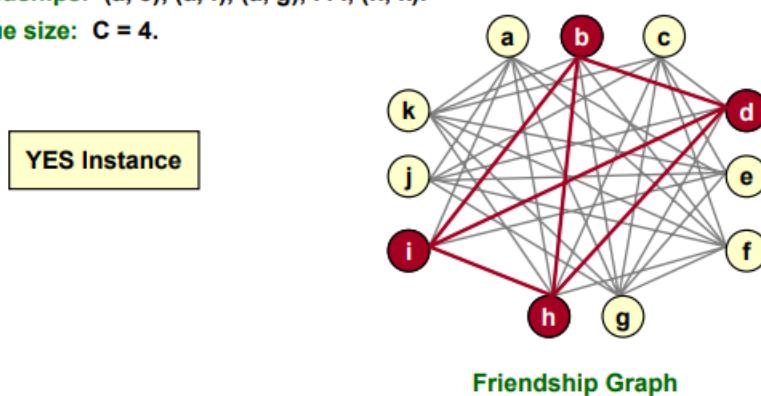
- Let  $G = (V, E)$ ,  $k$  be an instance of VERTEX-COVER.
- Create SET-COVER instance:
  - $k = k$ ,  $U = E$ ,  $S_v = \{e \in E : e \text{ incident to } v\}$
- Set-cover of size at most  $k$  if and only if vertex cover of size at most  $k$ .

## Clique

**CLIQUE:** Given N people and their pairwise relationships. Is there a group of C people such that every pair in the group knows each other.

Ex.

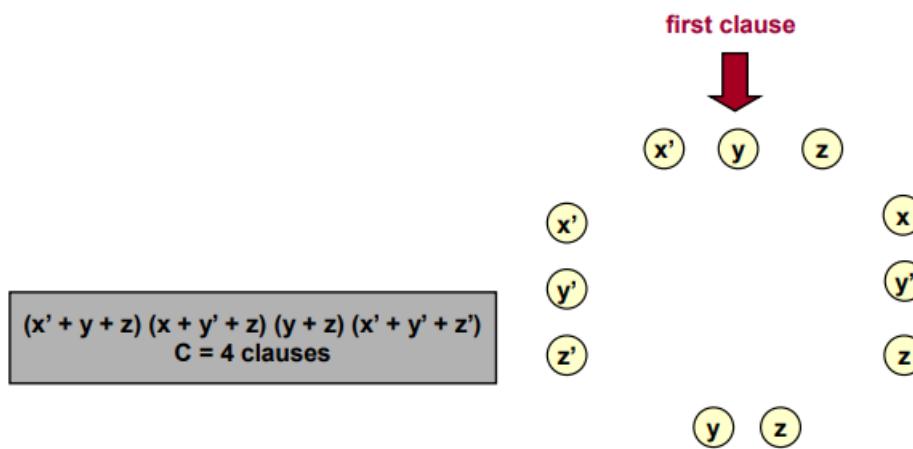
- People: a, b, c, d, e, . . . , k.
- Friendships: (a, e), (a, f), (a, g), . . . , (h, k).
- Clique size: C = 4.



## Satisfiability Reduces to Clique

**Claim.** CNF-SAT  $\leq_p$  CLIQUE.

- Given instance of CNF-SAT, create a person for each literal in each clause.

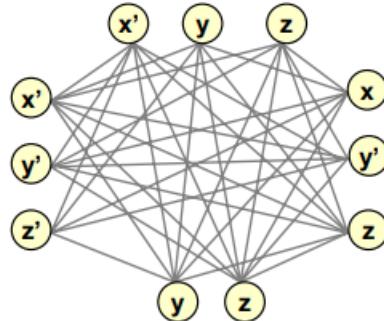


## Satisfiability Reduces to Clique

**Claim.** CNF-SAT  $\leq_p$  CLIQUE.

- Given instance of CNF-SAT, create a person for each literal in each clause.
- Two people know each other except if:
  - they come from the same clause
  - they represent a literal and its negation

$$(x' + y + z) (x + y' + z) (y + z) (x' + y' + z') \\ C = 4 \text{ clauses}$$

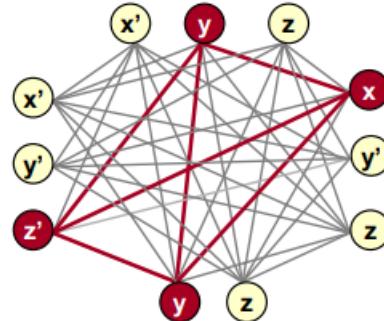


## Satisfiability Reduces to Clique

**Claim.** CNF-SAT  $\leq_p$  CLIQUE.

- Given instance of CNF-SAT, create a person for each literal in each clause.
- Two people know each other except if:
  - they come from the same clause
  - they represent a literal and its negation
- Clique of size C  $\Rightarrow$  satisfiable assignment.
  - set variable in clique to true
  - $(x, y, z) = (\text{true}, \text{true}, \text{false})$

$$(x' + y + z) (x + y' + z) (y + z) (x' + y' + z') \\ C = 4 \text{ clauses}$$

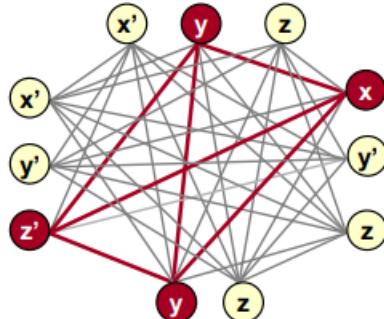


## Satisfiability Reduces to Clique

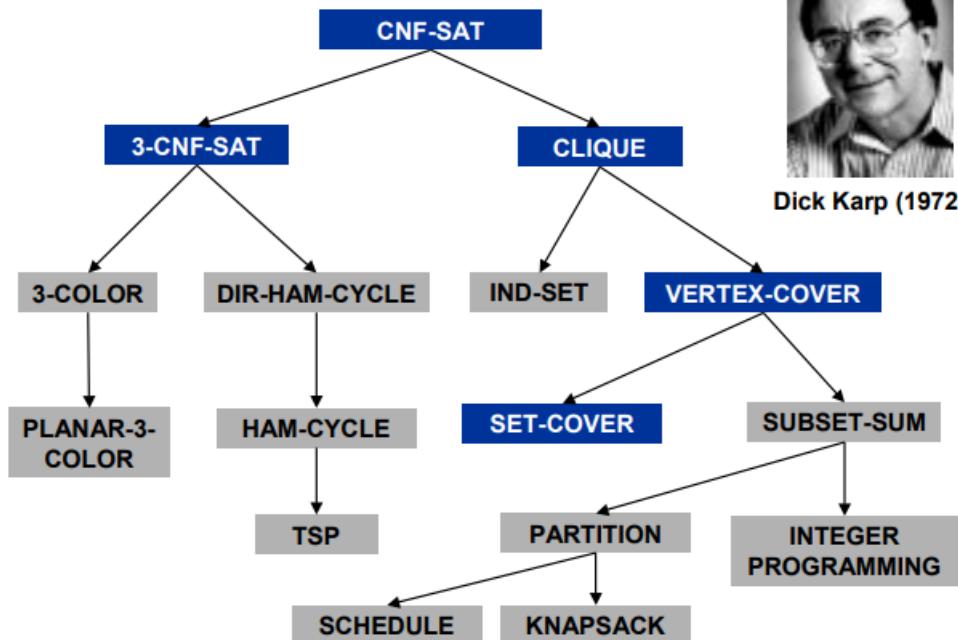
**Claim.** CNF-SAT  $\leq_p$  CLIQUE.

- Given instance of CNF-SAT, create a person for each literal in each clause.
- Two people know each other except if:
  - they come from the same clause
  - they represent a literal and its negation
- Clique of size C  $\Rightarrow$  satisfiable assignment.
- Satisfiable assignment  $\Rightarrow$  clique of size C.
  - $(x, y, z) = (\text{true}, \text{true}, \text{false})$
  - choose one true literal from each clause

$$(x' + y + z) (x + y' + z) (y + z) (x' + y' + z') \\ C = 4 \text{ clauses}$$



## Polynomial-Time Reductions



# Chapter 2

## NP Completeness II

By Sariel Har-Peled, December 17, 2012<sup>①</sup>

Version: 1.03

### 2.1 Max-Clique

We remind the reader, that a *clique* is a complete graph, where every pair of vertices are connected by an edge. The **MaxClique** problem asks what is the largest clique appearing as a subgraph of  $G$ . See Figure 2.1.

#### MaxClique

**Instance:** A graph  $G$

**Question:** What is the largest number of nodes in  $G$  forming a complete subgraph?

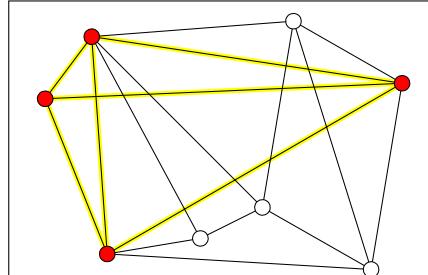


Figure 2.1: A clique of size 4 inside a graph with 8 vertices.

Note that **MaxClique** is an *optimization* problem, since the output of the algorithm is a number and not just true/false.

The first natural question, is how to solve **MaxClique**. A naive algorithm would work by enumerating all subsets  $S \subseteq V(G)$ , checking for each such subset  $S$  if it induces a clique in  $G$  (i.e., all pairs of vertices in  $S$  are connected by an edge of  $G$ ). If so, we know that  $G_S$  is a clique, where  $G_S$  denotes the *induced subgraph* on  $S$  defined by  $G$ ; that is, the graph formed by removing all the vertices are not in  $S$  from  $G$  (in particular, only edges that have both endpoints in  $S$  appear in  $G_S$ ). Finally, our algorithm would return the largest  $S$  encountered, such that  $G_S$  is a clique. The running time of this algorithm is  $O(2^n n^2)$  as can be easily verified.

<sup>①</sup>This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

**Suggestion 2.1.1.** When solving any algorithmic problem, always try first to find a simple (or even naive) solution. You can try optimizing it later, but even a naive solution might give you useful insight into a problem structure and behavior.

We will prove that **MaxClique** is **NP-HARD**. Before dwelling into that, the simple algorithm we devised for **MaxClique** shade some light on why intuitively it should be **NP-HARD**: It does not seem like there is any way of avoiding the brute force enumeration of all possible subsets of the vertices of  $G$ . Thus, a problem is **NP-HARD** or **NP-COMPLETE**, *intuitively*, if the only way we know how to solve the problem is to use naive brute force enumeration of all relevant possibilities.

**How to prove that a problem  $X$  is NP-Hard?** Proving that a given problem  $X$  is **NP-HARD** is usually done in two steps. First, we pick a known **NP-COMPLETE** problem  $A$ . Next, we show how to solve any instance of  $A$  in polynomial time, assuming that we are given a polynomial time algorithm that solves  $X$ .

Proving that a problem  $X$  is **NP-COMPLETE** requires the additional burden of showing that is in **NP**. Note, that only decision problems can be **NP-COMPLETE**, but optimization problems can be **NP-HARD**; namely, the set of **NP-HARD** problems is much bigger than the set of **NP-COMPLETE** problems.

**Theorem 2.1.2.** *MaxClique* is **NP-HARD**.

*Proof:* We show a reduction from **3SAT**. So, consider an input to **3SAT**, which is a formula  $F$  defined over  $n$  variables (and with  $m$  clauses).

We build a graph from the formula  $F$  by scanning it, as follows:

- (i) For every literal in the formula we generate a vertex, and label the vertex with the literal it corresponds to.

Note, that every clause corresponds to the three such vertices.

- (ii) We connect two vertices in the graph, if they are: (i) in different clauses, and (ii) they are *not* a negation of each other.

Let  $G$  denote the resulting graph. See Figure 2.2 for a concrete example. Note, that this reduction can be easily be done in quadratic time in the size of the given formula.

We claim that  $F$  is satisfiable iff there exists a clique of size  $m$  in  $G$ .

$\implies$  Let  $x_1, \dots, x_n$  be the variables appearing in  $F$ , and let  $v_1, \dots, v_n \in \{0, 1\}$  be the satisfying assignment for  $F$ . Namely, the formula  $F$  holds if we set  $x_i = v_i$ , for  $i = 1, \dots, n$ .

For every clause  $C$  in  $F$  there must be at least one literal that evaluates to **TRUE**. Pick a vertex that corresponds to such **TRUE** value from each clause. Let  $W$  be the

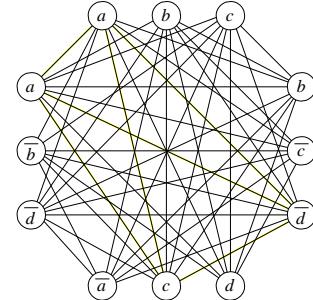


Figure 2.2: The generated graph for the formula  $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ .

resulting set of vertices. Clearly,  $W$  forms a clique in  $G$ . The set  $W$  is of size  $m$ , since there are  $m$  clauses and each one contribute one vertex to the clique.

$\Leftarrow$  Let  $U$  be the set of  $m$  vertices which form a clique in  $G$ .

We need to translate the clique  $G_U$  into a satisfying assignment of  $F$ .

- (i) set  $x_i \leftarrow \text{TRUE}$  if there is a vertex in  $U$  labeled with  $x_i$ .
- (ii) set  $x_i \leftarrow \text{FALSE}$  if there is a vertex in  $U$  labeled with  $\bar{x}_i$ .

This is a valid assignment as can be easily verified. Indeed, assume for the sake of contradiction, that there is a variable  $x_i$  such that there are two vertices  $u, v$  in  $U$  labeled with  $x_i$  and  $\bar{x}_i$ ; namely, we are trying to assign to contradictory values of  $x_i$ . But then,  $u$  and  $v$ , by construction will not be connected in  $G$ , and as such  $G_S$  is not a clique. A contradiction.

Furthermore, this is a satisfying assignment as there is at least one vertex of  $U$  in each clause. Implying, that there is a literal evaluating to **TRUE** in each clause. Namely,  $F$  evaluates to **TRUE**.

Thus, given a polytime (i.e., polynomial time) algorithm for **MaxClique**, we can solve **3SAT** in polytime. We conclude that **MaxClique** in **NP-HARD**. ■

**MaxClique** is an optimization problem, but it can be easily restated as a decision problem.

### Clique

**Instance:** A graph  $G$ , integer  $k$

**Question:** Is there a clique in  $G$  of size  $k$ ?

**Theorem 2.1.3.** *Clique* is **NP-COMPLETE**.

*Proof:* It is **NP-HARD** by the reduction of Theorem 2.1.2. Thus, we only need to show that it is in **NP**. This is quite easy. Indeed, given a graph  $G$  having  $n$  vertices, a parameter  $k$ , and a set  $W$  of  $k$  vertices, verifying that every pair of vertices in  $W$  form an edge in  $G$  takes  $O(u + k^2)$ , where  $u$  is the size of the input (i.e., number of edges + number of vertices). Namely, verifying a positive answer to an instance of **Clique** can be done in polynomial time.

Thus, **Clique** is **NP-COMPLETE**. ■

## 2.2 Independent Set

**Definition 2.2.1.** A set  $S$  of nodes in a graph  $G = (V, E)$  is an **independent set**, if no pair of vertices in  $S$  are connected by an edge.

### Independent Set

**Instance:** A graph  $G$ , integer  $k$

**Question:** Is there an independent set in  $G$  of size  $k$ ?