CS303 - Algorithms and Data Structures

Lecture 15

- Review

Professor : Mahmut Unan – UAB CS

Exam 2

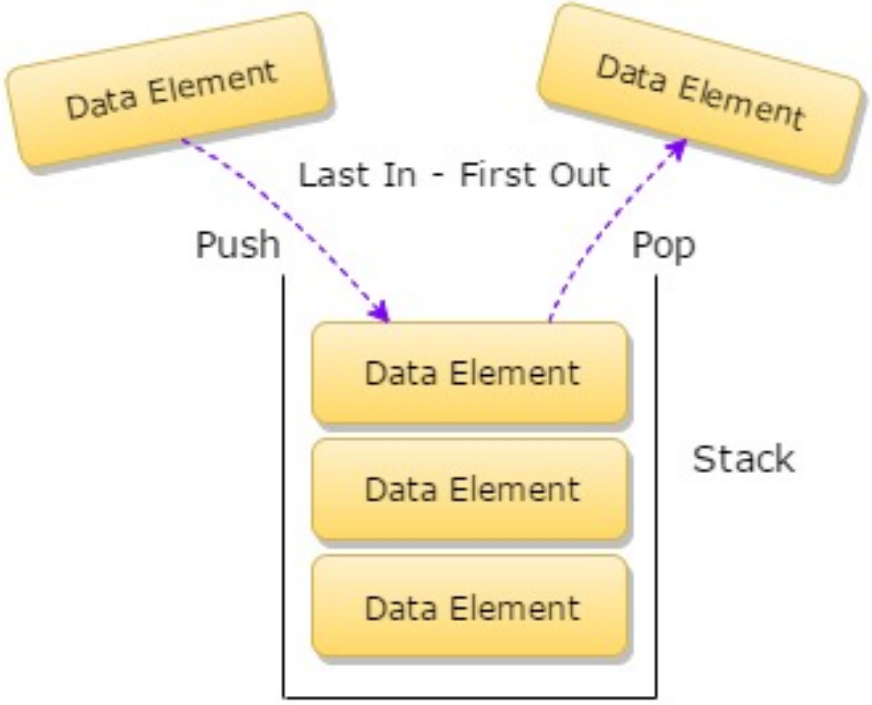
- **03/09/2023 Thursday**
  - **Regular Lecture time/place**
  - **Time Complexity**
  - **Data Structures**
  - **Searching**
  - **Graphs**
  - **Sorting Algorithms -- excluded**

Agenda

- Time Complexity
- Elementary Data Structures
- Binary Trees, Binary Search Trees
- Red Black Trees
- AVL Trees
- Tree Sort
- Elementary Graph Algorithms

Stacks and Queues

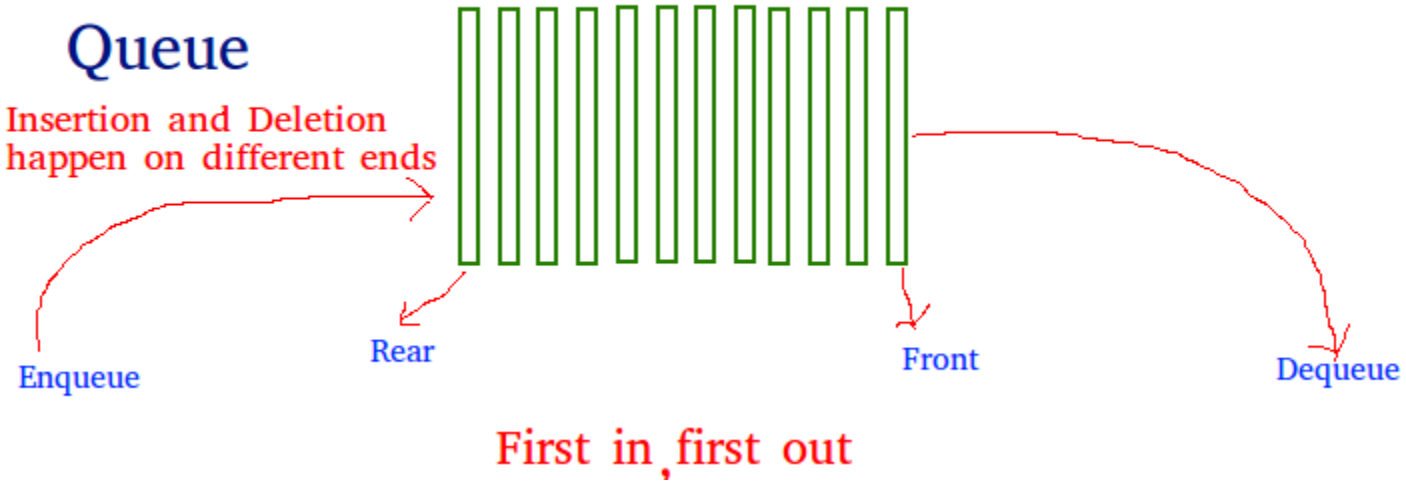
- Dynamic sets in which the element removed from the set by the DELETE operation is prespecified
- Stack
  - element deleted is the one most recently inserted
  - implements a **last-in, first-out (LIFO)** policy
  - INSERT operation is often called PUSH
  - DELETE operation is often called POP



https://www.w3schools.in/data-structures-tutorial/stack/

Stacks and Queues

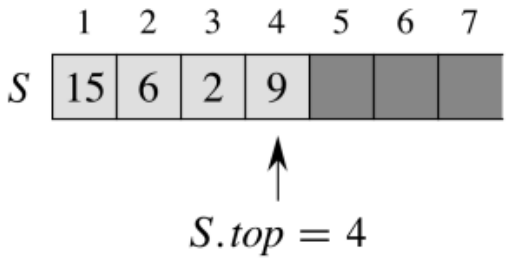
- Queue
  - element deleted is the one that has been in the set for the longest time
  - implements a first-in, first-out (FIFO) policy
  - INSERT operation is often called ENQUEUE
  - DELETE operation is often called DEQUEUE
- Deque – double-ended queue
  - a queue that allows insertion and deletion at both ends



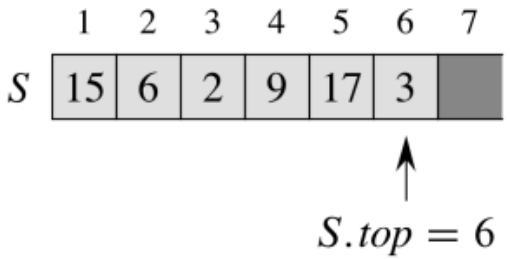
https://www.geeksforgeeks.org/queue-data-structure/

Implementing a stack using arrays

S[1..n] - array

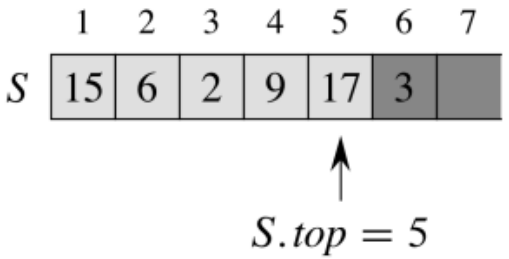


(a)



(b)

After
PUSH(S, 17)
PUSH(S, 3)



(c)

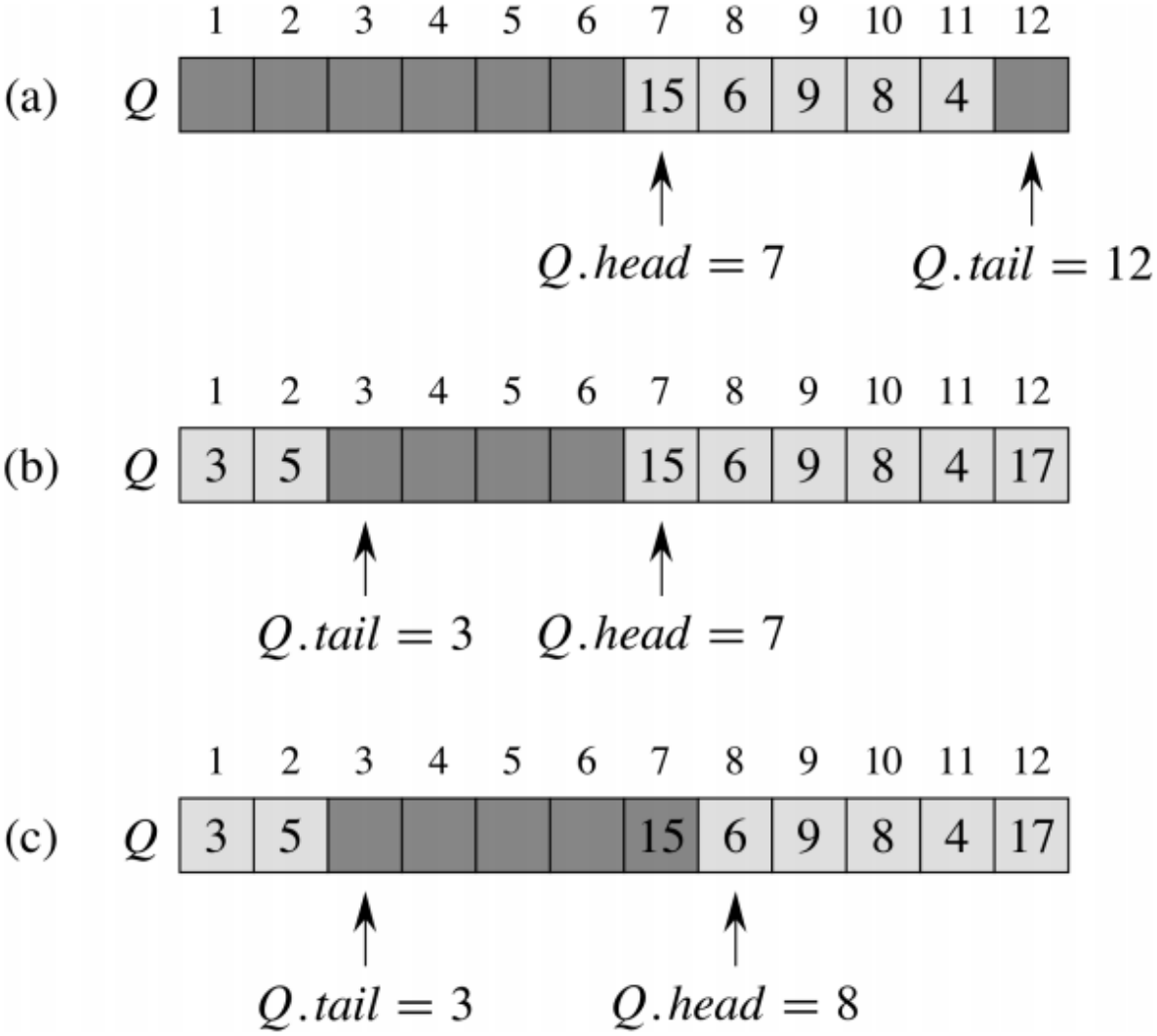
After
POP(S)

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  S.top = S.top + 1        // overflow not shown
2  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Implementing a queue using arrays



After
ENQUEUE(Q, 17)
ENQUEUE(Q, 3)

ENQUEUE(Q, 5)

After
DEQUEUE(Q)

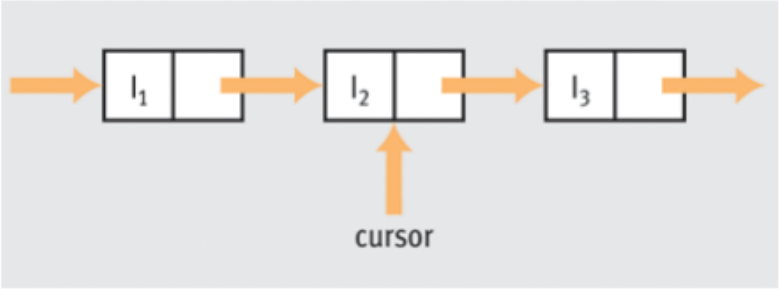
// overflow and underflow not shown

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Linked Lists

- Linked Lists provide a simple, flexible representation for dynamic sets
  - the order in a linked list is determined by a pointer in each object
  - retrieval, insertion, deletion allowed anywhere within the structure
- **List** – ordered collection of zero or more **nodes**
- Nodes contain two **fields**
  - **Information** field (**data** field)
  - **Pointer** field (**next** field)



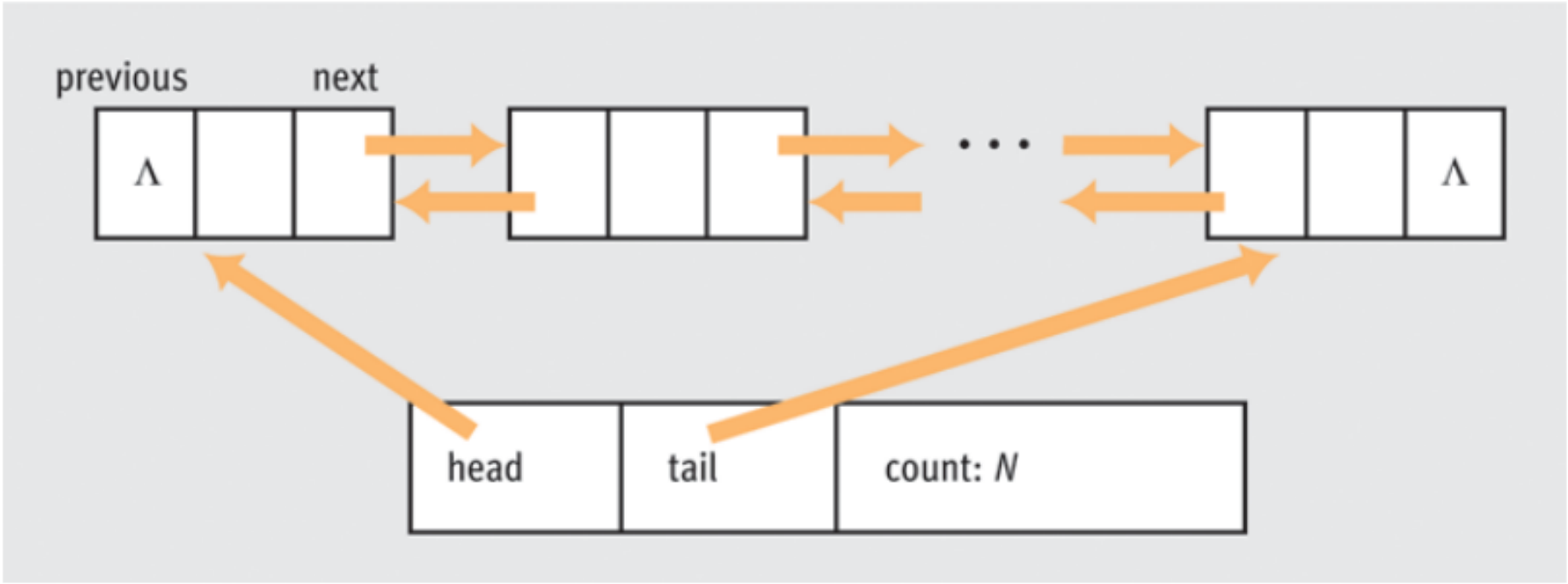
Implementations of Lists

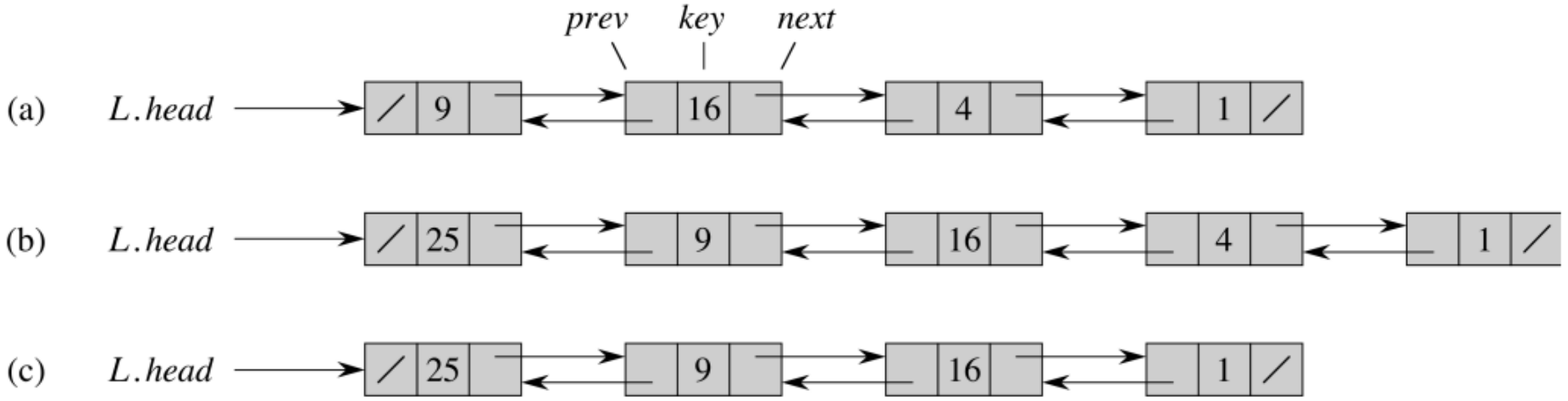
- Insert a node at the cursor:
  - Allocating space for the node
  - Assign the next field to the successor of the cursor
  - Assign the node to the next field of the cursor
  - Update total node count
- Adding a node is **O(1)** time complexity
- Many operations are **O(1)** because cursor points to the location, not using physical adjacency

Doubly Linked Lists

- **Singly linked list** – each node has a pointer to its successor
- **Doubly linked list** – nodes have a pointer to successor and predecessor
  - head and last nodes, cursor, and count
- Time complexity to search an element **O(n)**
  - Doubly linked list, list insertion/deletion time complexity **O(1)**

Doubly linked list data structure



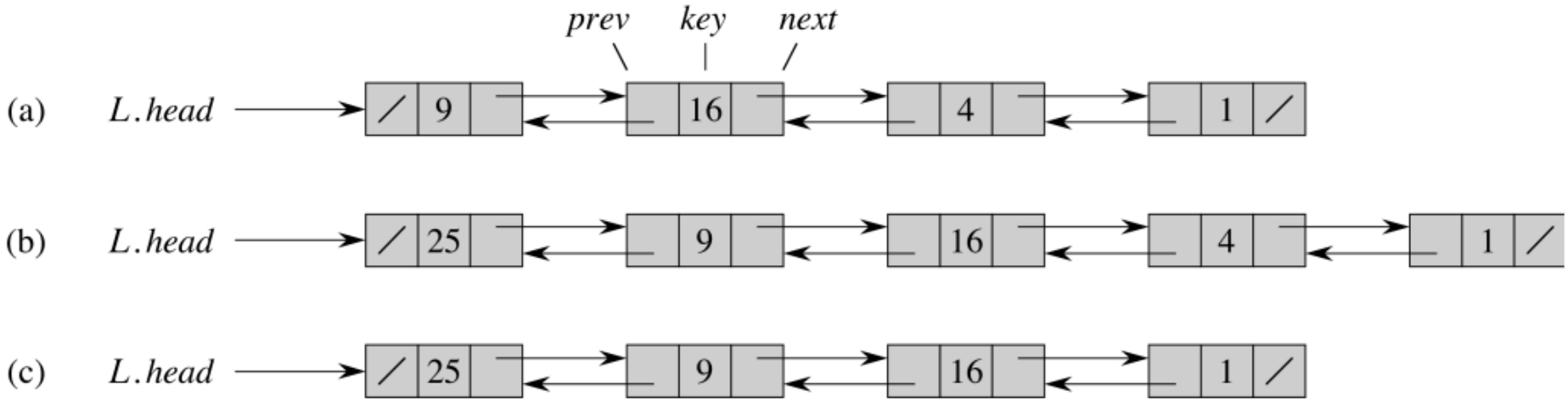


(b) After LIST_INSERT(L, 25)

(c) After LIST_DELETE(L, 4)

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```



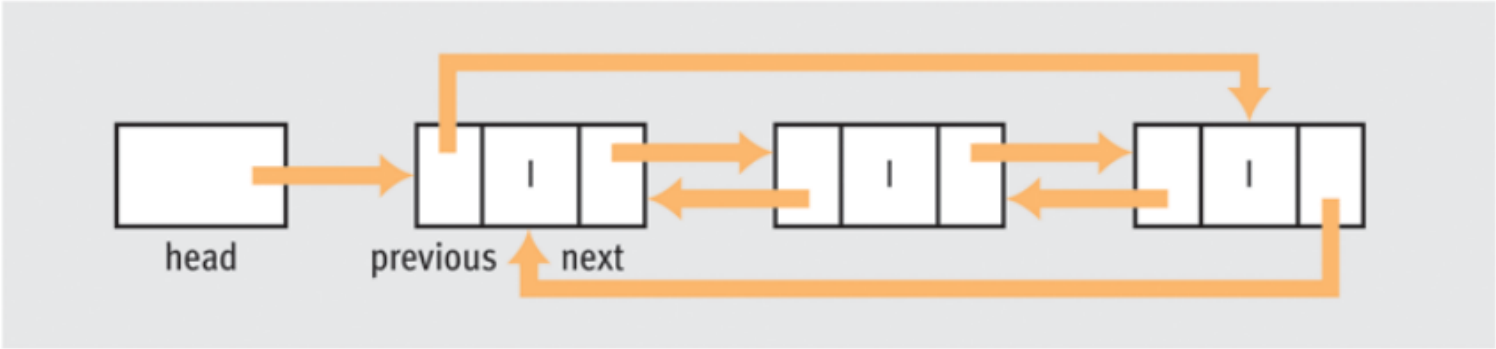
LIST-SEARCH(L, k)

1 $x = L.head$
2 **while** $x \neq$ NIL and $x.key \neq k$
3 $\quad x = x.next$
4 **return** x

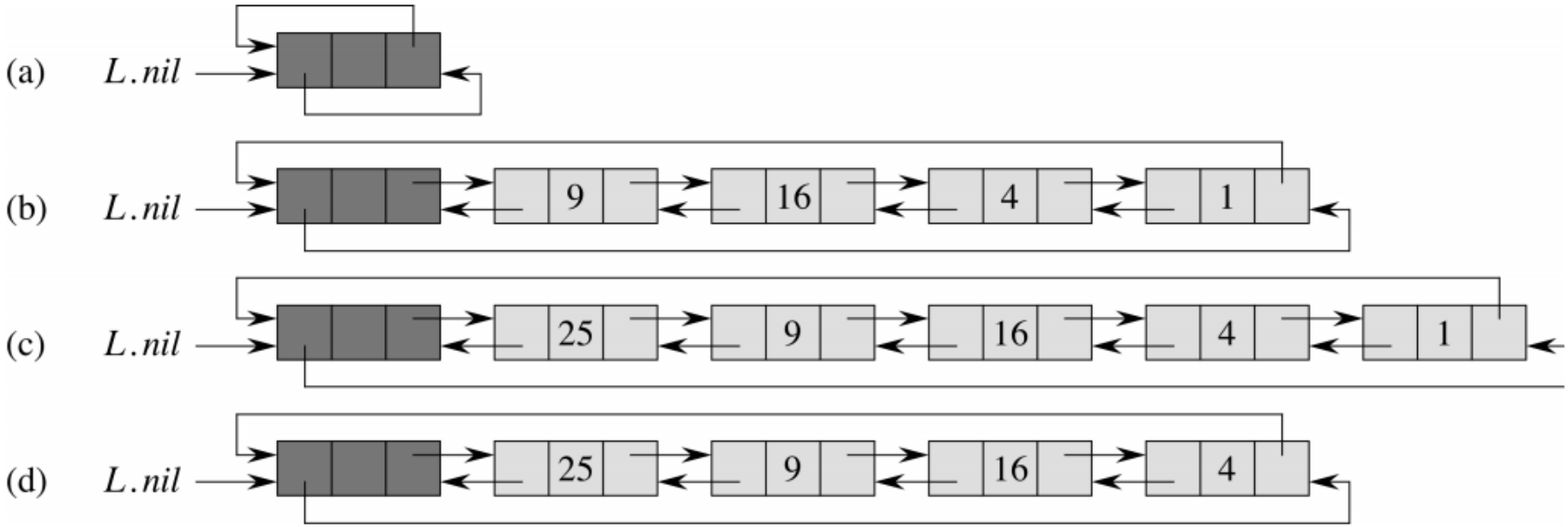
Circular Lists

- **Singly linked circular list (ring)**
  - last node next field points to the head of the list
  - No special case at ends of list
  - next() operation on last node returns first node
  - previous() operation on first node returns last node
- **Doubly linked circular list**
  - Last node successor points to the head node
  - Head node predecessor points to last node

Circular, doubly linked list



Circular, doubly linked list with a sentinel



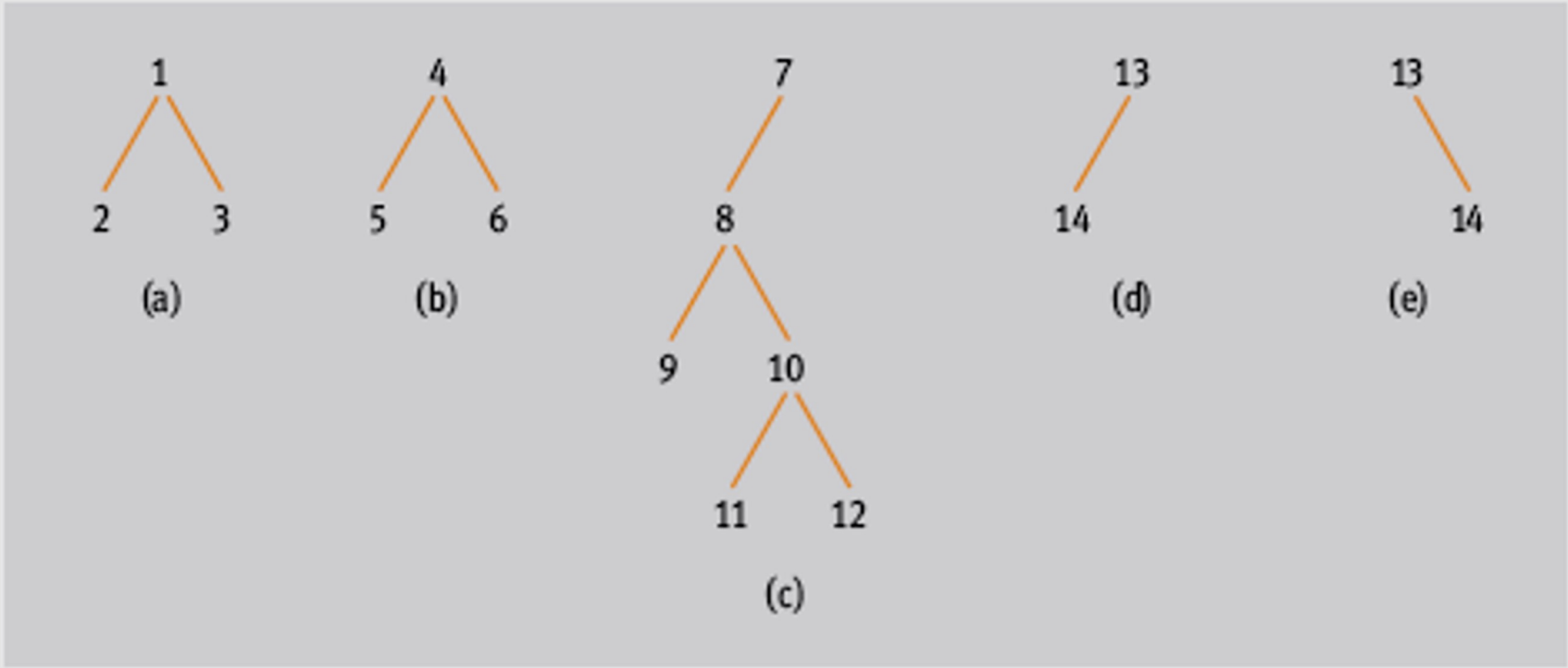
Time Complexity

Operation	doubly linked list	doubly linked circular list
a. SEARCH	O(n)	O(n)
a. INSERT	O(1)	O(1)
a. DELETE	O(1)	O(1)

Binary Trees

- Binary trees are distinct from general trees
  - Binary trees can be empty
  - Degree no greater than two
  - **Ordered trees** – every node is explicitly identifies as being either the left child or the right child of its parent
- **Binary tree**
  - Finite set of nodes, possibly empty
  - Consists of a root and two disjoint binary trees
    - Called left and right subtrees

Binary Trees (continued)



[FIGURE 7-6] Examples of binary trees

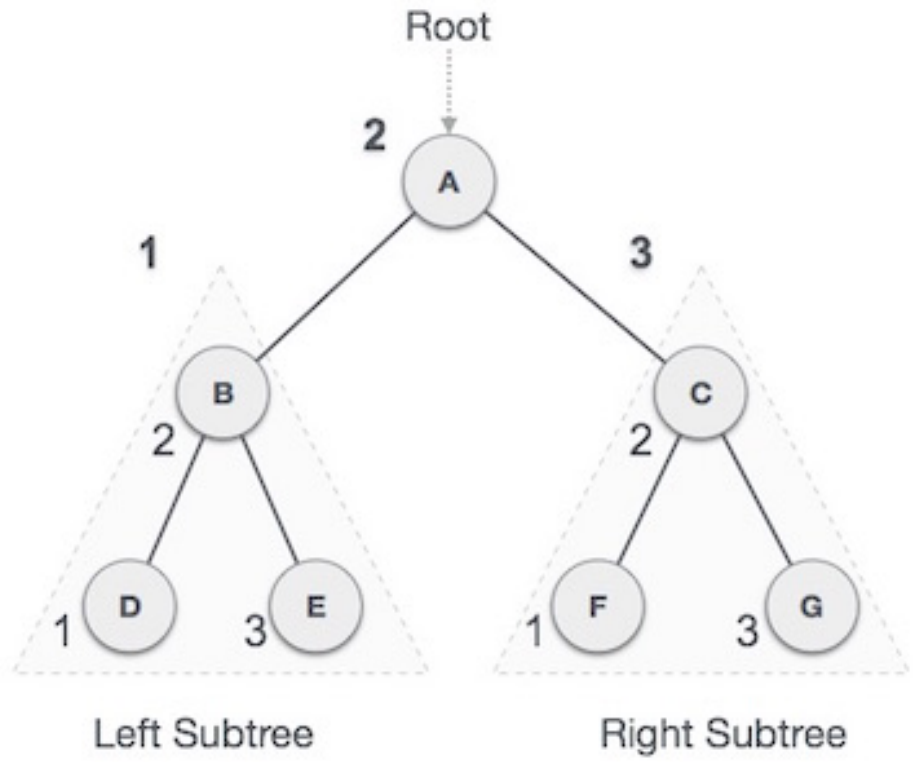
- Binary Tree
- Strict Binary Tree
- Complete Binary Tree
- Perfect Binary Tree

Operations on Binary Trees

- **Tree traversal** – start at root and visit every node exactly once
  - *Visit* – perform an operation on the data in a node
- **Preorder traversal** – visit the root, the left subtree, and the right subtree
- **Postorder traversal** – visit the left subtree, the right subtree, and the root
- **Inorder traversal** – visit the left subtree, the root, and the right subtree
- Complexity to traverse the tree:
  - $T(n) = c + 2T(n/2) = O(n)$

In-order Traversal

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



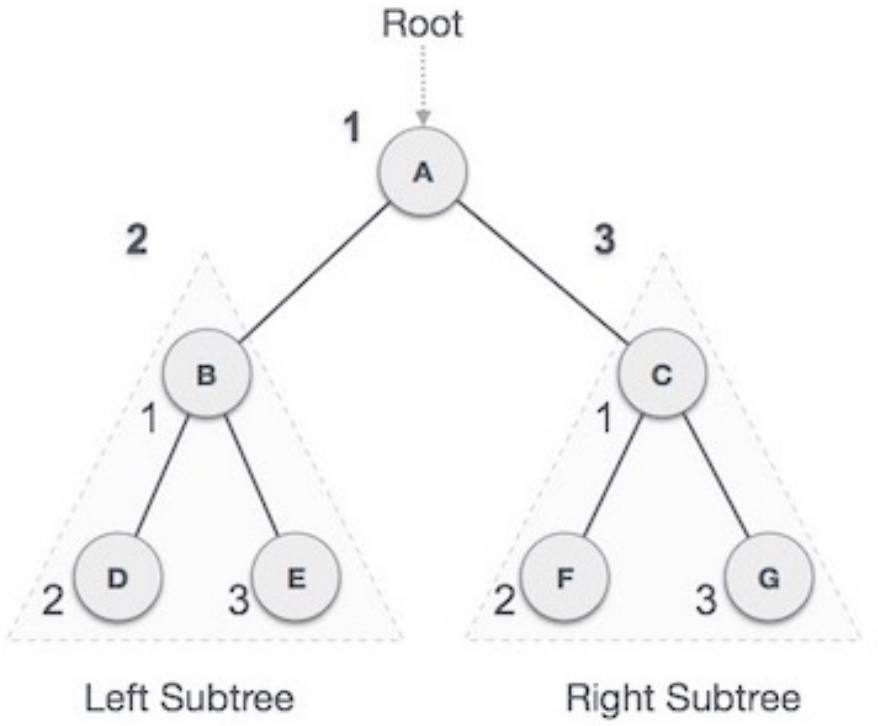
We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



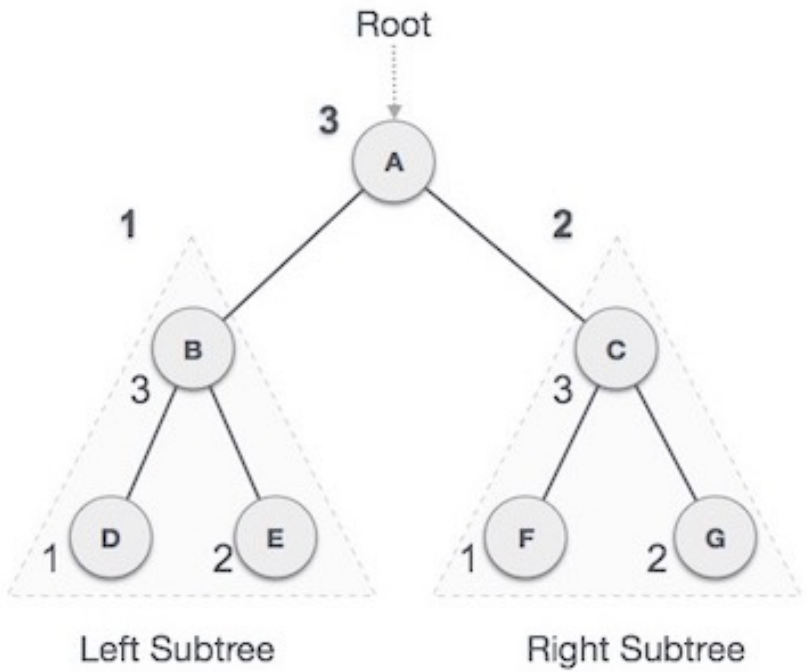
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm

Operations on Binary Trees (continued)

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

Operations on Binary Trees (continued)

- Natural correspondence between preorder, postorder, inorder traversals of an expression and the prefix, postfix, and infix representation of arithmetic expressions
  - **Prefix representation** – operator written before operands
  - **Postfix representation** – operator written after operand
  - **Infix representation** – operator written between operands

The Reference-Based Implementation of Binary Trees

- Two reference fields that point to the left and right subtrees
  - Locate the parent of a node by traversing the entire tree from the root
- If moving upward necessary, could add a pointer to the parent
- State information to maintain
  - Pointer to root of tree
  - Current position
  - Height, number of nodes, etc.

An Array-Based Implementation of Binary Trees

- One-dimensional array holds the information field of the node
- Two-dimensional array
  - First column holds the root of the left subtree
  - Second column holds the root of the right subtree
- Variable `root` is the index of the root of the tree
- Nodes that have no children store a value of -1 in the appropriate column in the children array

An Array-Based Implementation of Binary Trees (continued)

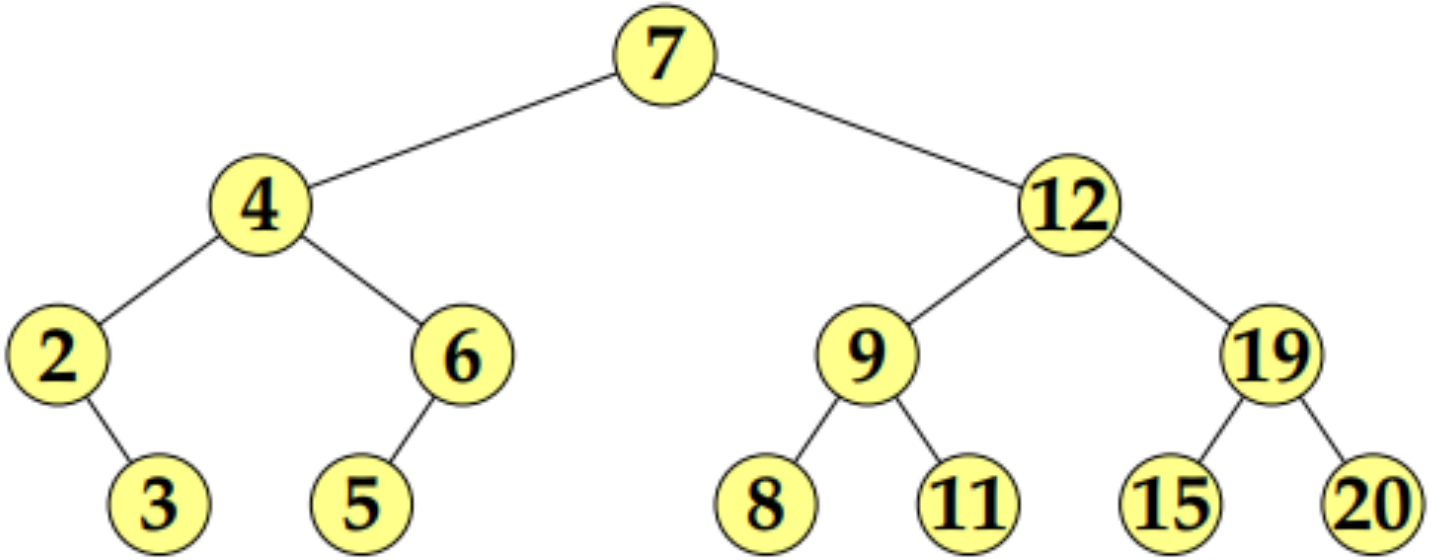
root = 3

	data	children (left)	children (right)
0	60	–1	–1
1	20	6	2
2	50	–1	–1
3	10	1	5
4	—	—	—
5	30	–1	0
6	40	–1	–1
7	—	—	—
8	—	—	—
9	—	—	—

[FIGURE 7-14] Implementation of a binary tree using arrays

Binary Search Trees

- **Binary search tree property** – an ordering on the nodes such that:
  - One data value in the information field of a node is the **key**
  - Every key value in the left subtree of a node is less than the key value in the node
  - Every key value in the right subtree of a node is greater than the key value in the node
- Usually assumed keys are unique


7
4
12
2
6
9
19
3
5
8
11
15
20

Using Binary Search Trees in Searching Operations

- `search(x, k)` method faster because it exploits binary tree property

```
TREE-SEARCH(x, k)
  if x == NIL or k == x.key
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH( x, k)
  while x ≠ NIL and k ≠ x.key
      if k < x.key
          x = x.left
      else
          x = x.right
  return x
```

Using Binary Search Trees in Searching Operations (continued)

- Θ(n) to locate a node if we do not know the location of a value
- Locate a given key, examine *H* nodes
  - *H* is the height of the tree
- **Degenerate binary search tree** – every nonterminal node has exactly one child
  - $H = n$, time to locate a node is $\Theta(n)$

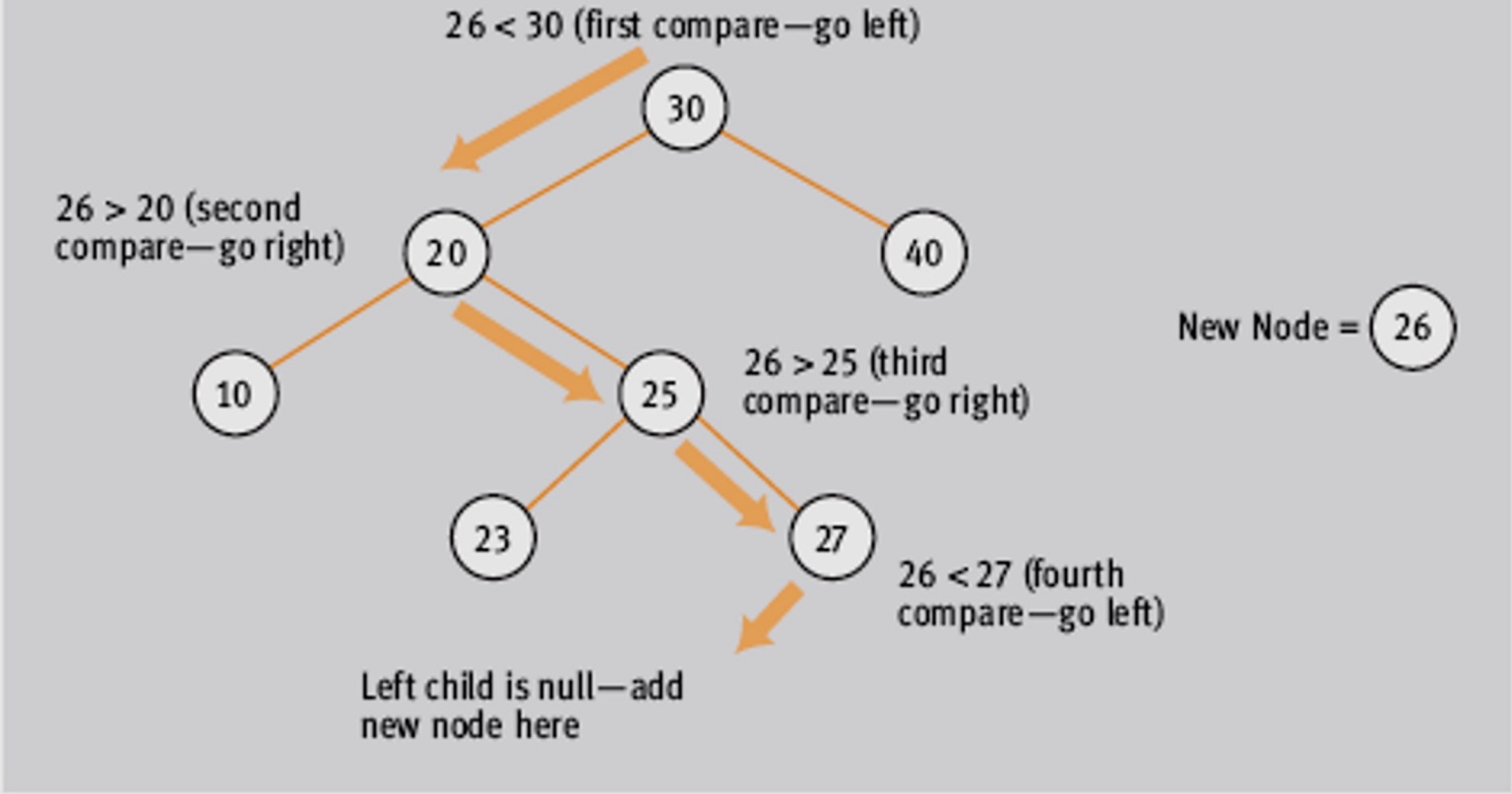
Using Binary Search Trees in Searching Operations (continued)

- **Full binary search tree** – all nonleaf nodes have degree = 2
  - $H = \Theta(\log n)$
- **Minimum height binary search tree** – least possible height for a binary tree
- Binary search tree constructed from a random sequence of values is roughly balanced
  - Height satisfies $H = \Theta(\log n)$
- Java Collection Framework uses a red-black tree

Insertion in Binary Search Trees

- Node insertions must maintain binary search tree property
- `insert(S, z)` method compares the value *v (=z.key)* to be inserted to the value *n* in the current node
  - If *v* < *n* follow the left branch
  - If *v* > *n* follow the right branch
  - Continue until *v* = *n* or Child pointer = **null**

Insertion in Binary Search Trees



[FIGURE 7-18] Insertion of a new node in a binary search tree

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
        y = x
        if z.key < x.key
              x = x.left
        else x = x.right
  z.p = y
  if y == NIL
        T.root = z          // tree T was empty
  elseif z.key < y.key
        y.left = z
  else y.right = z
```

Other Operations on BST

- minimum(x), x is any given node
- maximum(x), x is any given node
- successor(x), x is any given node
- predecessor(x), x is any given node
- delete(T, z), T is the tree and z is a node
- except delete, all other operations have trivial implementation

```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```

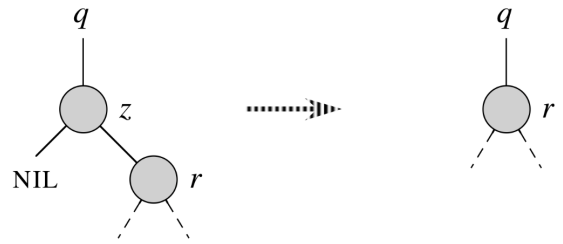
```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

Deleting a node from a BST

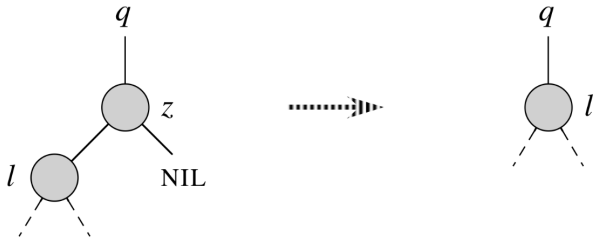
1. **If z has no children, then simply remove it by modifying the parent to z with NIL as its child**
2. **If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child**
3. **If z has two children, then find z's successor y, and**
   a. **if y is z's right child, then replace z by y, leaving y's right child alone**
   b. **otherwise, replace y with its own right child, and then replace z by y**

Case 2



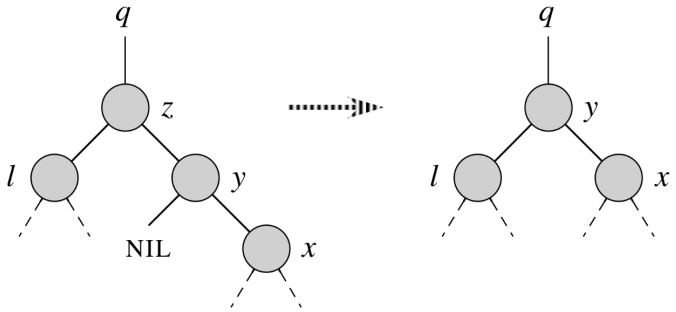
node z has no left child

Case 2



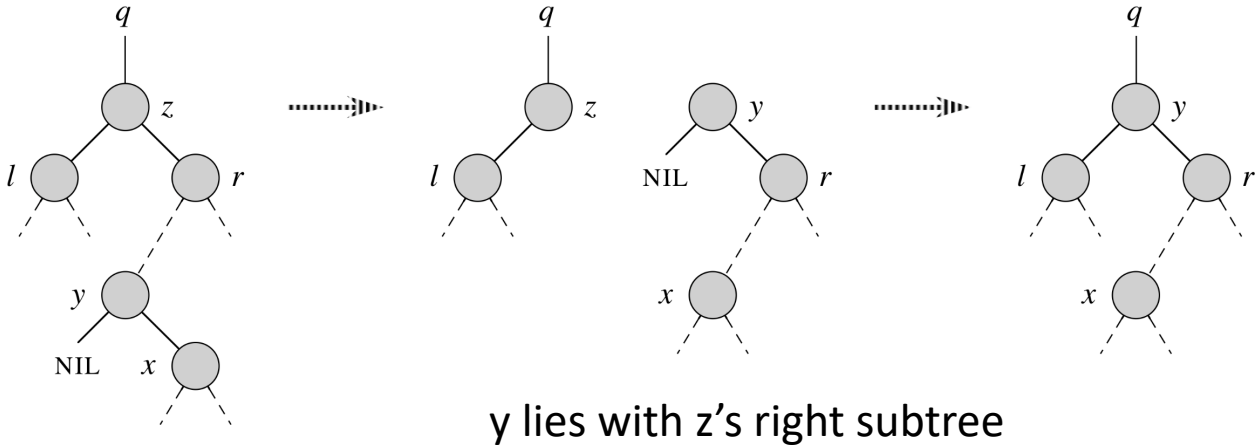
node z has no right child

Case 3a



y (z's successor) is z's right child

Case 3b

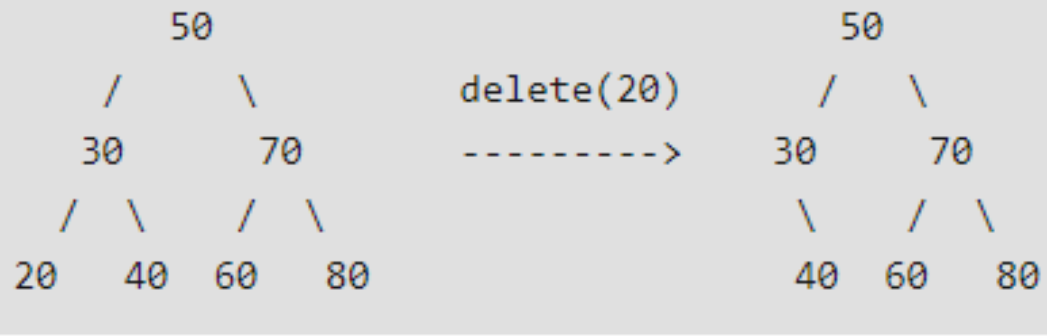


y lies with z's right subtree

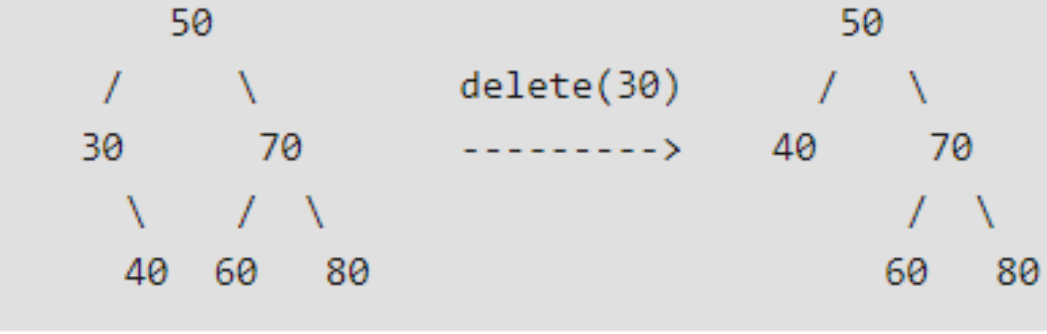
```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)            // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)             // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)            // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

1) ***Node to be deleted is leaf:*** Simply remove from the tree.



2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Exercises

- Draw the BST that results when you insert the following keys in exactly the order given below into an initially empty tree:
  a) 5, 8, 2, 3, 10, 12, 1, 15, 6, 9
  b) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  c) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
  d) 8, 23,17,1,92,13,44,21,19

- For the BST in case (a) above, draw the resulting BST when the root node is removed successively until there are no nodes in the tree

Time Complexity in Binary Search Trees

Assume 10,9,8,7,6,5 is a bst

- **Searching:** Searching in binary tree has worst case complexity of O(n). (searching for 5)

- **Insertion:** Insertion in binary tree has worst case complexity of O(n). (inserting 3)

- **Deletion:** Deletion in binary tree has worst case complexity of O(n). (delete 5)

Time Complexity in Binary Search Trees / 2

- **Searching:** In general, time complexity is O(h) where **h** is height of BST.
- **Insertion:** In general, time complexity is O(h).
- **Deletion:** In general, time complexity is O(h).

Tree Sort Algorithm

- Tree sort is a sorting algorithm that is based on Binary Search Tree data structure. It first creates a binary search tree from the elements of the input list or array and then performs an in-order traversal on the created binary search tree to get the elements in sorted order.
- Algorithm:
- **Step 1:** Take the elements input in an array.
- **Step 2:** Create a Binary search tree by inserting data items from the array into the binary search tree.
- **Step 3:** Perform in-order traversal on the tree to get the elements in sorted order.

https://www.geeksforgeeks.org/tree-sort/

Tree Sort

- Tree sort algorithm – inorder traversal of BST
- Time complexity of building the n-node binary search tree is $\Theta(n \log_2 n)$
- Time complexity of inorder traversal is time to visit every node once – $\Theta(n)$
- Time complexity of the tree sort algorithm:
  - $\max(\Theta(n \log n), \Theta(n)) = \Theta(n \log n)$
  - Average case behavior
- Worst case $\Theta(n^2)$

Tree Sort (continued)

ALGORITHM	n = 100,000 (ALL TIMES IN SECONDS) RANDOM ORDER	REVERSE ORDER
Merge sort	6.7	7.2
Quicksort	5.4	89.5
Tree sort	9.6	101.7
Heap sort	7.5	6.8

[FIGURE 7-22] Performance of the tree sort algorithm

Balanced Binary Search Trees

- BST built from random values has a height $H = \Theta(\log n)$
  - `search()` runs in logarithmic time
  - Worst case, BST highly unbalanced
- Worst-case behavior may be unacceptable
  - **Real-time programs** – deliver results in guaranteed time period
- Types of balanced search trees
  - **2-3 trees**
  - **Red-black trees**
  - **AVL trees**
  - **B-trees**

Balanced Binary Search Trees

- To guarantee that basic dynamic-set operations take O(lg n) time in the worst case
- Types of balanced search trees
  - **AVL trees**
  - **2-3 trees**
  - **2-3-4 trees**
  - **B-trees**
  - **Red-black trees**
  - **Skip lists**
  - **Splay tree**
  - **Treaps**

Properties of Red-Black Trees

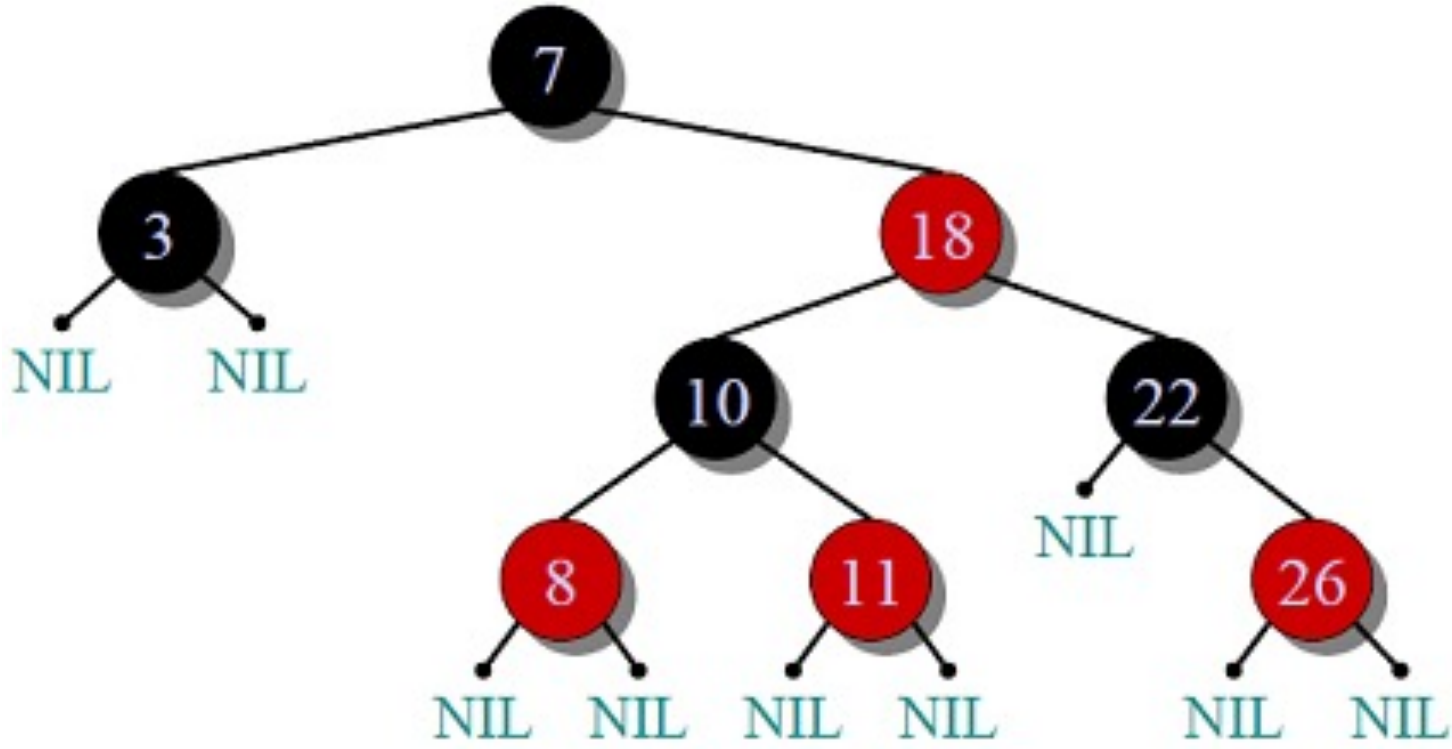
- A red-black tree is a binary search tree that satisfies the following ***red-black properties***:
  - every node is either **red** or **black**
  - the **root** node and **leaves** (NIL) are **black**
  - if a node is red, then both its children are black (no two consecutive red nodes in a path)
  - for each node, all simple paths from the node to descendant leaves contain the same number of black nodes
- Proposition: The height of a red-black tree with n internal nodes is $\leq 2 * \lg(n+1) = O(\lg n)$

By constraining the node colors on any simple path from the root to a leaf, red black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Shortest Path: all black nodes

Longest Path: alternating red and black

.

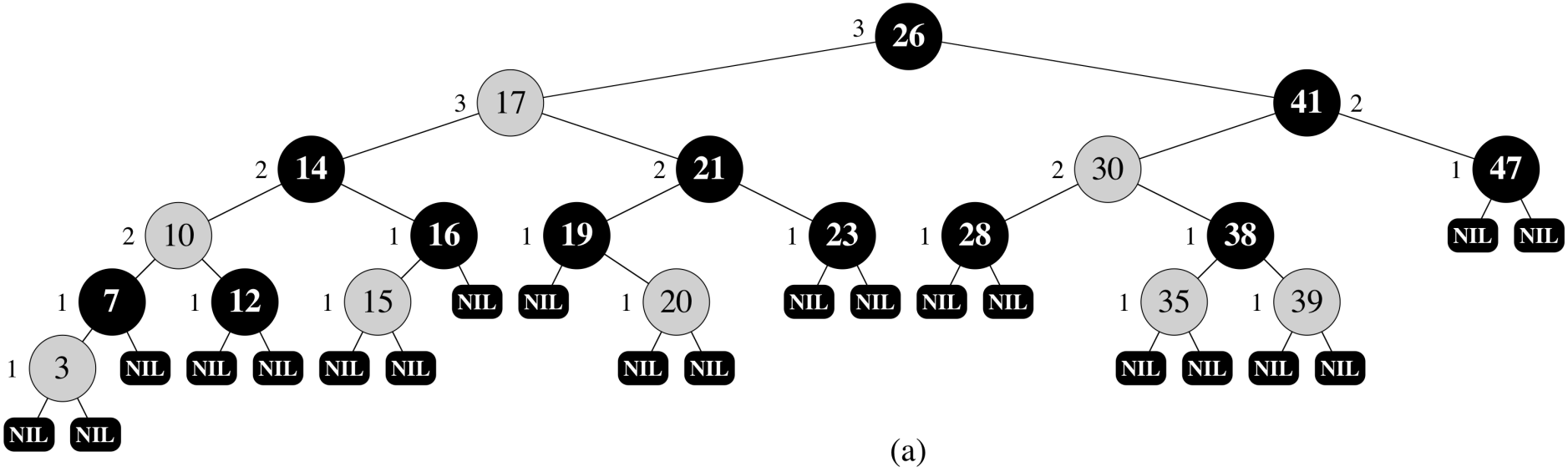


Why Red-Black Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)** time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations.
- The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

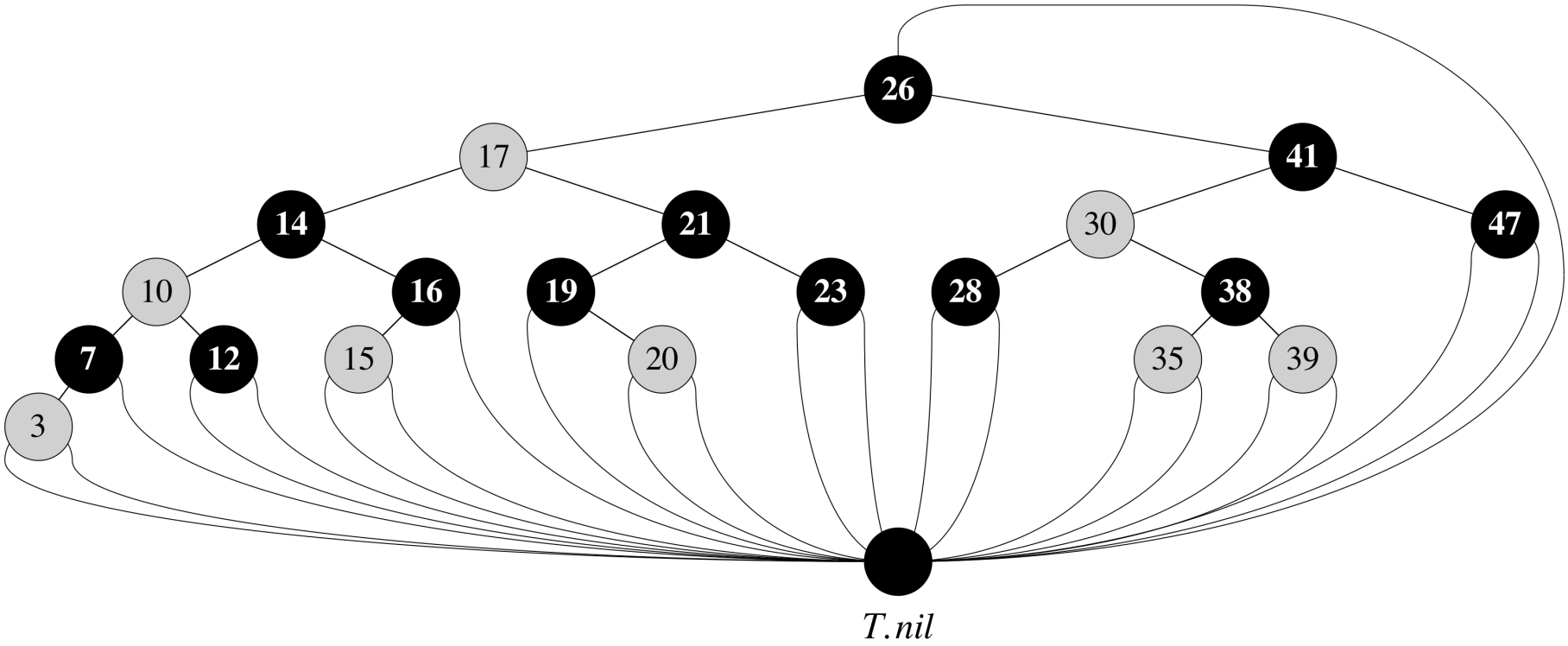
https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

Sample Red-Black Tree

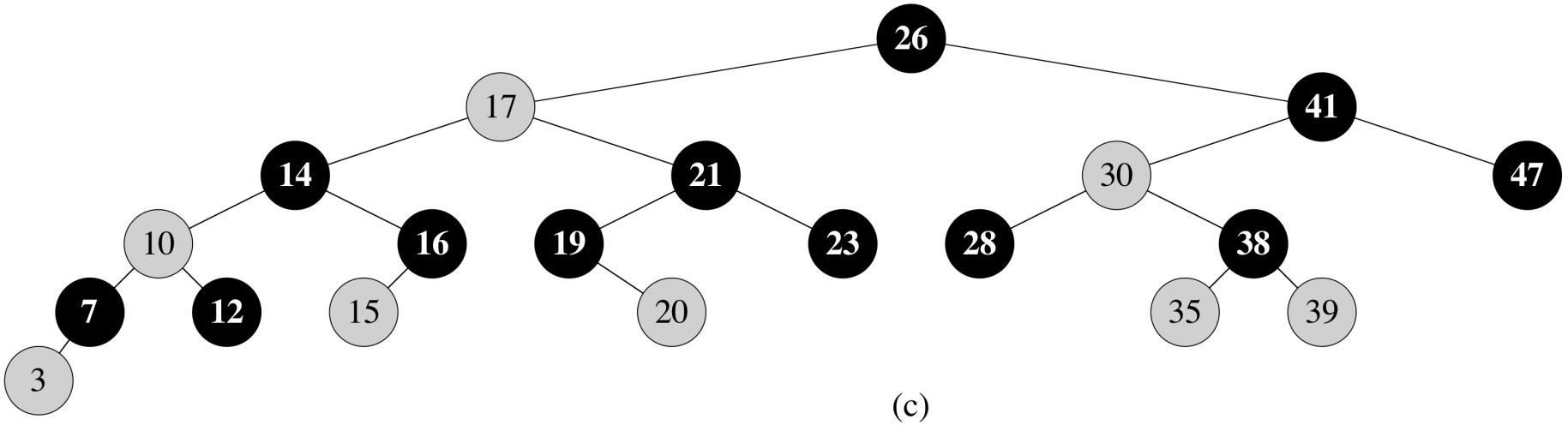


(a)

Alternative Representation



Common Representation



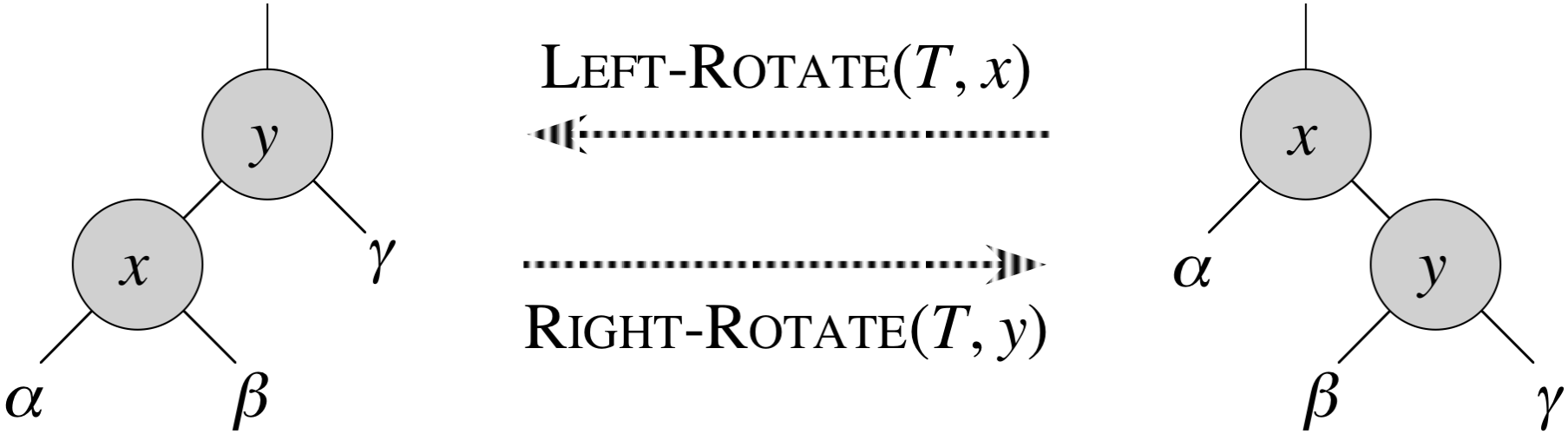
(c)

Common Operations

- Search
- Insert
- Delete

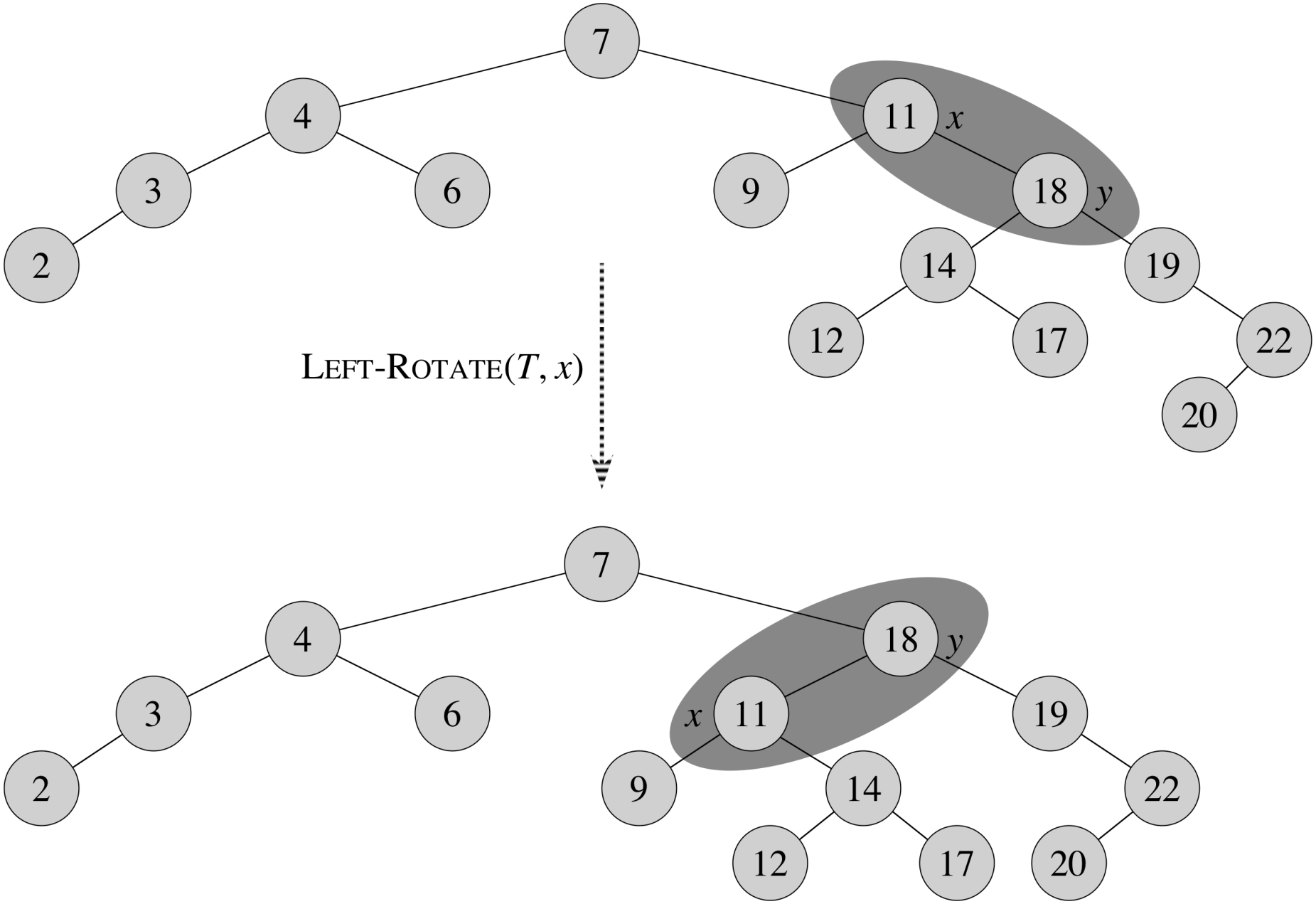
Require Rotation

Rotations



Rotation preserves BST property

$\alpha < x < \beta < y < \gamma$


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

```
LEFT-ROTATE(T, x)
  y = x.right                    // set y
  x.right = y.left               // turn y's left subtree into x's right subtree
  if y.left ≠ T.nil
       y.left.p = x
  y.p = x.p                      // link x's parent to y
  if x.p == T.nil
       T.root = y
  elseif x == x.p.left
       x.p.left = y
  else x.p.right = y
  y.left = x                     // put x on y's left
  x.p = y
```

Insertion

- Search
- Insert
- Delete

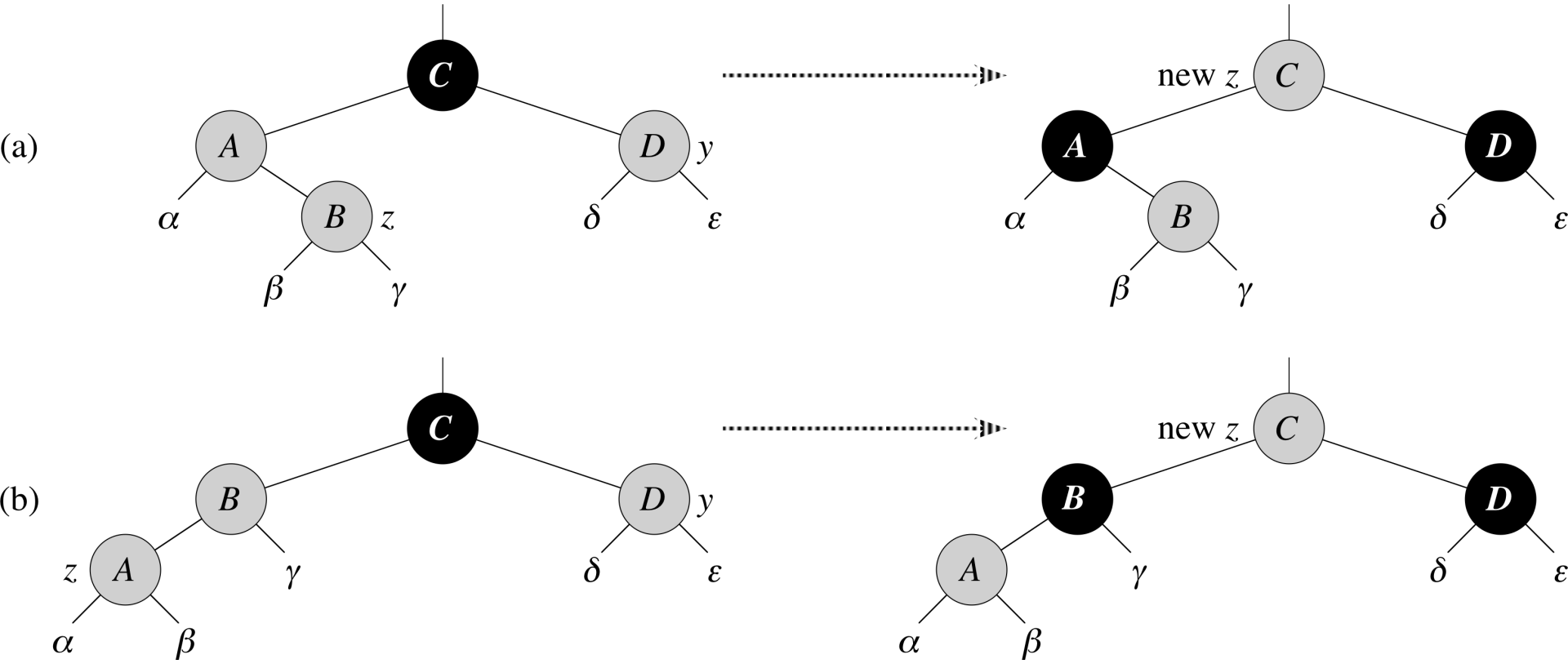
TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z          // tree T was empty
elseif z.key < y.key
      y.left = z
else y.right = z
```

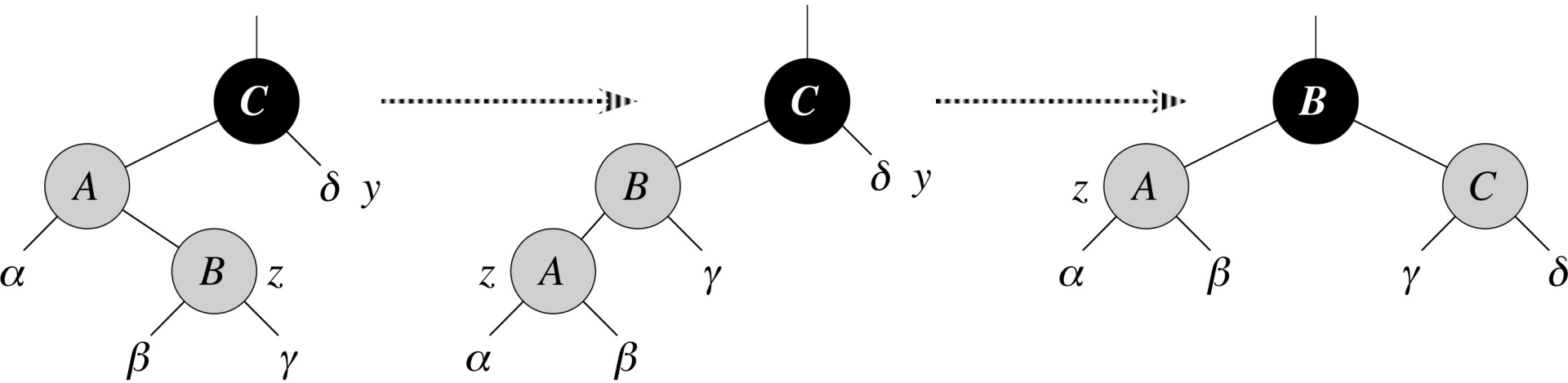
RB-INSERT(T, z)

```
y = T.nil
x = T.root
while x ≠ T.nil
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == T.nil
      T.root = z
elseif z.key < y.key
      y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK                          // case 1
              y.color = BLACK                            // case 1
              z.p.p.color = RED                          // case 1
              z = z.p.p                                  // case 1
          else if z == z.p.right
                  z = z.p                                // case 2
                  LEFT-ROTATE(T, z)                      // case 2
              z.p.color = BLACK                          // case 3
              z.p.p.color = RED                          // case 3
              RIGHT-ROTATE(T, z.p.p)                     // case 3
      else (same as then clause with "right" and "left" exchanged)
  T.root.color = BLACK
```

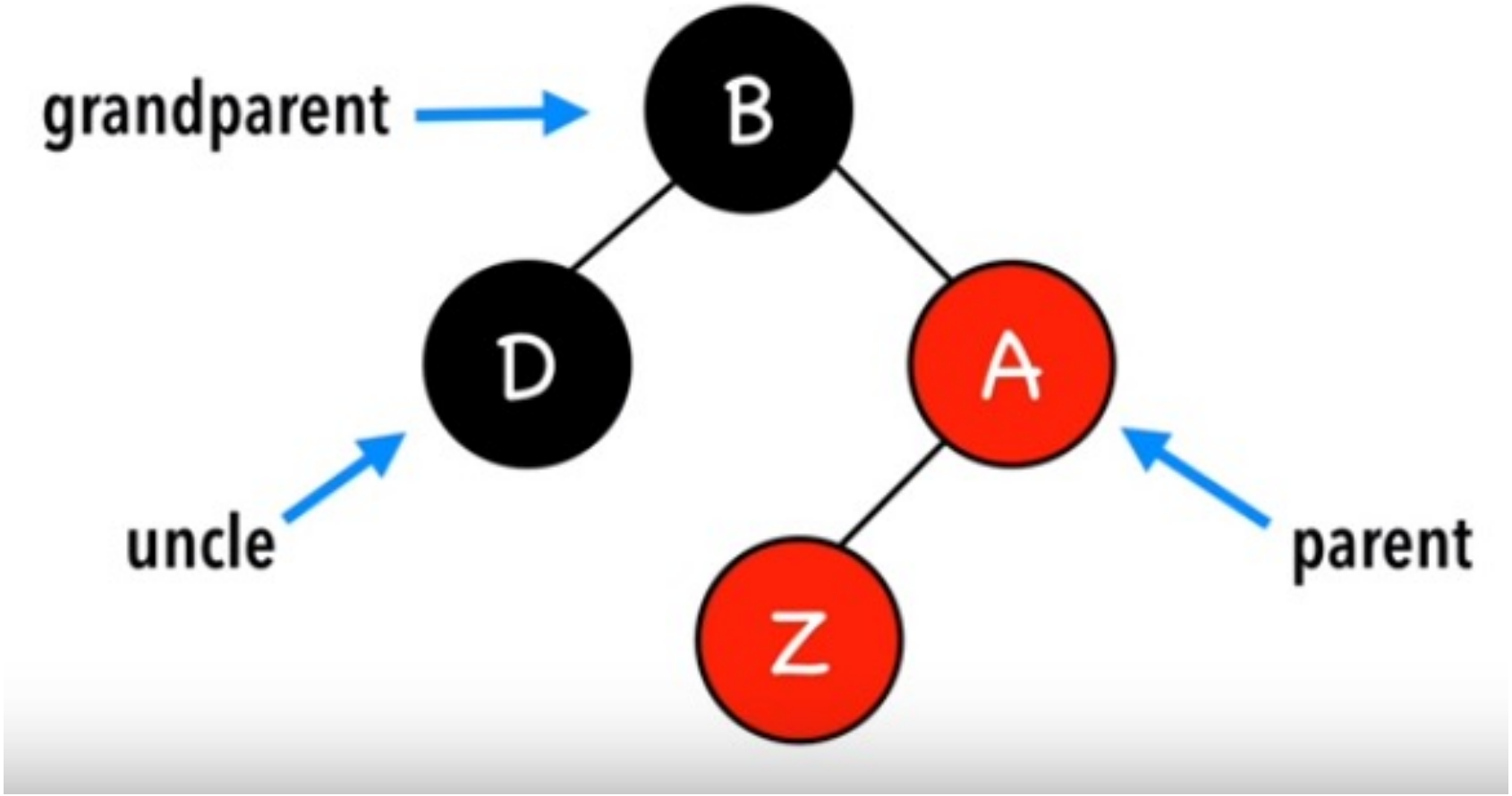

(a)
C
A
D
y
α
B
z
δ
ε
β
γ
new z
C
A
D
α
B
δ
ε
β
γ
(b)
C
B
D
y
z
A
γ
δ
ε
α
β
new z
C
B
D
A
γ
δ
ε
α
β

Case 1



Case 2 Case 3

Z's Relationships



https://www.youtube.com/watch?v=5IBxA-bZZH8

Insertion Strategy

1. Insert Z and color it red
2. Recolor and rotate nodes to fix violation
   a. Case 1= Z is the root
   b. Case 2= Z.uncle =red
   c. Case 3= Z.uncle = black (triangle)
   d. Case 4= Z.uncle black (line)

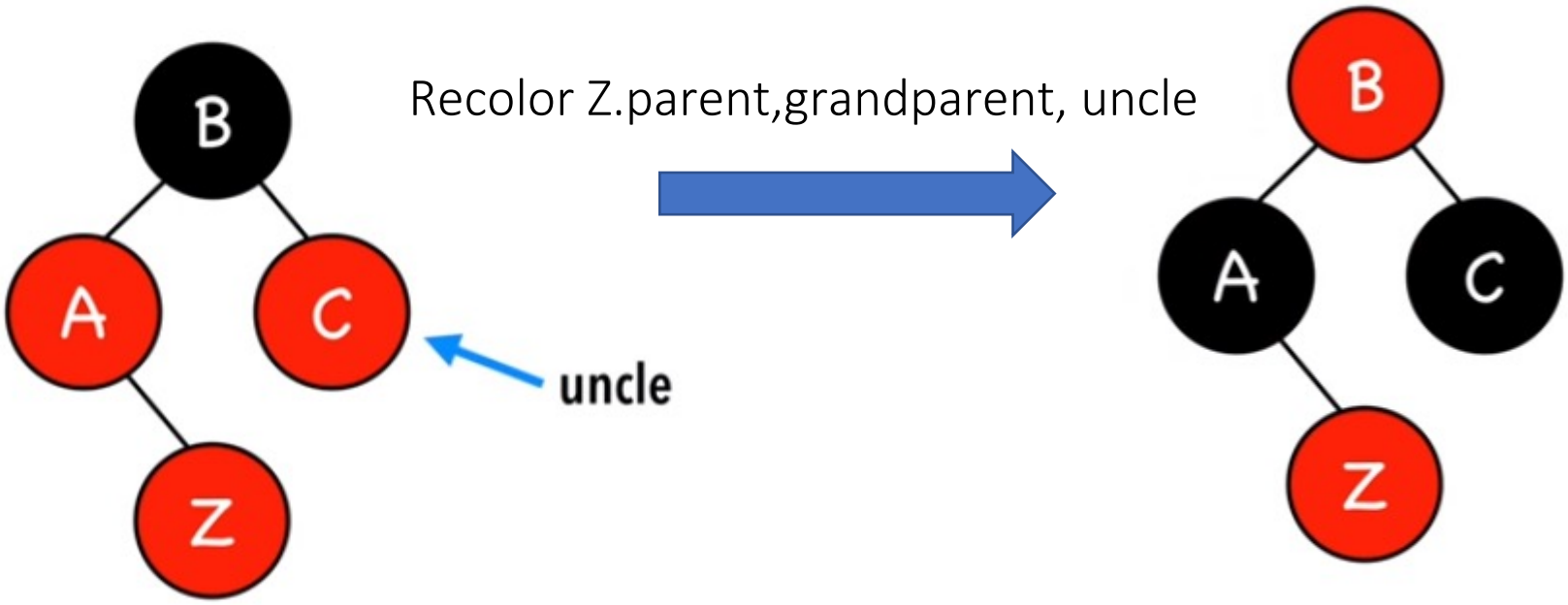
https://www.youtube.com/watch?v=5IBxA-bZZH8

Case 1 : Z is the root

Color Z → black

https://www.youtube.com/watch?v=5IBxA-bZZH8

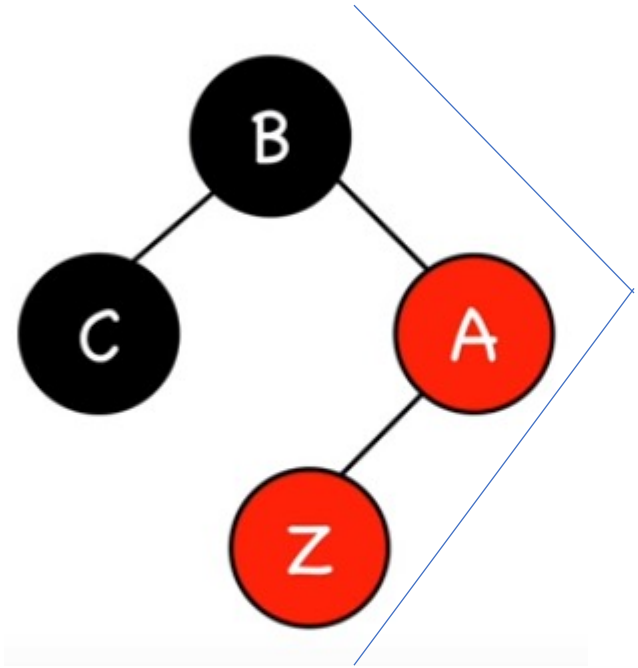
Case 2= Z.uncle =red



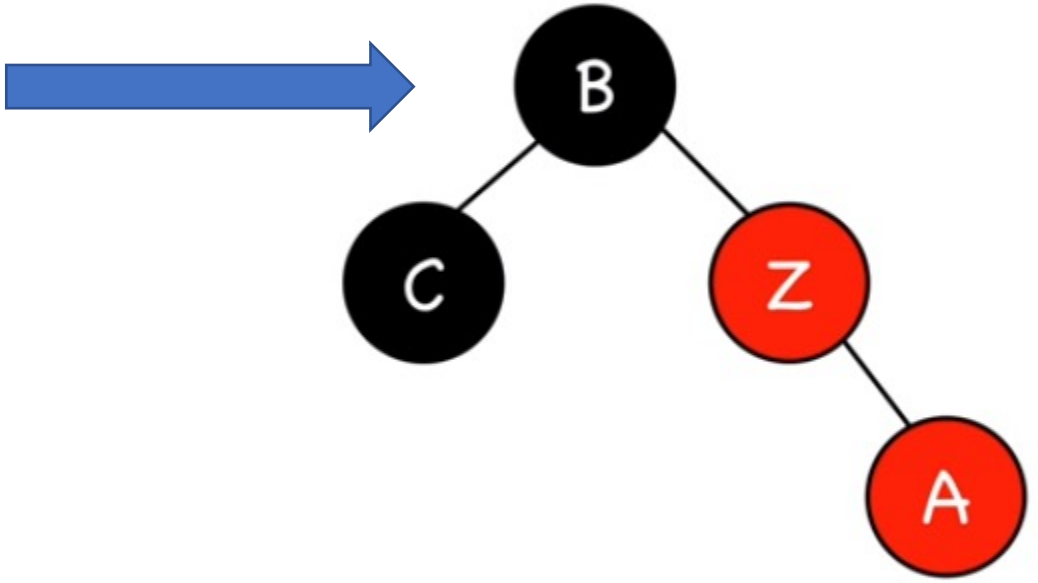
Assume this is a subtree

https://www.youtube.com/watch?v=5IBxA-bZZH8

Case 3= Z.uncle =black (triangle)



Rotate Z.parent

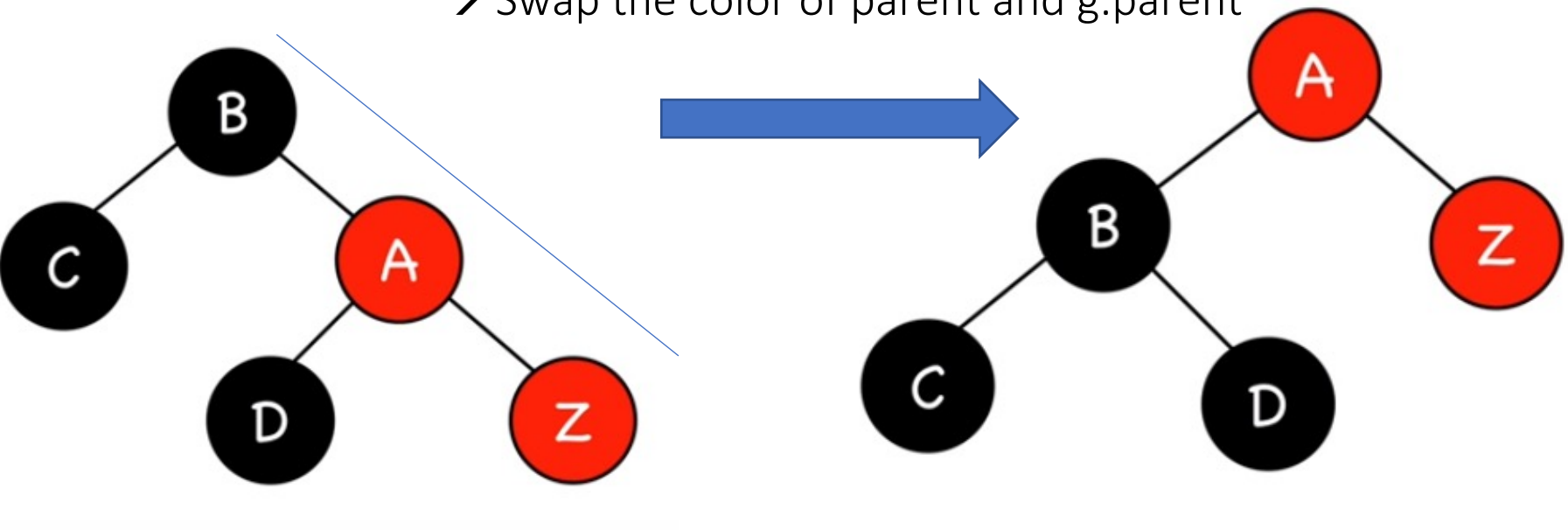


Assume this is a subtree

https://www.youtube.com/watch?v=5IBxA-bZZH8

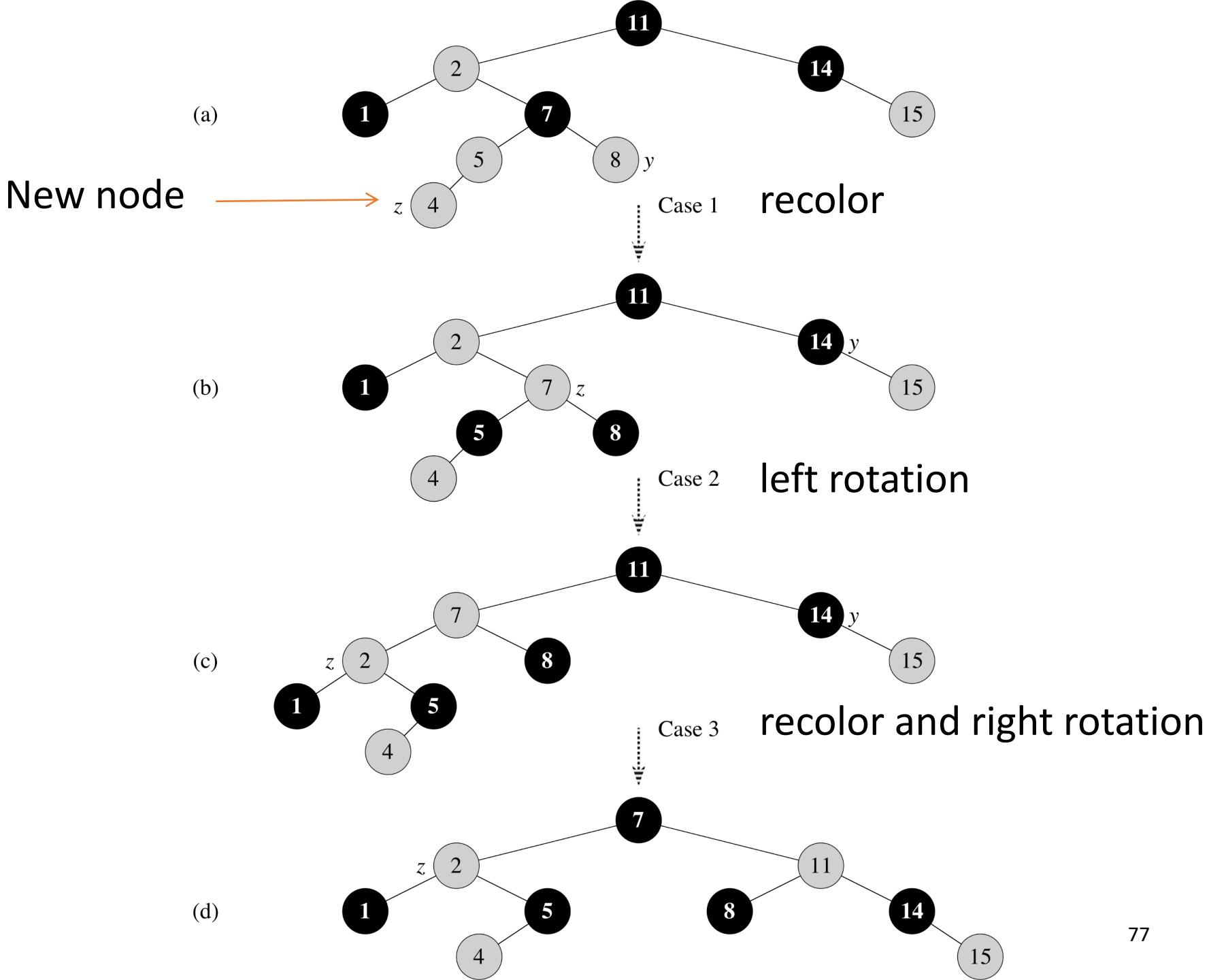
Case 4= Z.uncle =black (line)

Rotate Z.grandparent
recolor parent and grandparent
→ Swap the color of parent and g.parent

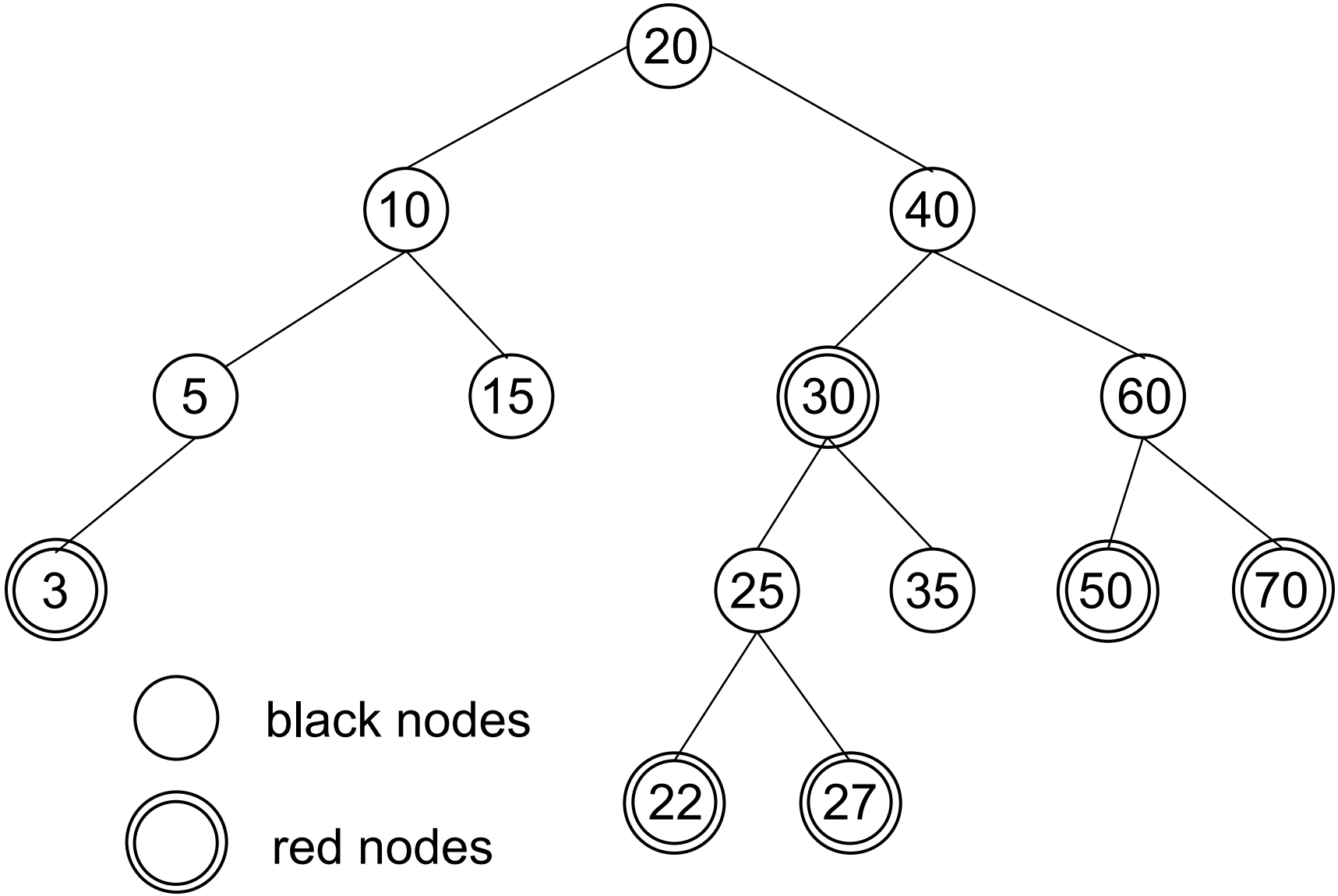


Assume this is a subtree

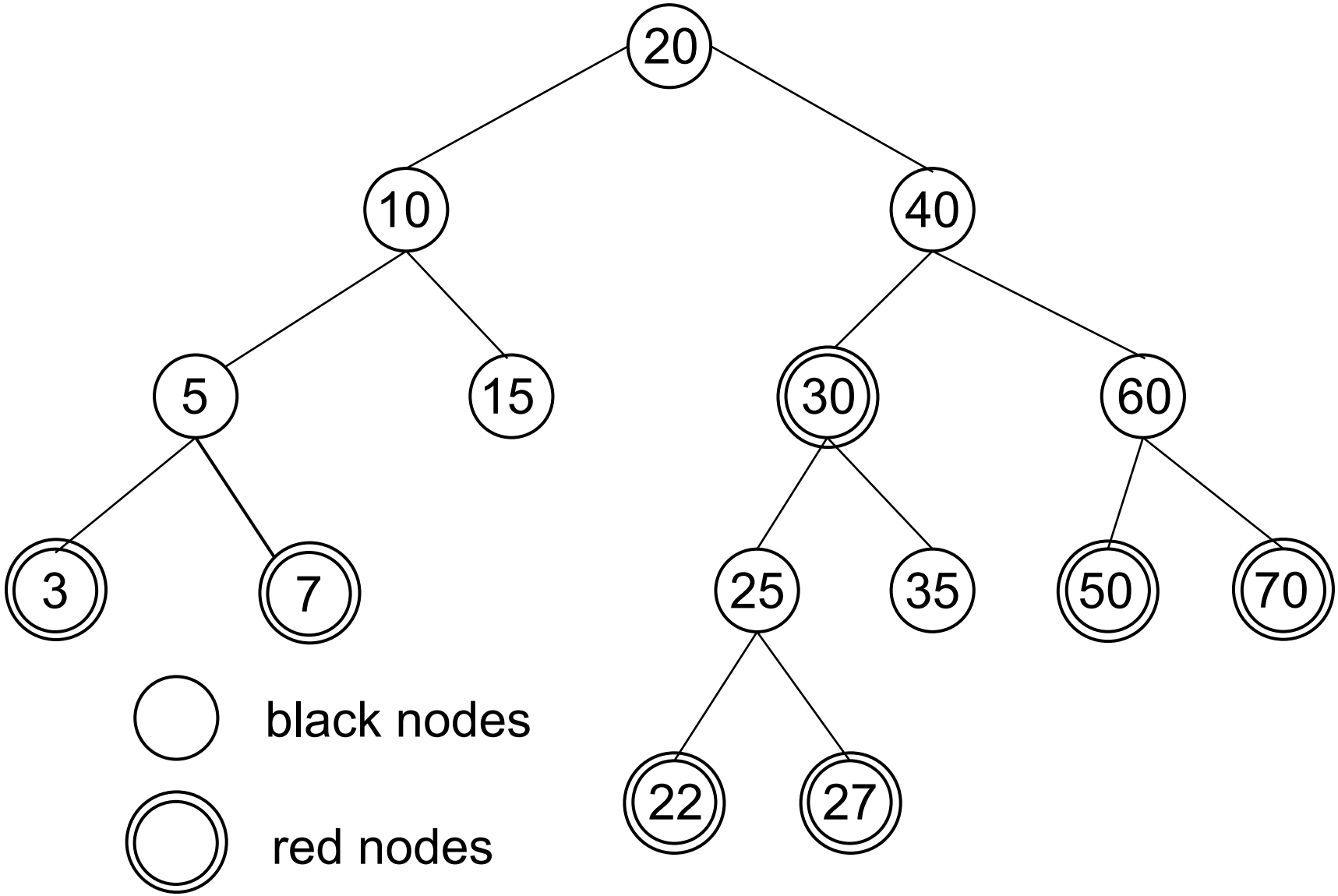
https://www.youtube.com/watch?v=5IBxA-bZZH8


(a)
11
2
14
1
7
15
5
8
y
New node
z
4
Case 1
recolor
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
left rotation
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
recolor and right rotation
(d)
7
z
2
11
1
5
8
14
4
15

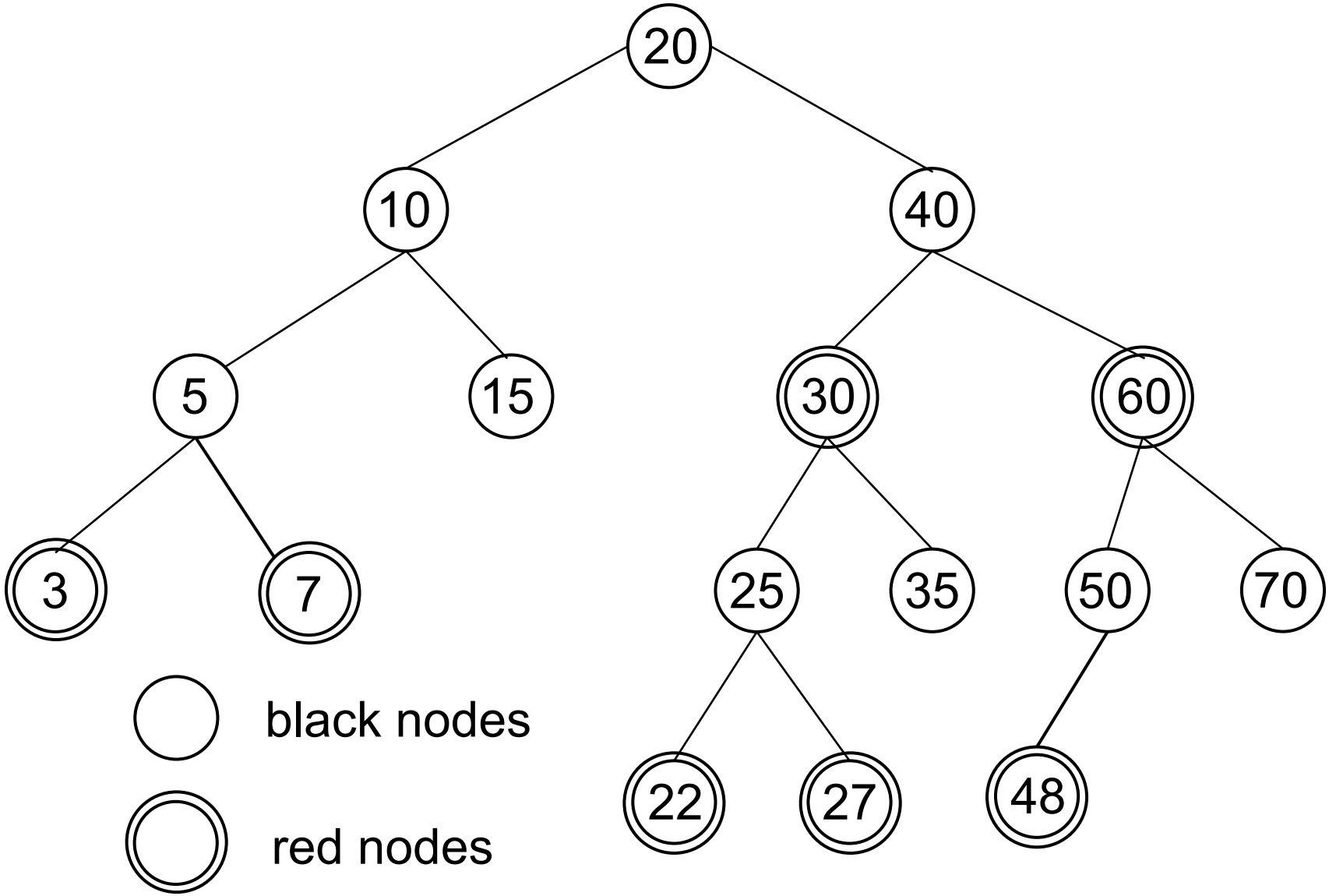
Red-Black Tree Insert Example



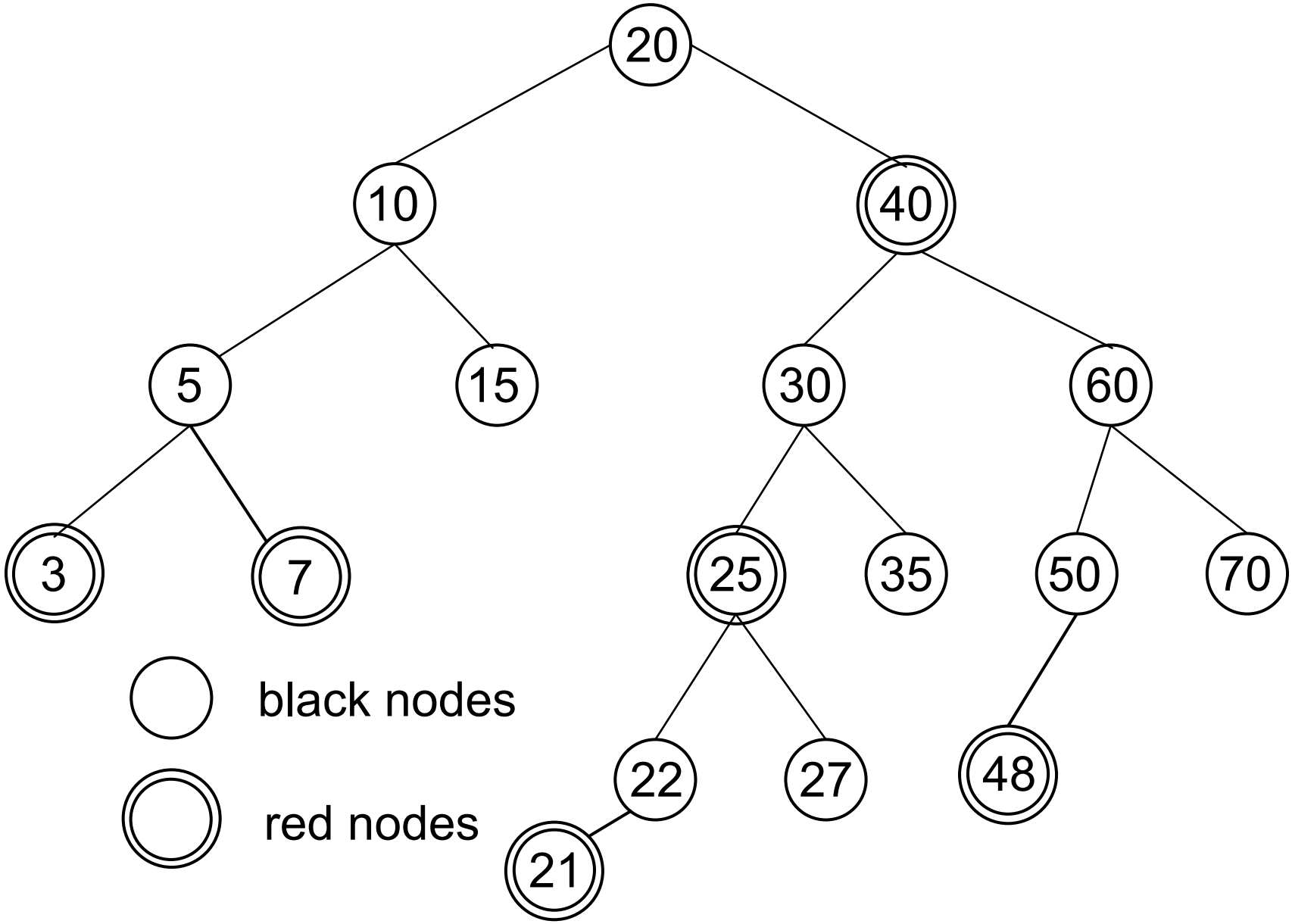
Insert 7



Insert 48



Insert 21



Useful website

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

Time Complexity

- Search → O (log n)
- insert → O (log n)
- Delete → O (log n)

Graphs

- Social Media
- Maps
- Networks
- WWW
- ….

GRAPHS

- Lots of problems formulated and solved in terms of graphs
  - Shortest path problems
  - Network flow problems
  - Matching problems
  - 2-SAT problem
  - Graph coloring problem
  - Traveling Salesman Problem (TSP)

Basics

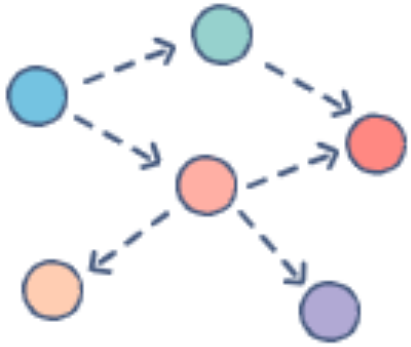
- **Graph** – a hierarchical data structure in which elements related to an arbitrary number of other elements
  - *many-to-many* data structure
  - **nodes** (**vertices**) connected by **edges**
  - two nodes **adjacent** (**incident**) if they share an edge
- Graph *G = (V, E)*, where *V* is the set of vertices and *E* is the set of edges

Basics

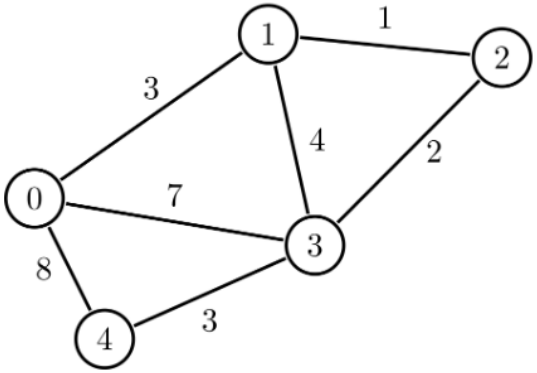
- **Directed graph** – edges have a direction
- **Undirected graph** – edges have no direction
- Edges are **weighted** if they have an associated value – **weighted graph**
- **Path** – finite sequence of nodes such that each pair of nodes connected by an edge
- **Cycle** – simple path in which the first and last nodes are connected
- **Directed acyclic graph** (DAG) contains no cycles, edges are directed

- **Simple Path:** A path in which no vertices (thus no edges) are repeated
- **Walk:** A sequence of vertices where each adjacent pair is connected by edge
- **Tail:** A walk in which no edges are repeated

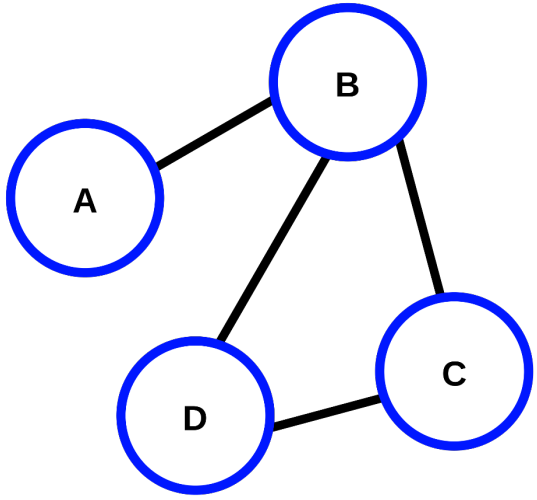
Undirected



Directed



Weighted Graph



Unweighted Graph

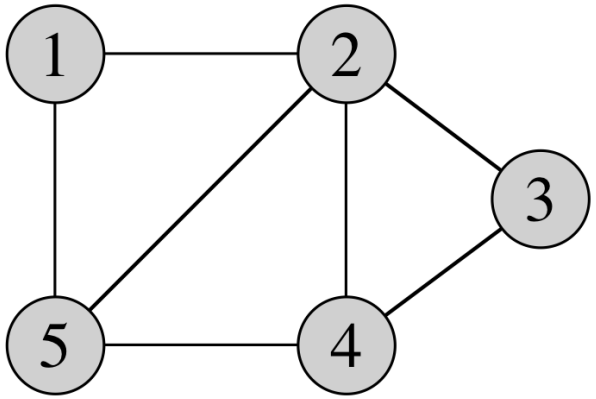
https://www.educative.io/edpresso/what-is-a-graph-data-structure
https://hyperskill.org/learn/step/5645

Representation of Graphs

- Two techniques for representing graphs
  - **Adjacency matrix**
  - **Adjacency list**
- Adjacency matrix uses *n*x*n* two-dimensional array of *A* boolean values
  - *A(x,y)* = true if there is an edge from x to y
    - Undirected graph *A(x,y) = A(y,x)*
    - Weighted graph *A(x,y)* has a real value
- Adjacency matrix appropriate for **dense** graphs, uses $\Theta(V^2)$ memory locations

Adjacency List Representation

- **Adjacency list** is more space efficient, requires $\Theta(V + E)$ memory locations
- Create an *n*-element one-dimensional array (**node list**)
- Each element is the head of a linked list of all neighbors (**neighbor list**)
  - Each entry is a **neighbor node**
- Implemented using an array of references to **neighbor** objects
  - Neighbor object contains a node identifier, weight, reference to another node



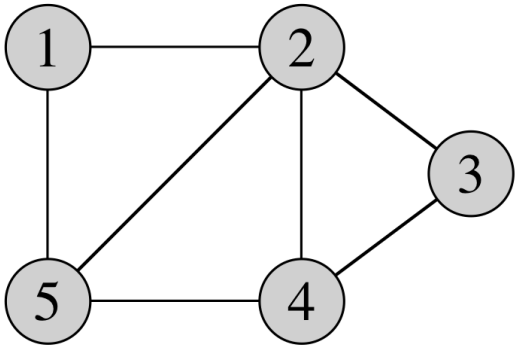
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

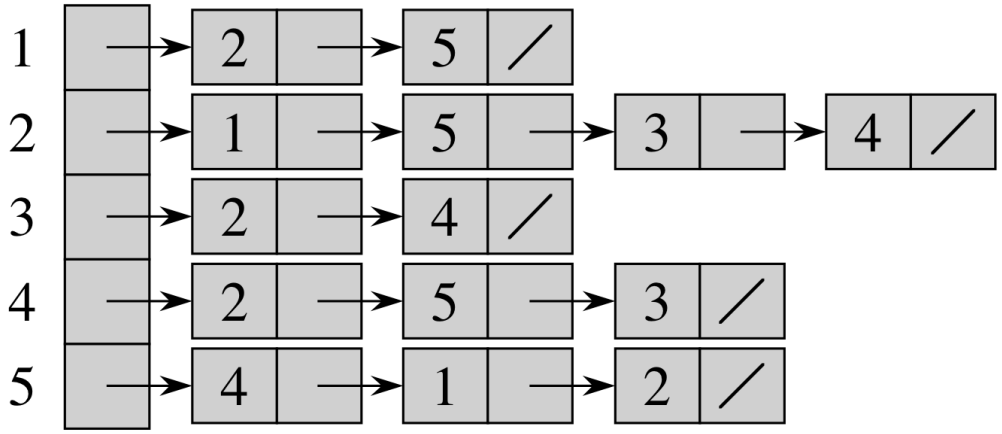
$$|V| \times |V| \text{ matrix } A = (a_{ij})$$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise .} \end{cases}$$

Time taken:

- to list all vertices adjacent to a node u – $\Theta(V)$
- to determine whether $(u, v) \in E$ – $\Theta(1)$



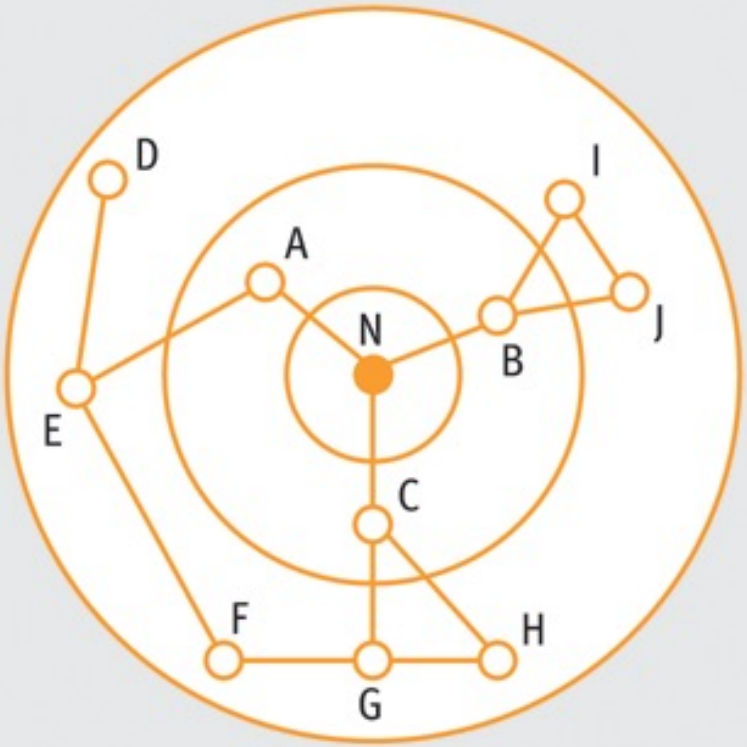


Time taken:

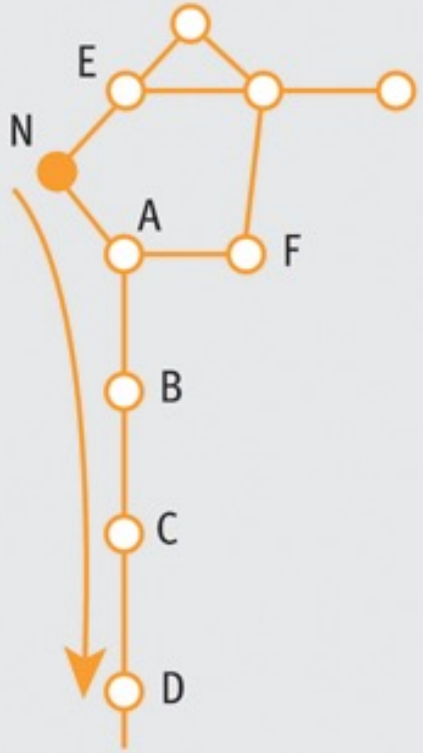
- to list all vertices adjacent to a node u – $\Theta(degree(u))$
- to determine whether $(u, v) \in E$ – $O(degree(u))$

Graph Traversal

- **Traversal** – visit every node in the graph exactly once
- **Breadth-first** – visit all neighbors before visiting non-neighbors
- Node B is a **neighbor** of node A if there is an edge between them
- **Depth-first** search follows a specific path as far as it leads
  - Begin another path when the first is exhausted



(a) Breadth-first traversal

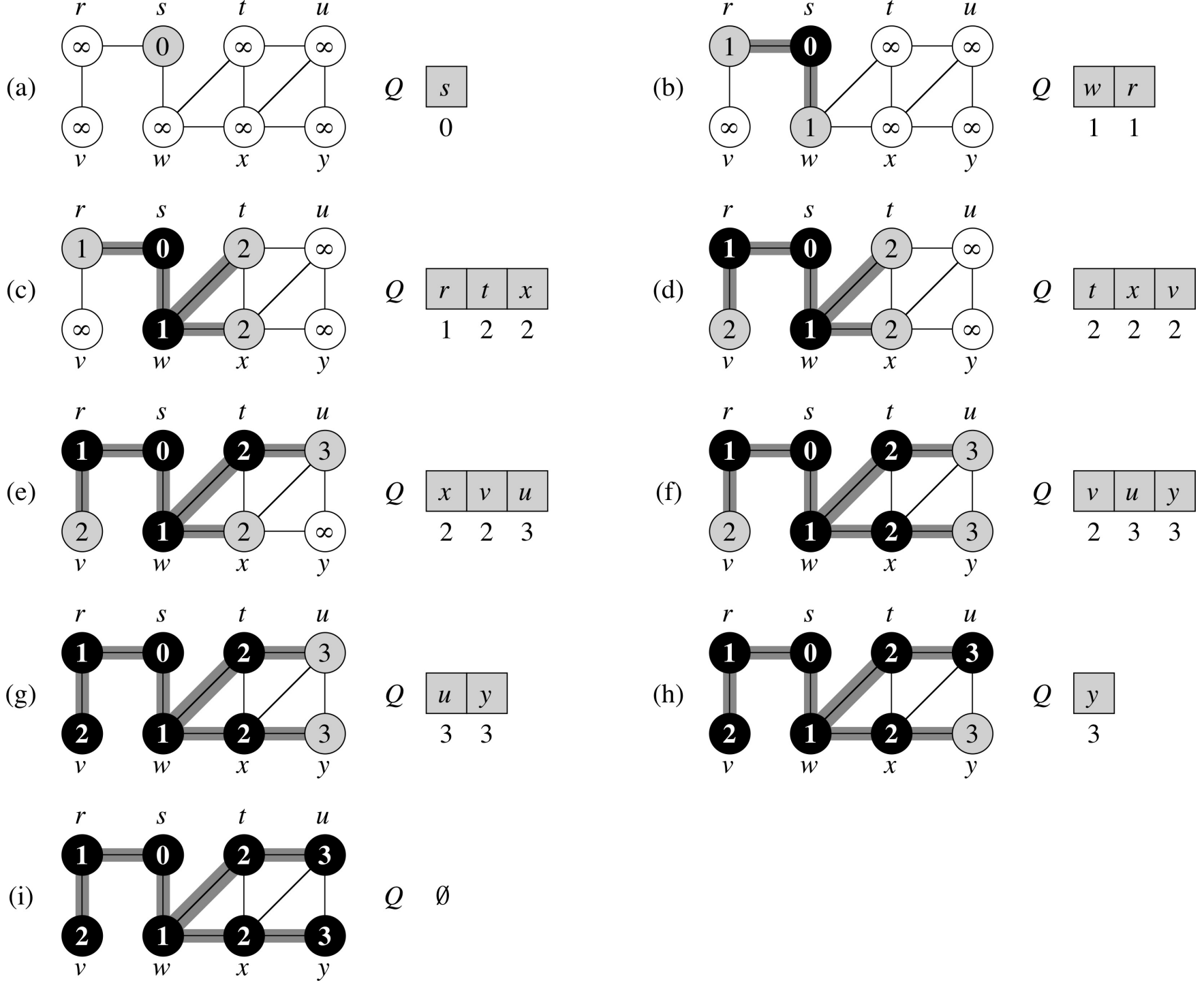


(b) Depth-first traversal

Breadth-first Search

- Each vertex keeps a **status value**: *visited, waiting, not visited*
  - All nodes start in the not-visited state
- Breadth-first search – **First In First Out structure**
  - Mark and enqueue node N, visit it, mark visited
  - Enqueue N's neighbors, mark as waiting
- Breadth-first search can produce a set of **connected subgraphs**
- Breadth-first search finds the shortest distance to each reachable vertex in a graph G(V, E) from a given source vertex $s \in V$ (also called the **shortest path**)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q
s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q
w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
∅

Exercise

Cost Analysis

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

- lines 1-4 – |V|
- lines 10-18
  - **while** loop iterates over all nodes (note that each node is added to the queue and removed only once)
  - **for** loop iterates over all edges from a given node (since a node is removed from the queue only once, we visit it's corresponding edges only once)
  - total time spent scanning adjacency lists =
- total time = O(V + E)

$$\sum_{u \in V} G.Adj[u] = |E|$$

Breadth-first trees

- BFS builds a breath-first tree (predecessor subgraph) while searching the graph (remember we did not do anything with the π attribute)
- We can display the shortest path from node s to node v using the breath-first tree created by BFS

```
PRINT-PATH(G, s, v)
1  if v == s
2        print s
3  elseif v.π == NIL
4        print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6        print v
```

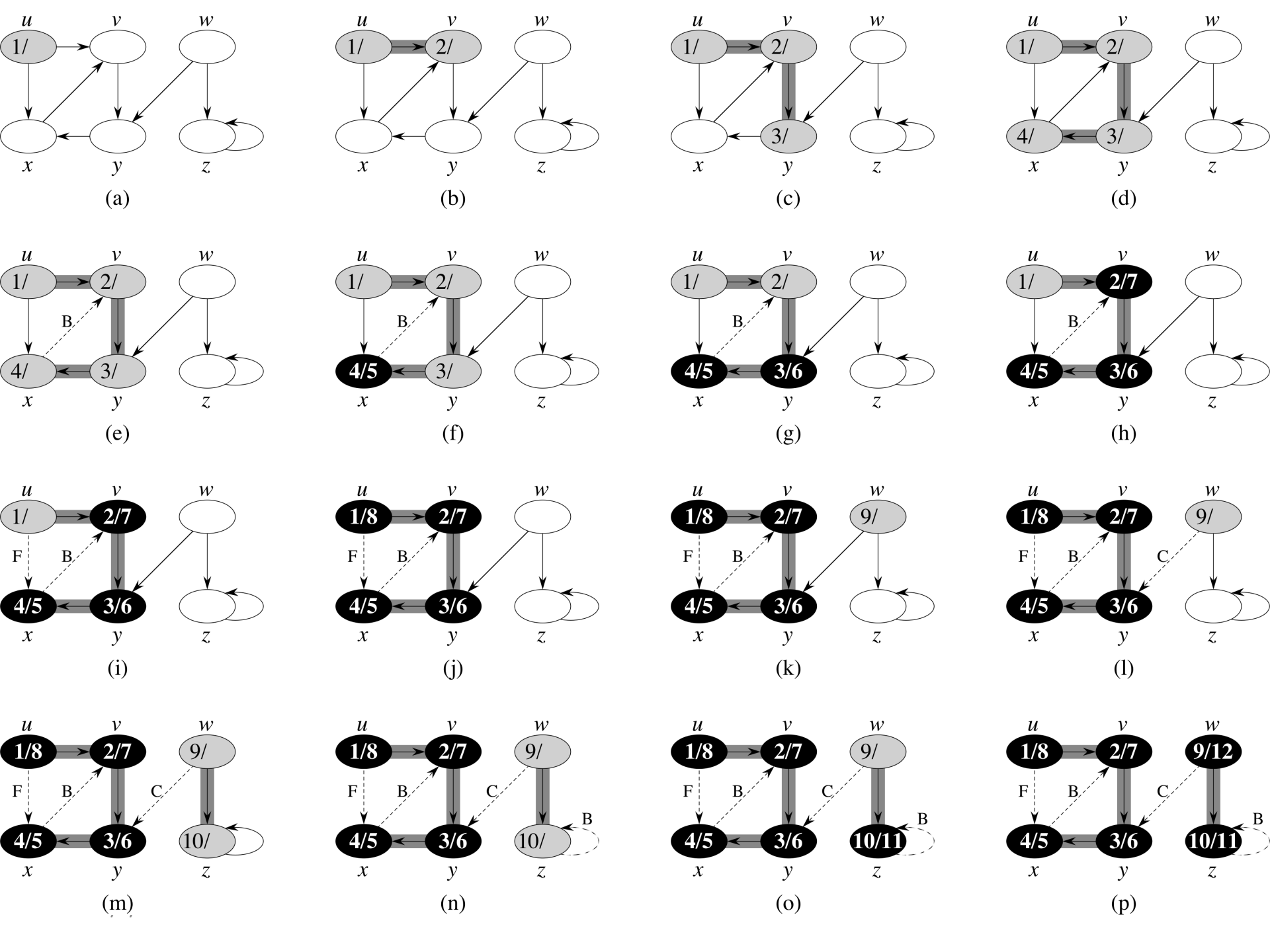
Depth-first Search

- Depth-first traversal – Last In First Out structure
  - Visit node N then stack up its neighbors
  - Pop the top node, visit it, stack its neighbors
- Visit the nodes that were added more recently
  - Continue following the current path
- When path end is reached, pop the next node and start on that path
  - New path begins at last node visited in current path

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

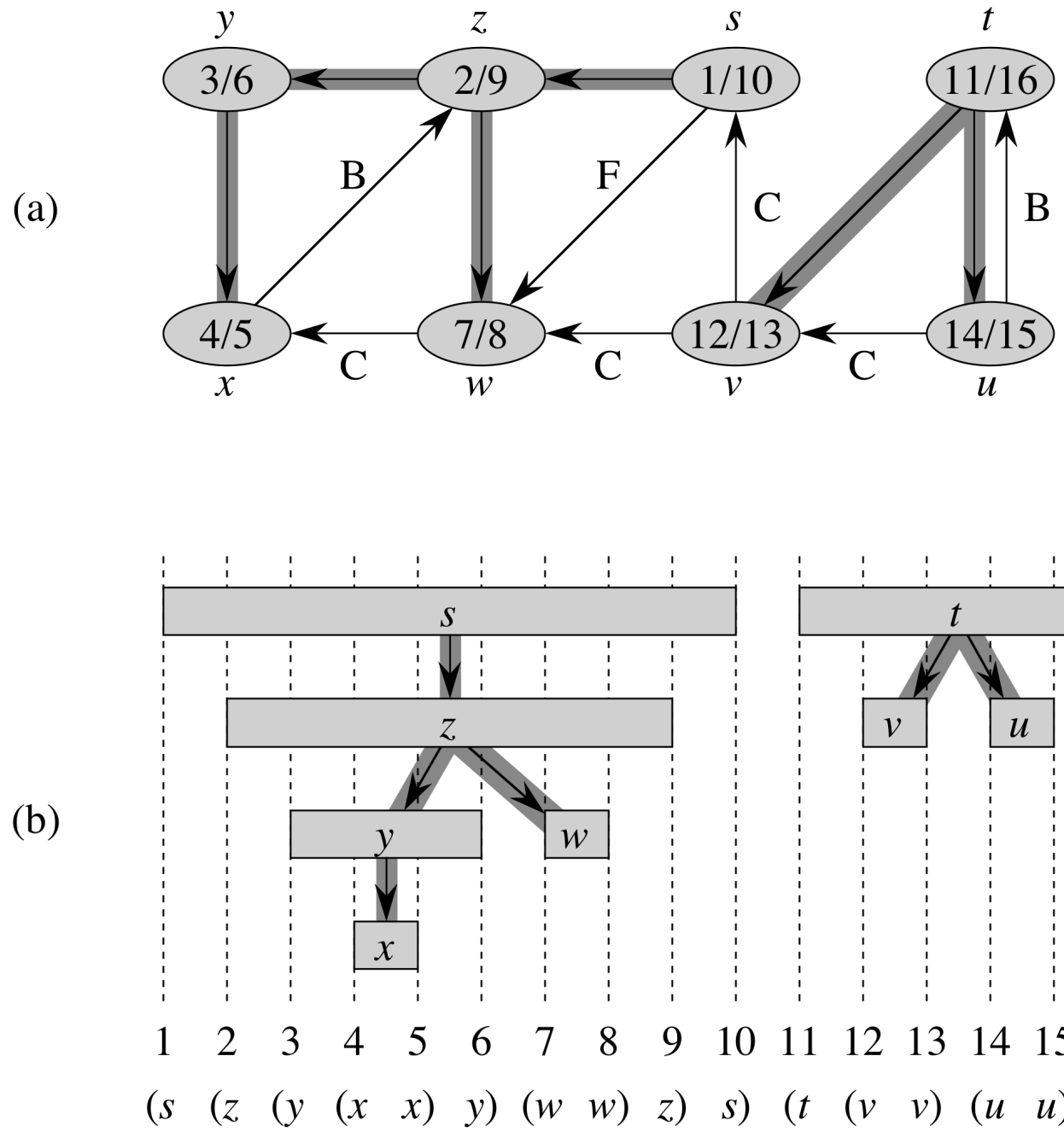
```
DFS(G)
1  for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6       if u.color == WHITE
7            DFS-VISIT(G, u)
```

Running time?


u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/
3/
x y z
(c)
u v w
1/ 2/
4/ 3/
x y z
(d)
u v w
1/ 2/
B
4/ 3/
x y z
(e)
u v w
1/ 2/
B
4/5 3/
x y z
(f)
u v w
1/ 2/
B
4/5 3/6
x y z
(g)
u v w
1/ 2/7
B
4/5 3/6
x y z
(h)
u v w
1/ 2/7
F B
4/5 3/6
x y z
(i)
u v w
1/8 2/7
F B
4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/ B
x y z
(n)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/11 B
x y z
(o)
u v w
1/8 2/7 9/12
F B C
4/5 3/6 10/11 B
x y z
(p)

DFS Observations

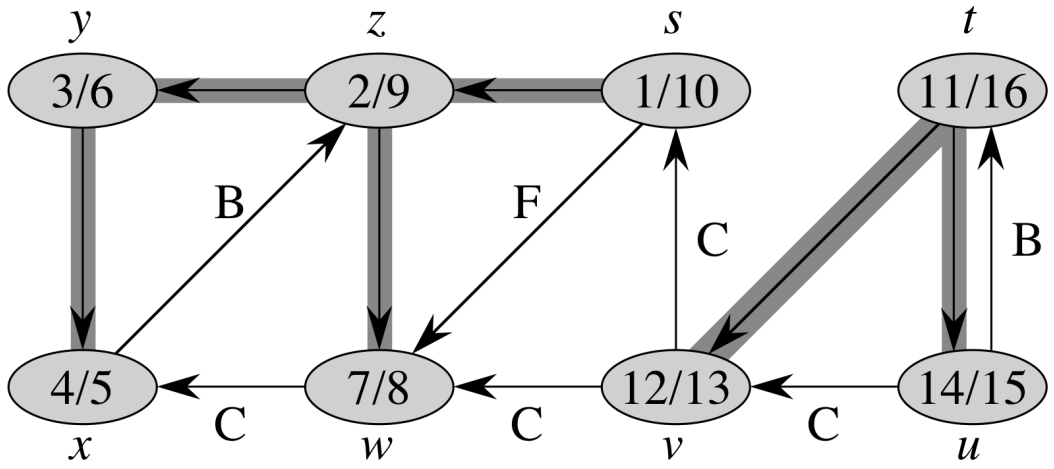
- DFS does not find the shortest path, but provides valuable information about the structure of a graph
- The predecessor subgraph form a forest of trees
- Discovery and finishing times have ***parenthesis structure*** (see diagram)
- Useful for
  - Edge classification
  - Cycle detection
  - Topological sort
  - Strongly connected components


(a)
y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u
(b)
s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

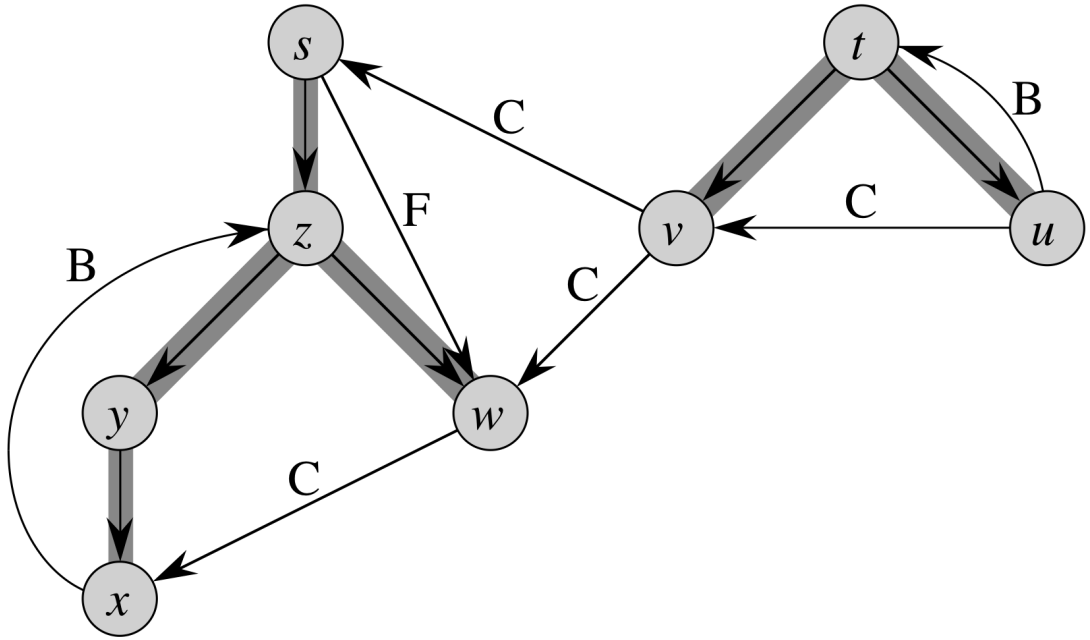
Edge classification

- Tree edges (parent pointer) – visit new vertex via this edge
- Forward edges – goes from a node to its descendant
- Back edges – goes from a node to its ancestor
- Cross edges – goes from one tree to another tree (between two non-ancestor related subtrees)
- *A directed graph is acyclic if and only if a depth-first search yields no "back" edges*
- *Forward and cross edges never occur in a depth-first search of an undirected graph*

(a)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u

(c)


s
t
C
B
F
C
z
v
u
B
C
y
w
C
x

Reachability in digraphs

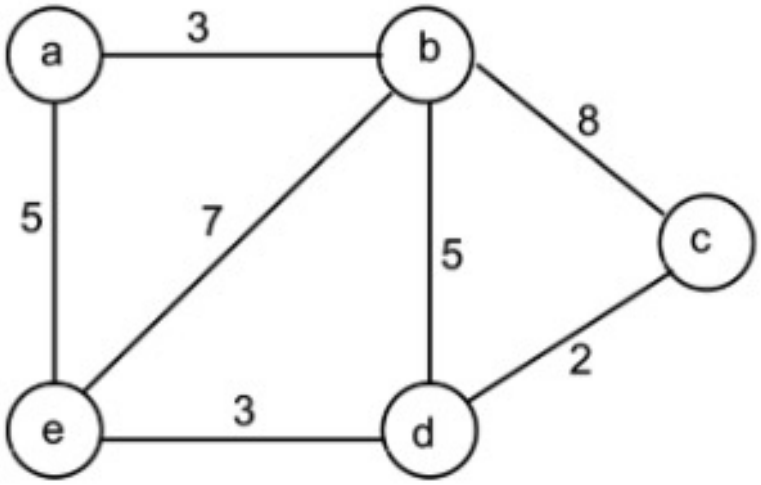
- **Single-source reachability:** Given a digraph and source s, is there a directed path from s to v? If so, find such a path.
- **Multiple-source reachability:** Given a digraph and a set of source vertices, is there a directed path from any vertex in the set to v?
- **Single-source directed paths:** Given a digraph and source s, is there a directed path from s to v? If so, find such a path.
- **Single-source shortest directed paths:** Given a digraph and source s, is there a directed path from s to v? If so, find a shortest such path.

Reachability in digraphs

- **Single-source reachability:** Given a digraph and source s, is there a directed path from s to v? If so, find such a path. **Use depth-first search to solve this problem.**
- **Multiple-source reachability:** Given a digraph and a set of source vertices, is there a directed path from any vertex in the set to v? **Use depth-first search to solve this problem**.
- **Single-source directed paths:** Given a digraph and source s, is there a directed path from s to v? If so, find such a path. **Use depth-first search to solve this problem**.
- **Single-source shortest directed paths:** Given a digraph and source s, is there a directed path from s to v? If so, find a shortest such path. **Use breadth-first search to solve this problem.**

Spanning Tree

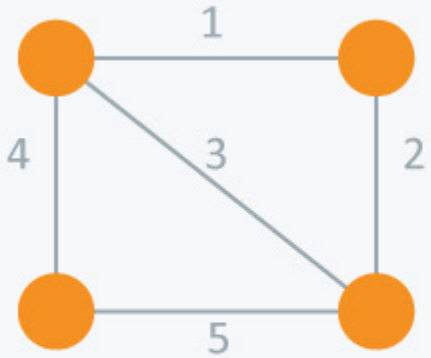
- **Spanning Tree:** A subset of the edges of a *connected, weighted and undirected* graph that connects all the nodes together **without cycle**
- Find the Spanning Trees of the following graph



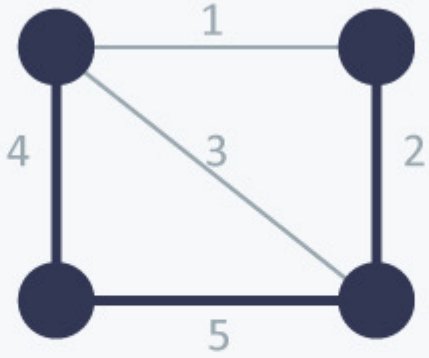
Minimum Spanning Trees (MST)

- MST is a spanning tree where the cost is minimum among all the spanning trees
  - There can be many minimum spanning trees in some cases
- There are different methods to find the MST of a graph
- Why it is useful?
  - Network design
  - Travelling Salesman Problem
  - Cluster Analysis
  - Handwriting Recognition
  - Image segmentation
  - ….

Minimum Spanning Trees

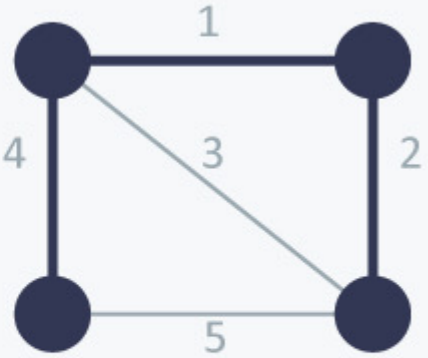


Undirected Graph



Spanning Tree

Cost = 11(=4+5+2)



Minimum Spanning Tree

Cost = 7(=4+1+2)

https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/

Motivation

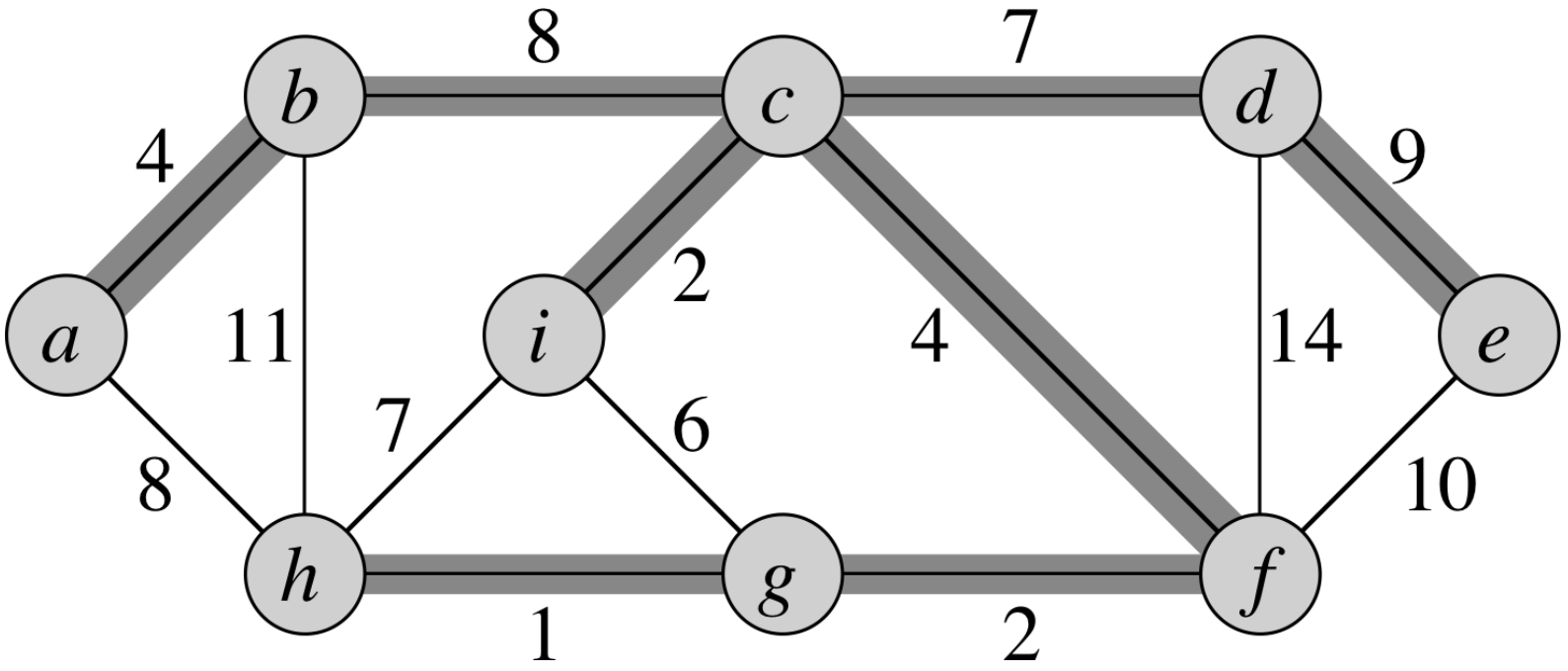
- Consider the problem of connecting a set of *n* pins using *n-1* wires (each connecting two pins) using the least amount of wire.
- This can be modeled as a graph *G = (V, E)* where *V* is the set of pins, *E* is the set of possible interconnections between pairs of pins, and for each edge *(u, v)* that belong to the set of edges E, we have a weight *w(u, v)* that specifies the cost (amount of wire needed) to connect the pins *u* and *v*.
- The goal is to find an acyclic subset of *E*, say *T*, that connects all the vertices and whose total weight *w(T)* is minimized:

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

Minimum Spanning Tree

- Since *T* is acyclic and connects all vertices, it must form a tree, which is called a spanning tree (since it "spans" the entire graph – connects all edges in *G*)
- Also since total weight of the tree *T, w(T)* is minimized we call this tree a ***minimum-weight spanning tree*** or simply a ***minimum spanning tree***

Input: A connected, undirected graph $G = (V, E)$ with the weight function $w : E \to R$



Output: A spanning tree T (connects all the vertices) of minimum weight

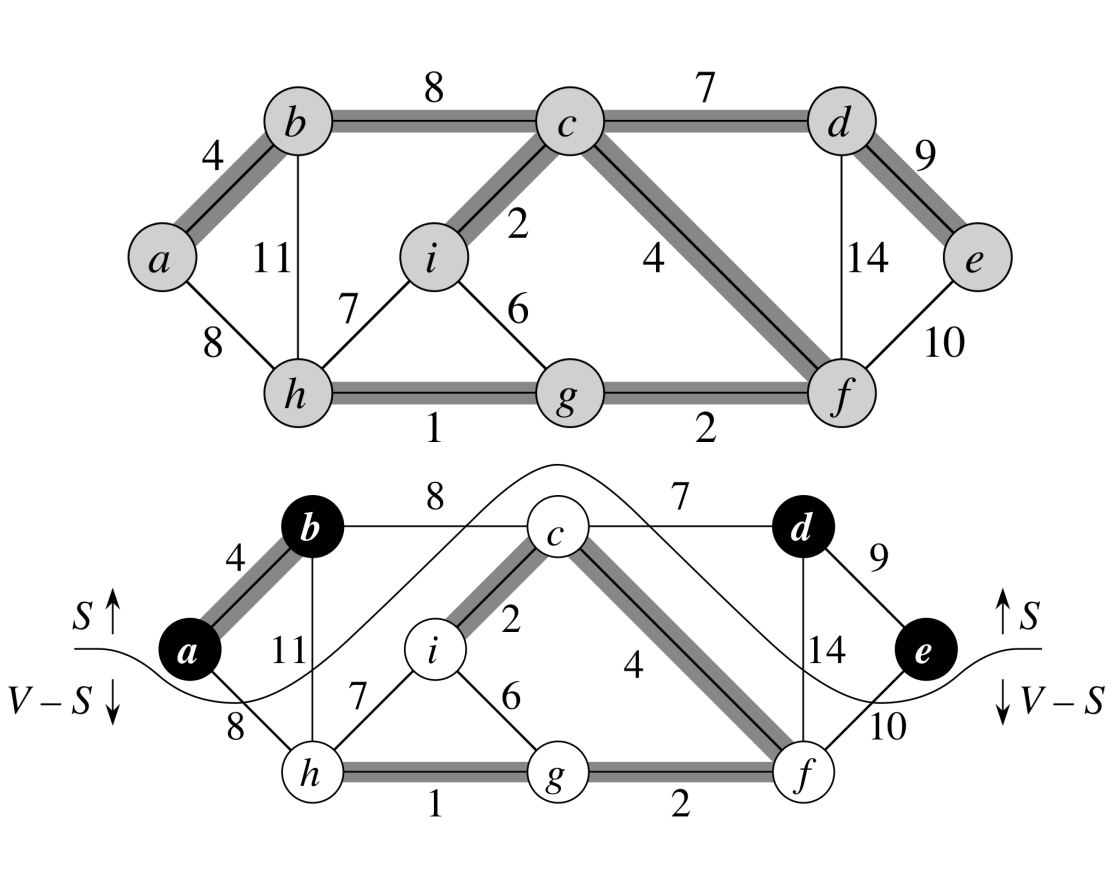
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Assumptions

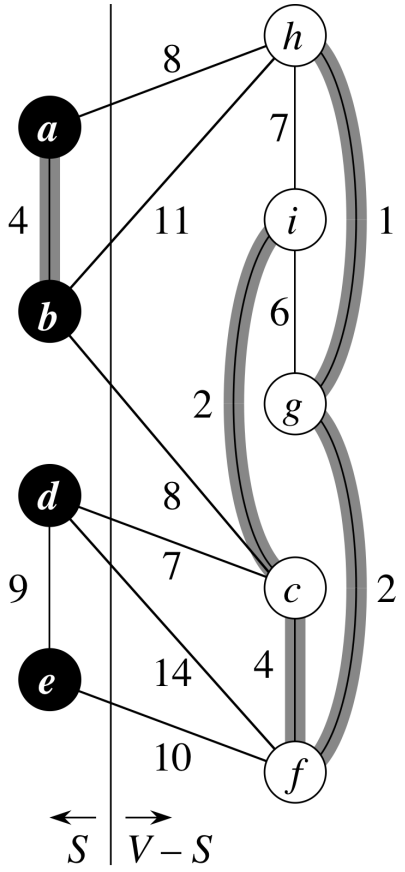
- The graph is connected – if not we can use the MST algorithms to compute the MSTs of each connected components (minimum spanning forest)
- The edge weights are not necessarily distances
- The edge weights may be zero or negative
- The edge weights are all different – if not the MST may not be unique (however, the algorithms we consider will still work)

Underlying Principles and Definitions

- A *cut* of a graph – a partition of its vertices into two disjoint sets
- A cut is said to ***respect*** a set A of edges if no edge in A crosses the cut
- A *crossing edge (light edge)* – an edge that connects a vertex in one set with a vertex in the other
- **Cut Property**: Given any cut in an edge-weighted graph (with distinct weights), the crossing edge (light edge) of minimum weight is in the MST of the graph (or that *crossing edge is a safe edge*)



(a)



(b)

(a) Black vertices are in the set S, and white vertices are in V - S. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut (S, V – S) respects A, since no edge of A crosses the cut.

(b) The same graph with the vertices in the set S on the left and the vertices in the set V - S on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

How to find MST????

- Compute each possible way and choose the minimum one???
- If we have 100 vertices how many possible MST?
- How about 1000000 vertices?
- We should compute it fast and accurate
- There are different algorithms to compute it
  - Greedy Algorithms
- …….

Greedy Algorithm

- Each step of a greedy algorithm must make one of several possible choices
- The greedy strategy advocates making the choice that is the best at the moment (local optimal choice is globally optimal)
- Such a strategy does not generally guarantee that it will always find globally optimal solutions to problems

GENERIC-MST(G, w)

$A = \emptyset$
while A is not a spanning tree
    find an edge (u, v) that is safe for A
    $A = A \cup \{(u, v)\}$
return A

Prim's Algorithm

- Prim's Algorithm uses Greedy Approach
- (Remove loops, self links and parallel edges)→ *optional*
- **Start with the minimum edge**
- **Always find the connected – minimum weighted edge**
- **Always check for the cycle**

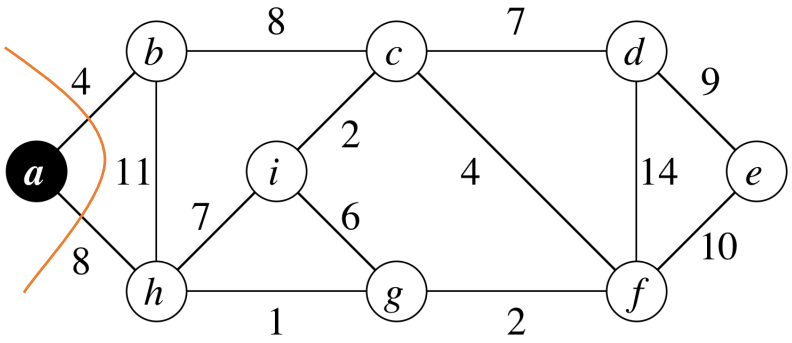
```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

1

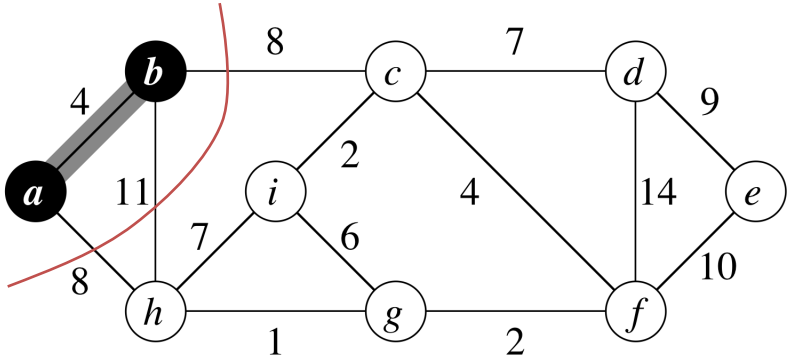
At the end, MST $A = \{(v, v.\pi) : v \in V - \{r\}\}$

1 – involves implicit DECREASE_KEY heap operation on min-heap

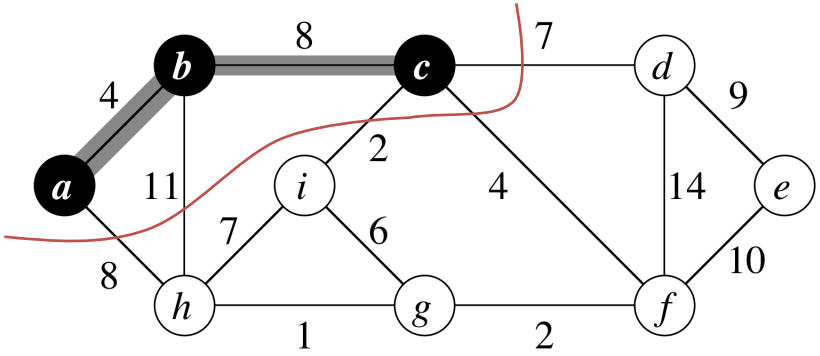
(a)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

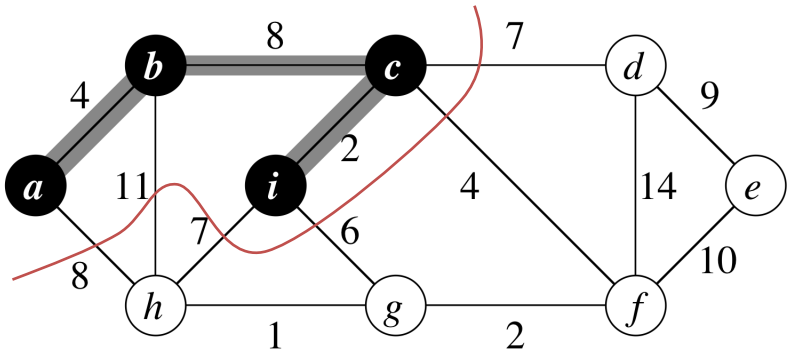
(b)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

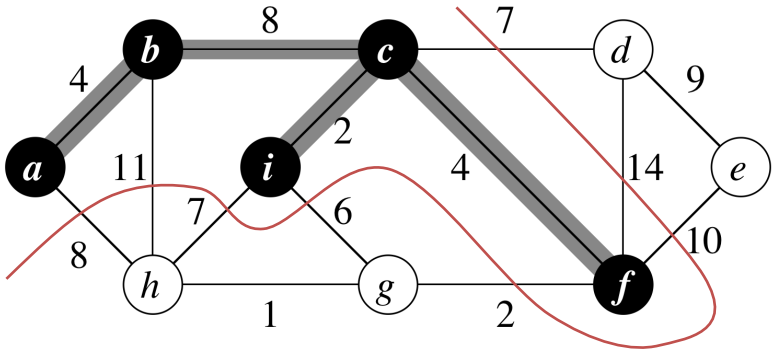
(c)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

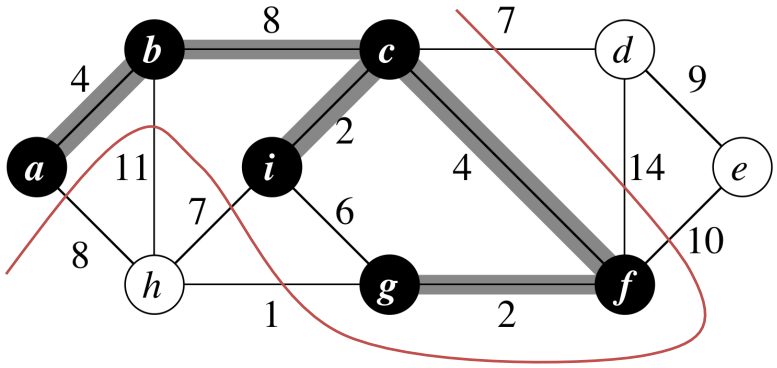
(d)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

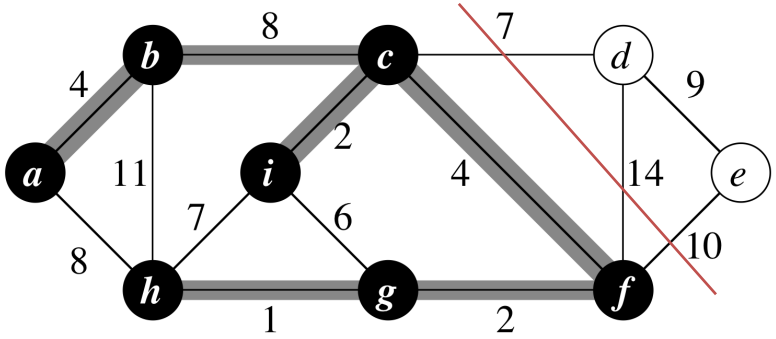
(e)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

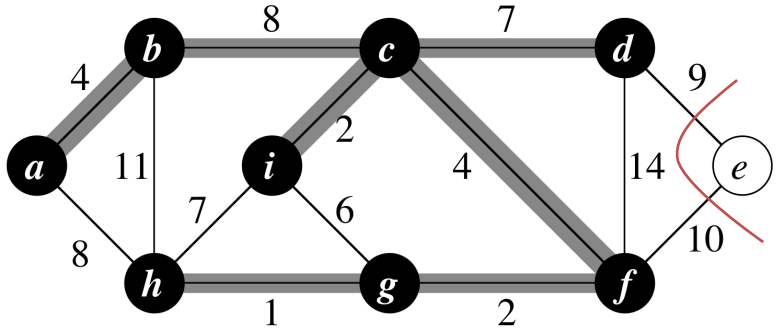
(f)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

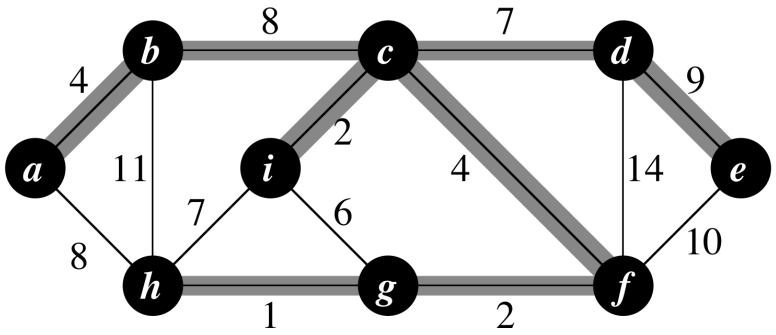
(g)

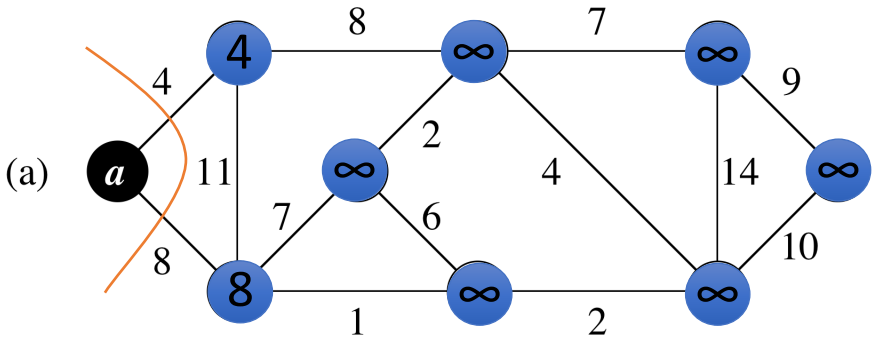

a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

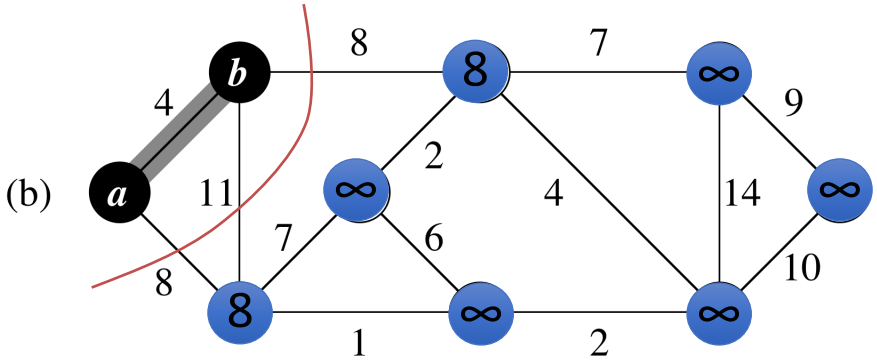
(h)

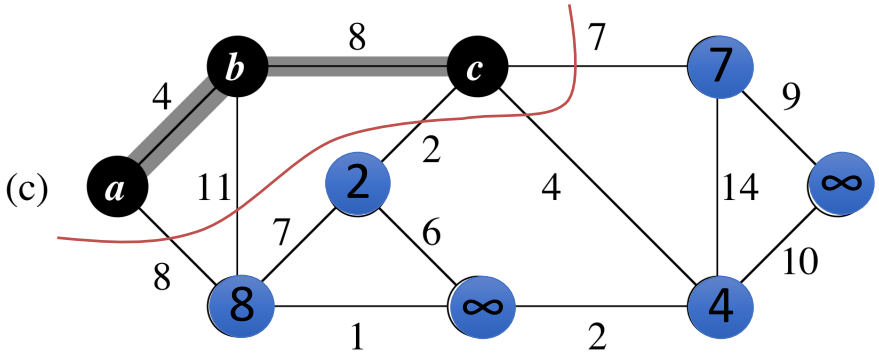

a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

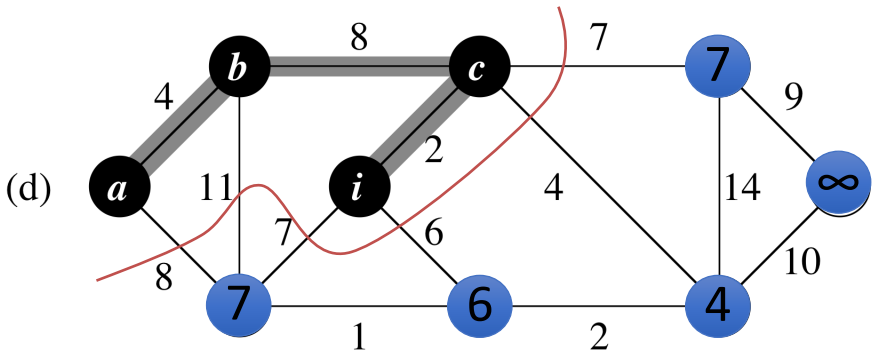
(i)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

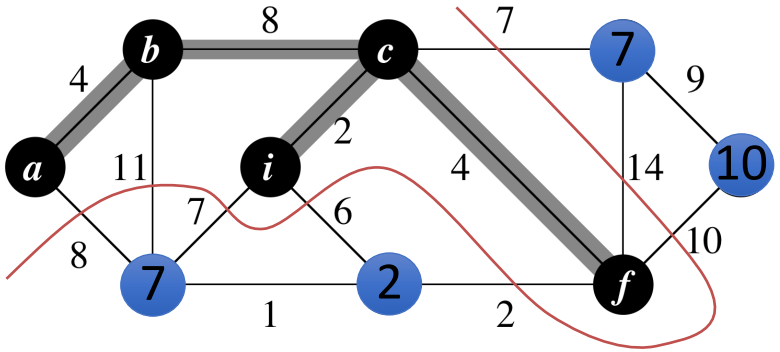

(a)
a
b
c
d
i
4
8
7
9
2
11
14
10
6
1
2
∞


(b)
a
b
8
7
9
4
11
2
14
∞
6
10
1
2

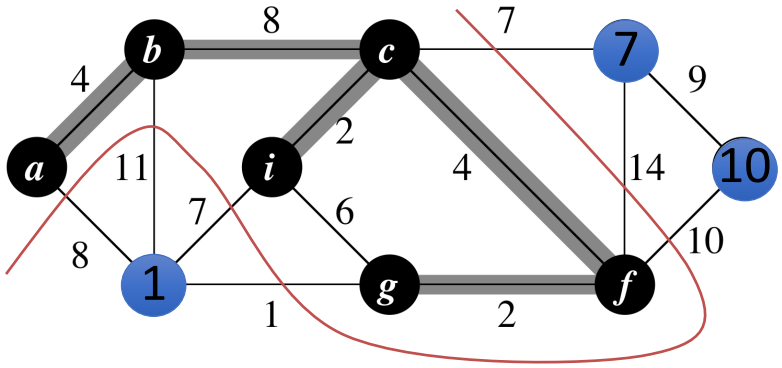

(c)
a
b
c
7
4
8
7
9
2
11
2
14
∞
6
10
1
4


(d)
a
b
c
7
4
8
7
9
2
11
i
14
∞
6
10
7
6
4

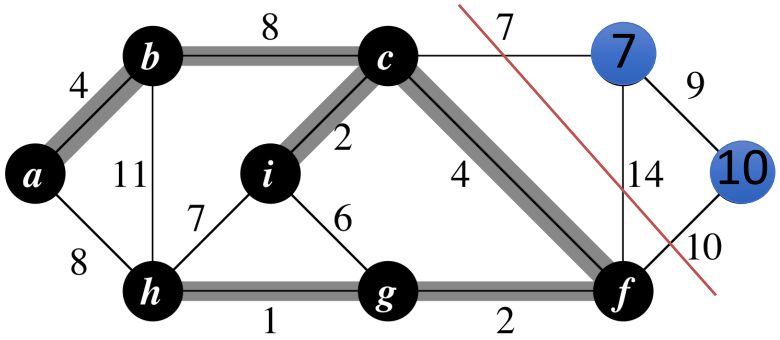
(e)


a
b
c
d
e
f
g
h
i
4
8
8
7
11
7
2
4
14
9
10
6
10
8
7
1
2
2

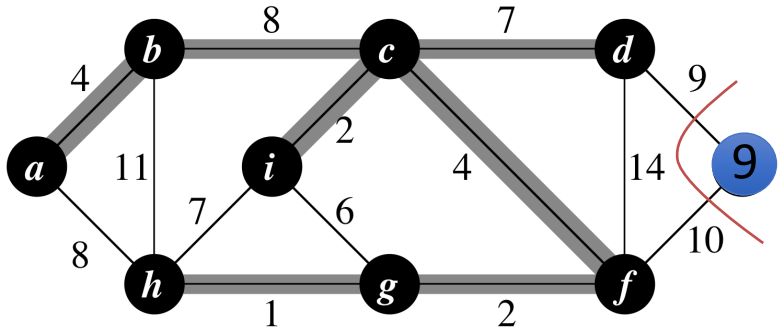
(f)


a
b
c
d
e
f
g
h
i
4
8
8
7
11
7
2
4
14
9
10
6
10
8
1
1
2

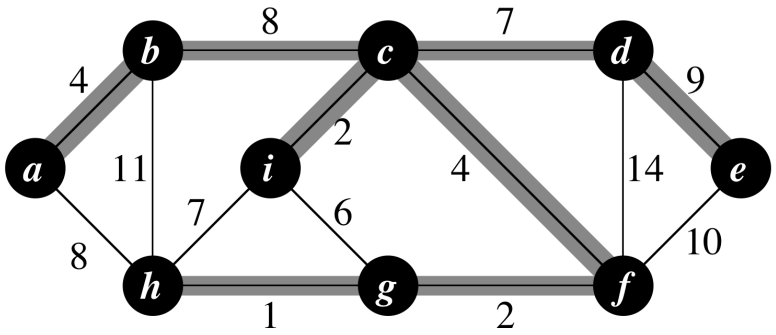
(g)


a
b
c
d
e
f
g
h
i
4
8
8
7
11
7
2
4
14
9
10
6
10
8
1
2

(h)


a
b
c
d
e
f
g
h
i
4
8
8
7
11
7
2
4
14
9
10
6
10
8
1
2

(i)


a
b
c
d
e
f
g
h
i
4
8
8
7
11
7
2
4
14
9
10
6
10
8
1
2

Exercises

Analysis

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V                          } Θ(V)
 2      u.key = ∞                             }
 3      u.π = NIL                             }
 4  r.key = 0
 5  Q = G.V                                   } Θ(V)
 6  while Q ≠ ∅                               } |V| times
 7      u = EXTRACT-MIN(Q)                    } ?
 8      for each v ∈ G.Adj[u]                 }
 9          if v ∈ Q and w(u, v) < v.key       } Σ_{u∈V} degree(u)
10              v.π = u                       }
11              v.key = w(u, v)               }
                        ?
```

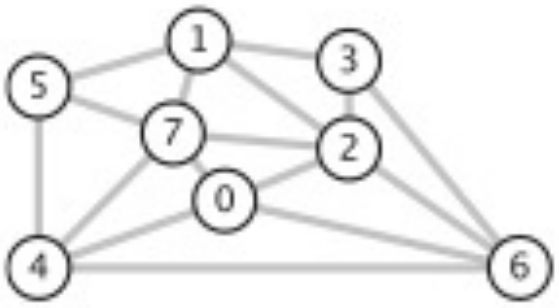
Analysis

Priority Queue Implementation	$T_{extract\_min}$	$T_{decrease\_key}$	Total
array	O(V)	O(1)	$O(V^2)$
binary heap	O(lg V)	O(lg V)	O(E lg V)

$$\text{Total Time} = O(V * T_{extract\_min} + E * T_{decrease\_key})$$

Prelab Assignment

Consider the following edge-weighted graph with 8 vertices and 16 edges (input consists of {u,v,w(u,v)}). Compute the minimum spanning tree for this graph using Prim's algorithm and Kruskal's algorithm.



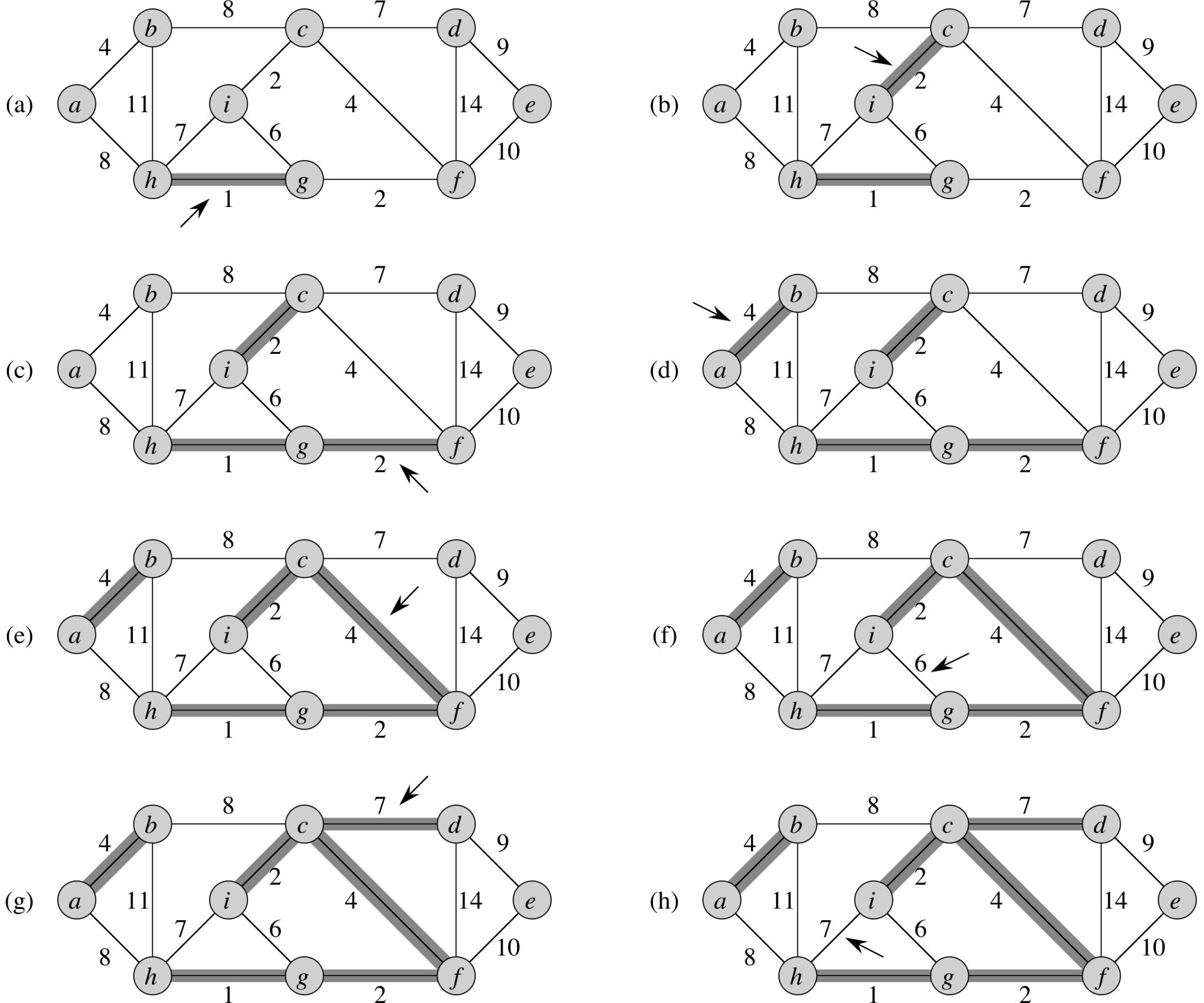
4 5 0.35
4 7 0.37
5 7 0.28
0 7 0.16
1 5 0.32
0 4 0.38
2 3 0.17
1 7 0.19
0 2 0.26
1 2 0.36
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

Kruskal's Algorithm

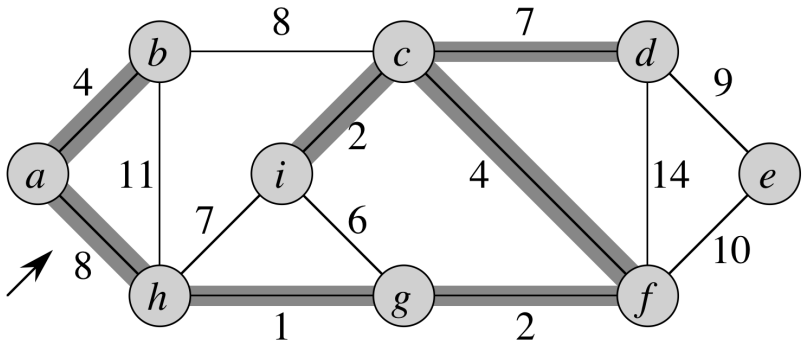
Key Idea:

- pick an edge with the lowest weight
- add that edge to the MST if it does not create a cycle
- note that while Prim's tries to keep a connected graph as it proceeds, Kruskal's forms a forest of trees that evolves gradually into a single tree, the MST

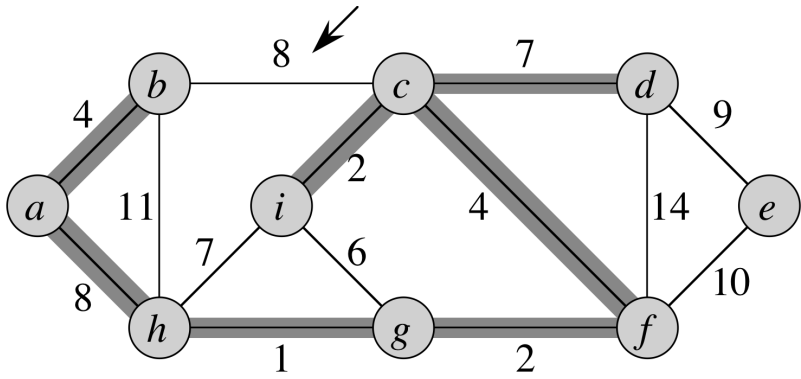
```
Initialize A = ⦸
for each edge (u,v) with the lowest weight in G.E
     if A U {(u,v)} has no cycles
        add (u,v) to A
return A
```


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
1
2

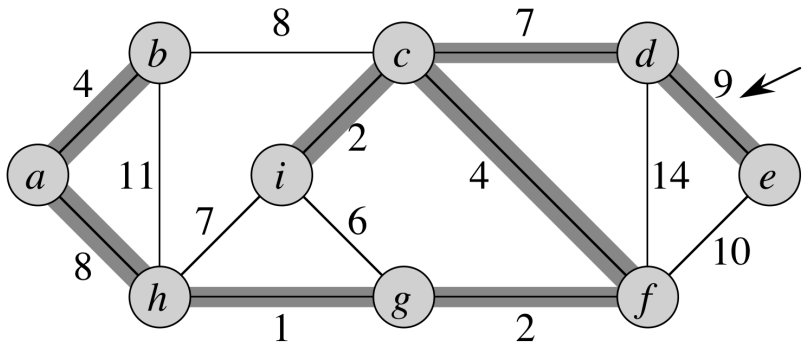
(i)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

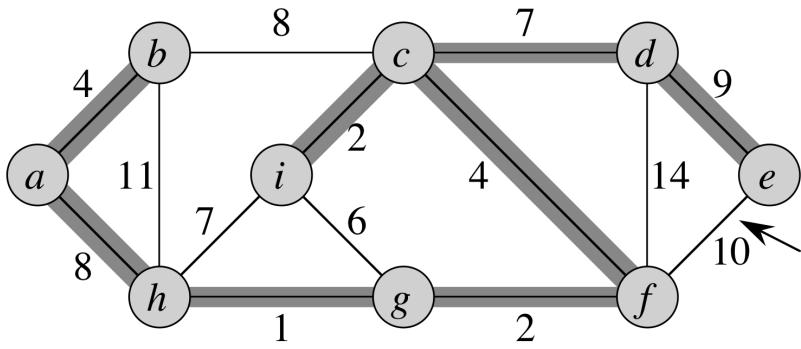
(j)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

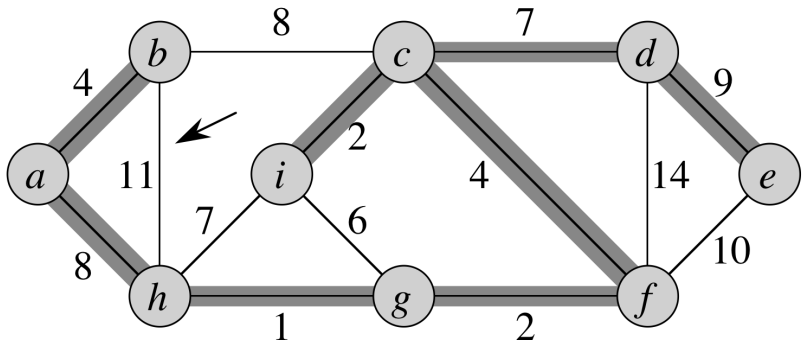
(k)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

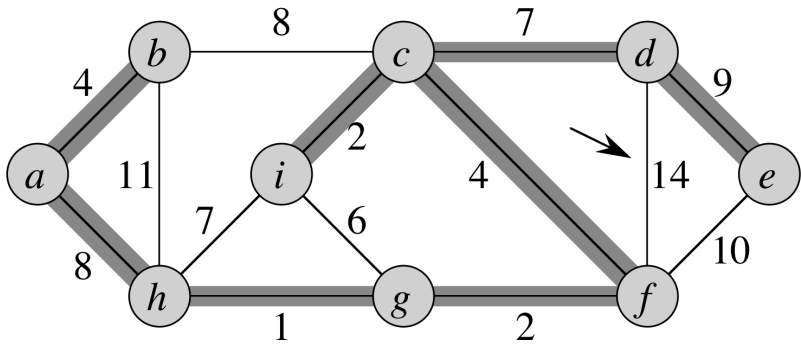
(l)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

(m)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

(n)


a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

Implementing Kruskal's Algorithm

- How do we implement Kruskal's algorithm?
  - How do we keep track of the emerging forest of trees?
  - How do we check cycles?
- What is the best data structure to implement Kruskal's algorithm?

```
Initialize A = ⦸
for each edge (u,v) with the lowest weight in G.E
    if A U {(u,v)} has no cycles
        add (u,v) to A
return A
```

Kruskal's Algorithm

KRUSKAL(G, w)

$A = \emptyset$

for each vertex $v \in G.V$

    MAKE-SET(v)

sort the edges of $G.E$ into nondecreasing order by weight w

for each (u, v) taken from the sorted list

    if FIND-SET$(u) \neq$ FIND-SET(v)

        $A = A \cup \{(u, v)\}$

        UNION(u, v)

return A

Time Complexity of Kruskal's Algorithm

- **O(ElogE) or O(ElogV).**
- Sorting of edges takes O(ELogE) time.
- After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time.
- So overall complexity is O(ELogE + ELogV) time.
- The value of E can be atmost $O(V^2)$, so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

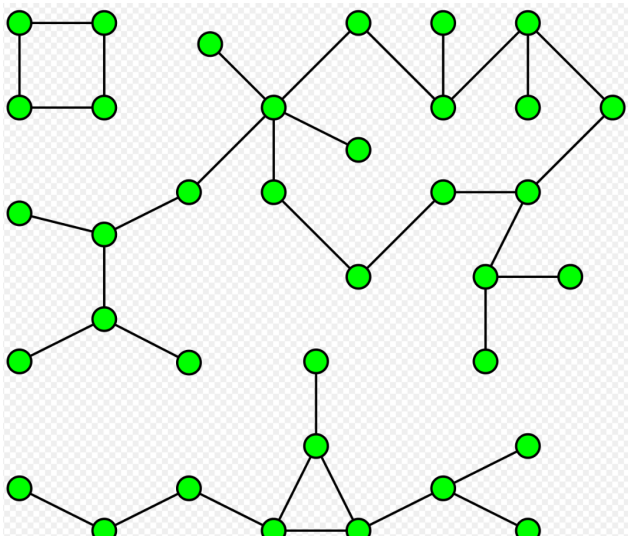
https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

Variations on MST

- The MST algorithms has several interesting properties that help solve several closely related problems;
  - Maximum Spanning Trees
  - Minimum Product Spanning Trees
  - Minimum Bottleneck Spanning Tree
  - Steiner Tree
  - Low-degree Spanning Tree
  - ….
  - …..

Minimum Spanning Forest

- It is a generalization of MST for unconnected graphs.
- For every component of the graph, we will take its MST and the resulting collection is a minimum spanning forest.
- It is union of the MSTs for the connected components



Picture: https://en.wikipedia.org/wiki/Pseudoforest#/media/File:Pseudoforest.svg