Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

Student Copy

AIEP Sendoff HS Session 7

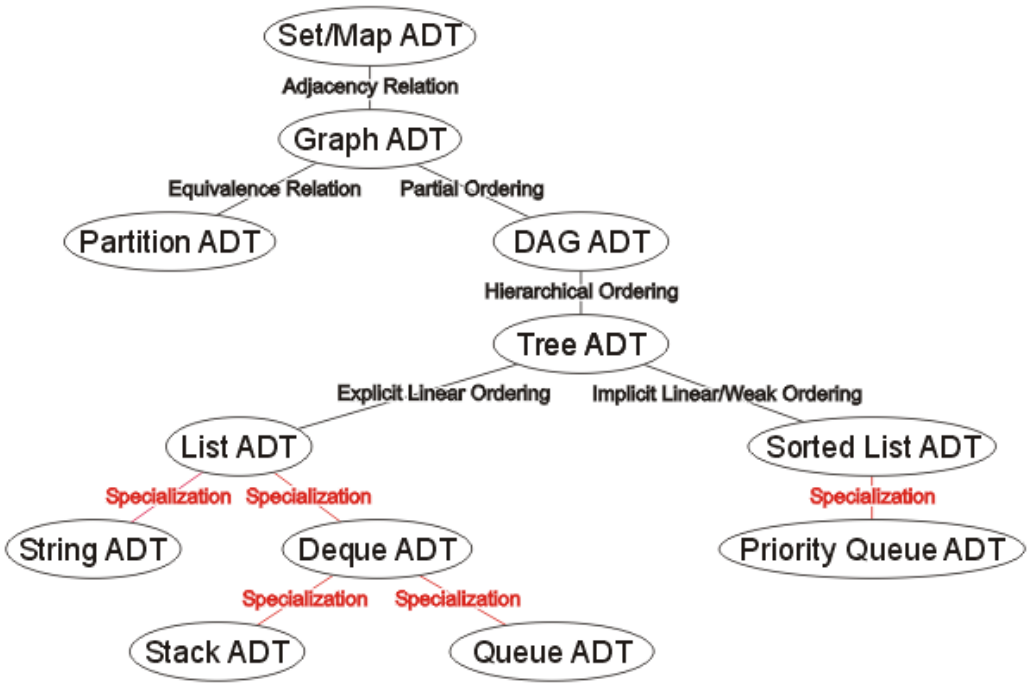
INTRODUCTION TO DATA STRUCTURES

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships"

~Linus Torvalds

As we progress with programming, we find that using only atomic data types (`int`, `float`, `char`, `bool`, etc.) are inadequate to accomplish more complex tasks. We need "containers" for our data; more importantly, we need proper containers that support the necessary and efficient operations to carry out our goal. These are properly termed as data structures.

Before we dive into some of the most common data structures in computer science, we first introduce the concept of an abstract data type (ADT). An ADT is essentially a "model" of a data structure concerned with operations and relationships rather than implementation.



https://ece.uwaterloo.ca/~dwharder/aads/Abstract_data_types/

Prepared by Mark Edward M. Gonzales

For example, we might want to have a data structure where the first to be retrieved is the first element stored; this describes the queue ADT. The actual implementation — whether it is via an array or a linked list — depends on the programmer (and the language). The implementation might be good, it might be bad, or it might be so-so. But it is not the chief concern. As long as the said scheme is satisfied, then it follows the queue ADT.

STACKS & QUEUES

In Session 3 of the training program, we have already mentioned the notions of stacks and queues. Analogous to their real-world counterparts, they are ADTs that differ only on their retrieval and insertion scheme:

1. Stacks follow a last-in, first-out (LIFO) scheme *(magkakapatong)*.
2. Queues follow a first-in, first-out (FIFO) scheme *(pila)*.

In computer science, stacks are used in the execution of recursive functions while queues are used in the scheduling of equal-priority jobs (for example, documents sent to a printer) following the order of their arrival.

Stack Operations

The following table shows the two fundamental operations related to **stacks** and how they are implemented in Python via lists:

Operation	Description	Python
`push(value)`	Inserts a value into the stack (top of the stack)	`append(value)`
`pop()`	Removes a value from the stack (top of the stack)	`pop()`

It is, indeed, possible to add some more operations, such as checking if the stack is empty, checking if it is already full, or just "peeking" at the value at the top. However, they must not violate LIFO and another core property: only the top can be accessed. Similar to how getting a plate sandwiched in a tower of plates will cause a collapse, it is incorrect to "get the middle element of a stack" or to "delete the bottommost value."

To check your understanding of the stack operations, it may be helpful to study this trace:

Code	Stack
`stack = []`	
`stack.append('a')`	a
`stack.append('m')`	a m
`stack.append('s')`	a m s
`stack.pop()`	a m
`stack.pop()`	a

Queue Operations

Although it is possible to use a list to implement a queue, the Python documentation warns against this and notes that "lists are not efficient for this purpose" since removing the front of the queue will incur the cost of shifting all the other elements. To avoid this, we will be using **`collections.deque`** (which stands for "double-ended queue").

The following table shows the two fundamental operations related to **queues** and how they are implemented in Python via `collections.deque`:

Operation	Description	Python
`enqueue(value)`	Inserts a value into the queue (front of the queue)	`append(value)`
`dequeue()`	Removes a value from the queue (back of the queue)	`popleft()`

It is, indeed, possible to add some more operations, such as checking if the queue is empty, checking if it is already full, or just "peeking" at the value at the front. However, they must not violate FIFO and another core property: only the front (head) and back (tail) can be accessed. Similar to how cutting in line *(singit)* is frowned upon, it is incorrect to "get the middle element of a queue" or to "delete the value at the back/tail."

To check your understanding of the queue operations, it may be helpful to study this trace:

Code	Stack
`queue = collections.deque()`	
`queue.append('a')`	a
`queue.append('m')`	a m
`queue.append('s')`	a m s
`queue.popleft()`	m s
`queue.popleft()`	s

Sample Usage (Postfix Evaluation)

Remember the infamous math "problem": $6 \div 2(1+2)$. For better or for worse, it exposed an intrinsic weakness of our current way of writing mathematical expressions (also known as infix notation): learning grouping and operator precedence is a bit of a chore.

A workaround is to employ the so-called reverse Polish notation. It is also referred to as postfix notation since operators are written after operands. Study the following examples in order to get an idea as to how postfix notation discards the need for parentheses and for the operator hierarchy.

Infix Notation	Postfix Notation	Value
1 * 2 + 3	1 2 * 3 +	5
1 + 2 * 3	1 2 3 * +	7
(1 + 2) * 3	1 2 + 3 *	9
30 / 3 + 2 * 3 - 1	30 3 / 2 3 * + 1 -	15
30 / 3 + 2 * (3 - 1)	30 3 / 2 3 1 - * +	14
30 / (3 + 2) * 3 - 1	30 3 2 + / 3 * 1 -	17

How then can we write a program that evaluates an expression written in postfix notation? The idea is to read the expression token by token, pushing the operands into the stack. Once an operator is encountered, the appropriate number of operands is popped, the result is evaluated, then this result gets pushed into the stack. This process is repeated until the entire expression is read, at which point the only number left inside the stack is the final answer.

The code belows shows a Python implementation of this algorithm. For simplicity, we make the following assumptions:

a. All operands consist of a single digit only, and there are only two operators: + and -.
b. There are no spaces in between the tokens.

```
def is_operator(token):
    operators = ['+', '-']
    return token in operators

def evaluate_postfix(exp):
    operand_stack = []
    for token in exp:
        if not is_operator(token):
            operand_stack.append(token)
        else:
            if token == '+':
                operand1 = int(operand_stack.pop())
                operand2 = int(operand_stack.pop())
                result = operand2 + operand1
                operand_stack.append(result)
            elif token == '-':
                operand1 = int(operand_stack.pop())
                operand2 = int(operand_stack.pop())
                result = operand2 - operand1
                operand_stack.append(result)
    return int(operand_stack.pop())
```

Another usage of stacks is related to checking whether the parentheses in a given string form matching pairs — a programming challenge that you have encountered in session 3. We will defer the sample usage of queues to the discussion of graph traversal.

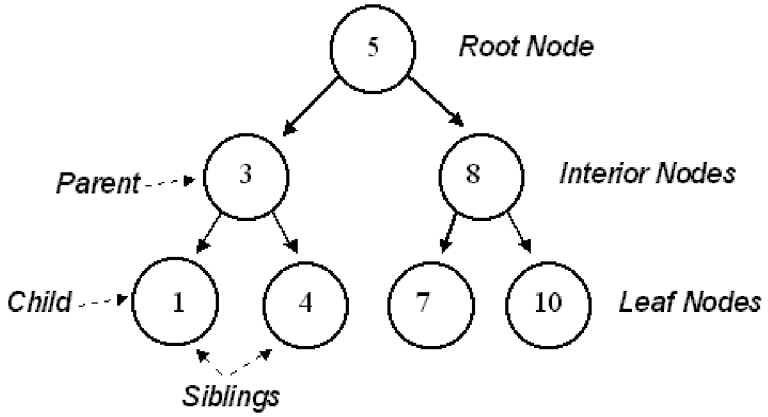
TREES

A tree is a graph (a collection of nodes and edges) that satisfies three properties:

1. **Acyclic** - There are no cycles.
2. **Undirected** - All the edges are bidirectional or two-way.
3. **Connected** - There is no isolated node.

Note that a tree is a recursive data structure; it is built by combining subtrees. Therefore, in writing algorithms for trees, we ought to take advantage of this property.

In computer science, they appear quite naturally in the context of maintaining hierarchical structures and relationships. For example, they are used by file management systems to keep track of folders (directories) and their subfolders.

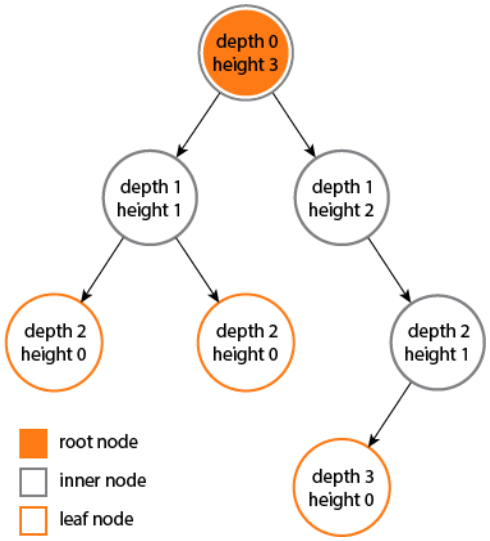


https://cs.stackexchange.com/questions/75416/how-is-data-in-a-tree-stored-in-memory

Tree Terminologies

1. **Root** - the topmost node ("origin") of the tree
2. **Interior Node** - a node that has a child
3. **Leaf** - a node that does not have any children

4. **Degree of a Node** - number of child(ren) of a node
    - The degree of a leaf is always 0
5. **Depth of a Node** - length of the path from the node to the root
    - The depth of the root is always 0
6. **Height of a Node** - length of the longest path from the node to a leaf
    - The height of a leaf is always 0
7. **Height of a Tree** - height of the root



https://stackoverflow.com/questions/29326512/what-is-the-difference-between-the-height-of-a-tree-and-depth-of-a-tree

Tree Traversal

1. **Breadth-First Search (BFS)**
    - Starts at the root
    - All the nodes at a certain depth are traversed (visited) before moving to the next depth.
    - For this reason, it is also referred to as level-order traversal.

2. **Depth-First Search (DFS)**
    - Starts at the root
    - Visits nodes (downward) into as much depth as possible before backtracking into an unvisited node.

Before we proceed to other traversal schemes, we first show how a node of a binary tree (a tree where each node has at most two children) is implemented in Python:

```
class BinaryTreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

For simplicity, we assume that visiting a node is printing the value it stores. Try to trace the given recursive functions to understand how the remaining tree traversals are done. Take special note of the position of the highlighted line of code (the "visit").

3. Preorder Traversal

```
def preorder(root):
    if root:
        print(root.val)
        preorder(root.left)
        preorder(root.right)
```

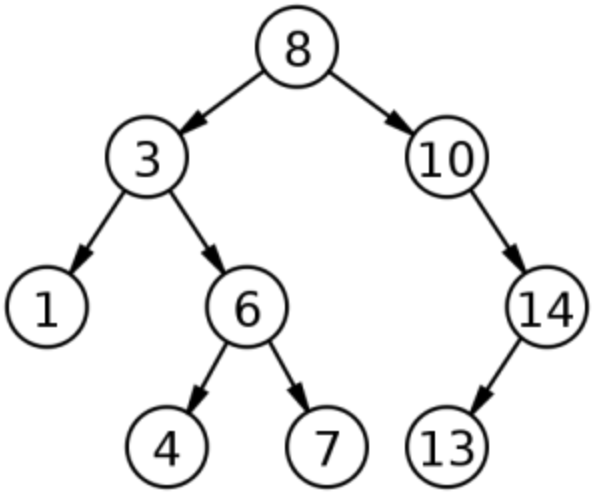
4. Inorder Traversal

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)
```

5. Postorder Traversal

```
def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val)
```

To check your understanding of the different tree traversal schemes, it may be helpful to study the example below:

Tree	Traversal
 https://humanwhocodes.com/blog/2009/06/16/computer-science-in-javascript-binary-search-tree-part-2/	**Breadth-First Search** 8 → 3 → 10 → 1 → 6 → 14 → 4 → 7 → 13
	Depth-First Search 8 → 3 → 1 → 6 → 4 → 7 → 10 → 14 → 13
	Preorder Traversal 8 → 3 → 1 → 6 → 4 → 7 → 10 → 14 → 13
	Inorder Traversal 1 → 3 → 4 → 6 → 7 → 8 → 10 → 13 → 14
	Postorder Traversal 1 → 4 → 7 → 6 → 3 → 13 → 14 → 10 → 8

Binary Search Tree

A binary search tree (BST) is a tree that obeys the following properties:

1. The key in any node is greater than all the keys in the node's left subtree.
2. The key in any node is less than all the keys in the node's right subtree.

The tree that we traversed earlier is an example of a BST. Besides the tree traversals, BSTs support other operations such as insertion, searching, deletion, getting the predecessor and successor, and getting the maximum and minimum keys. In this discussion, we will only focus on two: insertion and searching.

Focus on how these functions take advantage of both the recursive property of trees and the BST requirement. An important thing to emphasize is that we must always ascertain that the root is not `None` (or `NULL/NIL` in other programming languages). Otherwise, we will run into the trap of accessing an illegal memory address.

A. Insertion

```
def insert_bst(root, key):
    if root is None:
        root = BinaryTreeNode(key)
    elif root.val > key:
        root.left = insert_bst(root.left, key)
    elif root.val < key:
        root.right = insert_bst(root.right, key)
    return root
```

B. Searching

```
def search_bst(root, key):
    if root:
        if root.val == key:
            return root
        elif root.val > key:
            return search_bst(root.left, key)
        elif root.val < key:
            return search_bst(root.right, key)
    return None
```

The advantage of the unique structure of BST becomes clear when we trace the insertion and searching operations. For instance, searching for the node storing the value 4 in our BST involves only a total of 4 comparisons (8 → 3 → 6 → 4). Inserting 15 involves only 3 comparisons (8 → 10 → 14). On average, both insertion and searching run in $O(\log n)$ time. However, at the worst case (see NOT SO FAST! #2) , they still degrade to $O(n)$ time.

GRAPHS

A graph is a collection of vertices (or nodes) and edges. A tree is a special type of graph. A group of trees, aptly called a forest, is also a graph; note that the graph ADT is very broad and encompassing that it does not even require all the vertices to be connected.

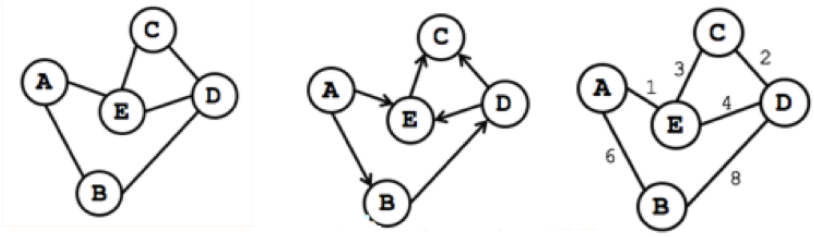
Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

Graph Terminologies

1. **Undirected Graph**
   - All edges are bidirectional or two-way.
   - An example is the graph representing Facebook friends. If Jack is a friend of Jill, then Jill is also a friend of Jack.

2. **Directed Graph**
   - There is at least one undirectional or one-way edge.
   - An example is the graph representing Twitter following-followers. If Jill follows Jack, then it does not mean that Jack follows Jill.

3. **Weighted Graph**
   - Each edge is associated with a numerical value called weight.
   - An example is a graph representing a trade route. The distances between stops can be taken as weights.
   - The weighted graph shown in the figure is a weighted undirected graph.



(A) Undirected Graph (B) Directed Graph (C) Weighted Graph

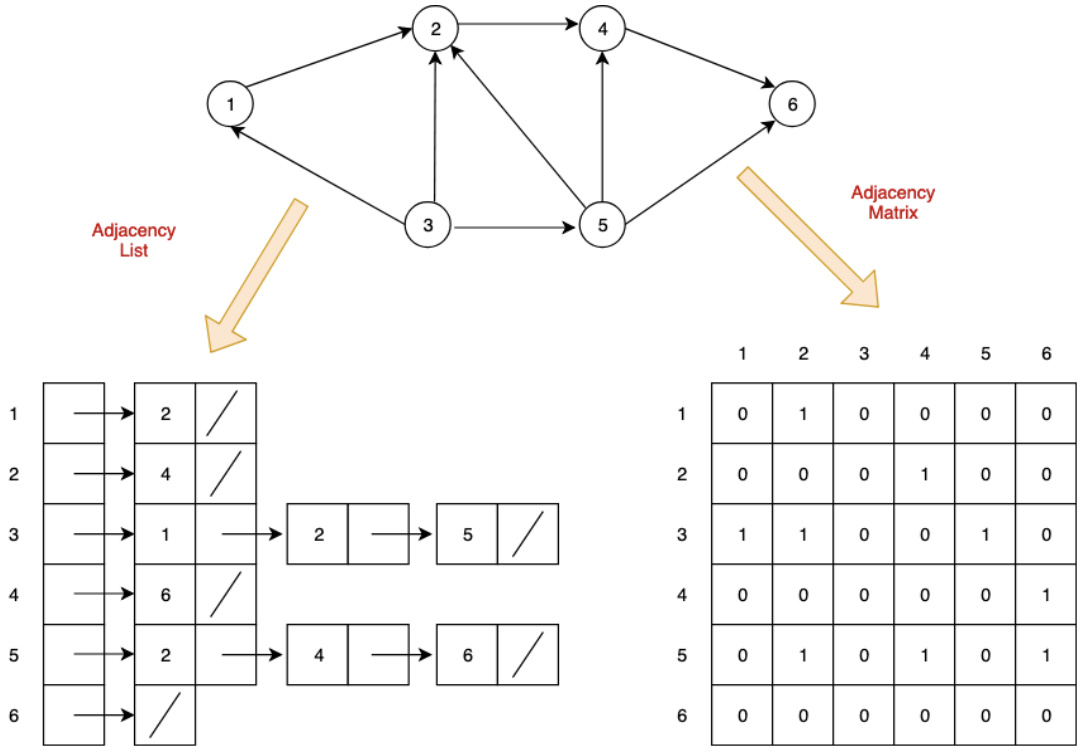
4. **Degree of a Vertex**
   - Number of edges connected to a vertex
   - Distinguished into two in the context of directed graphs:
     a. **Outdegree**
        - Number of edges "leaving" a vertex in a directed graph
        - In the directed graph shown, the outdegree of vertex D is 2.
     b. **Indegree**
        - Number of edges "entering" a vertex in a directed graph
        - In the directed graph shown, the indegree of vertex D is 1.

Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

Graph Representation

1. Adjacency List
   - A list of lists (usually an array of linked lists in C-style languages)
     - In Python, the implementation recommended by its creator Guido van Rossum is to use a dictionary (hash table) of lists.
   - Each list corresponds to a vertex in the graph and contains a list of vertices connected to the said vertex by an edge "leaving" it.
   - By the Handshaking Lemma, its space complexity is $O(V+E)$.

2. Adjacency Matrix
   - A square matrix (a two-dimensional array in C-style languages) where a cell is set to 1 if the two vertices are connected or 0, otherwise
   - Its space complexity is $O(V^2)$, making it a less desirable choice compared to an adjacency list, especially if the graph is just sparse (has few edges only).



https://algorithmtutor.com/Data-Structures/Graph/Graph-Representation-Adjacency-List-and-Matrix/

The code below shows how to represent a directed graph using an adjacency list. We will be using **`collections.defaultdict`** to signify our intention of having a dictionary of lists.

```
class Graph:
    def __init__(self):
        self.adjacency_list = collections.defaultdict(list)

    def add_edge(self, vertex1, vertex2):
        self.adjacency_list[vertex1].append(vertex2)
```

Graph Traversal

Graph traversal is a generalization of tree traversal. When there are two possible routes, tie-breaking schemes (for example, coin toss or getting the smaller key) are used. Note that, in both implementations of BFS and DFS, assume that visiting a vertex is printing the value that it stores. We will also be having a `visited` dictionary to keep track of the vertices that have already been visited (and avoid visiting them again).

1. Breadth-First Search (BFS)
    - BFS can be implemented using a queue:

```
def bfs(self, vertex):
    visited = collections.defaultdict(lambda : False)
    queue = collections.deque()
    queue.append(vertex)
    visited[vertex] = True

    while queue:
        vertex = queue.popleft()
        print(vertex)
        for adjacent_vertex in self.adjacency_list[vertex]:
            if not visited[adjacent_vertex]:
                queue.append(adjacent_vertex)
                visited[adjacent_vertex] = True
```

2. Depth-First Search (DFS)
   - DFS naturally lends itself to recursion and backtracking:

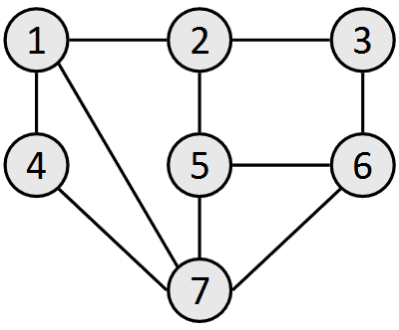
```
def dfs_helper(self, vertex, visited):
    visited[vertex] = True
    print(vertex)

    for adjacent_vertex in self.adjacency_list[vertex]:
        if not visited[adjacent_vertex]:
            self.dfs_helper(adjacent_vertex, visited)

def dfs(self):
    visited = collections.defaultdict(lambda: False)

    for vertex in self.adjacency_list.keys():
        if not visited[vertex]:
            self.dfs_helper(vertex, visited)
```

Can you identify the limitations of the implementations shown above? For what edge cases (for example, disconnected graphs) do they fail and how can we address them? To check your understanding of the two graph traversal schemes, it may be helpful to study the example below *(the tie-breaking scheme used is getting the smaller key)*:

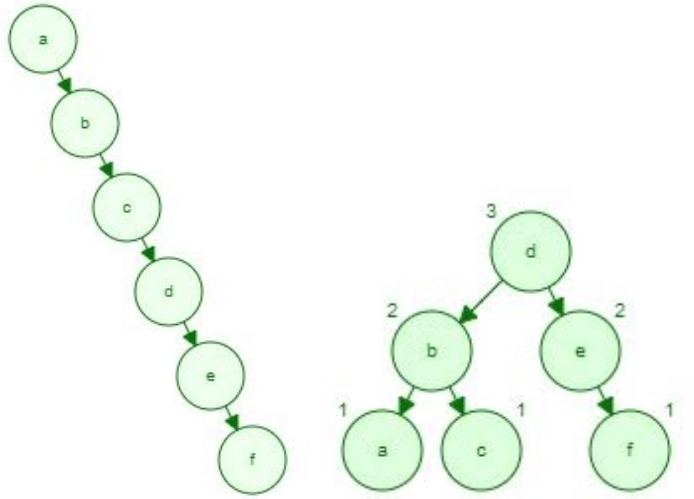
Graph	Traversal
 https://stanford.edu/class/archive/cs/cs106b/cs106b.1158/preview-graph-implementation.shtml	(Possible) Breadth-First Search 1 → 2 → 4 → 7 → 3 → 5 → 6
	(Possible) Depth-First Search 1 → 2 → 3 → 6 → 5 → 7 → 4

Graph Algorithms

The interesting nature of graphs and its manifold applications led to the development of a branch of mathematics devoted specifically to them: graph theory. Therefore, it comes as no surprise that there are also several algorithms related to graphs. For instance, we might be interested in getting the shortest path (Dijkstra's algorithm) or the minimum spanning tree (Prim's and Kruskal's algorithm). Some of these will be introduced in the next session.

NOT SO FAST!

1. The choice of data structures will affect the overall performance of the program, for better or for worse. As in real life, there is always a trade-off between space and time complexity, or a trade-off between the efficiencies of two operations. **There is no such thing as a universal or omnipotent data structure.**
2. To illustrate this point, try inserting the values 'a', 'b', 'c', 'd', 'e', and 'f' into a BST in this specified order. The resulting tree will be the one shown on the left:



Ordinary (Skewed) BST vs. AVL (Self-Balancing) Tree

This skewed BST gives the worst-case performance of BST. To resolve this, we can use **self-balancing binary search trees**, such as an **AVL tree** (shown on the right). The downside of this is the extra operation incurred to check if the tree is balanced at every insertion (and deletion), as well as to balance the tree if it is not.

Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

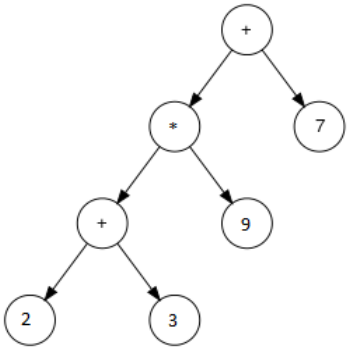
EXERCISES

Stacks & Queues

1. Write a function to convert a given expression in infix notation into an expression in postfix notation. Your function must return a queue with the tokens comprising the postfix notation as its elements. For simplicity, assume the following:
   - All tokens consist of a single character, and there are no spaces in between.
   - The only operators are (,), +, -, *, /, and ^ (exponentiation). (Caveat: Be careful with associativity: 2^3^4 is translated as 234^^.)

Trees

2. Create an expression tree given an expression in postfix notation (following the same simplifying assumptions as in #1). For example, the expression tree of 23+9*7+ is shown below. Then, by traversing this constructed tree:
   a. Write a function that prints the expression in prefix notation (operators are placed before the operands).
   b. Write a function that prints the expression in infix notation. (Caveat: The infix equivalent should respect the usual rules of grouping, associativity, and precedence.)
   c. Write a function that prints the expression in postfix notation.

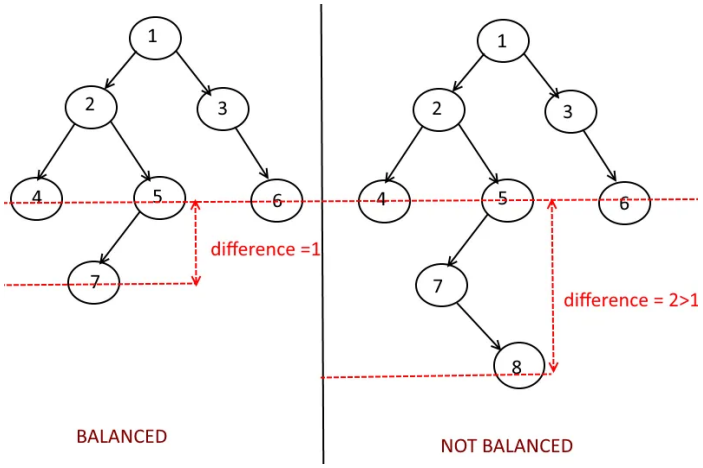


https://www.hackerrank.com/contests/data-structure-tasks-binary-tree-union-find/challenges/binary-expression-tree

3. Write a function to get the height of a given binary tree.
4. Write a function to count the number of leaves in a given binary tree.
5. Write a function to get the minimum key in a BST without doing inorder traversal.
6. Write a function to get the maximum key in a BST without doing inorder traversal.
7. Write a function to print the keys of a BST in descending order without the use of any sorting algorithm.
8. Write a function to print all the keys of a binary tree in a BFS fashion.

9. In NOT SO FAST!, we introduced the concept of an AVL tree, a self-balancing BST. To be precise, it avoids degrading into a skewed tree by performing "rotations" if the BST stops being height-balanced after an insertion or deletion.

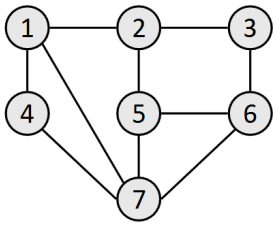
   A height-balanced tree is a tree with the property that all its subtrees satisfy this requirement: the height of their left and right subtrees differ by no more than 1. Write a function to check whether a tree is height-balanced or not.



https://algorithms.tutorialhorizon.com/find-whether-if-a-given-binary-tree-is-balanced/

Graphs

10. Write a function that uses DFS to check if a simple undirected graph has a cycle.



https://stanford.edu/class/archive/cs/cs106b/cs106b.1158/preview-graph-implementation.shtml

11. One of the privacy settings in Facebook is to restrict the audience of a post to at most "friends of friends." Write a function that uses BFS to print the "audience" of a given vertex following this scheme. For the graph shown above, the "audience" of 3 are 3, 2, 6, 1, 5, and 7.

```
-- return 0; --
```