CMPSC-132:Programming and Computation II

Department of Computer Science & Engineering
The Pennsylvania State University

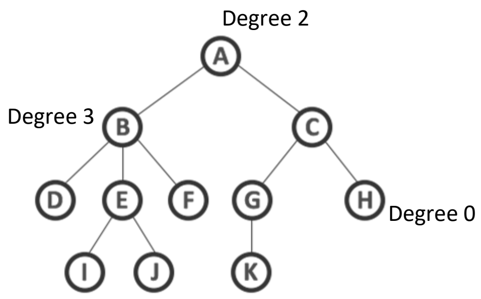
1. Other Data Structures

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular non-linear data structures.

1.1 Tree

Unlike arrays, linked lists, stack and queues, which are linear data structures, trees are hierarchical data structures that are widely used, with a root value and subtrees of children with a parent node, represented as a set of linked nodes. Unlike trees in nature, the tree data structure is upside down: the root of the tree is on top. A tree consists of nodes and its connections are called edges. The bottom nodes are also named leaf nodes or external nodes. A tree does not have a cycle. Additionally:

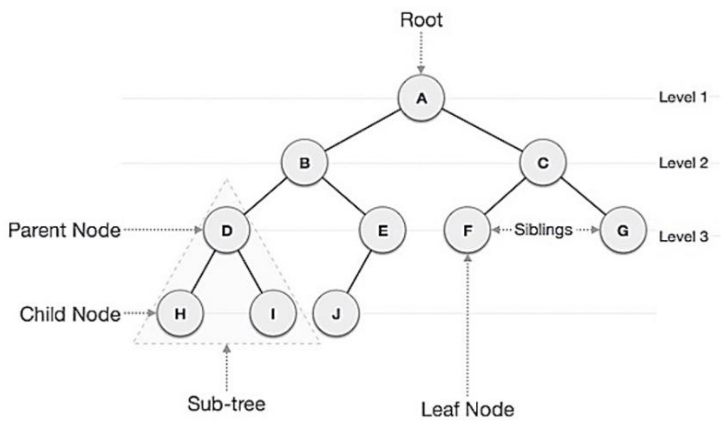
- Nodes with the same parent are called siblings
- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- The *degree* of a node is the number of children connected to the node.



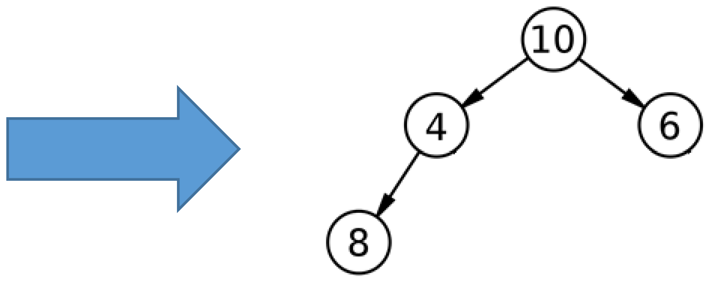
The degree of a tree is the maximum degree of node in a given tree. In the above example, node B has degree 3, the other nodes have less degree. Therefore, the degree of the tree is 3 (ternary tree)

1.1.1 Binary trees

A binary tree is a data structure where every node has at most two children (tree has degree 2). The root of a tree is on top. Since each element in a binary tree can have only 2 children, we typically name them the left and right child. Since Python does not have built-in support for trees, you need to define a class tree which has a left and right attribute.



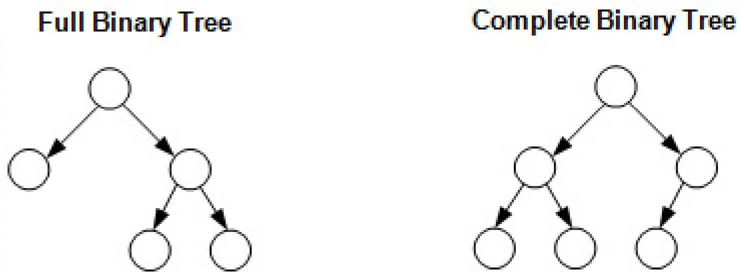
```
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.value = key

root = Node(10)
root.left = Node(4)
root.right = Node(6)
root.left.left = Node(8)

print(root.value)
print(root.left.value)
```

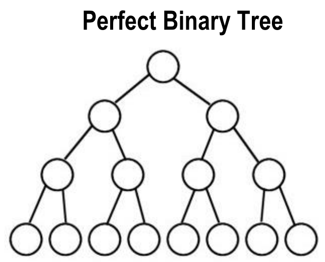


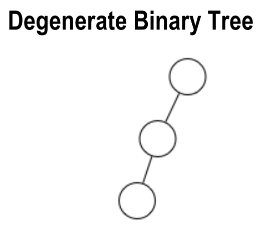
1.1.1.1Types of binary trees

The most common types of binary trees are:

- Full binary tree: a binary tree in which each node is either a leaf or possesses exactly two child nodes.
- Complete binary tree: a binary tree which is completely filled, with the possible exception of the bottom level, which is filled from left to right.
- Perfect binary tree: a binary tree in which all internal nodes have two children and all leaves are at same level.
- Degenerate binary tree: a binary tree where every internal node has one child. Such trees are performance-wise same as a linked list.







1.1.1.2Properties of binary trees

Binary trees have the following properties:

- A binary tree of n nodes has n-1 edges
- For every $k \geq 0$, there are no more than 2^k nodes in level k
- A binary tree with k levels has at most 2^{k-1} leaves
- The number of nodes on the last level is at most the sum of the number of nodes on all other levels plus 1
- A binary tree with k levels has no more than 2^k-1 node

Important terms

Below are important terms with respect to a tree:

- path: refers to the sequence of nodes along the edges of a tree
- root: the node at the top of the tree. There is only one root per tree and one path from the root node to any node
- parent: any node except the root node has one edge upward to a node called parent
- child: the node below a given node connected by its edge downward
- leaf: the node which does not have any child node
- subtree: represents the descendants of a node
- visiting: refers to checking the value of a node when control is on the node
- traversing: passing through nodes in a specific order
- levels: level of a node represents the generation of a node
- key: represents a value of a node based on which a search operation is to be carried out for a node

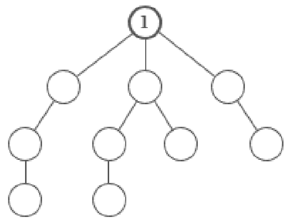
1.1.2 Traversal of binary trees

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. A tree is typically traversed in two ways:

- Breadth-First Traversal (Or Level Order Traversal)
- Depth-First Traversals:
    - ✓ Inorder Traversal (Left- Root -Right)
    - ✓ Preorder Traversal (Root-Left-Right)
    - ✓ Postorder Traversal (Left-Right- Root)

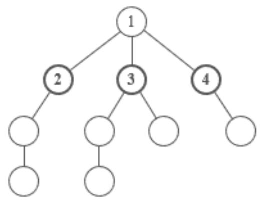
1.1.2.1Breadth-First Search

Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc. Here's a how a BFS would traverse this tree, starting with the root:

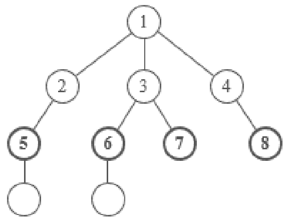


start at the root node (top of the tree)

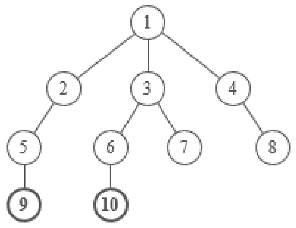
We would visit all the immediate children (all the nodes that are one step away from our starting node):



Then we would move on to all those nodes' children (all the nodes that are two steps away from our starting node):



And so on until we reach the end:



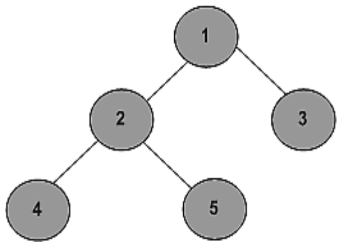
Advantages:

- BFS involves searching through a tree one level at a time. In a graph, BFS will find the shortest path between the starting point and any other reachable node.

Disadvantages:

- BFS on a binary tree generally requires more memory than a DFS.

There is only one kind of breadth-first traversal known as the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right:

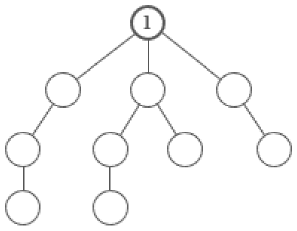


Level order traversal is:

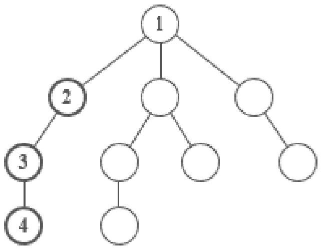
1 2 3 4 5

1.1.2.2Depth-First Search

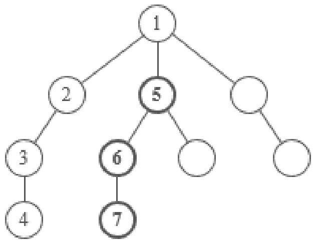
Depth-first search (DFS) is another method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one. Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one. Here's a how a DFS would traverse this tree, starting with the root:



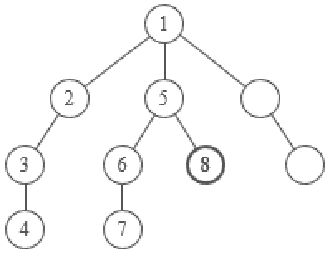
We would go down the first path we find until we hit a dead end:



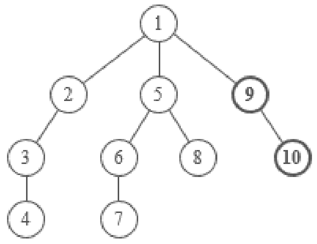
Then we would do the same thing again—go down a path until we hit a dead end:



And again:



And again:



Until we reach the end.

Advantages:

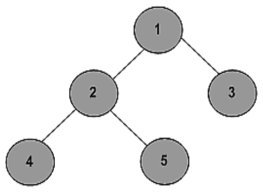
- DFS on a binary tree generally requires less memory than breadth-first.
- DFS can be easily implemented with recursion.

Disadvantages:

- DFS does not necessarily find the shortest path to a node

There are three commonly used depth-first traversals to visit all the nodes in a tree. The difference between these patterns is the order in which each node is visited. These three traversals are:

- Inorder Traversal (Left- Root -Right): we recursively do a Left- Root -Right traversal on the left subtree, visit the root node, and finally do a recursive Left-Root-Right traversal of the right subtree.
- Preorder Traversal (Root-Left-Right): we visit the root node first, then recursively do a DFS traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.
- Postorder Traversal (Left-Right- Root): we recursively do a Left-Right-Root traversal of the left subtree and the right subtree followed by a visit to the root node.



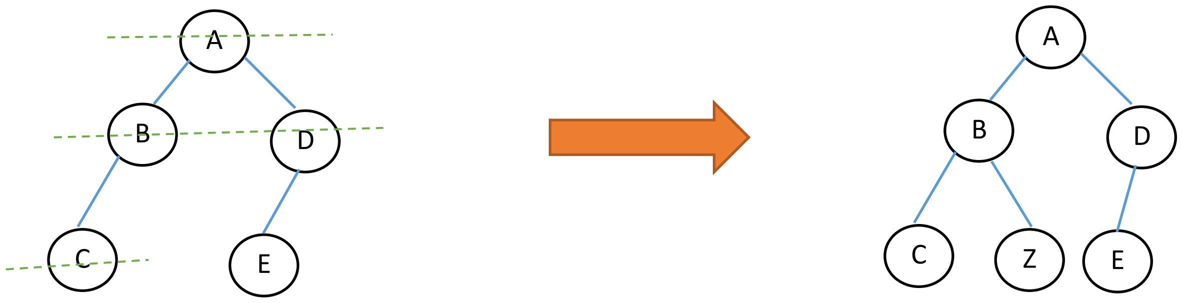
Inorder traversal: 4 2 5 1 3

Preorder traversal: 1 2 4 5 3

Postorder traversal: 4 5 2 3 1

1.1.3 Insertion in a Binary Tree in level order

Given a binary tree T and a node with a key (data), we can insert the key into T at first position available in level order. The main idea is to perform a level order traversal of the tree (using a queue is the most efficient way) and when we find a node whose left child is empty, we make the new node the left child of the node. If we find a node whose right child is empty, we make the new node key the right child. The point is to keep traversing the tree until we find a node whose either left or right is empty.



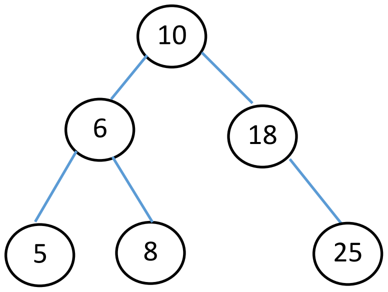
Insert node Z

1.1.4 Binary Search Trees

Binary trees are an excellent data structure for storing items of a map (key-value pair), assuming we have an order relation defined on the keys. In this context, a binary search tree (BST) is a binary tree T where each node contains one key (also known as data) with each position *p* storing a key-value pair (k,v) such that:

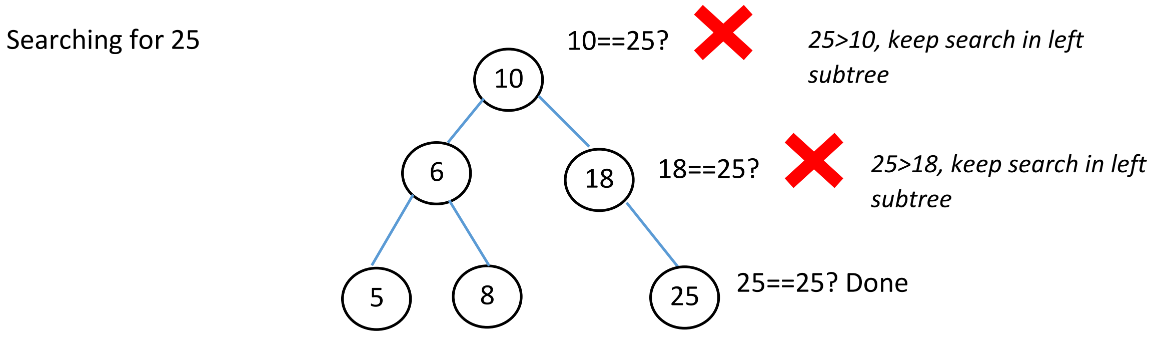
- Keys stored in the left subtree of *p* are less than k.
- Keys stored in the right subtree of *p* are greater than k.
- Duplicate keys are not allowed.
- The left and right subtrees must also be binary search trees

The basic idea behind this data structure is to have such a storing repository that provides an efficient way for data sorting, searching and retrieving data. In the tree below, all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree are > 10.



1.1.4.1 Searching

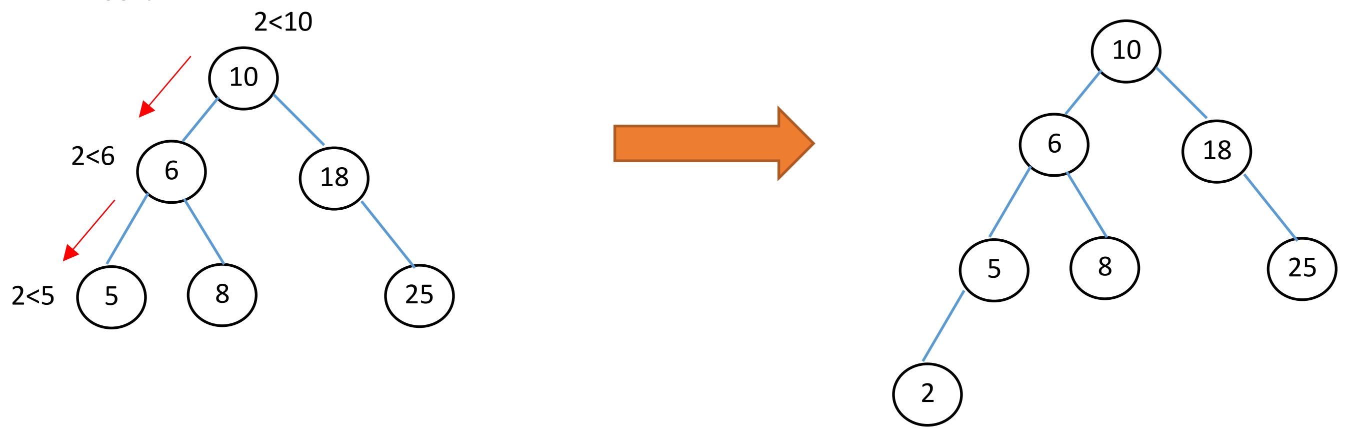
Searching in a BST always starts at the root. We compare a data stored at the root with the key we are searching for. If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise we go to the right child. The recursive structure of a BST commonly yields a recursive algorithm.


Searching for 25
10==25?
25>10, keep search in left subtree
18==25?
25>18, keep search in left subtree
25==25? Done
10
6
18
5
8
25

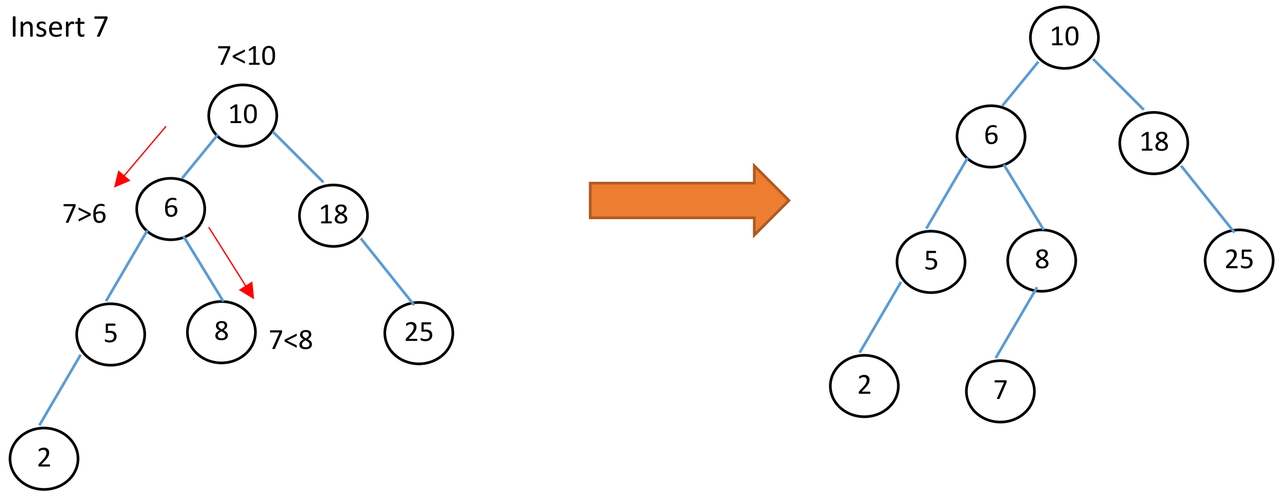
1.1.4.2 Insertion

The insertion procedure is quite similar to searching. We start at the root and recursively go down the tree searching for a location in a BST to insert a new node. If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace an empty child.

Insert 2



Insert 7

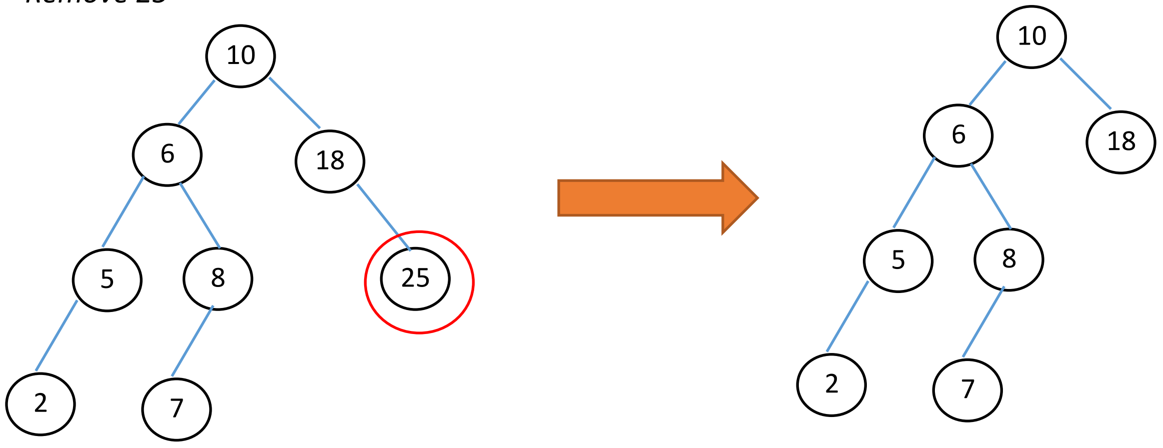


1.1.4.3 Deletion

The remove operation on binary search trees is more complicated, than inserting and searching. Basically, in can be divided into two stages, search for a node to remove, and if the node is found, perform the removal. We start the deletion at the root, but unlike searching, we should track the parent of the current node. After we found the node to be deleted, there are three possible cases:

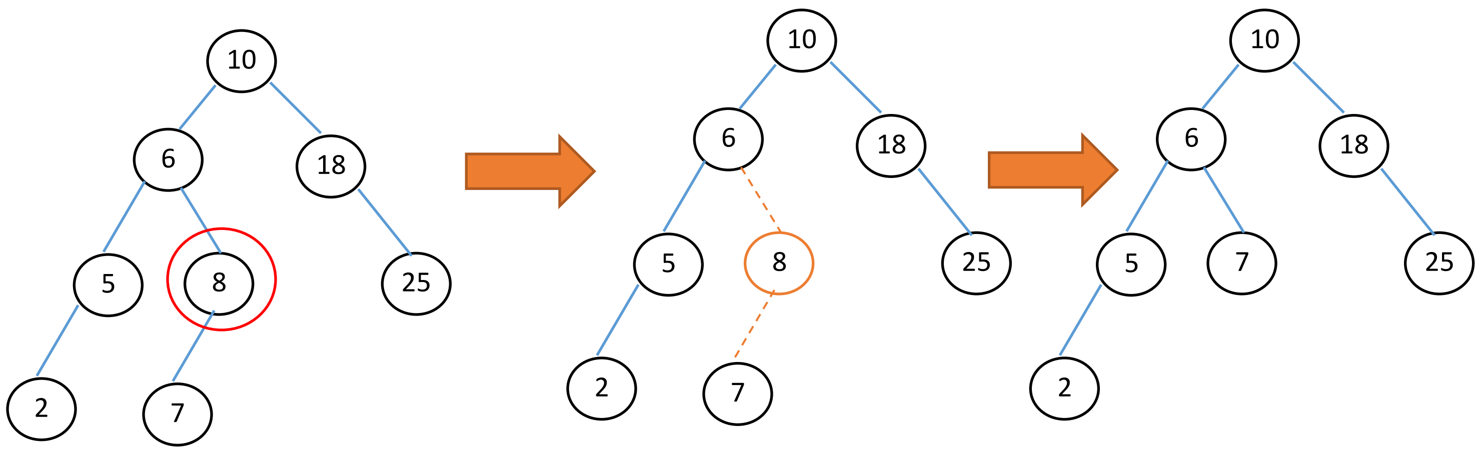
1. Node to be deleted has no children: The deletion process in this case is simple, we just set the corresponding link of the parent (edge) to None

Remove 25



2. Node to be deleted has one child: It this case, node is cut out from the tree, and we link the child directly to the parent of the removed node.

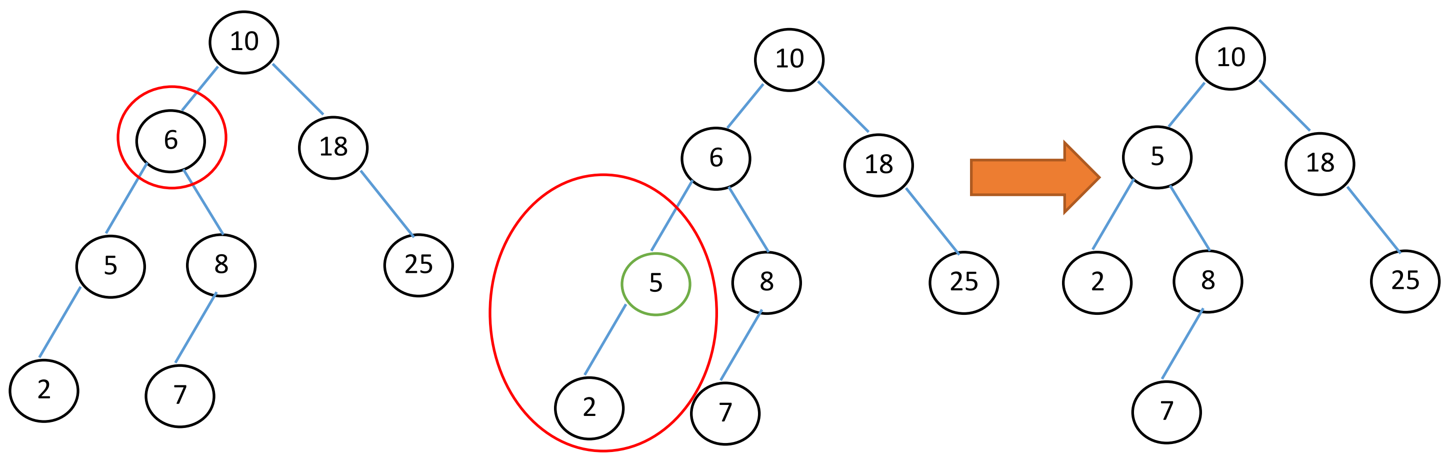
Remove 8



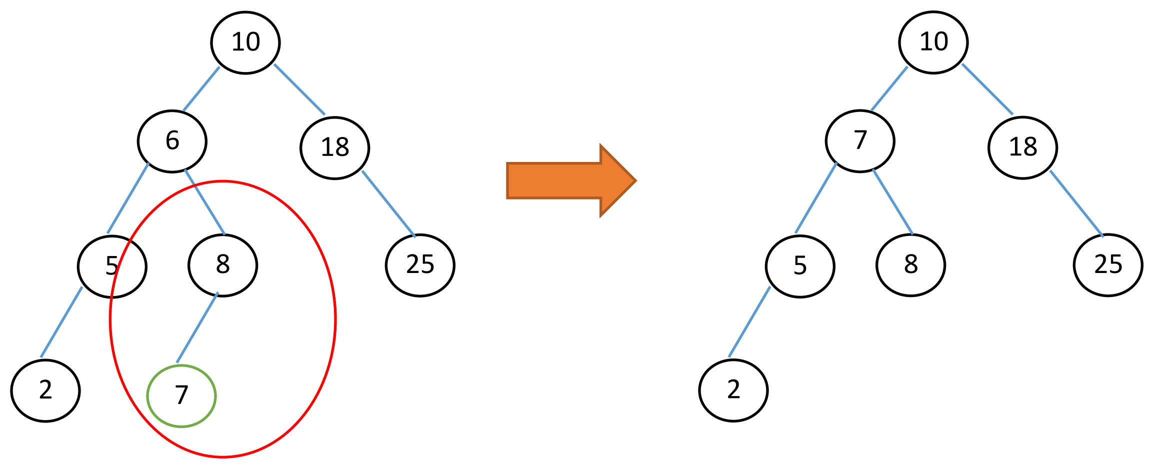
3. Node to be deleted has two children: In this case the strategy is to replace the node being deleted with the largest node in the left subtree of the node to be removed and then delete that largest node. By symmetry, the node being deleted can be swapped with the smallest node is the right subtree. This means that **two** trees can be the result after such removal operation

Remove 6

Option 1: Largest node in left subtree



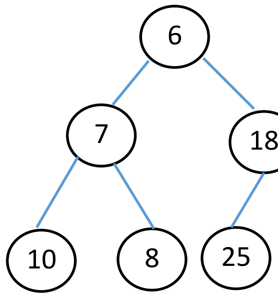
Option 2: Smallest node in right subtree



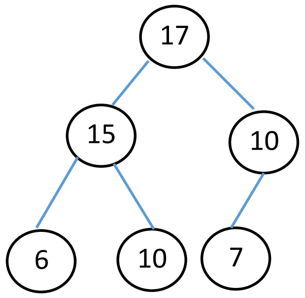
1.1.5 Binary Heaps

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



Min heap



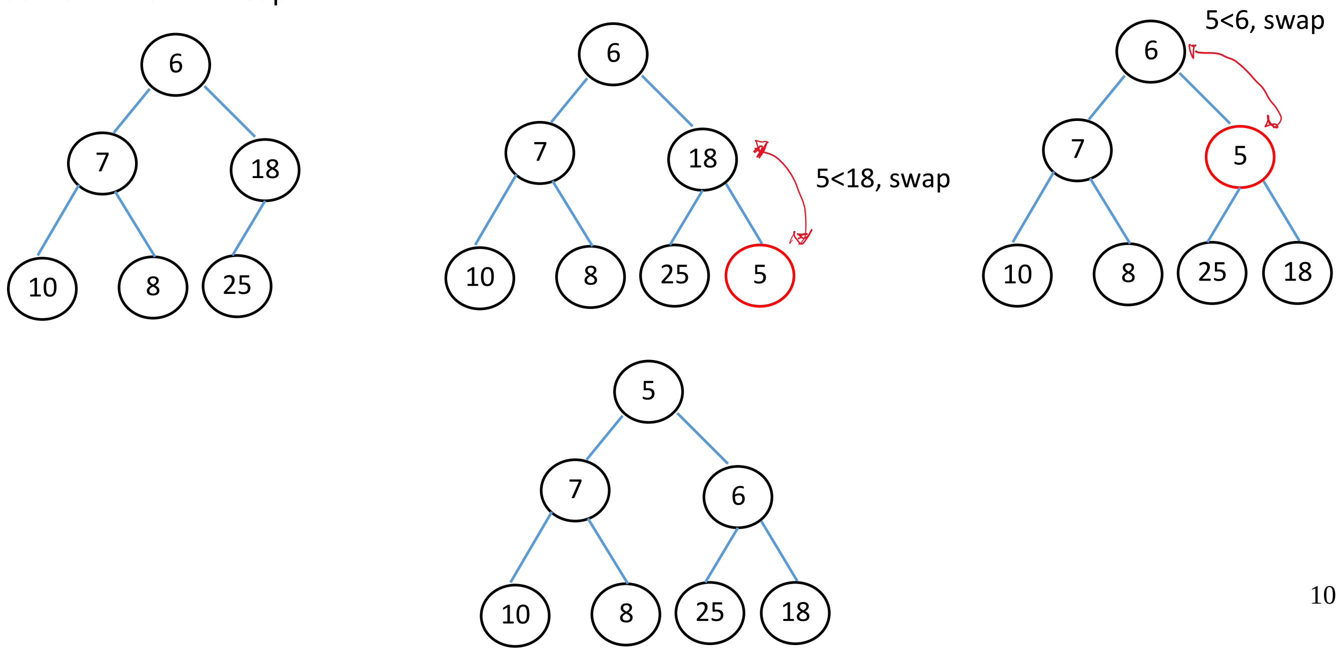
Max heap

In a heap the highest (or lowest) priority element is always stored at the root. A heap is not a sorted structure and can be regarded as partially ordered. As you see from the heaps above, there is no particular relationship among nodes on any given level, even among the siblings. Since a heap is a complete binary tree, it has the smallest possible height, a heap with N nodes always has log N height. A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

1.1.5.1 Insertion

New elements are initially added to the end of the heap. The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.

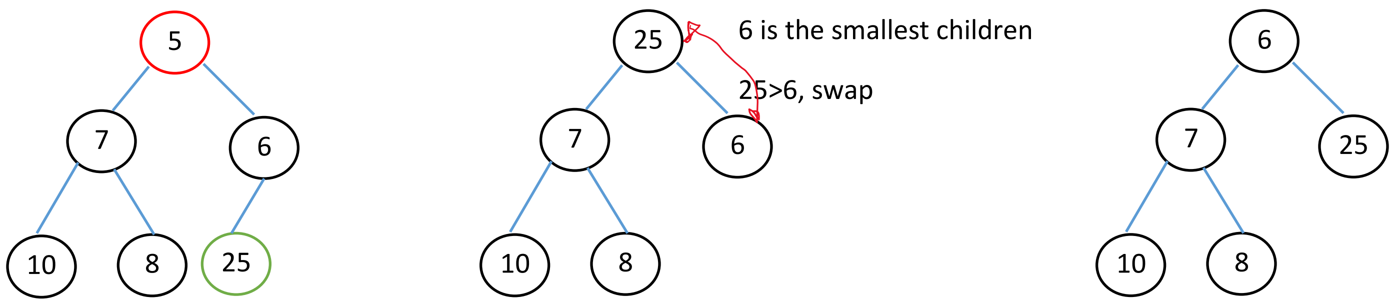
Insert 5 in the min heap



1.1.5.1 Deletion

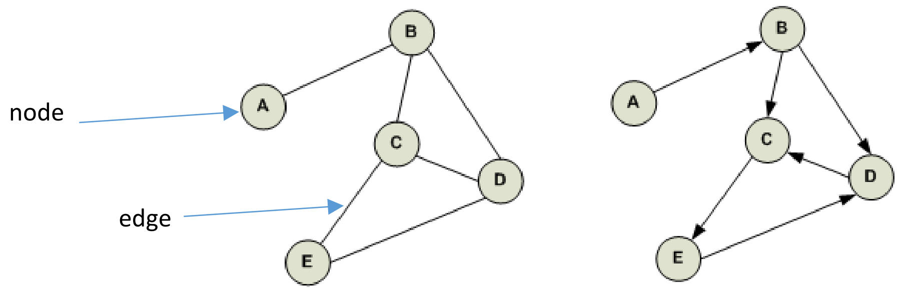
The minimum (or maximum) element can be found at the root. We remove the root and replace it with the last element of the heap and then restore the heap property by *percolating down*.

Remove 5



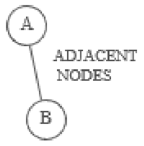
1.2 Graph

A graph is another abstract data structure with nodes (or vertices) that are connected by edges. Graphs can be directed or undirected. In directed graphs, edges point from the node at one end to the node at the other end. In undirected graphs, the edges simply connect the nodes at each end. The edges could represent distance or weight.

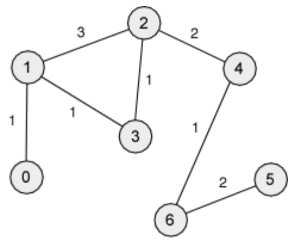


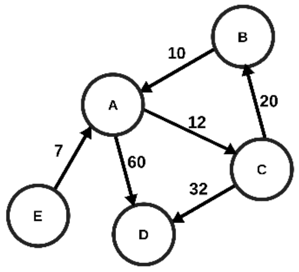
Undirected Graph Directed Graph

Graph are useful in cases where you have things that connect to other things. Nodes and edges could, for example, respectively represent cities and highways, routers and Ethernet cables, or Facebook users and their friendships. Two nodes connected by an edge are *adjacent* or neighbors.

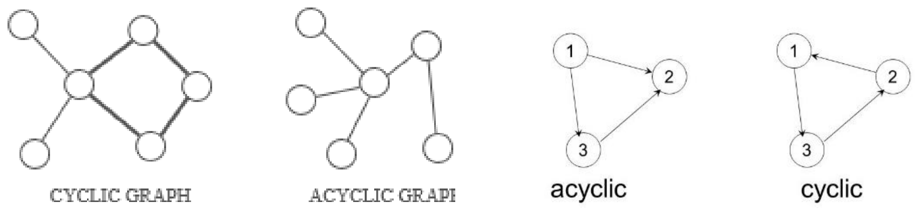


If a graph is weighted, each edge has a weight. The weight could, for example, represent the distance between two locations, or the cost or time it takes to travel between the locations.





A graph is cyclic if it has at least one cycle (an unbroken series of nodes with no repeating nodes or edges that connects back to itself). Graphs without cycles are acyclic.

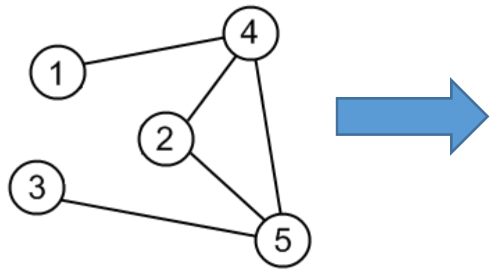


1.2.1 Graph Representation

There are two well-known implementations of a graph, the adjacency matrix and the adjacency list.

1.2.1.1 Adjacency Matrix

One of the easiest ways to implement a graph is to use a two-dimensional matrix of size $V \times V$ where V is the number of vertices in a graph. In this matrix implementation, each of the rows and columns represent a vertex in the graph. The value that is stored in the cell at the intersection of row v and column w indicates if there is an edge from vertex v to vertex w. A value in a cell represents the weight of the edge from vertex v to vertex w (for unweighted graphs we use 1).



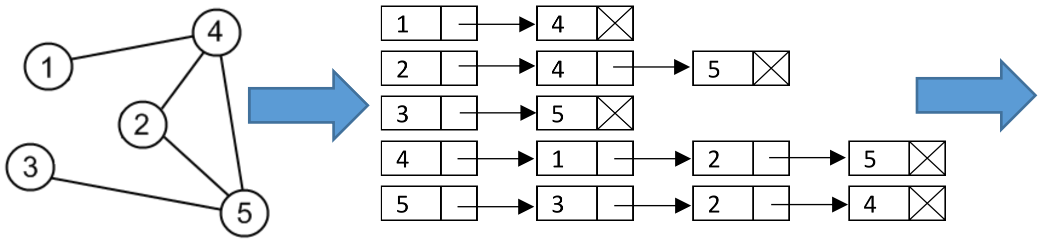
	1	**2**	**3**	**4**	**5**
1				1	
2				1	1
3					1
4	1	1			1
5		1	1	1	

```
graph = [
  [0, 0, 0, 1, 0],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 0, 1],
  [1, 1, 0, 0, 1],
  [0, 1, 1, 1, 0],
]
```

The advantage of the adjacency matrix is that it is simple, and for small graphs it is easy to see which nodes are connected to other nodes. However, notice that most of the cells in the matrix are empty. Because most of the cells are empty we say that this matrix is “sparse”. A matrix is not a very efficient way to store sparse data. The adjacency matrix is a good implementation for a graph when the number of edges is large (in the order of $|V|^2$). A matrix is full when every vertex is connected to every other vertex. There are few real problems that approach this sort of connectivity.

1.2.1.2 Adjacency List

A more space-efficient way to implement a sparsely connected graph is to use an adjacency list. In an adjacency list implementation we keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to.



```
With list:
graph = [
   [3],
   [3, 4],
   [4],
   [0, 1, 4],
   [2, 1, 3],
]
```

```
With dictionary:
graph = {
   0: [3],
   1: [3, 4],
   2: [4],
   3: [0, 1, 4],
   4: [3, 1, 3],
}
```

The advantage of the adjacency list implementation is that it allows us to compactly represent a sparse graph. The adjacency list also allows us to easily find all the links that are directly connected to a particular vertex.

Python does not have a graph data type. To use graphs you can either use the *networkx* module (`pip install networkx`) or implement it yourself

```
import networkx as nx

G=nx.Graph()
G.add_node("A")
G.add_node("B")
G.add_node("C")
G.add_edge("A","B")
G.add_edge("B","C")
G.add_edge("C","A")

print("Nodes: " + str(G.nodes()))
print("Edges: " + str(G.edges()))
```

```
Nodes: ['A', 'C', 'B']

Edges: [('A', 'C'), ('A', 'B'), ('C', 'B')]
```

For more details on networkx, read https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf

```
class Graph(object):
    def __init__(self, graph_dict=None):
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict

    def add_vertex(self, vertex):
        if vertex not in self.__graph_dict:
            self.__graph_dict[vertex] = []

    def add_edge(self, edge):
        edge = set(edge)
        (vertex1, vertex2) = tuple(edge)
        if vertex1 in self.__graph_dict:
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

    def vertices(self):
        return list(self.__graph_dict.keys())

    def edges(self):
        return self.__generate_edges()
```

initializes the graph object on adjacency list mode using a dictionary. If no dictionary or None is given, an empty dictionary will be used

edges can be provided using sets, lists or tuples

1.2.2 BFS and DFS for a graph

Given a graph *G* and a starting vertex *s*, a breadth first search proceeds by exploring edges in the graph to find all the vertices in *G* for which there is a path from *s*. The remarkable thing about a breadth first search is that it finds *all* the vertices that are a distance *k* from *s* before it finds *any* vertices that are a distance *k*+1.

BFS for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

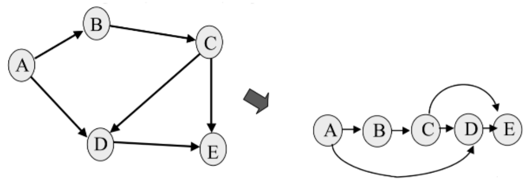
DFS explores the graph depth wise, that is we move down the graph from one node to another till the time we are out of all unexplored edges at a node, once we reach that condition, we backtrack and move to node one before and so on. In this traversal, we start with a node *s*, and mark it as visited. Now *s* be our current node *u*:

- Move to *v* where there is an edge (*u*,*v*).
- If *v* is already visited, we move back to *u*.
- If *v* is not visited, mark *v* as visited and make *v* as current node and repeated above steps.
- At some point all the edges at *u* will lead to already visited node, then we drop *u* and move to node which was visited before *u*.

This backtracking will make us reach the start node *s* again, and when there is no edge left to be explored at *s*, the graph traversal is done. DFS for a graph is similar to preorder traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we also use a boolean visited array.

1.2.3 Topological Sorting

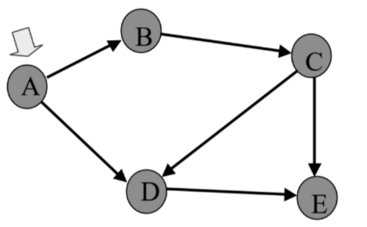
A topological sort takes a directed acyclic graph (DAG) and produces a linear ordering of all its vertices such that if the graph *G* contains an edge (*v*,*w*) then the vertex *v* comes before the vertex *w* in the ordering. DAGs are used in many applications to indicate the precedence of events. For example, software project schedules, precedence charts for optimizing database queries, and multiplying matrices. Any linear ordering in which all the arrows go to the right is a valid solution. Topological sorting for a graph is not possible if the graph is not a DAG.



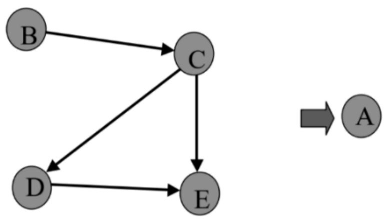
Topological sorting can be accomplished in two different ways:

- Identifying incoming edges:

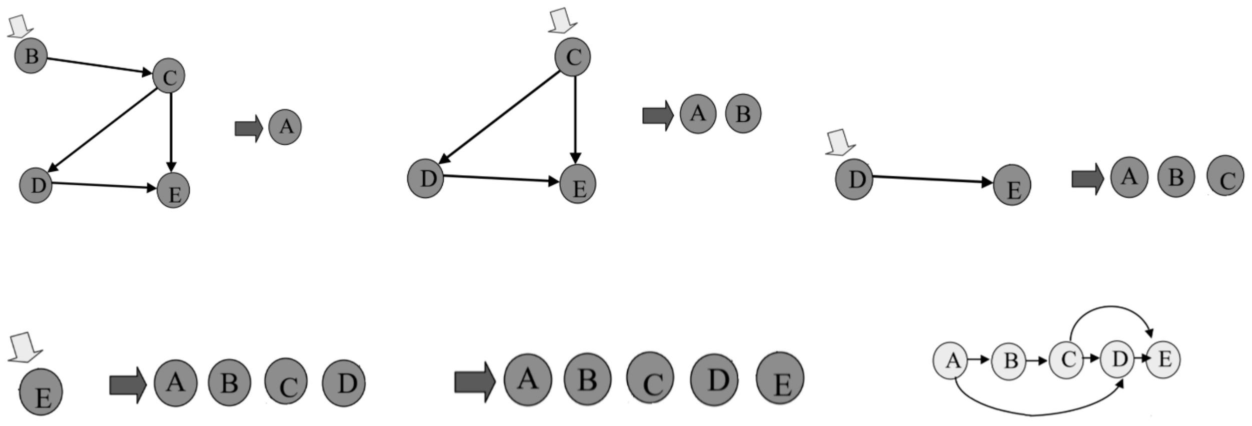
  Step 1- Identify vertices that have no incoming edge (in-degree is zero). If no such edges exist, graph is not DAG. If there are several vertices of in-degree zero, select one.



Step 2- Delete this vertex of in-degree zero and all its outgoing edges from the graph. Place it in the output.

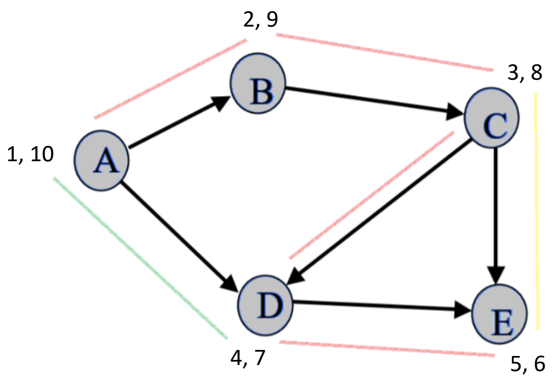


Step 3- Repeat Steps 1 and Step 2 until graph is empty

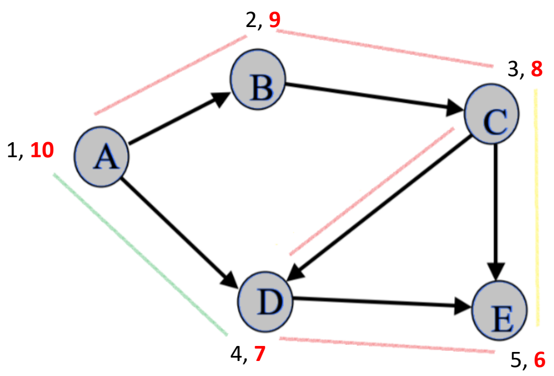


- Using DFS:
  The topological sort is a simple but useful adaptation of a depth first search:
  Step 1- Perform DFS for some graph *G*. (we call DFS to compute the finish times for each of the vertices)

DFS starting from node A

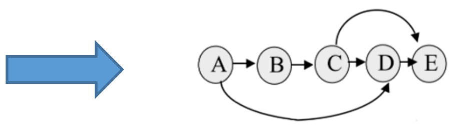


Step 2- Store the vertices in a list in decreasing order of finish time



Step 3- Return the ordered list as the result of the topological sort

list = [‘A’, ‘B’, ‘C’, ‘D’, ‘E’]



$$\begin{tikzcd}
A \arrow[r] \arrow[rrr, bend right] & B \arrow[r] & C \arrow[r] \arrow[rr, bend left] & D \arrow[r] & E
\end{tikzcd}$$

1.3 Hash Tables

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a *slot*, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value None.

hash table of size *m*=11

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

The mapping between an item and the slot where that item belongs in the hash table is called the hash function. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table. A good hash function must be efficiently computable and should uniformly distribute the keys (each table position equally likely for each key). Assume that we have the set of integer items 32, 53, 79, 15, 93, and 67. The modular method is the most common hash function, it simply takes an item and divides it by the table size, returning the remainder as its hash value (hash(item)=item%11).

Item	**Hash Value (item%11)**
32	10
53	9
79	2
15	4
93	5
67	1

Once the hash values have been computed, we can insert each item into the hash table at the designated position. Note that 6 of the 11 slots are now occupied. This is referred to as the *load factor*, and is commonly denoted by $\lambda = \frac{\text{\# of items}}{\text{table size}}$ (For this example, $\lambda = \frac{6}{11}$).

0	1	2	3	4	5	6	7	8	9	10
None	67	79	None	15	93	None	None	None	53	32

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation takes constant time, since a constant amount of time is required to compute the hash value and then index the hash table at that location. However, this technique is going to work only if each item maps to a unique location in the hash table. For example, if we want to add the item 42 to our collection, it would have a hash value of 9 (42%11== 9). Since 53 also had a hash value of 9, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a *collision*, and it creates problems for the hashing technique.

1.3.1 Hash Functions

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a *perfect hash function*. If we know the items and the collection will never change, then it is possible to construct a perfect hash function. Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function. Thus, the goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are several ways to extend the simple modular method.

1.3.1.1Folding method

This method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash item. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210%11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219%11=10.

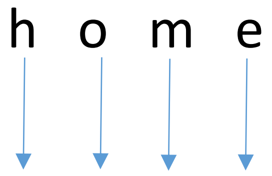
1.3.1.2Mid-square method

This method for constructing hash functions begins by squaring the item, and then extract some portion of the resulting digits. For example, for the item 44, we would first compute 44^2=1,936. By extracting the middle two digits, 93, and performing the remainder step, we get 93%11=5).

Item	Hash Value (item%11)	Mid-square
32	10	2
53	9	3
79	2	2
15	4	3
93	5	9
67	1	4

We can also create hash functions for character-based items such as strings. The word “home” can be thought of as a sequence of ordinal values. We can then take these four ordinal values, add them up, and use the modular method to get a hash value:

```
>>> ord('h')
104
>>> ord('o')
111
>>> ord('m')
109
>>> ord('e')
101
```



104 + 111 + 109 + 101 = 425

425 % 11 = 7

ord(c): Given a string representing one Unicode character, return an integer representing the Unicode code point of that character

You may be able to think of a number of additional ways to compute hash values for items in a collection. The important thing to remember is that the hash function has to be efficient so that it does not become the dominant part of the storage and search process.

1.3.2 Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

1.3.2.1Open Addressing

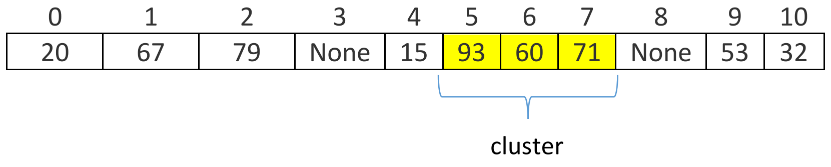
One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty (we may need to go back to the first slot (circularly) to cover the entire hash table). This collision resolution process is referred to as *open addressing*, where we try to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called *linear probing*.

When we attempt to place 60 into slot 5, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 6. Again, 71 should go in slot 5 but must be placed in slot 7 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slot 10 and finally find an empty slot at position 0.

0	1	2	3	4	5	6	7	8	9	10
20	67	79	None	15	93	60	71	None	53	32

Once we have built a hash table using open addressing and linear probing, it is essential that we utilize the same methods to search for items. Assume we want to look up the item 93. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return True. However, when we look for item 20, the hash value is 9, and slot 9 is currently holding 53. We cannot simply return False since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot.

A disadvantage to linear probing is the tendency for *clustering*; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, for example, when we try to add the item 26, a cluster of values hashing to 5 had to be skipped to finally find an open position.



0	1	2	3	4	5	6	7	8	9	10
20	67	79	None	15	93	60	71	None	53	32

One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs. Below there is a hash table where the collision resolution is done with a +3 probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

0	1	2	3	4	5	6	7	8	9	10
71	67	79	None	15	93	None	20	60	53	32

The general name for this process of looking for another slot after a collision is *rehashing*. In general, *hash_index=(hash_index + skip) % hash_table_size*. It is important to note that the size of the "skip" must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size be a prime number.

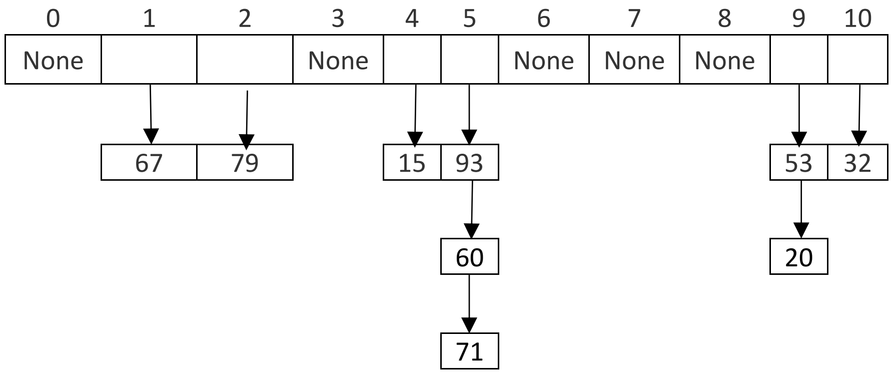
A variation of the *linear probing* idea is called *quadratic probing*. This method is similar to linear probing and the only difference is that if you were to try to insert into a space that is filled you would first check $1^2 = 1$ element away then $2^2 = 4$ elements away, then $3^2 =9$ elements away then $4^2=16$ elements away and so on.

0	1	2	3	4	5	6	7	8	9	10
None	67	79	71	15	93	60	20	None	53	32

With linear probing we know that we will always find an open spot if one exists (it might be a long search but we will find it). However, this is not the case with quadratic probing unless you take care in the choosing of the table size. In order to guarantee that the quadratic probes will hit every single available spot eventually, the hash table size must be a prime number and never be more than half full (even by one element).

1.3.2.2 Chaining

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.



When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient.

It is important to mention that if the load factor (λ) is small, then there is a lower chance of collisions, meaning that items are more likely to be in the slots where they belong. If λ is large, meaning that the table is filling up, then there are more and more collisions. This means that collision resolution is more difficult, requiring more comparisons to find an empty slot.

1.3.3 Hashing Implementation

Python has useful data types that could help us to develop our own hashing algorithm. For example, a dictionary that can store key-data pairs could be used to look up a data value based on a given key. However, we are living in a digitally disrupted world, where sensitive information flows among users and organizations. As a result, information security has become very important in most organizations. The main reason for this is that access to information and the associated resources has become easier because of the developments in distributed processing, for example the Internet and electronic commerce. The result is that organizations need to ensure that their information is properly protected and that they maintain a high level of information security.

Hash functions are used inside some cryptographic algorithms, in digital signatures, message authentication codes, manipulation detection, fingerprints, checksums (message integrity check), hash tables, password storage and much more. As a Python programmer you may need these functions to check for duplicate data or files, to check data integrity when you transmit information over a network, to securely store passwords in databases, or maybe some work related to cryptography. The Federal Information Processing Standards, as well as the Internet Engineering Task Force have defined and developed different hash algorithms that are widely use all over the Internet. Python's hashlib module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321).

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def hashfunction(self,key,size):
         return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size
```

Class structure to develop our own hashing techniques

```
list(map(hash, [0, 1, 2, 3]))
# Output: [0, 1, 2, 3]
```

hashing using Python's built-in hash function. Python is using different hash() function depending on the type of data

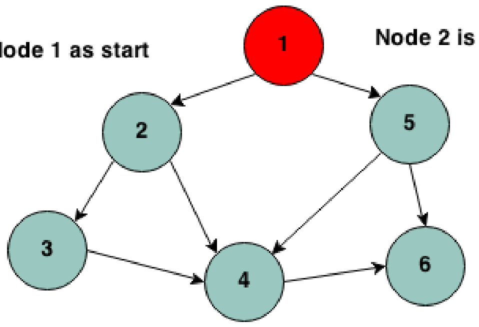
```
list(map(hash, ['0','1','2','3']))
#Output: [3512700405625046622, -2154559771013955726, 4558503884423229281, -5090714507869946363]

hash('0')
#Output: 3512700405625046622


import hashlib
```

hashing using Python's hashlib module

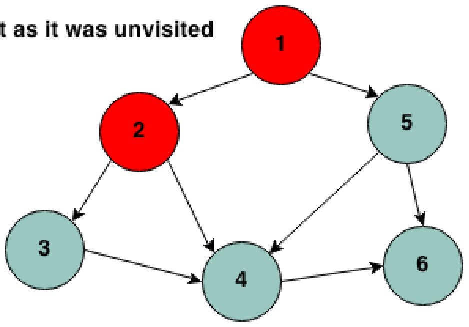
```
print(hashlib.algorithms_available)
# Output: {'sha', 'sha224', 'sha3_256', 'DSA', 'shake_128', 'sha512', 'SHA224', 'RIPEMD160',
'dsaEncryption',  'ripemd160',  'sha3_512',  'MD5',  'SHA384',  'SHA256',  'md5',  'blake2b',
'sha3_384', 'sha384', 'sha256', 'sha3_224', 'ecdsa-with-SHA1', 'whirlpool', 'SHA', 'shake_256',
'sha1', 'md4', 'SHA1', 'DSA-SHA', 'SHA512', 'blake2s', 'MD4', 'dsaWithSHA'}

hash_object = hashlib.md5(b'Hello World')
print(hash_object.hexdigest())
# Output: b10a8db164e0754105b7a99be72e3fe5

value = input('Enter string to hash: ')
hash_object = hashlib. sha256 (value.encode())
print(hash_object.hexdigest())
# Output:
       Enter String to hash: Hello World
       a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
```

Appendix 1. Depth First Search Example

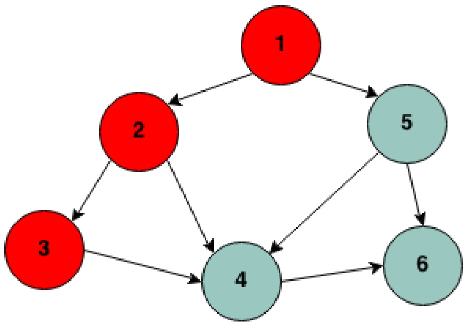
Take Node 1 as start



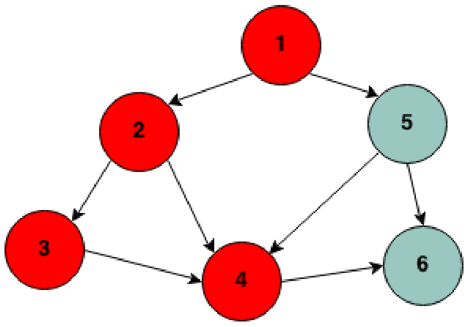
Node 2 is visited next as it was unvisited



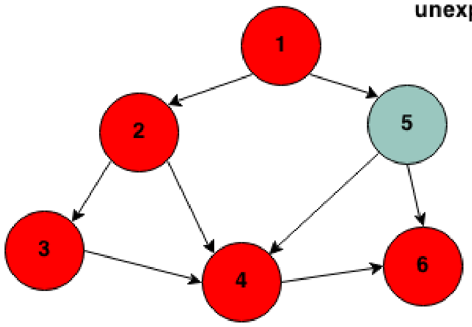
Node 3 is visited next as it was unvisited



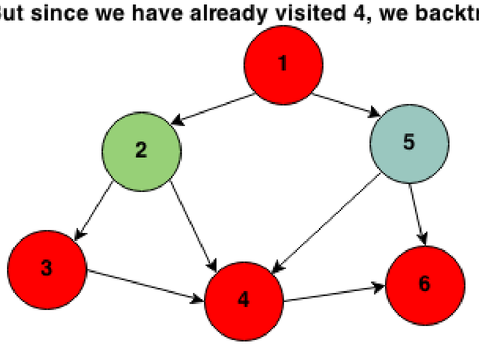
Node 4 is visited next as it was unvisited



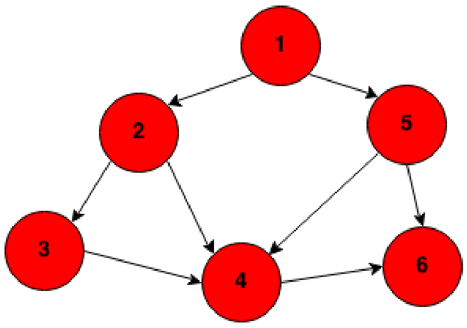
Node 6 is visited next as it was unvisited



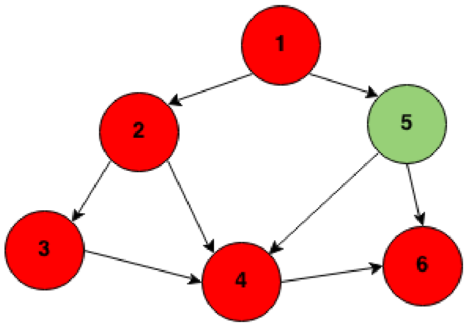
As there is no more nodes which can be traversed
from 6, we back track, first node where there is
unexplored edge is 2. But since we have already visited 4, we backtrack further



Node 5 from node 1 is visited next as it was unvisited

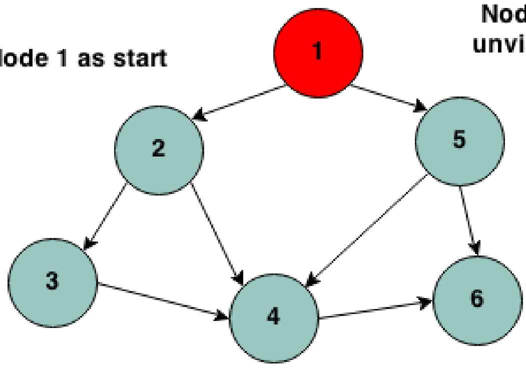


Again all neighbor nodes from 5 are already visited,
hence we back track to node 1

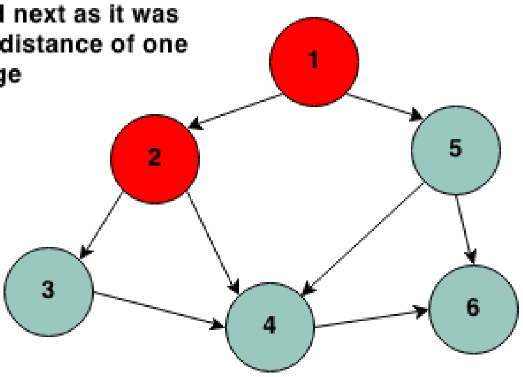


Appendix 2. Breadth First Search Example

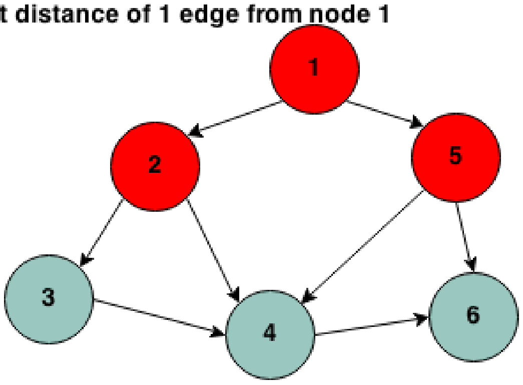
Take Node 1 as start



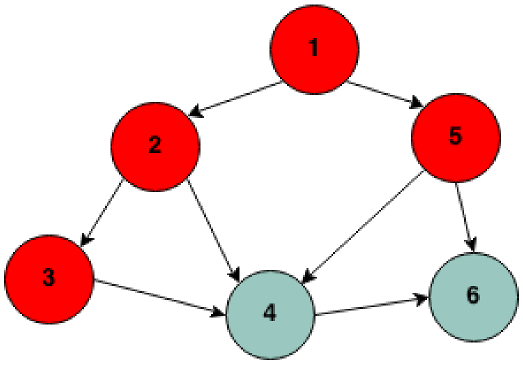
Node 2 is visited next as it was unvisited and at distance of one edge



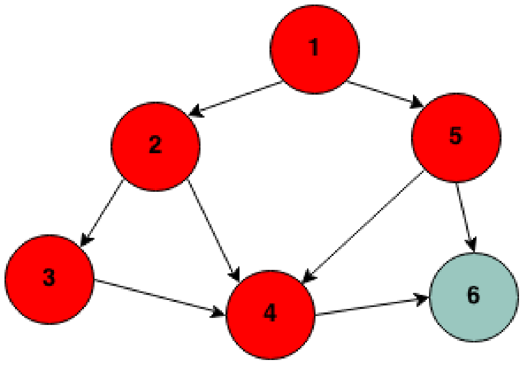
Node 5 is visited next as it was unvisited and at distance of 1 edge from node 1



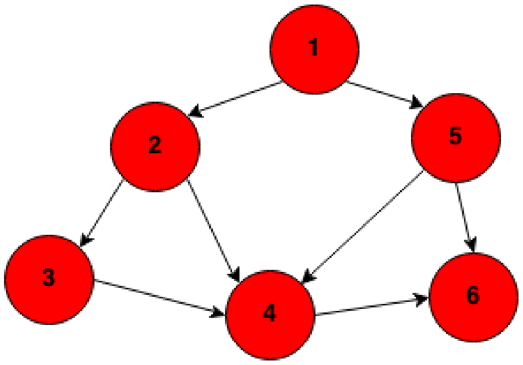
Node 3 is visited next as it was unvisited and at distance of one edge from node 2



Node 4 is visited next as it was unvisited and at distance of one edge from node 2



Node 6 is visited next as it was unvisited and at distance of one edge from node 5



References

Data Structures and Algorithms in Python. Michael H. Goldwasser, Michael T. Goodrich, and Roberto Tamassia. Chapters 6, 7, 8, 10, 14

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

http://en.wikipedia.org/wiki/Priority_queue

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

https://docs.python.org/3/library/hashlib.html