U O W

# Non-linear Data Structure

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

# Objective

- To understand what a tree data structure is and how it is used.
- To see how trees can be used to implement a map data structure.
- To implement trees using a list.
- To implement trees using classes and references.
- To implement a priority queue using a heap.
- To learn what a graph is and how it is used.
- To implement the **graph** abstract data type using multiple internal representations.
- To see how graphs can be used to solve a wide variety of problems

# What is Non-linear Data Structures

- Data structures where data elements are not arranged sequentially or linearly.
- Unlike linear data structure, a non-linear data structure involves not just a single level, but multi-level data structure.
- Hence, we needs to run recursively multiple time to traverse all the elements.
- Non-linear data structures are not easy to implement in comparison to linear data structure.
- Comparing to a linear data structure, non-linear data structures utilizes computer memory more efficiently.
- Examples of non-linear data structure are trees and graphs.

# What is Trees.

- Trees are used in many areas of computer science, including operating systems, graphics, database systems, and computer networking.
- Tree data structures have many things in common with their botanical cousins. A tree data structure has a root, branches, and leaves.
- The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom.
- Before we begin our study of tree data structures, let's look at a few common examples.

# What is Trees (Cons)

- The example of a tree is a classification tree from biology.
- An example of the biological classification of some animals is shown in the Figure 1.
- From this simple example, we can learn about several properties of trees. The first property this example demonstrates is that trees are hierarchical.
- By hierarchical, we mean that trees are structured in layers with the more general things near the top and the more specific things near the bottom.
- The top of the hierarchy is the Kingdom, the next layer of the tree (the “children” of the layer above) is the Phylum, then the Class, and so on.
- However, no matter how deep we go in the classification tree, all the organisms are still animals.

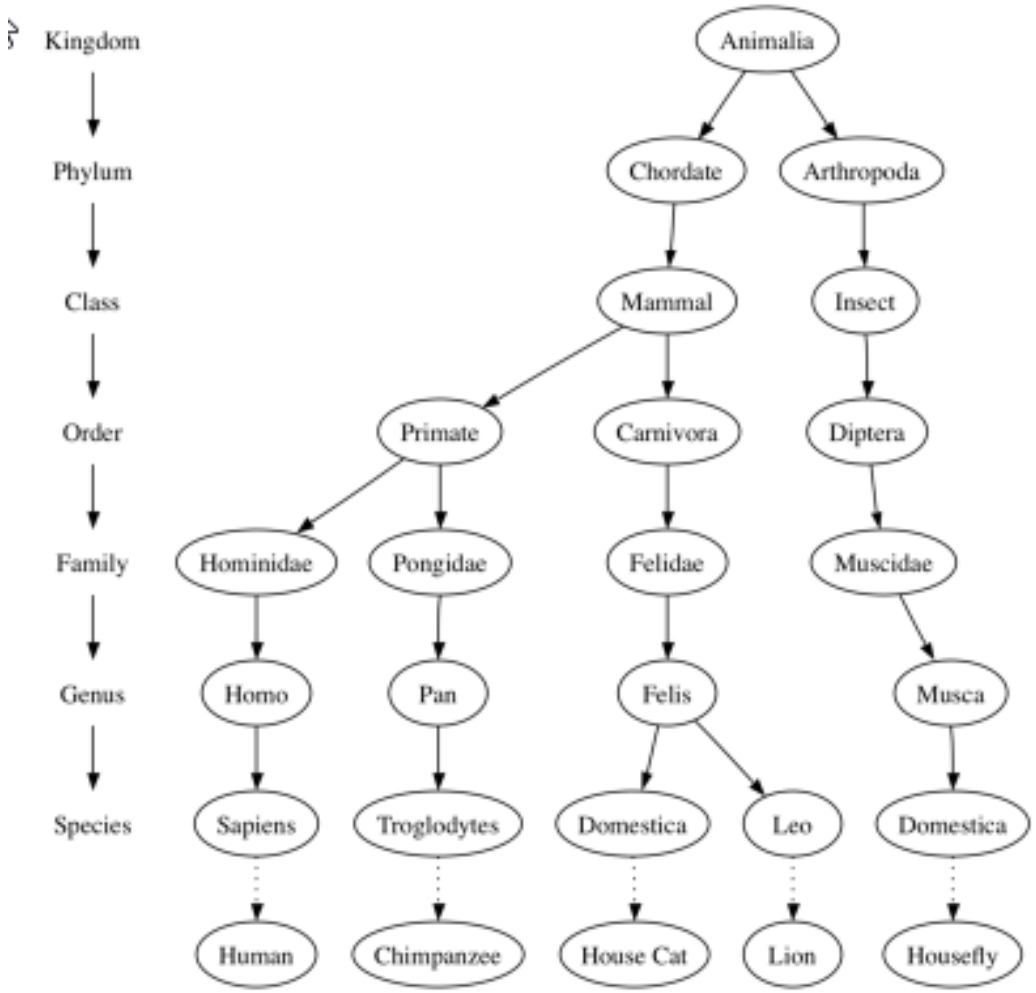



Figure 1: Taxonomy of Some Common Animals Shown as a Tree

Source Image by B. Miler and D. Ranum in Problem Solving with Algorithms and Data Structures using Python

# Definitions

- Node:
- Edge:
- Root:
- Path:
- Children:
- Parent:
- Sibling:
- Subtree:
- Leaf Node:
- Level:
- Height:

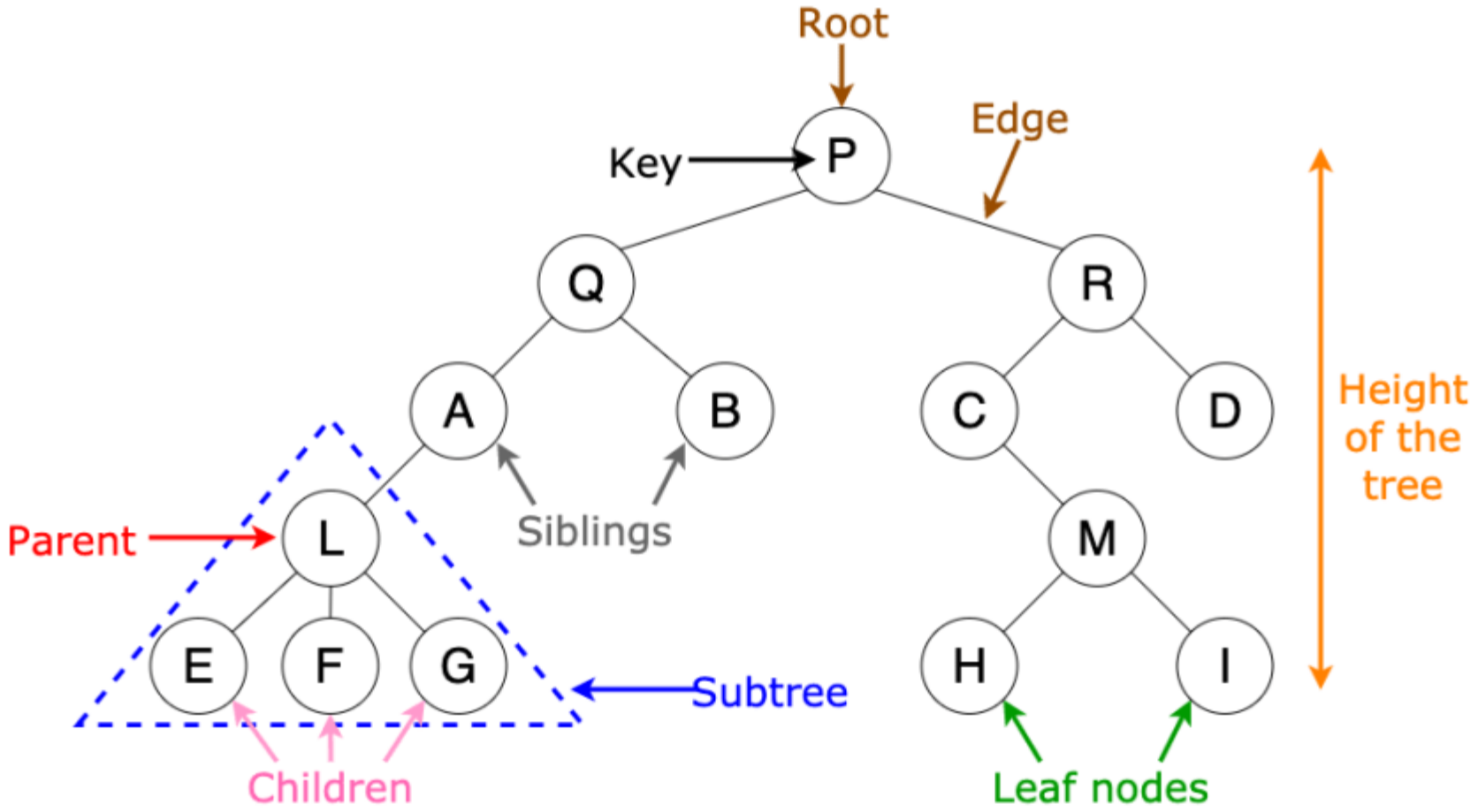



Figure 2: Trees

Source image by Christina in https://dev.to/christinamcmahon/understanding-binary-search-trees-4d90

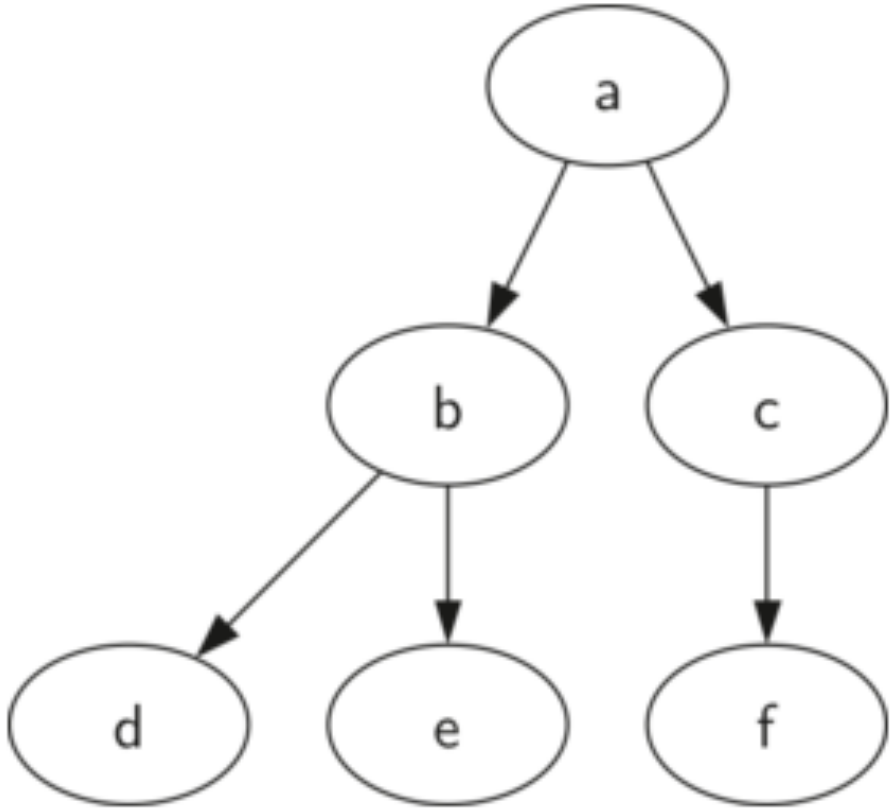
# Construct Tree from python List

- In a tree represented by a list of lists, we will begin with Python's list data structure and write the functions defined above.
- Although writing the interface as a set of operations on a list is a bit different from the other abstract data types we have implemented, it is interesting to do so because it provides us with a simple recursive data structure that we can look at and examine directly.
- In a list of lists tree, we will store the value of the root node as the first element of the list.
- The second element of the list will itself be a list that represents the left subtree.
- The third element of the list will be another list that represents the right subtree.
- To illustrate this storage technique, let's look at an example.

# Construct Tree from python List

- The below figure shows a simple tree and the corresponding list implementation.

```
myTree = ['a',   #root
    ['b',  #left subtree
     ['d', [], []],
     ['e', [], [] ],
    ['c',  #right subtree
     ['f', [], []],
     [] ]
   ]
```




Source image by Christina in https://dev.to/christinamcmahon/understanding-binary-search-trees-4d90

# Example code for tree construction

- myTree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c', ['f',[],[]], []] ]
- print(myTree)
- print('left subtree = ', myTree[1])
- print('root = ', myTree[0])
- print('right subtree = ', myTree[2])

# Binary Tree

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

# Binary Trees

- Let construct the binary tree data structure.
- Binary data structure is a tree structure that each level contain only itself value and the left and right subtree node.
- If the  subtree node is empty, it then contain the empty list.
- Let's formalize this definition of the tree data structure by providing some functions that make it easy for us to use lists as trees.
- Note that we are not going to define a binary tree class.
- The utility functions will just help us manipulate a standard list as though we are working with a tree.

# Example code

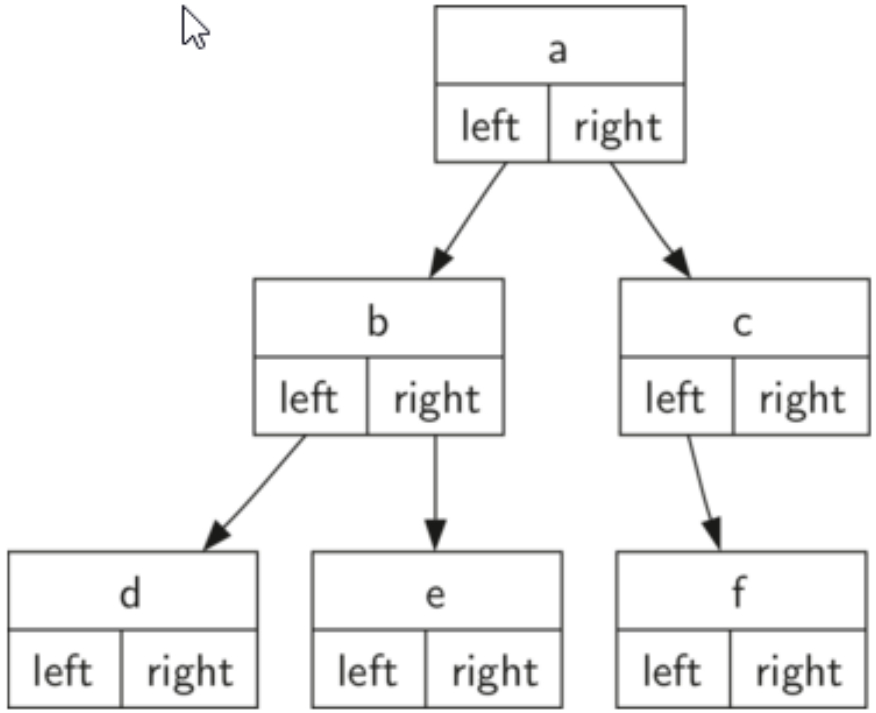
```
def BinaryTree(r):
    return [r, [], []]
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
def getRootVal(root):
    return root[0]
```

# Example code

```
def setRootVal(root,newVal):
    root[0] = newVal
def getLeftChild(root):
    return root[1]
def getRightChild(root):
    return root[2]
r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)
setRootVal(l,9)
print(r)
insertLeft(l,11)
print(r)
print(getRightChild(getRightChild(r)))
```

# Binary Tree Class

- Our second method to represent a tree uses nodes and references.
- In this case we will define a class that has attributes for the root value, as well as the left and right subtrees.
- Since this representation more closely follows the object-oriented programming paradigm and used in the previous construction for most of data structure in this subject, we will continue to use this representation.
- The right figure shows a simple tree and the corresponding list implementation.



# Binary Tree Class

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t
    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
```

# Binary Tree Class

```
def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

def setRootVal(self,obj):
    self.key = obj

def getRootVal(self):
    return self.key
```

# Binary Tree Class

```
r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

# Tree Traversals

There are three commonly used patterns to visit all the nodes in a tree. The term visitation of the nodes is called a “traversal”

**preorder**

In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

**inorder**

In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

**postorder**

In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

U O W

# Priority Queues with Binary Heaps

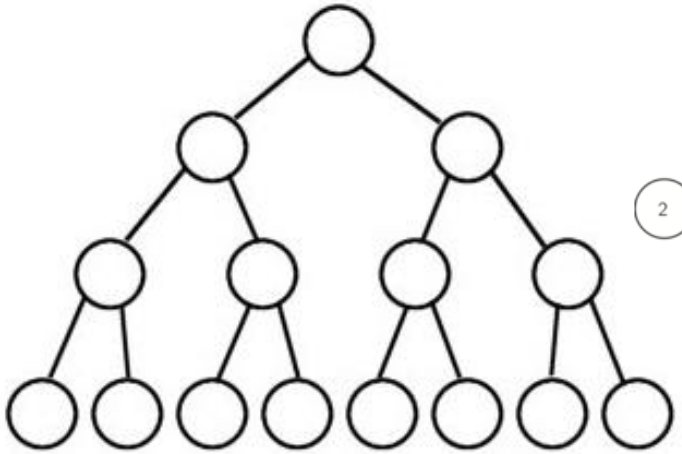
CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

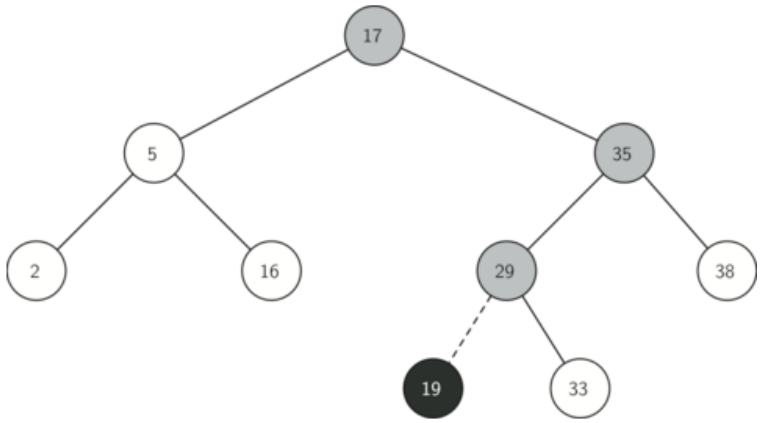
# Heap

- A heap is a tree data structure that is an (or almost) complete tree and satisfied the heap property:
- A max heap: the key value in the root or a parent node is more than or equal to the subtree node.
- A min heap: the key value in the root or a parent node is less than or equal to the subtree node.
- A heap is either the highest or lowest priority element  which is always stored at the root (top of the tree).
- A heap is not a sorted structure and it is only partially ordered.

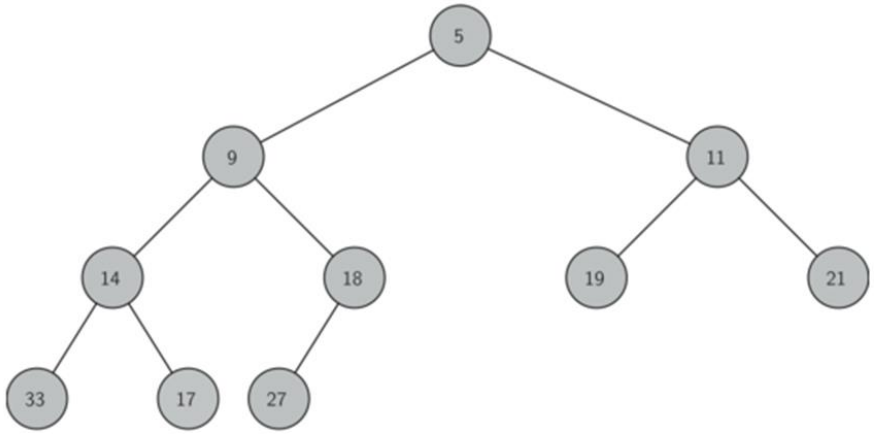
# Binary Heap Structure



Full binary Tree



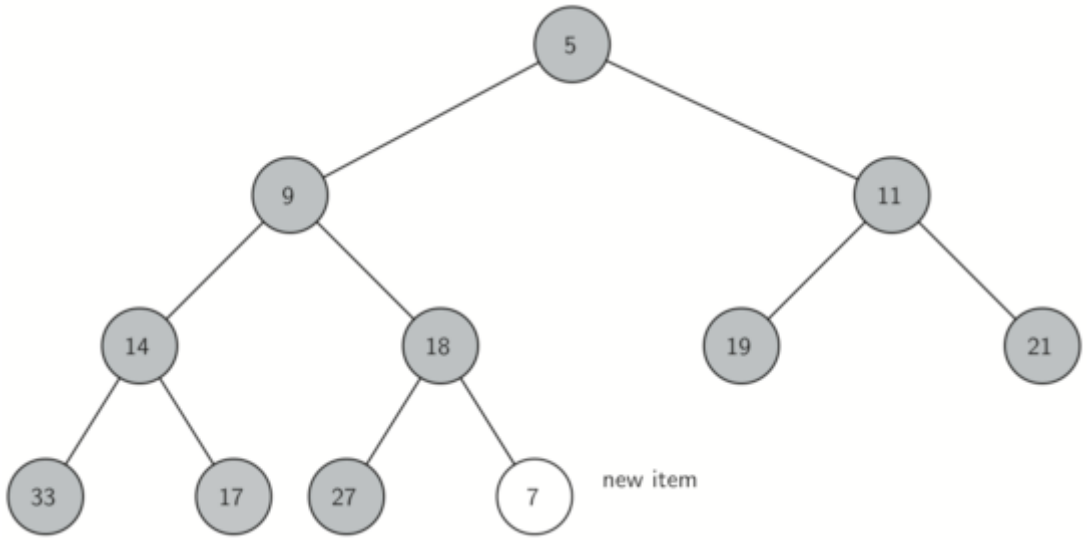

Strctly binary Tree

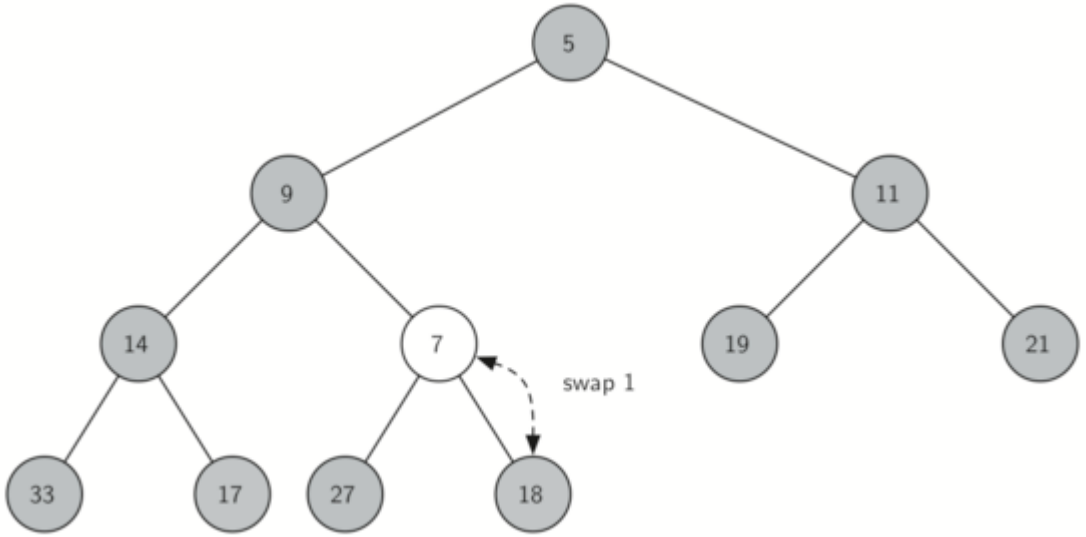



Complete binary Tree

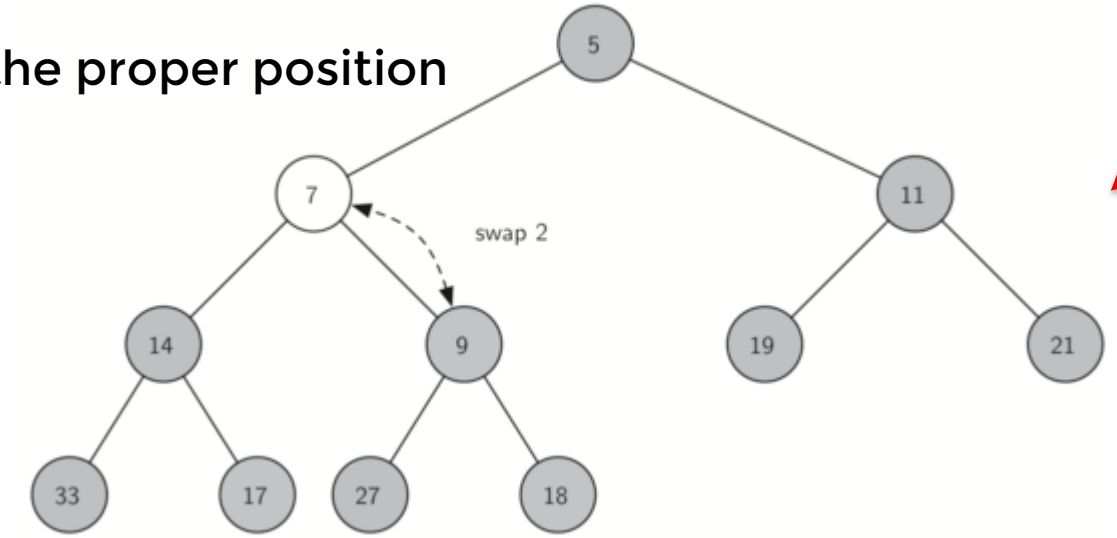
In order to make a heap work efficiently such that it will take advantage of the logarithmic nature of the binary tree to represent the complexity of the heap, the complete binary tree is needed.
In order to guarantee logarithmic performance, we must keep our tree balanced.

# Heap Operations: insert




Relocate its node to the proper position



# Queues with Binary Heap

- One important variation of a queue is called a priority queue.
- In a priority queue, the logical order of items inside a queue is determined by their priority.
- The highest priority items are  at the front.
- The lowest priority items  are at the back.
- Inserting to a list is O(n), sorting a list is O(n log(n)).
- A binary heap allows us to queue items within O(log n).
- In next slide, the min heap is implemented in the example code.

# Binary Heap Class

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
    def percUp(self,i):
        while i // 2 > 0:
          if self.heapList[i] < self.heapList[i // 2]:
              tmp = self.heapList[i // 2]
              self.heapList[i // 2] = self.heapList[i]
              self.heapList[i] = tmp
          i = i // 2
    def insert(self,k):
      self.heapList.append(k)
      self.currentSize = self.currentSize + 1
      self.percUp(self.currentSize)
```

# Binary Heap Class

```
    def minChild(self,i):
      if i * 2 + 1 > self.currentSize:
          return i * 2
      else:
          if self.heapList[i*2] < self.heapList[i*2+1]:
              return i * 2
          else:
              return i * 2 + 1
    def percDown(self,i):
      while (i * 2) <= self.currentSize:
          mc = self.minChild(i)
          if self.heapList[i] > self.heapList[mc]:
              tmp = self.heapList[i]
              self.heapList[i] = self.heapList[mc]
              self.heapList[mc] = tmp
          i = mc
  def delMin(self):
     retval = self.heapList[1]
     self.heapList[1] = self.heapList[self.currentSize]
     self.currentSize = self.currentSize - 1
     self.heapList.pop()
     self.percDown(1)
     return retval
```

# Binary Heap Class

```
    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1

bh = BinHeap()
bh.buildHeap([9,5,6,2,3])

print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
```

# Binary Search Tree

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

# Recall Map Data Type

Map() : Create a new, empty map.

put(key,val) : Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.

get(key) : Given a key, return the value stored in the map or None otherwise.

del : Delete the key-value pair from the map using a statement of the form del map[key].

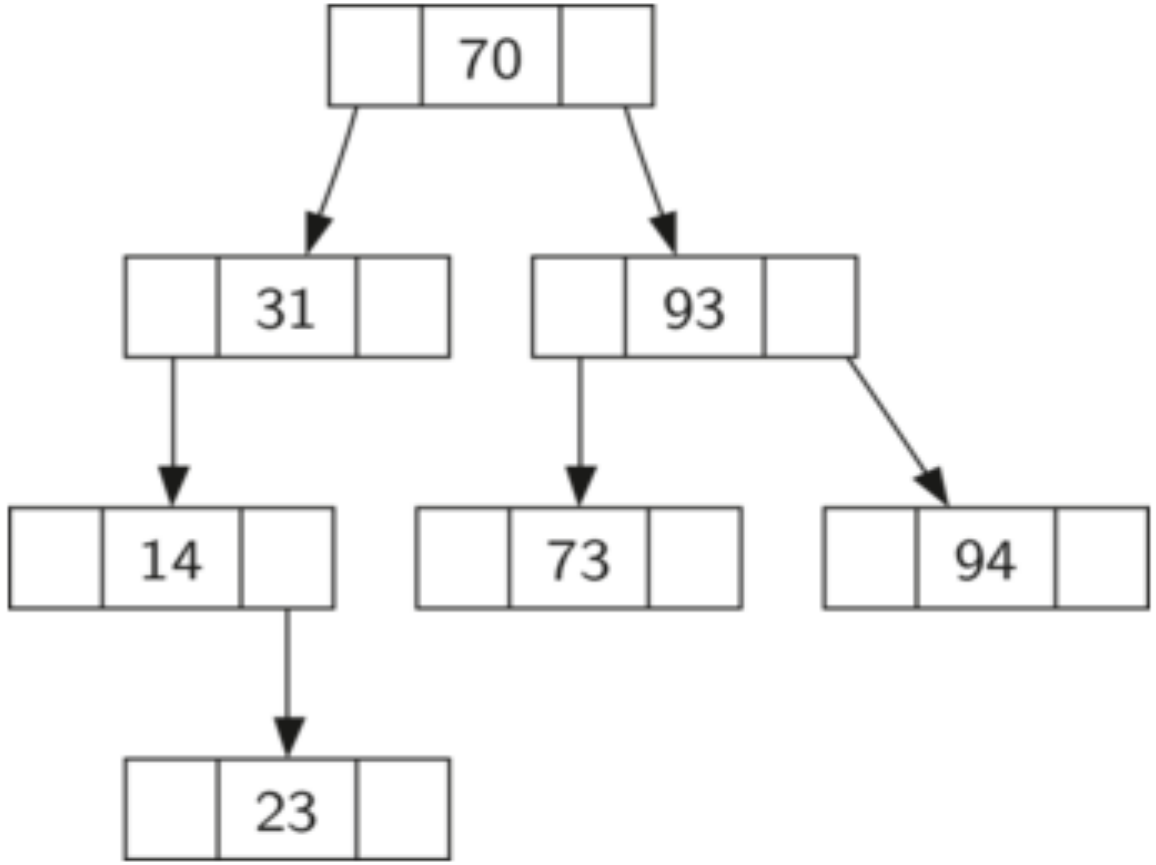
len() : Return the number of key-value pairs stored in the map.

in : Return True for a statement of the form key in map, if the given key is in the map.

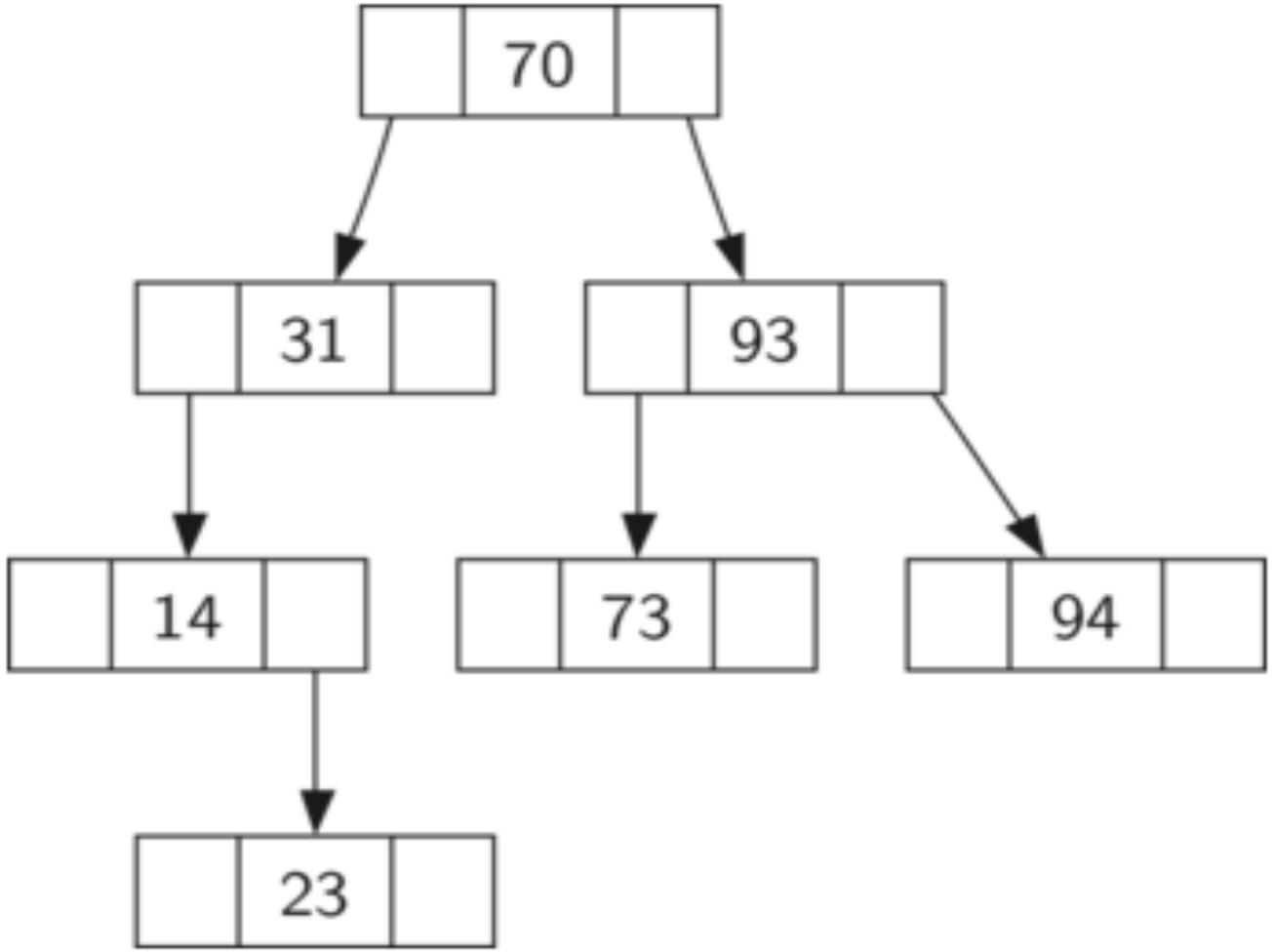
# Binary Search Tree

- The binary search tree property
  - ❑ keys that are less than the parent are found in the left subtree.
  - ❑ keys that are greater than the parent are found in the right subtree.

- Notice that the property holds for each parent and child. This is also apply for the root of the tree.



# Binary Search Tree



# Binary Search Tree: Design

- TreeNode Class:
  - hasLeftChild, hasRightChild
  - isLeftChild, isRightChild
  - isRoot, isLeaf
  - hasAnyChildren, hasBothChildren
  - spliceOut
  - findSuccessor, findMin
  - replaceNodeData
- BinarySearchTree Class
  - length, __len__
  - put, _put, __setitem__
  - gut, _gut, __getitem__, __contains__
  - delete, __delitem__, remove

# Tree Node Class

```
class TreeNode:
    def __init__(self,key,val,left=None,right=None,parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self
```

# Tree Node Class

```
def isRightChild(self):
    return self.parent and self.parent.rightChild == self

def isRoot(self):
    return not self.parent

def isLeaf(self):
    return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):
    return self.rightChild or self.leftChild

def hasBothChildren(self):
    return self.rightChild and self.leftChild
```

# Tree Node Class

```
def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
            self.parent.leftChild = None
        else:
            self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
            if self.isLeftChild():
                self.parent.leftChild = self.leftChild
            else:
                self.parent.rightChild = self.leftChild
            self.leftChild.parent = self.parent
        else:
            if self.isLeftChild():
                self.parent.leftChild = self.rightChild
            else:
                self.parent.rightChild = self.rightChild
            self.rightChild.parent = self.parent
```

# Tree Node Class

```
def findSuccessor(self):
    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
    else:
        if self.parent:
            if self.isLeftChild():
                succ = self.parent
            else:
                self.parent.rightChild = None
                succ = self.parent.findSuccessor()
                self.parent.rightChild = self
    return succ

def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current
```

# Tree Node Class

```
def replaceNodeData(self,key,value,lc,rc):
    self.key = key
    self.payload = value
    self.leftChild = lc
    self.rightChild = rc
    if self.hasLeftChild():
        self.leftChild.parent = self
    if self.hasRightChild():
        self.rightChild.parent = self
```

# Binary Search Tree Class

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):  # print (len(mytree))
        return self.size
```

# Binary Search Tree Class: Inserting Node

```
def put(self,key,val):
    if self.root:
        self._put(key,val,self.root)
    else:
        self.root = TreeNode(key,val)
    self.size = self.size + 1
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
        else:
                currentNode.leftChild = TreeNode(key,val,parent=currentNode)
    else:
        if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
        else:
                currentNode.rightChild = TreeNode(key,val,parent=currentNode)
def __setitem__(self,k,v):   #  mytree[3]="red" : k=3 v="red"
   self.put(k,v)
```

# Binary Search Tree Class: Retrieve Payload

```
def get(self,key):
    if self.root:
        res = self._get(key,self.root)
        if res:
                return res.payload
        else:
                return None
    else:
        return None
def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)
def __getitem__(self,key):  # print(mytree[6]): key = 6
    return self.get(key)
```

# Binary Search Tree Class: Delete Node

```
def __contains__(self,key):    # print(3 in mytree): key = 3
    if self._get(key,self.root):
        return True
    else:
        return False
def delete(self,key):
  if self.size > 1:
      nodeToRemove = self._get(key,self.root)
      if nodeToRemove:
          self.remove(nodeToRemove)
          self.size = self.size-1
      else:
          raise KeyError('Error, key not in tree')
  elif self.size == 1 and self.root.key == key:
      self.root = None
      self.size = self.size - 1
  else:
      raise KeyError('Error, key not in tree')
def __delitem__(self,key):  # del mytree[3] : key = 3
    self.delete(key)
```

# Binary Search Tree Class: Delete Node (2)

```
def remove(self,currentNode):
      if currentNode.isLeaf(): #leaf
        if currentNode == currentNode.parent.leftChild:
            currentNode.parent.leftChild = None
        else:
            currentNode.parent.rightChild = None
      elif currentNode.hasBothChildren(): #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
```

# Delete node from Binary Tree

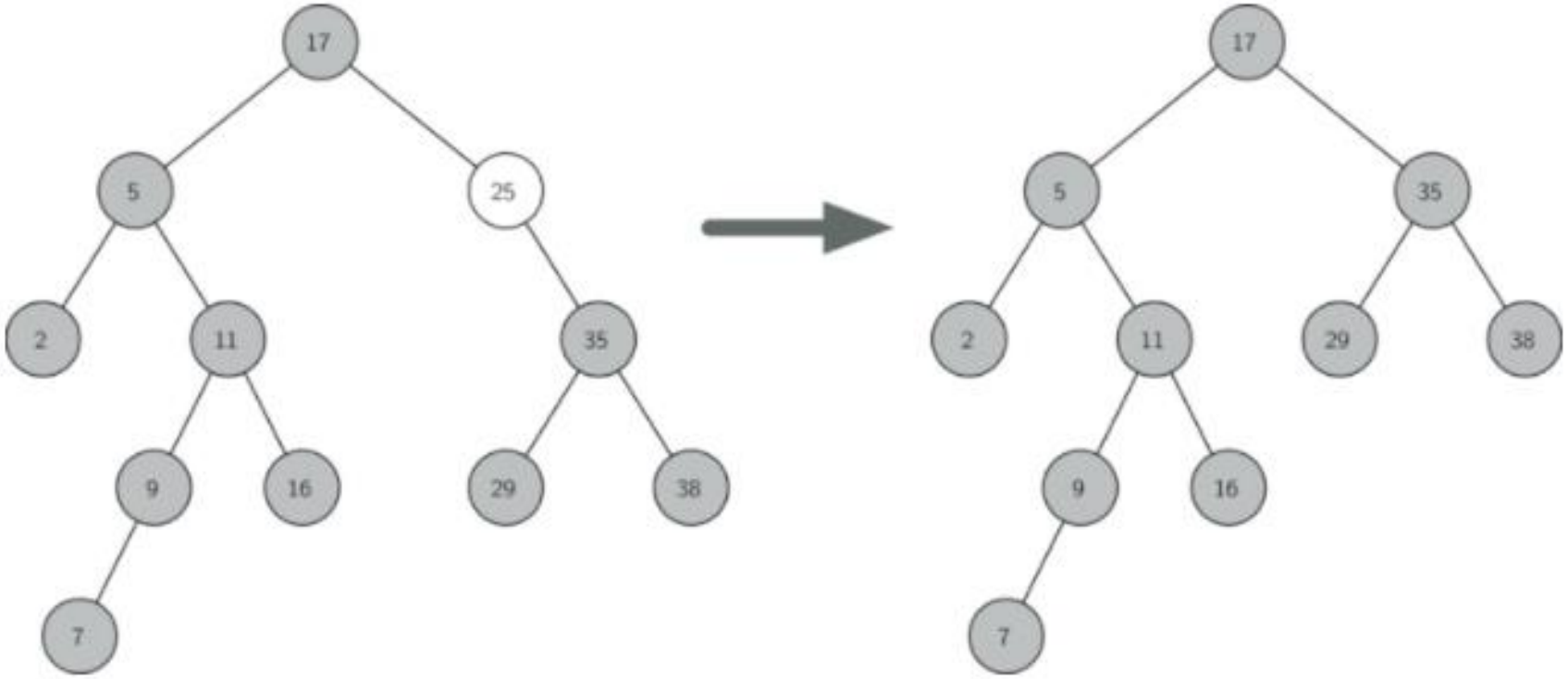



Figure: Delete Node 25 which has only a single child.

# Binary Search Tree Class: Delete Node (3)

```
else: # this node has one child
  if currentNode.hasLeftChild():
    if currentNode.isLeftChild():
        currentNode.leftChild.parent = currentNode.parent
        currentNode.parent.leftChild = currentNode.leftChild
    elif currentNode.isRightChild():
        currentNode.leftChild.parent = currentNode.parent
        currentNode.parent.rightChild = currentNode.leftChild
    else:
        currentNode.replaceNodeData(
           currentNode.leftChild.key,currentNode.leftChild.payload,
           currentNode.leftChild.leftChild,currentNode.leftChild.rightChild)
  else:
    if currentNode.isLeftChild():
        currentNode.rightChild.parent = currentNode.parent
        currentNode.parent.leftChild = currentNode.rightChild
    elif currentNode.isRightChild():
        currentNode.rightChild.parent = currentNode.parent
        currentNode.parent.rightChild = currentNode.rightChild
    else:
        currentNode.replaceNodeData(
          currentNode.rightChild.key,currentNode.rightChild.payload,
          currentNode.rightChild.leftChild,currentNode.rightChild.rightChild)
```

# Delete node from Binary Tree

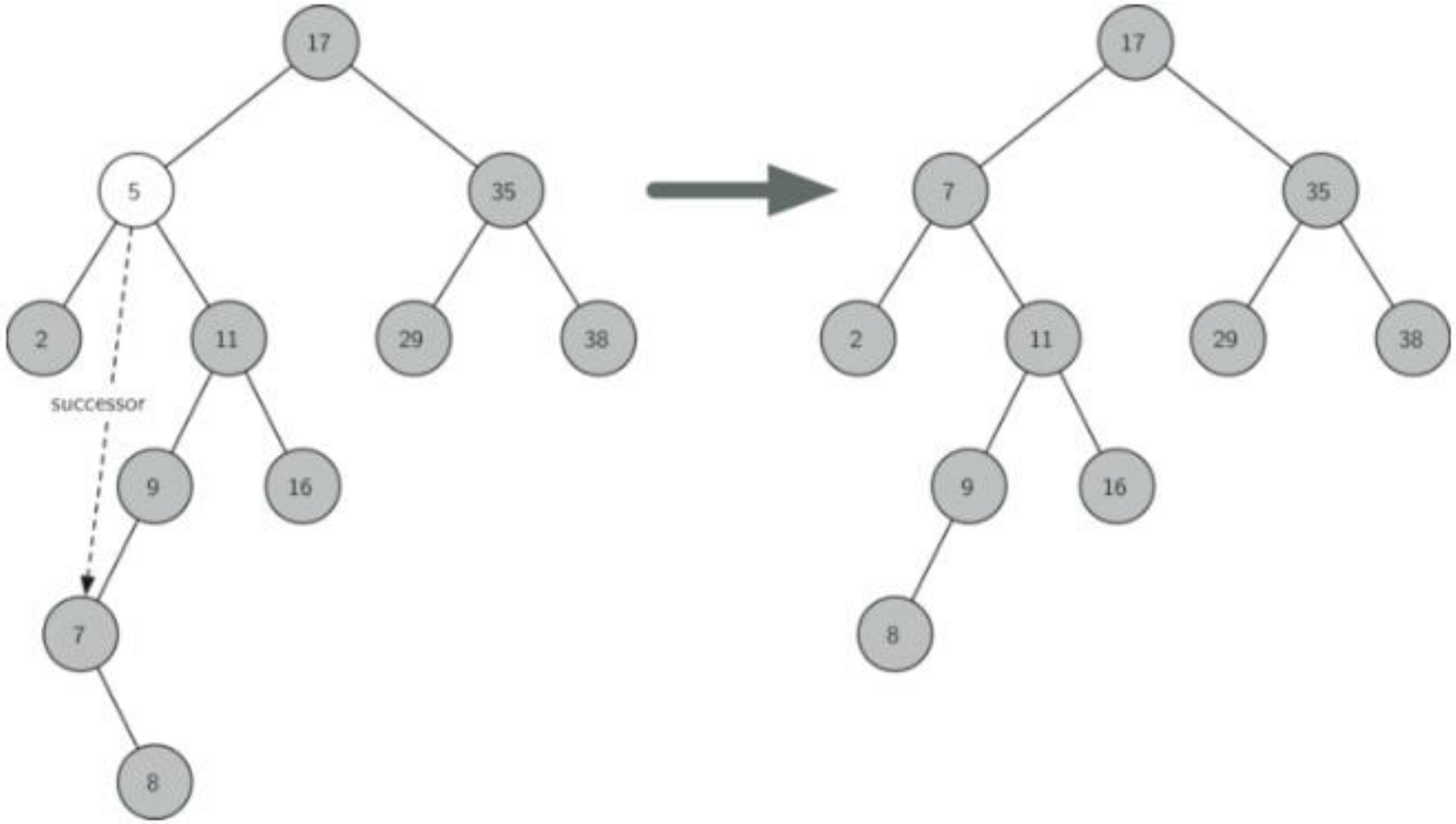



Figure: Delete Node 5 which has a two child.

# Binary Search Tree Class:  Delete Node (4)

```
elif currentNode.hasBothChildren(): #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
```

# Binary Search Tree Class

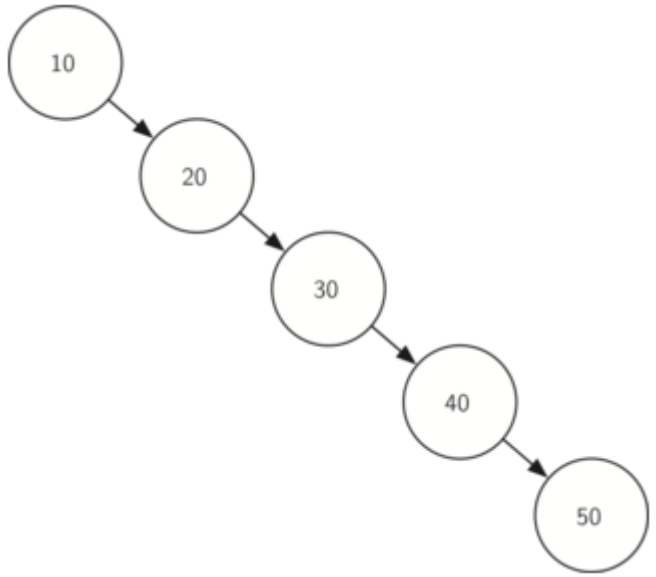
```
mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(mytree[6])
print(mytree[2])
print(10 in mytree)
print(3 in mytree)
del mytree[3]
print(3 in mytree)
print (len(mytree))
```

**Output:**

```
yellow
at
False
True
False
3
```

# Binary Search Tree Analysis

- The search algorithm computation complexity is implicitly related to the performance of the put Method.
- A perfectly balanced tree has the same number of nodes in the left subtree as the right subtree.
- Hence, the worst-case performance of put is O($log_2$ n), where n is the number of nodes in the tree.
- When adding new node, the maximum number of comparisons need in put method is $log_2$ n
- Nevertheless, it is possible to construct a search tree that has height n.
- It is simply inserting the keys in sorted order, for example, the figure in this slide.
- Hence, the performance of the put method in this case is is O(n) .



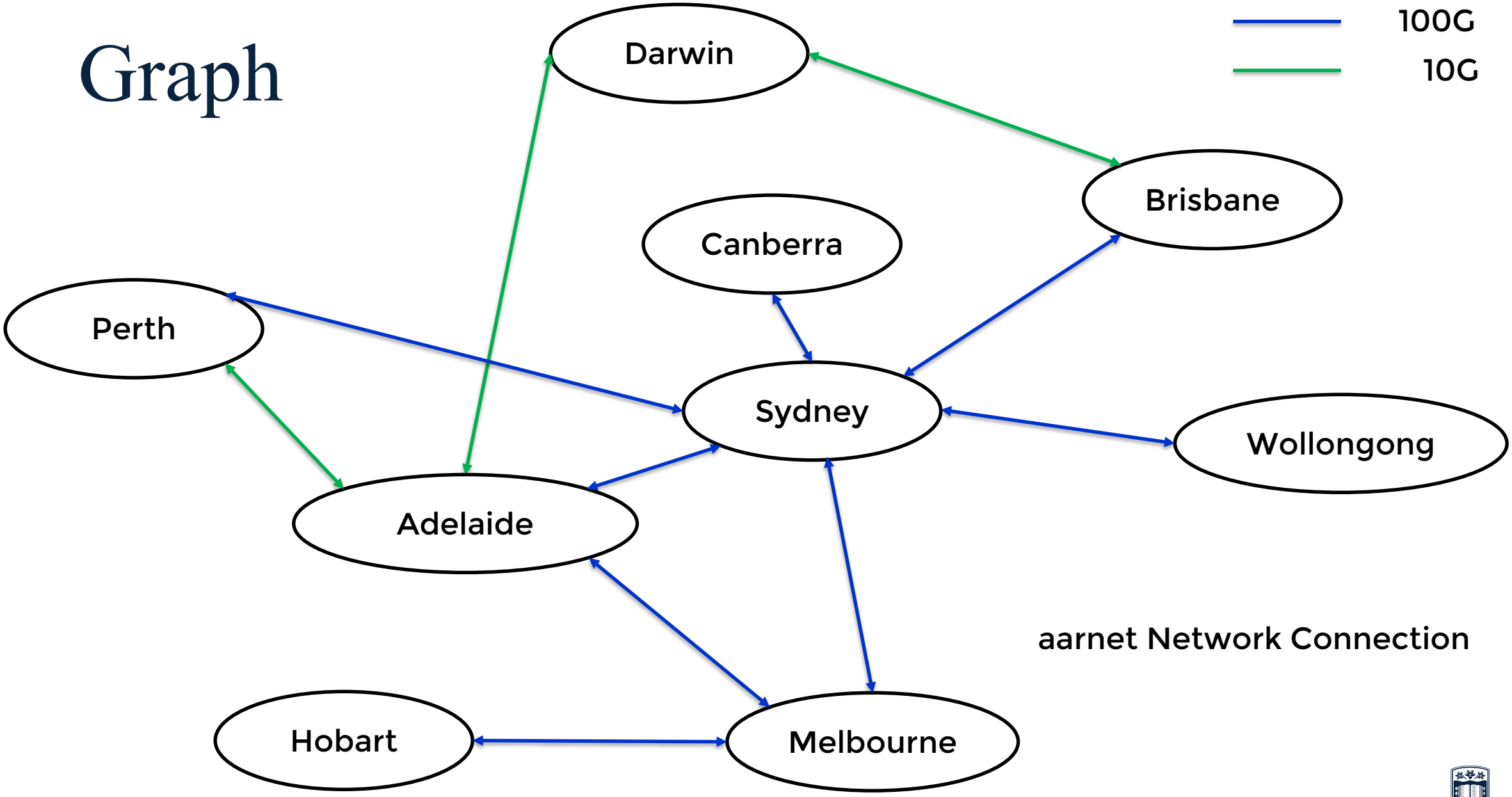
U O W

# Graphs

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

# Graph

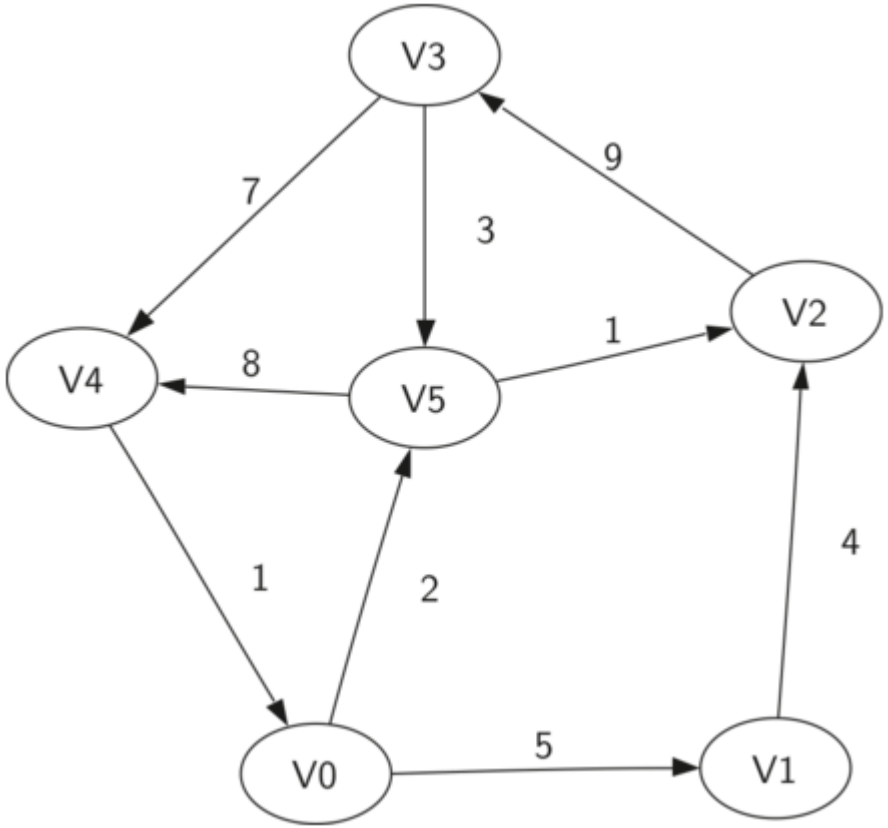

100G
10G
Darwin
Brisbane
Canberra
Perth
Sydney
Wollongong
Adelaide
Hobart
Melbourne
aarnet Network Connection

# Definitions

- Vertex
- Edge
- Directed graph or Digraph
- Graph G =( V, E )
- Weight
- Path
- Cycle

$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$E = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$



# Vertex Class

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight
    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])
    def getConnections(self):
        return self.connectedTo.keys()
    def getId(self):
        return self.id
    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

# Graph Class

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex
    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
```

# Graph Class

```
def __contains__(self,n):
    return n in self.vertList
def addEdge(self,f,t,weight=0):
    if f not in self.vertList:
        nv = self.addVertex(f)
    if t not in self.vertList:
        nv = self.addVertex(t)
    self.vertList[f].addNeighbor(self.vertList[t], weight)
def getVertices(self):
    return self.vertList.keys()
def __iter__(self):
    return iter(self.vertList.values())
```

# Graph Class

```
g = Graph()
for i in range(6):
    g.addVertex(i)
print(g.vertList)
g.addEdge(0,1,5)
g.addEdge(0,5,2)
g.addEdge(1,2,4)
g.addEdge(2,3,9)
g.addEdge(3,4,7)
g.addEdge(3,5,3)
g.addEdge(4,0,1)
g.addEdge(5,4,8)
g.addEdge(5,2,1)
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
```

Output:

```
{0: <__main__.Vertex object at 0x7fb8d67e1470>,
1: <__main__.Vertex object at 0x7fb8d67e15c0>,
2: <__main__.Vertex object at 0x7fb8d67e1400>,
3: <__main__.Vertex object at 0x7fb8d67e1550>,
4: <__main__.Vertex object at 0x7fb8d67e1588>,
5: <__main__.Vertex object at 0x7fb8d67e15f8>}
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```

# Breadth-First Search

CSIT-881 Python and Data Structures

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

# Breadth-First Search

- The graph algorithm we are going to use is called the “breadth first search” algorithm.
- Breadth first search (BFS) is one of the easiest algorithms for searching a graph. It also serves as a prototype for several other important graph algorithms.
- 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.
- One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time.
- A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.

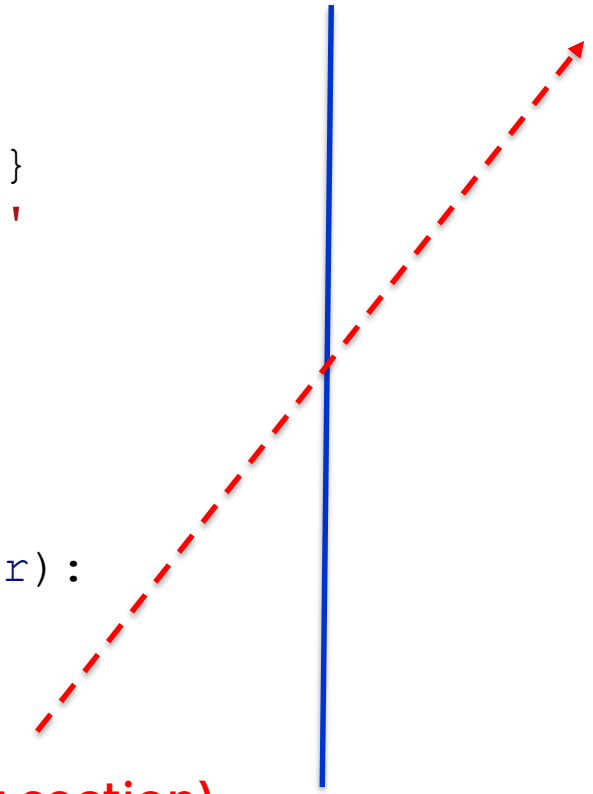
# Breadth-First Search

- To keep track of its progress, BFS colours each of the vertices white, grey, or black.
- All the vertices are initialized to white when they are constructed.
- A white vertex is an undiscovered vertex.
- When a vertex is initially discovered it is coloured grey, and when BFS has completely explored a vertex it is coloured black.
- This means that once a vertex is coloured black, it has no white vertices adjacent to it.
- A grey node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore.
- Hence, It goes through all the node, level by level until none of the node is paint in white or grey.

# Breadth-First Search Function

We first modify some codes in graph.py

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
        self.colour = 'white'
        self.distance=0
        self.pred = None

      ....

    def setColour(self,colour):
        self.colour =colour
```



```
def getColour(self):
    return self.colour

def setDistance(self,distance):
    self.distance=distance
def getDistance(self):
    return self.distance
def setPred(self,pred):
    self.pred=pred
def getPred(self):
    return self.pred
```

(Continue the code in the left section)

# Breadth-First Search Function

```
from graph.py import Graph, Vertex   # from the previous examples (graph.py)
from queue.py import Queue  # from the previous examples (queue.py)

def bfs(g,start):
  list1 = []
  start.setDistance(0)
  start.setPred(None)
  vertQueue = Queue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
        nbr.setColor('gray')
        nbr.setDistance(currentVert.getDistance() + 1)
        nbr.setPred(currentVert)
        vertQueue.enqueue(nbr)
    currentVert.setColor('black')
    list1.append(currentVert)
  return list1
```

# Breadth-First Search Function

```
g = Graph()
for i in range(6):
    g.addVertex(i)
g.addEdge(0,1,5)
g.addEdge(0,5,2)
g.addEdge(1,2,4)
g.addEdge(2,3,9)
g.addEdge(3,4,7)
g.addEdge(3,5,3)
g.addEdge(4,0,1)
g.addEdge(5,4,8)
g.addEdge(5,2,1)
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
start= g.getVertex(0)
list1 = bfs(g,start)
for i in list1:
    print(i)
```

**Output:**

```
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
0 connectedTo: [1, 5]
1 connectedTo: [2]
5 connectedTo: [4, 2]
2 connectedTo: [3]
4 connectedTo: [0]
3 connectedTo: [4, 5]
```