ESC190 Exam Review:
Data Structures and Algorithms

QiLin Xue

April 18, 2021

Contents

1 Data Structures

1.1 Linked List

A linked list uses *pointers* to link from one object to another. In C, this is a practical way of describing lists that allow for inserting and deleting new entries, which we will call nodes. This is because we do not need to allocate this memory beforehand, which is especially handy if we do not know how large our list will be. You should be familiar with how to implement the following functions in C related to linked lists:

- Creating a linked list
- Deleting the linked list
- Appending a node at a certain position
- Deleting a node at a certain position
- Finding a node which has a specific value
- Looking to see if there is a loop

The last node of a linked list should always point to the `null` pointer.

1.2 Stacks

Stacks retrieve information in a *last* in, *first* out order.

```
stack = []

# Add in entries
stack.append('a')
stack.append('b')
stack.append('c')

# The last entry added was 'c' so the first entry that comes out will be 'c'
out = stack.pop()
print(out) # 'c'
```

1.3 Queues

Queues retrieve information in a *first* in, *first* out order

```
queue = []

# Add in entries
queue.append('a')
queue.append('b')
queue.append('c')

# The first entry added was 'a' so the first entry that comes out will be 'a'
out = queue.pop(0)
print(out) # 'a'
```

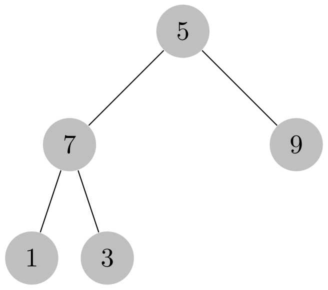
1.4 Priority Queues

Priority queues are similar to regular queues, except all entries are flagged with a specific weight. There are three primary operations:

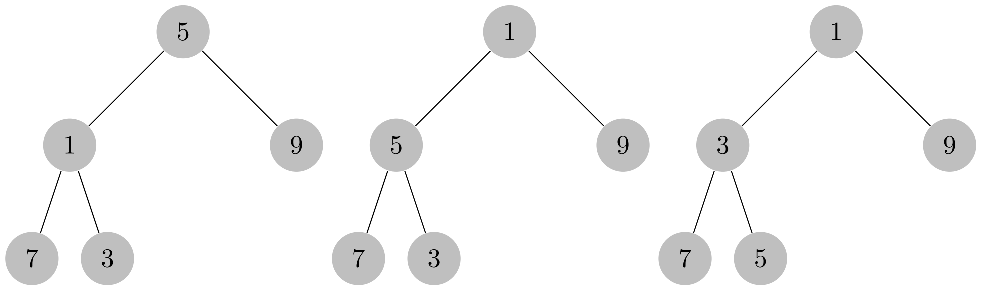
- Inserting an entry and its weight into a priority queue.
- Find the entry with the minimum / maximum weight
- Retrieve the entry with the minimum / maximum weight

1.5 Heaps

A heap is used to implement a priority queue. It takes the form of a binary tree, which can be read as a list. The only other condition for a heap is that the parent is smaller than both children. For example, $[1, 3, 9, 7, 5]$ refers to the following tree:



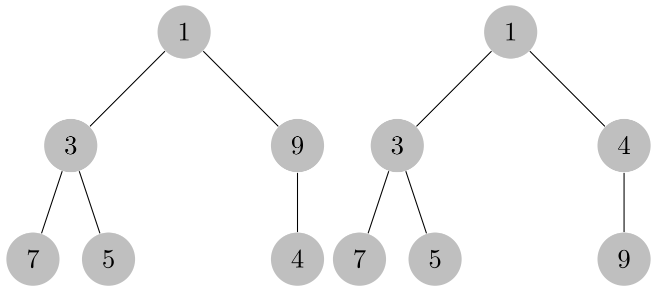
We can denote nodes that do not have any children as **leaves**. To turn this into a heap, we start at the topmost parent, and we see if it is smaller than both its children. Since this is true, we move on to its first child 7 , which is bigger than both its children. As a result, we swap it with the smallest of its two children 1 , and we start over the same process, going back to 5 . Both BFS (see 2.3.1) and DFS (see 2.3.2) can be used. The following illustrates the step by step tree diagram representations of the heap.



We can use Python to verify this:

```python
import heapq
heap = [5, 7, 9, 1, 3]
heapq.heapify(heap)
print(heap) # [1, 3, 9, 7, 5]
```

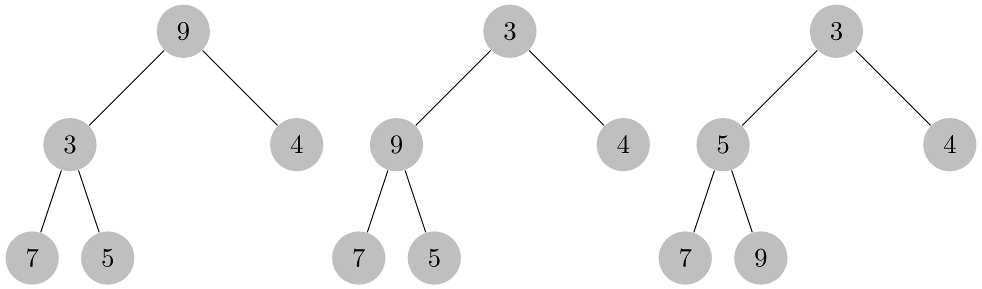
To add an element, we first place it at the next position of the tree (or the last position of the heap). We then "percolate" it up, comparing it with the parent until it is in the right spot. For example, suppose we wish to add 4 . Then the process will look like:



In Python, we can represent this as:

```python
heapq.heappush(heap,4)
print(heap) # [1, 3, 4, 7, 5, 9]
```

Since this is a priority queue, when we remove an element, we remove the smallest element which will always be at the very top. We can accomplish this by swapping this with the last element, and bubbling that element down. For example, suppose we wish to remove the smallest element 1 :



Using Python, we have:

```
heapq.heappop(heap)
print(heap) # [3, 5, 4, 7, 9]
```

1.5.1 Manual Implementation

Using this Python module makes working with heaps trivial and allows us to easily use a list. However, it is also possible to implement this manually with the following mathematical operations. Let the index of an element be i. Then:

- Index of parent: floor $\left(\frac{i}{2}\right)$
- Index of left child: $2i$
- Index of right child: $2i+1$

For convention, let $i=0$ correspond to a `Null` element. The manual implementation of heapify using Python is shown below:

```
# TBA
```

1.5.2 Runtime

If there are n entries, then we will percolate downwards on average $n/2$ times. The time complexity of percolating downwards is $\mathcal{O}(\log n)$, so the time complexity of heaipfying a list is $\mathcal{O}(n \log n)$.

However, it is possible to gain a better upper bound. We know that there are at least $n/2$ leaves in the heap. There are $n/4$ nodes at height 1 (let the bottommost layer be $h=0$). There are $n/8$ nodes at height 2, and so forth. Therefore at a height h, we perform:

$$N_h \propto h \cdot \frac{n}{2^{h+1}}$$

operations such that the total number of operations is proportional to:

$$N = \sum_{h=1}^{\log_2 n} h \cdot \frac{n}{2^{h+1}} \propto h$$

so the total runtime is $\mathcal{O}(n)$.

1.5.3 Heapsort

We can implement a heap to sort a list with a time complexity of $\mathcal{O}(n \log n)$. We perform the following steps:

1. Create a heap from the list.
2. Extract the minimum a total of n times, and placing the elements into a list in the original order.

2 Graph Theory

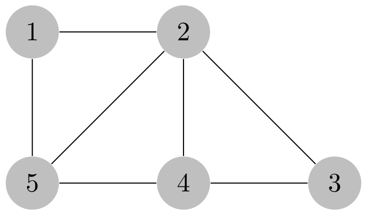
2.1 Background

- We can represent a graph as $G=(V,E)$ where V is the set of nodes $V=\{v_1, v_2, \ldots v_n\}$ and E is the set of edges $E=\{e_1, e_2, \ldots, e_m\}$.

- Each edge can be written as $e_k = (v_i, v_j)$.
- Directed graphs, or **digraphs** have edges with directions associated with them.
- Weighted graphs have a weight associated with each edge.
- Vertex v_1 is **adjacent** to vertex v_2 if an edge connects them.
- A *path* is a sequence of vertices in which each vertex is adjacent to the next one. The length of the path is the number of edges in it.
- A cycle in a path is a sequence $(v_1, \dots, v_n)$ such that $(v_i, v_{i+1}) \in E$ and $(v_n, v_1) \in E$.
- A graph with no cycles is known as **acyclic**.
- A directed graph which is acyclic is known as a **DAG**.
- A **simple path** and a **simple cycle** has no repeated vertices.
- Two vertices are **connected** if there is a path between them.
- A subset of vertices is a connected component of a graph G if each pair of vertices in the subset are connected.
- The **degree** of vertex v is the number of edges associated with v.

2.2 Representing Graphs

Suppose we take the following graph:



2.2.1 Adjacency Matrix

For the above graph, the adjacency matrix will look like:

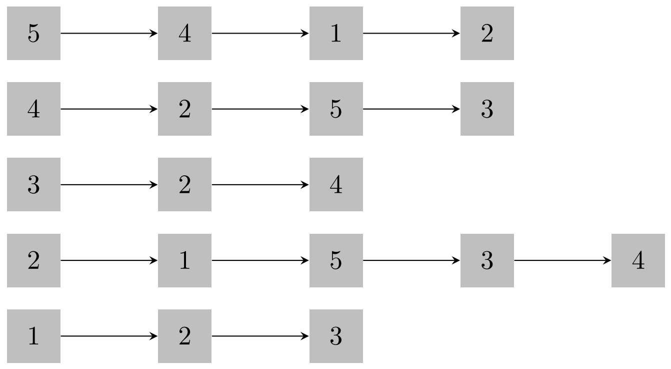
$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

To look up if 1 is connected to 2, we just need to look up `A[2,1] = A[1,2]` (or in terms of Python notation, `A[2][1]` or `A[1][2]`). Note that adjacency matrices are symmetric (e.g. $A^T = A$) for undirected graphs. It has the following complexities:

- Edge lookup: $\mathcal{O}(1)$
- Finding all vertices adjacent to vertex: $\mathcal{O}(|V|)$
- Space complexity: $\mathcal{O}(|V|^2)$

2.2.2 Adjacency List

An adjacency list uses linked lists (see sec. 1.1) to represent the graph:

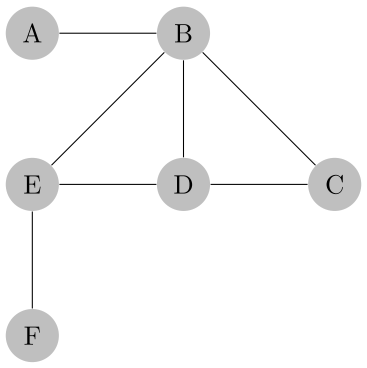


This has the following complexities:

- Edge lookup: $\mathcal{O}(d)$ where d is the maximum degree in the graph
- Finding all vertices adjacent to vertex: $\mathcal{O}(d)$
- Space complexity: $\mathcal{O}(|V| + |E|)$

2.3 Traversal Algorithms

We will apply both DFS and BFS to traverse the following graph:



We can represent this graph using Python:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False

def connect(node1, node2):
    node1.connections.append(node2)
    node2.connections.append(node1)

A = Node("A")
B = Node("B")
C = Node("C")
D = Node("D")
E = Node("E")
F = Node("F")

connect(A,B)
connect(E,F)
connect(B,E)
connect(E,D)
connect(B,D)
connect(B,C)
connect(D,C)
```

2.3.1 Breadth First Search

To implement BFS, we use a queue. Suppose we start our search from C. We would explore the nodes in the following order:

$$C \to B \to D \to A \to E \to F$$

We visit nodes in the order of the queue and every time a node is visited, the unvisited neighbours of that node is added to the back of the queue. See this Python implementation:

```python
def BFS(node):
    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop(0) # remove q[0] from q and put it in cur
        print(cur.name)
        for con in cur.connections:
            if not con.visited:
                q.append(con)
                con.visited = True

BFS(C) # C B D A E F
```

2.3.2 Depth First Search

In a depth first search, we make use of a stack instead. Nodes are visited in the order of the stack, and every time a node is visited, its unvisited neighbours are added to the stack. Starting from C, the nodes are explored in the following order:

$$C \to D \to E \to F \to B \to A$$

which can be shown using Python:

```python
def DFS(node):
    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop() # remove last element from q and put it in cur
        print(cur.name)
        for con in cur.connections:
            if not con.visited:
                q.append(con)
                con.visited = True

DFS(C) # C D E F B A
```

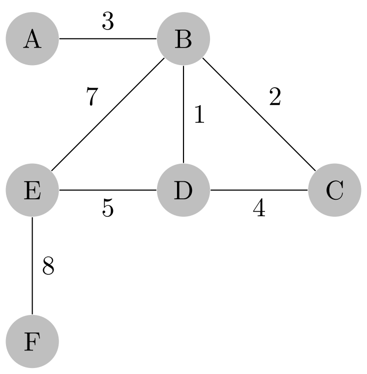
Alternatively, we can use recursion:

```python
def DFS_rec(node):
    '''Print out the names of all nodes connected to node using a
    recursive version of DFS'''
    print(node.name)
    node.visited = True
    for con in node.connections:
        if not con.visited:
            DFS_rec(con)

DFS_rec(C) # C B A E F D
```

Note that the nodes are traversed in a different order, since the implementation is slightly different. However, it is still DFS.

3 Shortest Path Algorithm

The general problem here is that we want to find the shortest path between two points through a graph. To do this, we can create a weighted graph, such as below:



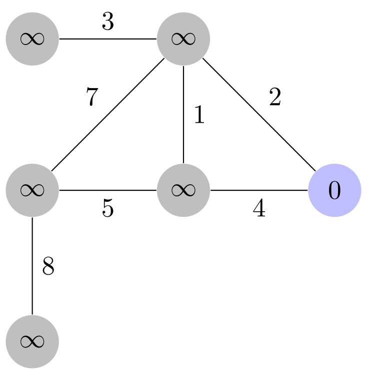
The shortest path from C to F is $C \to B \to D \to E \to F$, but there are several paths. A weighted graph can be represented using Python as:

```python
class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False


def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A")
B = Node("B")
C = Node("C")
D = Node("D")
E = Node("E")
F = Node("F")

connect(A,B,3)
connect(E,F,8)
connect(B,E,7)
connect(E,D,5)
connect(B,D,1)
connect(B,C,2)
connect(D,C,4)
```

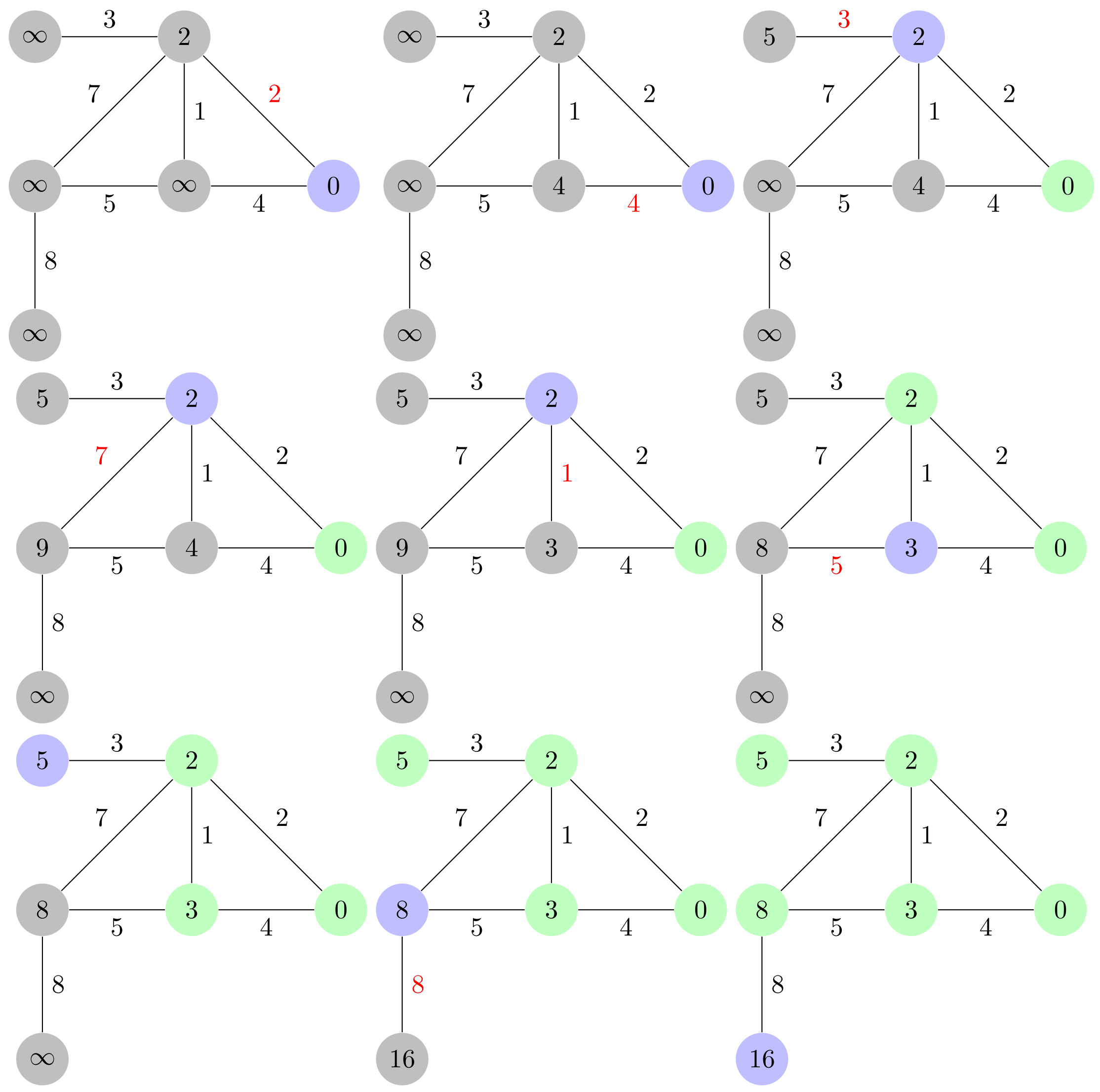
Search functions (e.g. BFS and DFS) can be slightly modified for this slightly different data type.

3.1 Dijkstra's Algorithm

The simplest method is to apply Dijkstra's Algorithm. There are several optimizations but we will look at the simplest. The overall idea is to apply to traverse through each unexplored node. For each unexplored node, the shortest distance to each of its unexplored neighbours are set. Initially, the distance from each node to the starting node (except the starting node) is infinite:



Here, a blue node represents the node we are currently exploring and nodes that we have already explored will be drawn as green.



We can show this below via Python:

```
def get_all_nodes(node):
    connections = []

    q = [node]
    node.visited = True
    while len(q) > 0:
        cur = q.pop(0) # remove q[0] from q and put it in cur
        connections.append(cur)
        for con in cur.connections:
            if not con["node"].visited:
                q.append(con["node"])
                con["node"].visited = True

    return connections

def dijsktra(node):

    S = [node] # Visited Nodes
    d = {node.name: 0} # Dictionary of distances to nodes
    prev = {}

    unexplored = get_all_nodes(node) # Unexplored nodes (S + unexplored = all nodes)

    # Set all nodes to infinity
    for n in unexplored:
        if n.name not in d:
            d[n.name] = 999999
    # Continue until everything is explored
    while len(unexplored) > 0:
        cur = unexplored.pop(0)
        for con in cur.connections:
            if con["node"] not in S:
                if con["weight"] + d[cur.name] < d[con["node"].name]:
                    d[con["node"].name] = con["weight"] + d[cur.name]

        S.append(cur)

    return d

print(dijsktra(C)) # {'C': 0, 'B': 2, 'D': 3, 'A': 5, 'E': 8, 'F': 16}
```

To obtain the path, we can implement a dictionary `prev` and whenever the path to a certain node improves, we call: `prev[con["node"].name] = cur.name`.

To add one vertex to S, we must search through all possible vertices so the runtime complexity is $\mathcal{O}(|V|^2)$.

3.2 Using Priority Queue

It is possible to improve the runtime to $\mathcal{O}(|E|\log|V|)$ using a priority queue:

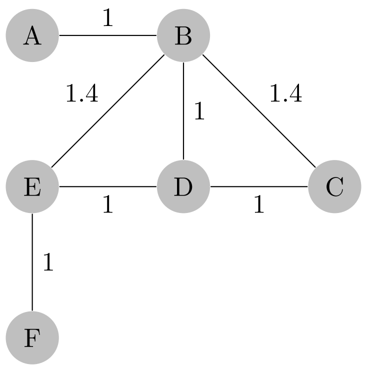
```
import heapq

class Node:
    def __init__(self, name):
        self.name = name
        self.connections = []
        self.visited = False
        self.distance = 999999

    def __lt__(self, other):
        # Comparison function for priority queue
        return self.distance < other.distance

A = Node("A"); B = Node("B"); C = Node("C"); D = Node("D"); E = Node("E"); F = Node("F")

connect(A,B,3); connect(E,F,8); connect(B,E,7); connect(E,D,5)
connect(B,D,1); connect(B,C,2); connect(D,C,4)

def dijsktra_pq(node):
    node.distance = 0

    S = [] # Visited Nodes
    pq = [node]
    heapq.heapify(pq)

    d = {node.name: 0} # Dictionary of distances to nodes

    # Continue until everything is explored
    while len(pq) > 0:
        cur = heapq.heappop(pq)

        if cur in S:
            continue

        d[cur.name] = cur.distance

        for con in cur.connections:
            con["node"].distance =  cur.distance + con["weight"]
            heapq.heappush(pq, con["node"])

        S.append(cur)

    return d

print(dijsktra_pq(C)) # {'C': 0, 'B': 2, 'D': 3, 'A': 5, 'E': 8, 'F': 16}
```

This algorithm is essentially the same, but instead of traversing explored nodes via BFS, we traverse it by sorting the nodes by their distance from the origin. The upper bound on the number of times a node is pushed is $2|E|$ and popping/pushing into `pq` has a time complexity of $\mathcal{O}(\log|V|)$. The time complexity of this algorithm is thus $\mathcal{O}(|E|\log|V|)$.

3.3 Greedy Best First Search

The general theme of search algorithms is to select a way to traverse explored nodes. In the simple method, it was via BFS. In the priority queue method, it was prioritizing nodes with a smaller distance. In the greedy best first search, it is by using a heuristic function $h(v)$ that gives a rough estimate of the distance between a node and the destination.

For example, take the below graph and suppose we wish to move from C to A. Unlike the previous examples, the length of each path corresponds to their weights, so we can imagine these as physical points in space. We can let our heuristic function be the *Manhattan distance* $\Delta x + \Delta y$:



We will start from C and look at the neighbours. The neighbour that minimizes the Manhattan distance is B, which directly leads to A. As you can expect, this is *really bad*. It doesn't even bother checking if going to D is faster, but if we have a good heuristic, this isn't necessary. We can implement it in Python below:

```python
class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.connections = []
        self.visited = False
        self.x = x
        self.y = y
        self.distance = 999999

def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A", 0, 0)
B = Node("B", 1, 0)
C = Node("C", 2, -1)
D = Node("D", 1, -1)
E = Node("E", 0, -1)
F = Node("F", 0, -2)

connect(A,B,1); connect(E,F,1); connect(E,D,1)
connect(B,D,1); connect(D,C,1); connect(B,C,3); connect(B,E,1.4)

def h(con):
    v = con["node"]
    return abs(v.x)+abs(v.y)

def greedy(source, dest):

    v = source
    S = [] # Visited Nodes
    d = {v.name: 0}

    # Continue until destination is reached
    while not v == dest:

        con = sorted(v.connections, key=h)[0]
        con["node"].distance = v.distance + con["weight"]
        d[con["node"].name] = con["weight"] + d[v.name]

        v = con["node"]
        S.append(v)

    return d

print(greedy(C, A)) # {'C': 0, 'B': 1.4, 'A': 2.4}
```

3.4 A*

The A* algorithm is more robust, and combines the advantages of using a heuristic function with a priority queue. It is extremely similar to the optimized version of dijkstra's, except the priority queue is sorted by the heuristic function. We can also implement it in Python:

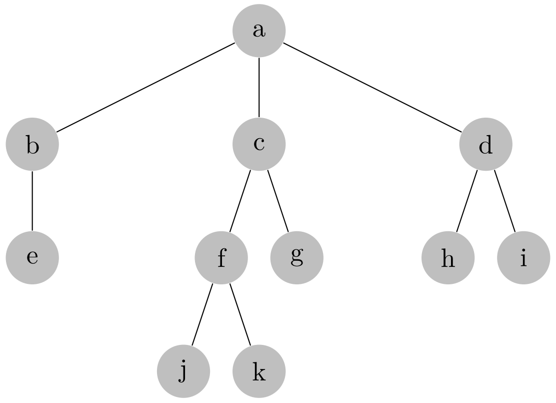
```
import heapq

class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.connections = []
        self.visited = False
        self.x = x
        self.y = y
        self.distance = 999999
        self.estimate = abs(self.x) + abs(self.y)

    def __lt__(self, other):
        # Comparison function for priority queue
        return self.estimate < other.estimate

def connect(node1, node2, weight):
    node1.connections.append({"node": node2, "weight": weight})
    node2.connections.append({"node": node1, "weight": weight})

A = Node("A", 0, 0)
B = Node("B", 1, 0)
C = Node("C", 2, -1)
D = Node("D", 1, -1)
E = Node("E", 0, -1)
F = Node("F", 0, -2)

connect(A,B,1); connect(E,F,1); connect(E,D,1)
connect(B,D,1); connect(D,C,1); connect(B,C,1.4); connect(B,E,1.4)

def h(v1, v2):
    return abs(v1.x - v2.x)+abs(v1.y - v2.y)

def greedy(source, dest):
    source.distance = 0

    S = [] # Visited Nodes
    pq = [source]
    heapq.heapify(pq)

    d = {source.name: 0} # Dictionary of distances to nodes

    # Continue until destination is reached
    while len(pq) > 0:
        cur = heapq.heappop(pq)

        if cur in S:
            continue

        d[cur.name] = cur.distance

        for con in cur.connections:
            con["node"].distance =  cur.distance + con["weight"]
            con["node"].estimate =  h(con["node"], dest) + con["node"].distance
            heapq.heappush(pq, con["node"])

        S.append(cur)
    return d

print(greedy(C, E)) # {'C': 0, 'B': 1.4, 'A': 2.4}
```

4 Trees

A *tree* is a collection of nodes and directed edges. Here are some additional definitions not defined above:

- The **length** of a path is the number of edges it contains, not the number of nodes.
- For each node n, the **depth** of n is the length of the unique path from the root to n.
- The **height** of n is the length of the longest path from n to a leaf.
- The height of a tree is equal to the height of the root node of a tree.
- The **branching factor** is the maximum number of children in any of its nodes.
- A **level** is the set of all nodes in the tree at a given depth.
- A **complete** tree is one where all levels are full except the bottom level, which has been filled from left to right.
- A **full tree** is a complete tree whose last level has been filled completely.

For example, consider the following graph:



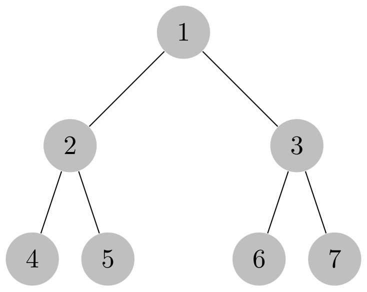
A **binary tree** is a tree with a branching factor of 2.

- The path c $\rightarrow$ f $\rightarrow$ k has length 2.
- The depth of f and the depth of g are both 2.
- The height of f is 1 and the height of g is 0.
- The depth of a is 0.
- The height of the tree is 3.

We will implement a tree from scratch, using Python. For simplicity, I will consider a binary tree (such that the same code can be used / expanded for future sections)

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

if 1:                   A = Node(1)
#                      /           \
if 1:      B = Node(2);             C = Node(3)
#         /       \                 /          \
D = Node(4);     E = Node(5);  F = Node(6);     G = Node(7)

A.left = B; A.right = C
B.left = D; B.right = E
C.left = F; C.right = G
```

which represents (if you can't tell from the cleverly structured code) the graph:



4.1 Traversal

Oftentimes, we wish to visit all the nodes in the tree in a particular order, and then apply some operation to each node. There are two types of traversal: preorder and postorder.

For **preorder**, we perform an operation on the node first, then recursively do the same thing to its children. Suppose we wish to print out all the nodes. Then, we can write the function `pre_visualize`

```
def pre_visualize(root):
    print(root.data)
    for node in [root.left, root.right]:
        if node is not None:
            pre_visualize(node)
pre_visualize(A)
```

In the above example, we get the output: 1, 2, 4, 5, 3, 6, 7. However, we can print the node *after* we recursively call the function. This is what **postorder** does:

```
def post_visualize(root):
    for node in [root.left, root.right]:
        if node is not None:
            post_visualize(node)
    print(root.data)
post_visualize(A)
```

which outputs: 4, 5, 2, 6, 7, 3, 1.

Specifically for a binary tree, we can traverse it in **inorder**, where we perform the operation in between the recursive calls:

```
def inorder_visualize(node):
    if node.left is not None:
        inorder_visualize(node.left)
    print(node.data)
    if node.right is not None:
        inorder_visualize(node.right)

inorder_visualize(A)
```

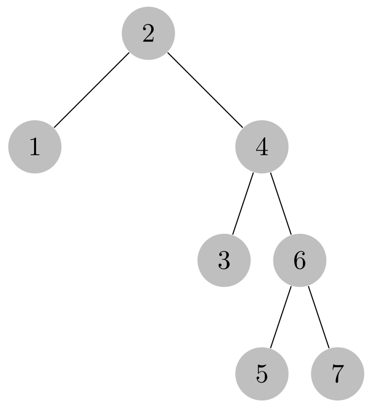
which outputs: 4, 2, 5, 1, 6, 3, 7.

4.2 Binary Search Tree

If we want to find a node with a specific value in a random tree, the worst case scenario is $O(n)$. However, we can improve this by using a **binary search tree**. The conditions for a tree to be a binary search tree are:

- Must be a binary tree.
- All nodes in the left subtree of a node with value k have a value less than k
- All nodes in the right subtree of a node with value k have a value greater than k

This means that all values are different. For example, an example of a binary search tree (though not unique) would be:



which is represented below:

```
if 1:                   B = Node(2)
#                      /           \
if 1:           A = Node(1);     D = Node(4)
#                               /          \
if 1:                     C = Node(3);   F = Node(6)
#                                        /         \
if 1:                           E = Node(5);     G = Node(7)

B.left = A; B.right = D
D.left = C; D.right = F
F.left = E; F.right = G
```

4.3 Searching

We can search this tree recursively:

```
def search_BST(node, value):
    if node == None: return None # Not Found
    if value == node.data: return node # Found
    if value < node.data: return search_BST(node.left, value) # Search Left
    if value > node.data: return search_BST(node.right, value) # Search Right
```

If the height is h, then the time complexity is $\mathcal{O}(h)$. However, the worst case scenario is still $\mathcal{O}(n)$. This is because $n = h$ is possible if we go from smallest to largest as we go from top to bottom.

4.3.1 Insertion

We can insert into a BST in a similar manner, where we traverse down, attempting to locate a certain value, and when we hit a null object, we insert it in that position. For example, in the below example, we insert 0 into our binary search tree:

```
def insert_BST(node, value):
    if value < node.data:
        if node.left == None:
            node.left = Node(value)
        else:
            insert_BST(node.left, value)

    if value > node.data:
        if node.right == None:
            node.right = Node(value)
        else:
            insert_BST(node.right, value)

insert_BST(B, 0)
```

Idea: Here, the C implementation may be even cleaner with the help of pointers. To implement a tree, the children of each node aren't stored as "objects," but instead they are stored as addresses. Therefore, all we need to do is recursively call `insert_BST` depending on whether the value is smaller or larger than each node, and when we hit the null pointer, we write in the desired node at that address. This approach *does not work* in Python (to the best of my knowledge).

This general pattern will continue, though for AVL trees, we will implement it in *C*.

4.3.2 Deletion

To delete a node, we again traverse the graph until we find the node we wish to delete. Unlike insertion, this isn't always a leaf node. Therefore, it is more complicated and there are three cases:

- **Case 1: No Children:** Easiest case. Simply remove it.
- **Case 2: One Child:** Replace the current node with the child.
- **Case 3: Two Children:** Do not delete the node. Instead, swap the node with its successor and delete the node from its new location.

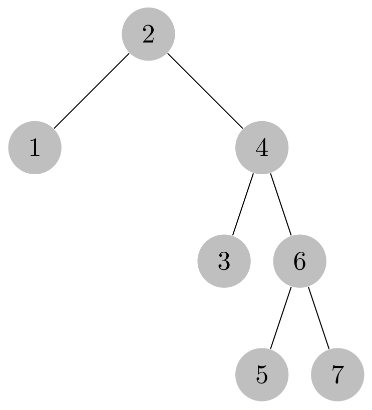
The **successor** is the next larger value in the tree such that it is larger or equal to the left child, and smaller or equal to the right child. We can find the successor by finding the smallest value in the right subtree:

```python
def successor_BST(node):
    cur_node = node.right
    while cur_node.left is not None:
        cur_node = cur_node.left
    return cur_node

print(successor_BST(D).data)
```

Here, we attempt to find the successor to 4, which happens to be 5. Using this, we can now implement deletion using the casework described above.

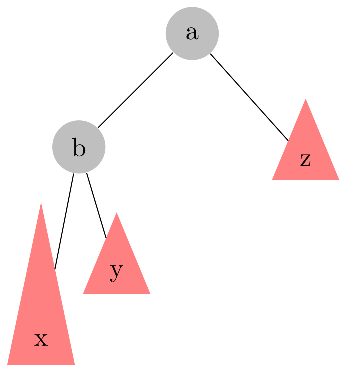
4.4 AVL Trees

AVL trees are a **balanced binary search tree**, with a *balance condition*: For every node, the height of the left and right subtree differs by at most 1. This is to ensure that we keep the height to be approximately $h \approx \log_2(n)$. The implementation is very similar to a regular binary search tree, except after inserting and deleting, we tweak the graph to balance it. For example, the following tree (that we saw above) is *not* balanced:

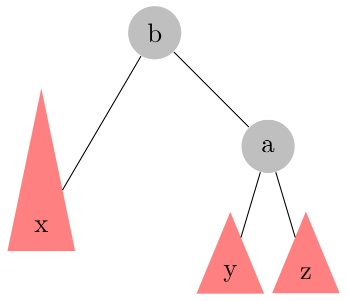


4.5 Balancing Outer Imbalances

For purposes of generalization, we will use x to represent a subtree. For example, the following is a graph where the left subtree of the left child is too high:



This can be fixed by performing a single rotation:



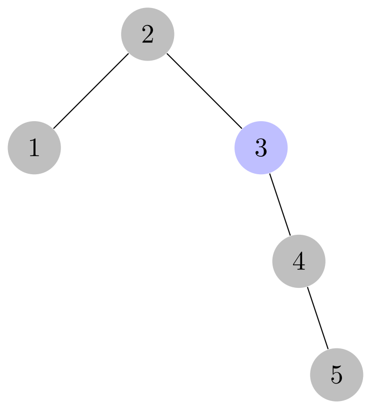
Notice that the BST property is still satisfied:

- x is still a left subtree of b.
- y is still in the right subtree of b.
- z is still in the right subtree of b.
- $b < a$, so a can be a right child of b.

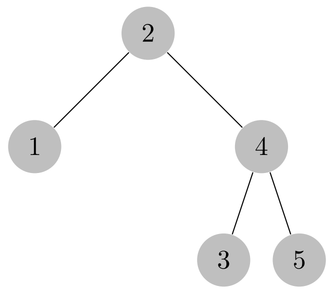
This is a **left rotation** that is quite simple to represent in programming. We simply have to write:

```
        a.left = b.right
        b.right = a
```

Similarly, we can also perform **right rotations**. Suppose, we have the graph:



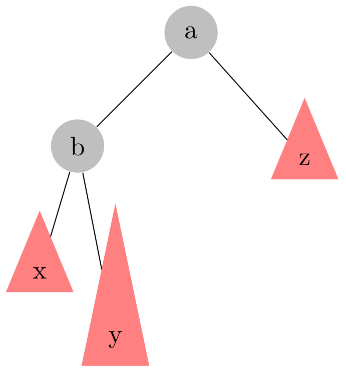
We can perform a right rotation on 3 to turn the tree into:

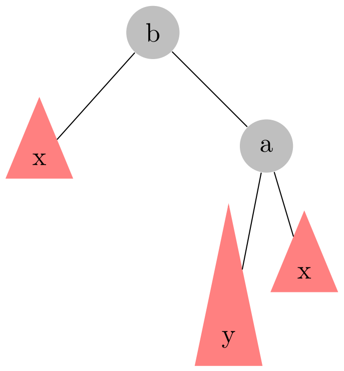


To be more precise, the above example is an **outside imbalance**, as the imbalance is caused in a subtree that is outside. This can be fixed using a single rotation.

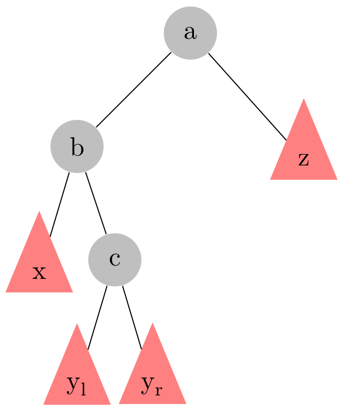
4.6 Balancing Inner Imbalances

If the imbalance is not at the outside, we need to use more than one rotation. For example, if we tried to perform a single rotation on the graph to the left, we will end up with the graph to the right

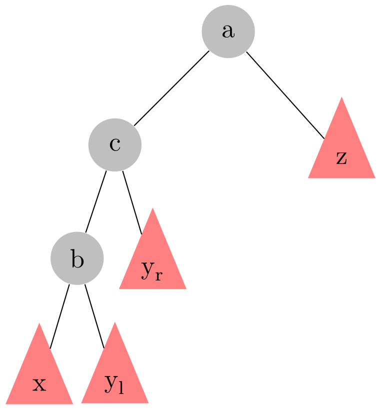


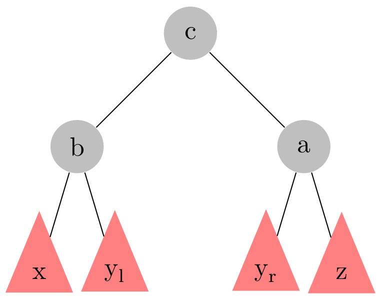


which is still not balanced! Instead, we have to expand our tree a bit and represent it as:



We can balance this by rotating b to the left (below left), then a to the right (below right).





While this may seem complicated, it is not too difficult to implement:

```
        c.left = b
        c.right = a
        b.right = root(yl)
        a.left = root(yr)
```

Implementation of AVL trees using *C* is provided on the course website. Make sure to be familiar with how the code works.

5 Hashing

A **map** is a collection of `(key, value)` pairs. The keys are unique, but the values may not be. This is equivalent to a *Python* dictionary. There are two main operations:

- `get(key)` returns the value corresponding to a given key.
- `put(key, value)` creates a new instance of a key and value pair.

Our motivation is to increase the time efficiency of `get(key)` as much as possible. There are a few methods:

- Store the keys in an unsorted array: The efficiency is $\mathcal{O}(n)$.
- Store the keys in an AVL tree (or something similar). The efficiency is $\mathcal{O}(n)$.
- Store the values in an array, where the index *is* the key. The efficiency is $\mathcal{O}(1)$.

While this last method may sound superior, it may take up a lot of space as keys can be spread apart a lot, and there is a lot of memory allocated that isn't used. The idea of hashing is to implement this same idea, but only using a table size of n.

We can define the hashing function to be:

$$h(x) = x\%n \tag{1}$$

Here, x is the key[1] and % gives the remainder once x is divided by n. It is possible to combine hash functions, say if the key is a compound object:

$$h(s,x) = (h(s)p_1 + h(x)p_2)\%n \tag{2}$$

where p_1 and p_2 are relatively prime (we don't want to factor the expression)

5.1 Collisions

The problem with this hashing function is that it is not a bijection. For example, if $n = 10$, then the key 13 and 23 would represent the same point in memory.

5.1.1 Chaining

We can solve this by using a linked list. For example, if we try to put the key 23 in memory (i.e. index 3) and there is already a linked list defined there (even if it contains only one key), then we just attach the desired key to the end of the linked list.

The runtime depends on the number of elements in a list on average. The **load factor** λ represents the average number of elements stored in each table entry:

$$\lambda = \frac{k}{\text{table size}} \tag{3}$$

where k is the number of entries. If lookup fails (key is not in the hash table), we need to search on average λ nodes. If lookup succeeds, we need to search on average $\lambda/2 + 1$ nodes. We want to keep $\lambda \approx 1$ and we can do this by increasing the table size as needed.

[1]This implies that the key must be an integer. If it was a string, we can convert it into an integer using the ASCII value of the characters, and perform similar things for other types.

5.1.2 Probing

Instead of storing a linked list at each entry, we can store just the key-value pair. If a collision occurs, we put the pair in an empty cell of the hash table instead. In **linear probing**, we empty cell we put it in is the *next empty cell.* During probing, the load factor is always smaller or equal to $\lambda \leq 1$. The table is full if $\lambda = 1$. In general, if there is a collision, we want to put our key-value pair in the index:

$$(h(x) + f(i))\%n \tag{4}$$

where $i = 1, 2, 3, \dots$. The type of probing depends on f:

- Linear Probing: Try the next cell: $f(i) = i$.
  - The biggest problem is *primary clustering.* Full cells tend to cluster with no free cells in between. The time required to find an empty cell can become very large if the table is almost full (i.e. if $\lambda \approx 1$)
- Quadratic Probing: $f(i) = i^2$.
  - Although primary clustering is no longer a prob lem, if the table gets too full (i.e. $\lambda > 0.5$), then it is possible that certain cells become unreachable.

  Theorem: The **Quadratic Problem Theorem**: If n is prime, then the first $n/2$ cells visited by quadratic probing are distinct. Therefore, it's always possible to find an empty cell if the table is at most half full.

  Proof. Suppose there is a repetition. Then:

  $$(h + i^2)\%n = (h + j^2)\%n \tag{5}$$

  Therefore:

  $$h + i^2 = h + j^2 + kn \tag{6}$$

  for some integer n. We can factor this to be:

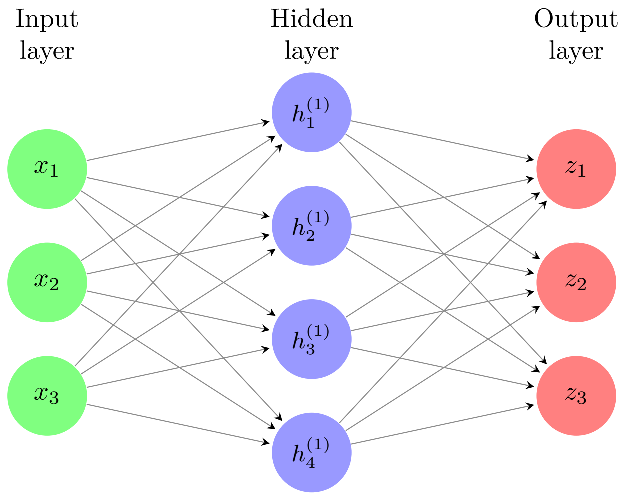
  $$kn = (i + j)(i - j) \tag{7}$$

  Since n is prime, either $i + j$ or $i - j$ must be divisible by T. But since $i < j < n$, both $i - j$ and $i + j$ are too small. □
- Double Hashing: $f(i) = ih_2(x)$
  - Do not worry too much about this.

6 Neural Networks

6.1 Overview

Machine Learning is in a nutshell, is just a graph theory problem. It takes in a series of inputs x_i (i.e. pixels in an image), and through a series of linear transformations, creates the outputs z_i (i.e. probability an image is Justin Bieber)



Intuitively, the hidden layers act as templates. For example, if x_i represent pixels, then $h_i^{(1)}$ can represent combinations of pixels (i.e. lines) and if we have another hidden layer, that can represent combinations of lines (i.e. shapes). Each arrow represents a certain weight, which corresponds to how important a particular node is for the template we're looking for.

Many of this is still traditional programming. The important thing about neural networks is that there can be thousands of nodes, and the weights are determined by the computer, not manually inputted by humans. This means that the computer will "learn" to recognize lines and shapes as setting these particular weights provides the highest rate of success.

In **supervised machine learning**, we train the neural network by providing it with a test set, where z_i is known. It then tries to minimize the error by shifting the weights. This shifting of weights is the mechanism we will be exploring.

The value of each node of interest is given by:

$$\sum(w_{(j,i)}h_i) + b_j \tag{8}$$

where h_i is the value of the node before it, $w_{(j,i)}$ is the weight of the connection, and b_j is the bias of the node of interest.

Warning: To be more rigorous, we can write each weight as:

$$W^{(h,j,i)} \tag{9}$$

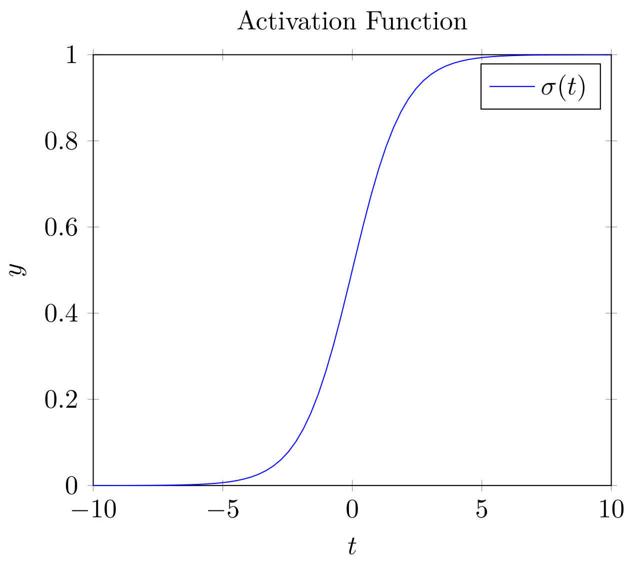
where h refers to the layer, i refers to node in that layer, and j refers to the node in the previous layer given by the connection. However, for our purposes, I chose to leave out this notation so I can focus on the concepts, not the syntax.

6.2 Activation Function

Each node can theoretically take a value from $-\infty$ to ∞, and we don't want one particularly high node to dominate. As a result, we normalize all values to between 0 and 1, using the activation function:

$$\sigma(t) = \frac{1}{1+\exp(-t)} \tag{10}$$

which graphically, looks like this:



6.3 Cost Function

A common cost function (we encountered this in ESC103) is the least square error. If o is the prediction, and y is the actual result, then we can let the cost be:

$$C(o,y)=\sum_{i=1}^{m}|y_i-o_i|^2 \tag{11}$$

Note that for larger datasets, we may have to pass in data as batches, and the cost would be the average cost of each batch.

6.4 Gradient Descent

Let W be a vector representing the weights. Then we want to set the new weight W_{new} to be:

$$W_{\text{new}}=W-\alpha\nabla_W C(o,y)=W-\alpha\begin{bmatrix}f_{w_1}C\\ f_{w_2}C\\ \vdots\end{bmatrix} \tag{12}$$

and the biases to be:

$$B_{\text{new}}=B-\alpha\nabla_B C(o,y) \tag{13}$$

The partial derivative of C with respect to $w_{(j,i)}$ is given by:

$$\frac{\partial C}{\partial w_{(j,i)}}=\frac{\partial o}{\partial w_{(j,i)}}\frac{\partial C}{\partial o} \tag{14}$$

Letting $o_i=\sigma\left(\sum w_{(j,i)}h_j+b_i\right)$, we can use the chain rule to get:

$$\begin{aligned}\frac{\partial C}{\partial w_{(j,i)}}&=\frac{\partial(\sum w_{(j,i)}h_j)}{\partial w_{(j,i)}}\frac{\partial\sigma}{\partial(\sum w_{(j,i)}h_j)}\frac{\partial C}{\partial o} &(15)\\ &=h_j\frac{\partial\sigma}{\partial(\sum w_{(j,i)}h_j)}\frac{\partial C}{\partial o} &(16)\\ &=h_j\sigma'\left(\sum_j w_{(j,i)}h_j+b_i\right)\frac{\partial}{\partial o_i}C &(17)\end{aligned}$$

Note that:

$$\sigma'(t)=\sigma(t)(1-\sigma(t)) \tag{18}$$

and:

$$\frac{\partial C}{\partial o_i}\sum_{i=1}^{N}(o_i-y_i)^2=2(o_i-y_i) \tag{19}$$

we can simplify this to:

$$\boxed{\frac{\partial C}{\partial w_{(j,i)}} = 2h_j o_i(1-o_i)(o_i-y_i)} \tag{20}$$

To re-iterate, $w_{(j,i)}$ refers to a certain weight between two nodes, h_j refers to the previous node, o_i refers to the node of interest, and y_i refers to the desired value. Since we can only have desired values for the output nodes, this equation *only* describes how the cost function changes with respect to how weights connected to the output nodes changes.

However, we can generalize this to other nodes using the chain rule:

$$\frac{\partial C}{\partial w_{(1,j,i)}} = \frac{\partial C}{\partial h_i}\frac{\partial h_i}{\partial w_{(1,j,i)}} \tag{21}$$

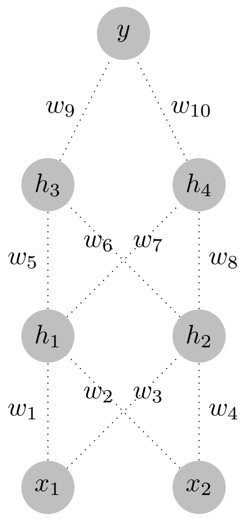
Here:

$$\frac{\partial C}{\partial h_i} = \sum_k \frac{\partial C}{\partial o_k}\frac{\partial o_k}{\partial h_i} \tag{22}$$

Idea: This really isn't too difficult, it's just the notation is hard to understand. Write out equations for the network, and apply ESC195 knowledge. To do so, we make use of an example:

6.5 A Quick Example:

Suppose we have the graph:



To calculate $\frac{\partial C}{\partial w_9}$, we have:

$$\frac{\partial C}{\partial w_9} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial w_{10}} \tag{23}$$

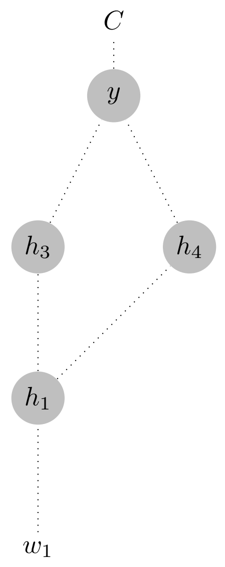
From earlier, we have:

$$\frac{\partial y}{\partial w_{10}} = h_4 y(1-y) \tag{24}$$

To calculate $\frac{\partial y}{\partial w_1}$, we have:

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial w_1} + \frac{\partial C}{\partial y}\frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial w_1} \tag{25}$$

This chain of partial derivatives can be made more clear by drawing a "chain rule tree," similar to what is done in Stewart. Each path corresponds to a different term in the sum. For the above example, we would draw:



Again, note that:

$$\frac{\partial h_1}{\partial w_1} = x_1 h_1(1-h_1) \tag{26}$$

and:

$$\frac{\partial h_3}{\partial h_1} = w_5 h_3(1-h_3) \tag{27}$$

and we can do similar computations for the other partial derivatives.