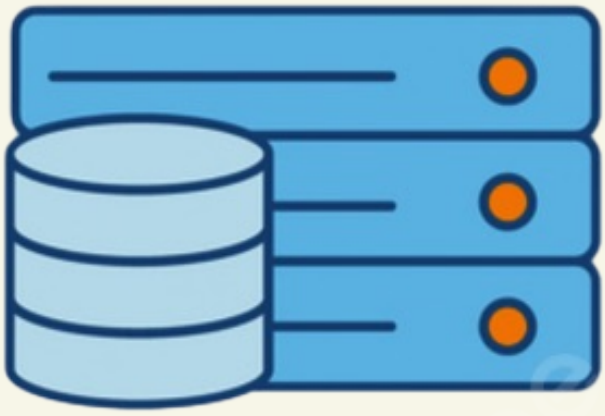
Data Struct & Algorithm



only for this mod!!
* lg → log base 2

Data Structures

- → ways to organise data items to promote efficient data processing
- → e.g. Array, Tree, Linked List

Algorithms

- → clearly specified set of instructions to be followed to solve a problem → receives input & produces output
- → operates on data items that are stored in data structure, carefully chosen data structure makes an algorithm more efficient
- → e.g. Sorting, searching, graph algorithms (shortest paths & minimum spanning trees), AI Algorithms

Sorting

→ Given a sequence of numbers (e.g. 11, 7, 14, 1, 5) → rearrange in increasing order (e.g. 1, 5, 7, 11, 14)

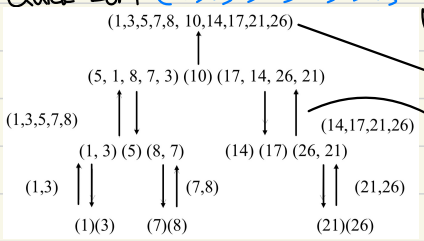
1. Insertion Sort (11, 7, 14, 1, 5, 9, 10)

Sorted sequence	Unsorted sequence
11	7, 14, 1, 5, 9, 10
7, 11	14, 1, 5, 9, 10
7, 11, 14	1, 5, 9, 10
1, 7, 11, 14	5, 9, 10
1, 5, 7, 11, 14	9, 10
1, 5, 7, 9, 11, 14	10
1, 5, 7, 9, 10, 11, 14	

→ 1. examine number from left to right one by one & insert no. into an appropriate place in an already sorted sequence

→ 2. compare 14 to 7 & 11 one by one from left to right & slot it in

2. Quick Sort (10, 5, 1, 17, 14, 8, 7, 26, 21, 3)



→ 1. Use first element 10 to divide numbers into 3 subsets → (smaller than 10) & (10) & (greater than 10)

→ 2. Then sort the left & right subsets in similar way

→ 3. Combine all & we have (1, 3, 5, 7, 8, 10, 14, 17, 21, 26)

Performance Comparision (Insertion & Quick sort)

n (no. of values)	Insertion sort	Quick sort
10	.000001	.000002
100	.000106	.000025
1,000	.011240	.000365
10,000	1.047	.004612
100,000	110.492	.058481
1,000,000	NA	.6842

→ Time (s) taken to sort n (When n=10, don't matter that much)

→ When n is large, insertion sort will take a longer time to execute than quick sort (different efficiency)

Mathematical Induction

→ Mathematical induction can be used to prove a statement is true for all $n \geq n_0$ if

1. Basis of induction → statement is true for $n = n_0$
2. Induction step → Assuming statement true for $n = k$, where $k \geq n_0$, prove statement true for $n = k+1$

E.g. Use mathematical induction to prove $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$ for all $n \geq 1$

Basic step: show eqn true for $n = 1 \Rightarrow$ LHS = 1 RHS = $\frac{(1)(2)(3)}{6} = 1$

Induction step: Assume eqn true for $n = k$, prove it is true for $k+1$

$$\sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2 = \frac{k(k+1)(2k+1)}{6} + (k+1)^2 = \frac{k(k+1)(2k+1) + 6(k+1)^2}{6}$$

$$= \frac{(k+1)(k+2)(2(k+1)+1)}{6}$$

similar

, eqn correct for $n \geq 1$

★ AP: $U_n = a + (n-1)d$, $S_n = \frac{n}{2}(a + a + (n-1)d)$

GP = $V_n = ar^{n-1}$, $S_n = \frac{a(1-r^n)}{1-r}$

E.g. Use mathematical induction to prove $(1+x)^n \geq 1+nx$, where $x \geq -1$

When $n=1$, $(1+x)^1 = 1+x$ → eqn is true.

Assume eqn is true for n, we want to prove that eqn is also valid for n+1. That is $(1+x)^{n+1} \geq 1+(n+1)x$, $x \geq -1$.

Since $(1+x)^n \geq 1+nx$

$LHS = (1+x)^{n+1} = (1+x)^n(1+x) \geq (1+nx) \times (1+x)$

$= 1+nx+x+nx^2 \geq 1+(n+1)x = RHS$ (make it like this)

Note that $nx^2 \geq 0$ when $x \geq -1$

Time Efficiency (Running Time)

- → analysed by determining number of repetitions of basic operations as a function of input size.
- → Basic operations → operation that contributes most towards the running time of the algorithm (largely independent from programming language)
- → take worst case scenarios → maximum no. of basic operations over inputs of size n

running time → $T(n) \approx C_{op} C(n)$ ← execution time for a basic operation; ← no. of times basic operation is executed; ← input

e.g. Counting Basic Operations

```
size = len(a)                # number of operation -> 1
largest = a[0]               # number of operation -> 1

count = 1                    # number of operation -> 1
while count < size:          # number of operation -> n - 1
    if a[count] > largest:   # number of operation -> n - 1
        largest = a[count]   # number of operation -> n - 1
    count = count + 1        # number of operation -> n - 1
print(largest)               # number of operation -> 1
```

→ executes n-1 times in general

$a \times (n-1) \leq T(n) \leq b \times (n-1)$

best time ← running time → worst time

→ Changing hardware/software environment affects T(n) by constant factor, but does not Δ growth rate T(n)

↳ How does running time ↑ as input size double?

e.g. Let $C(n) = \frac{n(n-1)}{2}$ be number of times basic operation executed for ($\approx \frac{n^2}{2}$)

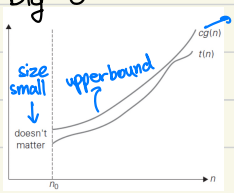
$\frac{T(2n)}{T(n)} = \frac{C_{op} C(2n)}{C_{op} C(n)} = \frac{(2n)^2}{2} \div \frac{n^2}{2} = 4$

∴ Input size double → running time 4 times longer for this algorithm

Order of growth

↳ for comparing the efficiency of different algorithms → notations used: O (big oh), Ω (big omega), Θ (big theta)

1 Big-O

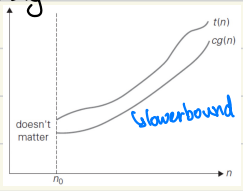


(the constant)

- If t(n) is bounded above by some constant multiple of g(n) for all large $n \geq n_0$

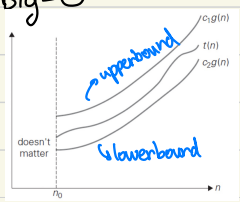
↳ $t(n) \leq cg(n)$ for all $n \geq n_0$ → $t(n) = O(g(n))$ ← upperbound of t(n)

2. Big-Ω



- If $t(n) \geq cg(n)$ for all $n \geq n_0$ → $t(n) = \Omega(g(n))$ ← lowerbound of t(n)

3. Big-Θ



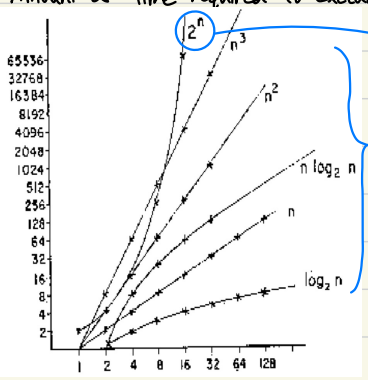
- If t(n) bounded above & below by some constant multiples of g(n) for $n \geq n_0$.

↳ $c_2 g(n) \leq t(n) \leq c_1 g(n)$ for $n \geq n_0$ → $t(n) = \Theta(g(n))$ & g(n) is tight bound

($c_2 g(n)$ ← lowerbound; $c_1 g(n)$ ← upperbound)

Several Important functions

→ Amount of time required to execute an algorithm usually depend on input size, n



exponential → no way computer can do it

The rest can run normally

Order of growth (lowest to highest)

↳ $5 \lg(n+100)^{10}, n \lg n, n^4+3n^3+1, 3^n, 2^{2n}, n!$

E.g. Given $f(n)=2n+3\lg n$ & $g(n)=n$, prove that $2n+3\lg n = \Theta(n)$

$2n+3\lg n \leq 2n+3n = 5n$ for all $n \geq 1$

We can take $k_1=5$ & $n_1=1$ & conclude $f(n)=O(n)$

Since $2n+3\lg n \geq 2n$ for all $n \geq 1$, we can take $k=2$ & $n_2=1 \longrightarrow f(n)=\Omega(n)$

$\therefore 2n+3\lg n = \Theta(n)$

Recursion

→ repetitive process in which an algorithm calls itself → each call solves an identical, but smaller problem

→ a test for base case (initial condition) enables the recursive calls to stop

→ To write a recursive function:

a) Determine input size

b) Determine base case(s) → answer is known

c) Determine general case → problem expressed as a smaller version of itself

E.g. Given that $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n \longrightarrow 0! = 1$ by definition [Base Case]

Let $f(n) = n! = n(n-1)! \longrightarrow f(n) = n \cdot f(n-1)$ [General Case]

```
def fact(n):
    if n==0:
        return 1          → 0! = 1
    else:
        return n*fact(n-1)   Recursive call

n = int(input("Enter n: "))
print("factorial({0}) = {1}".format(n, fact(n)))
```

Sample output

```
Enter n: 4
factorial(4) = 24
```

↳ Number of multiplications needed, let $T(n)$ = no. of multiplications needed to compute fact(n)

$T(0) = 0$ → according to code → n==0, return 1 (no multiplication)

$T(1) = T(0)+1 = 1$ → 1 multiplication

$T(2) = T(1)+1 = 2$

$T(3) = T(2)+1 = 3$

$T(100) \simeq 100$

Method of Backward Substitutions

→ From above, general case: $T(n) = T(n-1)+1$, Base Case: $T(0)=0$

→ Using method of backward substitution,

$T(n) = T(n-1)+1 = (T(n-2)+1)+1 = T(n-2)+2$

$T(n) = T(n-n)+n = T(0)+n = n$

→ Time efficiency of recursive algorithm is $O(n)$

E.g. Reversing an array (5, 3, 2, 6 → 6, 2, 3, 5)

```
def ReverseArray(A, i, j):
    if i < j:
        temp = A[i]
        A[i] = A[j]
        A[j] = temp
        ReverseArray(A, i+1, j-1)

A = [5, 3, 2, 6]
print("Original array: ", A)
first = 0
last = len(A) - 1
ReverseArray(A, first, last)
print("Reverse array: ", A)
```

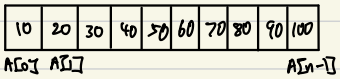
Swapping Operation

- Let T(n) be the number of swapping operations needed to reverse elements of an array
  - size $n \rightarrow T(0) = 0$ (reverse a subarray with n-2 elements & perform one swapping operation)
  - $T(n) = T(n-2) + 1 = [T(n-4) + 1] + 1$
  - $= T(n-n) + \frac{n}{2} = \frac{n}{2}$ → Time efficiency [O(n)]
- Recursive algorithm look simple but time consuming → total no. of calls grows exponentially

Basic Data Structure

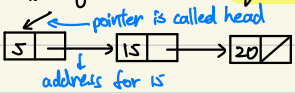
1. Array

- group of data elements stored in contiguous memory
- Advantage: Instant access to any element in array
- Disadvantage: Hard to insert new elements in middle of array, deletion may be time-consuming (have to move everything up & down)



2. Linked List

- Each node of linked list is composed of two parts: data part (store data value) & next part (hold address (pointer) of next node in list)
- Advantages: size of linked list can be changed dynamically, new nodes can be inserted
- Disadvantage: do not have quick access to members of linked list → go node to get address of next node



Abstract Data Types

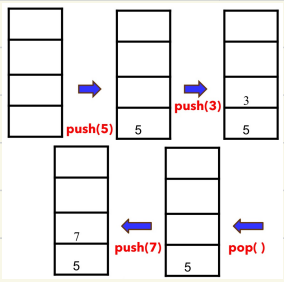
→ functions that operate on data, only behavior of ADT is specified, implementations of ADT functions are not specified

1. Stacks

- item can only be inserted & removed from the top → last-in-first-out (LIFO)
- e.g. Page-visited history in web browser, undo sequence in text-editor
- Implementing stacks using array: (add elements left to right)
  - t keeps track of index of top element.
  - item pushed into stack, t → t+1 & put item at t+1
  - item popped off stack, t → t-1

Operation	Output	Bottom – Stack – Top
S.stack_init( )	-	( )
S.push(5)	-	(5)
S.push(3)	-	(5, 3)
S.pop( )	-	(5)
S.push(7)	-	(5,7)
S.pop()	-	(5)
S.top( )	5	(5)
S.pop( )	-	( )
S.pop( )	"error"	( )
S.empty( )	*true*	( )
S.push(9)	-	(9)

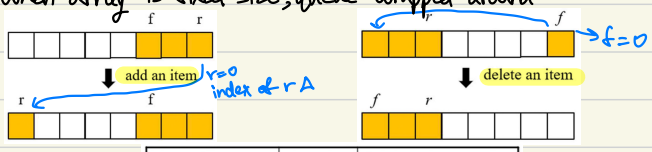
(make stack empty; add value; remove value; return value at top; return true if stack is empty, else false)

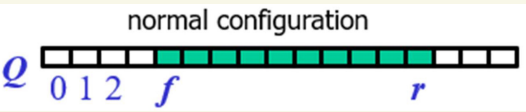


2. Queues

- insertions are at the rear of queue & removals are at the front of queue → first-in first-out (FIFO)
- e.g. waiting lines at banks, access to shared resources (printer)
- Implementing queue using array:
  - Need two variables, rear (r) & front (f) to track indexes of queue. Empty queue → r & f = -1

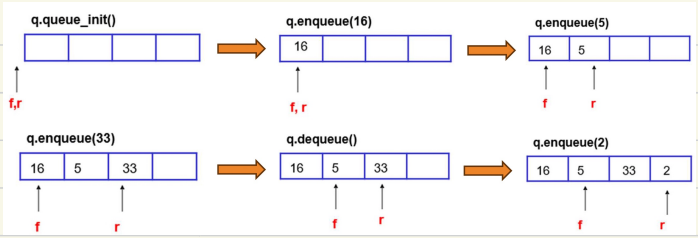
- item added into empty queue → r & f = 0 & put item in cell. Item added to non-empty queue, increase r & put item in cell r.
- item removed from non-empty queue → ↑f
- f=r → only one item in queue → deletion → f & r = -1
- When array is fixed size, queue wrapped around:





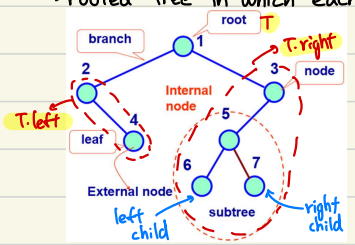
make queue empty / add value / remove item / return value in front / return true.

Operation	Output	front ← Q ← rear
Q.queue_init( )	-	( )
Q.enqueue(5)	-	(5)
Q.enqueue(3)	-	(5, 3)
Q.dequeue( )	-	(3)
Q.enqueue(7)	-	(3, 7)
Q.dequeue( )	-	(7)
Q.front( )	7	(7)
Q.dequeue( )	-	( )
Q.dequeue( )	"error"	( )
Q.empty( )	*true*	( )
Q.enqueue(9)	-	(9)

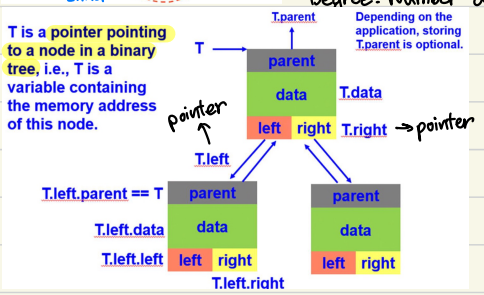


Binary Trees

↳ rooted tree in which each node has no children, one child or two children



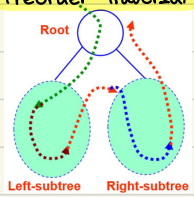
- Internal Node: node with at least one child
- External Node: node without any children
- Depth of Node: number of ancestors → e.g. F = 2
- Height of tree: maximum depth of any node (count lines) → e.g. 3 → empty tree height → -1, single node height → 0
- Degree: Number of children at a node → e.g. F = 3



Binary Tree Traversals

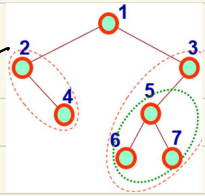
↳ To traverse a binary tree → visit each node in the tree in some prescribed order

1. Preorder Traversal



→ root – left – right

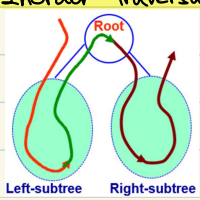
1, 2, 4, 3, 5, 6, 7




```
def Preorder(self, root):
    res = []  # At first, result is empty
    if root:  # if root is not empty
        res.append(root.data)    # read root.data
        res = res + self.Preorder(root.left)
        res = res + self.Preorder(root.right)
    return res
```

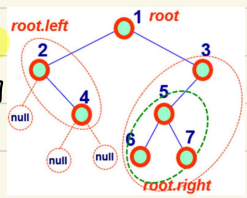
root.left replace root

2. Inorder Traversal



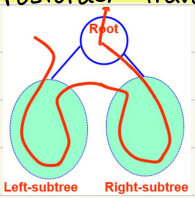
→ left – root – right

res = [2, 4, 1, 6, 5, 7, 3]



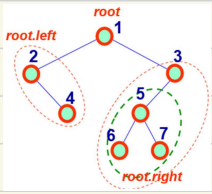

```
def Inorder(self, root):
    res = []  # At first, result is empty
    if root:  # if root is not empty
        res = self Inorder(root.left)
        res.append(root.data)  # read root.data
        res = res + self.Ineorder(root.right)
    return res
```

3. Postorder Traversal



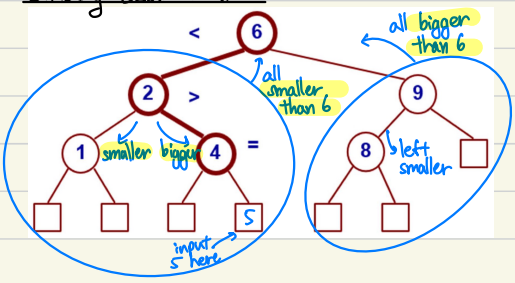
→ left-right-root

res = [4, 2, 6, 7, 5, 3, 1]




```
def Postorder(self, root):
    res = []  # At first, result is empty
    if root:  # if root is not empty
        res = self Postorder(root.left)
        res = res + self.Ineorder(root.right)
        res.append(root.data)  # read root.data
    return res
```

Binary Search Tree



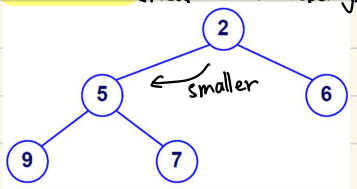
Binary Heap

→ All levels, except the last level have as many nodes as possible (Structural property)

↳ On last level, there is at most one node with one child, which must be a left child → called left-complete binary tree

1. Binary Minheap

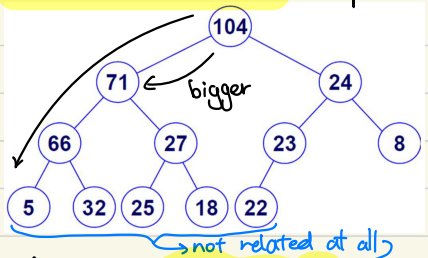
↳ heap structural in which values are assigned to the nodes so that value of each node is less than/equal to value of its children (Heap Order Property)

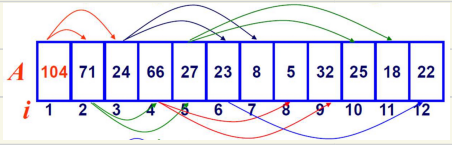


★ minheap must maintain two properties:
structural & order property

2. Binary Maxheap

↳ heap structure in which values are assigned to the nodes → value of each node is greater than/equal to values of its children (Heap Order Property)





→ floor($\frac{3}{2}$) = 1 (parent)

parent(i) { return $\lfloor \frac{i}{2} \rfloor$ }

left(i) { return 2*i; }

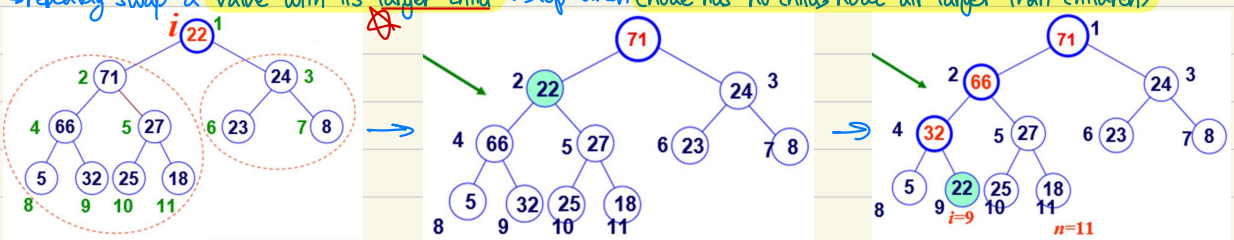
right(i) { return 2*i + 1 }

- MaxHeap is weakly sorted → values along path from root to a terminal node are in non-increasing order

Siftdown (Maintaining a Maxheap)

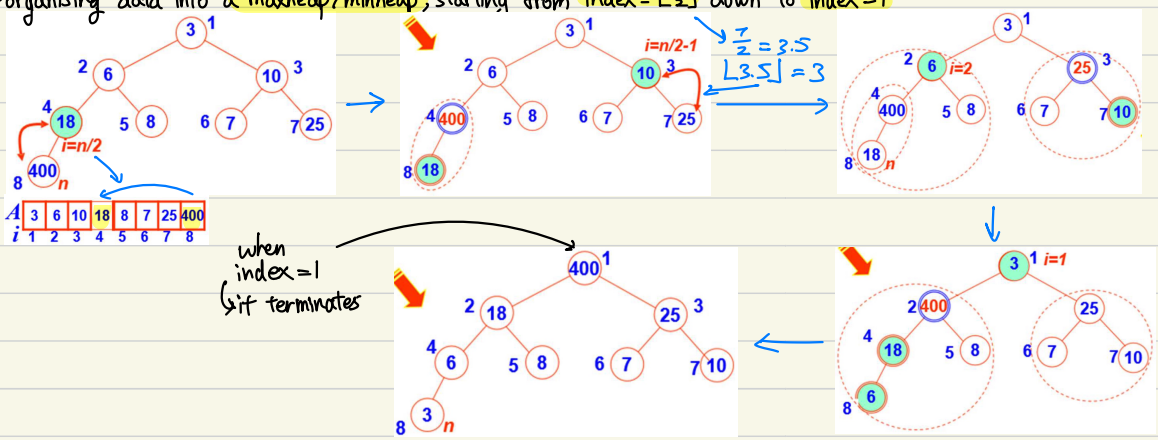
↳ value at root may be greater/smaller than value of its children in minheap/maxheap

↳ repeatedly swap a value with its larger child → stop when (node has no child, node alr larger than children)



Heapify

↳ organising data into a maxheap/minheap, starting from index = $\lfloor \frac{n}{2} \rfloor$ down to index = 1



BST Insert

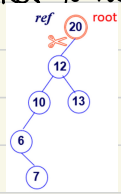
↳ insert a value into binary search tree (in jupyter notebook)

```
#insert a value into binary search tree!
def BSTinsert_recurs(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.value:
        root.left = BSTinsert_recurs(root.left, key)
    else:
        root.right = BSTinsert_recurs(root.right, key)
    return root
```

BST Node Deletion

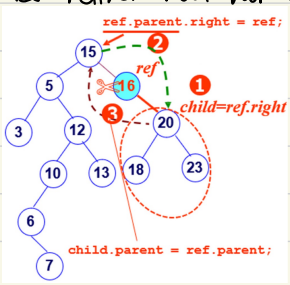
1. BST.replace(root, ref) → for ref has only one or no child

a) Ref is root → set child of ref as root, stop.



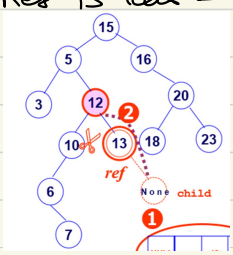

```
if ref.left == None:                    1
    child = ref.right
else:
    child = ref.left
if ref == root:
    if child != None:                   2
        child.parent = None             3
    return child    # root
```

b) Ref neither root nor leaf → set child of ref as ref.parent's child, stop.




```
if ref.left == None:
    child = ref.right                   1
else:
    child = ref.left
# is ref left child?
if ref.parent.left == ref:
    ref.parent.left = child
else:                                   2
    ref.parent.right = child

if child != None:                       3
    child.parent = ref.parent
return root
```

c) Ref is leaf → set ref.parent's child None, stop.

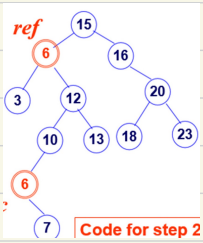



```
if ref.left == None:
    child = ref.right                   1
else:
    child = ref.left
# is ref left child?
if ref.parent.left == ref:
    ref.parent.left = child             2
else:
    ref.parent.right = child

if child != None:
    child.parent = ref.parent
return root
```

2. Two children

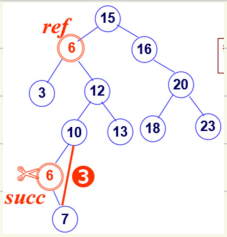

```
succ = ref.right
while succ.left != None:
    succ = succ.left
```

↳ search min until have no more left children




```
ref.data = succ.data
```

↳ replace value of node to be deleted with min value




```
delete succ
return BSTreplace (root, succ)
```

↳ delete succ

Sorting

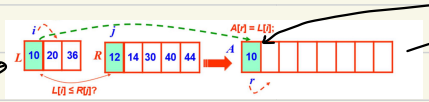
- Mergesort → divide array into 2 smaller arrays → sort them & combine them [Θ(n log n)]
- Heapsort → siftdown values [Θ(n log n)] → Average case complexity ← [Θ(n log n)] → best pratical choice in log n → smallest inp
- Quicksort → mentioned in front

1. Merge Algorithm

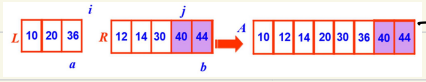
→ before this, both array are sorted in increasing order

```
def merge(A, p, q, n):
    a = q - p + 1    # number of elements in L subarray
    b = n - q        # number of elements in R subarray
    # Copy subarray A[p..q] to L and A[q+1..n] to R
    L = []
    for k in range(0, a):
        L = L + [A[p + k]]
    R = []
    for k in range(0, b):
        R = R + [A[q + 1 + k]]

i = 0
j = 0
r = p
while i < a and j < b:
    # copy smaller value to A
    if L[i] <= R[j]:
        A[r] = L[i]
        i = i + 1
    else:
        A[r] = R[j]
        j = j + 1
    r = r + 1

# copy remainder, if any, of first sub-array to A
while i < a:
    A[r] = L[i]
    i = i + 1
    r = r + 1
# copy remainder, if any, of second sub-array to A
while j < b:
    A[r] = R[j]
    j = j + 1
    r = r + 1
```

→ array
↳ must have 2 arrays

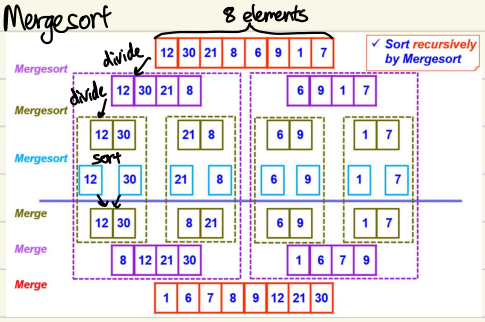


→ compare 10 & 12, 12>10



→ if all data from one array, can just move the remainder over

- Mergesort




```
def mergesort(A,i,j)
    # if only one element, just return
    if (i == j)
        return
    # divide A into two nearly equal parts
    m = int((i + j)/2)  # interger part
    # mergesort each half - recursive calls
    mergesort(A, i, m)
    mergesort(A, m + 1, j)
    # merge the two sorted halves
    merge(A, i, m, j)
```

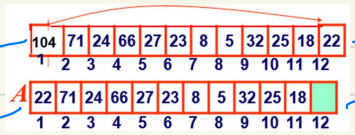
→ sort first group
→ sort second group

2. Siftdown: Recursive implementation of maxheap

```
def siftdown(A, i):
    n = len(A) - 1  # heap array 1..n
    largest = i
    left = 2 * i    # test for a left child
    if left <= n and A[left] > A[i]:
        largest = left
    right = 2 * i + 1  # test for a right child
    # if right child has a larger value, update largest
    if right <= n and A[right] > A[largest]:
        largest = right

    if largest != i:
        swap(A, i, largest)  # exchange using temp
        siftdown(A,largest)  # recursive call
```

- Delete root from heap

```
def heap_delete_root(A):
    n = len(A) - 1  # heap array 1..n
    A[1] = A[n]
    A.pop(n)    # delete last element
    n = n - 1  # adjust the heap size
    siftdown(A, 1)
```

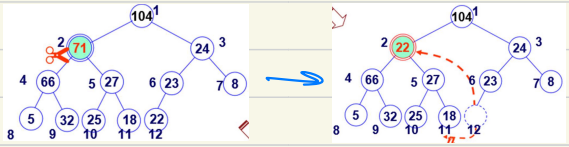


→ move 104 to the back, delete it

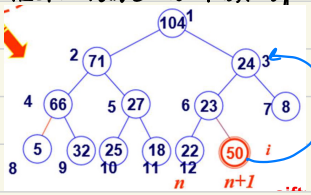
→ siftdown

- Delete any node from maxheap

```
if A[int(i/2)] >=A[i]:
    siftdown(A,i)
else:
    siftup(A,i)
```



- Insert value to maxheap



→ sift up until smaller than parent

code

→ instead of siftdown → can heapify (earlier)

Sequential (Linear) Search

→ search through array one by one

→ worst-time complexity → O(n) → need to search all elements in array

```
def sequential_search(L, key):
    for k in range(len(L)):
        if key == L[k]:
            return k
    return -1

L = [-1, 3, 7, 0, 17]

key = 17
result = sequential_search(L, key)
print(result)

key = 2
result = sequential_search(L, key)
print(result)
```

Binary Search

→ search for an item in a sorted array (increasing order)

→ binary-search algorithm begins by computing midpoint $k=L\frac{n-1}{2}$ & if L[k]=key → problem solved, else array divided into two parts of nearly equal size & searched, if key < L[k] → search in array 1

```
def binary_search(L, key):
    i = 0
    j = len(L) - 1
    while i <= j:
        k = (i + j) // 2
        if key == L[k]:
            return k
        elif key < L[k]:
            j = k - 1
        else:
            i = k + 1
    return -1

L = [3,14,15,17,28,31,40,51]
result = key = 51
binary_search(L, key)
print(result)
```

round down to nearest whole no.

increase index & loop again

3 14 15 17 28 31 40 51 (loop 1)

3, 6

28 31 40 51 (loop 2)

40 51 (loop 3)

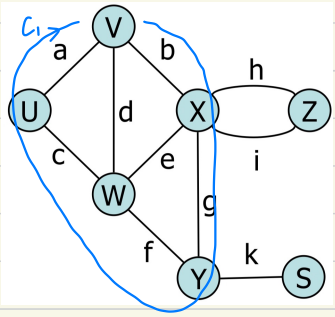
51 (loop 4)

↳ have to search & divide the array k times → time complexity → O(Log(n)) → better than linear growth

Graph

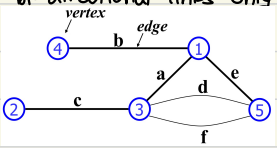
↳ graph G is a pair (V, E) → G=(V, E):
- a) V is a set of nodes called vertices
- b) E is set of connections between pair of vertices called edges → if e is an edge connecting vertices u & v → e=(u, v) / e=(v, u)

↳ terminologies:
- a) Edges are incident is shave a vertex (a, d, b incident on V)
- b) Adjacent vertices (U & V are adjacent)
- c) Degree of vertex (X has degree 5)
- d) Leaf → vertex with degree 1 (S)
- e) Path → sequence of alternating vertices & edges, begin & end with vertex
- f) Cycle → path whose initial vertex & terminal vertex are identical, no repeated edges



1. Undirected Graph

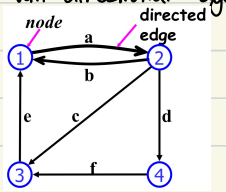
↳ bi-directional links only



V={1, 2, 3, 4, 5}
E={a, b, c, d, e, f}
a=(1, 3), b=(1, 4), c=(2, 3)

2. Directed Graph (Digraph)

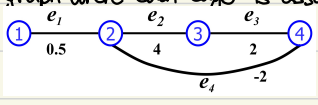
↳ uni-directional edge



V={1, 2, 3, 4}
E={a, b, c, d, e, f}
a=(1, 2), b=(2, 1), c=(2, 3) → but not c=(3, 2)

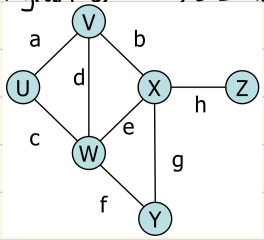
3. Weight Graphs

↳ graph where each edge is associated with a number (value) → actual meaning of no. depend on application

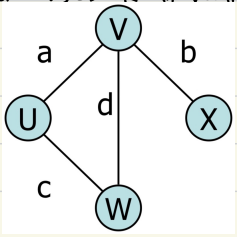


4. Subgraphs

↳ A graph G'=(V', E') is subgraph of G=(V, E) if all vertices & edges of G' are in G



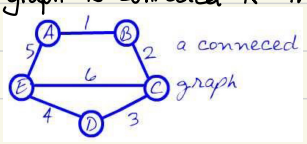
Graph G

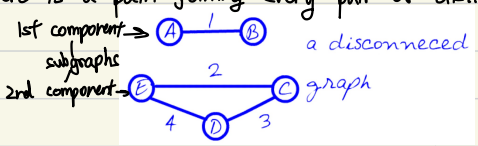


subgraph of G (G')

5. Connectivity

↳ graph is connected if there is a path joining every pair of distinct vertices, otherwise disconnected

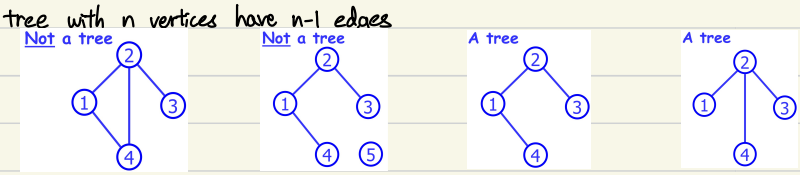




6. Trees

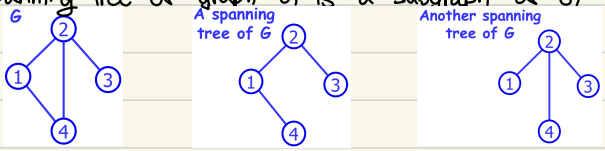
↳ graph called tree if it is connected & it contains no cycles

↳ tree with n vertices have n-1 edges



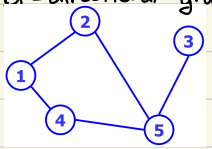
7. Spanning Tree

↳ spanning tree of graph G is a subgraph of G that is a tree & that includes all vertices of G



Adjacency Matrix

1. Bi-directional graph



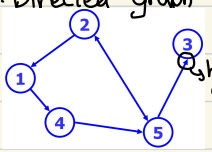
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

→ if 1 connect to 2 directly → put 1, else put 0

→ symmetric

→ diagonal entries are zero → if they do not have self-cycles

2. Directed graph



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

→ if weighted graph, there will be value instead of 1

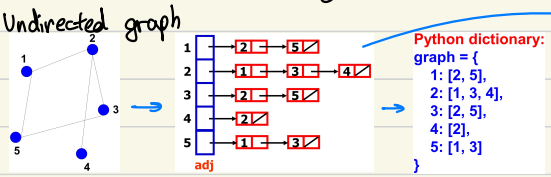
5 → 3

→ space complexity → $O(n^2)$

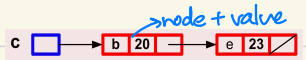
Adjacency List

↳ another way of representing an undirected graph is to use an array of linked list

1. Undirected graph



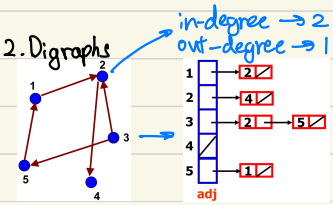
→ if weighted graph →



in-degree of v = no. of incoming edges

out-degree of v = no. of outgoing edges

2. Digraphs



→ adj[i] is a pointer

adj[i].data is 2

adj[i].next is pointer to next node in linked list

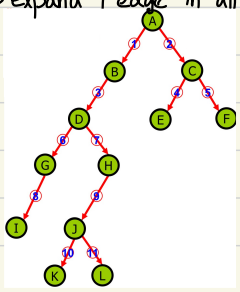
→ no. of vertices = n, number of edges in graph = m → space complexity → O(n+m)

→ if graph is sparse → each vertex is connected to very few vertices → adj list based algorithms more efficient than adj matrix based algorithm

→ if graph is dense → each vertex is connected to almost all vertices → adj matrix based algorithm is about equally efficient as adj list based algorithm.

Breadth-First Search (BFS) → FIFO queue

↳ expand 1 edge in all direction for each round of expansion. until all vertices are explored



queue → add A to q; add child

t	v	q	Visited
		A	
0	A	B,C	A
1	B	C,D	A,B
2	C	D,E,F, H	A,B,C
3	D	E,F,G, H	A,B,C,D
4	E	F,G,H	A,B,C,D,E
5	F	G,H	A,B,C,D,E,F
6	G	H,I	A,B,C,D,E,F,G
7	H	I,J	A,B,C,D,E,F,G,H
8	I	J	A,B,C,D,E,F,G,H,I
9	J	K,L	A,B,C,D,E,F,G,H,I,J
10	K	L	A,B,C,D,E,F,G,H,I,J, K
11	L		A,B,C,D,E,F,G,H,I,J, K,L

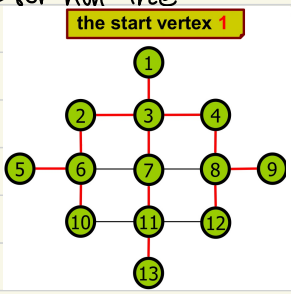
```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def bfs_search(root, target):
    if root is None:
        return False

    queue = []
    queue.append(root)
    while queue:
        node = queue.pop(0)
        print(node.val)
        if node.val == target:
            return True
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return False
```

```
# Create a binary tree as an example
#     1
#    / \
#   2  3
#  / \  \
# 4  5  6
#/ \
#7  8
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)
root.left.left.left = TreeNode(7)
root.left.left.right = TreeNode(8)

target = 5
result = bfs_search(root, target)

if result:
    print(f"Found {target} in the binary tree.")
else:
    print(f"{target} not found in the binary tree.")
```

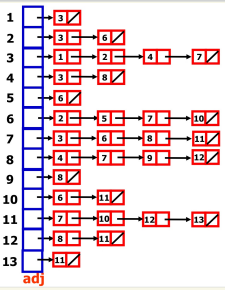
Program Results:

```
1
2
3
4
5
Found 5 in the binary tree.
```

↳ for non-tree




```
graph = {
    1: [3],
    2: [3, 6],
    3: [1, 2, 4, 7],
    4: [3, 8],
    5: [6],
    6: [2, 5, 7, 10],
    7: [3, 6, 8, 11],
    8: [4, 7, 9, 12],
    9: [8],
    10: [6, 11],
    11: [7, 10, 12, 13],
    12: [8, 11],
    13: [11]
}
```




```
def bfs(graph, start):
    visited = [False] * (len(graph) + 1)  # Initialize visited array
    queue = []  # Initialize a list as a queue for BFS
    traversal_order = []  # To store the order of visited vertices

    # Start BFS from the given vertex
    queue.append(start)
    visited[start] = True

    while queue:
        vertex = queue.pop(0)  # Dequeue the front element
        traversal_order.append(vertex)

        for neighbor in graph[vertex]:
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True

    return traversal_order
```

```
# Starting vertex for BFS
start_vertex = 1

# Perform BFS and get the traversal order
order = bfs(graph, start_vertex)

# Print the order of visited vertices
print("BFS Traversal Order:")
for vertex in order:
    print(vertex, end=" ")
```

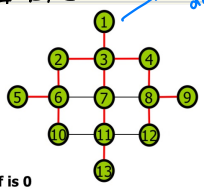
BFS Traversal Order:

1, 3, 2, 4, 7, 6, 8, 11, 5, 10, 9, 12, 13

→ time complexity → O(n+m) → visit all vertices O(n); visit all edges to explore adjacent vertices, O(m)

→ Finding shortest path length using BFS

```
from collections import deque

def bfs_shortest_path_lengths(graph, start):
    distances = {}  # To store shortest path lengths
    visited = [False] * (len(graph) + 1)  # Initialize
    queue = deque()  # Use deque as a queue for BFS

    # Start BFS from the given vertex
    queue.append(start)
    visited[start] = True
    distances[start] = 0  # Distance from start to itself is 0

    while queue:
        vertex = queue.popleft()  # Dequeue the front element

        for neighbor in graph[vertex]:
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True
                distances[neighbor] = distances[vertex] + 1  # Increment the distance

    return distances
```



increase in all direction 1 step at a time

	Length[i]
1	0
2	2
3	1
4	2
5	4
6	3
7	2
8	3
9	4
10	4
11	3
12	4
13	4

no. of lines

```
# Starting vertex for BFS
start_vertex = 1

# Compute shortest path lengths and print them
distances = bfs_shortest_path_lengths(graph, start_vertex)

print("Shortest Path Lengths from Vertex 1:")
for vertex, distance in distances.items():
    print(f"Vertex {vertex}: {distance}")
```

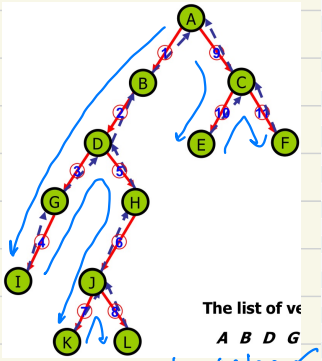
Program Results:

```
Shortest Path Lengths from Vertex 1:
Vertex 1: 0
Vertex 3: 1
Vertex 2: 2
Vertex 4: 2
Vertex 7: 2
Vertex 6: 3
Vertex 8: 3
Vertex 11: 3
Vertex 5: 4
Vertex 10: 4
Vertex 9: 4
Vertex 12: 4
Vertex 13: 4
```

```
graph = {
    1: [3],
    2: [3, 6],
    3: [1, 2, 4, 7],
    4: [3, 8],
    5: [6],
    6: [2, 5, 7, 10],
    7: [3, 6, 8, 11],
    8: [4, 7, 9, 12],
    9: [8],
    10: [6, 11],
    11: [7, 10, 12, 13],
    12: [8, 11],
    13: [11]
}
```

Depth-First Search (DFS) → LIFO stack

↳ explore branch as far as possible before backtracking → continue the process until we have visited all vertices reachable from start vertex



can also stop here, but algo don't know

visited? push? → stack

t	v	u	(top) Stack s	Visited
			A	A
0	A	B	B,A	A,B
1	B	D	D,B,A	A,B,D
2	D	G	G,D,B,A	A,B,D,G
3	G	I	I,G,D,B,A	A,B,D,G,I
4	I		G,D,B,A	A,B,D,G,I
5	G		D,B,A	A,B,D,G,I
6	D	H	H,D,B,A	A,B,D,G,I,H
7	H	J	J,H,D,B,A	A,B,D,G,I,H,J
8	J	K	K,J,H,D,B,A	A,B,D,G,I,H,J,K
9	K		J,H,D,B,A	A,B,D,G,I,H,J,K
10	J	L	L,J,H,D,B,A	A,B,D,G,I,H,J,K,L
11	L		J,H,D,B,A	A,B,D,G,I,H,J,K,L
12	J		H,D,B,A	A,B,D,G,I,H,J,K,L
13	H		D,B,A	A,B,D,G,I,H,J,K,L
14	D		B,A	A,B,D,G,I,H,J,K,L
15	B		A	A,B,D,G,I,H,J,K,L
16	A	C	C,A	A,B,D,G,I,H,J,K,L,C
17	C	E	E,C,A	A,B,D,G,I,H,J,K,L,C,E
18	E		C,A	A,B,D,G,I,H,J,K,L,C,E
19	C	F	F,A	A,B,D,G,I,H,J,K,L,C,E,F
20	F		A	A,B,D,G,I,H,J,K,L,C,E,F
21	A			A,B,D,G,I,H,J,K,L,C,E,F

remove

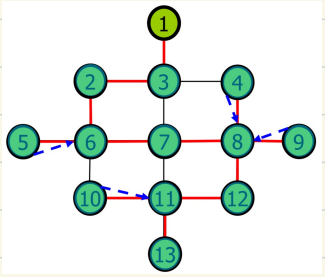
```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def dfs_search(root, target):
    if root is None:
        return False

    stack = []  # Stack for DFS traversal
    traversal_order = []  # List to store the order of node traversal
    stack.append(root)

    while stack:
        node = stack.pop()
        traversal_order.append(node.val)  # Record the order of node traversal

        if node.val == target:
            return traversal_order

        # Push the right child first so that left child is processed first (LIFO)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

    # If target is not found, return the traversal order
    return traversal_order
```

```
# Create a binary tree with 12 nodes as an example
#     1
#    / \
#   2  3
#  / \  \
# 4  5  6
#/ \
#7  8
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)
root.left.left.left = TreeNode(7)
root.left.left.right = TreeNode(8)

# Example DFS search with target = 5
target = 5
traversal_order = dfs_search(root, target)

if traversal_order[-1] == target:
    print(f"Found {target} in the binary tree.")
    print("Order of node traversal:", traversal_order)
else:
    print(f"{target} not found in the binary tree.")
```

Order of node traversal:

[1, 2, 4, 7, 8, 5]

↳ for non-tree




```
def dfs(graph, start):
    visited = [False] * (len(graph) + 1)  # Initialize visited array
    stack = []  # Initialize a list as a stack for DFS
    traversal_order = []  # To store the order of visited vertices

    # Start DFS from the given vertex
    stack.append(start)

    while stack:
        vertex = stack.pop()  # Pop the top element
        if not visited[vertex]:
            traversal_order.append(vertex)
            visited[vertex] = True

        for neighbor in reversed(graph[vertex]):
            if not visited[neighbor]:
                stack.append(neighbor)

    return traversal_order
```

```
graph = {
    1: [3],
    2: [3, 6],
    3: [1, 2, 4, 7],
    4: [3, 8],
    5: [6],
    6: [2, 5, 7, 10],
    7: [3, 6, 8, 11],
    8: [4, 7, 9, 12],
    9: [8],
    10: [6, 11],
    11: [7, 10, 12, 13],
    12: [8, 11],
    13: [11]
}
```

```
# Starting vertex for DFS
start_vertex = 1

# Perform DFS and get the traversal order
order = dfs(graph, start_vertex)

# Print the order of visited vertices
print("DFS Traversal Order:")
for vertex in order:
    print(vertex, end=" ")
```

DFS Traversal Order: (not unique)

1, 3, 2, 6, 5, 7, 8, 4, 9, 12, 11, 10, 13

→ time complexity → O(n+m)

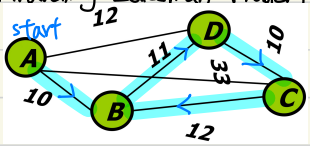
Application of DFS & BFS

- → can be used to test whether a graph is connected → graph connected if & only if all vertices are visited.
- → if any unvisited vertices remain (unconnected graph) → select one of them as new source & repeats search
- → algorithm ends only when every vertex has been visited

Greedy Search Techniques

- → in each step, adds part of solution → solution determined by greedy rule (heuristic)
- → in general, only exhaustive search algorithm (linear, binary search, BFS, DFS) guarantee to find globally optimal solutions → greedy only can guarantee global optimal solutions for some easier problems
- → Travelling Salesman Problem: given a set of n cities & distance b/w every pair of cities → find shortest route possible to visit every city. Greedy solution (nearest-neighbour solution) → in each step, travel to city with minimum distance from current city. (may not be optimal) → very fast, $O(n^2)$

E.g. Travelling Salesman Problem



Greedy solution → 10+11+10+34 = 64

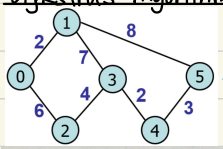
Optimal solution → 10+12+10+12 = 44

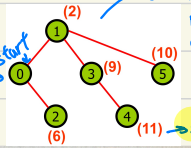
Worst route → 12+11+12+33 = 68

4 cities: (4-1)! = 6 routes

→ When greedy algorithm are optimal → shortest path problem (Dijkstra's Algorithm → optimal solution), minimum spanning tree problem (Kruskal's Algorithm)

Dijkstra's Algorithm





shortest path tree always remain as a tree when D Algorithm runs. → only include one path → can be multiple SPT from start vertex

(11) → shortest path to reach 4

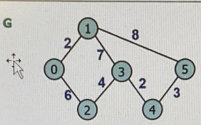
```
import heapq
# for priority queue, as opposed to FIFO queue
# each element in priority queue has a distance
# shorter the distance, higher the priority in queue
class Graph:
    def __init__(self, graph):
        self.adjacency_list = graph

    def dijkstra(self, src):
        V = len(self.adjacency_list)
        distances = [float('inf')] * V  # initialize to infinity
        distances[src] = 0  # source to source
        priority_queue = [(0, src)]  # initialize priority queue
        parent = [-1] * V  # parent node in the shortest path
        visited = set()  # visited nodes
```

distance

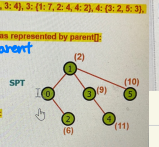
vertex

```
        while priority_queue:
            current_distance, current_node = heapq.heappop(priority_queue)
            if current_node in visited:
                continue
            visited.add(current_node)  # if not visited → add to visited

            for neighbor, weight in self.adjacency_list[current_node].items():
                if neighbor not in visited:
                    distance = current_distance + weight

                    if distance < distances[neighbor]:
                        distances[neighbor] = distance  # update
                        parent[neighbor] = current_node
                        heapq.heappush(priority_queue, (distance, neighbor))

        return distances, parent
```

```
graph = {
    0: {1: 2, 2: 8},  # weight
    1: {0:2, 3:7, 5:8},
    2: {0: 6, 3: 4},
    3: {1: 7, 2: 4, 4: 2},
    4: {3: 2, 5: 3},
    5: {1: 8, 4: 3}
}

source_vertex = 0
g = Graph(graph)
shortest_distances, parent = g.dijkstra(source_vertex)

print("Original Graph (adjacency list):")
print(graph)

print(f"\nShortest path tree from source vertex {source_vertex} as represented by parent[]:")
for i in range(len(parent)):
    print(f"parent[{i}] = {parent[i]}")

print(f"\nShortest distances from source vertex {source_vertex}:")
for i, distance in enumerate(shortest_distances):
    print(f"Shortest distance from {source_vertex} to {i}: {distance}")
```




```
Original Graph (adjacency list):
{0: {1: 2, 2: 8}, 1: {0: 2, 3: 7, 5: 8}, 2: {0: 6, 3: 4}, 3: {1: 7, 2: 4, 4: 2}, 4: {3: 2, 5: 3}, 5: {1: 8, 4: 3}}

Shortest path tree from source vertex 0 as represented by parent[]:
parent[0] = -1  → doesn't have parent
parent[1] = 0
parent[2] = 0
parent[3] = 1
parent[4] = 3
parent[5] = 1

Shortest distances from source vertex 0:
Shortest distance from 0 to 0: 0
Shortest distance from 0 to 1: 2
Shortest distance from 0 to 2: 8
Shortest distance from 0 to 3: 9
Shortest distance from 0 to 4: 11
Shortest distance from 0 to 5: 10
```



Dijkstra_Algorithm.ipynb
Dijkstra_Algorithm_trace.ipynb

```
V =  6
priority_queue =  [(0, 0)]

priority_queue BEFORE heappop =  [(0, 0)]
priority_queue AFTER heappop =  []
current_distance, current_node =  0 0
distances[0] = 0
visited = {0}

neighbor, weight =  1 2
distance =  2
distances[1] = inf > distance = 2: Update distances[1]!
priority_queue =  [(2, 1)]

neighbor, weight =  2 6
distance =  6
distances[2] = inf > distance = 6: Update distances[2]!
priority_queue =  [(2, 1), (6, 2)]
```

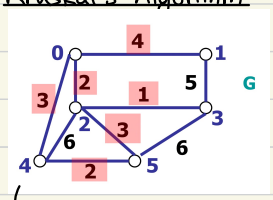
```
priority_queue BEFORE heappop =  [(2, 1), (6, 2)]
priority_queue AFTER heappop =  [(6, 2)]
current_distance, current_node =  2 1
distances[1] = 2
visited = {0, 1}


neighbor, weight =  3 7
distance =  9
distances[3] = inf > distance = 9: Update distances[3]!
priority_queue =  [(6, 2), (9, 3)]

neighbor, weight =  5 8
distance =  10
distances[5] = inf > distance = 10: Update distances[5]!
priority_queue =  [(6, 2), (9, 3), (10, 5)]

priority_queue BEFORE heappop =  [(6, 2), (9, 3), (10, 5)]
priority_queue AFTER heappop =  [(9, 3), (10, 5)]
current_distance, current_node =  6 2
```

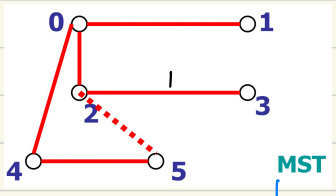
popped off

chosen at next node

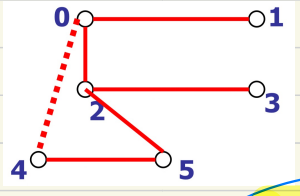
→ time complexity → $O(n^2)$

Kruskal's Algorithm





MST



↳ start drawing lines from the min weight

↳ minimum spanning tree (MST) → spanning tree of weighted graph with minimum total edge weight → no cycles

→ DisjointSet Class for Avoiding Cycles (disjoint set is not a minimum spanning tree)

↳ check if 2 sets roots are the same → if not same → merge two sets by making one set's root point to root of other set.

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = [-1] * vertices

    def find(self, node):
        if self.parent[node] == -1:
            return node
        return self.find(self.parent[node])

    def union(self, x, y):
        x_set = self.find(x)
        y_set = self.find(y)
        if x_set != y_set:
            self.parent[x_set] = y_set
```

all vertices are disjoint initially

recursive

find root of a set to which a vertex belongs

merge 2 sets

check if roots are same (if same, no join)

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.adjacency_list = []

    def add_edge(self, u, v, weight):
        self.adjacency_list.append((u, v, weight))

    def kruskal(self):
        self.adjacency_list.sort(key=lambda x: x[2])  # Sort edges by weight
        disjoint_set = DisjointSet(self.V)
        mst = []
        for edge in self.adjacency_list:
            u, v, weight = edge
            u_set = disjoint_set.find(u)
            v_set = disjoint_set.find(v)

            if u_set != v_set:
                mst.append((u, v, weight))
                disjoint_set.union(u, v)

        return mst
```

minimum spanning tree

check if they have same root

add edge to MST (no cycle)

```
graph = Graph(6)
graph.add_edge(0, 1, 4)
graph.add_edge(0, 2, 2)
graph.add_edge(0, 4, 3)
graph.add_edge(1, 3, 5)
graph.add_edge(2, 3, 1)
graph.add_edge(2, 4, 6)
graph.add_edge(2, 5, 3)
graph.add_edge(3, 5, 6)
graph.add_edge(4, 5, 2)

print("Minimum Spanning Tree (Kruskal's Algorithm):")
minimum_spanning_tree = graph.kruskal()

for edge in minimum_spanning_tree:
    u, v, weight = edge
    print(f"Edge: {u} - {v}, Weight: {weight}")
```

Program Output:

```
Minimum Spanning Tree (Kruskal's Algorithm):
Edge: 2 - 3, Weight: 1
Edge: 0 - 2, Weight: 2
Edge: 4 - 5, Weight: 2
Edge: 0 - 4, Weight: 3
Edge: 0 - 1, Weight: 4
```

Text Classifier

↳ assignment of text to one/more predefined categories based on text content

prediction function → $y = f_w(x)$ ← model parameters; output ← input

→ preprocessing text: removing stop words (punctuations, prepositions, pronouns), stemming (reducing words to root form), vectorisation (convert text into numerical vectors, bag-of-words)

A simple Python code for text classification (1)

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
# CountVectorizer: a class used to convert text into numerical features (a bag-of-words).
# train_test_split: a function to split the dataset into training and testing sets.
# MultinomialNB: the Multinomial Naive Bayes classifier
# accuracy_score: a function to calculate the accuracy of the classifier's predictions.

# Sample data: web page or document texts and their corresponding categories
# Total 6 categories, 24 data duples
data = [
    ("This is a technology news article.", "Technology"),
    ("Learn how to cook a delicious recipe.", "Cooking"),
    ("Financial market trends and analysis.", "Finance"),
    ("Discover the latest fashion trends.", "Fashion"),
    ("Health and wellness tips and advice.", "Health"),
    ("Travel destinations and vacation ideas.", "Travel"),
    ("New smartphone reviews and specifications.", "Technology"),
    ("Delicious dessert recipes and cooking tips.", "Cooking"),
    ("Stock market analysis and investment strategies.", "Finance"),
    ("Latest fashion trends and style inspiration.", "Fashion"),
    ("Fitness and nutrition advice for a healthy lifestyle.", "Health"),
    ("Travel stories and destination recommendations.", "Travel"),
]
```

A simple Python code for text classification (2)

```
# Split the combined data into features (text) and labels (categories)
texts, labels = zip(*data)

# Convert text data to numerical features using Count Vectorization
# These lines create a CountVectorizer object called vectorizer and
# use it to convert the text data (texts) into numerical features (X)
# using the bag-of-words representation. fit_transform both fits
# the vectorizer on the text data and transforms the text into numerical features.
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

# Split the dataset into training and testing sets
# These lines split the dataset into training and testing sets using train_test_split.
# X_train and y_train are the training features and labels, while X_test and y_test
# are the testing features and labels. The test_size parameter specifies the
# proportion of data to include in the testing set (in this case, 20%), and
# random_state is used to ensure reproducibility.
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2,
random_state=42)
```

A simple Python code for text classification (3)

```
# Train a Multinomial Naive Bayes classifier.
# create a Multinomial Naive Bayes classifier object called classifier and
# train it on the training data (X_train and y_train) using the fit method.
classifier = MultinomialNB()
classifier.fit(X_train, y_train)

# Predict the categories for test data.
# These lines use the trained classifier to make predictions on the test data (X_test)
# using the predict method. The predicted categories are stored in the y_pred
variable.
y_pred = classifier.predict(X_test)

# Calculate accuracyof the classifier's predictions by comparing the predicted labels
# (y_pred) with the true labels (y_test) using the accuracy_score function.
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")
```

A simple Python code for text classification (4)

```
# Additional input text to be classified
input_text = ["Get the latest updates on mobile technology."]
input_features = vectorizer.transform(input_text)
predicted_category = classifier.predict(input_features)[0]
print(f"Predicted category of {input_text}: {predicted_category}")

input_text = ["Smartphone buying guide"]
input_features = vectorizer.transform(input_text)
predicted_category = classifier.predict(input_features)[0]
print(f"Predicted category of {input_text}: {predicted_category}")
```

must use numerical value

Program Output:

```
Test Accuracy: 0.67
Predicted category of ['Get the latest updates on mobile technology.']: Fashion
Predicted category of ['Smartphone buying guide']: Technology
```

wrong

✓

prove $8^n > (2n-1)^2$ for $n \in \mathbb{Z}, n > 2$

base case:

$n = 3$

$8^3 = 512$

$[2(3)-1]^2 = 25$

$512 > 25$

Induction:

Assume $8^k > (2k-1)^2$

prove: $8^{k+1} > [2(k+1)-1]^2$

$[(2k-1)+2]^2 = [(2k-1)^2 + 4(2k-1) + 4]$

more

$8(8^k) = (1+7)8^k = 8^k + 7(8^k)$

more

since $k \geq 3$, $3(8^k) > 4$

$4(8^k) + 3(8^k)$

---

prove that $(n+1)! > 3^n$ for $n \in \mathbb{Z}, n \geq 4$

base case:

$(4+1)! = 120$

$3^4 = 81$

$120 > 81$

refer back to question

$((k+1)+1)! > 3^{(k+1)}$

$(k+2)! = (k+2)(k+1)! > (k+2)3^k$

for $k = 4$, $k+2 > 3$

$(k+2)3^k > 3(3^k)$

$> 3^{k+1}$

---

prove $n^4 + 3$ divided by 4 for $n \in \mathbb{Z}, n \geq 1$, n is an odd no.

$1^4 + 3 = 4$

therefore is $k+2$

$(k+2)^4 + 3$

$= (k+2)(k+2)(k+2)(k+2) + 3$

$= k^4 + 8k^3 + 24k^2 + 32k + 16 + 3$

$= (k^4 + 3) + 8(k^3 + 3k^2 + 4k + 2)$

can be divided by 4

8 can be divided by 4

$\lg(\frac{n}{2}) < \lg(n)$

---

show complexity of $\lg(n!)$ is $\Theta(n \lg n)$

$\lg(n!) \to \Theta(n \lg n)$

$\lg(n!) \leq c_1 n \lg n$

$\lg(n!) \geq (\frac{1}{2}) n \lg n$

$\lg(n!) = \lg(n \times (n-1) \times \dots \times 1)$

$= \lg n + \lg(n-1) + \dots + \lg 1$

$\leq \underbrace{\lg n + \lg n + \lg n \dots}_{n \text{ times}} = n \lg n$

$\lg(n!) = \lg(n \times (n-1) \times (n-2) \times \dots \times 2 \times 1)$

$= \lg n + \lg(n-1) + \dots + \lg 1$

$= \lg n + \lg(n-1) + \dots + \lg(\frac{n}{2}+1) + \lg(\frac{n}{2}) + \lg(\frac{n}{2}-1) + \dots \lg 2$

$\geq \underbrace{\lg(\frac{n}{2}) + \lg(\frac{n}{2}) + \dots + \lg(\frac{n}{2})}_{\frac{n}{2} \text{ times}} + \underbrace{\lg 2 + \lg 2 + \dots + \lg 2}_{\frac{n}{2}-1}$ (+ lg 1)

$= \frac{n}{2}\lg(\frac{n}{2}) + (\frac{n}{2}-1)\lg 2$

$= \frac{n}{2}\lg n + \frac{n}{2}\lg(\frac{1}{2}) + \frac{n}{2} - 1 = \frac{n}{2}\lg n - \frac{n}{2} + \frac{n}{2} - 1$

$= \frac{n}{2}\lg n - 1 = \frac{1}{2} n \lg n - 1$

not considering $\lg 1 = 0$

Determine order of growth of following function:

$(n+2)^2 \lg(\frac{n}{2}) + 2n \lg(n+2)^2$

Use $\Theta(g(n))$ notation with simplest $g(n)$ possible in your answers. Prove your assertions.

[$\lg x = \log_2 x$]

look for power bring down power

$\Theta(n)$?

look for $\lg(\frac{n}{2}) < \lg n$

look for integer values → smaller than n

$(n+2)^2 \lg(\frac{n}{2}) + 2n \lg(n+2)^2$

$\leq (n+2)^2 \lg(n) + 4n \lg(n+2)$ → $\lg(2n) = \lg 2 + \lg n = 1 + \lg n$

$\leq (n+n)^2 \lg(n) + 4n \lg(n+n)$

$= 4n^2 \lg(n) + 4n(\lg n + 1)$

$\leq 4n^2 \lg(n) + 4n(\lg n + \lg n)$

$= 4n^2 \lg(n) + 4n(2 \lg n)$

$= 4n^2 \lg(n) + 8n \lg n$

$\leq 4n^2 \lg(n) + 8n^2 \lg n$

$= 12n^2 \lg n \rightarrow O(n^2 \lg n)$

---

prove $\Omega(n^2 \lg n) \leq (n+2)^2 \lg(\frac{n}{2}) + 2n \lg(n+2)^2$ (A + B)

$A+B > A$ / $A+B > B$ can choose one to solve

$A+B \geq A$

$(n+2)^2 \lg(\frac{n}{2}) \geq n^2 \lg(\frac{n}{2})$

$= n^2(\lg n - 1)$

assume $n \geq 2$

$\geq n^2(\lg n - \frac{1}{2} \lg n)$ → if $n=2$, $\frac{1}{2} \lg 2 = \frac{1}{2}$ $(1 > \frac{1}{2})$

$= \frac{1}{2} n^2 \lg n$

→ $\Omega(n^2 \lg n)$