Proving Time Complexity

Proving upper bound:
$T(n) = O(f(n))$ iff $\forall n > k, |T(n)| \leq C|f(n)|$ for $C, k \in R$
Proving lower bound:
$T(n) = \Omega(f(n))$ iff $\forall n > k, |T(n)| \geq C|f(n)|$ for $C, k \in R$
Proving exact bound:
$T(n) = \Theta(f(n))$ iff $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Proving Correctness

Induction
(1) State predicate $P(n)$
(2) Show base case is true
(3) Assume $P(k)$ is true, label as inductive hypothesis
(4) Show that if $P(k)$ is true, then $P(k + 1)$ is also true
(5) Show that the function always terminates
(6) Conclude

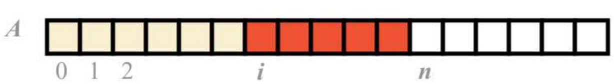
Loop invariant
(1) State loop invariant
(2) *Initialization* - Show that the invariant is true before the 1st iteration
(3) *Maintenance* - Show that the invariant is true for some arbitrary iteration
- USING INDUCTION: Assume the invariant holds after k iterations, and show that it still holds after $k + 1$ iterations

(4) Show that the function always terminates
(5) Conclude

Master Theorem

Given $T(n) = aT(n/b) + f(n)$,
Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
Case 3: If $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq \delta f(n)$ for $\epsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$

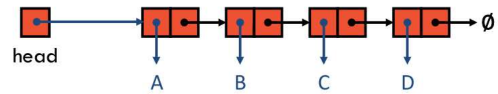
Arrays



Operations:
- **get(i):** $O(1)$
  Get element at index i
- **set(i, e):** $O(1)$
  Set element at index i
- **add(i, e):** $O(n)$
  Add element e at index i. Must shift every element after index i by $+ 1$.
- **remove(i):** $O(n)$
  Remove element e at index i. Must shift every element after index i by $- 1$.

Space complexity: $O(N)$ where N is the maximum number of elements the array can hold.

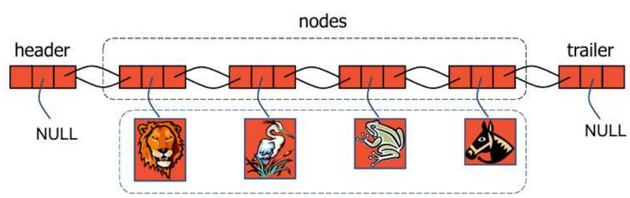
Singly Linked Lists



Operations:
- **first():** $O(1)$
  Get the element to which *head* points
- **last():** $O(n)$
  Get the element of which *next* points to *null*
- **before(p):** $O(n)$
  Get the previous element from node *p*
- **after(p):** $O(1)$
  Get the next element from node *p*
- **insert_before(p, e):** $O(n)$
  Insert a node (with element *e*) before node *p*
- **insert_after(p, e):** $O(1)$
  Insert a node (with element *e*) after node *p*
- **remove(p):** $O(n)$
  Remove the node to which *p* points
- **size():** $O(1)$ if size is tracked, otherwise $O(n)$
  Get the number of nodes in the SLL
- **is_empty():** $O(1)$
  Return *true* if *head* → *null,* otherwise return false

Space complexity: $O(n)$

Doubly Linked Lists



Operations: all SLL operations run in $O(1)$

Linked List
- good match to positional ADT
- efficient insertion and deletion
- simpler behaviour as collection grows
- modifications can be made as collection iterated over
- space not wasted by list not having maximum capacity

Arrays
- good match to index-based ADT
- caching makes traversal fast in practice
- no extra memory needed to store pointers
- allow random access (retrieve element by index)

Stack (LIFO)

Operations (with Array implementation)**:**

- **push(e):** $O(1)$
  Inserts an element e to the top of the stack
- **pop():** $O(1)$
  Removes and returns the top of the stack
- **top():** $O(1)$
  Returns the top of the stack without removing
- **size():**
  Returns the number of elements in the stack
- **isEmpty():** $O(1)$
  Return true if the stack is empty

Space complexity: $O(N)$ for Array implementation

Method stack frames: A method call causes a frame containing (1) local variable and return value, and (2) program counter to be pushed to the stack

Queue (FIFO)

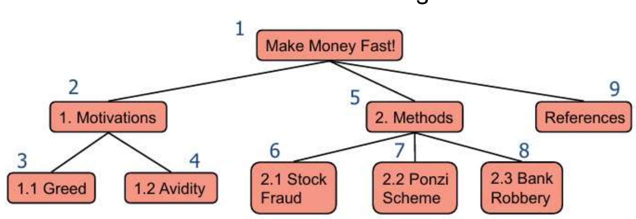
Operations: All operations run in $O(1)$ if the queue is implemented as an Array

Space complexity: $O(N)$ for Array implementation

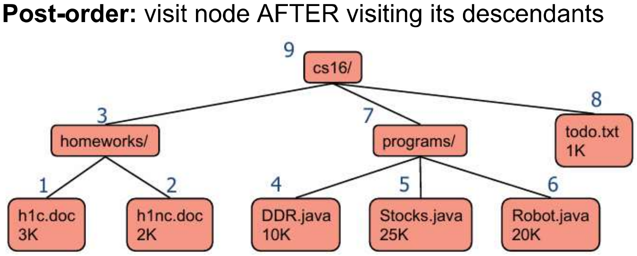
Tree

Root: node without parent
Depth: num of ancestors a node has, not including itself
Level: set of nodes with given depth
Height of a tree: maximum depth (depth of lowest leaf)
Subtree: tree made up of a node and its descendants
Edge: node pair (u, v) s. t. one is the parent of the other

Pre-order: visit node BEFORE visiting its descendants



Post-order: visit node AFTER visiting its descendants



Operations:

- **search(e):** $O(n)$, might need to traverse all node
- **size():** $O(1)$
- **isEmpty():** $O(1)$
- **root():** $O(1)$
- **parent(p):** $O(1)$
- **children(p):** $O(c)$ where c is the # children of p
- **numChildren(p):** $O(1)$
- **isInternal(p):** $O(1)$
- **isRoot(p):** $O(1)$

Space complexity: $O(n)$

Binary tree

Properties:

- Each node has at most 2 children
- Child ordering is left followed by right

Node structure:

```
class Node:
       element;
       parent;
       leftChild;
       rightChild;
```

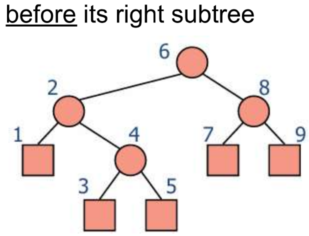
Proper: We say the binary tree is proper if every internal node has two children

Operations: Inherit all operations from `Tree`, with additionals the these additional operations,

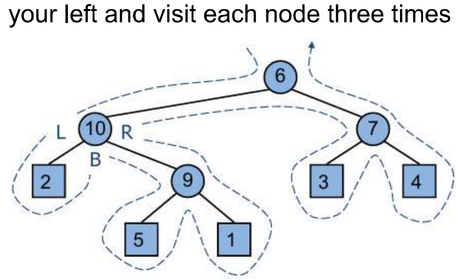
- **leftChild(p):** $O(1)$
- **rightChild(p):** $O(1)$
- **sibling(p):** $O(1)$

Space complexity: $O(n)$

Inorder traversal: visit node after its left subtree but before its right subtree



Euler tour traversal: walk around the tree, keeping it on your left and visit each node three times



Euler tour traversal example as above:
6,10,2,2,2,10,9,5,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,7,6

Binary Search Trees (BST)

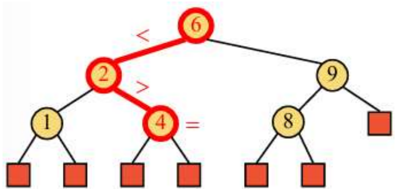
Defining property: A binary tree where for

- any node v,
- any node u in the left subtree of v
- any node w in the right subtree of v

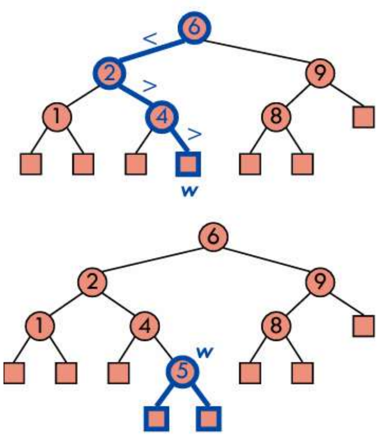
we have **key(u)** $<$ **key(v)** $<$ **key(w)**
which also means duplicate key values are not allowed

Space complexity: $O(n)$

search(v): $O(n)$, with $log\ n$ average
1) if **v** = key(**v**) or **v** is leaf, then return **v**
2) if **v** < key(**v**), then go to left child
3) if **v** > key(**v**), then go to right child



put(k, o): $O(n)$, with $log\ n$ average
1) if **k** is found, replace its value with **o**
2) if **k** isn't found, replace the leaf with an internal node holding with key **k** and value **o**



remove(k): $O(n)$, with $log\ n$ average

```
def remove(k)
w ← search(k, root)
 if w.isExternal() then
 return null
 else if w has at least one external child
z then
 remove z
 promote the other child of w to take w's
place
 remove w
 else
 y ← immediate successor of w
replace contents of w with entry from y
 remove y as above
```

Range query: $O(|output| + tree\ height)$
Find all keys k such that $k_1 \leq k \leq k_2$

- If key(v) $< k_1$ then search right subtree
- If $k_1 \leq$ key(v) $\leq k_2$ then search left subtree, add v to range output, and search right subtree
- If $k_2 <$ key(v) then search left subtree

Use `range_search` from the lecture slides

AVL Tree

Balance constraint: Ranks of the 2 children of every internal node differ by at most 1

Height of tree: $O(\log n)$
Space complexity: $O(\log n)$

Operations:

- **search(v):** $O(\log n)$
- **insert(k, v):** $O(\log n)$
- **remove(k):** $O(\log n)$

Trinode restructure: If after modification, the AVL tree loses its AVL property, then we do a trinode restructure to rebalance. This runs in $O(1)$ since it involves updating $O(1)$ pointers.

Hash Map

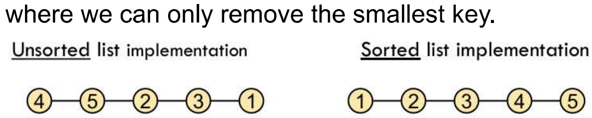
Operations:

- **get(k):** $O(n)$ worst, $O(1)$ average
- **put(k, v):** $O(n)$ worst, $O(1)$ average
- **remove(l):** $O(n)$ worst, $O(1)$ average
- **size():** $O(1)$
- **isEmpty():** $O(1)$
- **entrySet():** $O(n)$
- **keySet():** $O(n)$
- **values():** $O(n)$

Space complexity: $O(n)$

Priority Queue

A special type of map ADT to store key-value items where we can only remove the smallest key.



Method	Unsorted List	Sorted List
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Space complexity: $O(n)$

PQ-Sort

PQ-sort: $O(n^2)$

```
def priority_queue_sorting(A):
    pq ← new priority queue
    n ← size(A)
    for i in [0:n] do
        pq.insert(A[i])
    for i in [0:n] do
        A[i] = pq.remove_min()
```

Selection-sort

Inserting elements with n insert operations take $O(n)$

Rem. elements with n remove_min operations take $O(n^2)$

i	A	s
0	7, 4, 8, 2, 5, 3, 9	3
1	2, 4, 8, 7, 5, 3, 9	5
2	2, 3, 8, 7, 5, 4, 9	5
3	2, 3, 4, 7, 5, 8, 9	4
4	2, 3, 4, 5, 7, 8, 9	4
5	2, 3, 4, 5, 7, 8, 9	5
6	2, 3, 4, 5, 7, 8, 9	6

```
def selection_sort(A):
    n ← size(A)
    for i in [0:n] do
        # find s ⩾ i minimizing A[s]
        s ← i
        for j in [i:n] do
            if A[j] < A[s] then
                s ← j
        # swap A[i] and A[s]
        A[i], A[s] ← A[s], A[i]
```

Insertion-sort

Inserting elements with n insert operations take $O(n^2)$

Rem. elements with n remove_min operations take $O(n)$

i	A	j
1	7, 4, 8, 2, 5, 3, 9	0
2	4, 7, 8, 2, 5, 3, 9	2
3	4, 7, 8, 2, 5, 3, 9	0
4	2, 4, 7, 8, 5, 3, 9	2
5	2, 4, 5, 7, 8, 3, 9	1
6	2, 3, 4, 5, 7, 8, 9	6

```
def insertion_sort(A):
    n ← size(A)
    for i in [1:n] do
        x ← A[i]
        # move forward entries > x
        j ← i
        while j > 0 and x < A[j-1] do
            A[j] ← A[j-1]
            j ← j - 1
        # if j>0 ⇒ x ≥ A[j-1]
        # if j<i ⇒ x < A[j+1]
        A[j] ← x
```

Heap-sort: PQ-sort with heap, runs in $O(n \log n)$ time

Heap

Binary tree storing (key, value) pairs at its node satisfying the following properties:

1. **Heap-Order:** For every node $m \neq$ root, we have key(m) ≥ key(parent(m))
2. **Complete binary tree:** Let h be height
   - Every level $i < h$ has 2^i nodes
   - Nodes at level h take the leftmost pos., where last node is the rightmost node

Height: $O(\log n)$

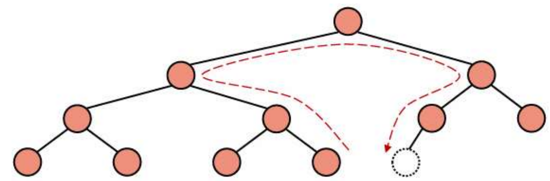
Space complexity: $O(n)$

Min heap: The minimum key is at the root

Max heap: The maximum key is at the root

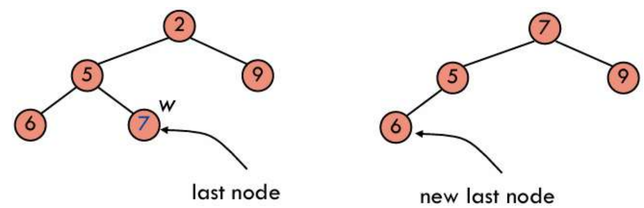
Insert: $O(\log n)$

1. Start from last node
2. Go up until left child or the root is reached
3. If root reached, then we need to open new lvl
4. Otherwise, go to sibling (right child of parent)
5. Go down left until a leaf is reached
6. Create node and restore heap-order property



Removal: $O(\log n)$

1. Replace root key with key of last node
2. Delete last node
3. Restore heap-order property



Upheap: $O(\log n)$

Restore heap-order property by swapping keys along upward path from insertion point

```
def up_heap(z):
    while z ≠ root and
    key(parent(z)) > key(z) do
        swap key of z and parent(z)
        z ← parent(z)
```

Downheap: $O(\log n)$

Restore heap-order property by swapping keys along downward path from root

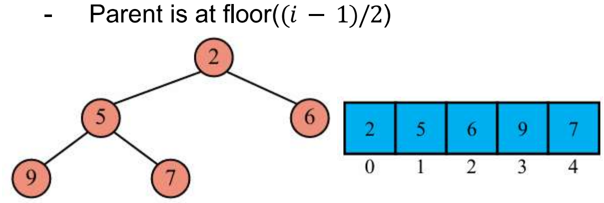
```
def down_heap(z):
    while z has child with
     key(child) < key(z) do
        x ← child of z with smallest key
        swap keys of x and z
        z ← x
```

Operations:

Operation	Time
size, isEmpty	$O(1)$
min,	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Array implementation:

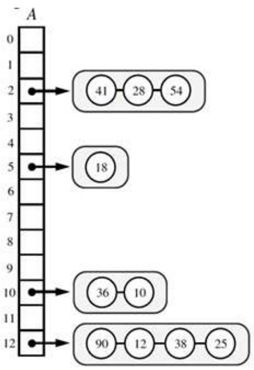
- Root is at 0
- Last node is at $n - 1$
- Left child is at $2i + 1$ and right child is at $2i + 2$
- Parent is at floor($(i - 1)/2$)



Collision Handling

Collision: A situation where 2 or more elements are hashed to the same location.

Separate chaining: Let each cell in the table point to a linked list holding the entries that map there



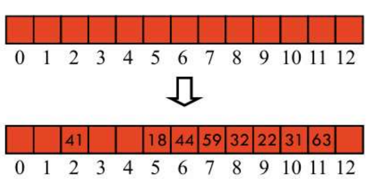

```
def get(k):
    return A[h(k)].get(k)

def put(k,v):
    return A[h(k)].put(k,v)

def remove(k):
    return A[h(k)].remove(k)
```

Operations: All map operations with separate chaining run in:

- $O(1 + \alpha)$ EXPECTED time where α is the load factor n/N of the hash table
- $O(n)$ WORST time (occurring when all items collide into a single chain)

Linear probing: Place colliding item in the next (circularly) available cell.

Example with h(x) = x mod 13.
Suppose we sequentially insert:
18 [5], 41 [2], 22 [9],
44 [5], 59 [7], 32 [6],
31 [5], 63 [11]



DEFUNCT: A special object to replace deleted elements

get(k): $O(n)$ WORST, $O(1)$ EXPEC. for load factor < 1
Must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell

put(k,v): $O(n)$ WORST, $O(1)$ EXPEC. for load factor < 1
Search for the entry as in get(k), but we also remember the index j of the first cell we find that has DEFUNCT or empty. If we find key k, we replace the value there with v and return the previous value. If we don't find k, we store (k, v) in cell with index j

remove(k): $O(n)$
Search for the entry as in get(k). If found, replace it with the special item DEFUNCT and return element v

Cuckoo hashing:

```
def get(k):
 if T1[h1(k)] ≠ null and T1[h1(k)].key = k
then
 return T1[h1(k)].value
 if T2[h2(k)] ≠ null and T2[h2(k)].key = k
then
 return T2[h2(k)].value
 return null

def remove(k):
 temp ← null
 if T1[h1(k)] ≠ null and T1[h1(k)].key = k:
   temp ← T1[h1(k)].value
   T1[h1(k)] ← null
 if T2[h2(k)] ≠ null and T2[h2(k)].key = k:
   temp ← T2[h2(k)].value
   T2[h2(k)] ← null
 return temp
```

Operations:

- **get(k):** $O(1)$
- **put(k, v):** $O(n)$
- **remove(k):** $O(1)$

Set

Unordered collection of elements WITHOUT DUPLICATES, usually implemented with hashmap

Operations:

- **add(e):** $O(1)$
- **remove(e):** $O(1)$
- **contains(e):** $O(1)$
- **iterator():** $O(n)$
- **addAll(T):** $O(|T|)$, this is same as $S \cup T$
- **retainAll(T):** $O(n)$, this is same as $S \cap T$
- **removeAll(T):** $O(n)$, this is same as $S \cup T$

MultiSet

Like the Set ADT, but allows duplicates. It is implemented by a Map where the element is the key, and the number of occurrences is the value.

Operations:
extends Set ADT

- **count(e):** $O(1)$
- **remove(e):** $O(1)$

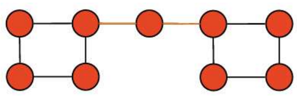
Graph

A graph G is a pair (V, E) where
- V: set of vertices
- E: collection of pairs of vertices called edges

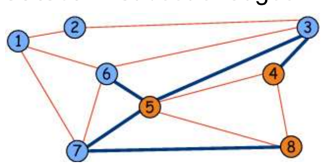
Path: A sequence of vertices such that every pair of consecutive vertices is connected by an edge

Simple path: A path where all vertices are distinct
Cycle: Path that starts and ends at the same vertex
Simple cycle: Cycle where all vertices are distinct
Acyclic graph: Graph that has no cycles

Sum of degrees: $\sum_{v \in V} \deg(v) = 2m$ where m is # edges
Cut edge: For a connected graph $G = (V, E)$, $(u, v) \in E$ is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected.

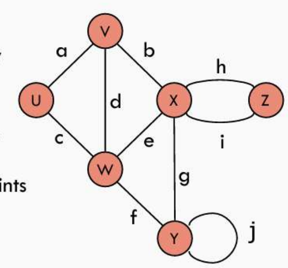


Cutset: A subset of edges



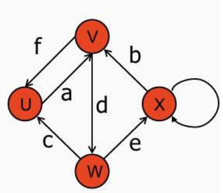
Terminology (Undirected graphs)

- Edges connect endpoints
  e.g., W and Y for edge f
- Edges are incident on endpoints
  e.g., a, d, and b are incident on V
- Adjacent vertices are connected
  e.g., U and V are adjacent
- Degree is # of edges on a vertex
  e.g., X has degree 5
- Parallel edges share same endpoints
  e.g., h and i are parallel
- Self-loop have only one endpoint
  e.g., j is a self-loop
- Simple graphs have no parallel or self-loops



Terminology (Directed graphs)

- Edges go from tail to head
  e.g., W is the tail of c and U its head
- Out-degree is # of edges out of a vertex
  e.g., W has out-degree 2
- In-degree is # of edges into a vertex
  e.g., W has in-degree 1
- Parallel edges share tail and head
  e.g., no parallel edge on the right
- Self-loop have same head and tail
  e.g., X has a self-loop
- Simple directed graphs have no parallel or self-loops, but are allowed to have anti-parallel loops like f and a

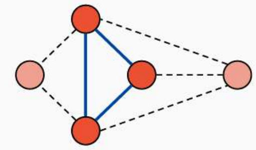


Graph concepts: Subgraphs

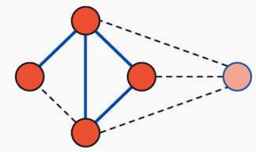
Let G=(V, E) be a graph. We say S=(U, F) is a subgraph of G if U ⊆ V and F ⊆ E

A subset U ⊆ V induces a graph G[U] = (U, E[U]) where E[U] are the edges in E with endpoints in U

A subset F ⊆ E induces a graph G[F] = (V[F], F) where V[F] are the endpoints of edges in F



Subgraph induced by red vertices

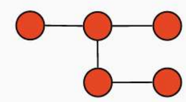


Graph concepts: Trees and Forests

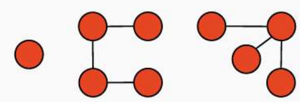
An unrooted tree T is a graph such that
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has n-1 edges



Tree



Forest

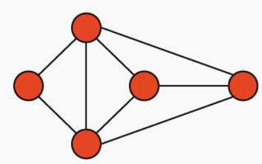
Spanning Trees and Forests

A spanning tree is a connected subgraph on the same vertex set

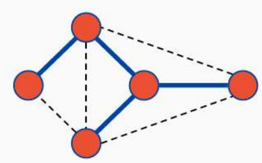
A spanning tree is not unique unless the graph is a tree

Spanning trees have applications to the design of communication networks

A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Represent
- **numVertices():** $O(1)$
- **vertices():** $O(1)$
- **numEdges:** $O(1)$
- **edges():** $O(1)$
- **getEdges(u, v):** $O(1)$
- **endVertices(e):** $O(1)$
- **opposite(v, e):** $O(1)$
- **outDegree(v):** $O(|E|)$
- **inDegree(v)”** $O(|E|)$
- **outgoingEdges(v):** $O(|E|)$
- **incomingEdges(v):** $O(|E|)$
- **insertVertex(x):** $O(1)$
- **insertEdge(u, v, x):** $O(1)$
- **removeVertex(v):** $O(|V| + E)$
- **removeEdge(e):** $O(1)$

Ways to represent a graph:

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n+m)$	$O(n+m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	O(deg(v))	$O(n)$
getEdge(u, v)	$O(m)$	O(min(deg(u), deg(v)))	O(1)
insertVertex(x)	O(1)	O(1)	$O(n^2)$
insertEdge(u, v, x)	O(1)	O(1)	O(1)
removeVertex(v)	$O(m)$	O(deg(v))	$O(n^2)$
removeEdge(e)	O(1)	O(1)	O(1)

Set up edge list: $O(1)$
Assume the input is given as (V, E), then

```
class Graph:
    def __init__(V, E):
        edges = E
```

Set up adjacency list: $O(|V| + |E|)$
Assume the input is given as (V, E), then

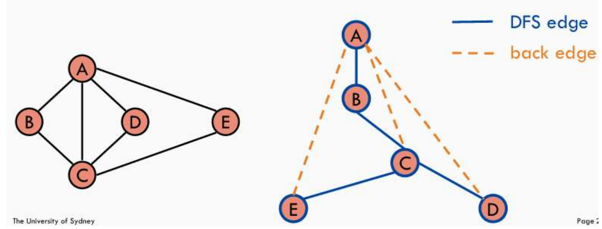
```
class Graph:
    def __init__(V, E):
        adj_list = {}
        for v in V:
            adj_list[v] = []
        for (u, v) in E:
            adj_list[u].append(v)
            # For undirected graph
            adj_list[v].append(u)
```

Set up adjacency matrix: $O(|V|^2) + O(|E|) = O(|V|^2)$
Assume the input is given as (V, E), then

```
class Graph:
    def __init__(V, E):
        adj_matrix = |V| x |V| zero matrix
        for (u, v) in E:
            adj_matrix[u][v] = 1
            # For undirected graph
            adj_matrix[v][u] = 1
```

Depth-First Search (DFS)

Follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if stuck.




```
def DFS(G):

  # set things up for DFS
  for u in G.vertices() do
    visited[u] ← False
    parent[u] ← None

  # visit vertices
  for u in G.vertices() do
    if not visited[u] then
      DFS_visit(u)

  return parent
```

```
def DFS_visit(u):

  visited[u] ← True

  # visit neighbors of u
  for v in G.incident(u) do
    if not visited[v] then
      parent[v] ← u
      DFS_visit(v)
```

Time complexity: $O(|V| + |E|)$ assuming adjacency list
Applications:

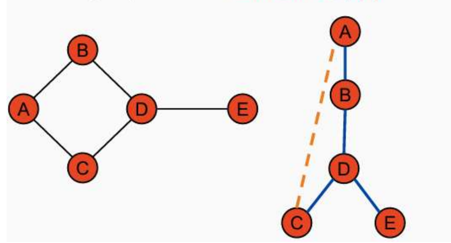
- Find a path between 2 vertices, if any
- Find a cycle in the graph
- Test if a graph is connected
- Find connected components of a graph
- Find spanning tree of a graph (if connected)

Identifying cut edges in O(n+m) time

Compute a DFS tree of the input graph G=(V, E)

For every u in V, compute level[u], its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this down_and_up[v]



	level	d&u
A	0	0
B	1	0
C	3	0
D	2	0
E	3	3

Breadth-First Search (BFS)

Visit all vertices at distance k from a start vertex s before visiting vertices at distance $k + 1$

```
def BFS(G,s):

  # set things up for BFS
  for u in G.vertices() do
     seen[u] ← False
     parent[u] ← None

  seen[s] ← True
  layers ← []
  current ← [s]
  next ← []

  # process current layer
  while not current.is_empty() do
     layers.append(current)
     # iterate over current layer
     for u in current do
        for v in G.incident(u) do
           if not seen[v] then
              next.append(v)
              seen[v] ← True
              parent[v] ← u
     # update curr and next layers
     current ← next
     next ← []

  return layers
```

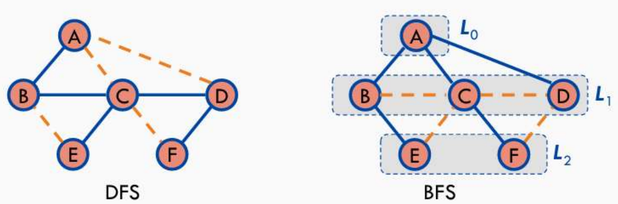
O(n) time

O($\sum_u$ deg(u)) = O(m) time

Time complexity: $O(|V| + |E|)$ assuming adjacency list

Applications:
- Find shortest path between 2 vertices
- Find a cycle in the graph
- Test if a graph is connected
- Find the spanning tree of a graph (if connected)

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Cut edges	√	



DFS BFS

Dijkstra's Shortest Path Algorithm

Finding shortest path from one vertex to another in a weighted graph

Inputs:
- Graph $G = (V, E)$
- Edges weights: $w: E \to R_+$
- Start vertex: s

Output:
- Distance from s to all $v \in V$
- Shortest path tree rooted at s

Assumptions:
- G is connected and undirected
- Edge weights are nonnegative

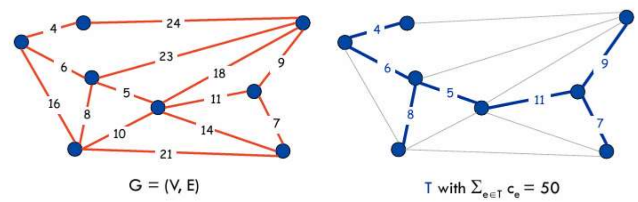
Pseudocode:

```
def Dijkstra(G, w, s):
 for v in V do
     D[v] ← ∞
     parent[v] ← ∅
 D[s] ← 0
 Q ← new PQ for { (v, D[v]) : v in V }
 while Q is not empty do
     u ← Q.remove_min()
     for z in G.neighbors(u) do
         if D[u] + w[u, z] < D[z] then
             D[z] ← D[u] + w[u, z]
             Q.update_priority(z, D[z])
             parent[z] ← u
 return D, parent
```

Time complexity: $O(|E| \log |V|)$ using HEAP,

Minimum Spanning Tree (MST)

MST: Given weighted connected graph $G = (V, E)$, an MST is a subset of the edges $T \subseteq E$ s.t. T is a spanning tree whose sum of edge weights is minimised



Simplifying assumption: all edge costs are distinct

Cut property: Let S be any subset of nodes and let e be the min. Cost edge with exactly one endpoint in S, then the MST contains e

Cycle property: Let C be any cycle, and let f be the max cost edge belonging to C. Then the mST does not contain f

Union Find

Assume an implementation using the Tree structure

Operations:
- **makeSets(A):** $O(n)$
  makes |A| singleton sets with elements in A
- **find(a):** $O(\log n)$, assuming tree structure
  search up the tree recursively until it finds rep.
- **union(a, b):** $O(\log n)$, assuming tree structure
  let the rep. vertex of one connected component be the parent of the rep. of another component

  i.e., union(1, 2) => Parent(find(2)) = find(1)

Space complexity: $O(n)$

Prim's Algorithm for MST

```
def prim(G, c):
 u ← arbitrary vertex in V
 S ← { u }
 T ← Ø
 while |S| < |V| do
  (u, v) ← min cost edge s.t. u in S and v not in S
  add (u, v) to T
  add v to S
 return T
```

Intuitive procedure:

1. Select the smallest edge (lowest weight)
2. Select the smallest connect edge
3. Repeat

Time complexity: $O(|E| \log |V|)$ using HEAP

Kruskal's Algorithm for MST

```
def Kruskal(G,c):

 sort E in increasing c-value
 answer ← [ ]
 comp ← make_sets(V)
 for (u,v) in E do
  if comp.find(u) ≠ comp.find(v) then
   answer.append( (u,v) )
   comp.union(u, v)
 return answer
```

Intuitive procedure: Select minimum cost edges such that it does not form a cycle

Time complexity: $O(|V||E|)$, or $O(n^2)$ if $|V| = |E|$

Space complexity: $O(|V| + |E|)$

Generic Greedy Template

```
def generic_greedy(input):

 # initialization
 initialize result

 determine order in which to consider input

 # iteratively make greedy choice
 for each element i of the input (in above order) do
  if element i improves result then
   update result with element i

 return result
```

Fractional Knapsack

Given: A set S of n items, each having a positive benefit b_i and a positive weight w_i

Objective: Maximise $\sum_{i \in S} b_i (x_i / w_i)$

Constraints:

- Total weight is bounded by W i.e, $\sum_{i \in S} x_i \leq W$
- Individual weight is bounded i.e., $0 \leq x_i \leq w_i$

Pseudocode:

```
def fractional_knapsack(b, w, W):

 # initialization
 x ← array of size |b| of zeros
 curr ← 0

 # iteratively do greedy choice
 for i in descending b[i]/w[i] order do
  x[i] ← min(w[i], W - curr)
  curr ← curr + x[i]
 return x
```

Time complexity: $O(n \log n)$ to sort the items, and $O(n)$ to process them in the for-loop

Interval Partitioning

```
def interval_partition(S):

 # initialization
 sort intervals in increasing starting time order
 d ← 0   # number of allocated classrooms

 # iteratively do greedy choice
 for i in increasing starting time order do
  if lecture i is compatible with some classroom k then
   schedule lecture i in classroom 1 ≤ k ≤ d
  else
   allocate a new classroom d+1
   schedule lecture i in classroom d+1
   d ← d+1
 return d
```

Time complexity: $O(n \log n)$

Huffman Coding

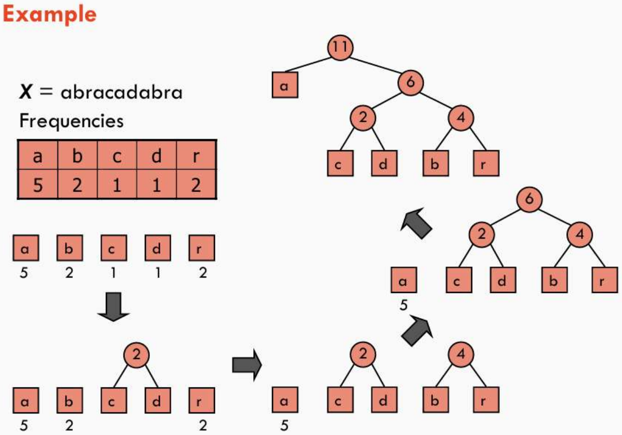
Objective: Minimise bits needed to encode the given string X i.e., $\sum_{c \in C} f(c) \times depth_T(c)$

```
def huffman(C, f):

  # initialize priority queue
  Q ← empty priority queue
  for c in C do
    T ← single-node binary tree storing c
    Q.insert(f[c], T)

  # merge trees while at least two trees
  while Q.size() > 1 do
    f1, T1 ← Q.remove_min()
    f2, T2 ← Q.remove_min()
    T ← new binary tree with T1/T2 as left/right subtrees
    f ← f1 + f2
    Q.insert(f, T)

  # return last tree
  f, T ← Q.remove_min()
  return T
```

Time complexity: $O(|C| \log |C|)$, or $O(n \log n)$ where n is # of unique characters

Space complexity: $O(n)$ for storing the Huffman tree and code table.

Example



Divide and Conquer

1. **Divide:** If it's a base case, solve it directly. Otherwise, break up the problem into several parts
2. **Recur:** Recursively solve each part
3. **Combine:** combine the solutions of each part into the overall solution

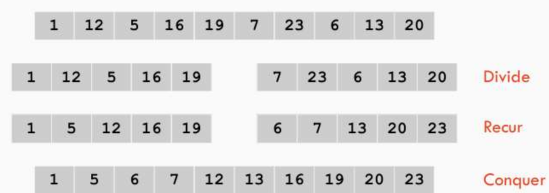
Recurrence relation:

$$T(n) = \begin{cases} \text{“Recur” + “Divide and Conquer”} & \text{for } n > 1 \\ \text{“Base case” (typically } O(1)\text{)} & \text{for } n = 1 \end{cases}$$

Merge-Sort

Merge-Sort

1. Divide the array into two halves.
2. Recur recursively sort each half.
3. Conquer two sorted halves to make a single sorted array.




```
def merge_sort(S):
    # base case
    if |S| < 2 then
        return S

    # divide
    mid ← ⌊|S|/2⌋
    left ← S[:mid]   # doesn't include S[mid]
    right ← S[mid:]  # includes S[mid]


    # recur
    sorted_left ← merge_sort(left)
    sorted_right ← merge_sort(right)

    # conquer
    return merge(sorted_left, sorted_right)
```

Time complexity:

$$T(n) = \begin{cases} 2\,T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n \log n)$

Some recurrence solutions

Recurrence	Solution
$T(n) = 2\,T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2\,T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2\,T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

Quick-Sort

Quick sort

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively sort the less than and greater than lists
3. Conquer Join the sorted 3 lists together

pivot

1	7	6	16	19	7	13	5	13	19	
1 6 5			7 7			16 19 13 13 19				divide
1 5 6			7 7			13 13 16 19 19				recur
1	5	6	7	7	13	13	16	19	19	conquer

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n \log n)$ expected time

Logarithms facts

Base exchange rule:

$$\log_a x = (\log_b x)/(\log_b a)$$

Product rule:

$$\log_a (xy) = (\log_a x) + (\log_a y)$$

Power rule:

$$\log_a x^b = b \log_a x$$

Master Theorem

Master Theorem

Let $f(n)$ and $T(n)$ be defined as follows:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Depending on a, b and $f(n)$ the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \leq \delta\, f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,