GRAPHS

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Data structures
  - Array-based
  - Stacks and queues
  - Linked Lists
  - Trees
  - Priority Queues
  - Maps and Hash tables
  - Search Trees
- Algorithms
  - Sorting and Selection

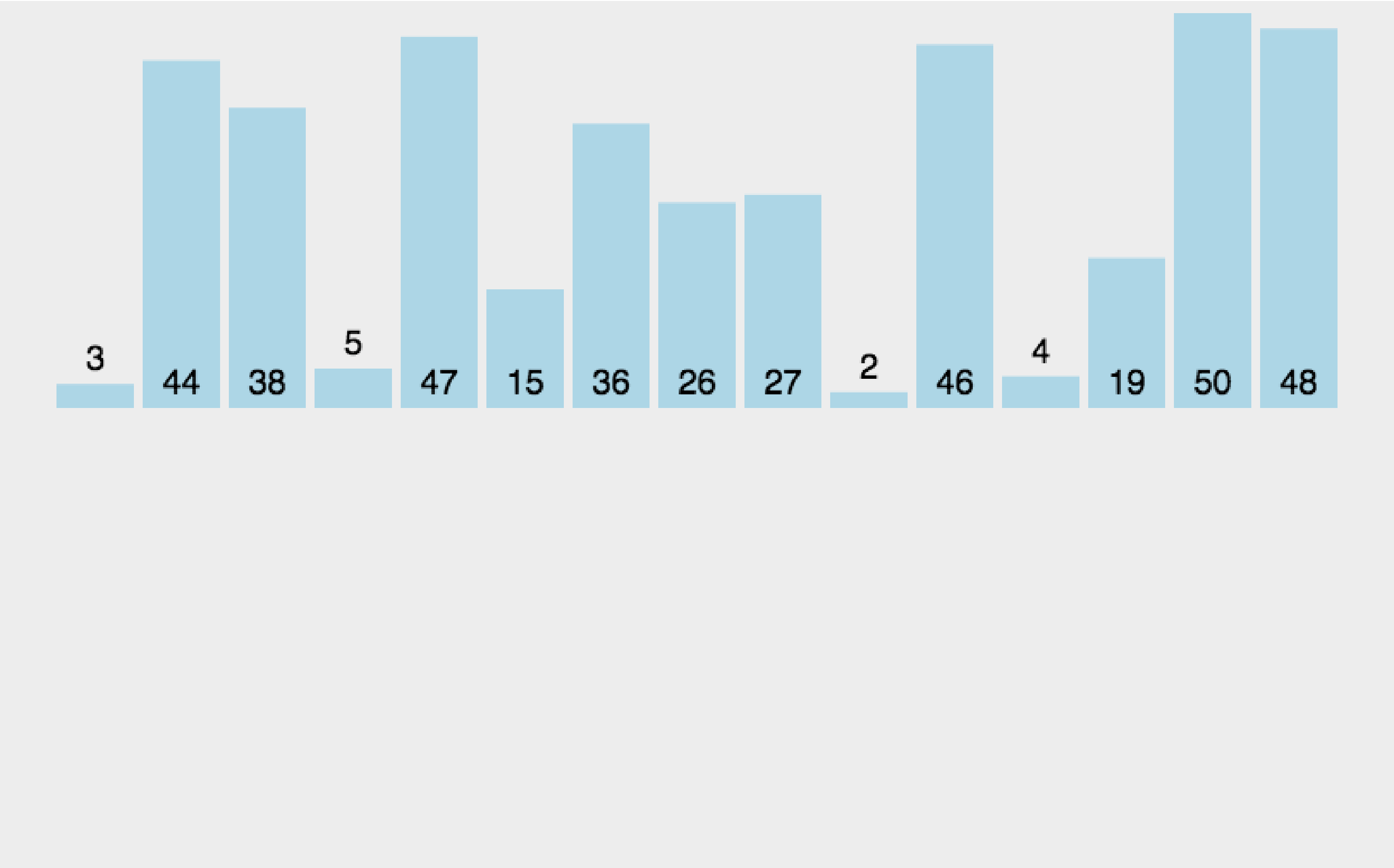
INTERVIEW QUESTIONS

- From Facebook
- Given an array of n elements, ranging from 0 to N, determine if the array contains at least 1 element from 0 to N. What would be the space efficiency and the time efficiency?
- You are given a collection of (city, population) entries, provide an algorithm, that randomly selects a pair, so that its population is proportional to the probability that the entry is selected.

INSERTION SORT

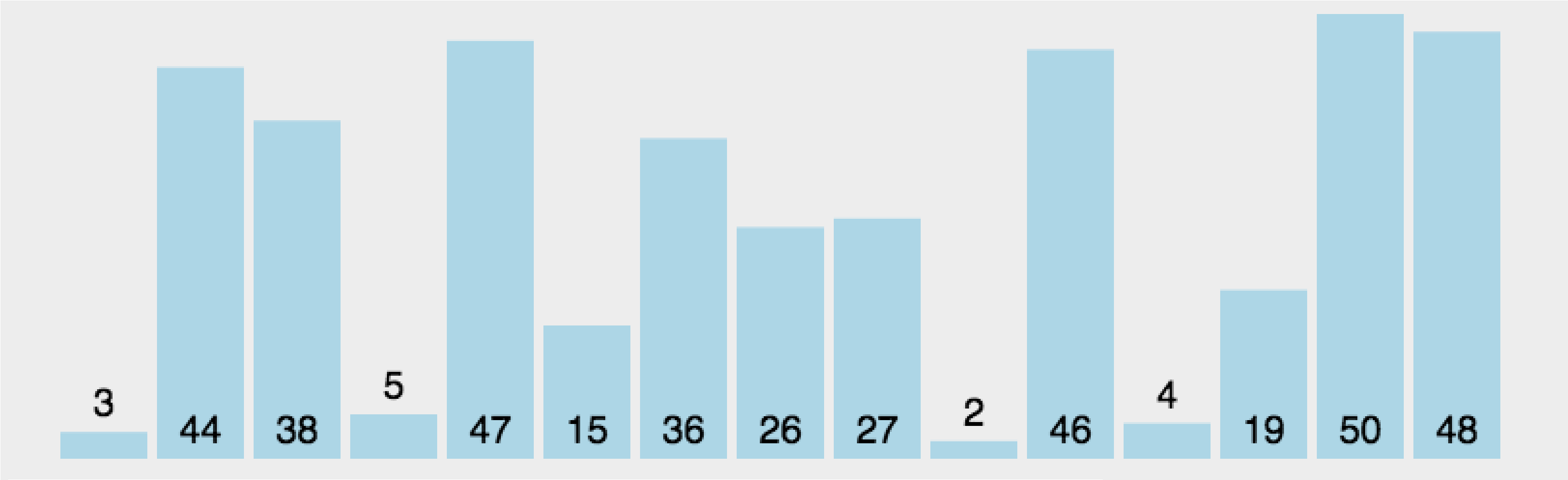
- Insertion sort

```
def insertion_sort(A):
  """Sort list of comparable elements
  for k in range(1, len(A)):
    cur = A[k]
    j = k
    while j > 0 and A[j-1] > cur:
      A[j] = A[j-1]
      j -= 1
    A[j] = cur
```



SELECTION SORT

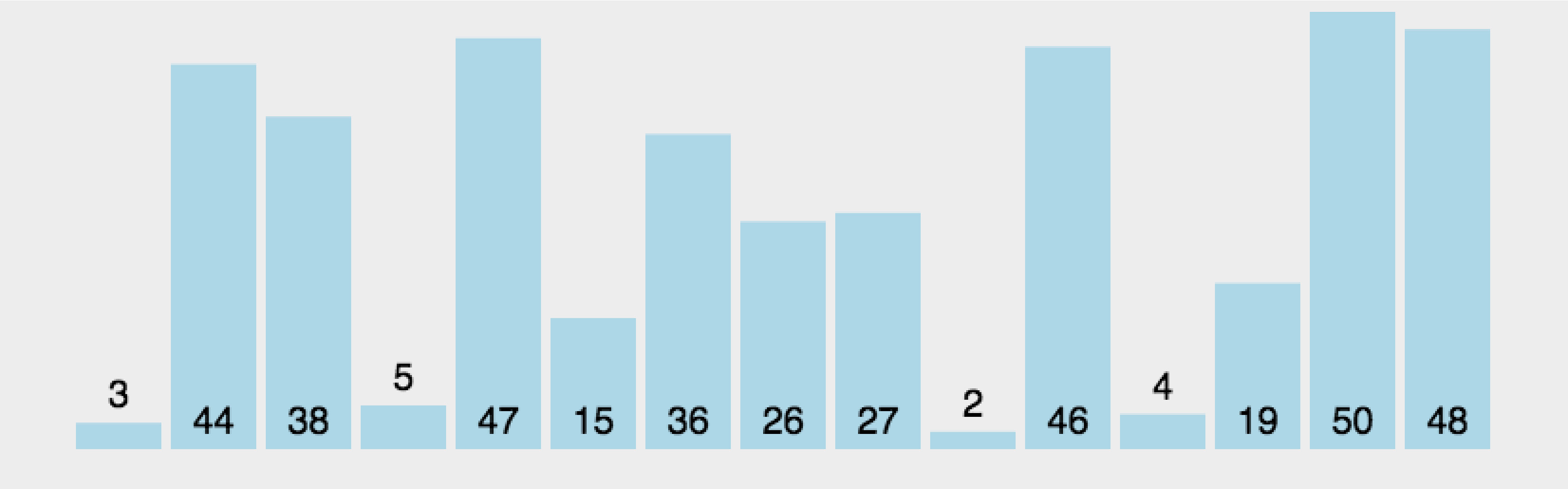
- Selection sort
- We have seen this
  - In Priority Queue




```python
def selectionSort(arr):
    for i in range(len(arr) - 1):
        # 记录最小数的索引
        minIndex = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[minIndex]:
                minIndex = j
        # i 不是最小数时，将 i 和最小数进行交换
        if i != minIndex:
            arr[i], arr[minIndex] = arr[minIndex], arr[i]
    return arr
```

BUBBLE SORT

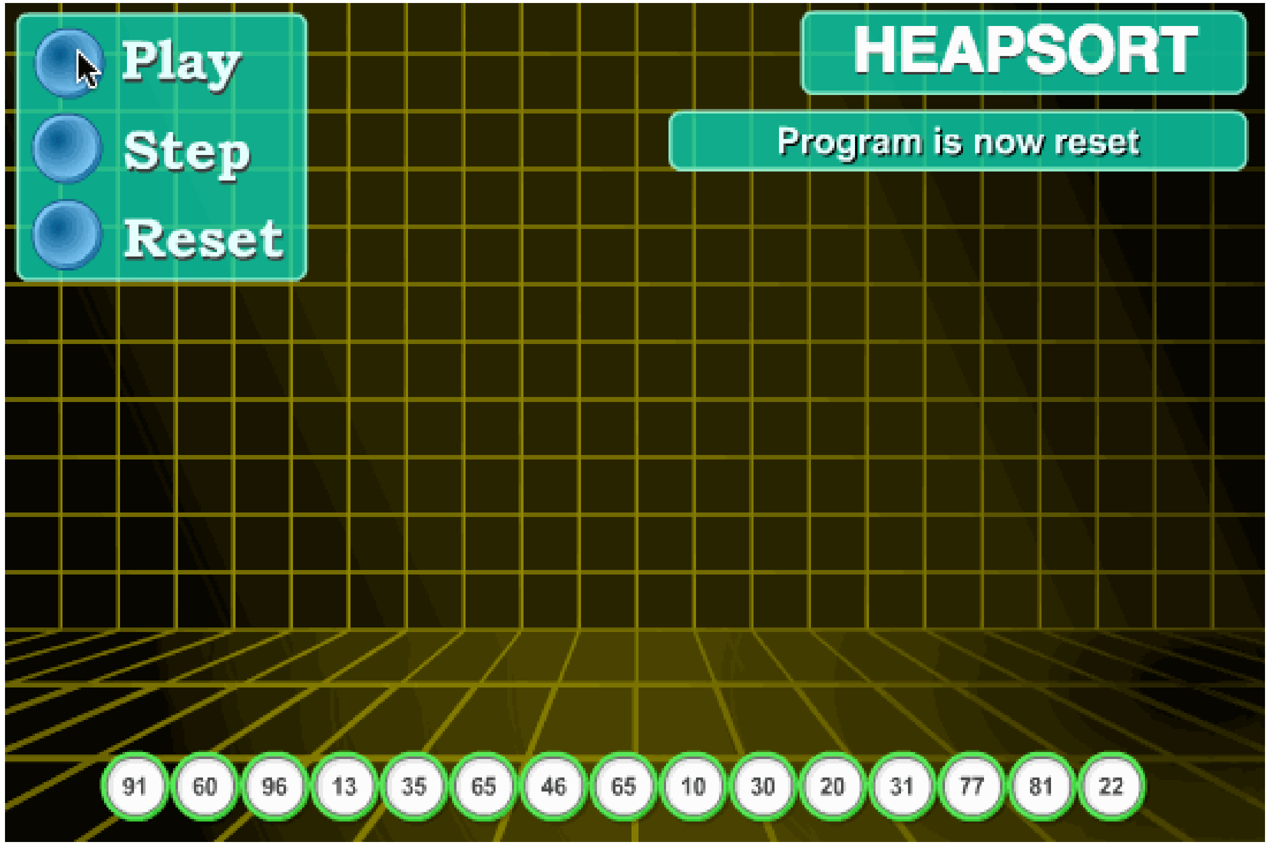
- Bubble sort




```python
def bubbleSort(arr):
    for i in range(1, len(arr)):
        for j in range(0, len(arr)-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

HEAP SORT

- Heap sort



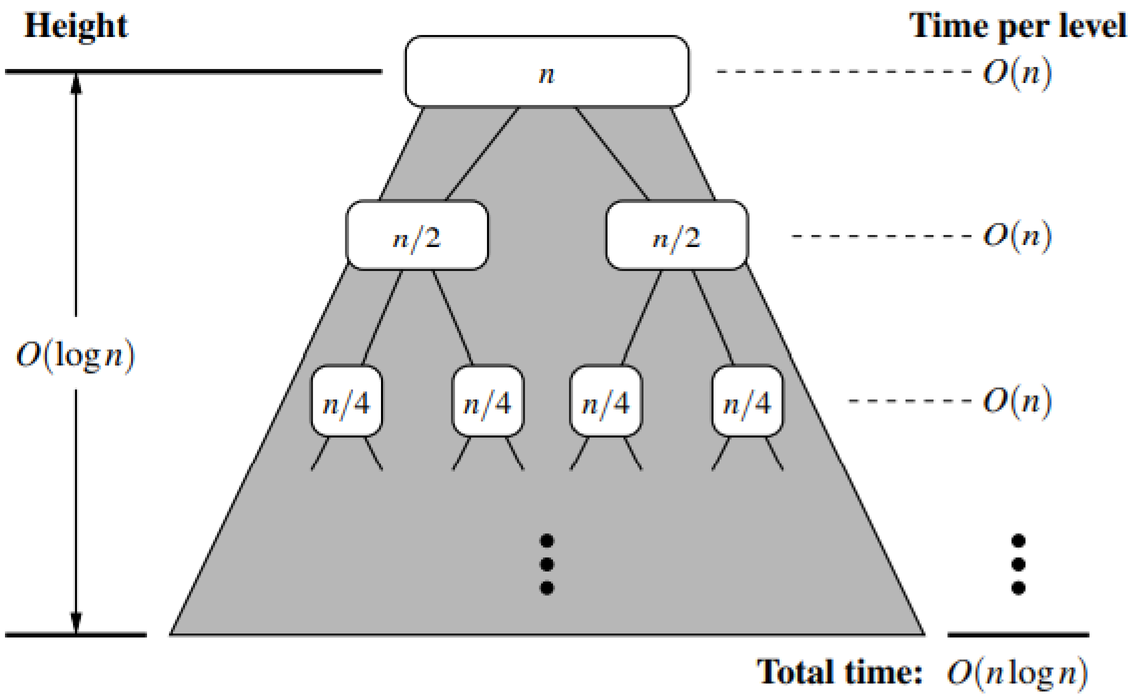
MERGE SORT

- Divide and conquer for sorting
  - **Divide**: if S has zero or one element, return S. Otherwise, remove all the elements from S and put them into two sequences, S1 and S2, each containing half of the elements of S
  - **Conquer**: recursively sort sequence S1 and S2
  - **Combine**: put back the elements into S by merging the sorted sequences S1 and S2 into a sorted sequence

MERGE SORT

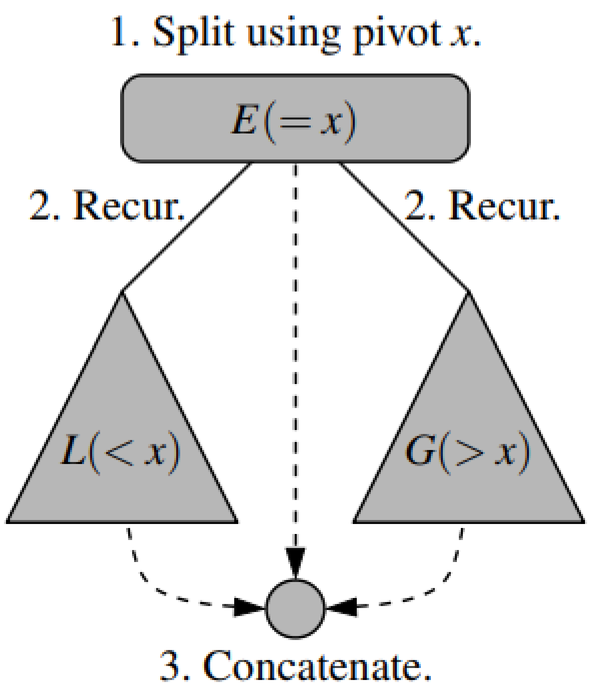
- merge_sort(A[1..n])

1. If n = 1, done
2. recursively sort A[1..n//2], A[n//2+1..n]
3. Merge the two sorted list



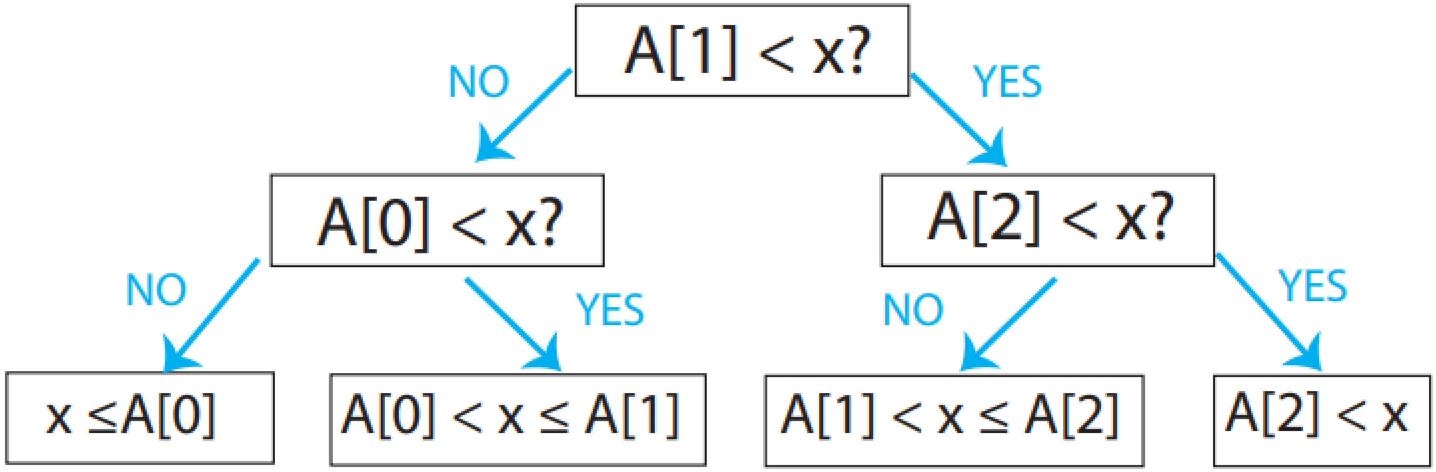
QUICK SORT

- **Divide**: if S has at least two elements, select a specific element x from S, which is called the pivot(枢纽). <u>The common practice is to choose the last element of S</u>. Remove all the elements from S and put them into three sequences:
  - L, storing elements in S less than x
  - E, storing elements in S equal to x
  - G, storing elements in S greater than x
- **Conquer**: recursively sort sequences L and G
- **Combine**: Put back the elements into S in order by first inserting the elements of L, then E, then G



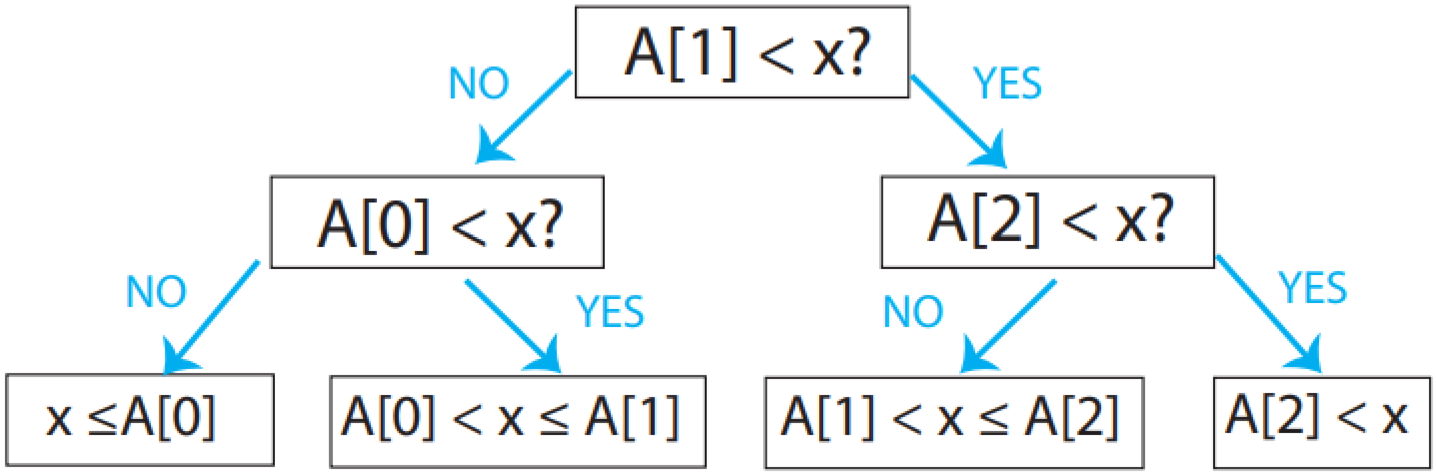
SEARCH LOWER BOUND

- N preprocessed items
- Finding a given item among them in comparison model requires Ω(log n) in worst case
  - Why?
- Decision tree is binary, and therefore must have at least n leaves for each answer
  - Height >= log n



SORTING LOWER BOUND

- N preprocessed items
- Finding a given item among them in comparison model requires Ω(n log n) in worst case
  - Why?
- Decision tree is binary
- # leaves >= # possible answers
- Possible answers?
  - n!

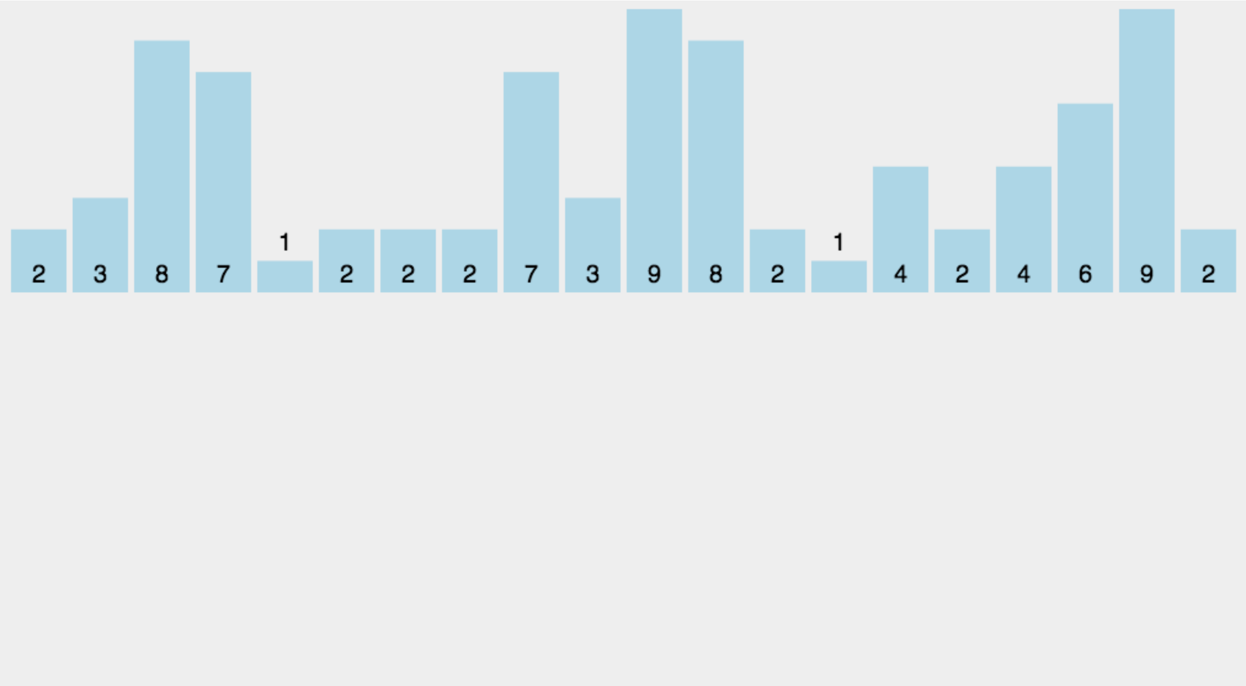


LINEAR TIME SORTING

- Counting sort

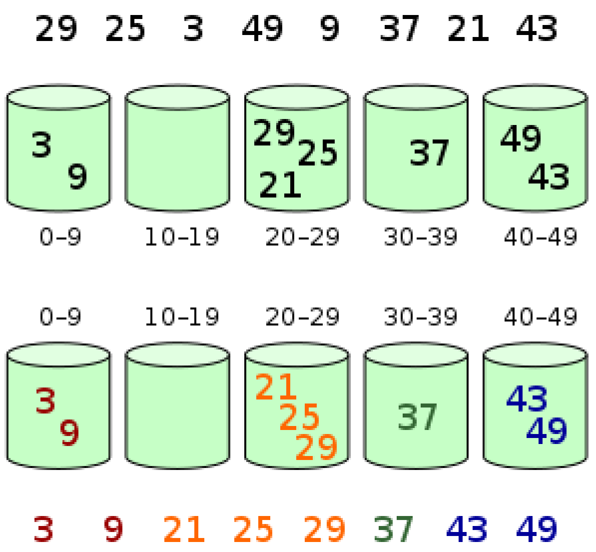
```
L = array of k empty lists
For j in range n:
        L[key(A[j])].append(A[j])
Output = []
For i in range k:
        Output.extend(L[i])
```

  - Time: O(n + k)
  - Count key occurrences using RAM output <count> copies of each key in order
  - But item is more than just a key



LINEAR TIME SORTING

- Bucket sort
  - Sequence S of n entries
  - Keys are integers in [0, N-1], N>=2
  - It is possible to sort S in O(n+N)
  - O(n) if N is O(n)



Algorithm bucketSort(S):

 Input: Sequence S of entries with integer keys in the range $[0, N-1]$

 Output: Sequence S sorted in nondecreasing order of the keys

 let B be an array of N sequences, each of which is initially empty

 for each entry e in S **do**

  k = the key of e

  remove e from S and insert it at the end of bucket (sequence) B[k]

 for i = 0 to N−1 **do**

  for each entry e in sequence B[i] **do**

   remove e from B[i] and insert it at the end of S

LINEAR TIME SORTING

- Radix sort
  - Imagine each integer as base b
  - # digits = d = $\log_b k + 1$
  - Sort integers by least significant digit
  - …
  - Sort by most significant digit
- Sort using counting sort
  - Each iteration: $O(n + b)$
  - Total: $O((n+b)*d)$
  - $= O((n+b)*\log_b k)$
  - When $b = \Theta(n)$
  - $O(n*\log_n k)$
  - If $k = n^c$ then $O(nc)$

SORTING ALGORITHMS

- $O(n^2)$ algorithms
  - Selection sort, insertion sort
- $O(n \log n)$ algorithms
  - Heap sort, merge sort, quick sort
- Linear time algorithms
  - Counting sort, bucket sort, radix sort
- Poorest choice of algorithms?
  - Selection sort
  - Best case $O(n^2)$

SORTING ALGORITHMS

- Insertion sort
  - O(n + m), where m is number of inversions
  - Good when n is small
  - Good when m is small
  - $O(n^2)$ when m is large and n is large
- Heap sort
  - O(n log n) time worst case
  - In-place easy to implement
  - Slower than quick sort and merge sort on larger sequences

SORTING ALGORITHMS

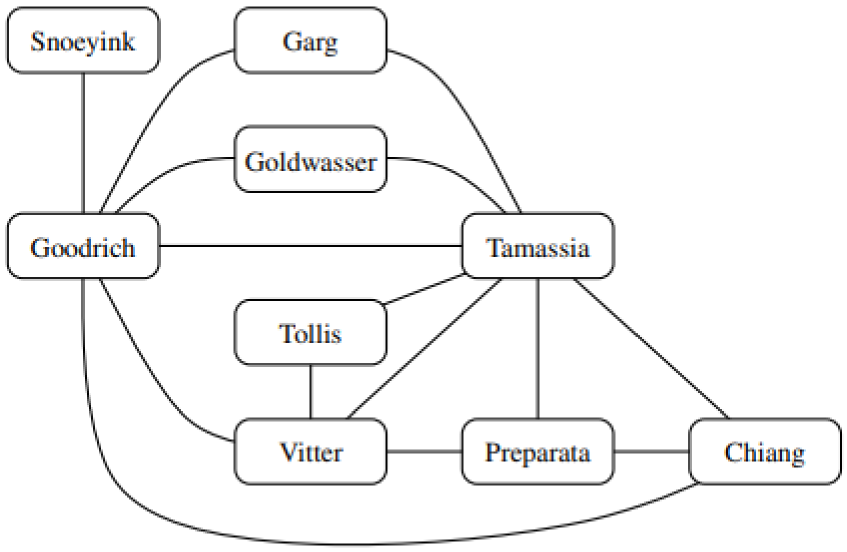
- Quick sort
  - Worst case $O(n^2)$
  - O (n log n) expected
  - Faster than heap sort and merge sort on may tests
  - Can be implemented in-place
- Merge sort
  - O(n log n) worst case
  - In-place implementation: difficult
  - Hence slower than quick sort
- Bucket sort and radix sort
  - Limitations: small integer keys, character strings, d-tuples from a discrete range
  - O(d(n+N) time, in range [0, N-1]

THIS LECTURE

- Graphs: relationships between pairs of objects: vertices together with a collection of connections between them (edges).
  - Mapping, transportation, computer networks electrical engineering
- Definition:
  - Set V of **vertices** and a collection E of pairs of vertices from V, called **edges**
- Edges
  - Directed: an edge (u, v) is directed from u to v if the pair (u, v) is ordered
  - Undirected: an edge (u, v) is not ordered, sometimes denoted as {u, v}
- Undireged graph: the edges are all undirected
- Directed graph (digraph): edges are all directed
- Mixed graph: directed edges + undirected edges
- Undirected graph => directed graph

GRAPHS

- Example
  - Collaboration graph (undirected)
  - Class relationships within an Object-Oriented program (directed)
  - City map (mixed graph)
  - Electric wiring/Schematics
    - Directed/undirected

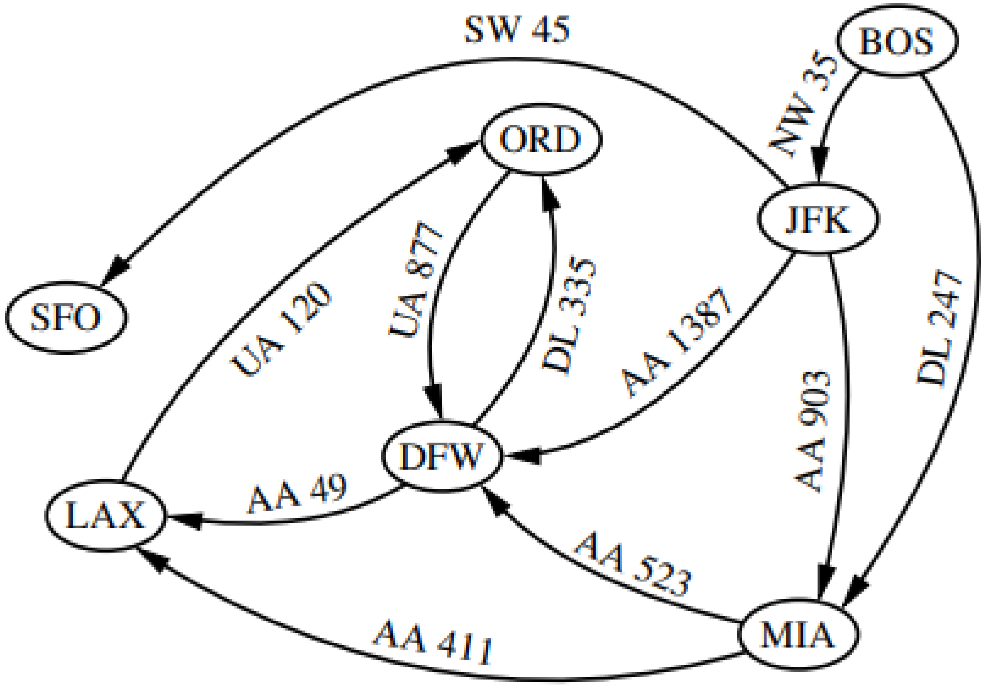


GRAPHS

- End vertices (endpoints)
  - Two vertices joined by an edge
  - Directed edge: origin -> destination
- Adjacent vertices: u and v are adjacent if there is an edge whose end vertices are u and v
- Incident: an edge is incident to a vertex if the vertex is one of the edge’s endpoints
- Outgoing edges of a vertex: directed edges whose origin is the vertex
- Incoming edges of a vertex: directed edges whose destination is the vertex
- Degree of a vertex: # of incident edges of v
- In-degree: # of incoming edges of v
- Out-degree: # of outgoiong edges of v

GRAPHS

- End vertices (endpoints)
  - Two vertices joined by an edge
  - Directed edge: origin -> destination
- Adjacent vertices: u and v are adjacent if there is an edge whose end vertices are u and v
- Incident: an edge is incident to a vertex if the vertex is one of the edge's endpoints
- Outgoing edges of a vertex: directed edges whose origin is the vertex
- Incoming edges of a vertex: directed edges whose destination is the vertex
- Degree of a vertex: # of incident edges of v
- In-degree: # of incoming edges of v
- Out-degree: # of outgoiong edges of v

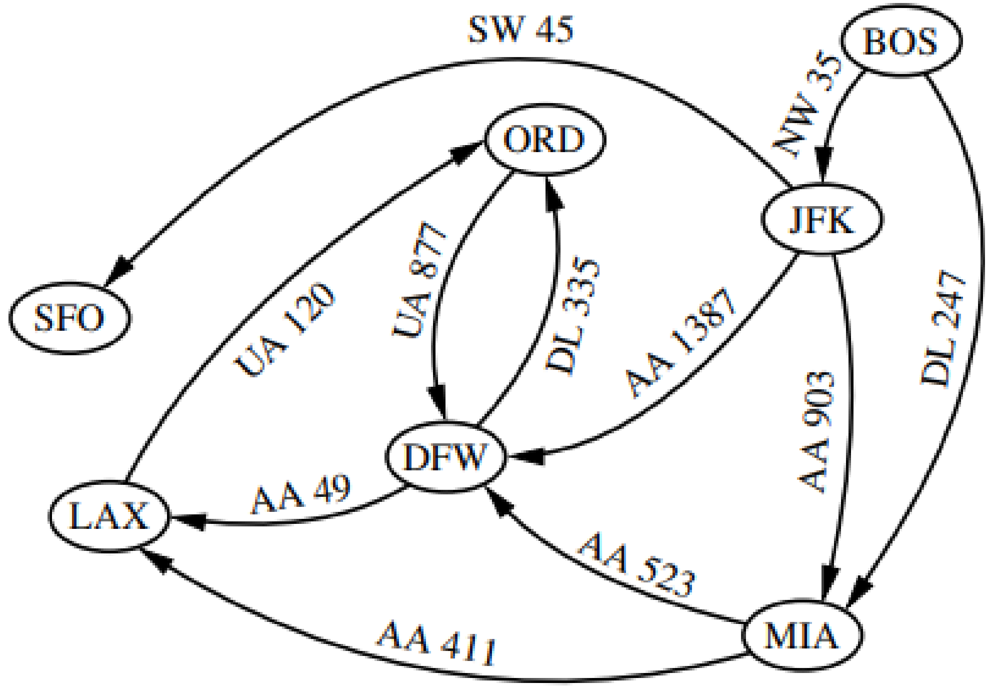


GRAPHS

- Definition:
  - Set V of **vertices** and a collection E of pairs of vertices from V, called **edges**
- E: collection, not a set
  - Two edges can have the same end vertices
  - Such edges are called **parallel** or **multiple** edges
  - **Self-loop** edges: endpoints are the same
- **Simple** graphs: graphs do not have parallel edges or self loops
- **Path**: sequence of vertices and edges, start at a vertex and ends at a vertex
  - Simple: if each vertex in the path is distinct
- **Cycle**: path that starts and ends at the same vertex, and includes at least one edge
  - Simple: if each vertex in the cycle is distinct(except for the first and last one)
- **Acyclic**: directed graph that has no directed cycles

GRAPHS

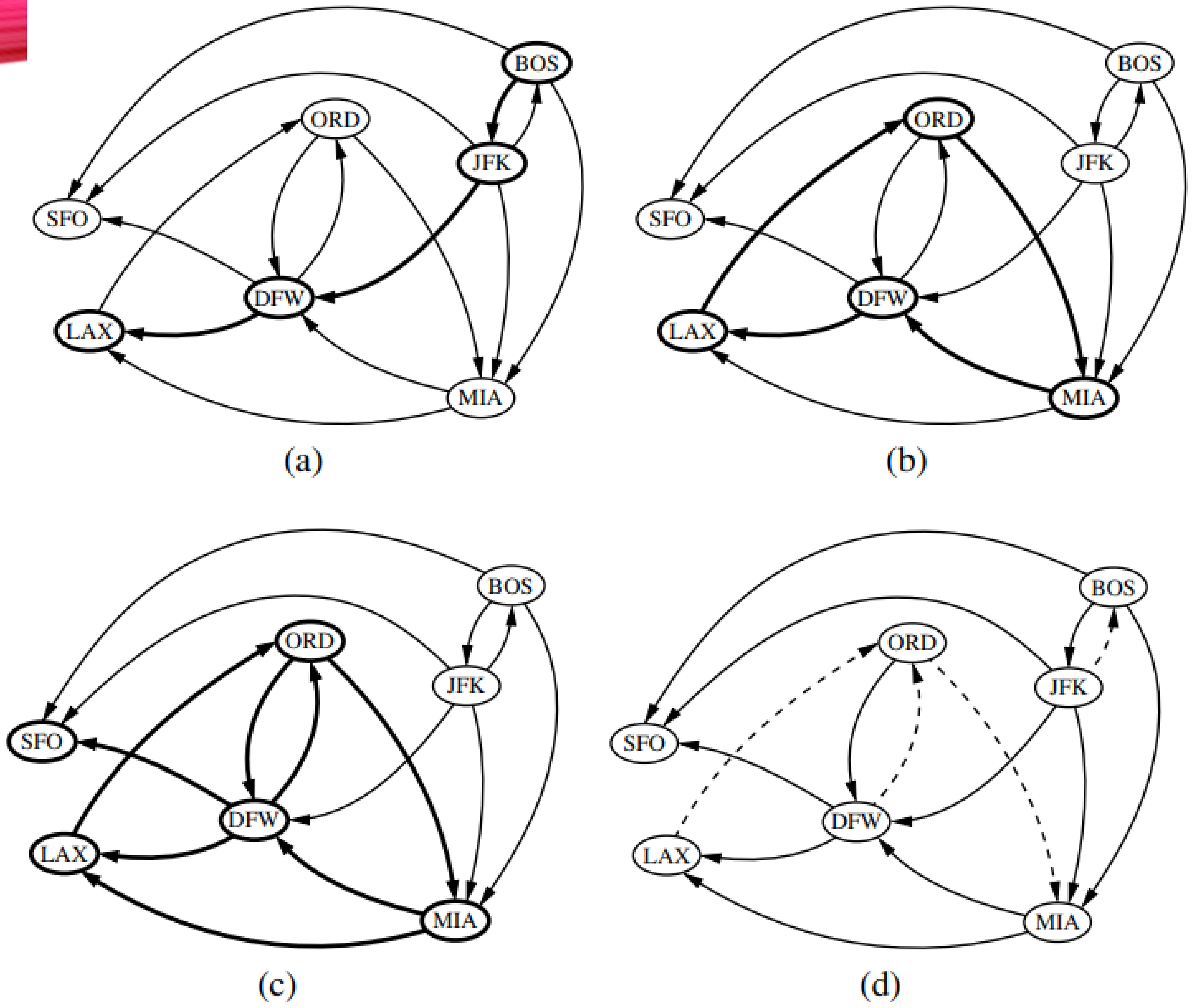
- Directed path
  - A path such that all edges are directed and are traversed along their direction
  - (BOS, NW35, JFK, AA1387, DFW)
- Directed cycle
  - A cycle such that all edges are directed and are traversed along their direction
  - (LAX, UA1200, ORD, UA877, DFW, AA49, LAX)



GRAPHS

- Reachability
  - In a graph G, u **reaches** v, and v is reachable from u, if G has a path from u to v
- Connectivity
  - G is **connected**, if for any two vertices, there is a path between them
  - Directed graph: **strongly connected**
- Subgraph
  - Graph H whose vertices and edges are subsets of the vertices and edges of G
- Spanning subgraph
  - Subgraph of G that contains all the vertices of the graph
- Forest
  - Graph without cycles
- Tree
  - Connected forest
- Spanning tree
  - Spanning subgraph that is a tree

GRAPHS



(a) (b) (c) (d)

GRAPHS

- If G is a graph with m edges and vertex set V, then
- If G is a directed graph with m edges and vertex set V, then
- Let G be a simple graph with n vertices and m edges.
  - If G is undirected, then m <= n(n-1)/2
  - If G is directed, then m <= n(n-1)
- Let G be a undirected graph with n vertices and m edges
  - If G is connected, then m >= n-1
  - If G is a tree, then m = n-1
  - If G is a forest then m <= n-1

$$\sum_{v \text{ in } V} deg(v) = 2m.$$

$$\sum_{v \text{ in } V} indeg(v) = \sum_{v \text{ in } V} outdeg(v) = m.$$

GRAPHS

- Graph ADT
- Endpoints: return a tuple (u, v) such that vertex u is the origin and v the destination
- Opposite(v): if v is one endpoint of the edge, return the other endpoint
- Vertex_count(): returns # of vertices
- Vertices(): returns an iteration of all vertices of the graph
- Edge_count(): returns # of edges
- Edges(): returns an iteration of all edges
- Get_edge(u, v): returns the edge from u to v, if one exists, otherwise return None

GRAPHS

- Graph ADT
- degree(v, out= True): returns the # of edges incident to v
- incident_edges(v, out=True): returns an iteration of edges incident to v
- insert_vertex(x = None): create and return a new Vertex storing element x
- insert_edge(u, v, x = None): create and return a new Vertex from u to v
- remove_vertex(v): remove v and all its incident edges from the graph
- remove_edge(e): remove e from the graph

DATA STRUCTURES FOR GRAPHS

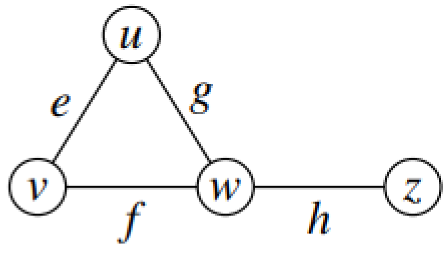
- Edge list
  - An unordered list of all edges, no efficient way to locate a particular edge (u, v), or the set of all edges incident to a vertex v
- Adjacency list
  - For each vertex, a list containing edges that are incident to the vertex
  - Complete set of edges can be determined by taking the union of the smaller sets
- Adjacency map
  - Similar to adjacency list, but secondary container for edges are maps
  - O(1) expected time to access a specific edge(u,v)
- Adjacency matrix
  - Worst case O(1) access to a specific edge (u, v) by maintaining a n*n matrix for each graph with n vertices

DATA STRUCTURES FOR GRAPHS

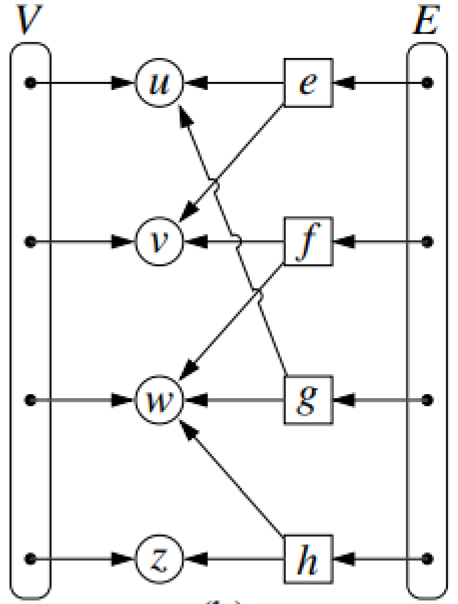
Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

EDGE LIST

- Collections V and E are represented with doubly linked lists (PositionalList)
- Vertex objects
  - Reference to element x
  - Reference to position of the vertex instance in the list V
- Edge objects
  - Reference to element x
  - Reference to the vertex objects associated with the endpoint vertices of e
  - Reference to the position of the edge instance in list E



(a)



(b)

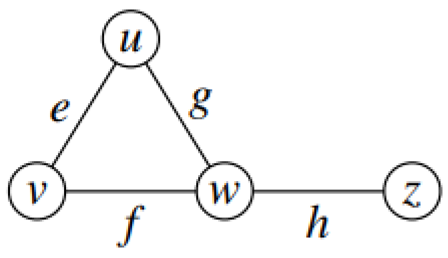
EDGE LIST

- Performance
- Space: $O(n + m)$
  - n vertices and m edges
- Running time
  - vertices(): $O(n)$
  - edges(): $O(m)$
  - get_edge(): $O(m)$
    - Most significant limitation
  - remove_vertex(v): $O(m)$
    - Why?

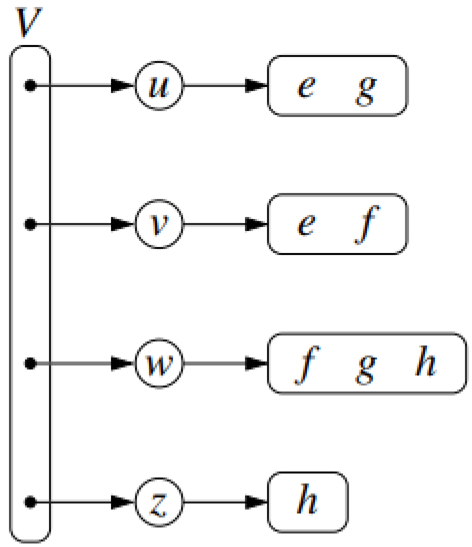
Operation	**Running Time**
vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

ADJACENCY LIST

- Secondary containers for edges that are associated with each individual vertex
- For each v, maintain a collection I(v) called **incidence collection** of v
- Primary structure: collection V of vertices
  - Positional list
- Each vertex instance
  - Direct reference to its I(v) incidence collection



(a)



(b)

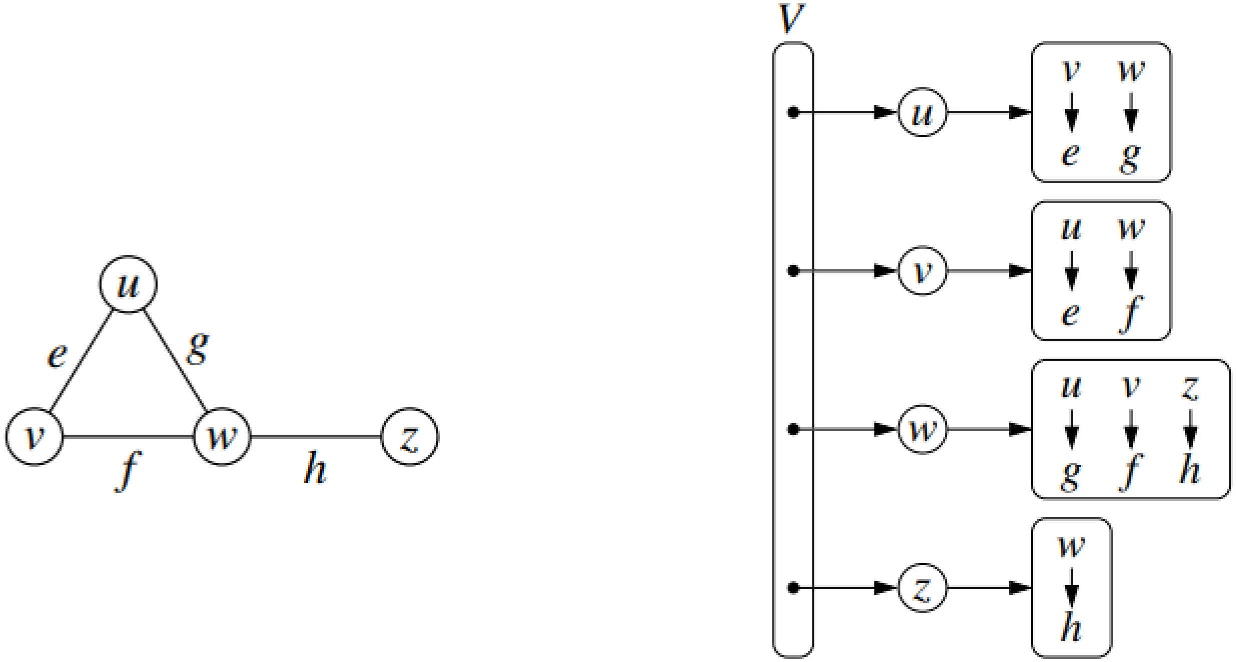
ADJACENCY LIST

- Performance
- Space: O(n + m)
  - n vertices and m edges
- Running time
  - vertices(): O(n)
  - edges(): O(m)
  - get_edge(): O(min(deg(u), deg(v)))
    - Search through either I(u) or I(v)
  - remove_vertex(v): O(dev(v))

Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

ADJACENCY MAP

- Adjacency list
  - I(v) uses O(deg(v)) space
  - O(dev(v)) time
- Performance improvement
  - Hash-based map for I(v)
  - get_edge(u,v) can run in expected O(1) time, worst case O(min(deg(u), deg(v))



(a) (b)

ADJACENCY MATRIX

- Matrix A (n by n) to locate an edge between a given pair of vertices in worst-case O(1) time
- Verticeis as integers in {0, 1, …, n-1}
- A[I, j] holds a reference to the edge (u, v) if one exists
- Edge (u, v) can be accessed in worst-case O(1) time
- $O(n^2)$ space usage
- Matrix can be used to store only Boolean values, if edges do not store any additional data

(a) (b)

F

```
#------------------------ nested Vertex class --------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's i
    self._element = x

  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):            # will allow vertex tc
    return hash(id(self))
```

```
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x).
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):            # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

PYTHON IMPLEMENTATION

```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for un
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected

    Property is based on the original declaration of the graph, no
    """
    return self._incoming is not self._outgoing # directed if ma

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)
```

```
  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count ed
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )        # avoid double-reporting edges of u
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))      # add edges
    return result
```

PYTHON IMPLEMENTATION

```
def get_edge(self, u, v):
  """Return the edge from u to v, or None if not adjac
  return self._outgoing[u].get(v)               # returns N

def degree(self, v, outgoing=True):
  """Return number of (outgoing) edges incident to ve

  If graph is directed, optional parameter used to count
  """
  adj = self._outgoing if outgoing else self._incoming
  return len(adj[v])

def incident_edges(self, v, outgoing=True):
  """Return all (outgoing) edges incident to vertex v ir

  If graph is directed, optional parameter used to reque
  """
  adj = self._outgoing if outgoing else self._incoming
  for edge in adj[v].values():
    yield edge
```

```
def insert_vertex(self, x=None):
  """Insert and return a new Vertex with elei
  v = self.Vertex(x)
  self._outgoing[v] = { }
  if self.is_directed():
    self._incoming[v] = { }              # need c
  return v

def insert_edge(self, u, v, x=None):
  """Insert and return a new Edge from u to
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

THANKS

See you in the next session!