MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

31 Sorting Continued, Graphs

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

April 7, 2022

Overview

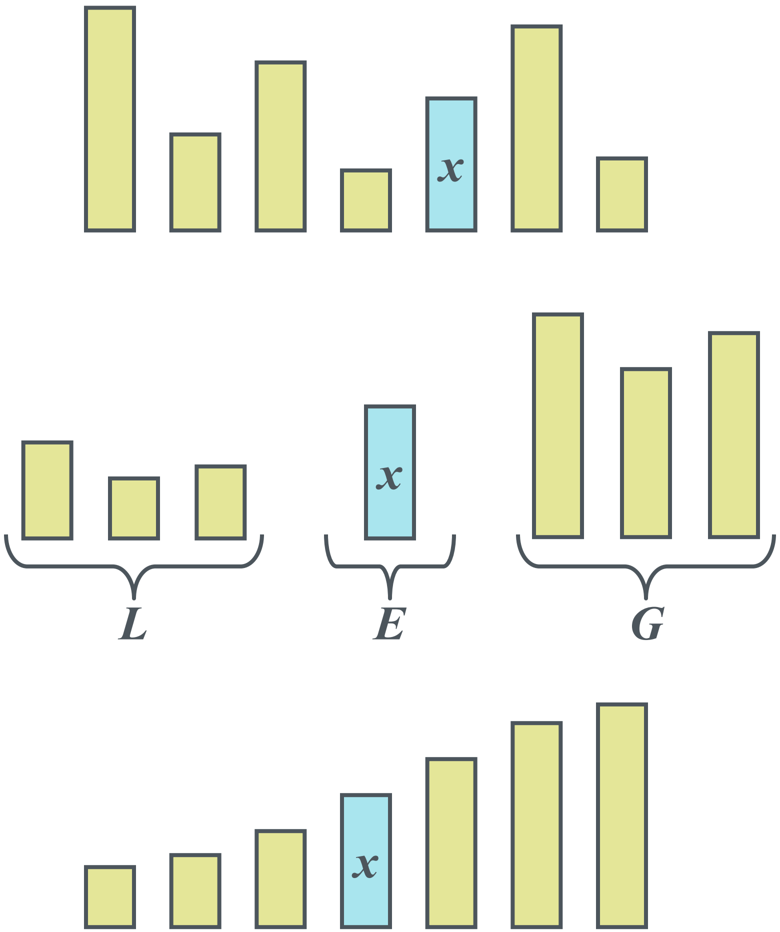
- Sorting: What we have seen so far?

Sorting Algorithm	Time Complexity	Properties
Insertion sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Selection sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Heap sort	O(nlogn)	• fast • in-place • Suitable for large datasets (1K - 1M)
Merge sort	O(nlogn)	• fast • sequential data access • Suitable for for huge datasets (>1M)

- We will talk about Quick sort

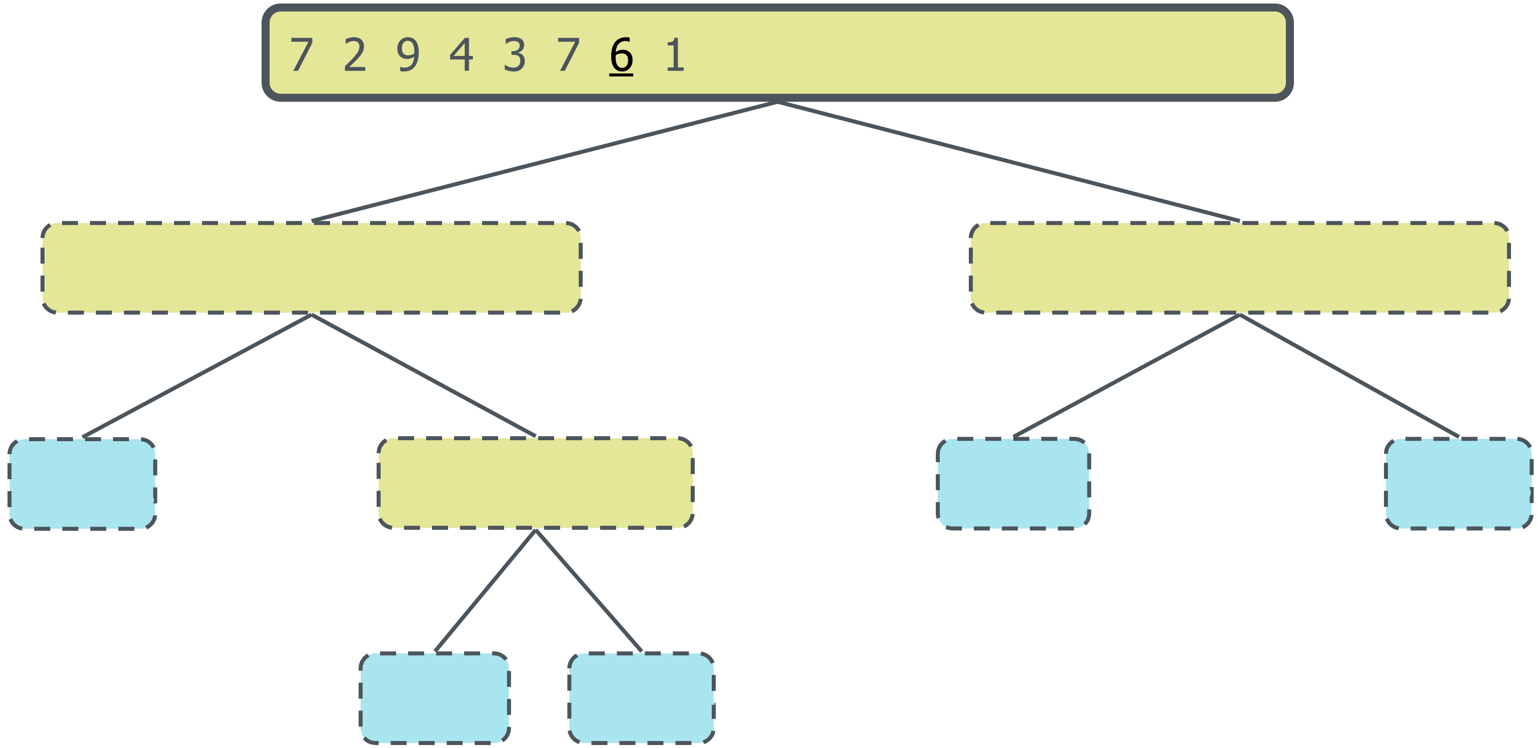
Quicksort

- Quicksort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into:
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



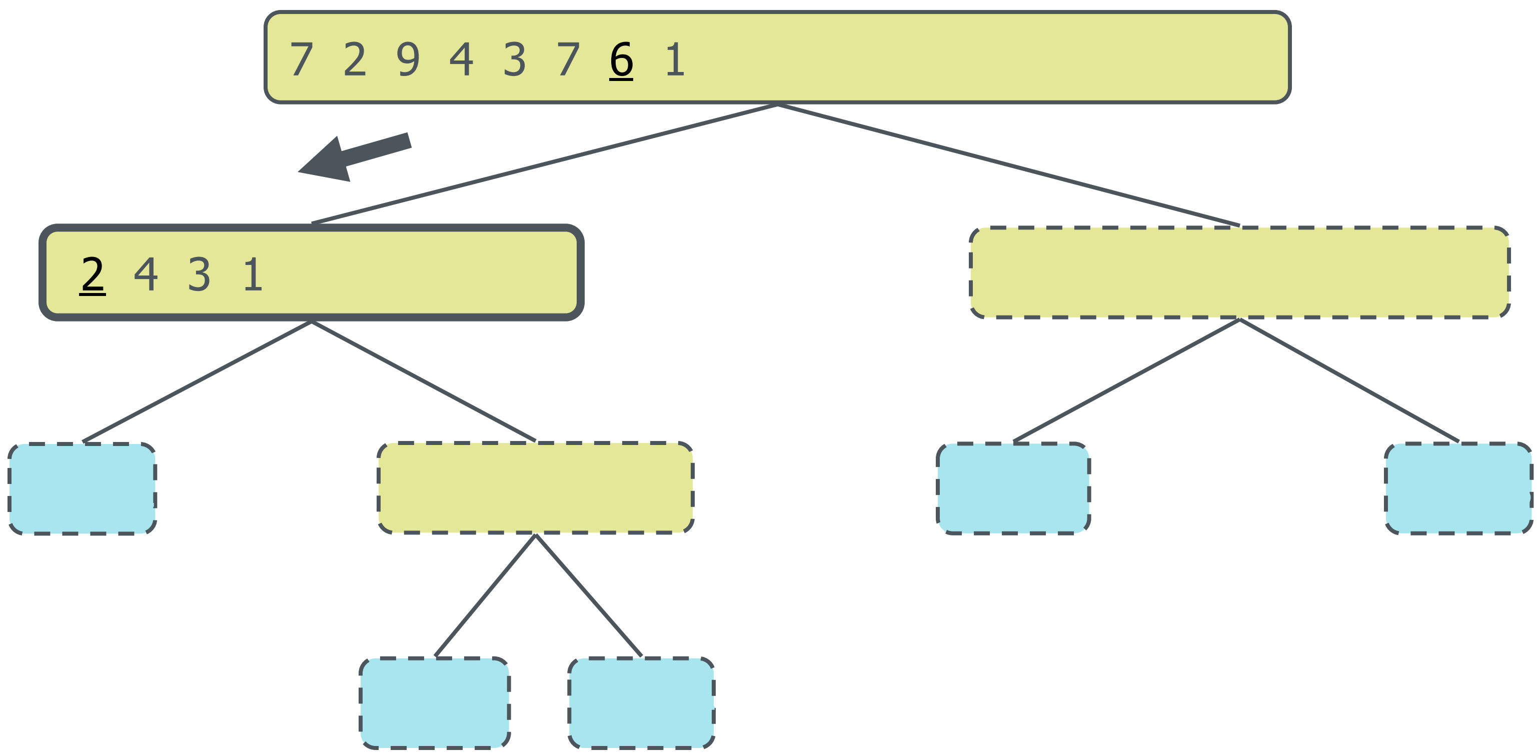
Quicksort - Execution Example

- **Pivot Selection**



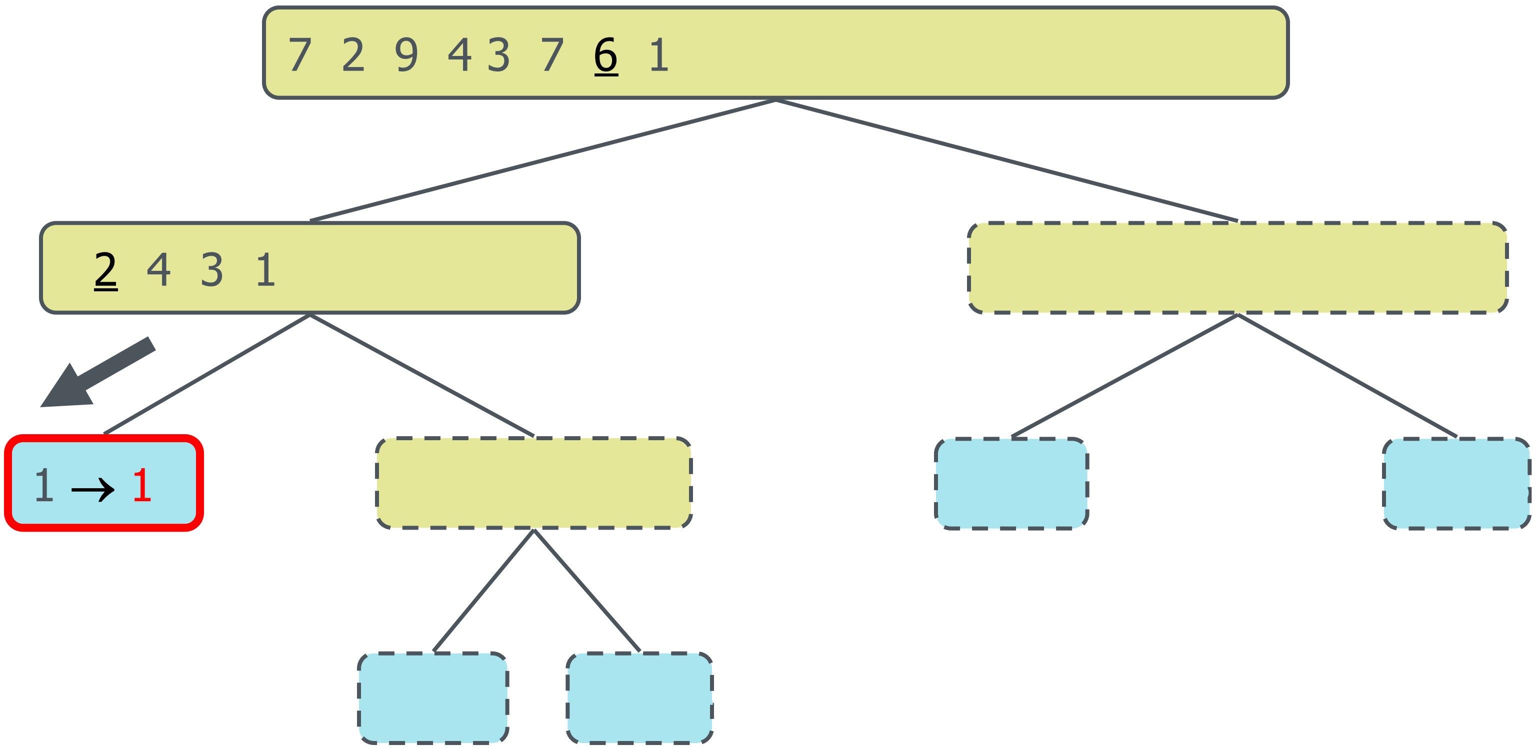
Quicksort - Execution Example

- **Partition, recursive call, pivot selection**



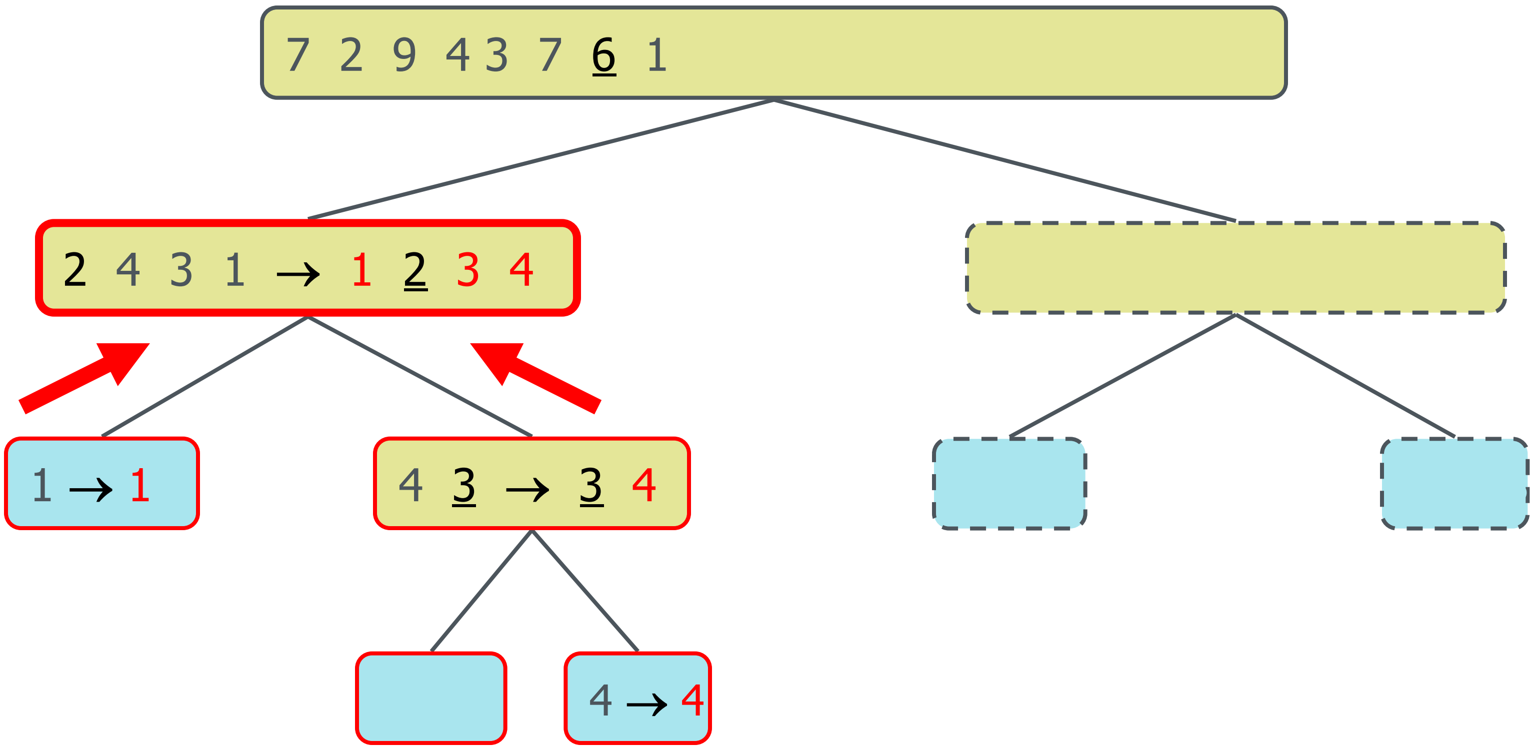
Quicksort - Execution Example

- **Partition, recursive call, base case**



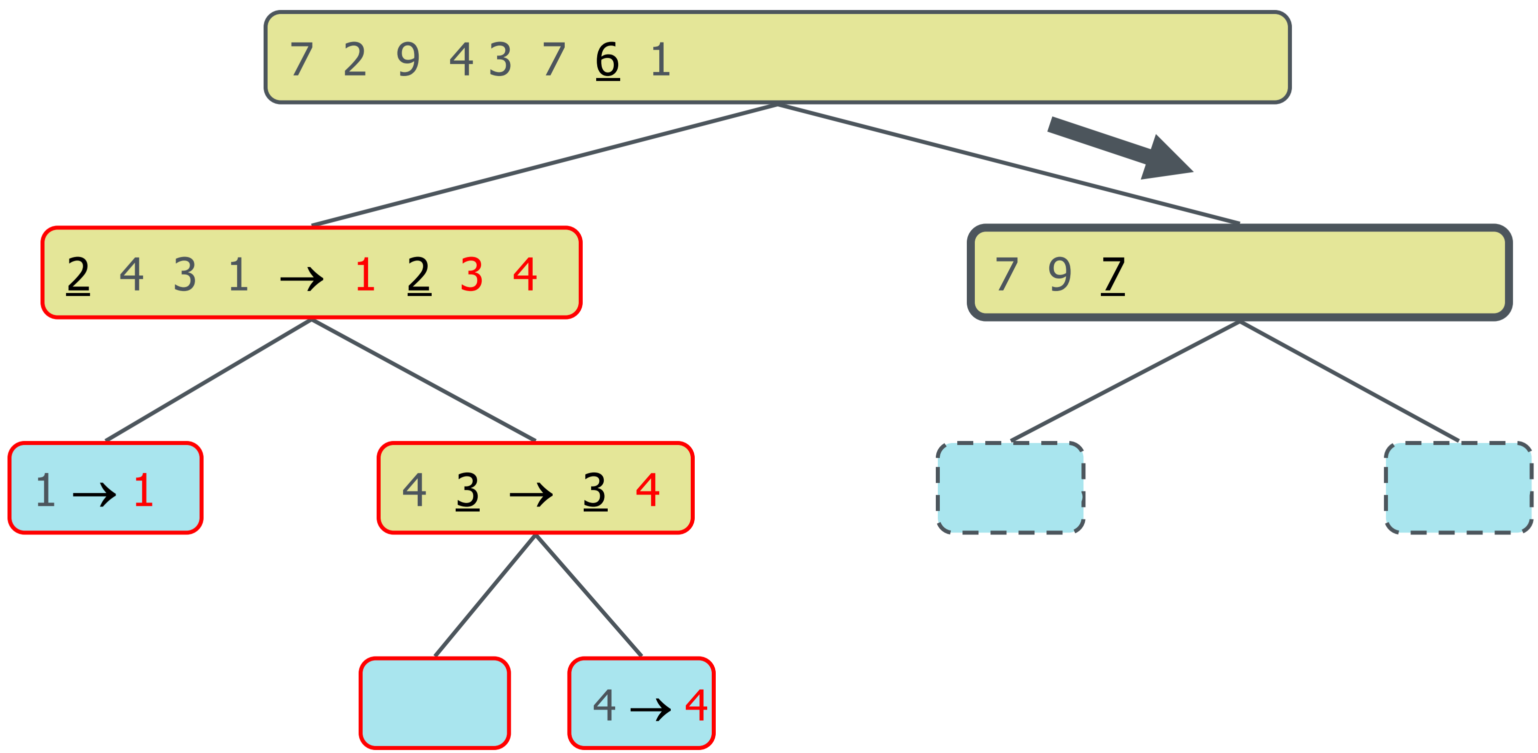
Quicksort - Execution Example

- **Recursive call, …, base case, join**



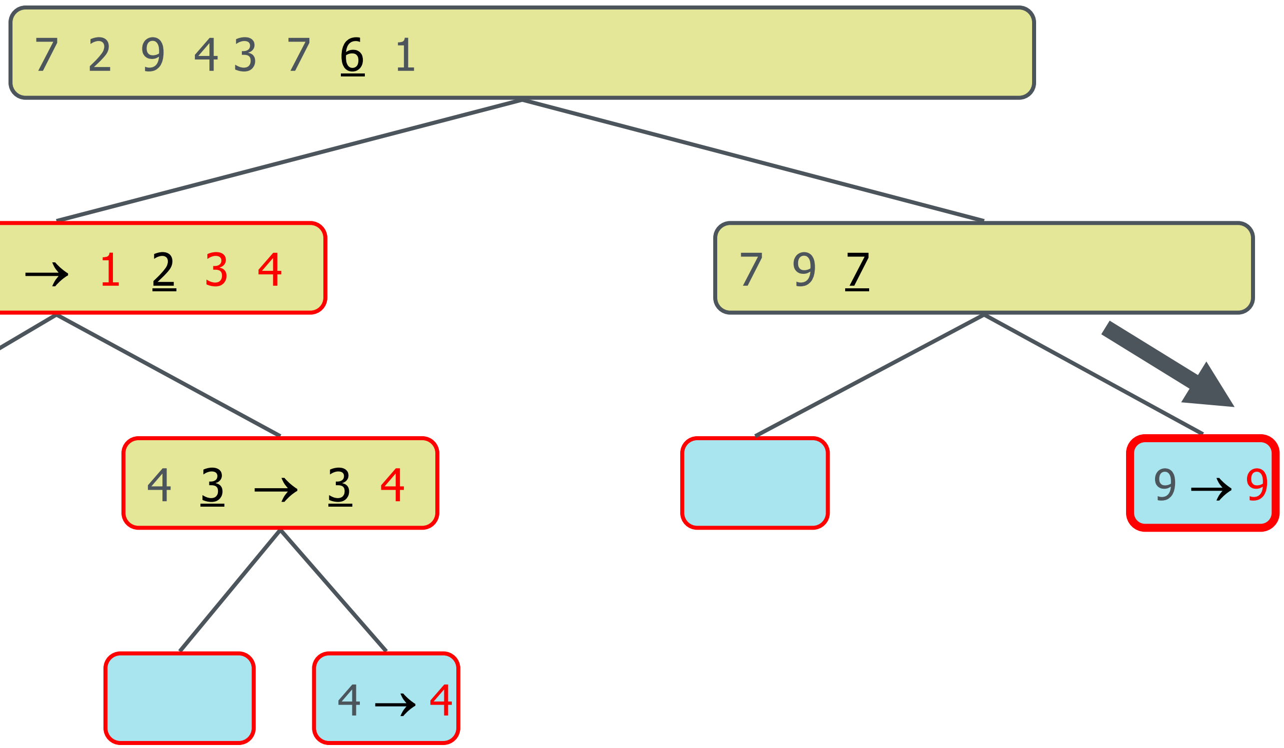
Quicksort - Execution Example

- **Recursive call, pivot selection**



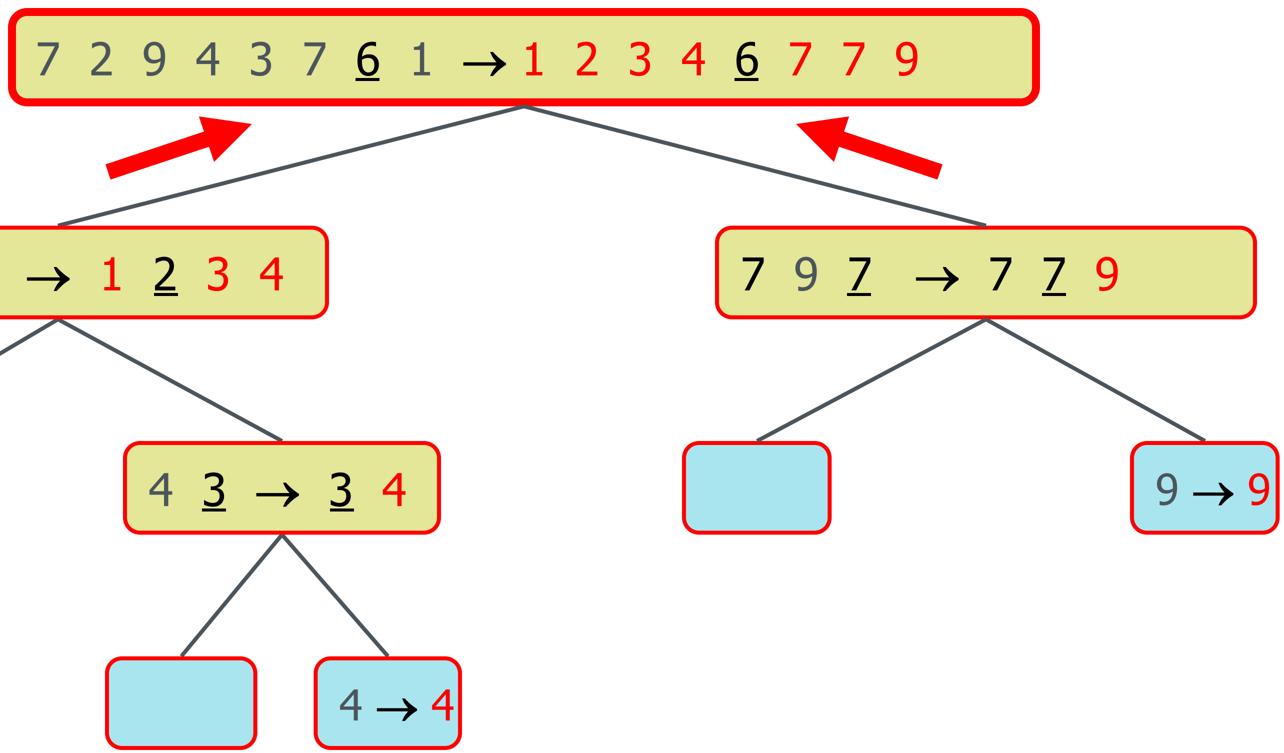
Quicksort - Execution Example

- **Partition, …, recursive call, base case**



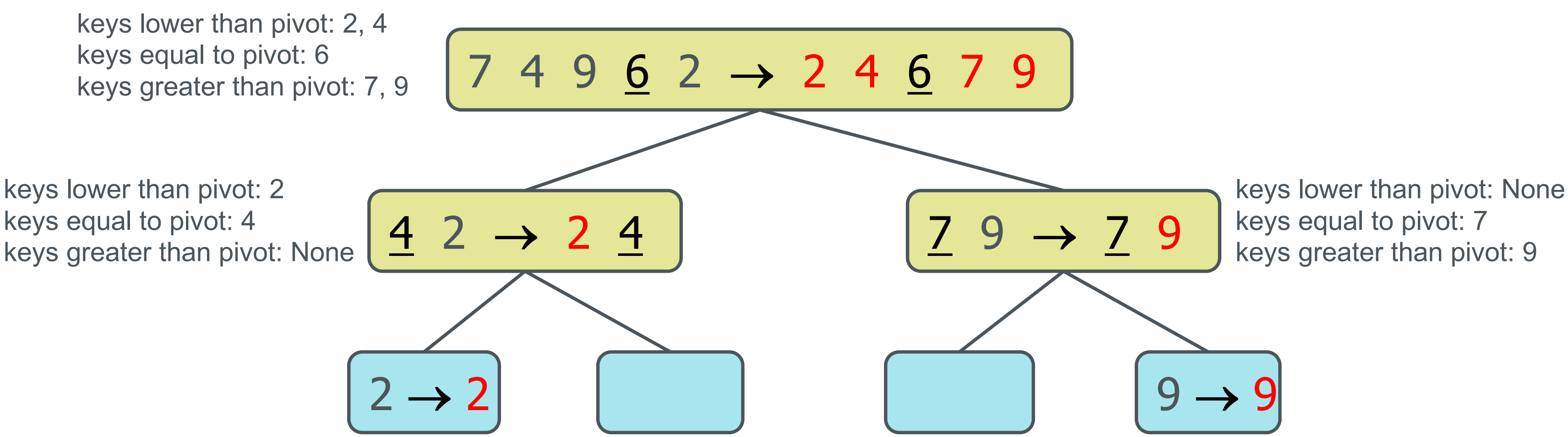
Quicksort - Execution Example

- **join, join**



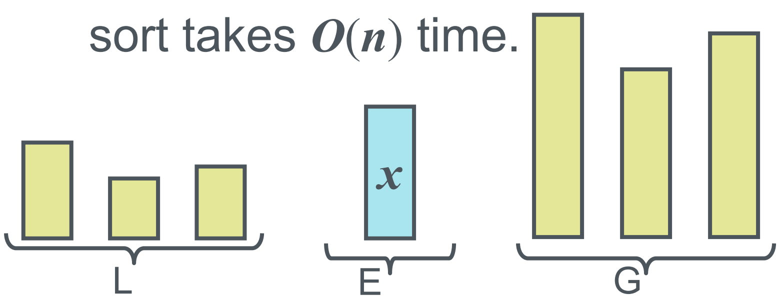
Quicksort Tree

- An execution of quicksort is depicted by a binary tree
  - Each node represents a recursive call of quicksort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



Partition

- We partition an input sequence as follows:
    - We remove, in turn, each element y from S and
    - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time.




```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
      elements of S less than, equal to,
      or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.erase(p)
    while ¬S.empty()
        y ← S.eraseFront()
        if y < x
            L.insertBack(y)
        else if y = x
            E.insertBack(y)
        else  // y > x
            G.insertBack(y)
    return L, E, G
```

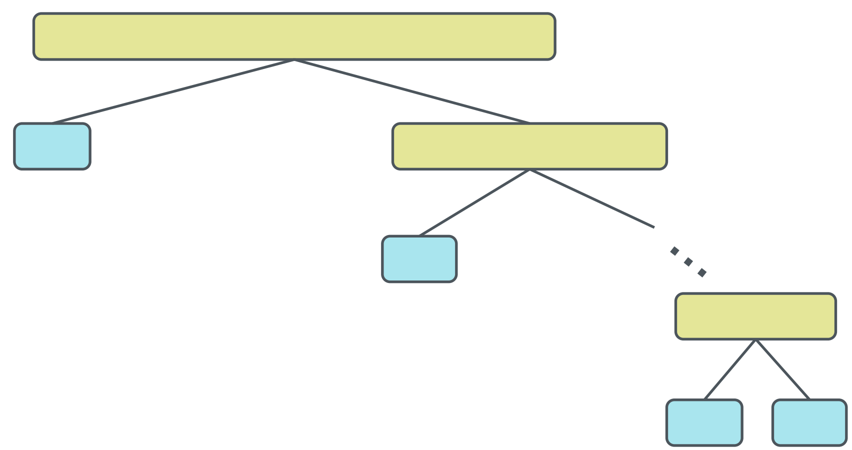
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

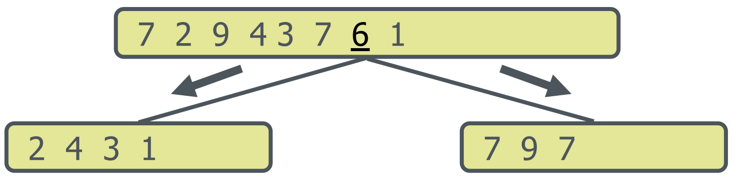
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1

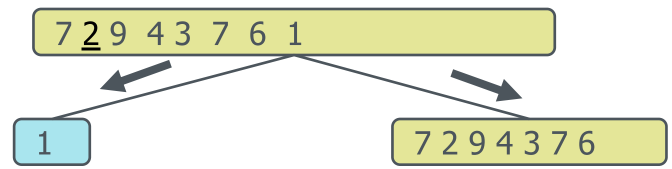


Expected Running Time (This slide is not a course content)

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call**: the sizes of L and G are each less than $3s/4$
  - **Bad call**: one of L and G has size greater than $3s/4$

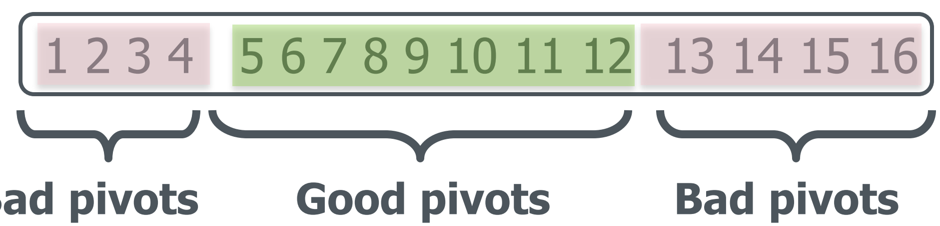


Good call



Bad call

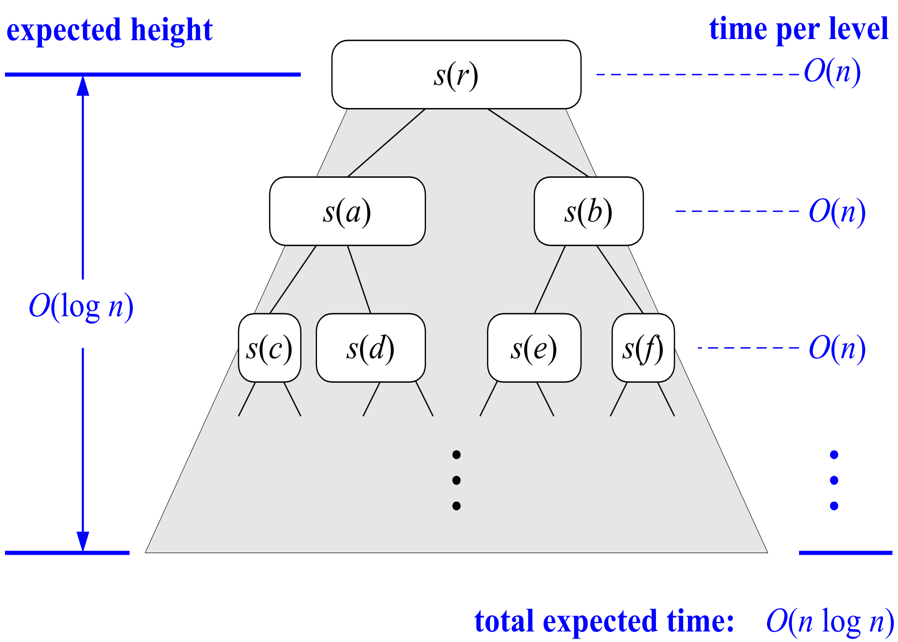
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Expected Running Time (This slide is not a course content)

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$

- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



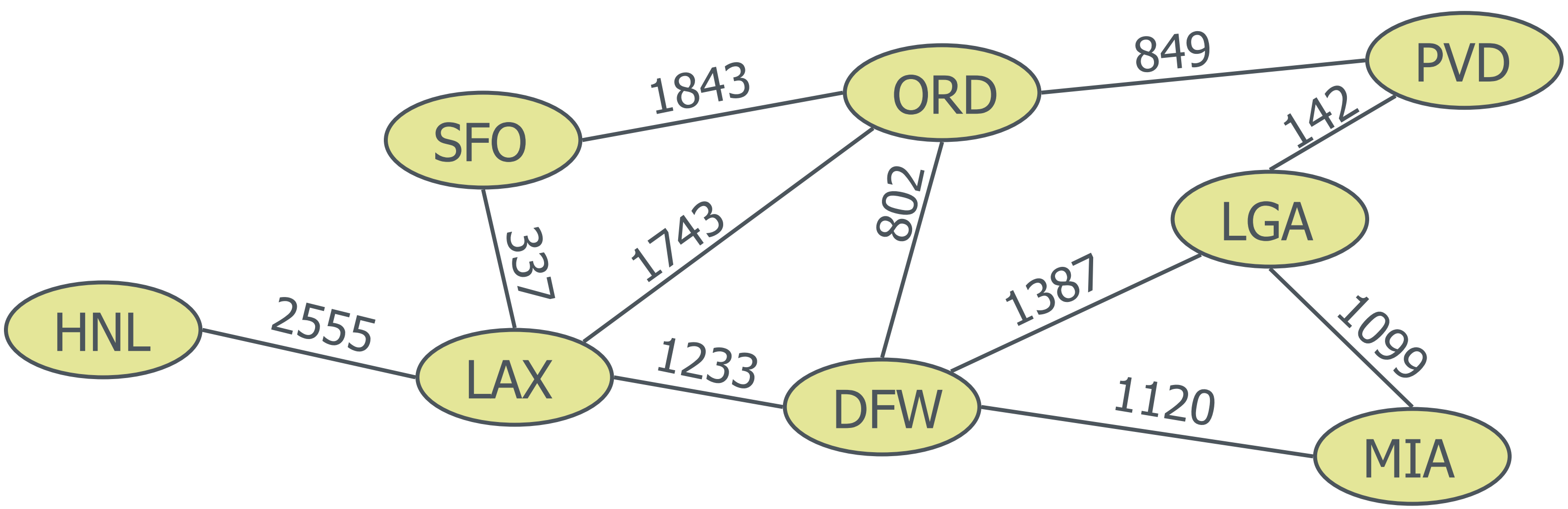
Recall

Sorting Algorithm	Time Complexity	Properties
Insertion sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Selection sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Heap sort	O(nlogn)	• fast • in-place • Suitable for large datasets (1K - 1M)
Merge sort	O(nlogn)	• fast • sequential data access • Suitable for for huge datasets (>1M)
Quicksort	O(nlogn) expected	• in-place, randomized • fastest (good for large inputs)

Graphs

Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

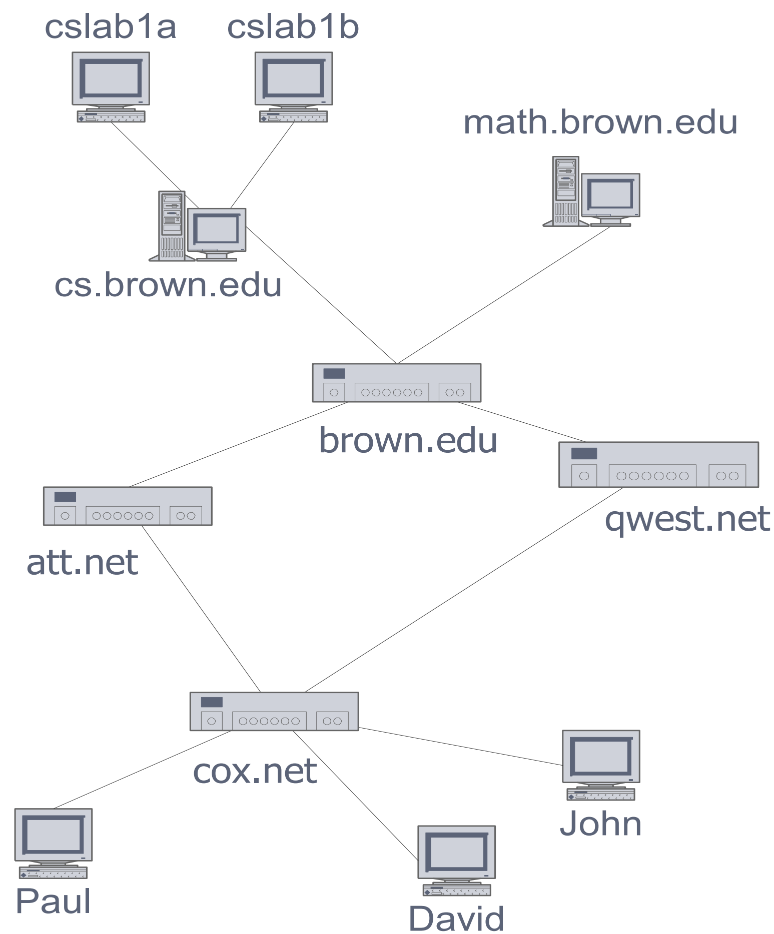


Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

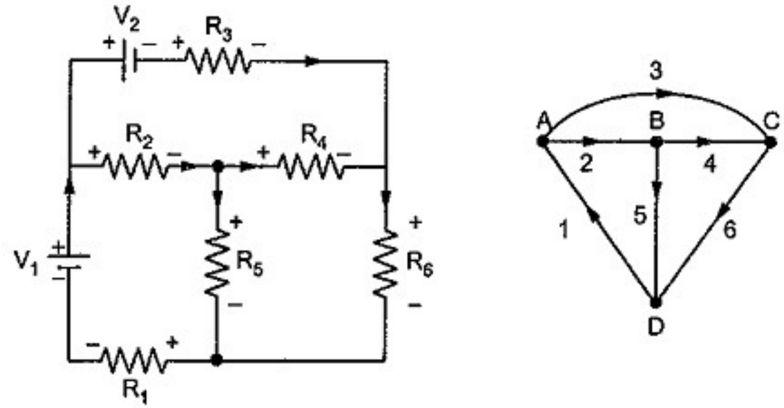
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
  - Social Networks



Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
  - Social Networks



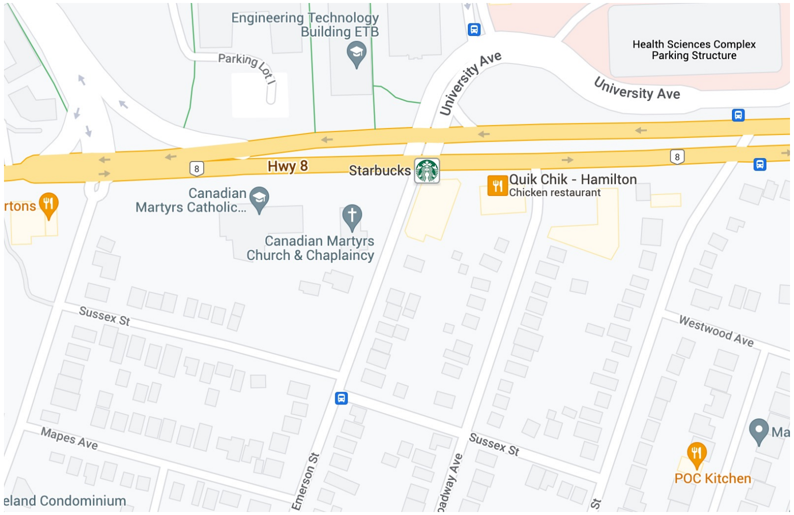
(a) Network

(b) Oriented graph

Fig. 5.19

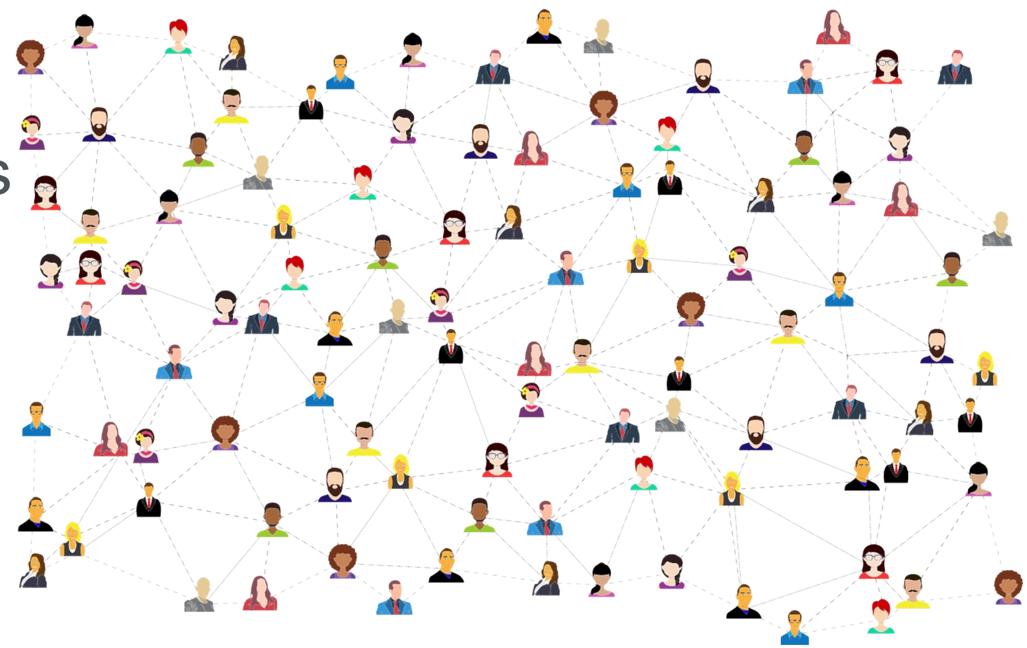
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
  - Social Networks



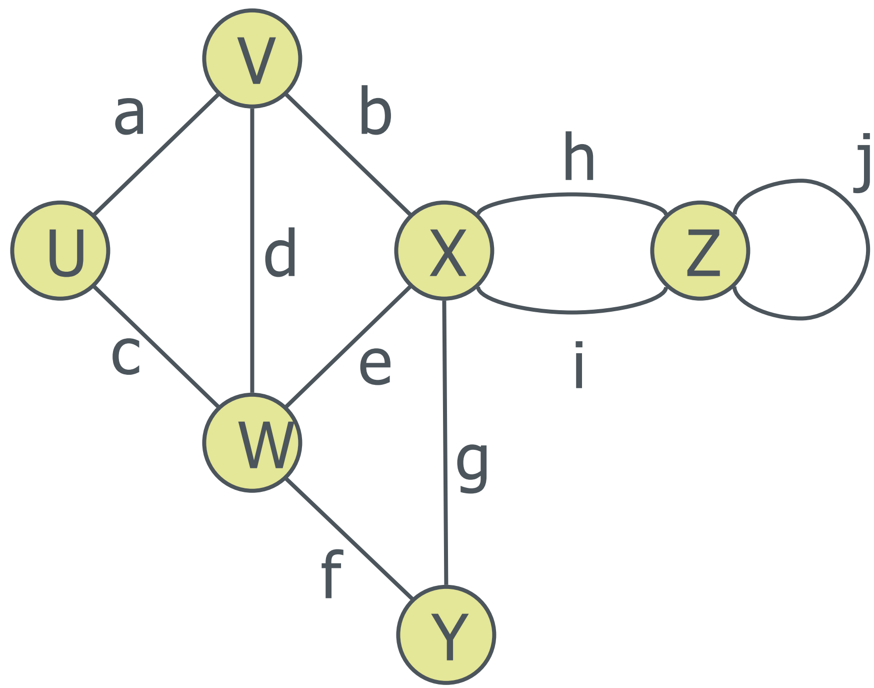
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
  - Social Networks



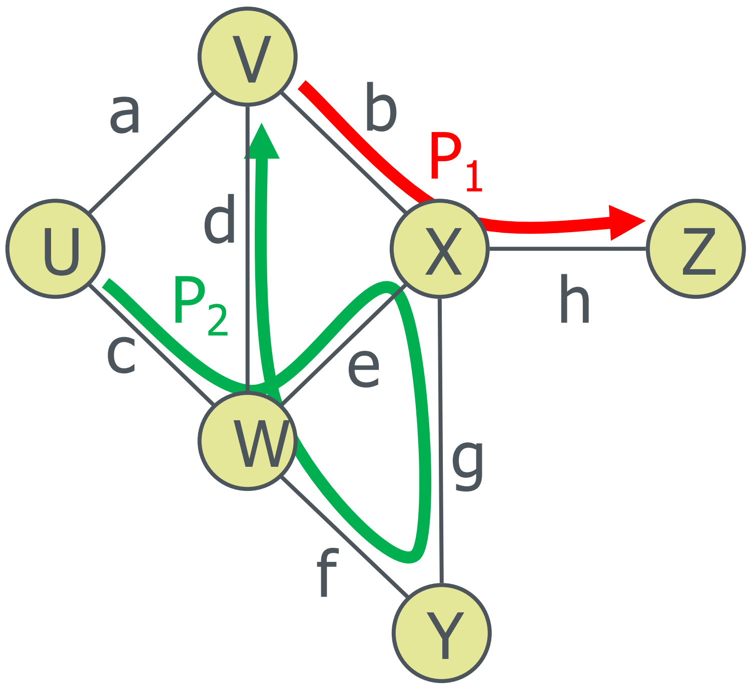
Terminology

- End vertices (or endpoints) of an edge
  - **U** and **V** are the endpoints of **a**
- Edges incident on a vertex
  - **a**, **d**, and **b** are incident on **V**
- Adjacent vertices
  - **U** and **V** are adjacent
- Degree of a vertex
  - **X** has degree **5**
- Parallel edges
  - **h** and **i** are parallel edges
- Self-loop
  - **j** is a self-loop



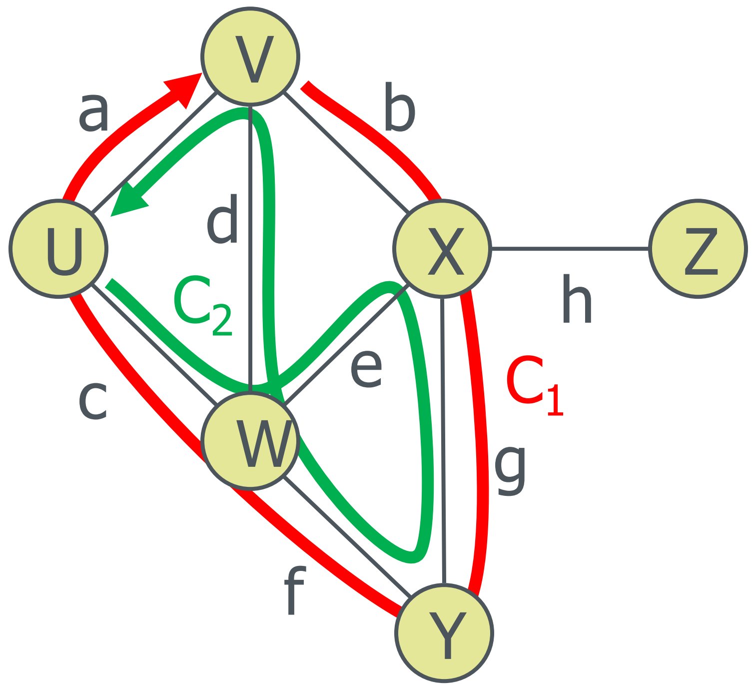
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

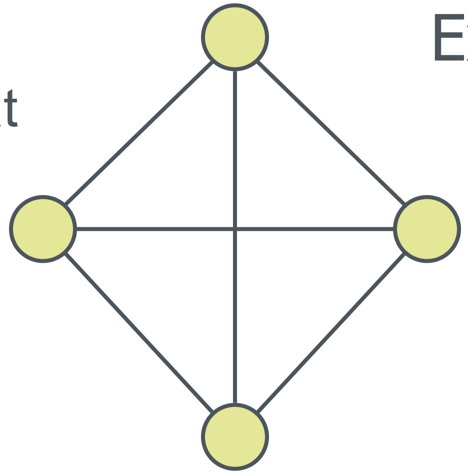
In an undirected graph with no self-loops and no multiple edges

$m \leq n\,(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

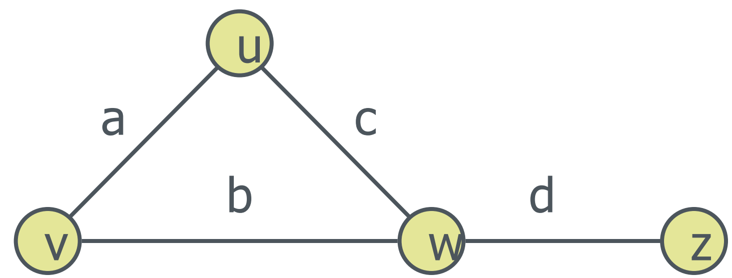
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

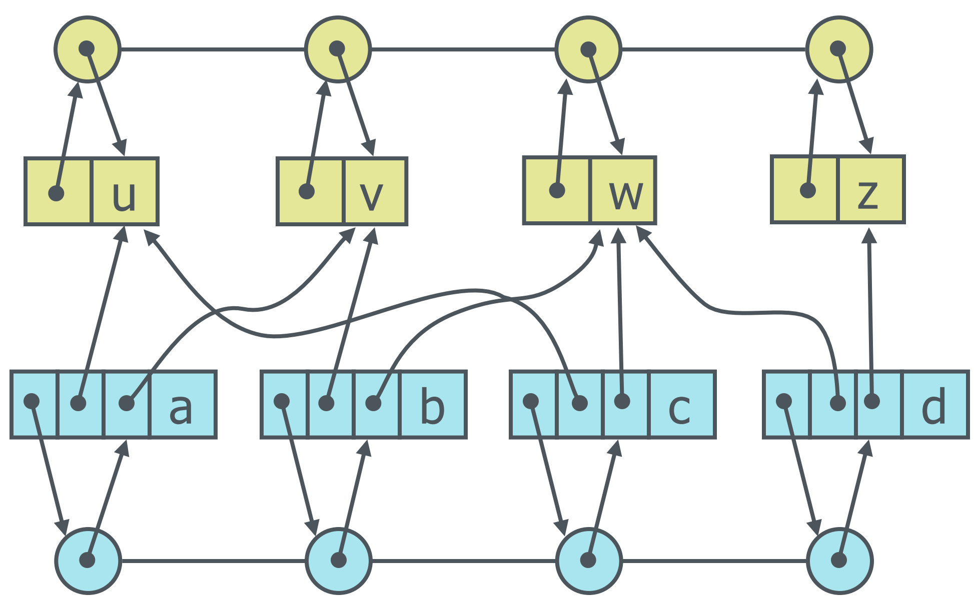
Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - e.endVertices(): a list of the two endvertices of e
  - e.opposite(v): the vertex opposite of v on e
  - u.isAdjacentTo(v): true iff u and v are adjacent
  - *v: reference to element associated with vertex v
  - *e: reference to element associated with edge e
- Update methods
  - insertVertex(o): insert a vertex storing element o
  - insertEdge(v, w, o): insert an edge (v,w) storing element o
  - eraseVertex(v): remove vertex v (and its incident edges)
  - eraseEdge(e): remove edge e
- Iterable collection methods
  - incidentEdges(v): list of edges incident to v
  - vertices(): list of all vertices in the graph
  - edges(): list of all edges in the graph

Edge List Structure

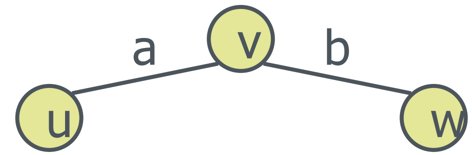
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects

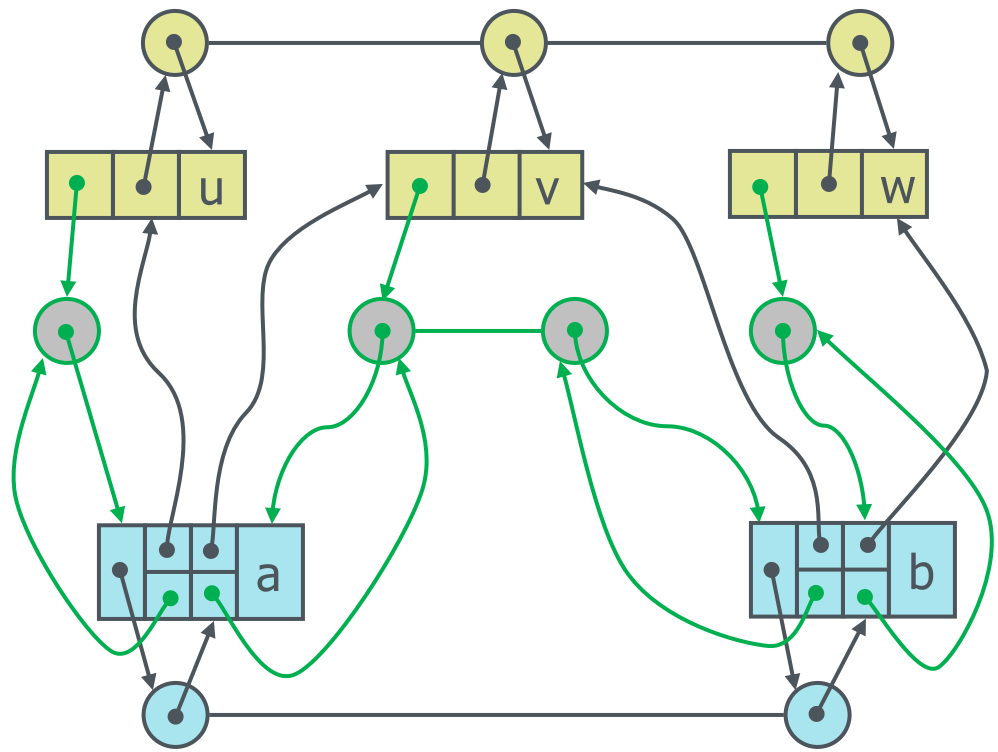




Adjacency List Structure

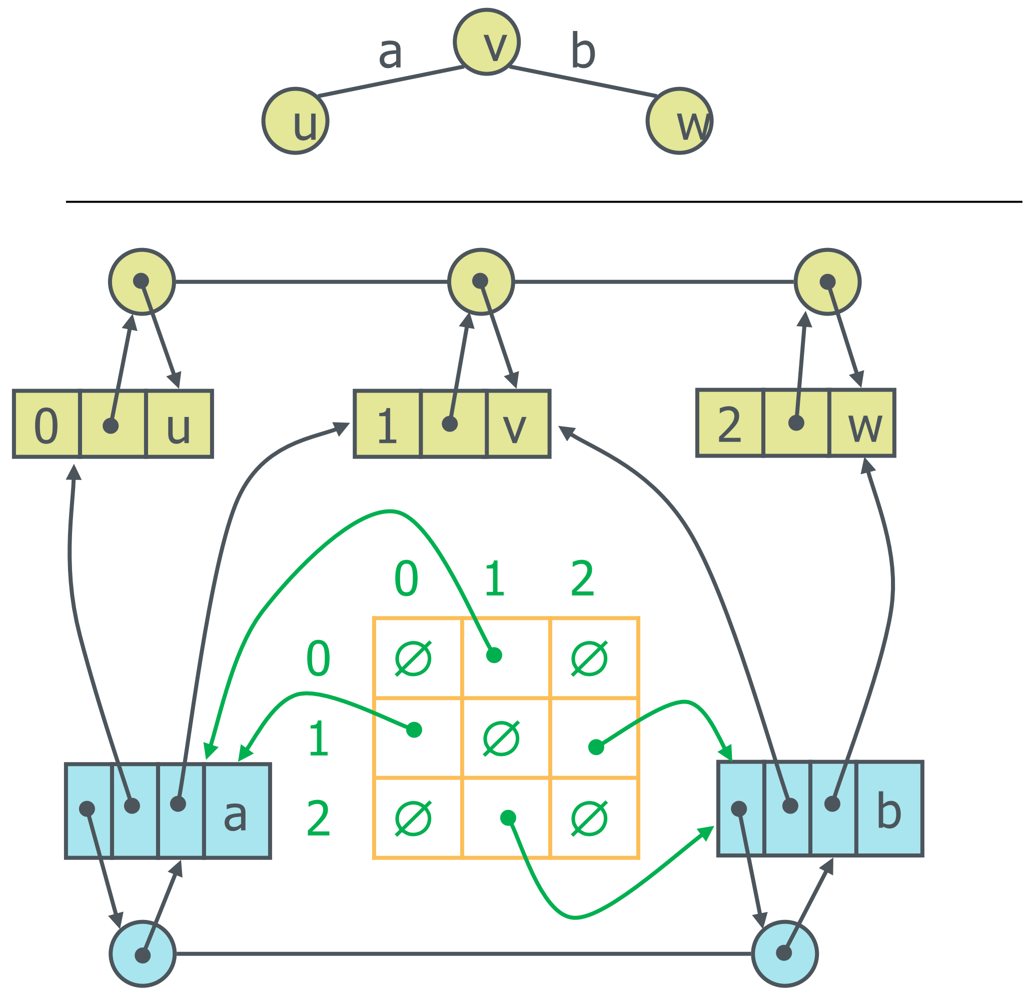
- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices





Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



What next?

- We will see the BFS and DFS algorithms next
  - We will only consider the algorithms and not the implementation

Questions?

Please evaluate this course!
https://evals.mcmaster.ca/
Thank you