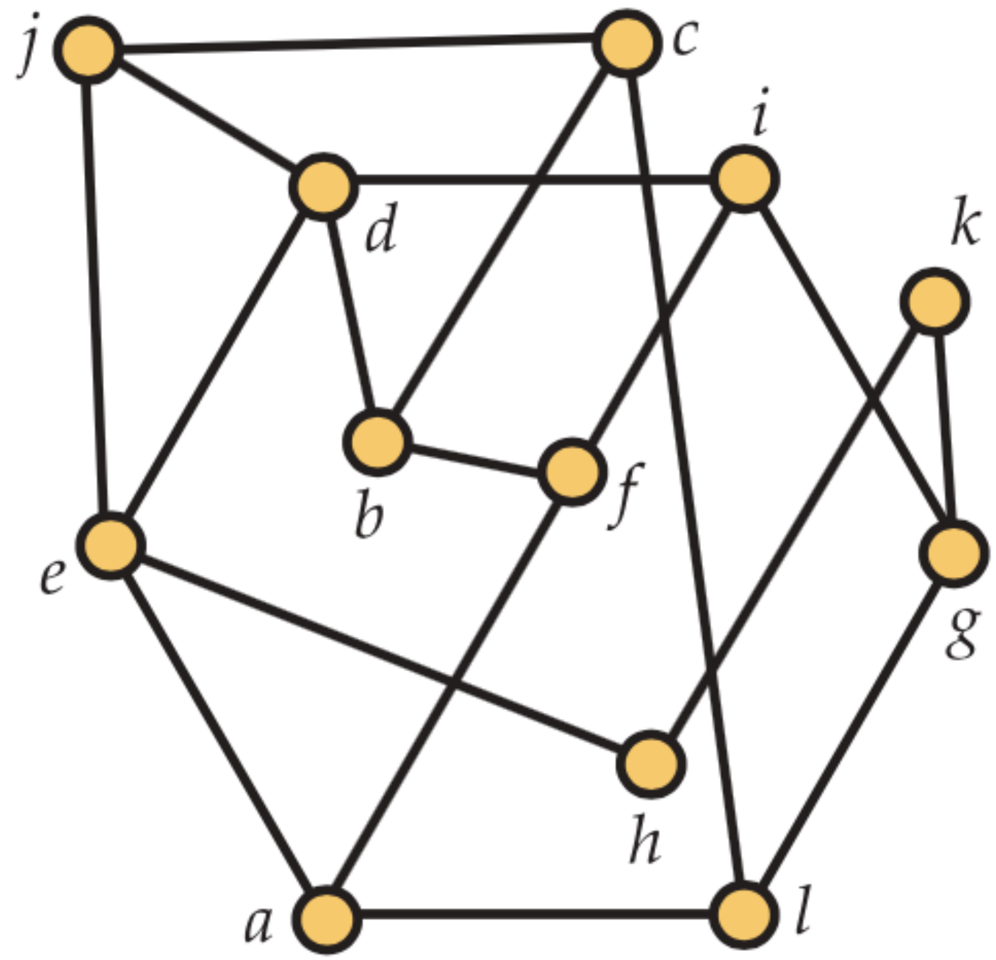


CSCI 241

Lecture 22

Miscellaneous, Review

Announcements

- Material through today is on the exam. Just a few miscellaneous topics, review thereafter.
- There will be in-class exercises today and Friday

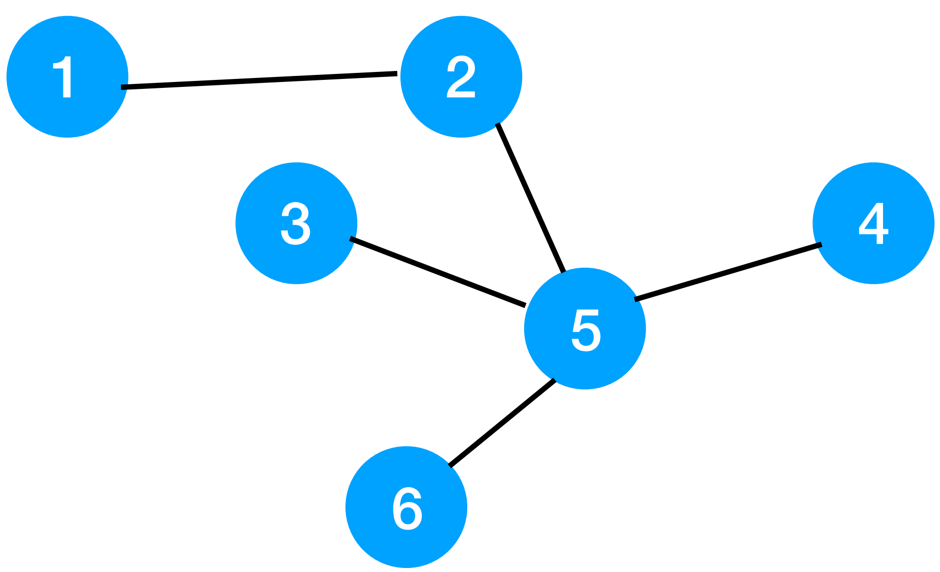
Goals

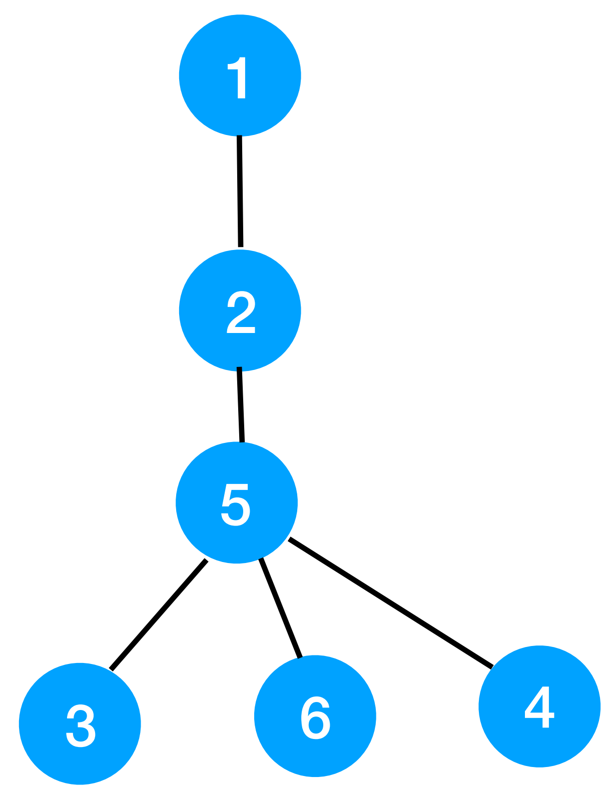
- Know the definition of planarity in graphs
- Know what it means for a sorting algorithm to be in-place
- Understand the heap sort algorithm.
- Work on some review problems.

Drawing Graphs

- The same graph can be drawn (infinitely!) many different ways.

$V = \{1,2,3,4,5,6\}$
$E = \{(1,2), (2,5), (3,5)$
$(4,5), (5,6)\}$

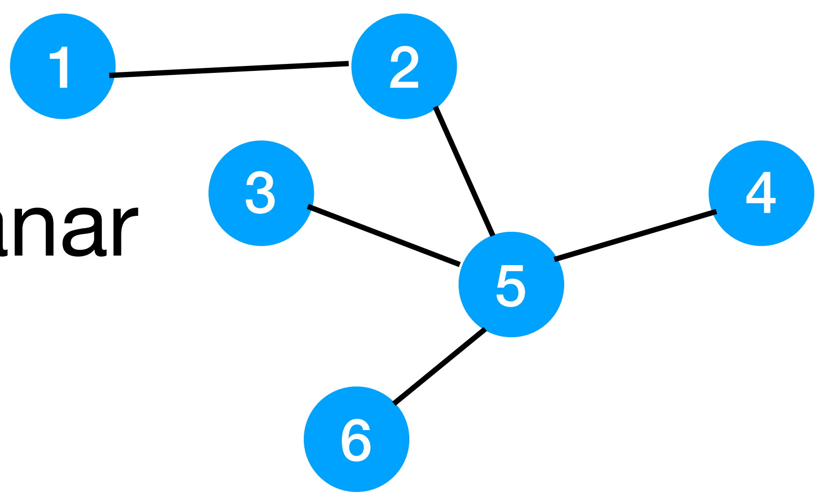


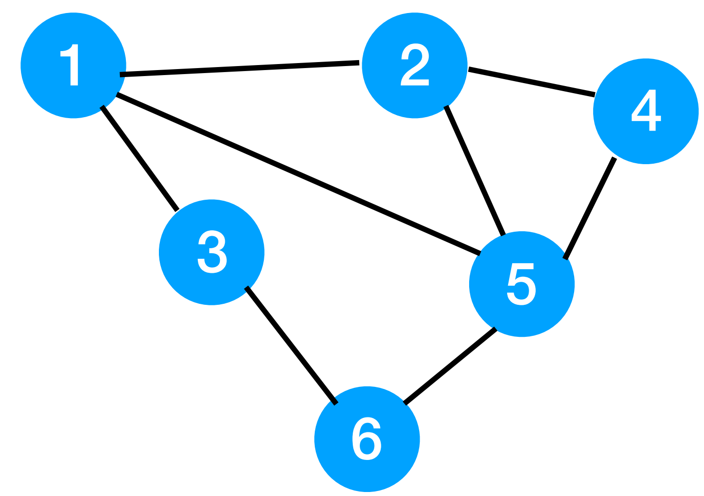


Planarity

- If a graph can be drawn without crossing edges, it is planar.

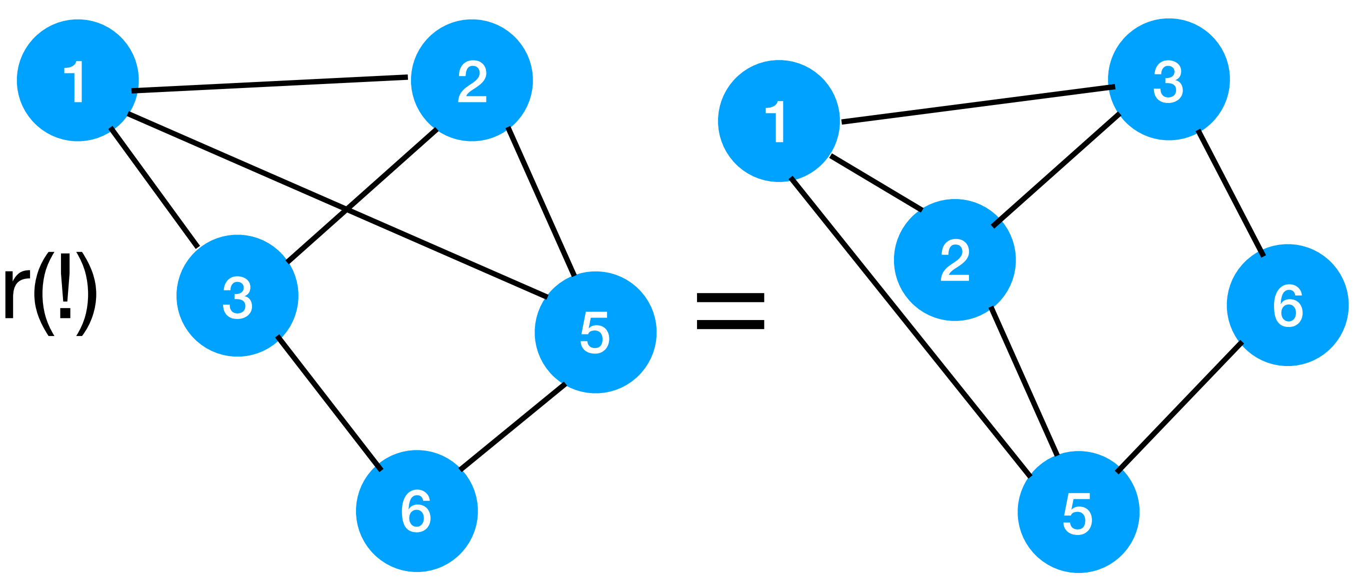
Planar

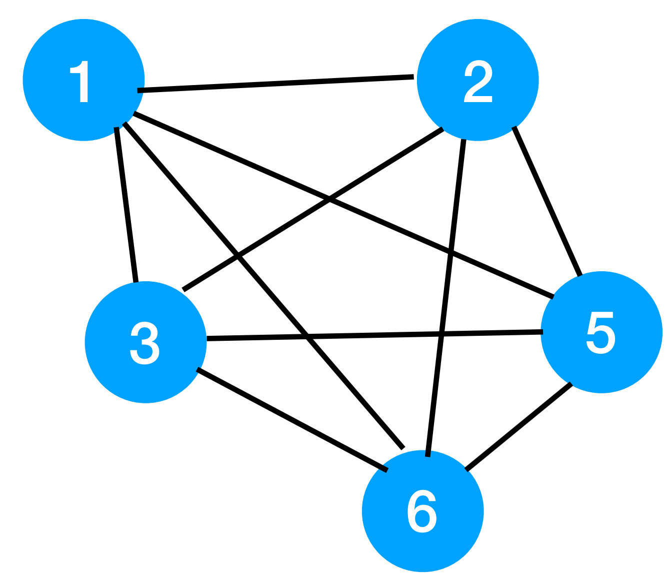




Planar

Planar(!)



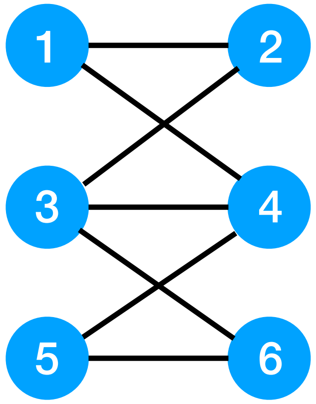


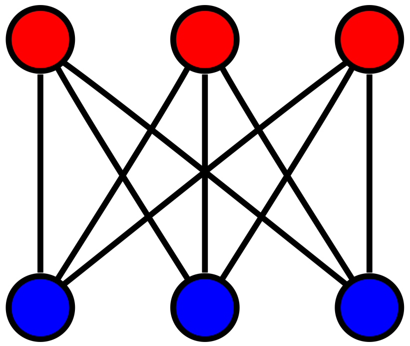
Not planar

Planar Graphs

A complete graph is a graph with all possible edges.

- Which of the following is planar?

1. The complete graph of 4 nodes
2. The complete graph of 5 nodes
3. This graph: 

4. This graph: 

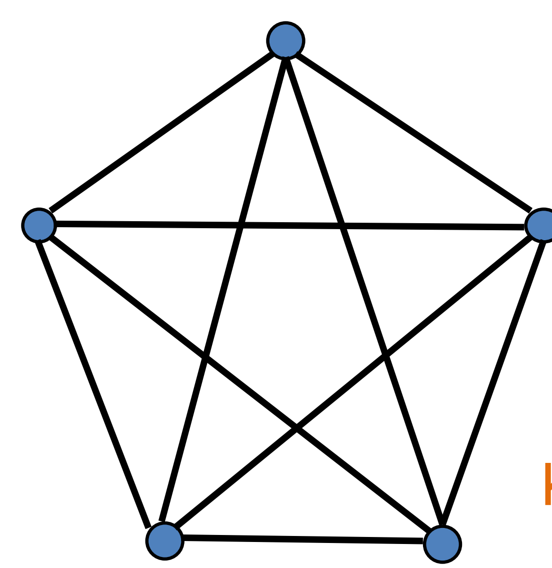
Aside: Detecting Planarity

A subgraph of a graph is a graph whose vertex and edge sets are subsets of the larger graph's.

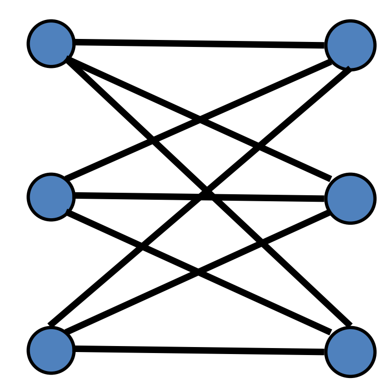
- Elements of the edge subset can only contain nodes in the vertex subset.

- There's a (non-obvious) theorem that says a graph is planar if and only if it does not contain* one of these as a subgraph:

*The definition of "contain" is slightly more general than having one of these directly as a subgraph.



K_5



$K_{3,3}$

Magic trick time!

- Remember that heap lecture when I ran out of time for my magic trick?

Heapsort

```
public static void heapsort(int[] b) {




}
```

Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap<Integer>();
  // put everything into a heap - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

  // pull everything out in order - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    b[k] = poll(b, k);
  }
}
```

Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap<Integer>();
  // put everything into a heap - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

  // pull everything out in order - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    b[k] = poll(b, k);
  }
}
```

Worst-case runtime: O(n log n) !

In-Place

- Time complexity: how many operations?
- Space complexity: how much (extra) memory?
  - Usually don't count the size of the input, because we have no choice but to store it.

In-Place

- Time complexity: how many operations?
- Space complexity: how much (extra) memory?
  - Usually don't count the size of the input, because we have no choice but to store it.

ABCD:
How much extra space does insertion sort use?

A. O(1)
B. O(log n)
C. O(n)
D. O(n^2)

```
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] > A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

In-Place

A sort is considered **in-place** if it requires O(1) storage space in addition to the input.

ABCD:
How much extra space does insertion sort use?

A. O(1)
B. O(log n)
C. O(n)
D. O(n^2)

```
insertionSort(A):
  i = 0;
  while i < A.length:
    j = i;
    while j > 0 and A[j] > A[j-1]:
      swap(A[j], A[j-1])
      j--
    i++
```

Sort Space Complexity

- Which of the following are in-place sorts?

    1. Insertion
    2. Selection
    3. Quick
    4. Merge
    5. Radix
    6. Heap