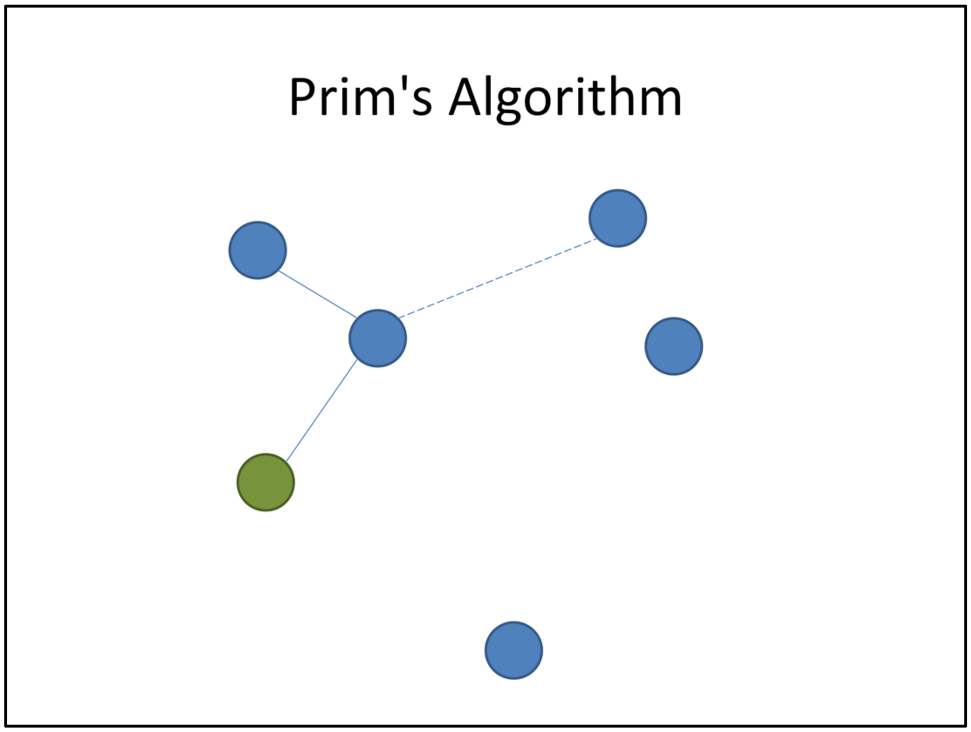
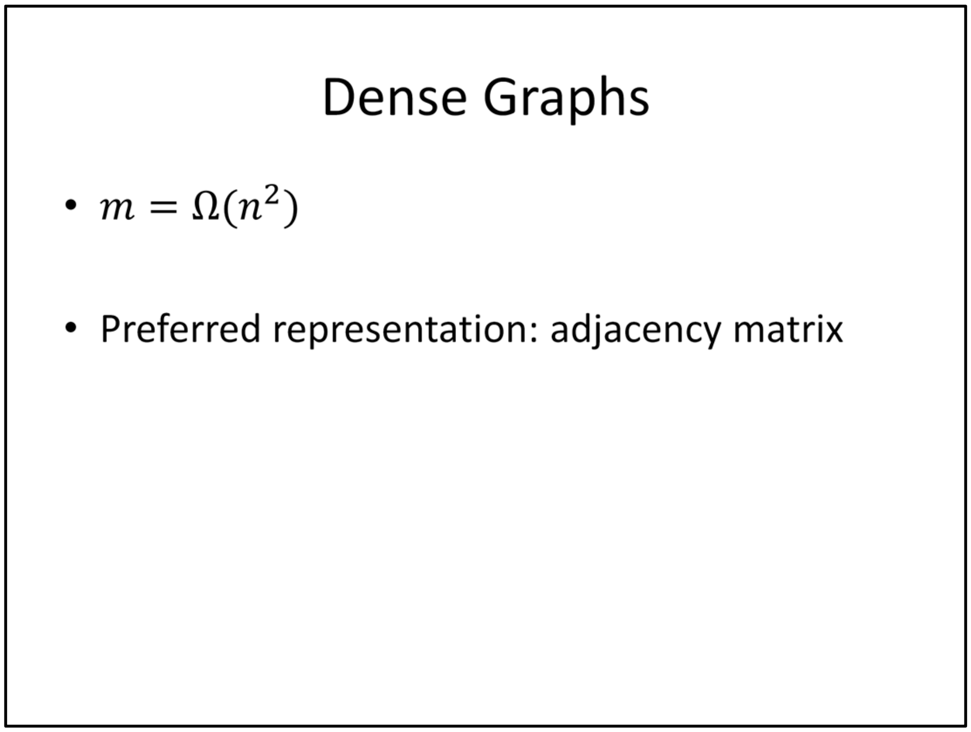
# CS161L: Implementation of Algorithms

Friday, May 30, 2014



Prim's Algorithm

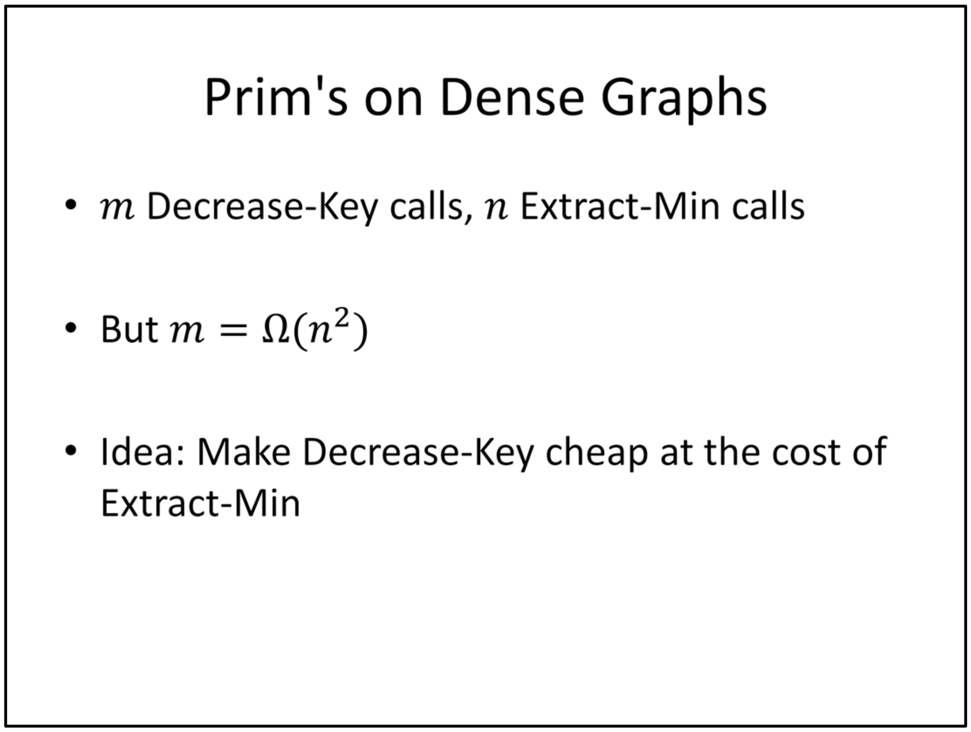
Last week we talked about implicit graph representations. This week we're going to be implementing Prim's Algorithm in two different ways to practice working with explicit graph representations, namely, adjacency matrices and adjacency lists. Remember that Prim's Algorithm grows an MST from a single node outward. At each step, we add the node that's closest to the tree we've built up so far.



## Dense Graphs

- $m = \Omega(n^2)$
- Preferred representation: adjacency matrix

Now, we can loosely classify graphs into dense graphs and sparse graphs. Dense graphs are graphs where there are edges between most of the nodes. This makes the adjacency matrix representation better suited for our purposes because the space drawback isn't as big a deal when you would already have to spend that much space no matter what.



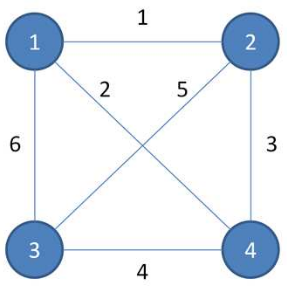
## Prim's on Dense Graphs

- $m$ Decrease-Key calls, $n$ Extract-Min calls
- But $m = \Omega(n^2)$
- Idea: Make Decrease-Key cheap at the cost of Extract-Min

Now, if you'll remember what happens in Prim's algorithm, we're going to maintain the nodes of the graph in a priority queue, and then we're going to call Decrease-Key once for every edge, and Extract-Min once for every node. But remember that in a dense graph with n nodes, we have order n^2 edges, which means we can afford to be inefficient in our Extract-Min calls, especially if it means we can make Decrease-Key more efficient.

## Prim's on Dense Graphs

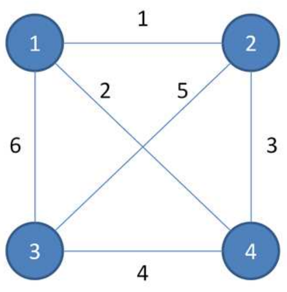
| | | | |
|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ |



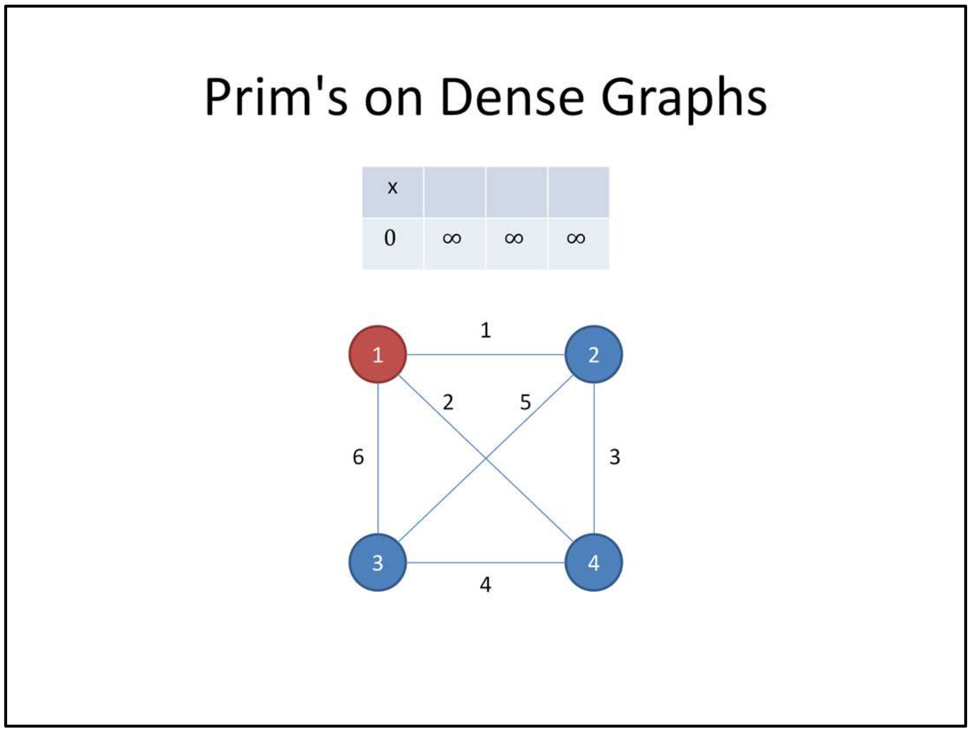
So what kind of a priority queue should we use? Well, I'm going to propose the following: We make two arrays, one that tells us whether each node has already been added to the MST, and one that tells us the distance from the node to our MST so far. We start with no nodes added, and all distances at infinity.

## Prim's on Dense Graphs

| | | | |
|---|---|---|---|
| 0 | ∞ | ∞ | ∞ |

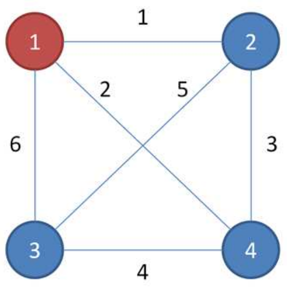


We pick a node, say the first node, and set its distance to 0 to start off the algorithm. Now the main loop of the algorithm begins. We scan over our array to find the smallest distance to a node that hasn't been added yet.



## Prim's on Dense Graphs

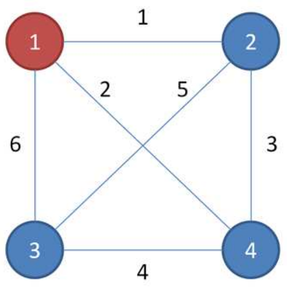
| x | | | |
|---|---|---|---|
| 0 | ∞ | ∞ | ∞ |



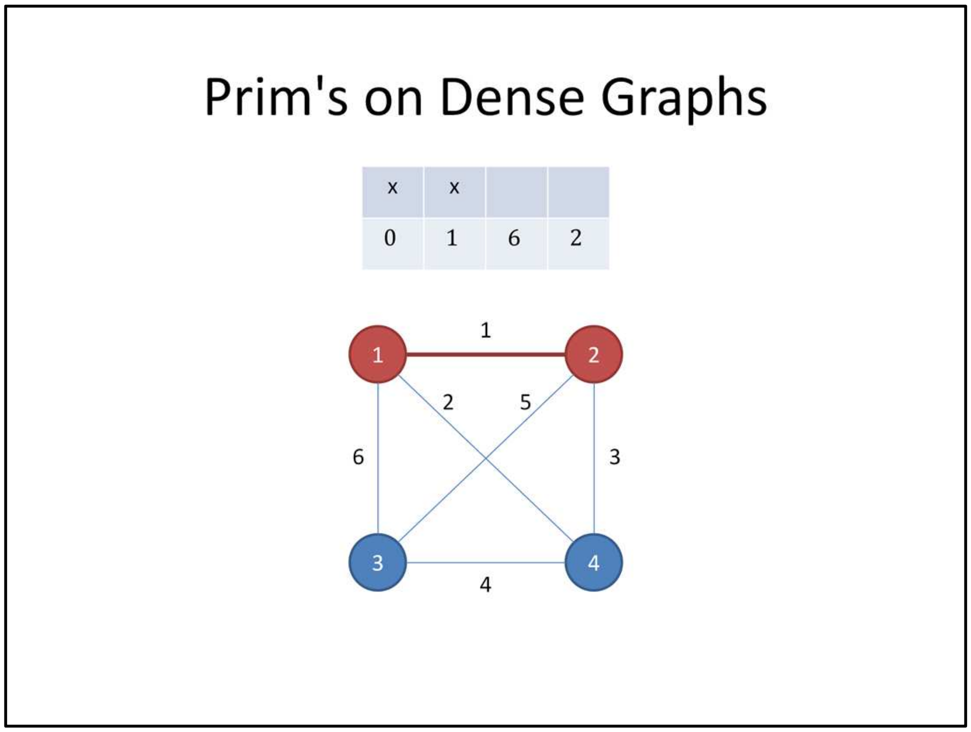
In this case, that would be the first node. We add it to our MST.

## Prim's on Dense Graphs

| x | | | |
|---|---|---|---|
| 0 | 1 | 6 | 2 |

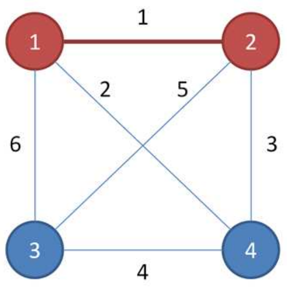


Then we update the distances to all nodes that haven't been added to the MST yet. How do we do that? By looping over all edges connected to our current node. But in adjacency matrix format, that means all we have to do is loop over all the numbers in a row of the adjacency matrix, and overwrite the values in our queue if they're better, and if the corresponding node hasn't been added yet.

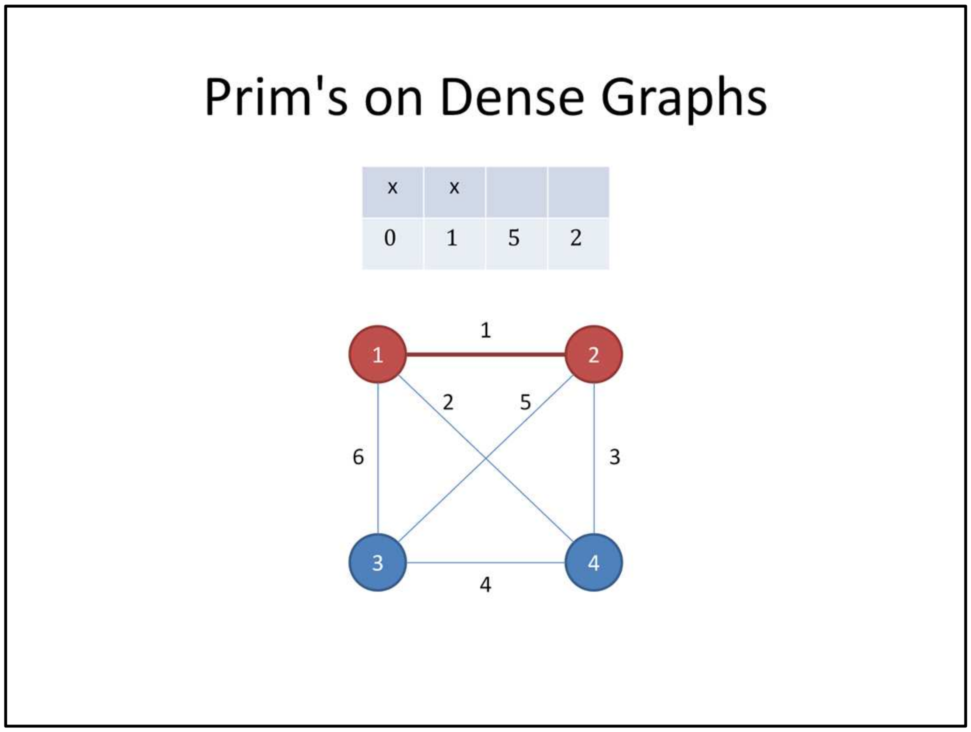


## Prim's on Dense Graphs

| x | x |  |  |
|---|---|---|---|
| 0 | 1 | 6 | 2 |

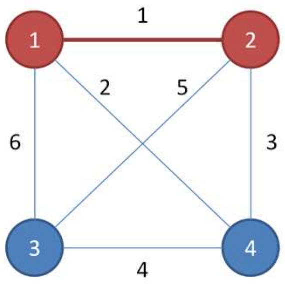


Then we scan over the array again to find the closest node that hasn't been added yet.  In this case, it's the second node.

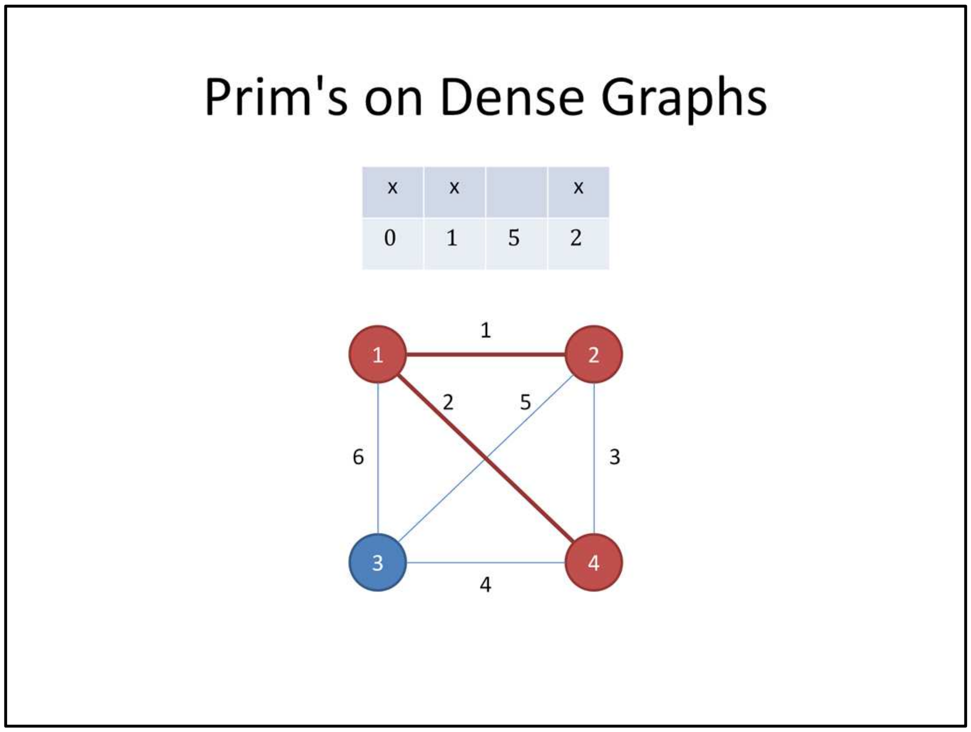


## Prim's on Dense Graphs

| x | x |  |  |
|---|---|---|---|
| 0 | 1 | 5 | 2 |

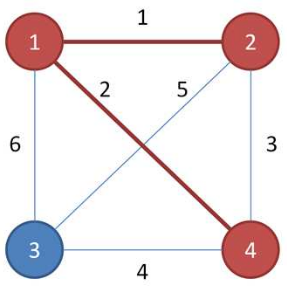


Then we update the distances.

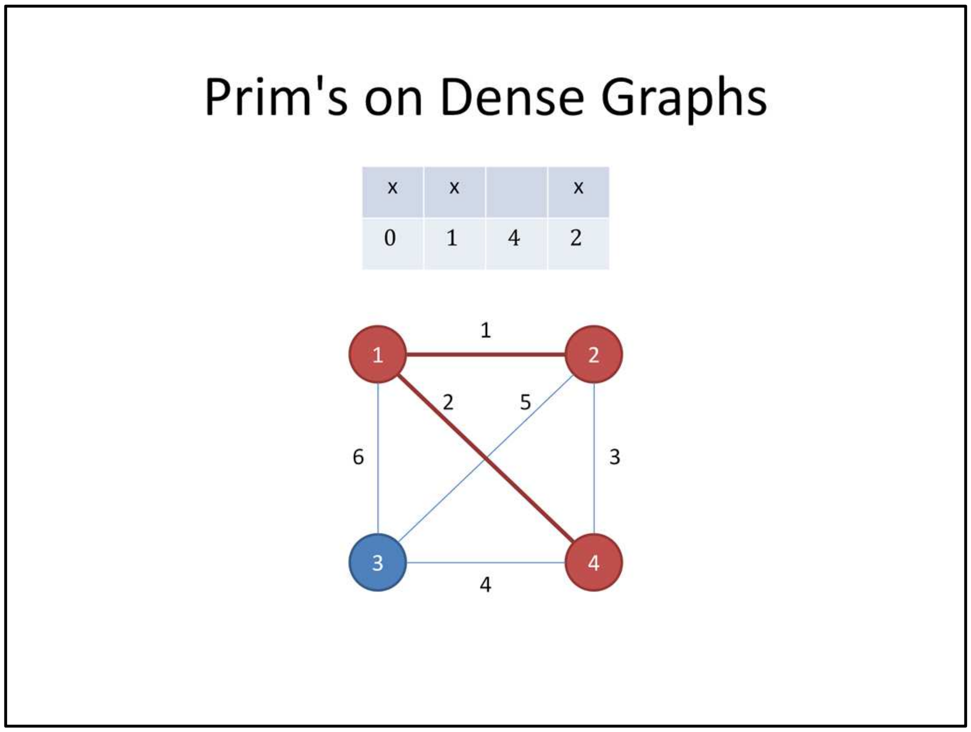


## Prim's on Dense Graphs

| x | x |  | x |
|---|---|---|---|
| 0 | 1 | 5 | 2 |

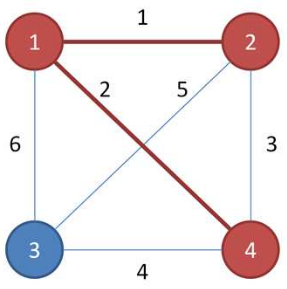


We continue this pattern,

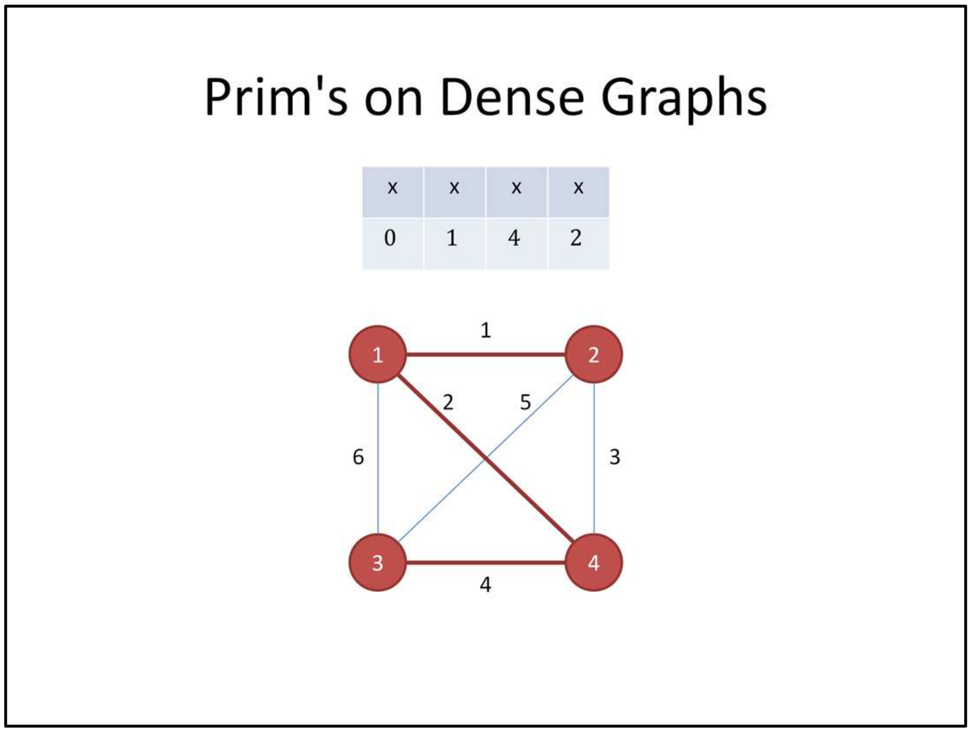


## Prim's on Dense Graphs

| x | x |  | x |
|---|---|---|---|
| 0 | 1 | 4 | 2 |

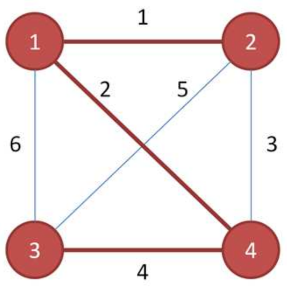


looping over the entire array each time,

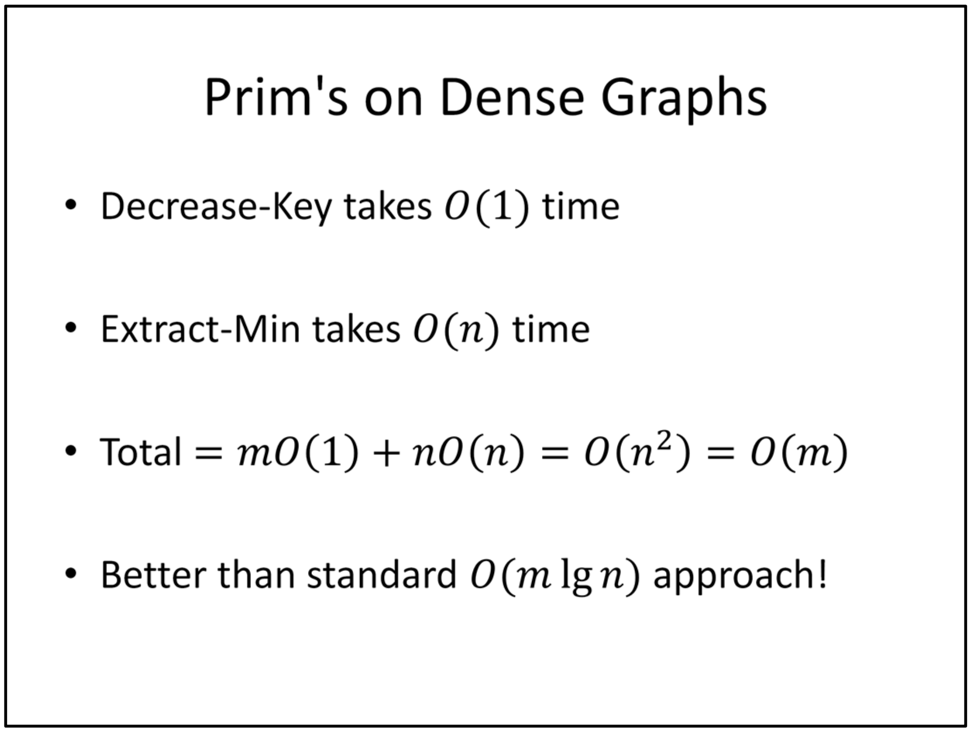


## Prim's on Dense Graphs

| x | x | x | x |
|---|---|---|---|
| 0 | 1 | 4 | 2 |



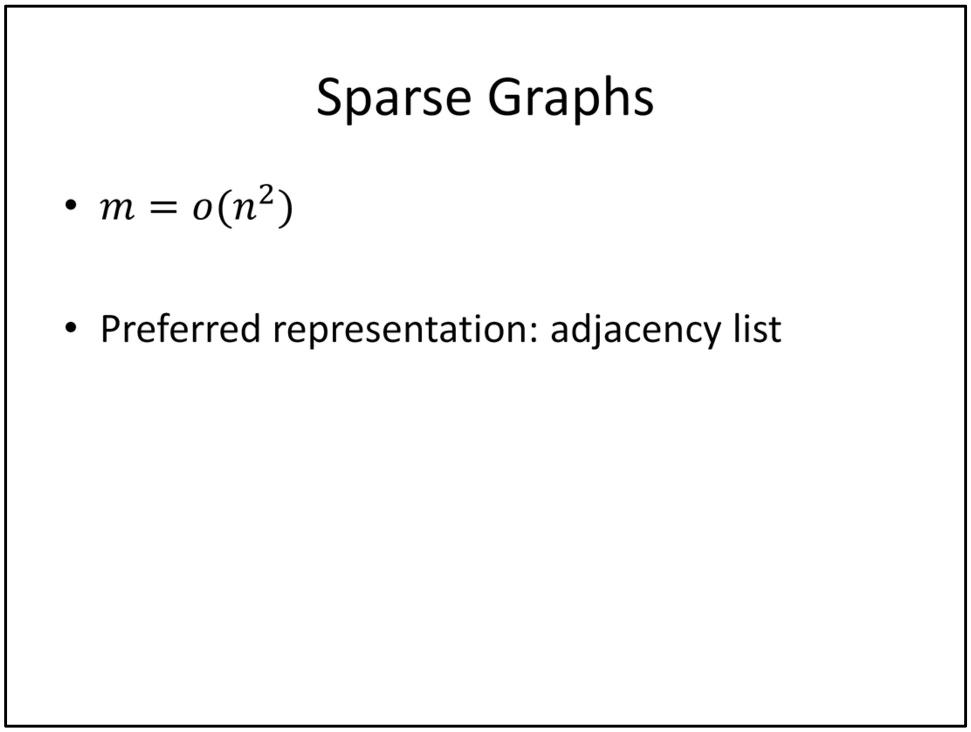
until we've added all the nodes to our MST.



## Prim's on Dense Graphs

- Decrease-Key takes $O(1)$ time
- Extract-Min takes $O(n)$ time
- Total $= mO(1) + nO(n) = O(n^2) = O(m)$
- Better than standard $O(m \lg n)$ approach!

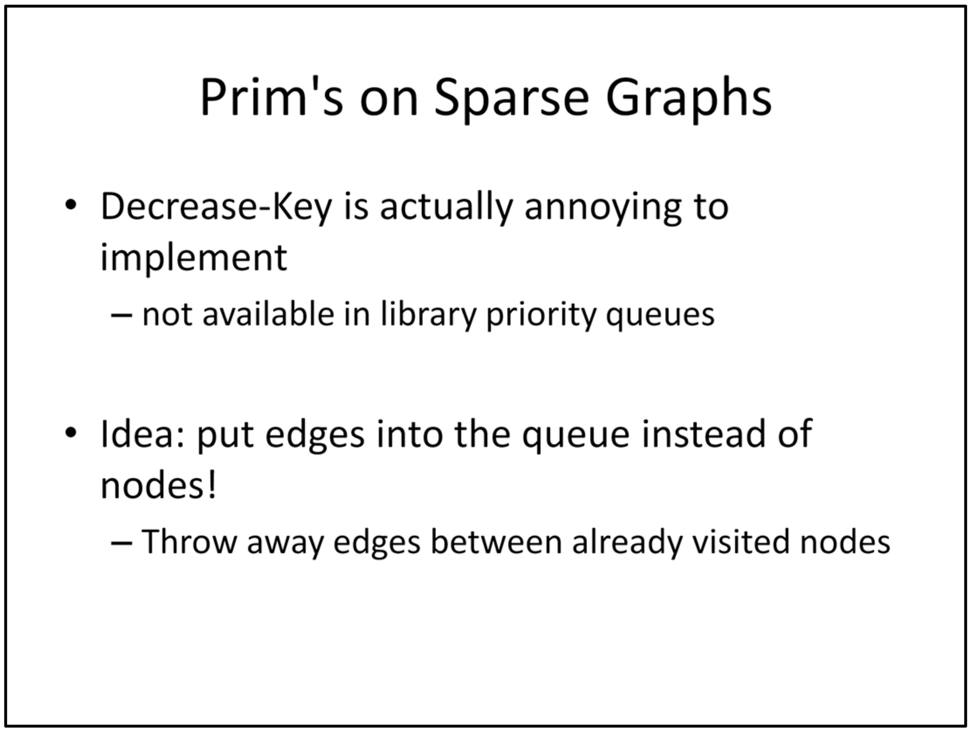
Now how expensive is this to do? Well, each Decrease-Key operation is just changing one entry in our array, so it takes constant time. Extract-Min requires looping over the whole array, so it takes order n time, but because there are so many more edges than nodes, the total runtime is just n^2, or m. This actually beats what we would get if we used a binary heap as described in lecture on Wednesday.



## Sparse Graphs

- $m = o(n^2)$
- Preferred representation: adjacency list

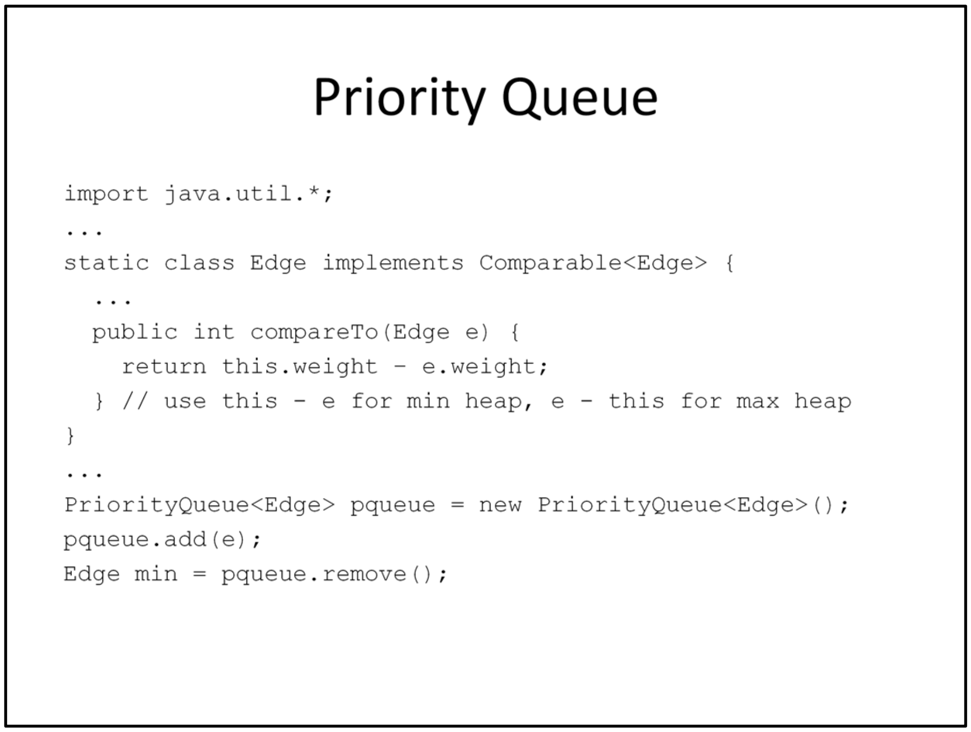
Next, let's move on to sparse graphs. Sparse graphs are ones where most possible edges are NOT present in the graph. In this case, we do get asymptotic savings from using an adjacency list instead of an adjacency matrix.



## Prim's on Sparse Graphs

- Decrease-Key is actually annoying to implement
  - not available in library priority queues
- Idea: put edges into the queue instead of nodes!
  - Throw away edges between already visited nodes

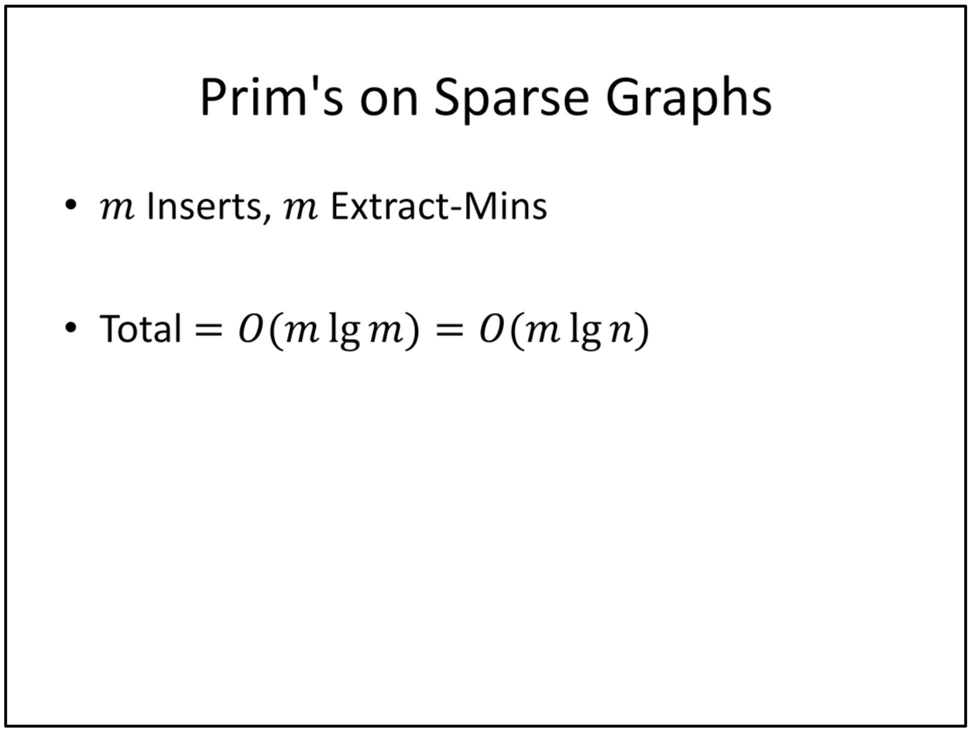
Now, Prim's Algorithm as described in lecture actually has a technical detail that's really annoying to deal with, and that's the fact that Decrease-Key is hard to implement in log time. The problem is that you need to also find the key that you want to decrease, and that's hard to do in general. In fact, the library implementation of priority queues doesn't even bother doing this. Thankfully, there's a workaround that we can use for Prim's Algorithm. What we can do is instead of maintaining a priority queue on the nodes, we can maintain a priority queue of edges. This way, we never have to call Decrease-Key. Instead, when we Extract-Min, we look at the nodes that belong to the edge, and if both of them are already in our MST, then we throw away the edge and try again. When we find an edge that has a new node, then we add all the edges from that node to its unvisited neighbors.



## Priority Queue

```
import java.util.*;
...
static class Edge implements Comparable<Edge> {
  ...
  public int compareTo(Edge e) {
    return this.weight - e.weight;
  } // use this - e for min heap, e - this for max heap
}
...
PriorityQueue<Edge> pqueue = new PriorityQueue<Edge>();
pqueue.add(e);
Edge min = pqueue.remove();
```

So what does a PriorityQueue look like in Java? Well, it takes in an object that it's capable of comparing, which it uses to determine priority. You can either pass in a comparator, or make the object itself Comparable. This saves you from having to write an extra comparator class; instead, you just implement this compareTo function which returns a negative number, 0, or a positive number if you are smaller than, equal to, or larger than the argument you passed in. A convenient shorthand for doing this is to take our weight and subtract the argument's weight. Be warned, though, that this shorthand is susceptible to overflow, so make sure the numbers you're comparing aren't too far apart. For this assignment, the numbers are safe, so you can get away with this. As for the PriorityQueue class itself, notice that it calls insert "add" and extract-min "remove".



## Prim's on Sparse Graphs

- $m$ Inserts, $m$ Extract-Mins
- Total $= O(m \lg m) = O(m \lg n)$

So how fast is this algorithm?  Well, we're inserting each edge into the priority queue once, and we're extracting every edge out eventually.  This means the whole process is going to cost us m lg m time.  But remember that since we have at most n^2 edges, this is equivalent to m lg n time, which is the same as the runtime we got from the version taught on Wednesday.