CS 2110 Fall 2009

Prelim 2 Review

Prelim 2 Information

- Prelim 2
  - Tuesday, November 16th (Tomorrow!)
  - Uris G01 (same as Prelim 1)
  - 7:30-9:00
- Topics:
  - Threads and Concurrency (basic)
  - Big O
  - ADTs: Stacks, Queues, PriorityQueues, Maps, Sets
  - Graph Algorithms: Prim's, Kruskal's, Dijkstra's
  - No Induction ☹

For the prelim...

- Don’t spend your time memorizing Java APIs!
- If you want to use an ADT, it’s acceptable to write code that looks reasonable, even if it’s not the exact Java API. For example,

```
Queue<Integer> myQueue = new Queue<Integer>();
myQueue.enqueue(5);
...
int x = myQueue.dequeue();
```

- This is not correct Java (Queue is an interface! And Java calls *enqueue* and *dequeue* “add” and “poll”)
- But it’s fine for the exam.

Big-O notation

- Big-O is an asymptotic upper bound on a function
  - *"f(x) is O(g(x))"*

    Meaning: There exists some constant *k* such that

    $f(x) \le k\, g(x)$

    ...as x goes to infinity
- Often used to describe upper bounds for both worst-case and average-case algorithm runtimes
  - Runtime is a function: The number of operations performed, usually as a function of input size

Big-O notation

- For the prelim, you should know…
  - Worst case Big-O complexity for the algorithms we've covered and for common implementations of ADT operations
    - Examples
      - Mergesort is worst-case O(n log n)
      - PriorityQueue insert using a heap is O(log n)
  - Average case time complexity for some algorithms and ADT operations, if it has been noted in class
    - Examples
      - Quicksort is average case O(n log n)
      - HashMap insert is average case O(1)

Big-O notation

- For the prelim, you should know…
  - How to estimate the Big-O worst case runtimes of basic algorithms (written in Java or pseudocode)
    - Count the operations
    - Loops tend to multiply the loop body operations by the loop counter
    - Trees and divide-and-conquer algorithms tend to introduce *log(n)* as a factor in the complexity
    - Basic recursive algorithms, i.e., binary search or mergesort

Abstract Data Types

- What do we mean by “abstract”?
  - Defined in terms of operations that can be performed, not as a concrete structure
    - Example: Priority Queue is an ADT, Heap is a concrete data structure
- For ADTs, we should know:
  - Operations offered, and when to use them
  - Big-O complexity of these operations for standard implementations

ADTs: The Bag Interface

```
interface Bag<E> {
   void insert(E obj);
   E extract(); //extract some element
   boolean isEmpty();
   E peek(); // optional: return next
                           element without removing
}
```

Examples: Queue, Stack, PriorityQueue

Queues

- First-In-First-Out (FIFO)
  - Objects come out of a queue in the same order they were inserted
- Linked List implementation
  - **insert(obj):** O(1)
    - Add object to tail of list
    - Also called **enqueue**, **add** (Java)
  - **extract():** O(1)
    - Remove object from head of list
    - Also called **dequeue**, **poll** (Java)

Stacks

- Last-In-First-Out (LIFO)
  - Objects come out of a queue in the opposite order they were inserted
- Linked List implementation
  - **insert(obj):** O(1)
    - Add object to tail of list
    - Also called **push** (Java)
  - **extract():** O(1)
    - Remove object from head of list
    - Also called **pop** (Java)

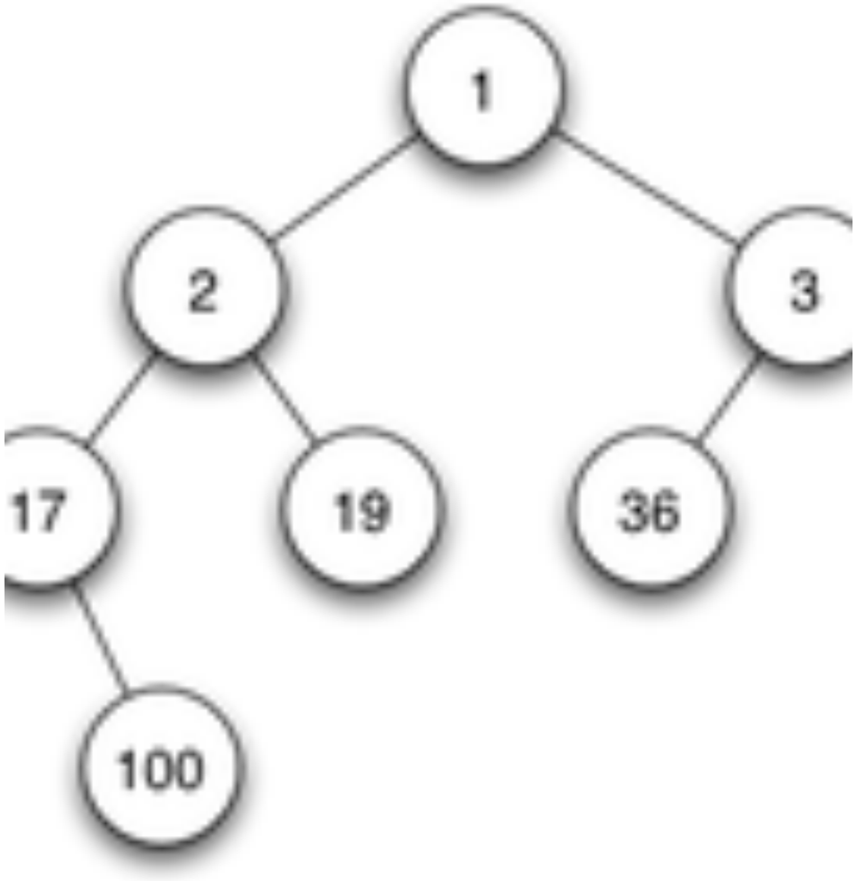
Priority Queues

- Objects come out of a Priority Queue according to their priority
- Generalized
  - By using different priorities, can implement Stacks or Queues
- Heap implementation (as seen in lecture)
  - **insert(obj, priority):** O(log n)
    - insert object into heap with given priority
    - Also called **add** (Java)
  - **extract():** O(log n)
    - Remove and return top of heap (minimum priority element)
    - Also called **poll** (Java)

Heaps

- Concrete Data Structure
- Balanced binary tree
- Obeys **heap order invariant:**

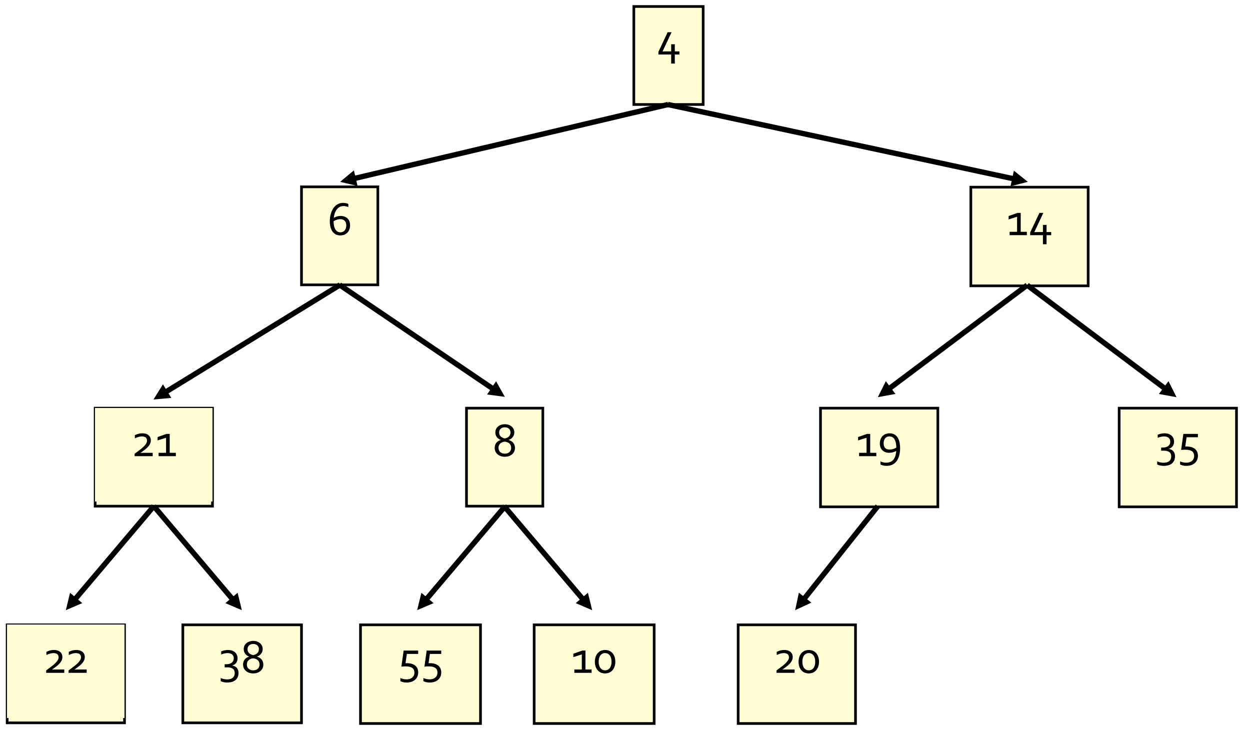
  Priority(child) ≥ Priority(parent)
- Operations
  - insert(value, priority)
  - extract()



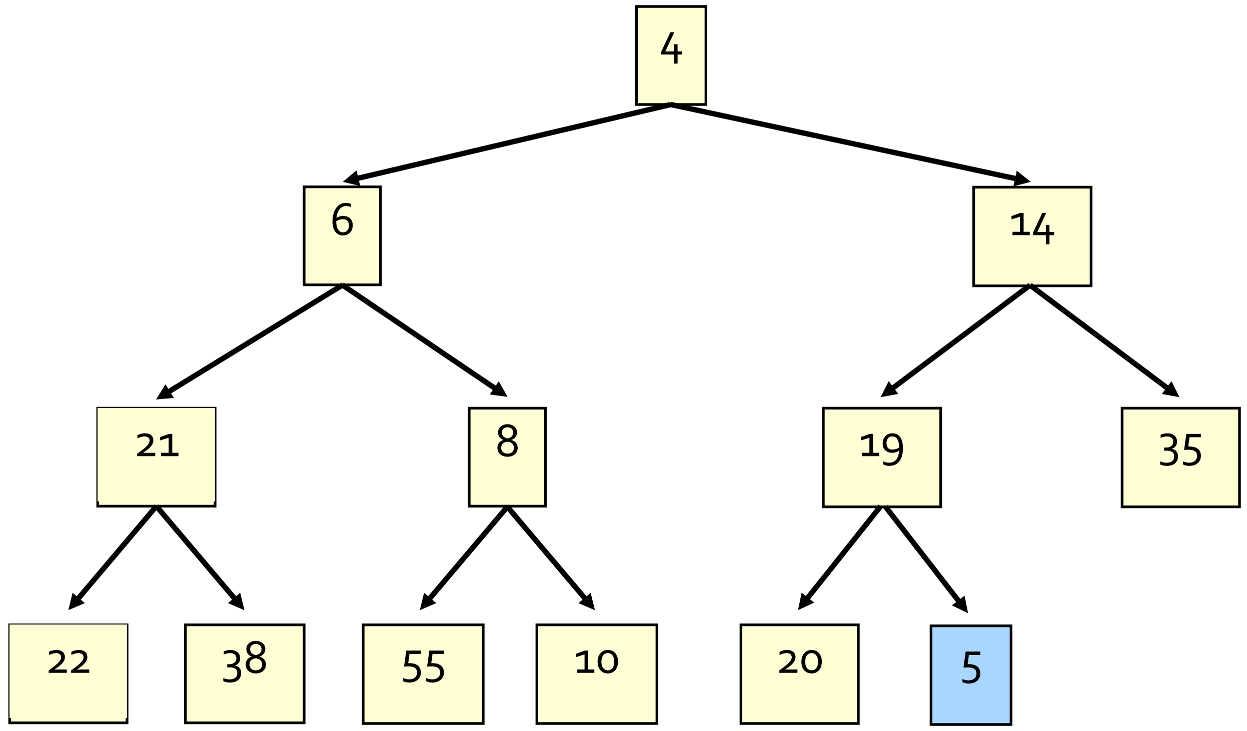
Heap insert()

- Put the new element at the end of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place
- The heap invariant is maintained!

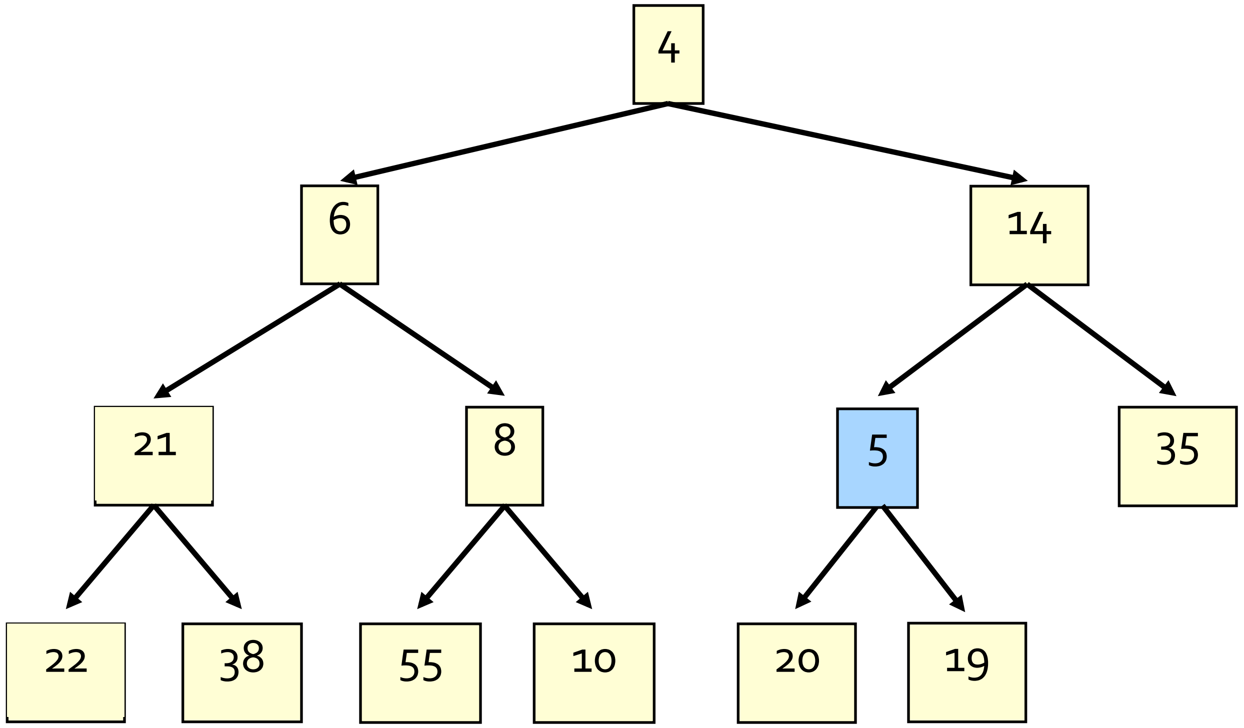
Heap insert()



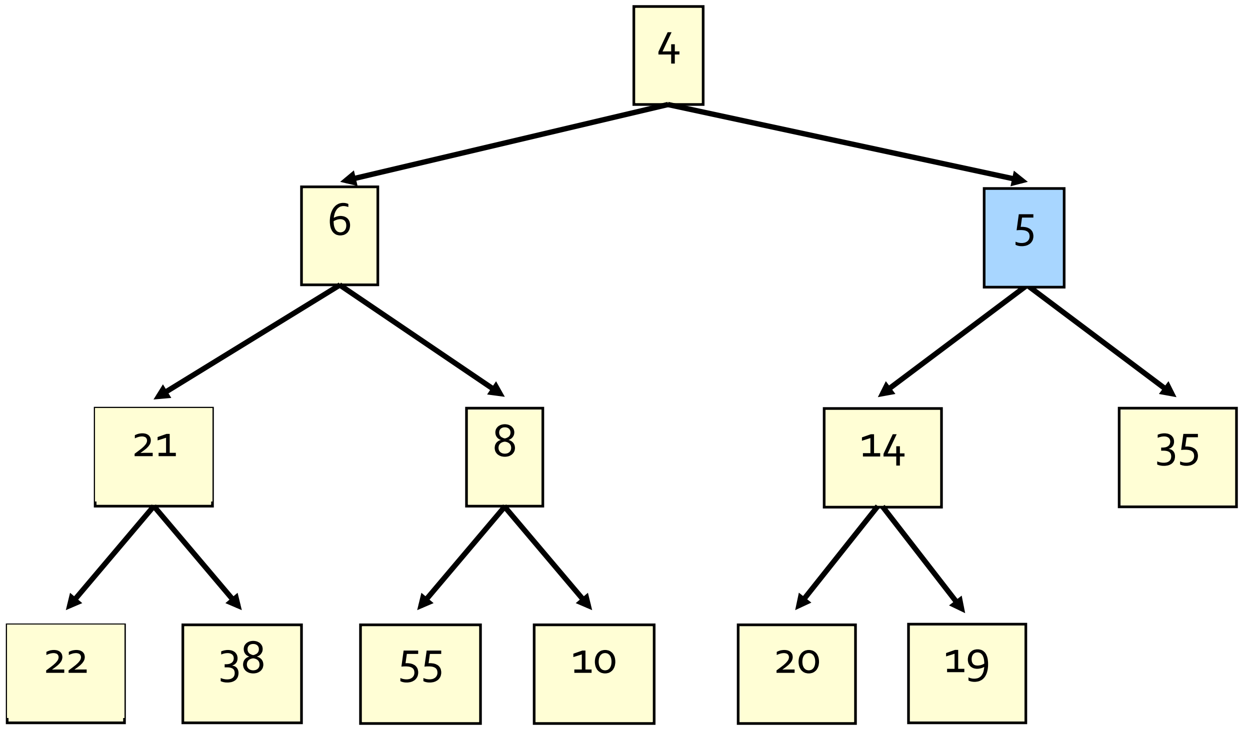
Heap insert()



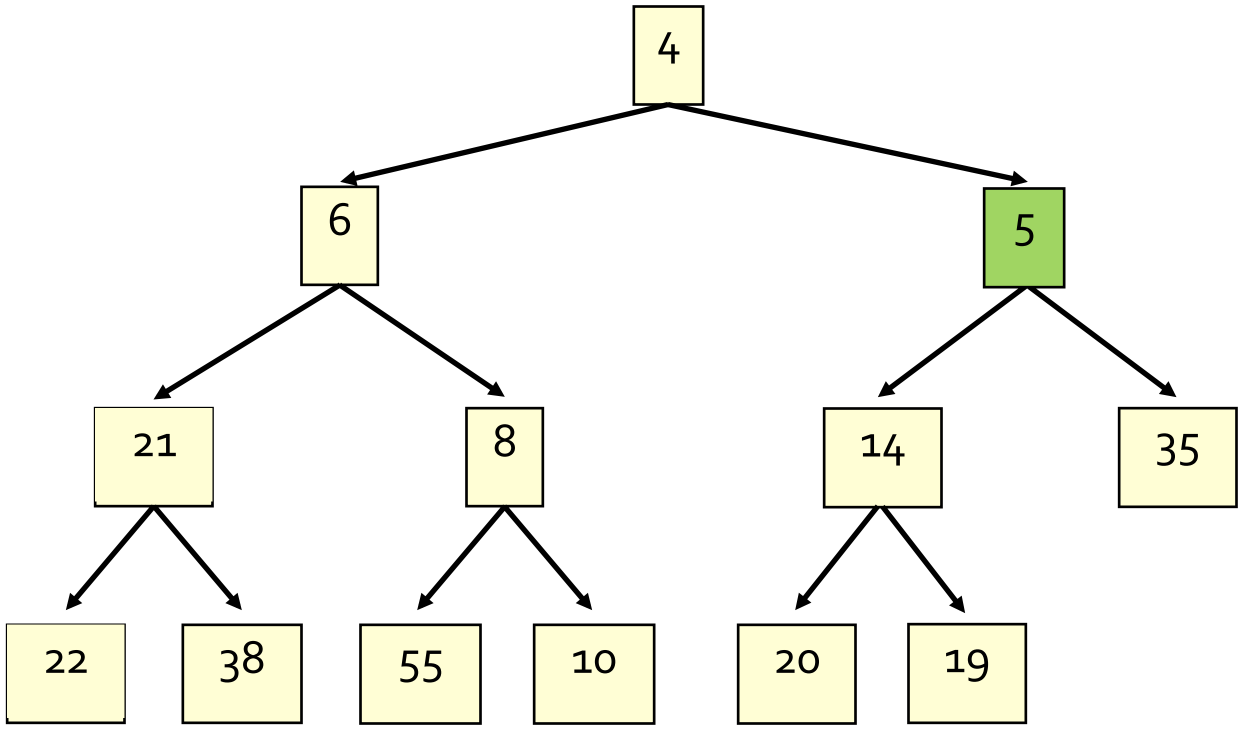
Heap insert()



Heap insert()



Heap insert()



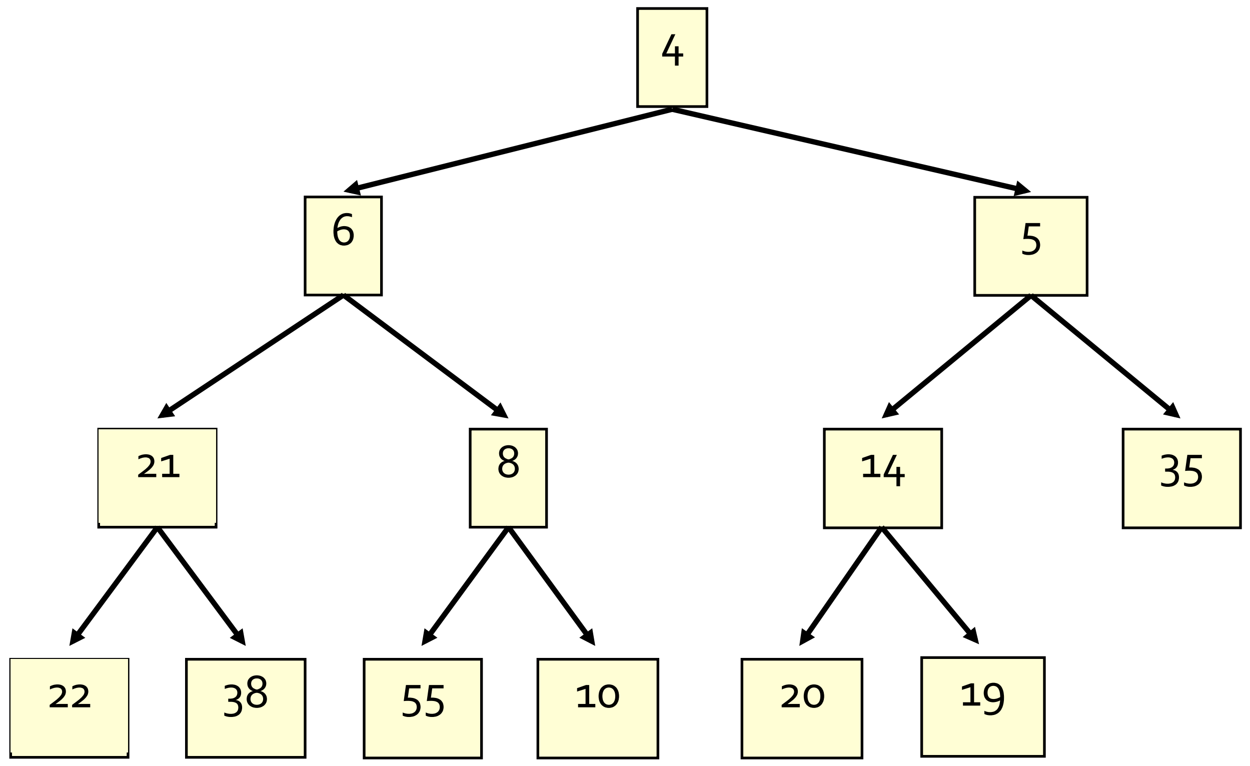
insert()

- Time is O(log n), since the tree is balanced
  - size of tree is exponential as a function of depth
  - depth of tree is logarithmic as a function of size

`extract()`

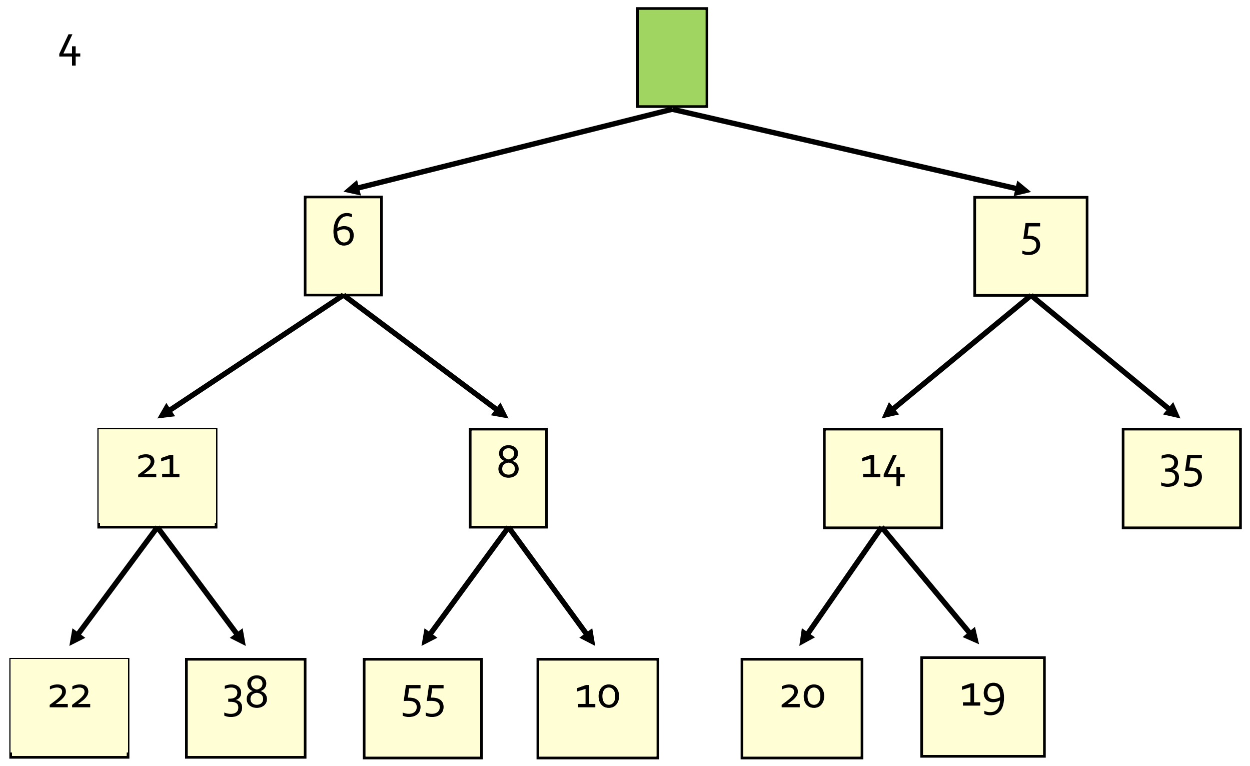
- Remove the least element – it is at the root
- This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place
- The heap invariant is maintained!

extract()



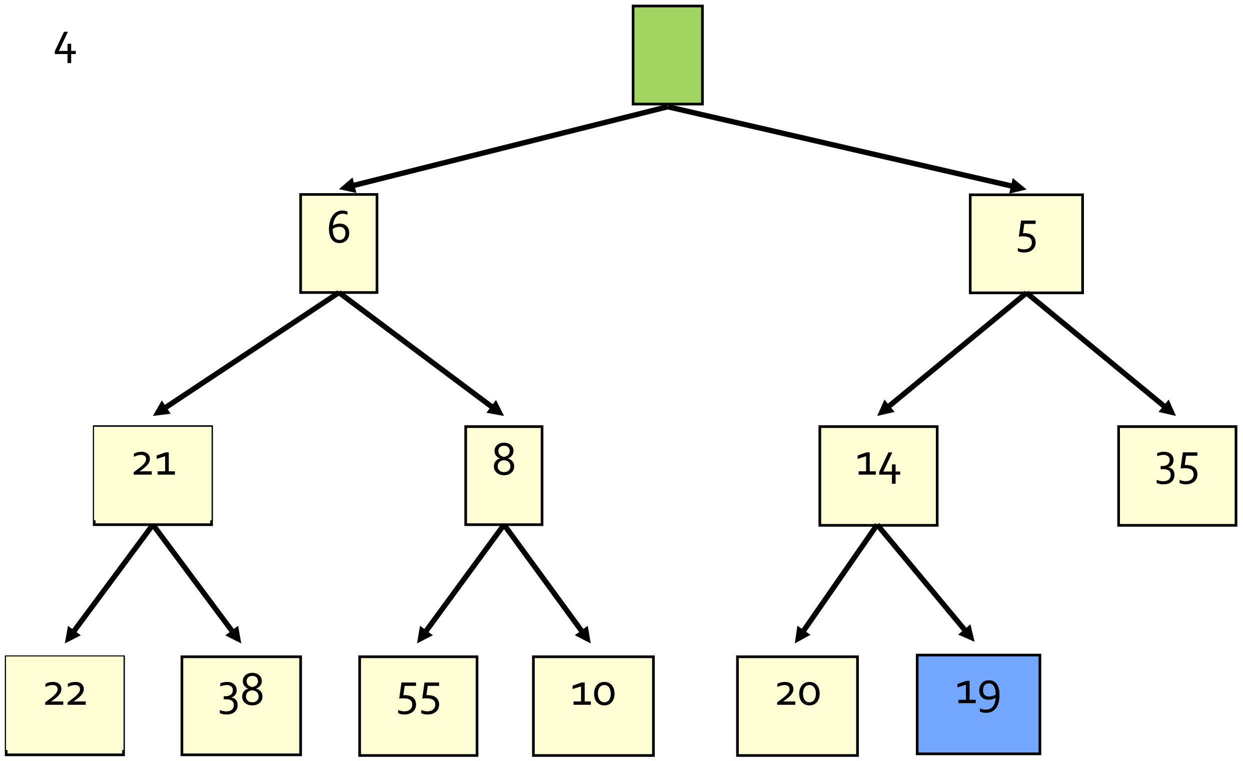
extract()

4



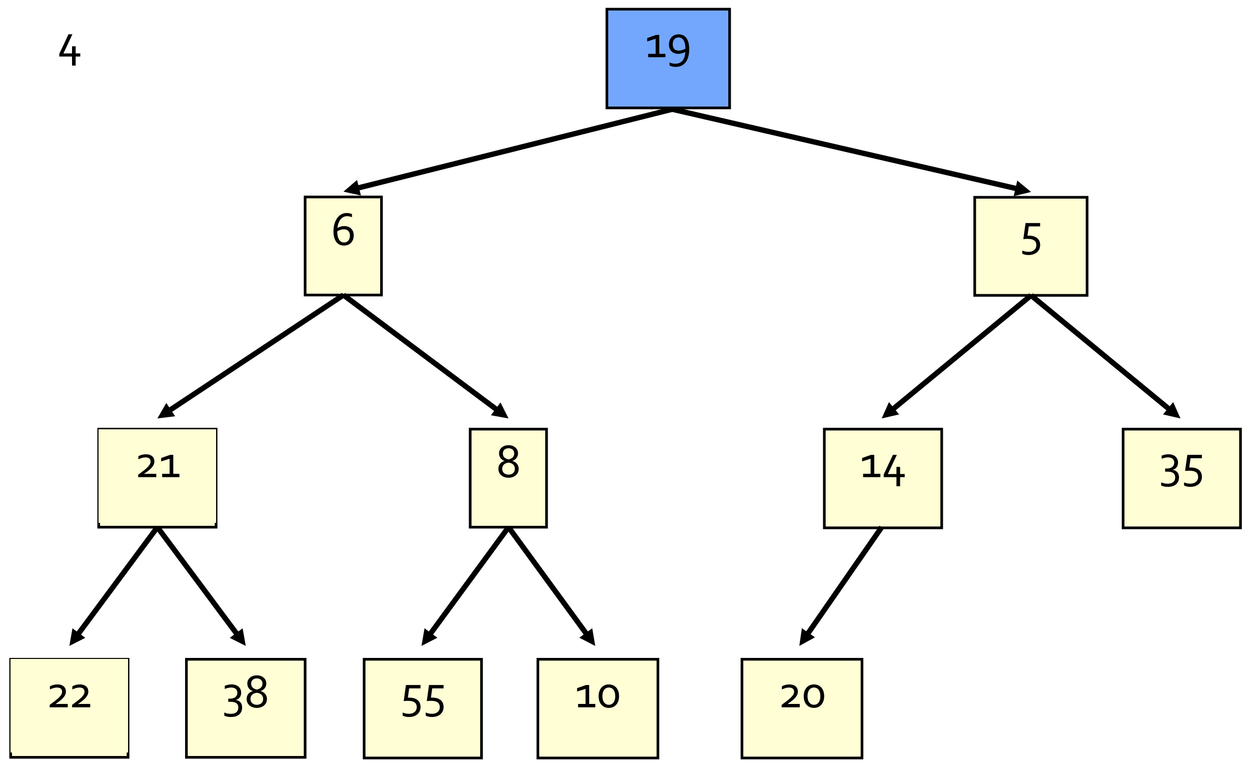
`extract()`

4



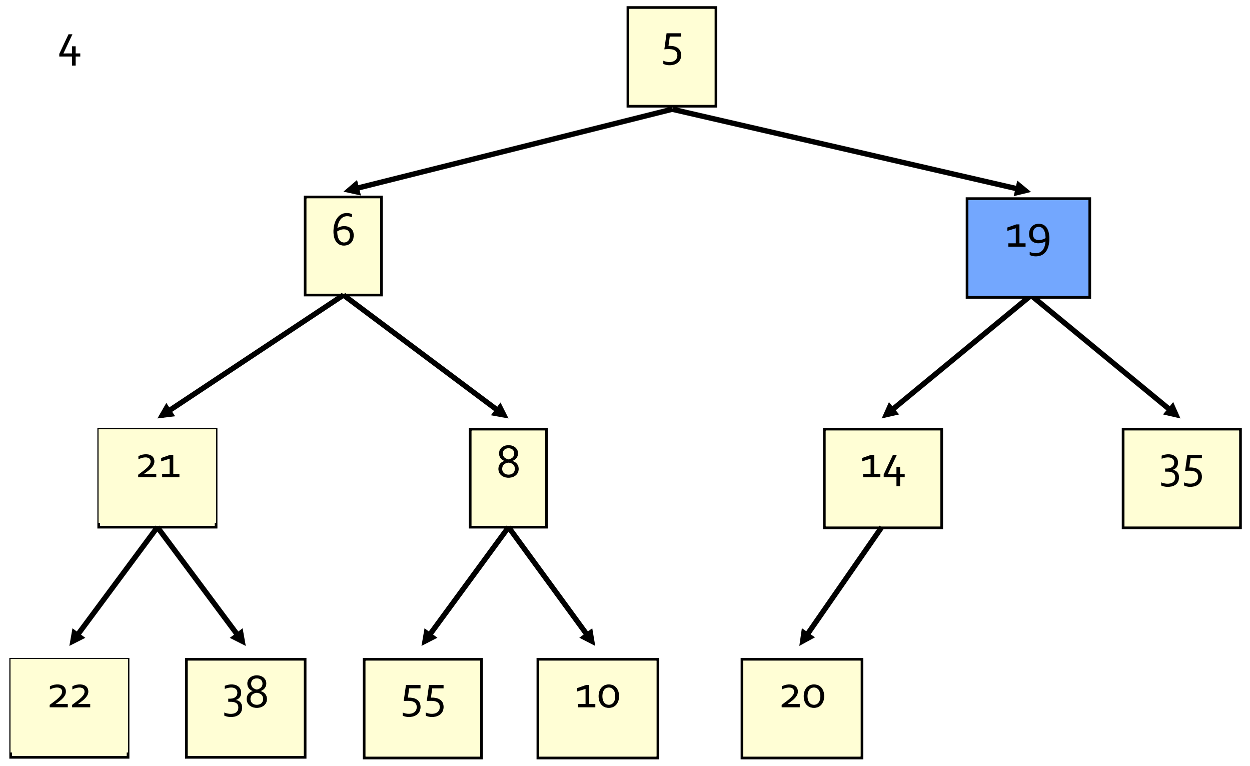
`extract()`

4



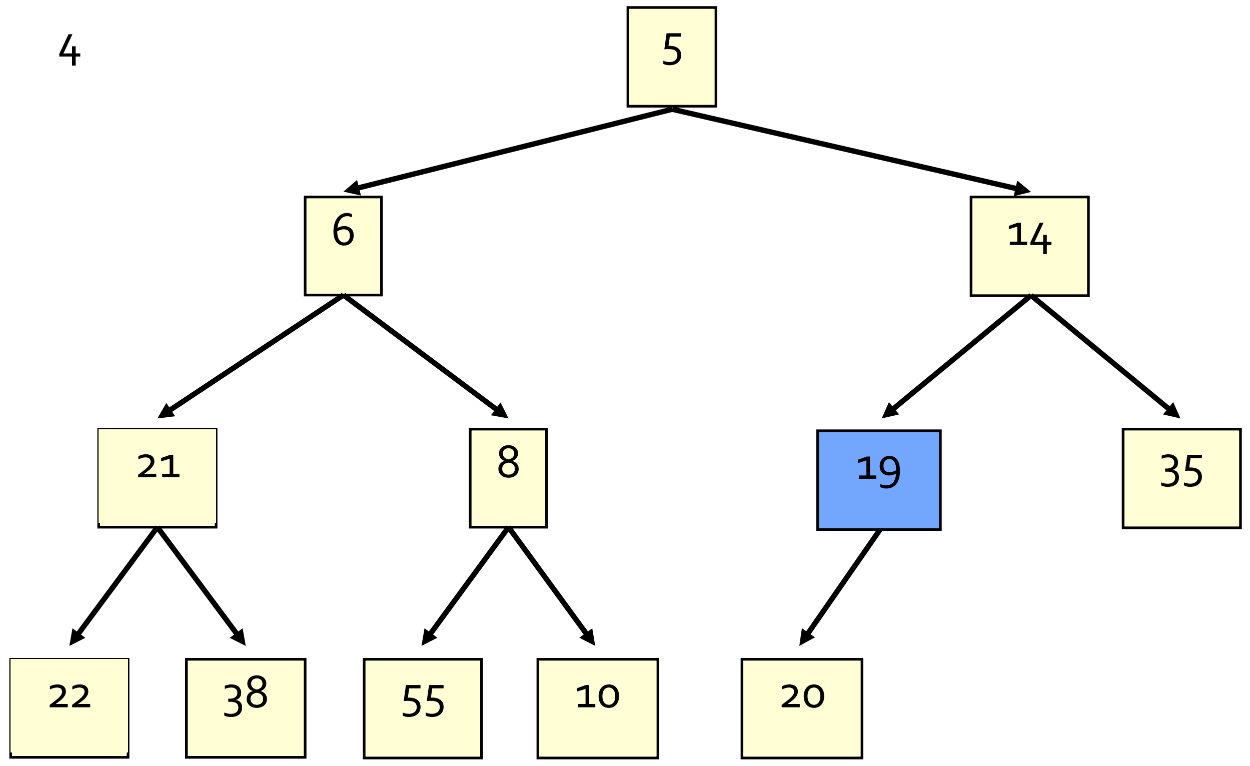
extract()

4



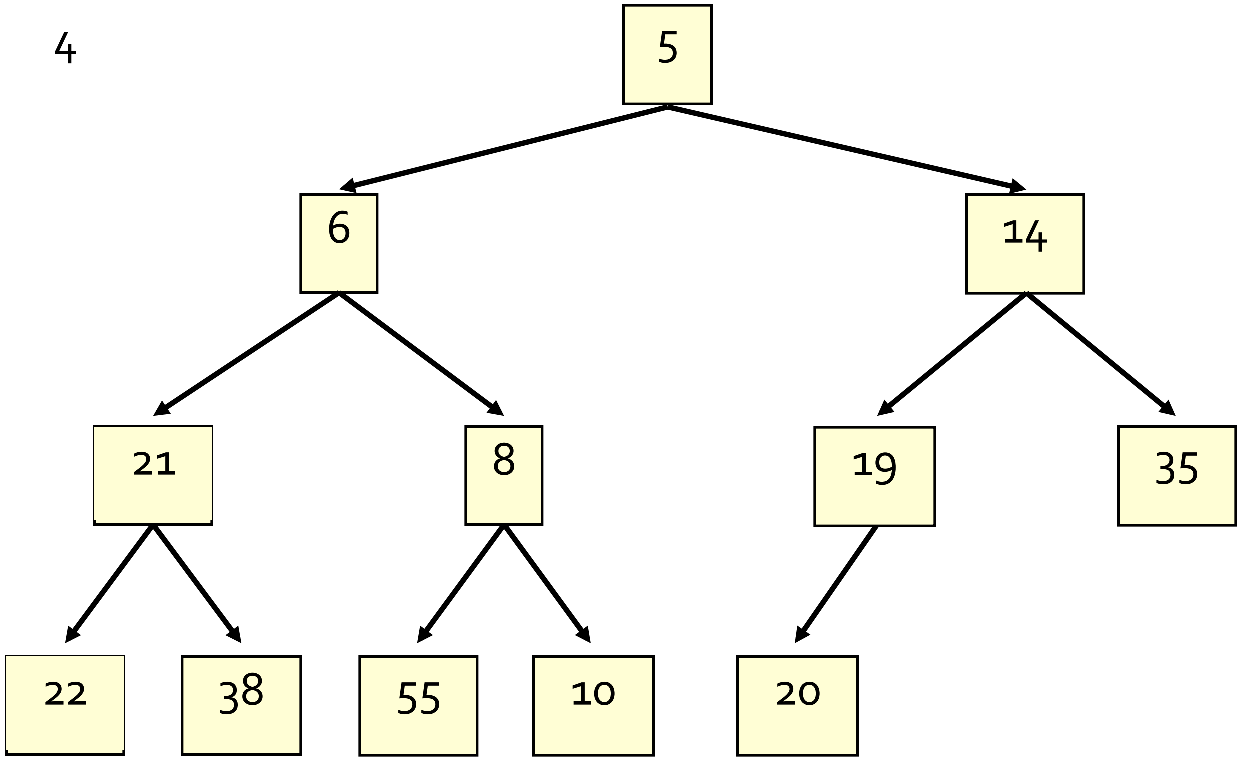
extract()

4



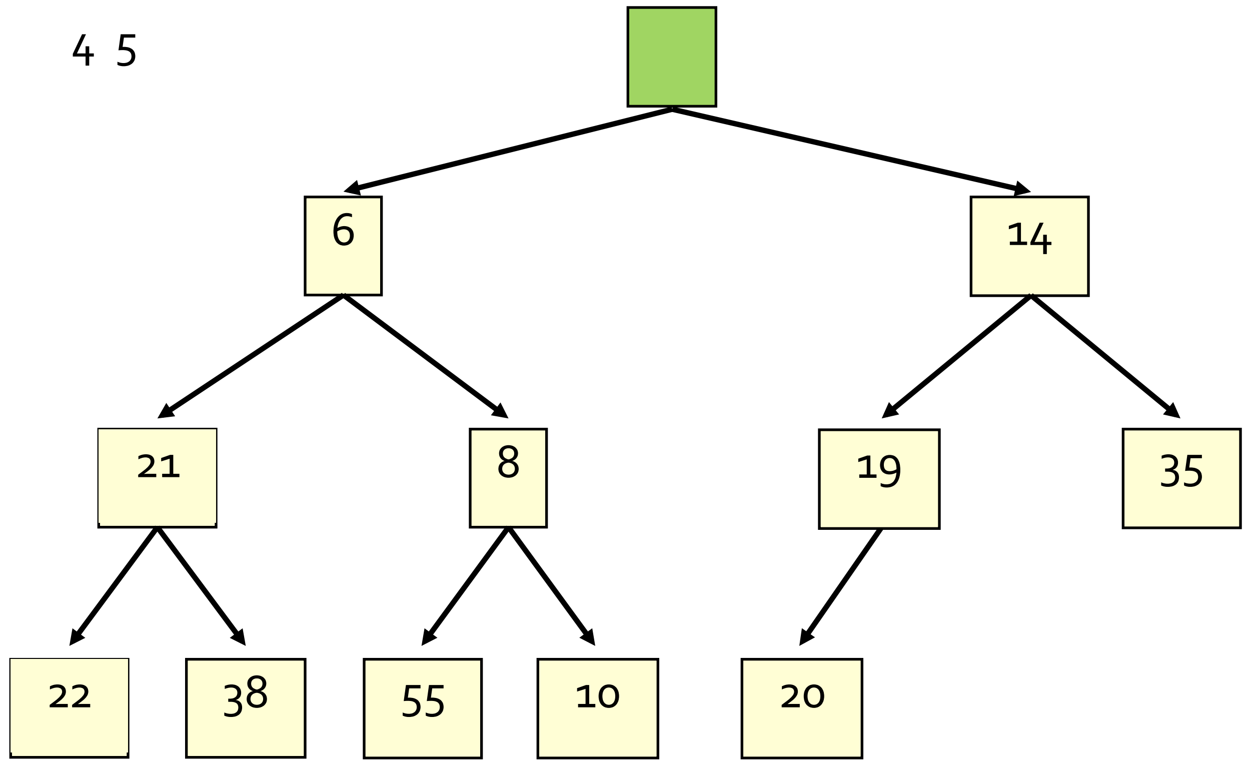
extract()

4



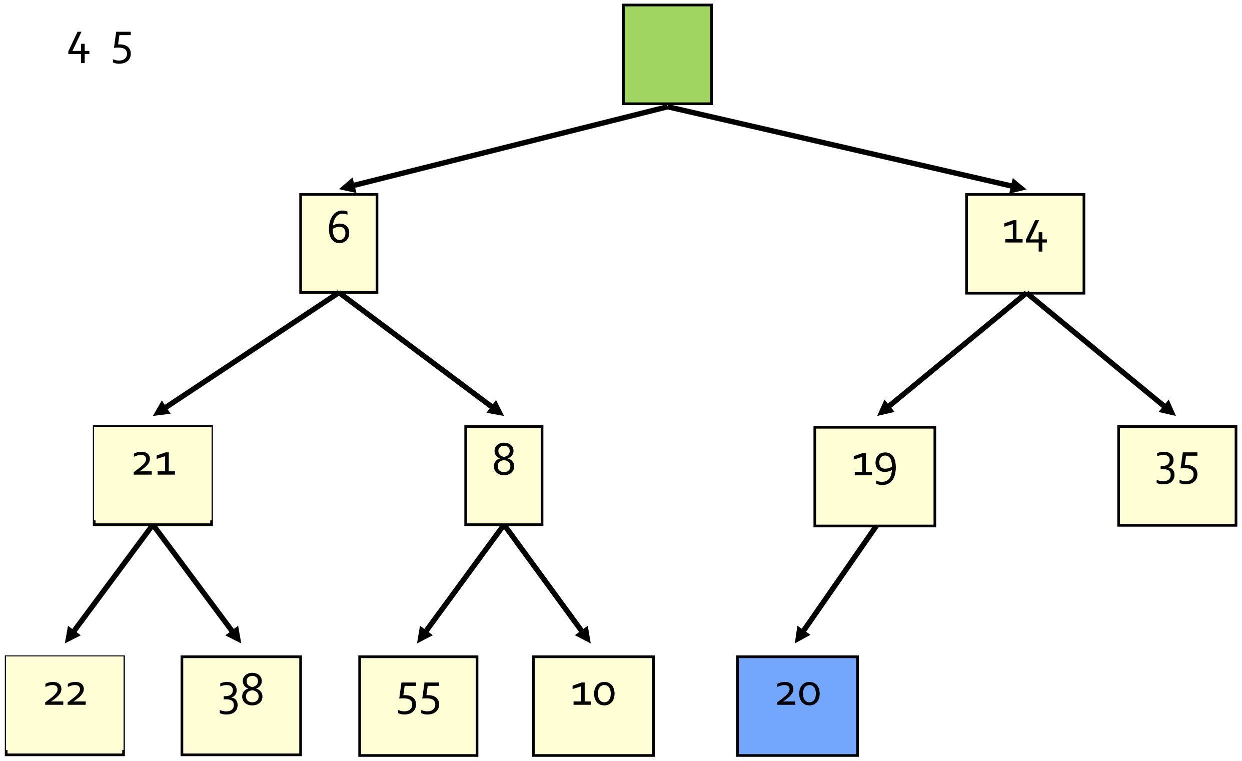
extract()

4 5



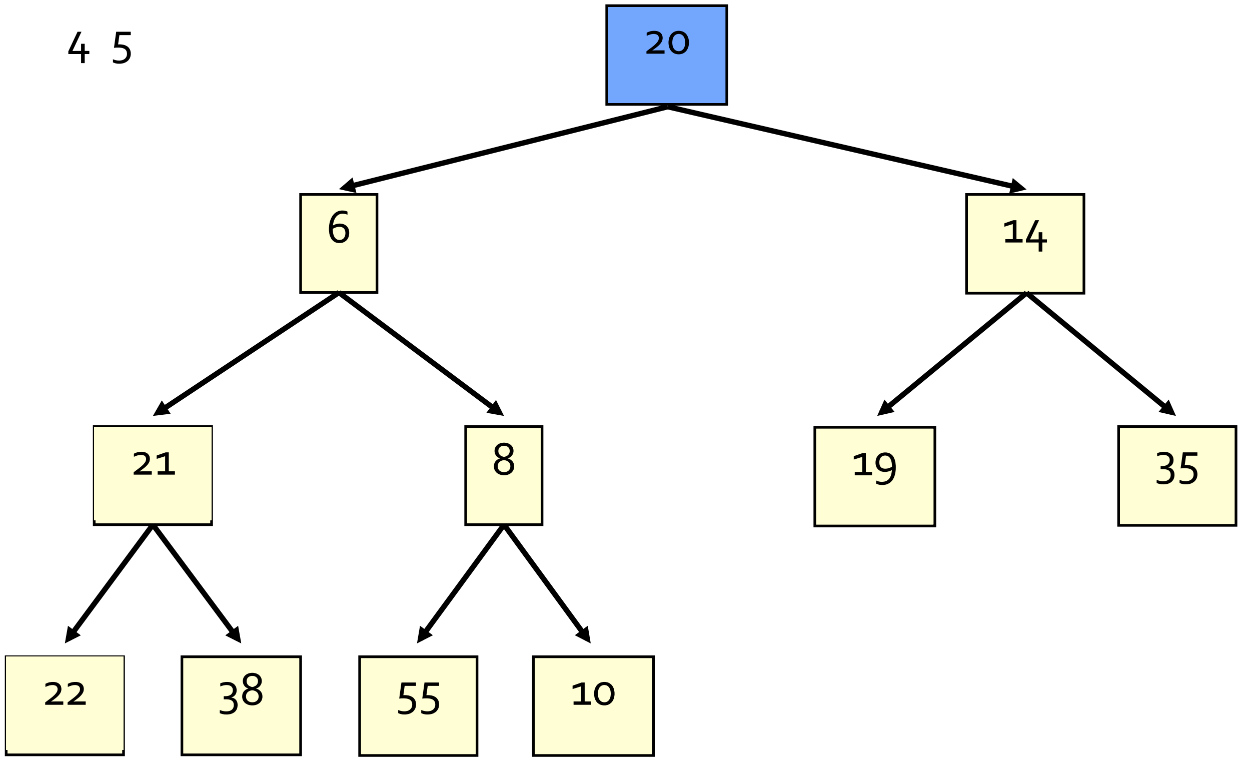
extract()

4 5



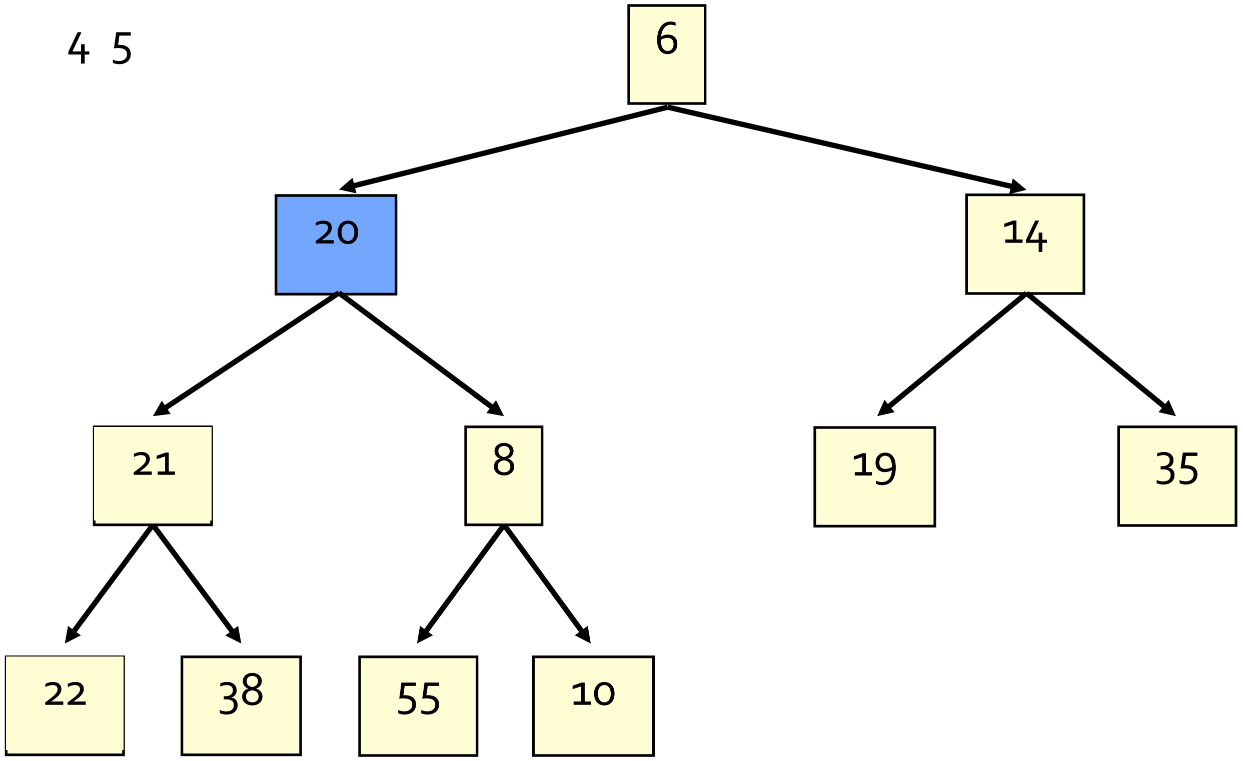
extract()

4 5



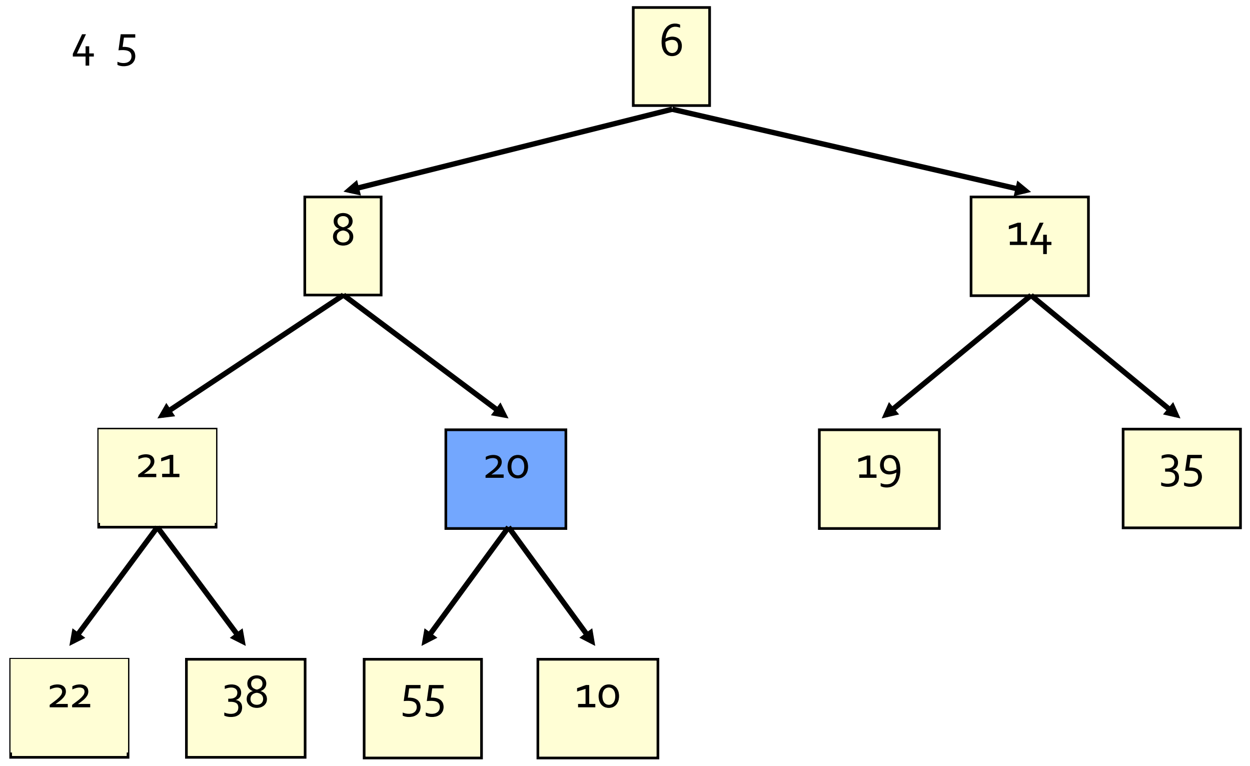
extract()

4 5



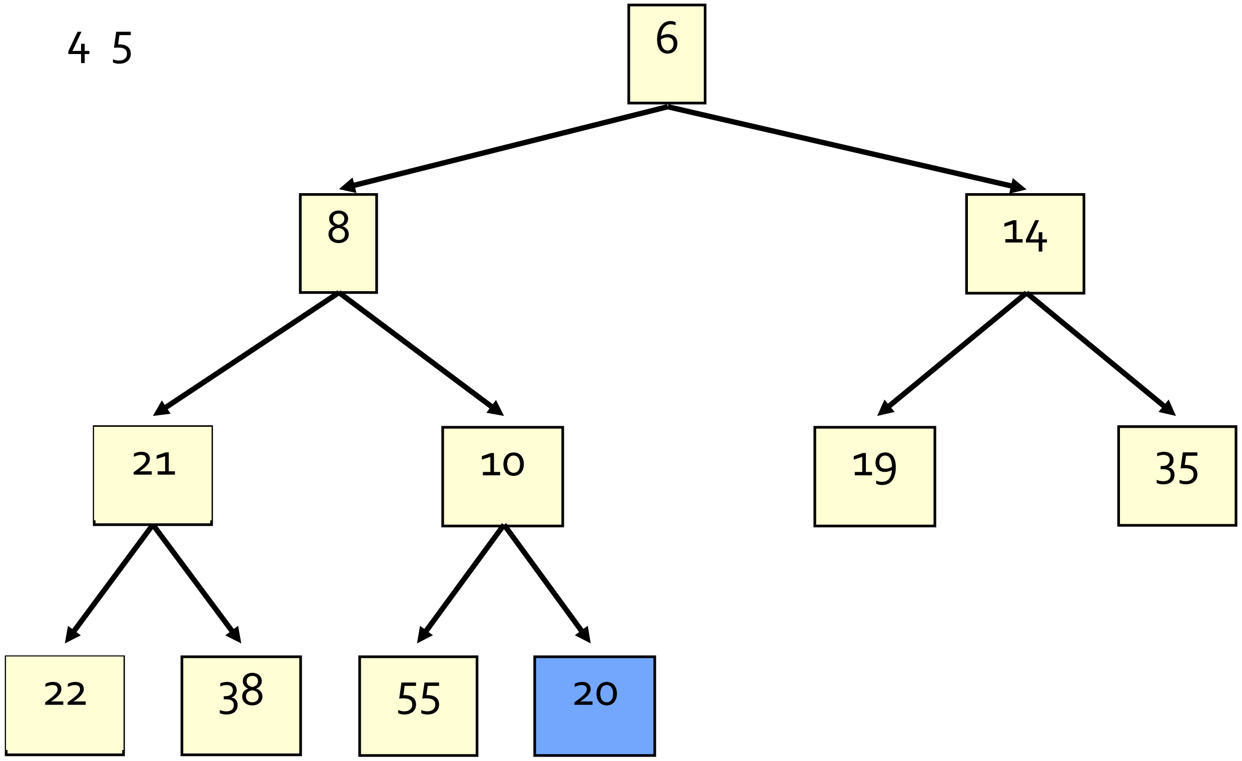
extract()

4 5



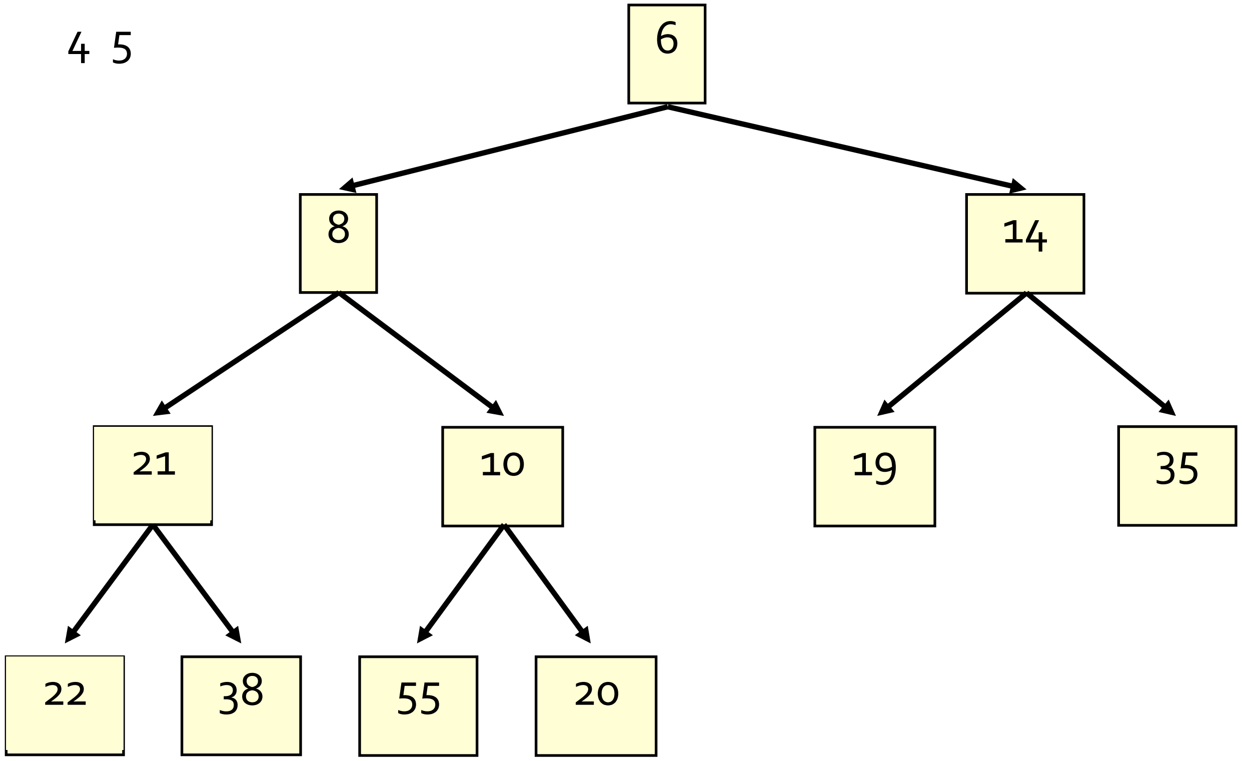
extract()

4 5



extract()

4 5



extract()

- Time is O(log n), since the tree is balanced

Store in an ArrayList or Vector

- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom
- The children of the node at array index n are found at 2n + 1 and 2n + 2
- The parent of node n is found at (n – 1)/2

Sets

- ADT Set
  - Operations:
    - `void insert(Object element);`
    - `boolean contains(Object element);`
    - `void remove(Object element);`
    - `int size();`
    - `iteration`
- No duplicates allowed
- Hash table implementation: O(1) *insert* and *contains*
- SortedSet tree implementation: O(log n) *insert* and *contains*

A set makes no promises about ordering, but you can still iterate over it.

Dictionaries

- ADT Dictionary (aka Map)
  - Operations:
    - `void insert(Object key, Object value);`
    - `void update(Object key, Object value);`
    - `Object find(Object key);`
    - `void remove(Object key);`
    - `boolean isEmpty();`
    - `void clear();`
- Think of: key = word; value = definition
- Where used:
  - Symbol tables
  - Wide use within other algorithms

A HashMap is a particular implementation of the Map interface

Dictionaries

- Hash table implementation:
  - Use a **hash function** to compute hashes of keys
  - Store values in an array, indexed by key hash
  - A **collision** occurs when two keys have the same hash
  - How to handle collisions?
    - Store another data structure, such as a linked list, in the array location for each key
    - Put (key, value) pairs into that data structure
  - insert and find are O(1) when there are no collisions
    - Expected complexity
  - Worst case, every hash is a collision
    - Complexity for insert and find comes from the tertiary data structure's complexity, e.g., O(n) for a linked list

A HashMap is a particular implementation of the Map interface

Graphs

- Set of vertices (or nodes) V, set of edges E
- Number of vertices **n = |V|**
- Number of edges **m = |E|**
  - Upper bound $O(n^2)$ on number of edges
    - A **complete** graph has $m = n(n-1)/2$
- Directed or undirected
  - Directed edges have distinct **head** and **tail**
- Weighted edges
- Cycles and paths
- Connected components
- DAGs
- Degree of a node (in- and out- degree for directed graphs)

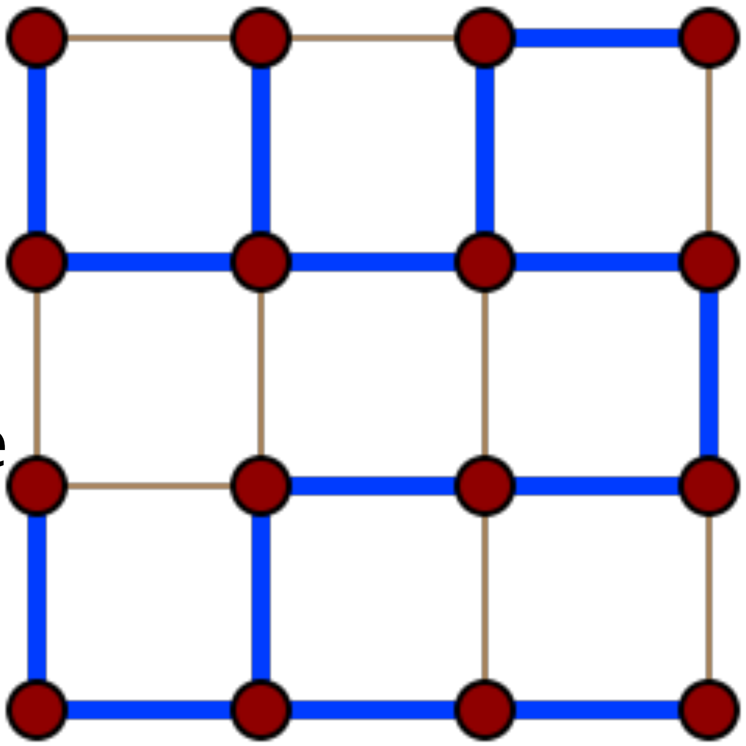
Graph Representation

- You should be able to write a *Vertex* class in Java and implement standard graph algorithms using this class
- **However,** understanding the algorithms is much more important than memorizing their code

Spanning Trees

- A **spanning tree** is a subgraph of an undirected graph that:
  - Is a tree
  - Contains every vertex in the graph
- Number of edges in a tree

m = n-1



Minimum Spanning Trees (MST)

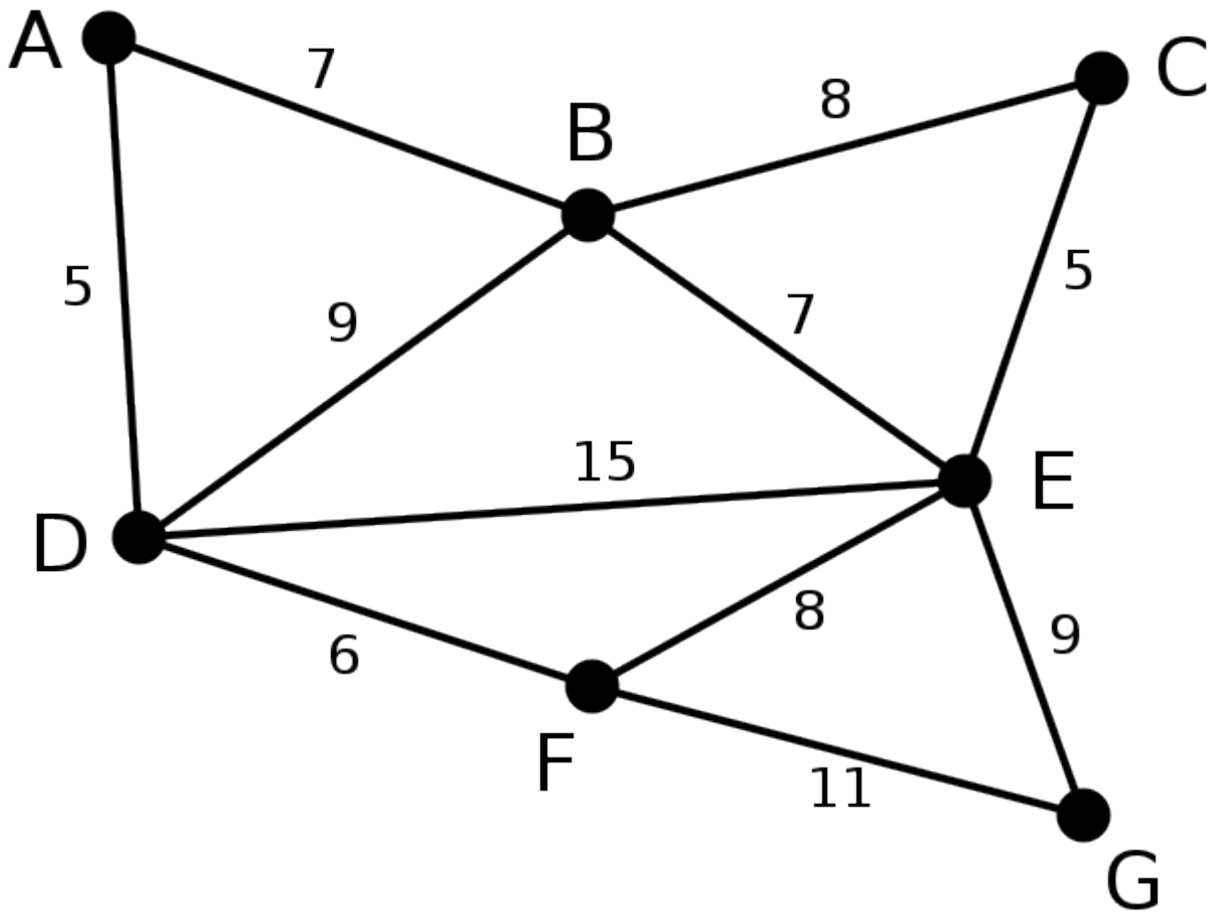
- Spanning tree with minimum sum edge weights
  - Prim’s algorithm
  - Kruskal’s algorithm
  - Not necessarily unique

Prim's algorithm

- Graph search algorithm, builds up a spanning tree from one root vertex
- Like BFS, but it uses a priority queue
  - Priority is the weight of the edge to the vertex
  - Also need to keep track of which edge we used
- Always picks smallest edge to an unvisited vertex
- Runtime is O(m log m)
  - O(m) Priority Queue operations at log(m) each

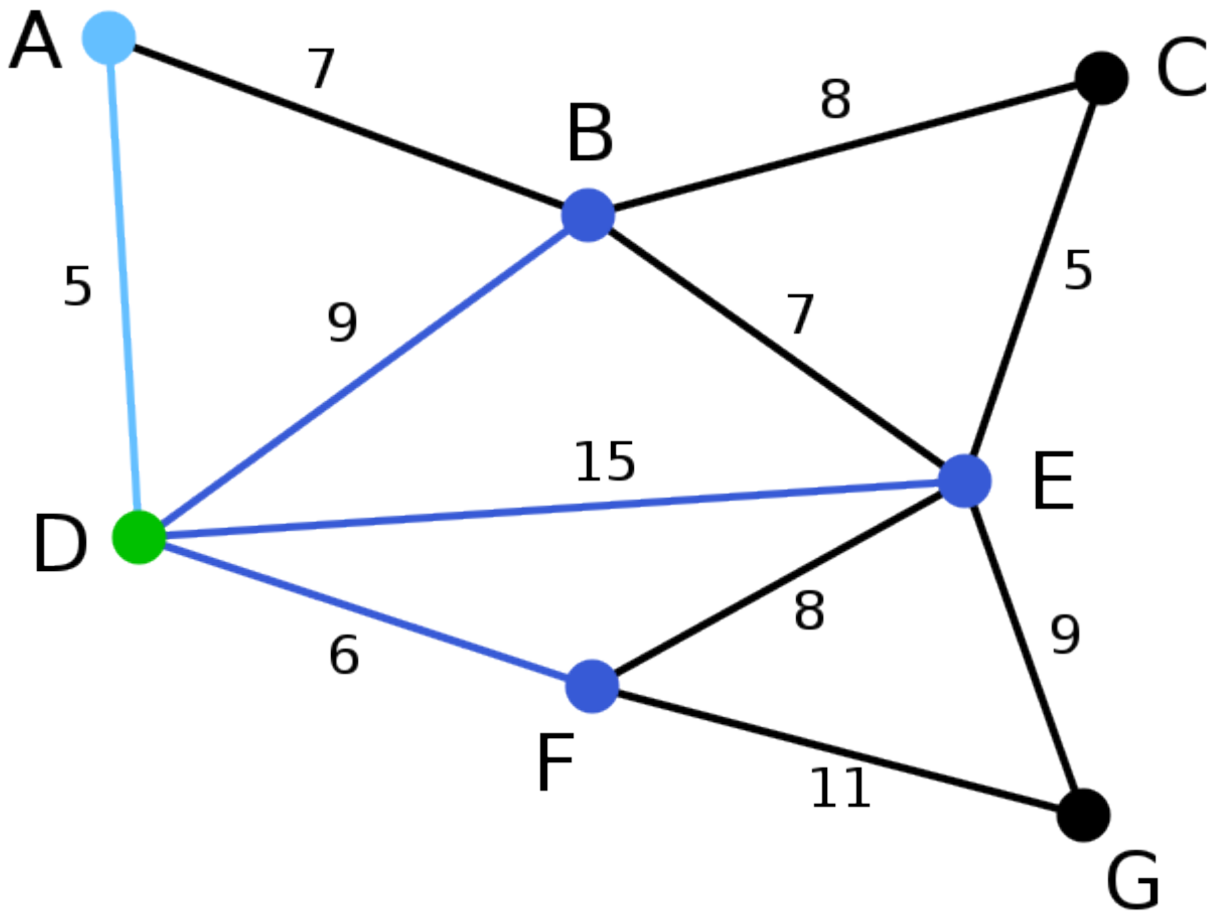
Prim's Algorithm Example

This is our original weighted graph. The numbers near the edges indicate their weight.



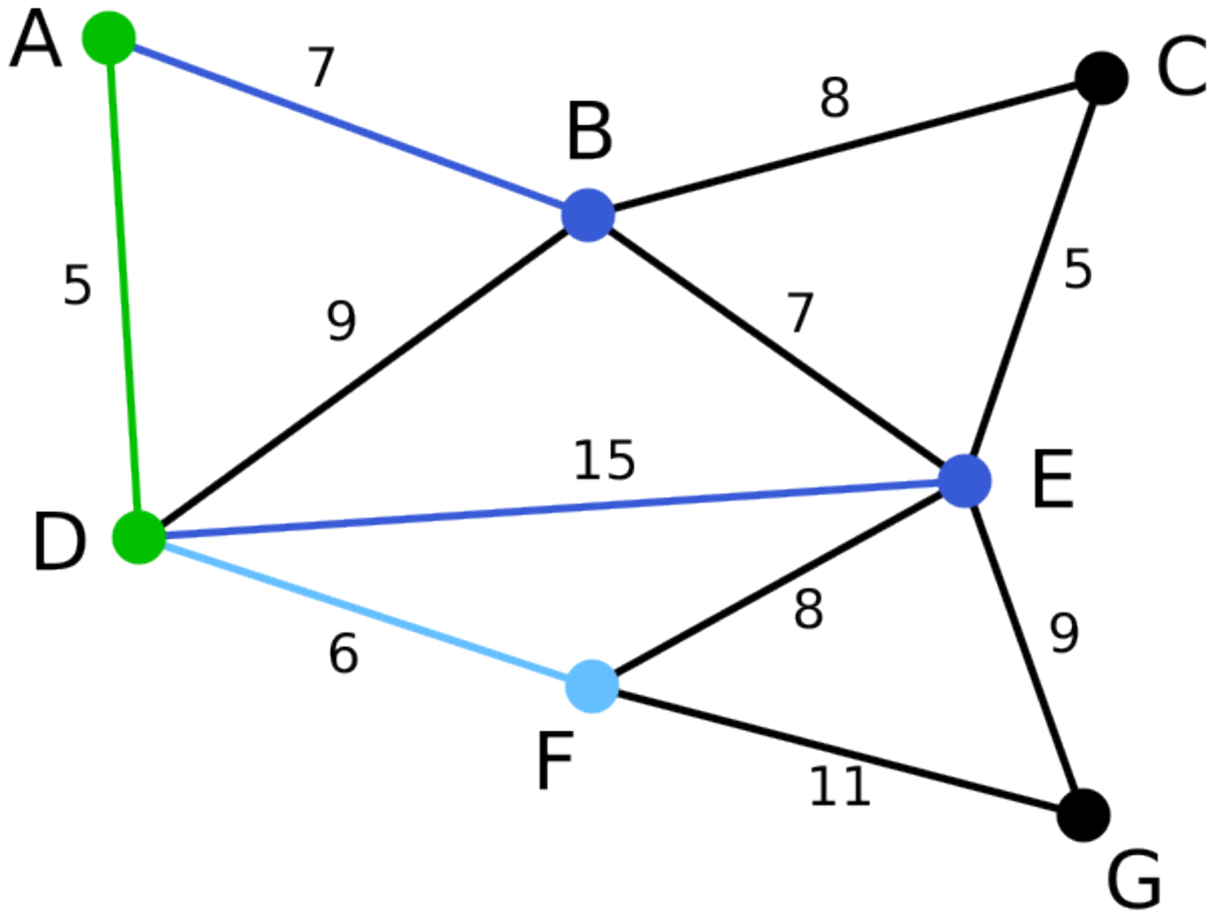
Prim's Algorithm Example

Vertex D has been arbitrarily chosen as a starting point. Vertices A, B, E and F are connected to D through a single edge. A is the vertex nearest to D and will be chosen as the second vertex along with the edge AD.



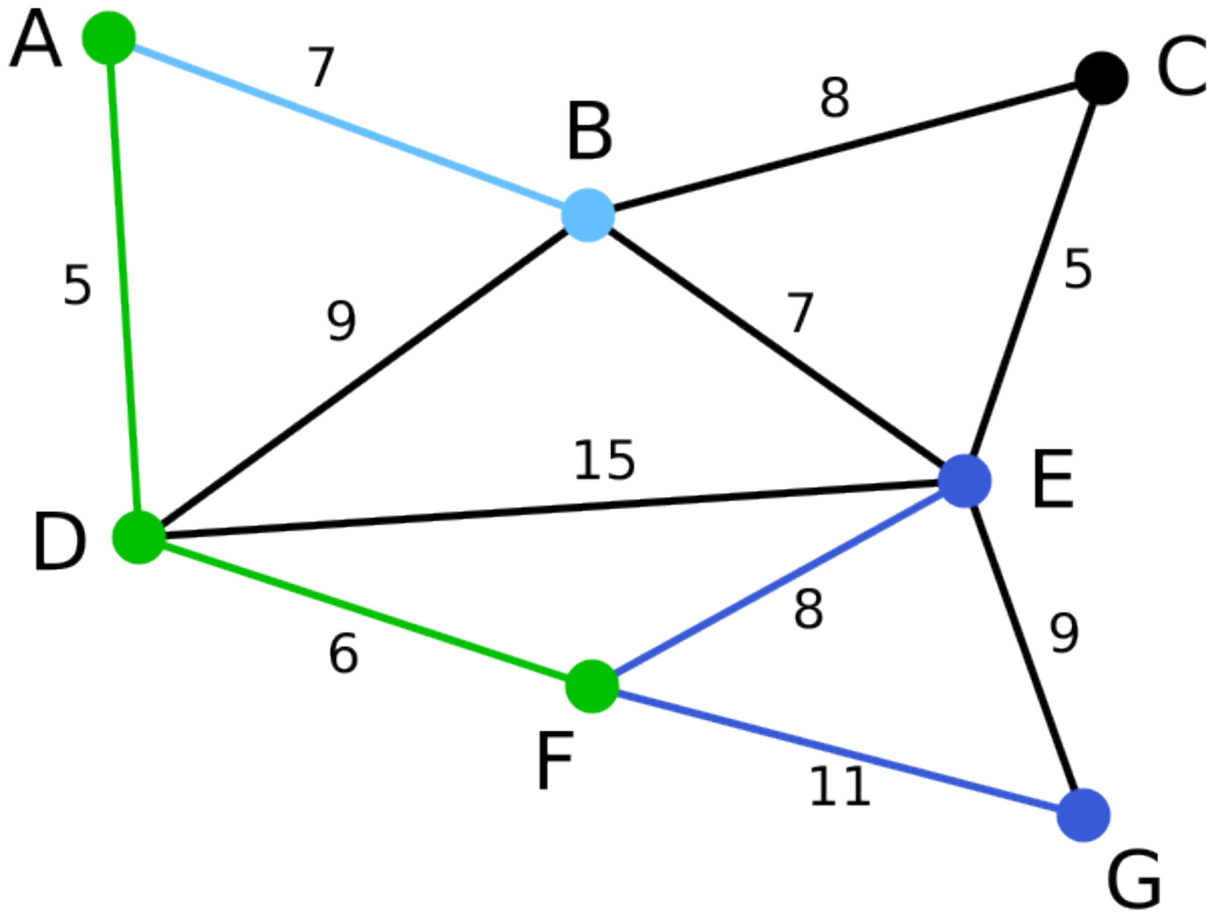
Prim's Algorithm Example

The next vertex chosen is the vertex nearest to either D or A. B is 9 away from D and 7 away from A, E is 15, and F is 6. F is the smallest distance away, so we highlight the vertex F and the arc DF.



Prim's Algorithm Example

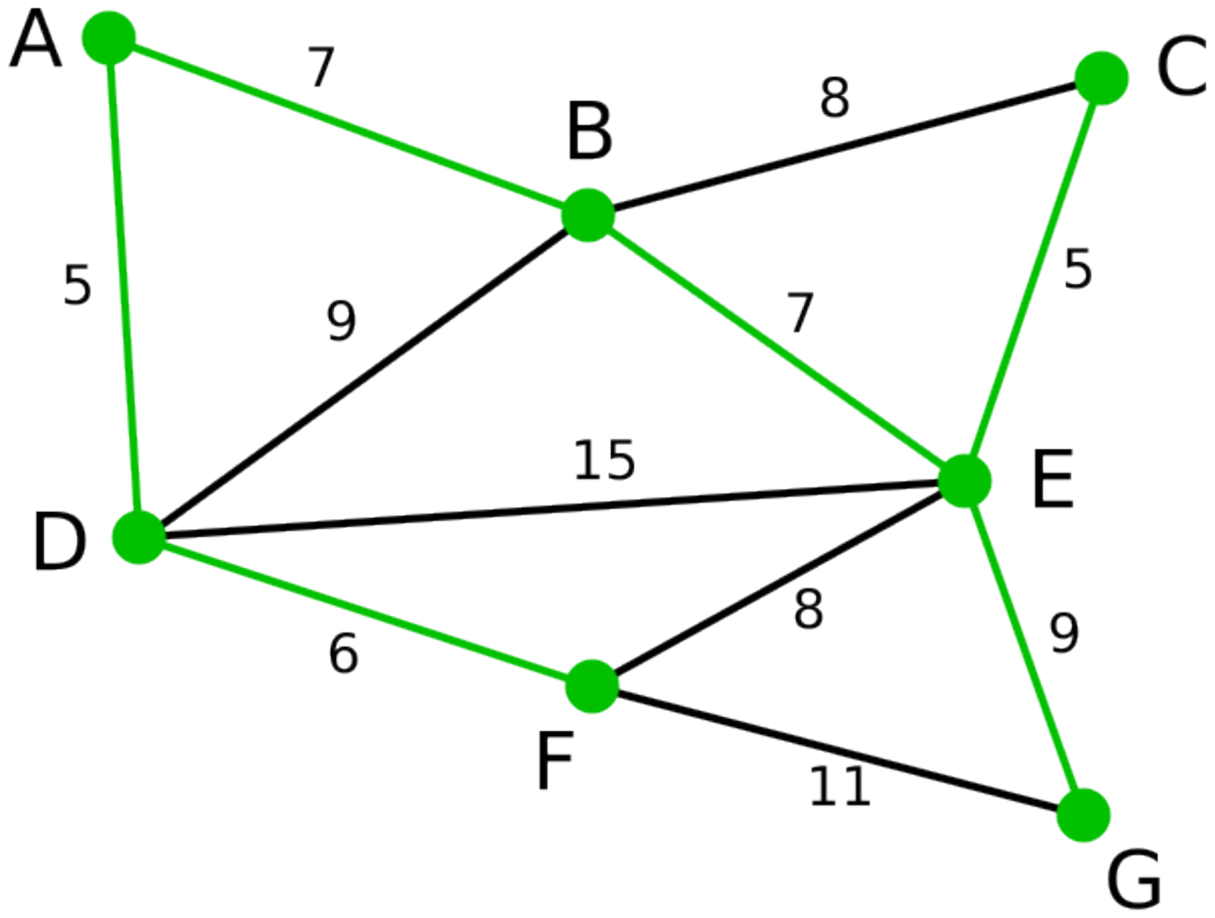
The algorithm carries on as above. Vertex B, which is 7 away from A, is highlighted.



Prim's Algorithm Example

End Result

Notice how each vertex has at least 1 edge connecting to it and that the edge is the least of the edges connected to the vertex.



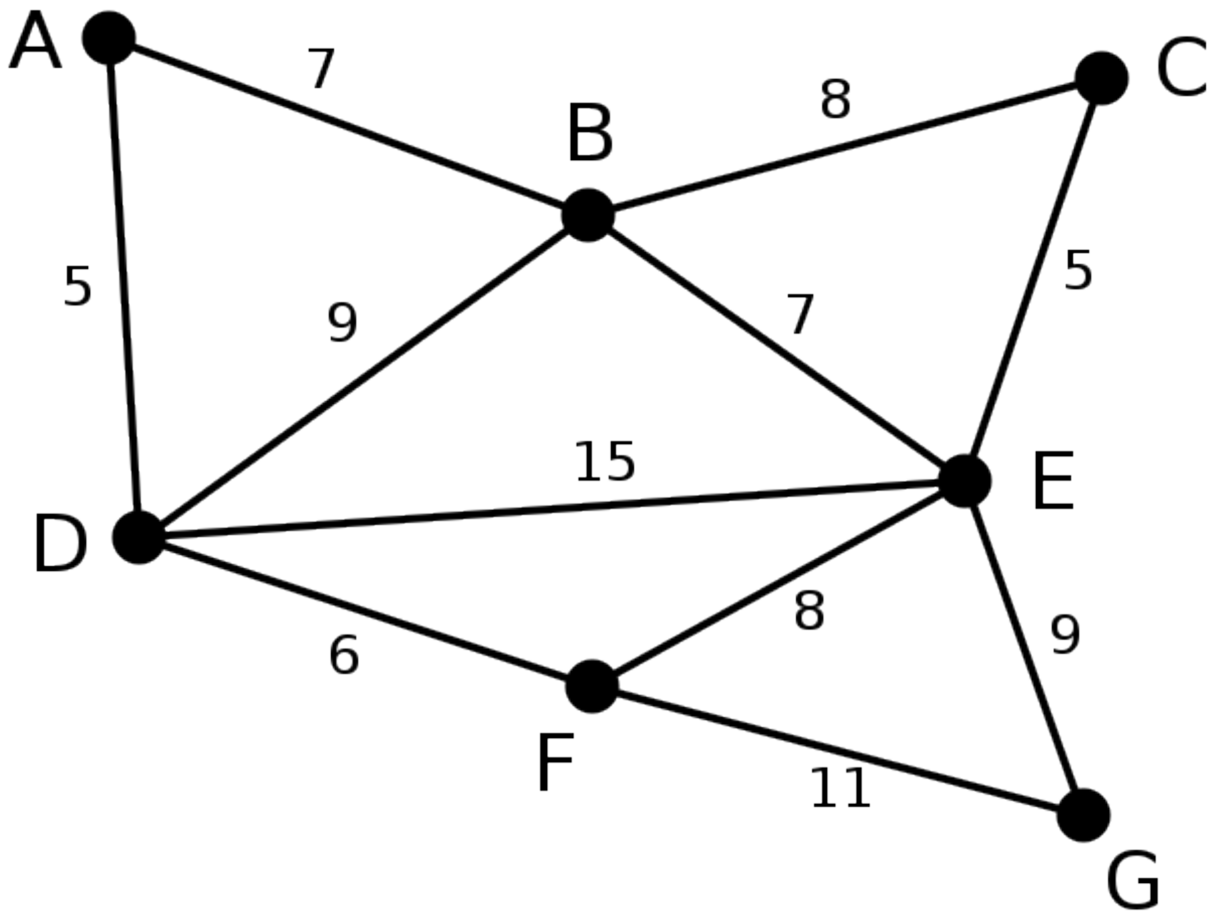
Kruskal’s Algorithm

- Idea: Find MST by connecting forest components using shortest edges
  - Process edges from least to greatest
  - Initially, every node is its own component
  - Either an edge connects two different components or it connects a component to itself
    - Add an edge only in the former case
  - Picks smallest edge between two components
  - O(m log m) time to sort the edges
    - Also need the union-find structure to keep track of components, but it does not change the running time

Kruskal's Algorithm Example

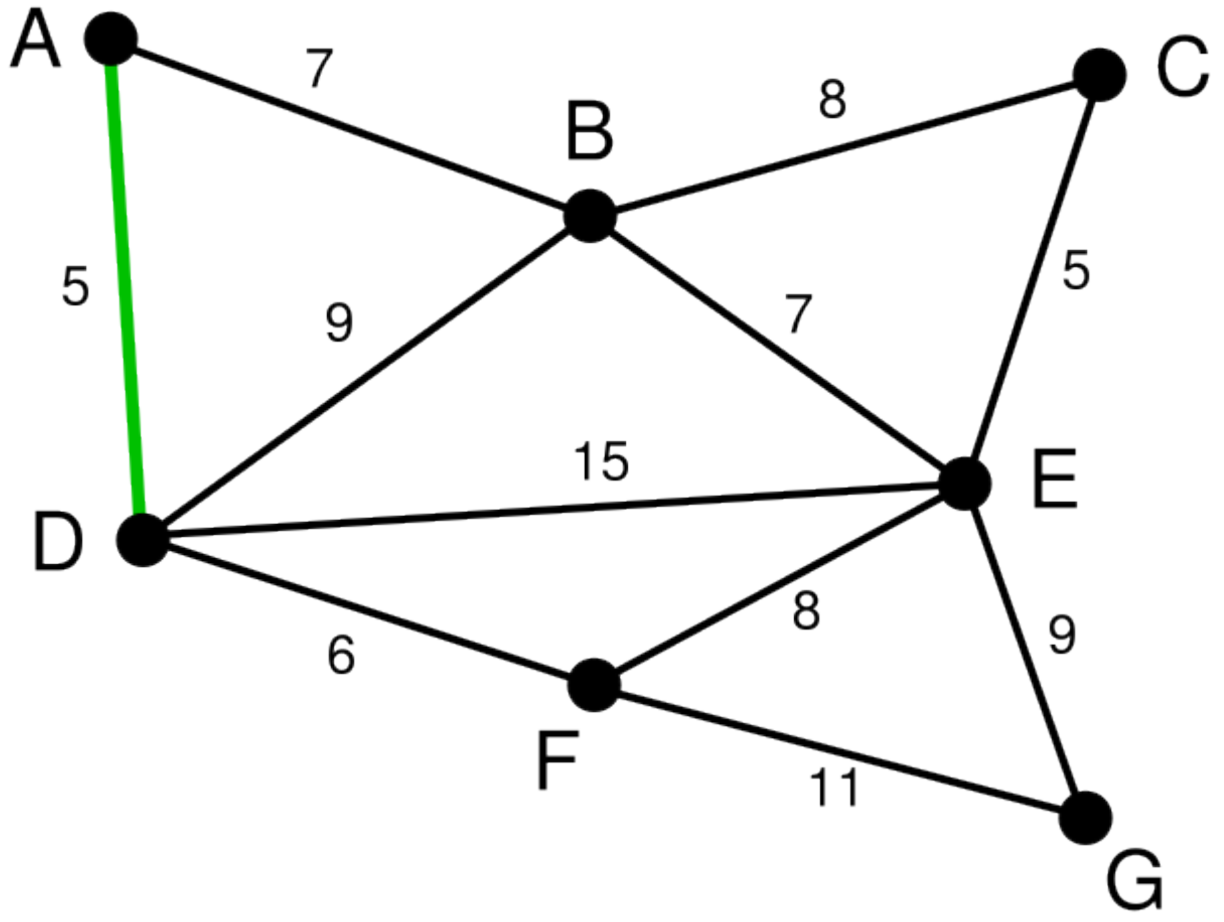
This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

∞



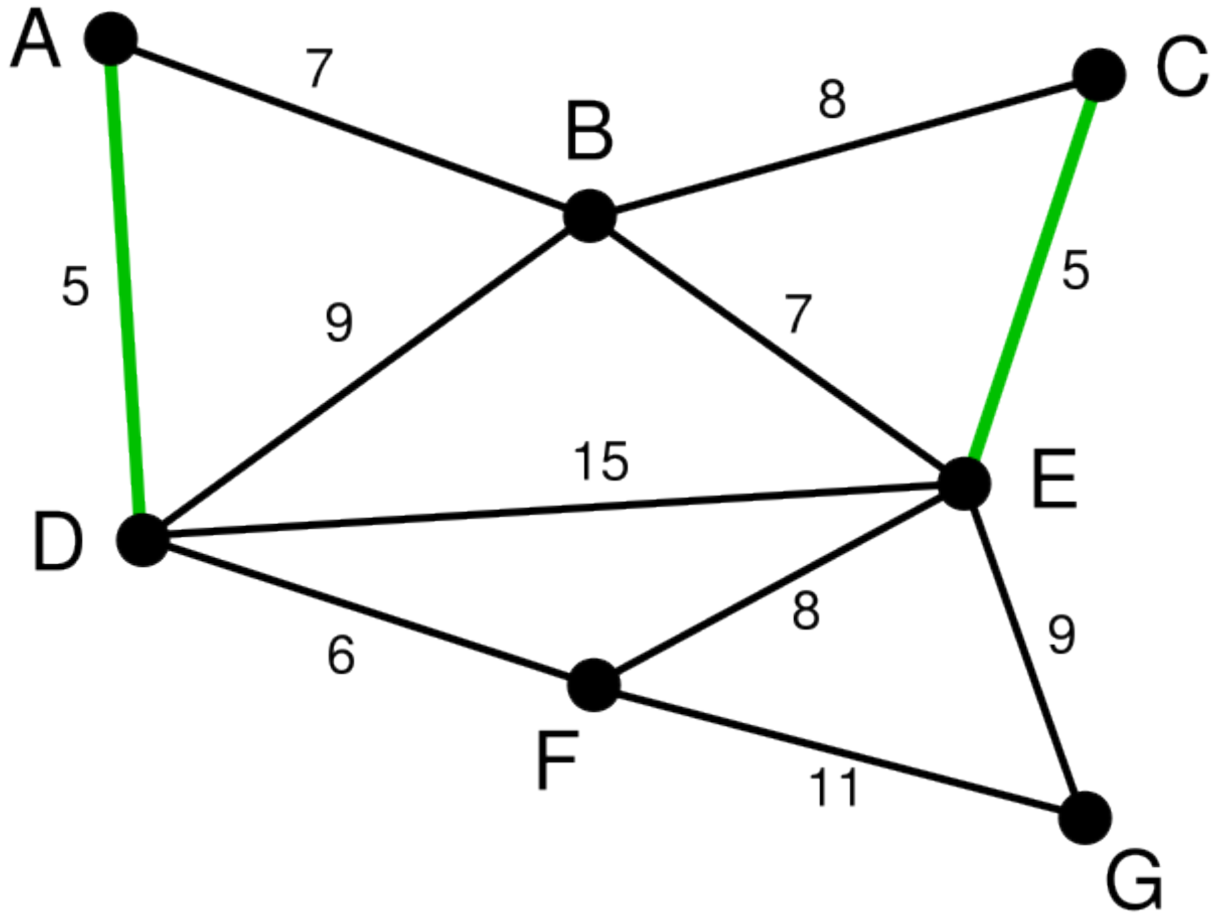
Kruskal's Algorithm Example

AD and CE are the shortest arcs, with length 5, and AD has been arbitrarily chosen, so it is highlighted.



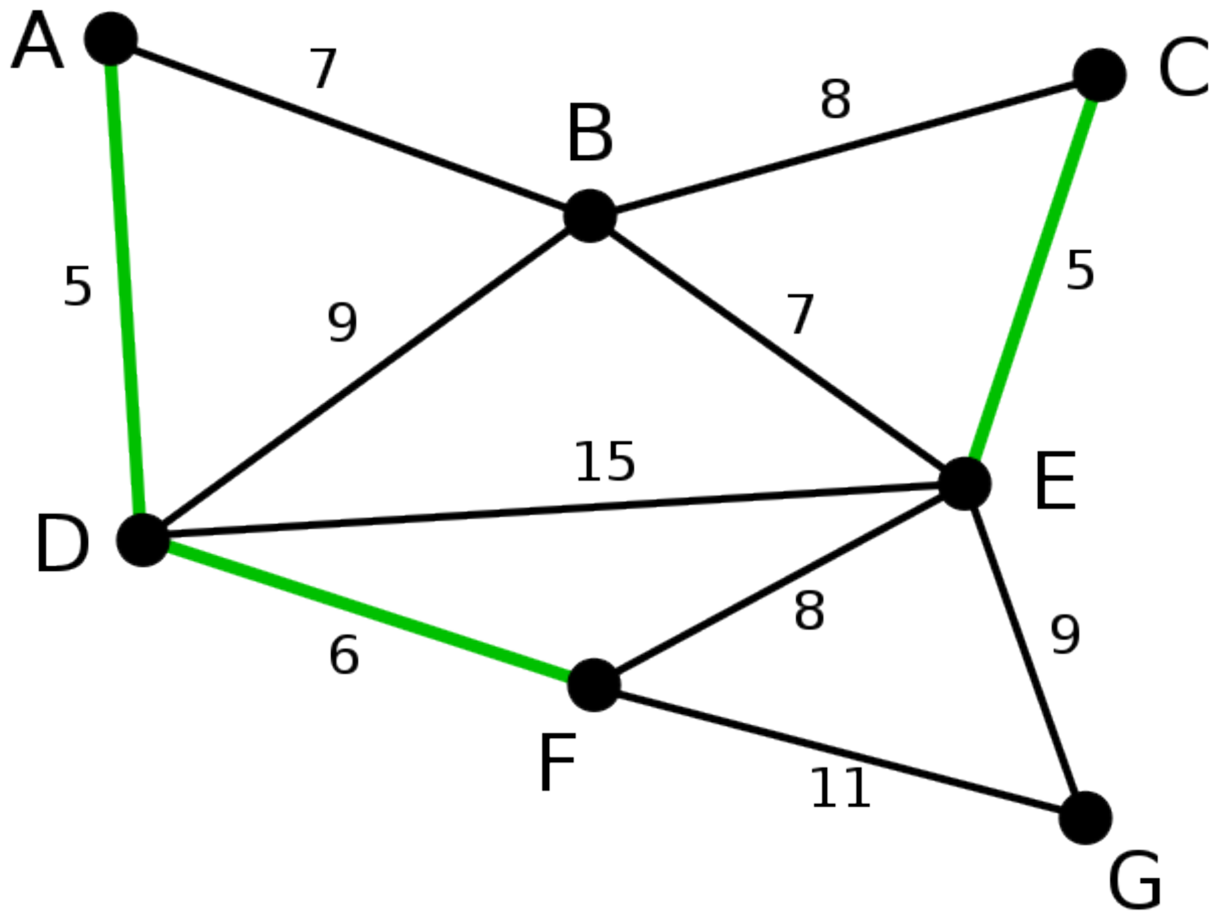
Kruskal's Algorithm Example

CE is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.



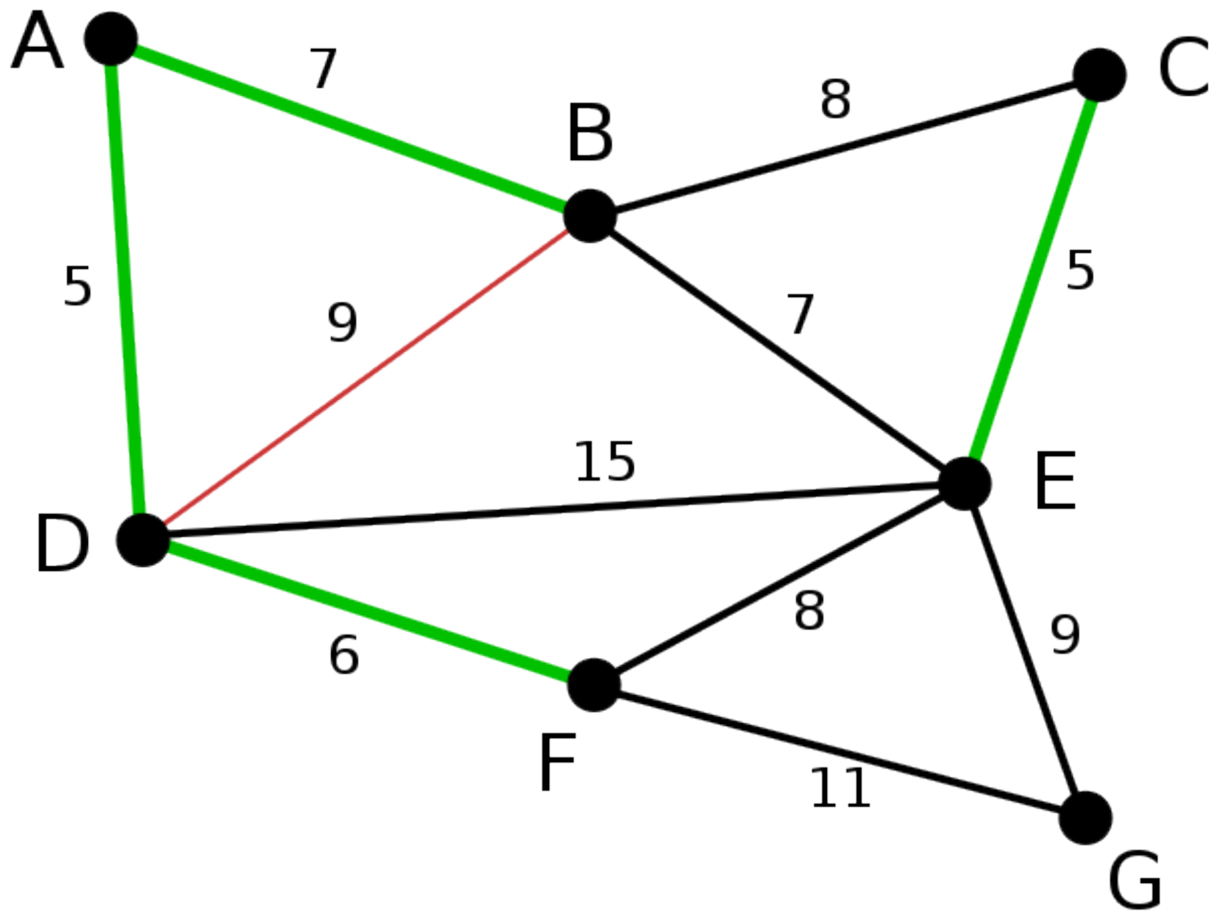
Kruskal's Algorithm Example

The next arc, DF with length 6, is highlighted using much the same method.



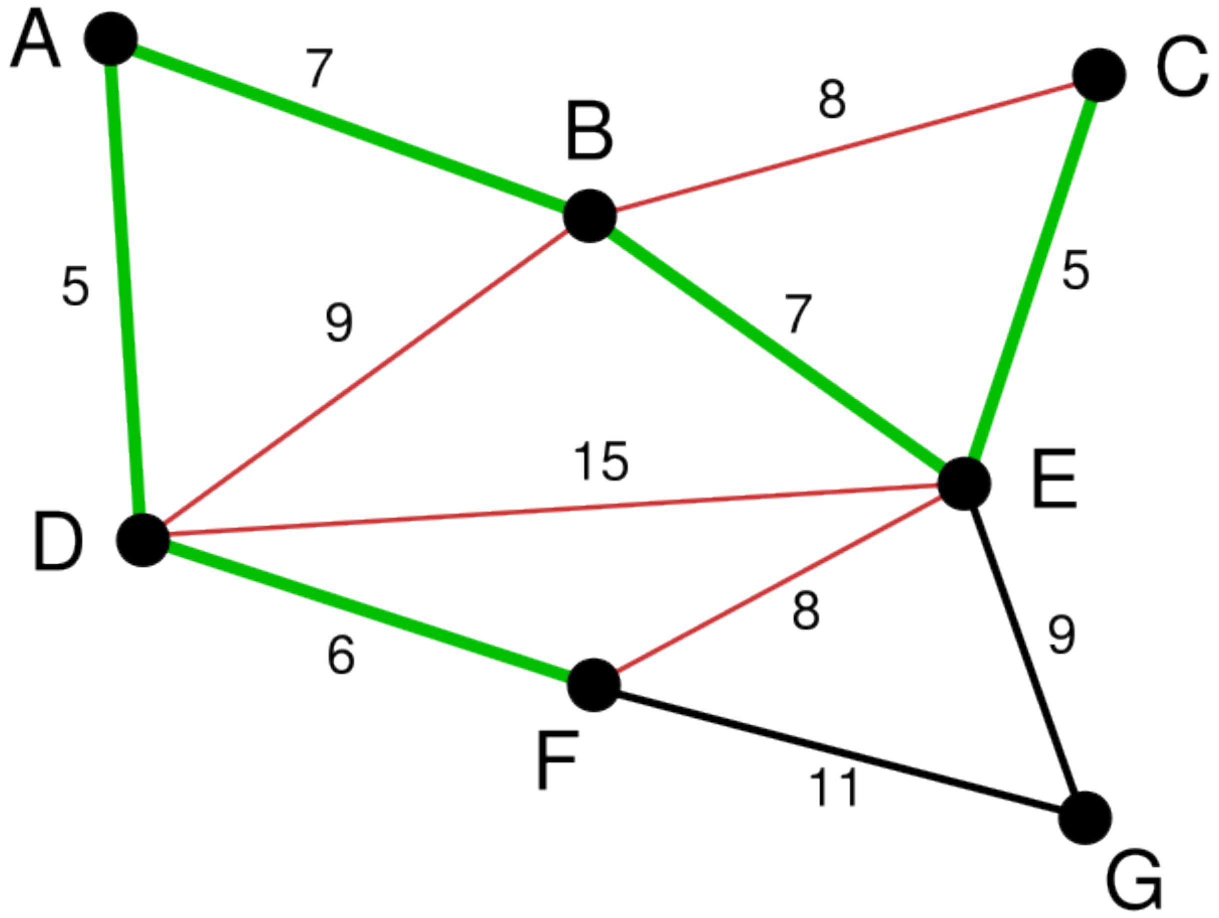
Kruskal’s Algorithm Example

The next-shortest arcs are AB and BE, both with length 7. AB is chosen arbitrarily, and is highlighted. The arc BD has been highlighted in red, because there already exists a path (in green) between B and D, so it would form a cycle (ABD) if it were chosen.



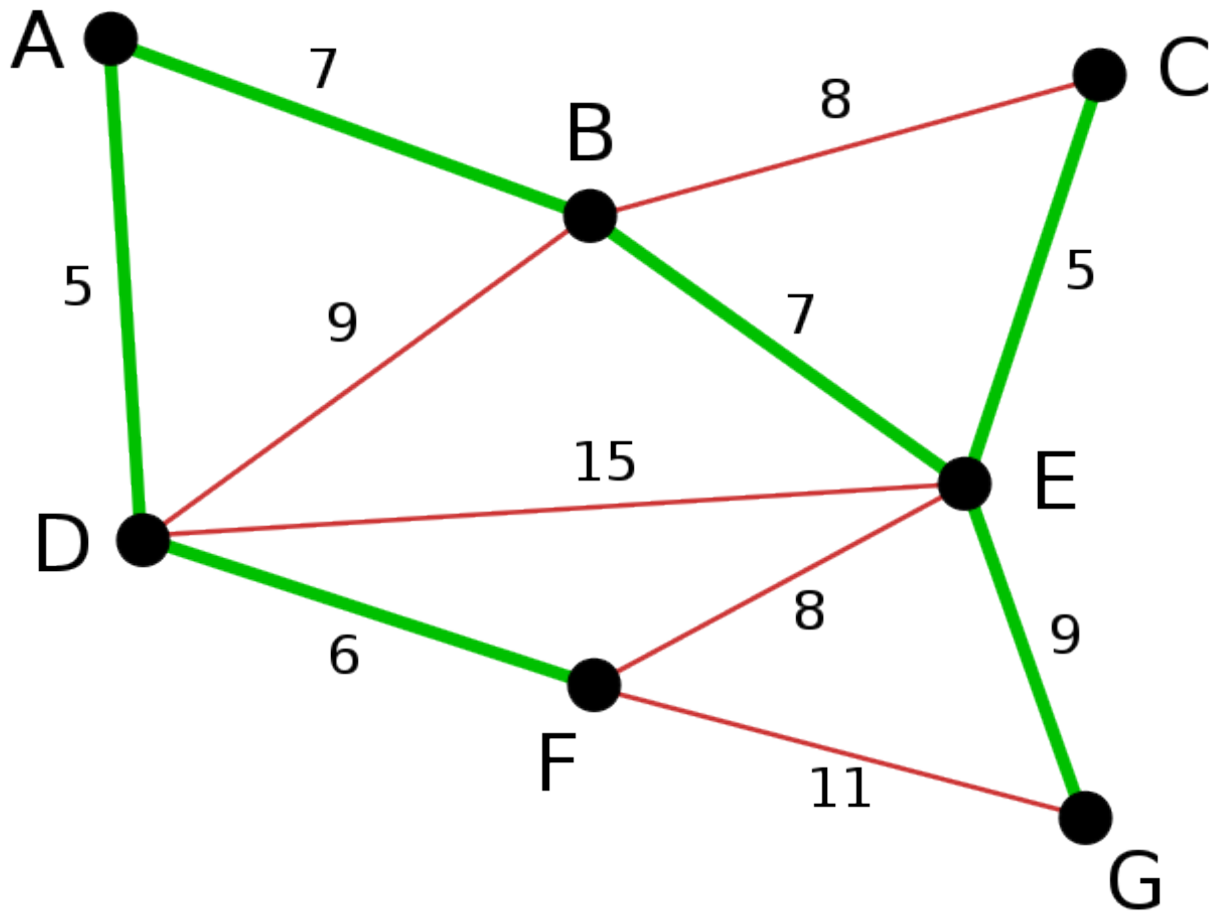
Kruskal’s Algorithm Example

The process continues to highlight the next-smallest arc, BE with length 7. Many more arcs are highlighted in red at this stage: BC because it would form the loop BCE, DE because it would form the loop DEBA, and FE because it would form FEBAD.



Kruskal's Algorithm Example

Finally, the process finishes with the arc EG of length 9, and the minimum spanning tree is found.



Dijkstra’s Algorithm

- Compute length of shortest path from source vertex to every other vertex
- Works on directed and undirected graphs
- Works only on graphs with non-negative edge weights
- **O(m log m)** runtime when implemented with Priority Queue, same as Prim’s

Dijkstra's Algorithm

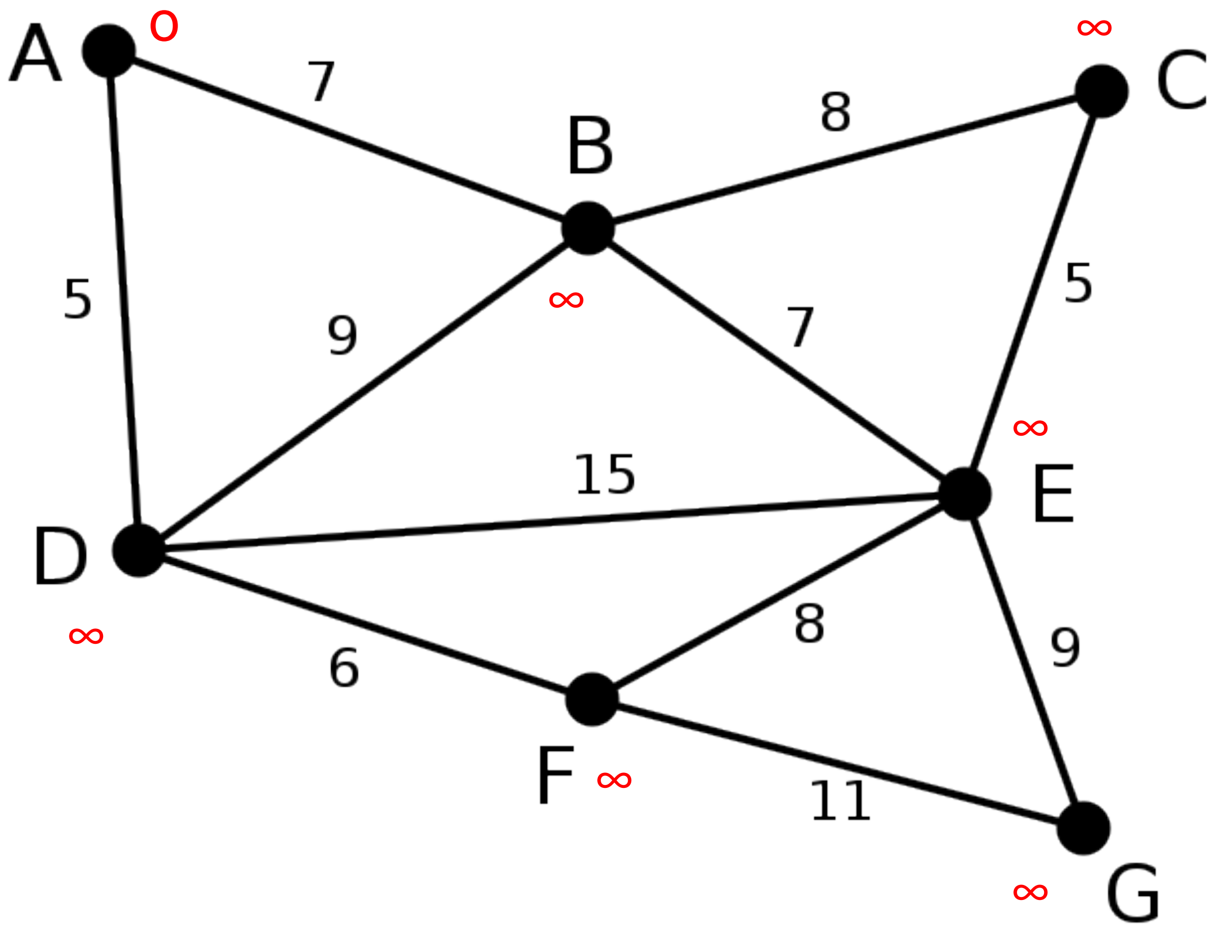
- Similar to Prim's algorithm
- Difference lies in the priority
  - Priority is the length of shortest path to a visited vertex + cost of edge to unvisited vertex
  - We know the shortest path to every visited vertex
- On unweighted graphs, BFS gives us the same result as Dijkstra's algorithm

Dijkstra's Algorithm

1. Assign to every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes as unvisited. Set initial node as current.
3. For current node, consider all its unvisited neighbors and calculate their tentative distance (from the initial node) If this distance is less than the previously recorded distance, overwrite the distance.
4. When we are done considering all neighbors of the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal.
5. If all nodes have been visited, finish. Otherwise, set the unvisited node with the smallest distance (from the initial node) as the next "current node" and continue from step 3.

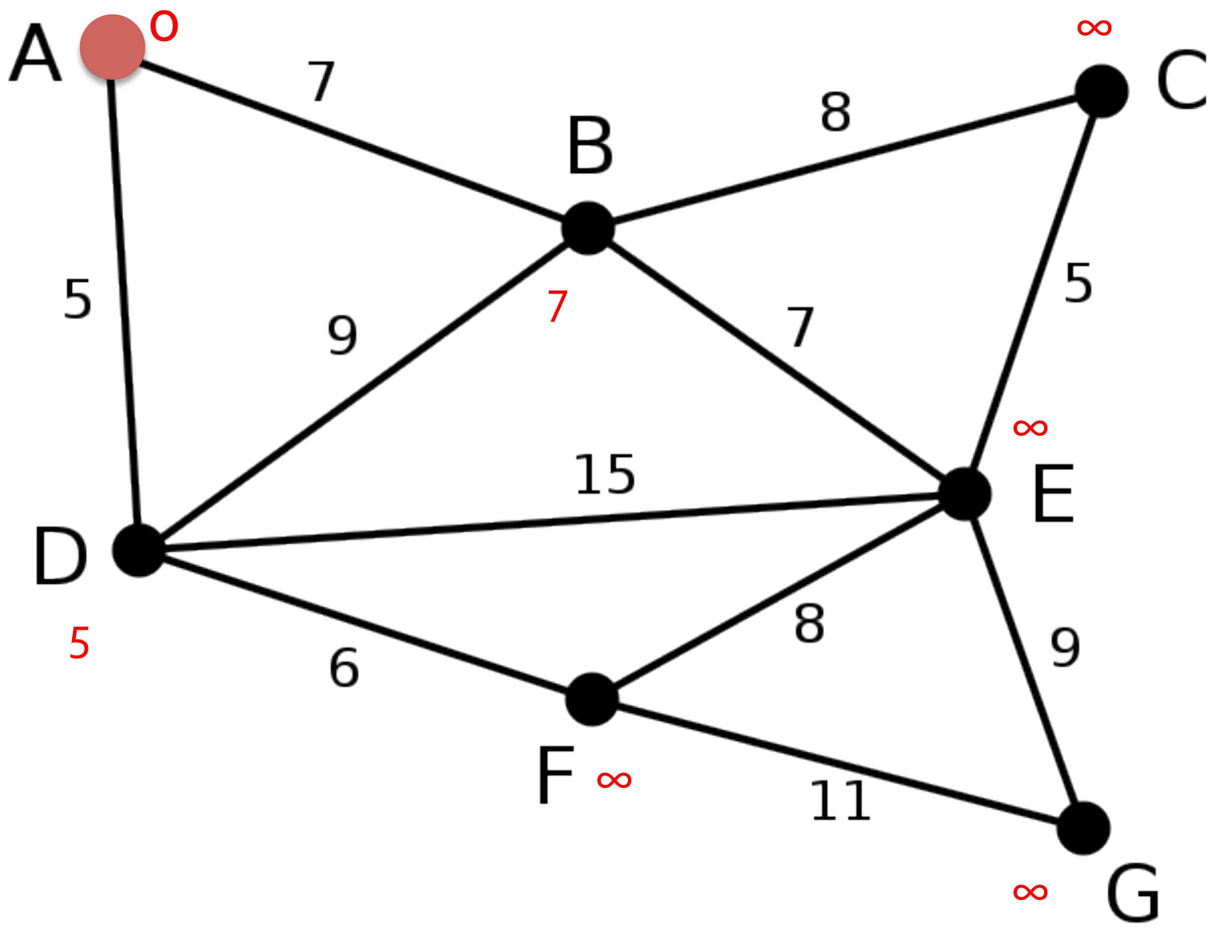
Dijkstra’s Algorithm Example

Initial distances set to 0 for initial node and ∞ for all other nodes.



Dijkstra's Algorithm Example

Set distances for all nodes connected to the initial node. Mark the initial node as done (red).



Dijkstra's Algorithm Example

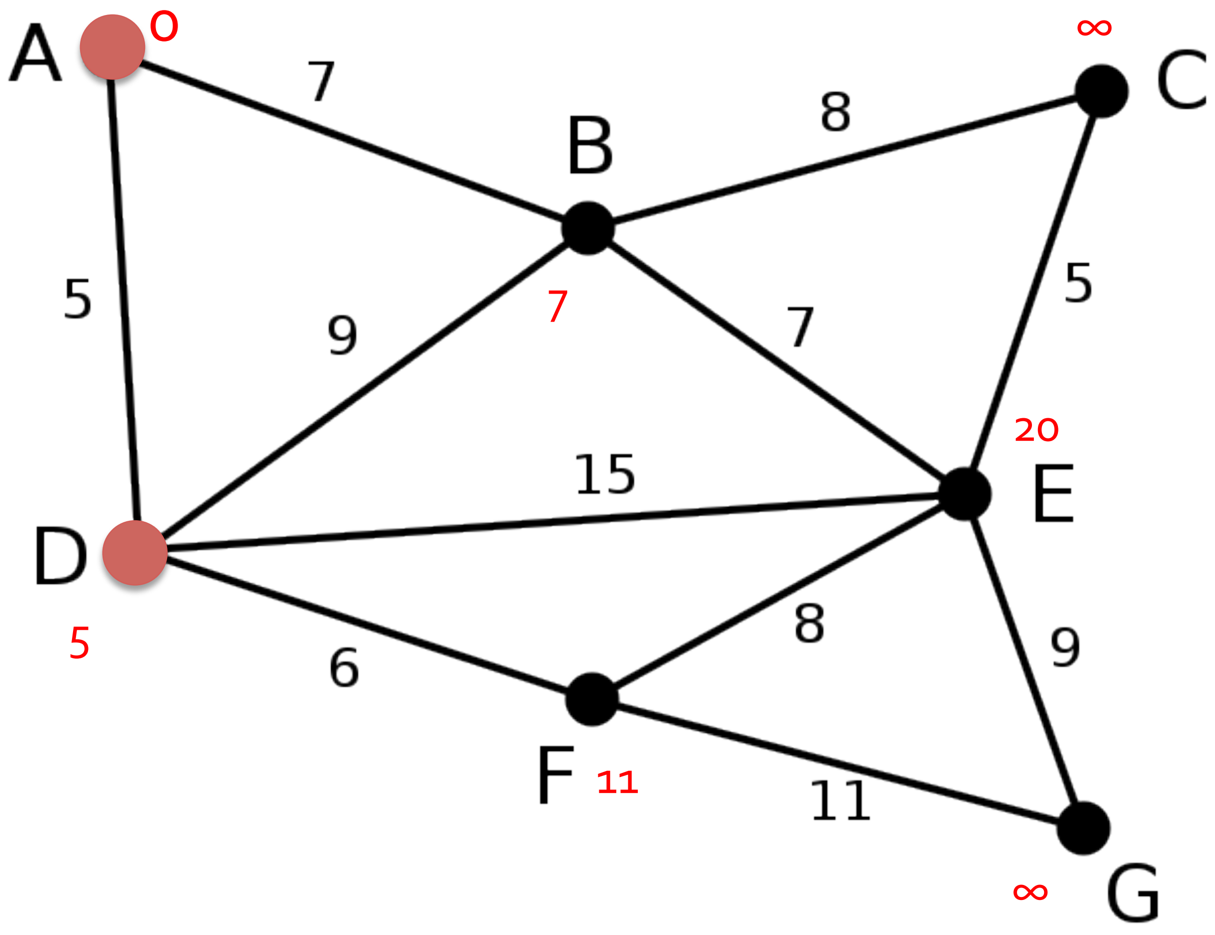
Select the node is with the smallest distance that isn't done, and update the distances to its neighbors.

F = 11 : 5 + 6 = 11

B = 7: 5 + 9 = 14 > 7

E = 20: 5 + 15 = 20

Mark D as visited.



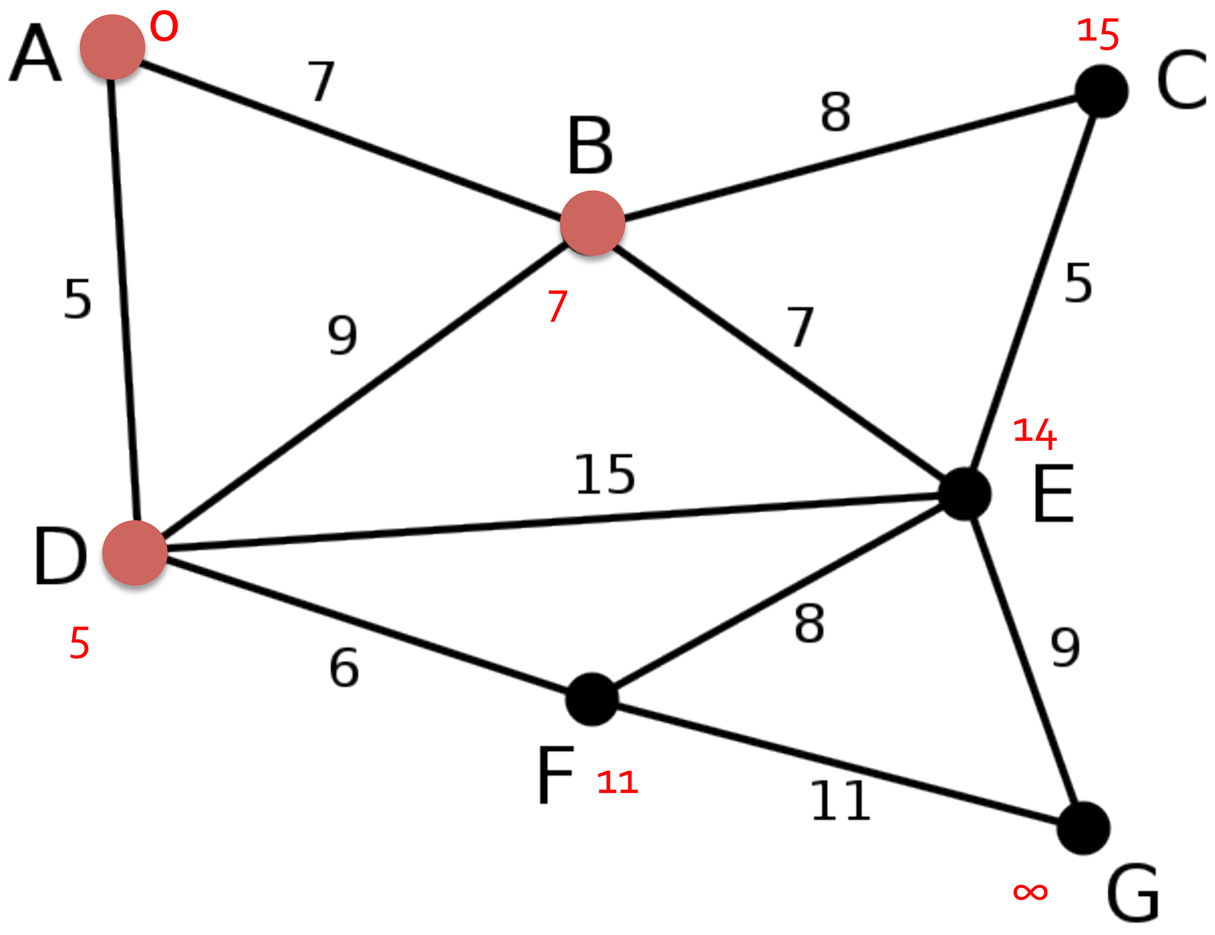
Dijkstra's Algorithm Example

Set the current node to B.

E = 14: 7 + 7 = 14

C = 15: 7 + 8 = 15

Mark B as visited.



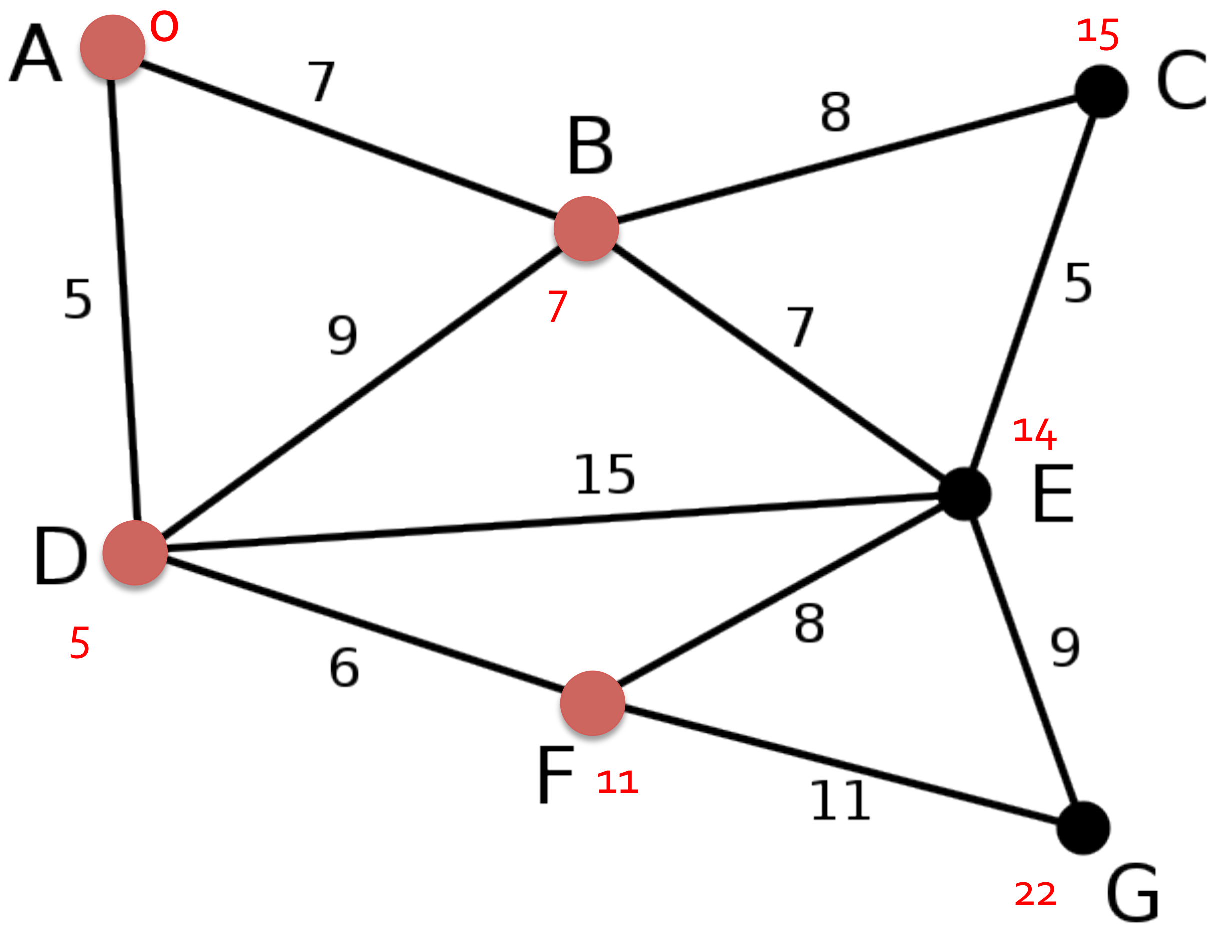
Dijkstra's Algorithm Example

Repeat the process:

E = 14: 11 + 8 = 19 > 14

G = 22: 11 + 11 = 22

Mark F as visited



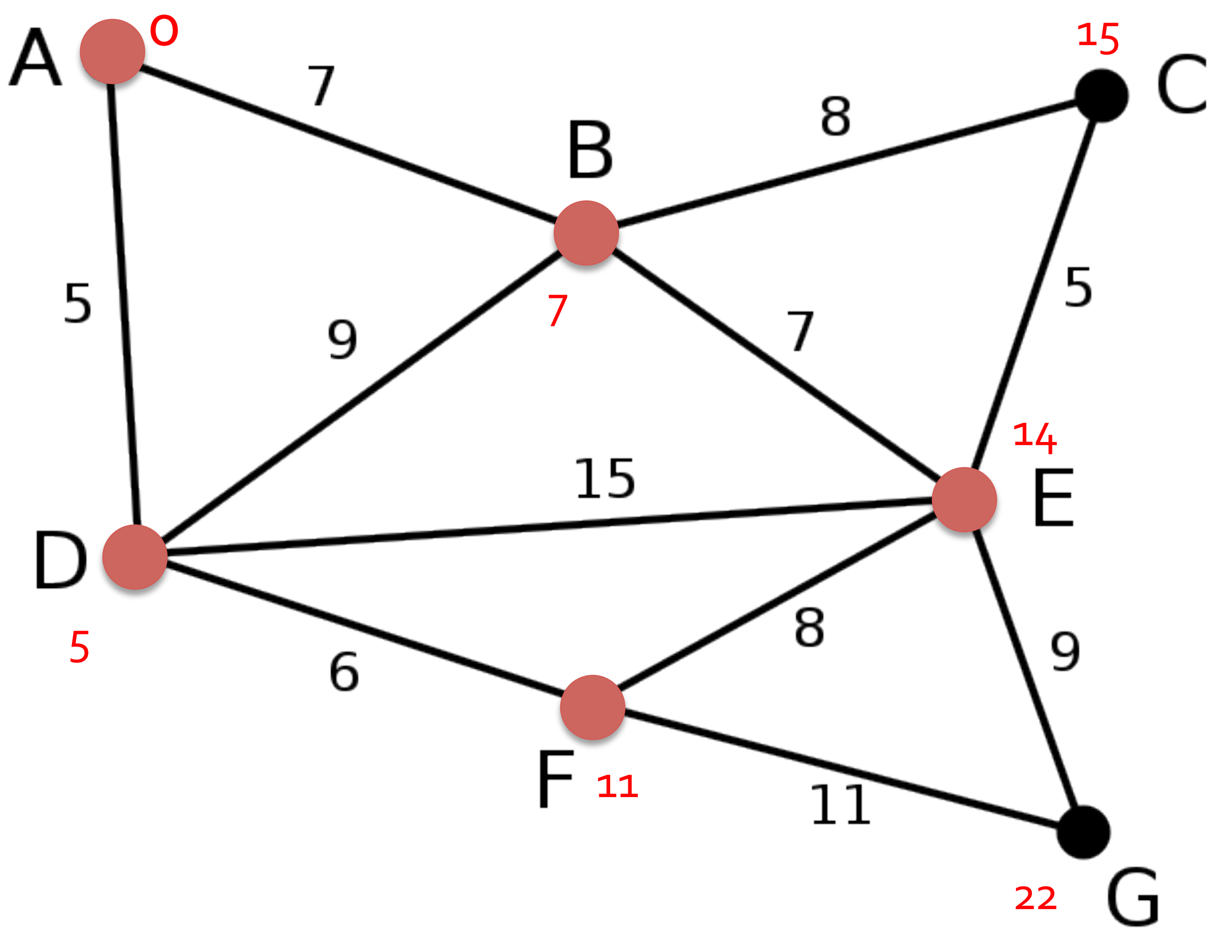
Dijkstra's Algorithm Example

Repeat the process:

C = 15: 14 + 5 = 19 > 15

G = 22: 14 + 9 = 23 > 22

Mark E as visited



Question Time

- Now we’ll take a 5-10 minute break
- We’ll begin Q&A session afterwards