CS 2110 FINAL REVIEW

Johnathon Schultz

Fall 2010

Final Information

- Final
  - Thursday, December 16th
  - Barton Hall – West
  - 2:00 - 4:30
- How to Review:
  - Review previous prelims (esp. this years!)
  - Review previous finals
  - Review lecture slides
  - Review previous review slides
  - Attend this session

Material

- In short, everything ever
- Types
- Recursion
- Lists and Trees
- Big-O
- Induction
- Threads & Concurrency
- ADTs:
  - Stacks
  - Queues
  - Priority Queues
  - Maps
  - Sets
- Graph Algorithms:
  - Prim's
  - Kruskal's
  - Dijkstra's

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.

Inheritance

Inheritance

- Subclasses inherit fields & methods of superclass
- Overriding – subclass contains the same method as superclass (same name, parameters, static/not)
- Shadowing – subclass contains the same field (instance variable) as superclass (this is BAD)
- Casting – upcasting is always type-safe and OK
  - Downcasting is bad – sometimes doesn't work (hard to predict)
- Java is single implementation inheritance and multiple interface inheritance

Typing

- Suppose type B implements or extends type A
  - B is a subtype of A; A is a supertype of B
- Each variable has a static type
  - List<Integer> x; List<Integer> is the static type
  - Note: List<SubtypeOfInteger> is not a subtype of List<Integer>
- Can safely assign x a dynamic subtype of List<Integer>
  - x = new ArrayList<Integer>;
- Static type can differ from dynamic type at runtime
  - The dynamic type cannot be an interface

Typing examples

- B var = new C();
- Static type = B
  - Static type is used when calling fields; i.e. var.x will call the field x in class B
  - NEVER CHANGES
- Dynamic type = C
  - Used when calling methods; i.e. var.hello() will call the method hello() in C (not the one in B!)
  - Changed by: var = new B();
  - Now, the dynamic type is B
  - Casting: var = (B) var – does not change any type

Polymorphism

- Previous slide: "Used when calling methods; i.e. var.hello() will call the method hello() in C (not the one in B!)"
- This is called Polymorphism
- When a method is called on an object, the method in the object's dynamic type is the method that is actually run!
- NOT the method in its static type, even if the method is defined there.

The instanceof Operator

Code	Effect
`dog instanceof Dog`	true
`dalmatian instanceof Dog`	true
`dog instanceof Dalmatian`	false
`dalmatian instanceof ShowDogInterface`	true
`dalmatian instanceof dog`	syntax error
`! dalmatian instanceof Dog`	syntax error
`! (dalmatian instanceof Dog)`	false
`dog instanceOf Dalmatian`	syntax error
`dalmatian instanceof "Dog"`	syntax error
`dalmatian instanceof Class.forName ("DogPackage.Dog" )`	syntax error
`null instanceof String`	false

Recursion

Recursion

- *To understand recursion, you must first understand recursion.*
- A procedure or subroutine whose implementation references itself
- Examples
  - Fibonacci: Fib(n) = Fib(n-1) + Fib(n-2)
  - Factorial: n! = n * (n-1)!
  - Grammar Parsing
- Must have one or more base cases and a recursive case
  - You don't need to know proofs by induction for Prelim 1
- If a problem asks you to write a recursive method, you must call that method within itself
- You should know how to run through a recursive method and figure out its output
  - See Recursion.java

How to Write Recursive Methods

- Start with the base case
  - Ask yourself, what am I doing when I'm done?
  - That's your base case
- Next, what is the simplest case that isn't the base case?
  - Because of the nature of recursion, this case is exactly like all the other cases
- For example, let's write a LinkedList Reverser…

Reversing a LinkedList

```
Public void reverse(???){



}
```

Reversing a LinkedList

```
Public void reverse(Node curr){
        if(curr.next == null){
                head = curr;
        }
}
```

Reversing a LinkedList

```
Public void reverse(Node prev, Node curr){
        if(curr.next == null){
                head = curr;
        }
        else{
                reverse(curr, curr.next);
        }
        curr.next = prev;
}
```

Reversing a LinkedList

But wait! We forgot one thing!

What if the list is empty?

Time for a recursive helper function!

Reversing a Linked List

- Recursive helper functions are useful for edge cases that don't appear in the general recursion

```
public void reverse(){
        if(! isEmpty() && head.next != null)
                return reverse(null, head);
}
```

Reversing a LinkedList

```
public void reverse(){
        if(! isEmpty() && head.next != null)
                return reverse(null, head);
}
Public void reverse(Node prev, Node curr){
        if(curr.next == null){
                head = curr;
        }
        else{
                reverse(curr, curr.next);
        }
        curr.next = prev;
}
```

Grammars and Parsing

- Refer to the following grammar (ignore spaces). <S> is the start symbol of the grammar. (Note that P → a | b is really two rules, P → a and P → b)
  - <S> → <exp>
  - <exp> → <int> + <int> | <int> - <med_int> | <int> + <exp>
  - <int> → <small_int> | <med_int> <large_int> | <small_int>.<large_int>
  - <large_int> → 8 | 9
  - <med_int> → 5 | 6 | 7
  - <small_int> → 0 | 1 | 2 | 3 | 4
- Is “4 + 2.8 – 49” a valid sentence?

Grammars and Parsing

- Refer to the following grammar (ignore spaces). <S> is the start symbol of the grammar. (Note that P → a | b is really two rules, P → a and P → b)
  - <S> → <exp>
  - <exp> → <int> + <int> | <int> - <med_int> | <int> + <exp>
  - <int> → <small_int> | <med_int> <large_int> | <small_int>.<large_int>
  - <large_int> → 8 | 9
  - <med_int> → 5 | 6 | 7
  - <small_int> → 0 | 1 | 2 | 3 | 4
- Is “30 + 0 + 0.99” a valid sentence?

Grammars and Parsing

- Refer to the following grammar (ignore spaces). <S> is the start symbol of the grammar. (Note that P → a | b is really two rules, P → a and P → b)
  - <S> → <exp>
  - <exp> → <int> + <int> | <int> - <med_int> | <int> + <exp>
  - <int> → <small_int> | <med_int> <large_int> | <small_int>.<large_int>
  - <large_int> → 8 | 9
  - <med_int> → 5 | 6 | 7
  - <small_int> → 0 | 1 | 2 | 3 | 4
- Which rule makes the grammar infinite?

Recursive Descent Parsers

- Recursively parse the data by descending from the top level into smaller and smaller chunks.
- Cannot handle all Grammars
  - Ex:
    - $S \rightarrow b$
    - $S \rightarrow Sa$
- Grammar can be rewritten:
  - $S \rightarrow b$
  - $S \rightarrow bA$
  - $A \rightarrow a$
  - $A \rightarrow aA$

Example

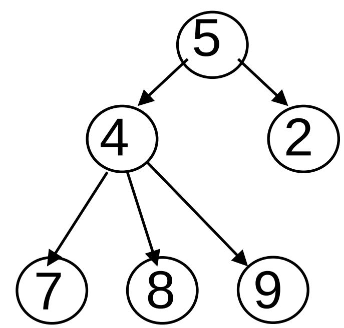
A := A B

```
boolean A() {
    if (A()) {
        return B();
    }
    return false;
}
```

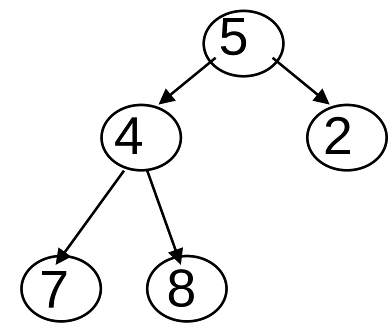
Trees

trees

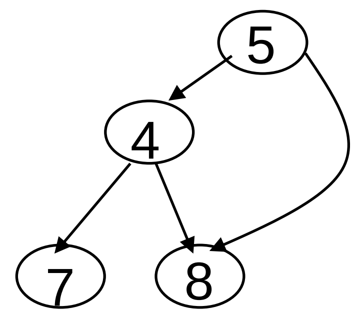
- *Tree*: recursive data structure (similar to lists)
  - Each cell may have zero or more *successors* (children)
  - Each cell has exactly one *predecessor* (parent) except the *root*, which has none
  - All cells are reachable from *root*



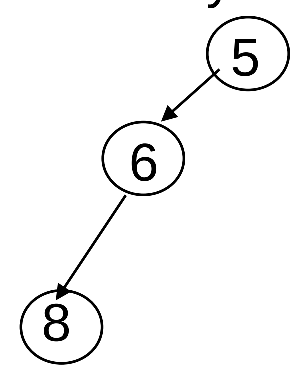
General tree



Binary tree



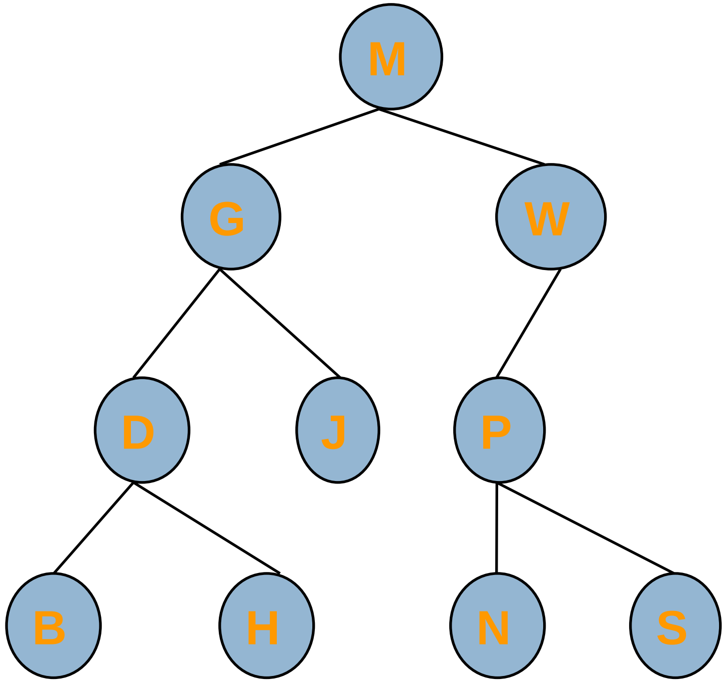
Not a tree



List-like tree

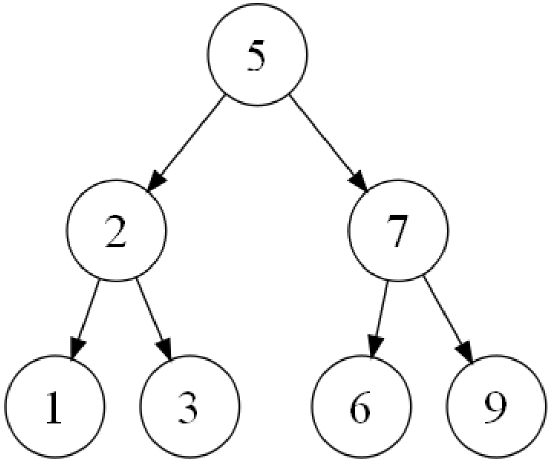
Tree terminology

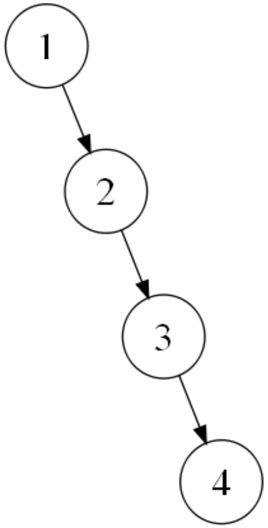
- M is the *root* of this tree
- G is the *root* of the *left subtree* of M
- B, H, J, N, and S are *leaves*
- N is the *left child* of P; S is the *right child*
- P is the *parent* of N
- M and G are *ancestors* of D
- P, N, and S are *descendants* of W
- Node J is at *depth* 2 (i.e., *depth* = length of path from root)
- Node W is at *height* 2 (i.e., *height* = length of longest path to a leaf)
- A tree is complete if it all the levels are completely filled except for the last



Binary search trees

- Also known as BSTs
- Children to the left are less than the current node
- Children to the right are greater than the current node



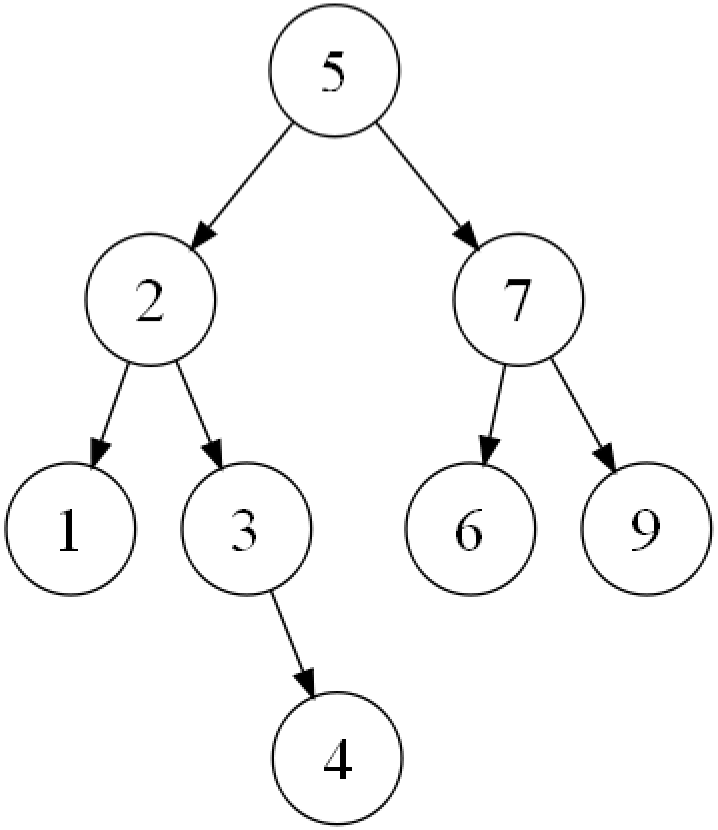


Tree traversals

- Preorder
  - Root node, Left Node, Right Node
- Postorder
  - Left Node, Right Node, Root Node
- Inorder
  - Left Node, Root Node, Right Node
- Breadth First
  - First level, Second Level, Third Level, …
- Depth First
  - Root, Child, Child, Child, …, Leaf, Up One, Child, Child, Leaf, …

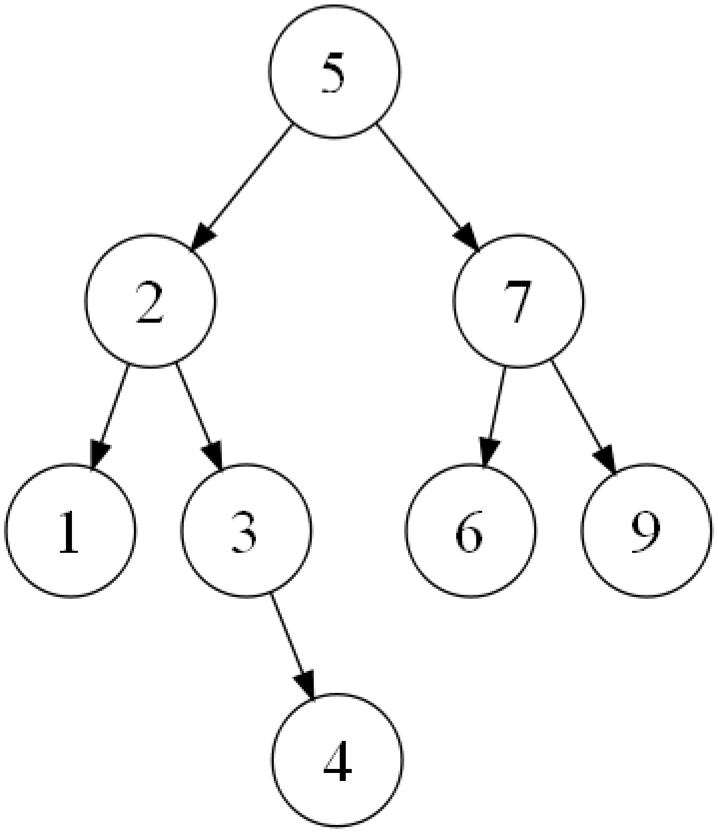
Preorder Traversal

- **Pre(5)**
- 5, **Pre(2)**, Pre(7)
- 5, 2, 1, **Pre(3)**, Pre(7)
- 5, 2, 1, 3, 4, **Pre(7)**
- 5, 2, 1, 3, 4, 7, 6, 9



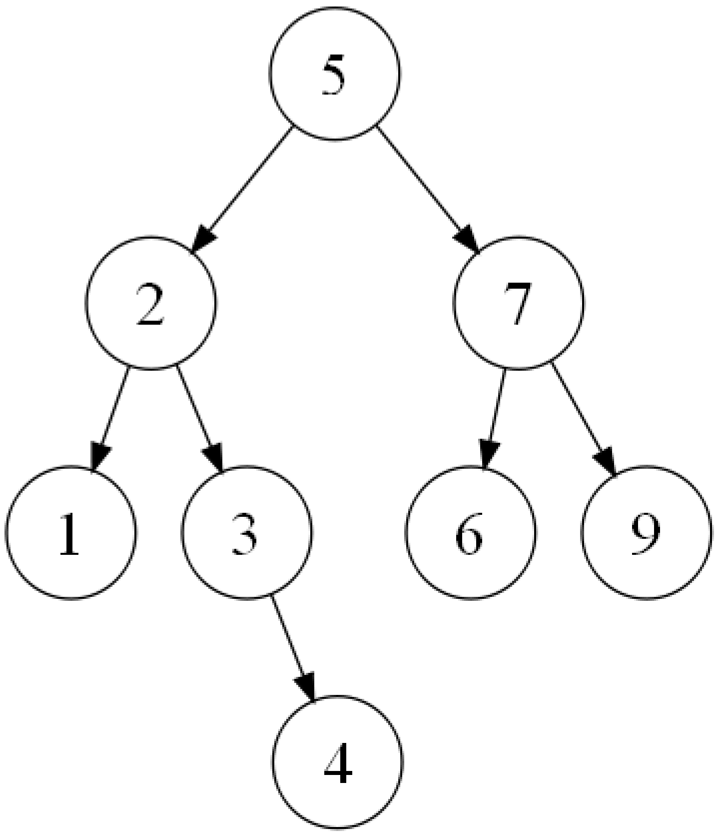
Inorder Traversal

- **In(5)**
- **In(2)**, 5, In(7)
- 1, 2, **In(3)**, 5, In(7)
- 1, 2, 3, 4, 5, **In(7)**
- 1, 2, 3, 4, 5, 6, 7, 9



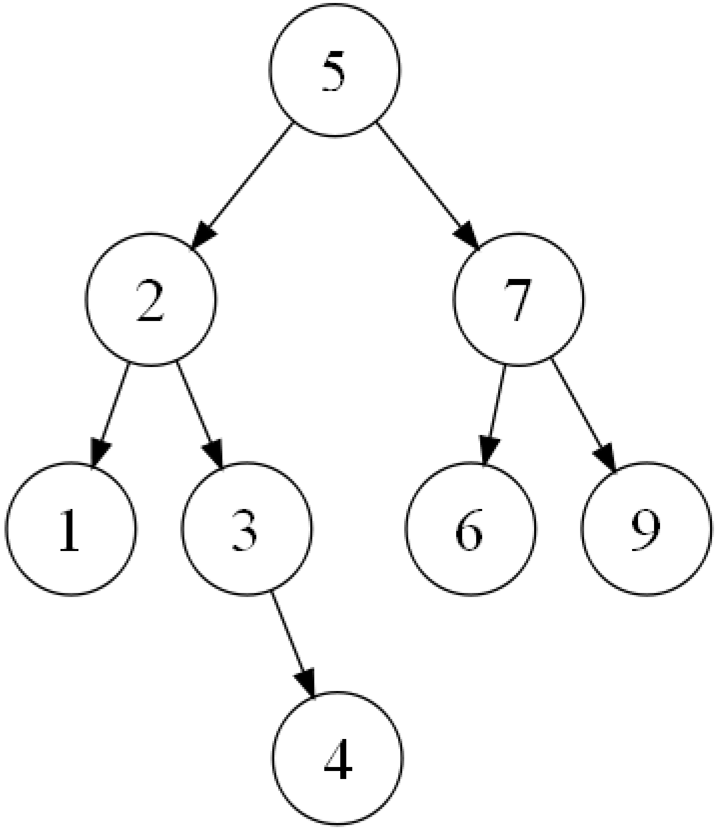
Postorder Traversal

- **Post(5)**
- **Post(2)**, Post(7), 5
- 1, **Post(3)**, 2, Post(7), 5
- 1, 4, 3, 2, **Post(7)**, 5
- 1, 4, 3, 2, 6, 9, 7, 5



Breadth first and depth first

- Breadth First:
  - 5, Depth 1, Depth 2, Depth 3
  - 5, 2, 7, Depth 2, Depth 3
  - 5, 2, 7, 1, 3, 6, 9, Depth 3
  - 5, 2, 7, 1, 3, 6, 9, 4
- Depth First:
  - 5, 5.Left, 5.Right
  - 5, 2, 2.Left, 2.Right, 5.Right
  - 5, 2, 1, 2.Right, 5.Right
  - 5, 2, 1, 3, 3.Right, 5.Right
  - 5, 2, 1, 3, 4, 5.Right



Efficiency

Big O notation

- F is O(n) means F is on the order of n.
- Big O provides an **upper bound** for the number of operations the function is performing, based on the size of its input

Notation	Name	Example
$O(1)$	Constant	Determining if a number is odd
$O(\log n)$	Logarithmic	Finding an item in a sorted list or tree
$O(n)$	Linear	Finding an item in an unsorted list or tree.
$O(n \log n)$	Quasilinear	Quicksort (best and average case), Merge sort
$O(n^2)$	Quadratic	Bubble sort, Quicksort (worst case)
$O(n^c)$	Polynomial	Maximum matching for bipartite graphs
$O(c^n)$	Exponential	Travelling salesman advanced, K-SAT brute-force
$O(n^n)$ $O(n!)$	Factorial	Travelling salesman brute-force

Formal Definition

Let *f*(*x*) and *g*(*x*) be two functions defined on some subset of the real numbers.

$$f(x) = O\big(g(x)\big) as\ x \ \rightarrow \infty$$

if and only if

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0$$

M is some constant multiplier

X_0 is a value of x above which this statement is always true

Also…

- Big O – Upper Bound
- Big Omega – Lower Bound (Ω)
- Big Theta – Tight Upper and Lower Bound (Θ)

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

Induction

Induction

- We are going to spell the problem out for you
- Trick is to take the information, write it down correctly, know algebra
- You can get most of the points on an induction question just by writing down:
  - Base Cases
  - Inductive Hypothesis
  - Using the I.H.

Induction Form

- Base Case(s)
- Inductive Hypothesis
- Proof (n+1 case):
  - Given (Definition) Statement = What you want to prove
  - …
  - Substitution using the I.H.
  - …
  - Statement you want to Prove

Example

Prove that for $n \geq 1$,

$$2 + 2^2 + 2^3 + 2^4 + \cdots + 2^n = 2^{n+1} - 2$$

Let $n = 1$. Then:

$$2^1 = 2 = 2^{1+1} - 2 \text{ (Base Case)}$$

Assume that the equation holds for all n

$$2 + 2^2 + 2^3 + \cdots + 2^n = 2^{(n+1)} - 2 \text{ (I.H.)}$$

Example Cont.

$$2 + 2^1 + 2^2 + \cdots + 2^n + 2^{n+1} = 2^{n+2} - 2$$

$$2^{n+1} - 2 + 2^{n+1} = 2^{n+2} - 2 \text{ (Via I.H.)}$$

$$2 \cdot 2^{n+1} - 2 = 2^{n+2} - 2$$

$$2^1 \cdot 2^{n+1} - 2 = 2^{n+2} - 2$$

$$2^{n+2} - 2 = 2^{n+2} - 2 \text{ (Q.E.D.)}$$

Reviewing Induction

- Look at previous exams
- Look at lecture notes from CS 2800: Discrete Structures
- Even if you can't figure out the algebra, you'll get a majority of the credit if you do what I told you

Threads & Concurrency

Threads and Concurrency

- What you need to know is very simple
- Threads allow for multiple paths of execution in the code – parallel computing
- Do Not:
  - Access one variable from multiple threads without synchronization
- Operations you think are atomic are NOT
  - Cause of most thread synchronization problems
  - "i++" is actually three instructions:
    - Read-Update-Write

Threads and Concurrency

- Do: Synchronize access to variables
  - wait(), notify(), notifyAll()
  - synchronized(Object){ } blocks
  - Use thread-safe objects, for example:
    - AtomicInteger
    - BlockingQueue
    - ConcurrentHashMap,
    - ConcurrentSkipListMap (TreeMap).

Threads & Concurrency

- See the last question on last year's final
- We're not going make you write threaded code, only talk about what is wrong or right with existing code

Abstract Data Types

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)

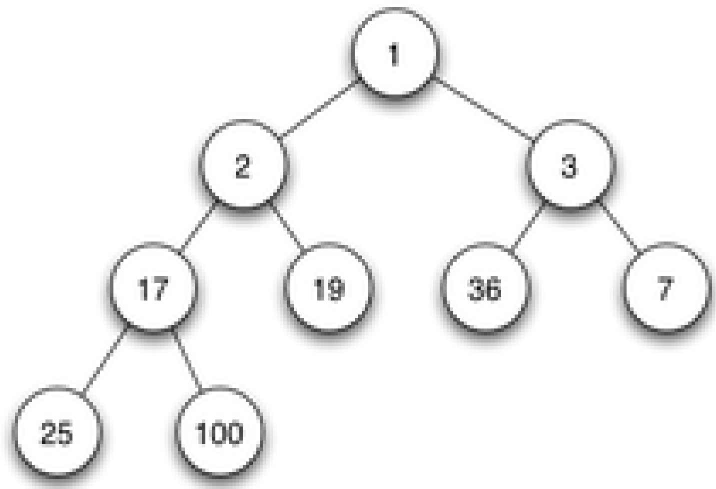
Priority Queues

- 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:**

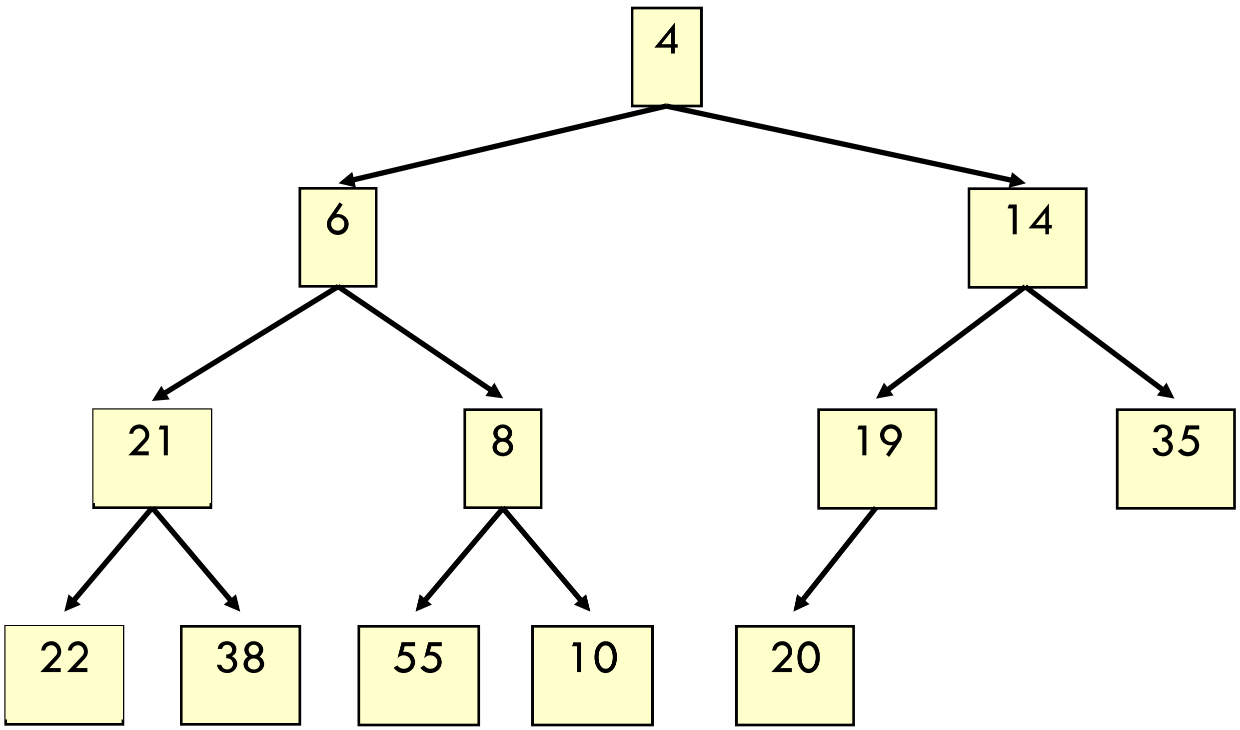
  Priority(child) $\geq$ Priority(parent)
- Operations
  - insert(value, priority)
  - extract()



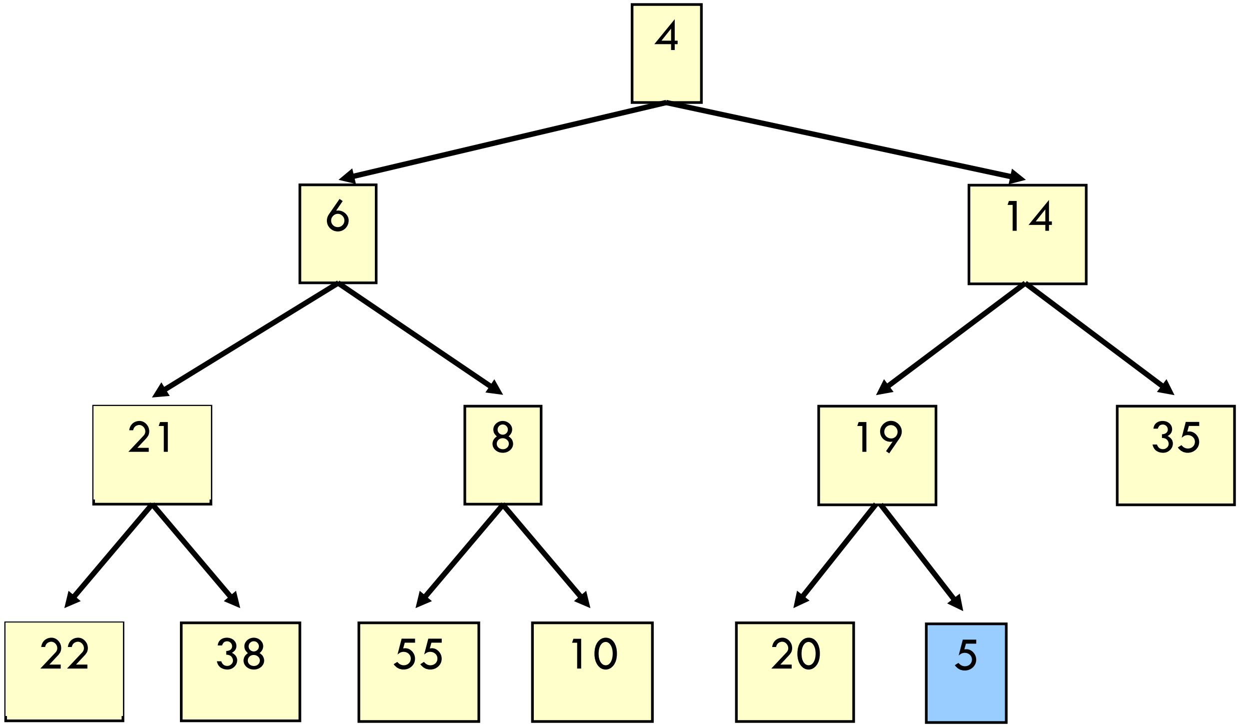
Heap insert()

- 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!

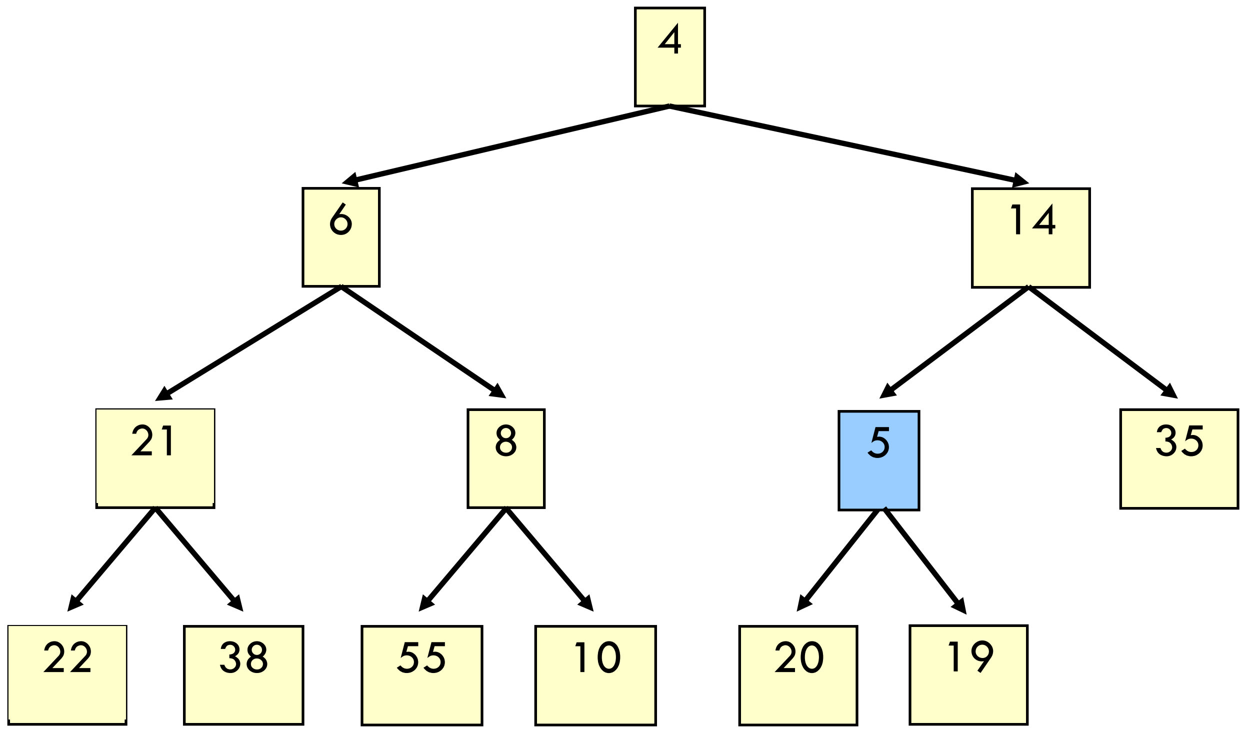
Heap insert()



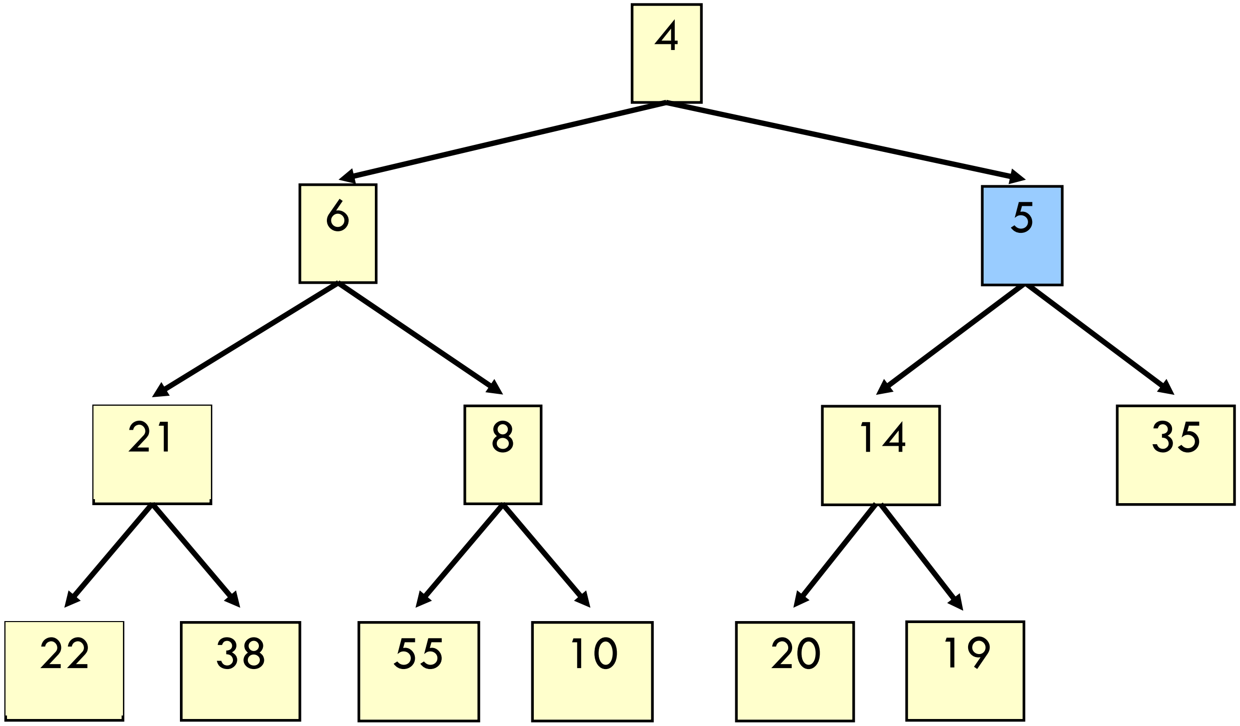
Heap insert()



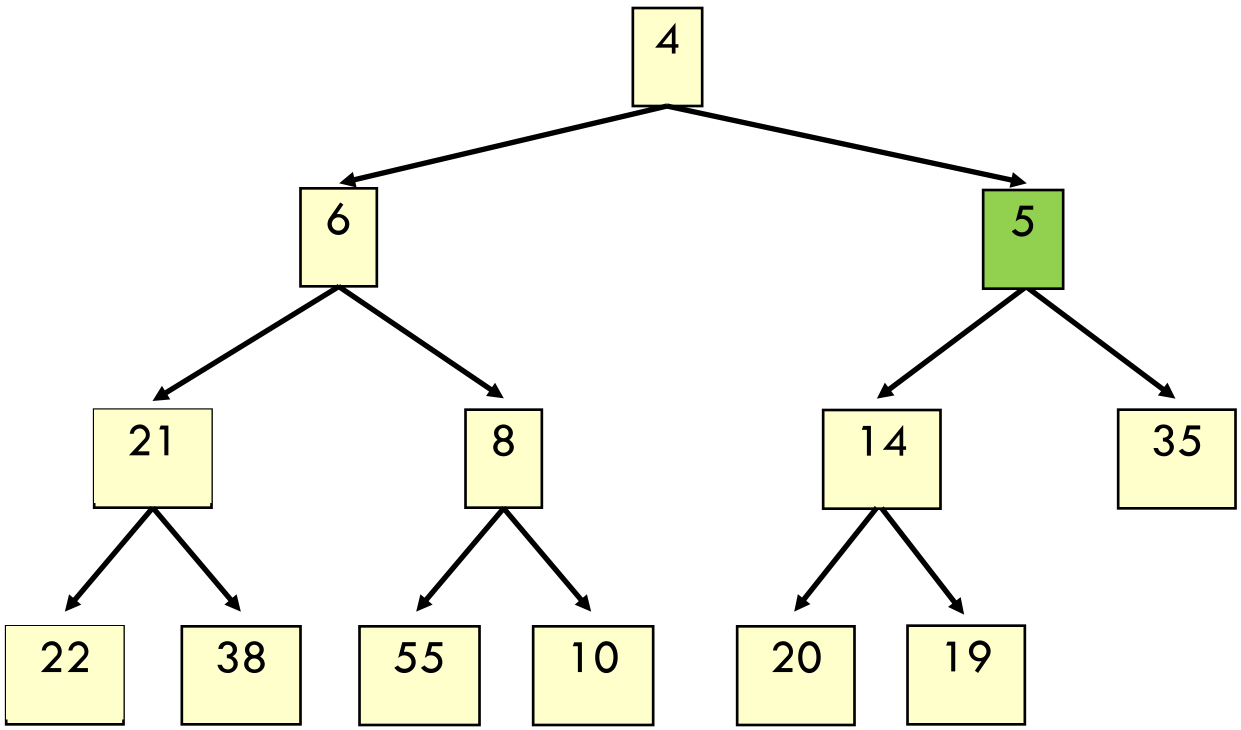
Heap insert()



Heap insert()



Heap insert()



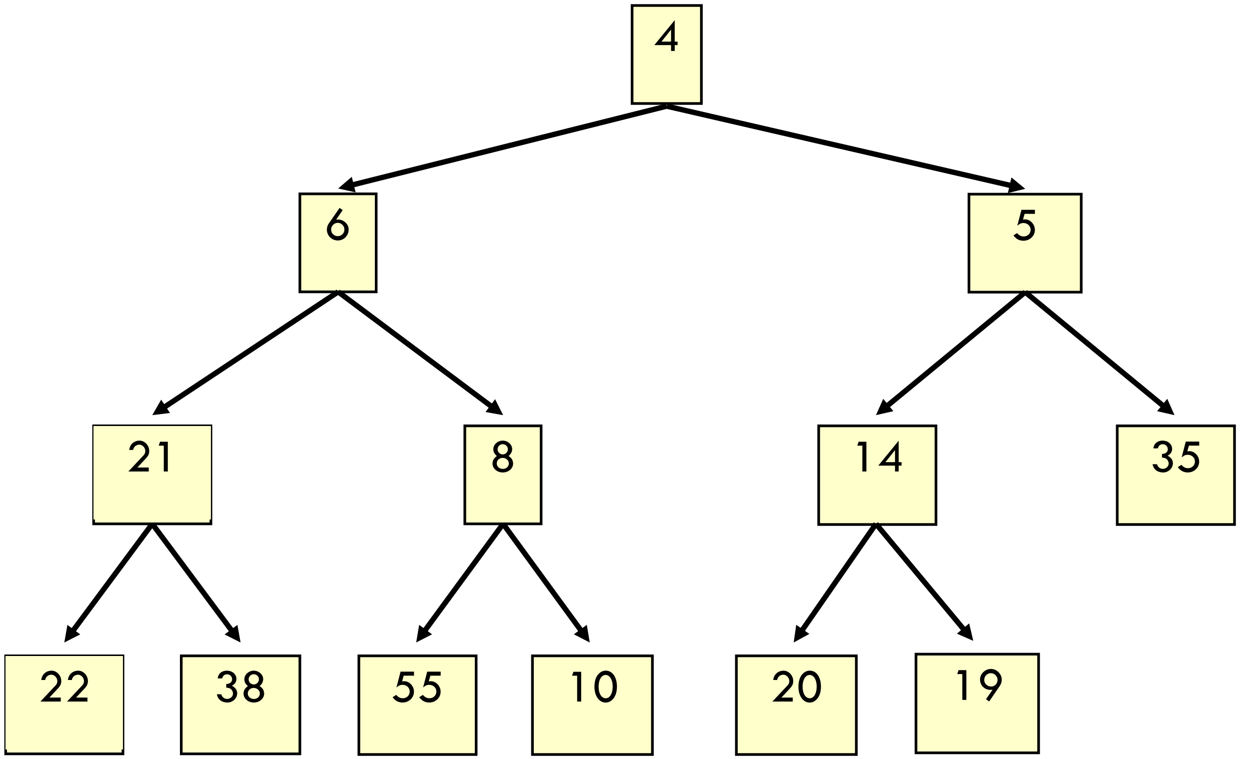
insert()

- 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()`

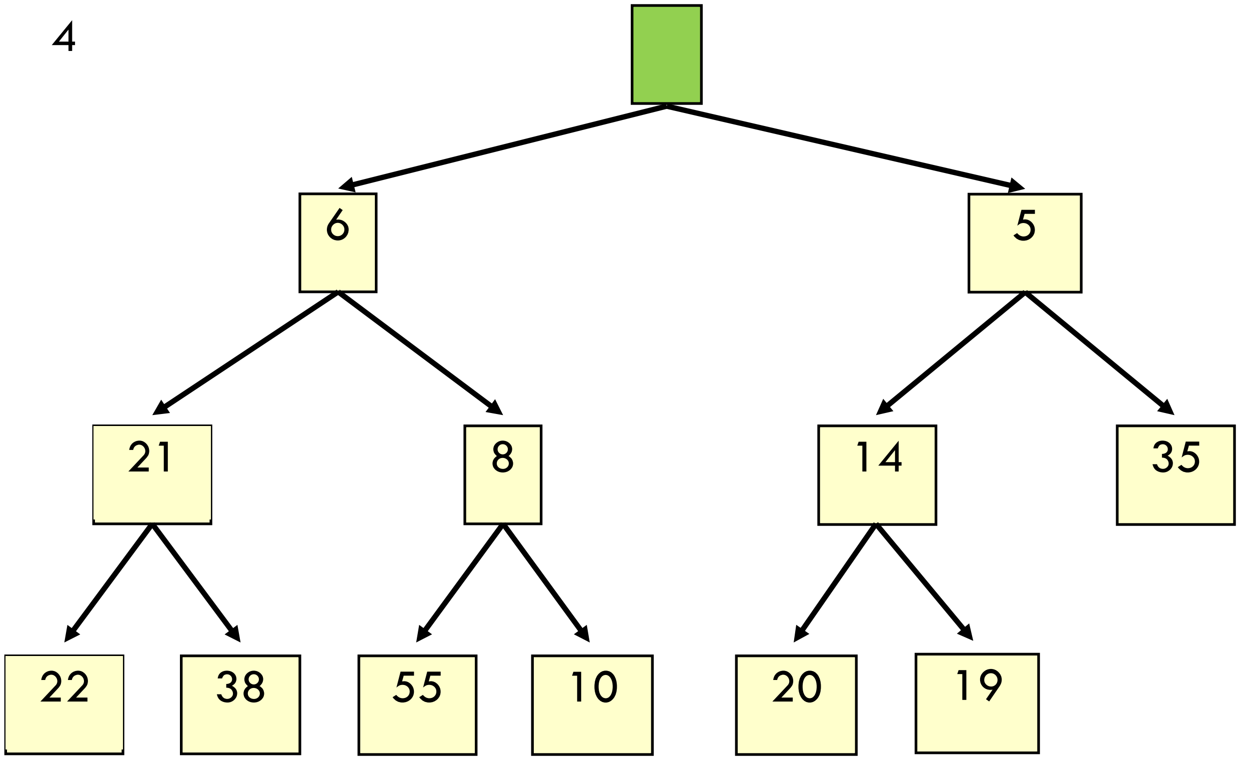
- Remove the least element – it is at the root
- This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place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

`extract()`



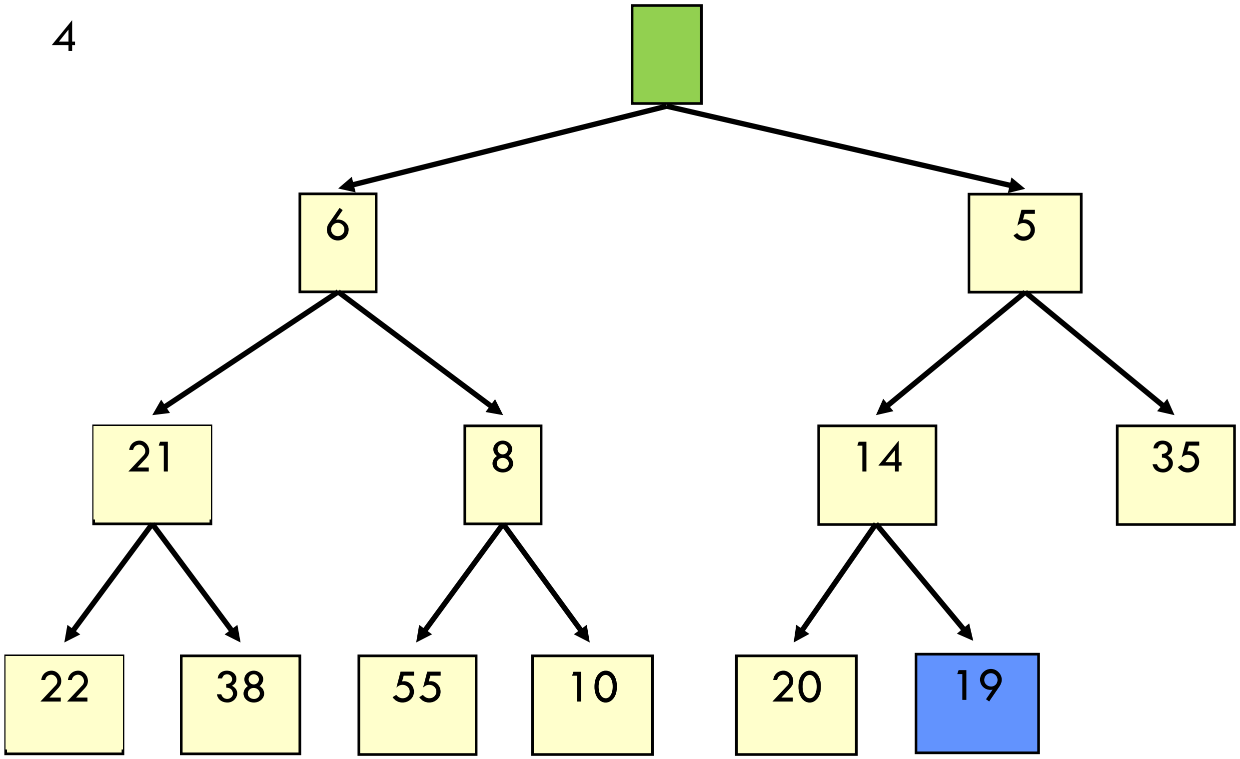
`extract()`

4



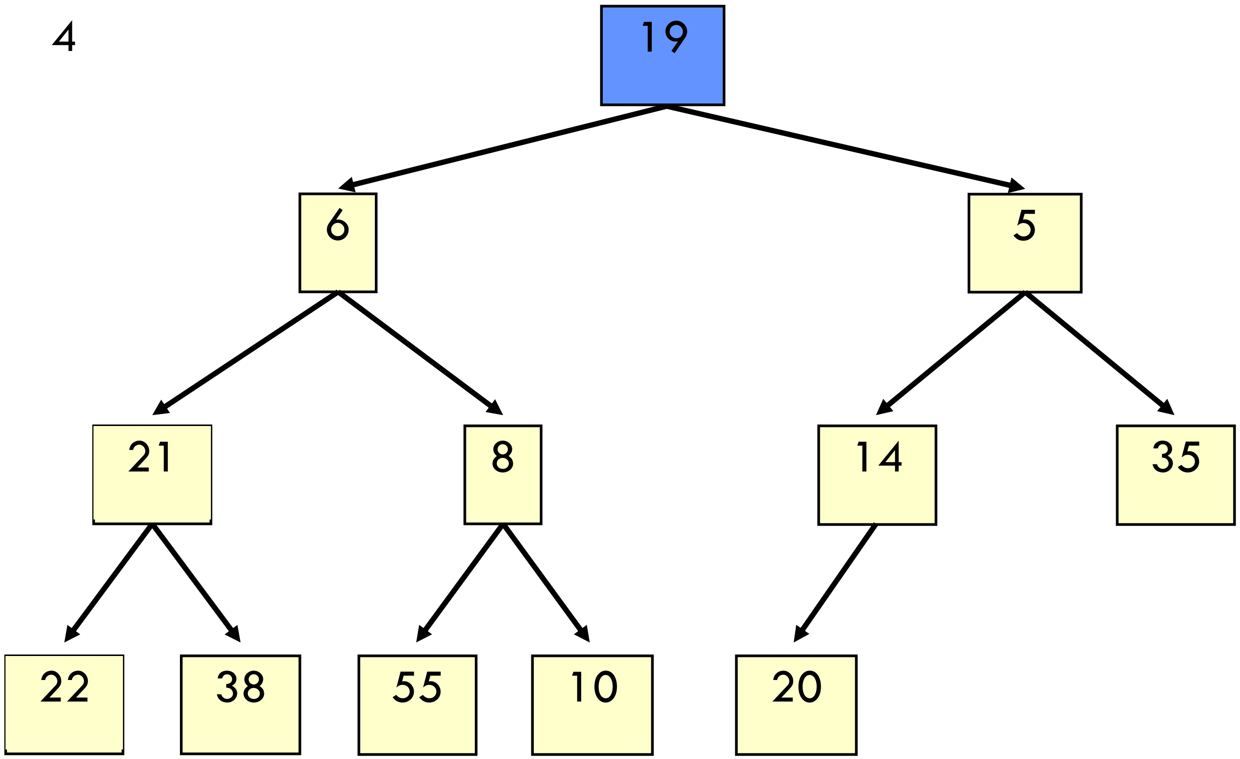
extract()

4



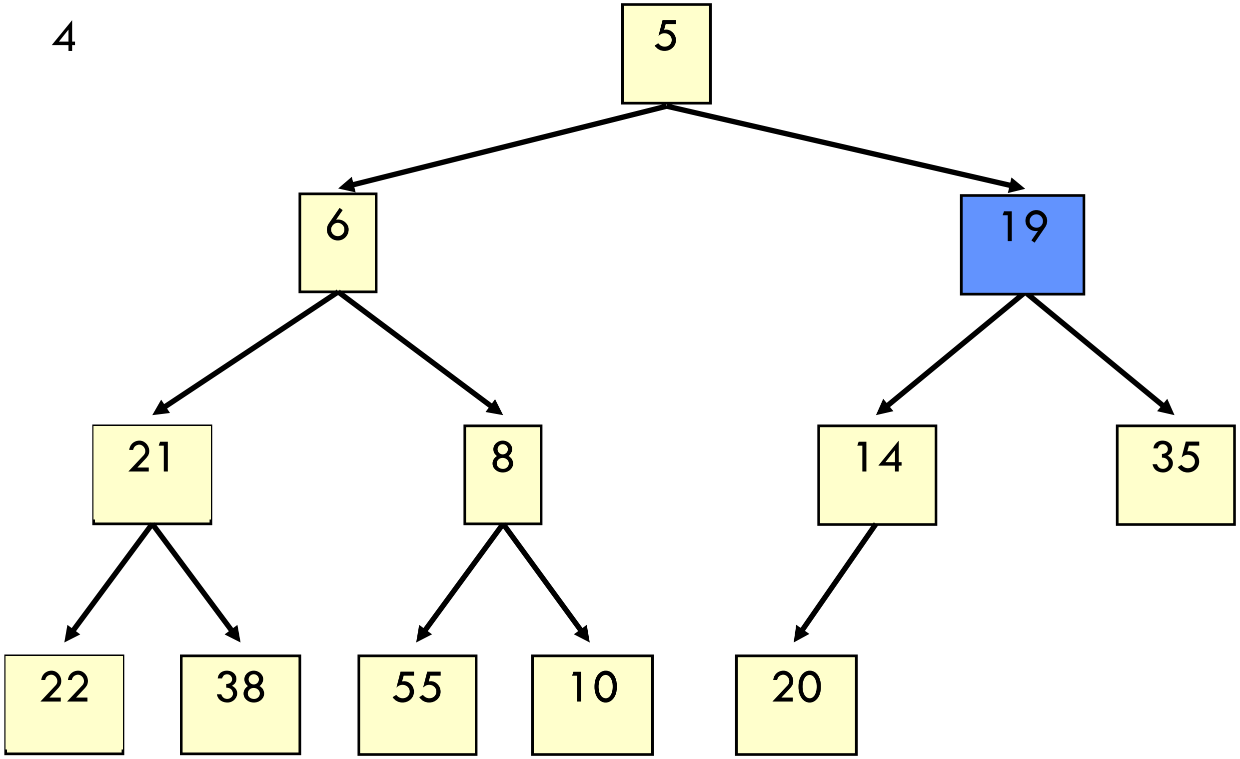
extract()

4

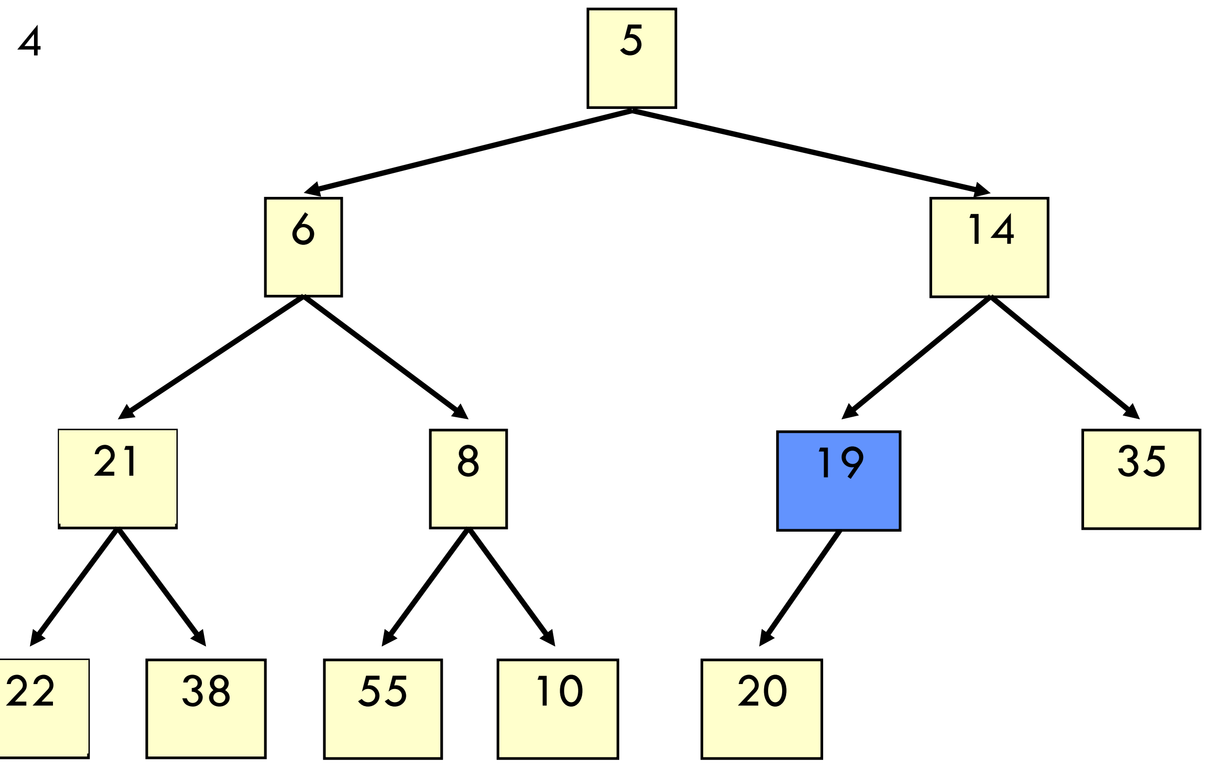


extract()

4

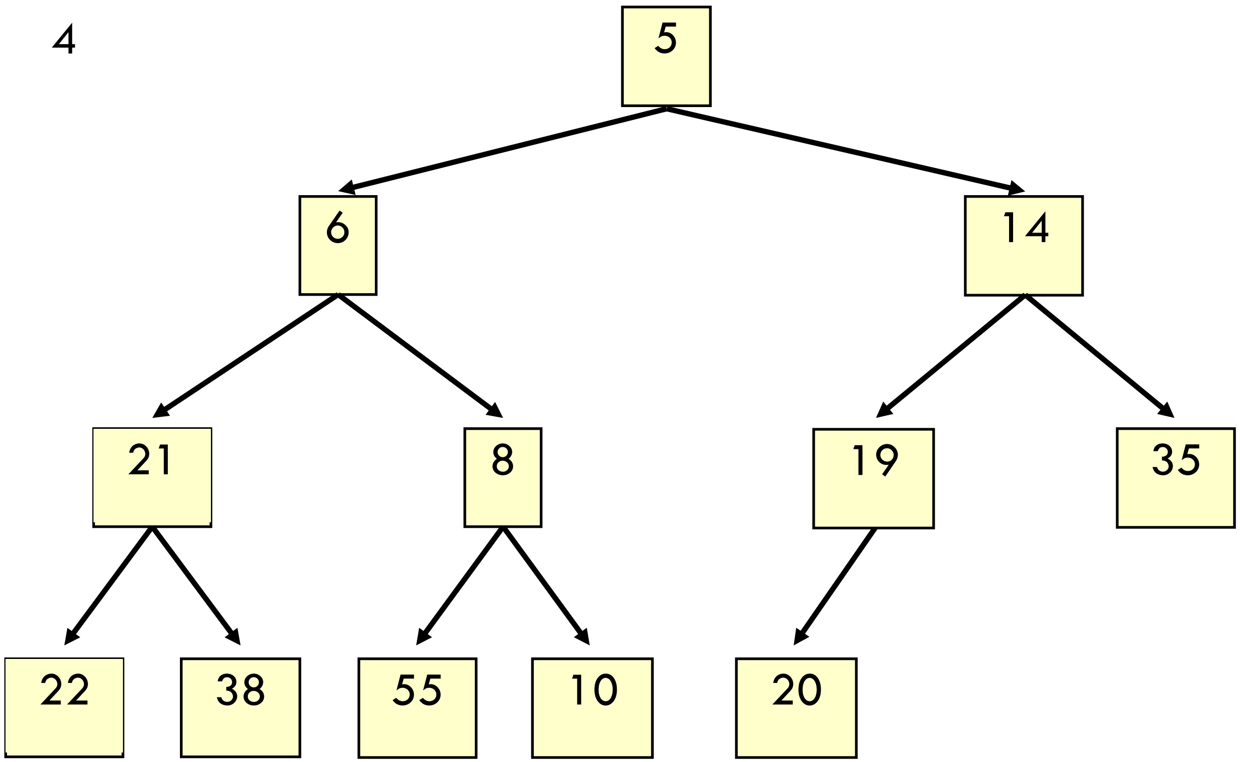


`extract()`



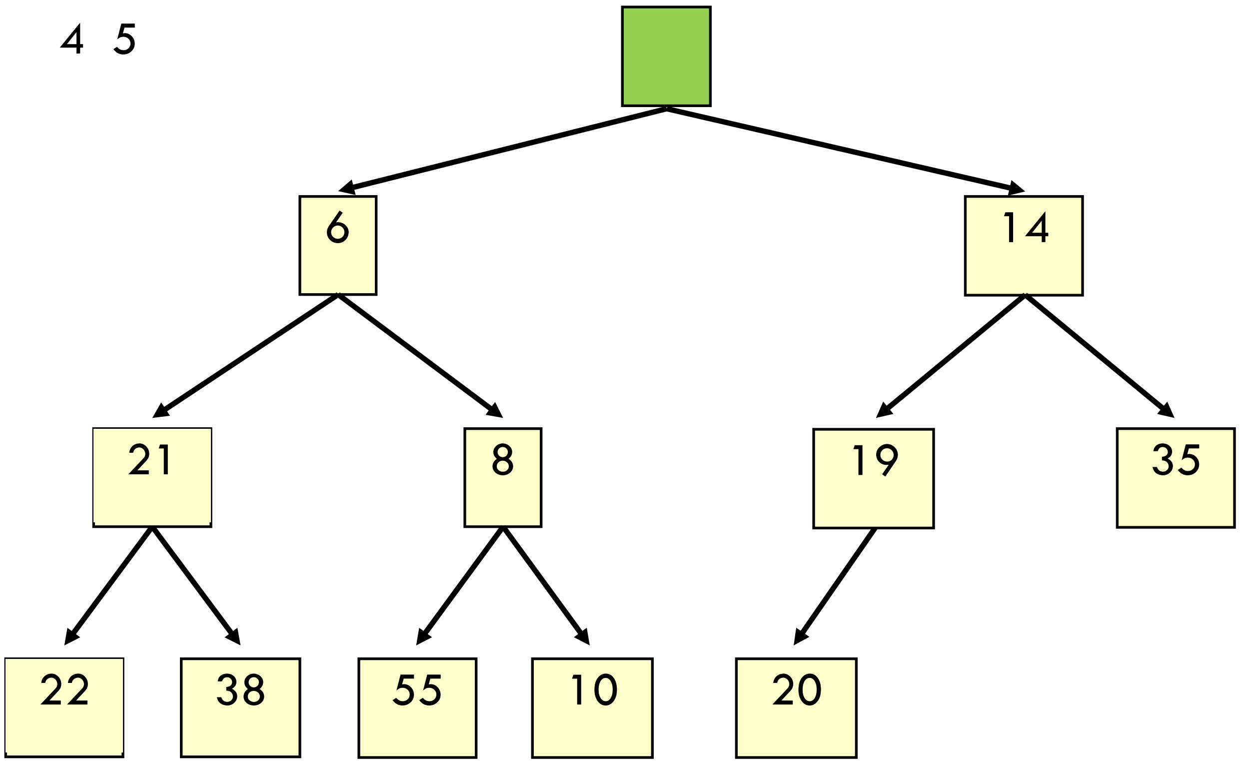
extract()

4



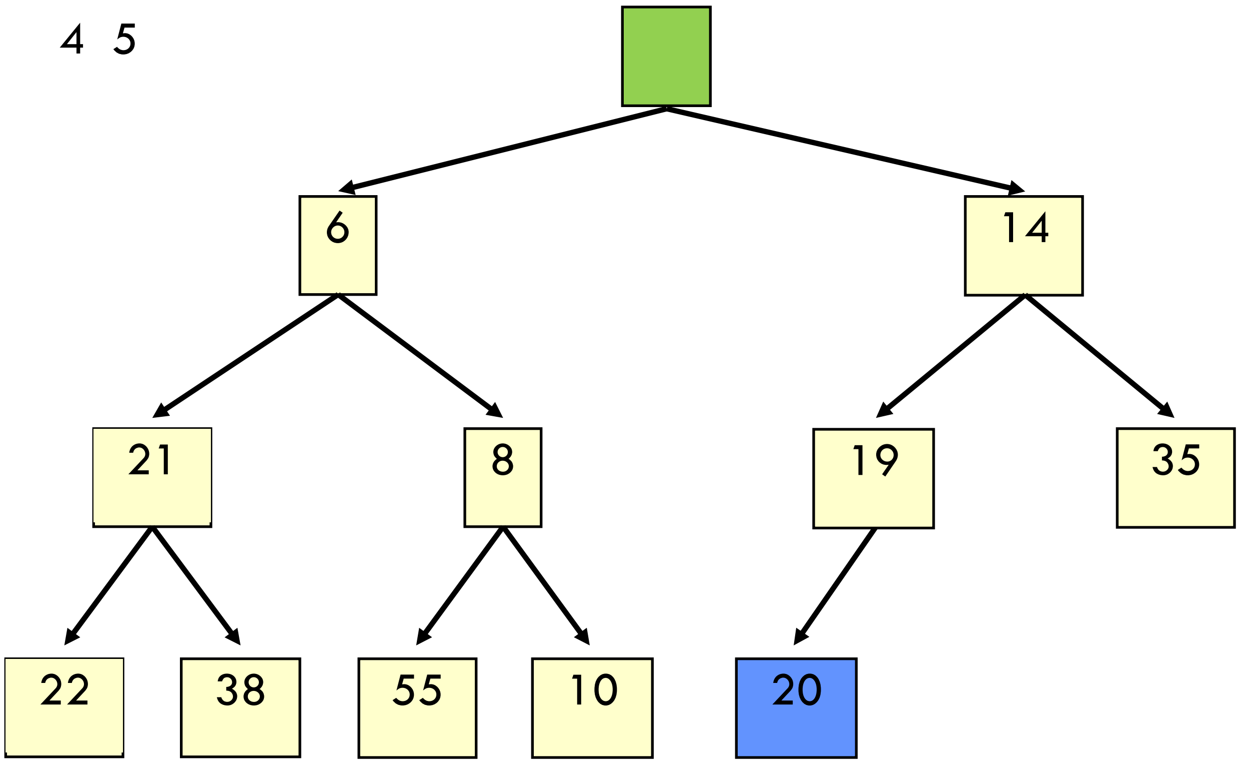
extract()

4 5



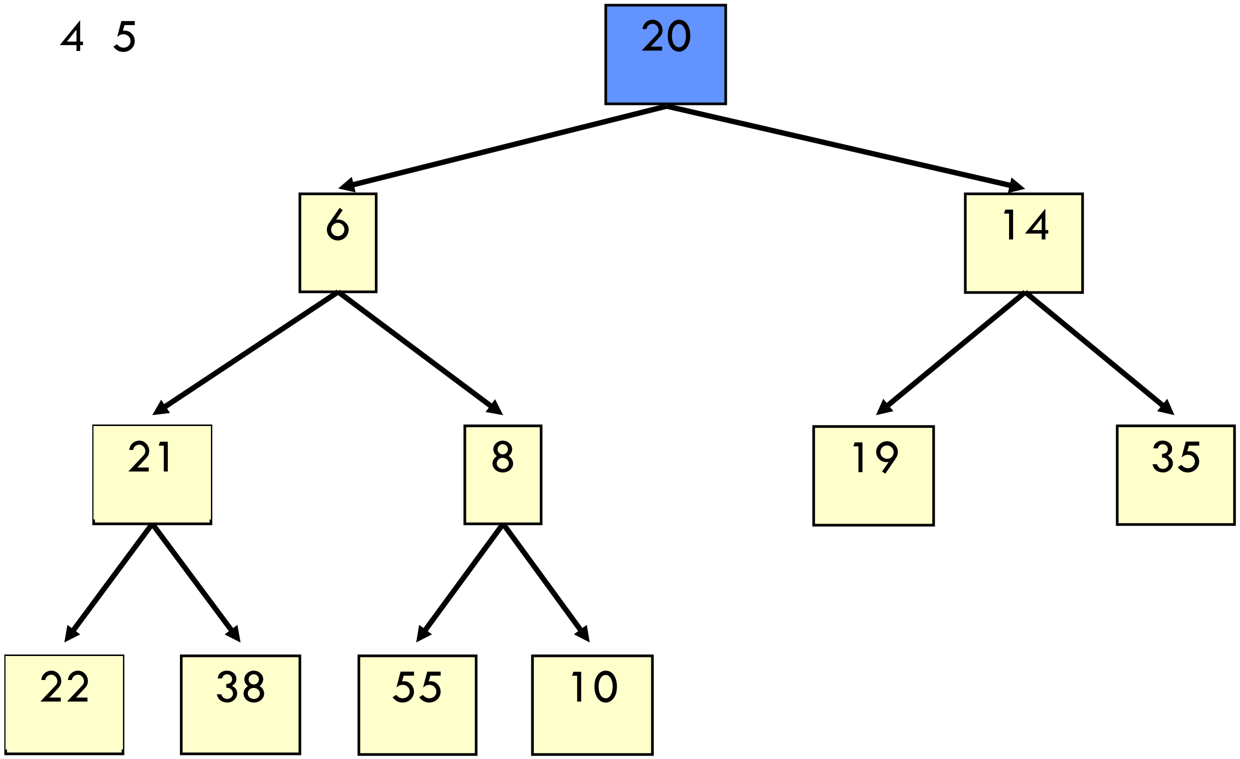
extract()

4 5



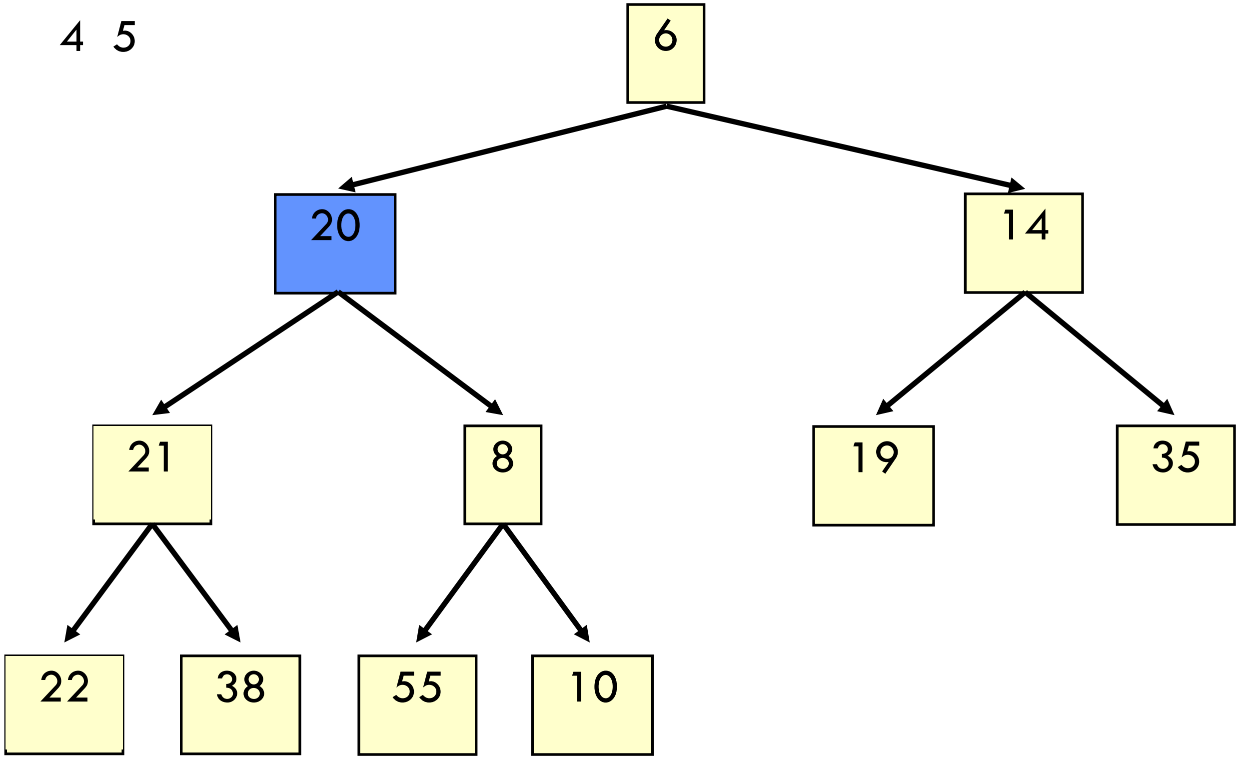
extract()

4 5

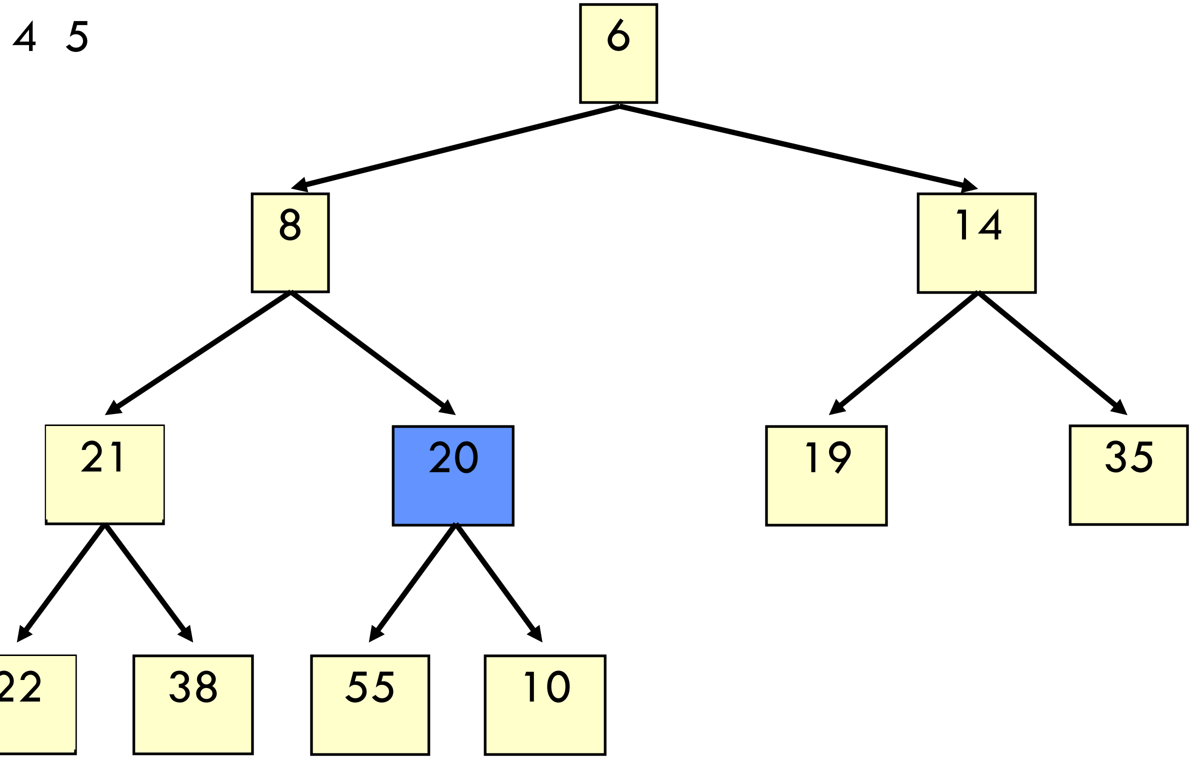


extract()

4 5

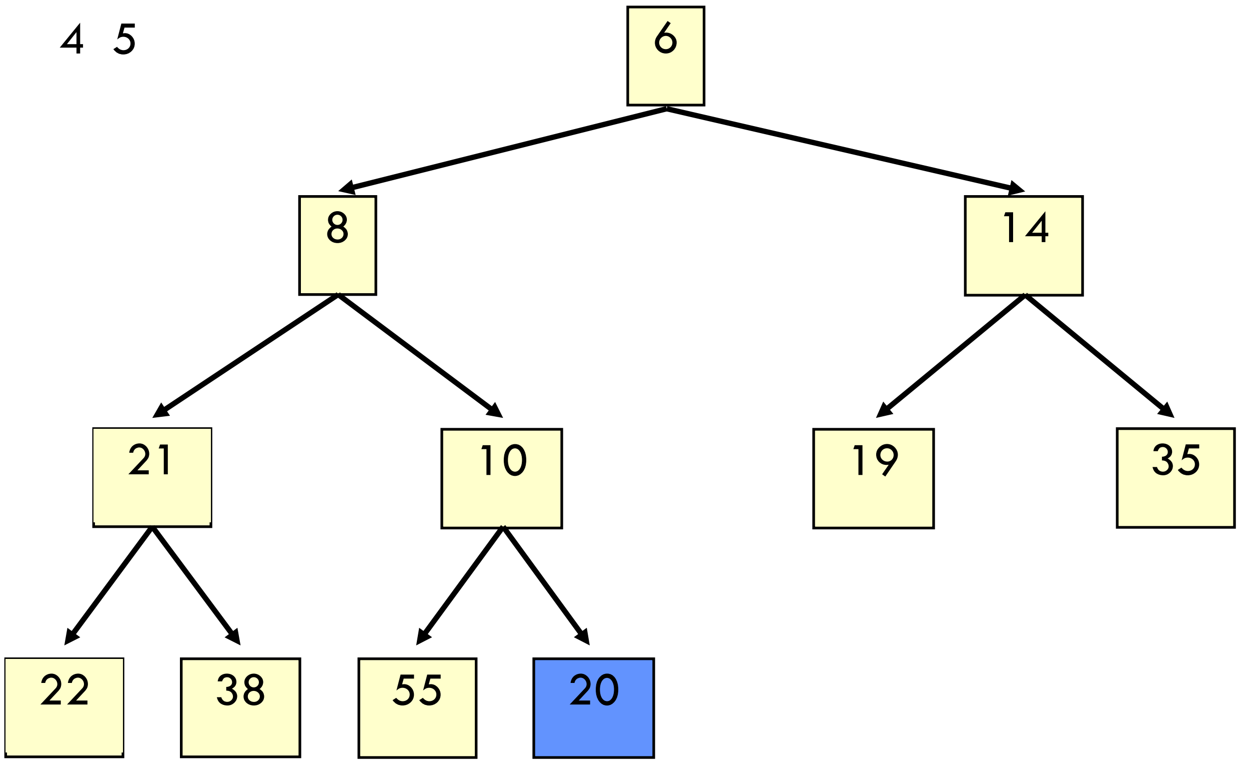


`extract()`



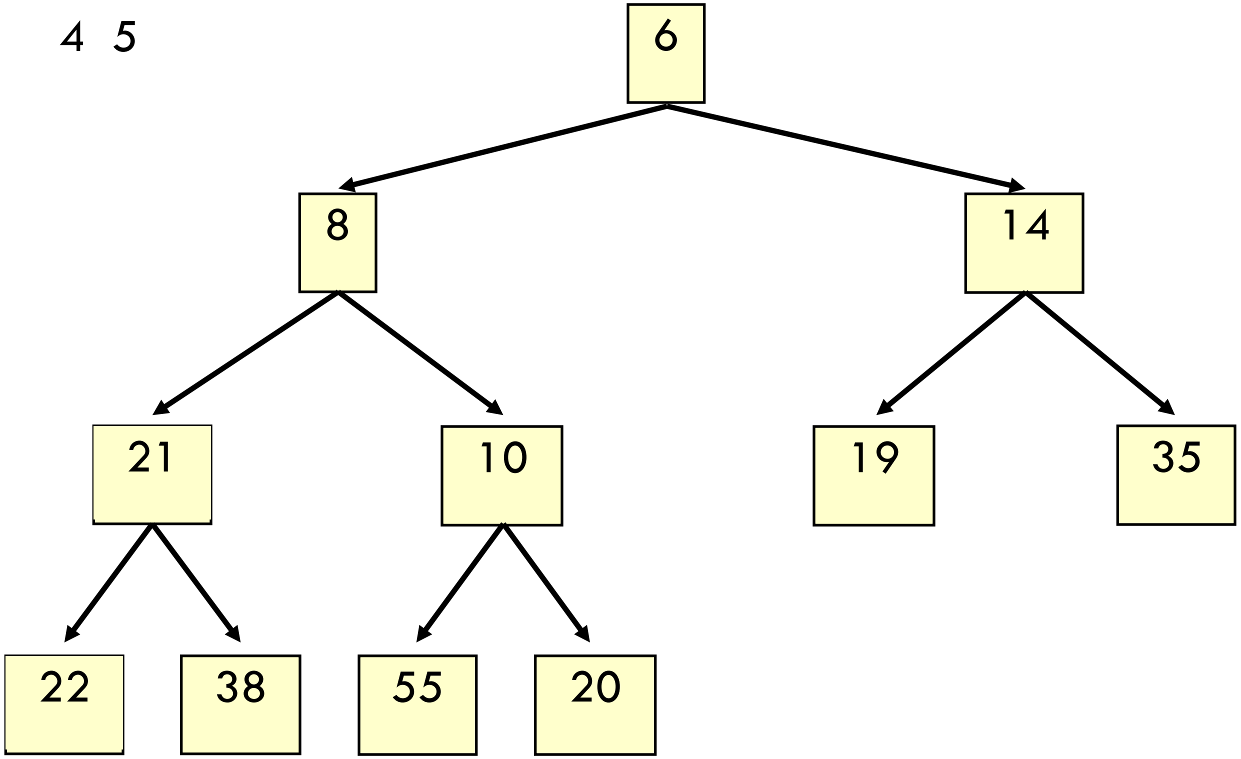
extract()

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 (called bucketing or chaining)
    - 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
- Be familiar with the alternative of bucketing: linear probing

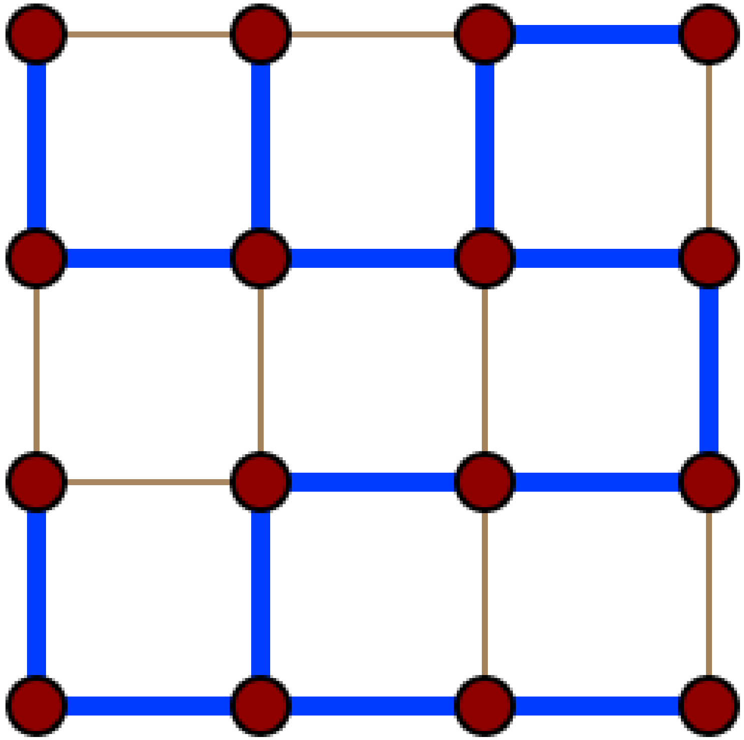
A HashMap is a particular implementation of the Map interface

Graphs & Graph Algorithms

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)

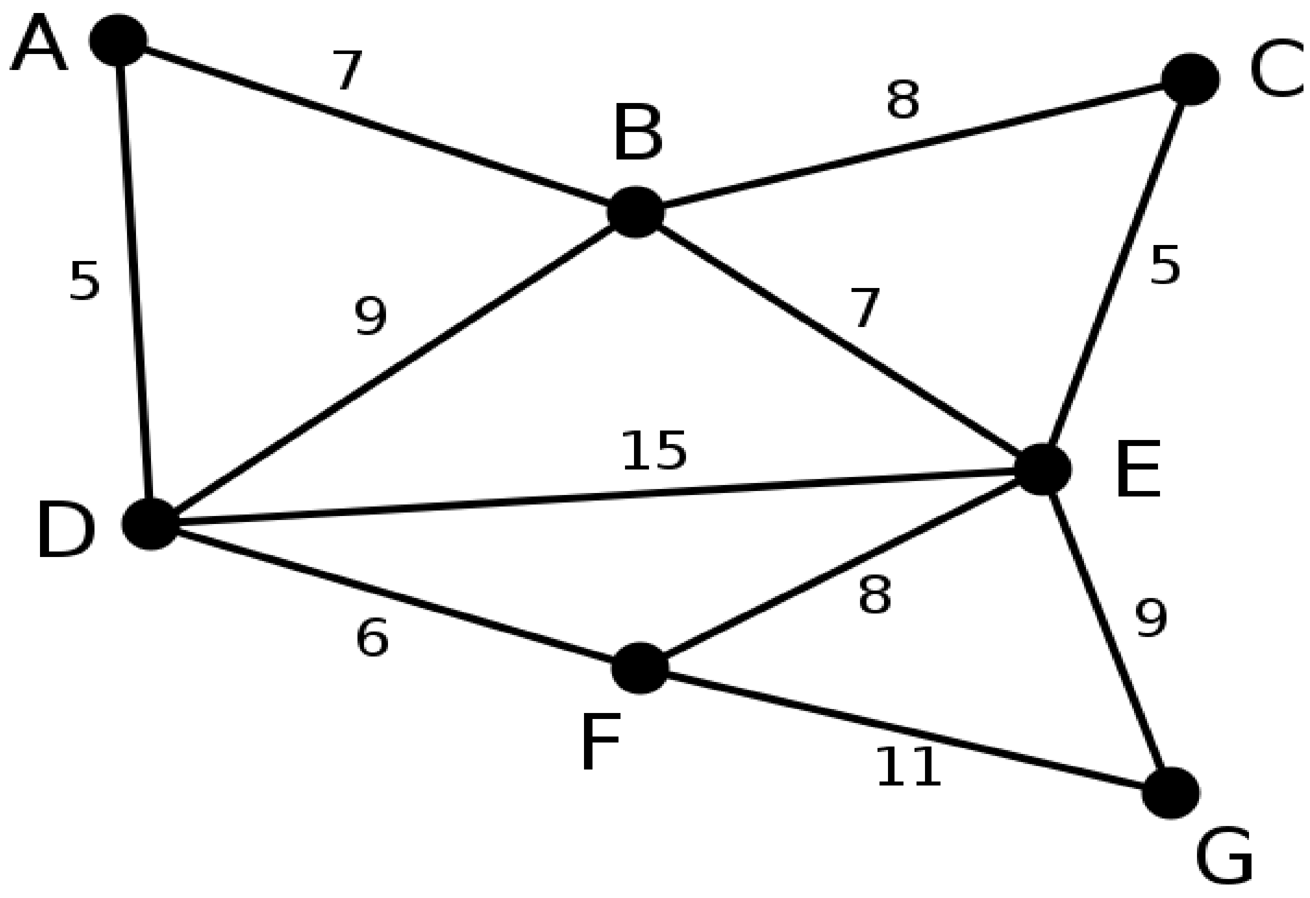
- Spanning tree with minimum sum edge weights
  - Prim's algorithm
  - Kruskal's algorithm
  - Not necessarily unique

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

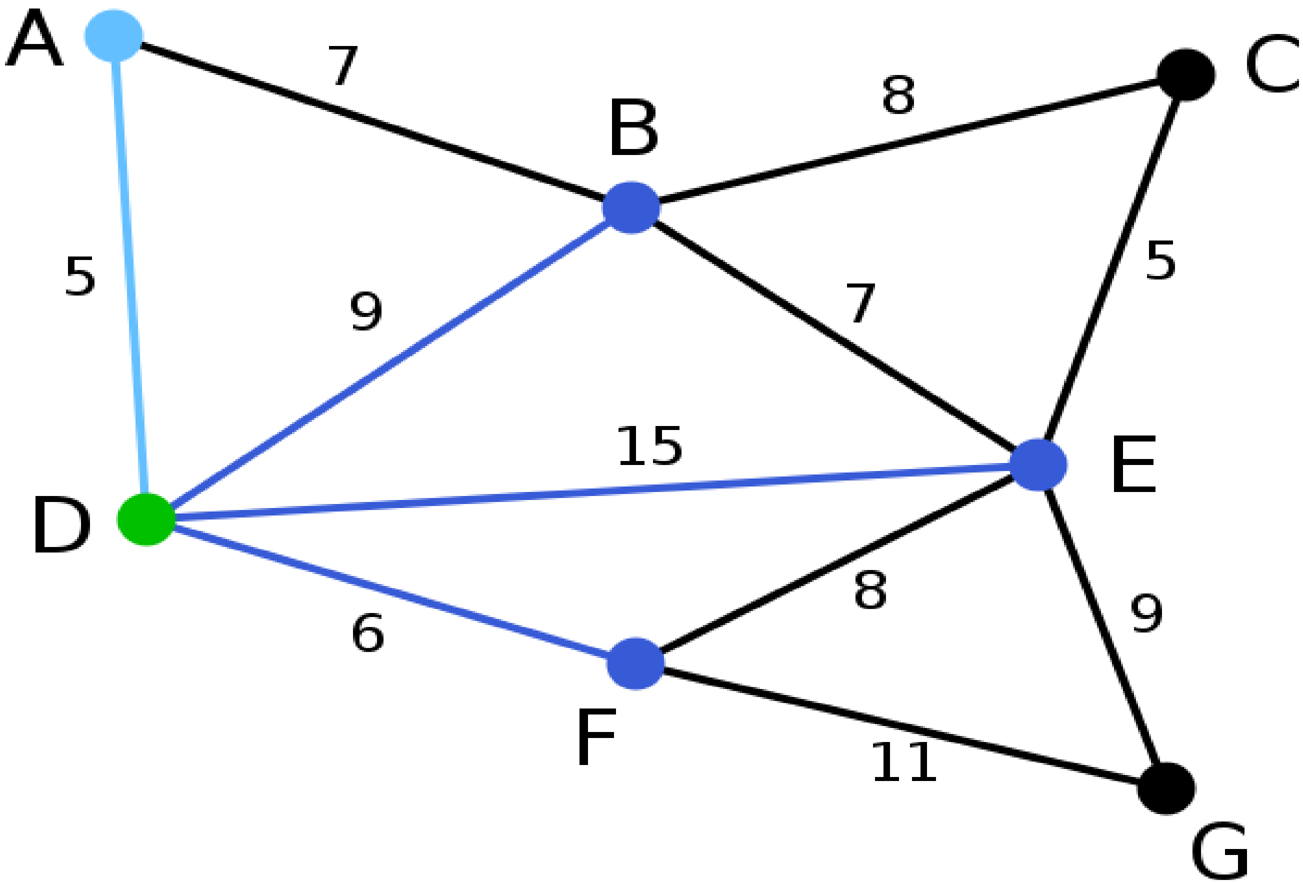
Prim's Algorithm Example

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



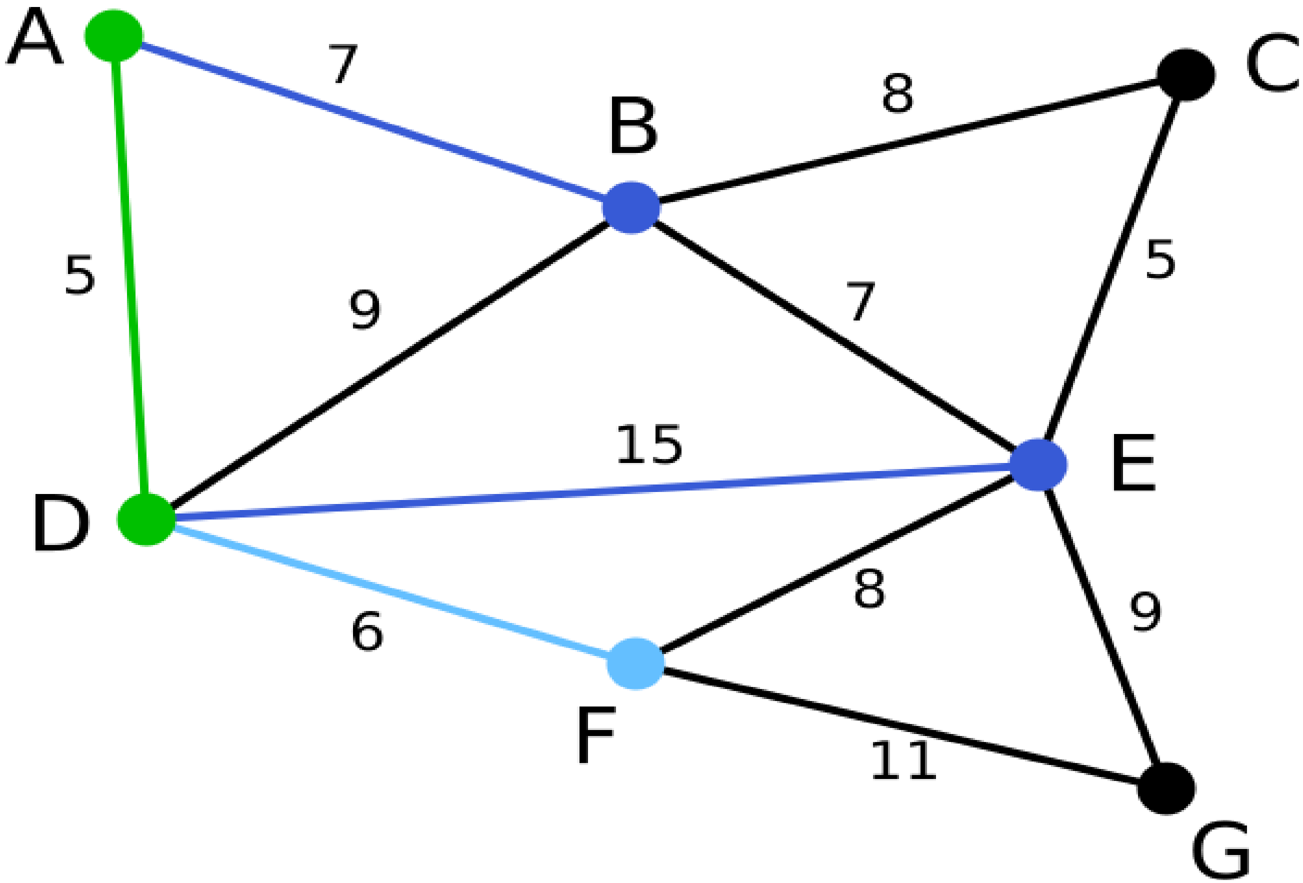
Prim's Algorithm Example

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.



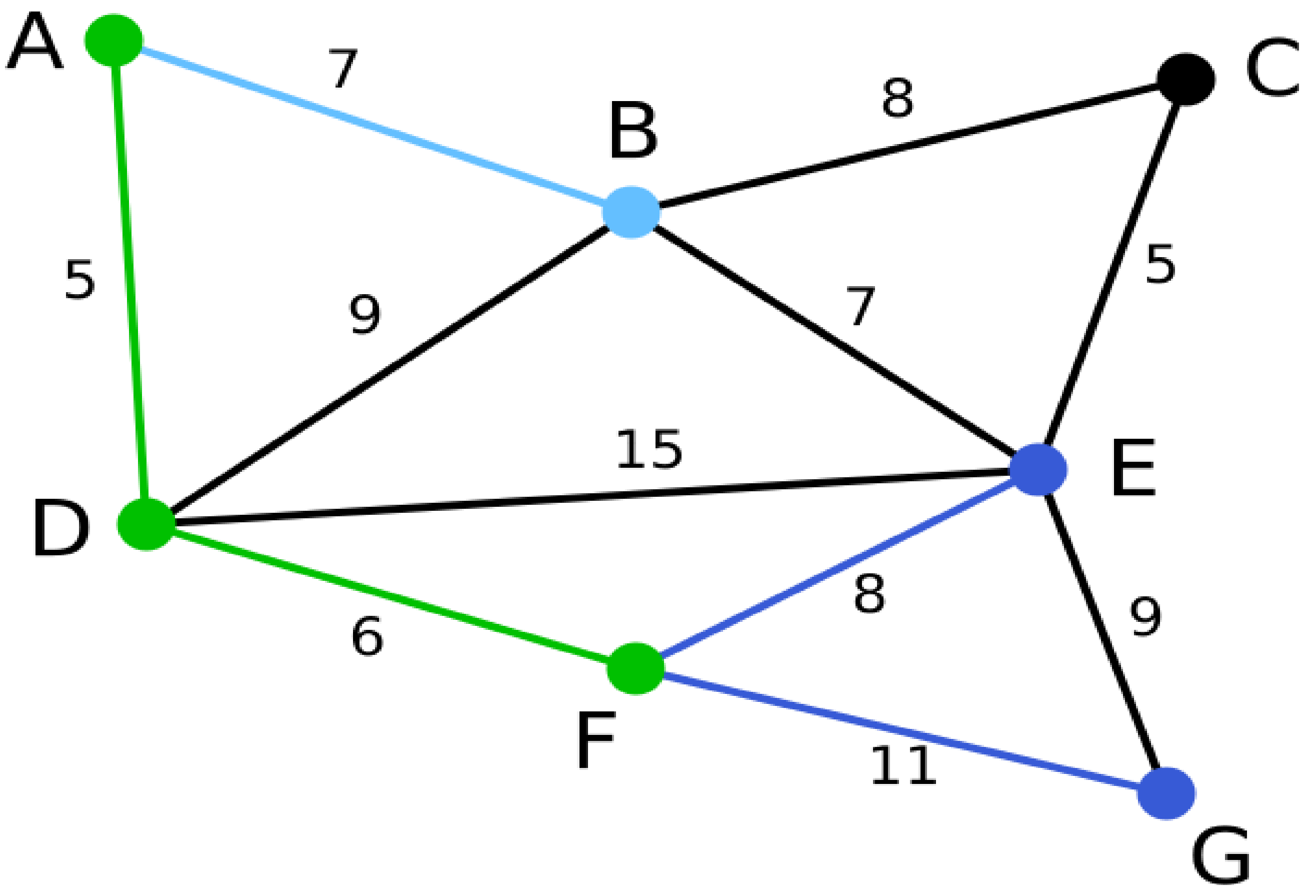
Prim's Algorithm Example

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

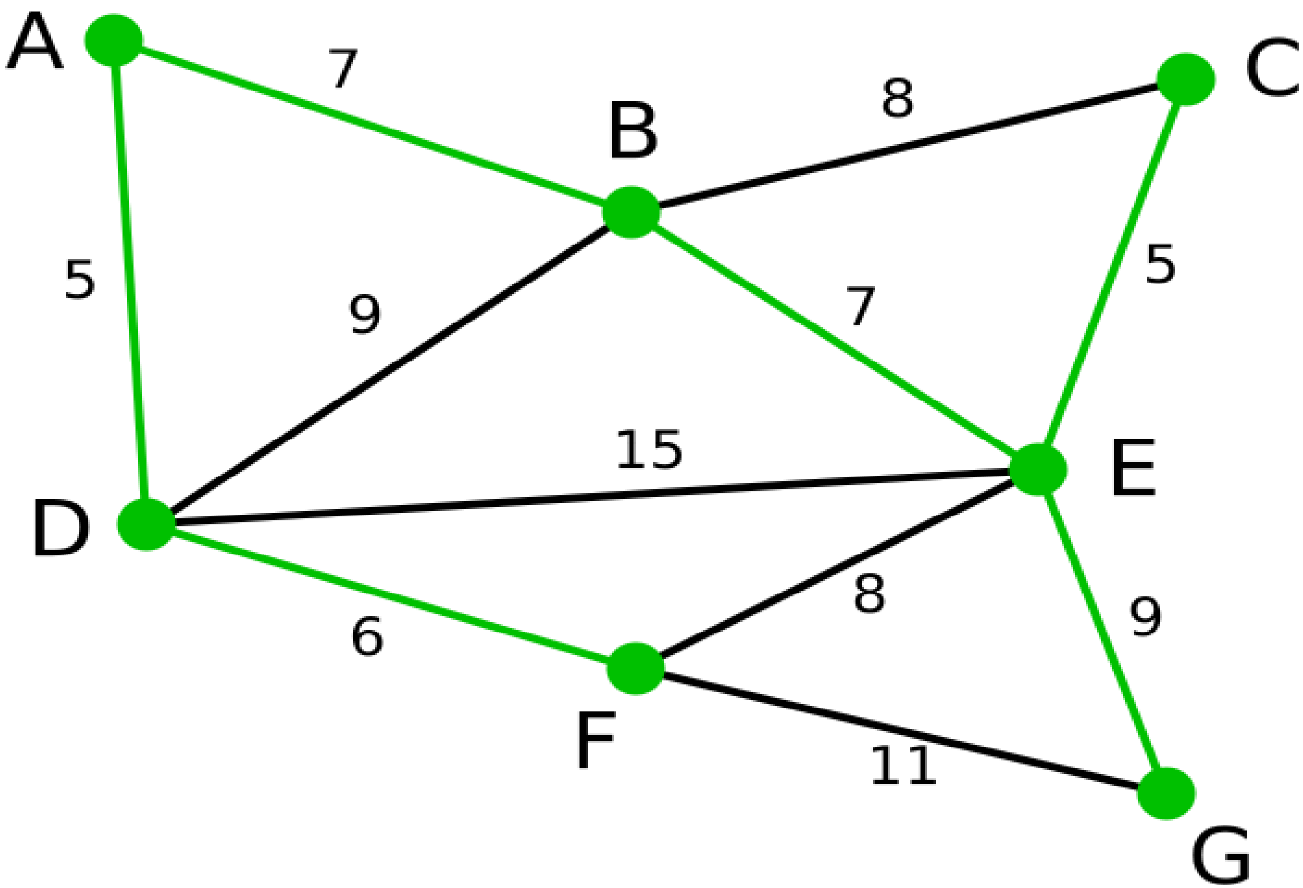
The algorithm carries on as above. Vertex B, which is 7 away from A, is highlighted.



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.



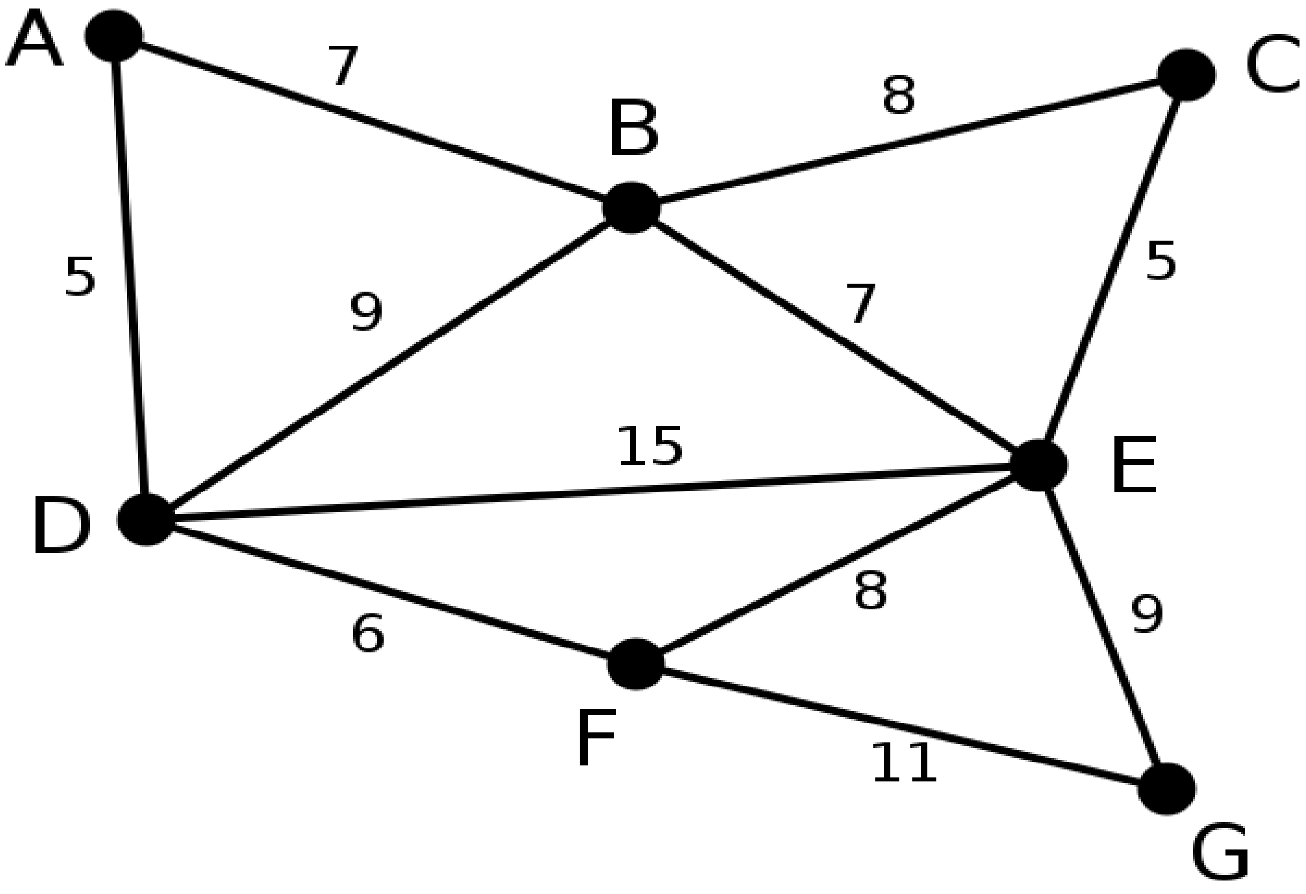
Kruskal's Algorithm

- 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

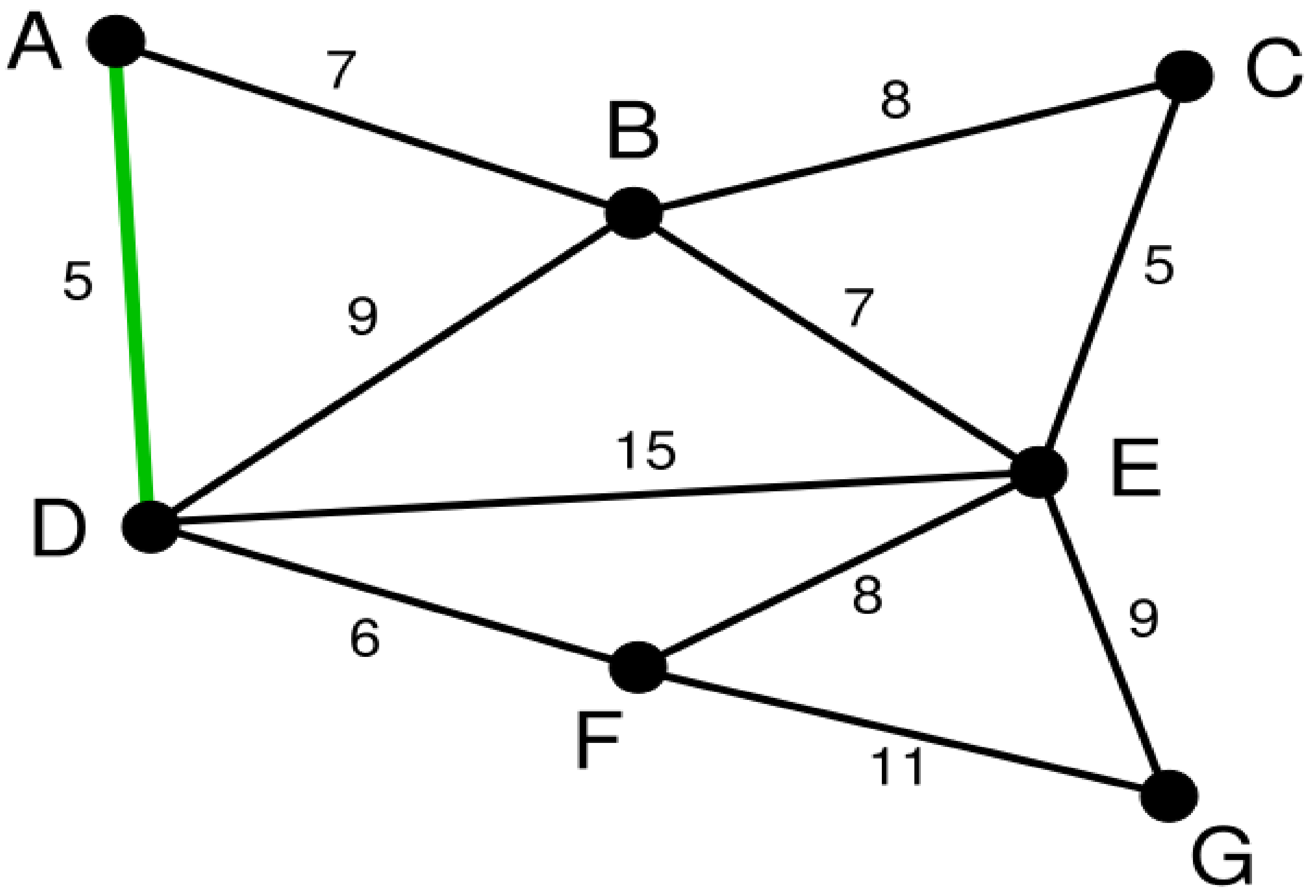
This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

∞



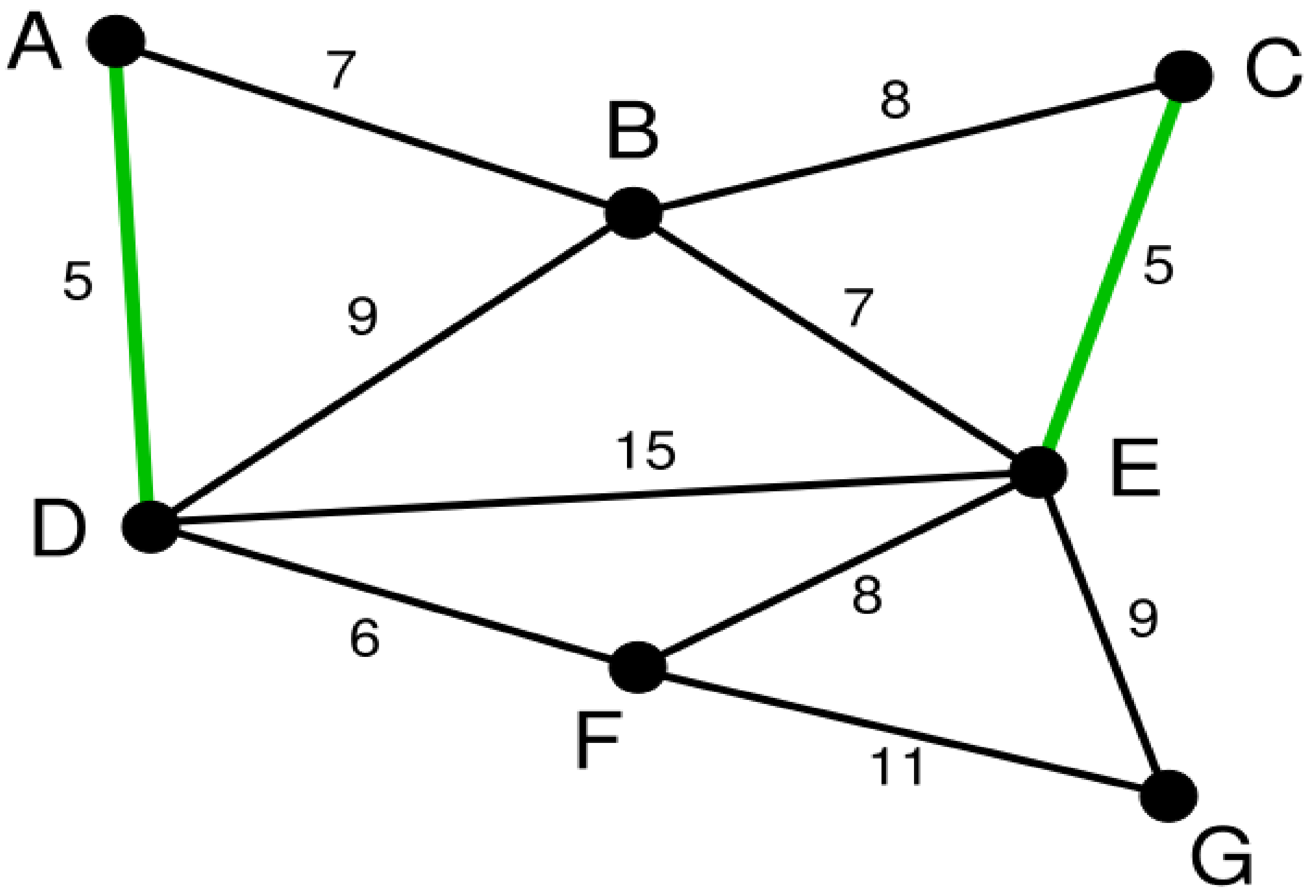
Kruskal's Algorithm Example

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



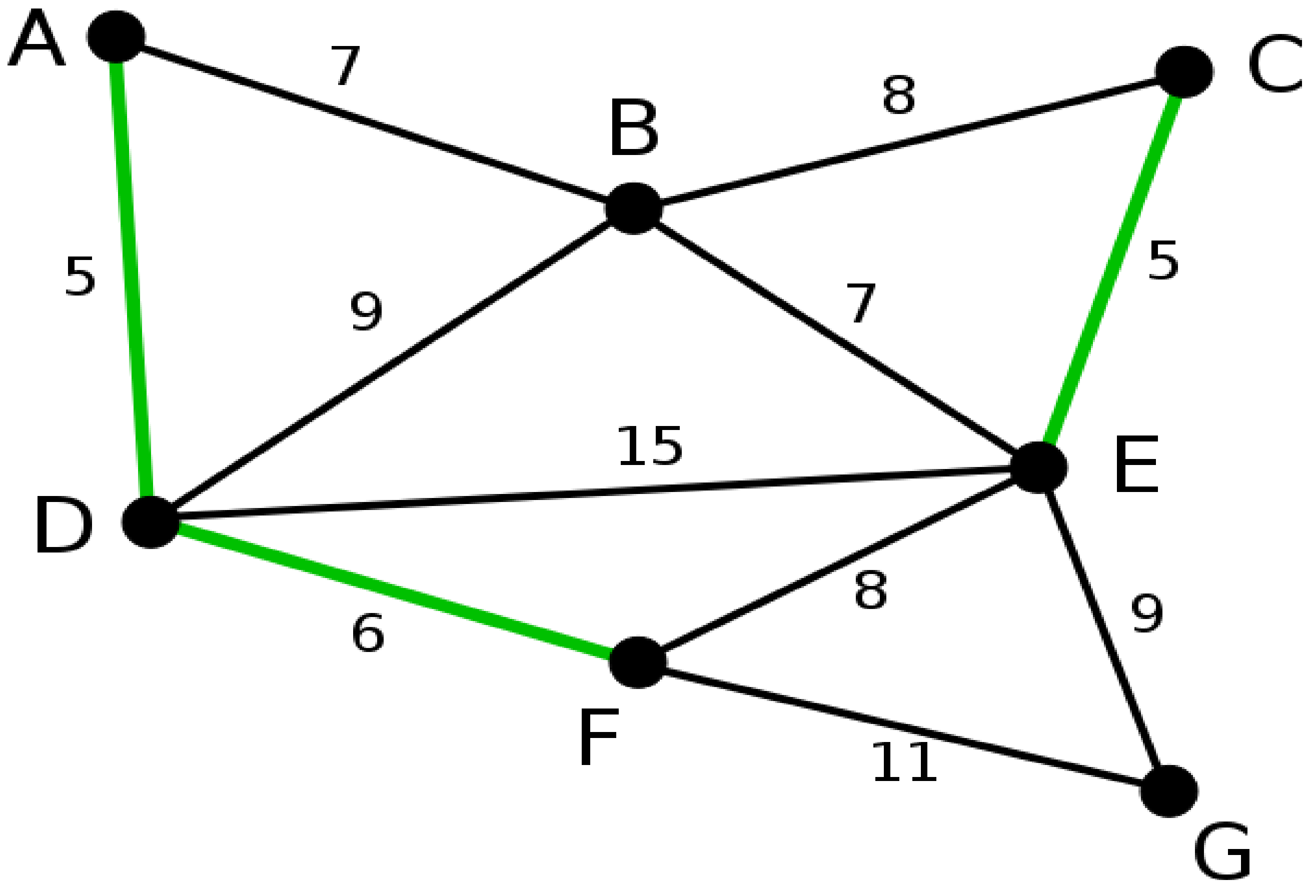
Kruskal's Algorithm Example

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



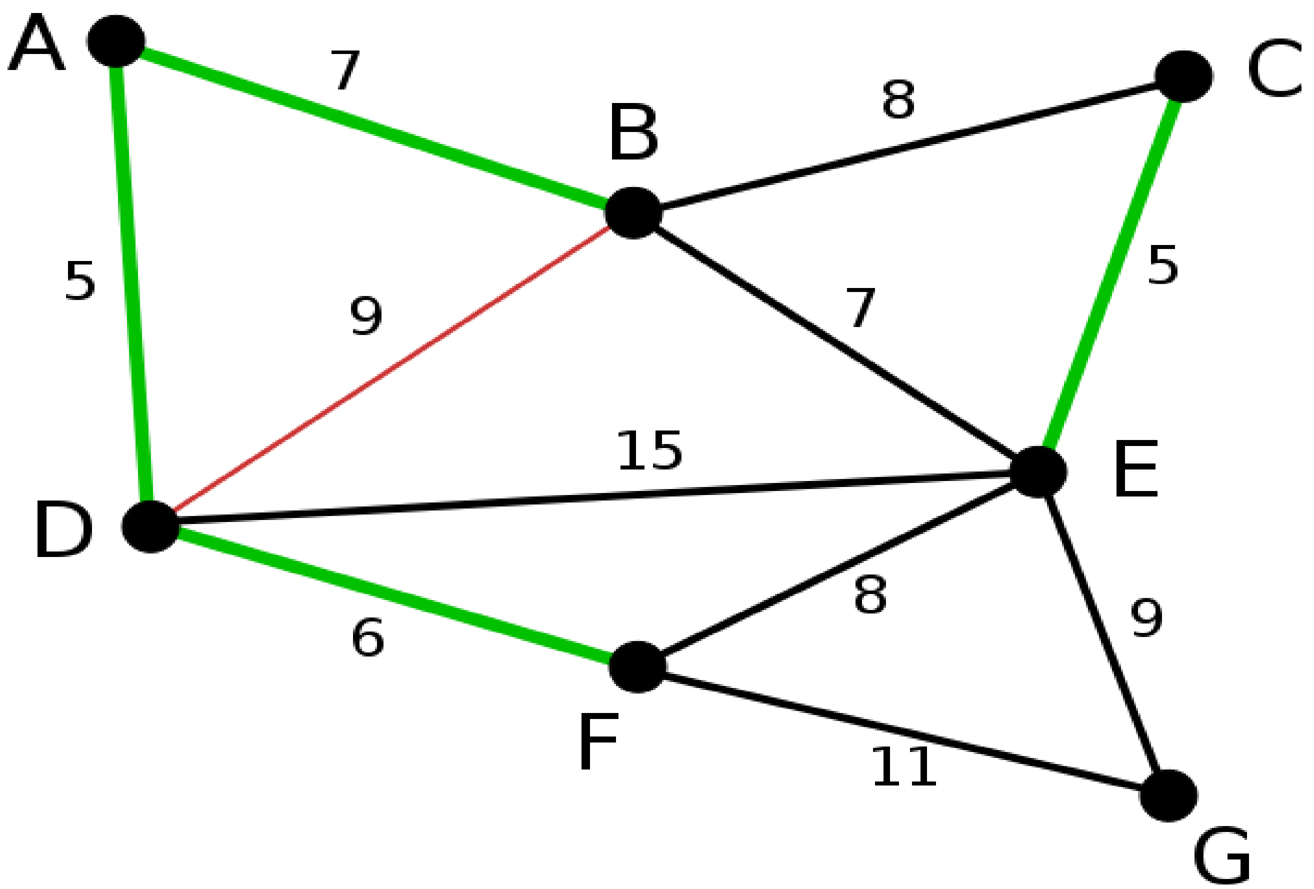
Kruskal's Algorithm Example

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



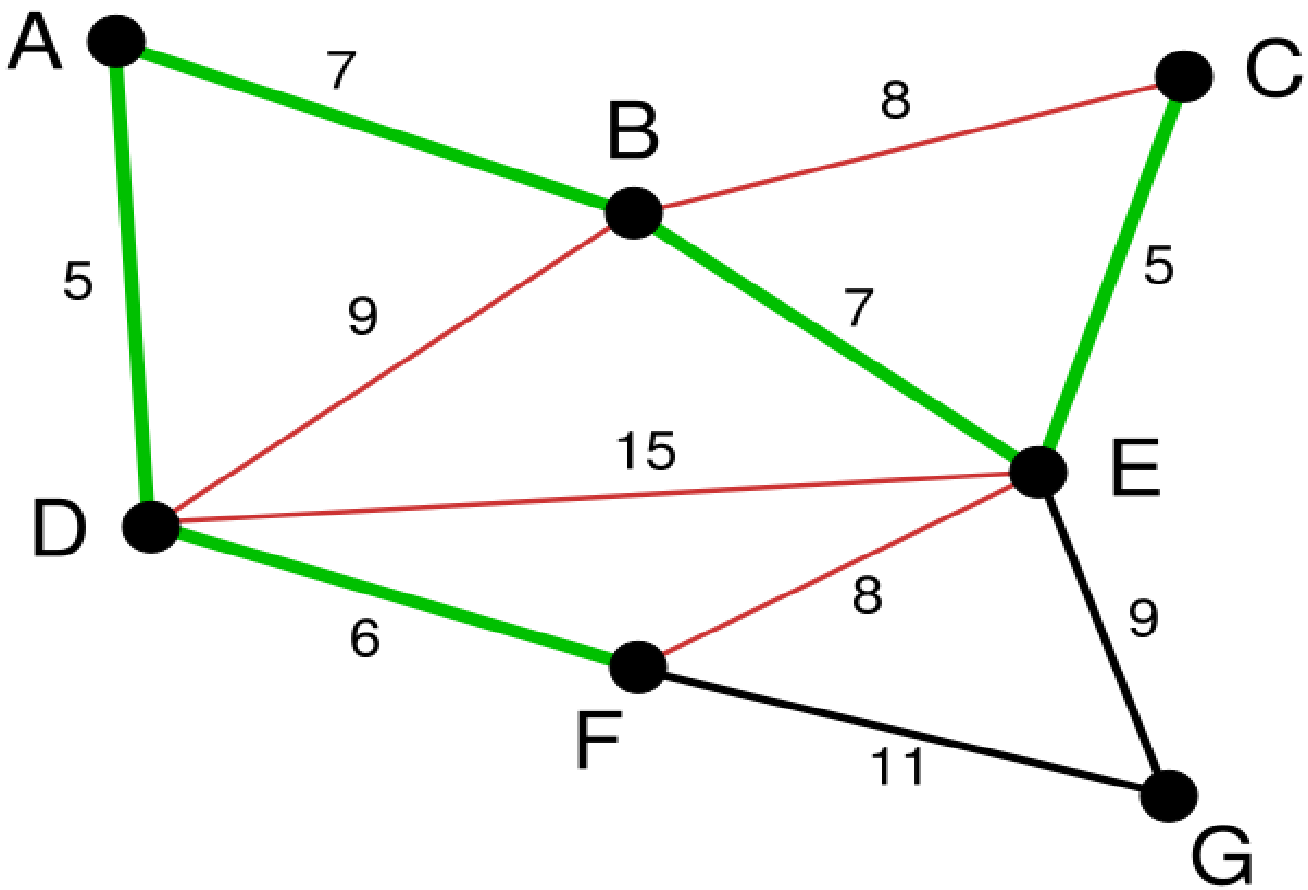
Kruskal's Algorithm Example

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.



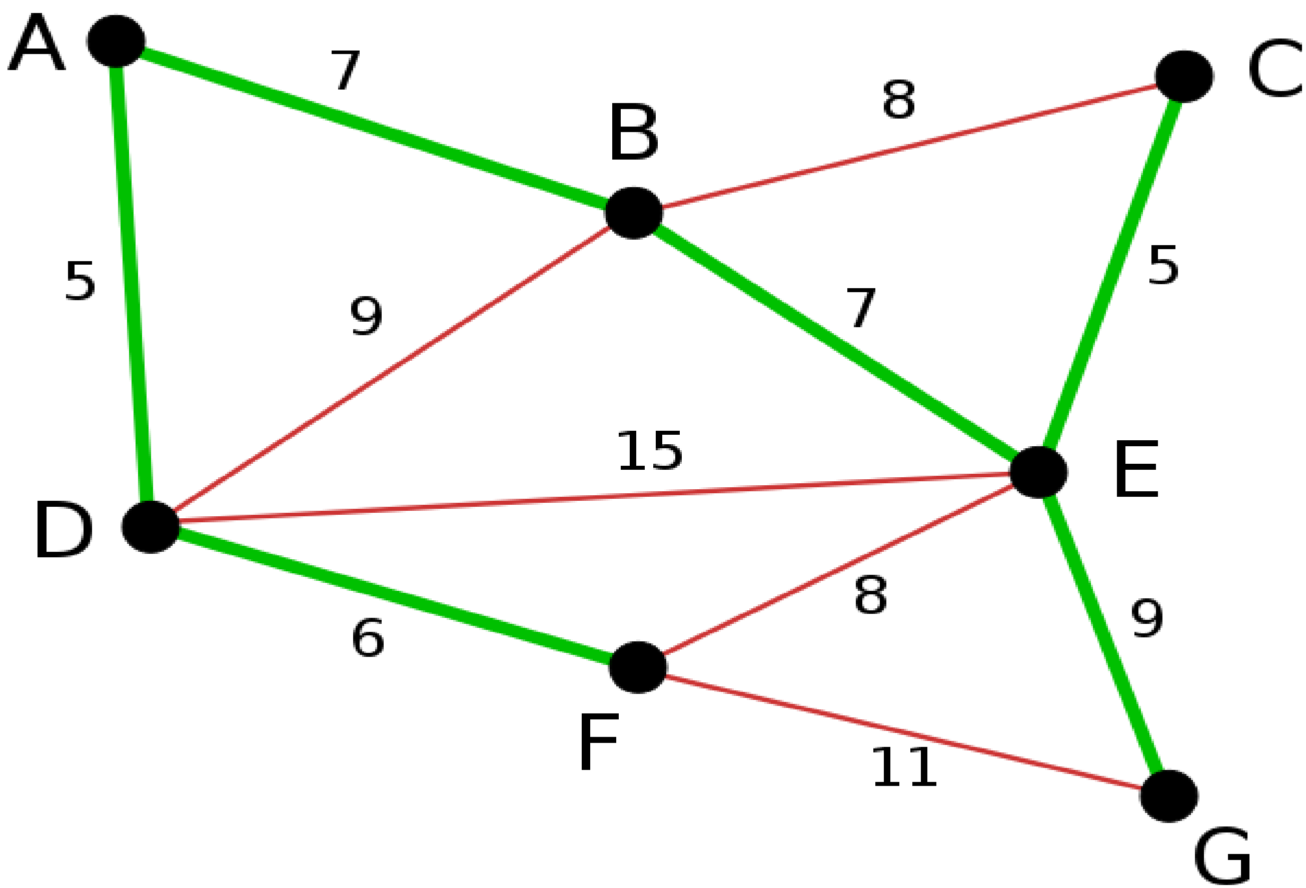
Kruskal's Algorithm Example

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

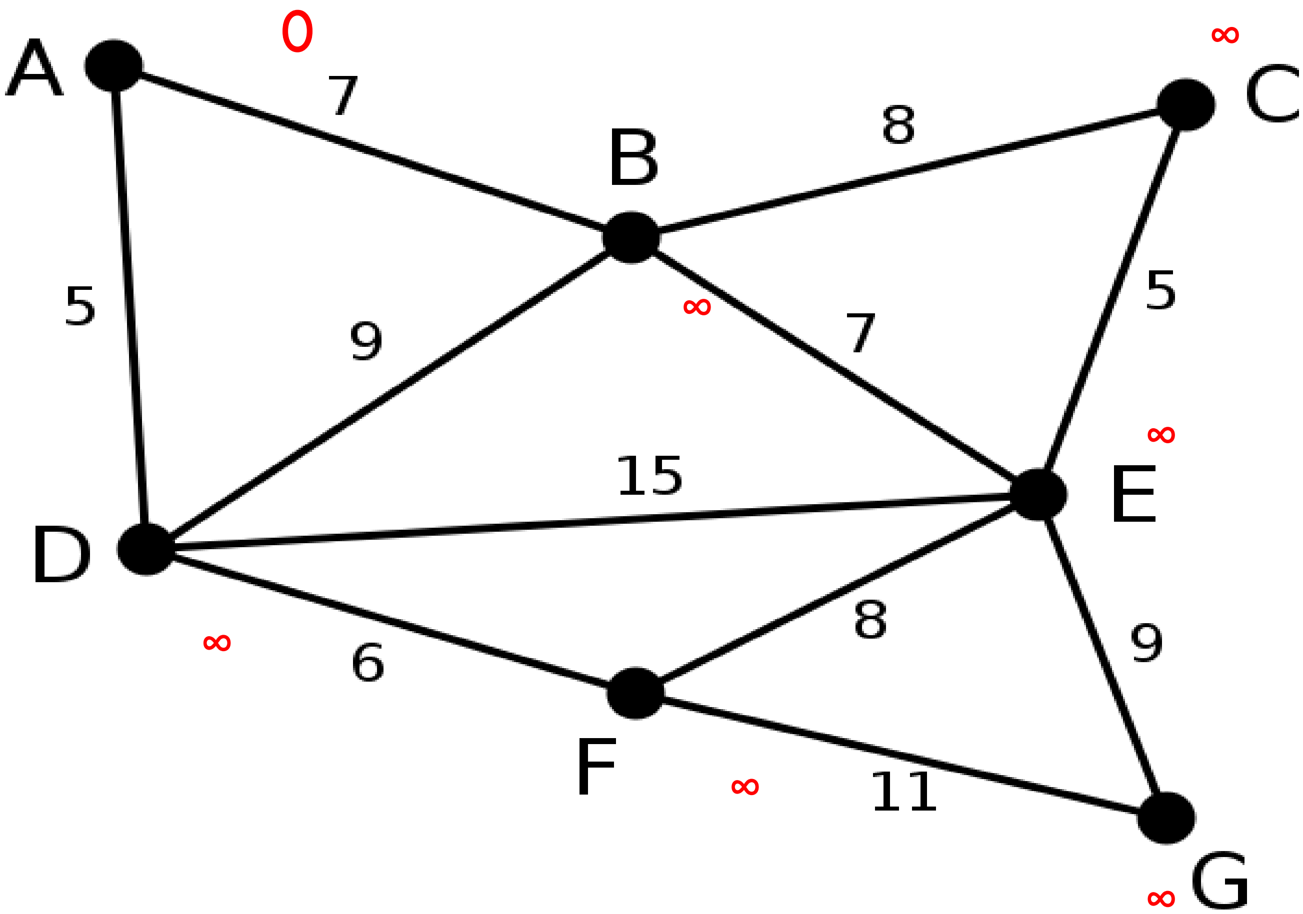
- Similar to Prim's algorithm
- Difference lies in the priority
  - Priority is the length of shortest path to a visited vertex + cost of edge to unvisited vertex
  - We know the shortest path to every visited vertex
- On unweighted graphs, BFS gives us the same result as Dijkstra's algorithm

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.

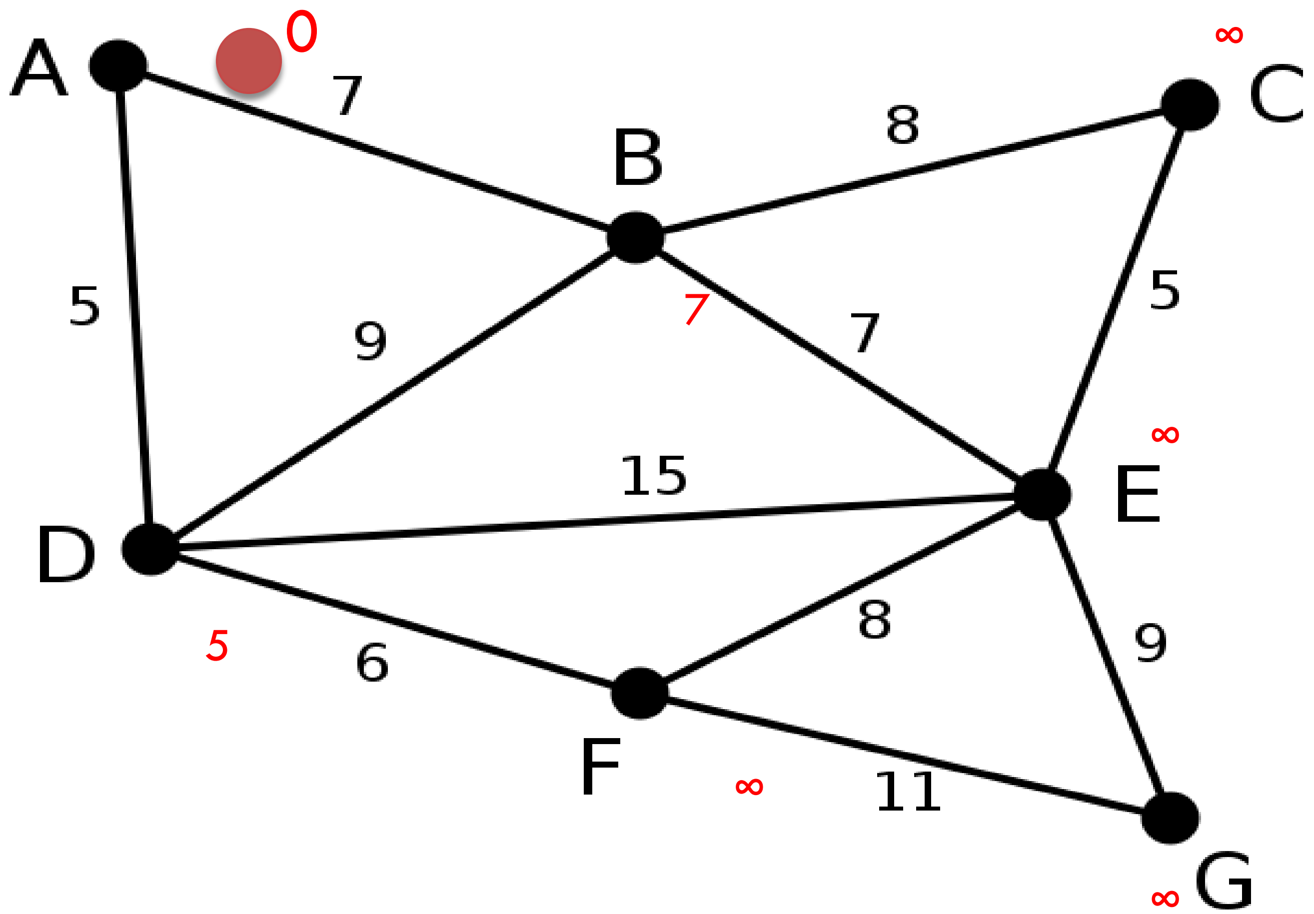
Dijkstra's Algorithm Example

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

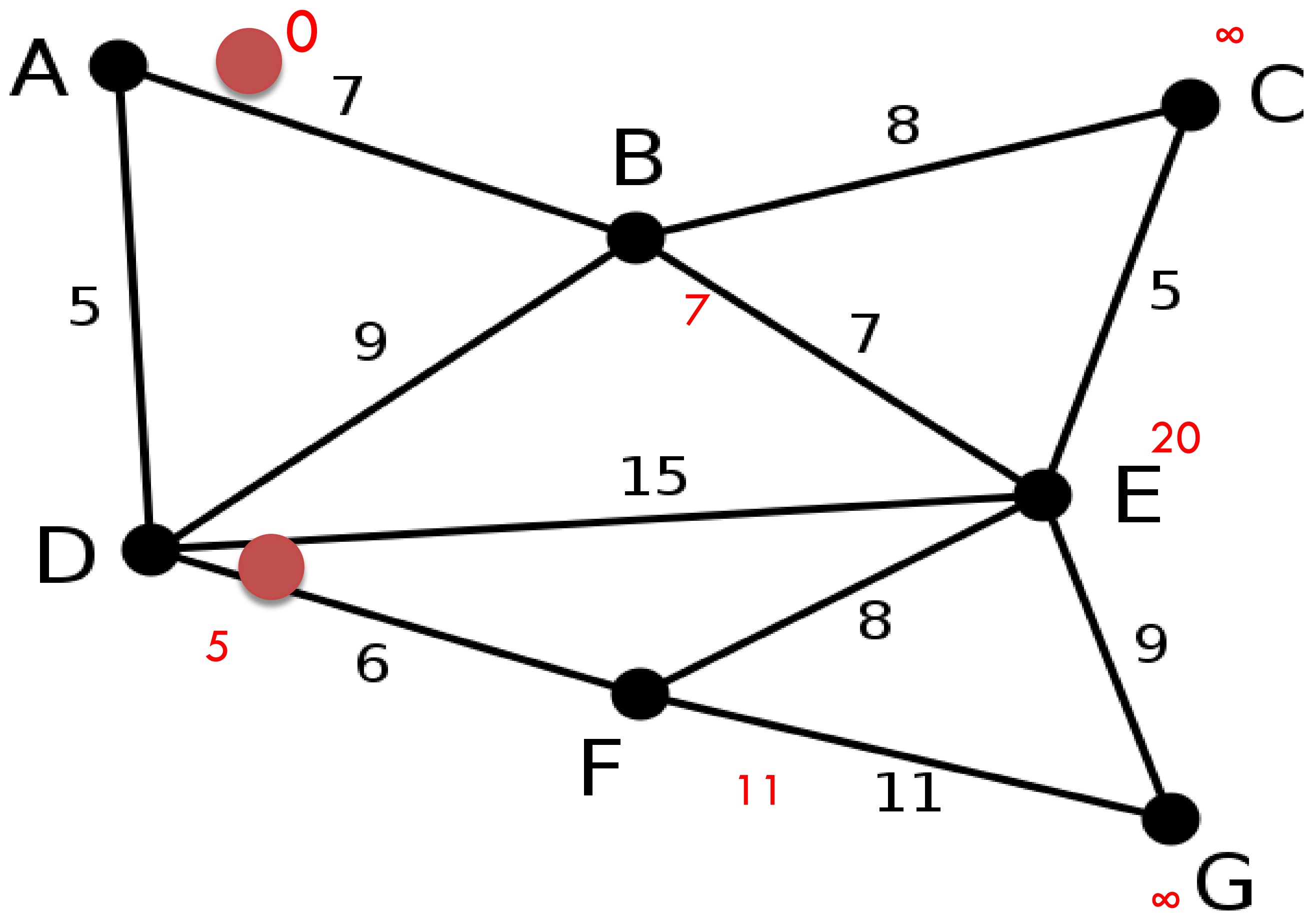
Select the node is with the smallest distance that isn't done, and update the distances to its neighbors.

F = 11 : 5 + 6 = 11

B = 7: 5 + 9 = 14 > 7

E = 20: 5 + 15 = 20

Mark D as visited.



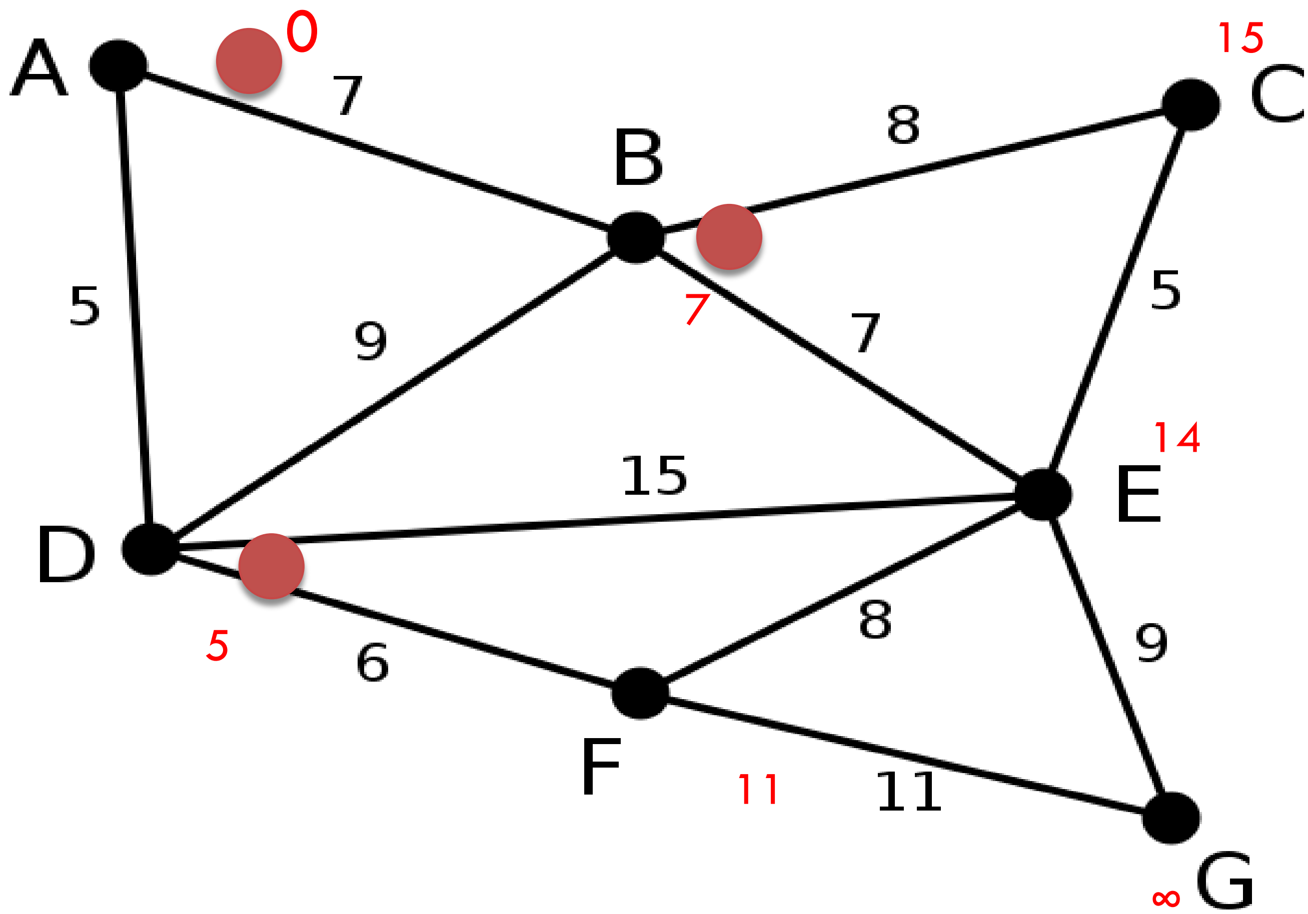
Dijkstra's Algorithm Example

Set the current node to B.

E = 14: 7 + 7 = 14

C = 15: 7 + 8 = 15

Mark B as visited.



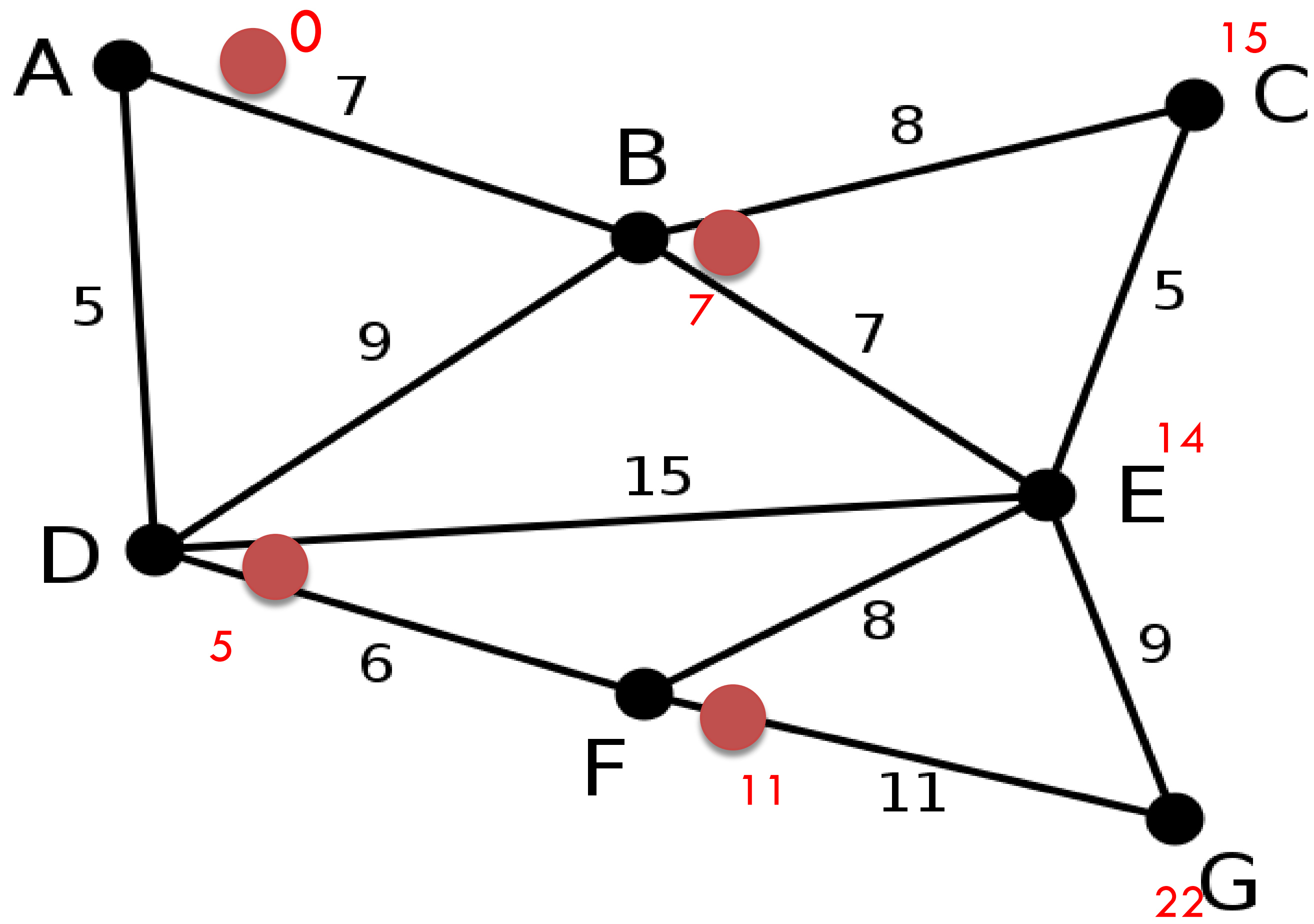
Dijkstra's Algorithm Example

Repeat the process:

E = 14: 11 + 8 = 19 > 14

G = 22: 11 + 11 = 22

Mark F as visited



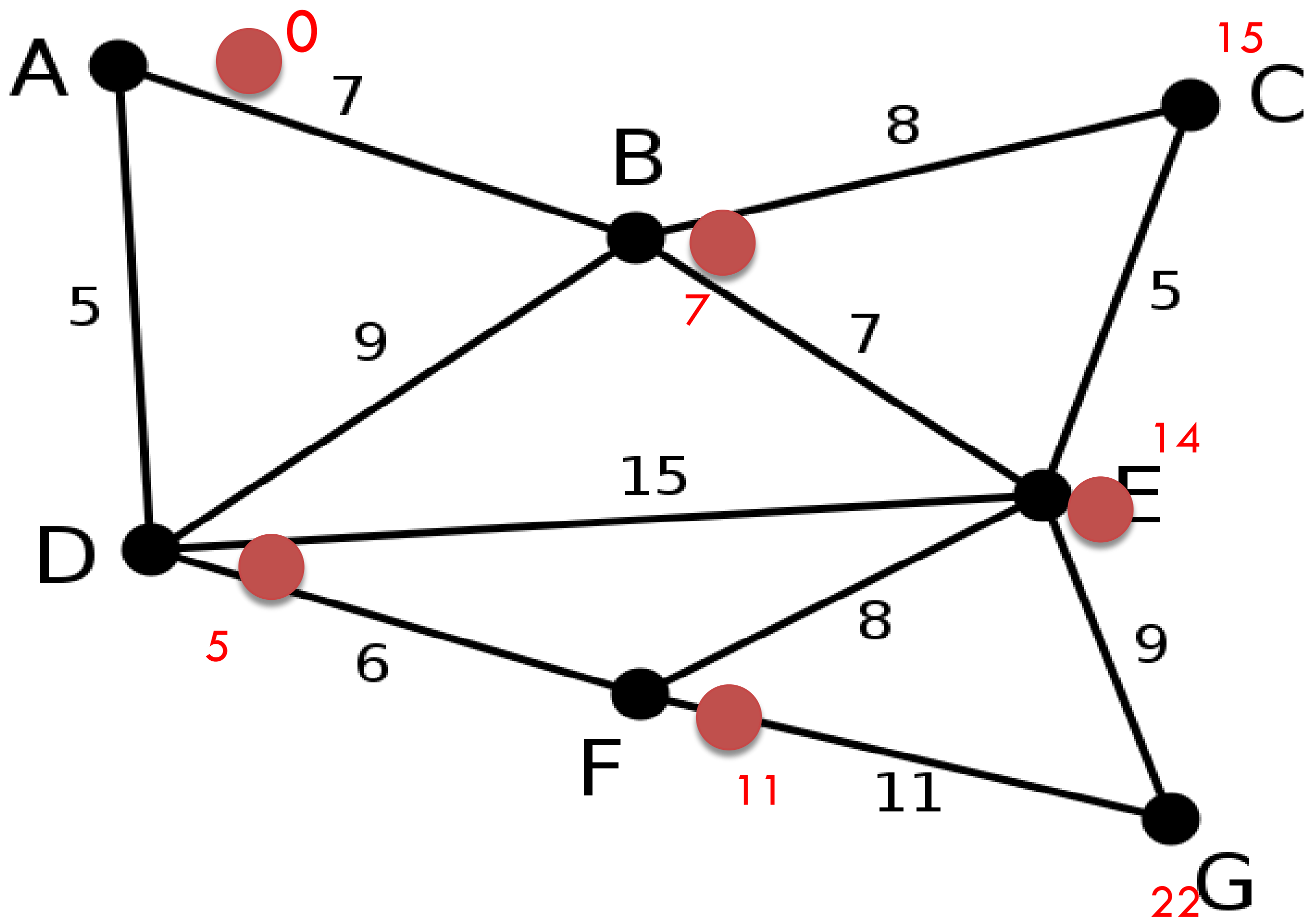
Dijkstra's Algorithm Example

Repeat the process:

C = 15: 14 + 5 = 19 > 15

G = 22: 14 + 9 = 23 > 22

Mark E as visited



Sorting

http://www.sorting-algorithms.com/

Question Time

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