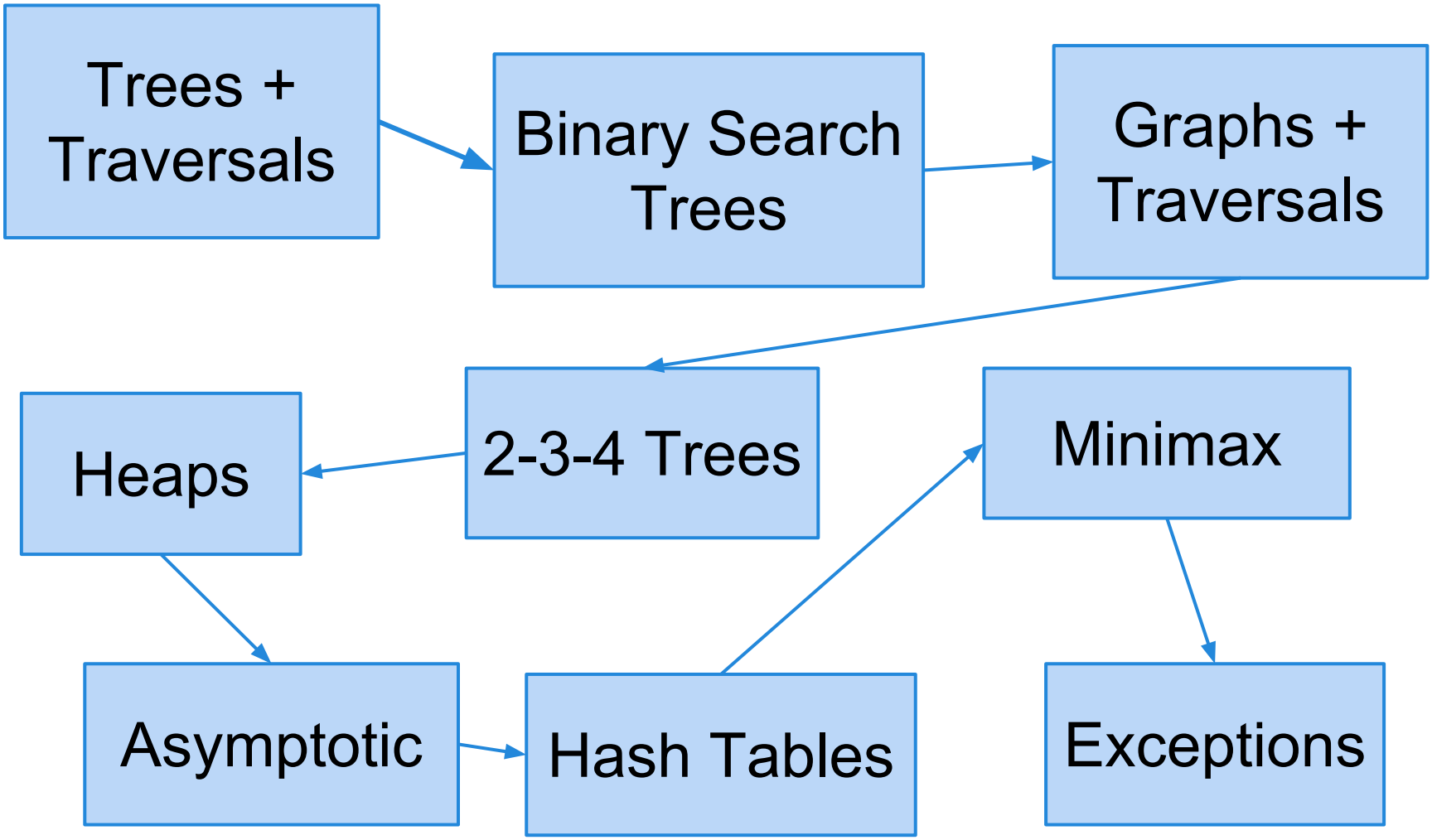
CS61B MT2 Review

Mona Gupta, Kevin Tee, Iris Wang, Evan Ye

4/10/14

Roadmap


Trees + Traversals
Binary Search Trees
Graphs + Traversals
Heaps
2-3-4 Trees
Minimax
Asymptotic
Hash Tables
Exceptions

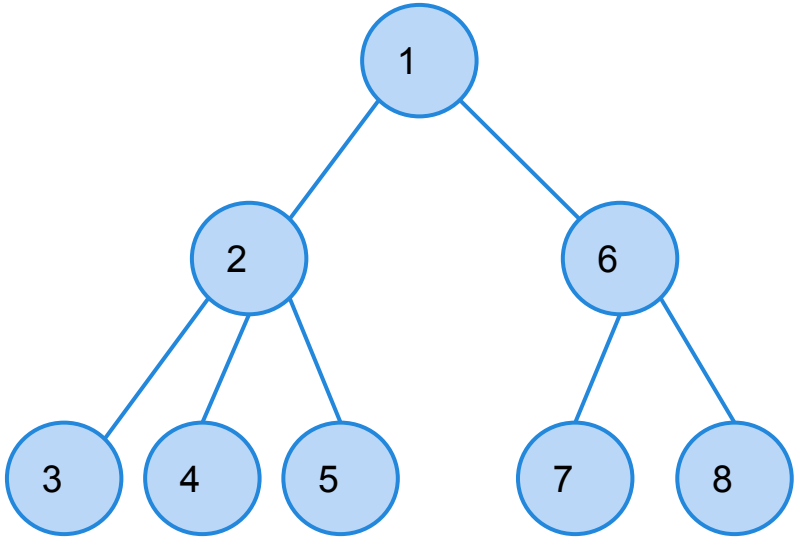
Tree Traversals

Tree Traversals: pre-order

- Visit each node before recursively visiting its children, which are visited left to right.

Code:

```
class SibTreeNode {
  public void preorder() {
    this.visit();
    if (firstChild != null) {
      firstChild.preorder();
    }
    if (nextSibling != null) {
      nextSibling.preorder();
    }
  }
}
```

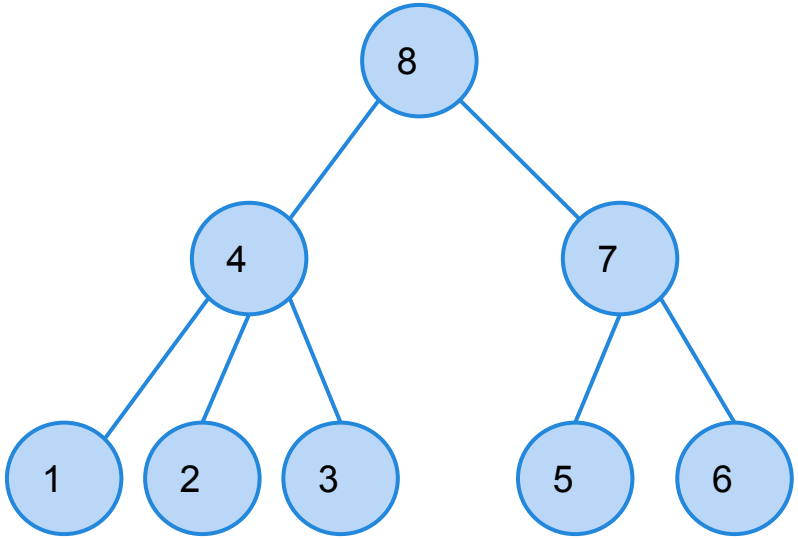


Tree Traversals: post-order

- Visit each node's children (in left-to-right order) before the node itself.

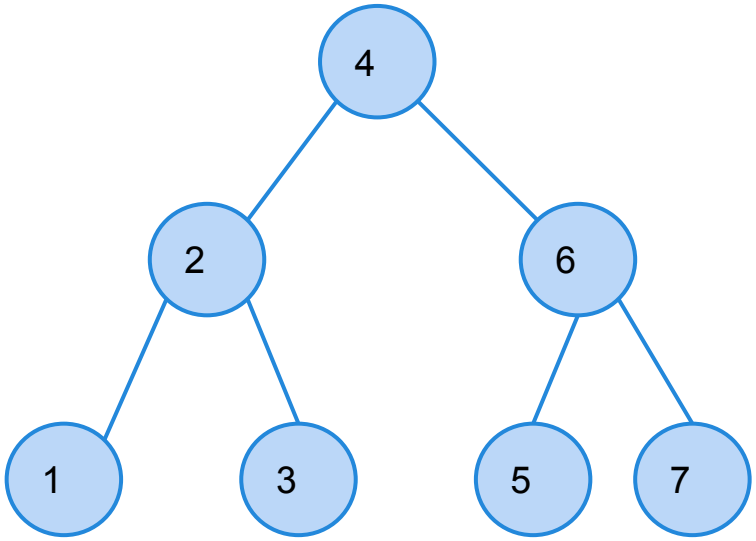
Code:

```
public void postorder() {
    if (firstChild != null) {
      firstChild.postorder();
    }
    this.visit();
    if (nextSibling != null) {
      nextSibling.postorder();
    }
  }
```



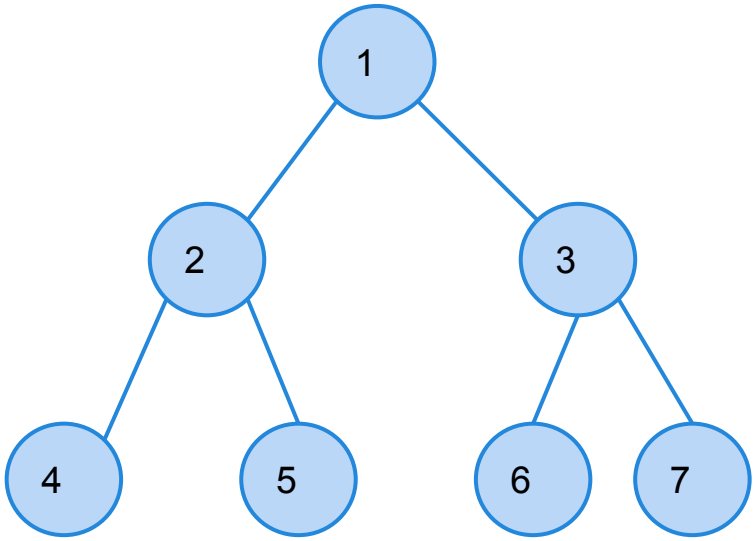
Tree Traversals: in-order (Allowed in Binary Search Trees)

- Recursively traverse the root's left subtree, then the root itself, then the root's right subtree



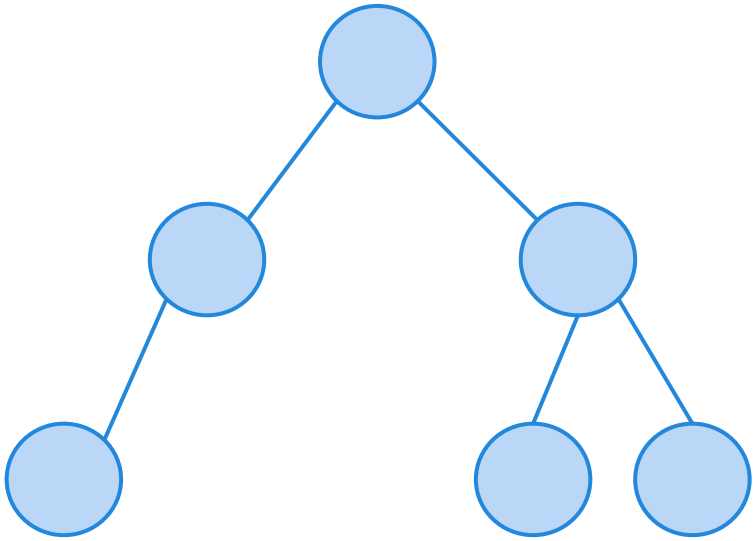
Tree Traversals: level-order

- Visit the root, then all the depth-1 nodes (from left to right), then all the depth-2 nodes, et cetera



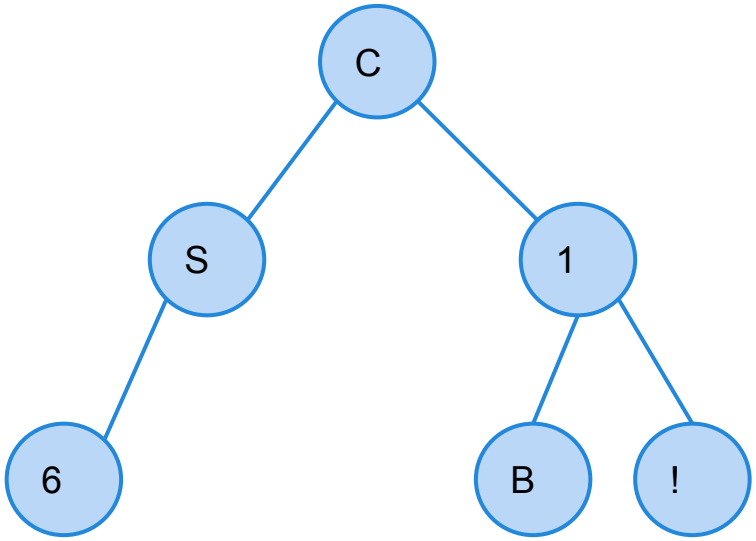
Practice question!

Fill in the following tree such that its preorder traversal is: C S 6 1 B !



Practice question (solution)

Fill in the following tree such that its preorder traversal is: C S 6 1 B !



Binary Search Trees

Binary Search Trees: Overview

A binary search tree is a tree in which no node has more than two children.

For any node X:

- Every key in the left subtree is less than or equal to X's key
- Every key in the right subtree of X is greater than or equal to X's key

Binary Search Trees: Find

- Starting at the root, if the node matches the key you are searching for, return that node.
- Otherwise, traverse the node's right child if key > node, the node's left child if key < node.
- If you reach a null reference, return not found

Binary Search Trees: Insert

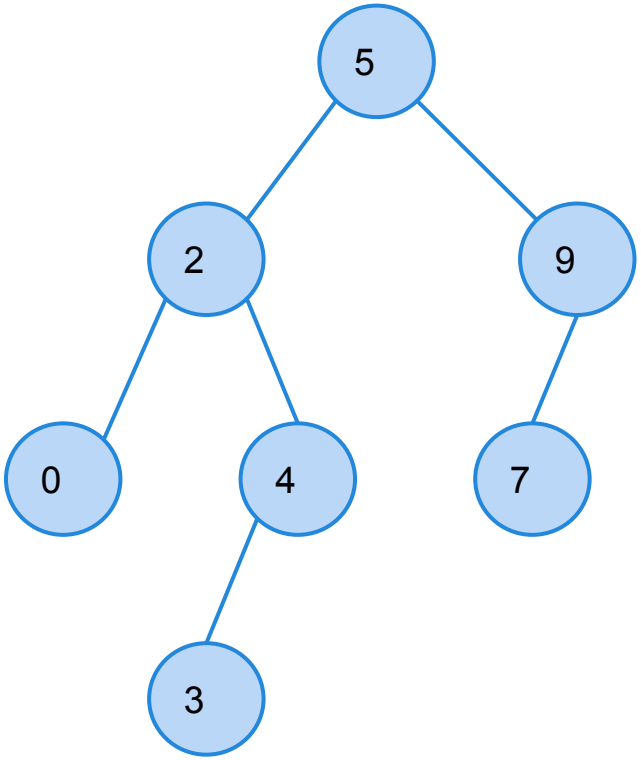
- Following find procedure until you reach a null reference, and insert your number.
- If there are duplicate keys (i.e. if you insert a key that is already in the tree), move to the left child.

Binary Search Trees: Remove

- If the node has no children, just remove it.
- If the node has one child, then join its child to the node's parent.
- If the node has two children, find the next largest value and replace it with node.
- To find the next largest value, go to the node's right child, then left child, left child, etc. until there is no left child. This is the next largest node, note it only has one child.

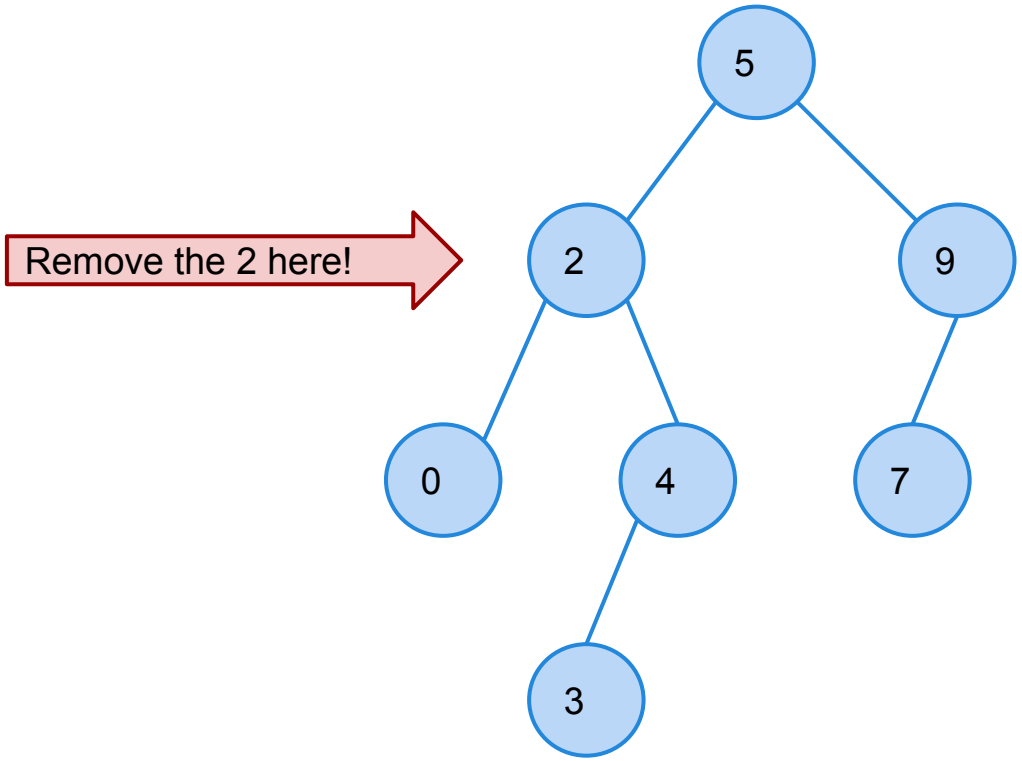
Binary Search Trees: Example

remove(2)



Binary Search Trees: Example

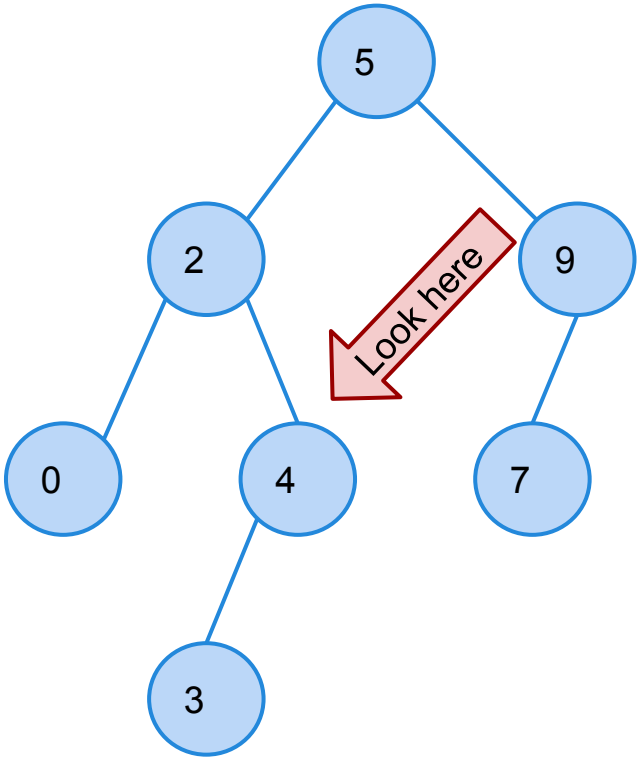
remove(2)



Binary Search Trees: Example

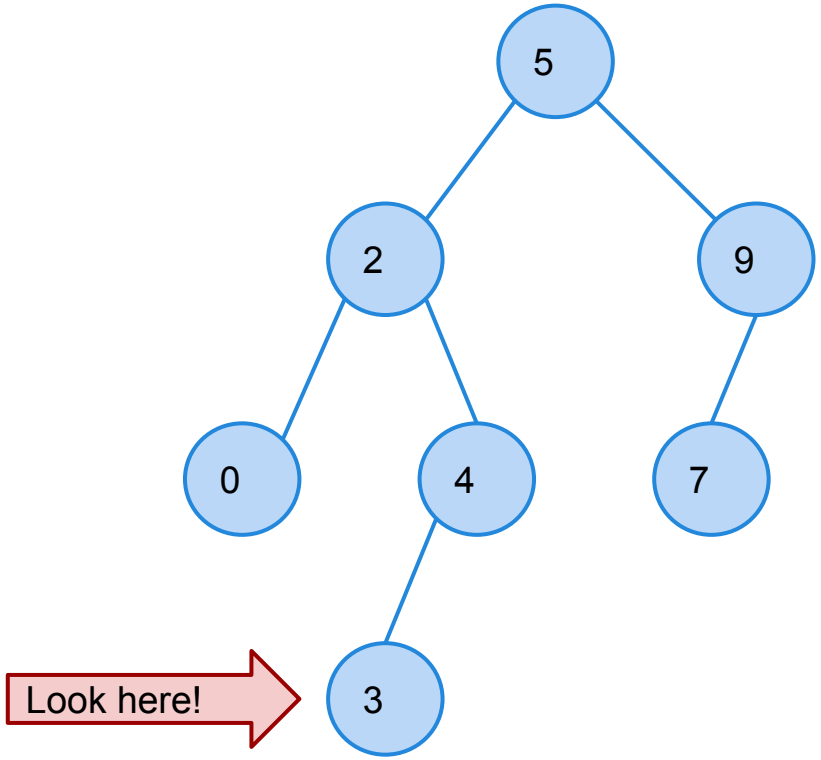
remove(2):

Looking for a replacement



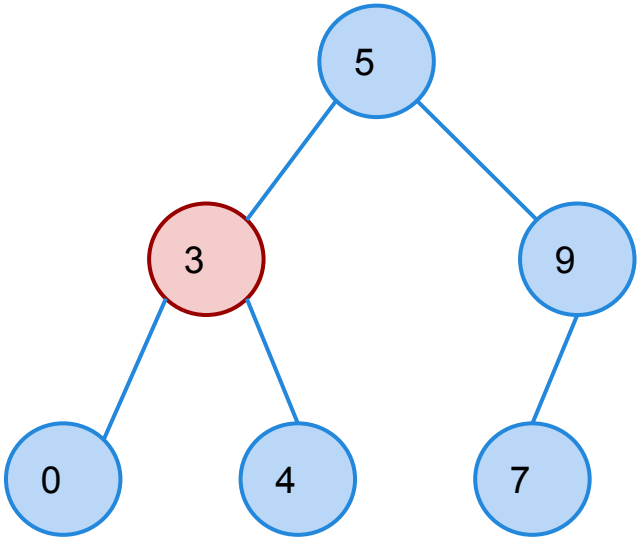
Binary Search Trees: Example

remove(2): We've found a replacement, the 3 node



Binary Search Trees: Example

remove(2): Replace the 2 node with 3

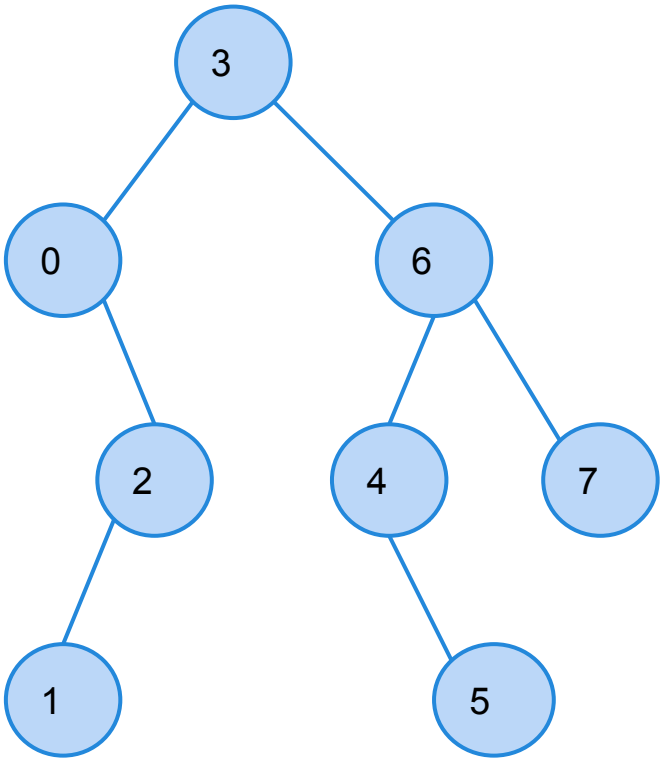


Past midterm question:

Draw the binary search tree whose preorder traversal is: 3 0 2 1 6 4 5 7

Past midterm question: Solution

Draw the binary search tree whose preorder traversal is: 3 0 2 1 6 4 5 7

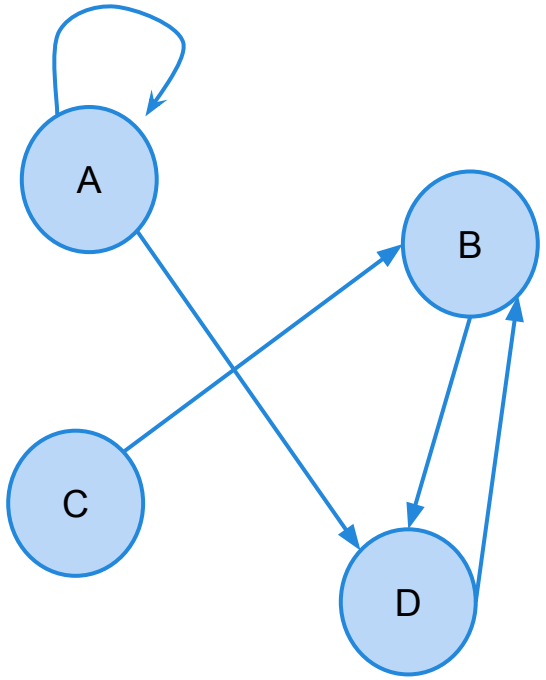


Graph/Graph Traversals

Graphs

What is a graph?

A graph is a collection of nodes (also called vertices)

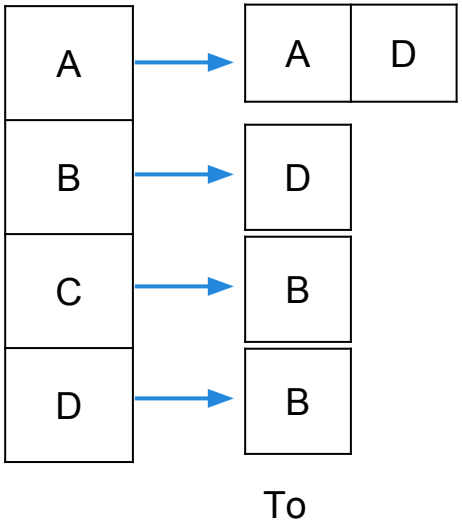


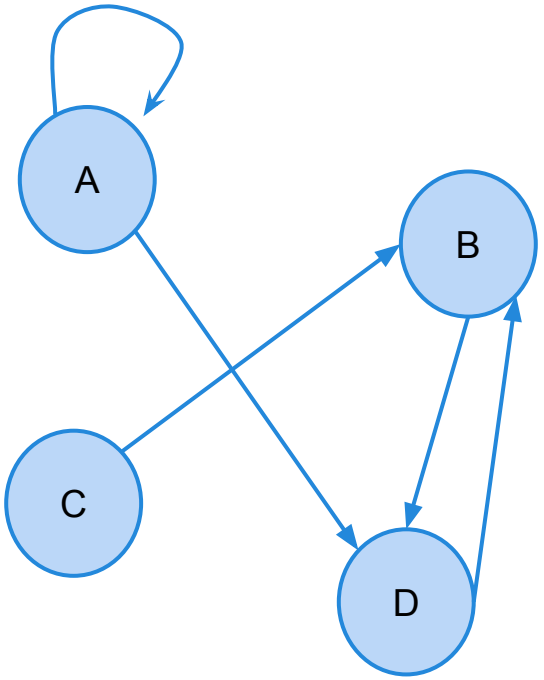
as well as edges that connect vertices

V = # vertices, E = # edges

Graphs Representations

Adjacency List





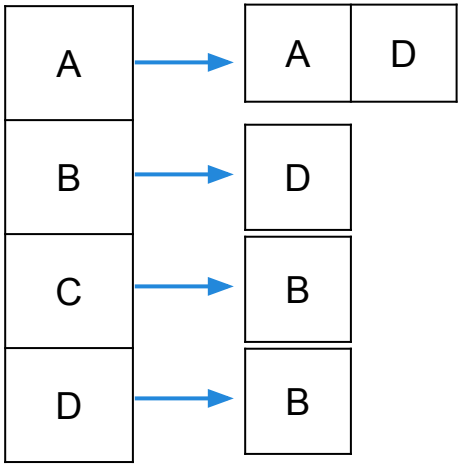
Adjacency Matrix

From

	A	B	C	D
A	1			1
B				1
C		1		
D		1		

Graphs Representations

Adjacency List



Space Complexities

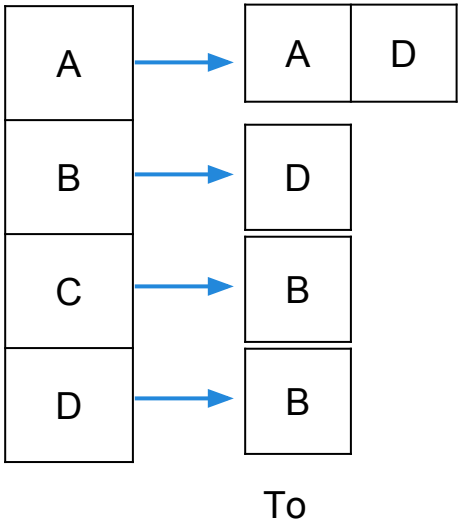
Adjacency Matrix

From

	A	B	C	D
A	1			1
B				1
C		1		
D		1		

Graphs Representations

Adjacency List



Space Complexities

O(V+E)

Adjacency Matrix

From

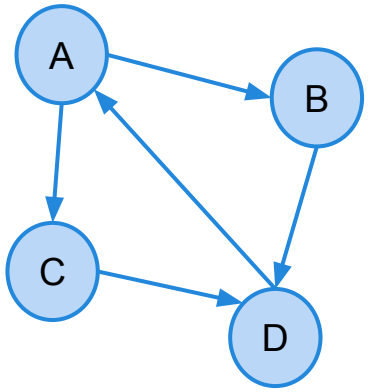
	A	B	C	D
A	1			1
B				1
C		1		
D		1		

$O(V^2)$

Graph Traversals

Depth First Search

```
public void dfs(Vertex u) {
    u.visit();
    u.visited = true;
    for (each vertex v
            reachable from u) {
      if (!v.visited) {
        dfs(v);
      }
    }
  }
}
```



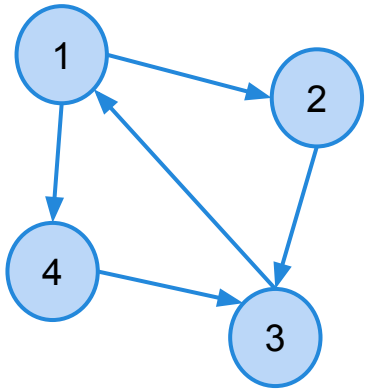
Breadth First Search

```
public void bfs(Vertex u) {
    make a queue q
    q.add(u);
    while(q is not empty) {
        cur = q.dequeue();
        cur.visited = true;

        for(each vertex v
            reachable from cur) {
            if(!v.visited)
                q.add(v);
        }
    }
}
```

Graph Traversals

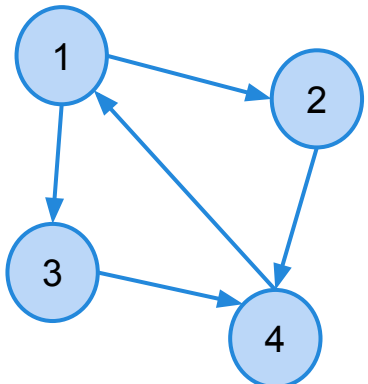
Depth First Search

```
public void dfs(Vertex u) {
    u.visit();
    u.visited = true;
    for (each vertex v
            reachable from u) {
      if (!v.visited) {
        dfs(v);
      }
    }
  }
}
```



Breadth First Search

```
public void bfs(Vertex u) {
    make a queue q
    q.add(u);
    while(q is not empty) {
        cur = q.dequeue();
        cur.visited = true;

        for(each vertex v
            reachable from cur) {
            if(!v.visited)
                q.add(v);
        }
    }
}
```



Graph Traversals

Depth First Search

```
public void dfs(Vertex u) {
    u.visit();
    u.visited = true;
    for (each vertex v
            reachable from u) {
      if (!v.visited) {
        dfs(v);
      }
    }
  }
}
```

Breadth First Search

```
public void bfs(Vertex u) {
    make a queue q
    q.add(u);
    while(q is not empty) {
        cur = q.dequeue();
        cur.visited = true;

        for(each vertex v
            reachable from cur) {
            if(!v.visited)
                q.add(v);
        }
    }
}
```

Time Complexities

Graph Traversals

Depth First Search

```
public void dfs(Vertex u) {
    u.visit();
    u.visited = true;
    for (each vertex v
            reachable from u) {
      if (!v.visited) {
        dfs(v);
    }
  }
}
```

Time Complexities
Both O(V+E)

Breadth First Search

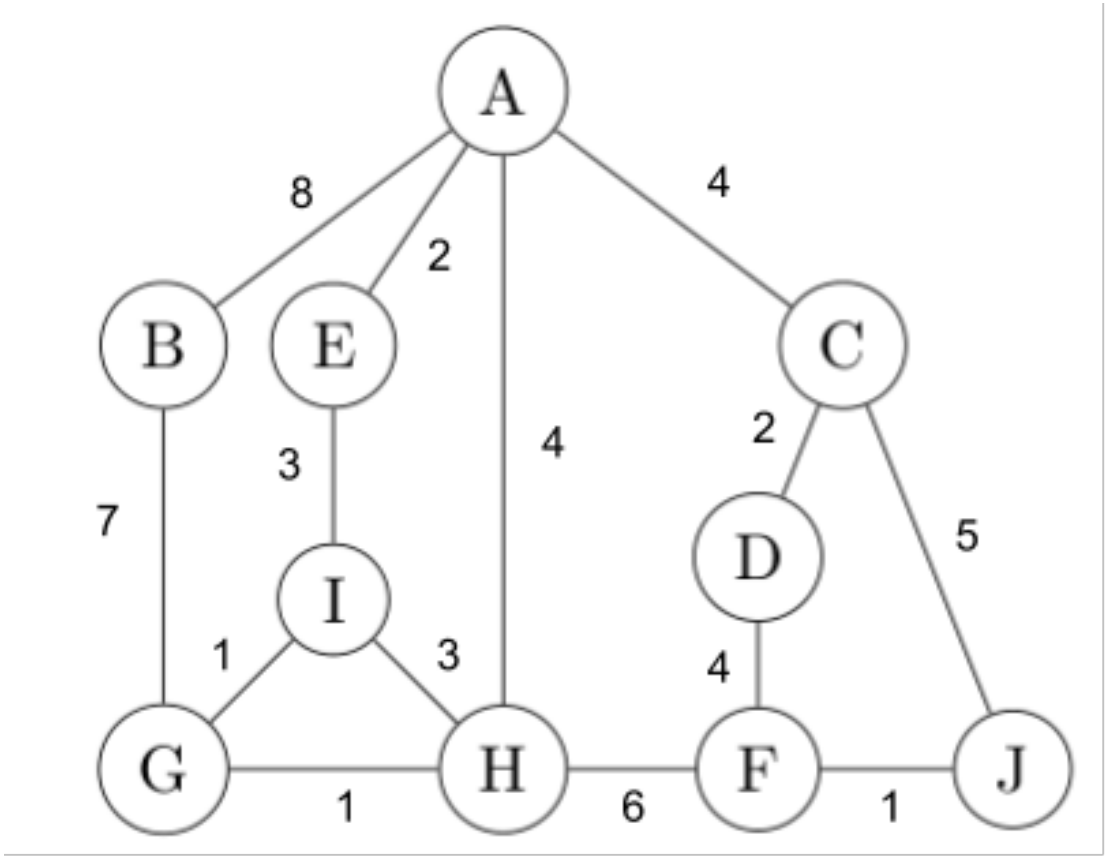
```
public void bfs(Vertex u) {
    make a queue q
    q.add(u);
    while(q is not empty) {
        cur = q.dequeue();
        cur.visited = true;

        for(each vertex v
            reachable from cur) {
            if(!v.visited)
                q.add(v);
        }
    }
}
```

Graph Questions!

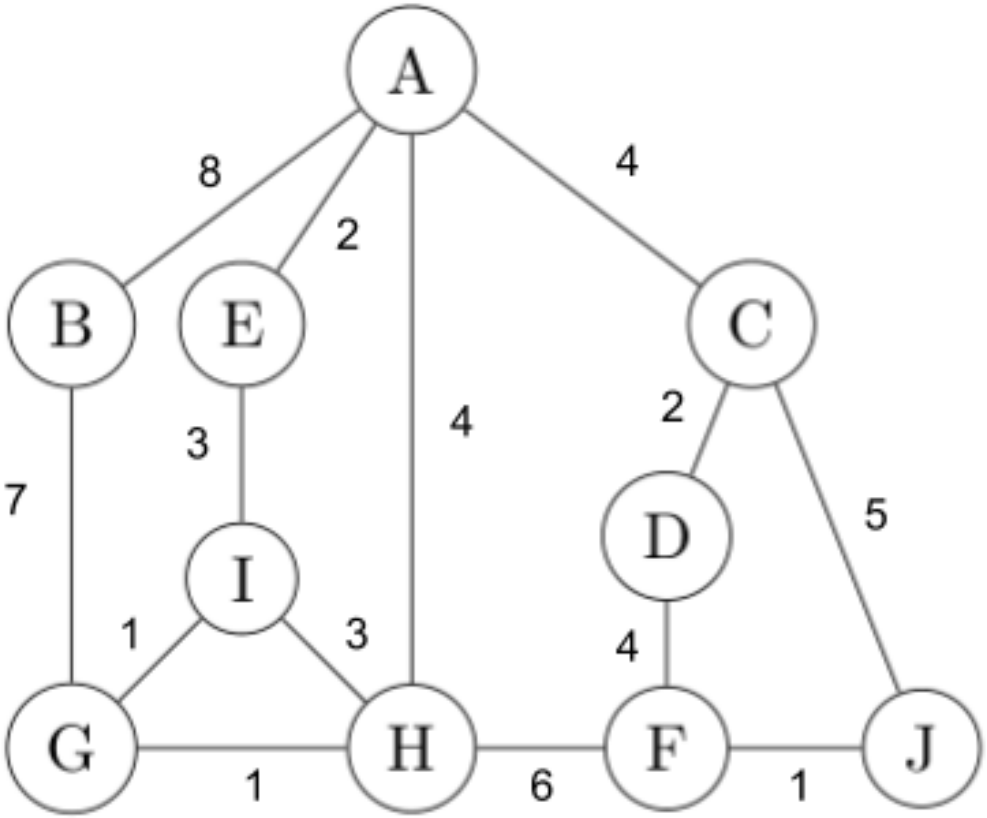
	A	B	C	D	E	F	G	H	I	J
A	-	8	4	-	2	-	-	4	-	-
B	8	-	-	-	-	-	7	-	-	-
C	4	-	-	2	-	-	-	-	-	5
D	-	-	2	-	-	4	-	-	-	-
E	2	-	-	-	-	-	-	-	3	-
F	-	-	-	4	-	-	-	6	-	1
G	-	7	-	-	-	-	-	1	1	-
H	4	-	-	-	-	6	1	-	3	-
I	-	-	-	-	3	-	1	3	-	-
J	-	-	5	-	-	1	-	-	-	-

Convert the following adjacency matrix into an weighted undirected graph.

Graphs

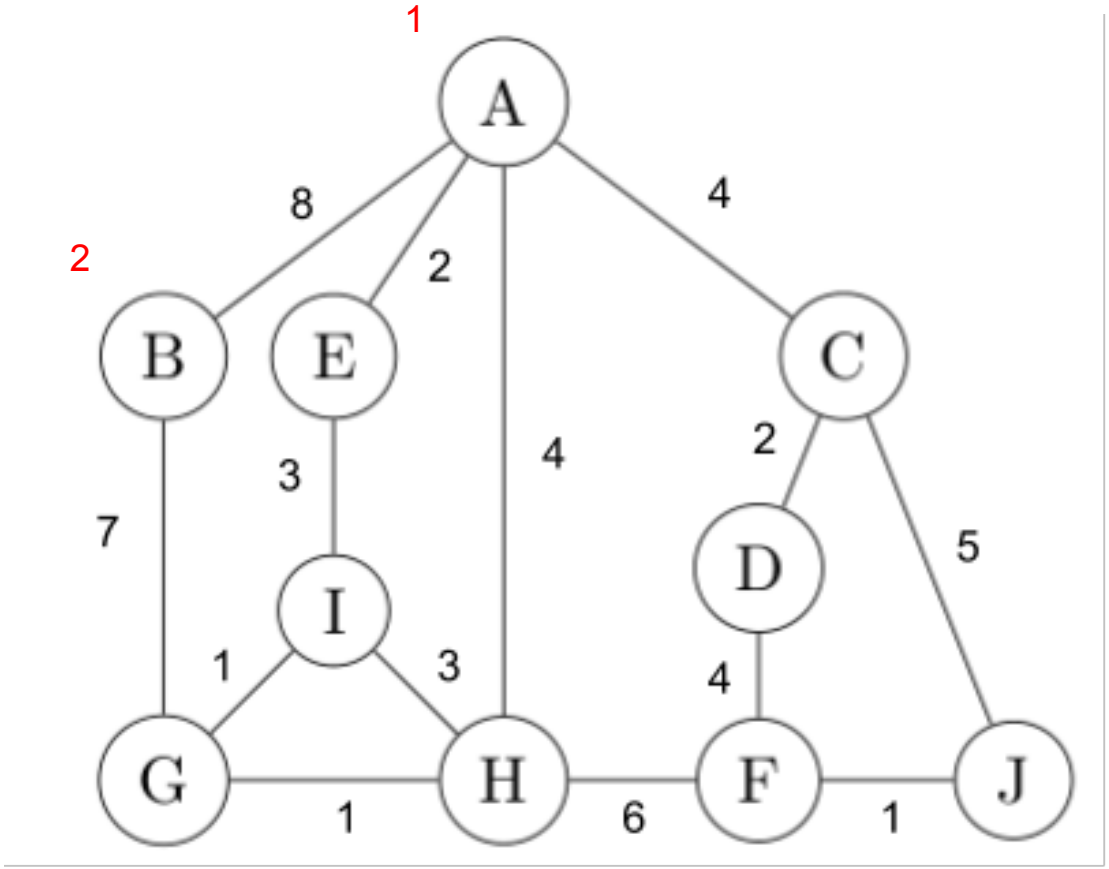


Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

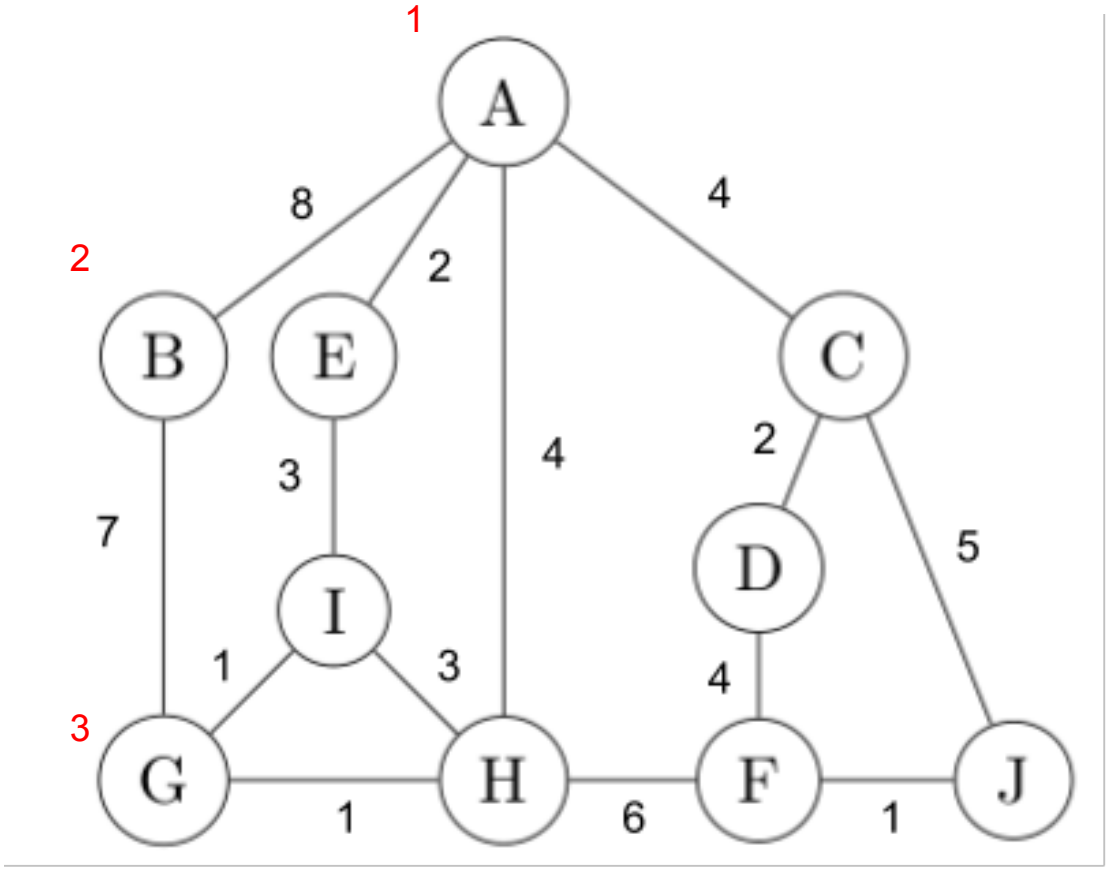
Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B

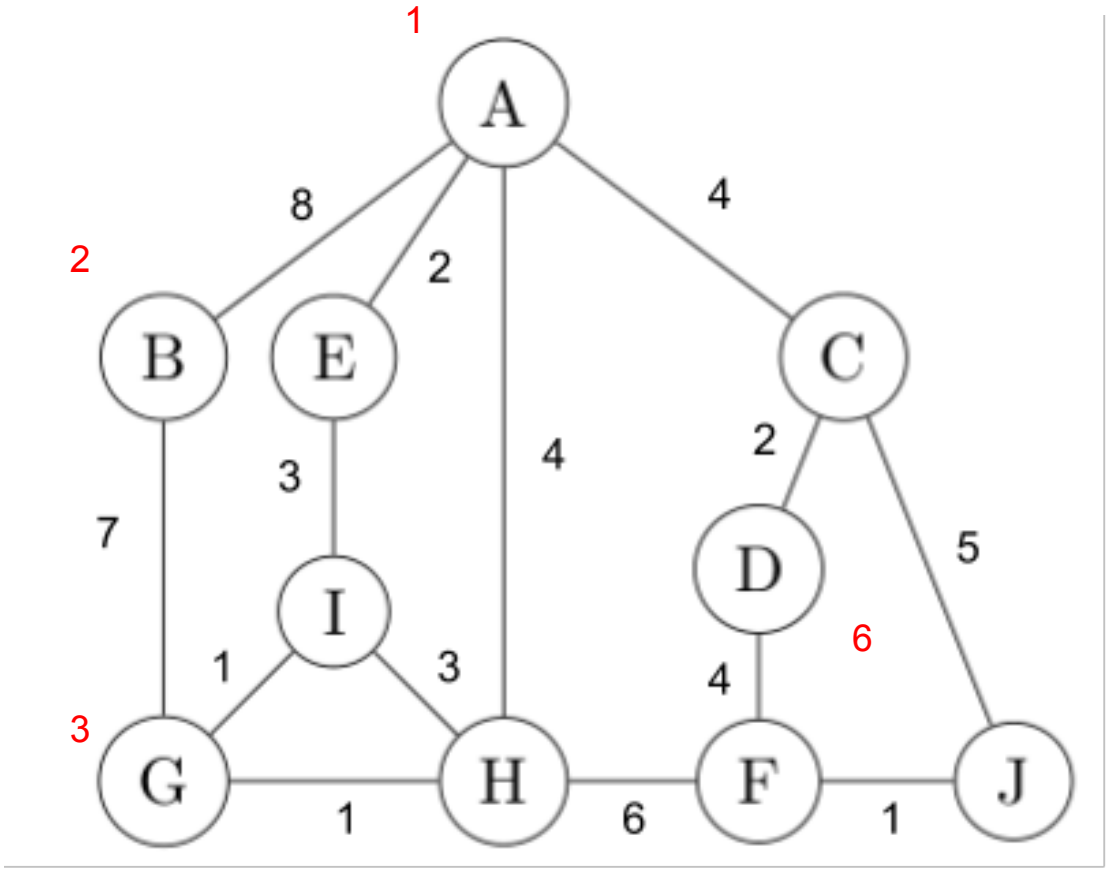
Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B G H

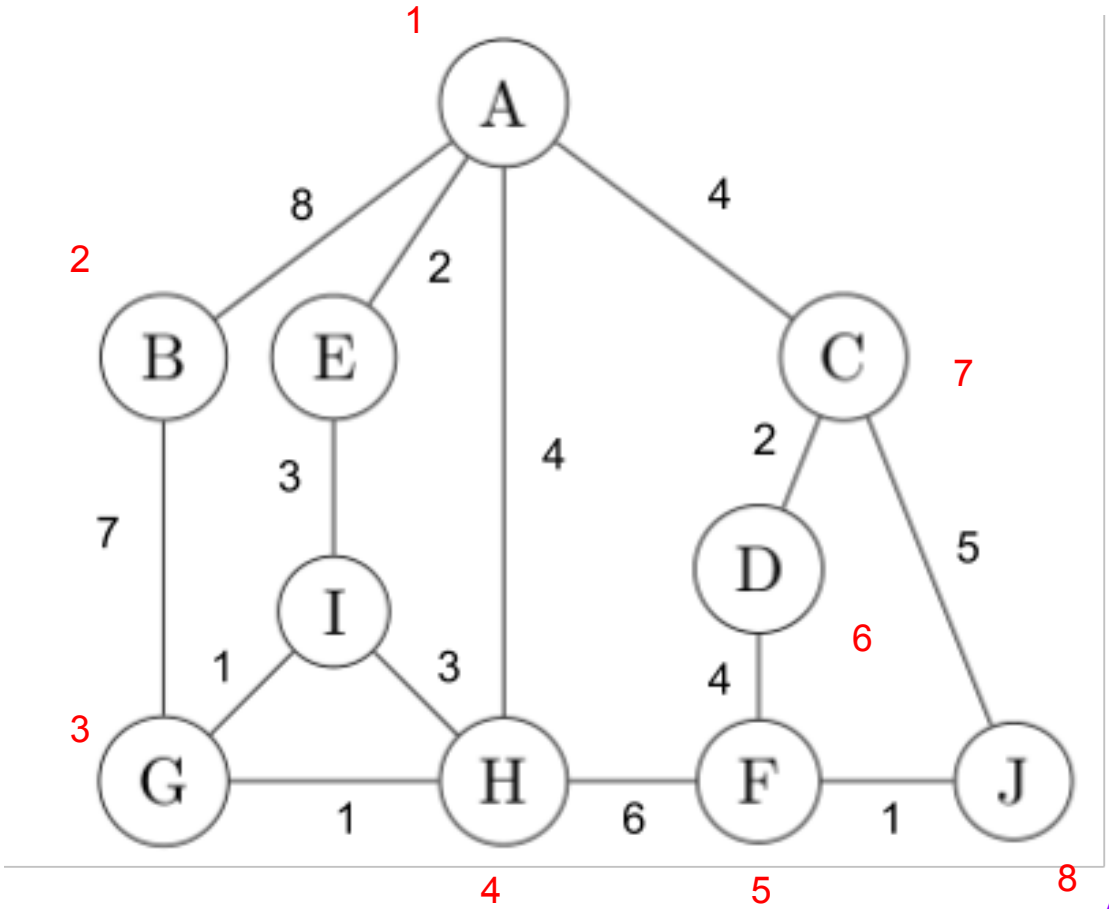
Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B G H F D

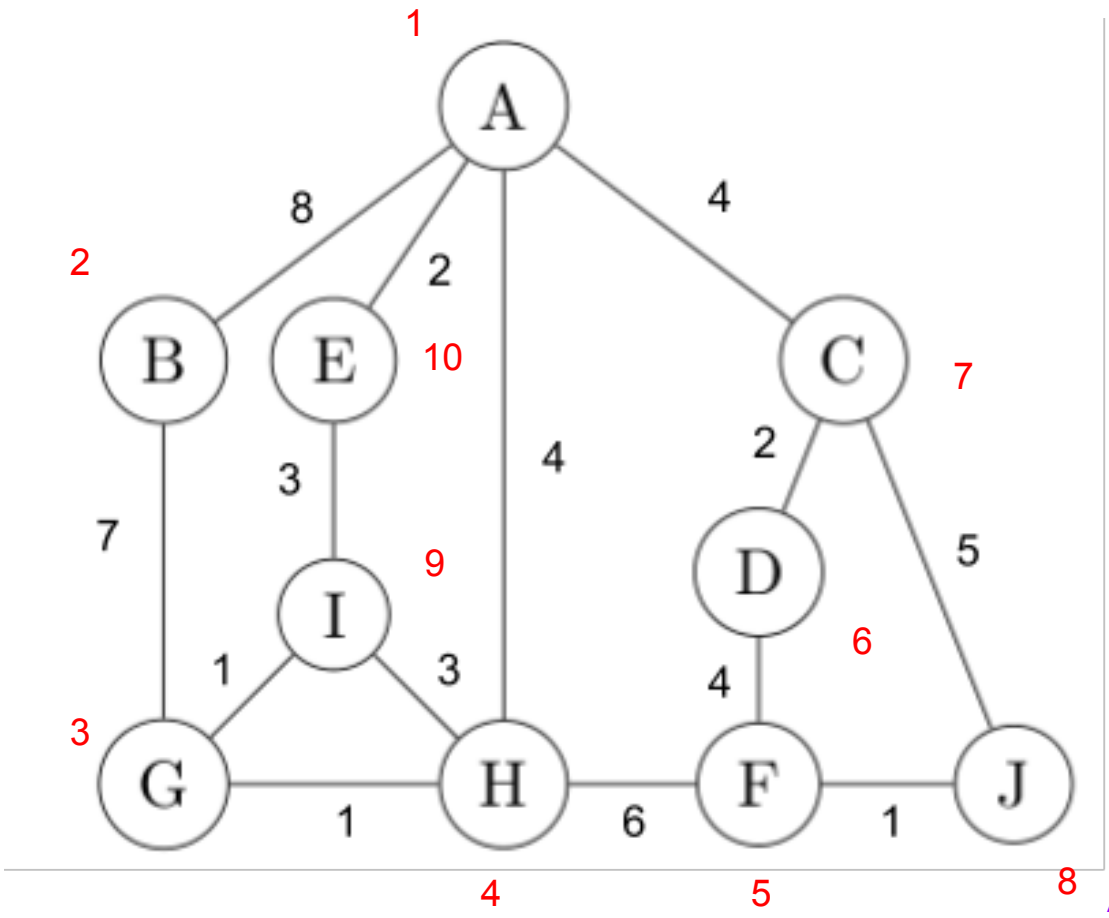
Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B G H F D C J

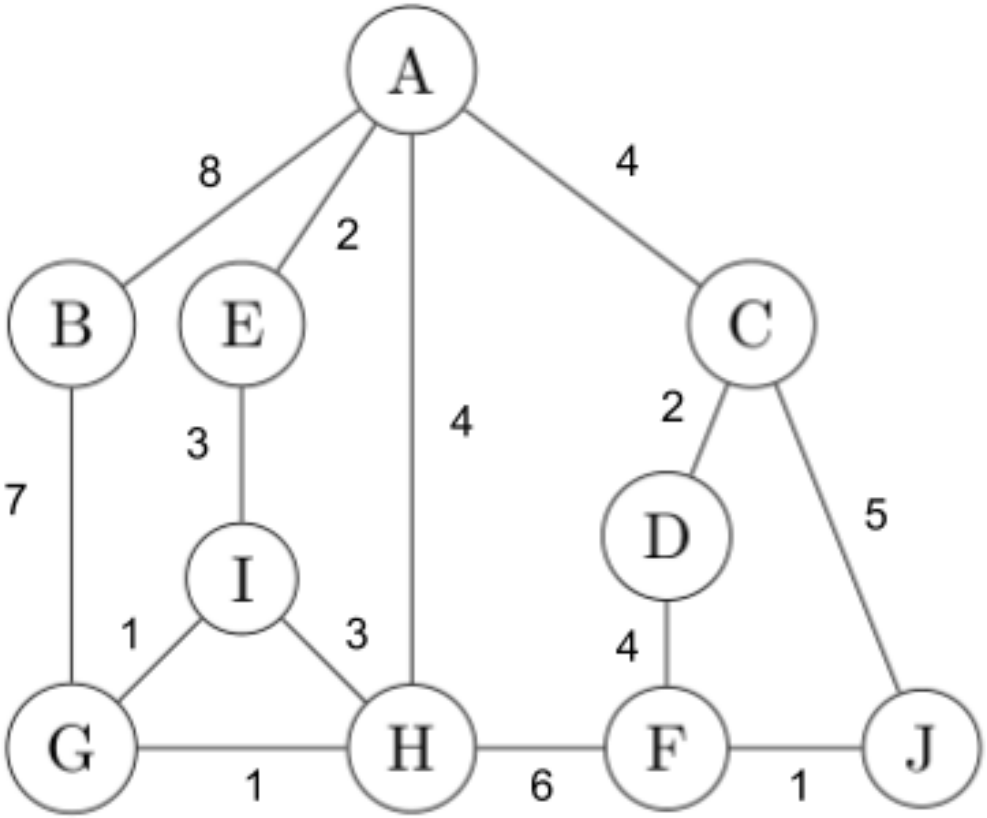
Graphs



a. Write the order in which a DFS starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

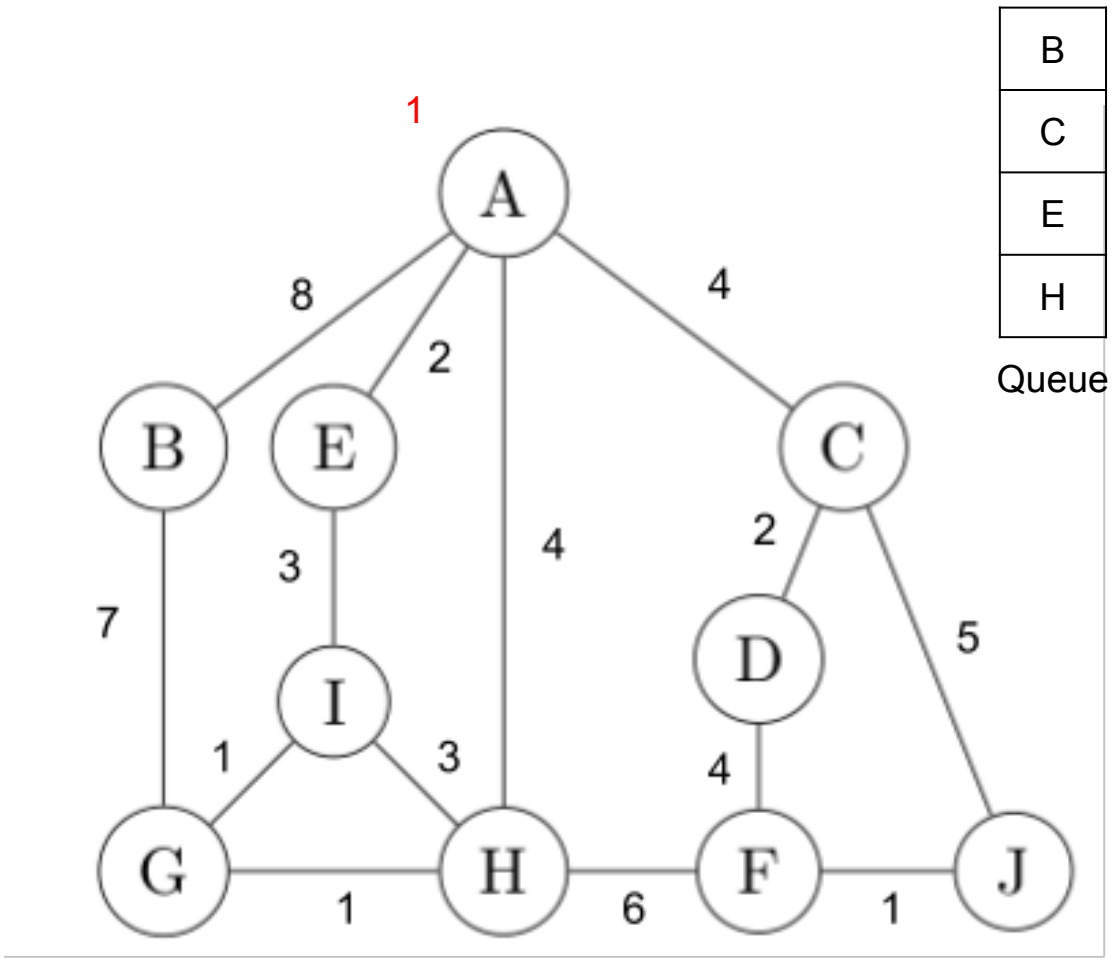
Answer: A B G H F D C J I E

Graphs, contd.



a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Graphs, contd.



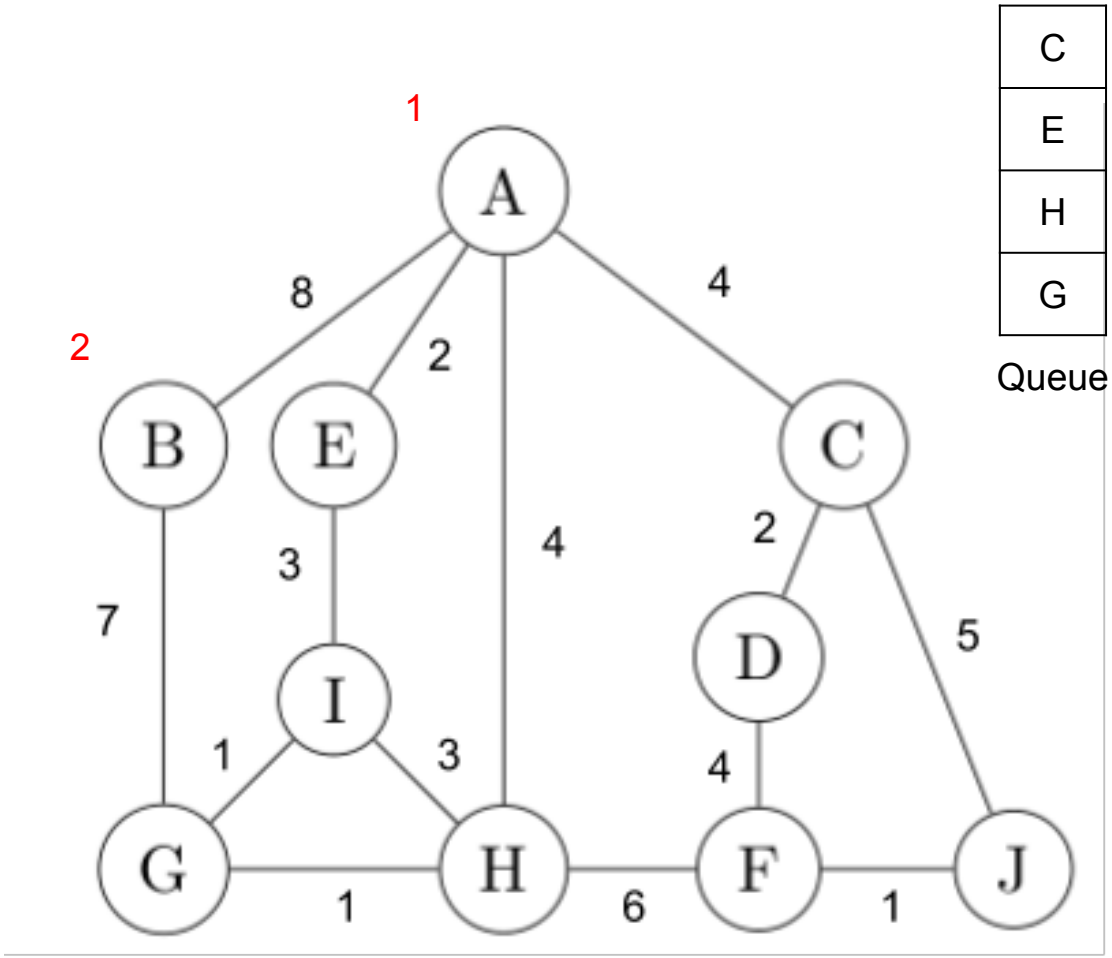
B
C
E
H

Queue

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A

Graphs, contd.



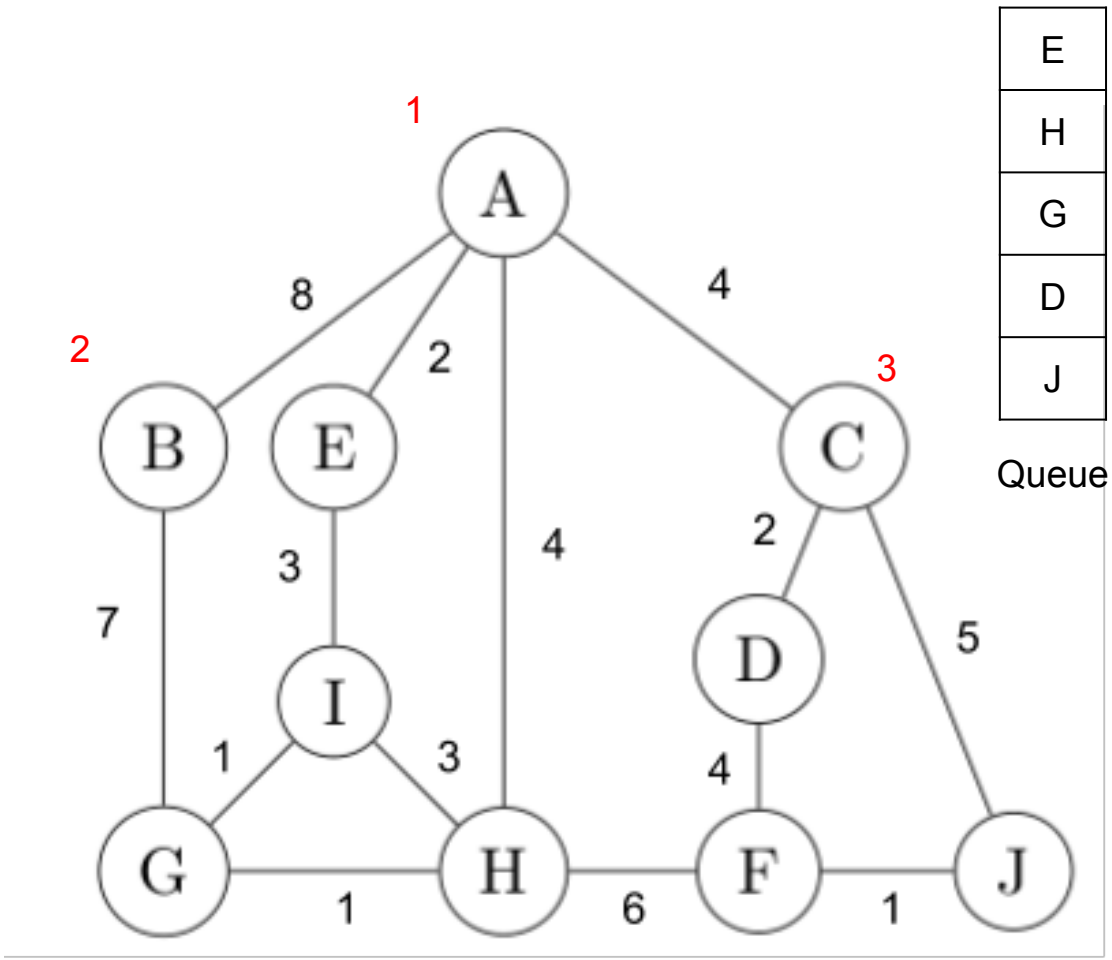
C
E
H
G

Queue

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B

Graphs, contd.



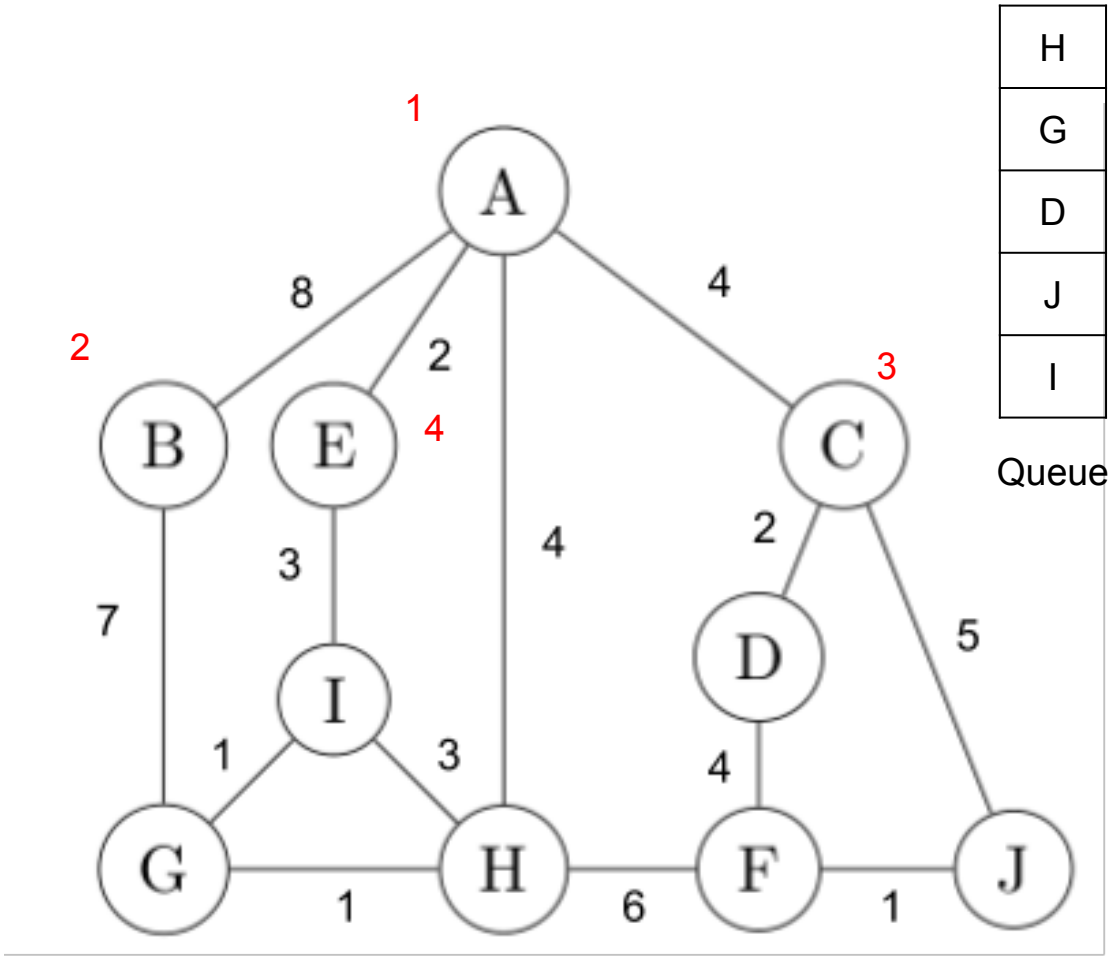
E
H
G
D
J

Queue

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C

Graphs, contd.



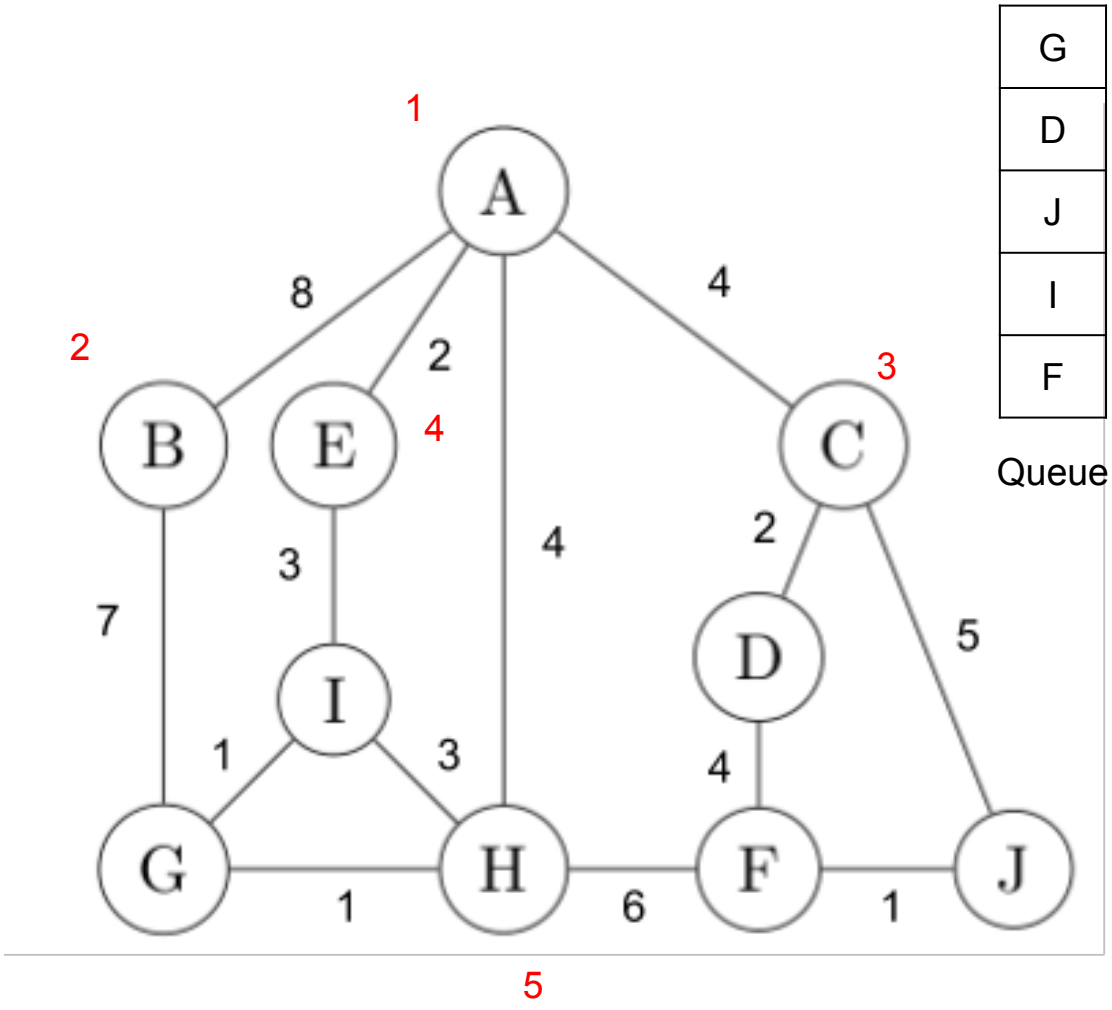
H
G
D
J
I

Queue

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E

Graphs, contd.

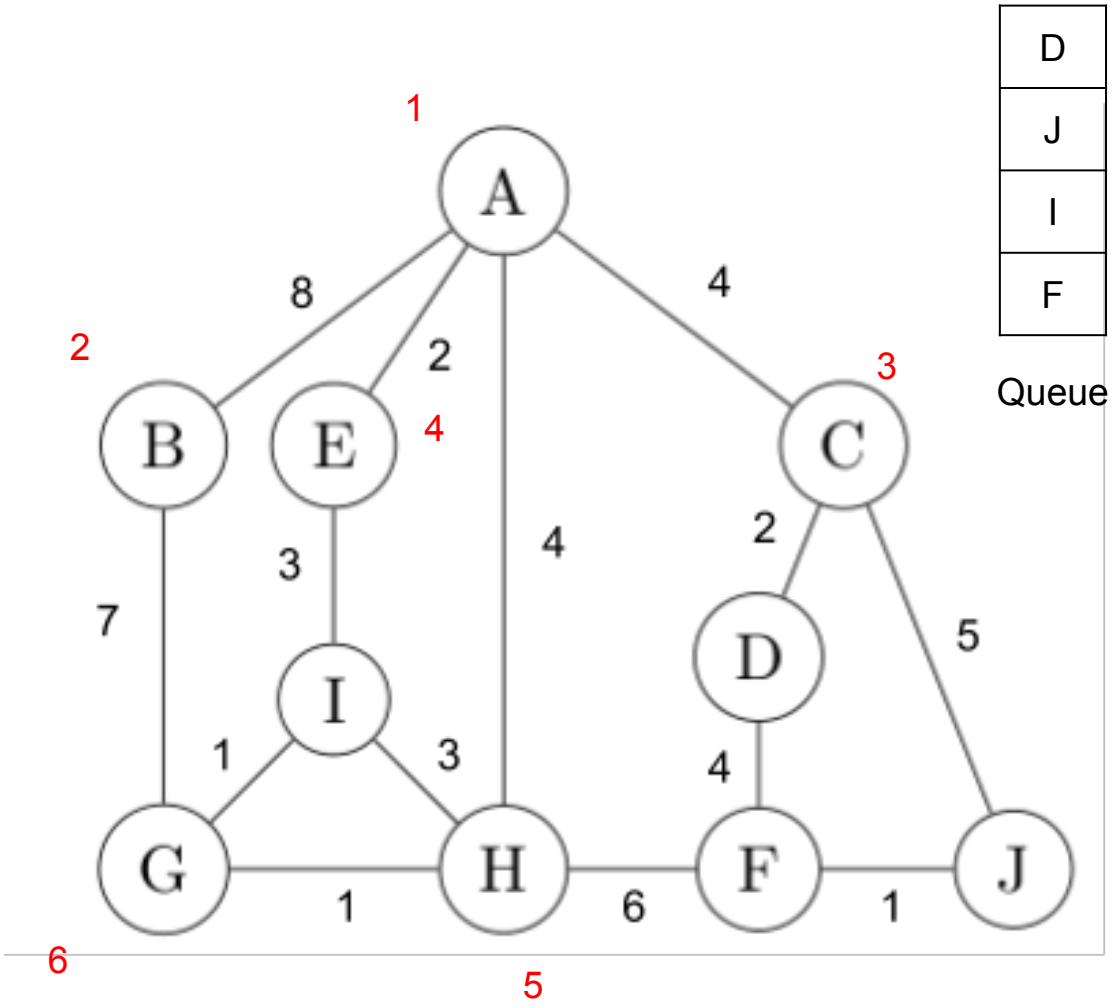


Queue
G
D
J
I
F

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H

Graphs, contd.

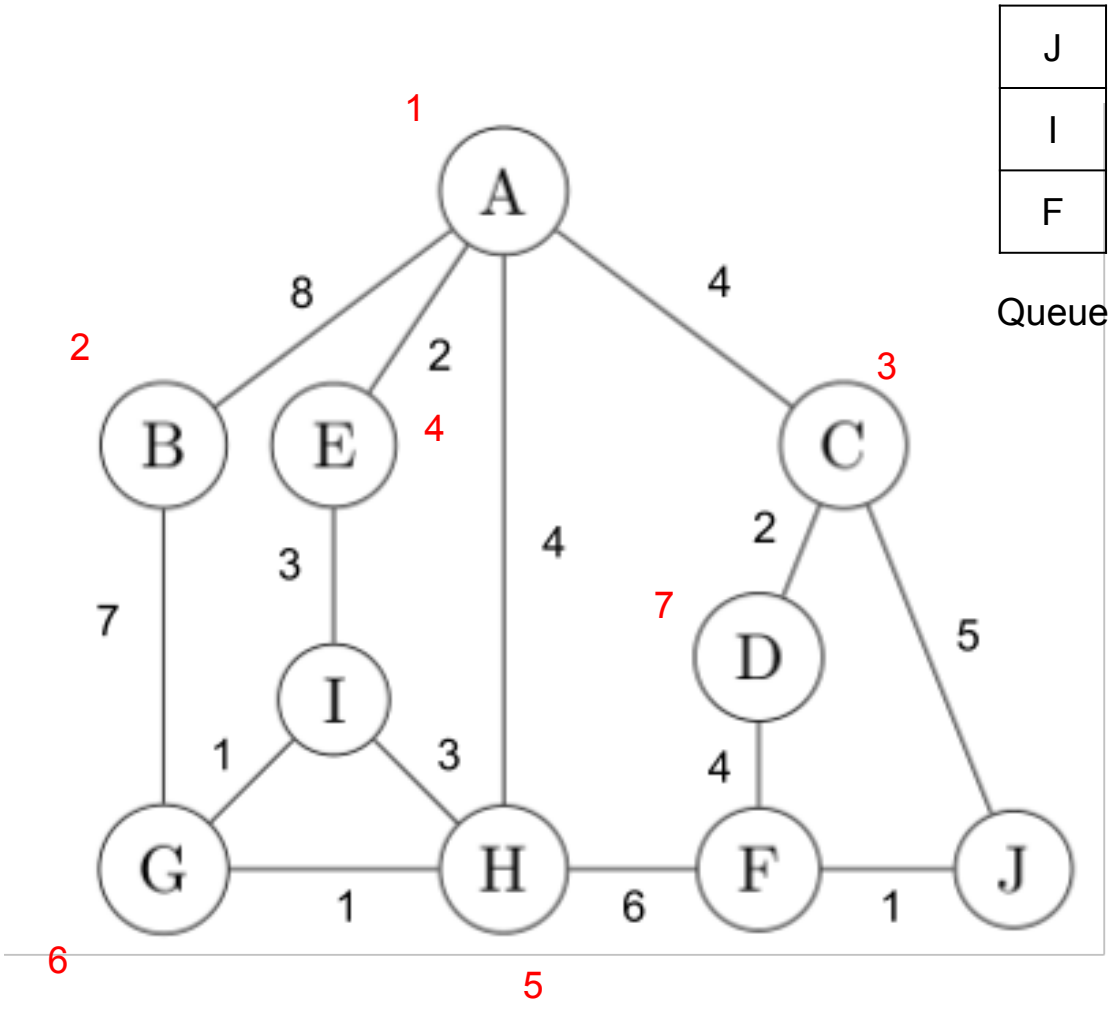


Queue
D
J
I
F

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H G

Graphs, contd.



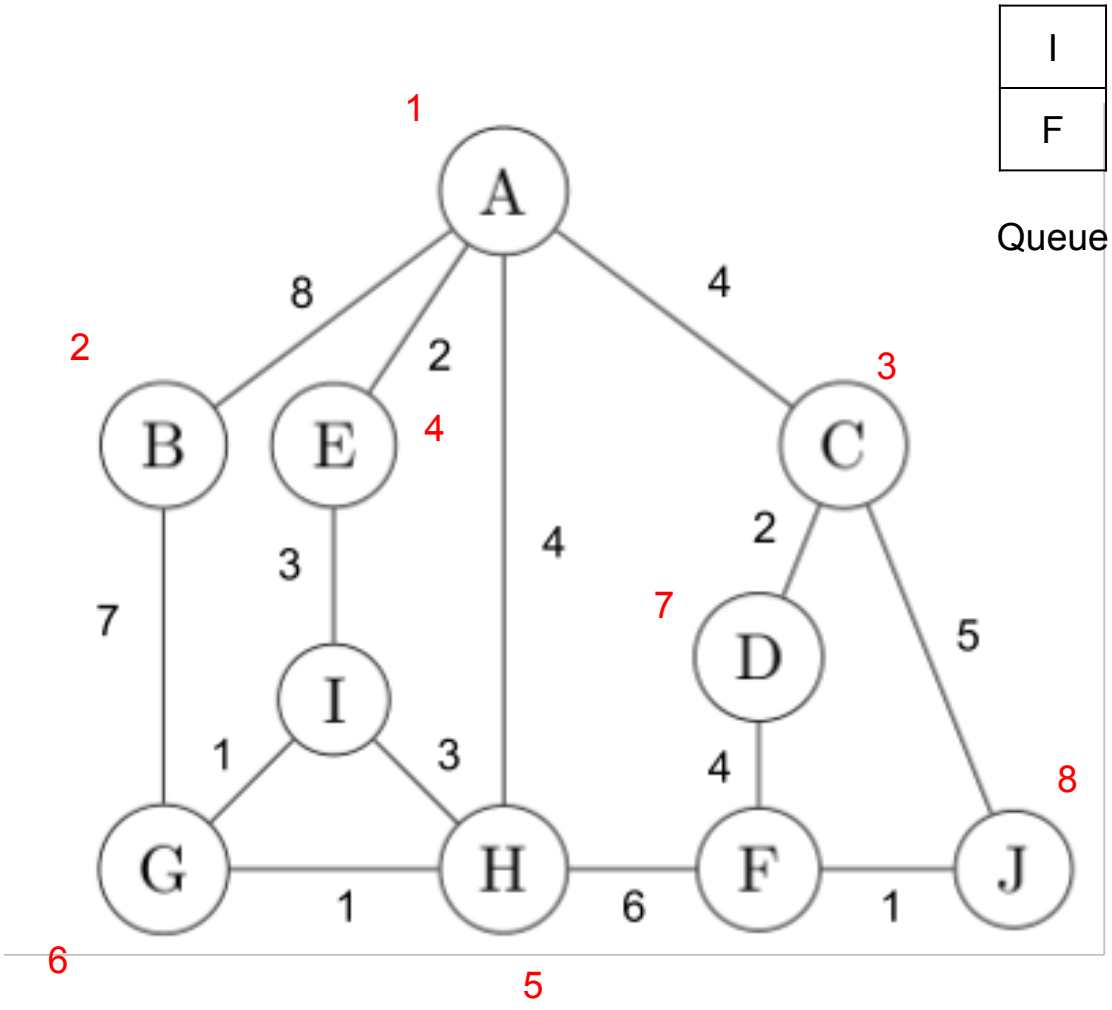
J
I
F

Queue

a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H G D

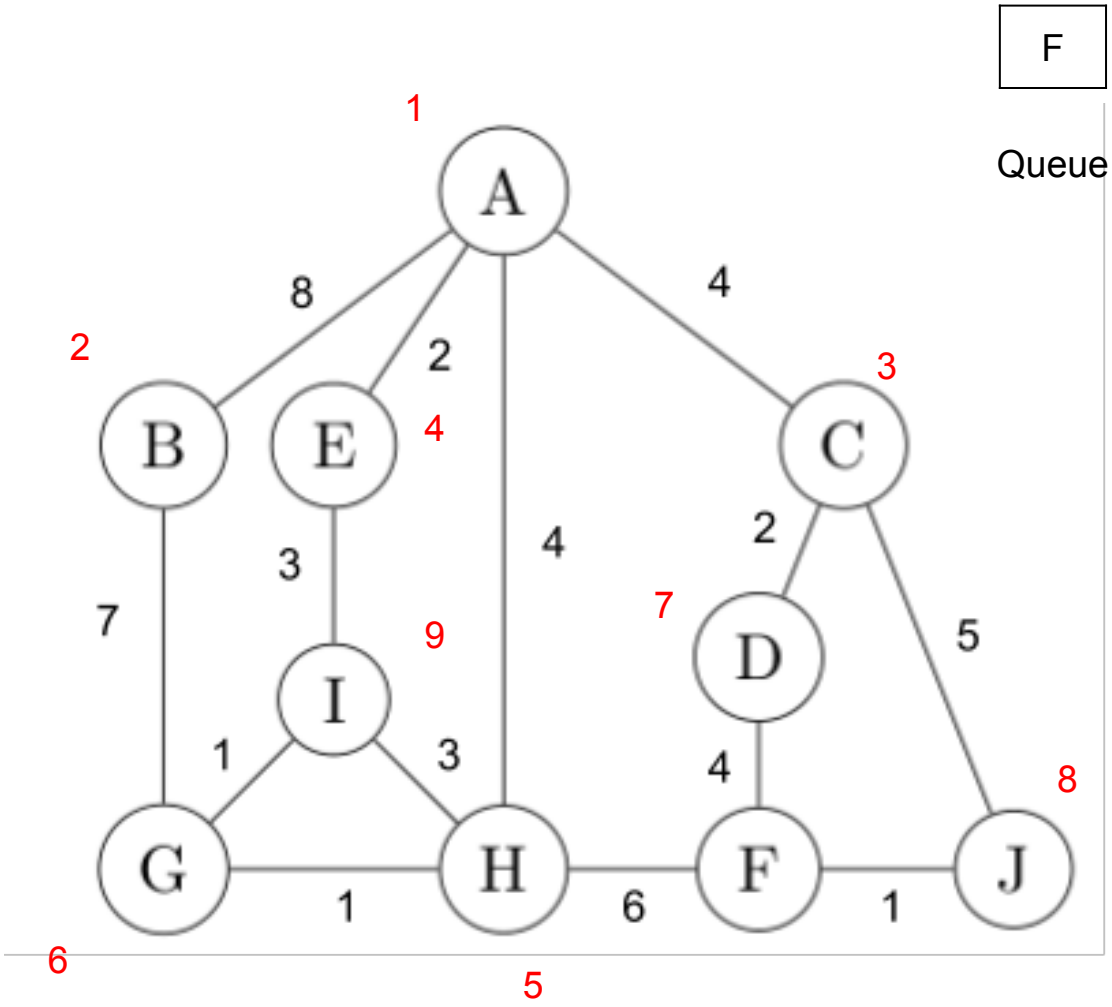
Graphs, contd.



a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H G D J

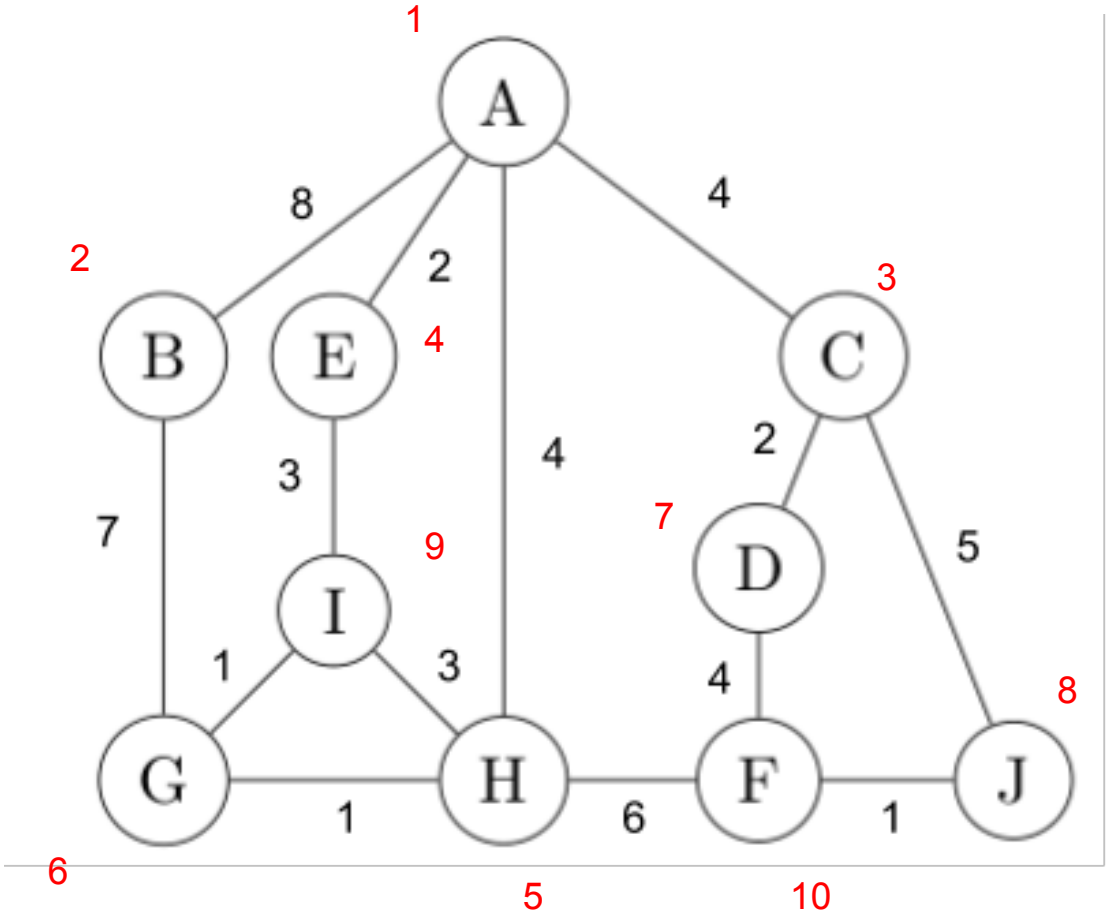
Graphs, contd.



a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H G D J I

Graphs, contd.



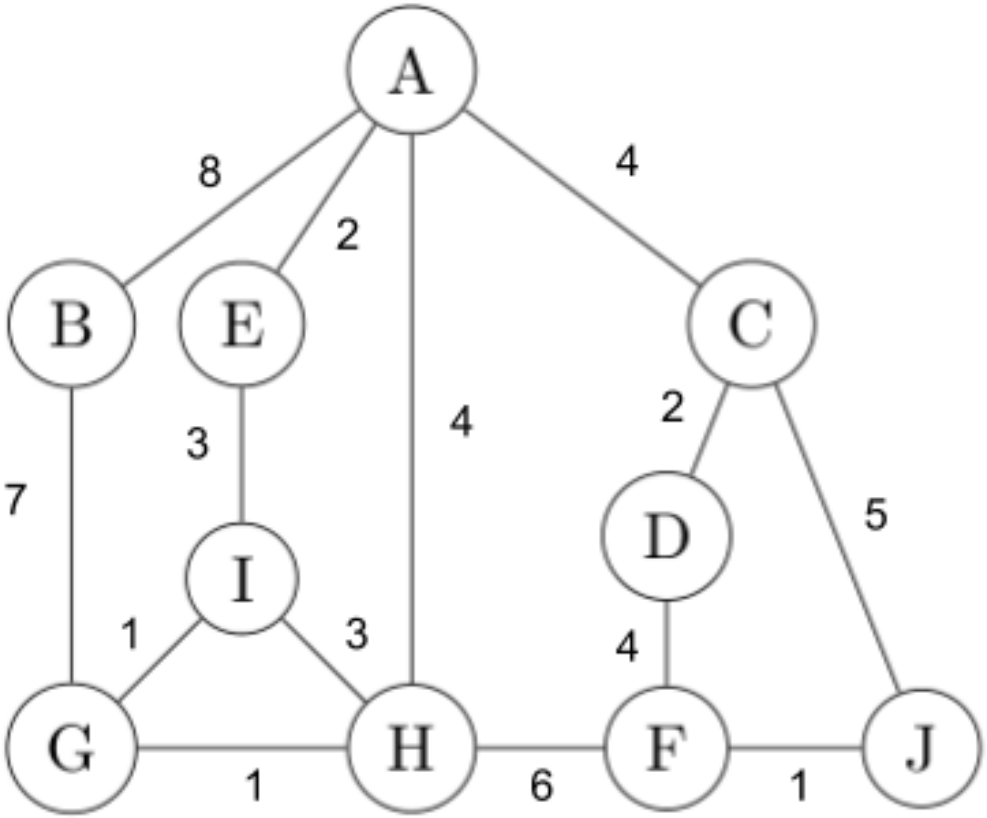
a. Write the order in which a **BFS** starting at vertex A would visit the nodes on the given graph. Whenever there is a choice of which node to visit next, visit the nodes in alphabetical order.

Answer: A B C E H G D J I F

Graphs, Kruskals Algorithm

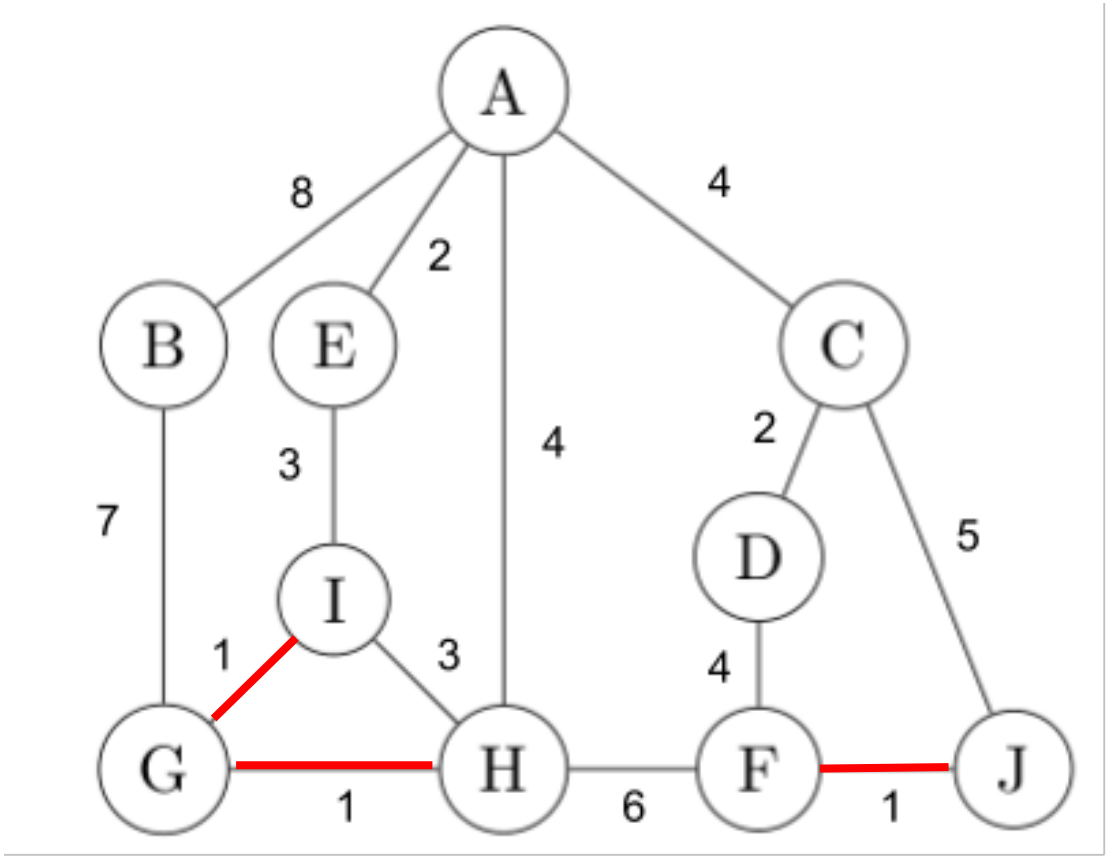
1. Create a new graph T with the same vertices as G, but no edges (yet).
2. Make a list of all the edges in G, and sort the edges by weight, from least to greatest.
3. For each edge (u,w):
   a. If u and w are not connected by a path in T, add (u, w) to T.
4. ???
5. Profit!!

Graphs, contd.



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

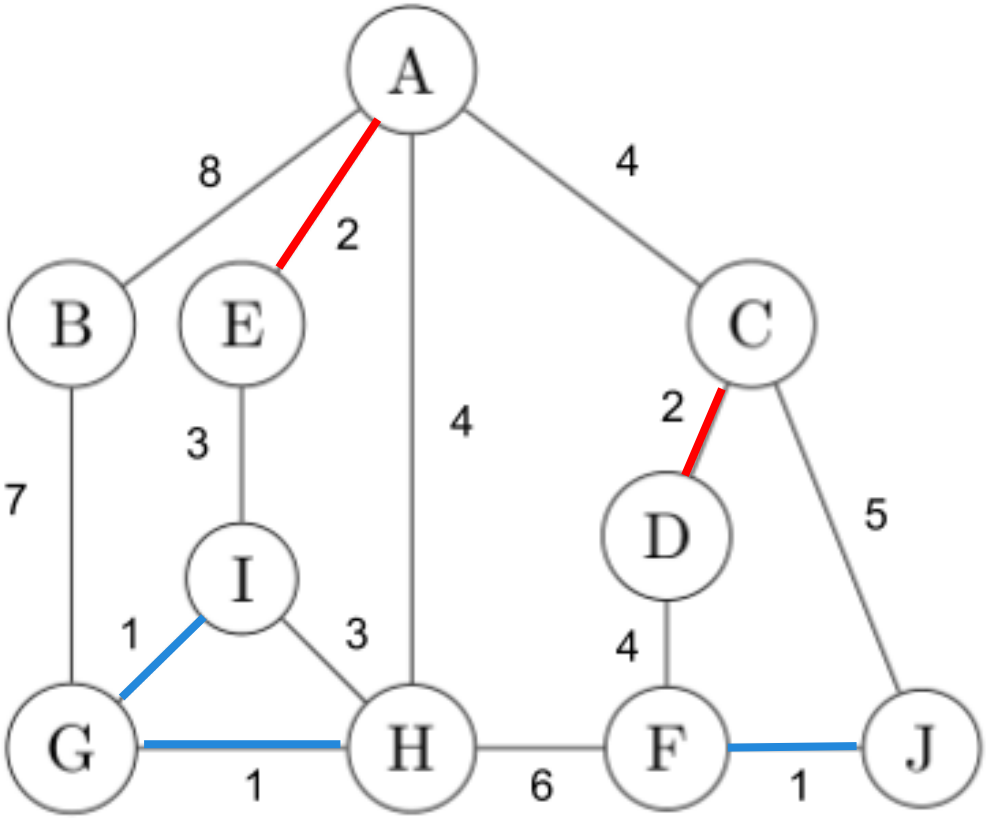
Graphs, Minimal Spanning Trees



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

Edges in the tree: (G,I), (G,H), (F, J). Cost = 3

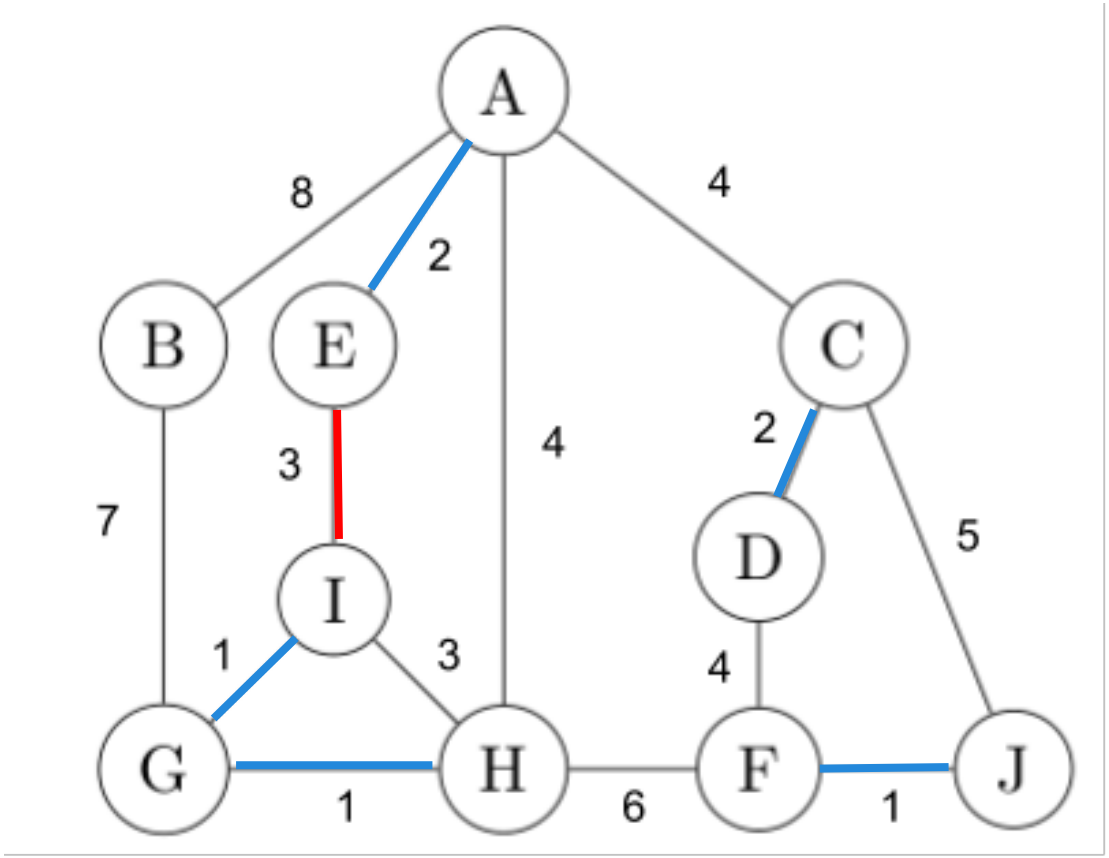
Graphs, Minimal Spanning Trees



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

Edges in the tree: (G,I), (G,H), (F, J), (A, E), (C, D). Cost = 7

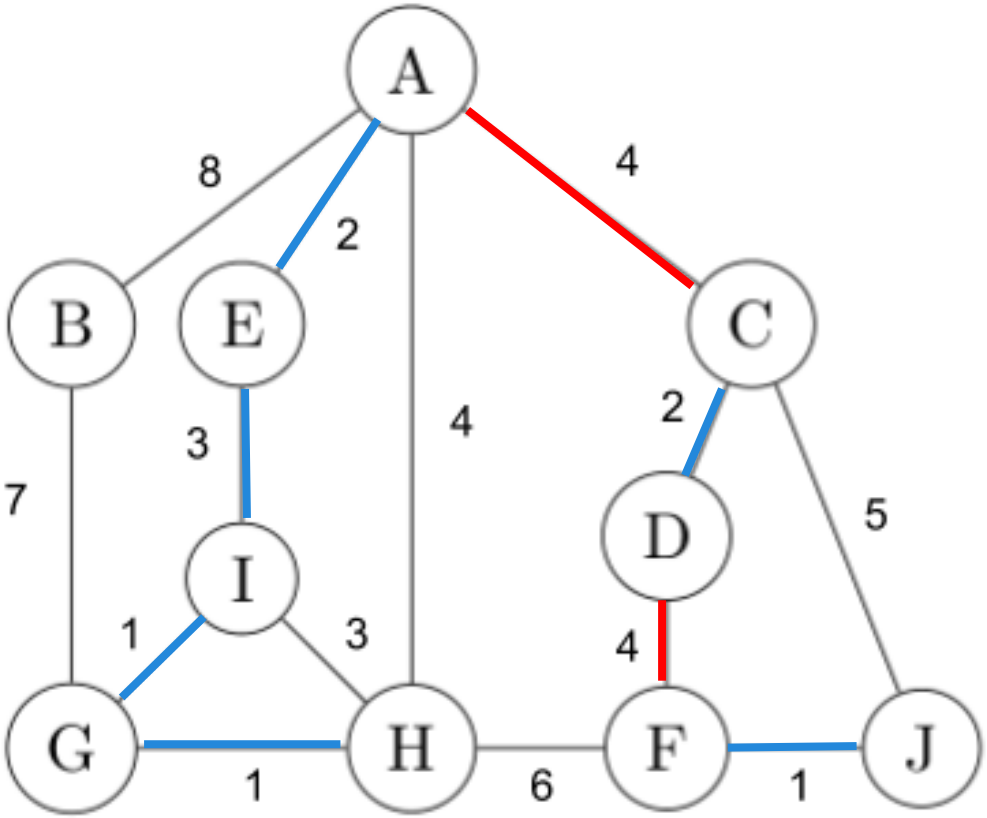
Graphs, Minimal Spanning Trees



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

Edges in the tree: (G,I), (G,H), (F, J), (A, E), (C, D), (E, I). Cost = 10

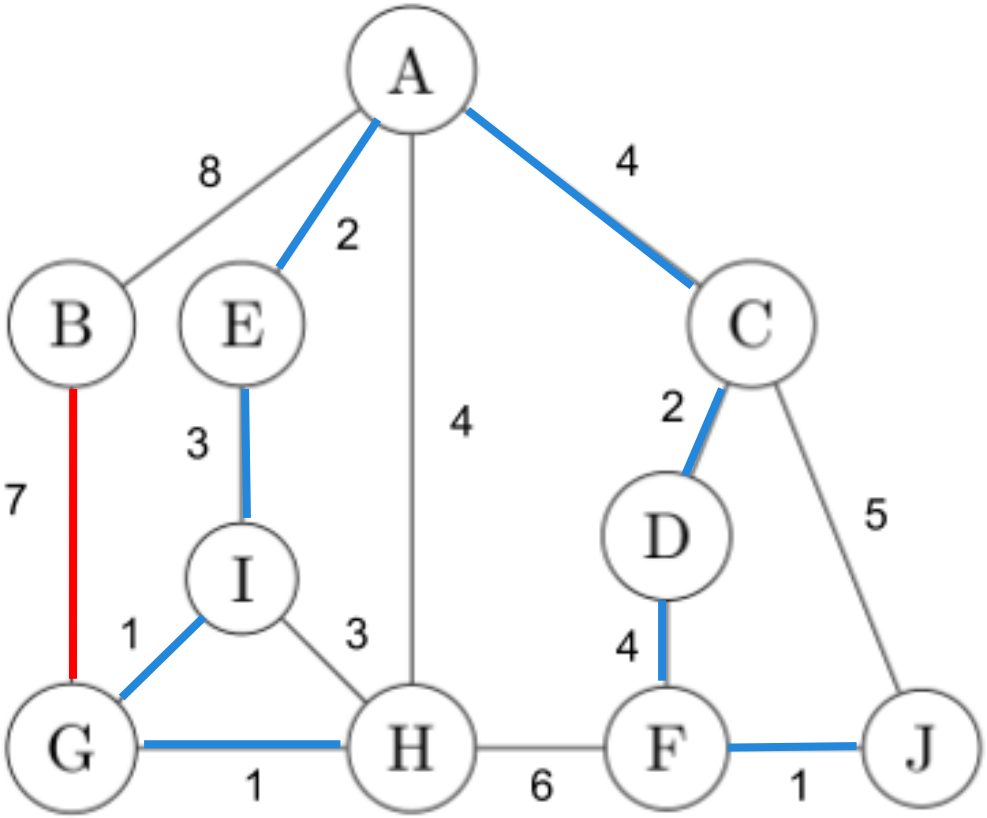
Graphs, Minimal Spanning Trees



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

Edges in the tree: (G,I), (G,H), (F, J), (A, E), (C, D), (E, I), (A,C), (D,F). Cost = 18

Graphs, Minimal Spanning Trees



c. Run Kruskal's Algorithm to find the minimum spanning tree. List the edges that are part of the tree. What is the total cost of the tree?

Edges in the tree: (G,I), (G,H), (F, J), (A, E), (C, D), (E, I), (A,C), (D,F), (B,G). Cost = 25

Time to get a bit tricky

1. How would you find the *maximum* spanning tree of a graph? Describe in one or two sentences.

2. Suppose a graph has V vertices with E > V. Under what condition is the minimal spanning tree unique?

Time to get a bit tricky

1. How would you find the *maximum* spanning tree of a graph? Describe in one or two sentences.

Run Kruskal's but process edges from longest to shortest

2. Suppose a graph has V vertices with E > V. Under what condition is the minimal spanning tree unique?

Time to get a bit tricky

1. How would you find the *maximum* spanning tree of a graph? Describe in one or two sentences.

Run Kruskal's but process edges from longest to shortest

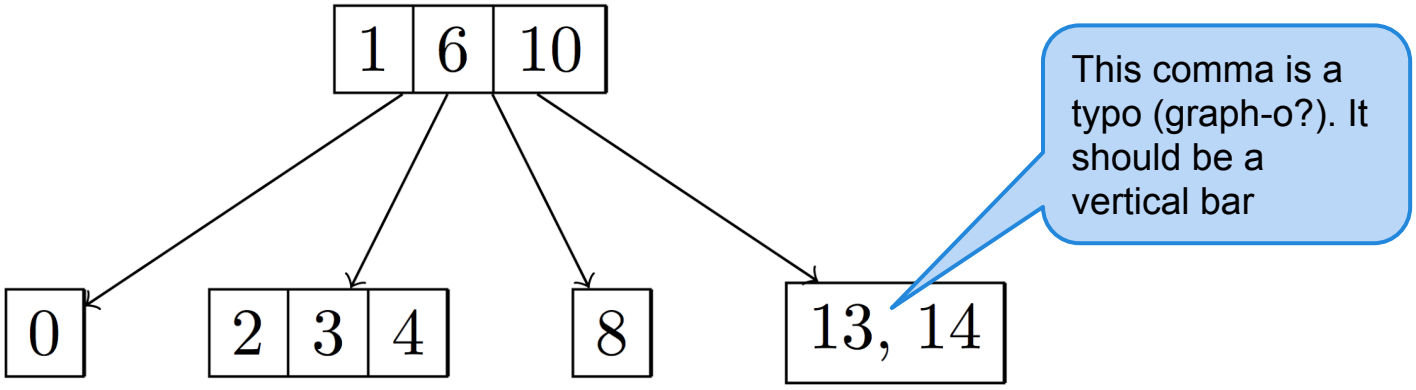
2. Suppose a graph has V vertices with E > V. Under what condition is the minimal spanning tree unique?

If every edge has a different weight

2-3-4 Trees

2-3-4 Trees

- Each node contains 1, 2, or 3 keys
  - Each node has one more child than keys
  - Each child's subtree contains values between the keys on either side of the child
- Perfectly balanced
  - O(log n) runtime find, insert, remove

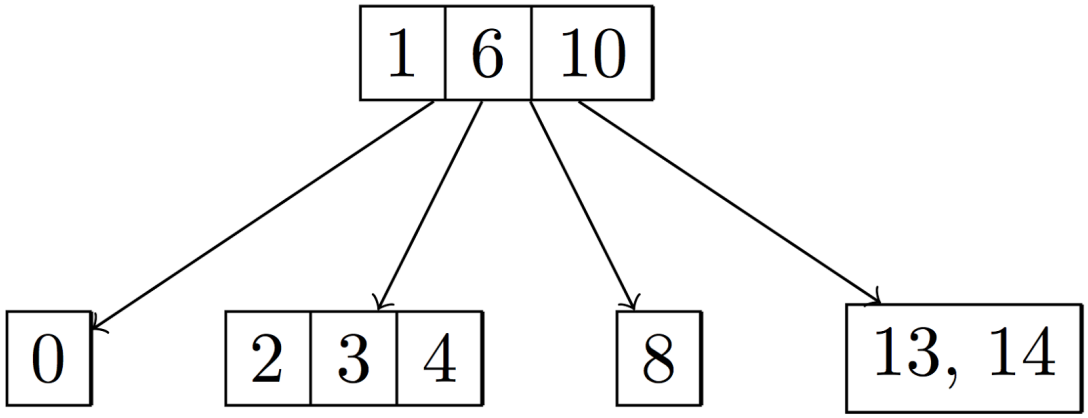


2-3-4 Trees

- Find: Starting from the root, walk down the tree, choosing an appropriate child by comparing the search key with each entry.
- Insert: Push middle key of 3-key nodes up as you go down the tree. Insert key in leaf.
- Remove: Find the next largest value to replace the value you are replacing.
  - As you traverse down the tree, eliminate all one-key nodes.

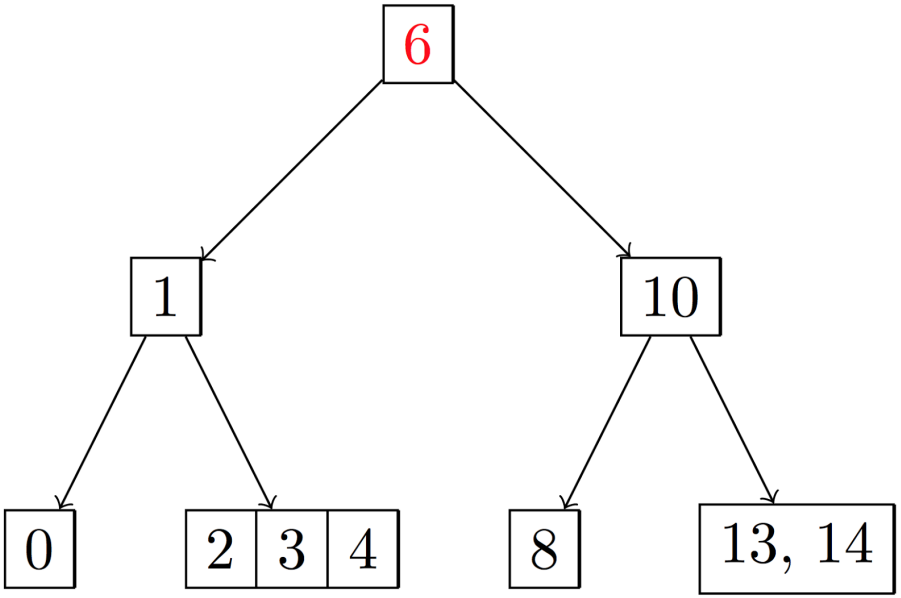
2-3-4 Trees

insert(5)



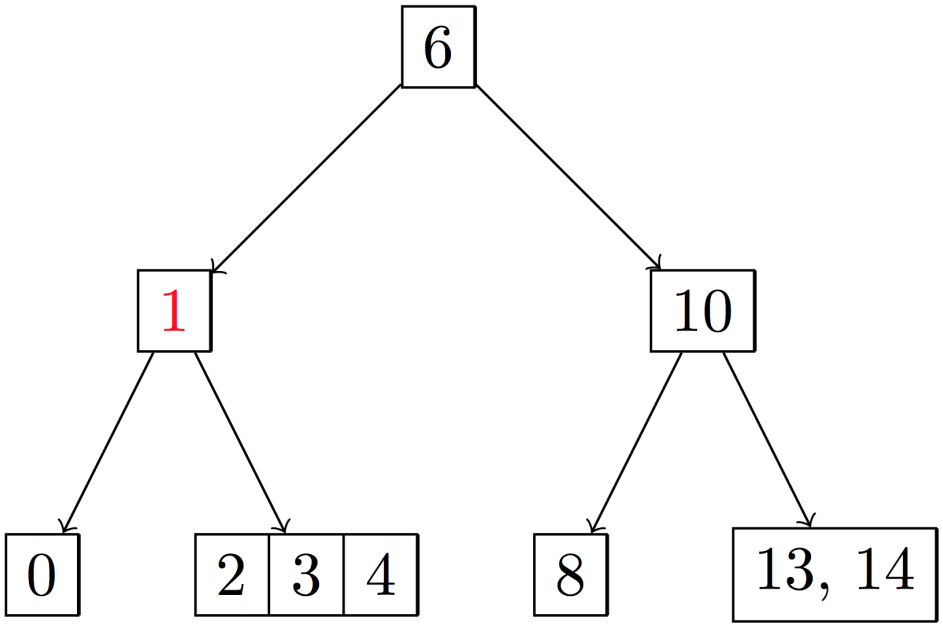
2-3-4 Trees

insert(5)



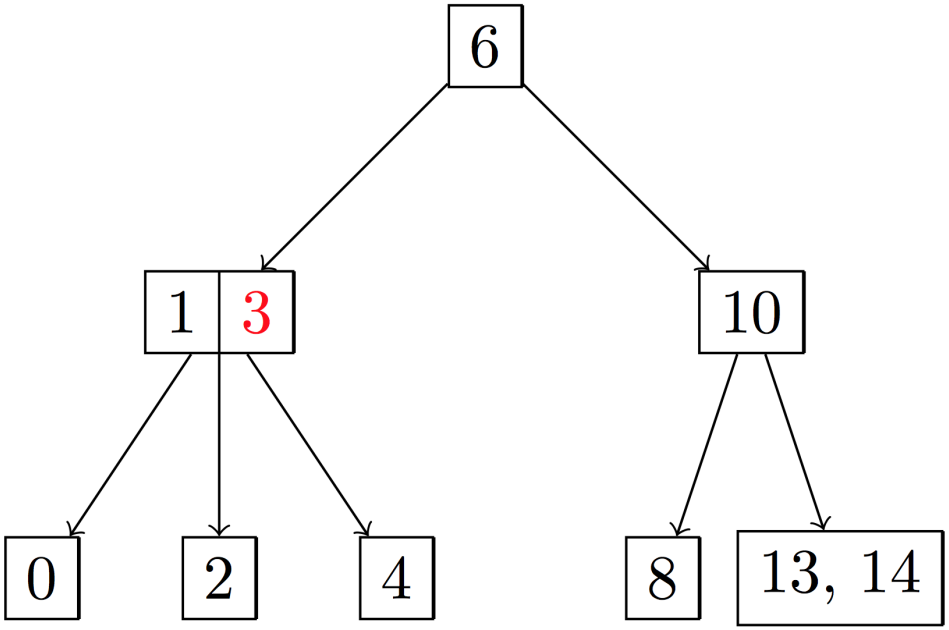
2-3-4 Trees

insert(5)



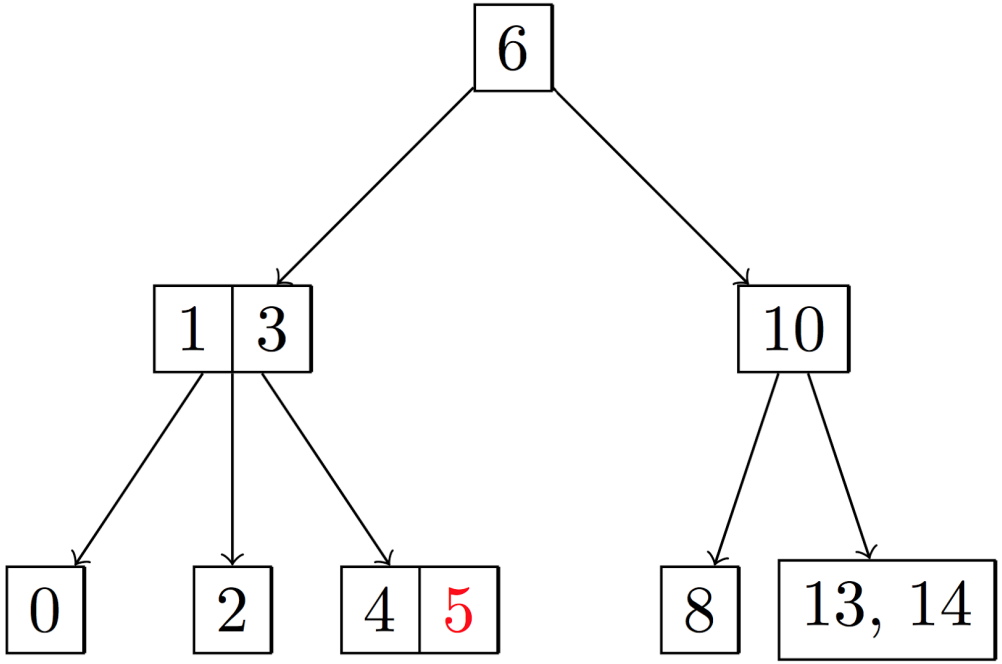
2-3-4 Trees

insert(5)



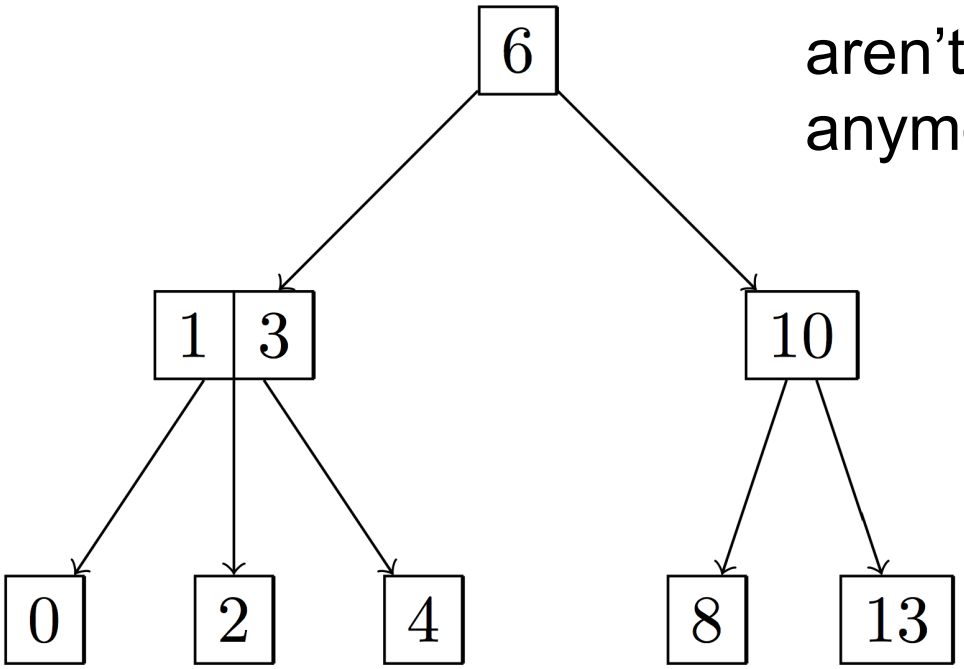
2-3-4 Trees

insert(5)



2-3-4 Trees

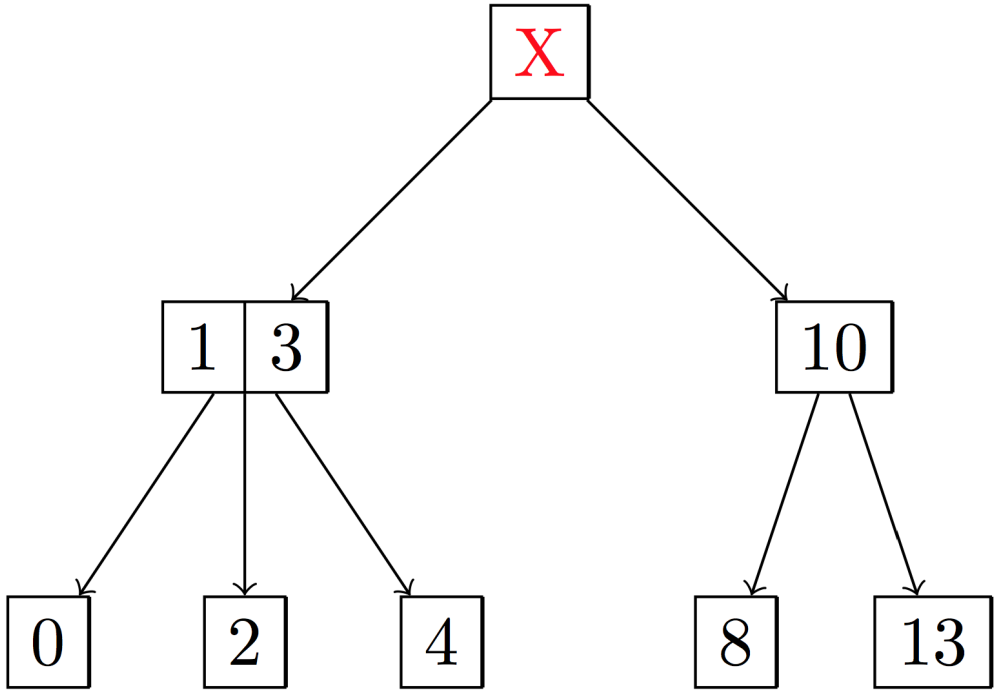
remove(6)



Note that 5 and 14 aren't in the tree anymore.

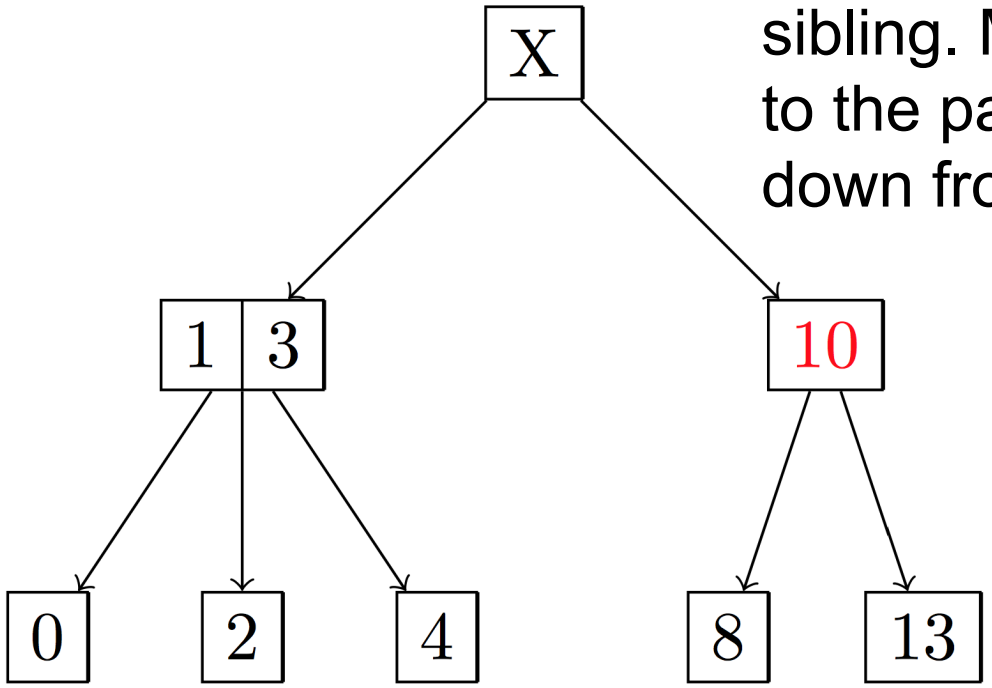
2-3-4 Trees

remove(6)



2-3-4 Trees

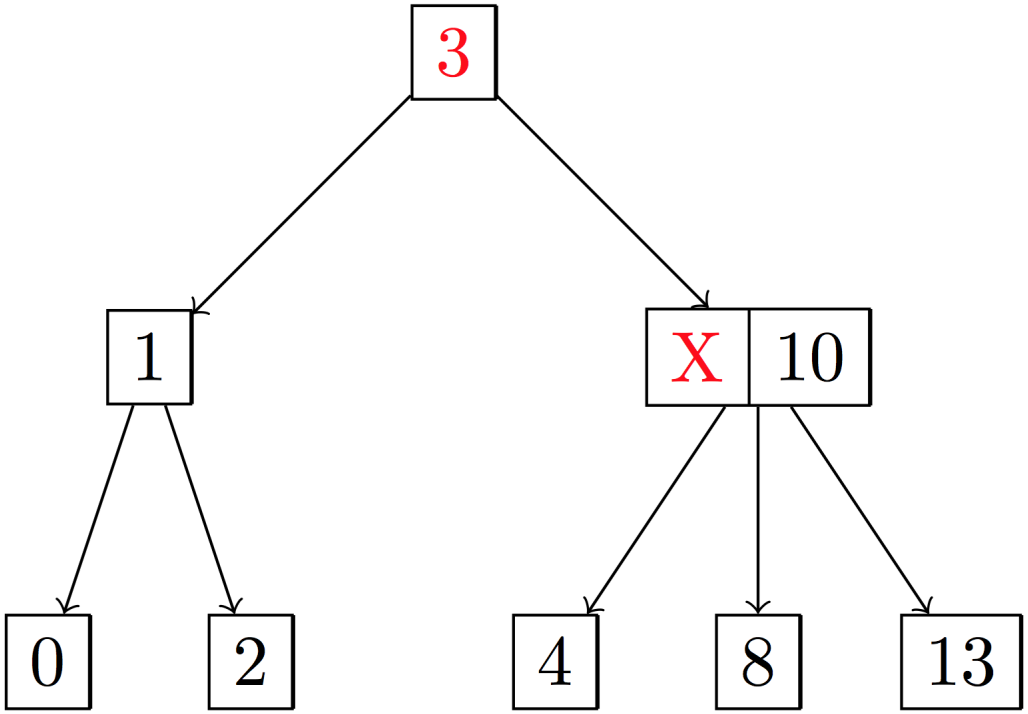
remove(6): Rotation



Idea: Steal a key from a sibling. Move that key up to the parent, and a key down from the parent.

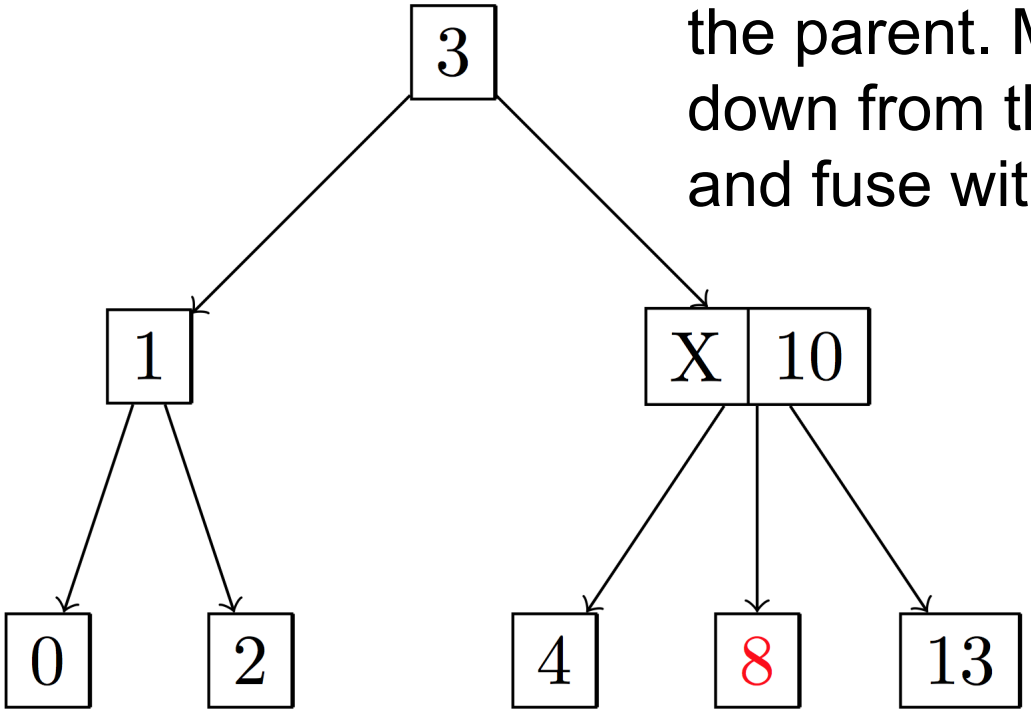
2-3-4 Trees

remove(6): Rotation



2-3-4 Trees

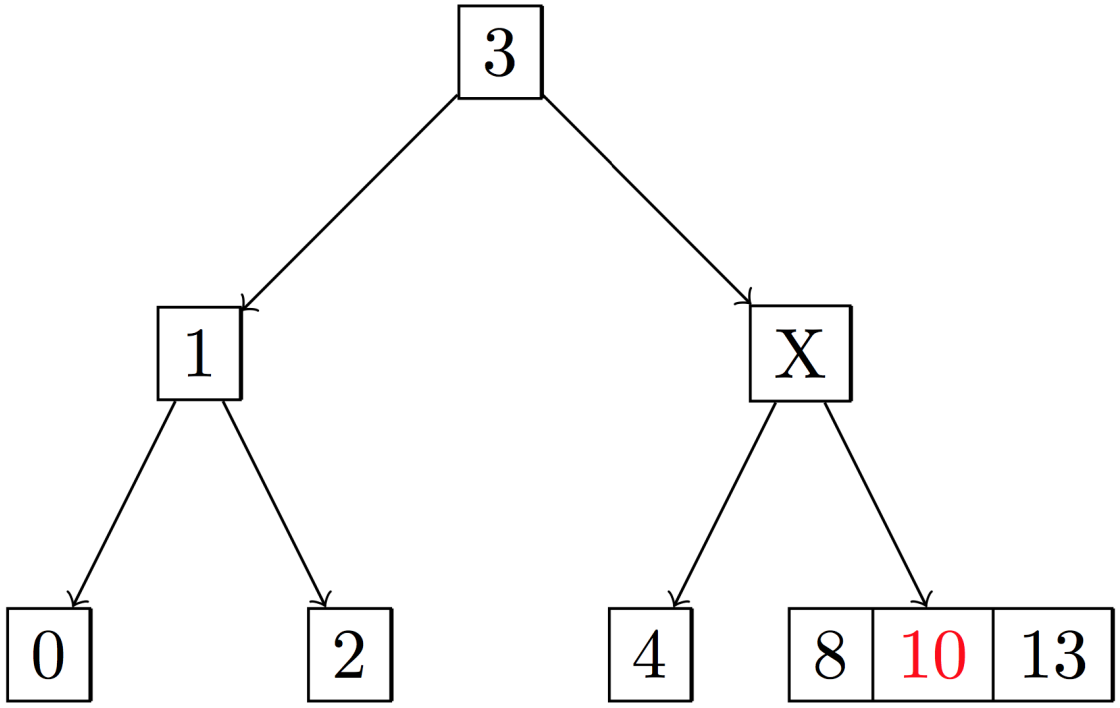
remove(6): Fusion



Idea: Steal a key from the parent. Move a key down from the parent, and fuse with a sibling.

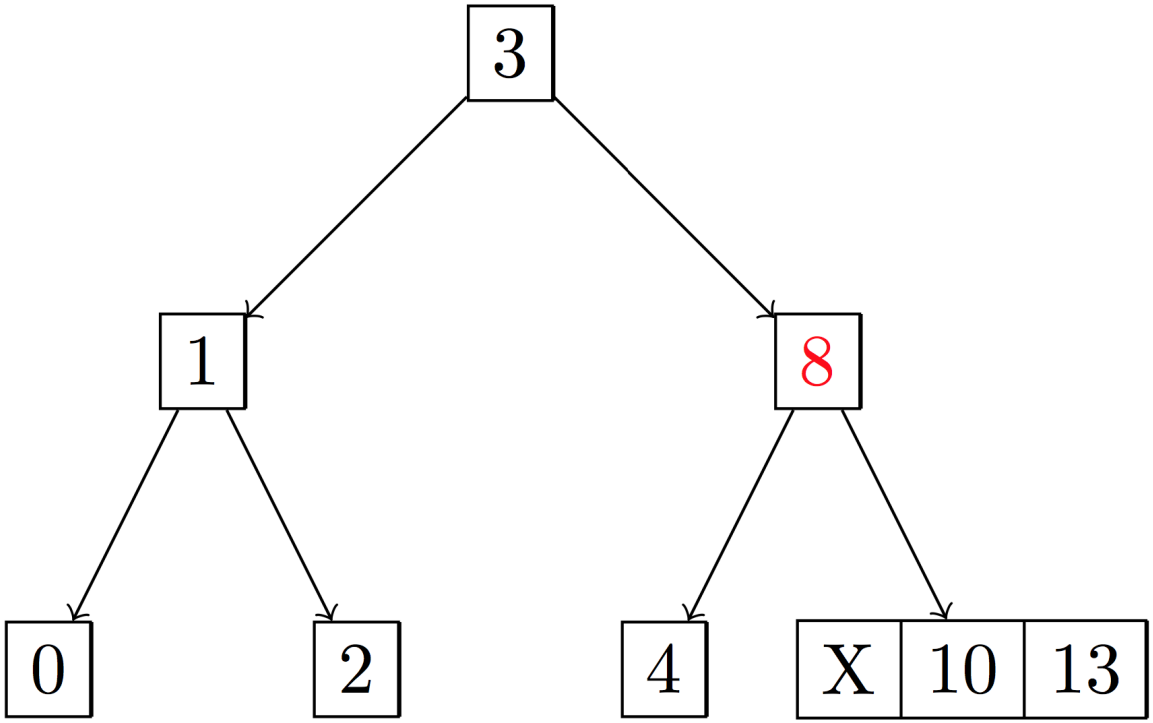
2-3-4 Trees

remove(6): Fusion



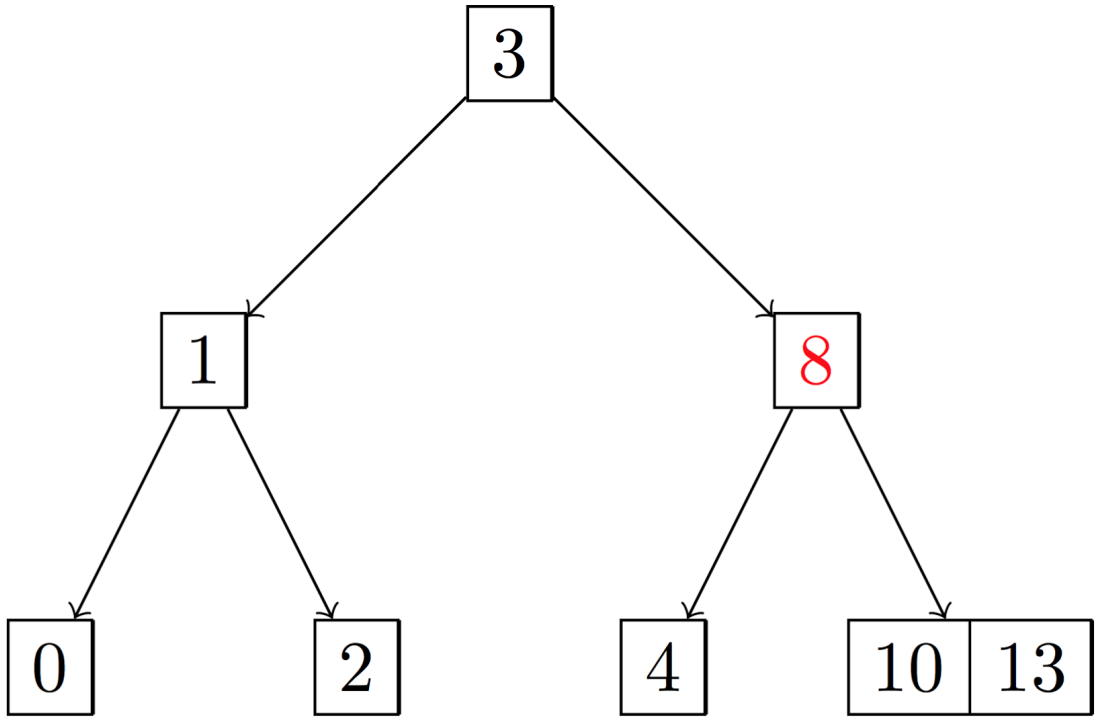
2-3-4 Trees

remove(6)



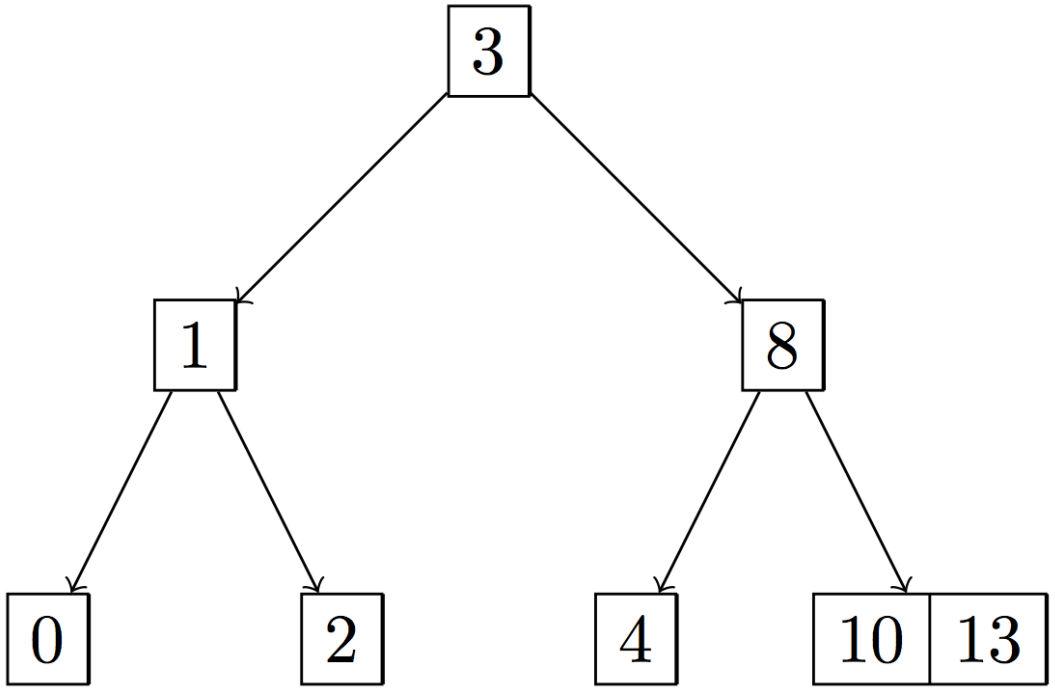
2-3-4 Trees

remove(6)



2-3-4 Trees

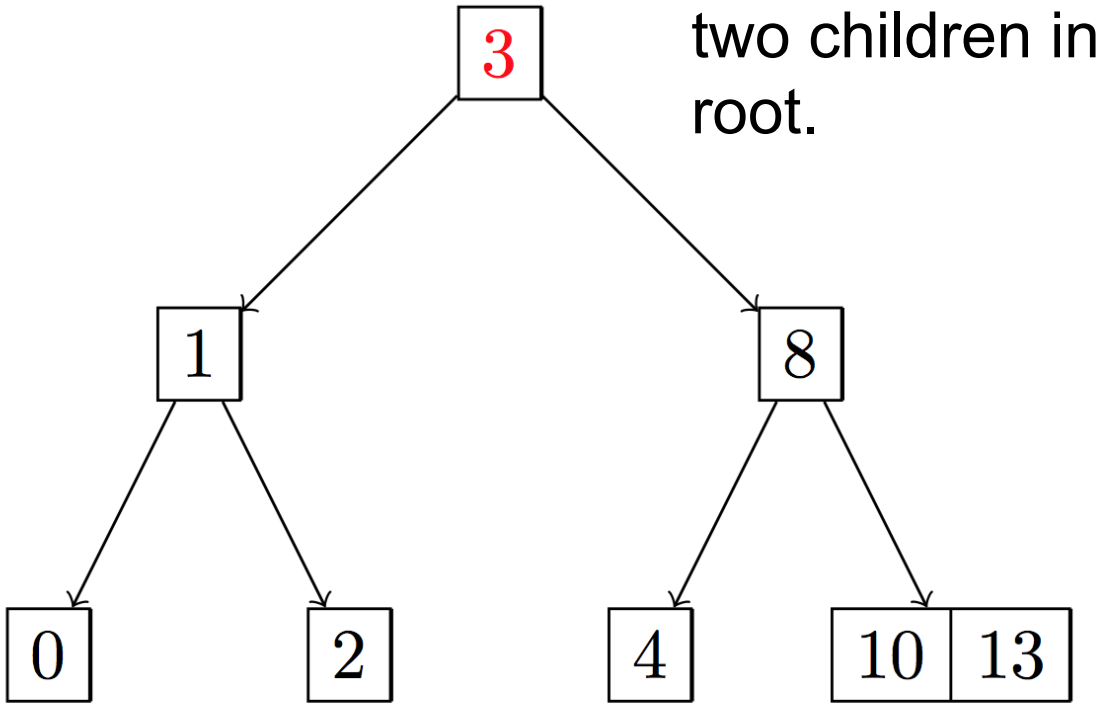
remove(4)



2-3-4 Trees

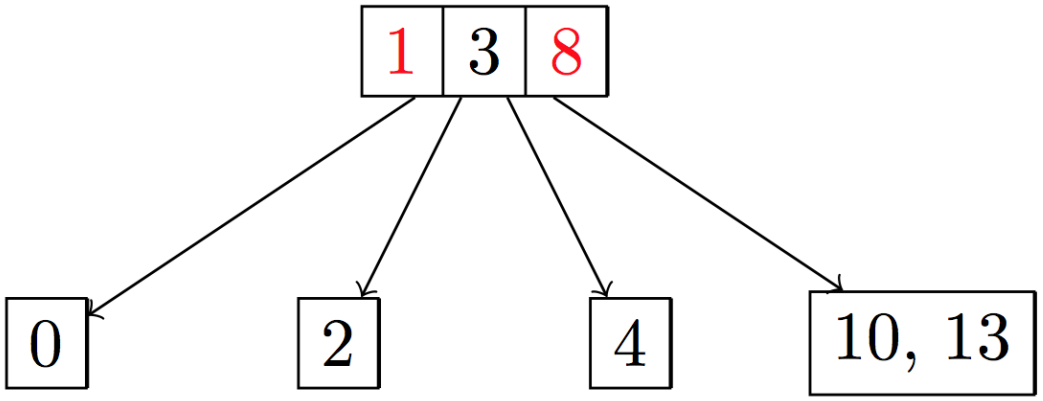
remove(4): Root fusion

Idea: Fuse the root and two children into a new root.



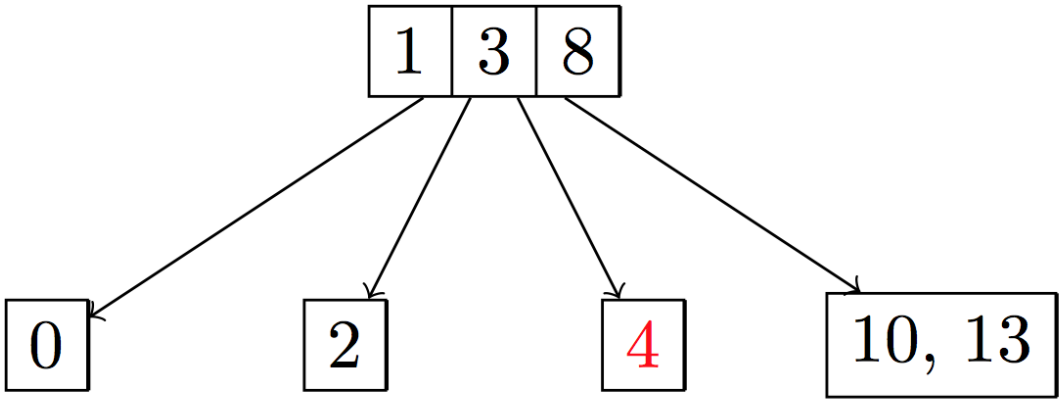
2-3-4 Trees

remove(4): Root fusion



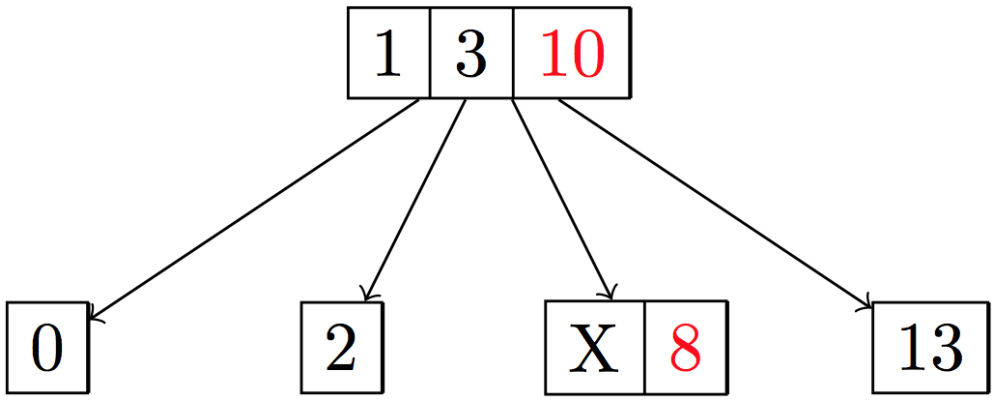
2-3-4 Trees

remove(4): Rotation



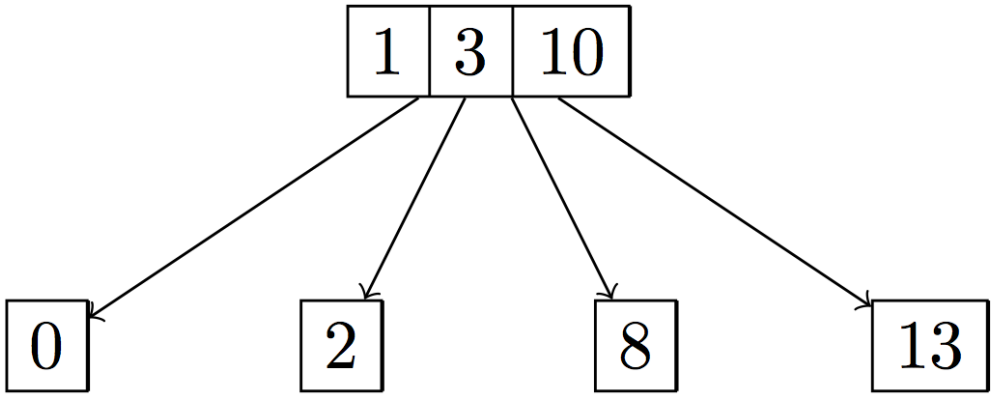
2-3-4 Trees

remove(4): Rotation



2-3-4 Trees

remove(4)



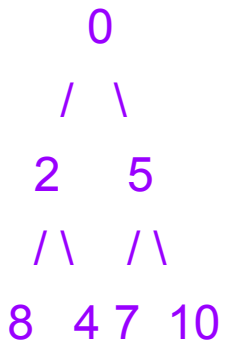
Heaps

Binary Heaps

- Invariants: No child has a value less than its parent, tree is complete
- min(): top element
- insert(): insert element into next spot in the tree and bubble up
- removeMin(): take last element and replace it with min, then bubble that element down

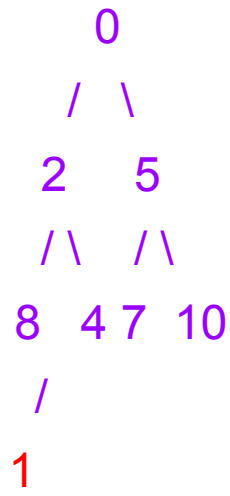
Binary Heaps: Insert

insert(1)



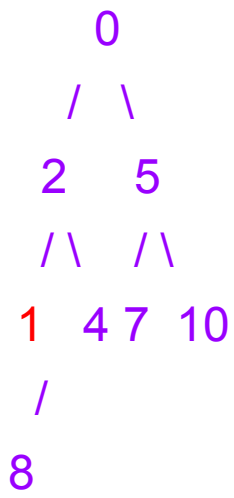
Binary Heaps: Insert

insert(1)



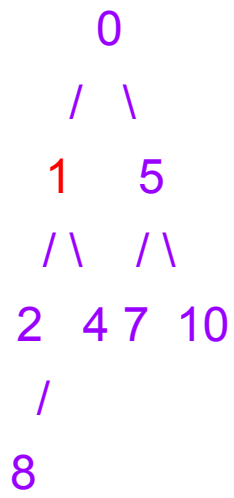
Binary Heaps: Insert

insert(1)



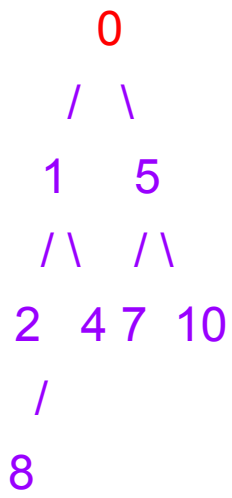
Binary Heaps: Insert

insert(1)



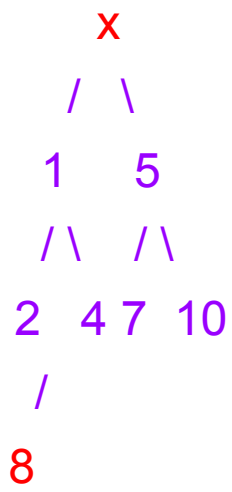
Binary Heaps: Insert

removeMin()



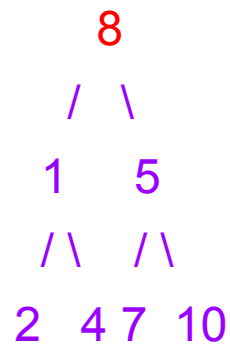
Binary Heaps: Insert

removeMin()



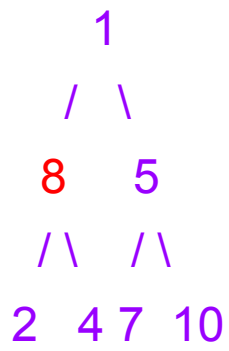
Binary Heaps: Insert

removeMin()



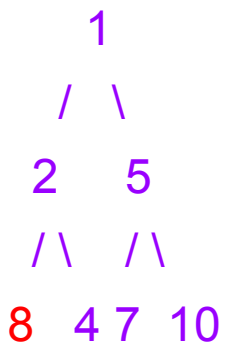
Binary Heaps: Insert

removeMin()



Binary Heaps: Insert

removeMin()

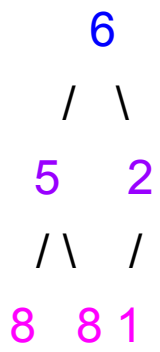


Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1

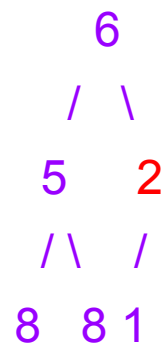
Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1



Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1

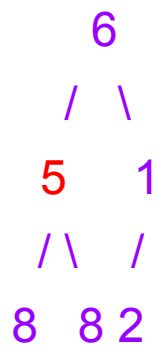


Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1

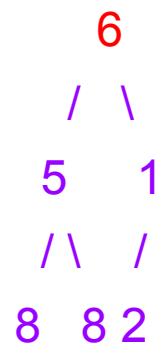
Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1



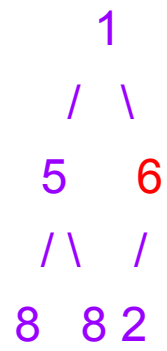
Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1



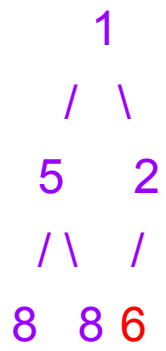
Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1



Binary Heaps: Bottom Up

Array: x, 6, 5, 2, 8, 8, 1



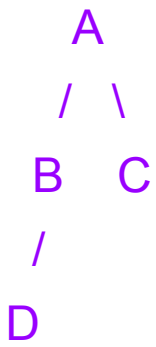
BSTs & Heaps

Draw a binary tree with 4 unique keys that is both a BST *and* a binary heap, or explain why no such tree exists.

BSTs & Heaps

Draw a binary tree with 4 unique keys that is both a BST *and* a binary heap, or explain why no such tree exists.

Not possible. Any valid binary heap with 4 nodes must have the following structure (because binary heaps are complete):



Look at the relationship between A and B. In a valid BST, B must be <= A. In a valid heap, B must be >= A. The only way both these conditions hold true is if A = B. but we have specified that all keys are unique, so the original statement must have no solution. Note: here I assumed a min-heap, but you can easily modify the explanation for a max-heap.

Asymptotic Analysis

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) <= c*h(n), n >= N

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) <= c*h(n), n >= N
- g(n) <= c'*h(n), n >= N'

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) <= c*h(n), n >= N
- g(n) <= c'*h(n), n >= N'
- f(n)*g(n) <= c*h(n)*g(n), n >= N

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) <= c*h(n), n >= N
- g(n) <= c'*h(n), n >= N'
- f(n)*g(n) <= c*h(n)*g(n), n >= N
- f(n)*g(n) <= c*h(n)*g(n) <= c*h(n)*c'*h(n), n >= max{N, N'}

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) <= c*h(n), n >= N
- g(n) <= c'*h(n), n >= N'
- f(n)*g(n) <= c*h(n)*g(n), n >= N
- f(n)*g(n) <= c*h(n)*g(n) <= c*h(n)*c'*h(n), n >= max{N, N'}
- f(n)*g(n) <= c*c'*h(n)*h(n), n >= max{N, N'}

Asymptotic Analysis

- If f(n) is in O(h(n)), and g(n) is also in O(h(n)), prove f(n)*g(n) is in O(h(n)*h(n))

- f(n) < c*h(n), n > N
- g(n) < c'*h(n), n > N'
- f(n)*g(n) < c*h(n)*g(n), n > N
- f(n)*g(n) < c*h(n)*g(n) < c*h(n)*c'*h(n), n > max{N, N'}
- f(n)*g(n) < c*c'*h(n)*h(n), n > max{N, N'}
- f(n)*g(n) is in O(h(n)*h(n))

Big Oh Quickies

Is the following statement True or False? Make sure you understand why!

1. $x^3 \sin(x) \in \Omega(x)$

Big Oh Quickies

Is the following statement True or False? Make sure you understand why!

1. $x^3 \sin(x) \in \Omega(x)$

   False.

   If this were true, then we would have
   $x^3 \sin(x) >= cx$ for all $x >= X$
   for some positive c.
   Consider x = 2*k*pi for some integer k s.t. 2*k*pi >= X.
   Then sin(x) = 0 and cx > 0 no matter what c we choose, which is a contradiction.

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

2. $3^x \in O(x^8)$

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

2. $3^x \in O(x^8)$

False.

Reason: Exponential functions grow much faster than polynomials.

Explanation: If you differentiate 3^x 8 times, you get $(\ln(3))^8 3^x$, but when you differentiate x^8 8 times, you get a constant, which intuitively should mean that 3^x grows faster than x^8, using L'Hopital's Rule and taking x to infinity.

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

3. $3^x \in O(x!)$

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

3. $3^x \in O(x!)$

True.

Reason:
$3x! = 3 (1 * 2 * 3 * \ldots$ x times$) = 3 * 6 * 9 * 12 * 15 * 18 \ldots$
$> 3 * 3 * 3 \ldots$
$= 3^x$. So, c = 3, X = 1

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

4. $7x^2 + 45x + 9 \in O(x^2)$

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

4. $7x^2 + 45x + 9 \in O(x^2)$

True.

Choose c = 7 + 45 + 9 = 61, X = 1. (This is similar to the proof given to you in the lab on asymptotic analysis.)

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

5. $100n + 100x \in \Theta(n+x)$

Big Oh Quickies (contd.)

Is the following statement True or False? Make sure you understand why!

5. $100n + 100x \in \Theta(n+x)$

True. Proof: We need to prove that

$d(n+x) <= 100n + 100x <= c(n+x)$ for all $n >= N$, $x >= X$.

i.e. we need to prove that

$d(n+x) <= 100(n+x) <= c(n+x)$ for all $n >= N$, $x >= X$.

But it is trivial to choose constants that make this true. Choose d<=100, c>=100, and N and X are arbitrary. For example,

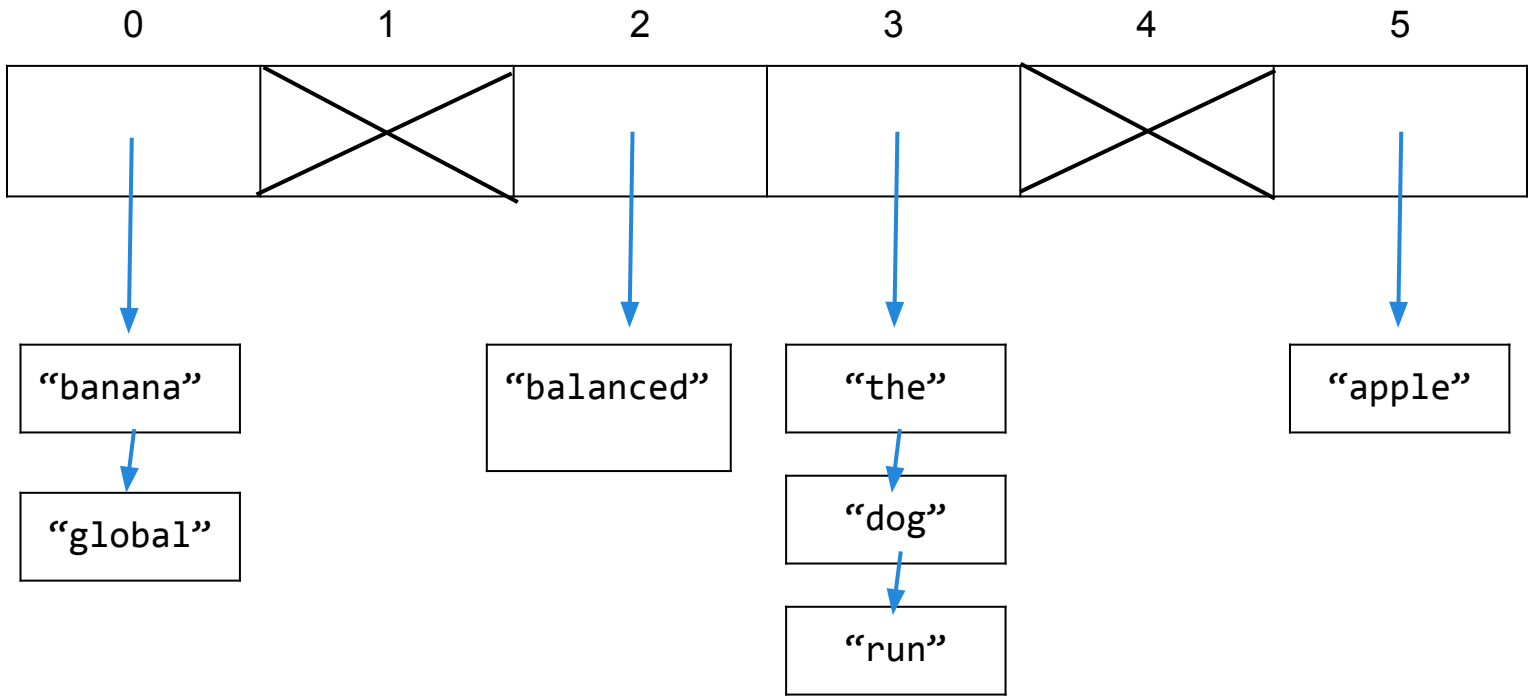
$99(n+x) <= 100(n+x) <= 101(n+x)$ for all $n >= 1$, $x >= 1$.

Hash Tables

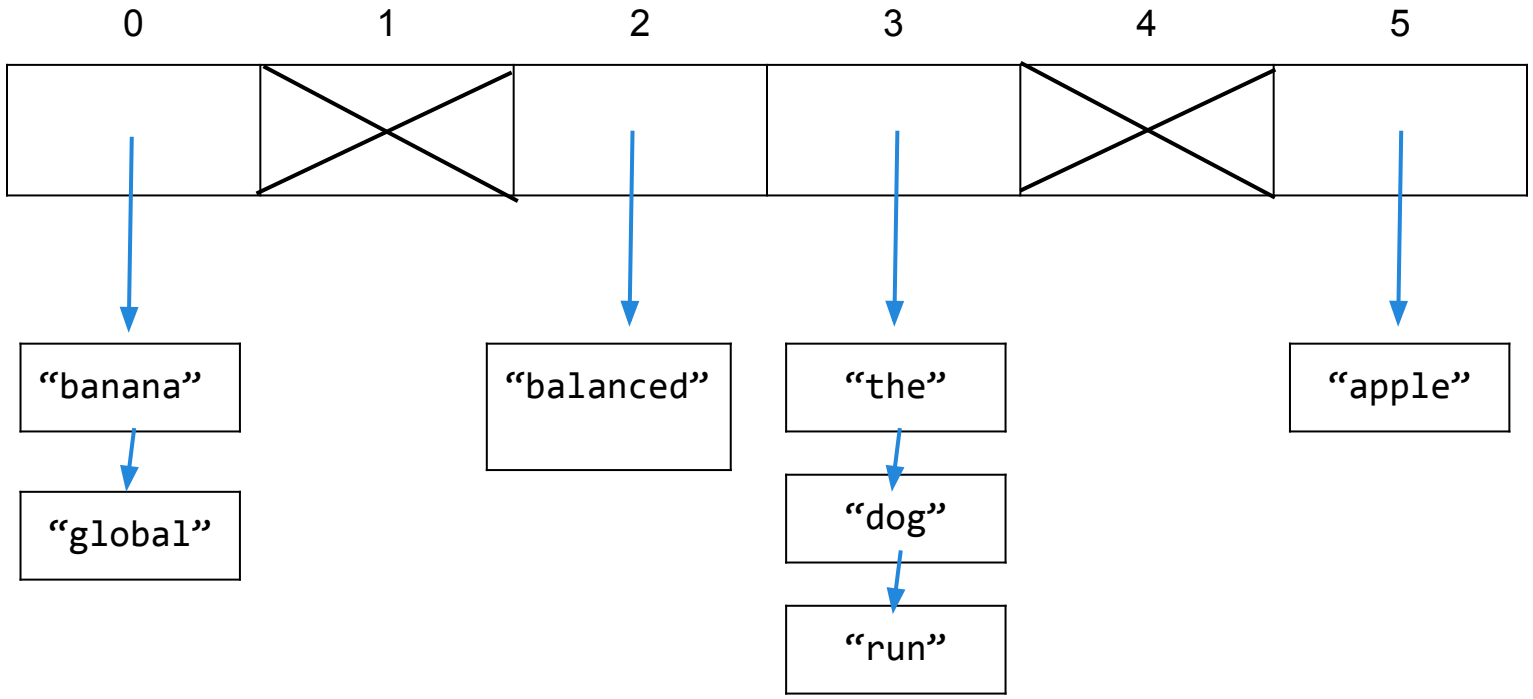
Hash Tables

- One implementation of a **dictionary**
- Allows for very fast operations:
  - `find(key)`
  - `insert(entry)`
  - `remove(key)`
- Θ(1) usually, O(n) worst-case
- Implemented as an array of linked lists

Hash Tables

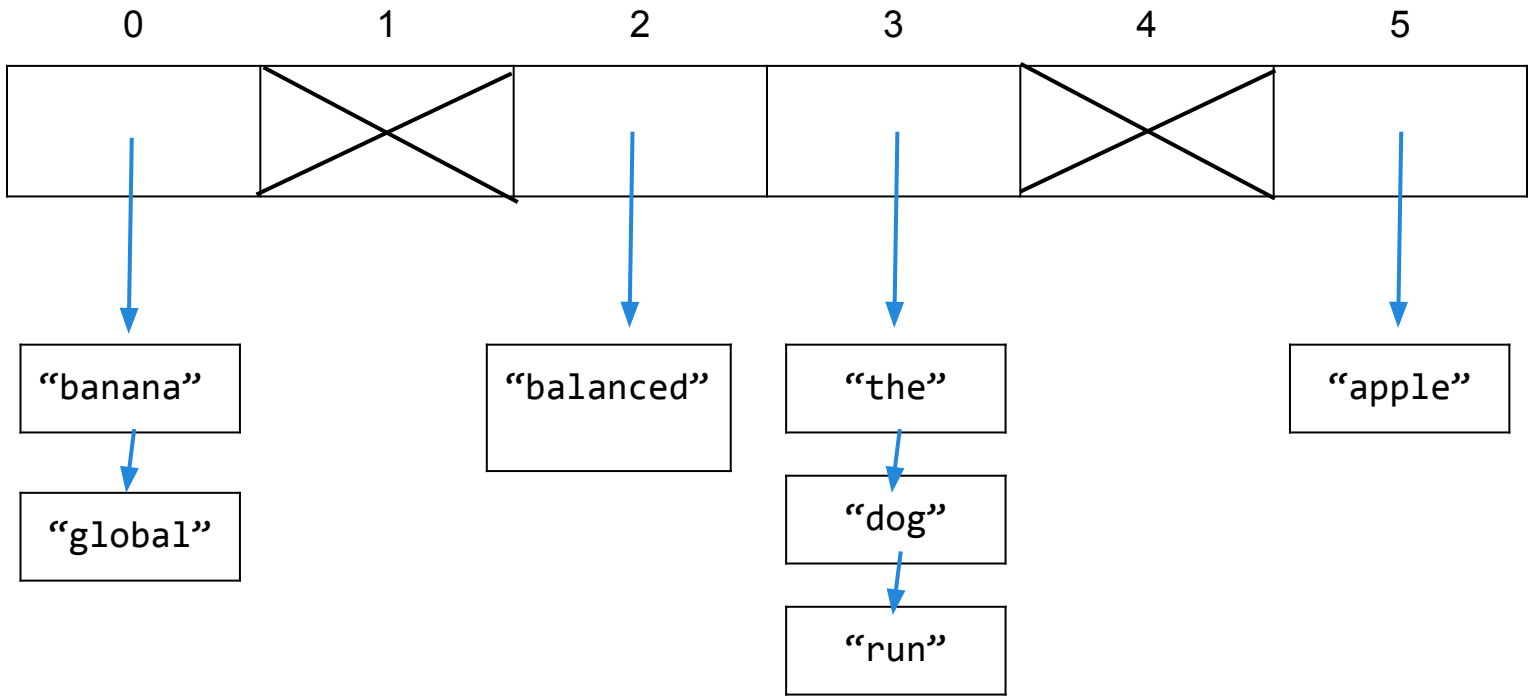


Hash Tables



What is the load factor?

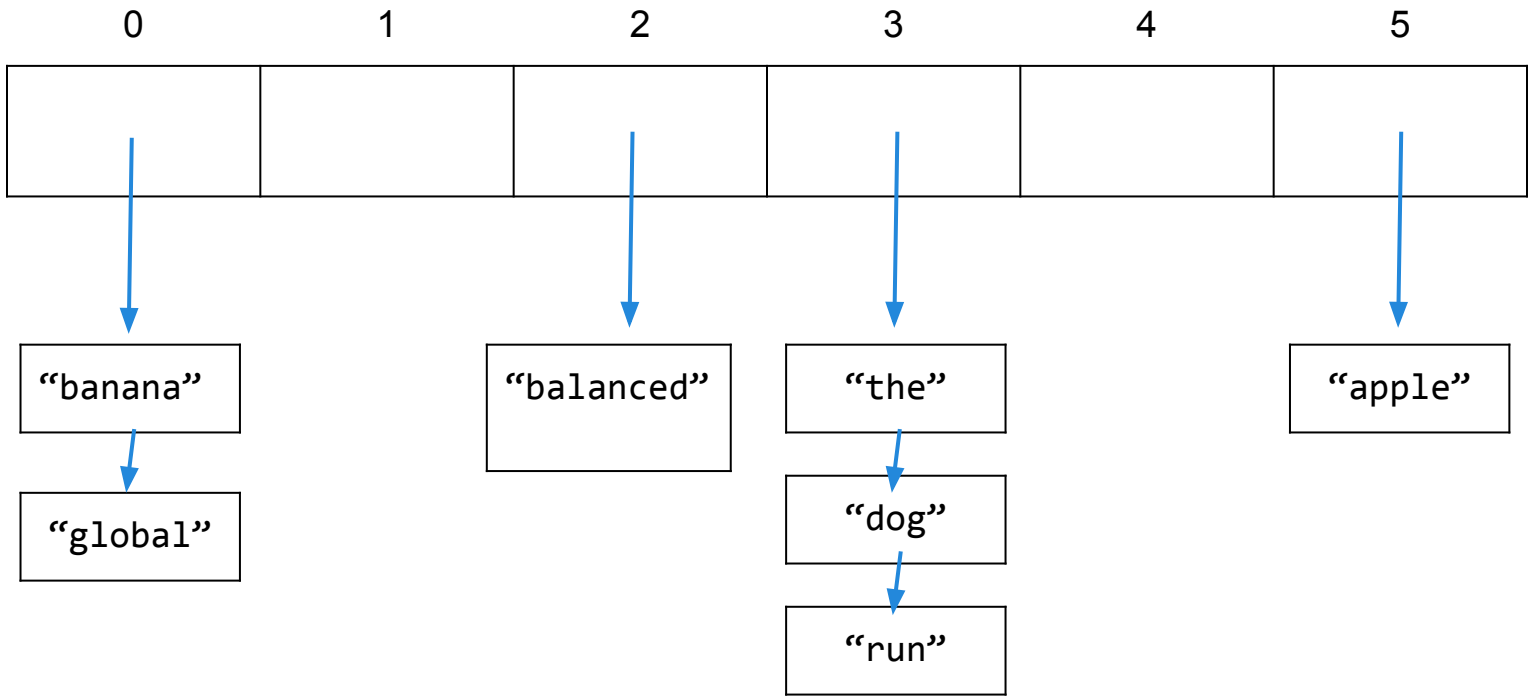
Hash Tables



What is the load factor?

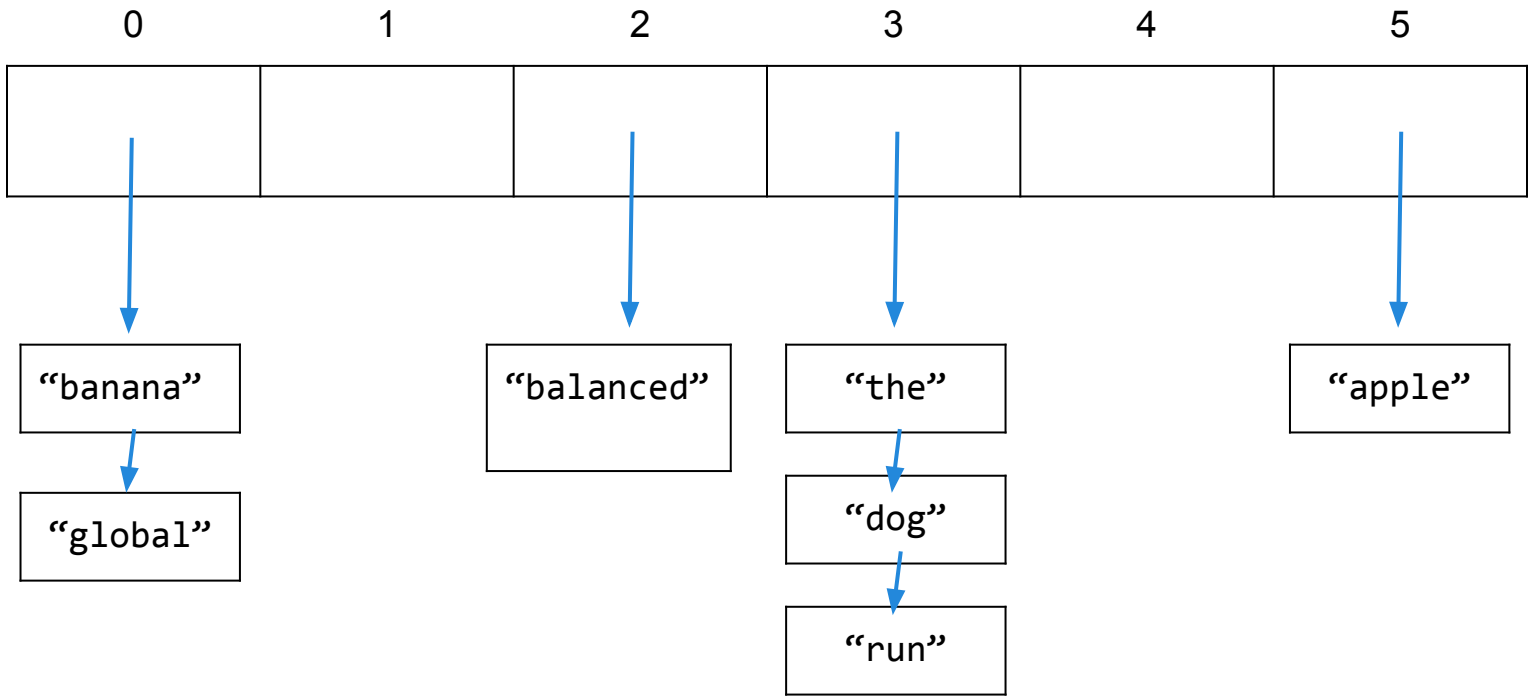
n/N = 7 / 6 ~ 1.17 Very high!

Hash Tables



Which bucket does an object hash to?

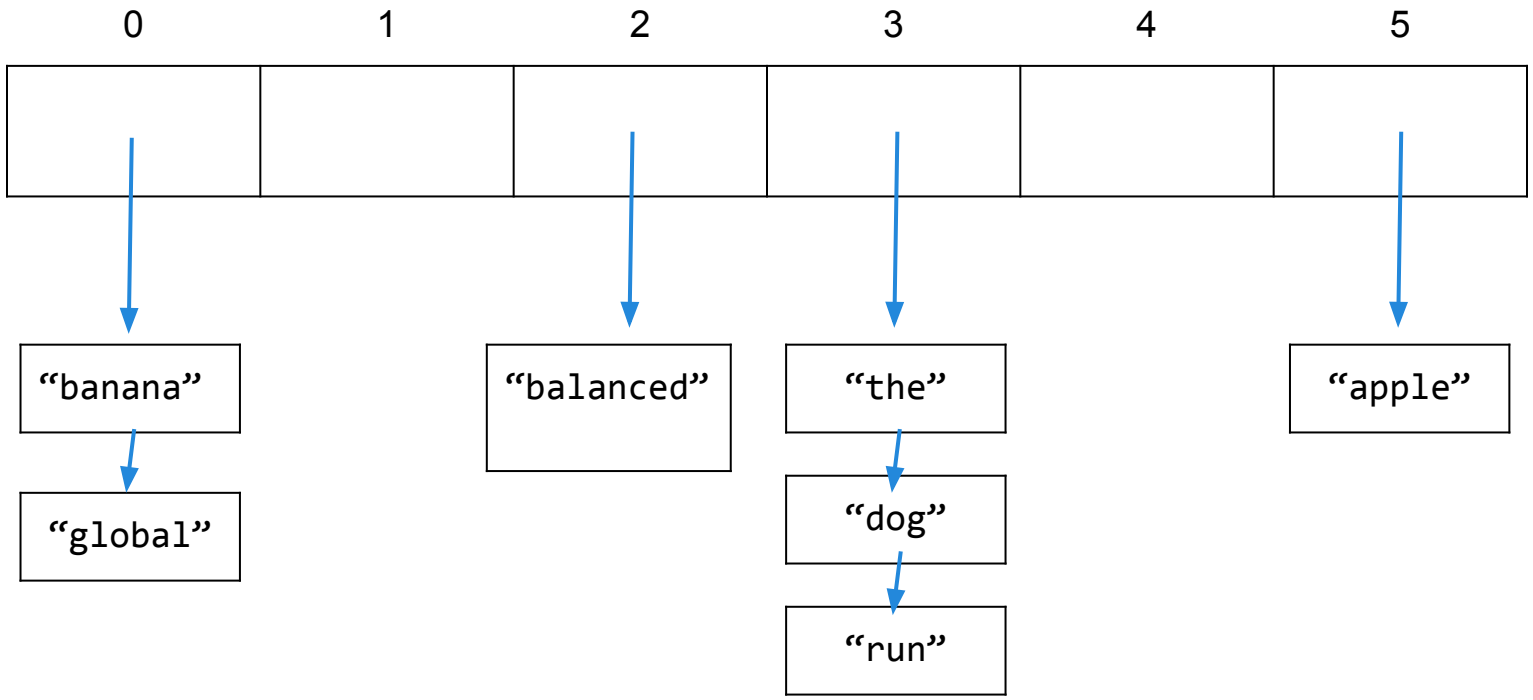
Hash Tables



Which bucket does an object hash to?

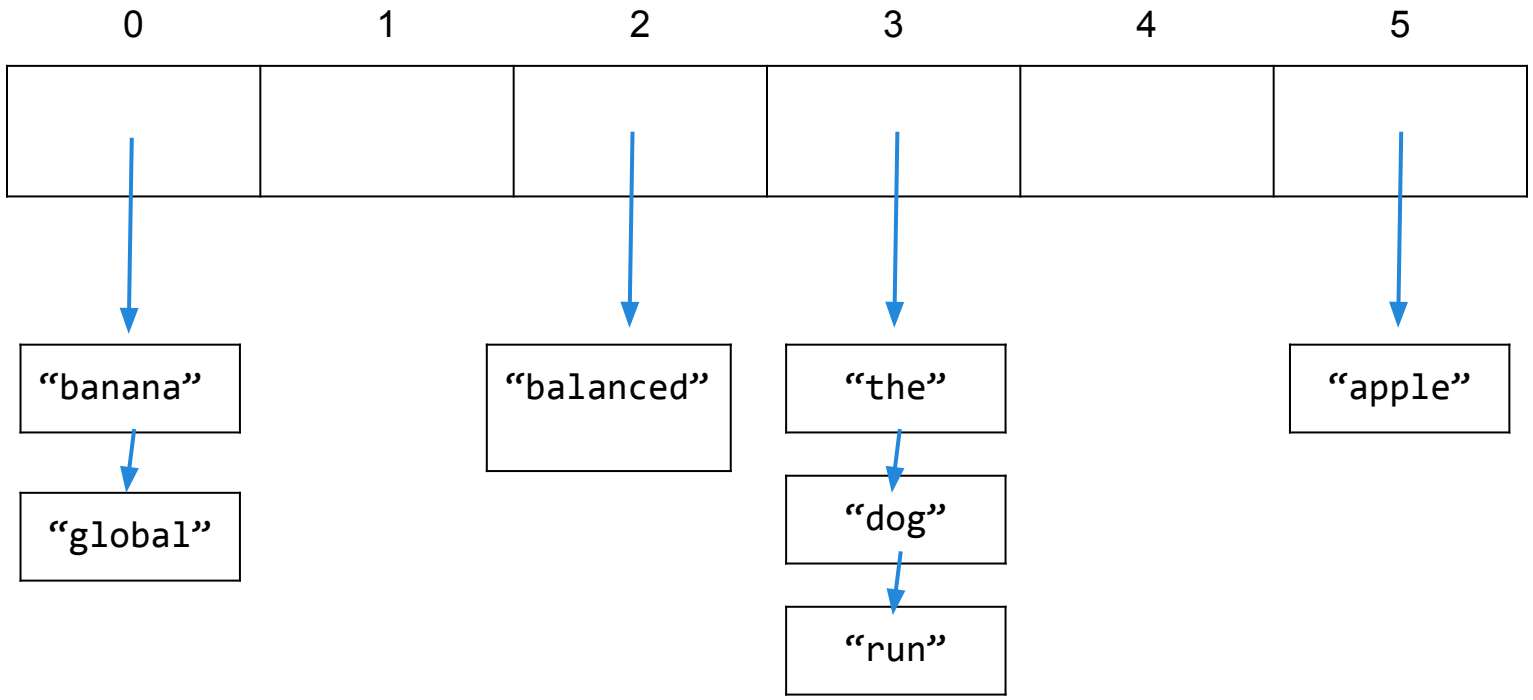
`compressionFunction(obj.hashCode())`

Hash Tables



Why do we need chaining?

Hash Tables



Why do we need chaining?

It’s one way to deal with collisions.

Hash Tables

What's wrong with the following hash functions?

```
int hashCode(String s) {
    return s.length();
}

int hashCode(Integer i) {
    return i*3571;
}
```

Hash Tables

What's wrong with the following hash functions?

```
int hashCode(String s) {
    return s.length();
}
```

Most strings are relatively short, which means a lot of strings will have the same or very similar hashCodes, while very few strings will hash to large numbers. In short the hashCodes will not be evenly distributed!

Hash Tables

What's wrong with the following hash functions?

```
int hashCode(Integer i) {
    return i*3571;
}
```

If your application uses mostly multiples of 4, for example, this function will make it so all of the hashCodes are still multiples of 4 coming out. This means the hashCode is not very likely to be evenly distributed.

Hash Tables

What's wrong with the following compression function? Assume `arraySize` is the number of buckets.

```
int compress(int hashCode) {
    return hashCode % arraySize;
}
```

Hash Tables

What's wrong with the following compression function? Assume `arraySize` is the number of buckets.

```
int compress(int hashCode) {
    return hashCode % arraySize;
}
```

hashCode could be negative, which would lead to a negative output. Additionally, we're not doing anything to jumble the bits more. Any pattern in the hashCode (e.g. only even hashCodes) will carry over to the output of the compression function, which means many buckets may go unused.

Hash Tables

True or false?

1. Hash tables are one implementation of an ordered dictionary.

2. Two objects have the same `hashCode` if and only if they are `.equals()`

3. When resizing a hash table, each entry has to be individually hashed and compressed into the new hash table.

Hash Tables

True or false?

1. Hash tables are one implementation of an ordered dictionary. False. They're not ordered!

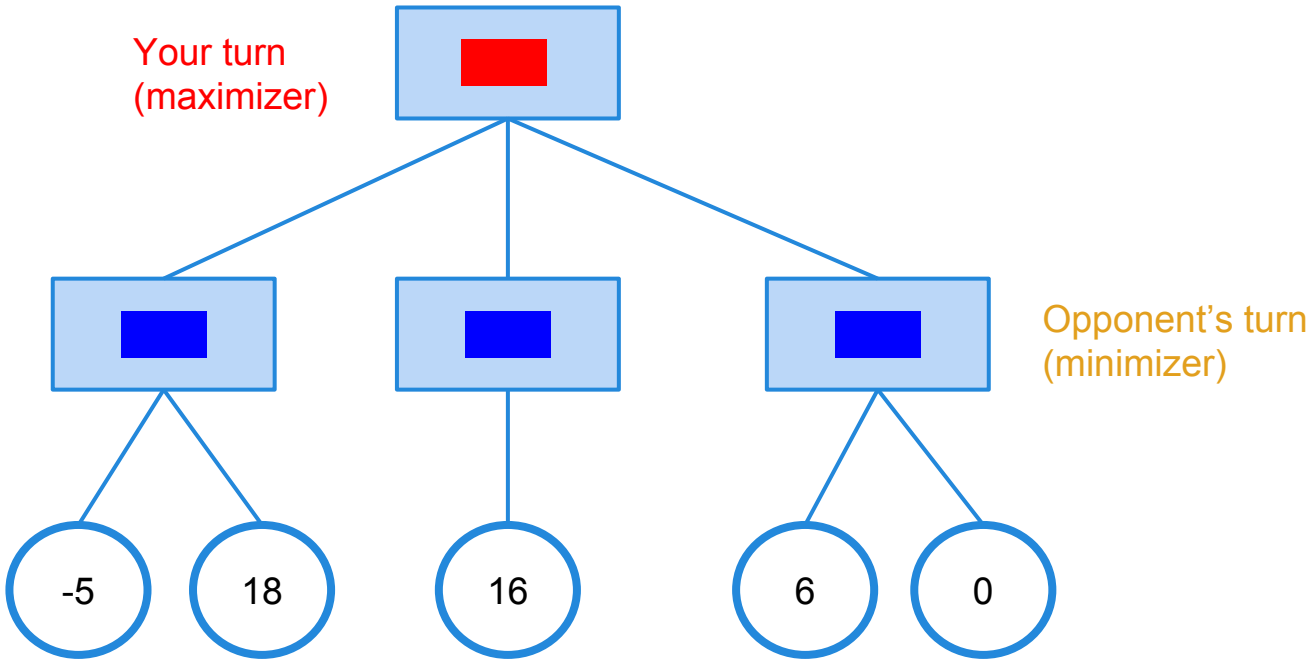
2. Two objects have the same `hashCode` if and only if they are `.equals()` False. Two objects with the same hashCode aren't necessarily `.equals()`

3. When resizing a hash table, each entry has to be individually hashed and compressed into the new hash table. True. Necessary so all the elements are in the right place

Minimax/Alpha Beta

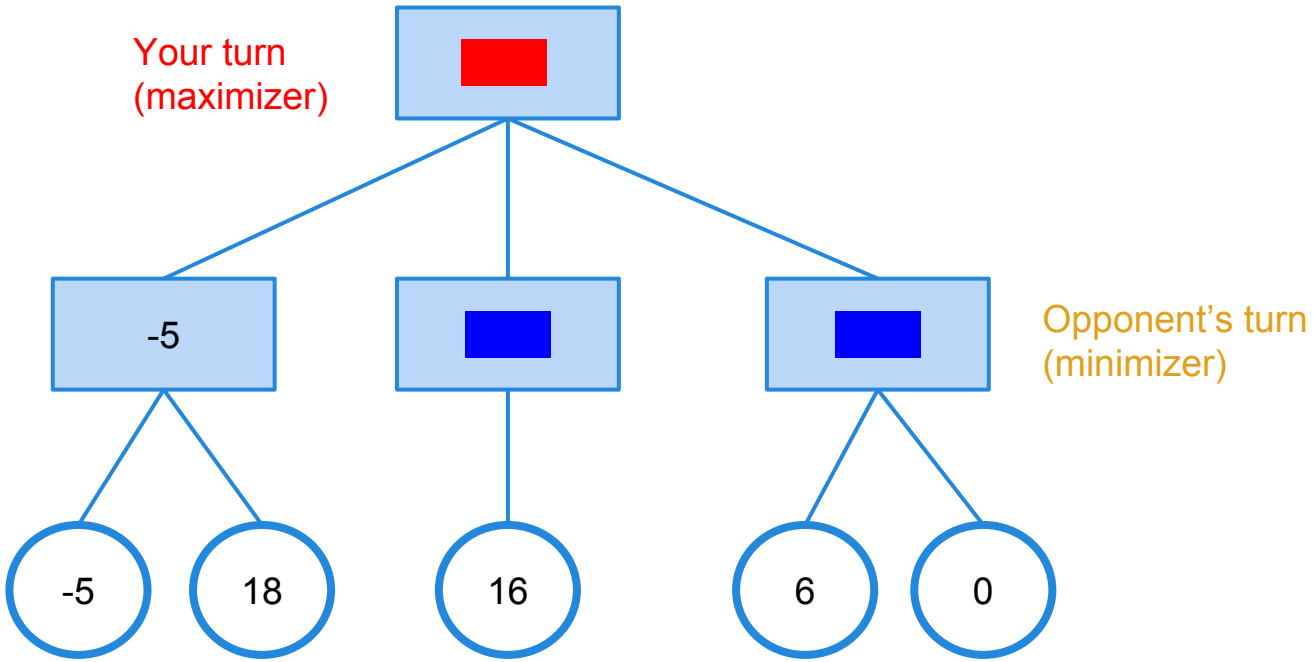
Minimax

Complete minimax on the tree below:



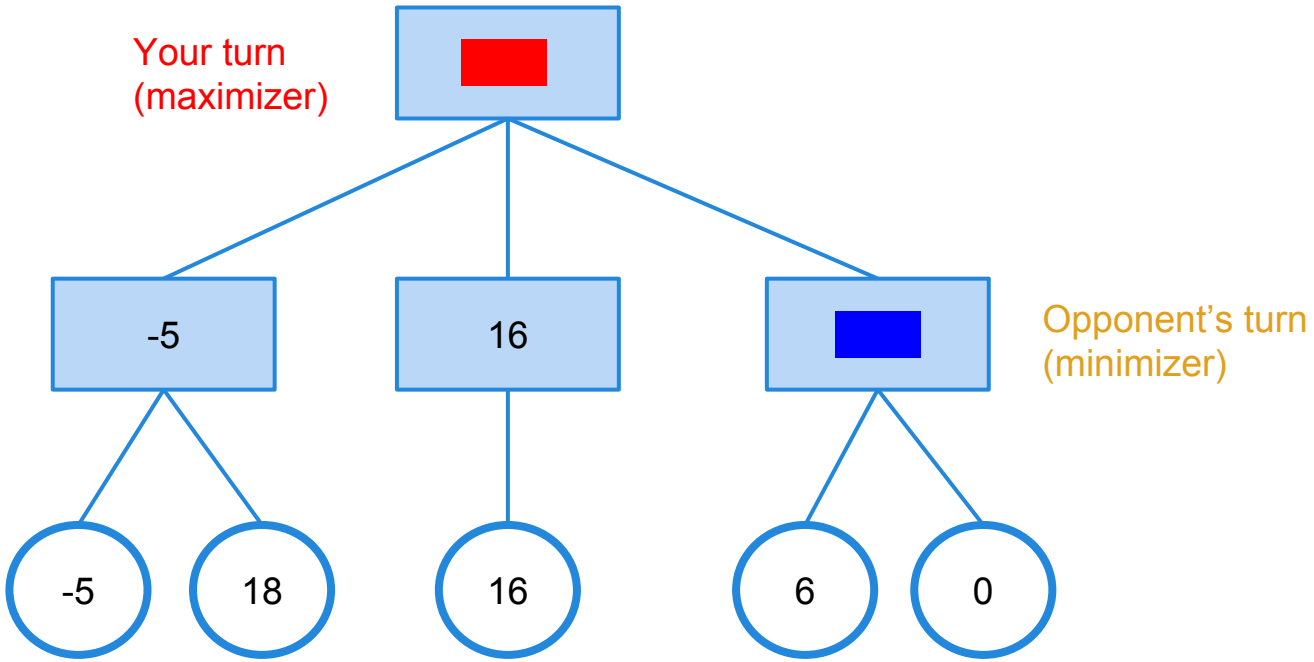
Minimax

Complete minimax on the tree below:



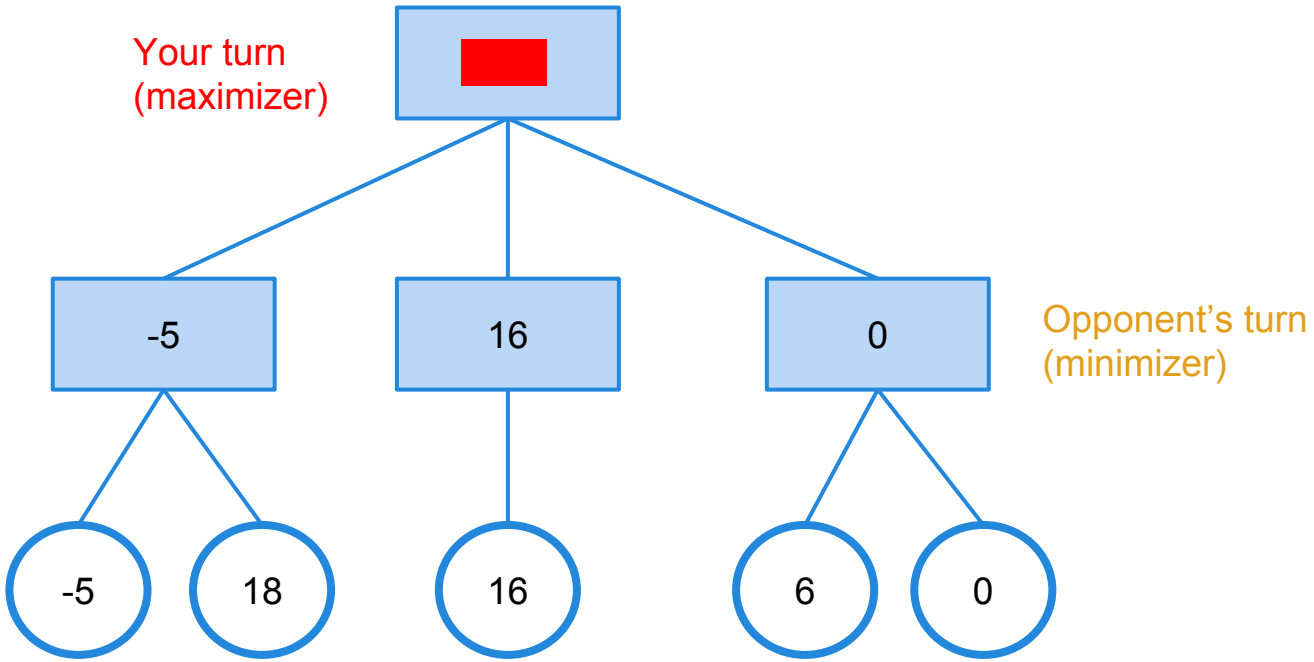
Minimax

Complete minimax on the tree below:



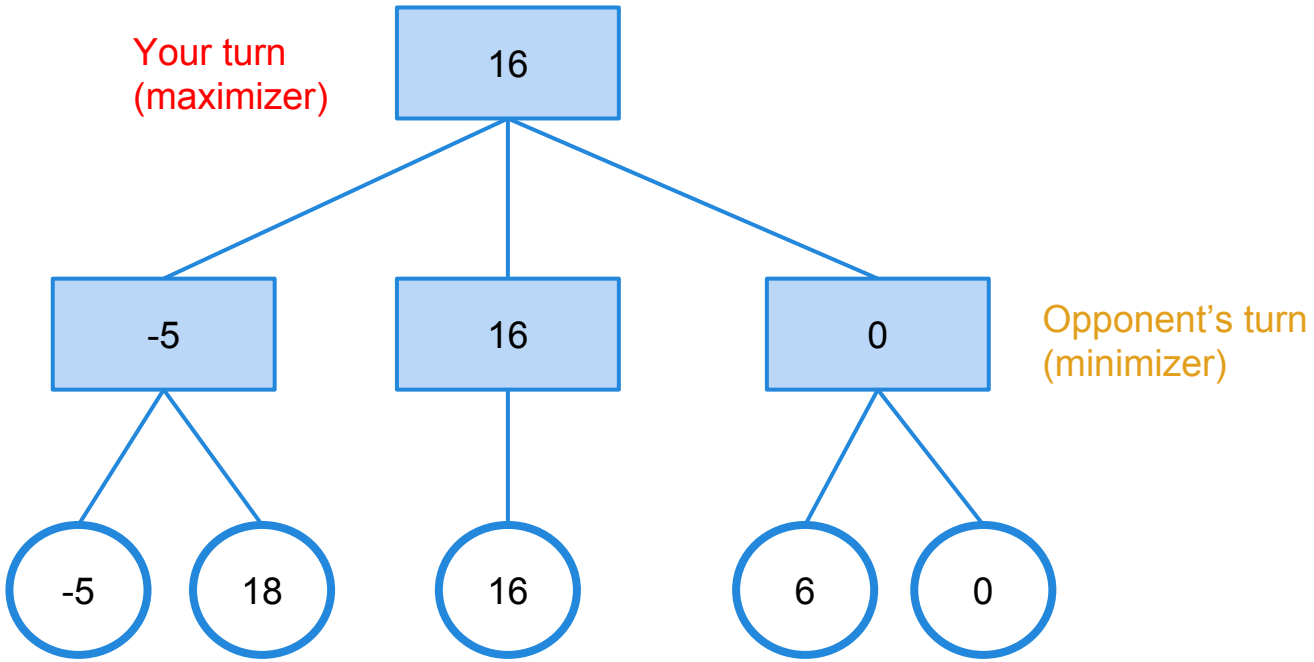
Minimax

Complete minimax on the tree below:



Minimax

Complete minimax on the tree below:



Alpha Beta Pruning

Idea: make minimax faster!

α - best guaranteed score seen so far for YOU

- YOU = maximizer node, YOU search for HIGHER scores
- Starts at negative infinity

β - best guaranteed score seen so far for your OPPONENT

- OPPONENT = minimizer node, searches for LOWER scores
- Starts at positive infinity

Prune Cases

- prune at a minimizer node whose β value is less than or equal to the α value

- prune at a maximizer node whose α value is greater than or equal to the β value

Alpha Beta Pruning

Idea: make minimax faster!

α - best guaranteed score seen so far for YOU

- YOU = maximizer node, YOU search for HIGHER scores
- Starts at negative infinity

β - best guaranteed score seen so far for your OPPONENT

- OPPONENT = minimizer node, searches for LOWER scores
- Starts at positive infinity

Prune Cases

Basically, when $\alpha \geq \beta$

Alpha Beta Pruning

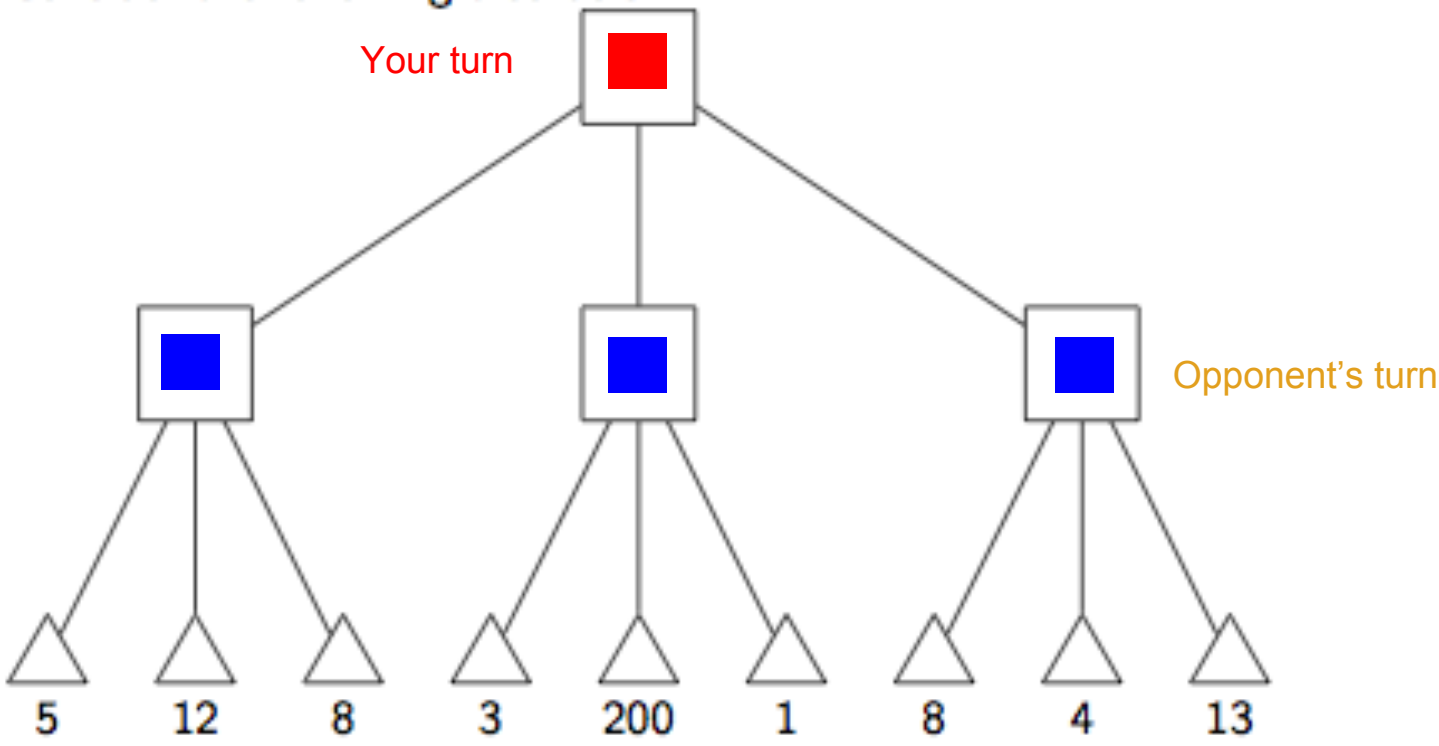
How to:

1. Do the minimax search
2. Keep track of alpha, beta
   a. if you are maximizing, update alpha
   b. if you are minimizing, update beta
3. If you ever see **α** >= **β**, PRUNE

Note*: **α** and **β** are INHERITED from parent nodes.

Alpha Beta Pruning

Consider the following tree below:



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

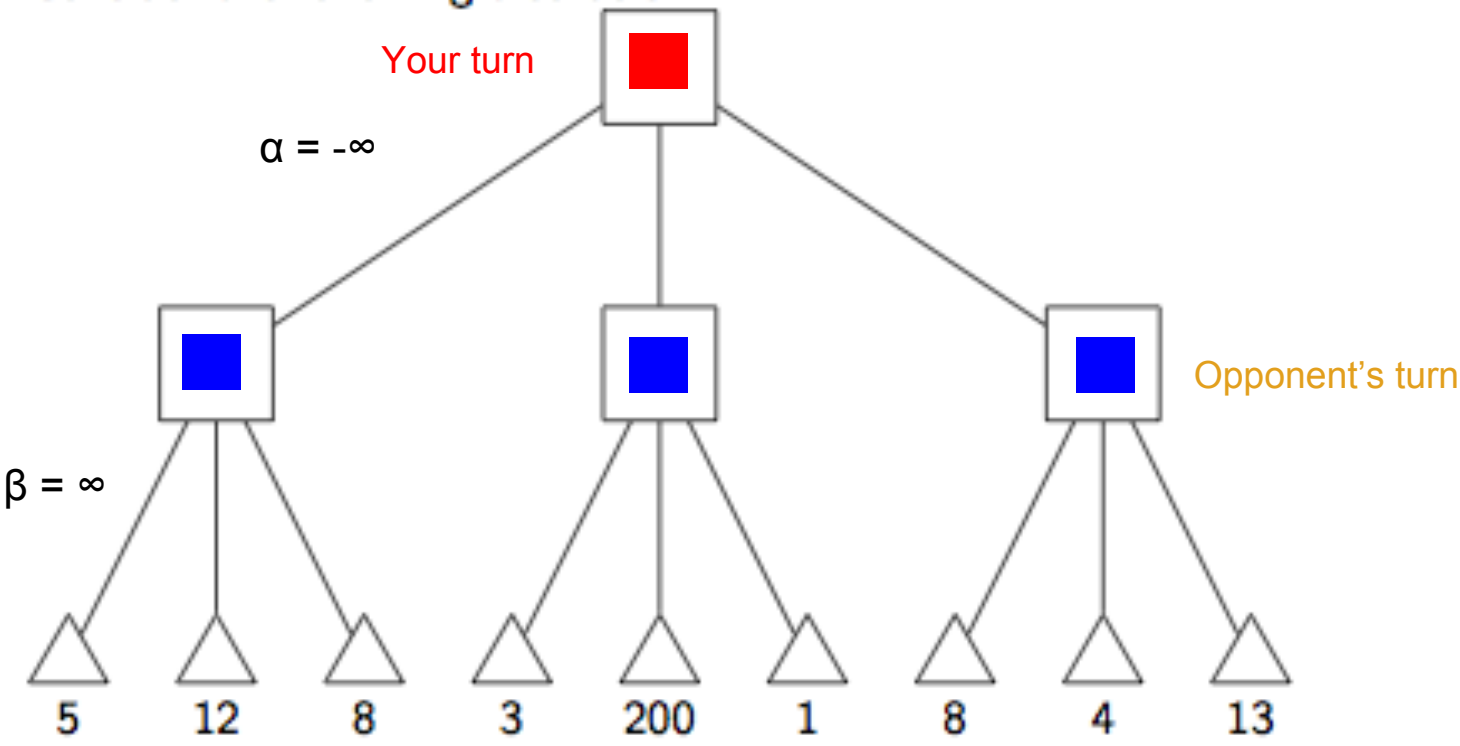
Also assume we traverse children from left to right.

A few things to note, before looking at the solution

- Everyone has their own way of “notating” the alpha-beta search
- The following way tries to keep variables clean, by ONLY keeping track of the current variable (alpha or beta, whether we are at a minimizer or maximizer node)
- The value of the variable (alpha or beta) going INTO a branch is the variables value BEFORE evaluating the node.
- After looking at the node, the variable value MAY CHANGE
- Professor Shewchuk has a more extensive system, which you can see here: http://www.cs.berkeley.edu/~jrs/61b/lec/17.pdf

Alpha Beta Pruning

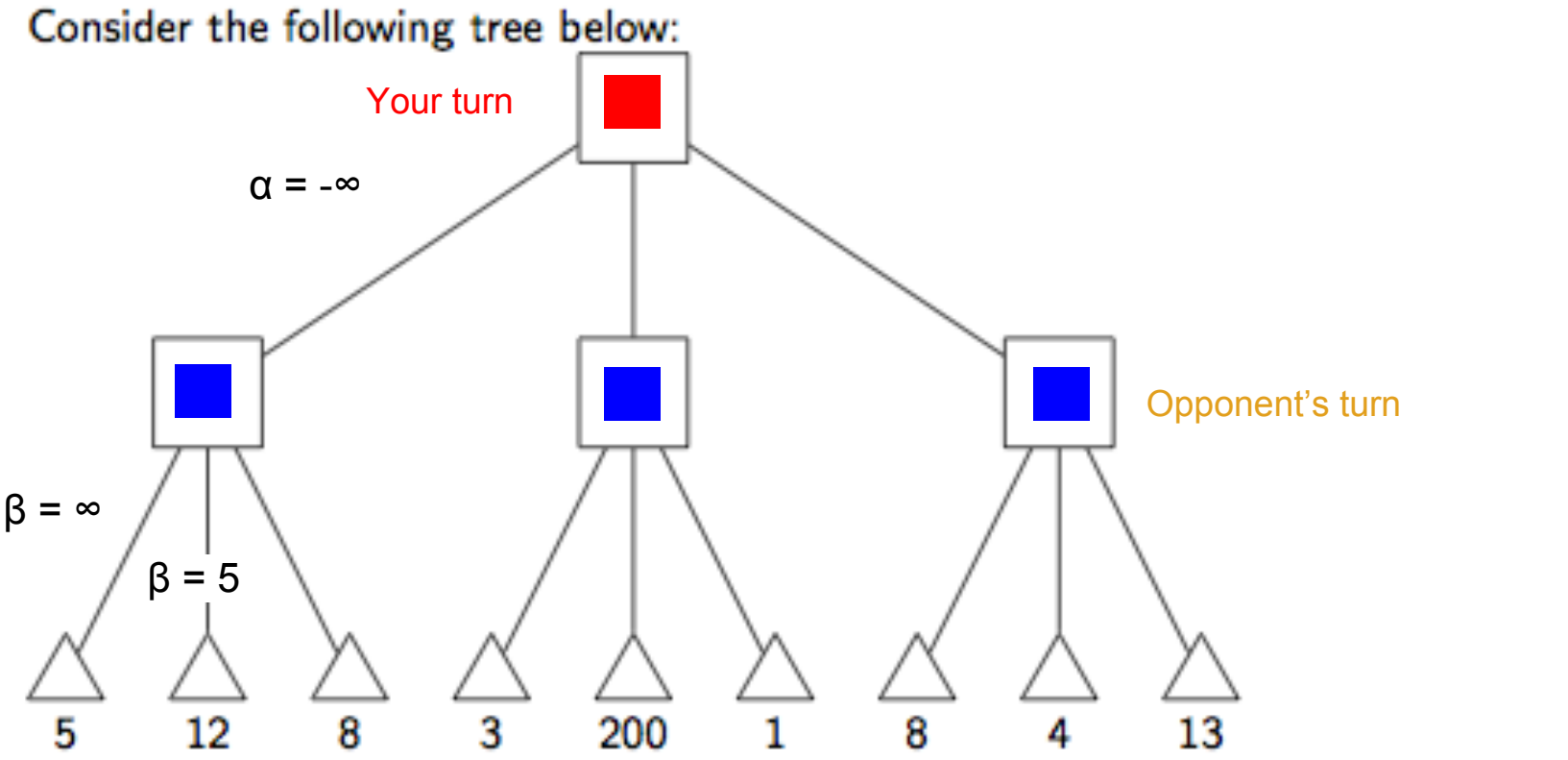
Consider the following tree below:



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

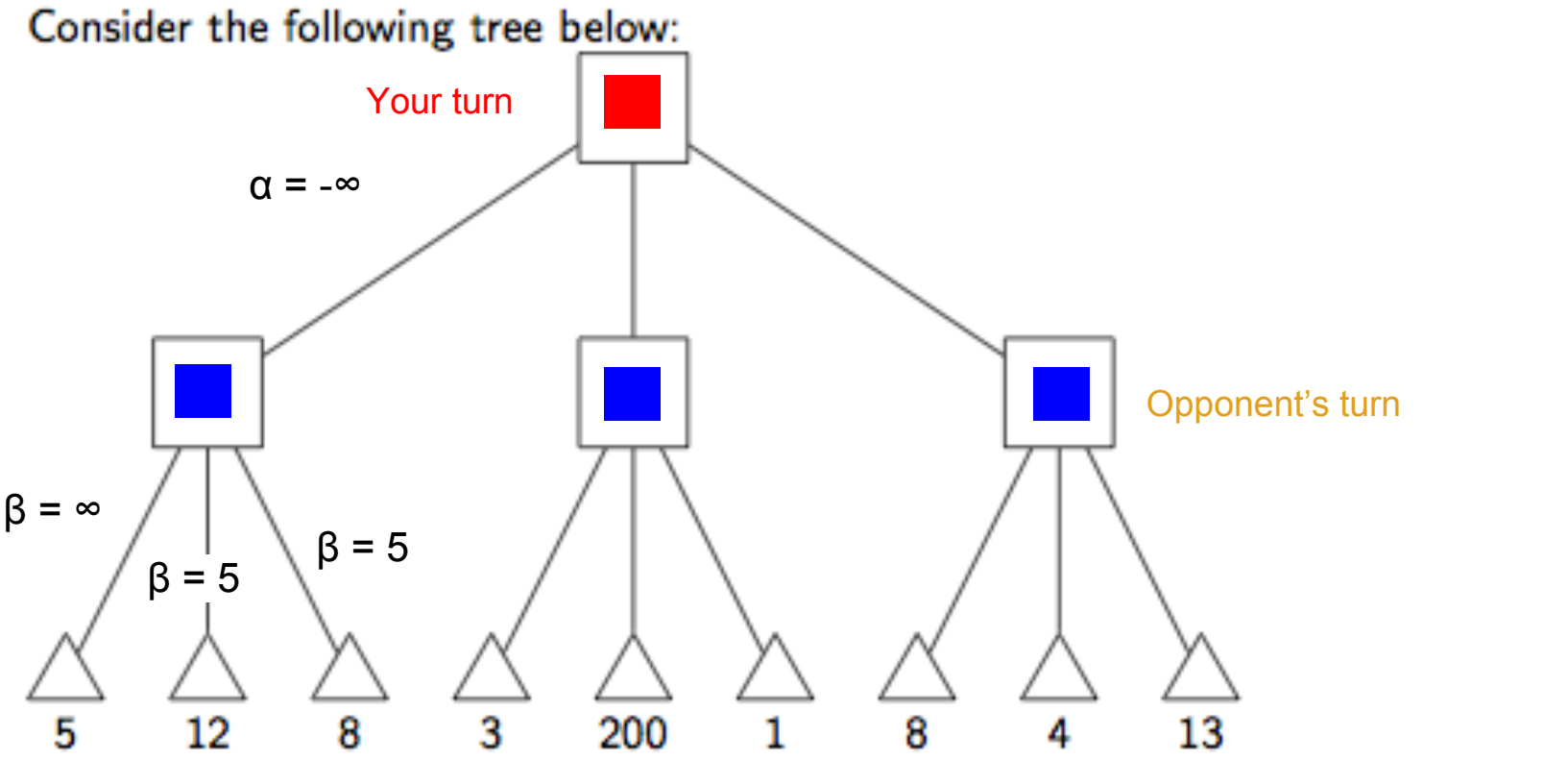
Alpha Beta Pruning



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

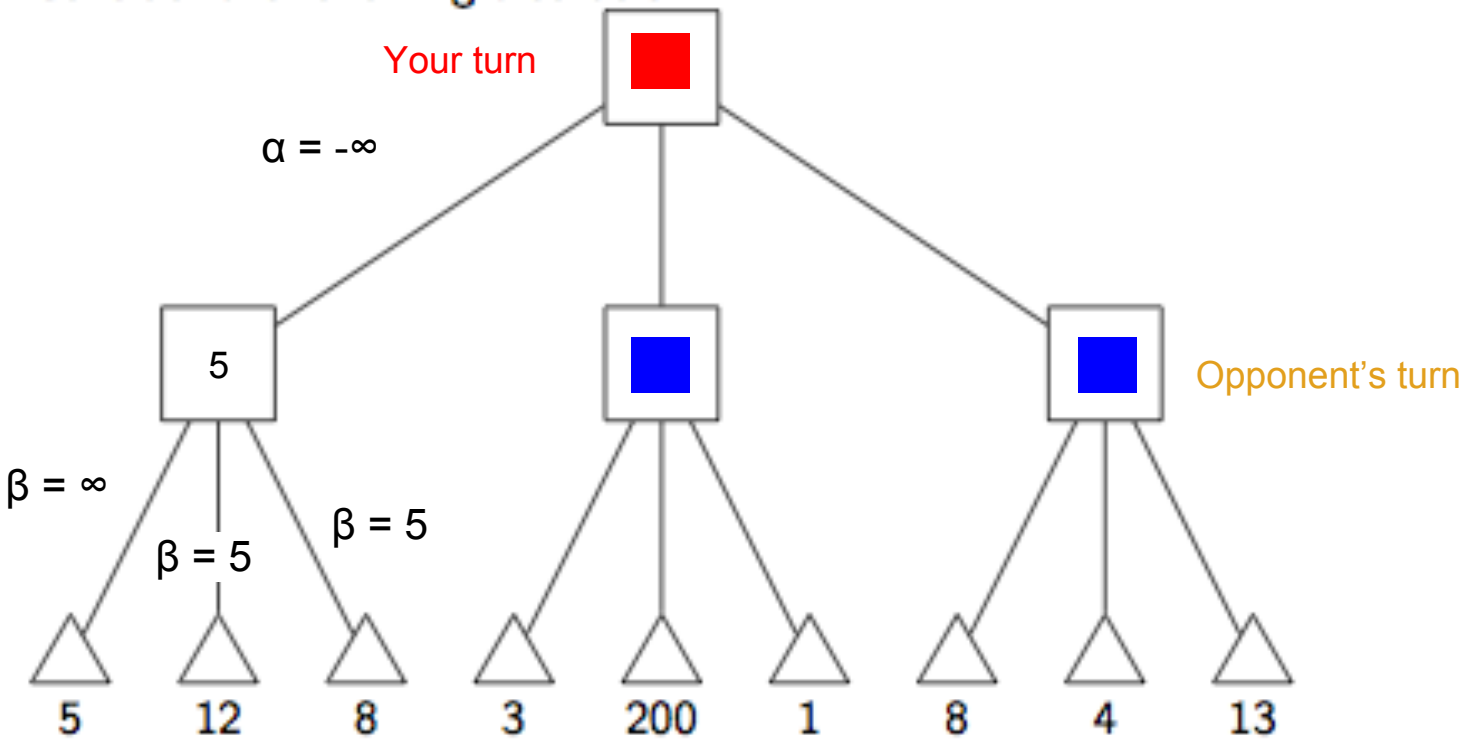
Alpha Beta Pruning



Also assume we traverse children from left to right.

Alpha Beta Pruning

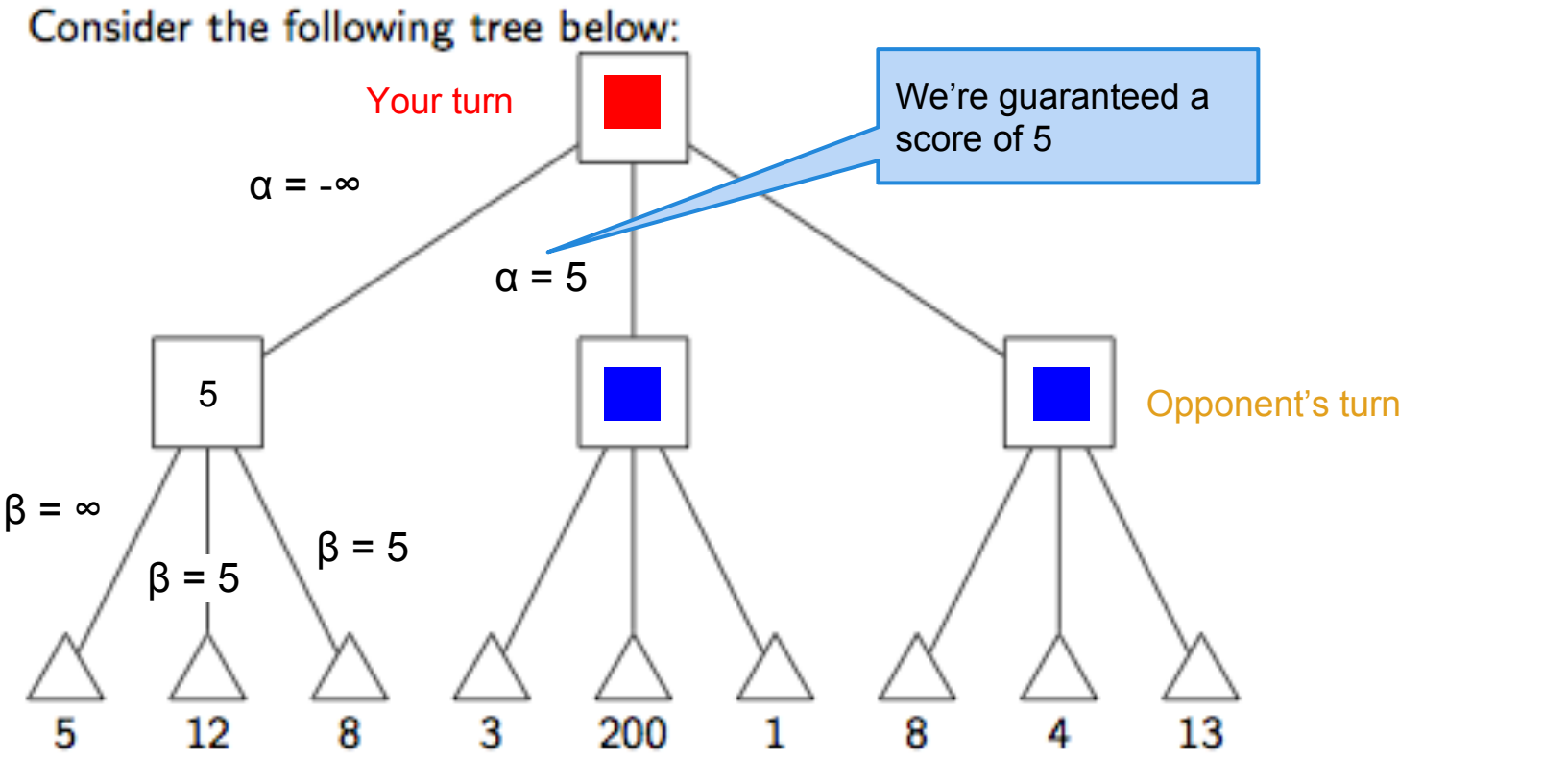
Consider the following tree below:



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

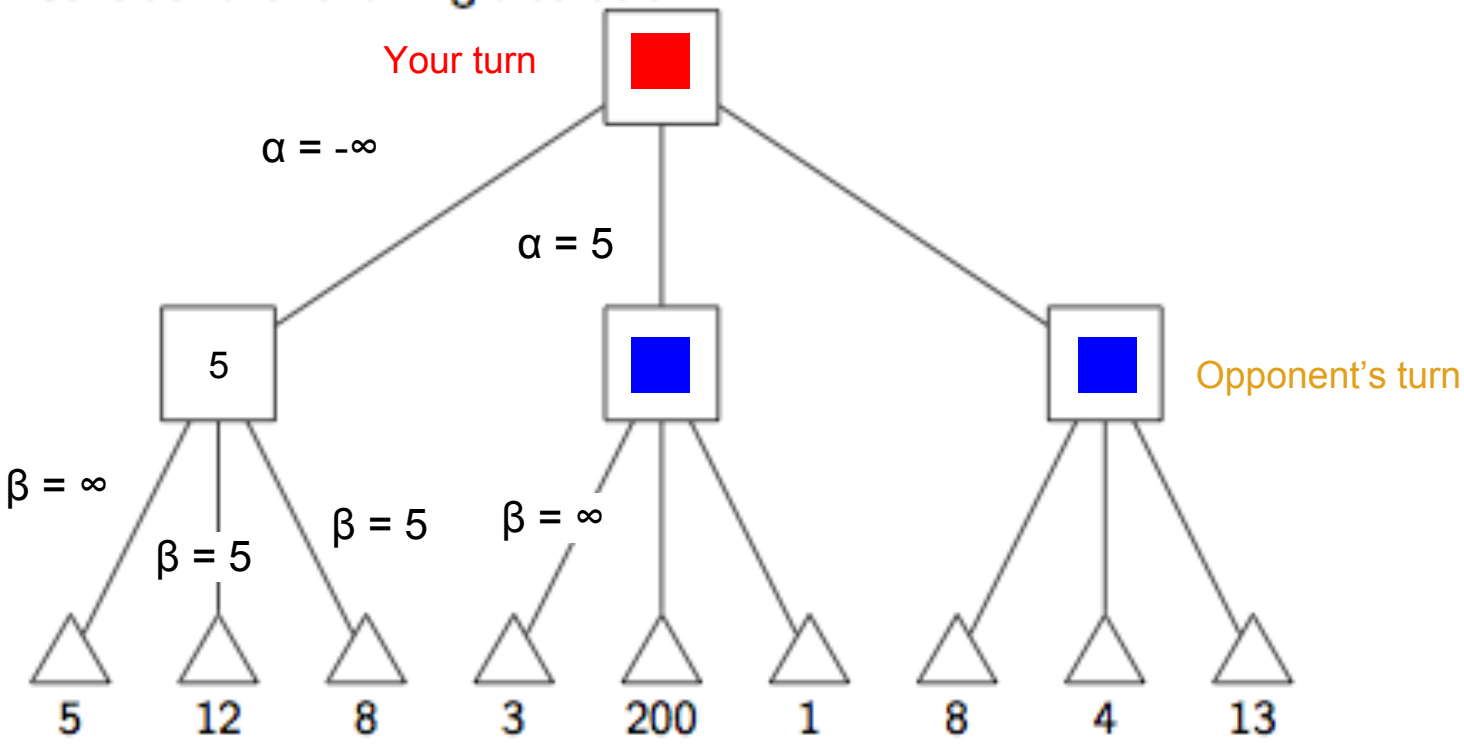


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

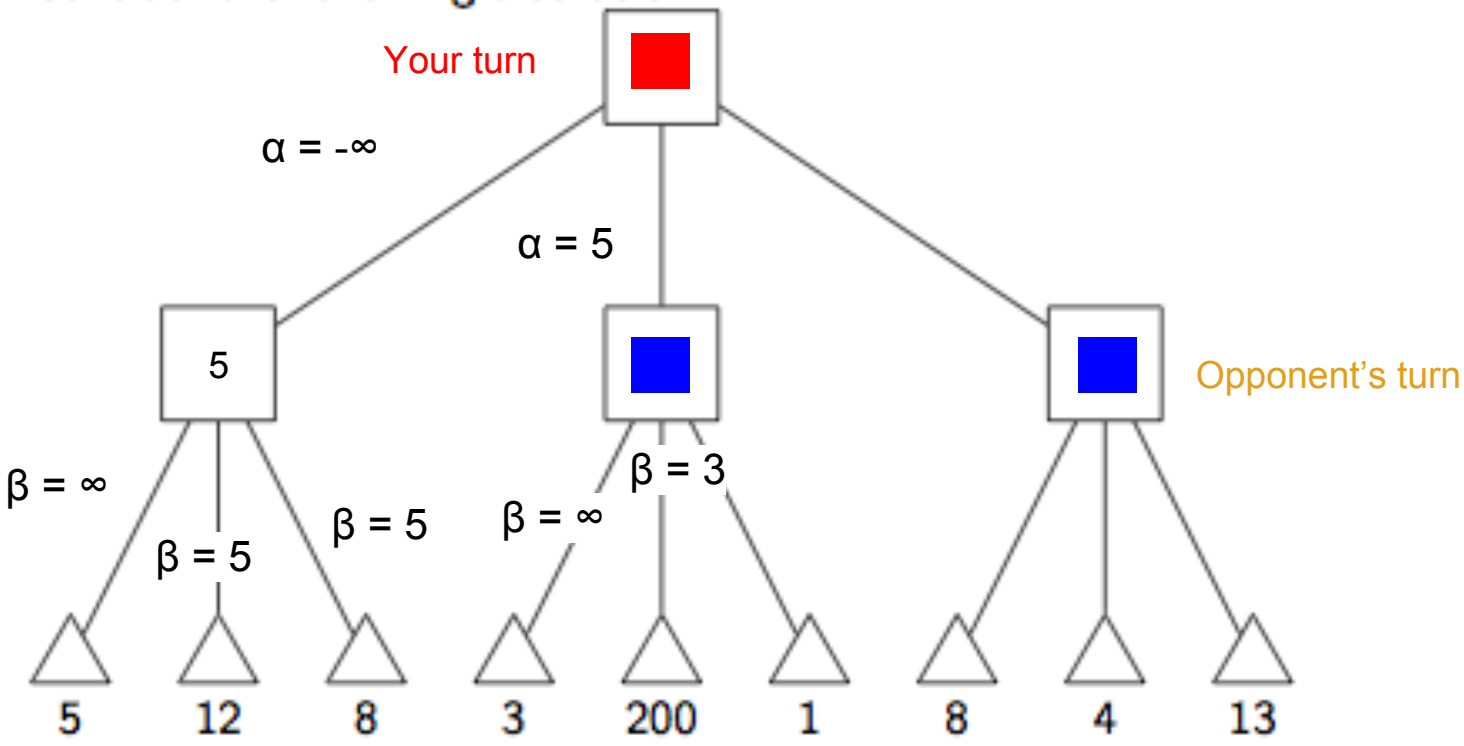


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

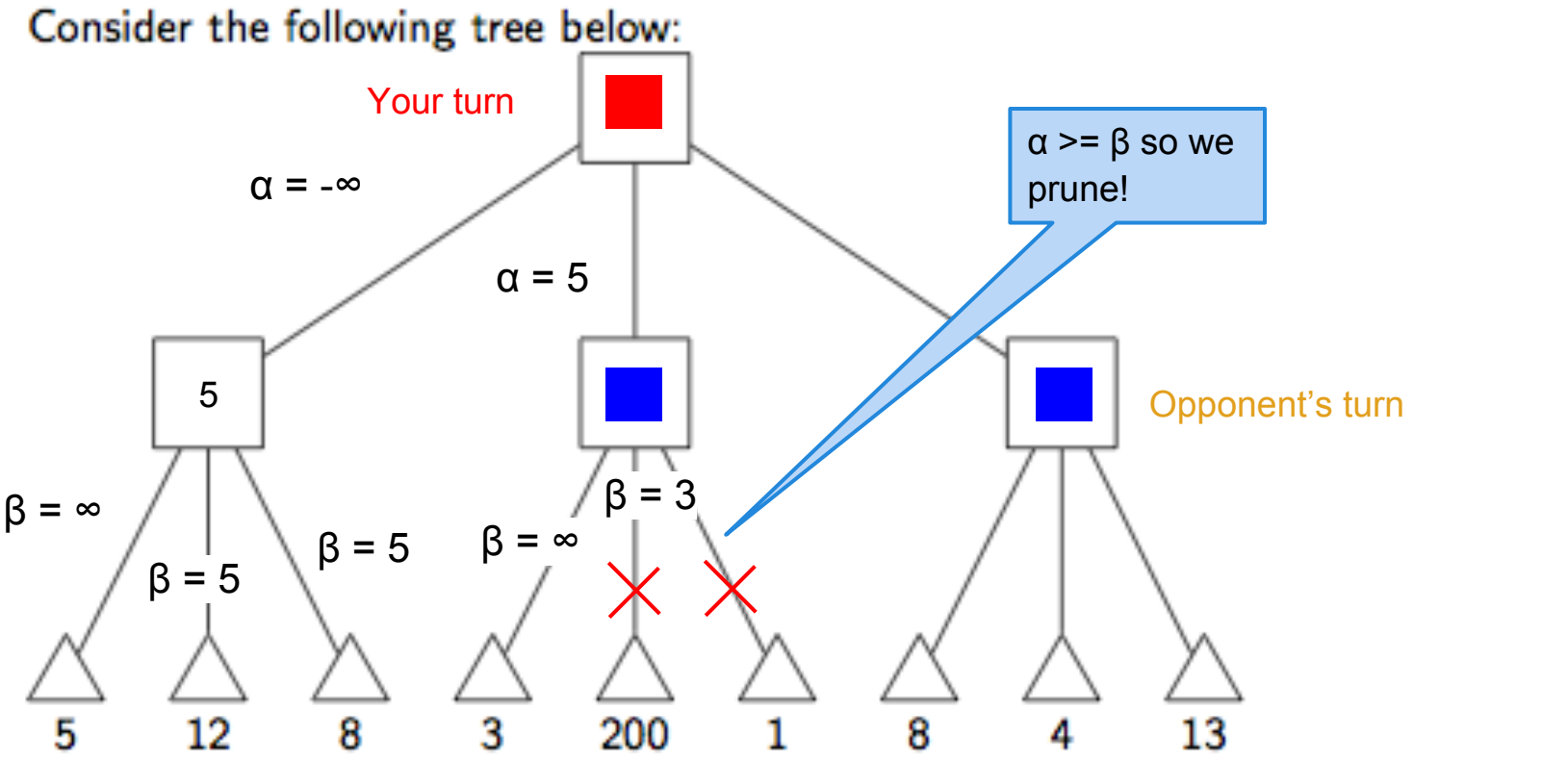
Consider the following tree below:



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

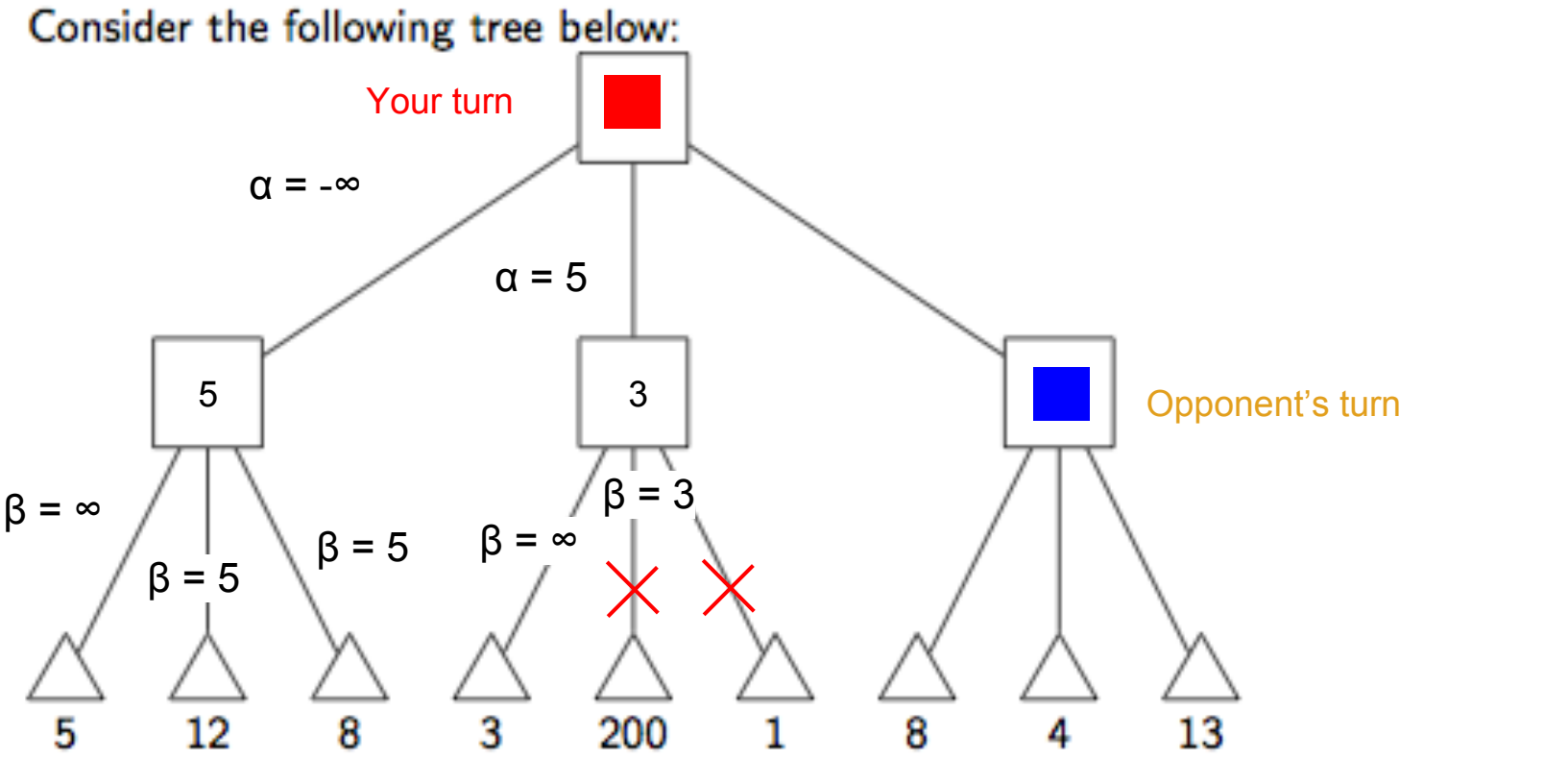
Also assume we traverse children from left to right.

Alpha Beta Pruning



Also assume we traverse children from left to right.

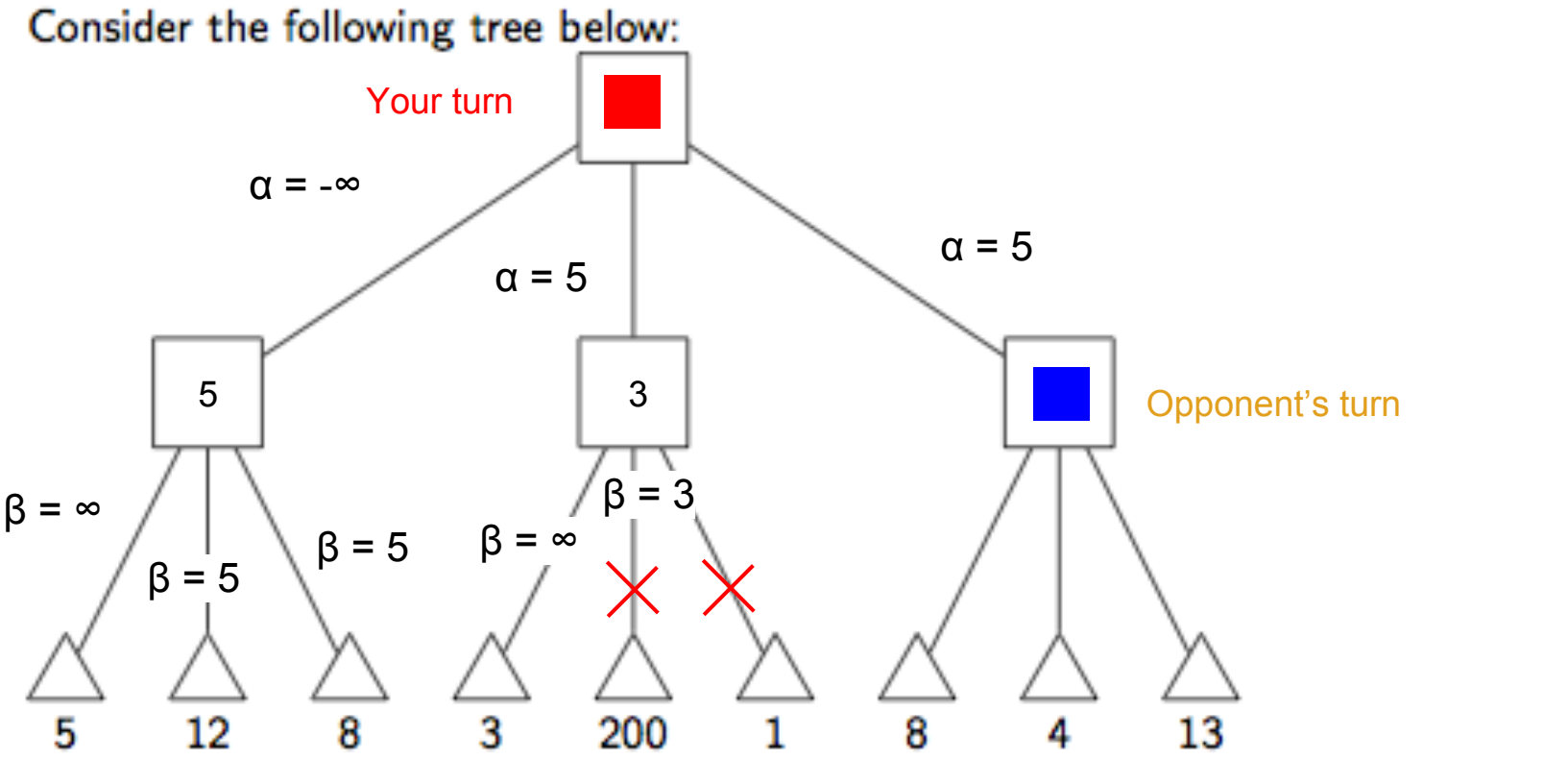
Alpha Beta Pruning



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

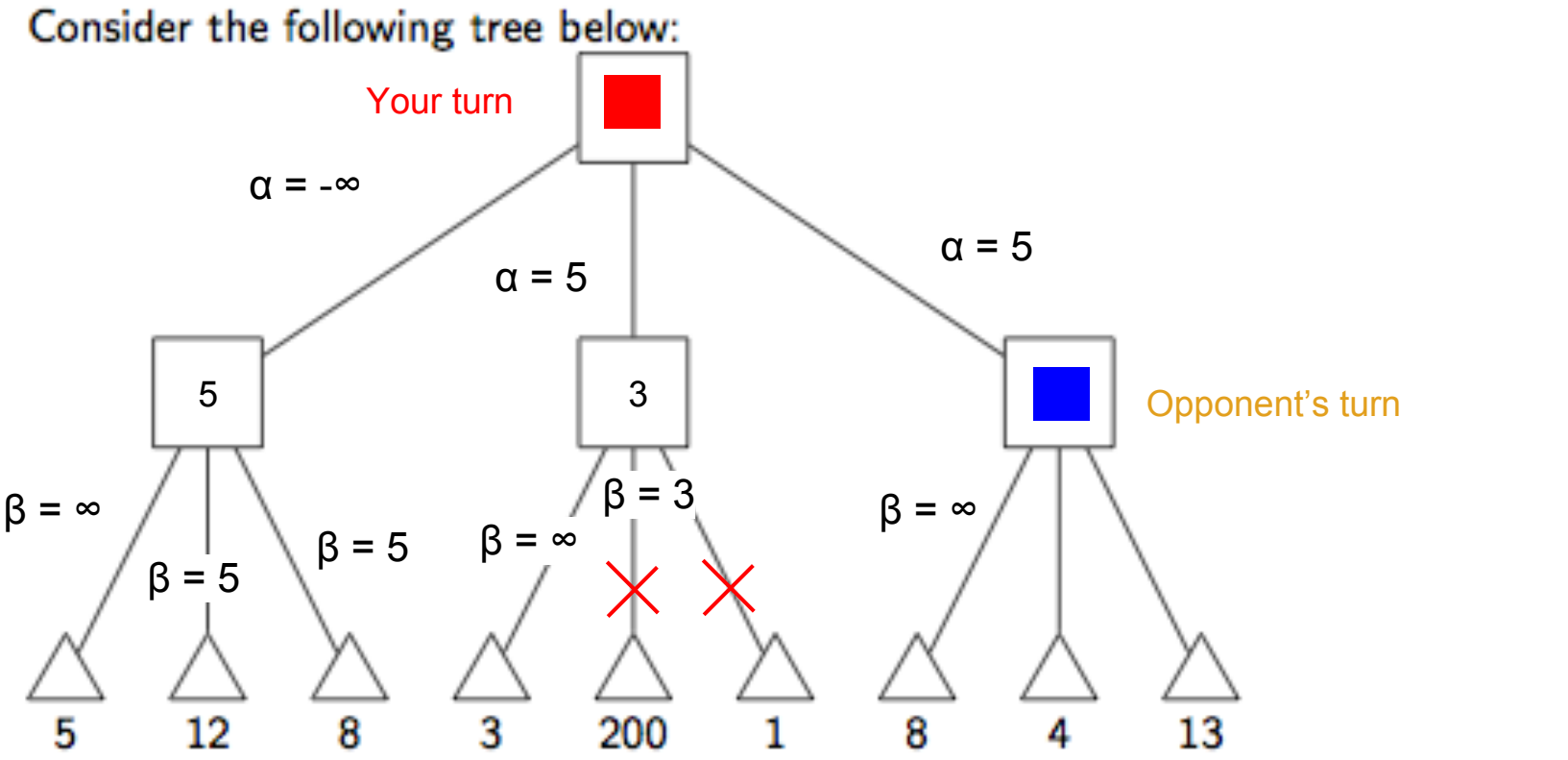
Alpha Beta Pruning



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

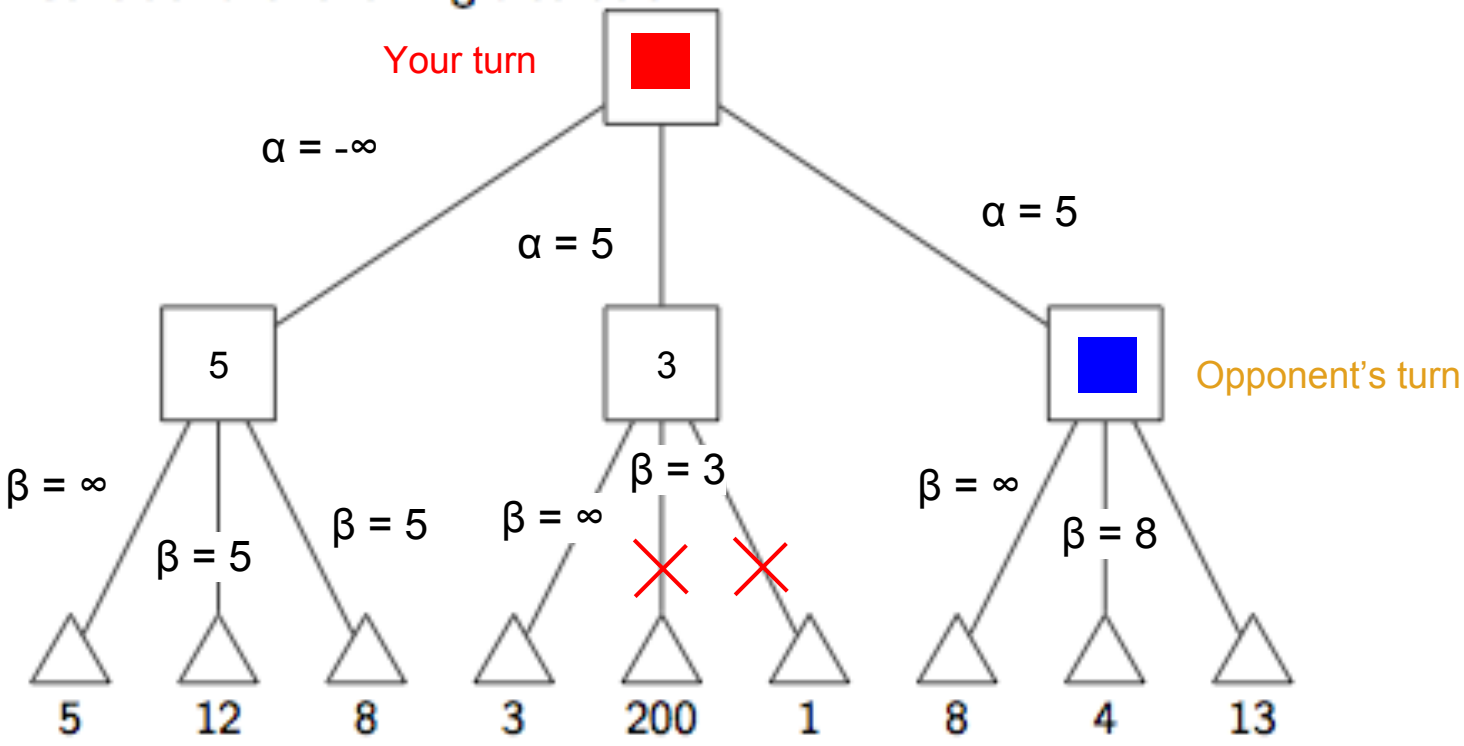


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

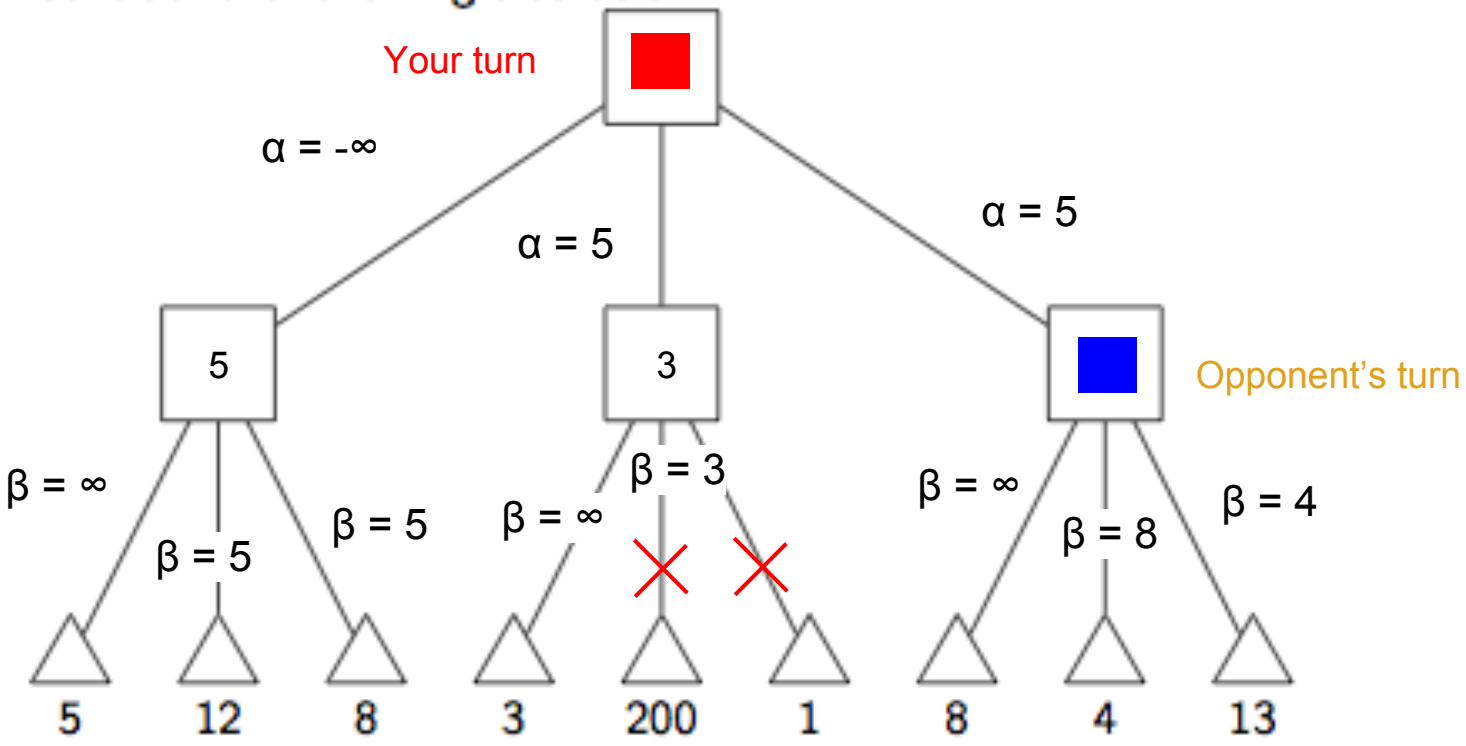


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

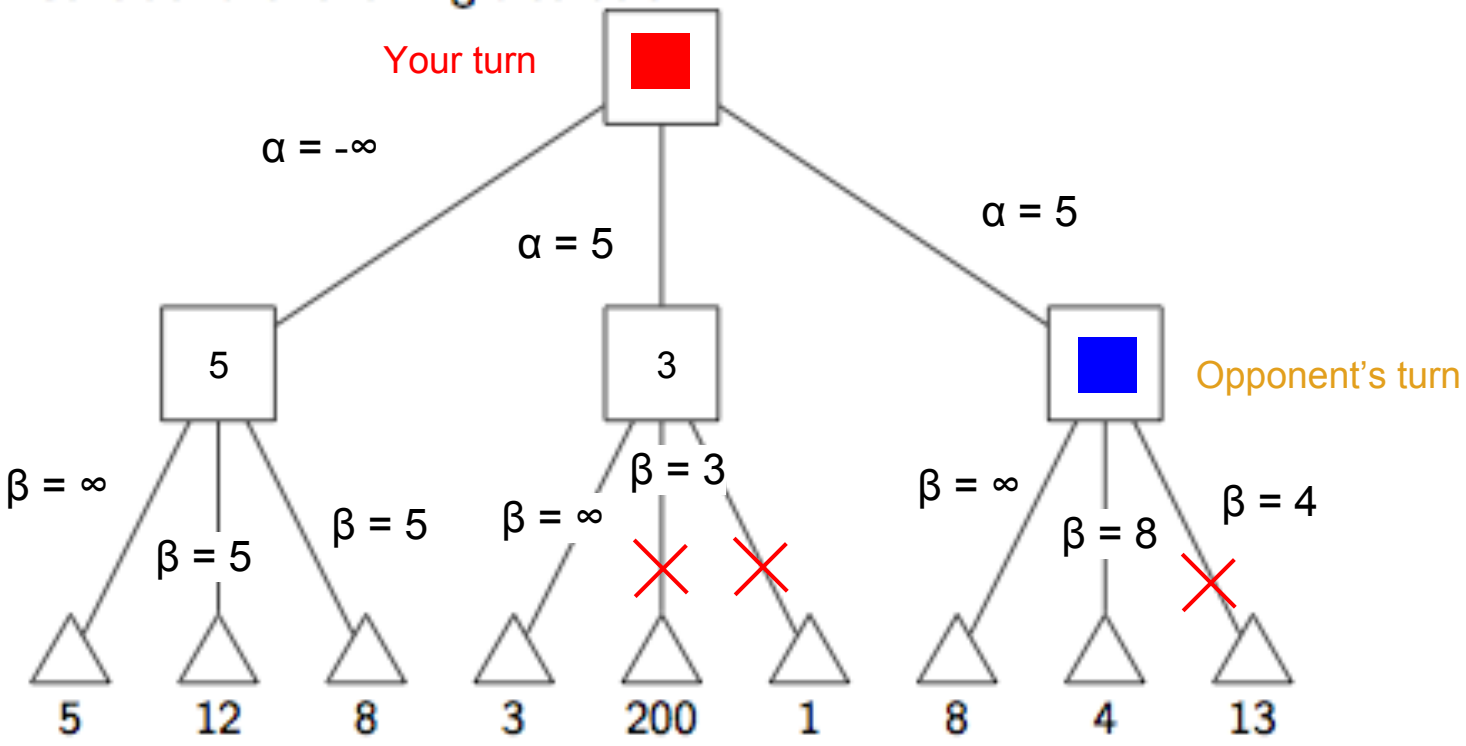


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

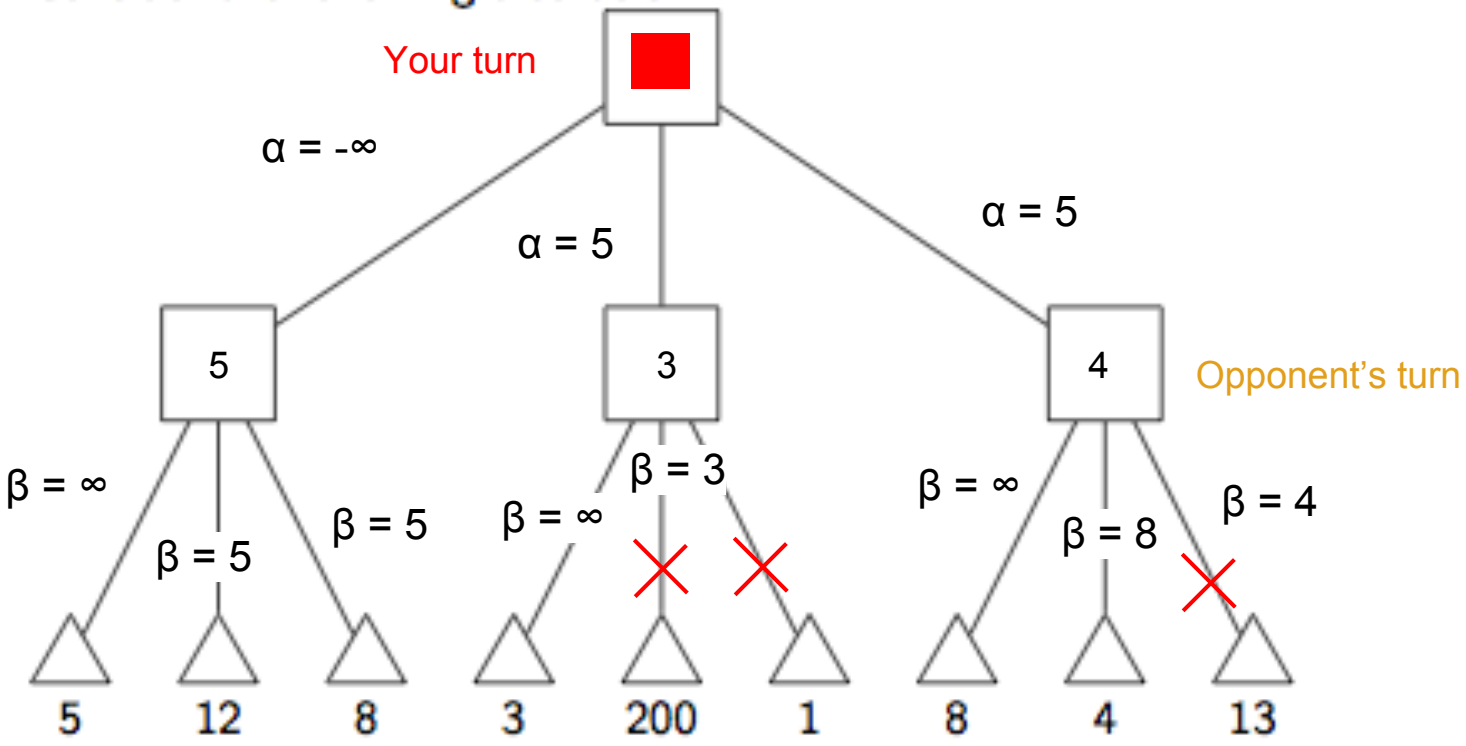


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

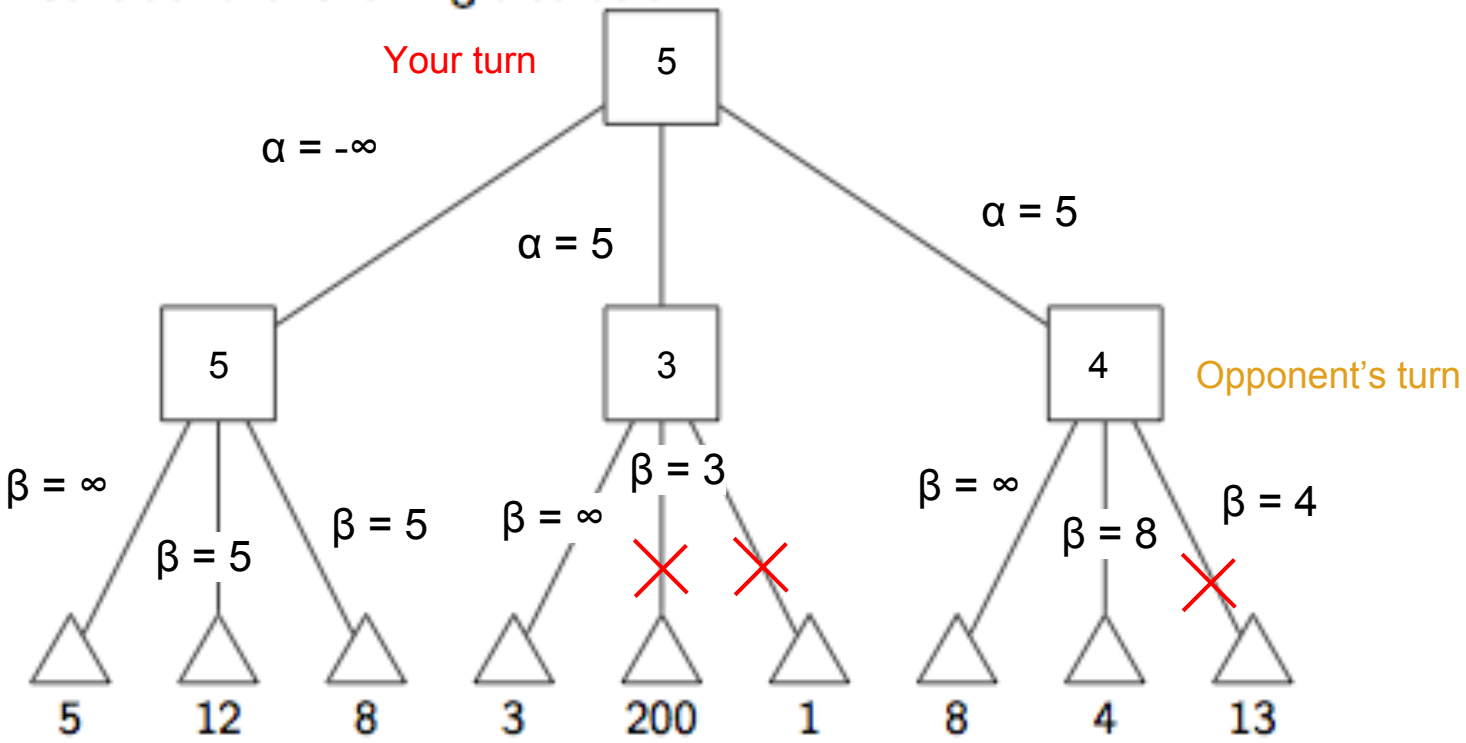


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

Consider the following tree below:

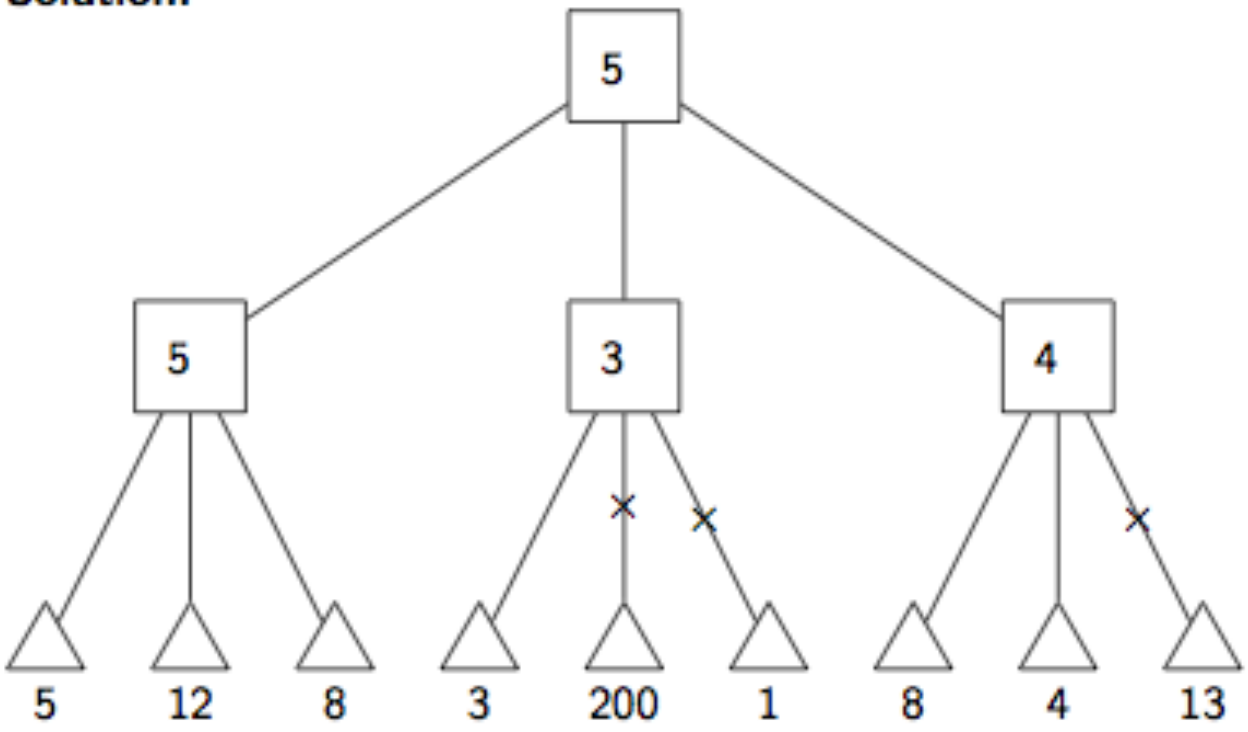


Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

Also assume we traverse children from left to right.

Alpha Beta Pruning

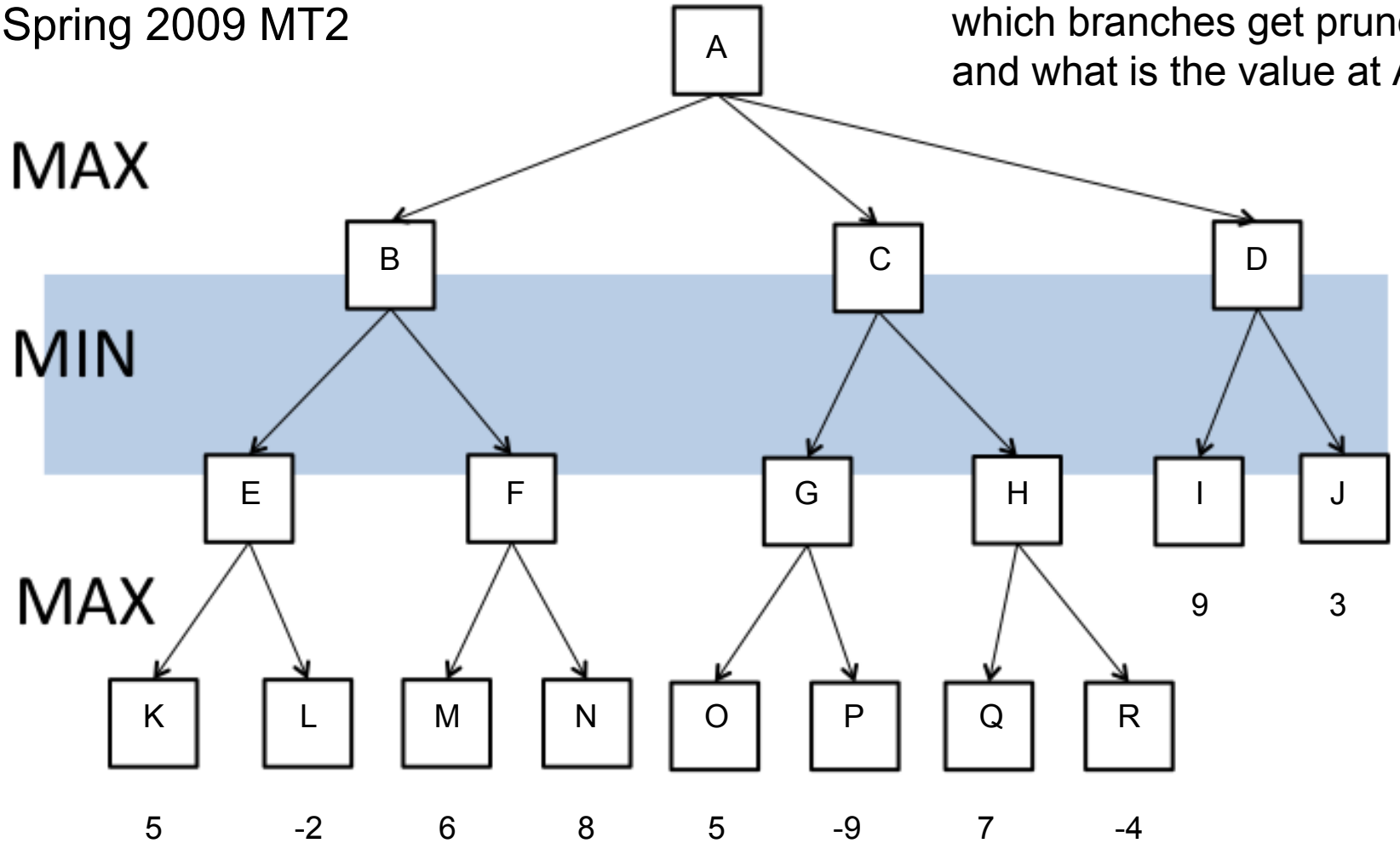
Solution:



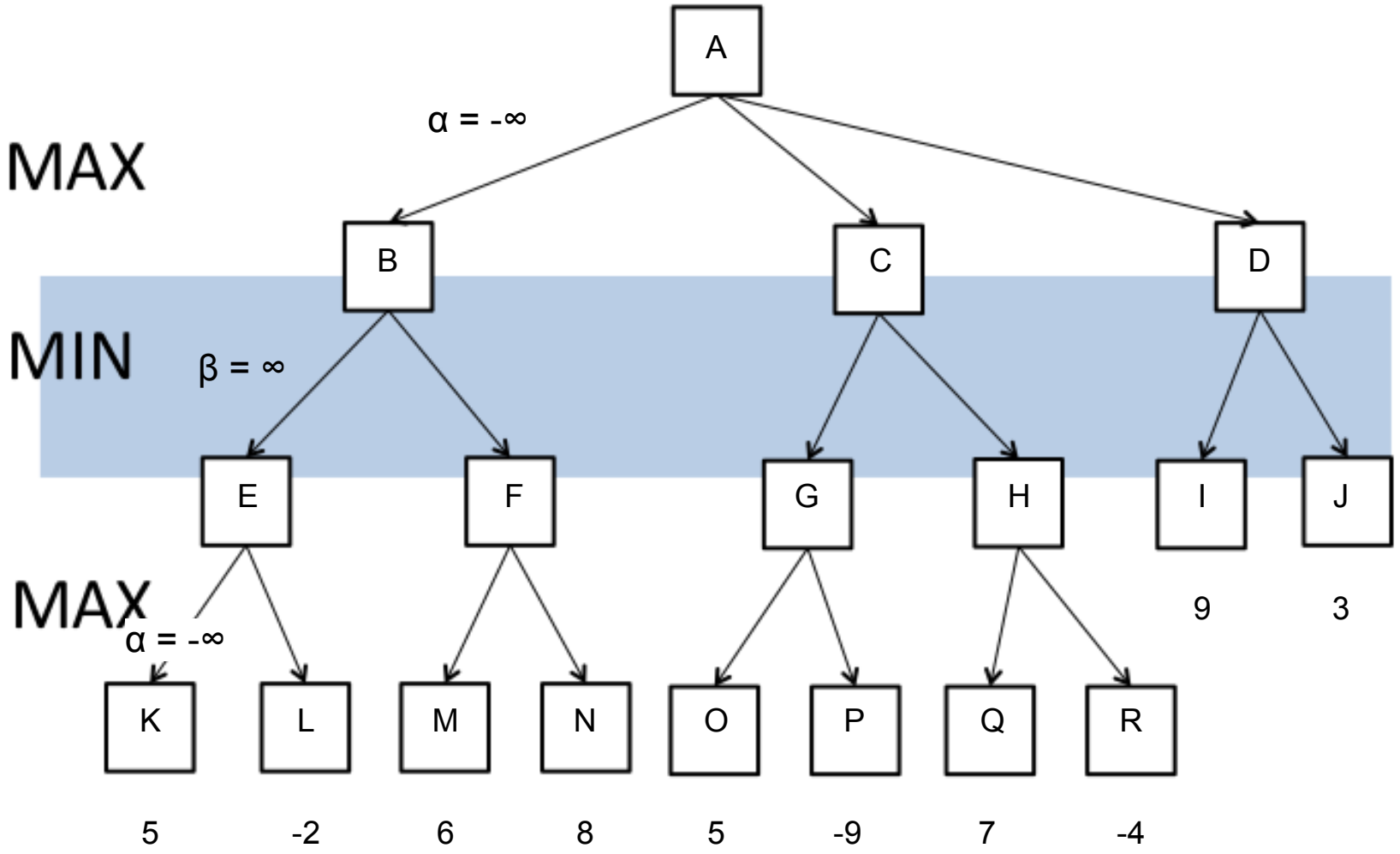
Alpha Beta Pruning

Spring 2009 MT2

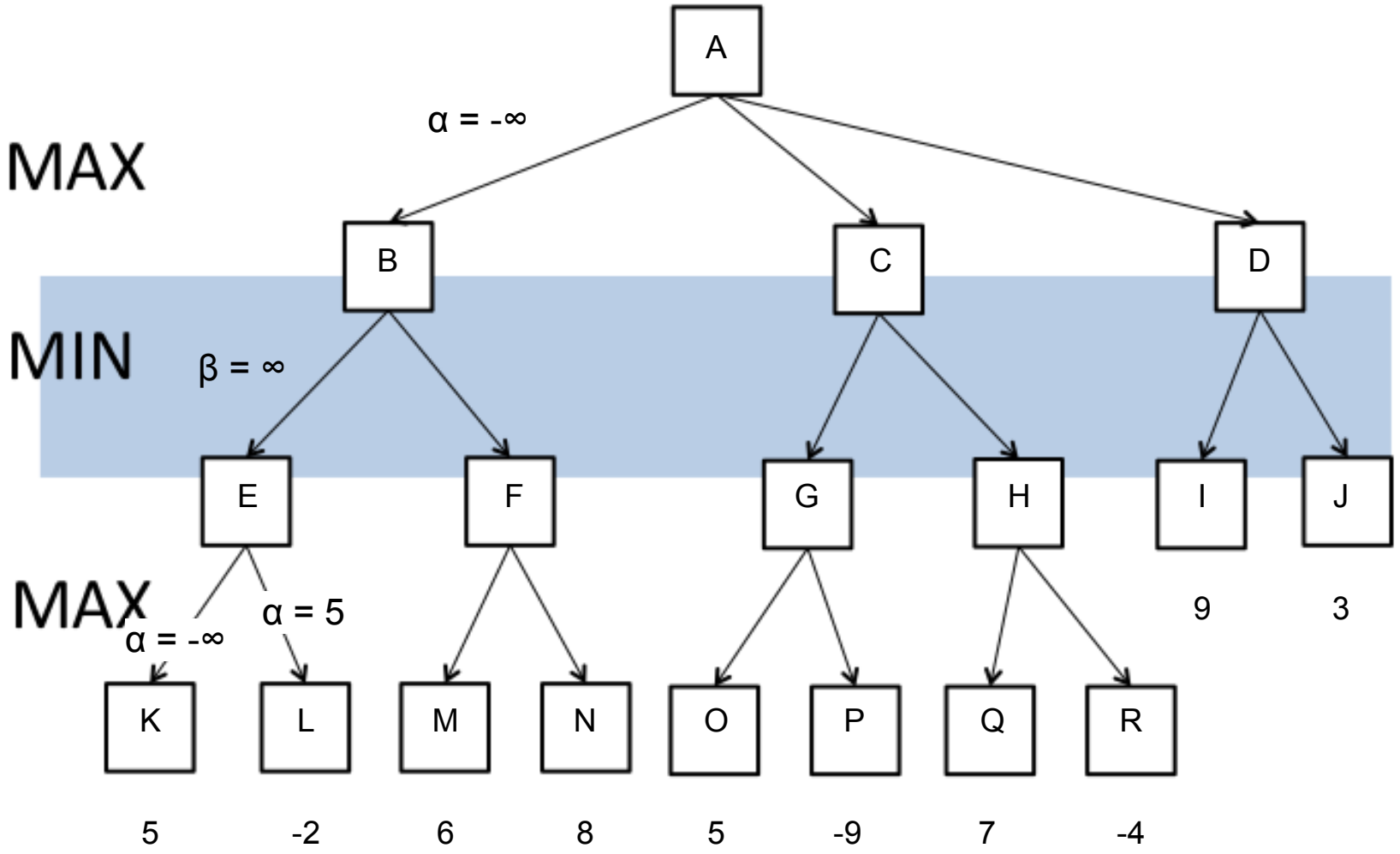
which branches get pruned, and what is the value at A?



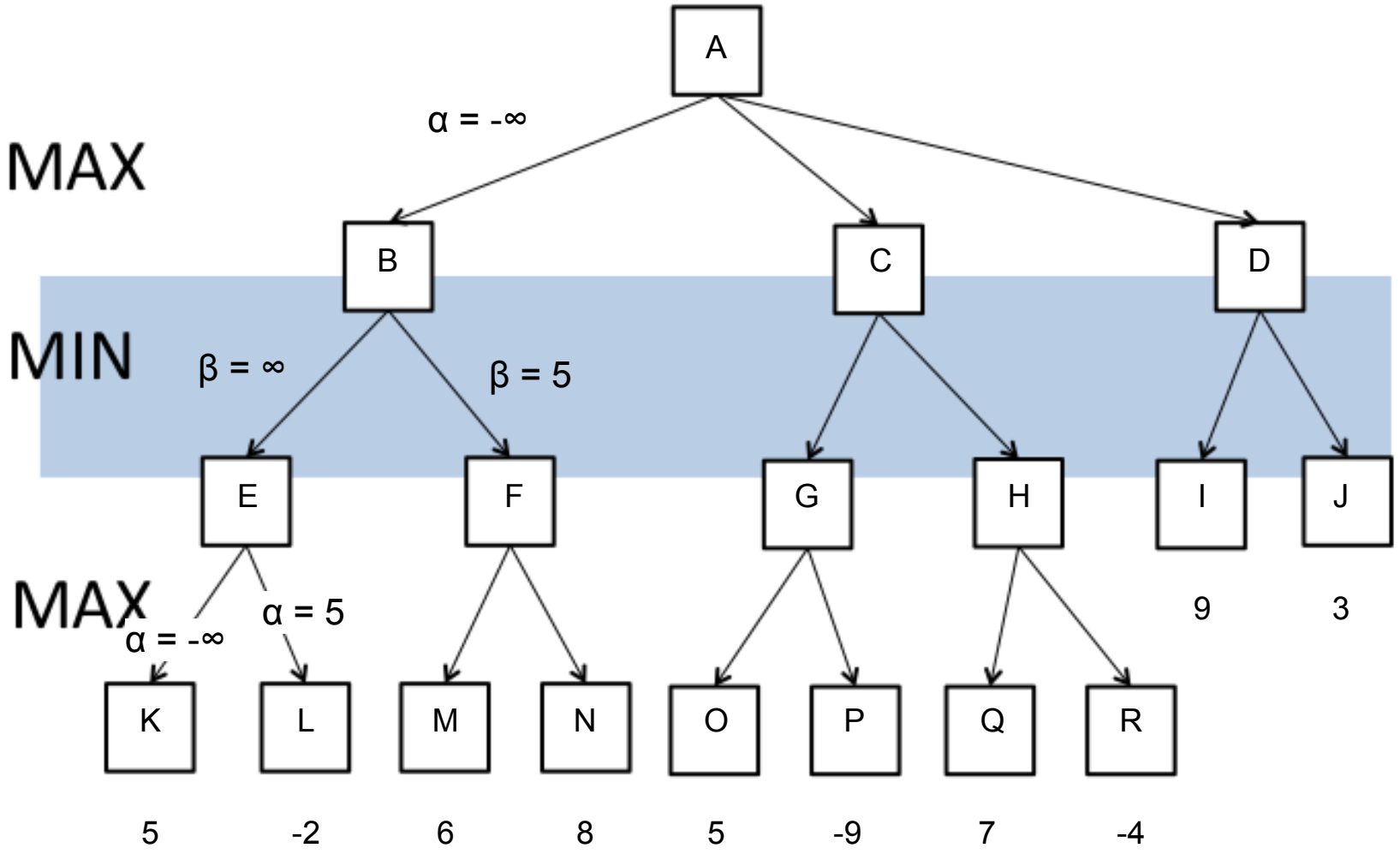
Alpha Beta Pruning



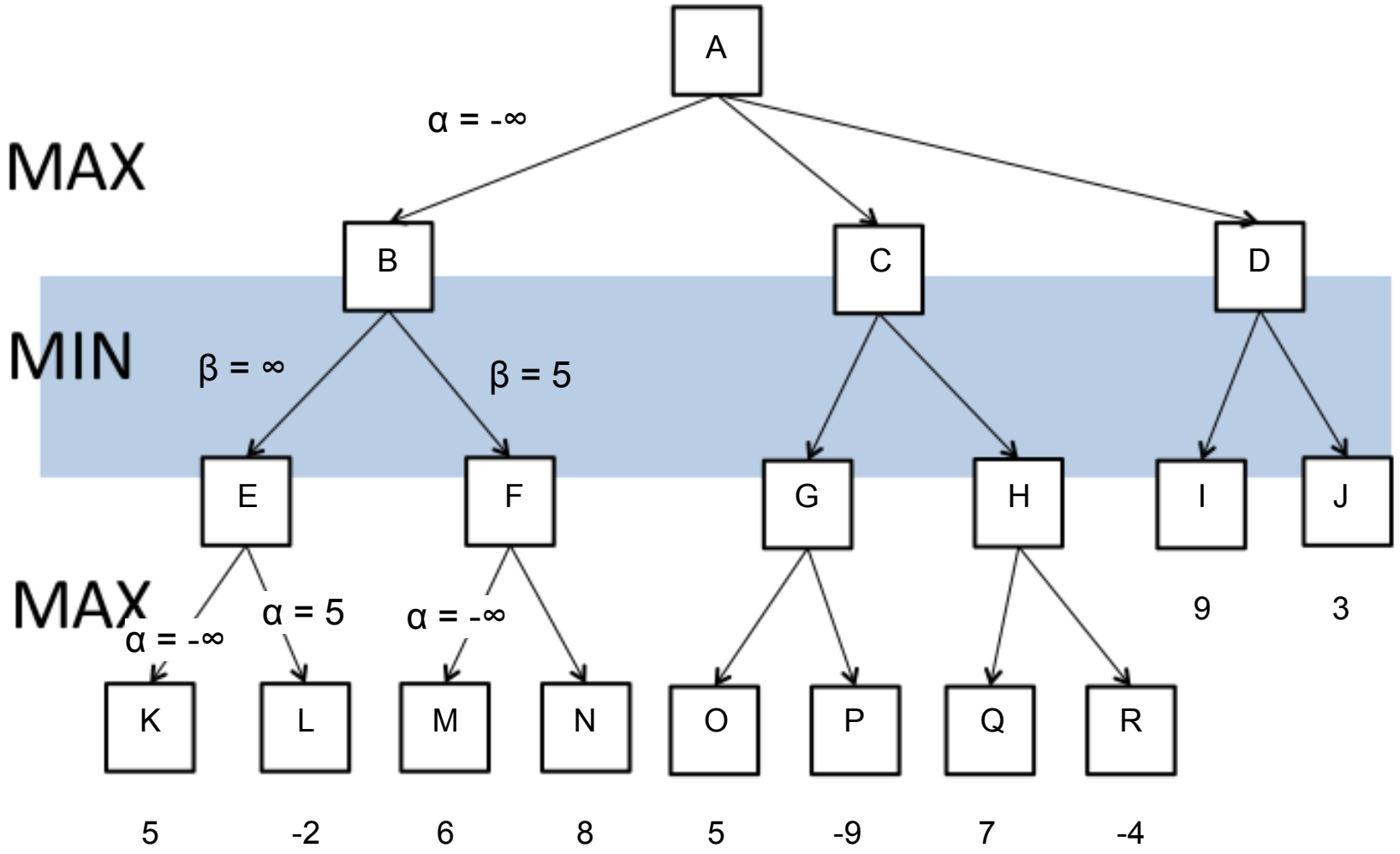
Alpha Beta Pruning



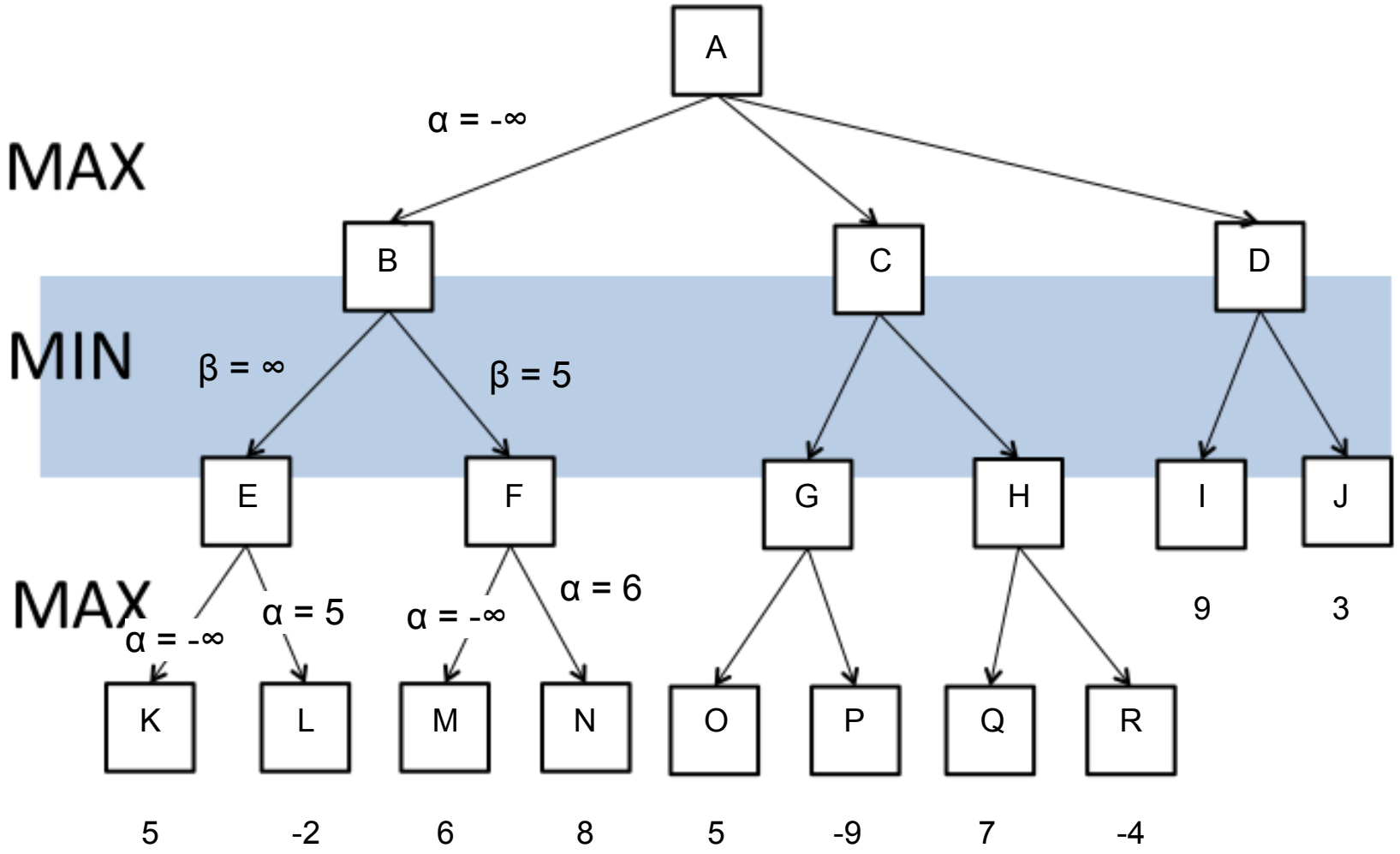
Alpha Beta Pruning



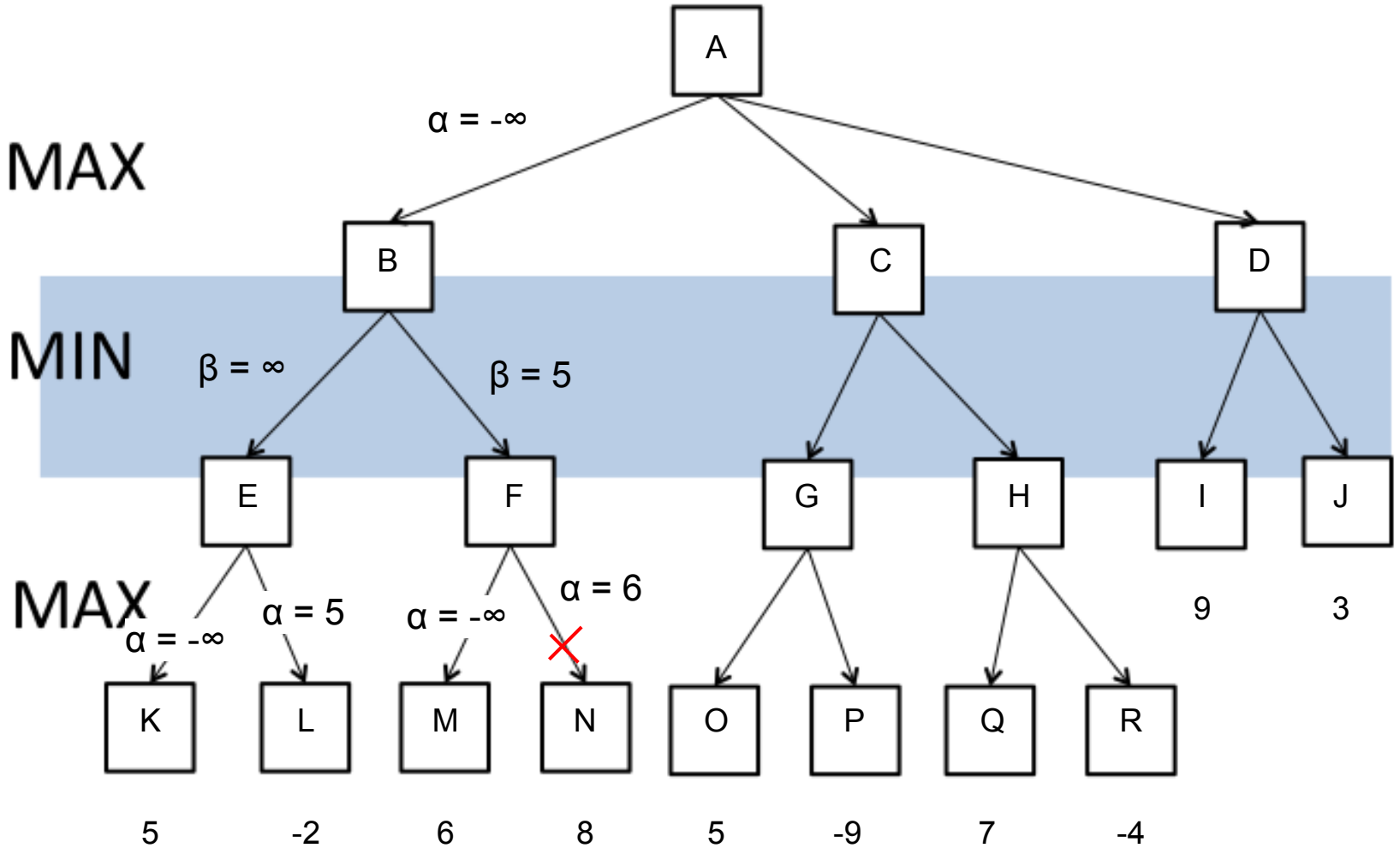
Alpha Beta Pruning



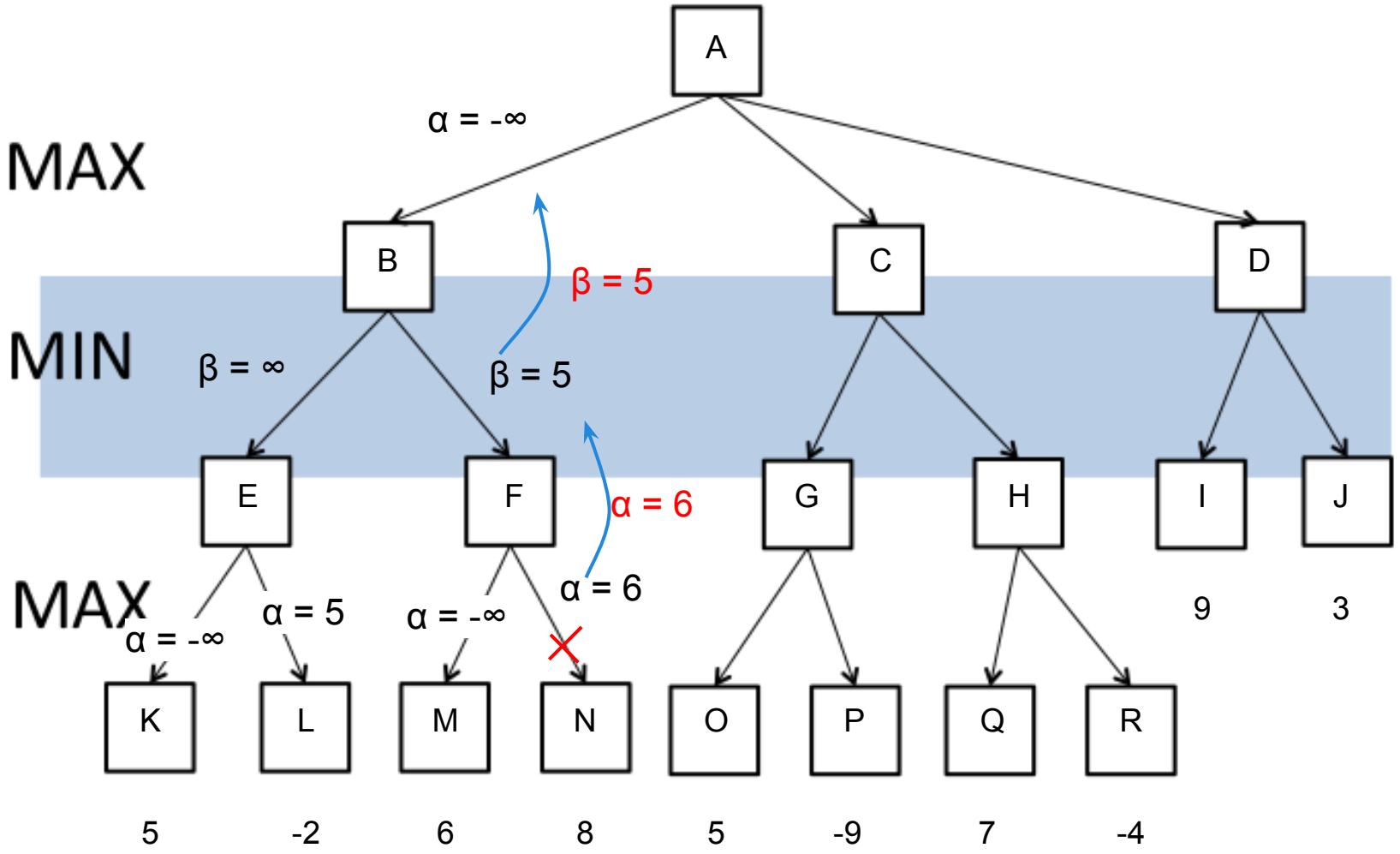
Alpha Beta Pruning



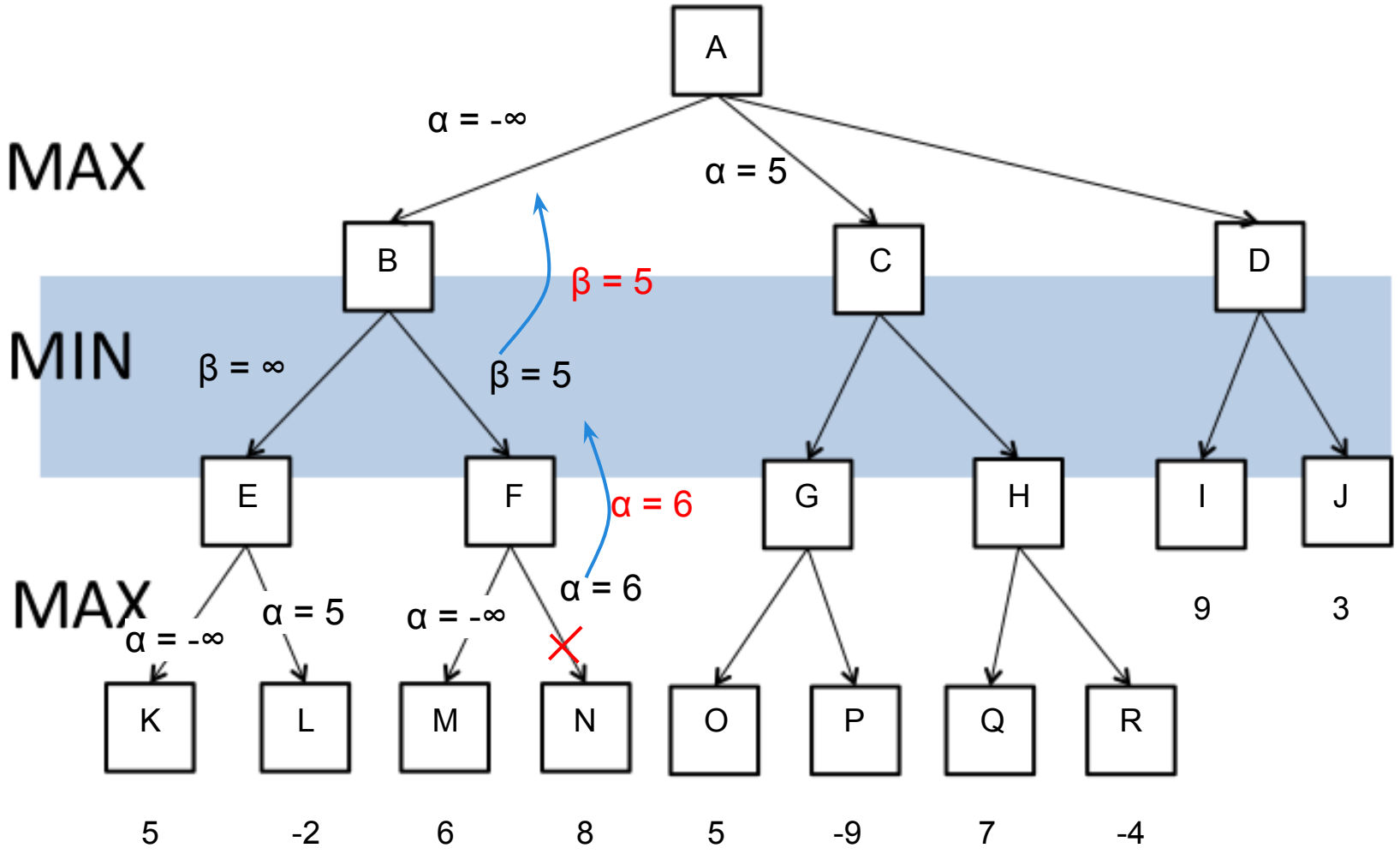
Alpha Beta Pruning



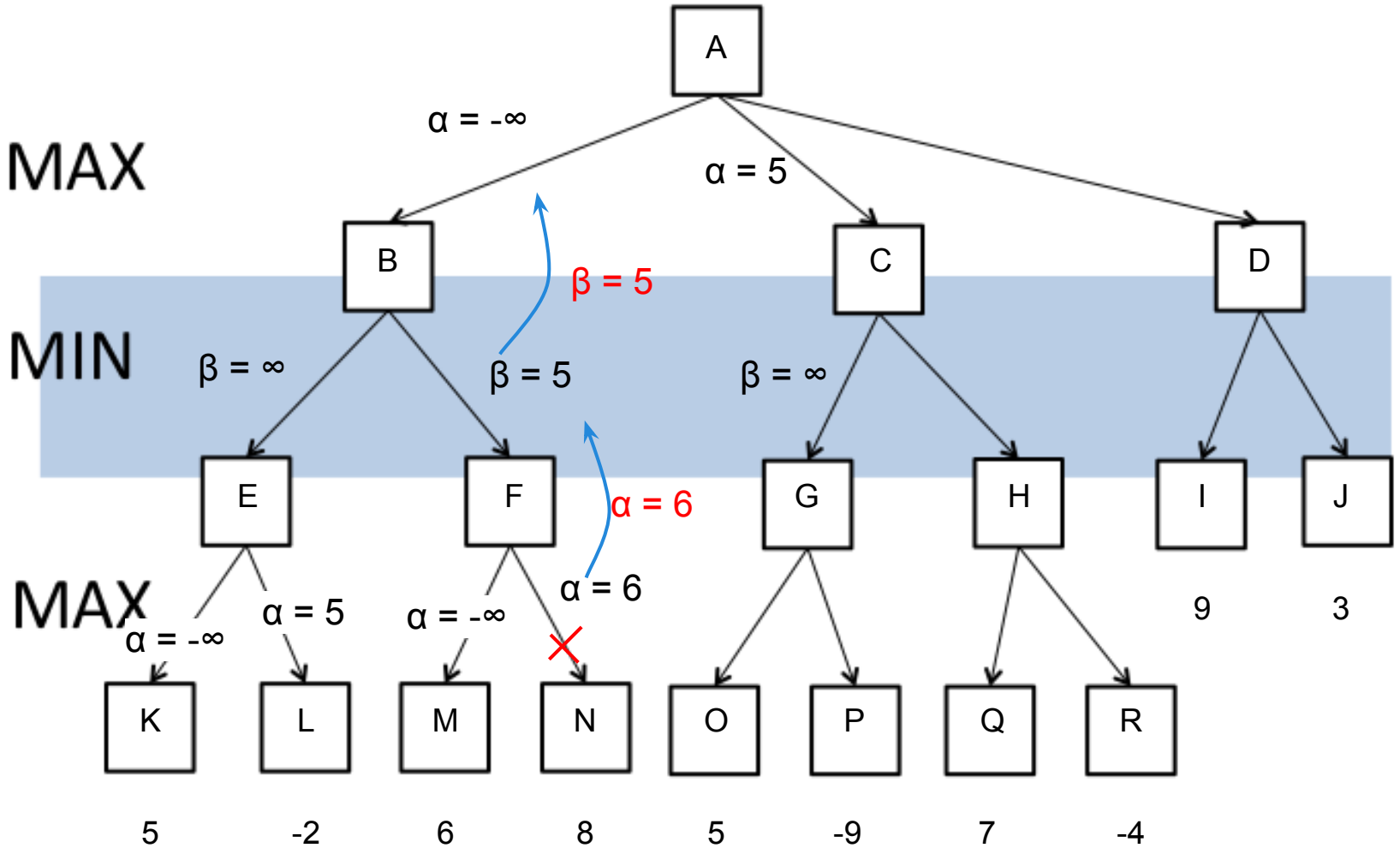
Alpha Beta Pruning



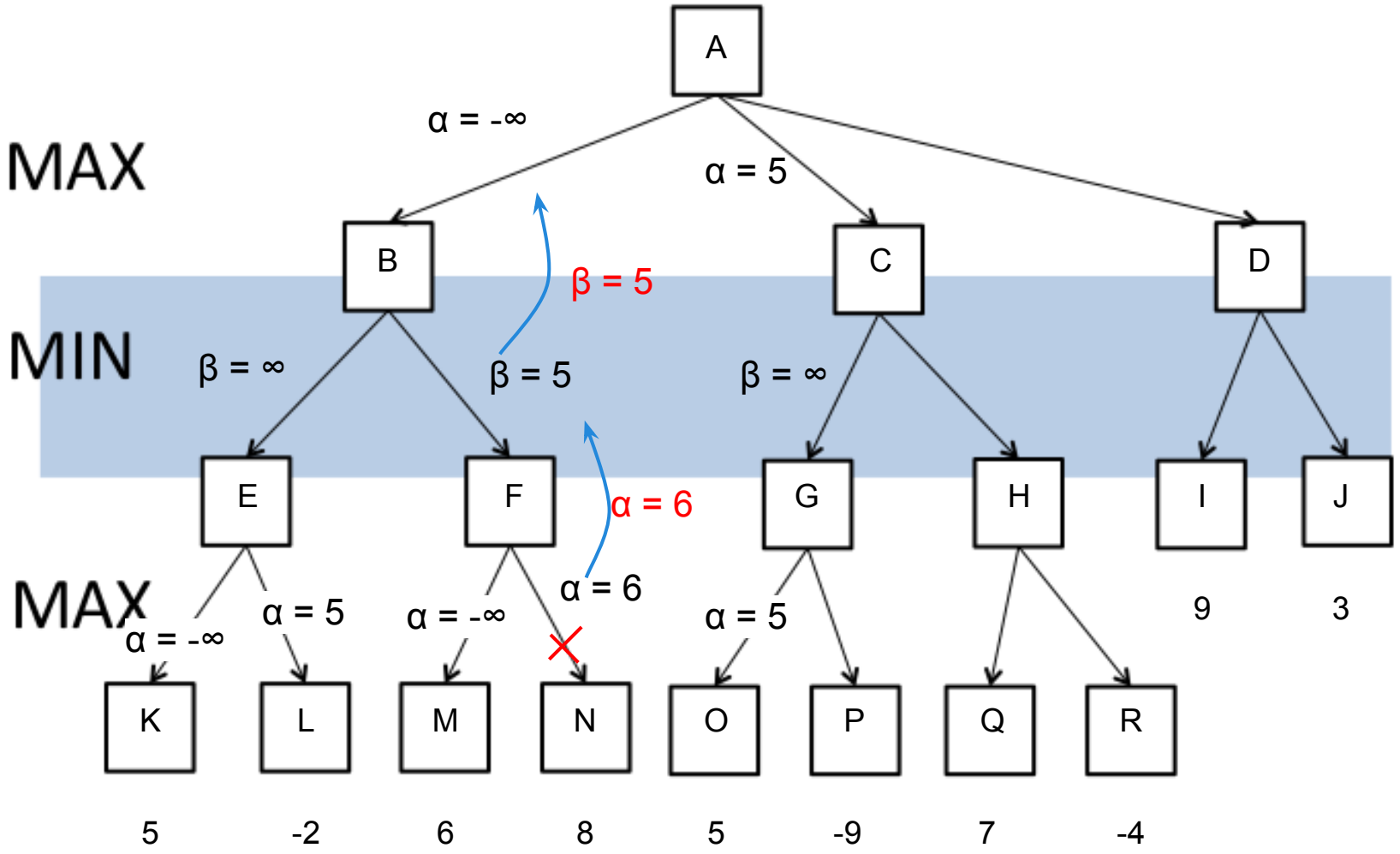
Alpha Beta Pruning



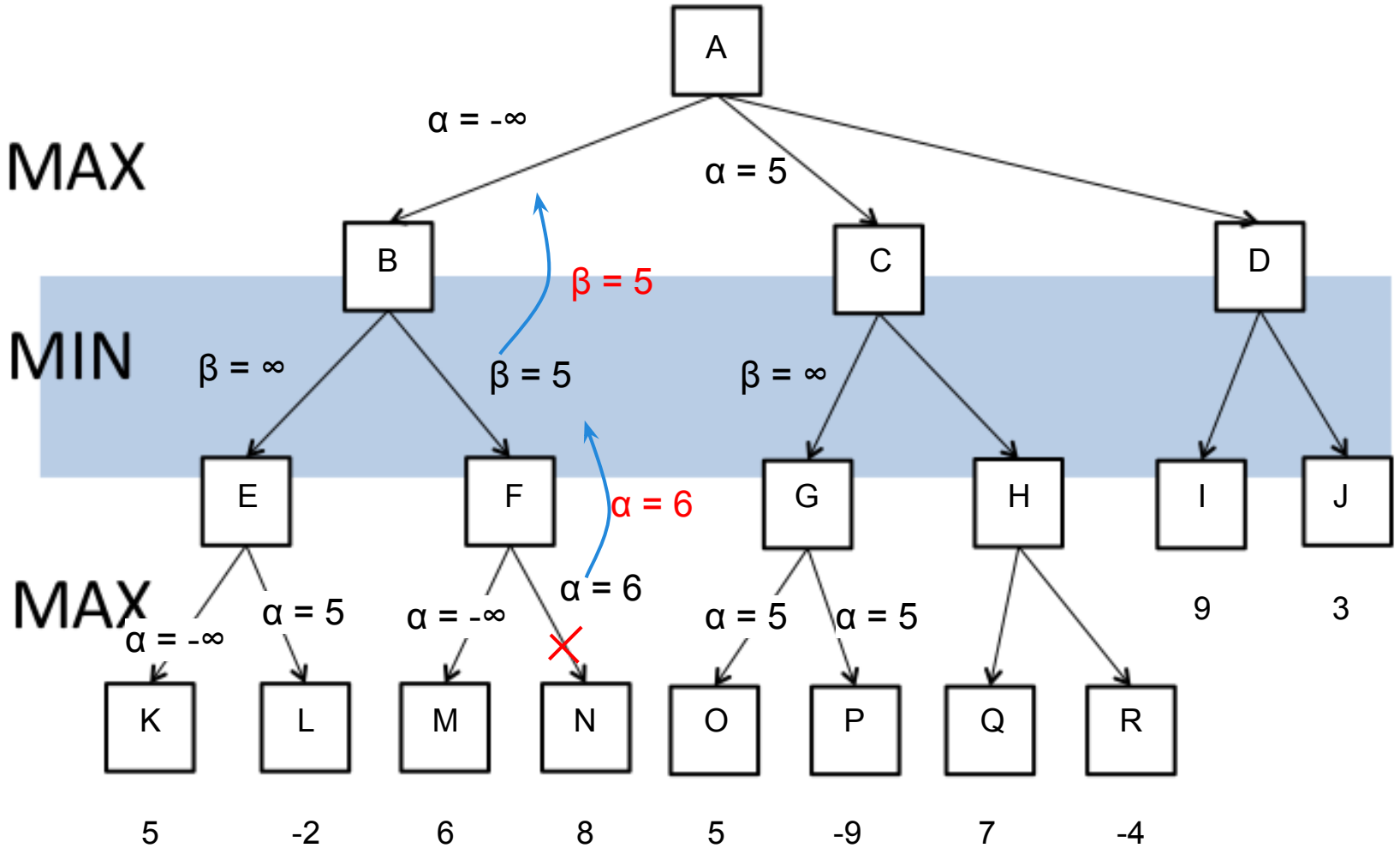
Alpha Beta Pruning



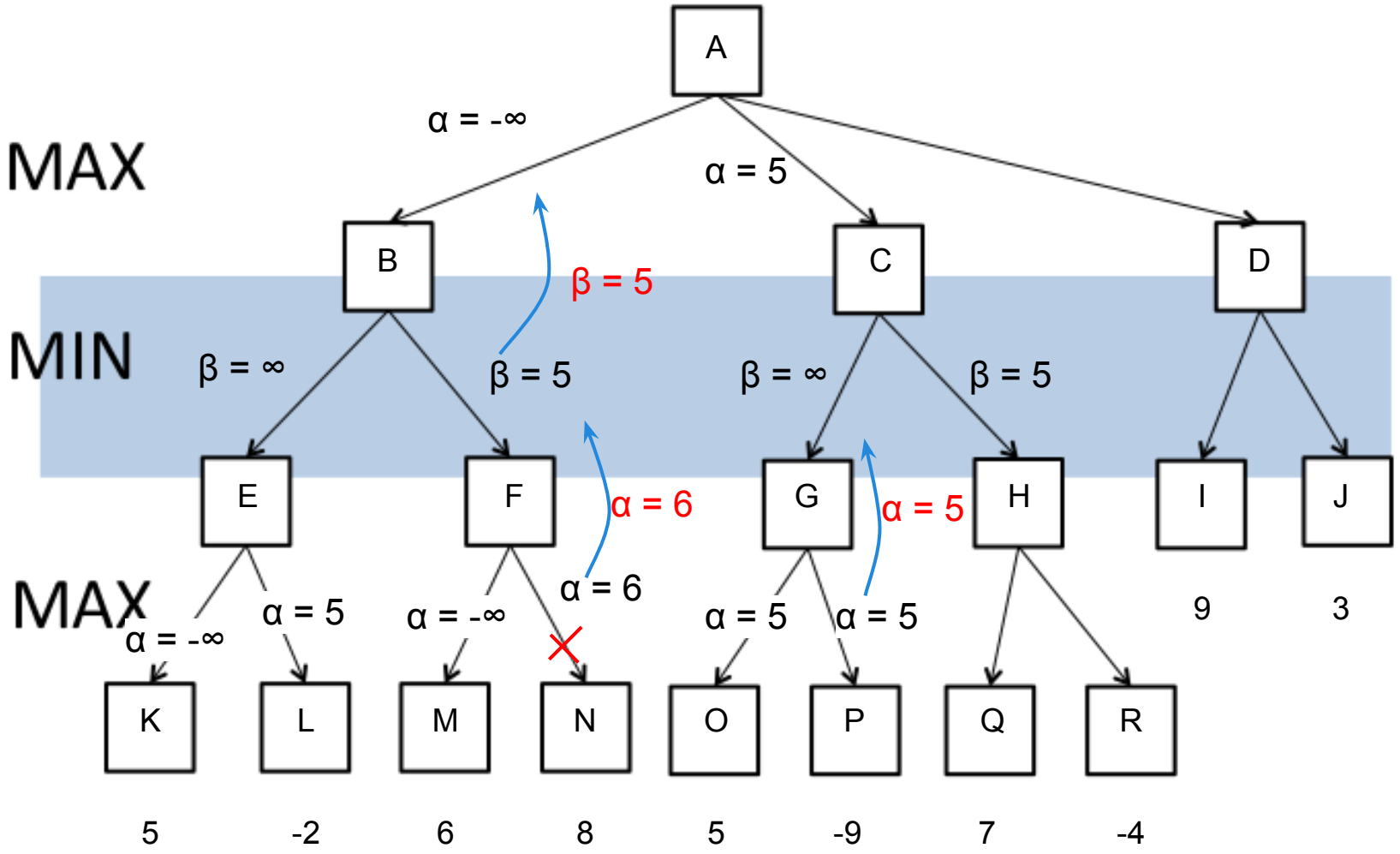
Alpha Beta Pruning



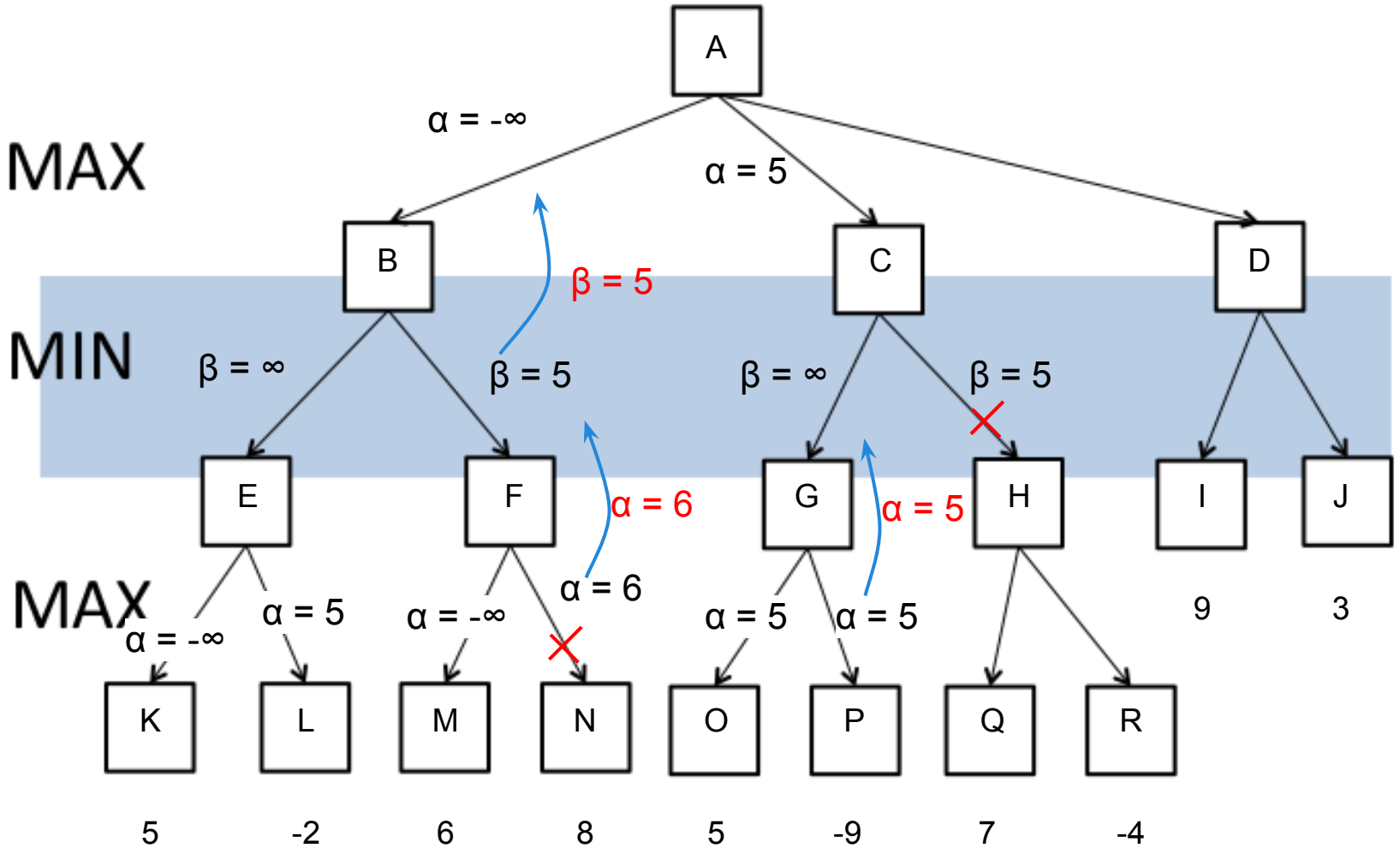
Alpha Beta Pruning



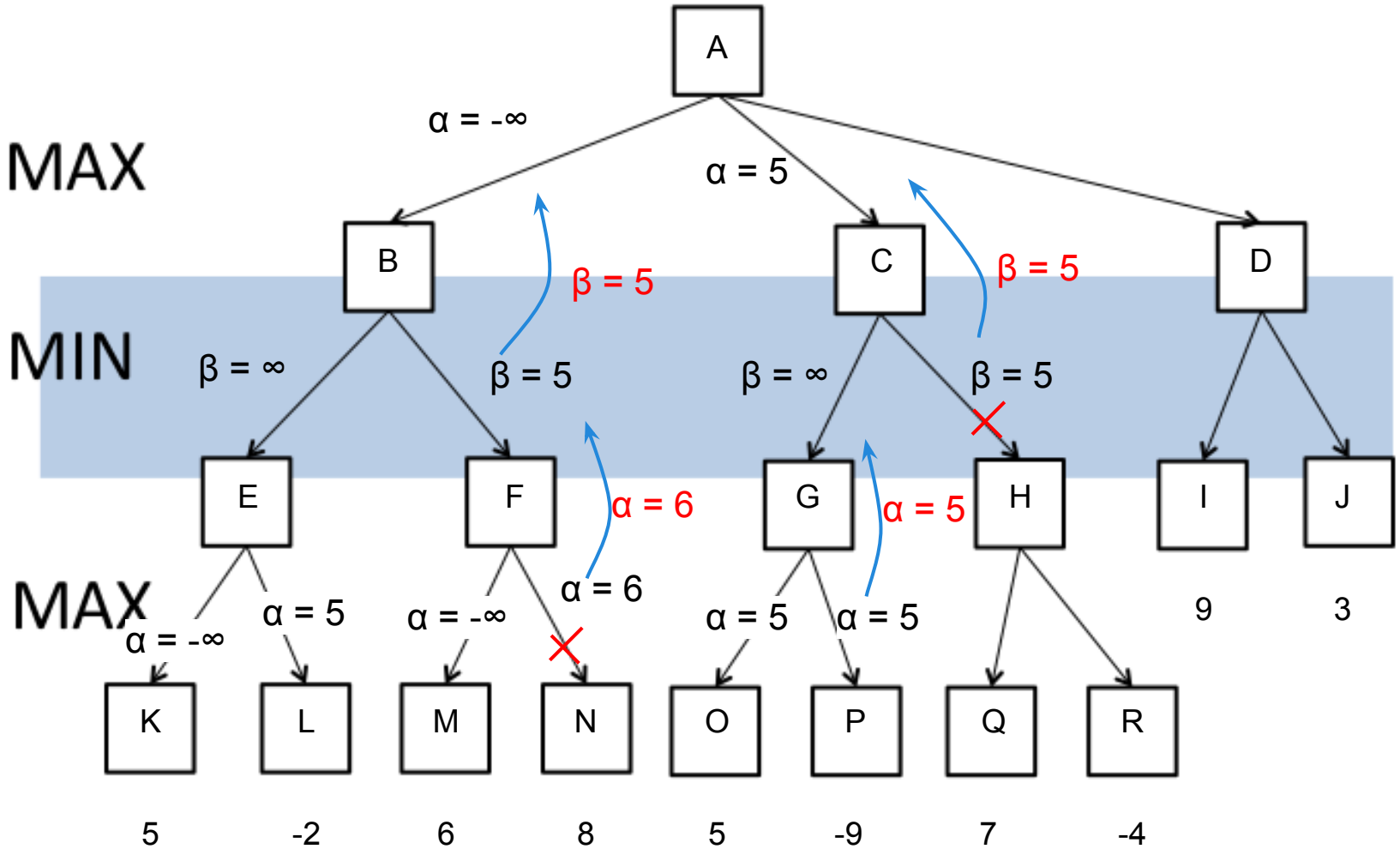
Alpha Beta Pruning



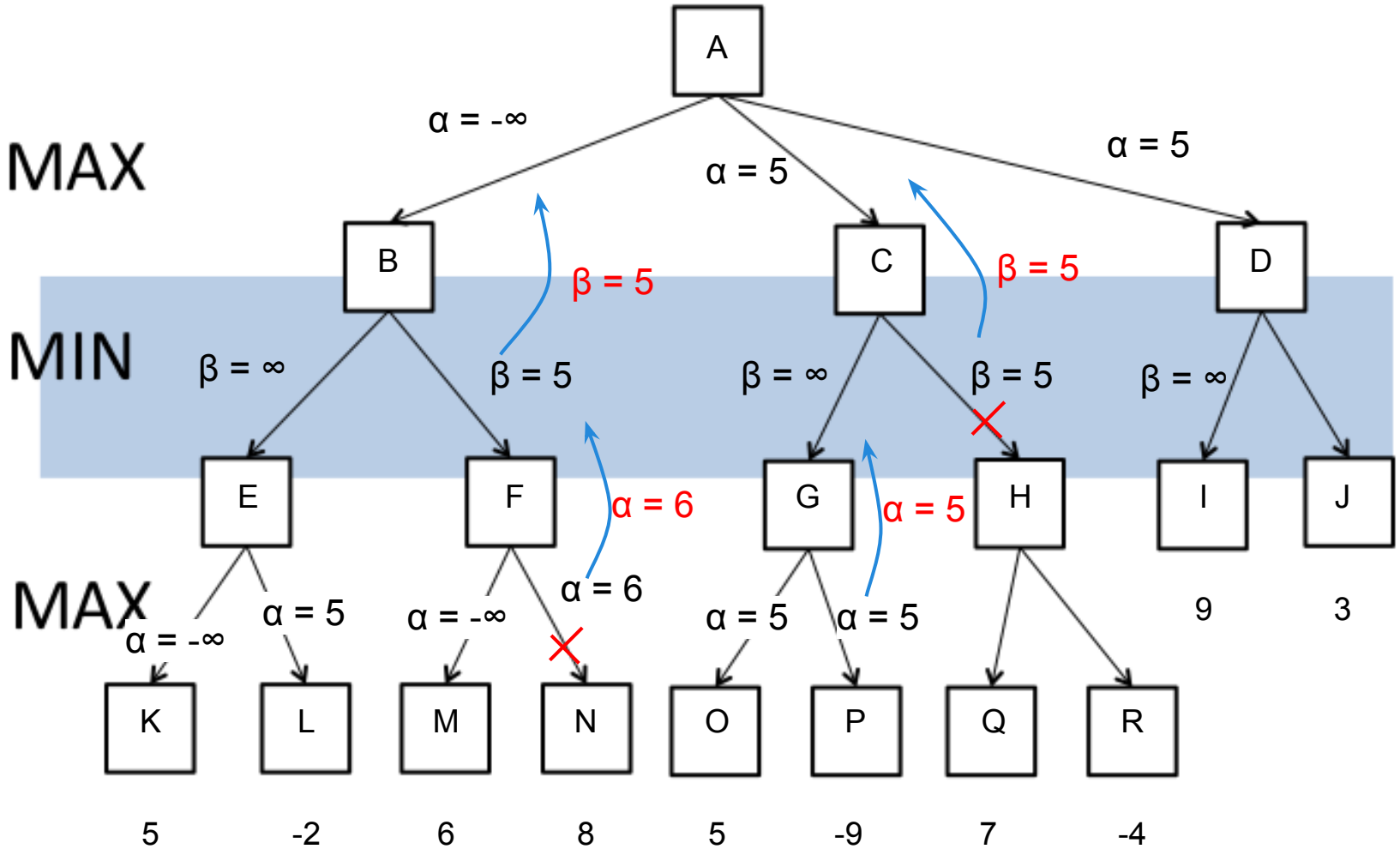
Alpha Beta Pruning



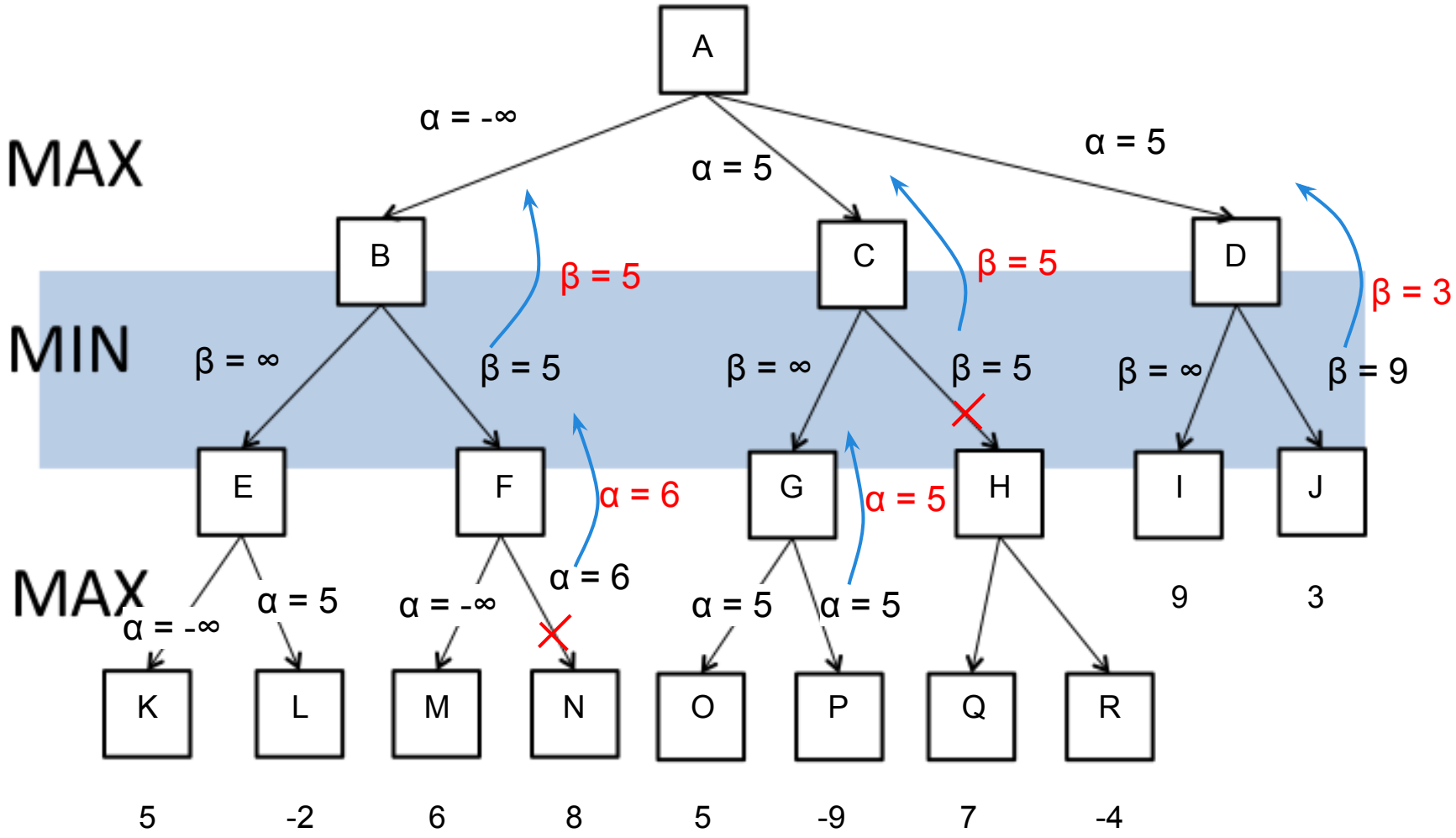
Alpha Beta Pruning



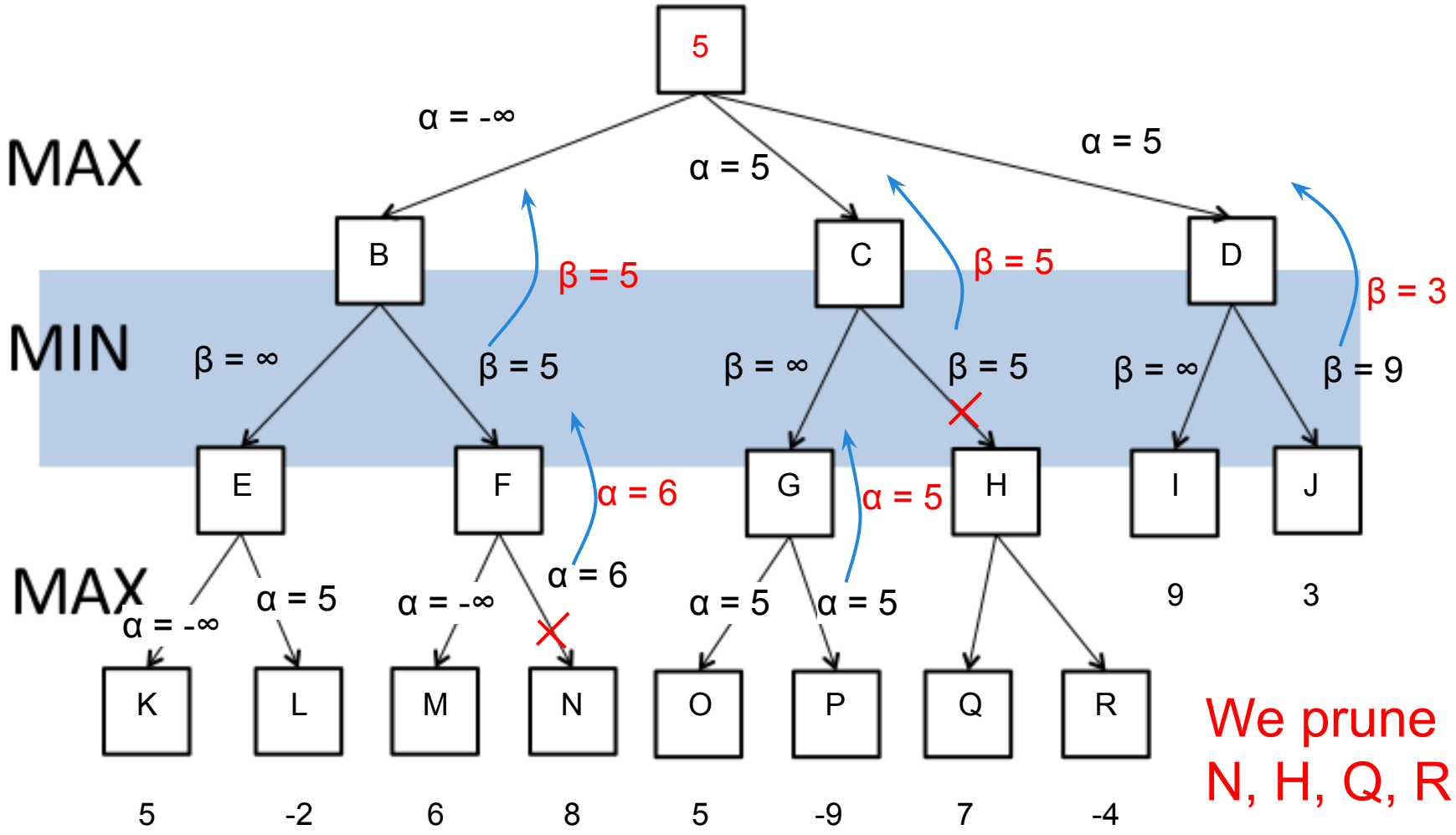
Alpha Beta Pruning



Alpha Beta Pruning

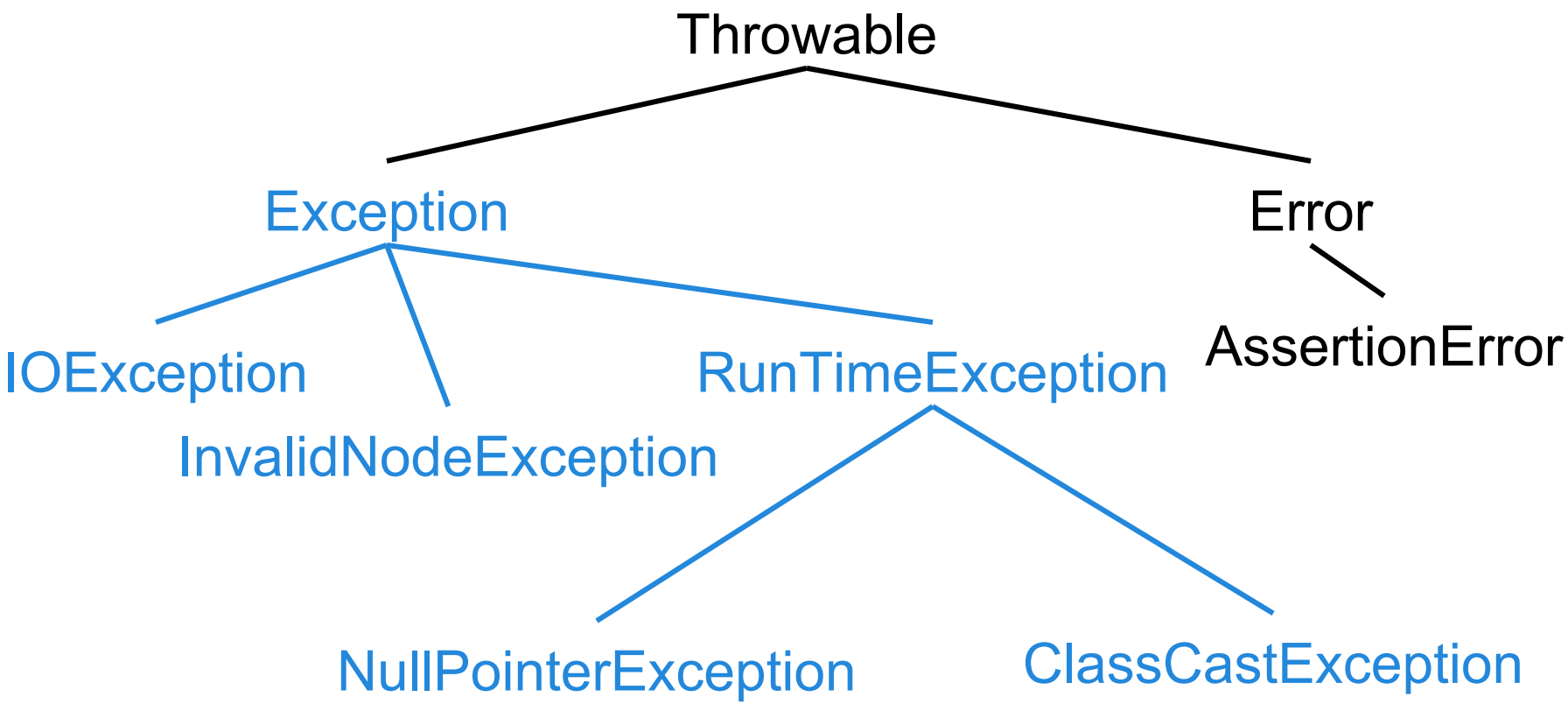


Alpha Beta Pruning



Exceptions

Throwable Class Hierarchy



Exceptions

- Checked Throwable
  - Any Exception that is not a RunTimeException
  - Must be either caught or declared!

Exceptions

- Checked Throwable
  - Any Exception that is not a RunTimeException
  - Must be either caught or declared!

```
Declared:
int riskyStuff() throws ScaryException {
    throw new ScaryException();
}
```

Exceptions

- Checked Throwable
  - Any Exception that is not a RunTimeException
  - Must be either caught or declared!

Caught:

```
void saferStuff() {
    try {
        riskyStuff();
    } catch (ScaryException e) {
        System.out.println(“Oh no!”);
    }
}
```

Finally

- Can be added to a try
- The code in the finally block will always be executed!
- Execution will jump to finally clause either when `try/catch` block finishes or right before a `return`

Exceptions

```
try {
    System.out.println(“in try”);
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
} finally {
    System.out.println(“in finally”);
}
```

Exceptions

```
try {
    System.out.println(“in try”);
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
} finally {
    System.out.println(“in finally”);
}
```

```
in try
in finally
-> returns 5
```

Exceptions

```
try {
    throw new ScaryException();
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
    return 3;
} finally {
    System.out.println(“in finally”);
}
```

Exceptions

```
try {
    throw new ScaryException();
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
    return 3;
} finally {
    System.out.println(“in finally”);
}
```

```
in catch
in finally
-> return 3
```

Exceptions

```
try {
    throw new ScaryException();
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
    return 3;
} finally {
    System.out.println(“in finally”);
    return 7;
}
```

Exceptions

```
try {
    throw new ScaryException();
    return 5;
} catch (ScaryException e) {
    System.out.println(“in catch”);
    return 3;
} finally {
    System.out.println(“in finally”);
    return 7;
}
```

```
in catch
in finally
-> return 7
```

Questions?

Good luck on the midterm!