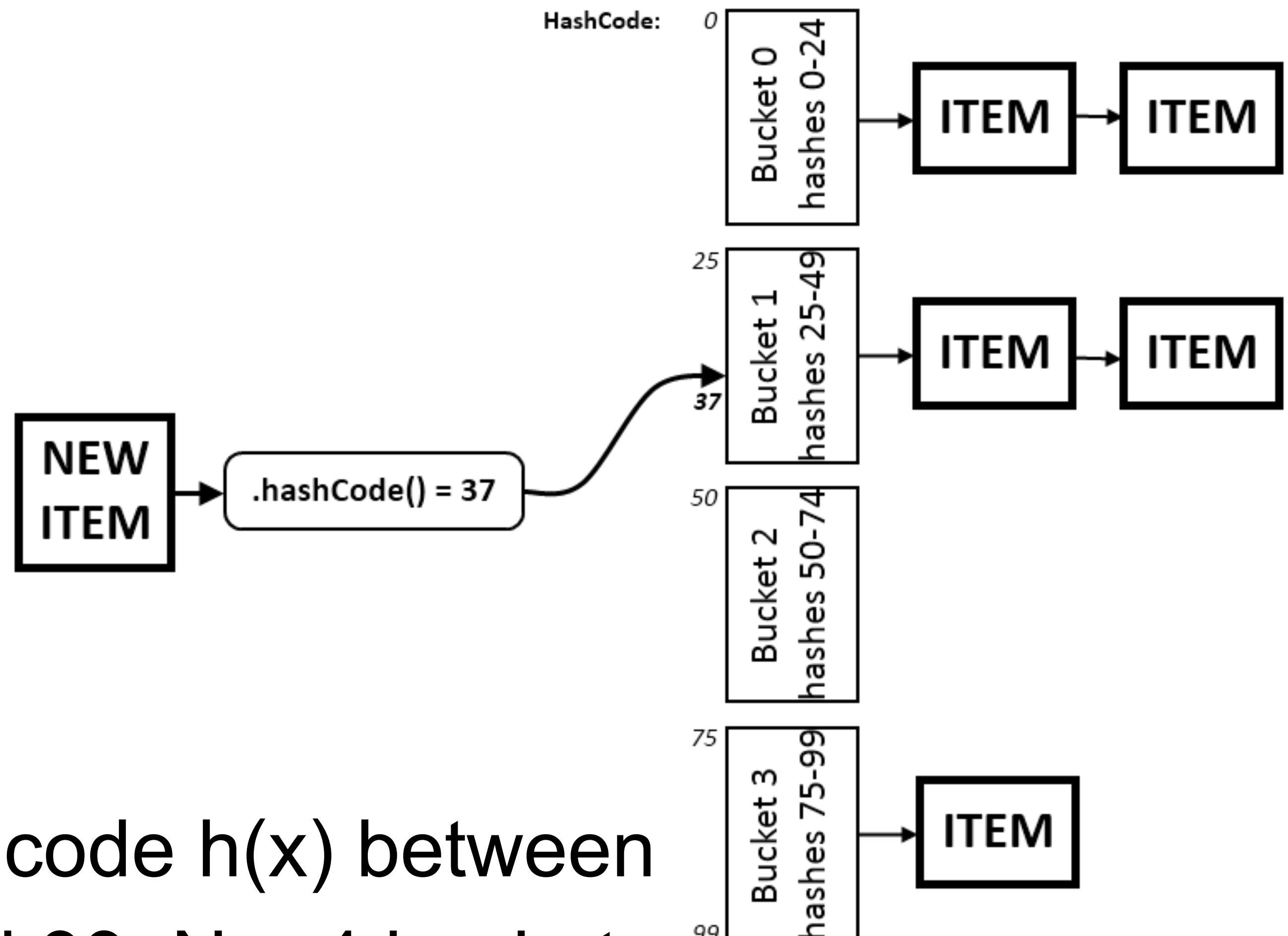
HKN CS61B Midterm 2 Review

Jimmy Lee
Evan Ye
Joey Moghadam
Chris Xie

Hashing overview

- hashCode() maps an **object** to a **code**
- A compression function maps a **code** to a **bin/bucket**
- When two objects map to the same bin, it is called a **collision**
  - collisions cause linked lists within the bins to grow, slowing down the structure
- A good hash function is *unlikely* to map several items in a set to the same bin

Basic Hash Table



Hashcode h(x) between
0 and 99, N = 4 buckets

Awful hash function example

Q: You want to add a set of words to a hash table. Why is it a bad idea to simply assign each word to a bin based on its first letter?

Awful hash function example

Q: You want to add a set of words to a hash table. Why is it a bad idea to simply assign each word to a bin based on its first letter?

A: It is very easy for certain data to slow down the hash table. For example, if every word in the set starts with either 'a' or 'b', then all of the words end up in two bins, and access time becomes O(n) instead of O(1).

Important points

- Items in a hash table are **not** in order
- Bins in a hash table are usually linked lists, but they don't have to be
- If we have *n* items in a hash table and *N* bins, then the **load factor** is *n*/*N.*
- If the load factor is < 1, we have constant access time.
- Choose hash codes and functions that distribute items evenly across the bins

More Buckets!

We notice that our load factor is increasing past 0.9. What should we do?

More Buckets!

We notice that our load factor is increasing past 0.9. What should we do?

Add more buckets! What else?

More Buckets!

We notice that our load factor is increasing past 0.9. What should we do?

Add more buckets! What else?

Rehash all of the entries. Chances are they don’t all belong in the same bucket anymore.

Design Question

We have a set of items (represented by strings). At any point we can access any item in the set.

Q: We want to be able to access any item in the set in constant time. We also want to be able to tell what is the least recently accessed item in constant time. How can we make a data structure that allows us to do this?

Design Question

We have a set of items (represented by strings). At any point we can access any item in the set.

Q: We want to be able to access any item in the set in constant time. We also want to be able to tell what is the least recently accessed item in constant time. How can we make a data structure that allows us to do this?

A: We need a hash table to access items quickly. We also need a list to keep track of the order in which the items have been accessed. This is essentially a LRU cache (see next slide)

LRU Cache

How the Least Recently Used Cache works:

```
cache = [“A”, “B”, “C”, “D”, “E”];
cache.access(“B”); // cache = [“A”, “C”, “D”, “E”, “B”]
cache.access(“E”); // cache = [“A”, “C”, “D”, “B”, “E”]
For (elem e : [“E”, “D”, “C”, “B”, “A”]) {
    cache.access(e);
}
```

LRU Cache

How the Least Recently Used Cache works:

```
cache = [“A”, “B”, “C”, “D”, “E”];
cache.access(“B”); // cache = [“A”, “C”, “D”, “E”, “B”]
cache.access(“E”); // cache = [“A”, “C”, “D”, “B”, “E”]
For (elem e : [“E”, “D”, “C”, “B”, “A”]) {
	cache.access(e);
}
cache.getLRU() = ???
```

LRU Cache

How the Least Recently Used Cache works:

```
cache = [“A”, “B”, “C”, “D”, “E”];
cache.access(“B”); // cache = [“A”, “C”, “D”, “E”, “B”]
cache.access(“E”); // cache = [“A”, “C”, “D”, “B”, “E”]
For (elem e : [“E”, “D”, “C”, “B”, “A”]) {
    cache.access(e);
}
cache.getLRU() → “E”
```

E was accessed first followed by the rest.

LRU Cache Implementation

A -> B -> C -> D -> E Keep a Linked List of items **and** have a hash table mapping a string (ie “A”) to a pointer to the node that contains A.

Here’s how the methods work:

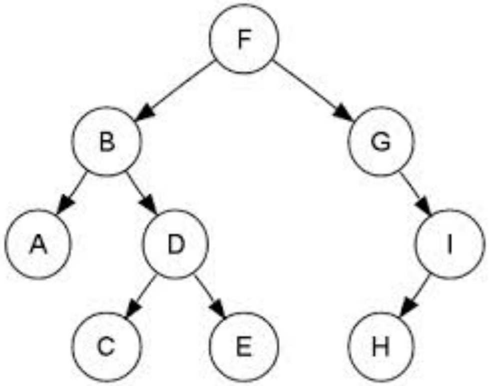
.getLRU() return LL.head.item

.access(“B”) node = hashTable.get(“B”);
LL.remove(node);
LL.addToEnd(node);

Binary Trees

Binary Tree: Each tree has at most 2 children.

No other restrictions or patterns.

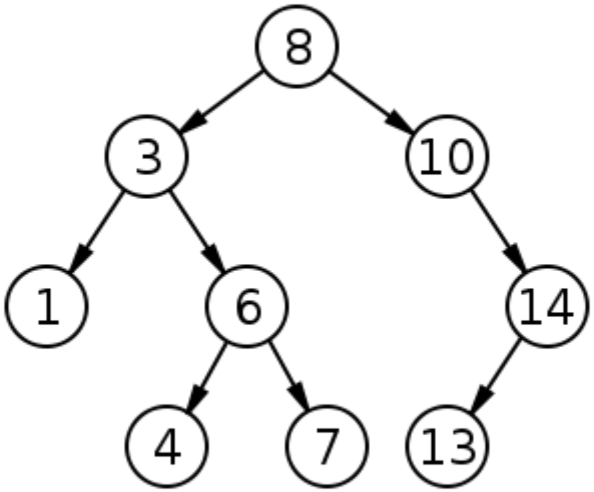


Binary Search Trees

Binary tree with property:

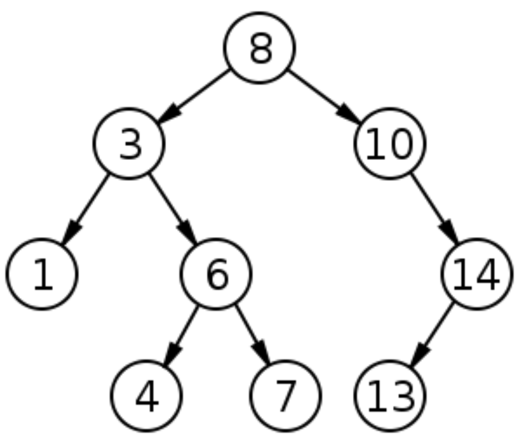
- All nodes in LEFT subtree < root
- All nodes in RIGHT subtree > root

in-order traversal gives sort



Some BST Runtimes

Operation	Average Case	Worst Case
Find	O(log n)	O(n)
Insert	O(log n)	O(n)
Remove	O(log n)	O(n)
Construct	O(nlog n)	

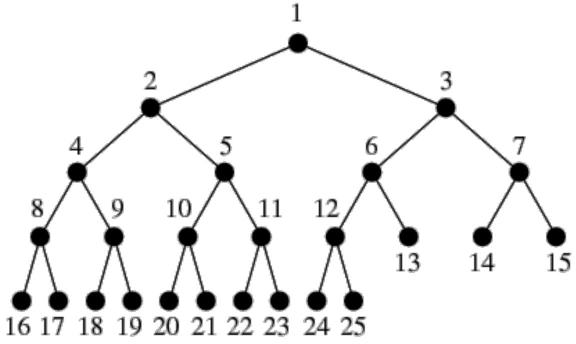


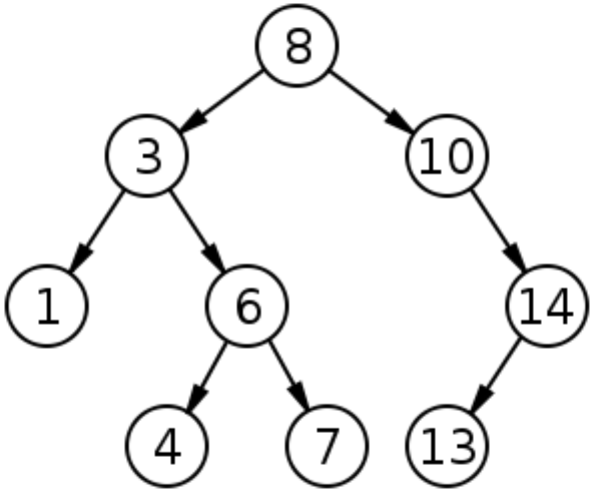
Everything is O (n) worst case!

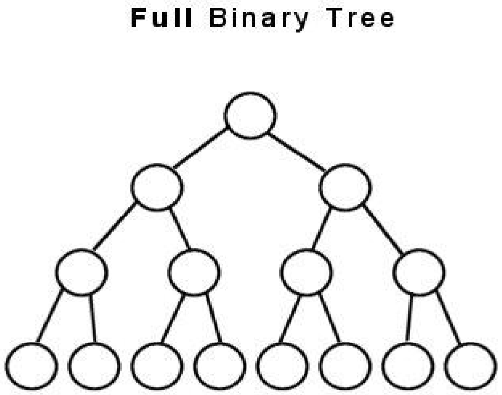
Why?

BST Quiz Questions

- Write functions for:
  - Finding the 2nd-least element
  - Checking if a tree is complete.
  - Checking if a tree is a BST







Checking BST Property: First Try

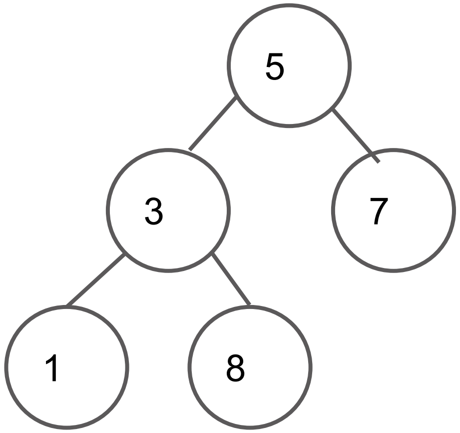
```
is_bst(tree) {
  if(!tree) return true;
  return tree.left.value < tree.value
    && tree.right.value > tree.value
    && is_bst(tree.left)
    && is_bst(tree.right)
}
```

What's wrong with this?

Checking BST Property: First Try

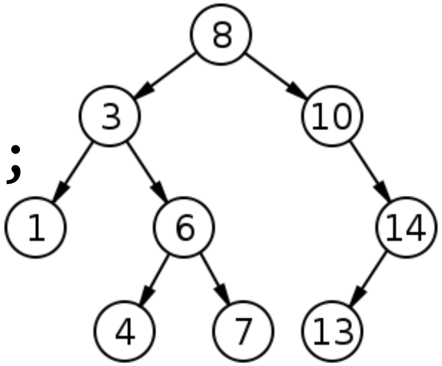
```
is_bst(tree) {
  if(!tree) return true;
  return tree.left.value < tree.value
    && tree.right.value > tree.value
    && is_bst(tree.left)
    && is_bst(tree.right)
}
```

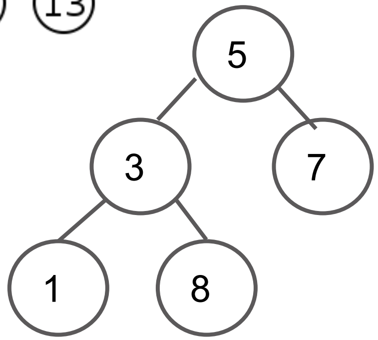
Fails for trees like:



Checking BST Property: Correct

```
is_bst(tree) {
   return is_bst(tree, -inf, +inf);
}
```

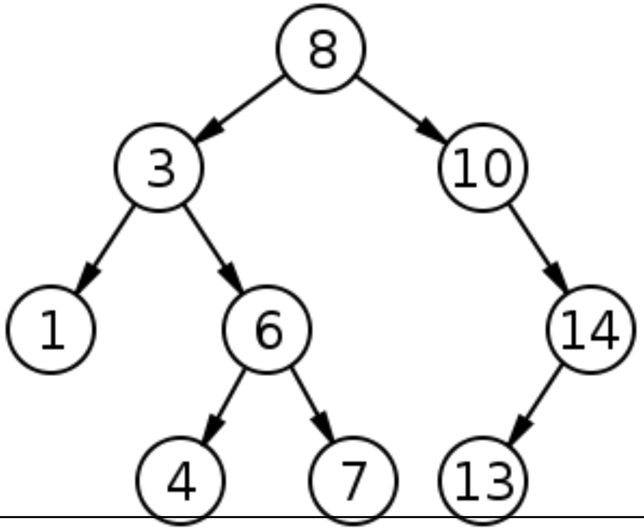





```
is_bst(tree, min, max) {
   if (!tree) return true;
   return     tree.value > min
         && tree.value < max
         && is_bst(tree.left, min, tree.value)
         && is_bst(tree.right, tree.value, max);
}
```

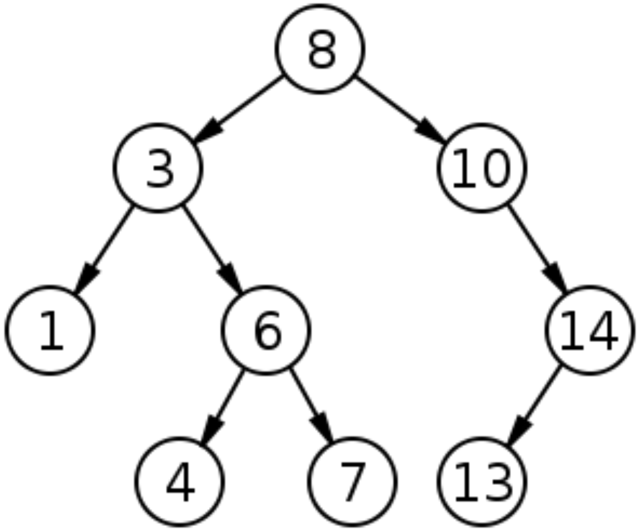
BST Intermediate Questions

- Reconstruct a BST given pre-order and in-order traversals.
- Example:
  - Pre-order: 8,3,1,6,4,7,10,14,13
  - In-order: 1,3,4,6,7,8,10,13,14



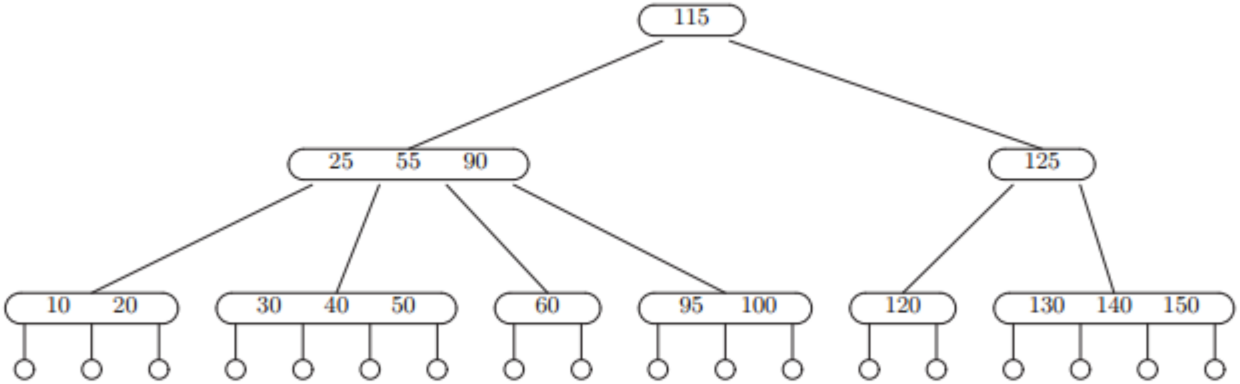
BST Intermediate Questions

- Reconstruct a BST given pre-order and in-order traversals.
- Example:
  - Pre-order: 8,3,1,6,4,7,10,14,13
  - In-order: 1,3,4,6,7,8,10,13,14



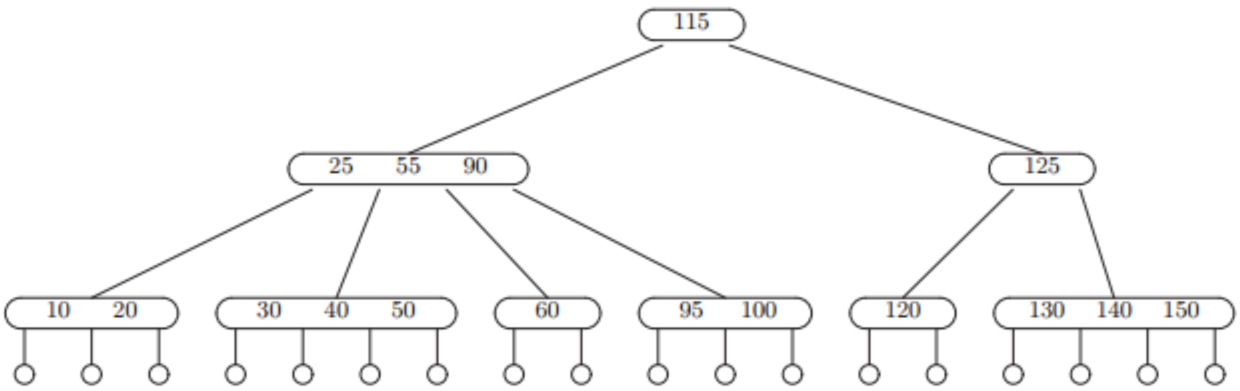
2-3-4 Trees

- Each node contains:
  - 2 to 4 children
  - 1 to 3 keys
- All empty children occur at same level ("balanced")



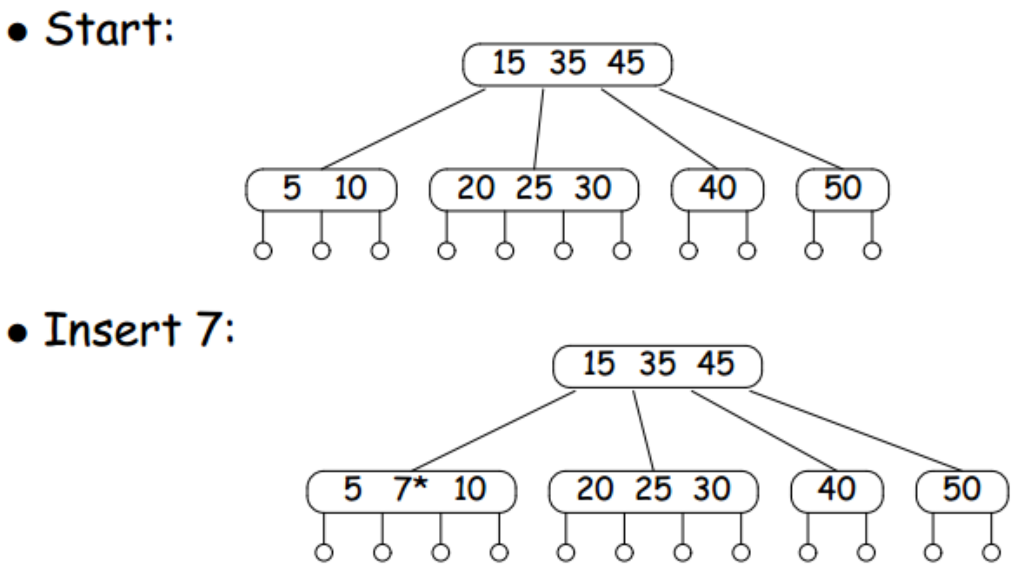
2-3-4-tree Find

- Essentially same as BST-find.
- Only difference: > 1 comparison per node
  - How many, worst case?
- height = O(log n) always!

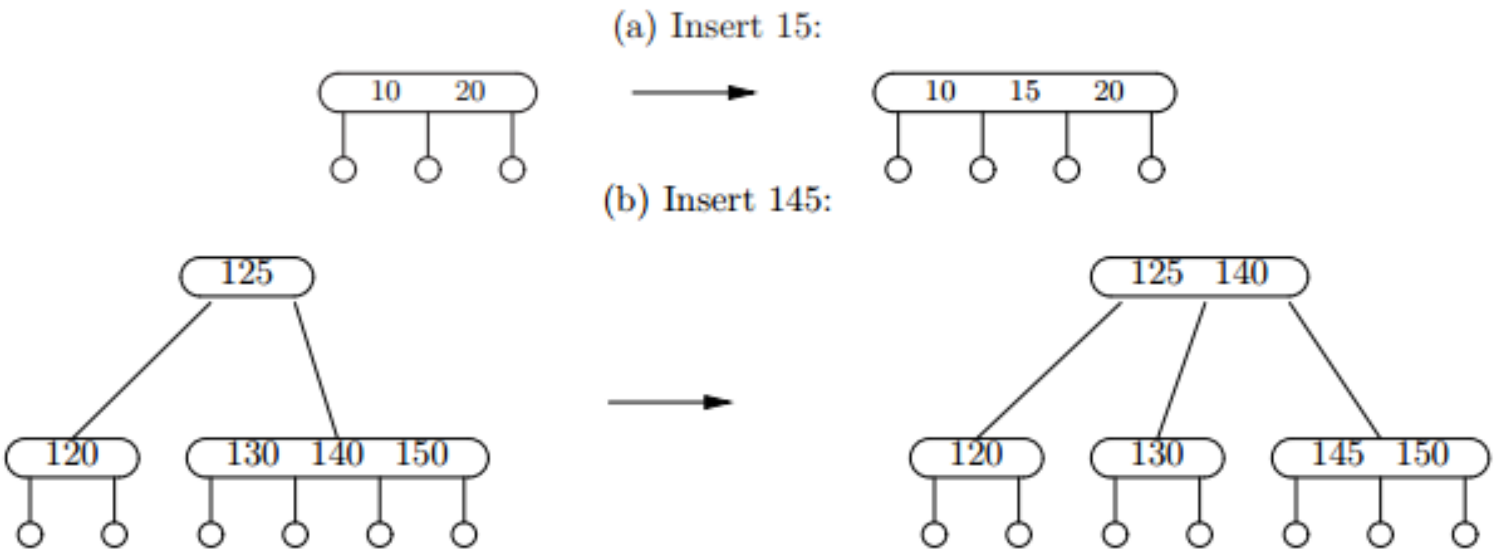


2-3-4-tree Insert (Simple case)

- Insert new key into existing node at bottom of tree



2-3-4 tree Insert (Additional Example)



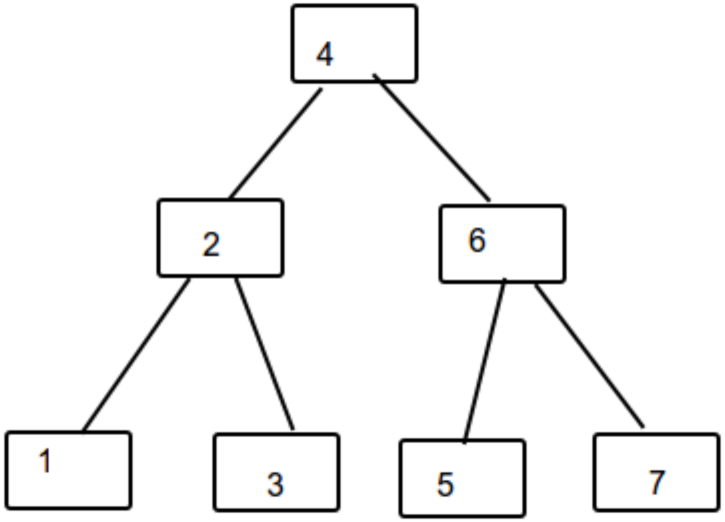
2-3-4-tree Delete

- Deleting from inner nodes complicated...

- While traversing, **combine** when you find a 1-key node
  - **steal from neighbors/rotate** - if your left or right neighbor has more than 1 key, rotate
  - **otherwise - pull down (fusion)** by stealing a parent node and combining with a neighbor

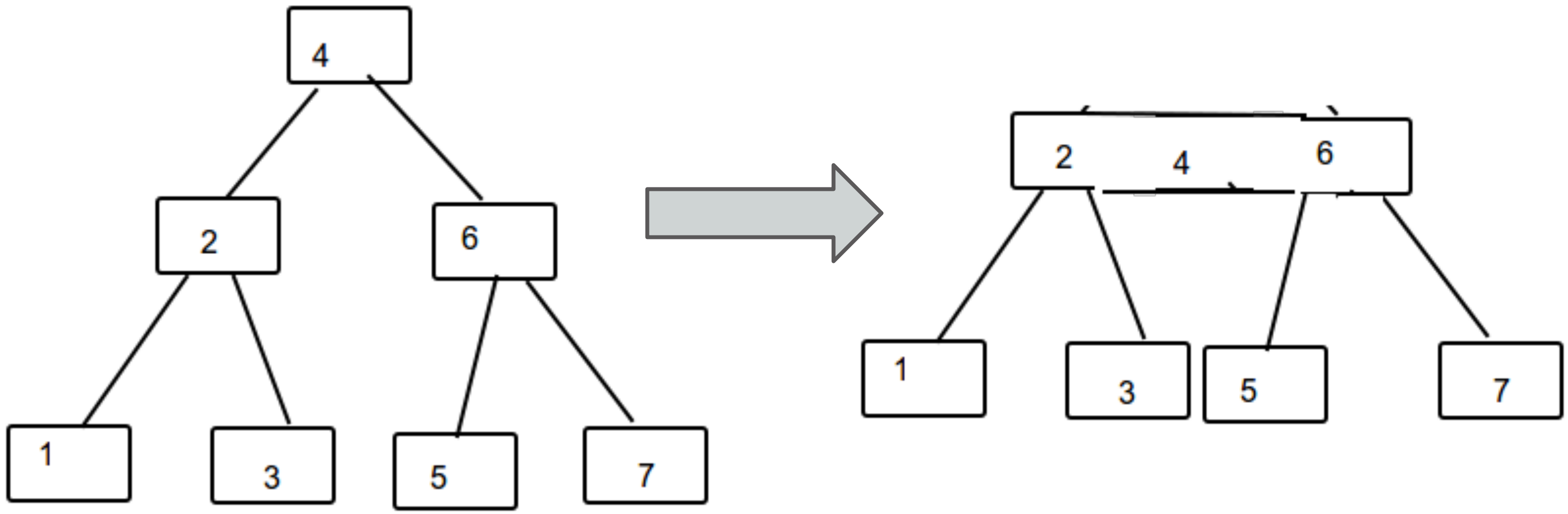
Tricky Deletion

Remove the number 7 from the 2 3 4 Tree.



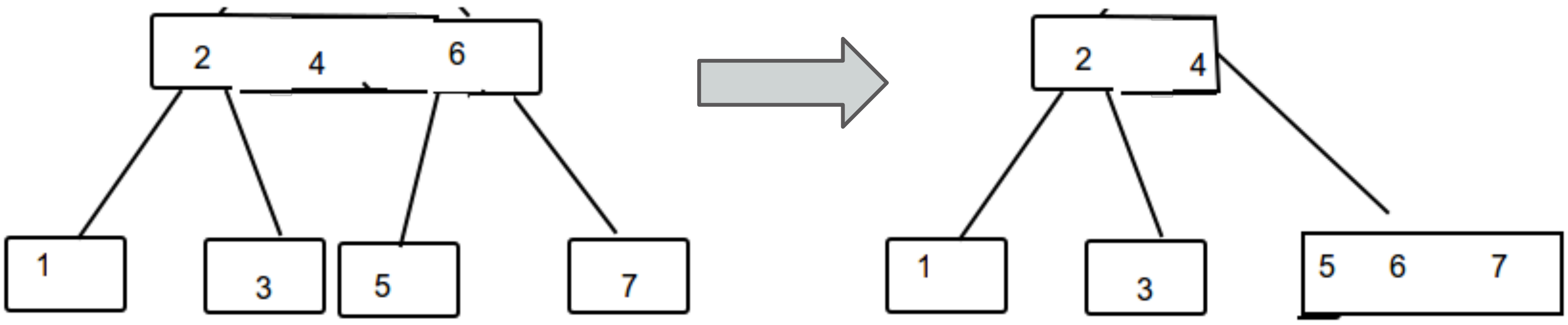
Tricky Deletion

Fusion!



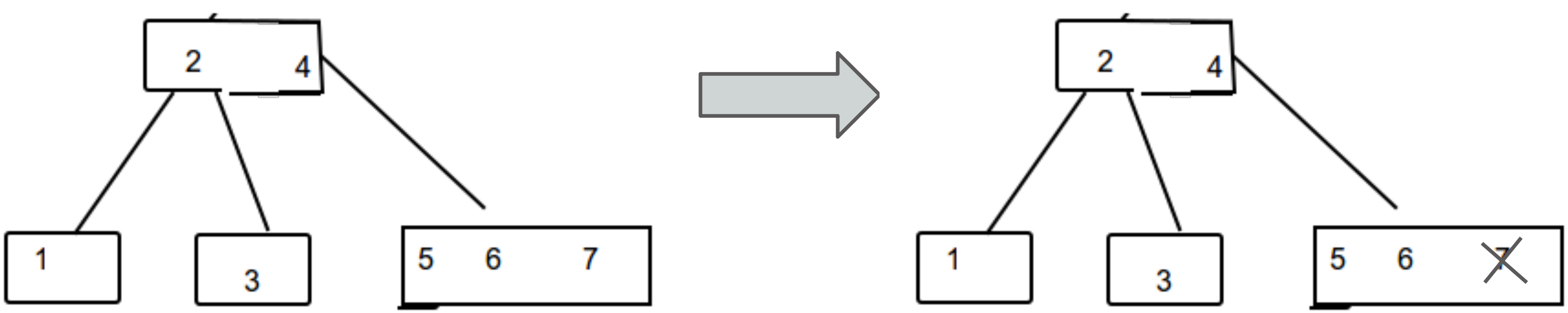
Tricky Deletion

Fusion Again!



Tricky Deletion

Remove 7



Some 2-3-4-tree Runtimes

Operation	Average Case	Worst Case
Find	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Remove	O(log n)	O(log n)
Construct	O(nlog n)	O(nlog n)

Everything is O(log n)!

B-trees are *balanced search trees*

Algorithm Analysis

Goal

- Understand how ***efficient*** our programs are

Approach

- Write the number of steps a program does as a function of the input size. Consider the ***best*** and ***worst cases***

- Understand how the function grows using ***asymptotic analysis***

Asymptotic Analysis

Goal

- Understand how a function grows

Approach

- Find simple functions that can ***upperbound*** our function (big $\boldsymbol{O}$) and ***lowerbound*** our function (big $\boldsymbol{\Omega}$)

Algorithm Analysis

What are the best and worst-case runtimes of this method?
How could the worst-case runtime be improved?

```
public static int explode(String s){
   if (s.length()==0){
      return 0;
   } else if (s.charAt(0)=='x'){
      return 1 + explode(s.substring(1));
   } else {
      return 1 + explode(s.substring(1)) +
      explode(s.substring(1));
   }
}
```

Algorithm Analysis Answer

If **n** is the length of the String **s**...

Best case upperbound: **O(n)**

Best case lowerbound: **Ω(n)**

Worst case upperbound: **O(2^n)**

Worst case lowerbound: **Ω(2^n)**

To fix, replace the else case with:

```
return 1 + 2*explode(s.substring(1));
```

Now the worst case bounds are **O(n)** and **Ω(n)**

Asymptotic Analysis

Show that

$$\mathbf{n! + (n-1)! + (n-2)! + \ldots + 1}$$

is in O(n!)

Asymptotic Analysis Answer

$n! + (n-1)! + (n-2)! + \ldots + 1$

$< n! + (n-1)! + (n-1)! + \ldots + (n-1)!$, for $n > 2$

$< n! + (n-1)*(n-1)!$

$< n! + n*(n-1)!$

$< n! + n!$

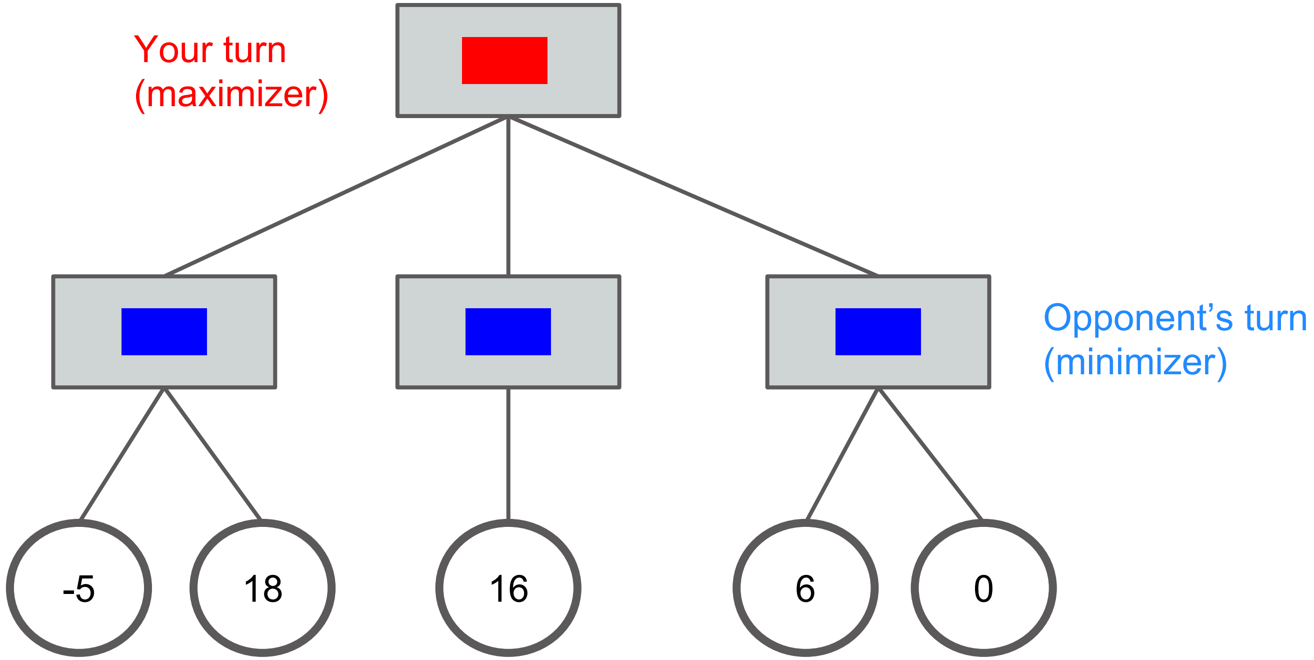
$< 2*n!$

So we can choose $c = 2$ and $N = 3$ to prove it is in $O(n!)$

Alpha Beta

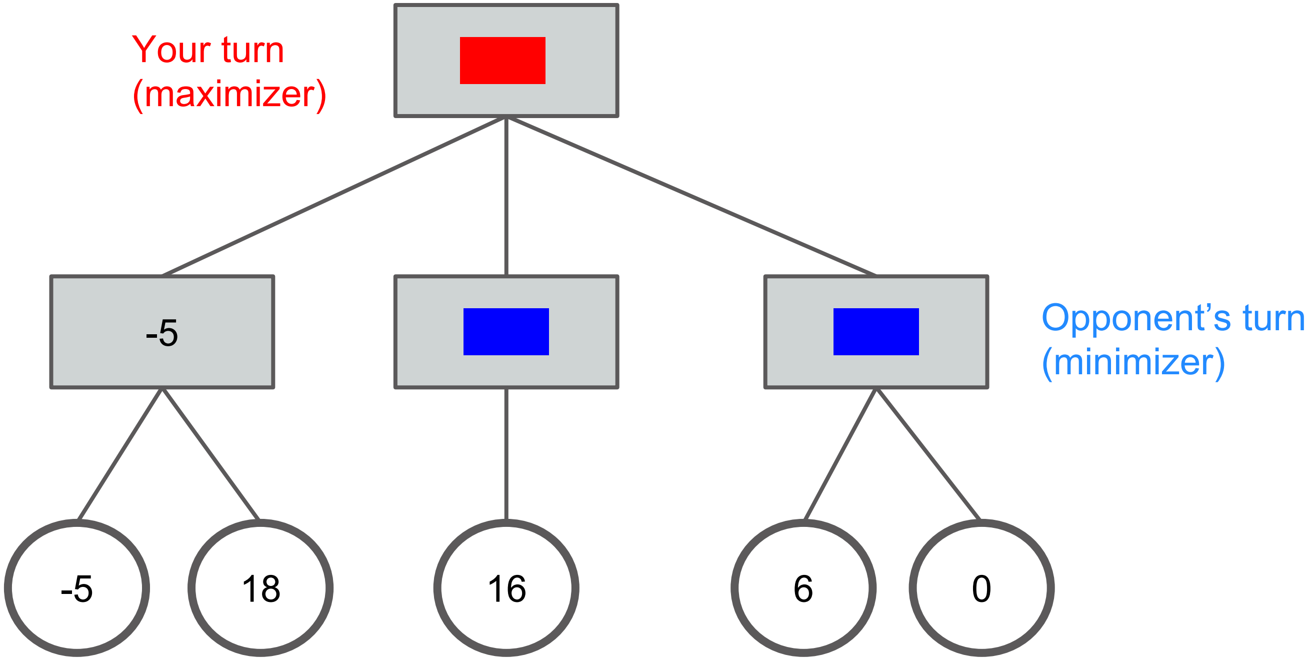
Minimax

Complete minimax on the tree below:



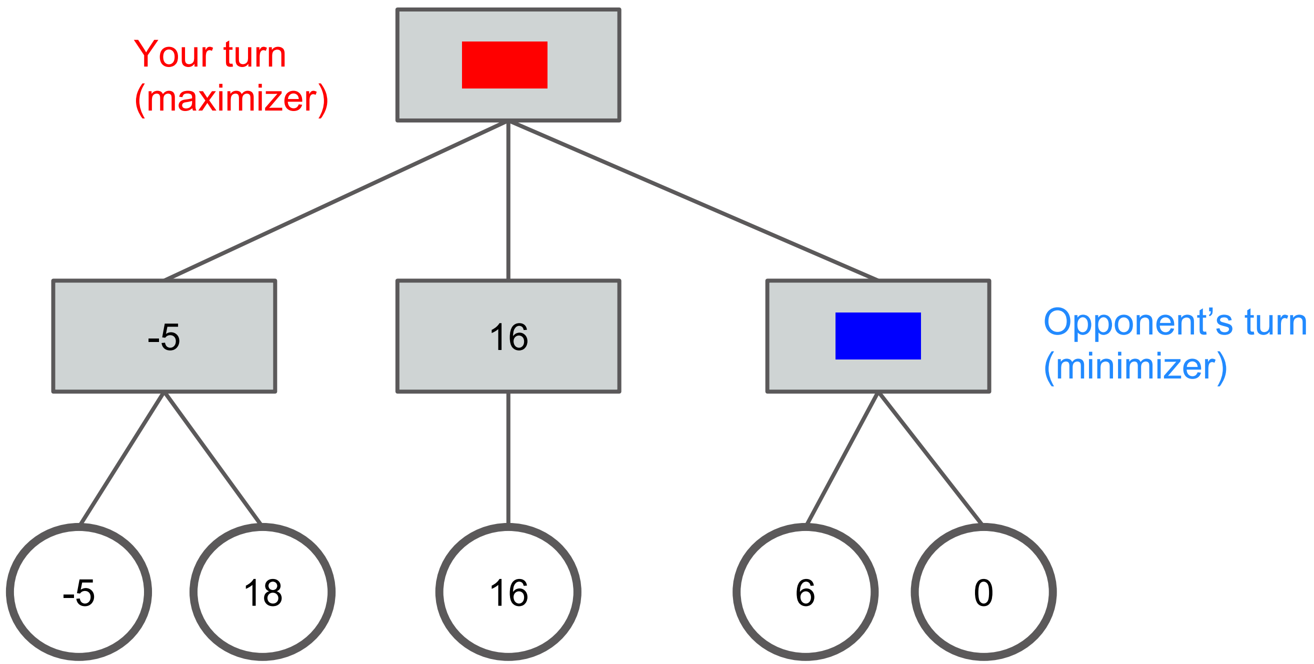
Minimax

Complete minimax on the tree below:



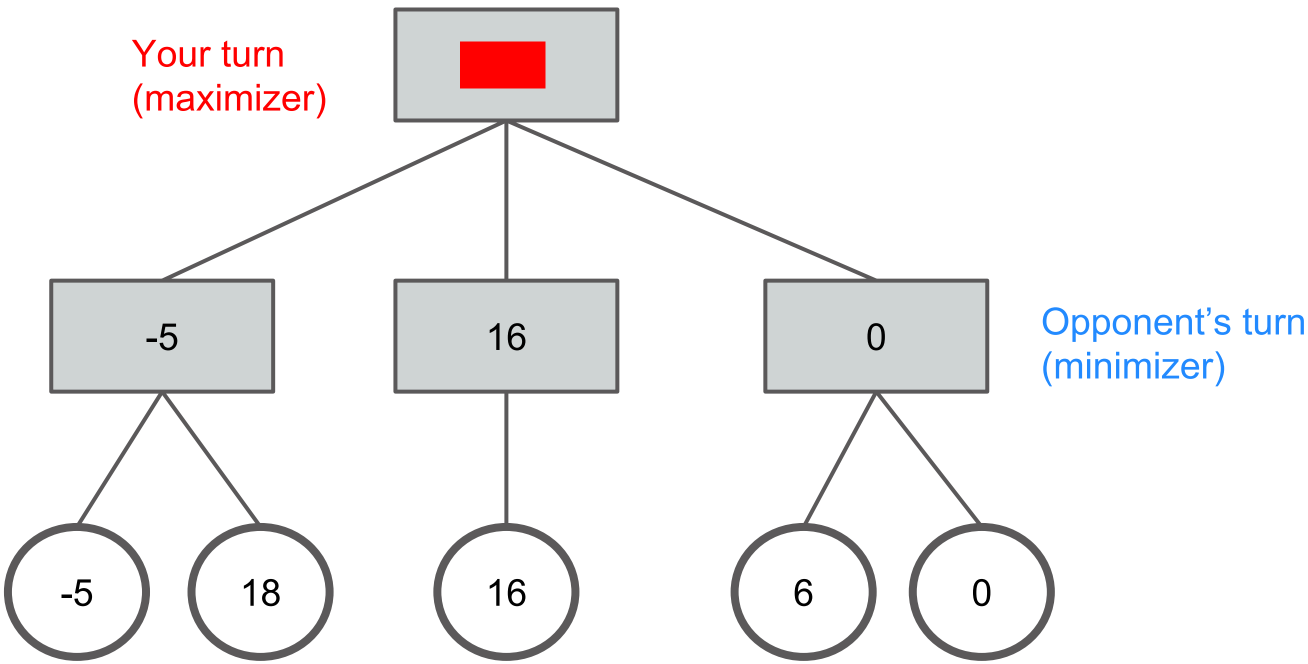
Minimax

Complete minimax on the tree below:



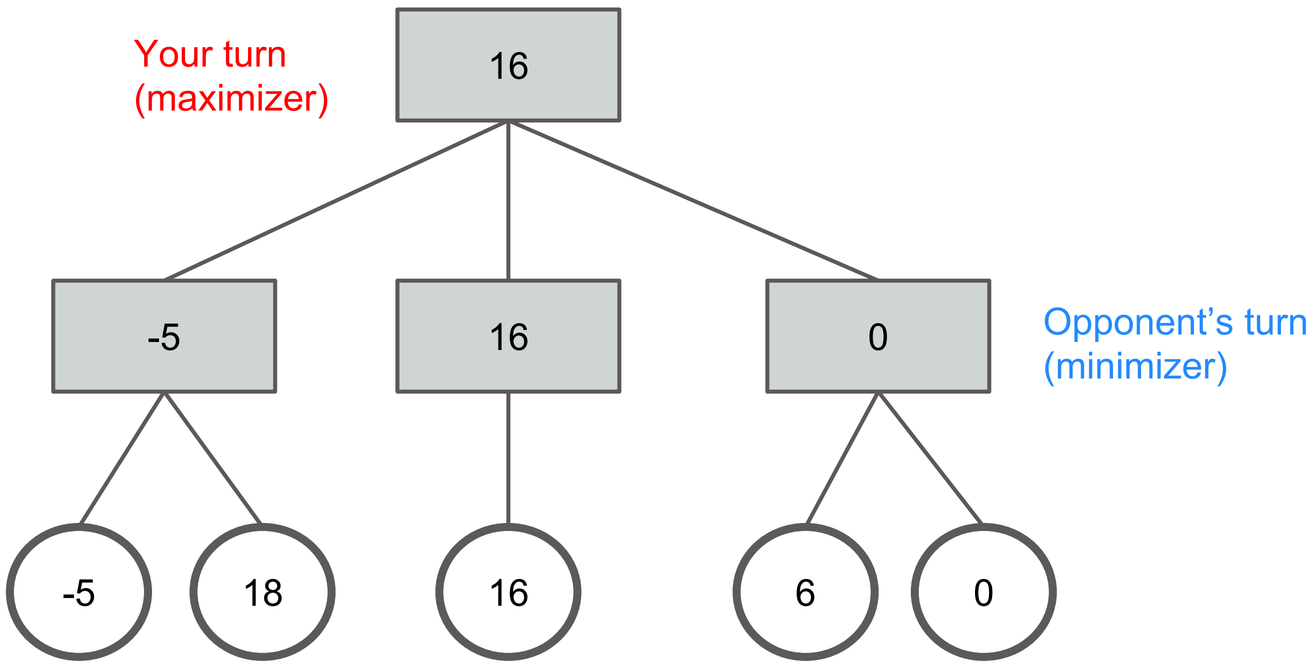
Minimax

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

Basically, when $\alpha \geq \beta$

Alpha Beta Pruning

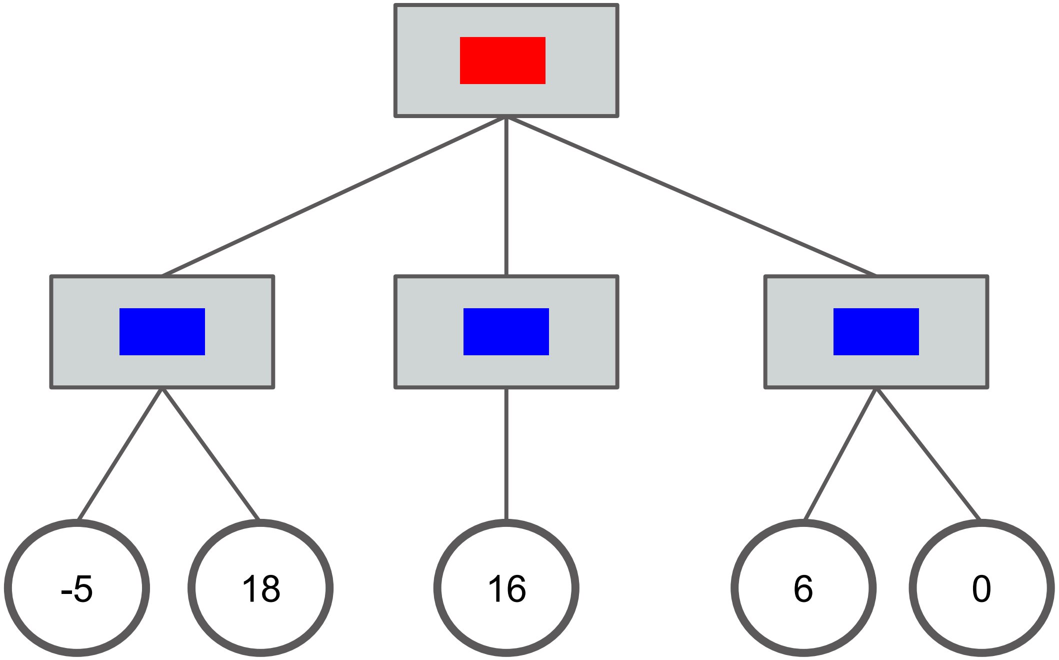
How to:

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.

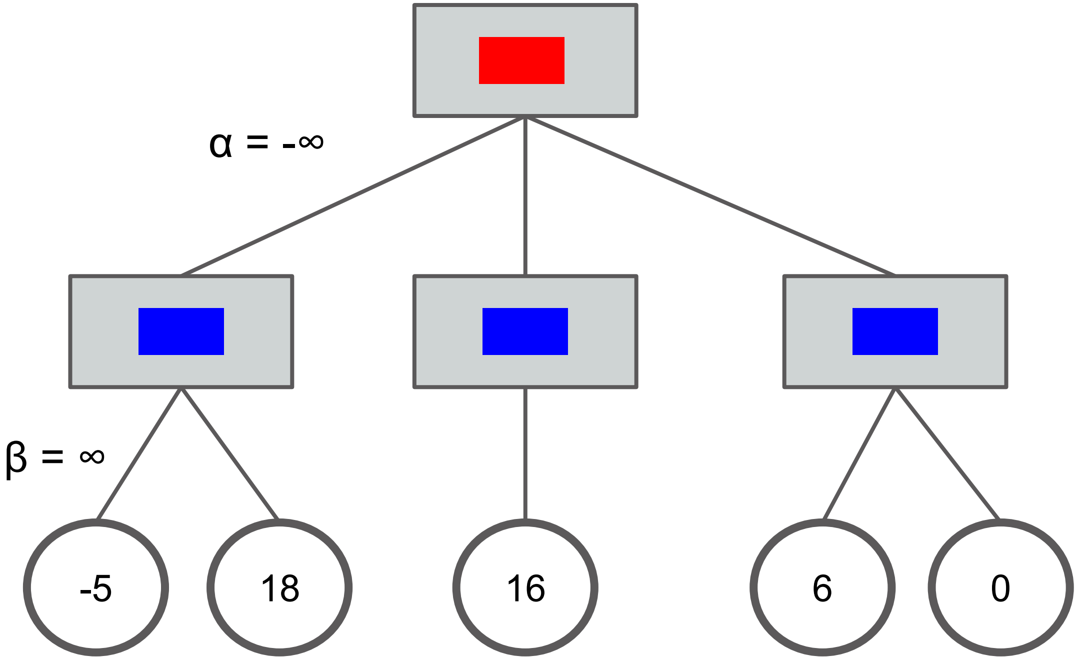
Alpha-Beta example

Perform alpha-beta pruning on the following tree



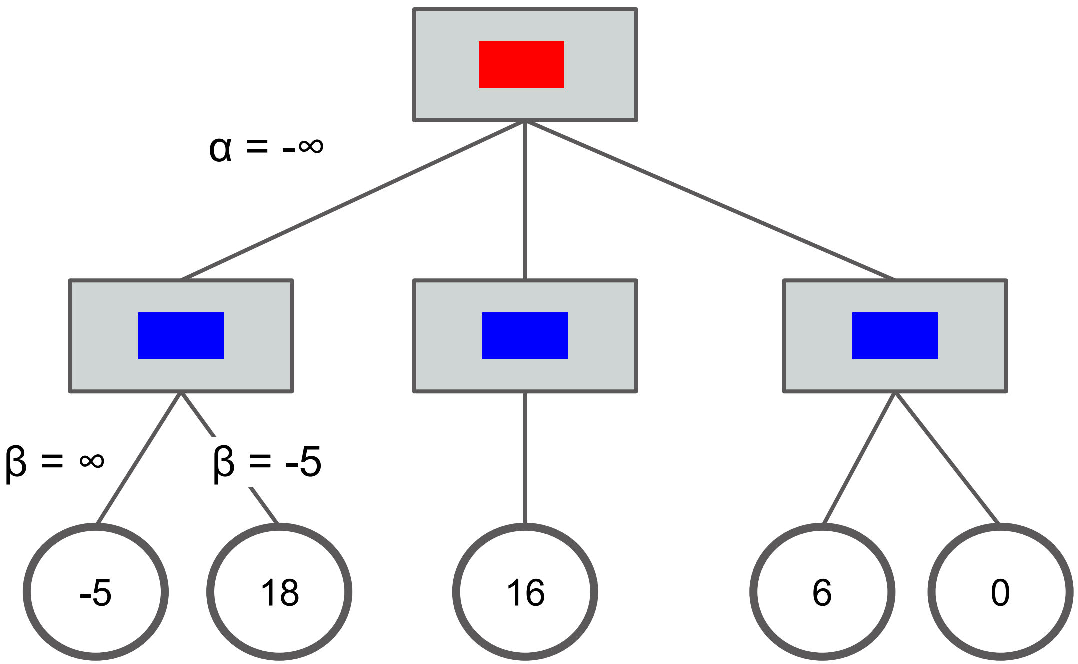
Alpha-Beta example

Perform alpha-beta pruning on the following tree



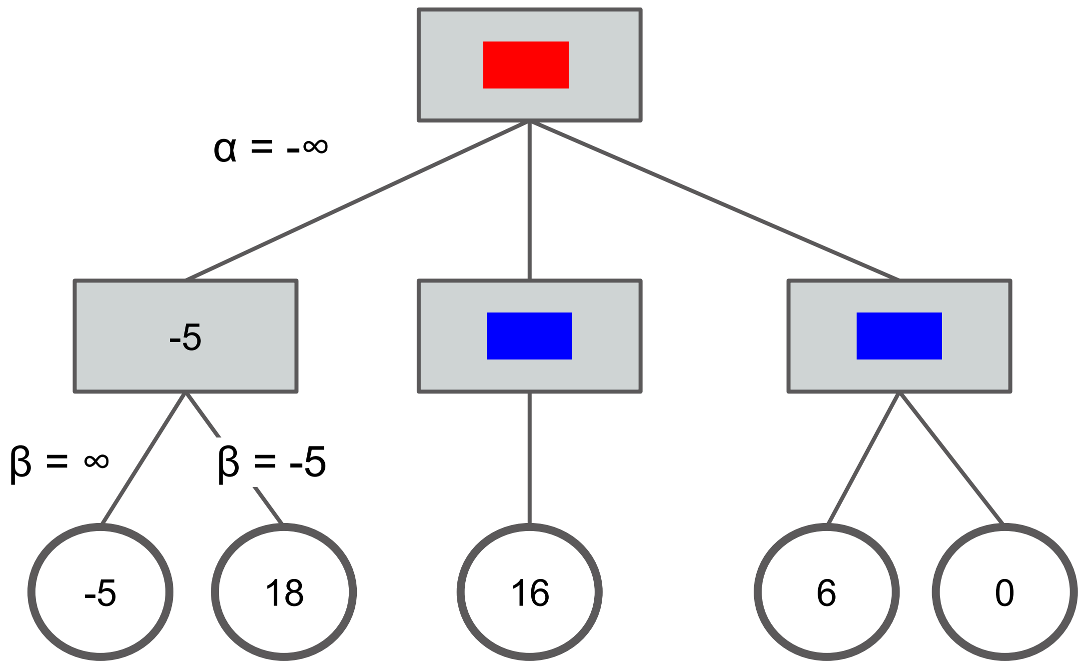
Alpha-Beta example

Perform alpha-beta pruning on the following tree



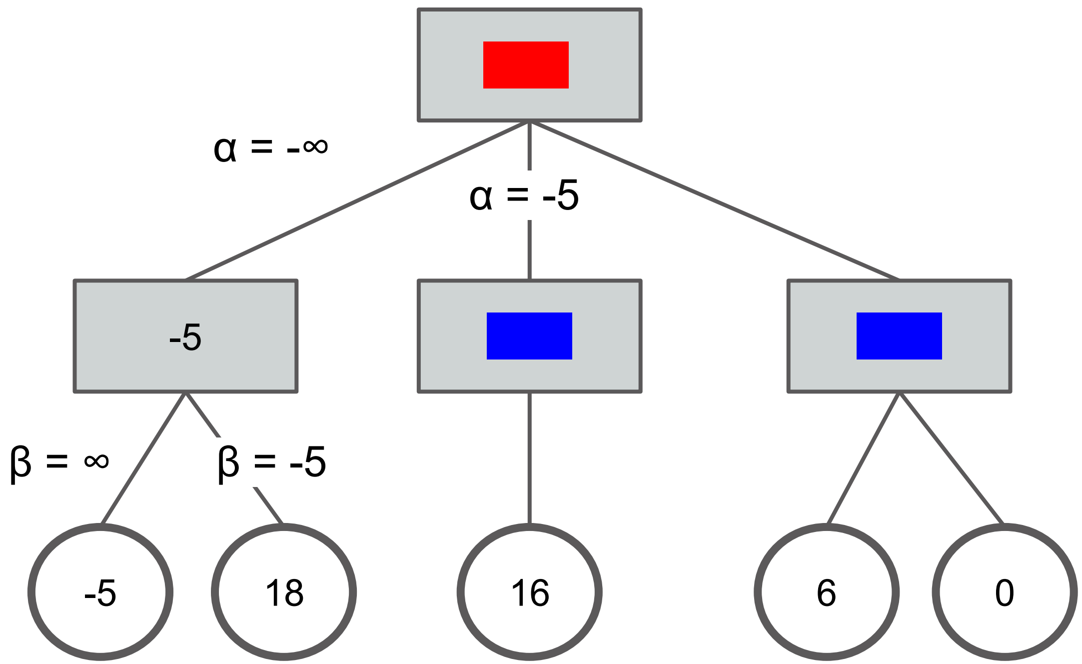
Alpha-Beta example

Perform alpha-beta pruning on the following tree



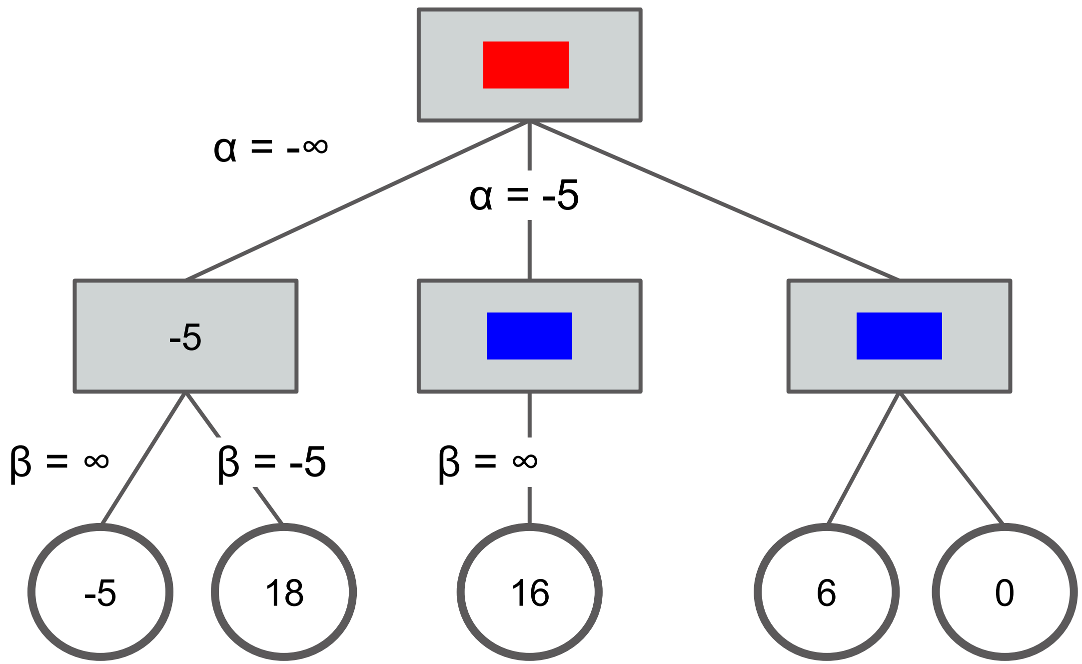
Alpha-Beta example

Perform alpha-beta pruning on the following tree



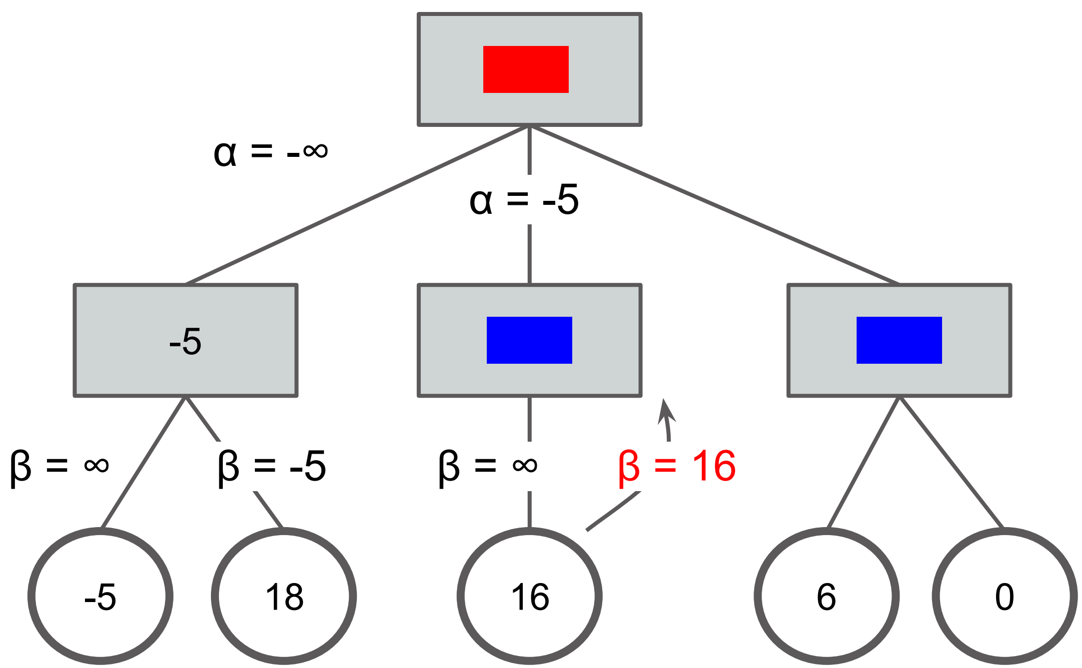
Alpha-Beta example

Perform alpha-beta pruning on the following tree



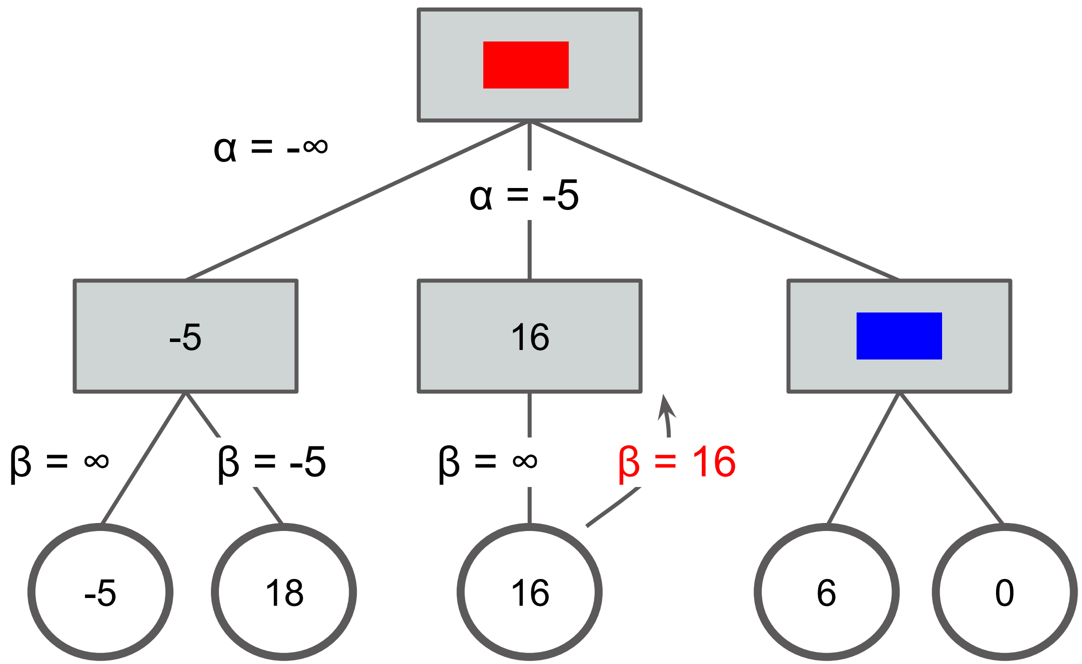
Alpha-Beta example

Perform alpha-beta pruning on the following tree



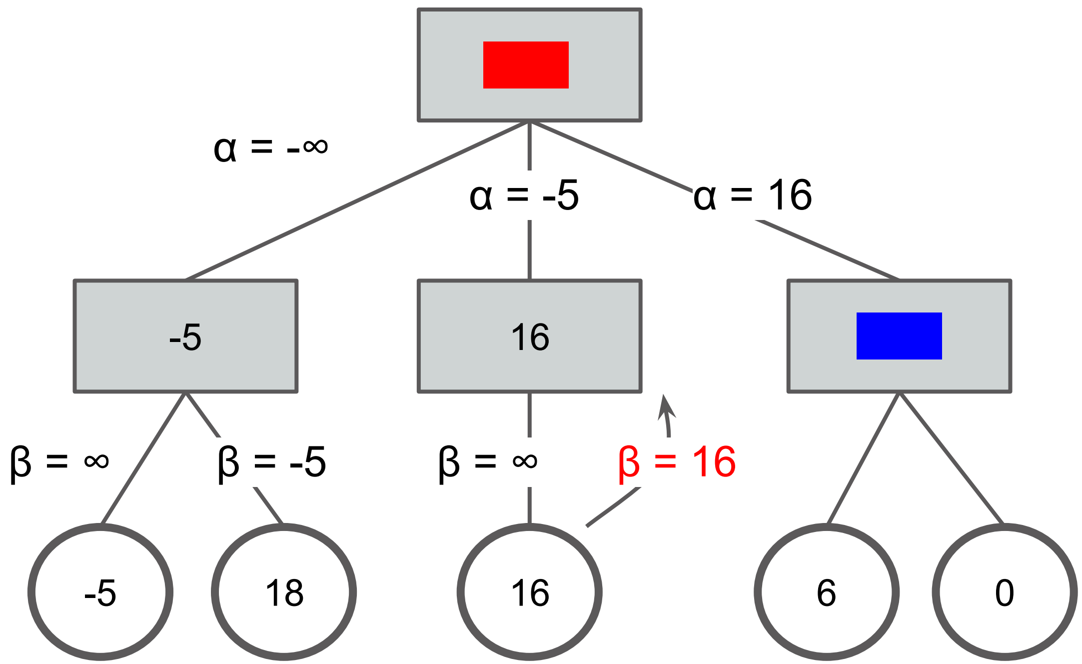
Alpha-Beta example

Perform alpha-beta pruning on the following tree



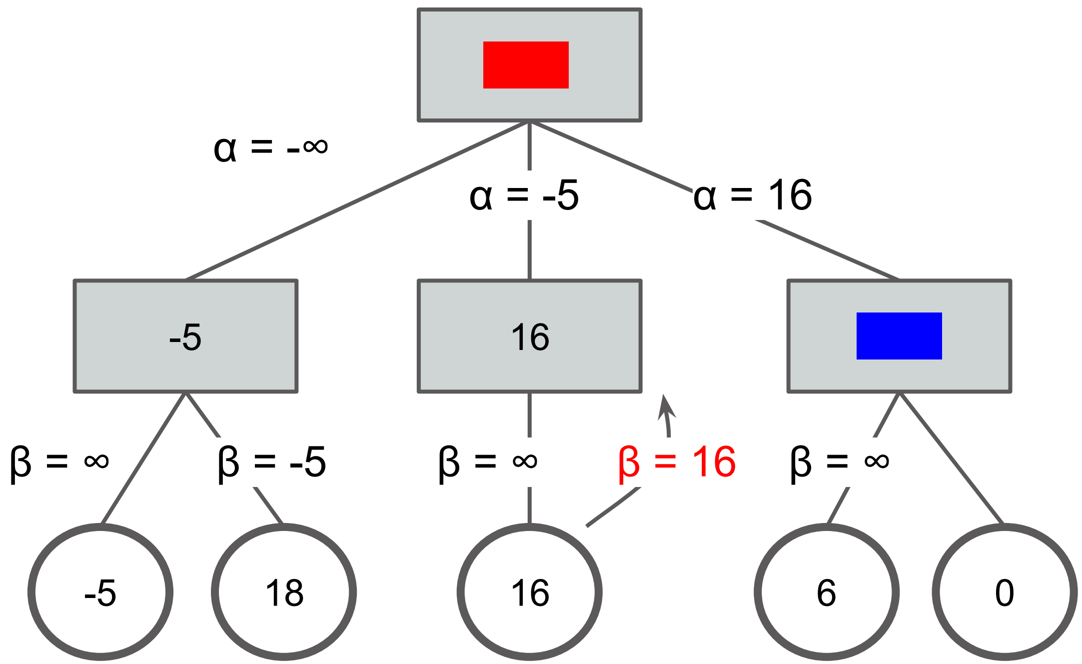
Alpha-Beta example

Perform alpha-beta pruning on the following tree



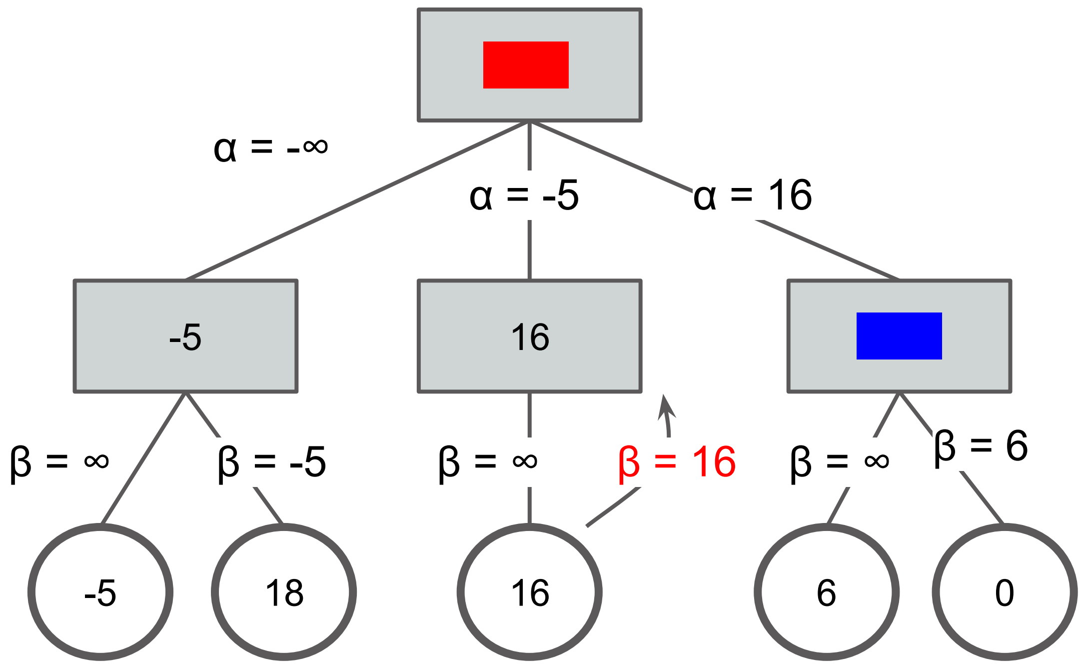
Alpha-Beta example

Perform alpha-beta pruning on the following tree



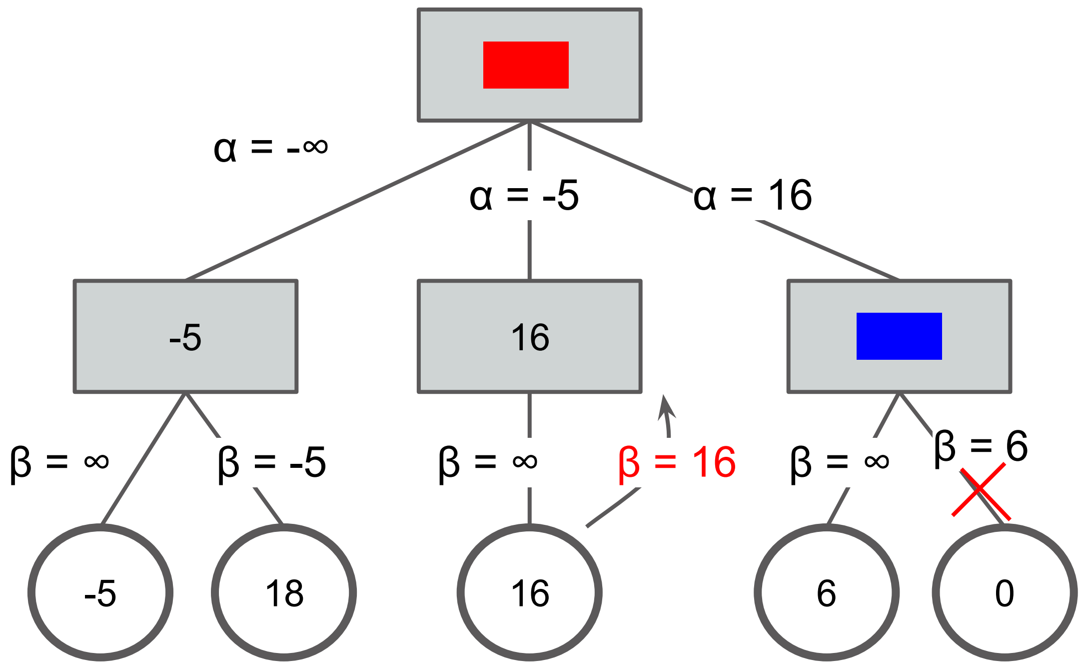
Alpha-Beta example

Perform alpha-beta pruning on the following tree



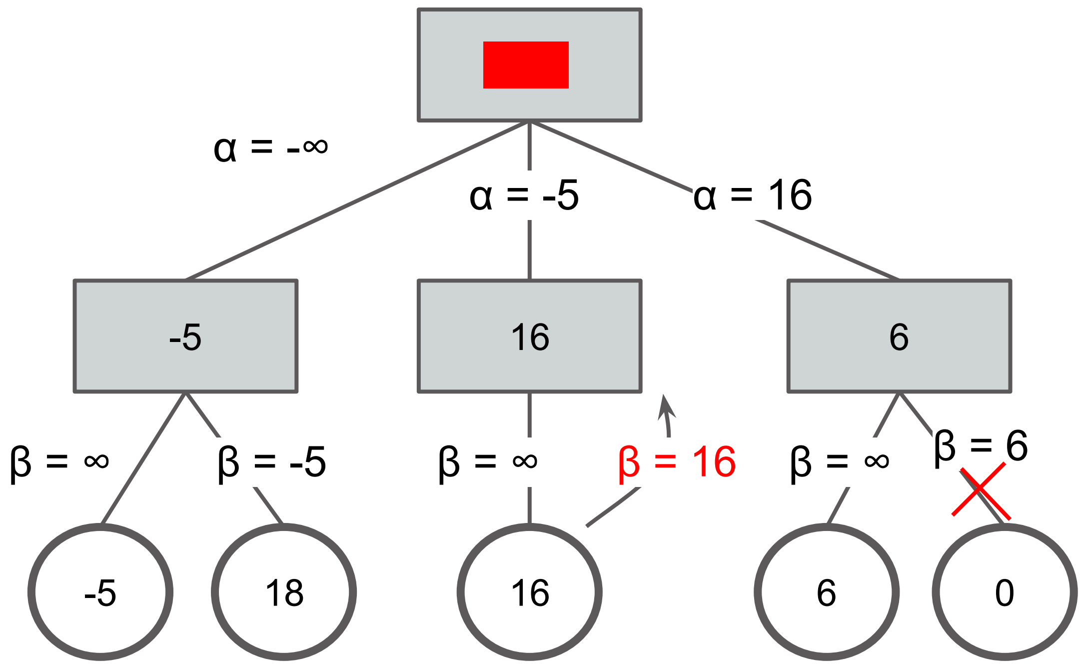
Alpha-Beta example

Perform alpha-beta pruning on the following tree



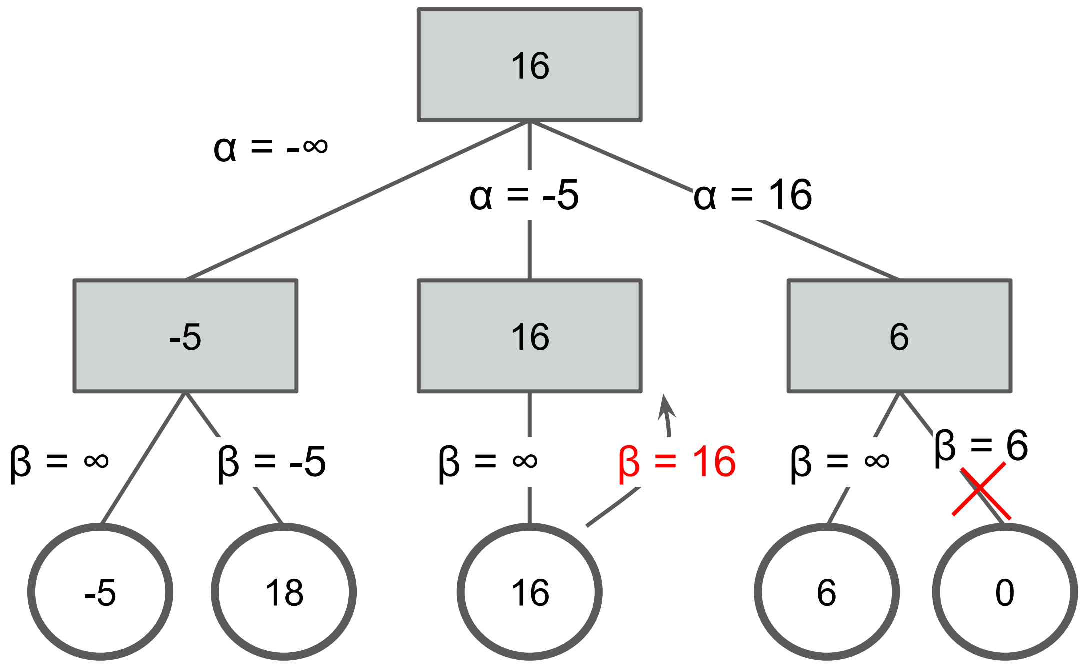
Alpha-Beta example

Perform alpha-beta pruning on the following tree



Alpha-Beta example

Perform alpha-beta pruning on the following tree



Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

   True -- try doing minimax on our example

2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

   True -- try doing minimax on our example

2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

   False -- try doing minimax on our example

3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

Alpha Beta Pruning

True or False:

1. After alpha-beta pruning, the root maximizer node will never have the wrong value.

   True -- try doing minimax on our example

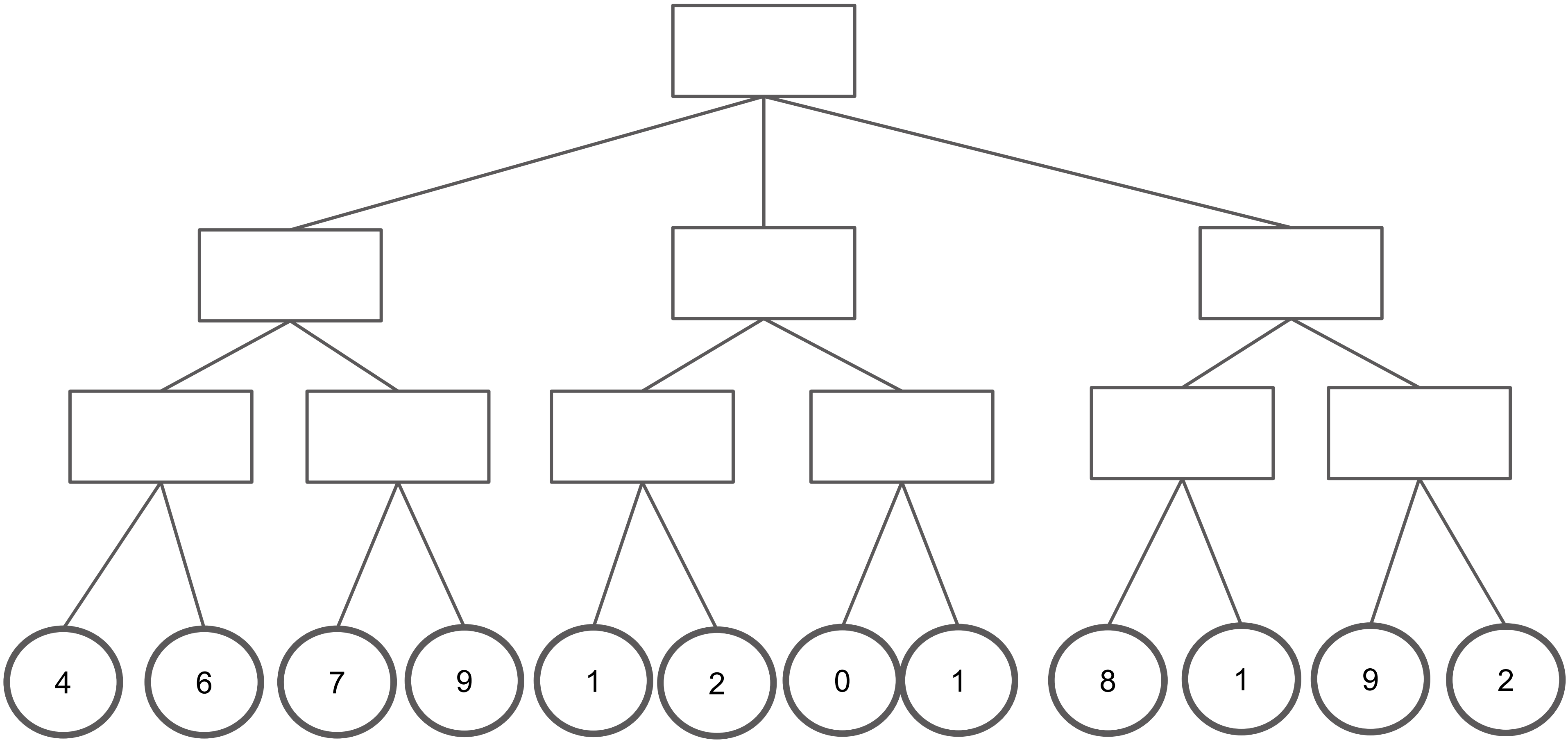
2. During alpha-beta pruning, none of the minimizer or maximizer intermediate nodes will differ from values found when using the normal minimax algorithm.

   False -- try doing minimax on our example

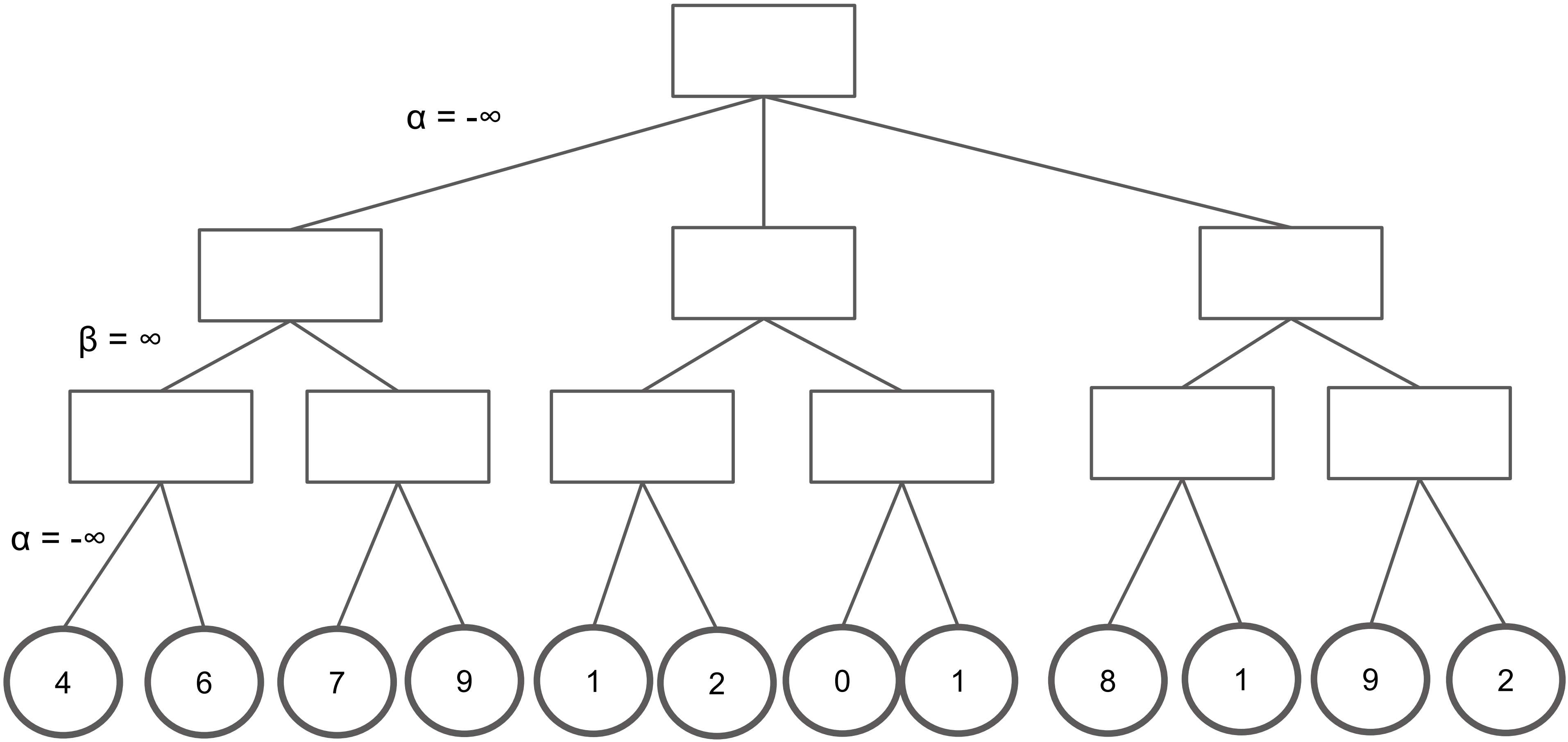
3. Alpha-beta pruning can have different prunings based on the order in which the algorithm traverses the tree.

   True -- what would have happened if we started with the right child first?

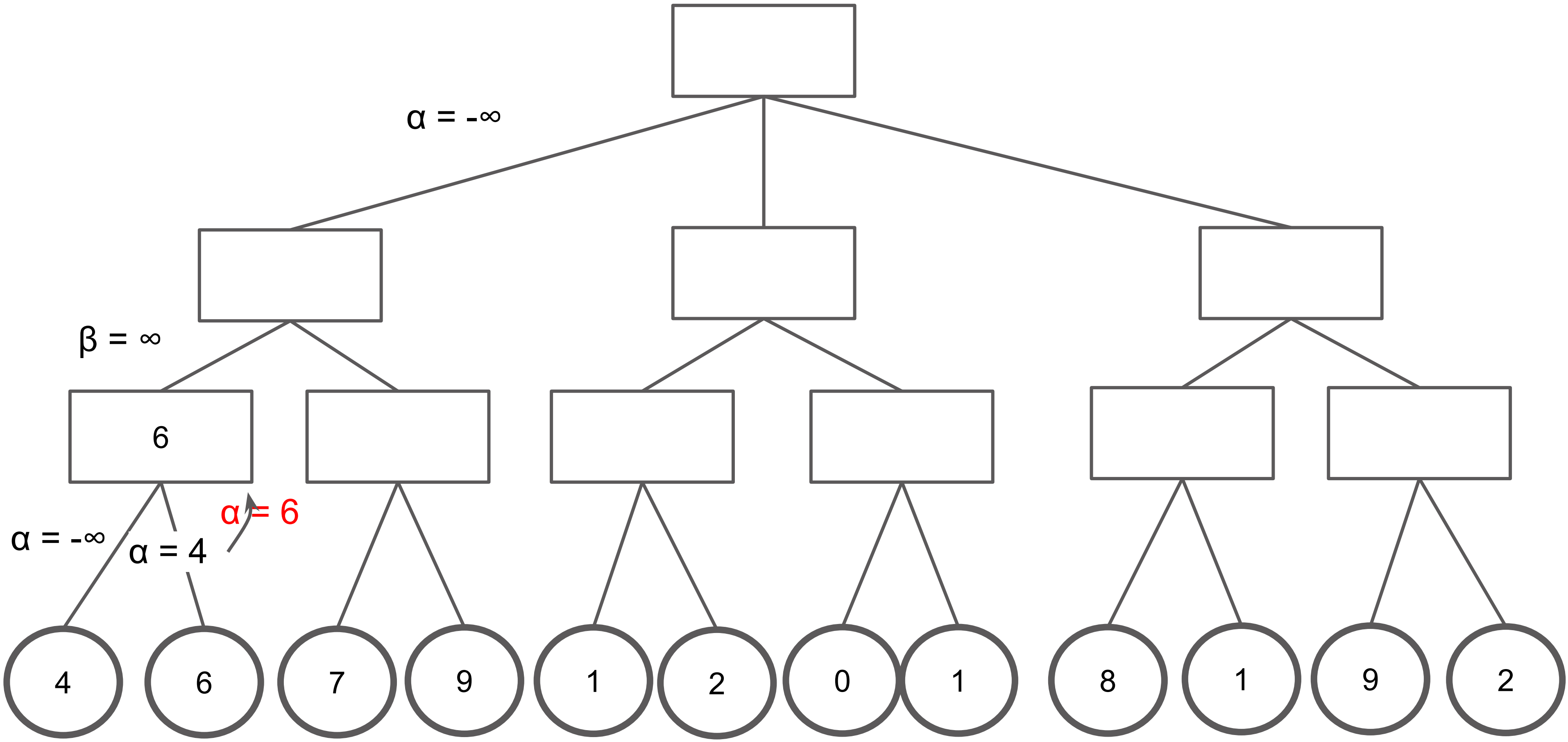
Alpha Beta Pruning



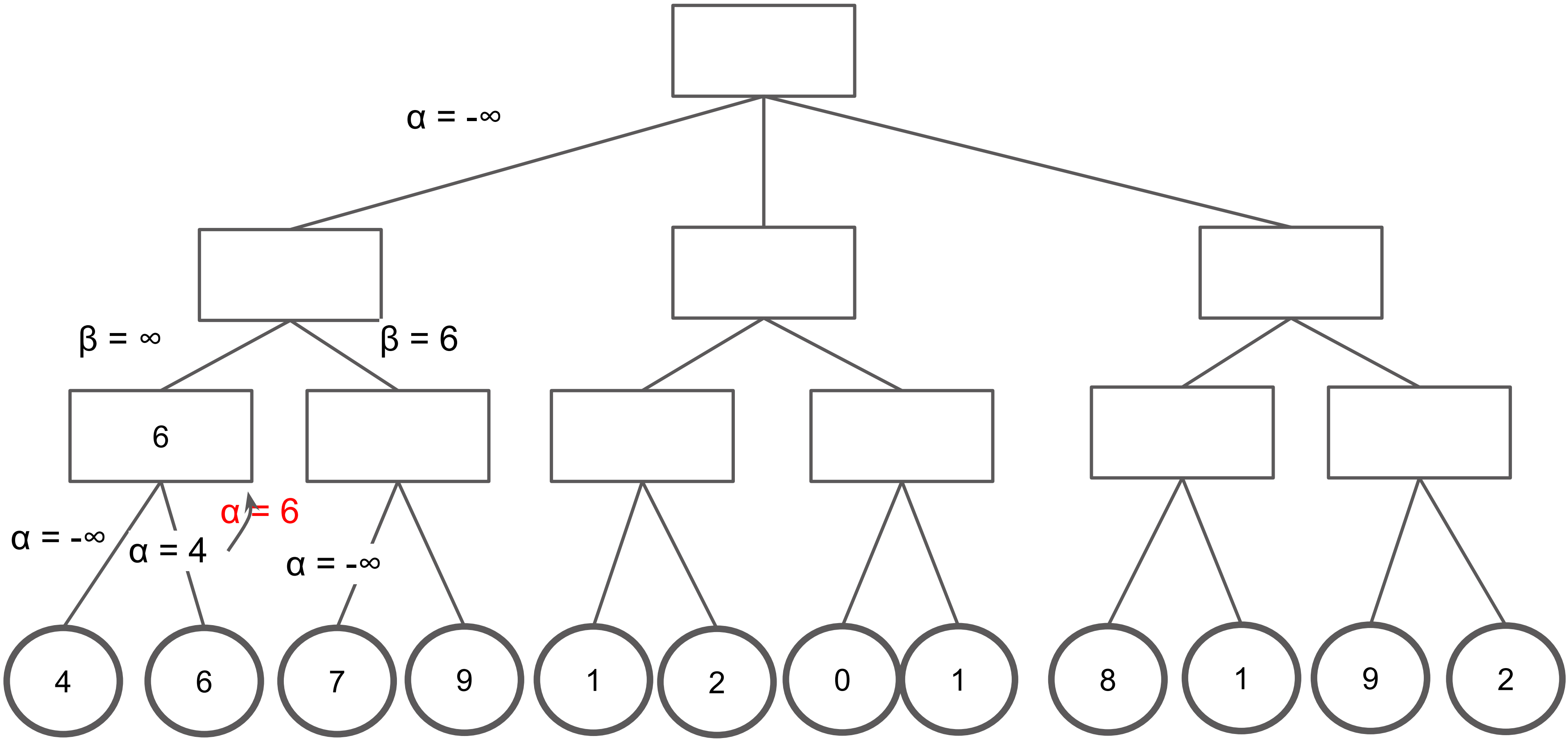
Alpha Beta Pruning



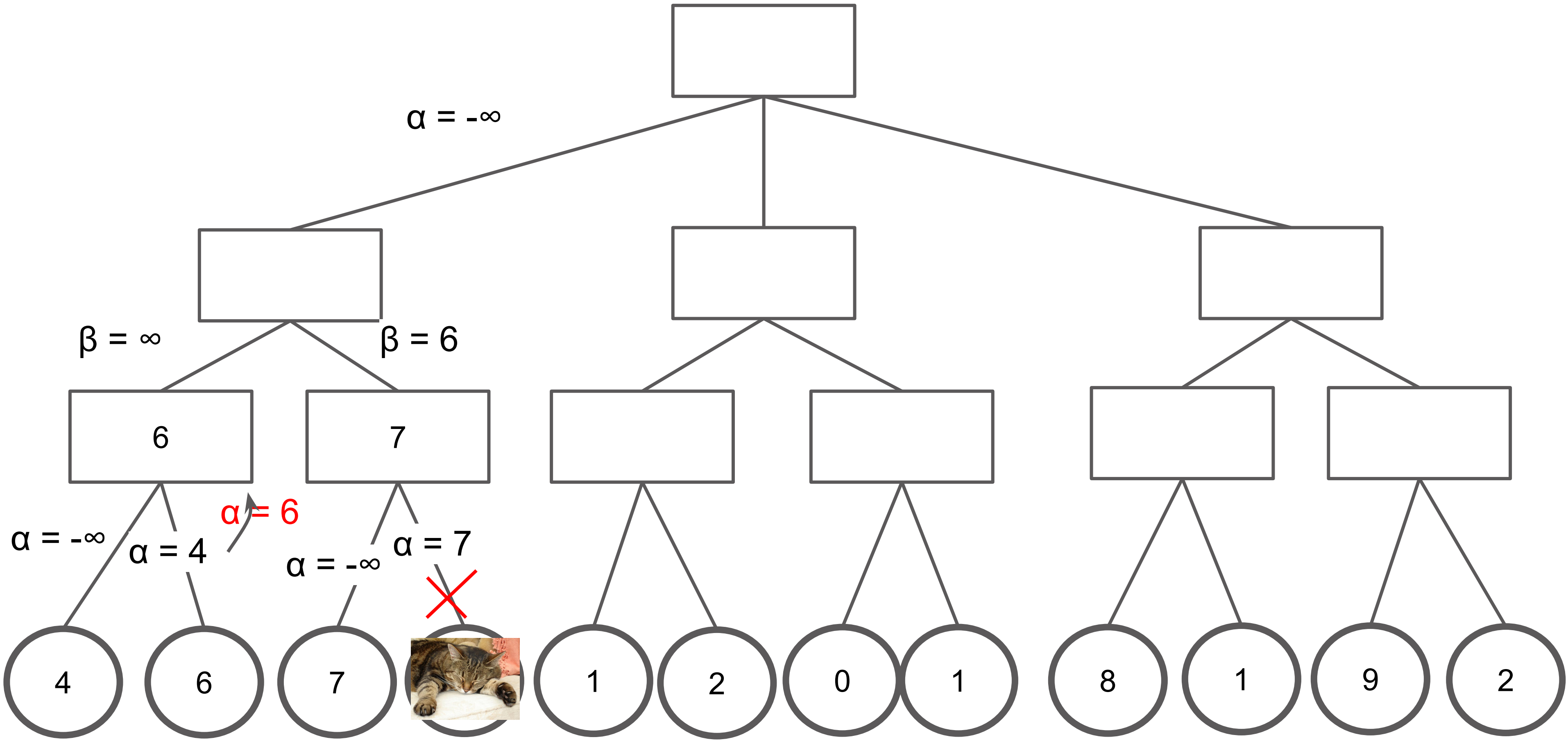
Alpha Beta Pruning



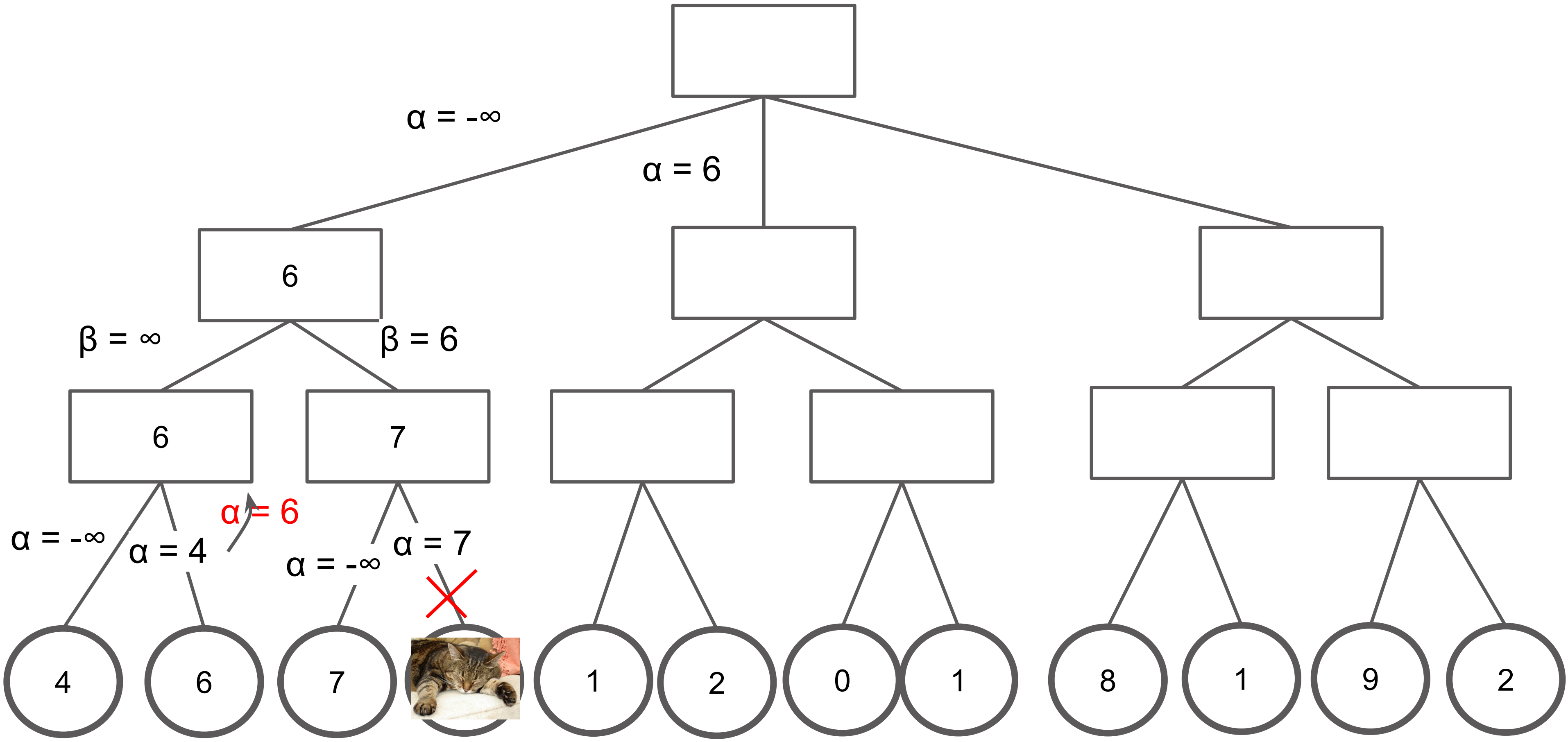
Alpha Beta Pruning



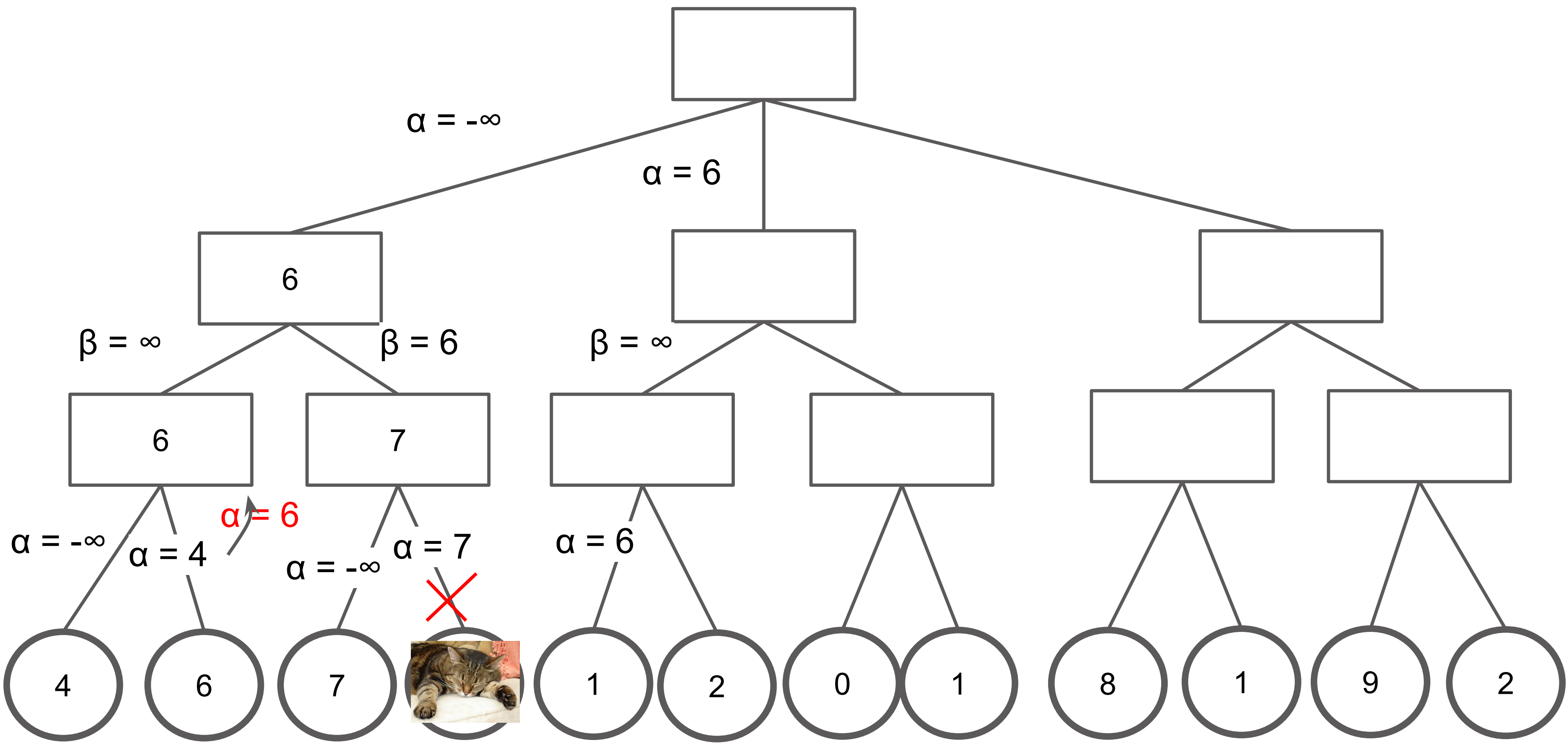
Alpha Beta Pruning



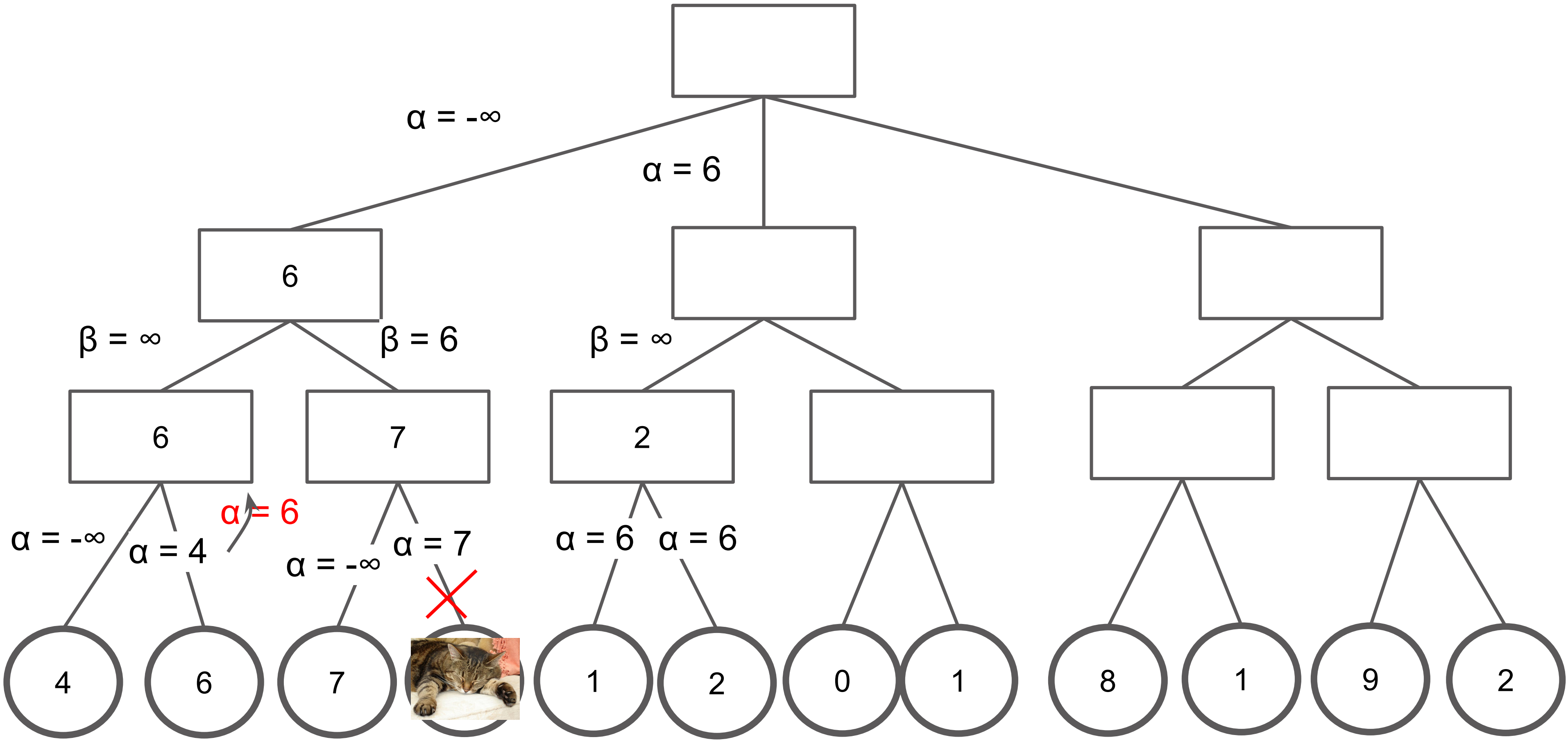
Alpha Beta Pruning



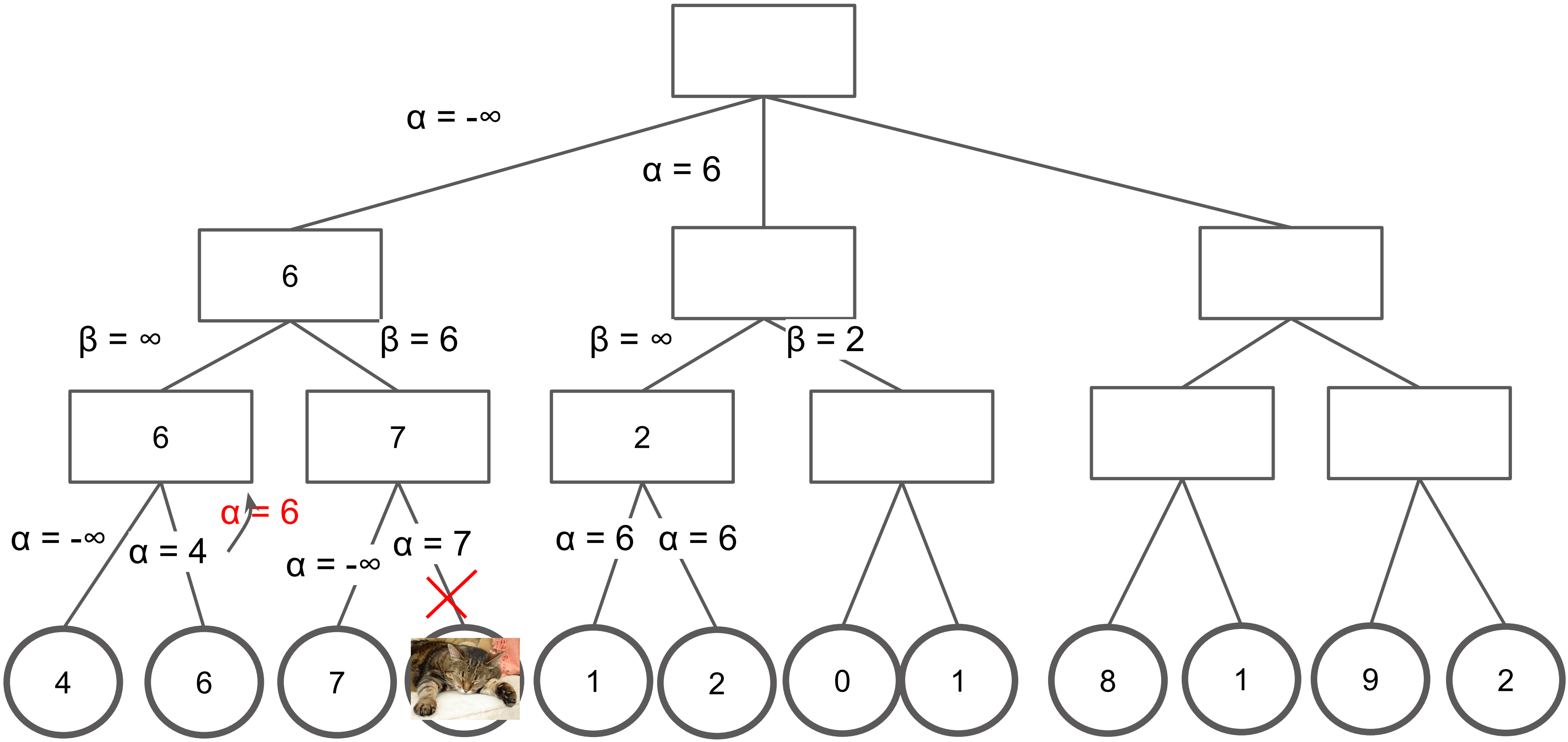
Alpha Beta Pruning



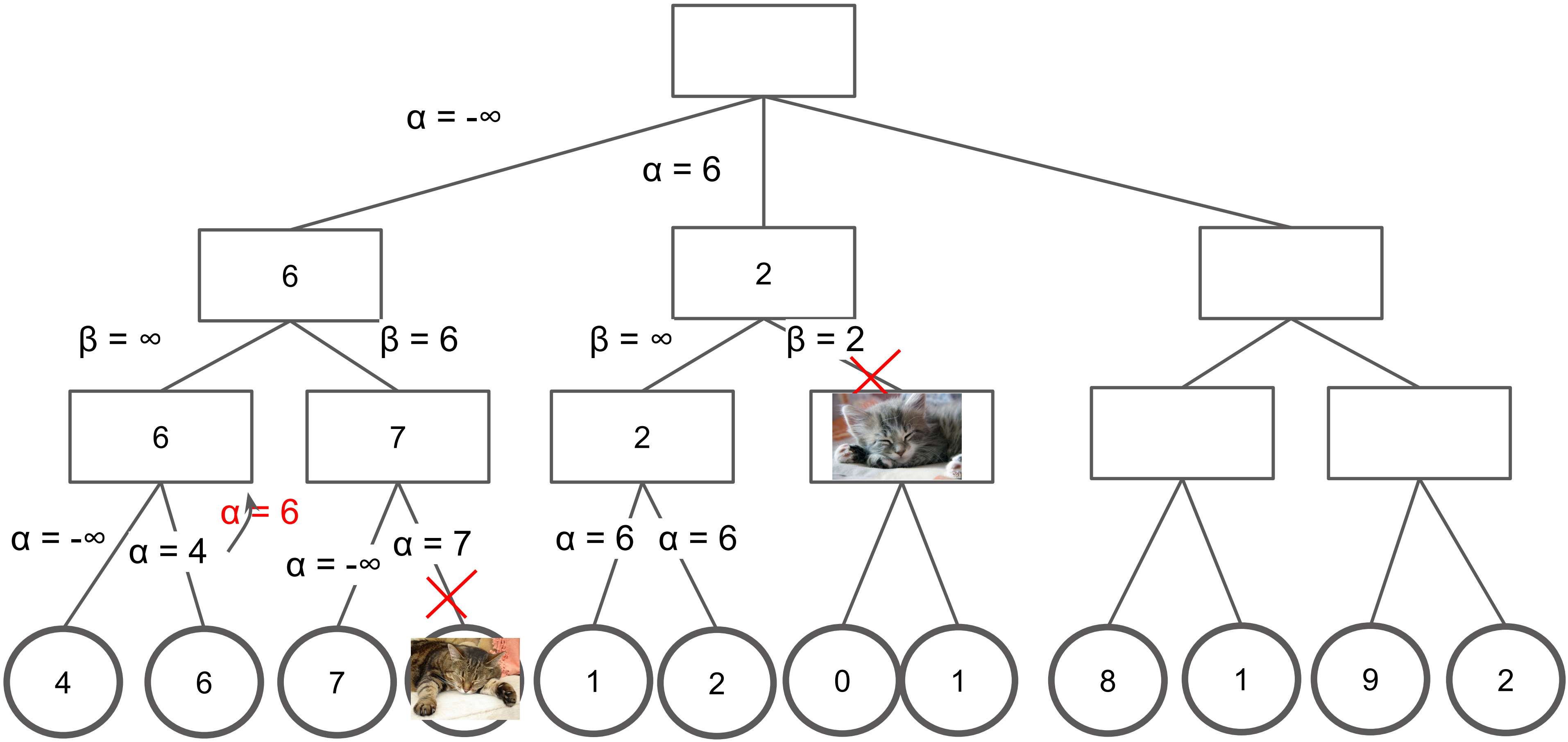
Alpha Beta Pruning



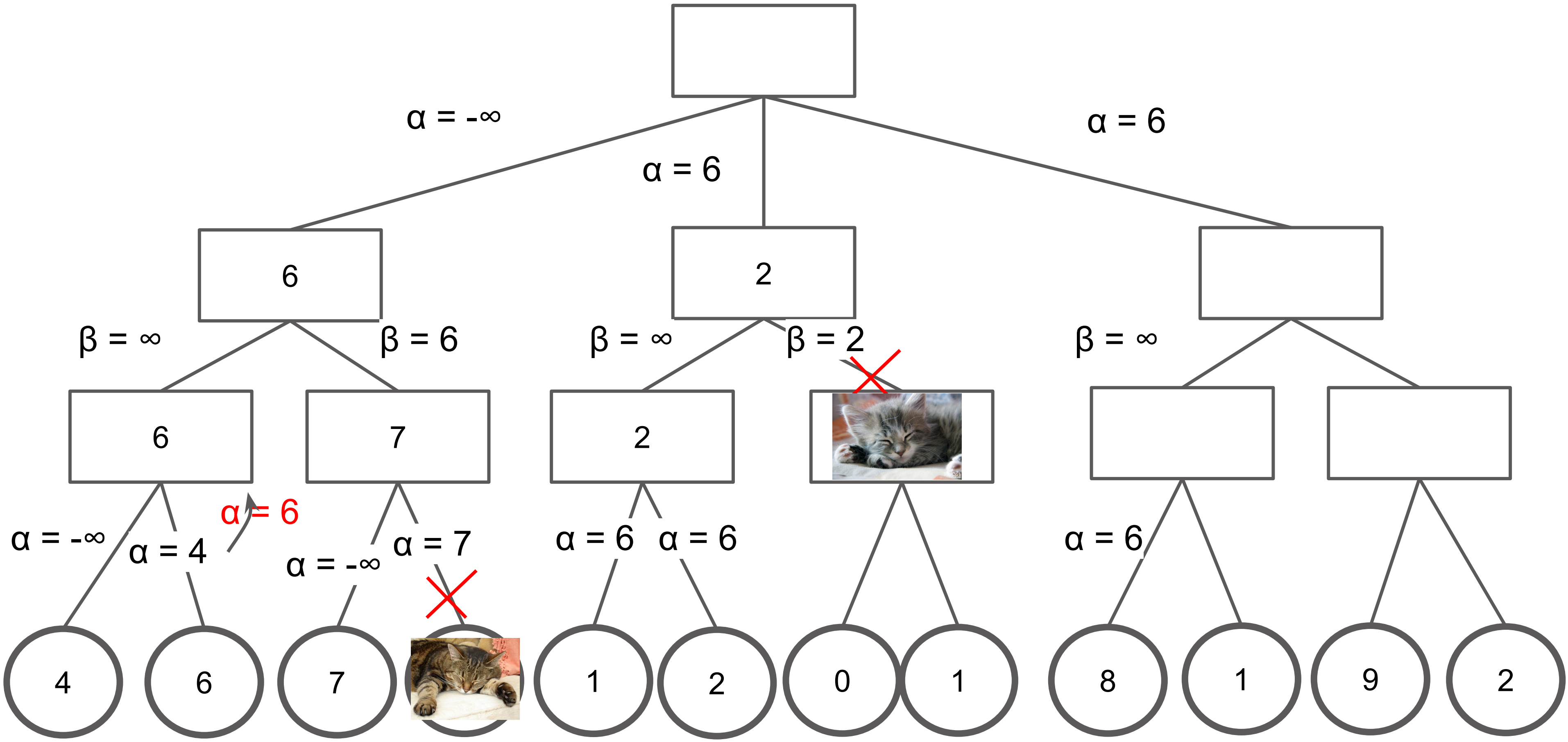
Alpha Beta Pruning



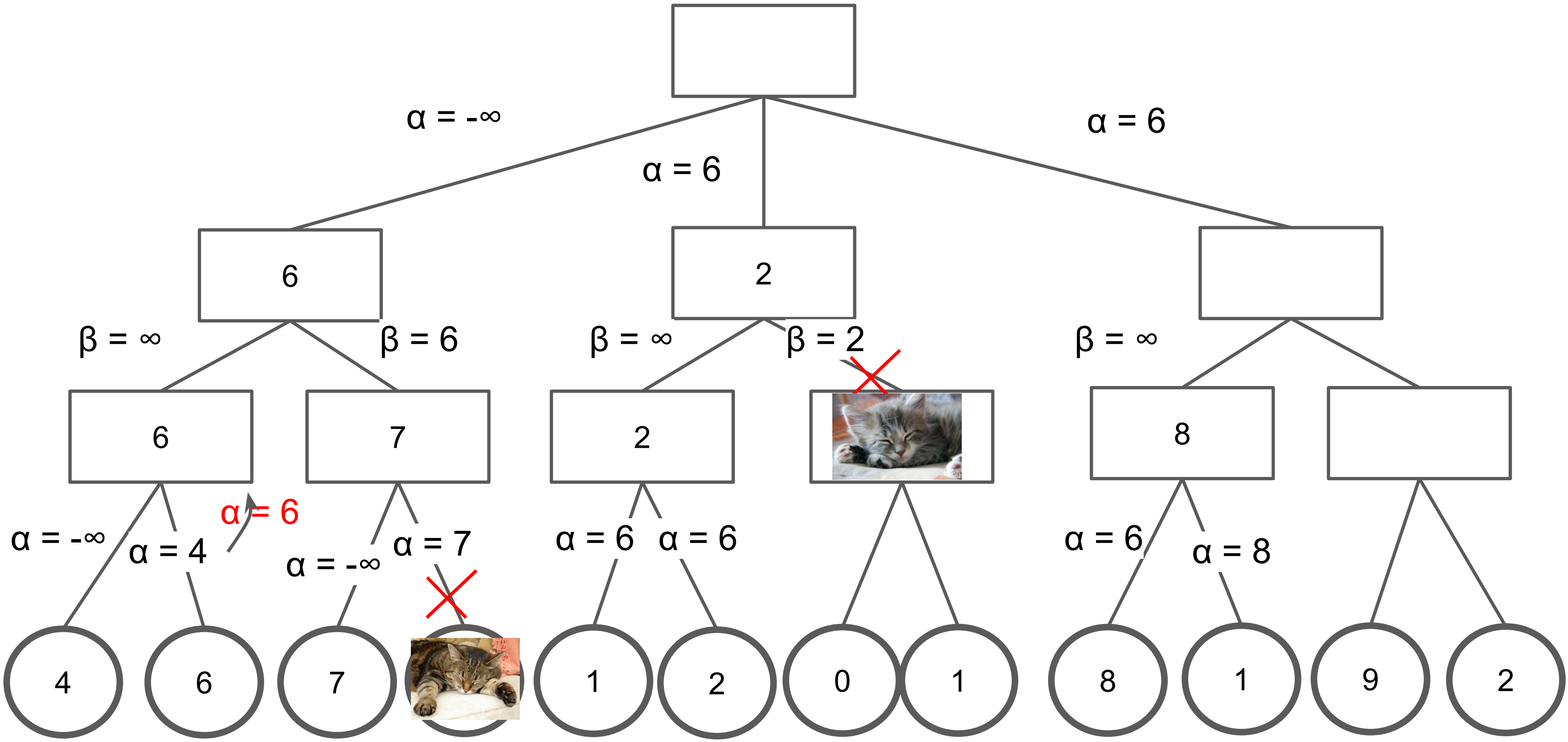
Alpha Beta Pruning



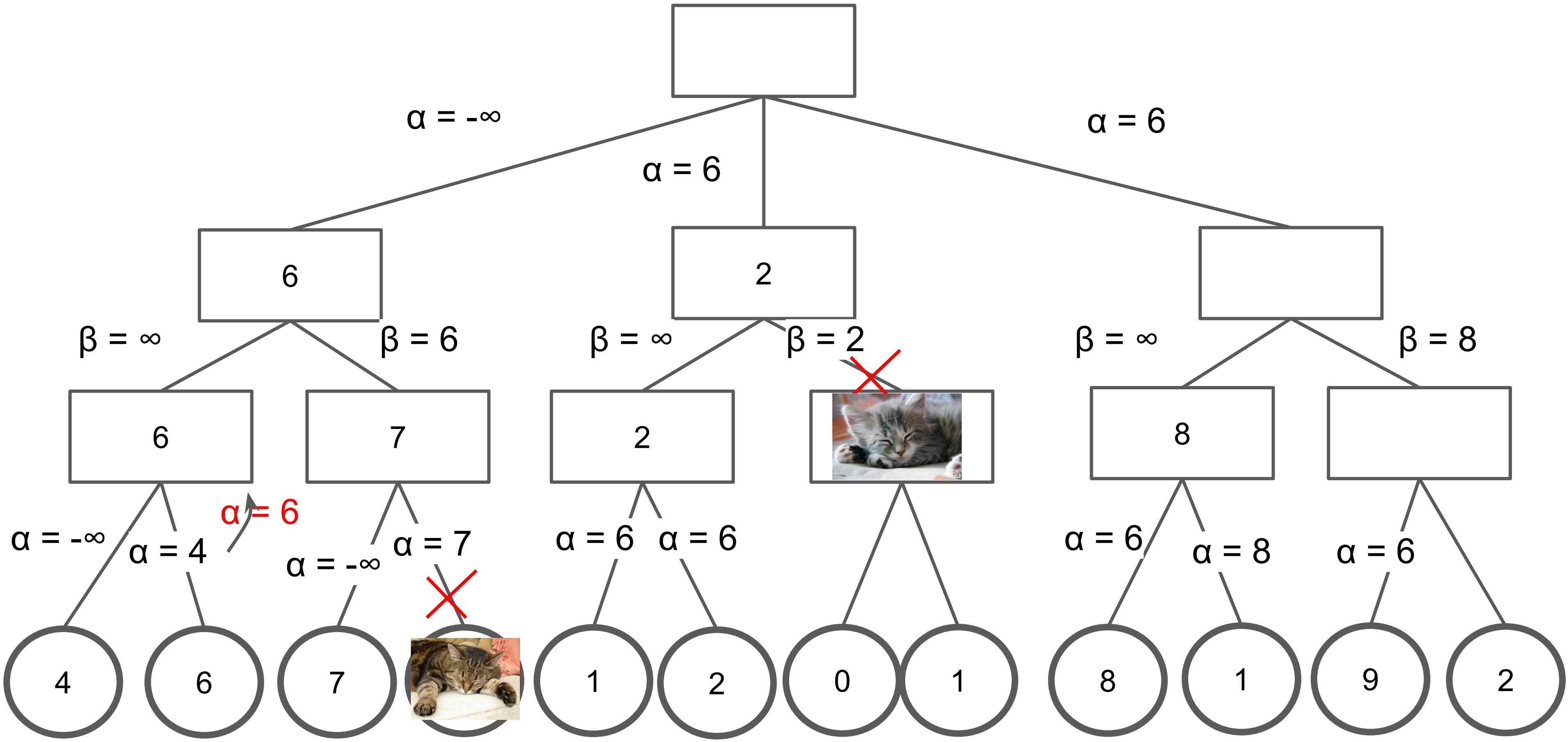
Alpha Beta Pruning



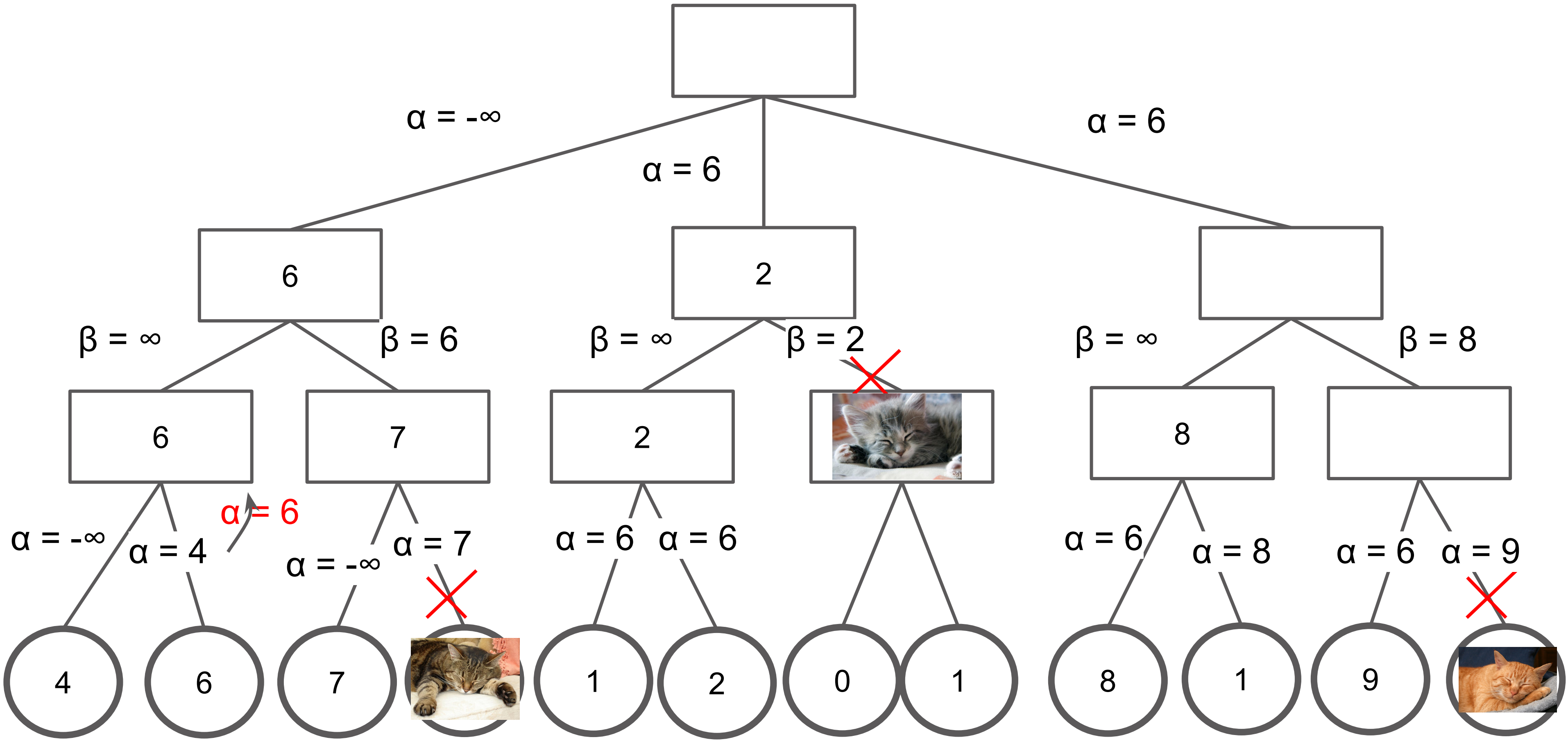
Alpha Beta Pruning



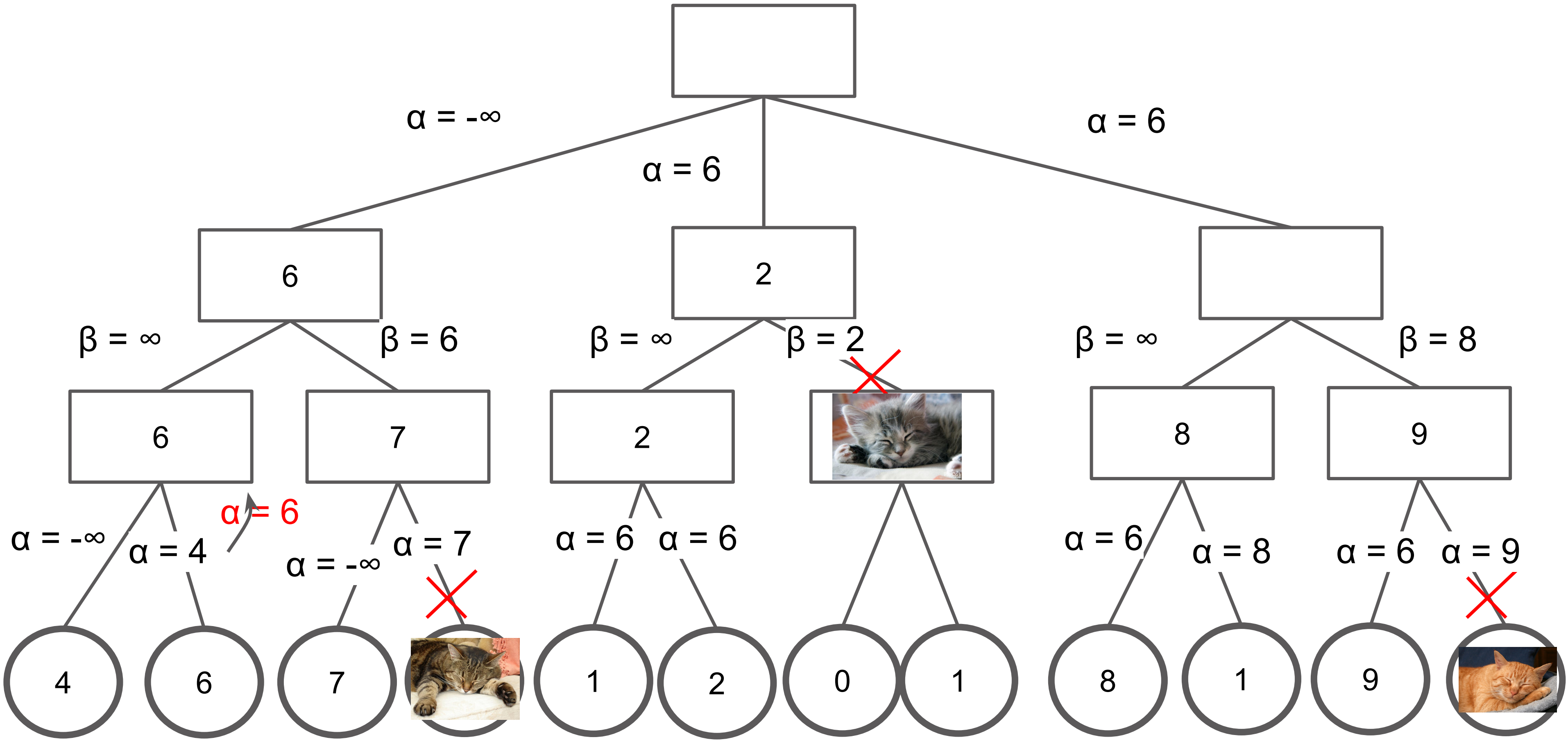
Alpha Beta Pruning



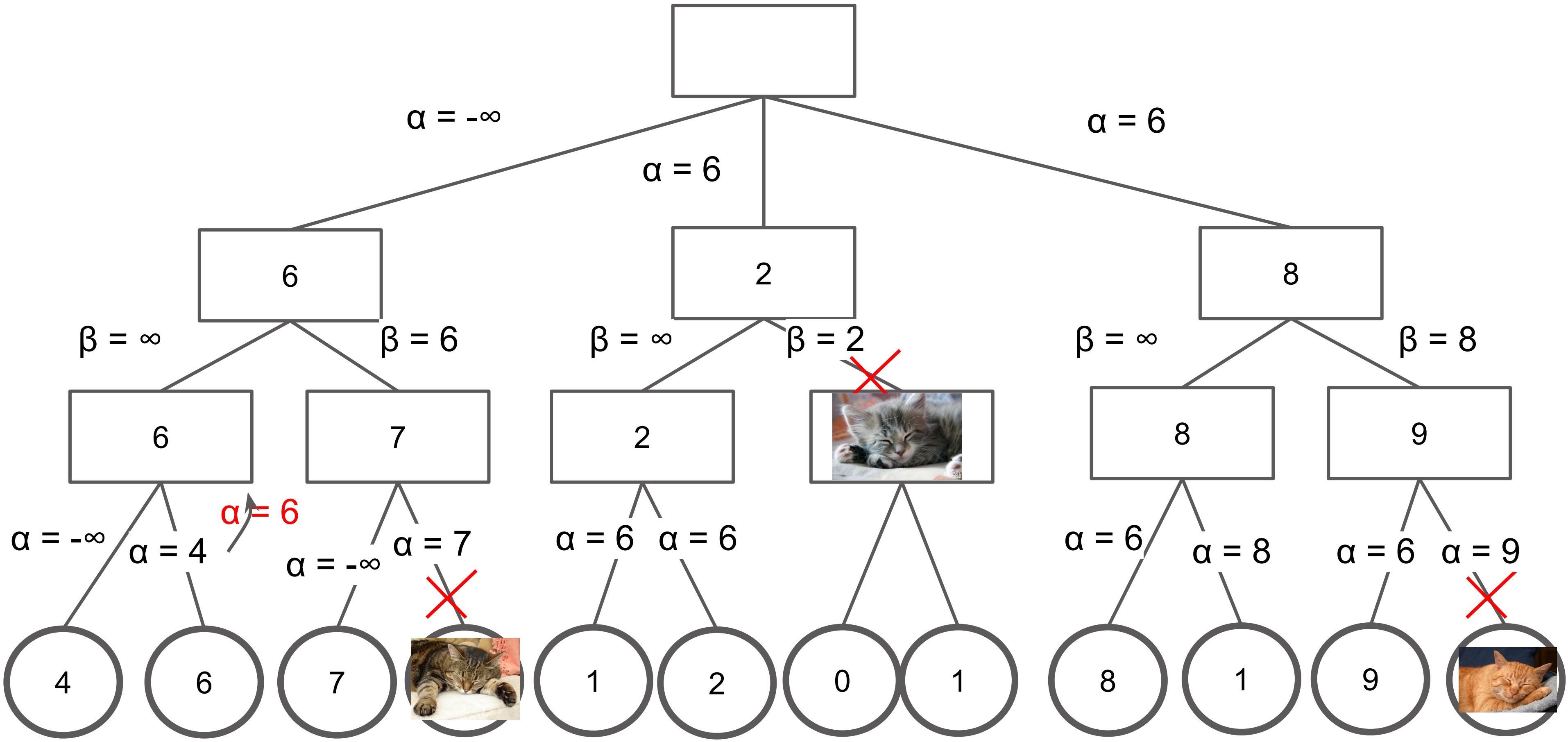
Alpha Beta Pruning



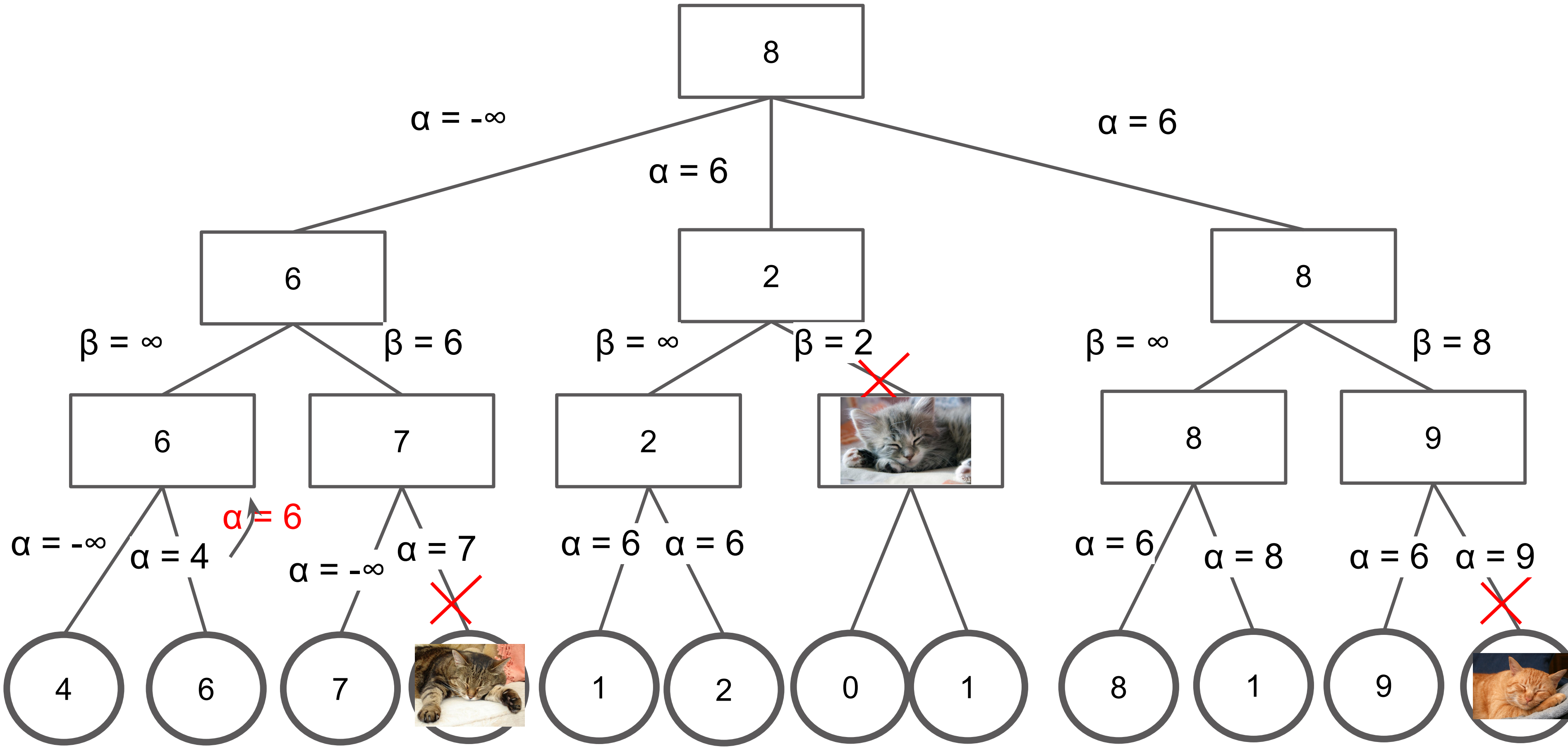
Alpha Beta Pruning



Alpha Beta Pruning



Alpha Beta Pruning



Heaps

Heaps

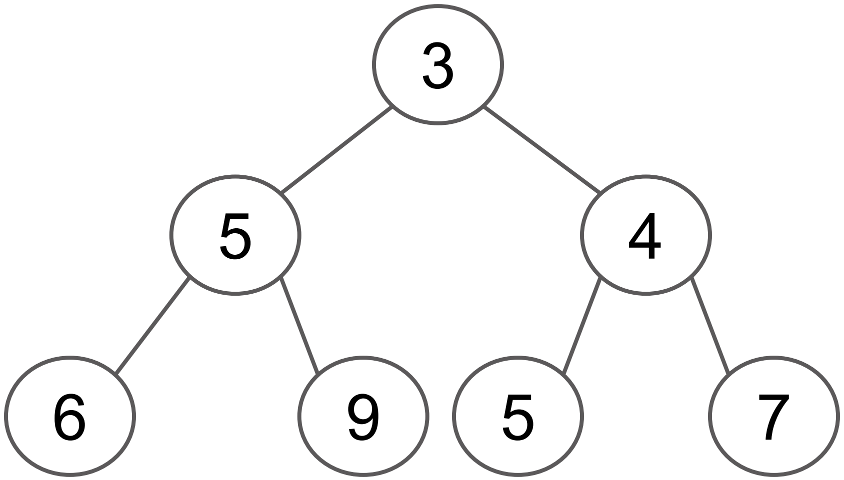
A heap is a binary tree with extra properties:

- The tree is complete
  - Recall: complete means that every level is filled, except possibly the last, which is filled left to right.

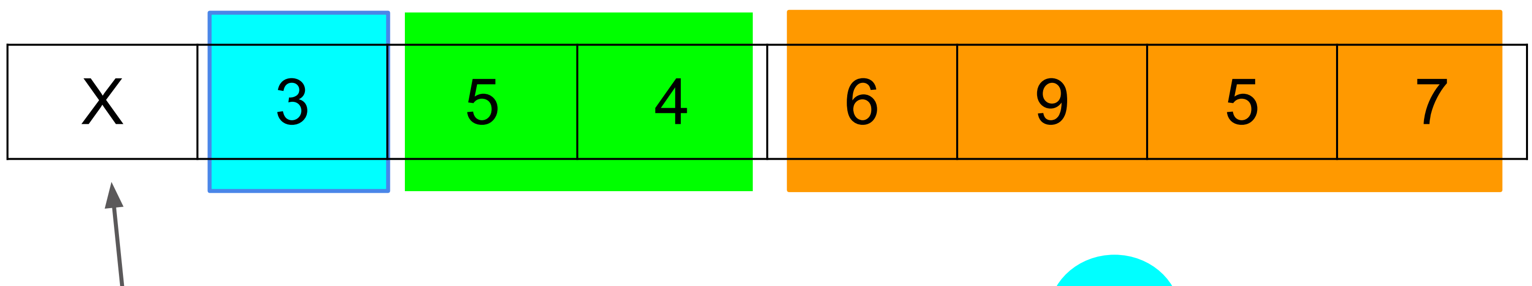
- Heap-order property: If node B is a descendant of node A:
  - **Min** heap: key of B >= key of A
  - **Max** heap: key of B <= key of A

Heaps

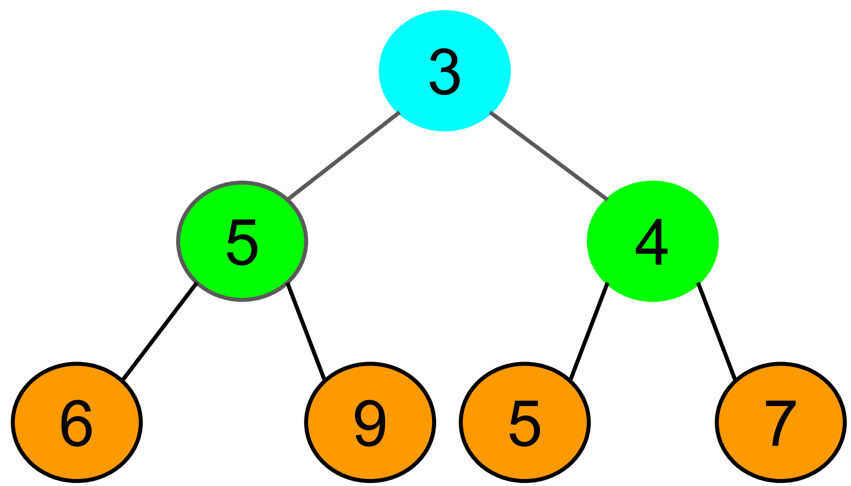
X	3	5	4	6	9	5	7



Heaps



Why do we X out the zero-th index?



Heaps, Array Representation

You are at a node with index **i**

Parent is at **floor(i/2)**

Children are at indices **2i** and **2i+1**

Ex: Key at index 3.

Parent: index 1.

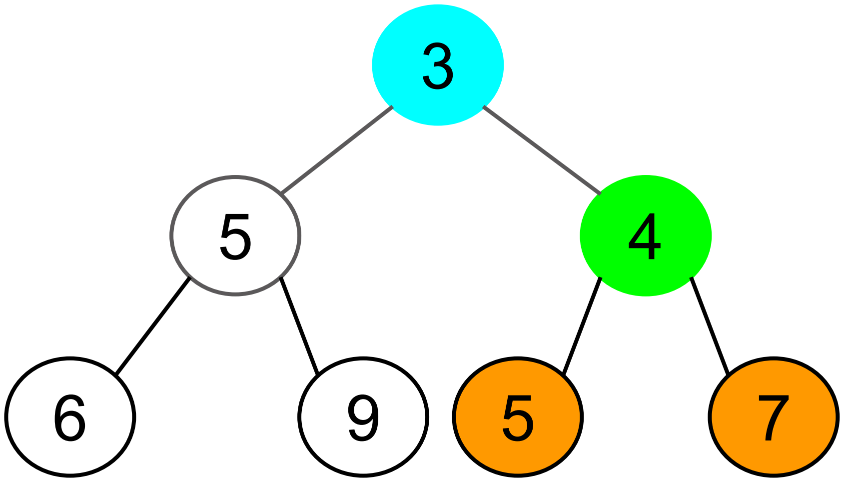
Children: indices 6 and 7.

Heaps

0	1	2	3	4	5	6	7
X	3	5	4	6	9	5	7

Let i = 3

parent: floor(i/2) = 1
children: 2i, 2i+1 = 6, 7



Heaps, a possibly tricksy question

True/False:

In a min heap:

Key $\mathbf{k_1}$ is at level $\mathbf{l_1}$ and key $\mathbf{k_2}$ is at level $\mathbf{l_2}$.

If $\mathbf{k_1 < k_2}$, then $\mathbf{l_1 <= l_2}$?

What if instead, we are given $\mathbf{l_1 < l_2}$.

Is $\mathbf{k_1 <= k_2}$?

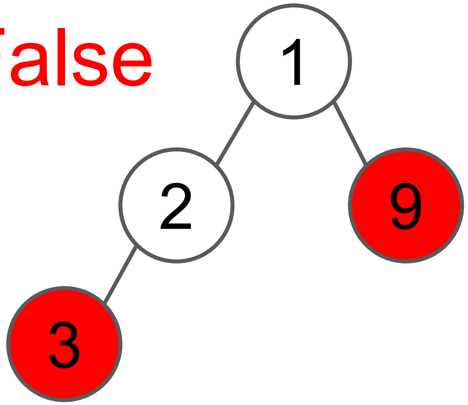
Heaps, a possibly tricksy question

True/False: In a min heap:

Key k_1 is at level l_1 and key k_2 at level l_2.
If $k_1 < k_2$, then $l_1 <= l_2$?
Or if $l_1 < l_2$, does $k_1 <= k_2$?

Ans: Both False

Ex:



k_1: 3
k_2: 9
l_1: 3
l_2: 2

Heaps, insert and removeMin

To `insert()` in a heap:

- Insert item at the end of array.
- Bubble up item by repeatedly swapping with parents until heap-order property is satisfied.

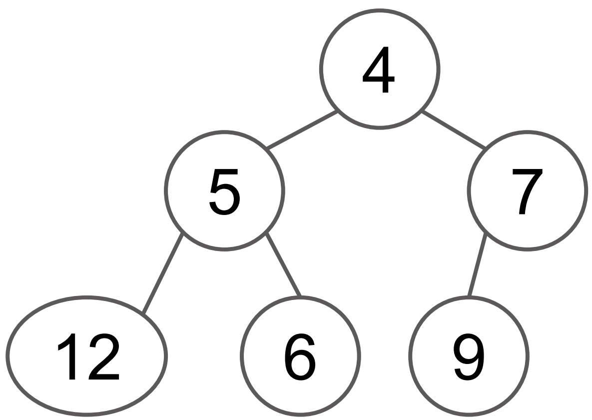
To `removeMin()`:

- Replace first element (root) with last element, and remove last node.
- Bubble down new root by repeatedly swapping with smaller of two children until heap-order property is satisfied.

Inserting into Heaps

Let's insert **-1**

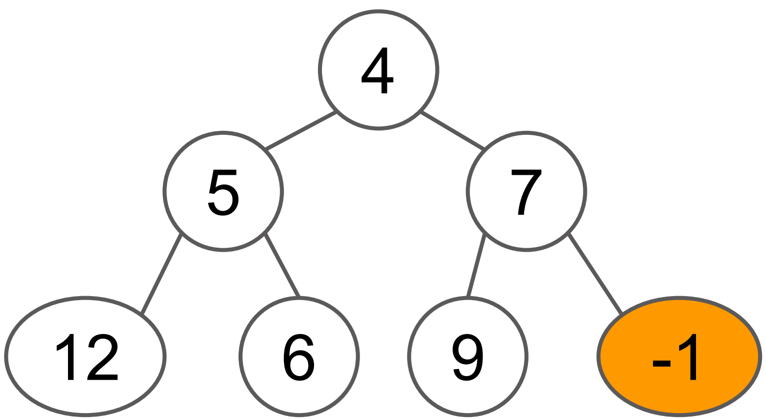
X	4	5	7	12	6	9	



Inserting into Heaps

put it at the beginning

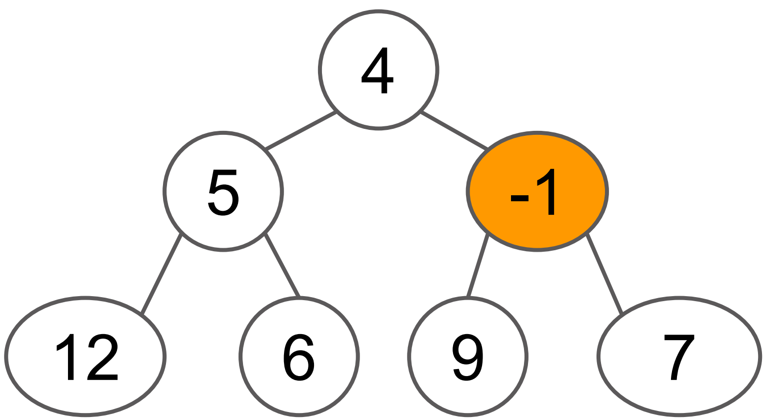
X	4	5	7	12	6	9	-1



Inserting into Heaps

bubble up...

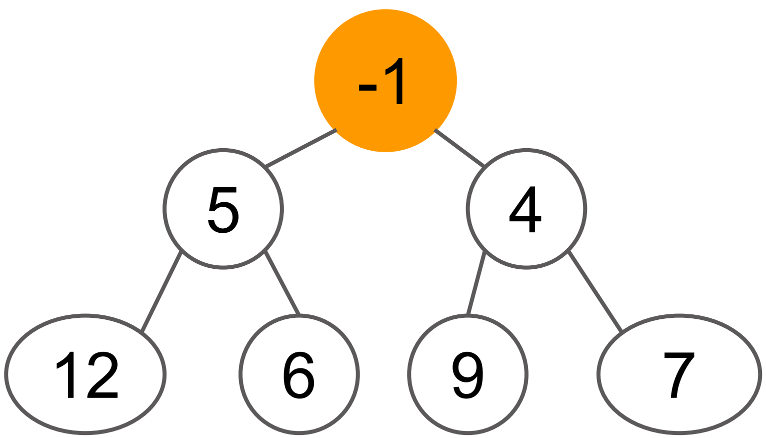
X	4	5	-1	12	6	9	7



Inserting into Heaps

bubble up.. and done!

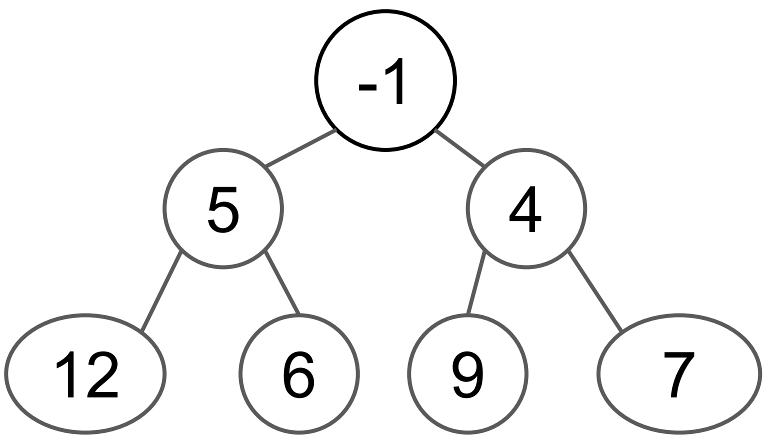
X	-1	5	4	12	6	9	7



Removing from Heaps

(a) `removeMin()`

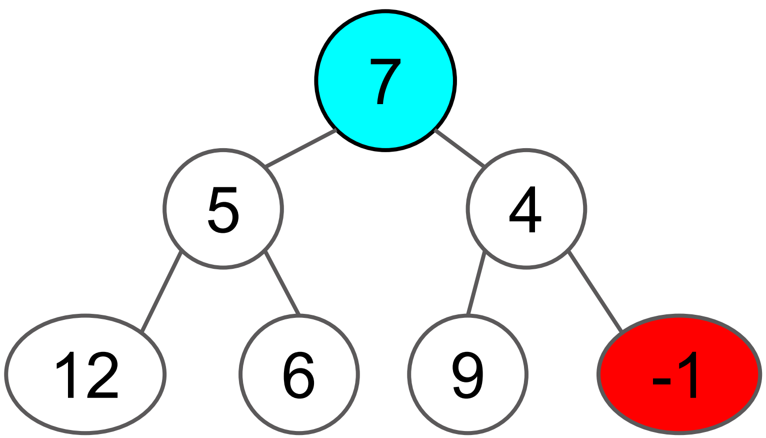
X	-1	5	4	12	6	9	7



Removing from Heaps

Switch first and last elements

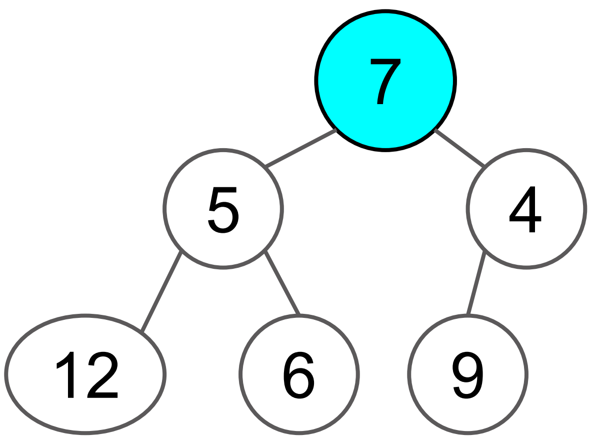
X	7	5	4	12	6	9	-1



Removing from Heaps

pop the last element

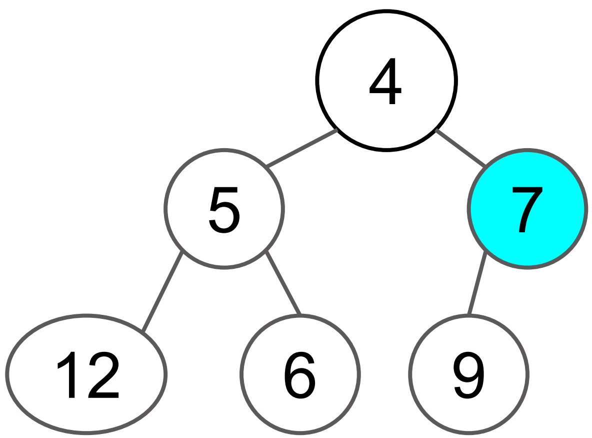
X	7	5	4	12	6	9	



Removing from Heaps

bubble down… and done!

X	4	5	7	12	6	9	



How do we know which way to bubble down?

Heap Complexities

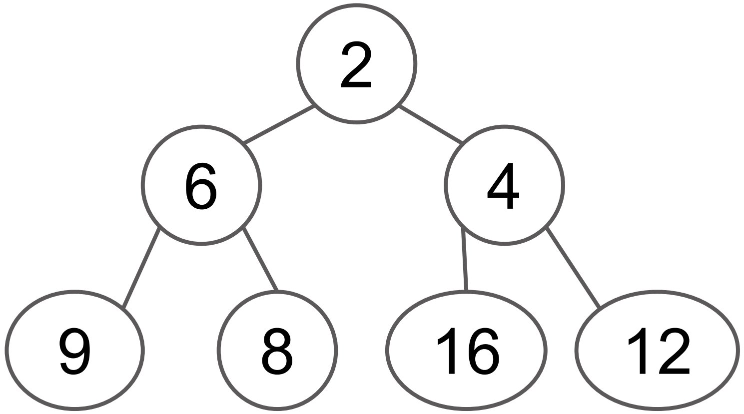
Running Times			
	Binary Heap	Sorted List/Array	Unsorted List/Array
min()	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
insert() (worst case)	$\Theta(logn)^*$	$\Theta(n)$	$\Theta(1)^*$
insert() (best case)	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$
removeMin() (worst case)	$\Theta(logn)$	$\Theta(1)$	$\Theta(n)$
removeMin() (best case)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

* If you are using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take Θ(n) time to allocate a larger array and copy the entries into it. However, if you double the array size each time, the average running time will still be as indicated.

Heap Practice

Given the following min heap, draw what it looks like after each following consecutive method calls:

- `insert(10)`
- `removeMin()`
- `insert(3)`
- `removeMin()`



Heap Practice

After insert(10)

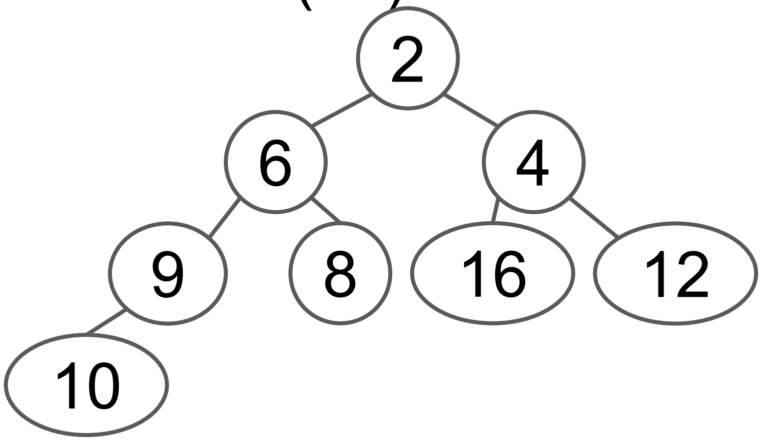
After insert(3)

After removeMin()

After removeMin()

Heap Practice

After insert(10)



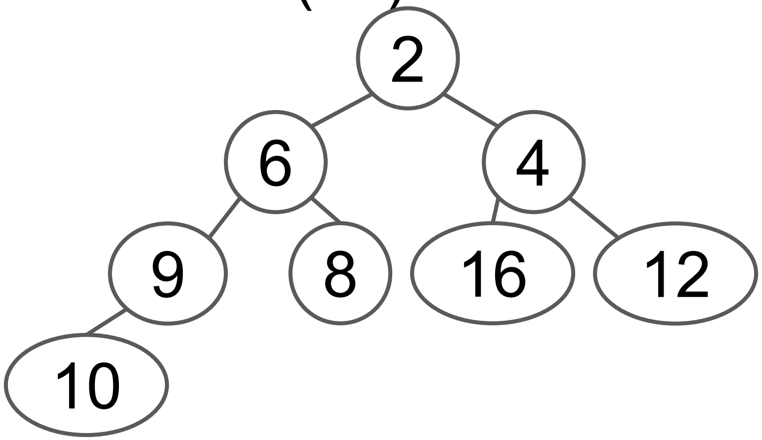
After insert(3)

After removeMin()

After removeMin()

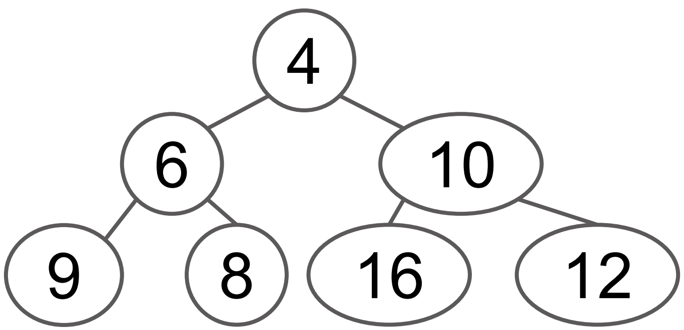
Heap Practice

After insert(10)



After insert(3)

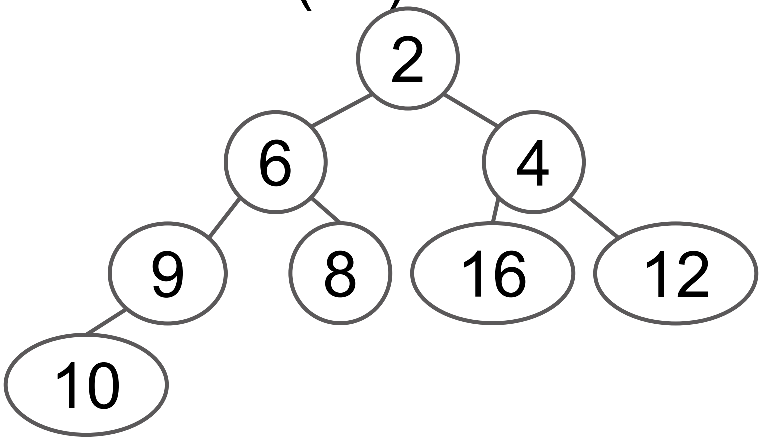
After removeMin()



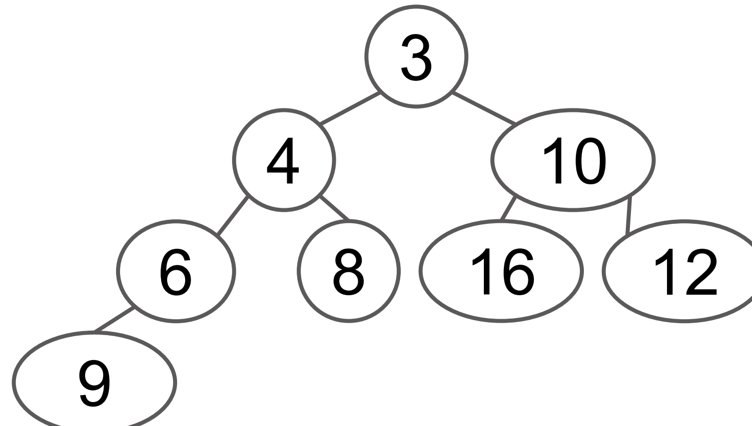
After removeMin()

Heap Practice

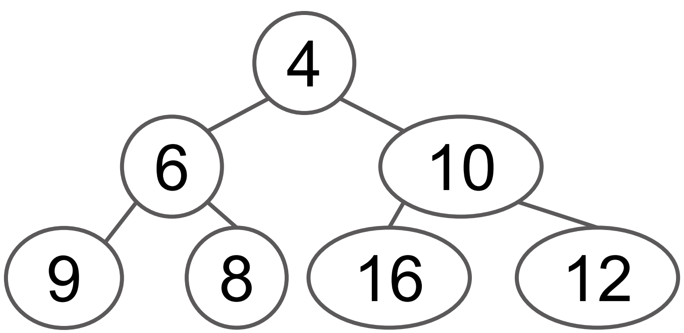
After insert(10)



After insert(3)



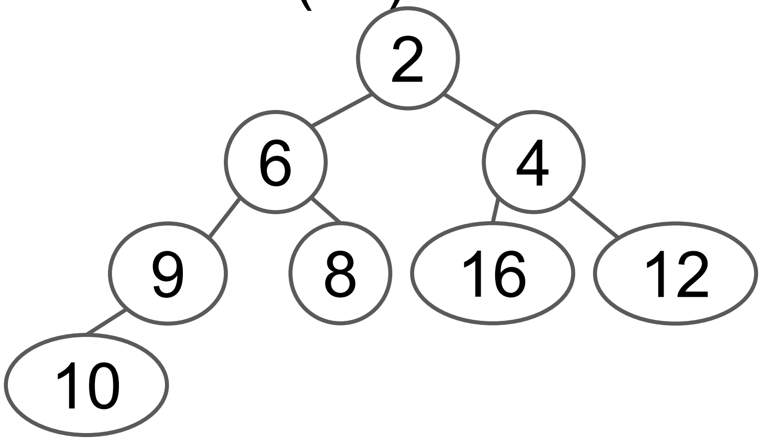
After removeMin()



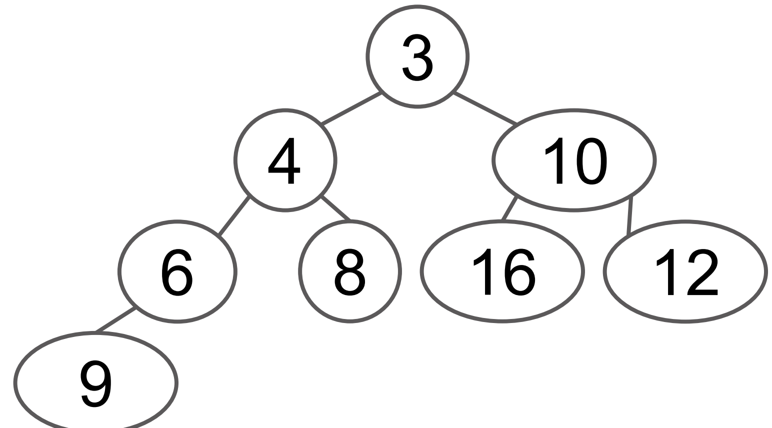
After removeMin()

Heap Practice

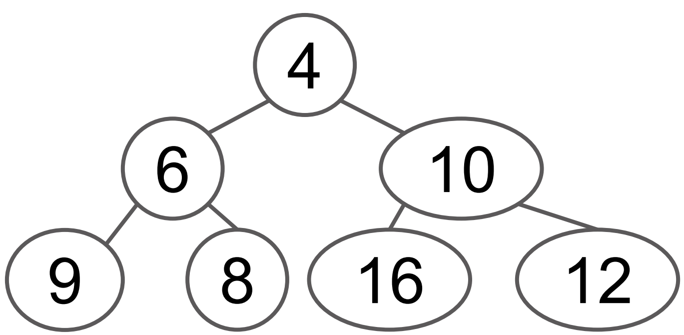
After insert(10)



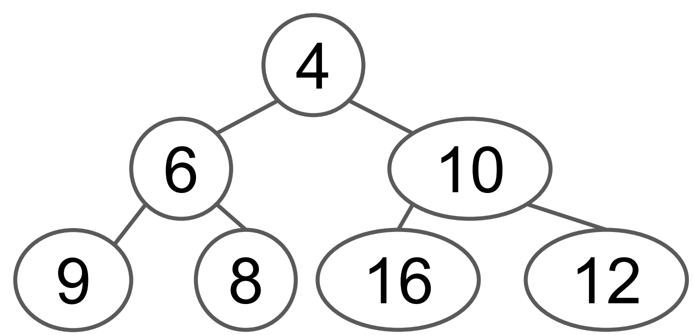
After insert(3)



After removeMin()



After removeMin()

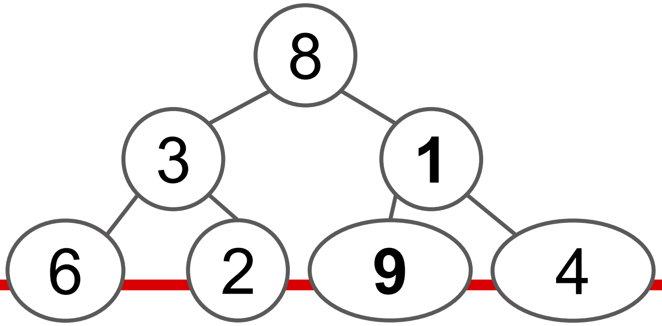


Heaps, bottom up

Perform bottomUpHeap on the following array, redrawing the array after every swap.

X	8	3	9	6	2	1	4

Heaps

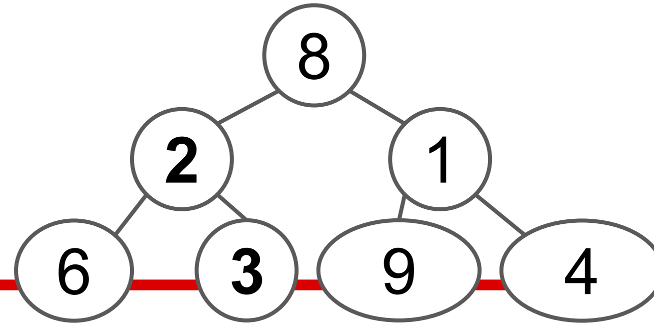


X	8	3	9	6	2	1	4

Ans:

X	8	3	**1**	6	2	**9**	4

Heaps



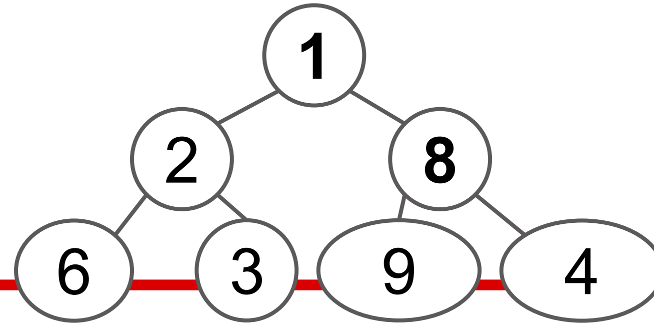
X	8	3	9	6	2	1	4

Ans:

X	8	3	**1**	6	2	**9**	4

X	8	**2**	1	6	**3**	9	4

Heaps



X	8	3	9	6	2	1	4

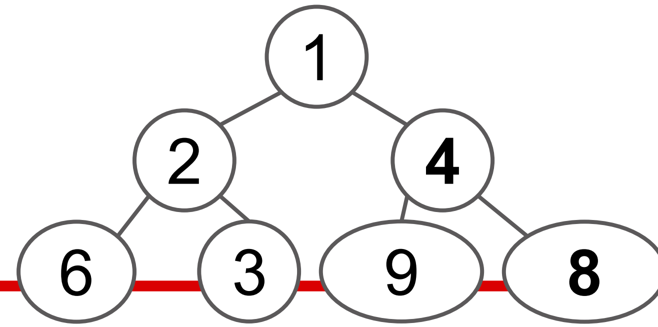
Ans:

X	8	3	**1**	6	2	**9**	4

X	8	**2**	1	6	**3**	9	4

X	**1**	2	**8**	6	3	9	4

Heaps



X	8	3	9	6	2	1	4

Ans:

X	8	3	**1**	6	2	**9**	4

X	8	**2**	1	6	**3**	9	4

X	**1**	2	**8**	6	3	9	4

X	1	2	**4**	6	3	9	**8**

Interview Question of the Day

Write a function **int[] kLargest(int[] a, int k)** that takes an **unsorted integer array** of size **N** and finds the **k** largest elements.

You also know that **k << N**

Interview Answer of the Day

Ans:

1. Build a **min** heap of size **k** using the first k elements.
2. For each element **x** of **a**:
   a. Compare heap.peekMin() and **x**
   b. If x is larger, heap.removeMin(), and heap.insert(x)
3. Output the elements of the heap

Running time: **O(nlogk)**

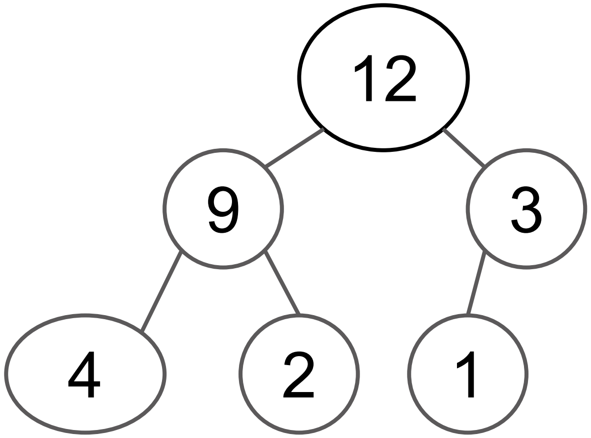
Heaps, a silly answer

```
int[] kLargest(int[] a, int k){
  Heap bestOfTheBest = new MinHeap();
  for(int i = 0; i < k; i++)
    bestOfTheBest.insert(a[i]);

  for(int i = k; i < a.length; i++){
    if(bestOfTheBest.peekMin() < a[i]){
      bestOfTheBest.removeMin();
      bestOfTheBest.insert(a[i]);
    }
  }

  int[] largest = new int[k];
  for(int i = 0; i < k; i++){
    largest[i] = bestOfTheBest.removeMin();
  }
  return largest;
}
```

Reminder: Max Heaps

Same as before, but now the root is bigger than its two children

X	12	9	3	4	2	1	



Heaps, a better question

Design an efficient data structure that supports the following method calls:

- void insert(int n)
- int getMedian()

Heaps, a better question

Example input = {42, 0, 100}

>>> insert(42);

>>> getMedian();

42

>>> insert(0);

>>> getMedian();

21

>>> insert(100); getMedian();

42

For heaps, you are given the following methods:

- int size()
- int peakMin() or peakMax()
- int removeMin() or removeMax()
- int insert(int num)

Note: an **O(n)** insert is too slow!

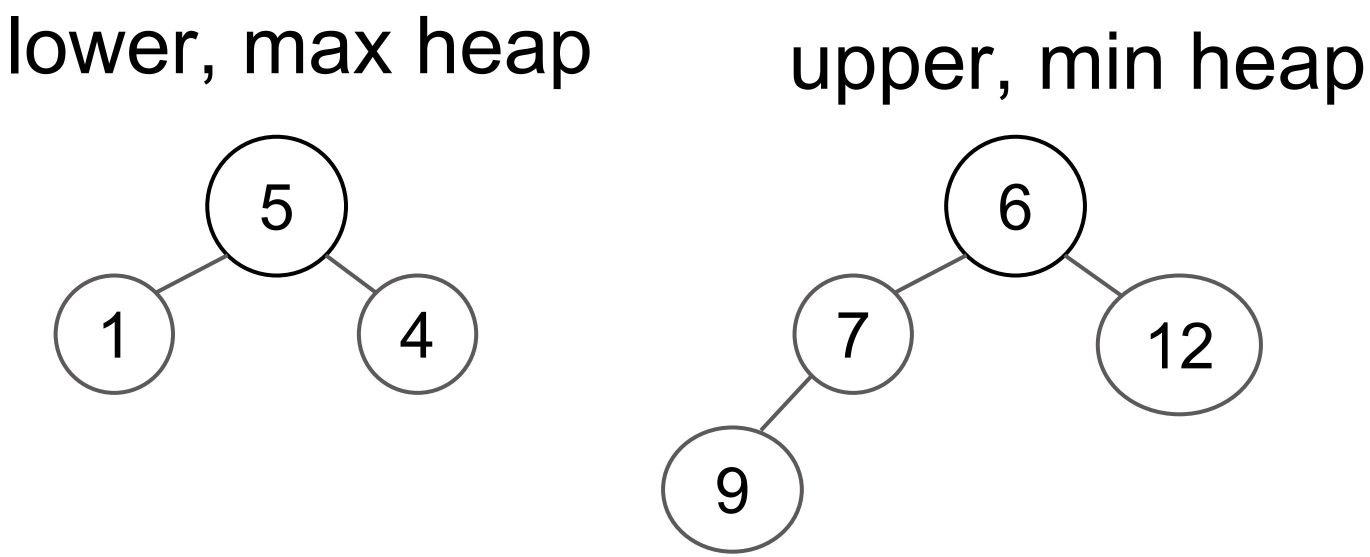
Heads, a better solution

The Idea: split data into two halves

1. Use two heaps, **upper** and **lower.**
   a. **upper** is a **min-heap**, **lower** is a **max-heap**
2. getMedian()
   a. check the sizes of **upper, lower**
   b. If **upper** is larger, return **upper.peekMin()**
   c. If **lower** is larger, return **lower.peekMax()**
   d. If they are the same size, return the average of **upper.peekMin()** and **lower.peekMax()**
3. insert(int n)
   a. If **n** is smaller than **upper.peekMin()**, insert into **lower**
   b. Otherwise, insert into **upper**
   c. rebalance **lower** and **upper**

Heaps, a better solution

Data							
Data	1	4	5	7	12	6	9



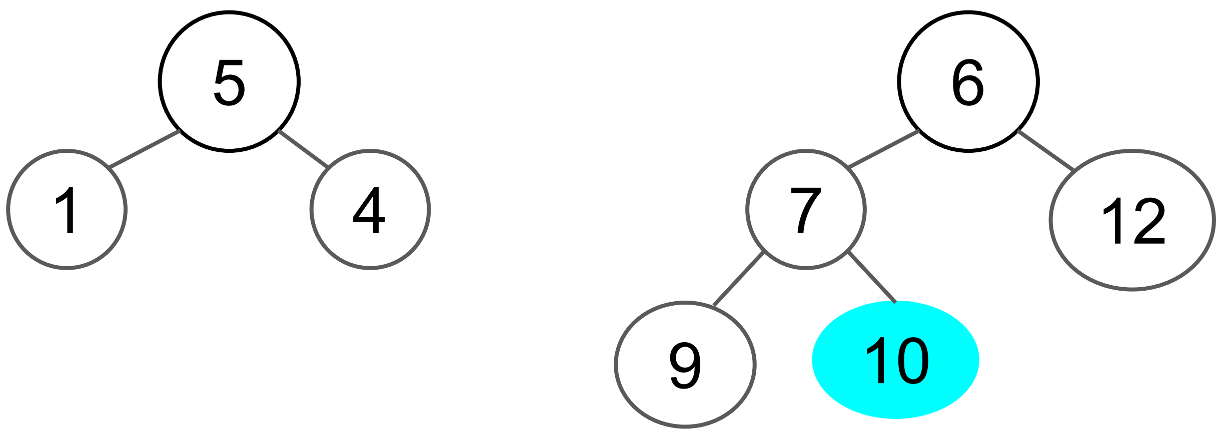
getMedian() returns **6**, the min of the upper heap

Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap

upper, min heap



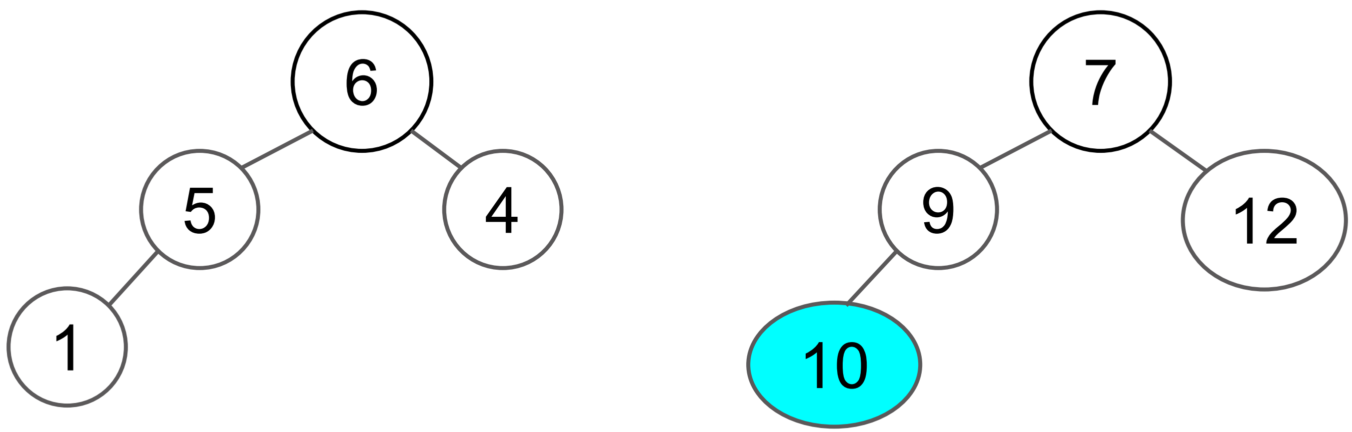
needs rebalancing!

Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap

upper, min heap



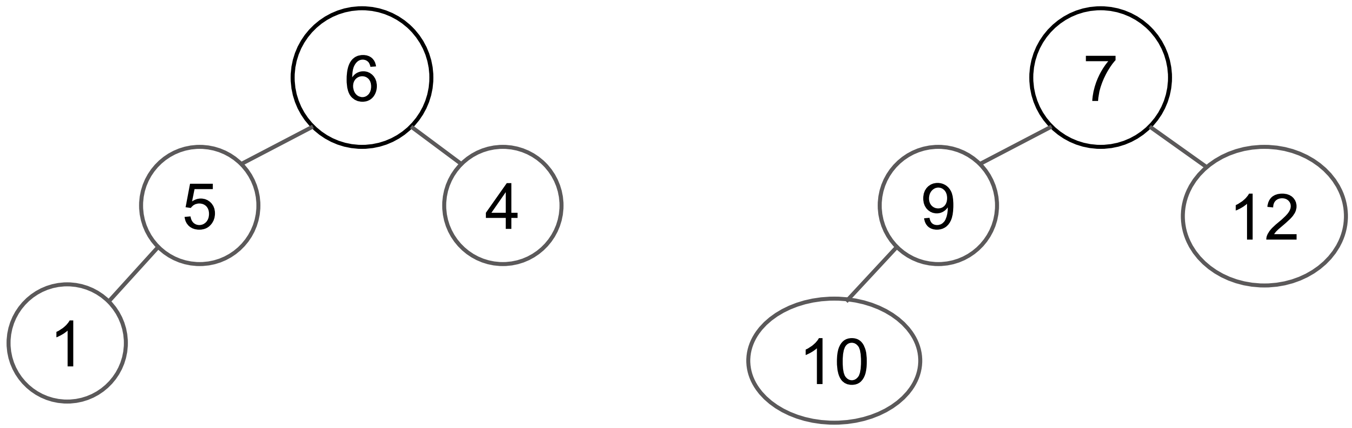
we pop **upper's** min, and insert it into **lower**

Heaps, a better solution

Data	1	4	5	7	12	6	9	10

lower, max heap

upper, min heap



getMedian() returns the average between **6, 7 = 6.5**

Heaps, a better solution

```
public class SexyHeaps{
  private Heap lower, upper;
  public SexyHeaps(){
    lower = new MaxHeap();
    upper = new MinHeap();
  }

  public double getMedian(){
    if(upper.size() > lower.size())
      return upper.peekMin();
    else if(lower.size() > upper.size())
      return lower.peekMax();
    else
      return (upper.peekMin() + lower.peekMax()) / 2.0;
  }
```

Heaps, a better solution

```
    public void insert(int n){
        if(n < upper.peekMin())
            lower.insert(n);
        else
            upper.insert(n);
        //rebalance
        if(lower.size() >= upper.size() + 2)
            upper.insert(lower.removeMax());
        else if(upper.size() >= lower.size() + 2)
            lower.insert(upper.removeMin());
    }
}
```

Heaps, a better solution

Arrays	Complexity	Total time over **n** inserts/getMedians
insert	O(1)	O(n)
getMedian	O(nlogn) for sorting more sophisticated algorithms can yield O(n)	O(n^2)

SexyHeaps	Complexity	Total time over **n** inserts/getMedians
insert	O(logn)	O(nlogn)
getMedian	O(logn)	O(nlogn)

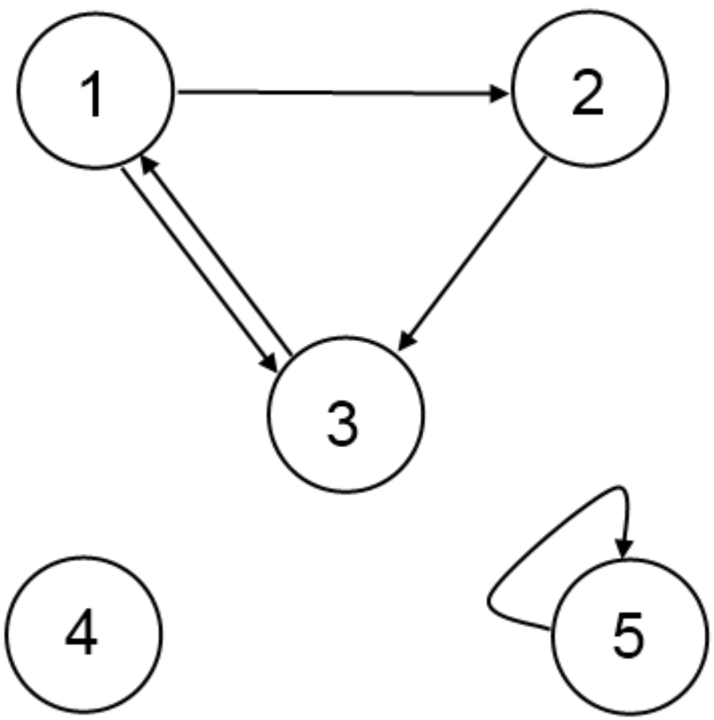
Graphs overview

- Graphs consist of **vertices** and **edges**
- **Vertex:** a point (v) in the graph
- **Edge:** connects two vertices (v, u)
  - If a graph is **directed**, edges only go one way
  - Edges can have **weight** associated with them, e.g. the distance between two points
- The **degree** of a vertex is the number of edges touching it
  - Vertices in a directed graph have **indegree** and **outdegree**
- Graph is **connected** if there is a path between every two vertices

Adjacency Matrix

Create the adjacency matrix for this graph:

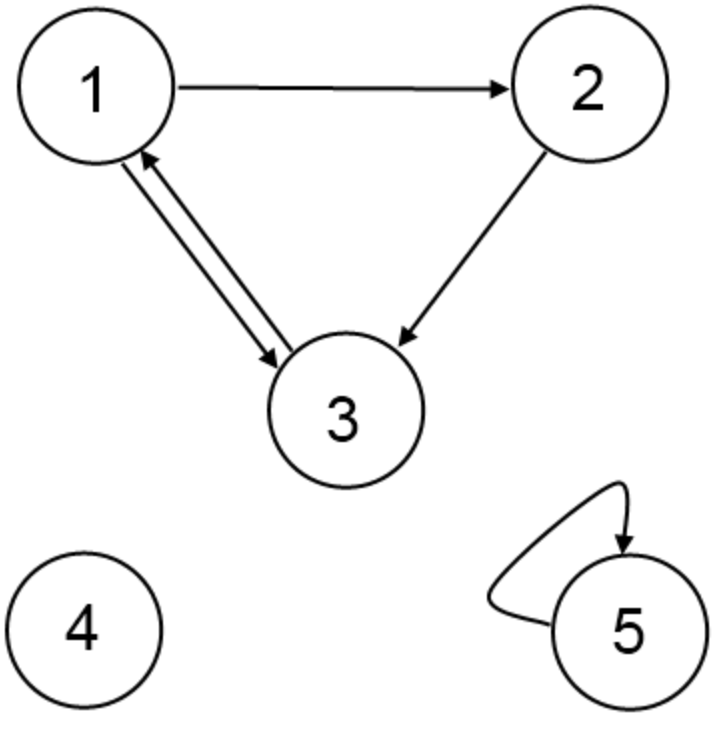
	1	2	3	4	5
1					
2					
3					
4					
5					



Adjacency Matrix

Create the adjacency matrix for this graph:

	1	2	3	4	5
1	-	T	T	-	-
2	-	-	T	-	-
3	T	-	-	-	-
4	-	-	-	-	-
5	-	-	-	-	T



Memory required: $O(V^2)$

Adjacency List

Now create the adjacency list for this graph:

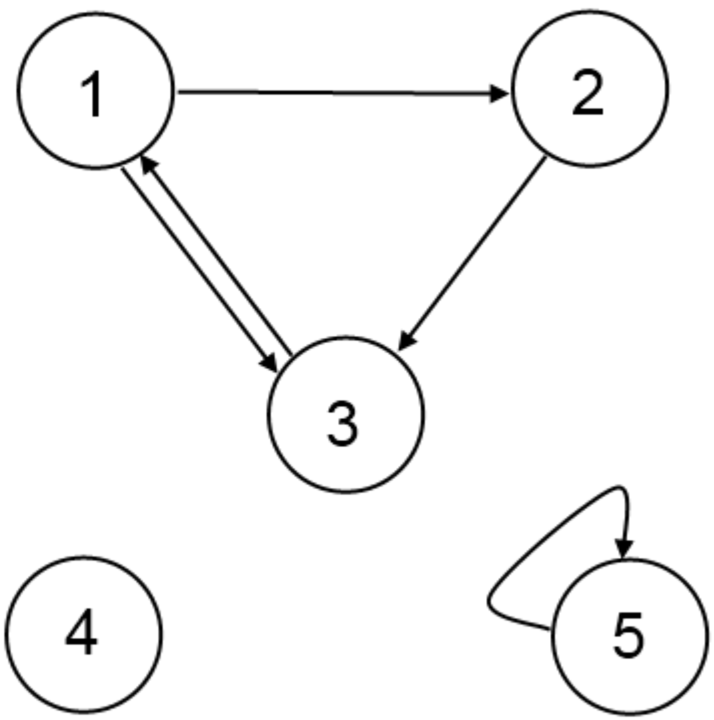
1:

2:

3:

4:

5:



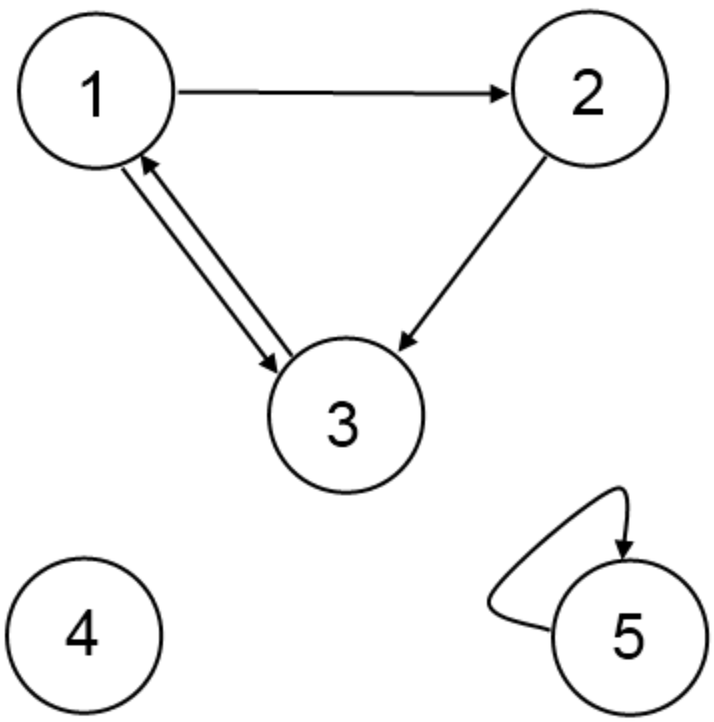
Adjacency List

Now create the adjacency list for this graph:

```
1: 2, 3
2: 3
3: 1
4:
5: 5
```

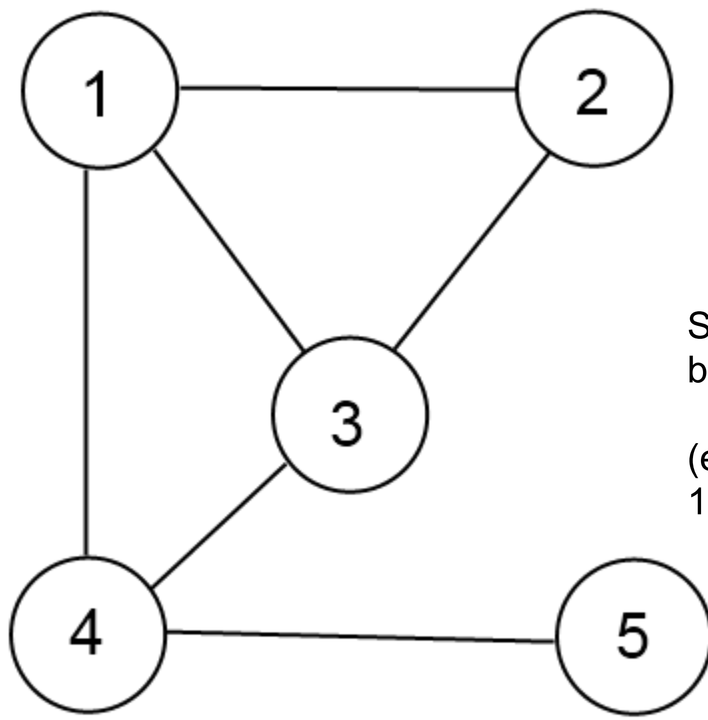
Memory required:

Theta(V+E)



BFS, DFS

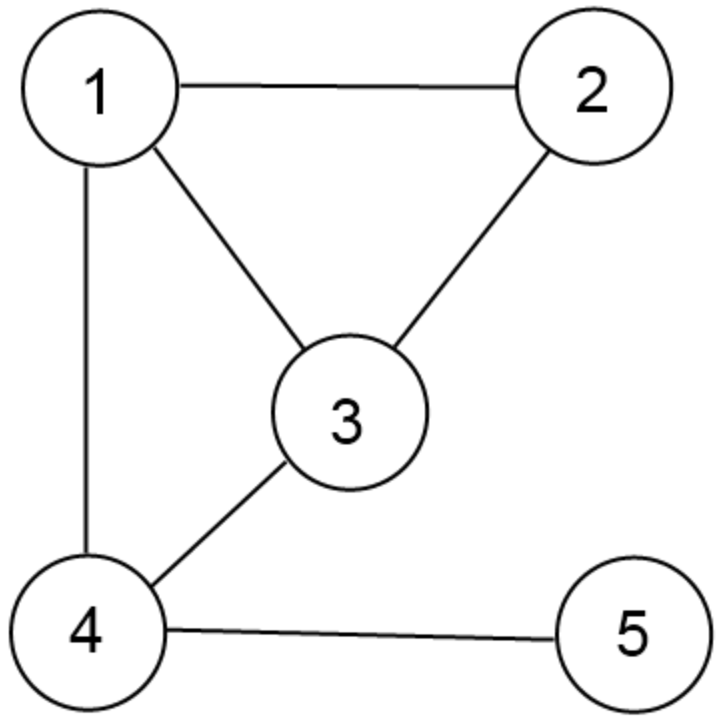
Specify the Breadth First and Depth First Search (BFS & DFS) orders of this graph starting at 2.



Start your BFS/DFS at 2, then break ties from least to greatest:

(ex: if you are tied between 9 and 100, pick 9)

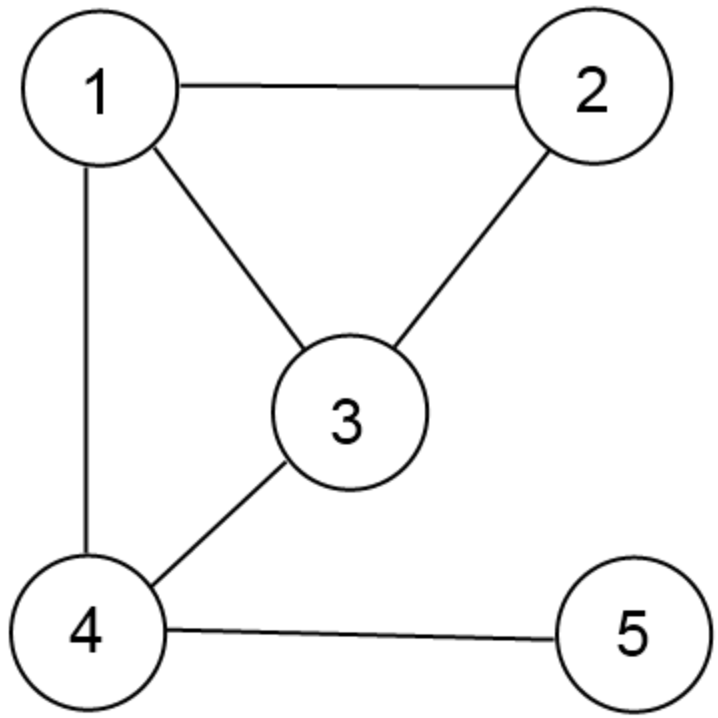
BFS, DFS



BFS search order:

DFS search order:

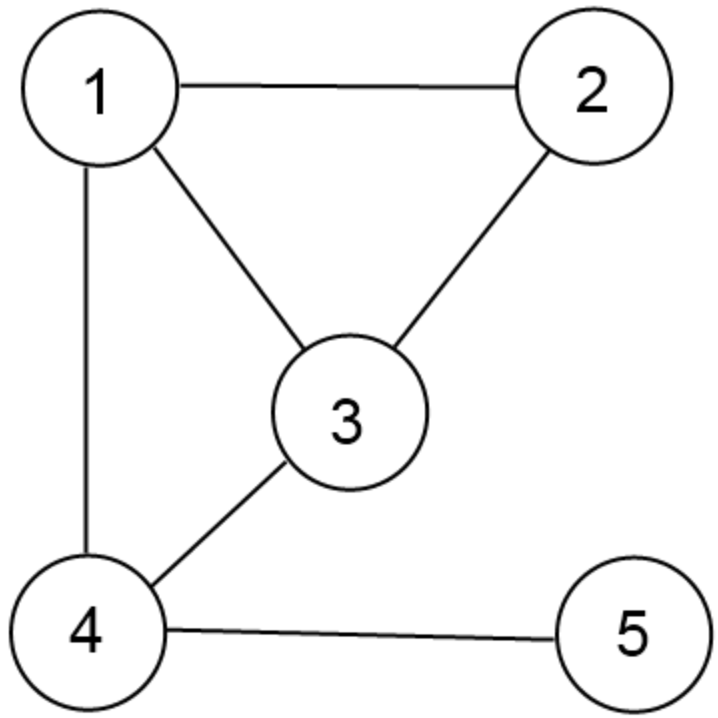
BFS, DFS



BFS search order: 2, 1, 3, 4, 5

DFS search order:

BFS, DFS



BFS search order: 2, 1, 3, 4, 5

DFS search order: 2, 1, 3, 4, 5

BFS, DFS

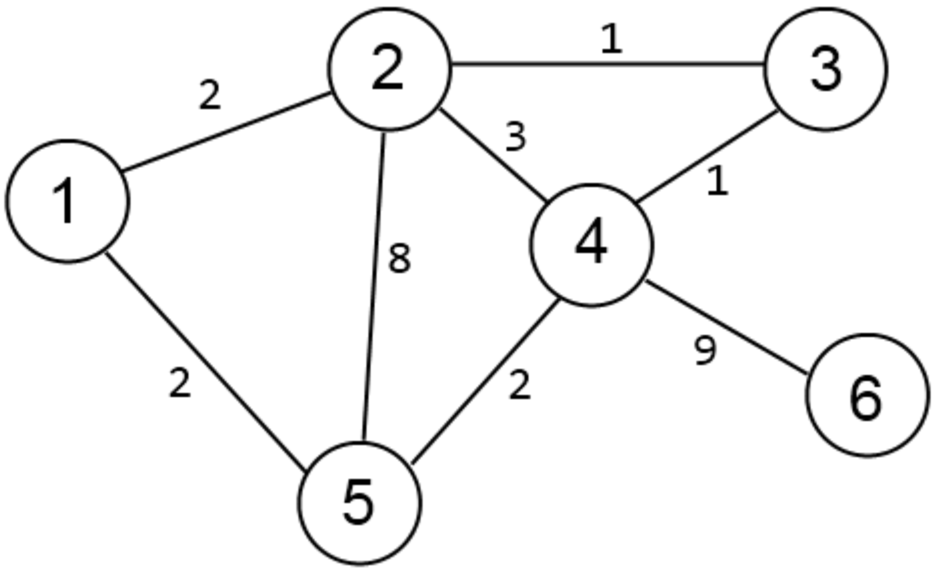
- **DFS** is an algorithm based on recursively checking a node's children
  - See the pseudocode in Lecture 28
  - $O(V + E)$ for adjacency list, $O(V^2)$ for matrix
- **BFS** uses a queue to iteratively deepen in a graph
  - See the pseudocode in Lecture 29
  - Also $O(V + E)$ for adjacency list and $O(V^2)$ for matrix
- Adjacency lists are good for **sparse** trees, but as $E \to V^2$, adjacency matrices become faster

Kruskal's Minimum Spanning Tree

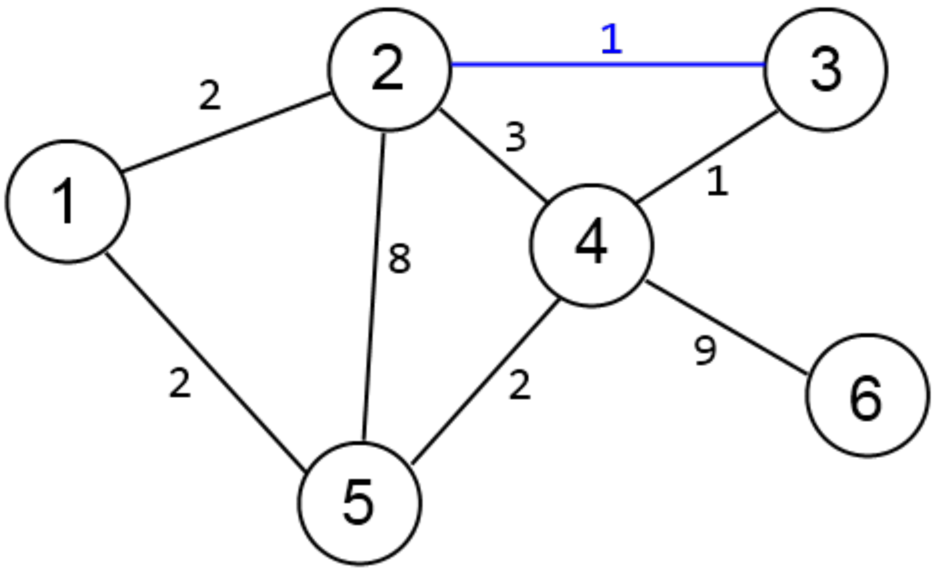
- The algorithm goes through each edge (u, v) in increasing weight order, and if the two points are not already connected, it adds the edge to the set
- Sorting the edges takes O(E log E) time
- Checking connectivity with a DFS takes Theta(E*(V + E)), which is rather slow
  - A trick exists to make the check time O(E log E), you haven't gone over it yet
- The total time you *need to know* is O(V + E log E) = **O(V + E log V)**

Kruskal's Minimum Spanning Tree

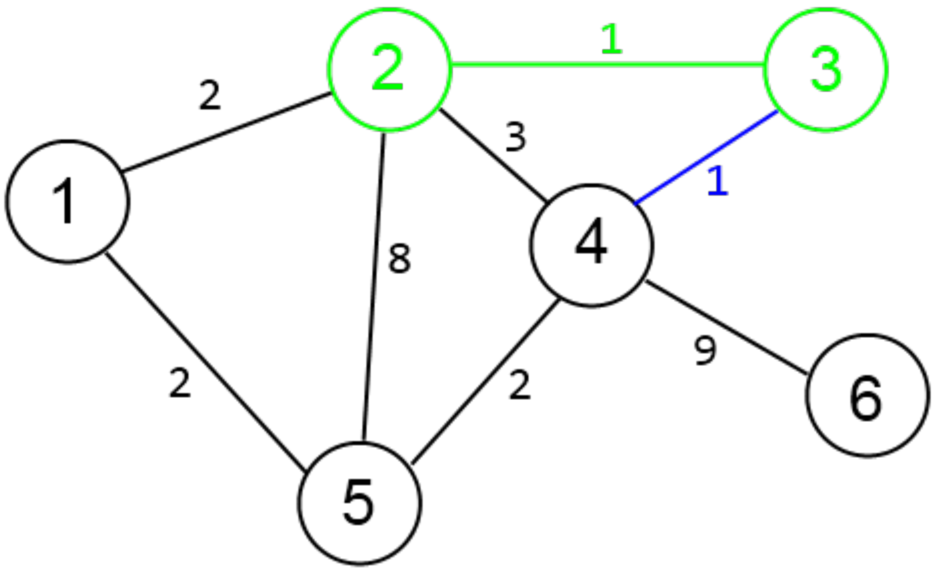
Find a subset of the edges in E that connects every vertex in V with the shortest possible total length:



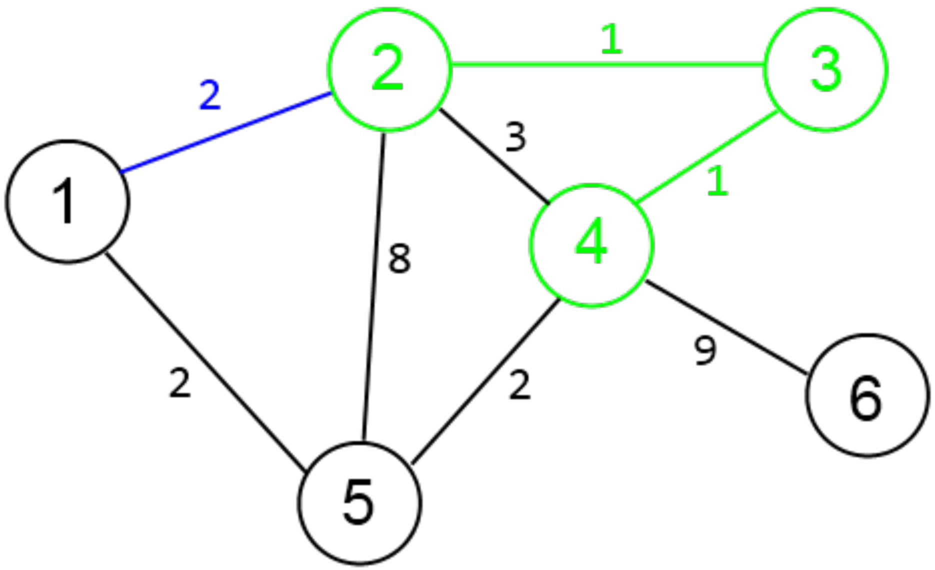
Kruskal's Minimum Spanning Tree



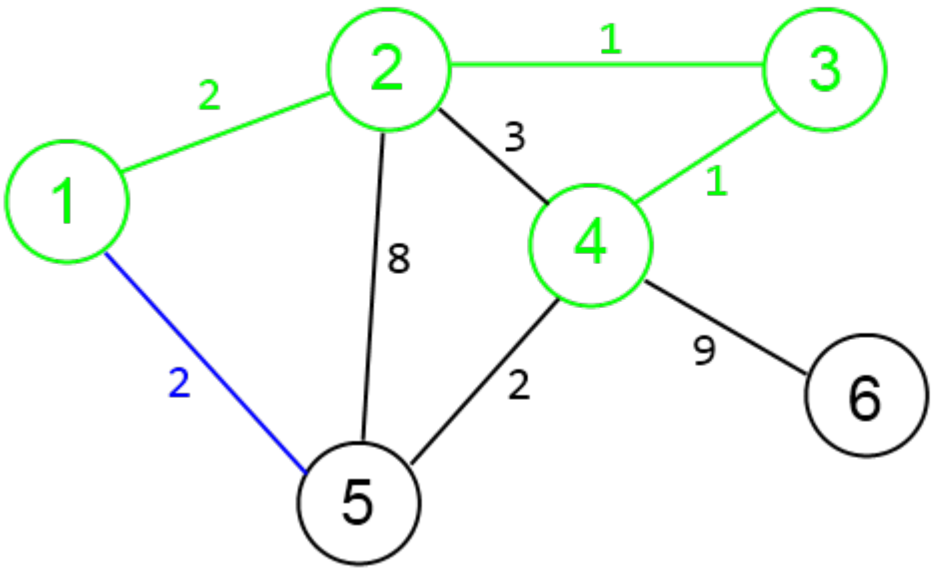
Kruskal's Minimum Spanning Tree



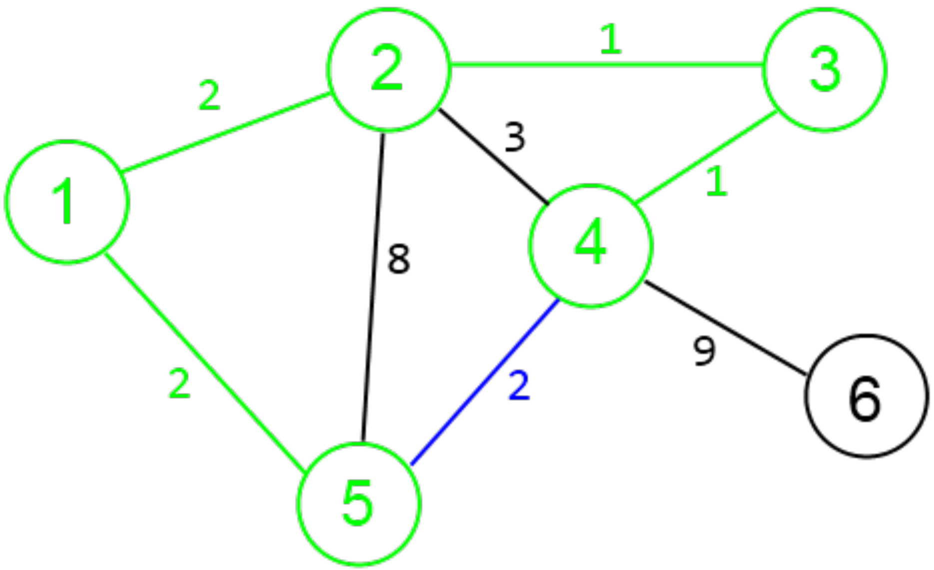
Kruskal's Minimum Spanning Tree



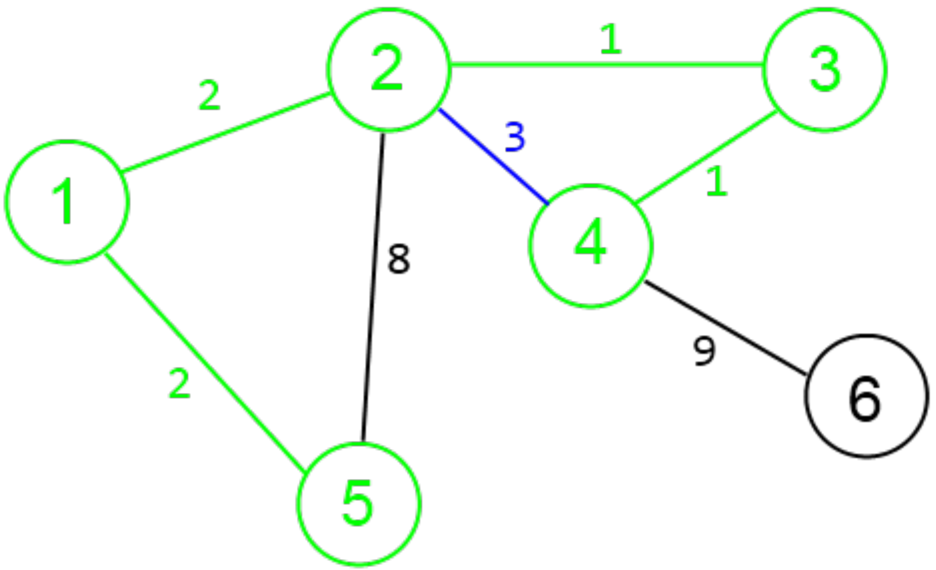
Kruskal's Minimum Spanning Tree



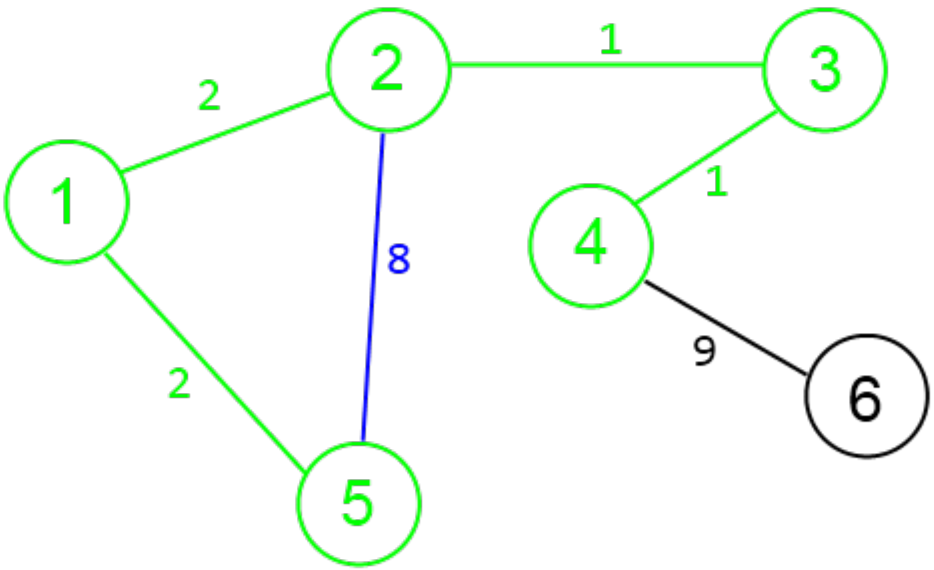
Kruskal's Minimum Spanning Tree



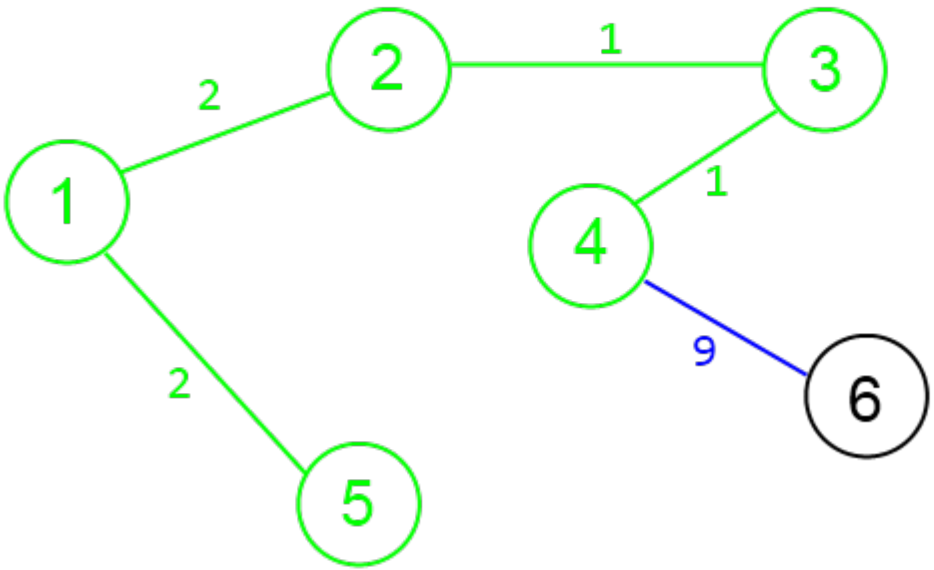
Kruskal's Minimum Spanning Tree



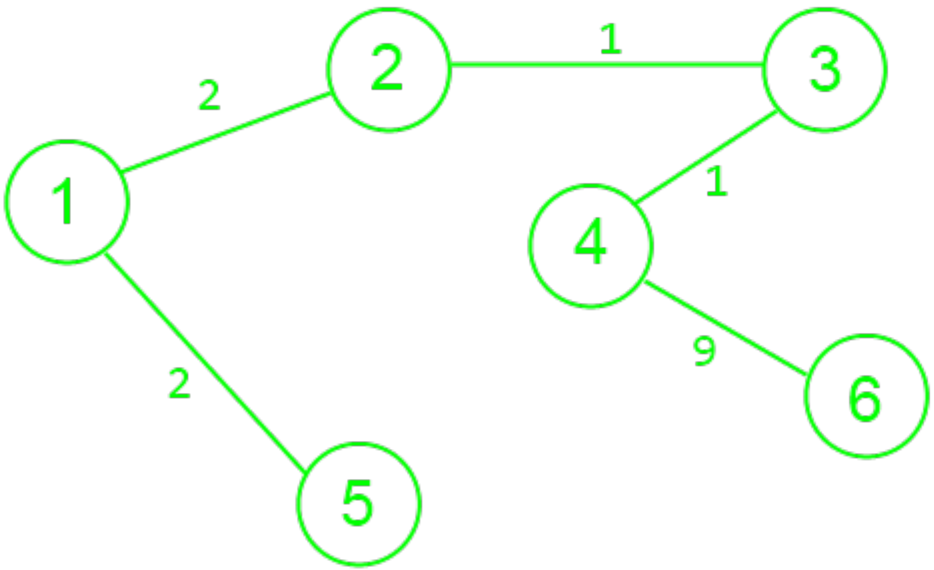
Kruskal's Minimum Spanning Tree



Kruskal's Minimum Spanning Tree



Kruskal's Minimum Spanning Tree



Note: there are two other minimum spanning trees for this graph, depending on how equal-length edges are ordered.

T/F MST Questions

- If a graph G has more than |V| - 1 edges, and there is a unique heaviest edge, then that edge cannot be a part of a minimum spanning tree.

- If G has a cycle with a unique heaviest edge, then that edge cannot be a part of any MST.

- If G has an edge e that is of minimum weight, then e is a part of every MST.

T/F MST Questions

- If a graph G has more than |V| - 1 edges, and there is a unique heaviest edge, then that edge cannot be a part of a minimum spanning tree. F
- If G has a cycle with a unique heaviest edge, then that edge cannot be a part of any MST.
- If G has an edge e that is of minimum weight, then e is a part of every MST.

T/F MST Questions

- If a graph G has more than |V| - 1 edges, and there is a unique heaviest edge, then that edge cannot be a part of a minimum spanning tree. F
- If G has a cycle with a unique heaviest edge, then that edge cannot be a part of any MST. T
- If G has an edge e that is of minimum weight, then e is a part of every MST.

T/F MST Questions

- If a graph G has more than |V| - 1 edges, and there is a unique heaviest edge, then that edge cannot be a part of a minimum spanning tree. F

- If G has a cycle with a unique heaviest edge, then that edge cannot be a part of any MST. T

- If G has an edge e that is of minimum weight, then e is a part of every MST. F

Exceptions

Consider the following code. Why doesn't it compile? Without changing the method signature, how can we fix it?

```
import java.io*;
public class Mathy{
public static int doMath(String fileName) throws DoMathException{
    fs = new FileInputStream(fileName); //may generate a
                                        // FileNotFoundException
    int i = fs.read(); //may generate an IOException
    return 1/i; //may generate an ArithmeticException
}}
-----------------------------------------
public class DoMathException extends Exception{
}
```

Exceptions Answer

```
import java.io*;
public class Mathy{
public static int doMath(String fileName) throws DoMathException{
    try{
        fs = new FileInputStream(fileName);
        int i = fs.read();
    } catch (FileNotFoundException e){
        throw new DoMathException();
    } catch (IOException e){
        throw new DoMathException();
    }
    return 1/i;
}}
```

Good luck on your Midterm!