Announcements

No lecture for the next two weeks (Stuvac and ANZAC day).

Programming Assignment 2 launches April 29. It will be about min heaps and shortest paths.

I have shuffled around the lectures to prepare you for PA2. We have postponed the lecture on Binary Search Trees.

set and map are based on balanced binary search trees.

Last Time

Path: sequence of vertices $v_0, \ldots, v_k$ where each $\{v_{i-1}, v_i\}$ is an edge for $i = 1$ to k .

We allow vertices to repeat, and call it a simple path when all vertices are distinct.

A cycle is a simple path $v_0, \ldots, v_k$ where $\{v_0, v_k\}$ is also an edge and $k \geq 2$.

Last Time

Heap memory seems to be named after the informal meaning of a "pile", and is not related to the data structure.

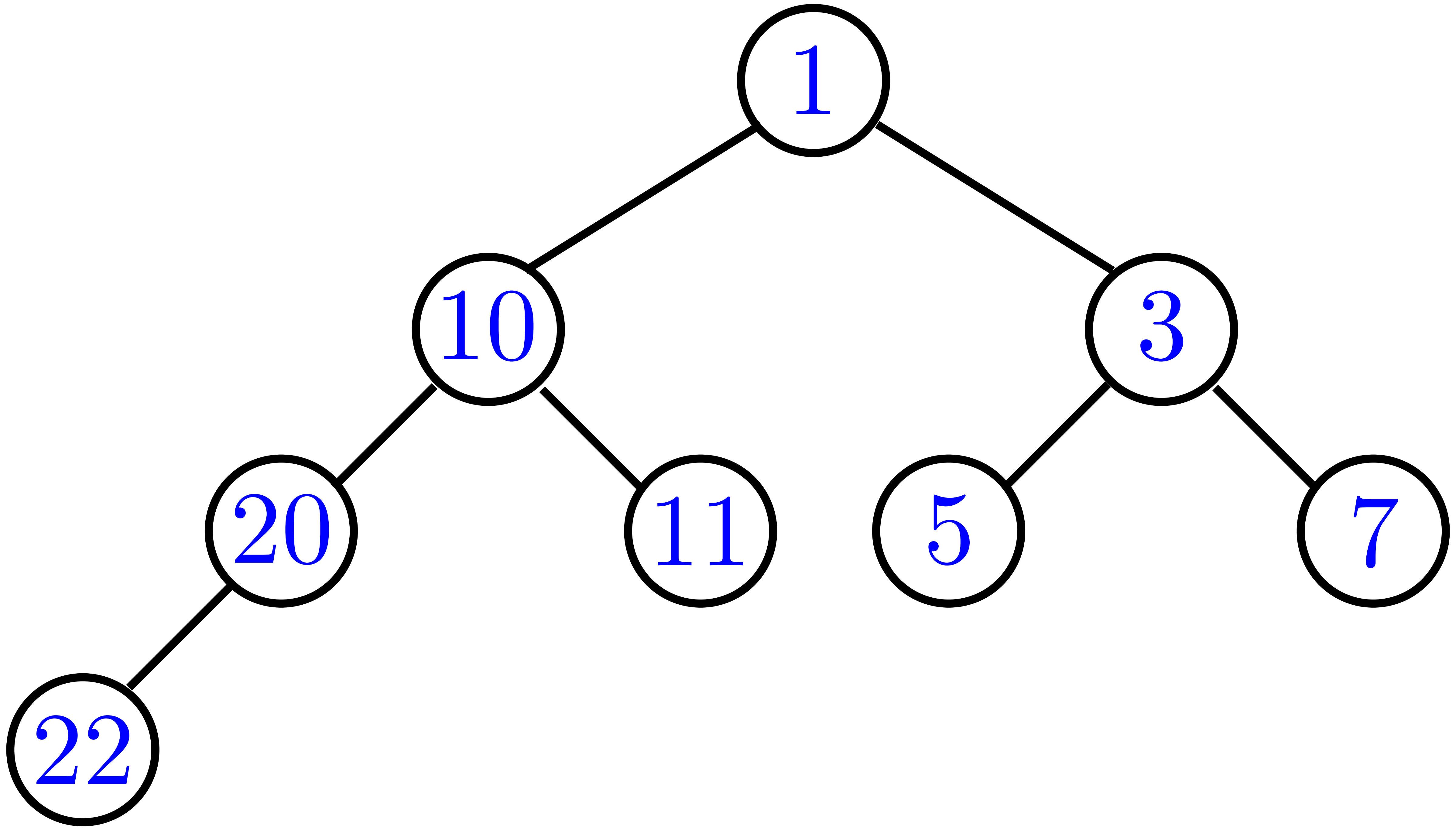
Bjarne Stroustrup calls heap memory the "free store".

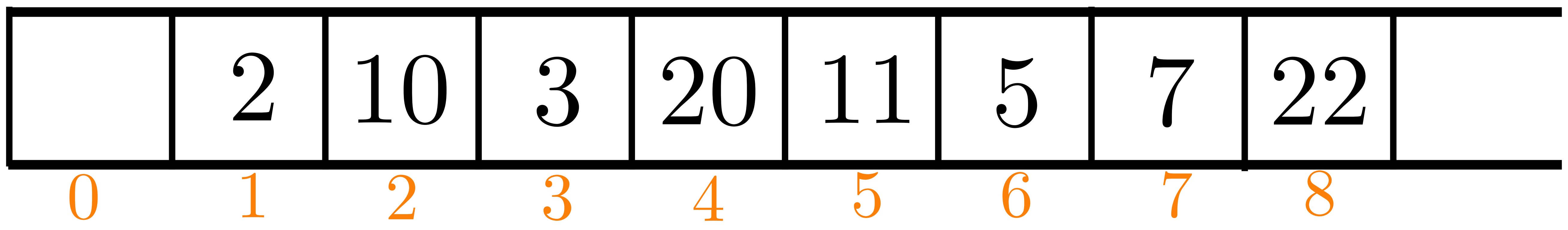
Min Heap

Review:

What are the invariants?

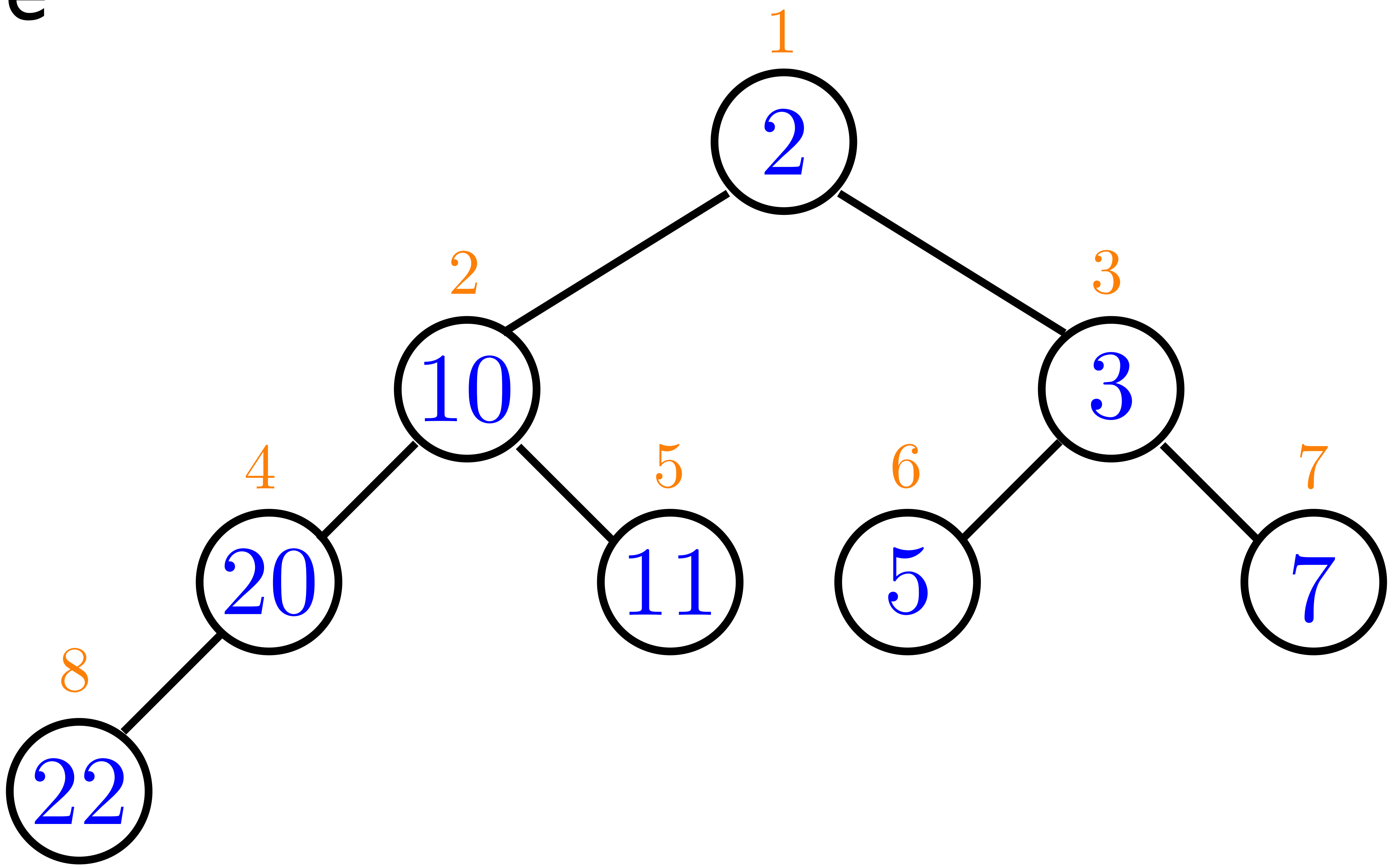
What are the operations?





A nice feature of heaps is that they are relatively simple to implement.

We can represent the heap by a vector.



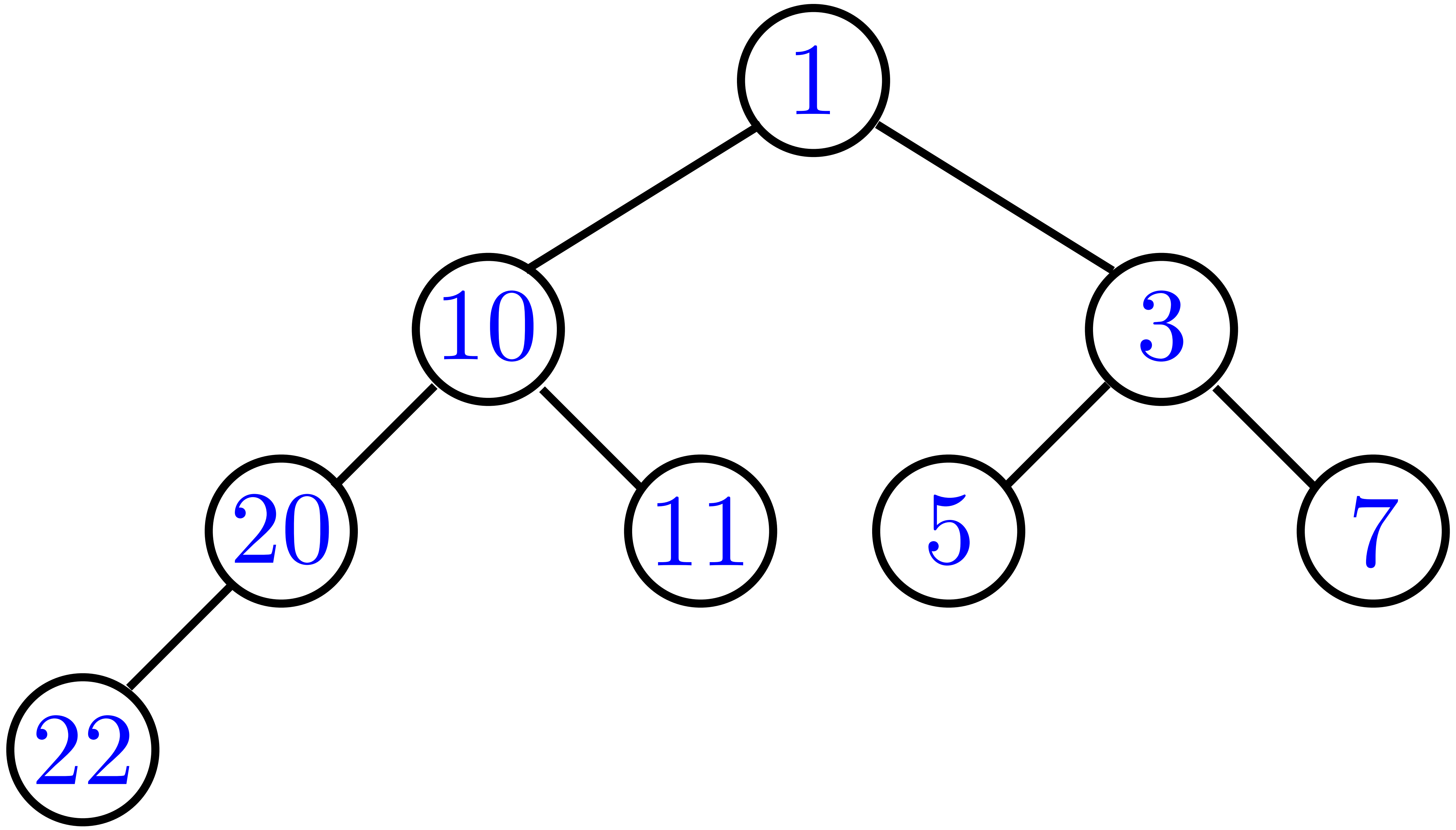
Complexity

What is the complexity of our 3 operations on a min heap?

top:

push:

pop:



Heapsort

Let's see how we can use a heap to sort a vector of n elements.

	3	7	6	5	3	5

Heapsort

Let's see how we can use a heap to sort a vector of n elements.

	3	7	6	5	3	5

To sort the vector from smallest to largest, it is best to use a max heap.

Heapsort

Let's see how we can use a heap to sort a vector of n elements.

	3	7	6	5	3	5

To sort the vector from smallest to largest, it is best to use a max heap.

In a max heap the key at a node is not smaller than the keys of its children.

Heapsort

Let's see how we can use a heap to sort a vector of n elements.

	3	7	6	5	3	5

To sort the vector from smallest to largest, it is best to use a max heap.

In a max heap the key at a node is not smaller than the keys of its children.

In a max heap, the root holds the largest element.

Heapsort

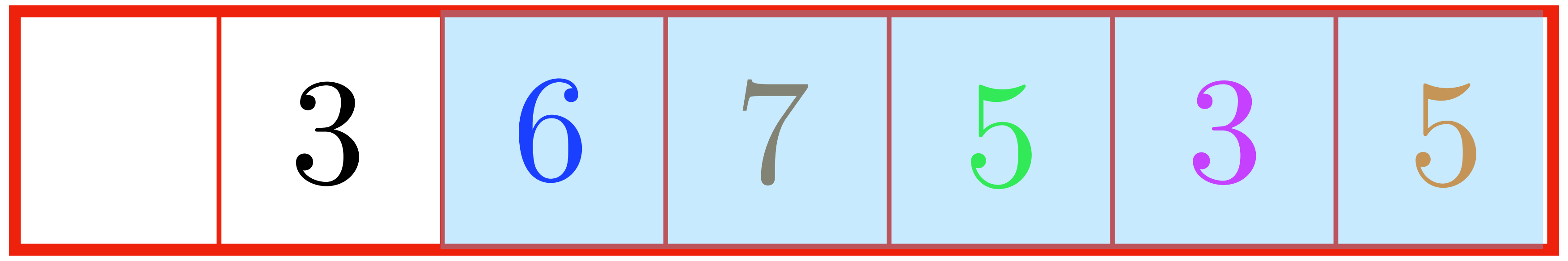
Heapsort consists of two phases. In the first phase we create a max heap with the elements of the vector.

	3	6	7	5	3	5

We grow a heap by inserting each element of the vector into it.

Heapsort

Heapsort consists of two phases. In the first phase we create a max heap with the elements of the vector.



We grow a heap by inserting each element of the vector into it.

Heapsort

Heapsort consists of two phases. In the first phase we create a max heap with the elements of the vector.

	3	6	7	5	3	5

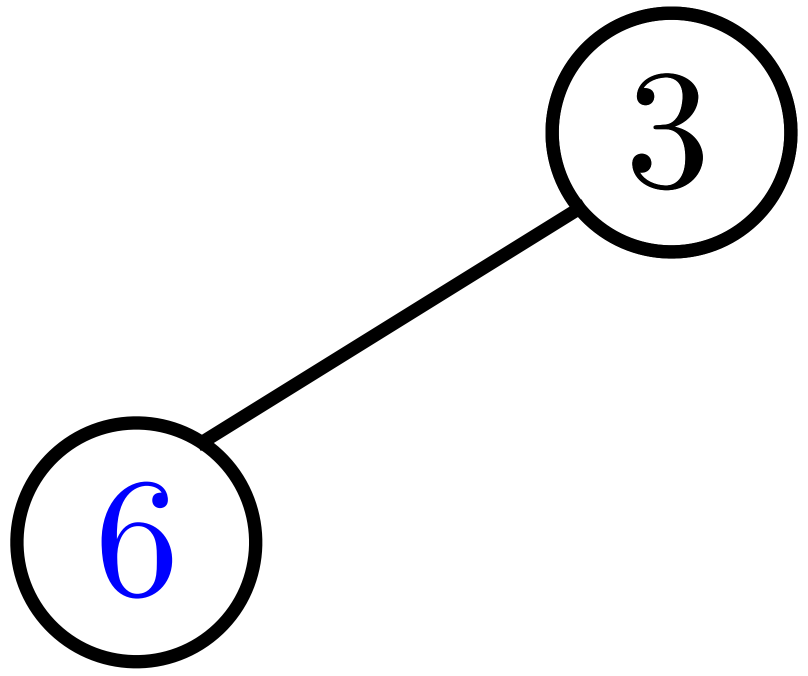
We grow a heap by inserting each element of the vector into it.

Not much to do with the first element.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	3	6	7	5	3	5

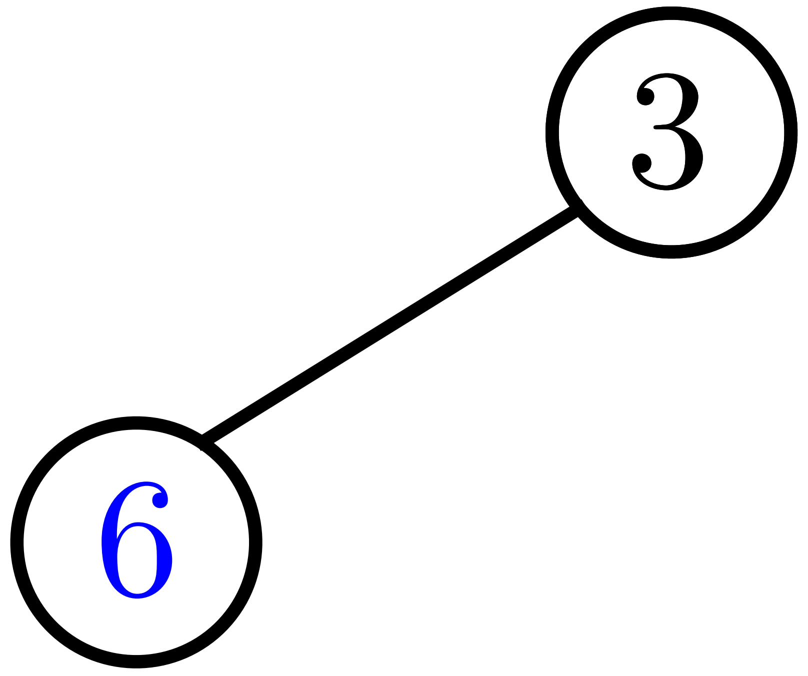


Next we insert 6.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	3	6	7	5	3	5



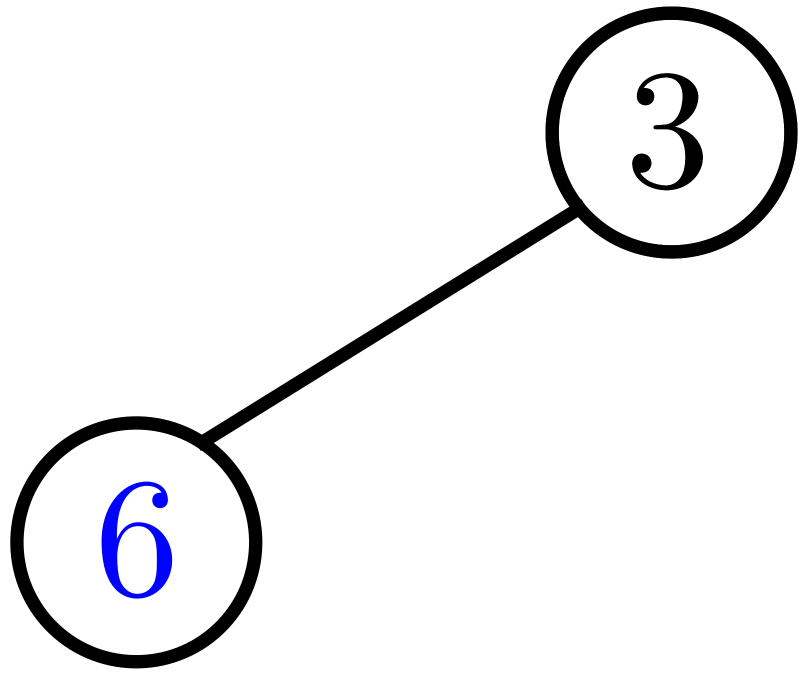
Next we insert 6.

Now we "swim" with 6.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	3	6	7	5	3	5



Next we insert 6.

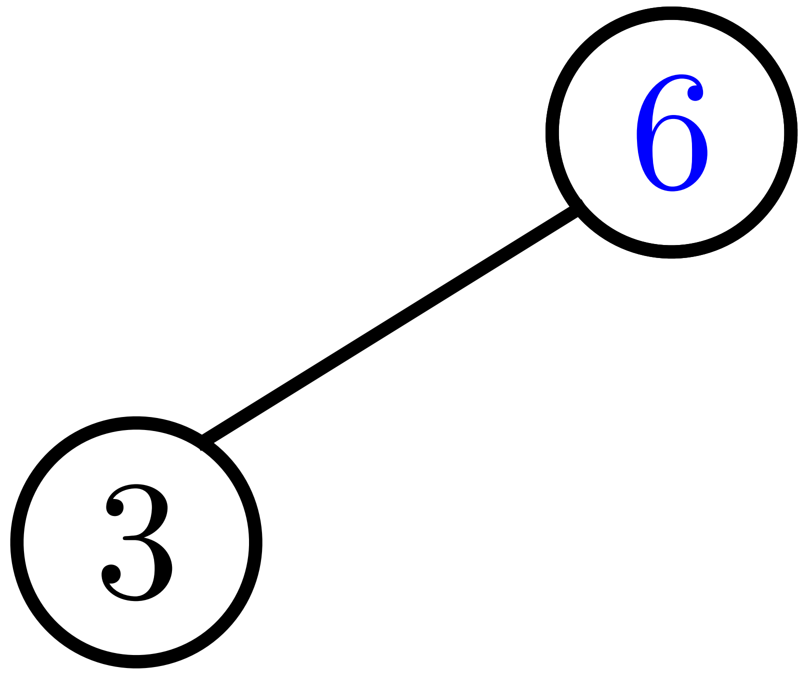
Now we "swim" with 6.

Is $3 < 6$? Yes, so the max heap property is violated. We swap them.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	6	3	7	5	3	5

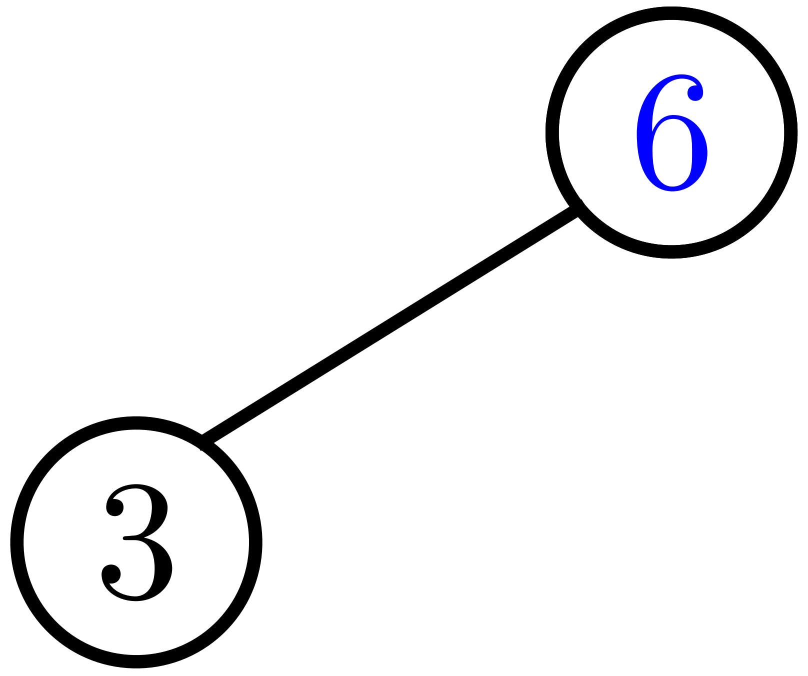


Now we have a max heap with the first two elements of the vector.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	6	3	7	5	3	5



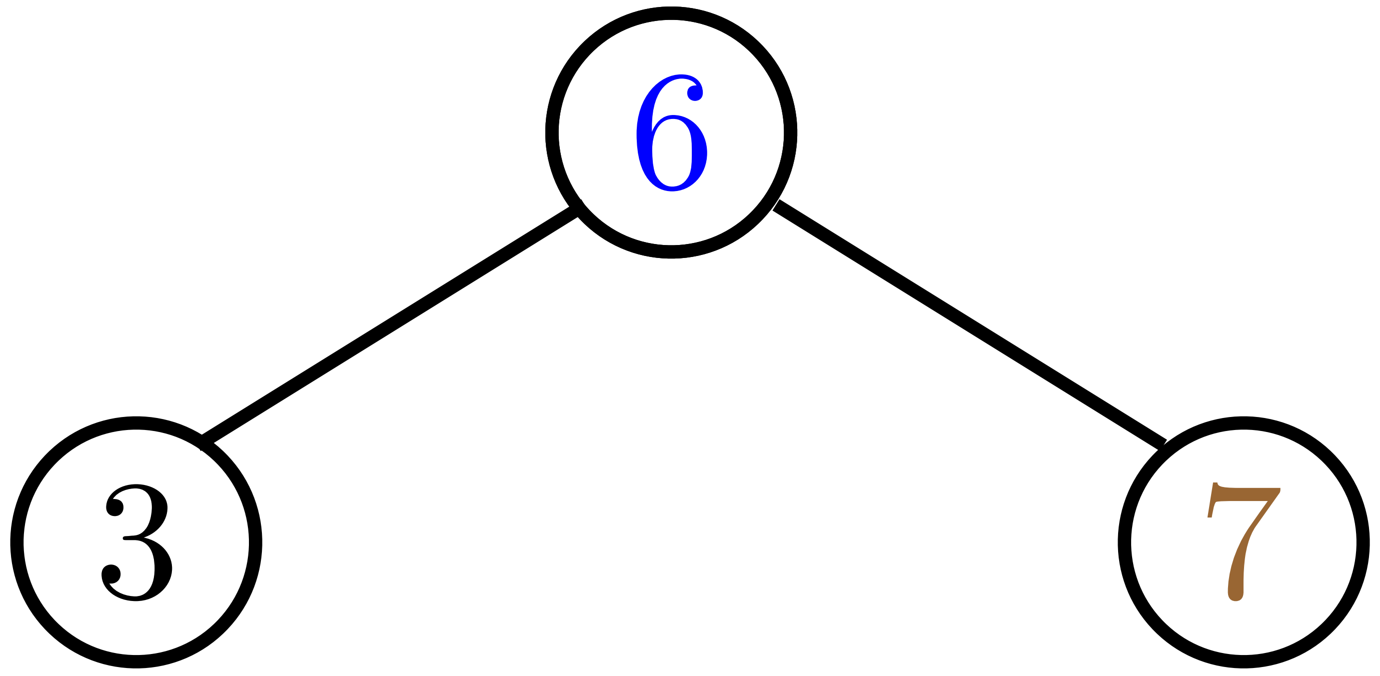
Next we insert 7 into the heap.

Heapsort

In the first phase we create a max heap with the elements of the vector.

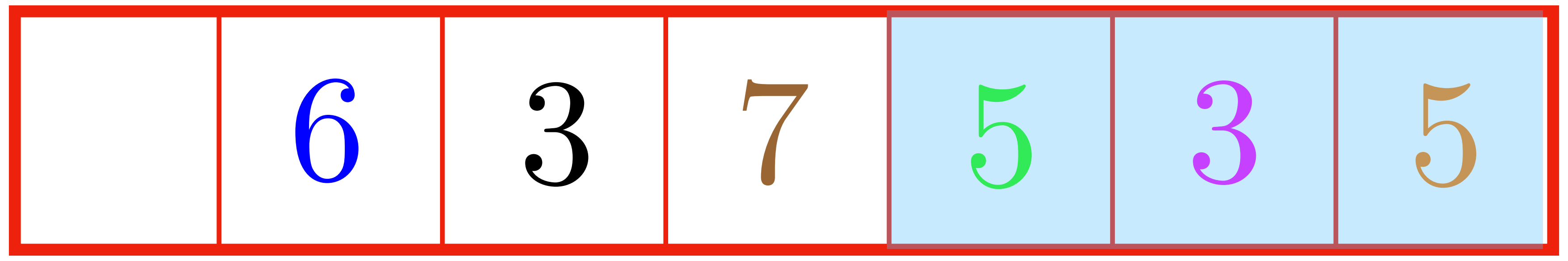
	6	3	7	5	3	5

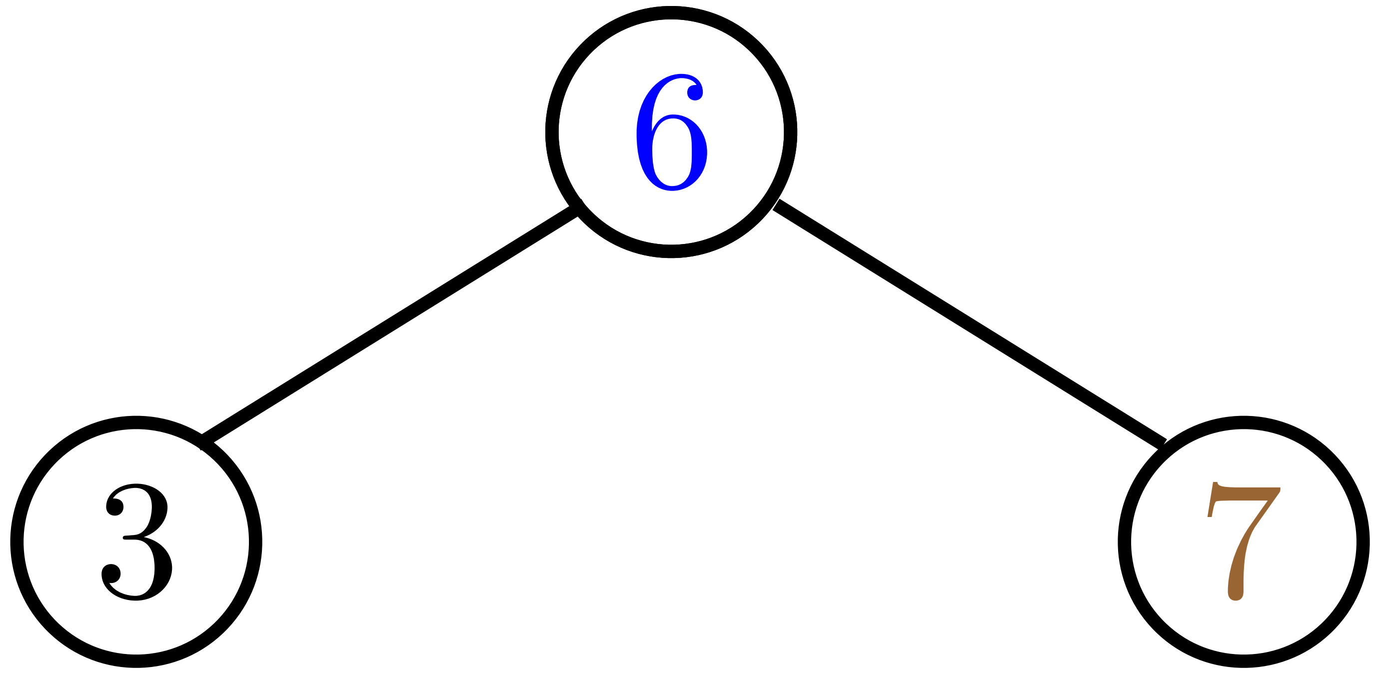
Next we insert 7 into the heap.



Heapsort

In the first phase we create a max heap with the elements of the vector.





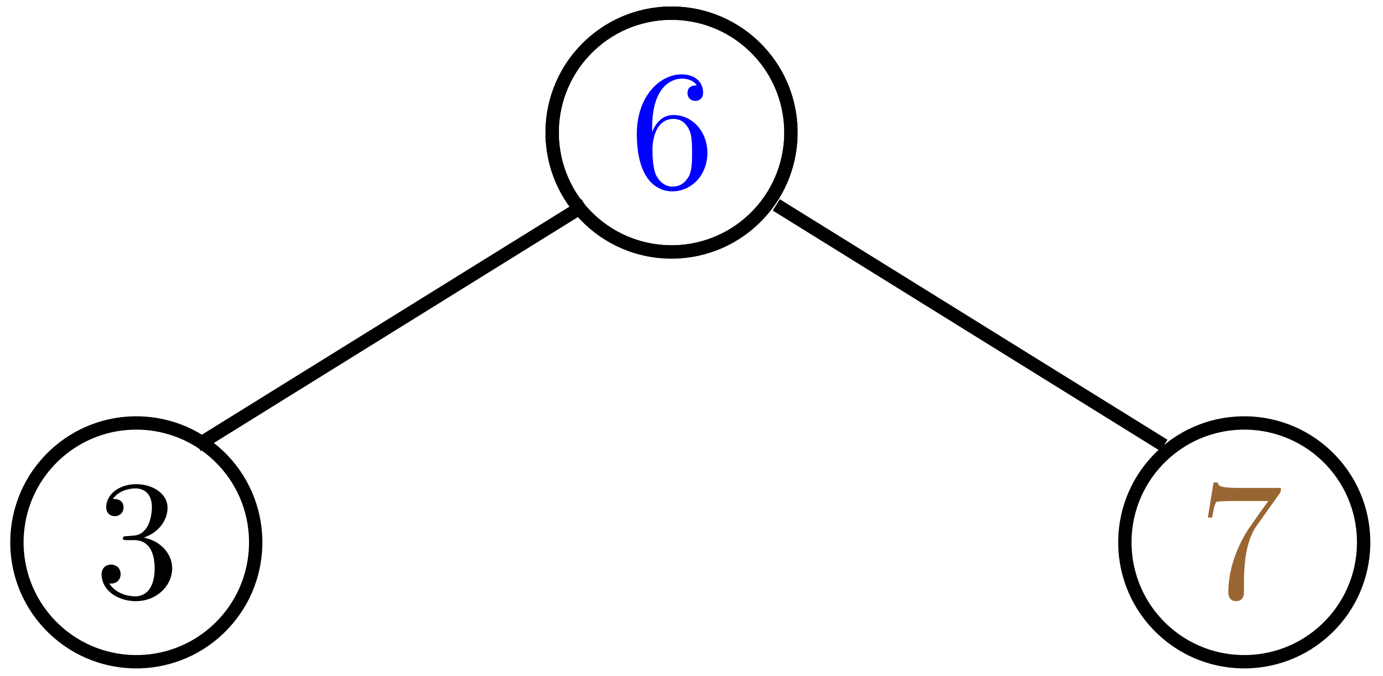
Next we insert 7 into the heap.

Now "swim" with 7.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	6	3	7	5	3	5



Next we insert 7 into the heap.

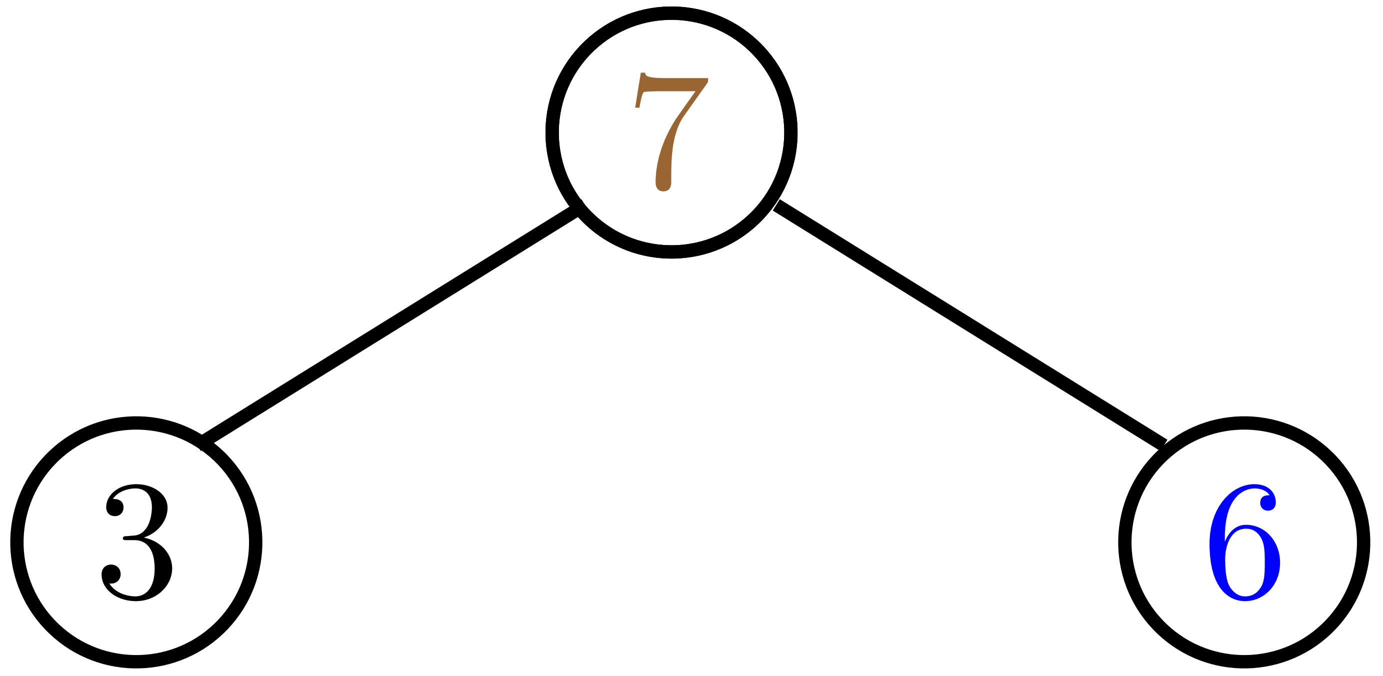
Now "swim" with 7.

Is $6 < 7$? Yes, so the max heap property is violated. We swap them.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	3	6	5	3	5

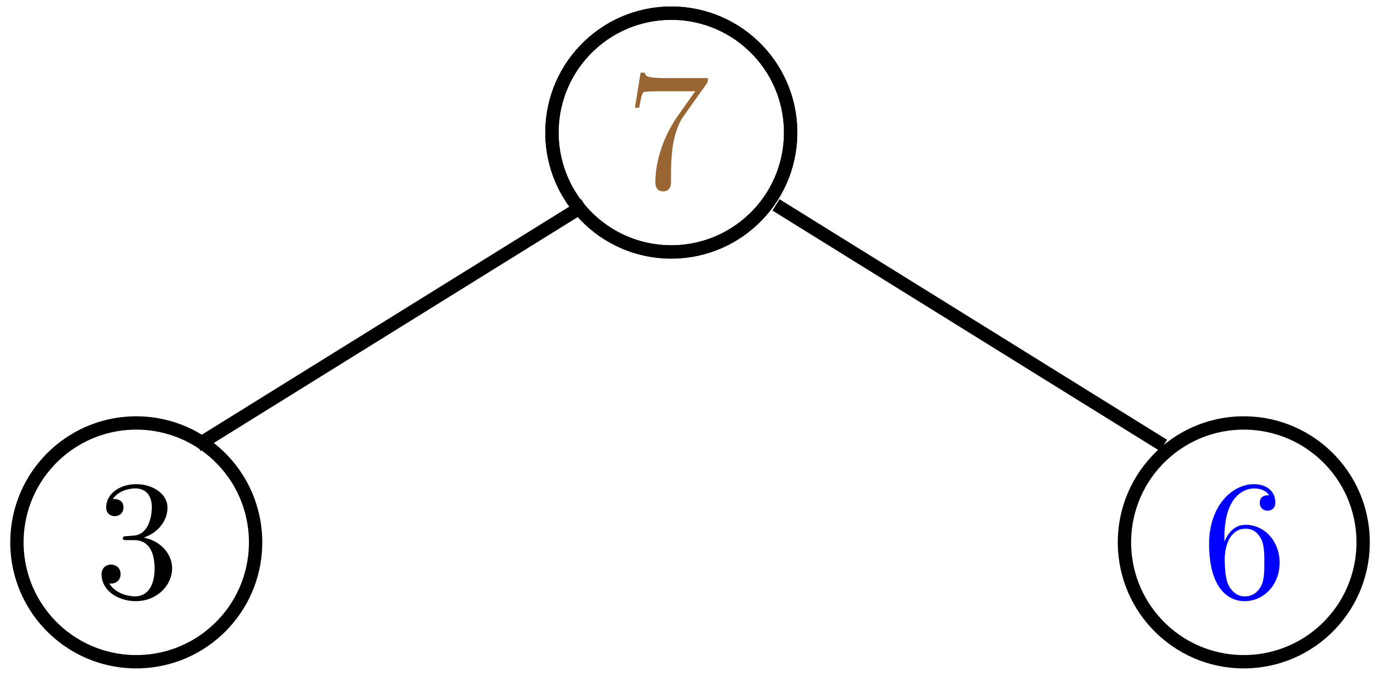


Now we have a max heap with the first three elements of the vector.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	3	6	5	3	5



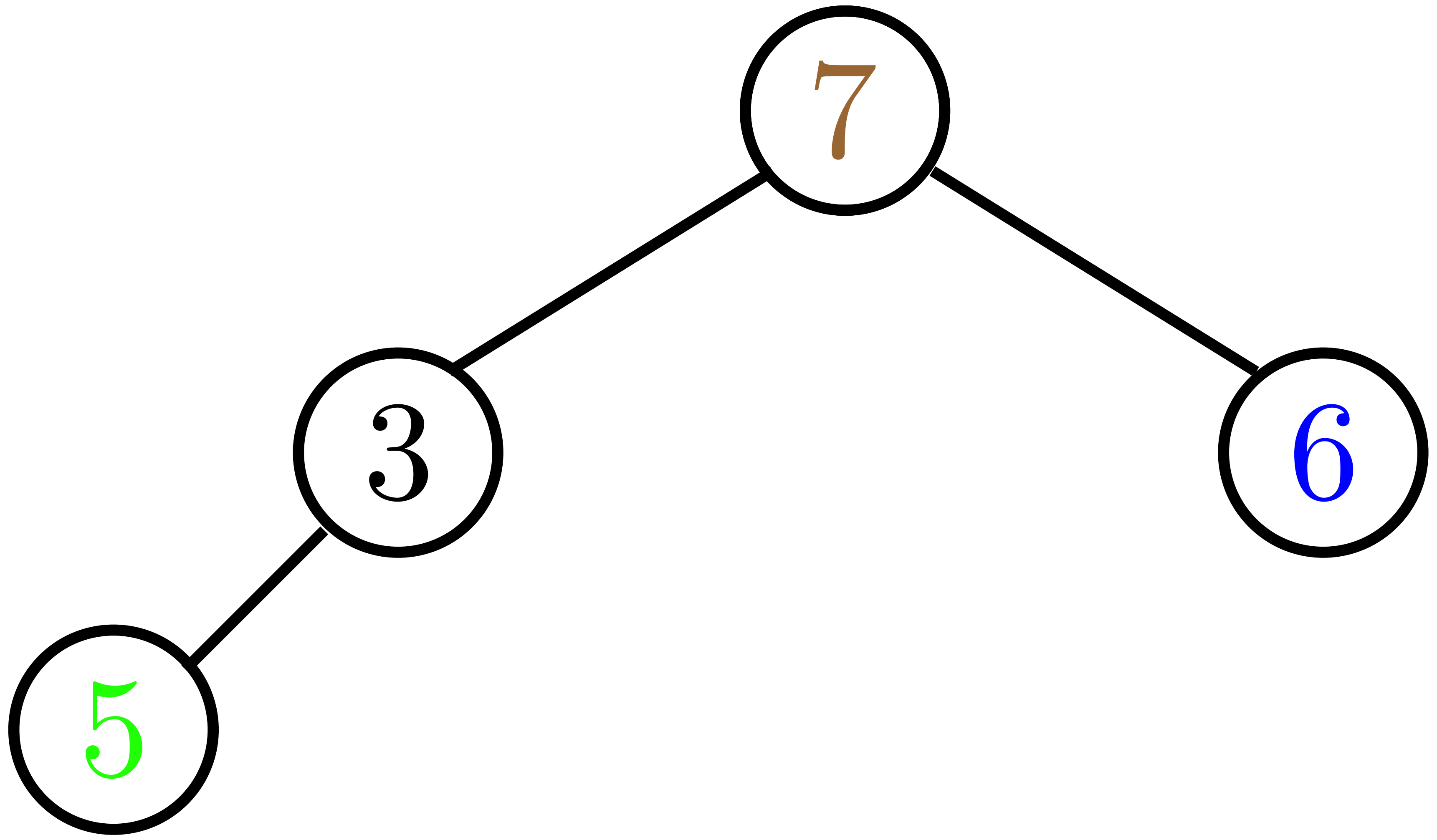
Next we insert 5.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	3	6	5	3	5

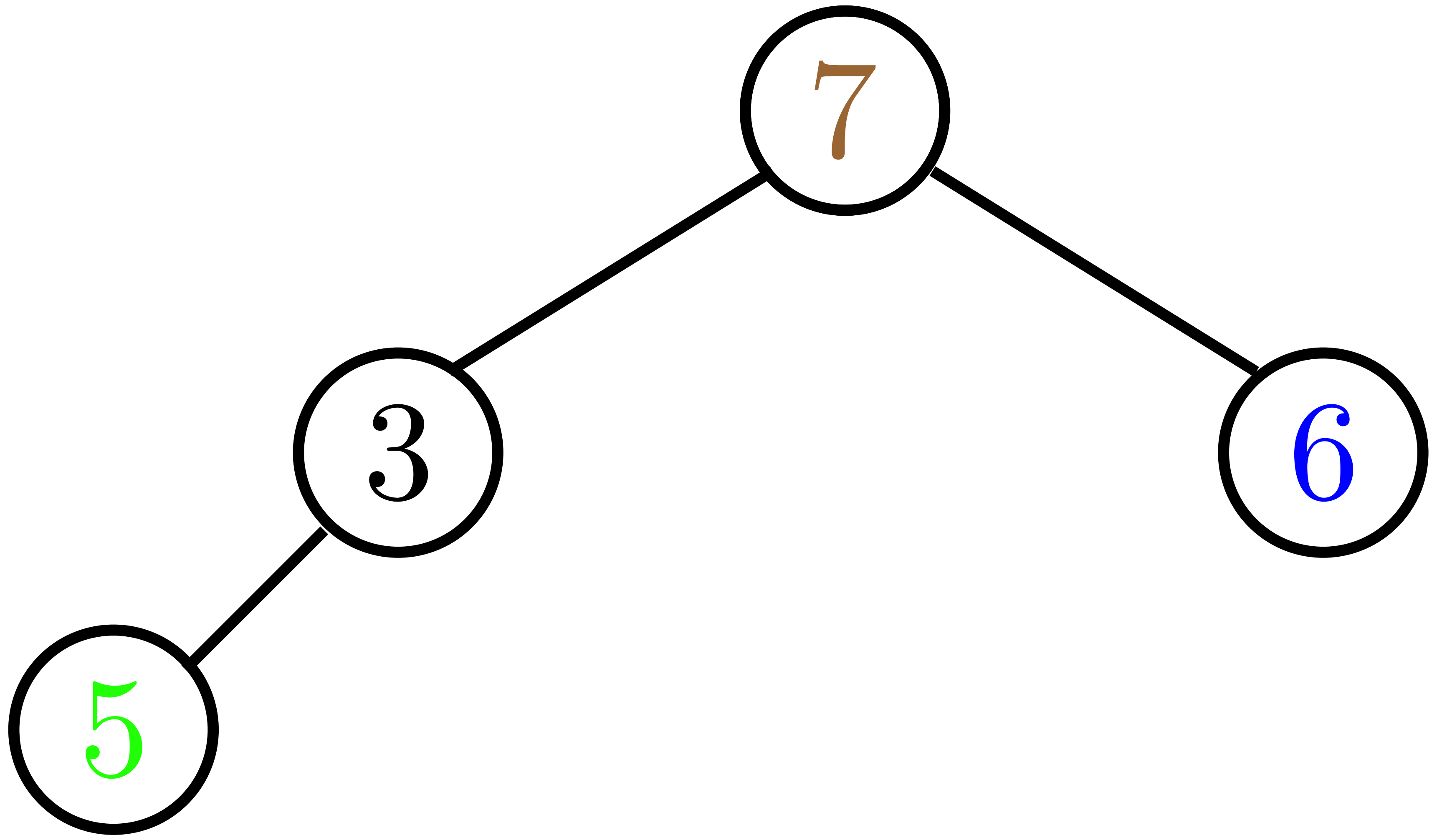
Next we insert 5.



Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	3	6	5	3	5



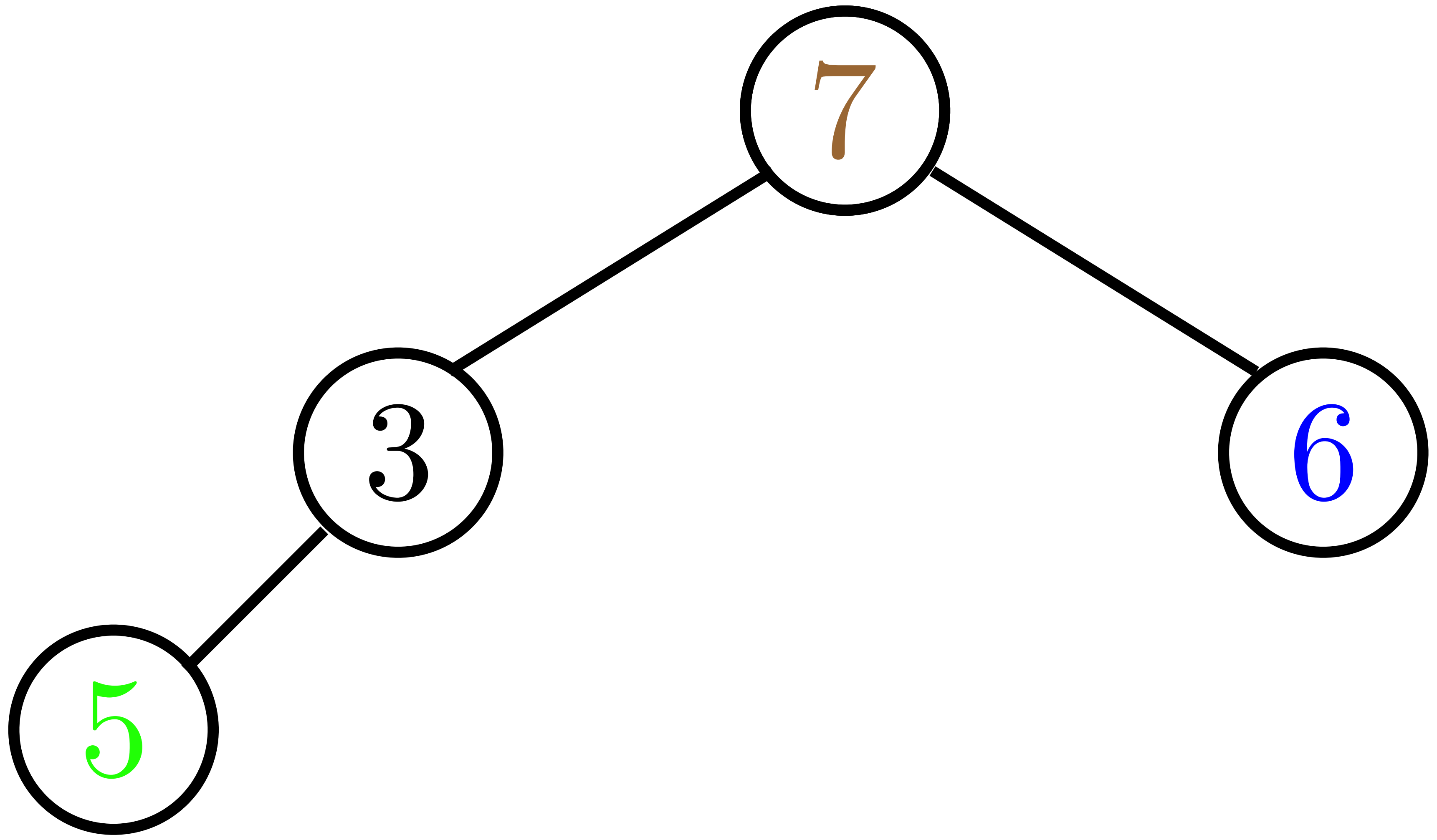
Next we insert 5.

Swim with 5 .

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	3	6	5	3	5



Next we insert 5.

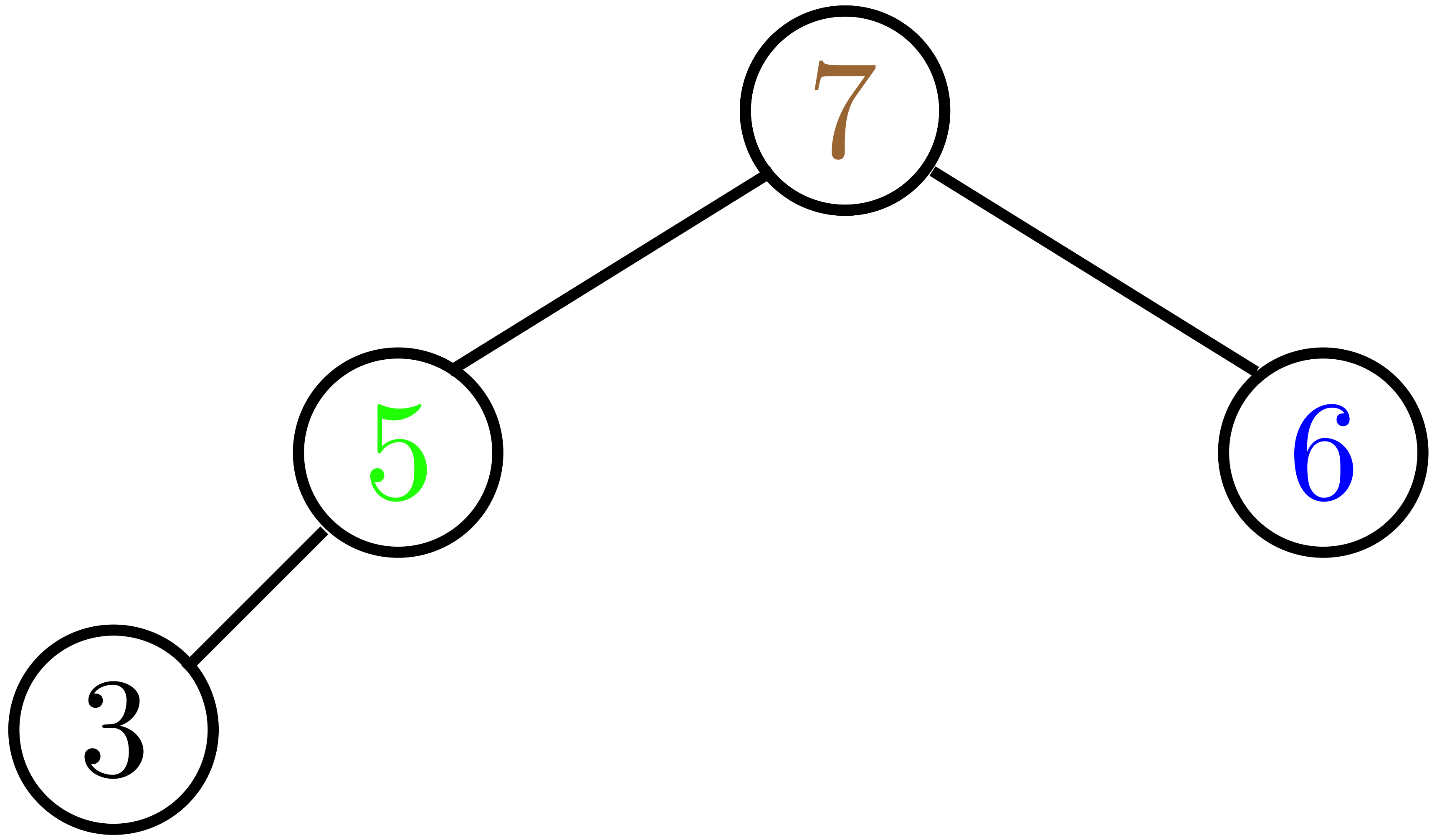
Swim with 5 .

Is $3 < 5$? Yes, so the max heap property is violated. We swap them.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5



Swim with 5 .

Is $7 < 5$?

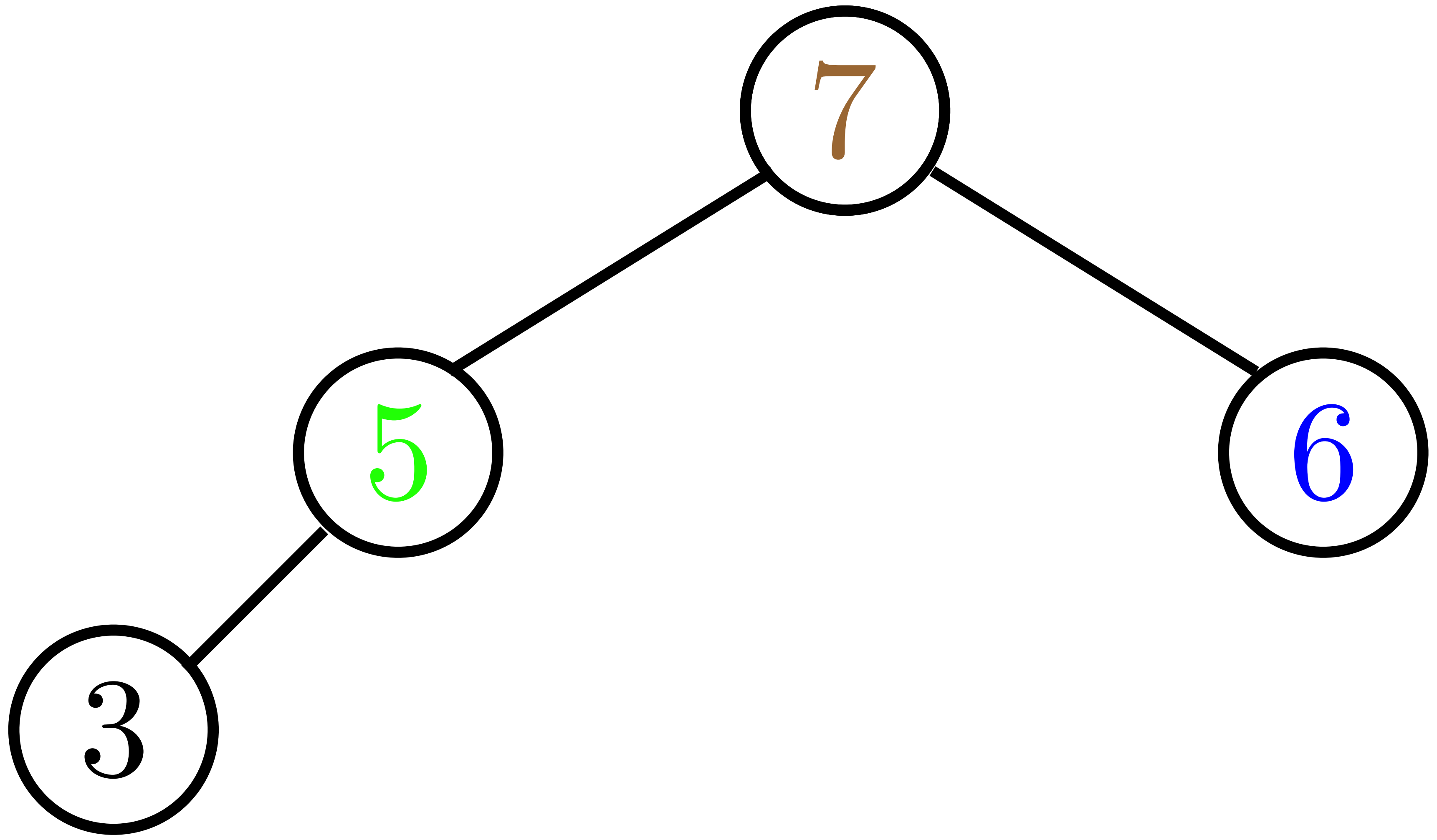
No, we have a max heap on the first four elements of the vector.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

Next we insert 3 .

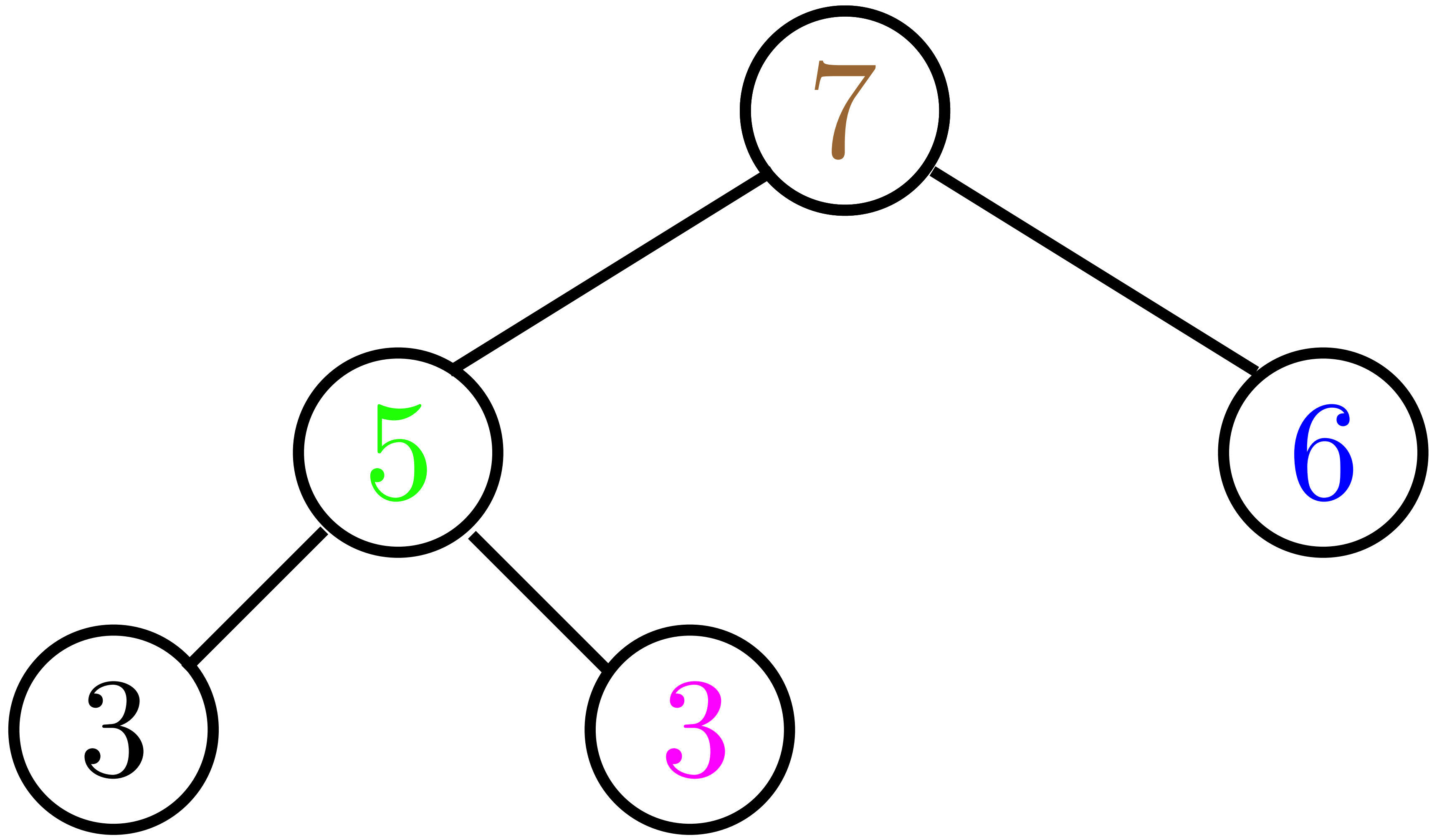


Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

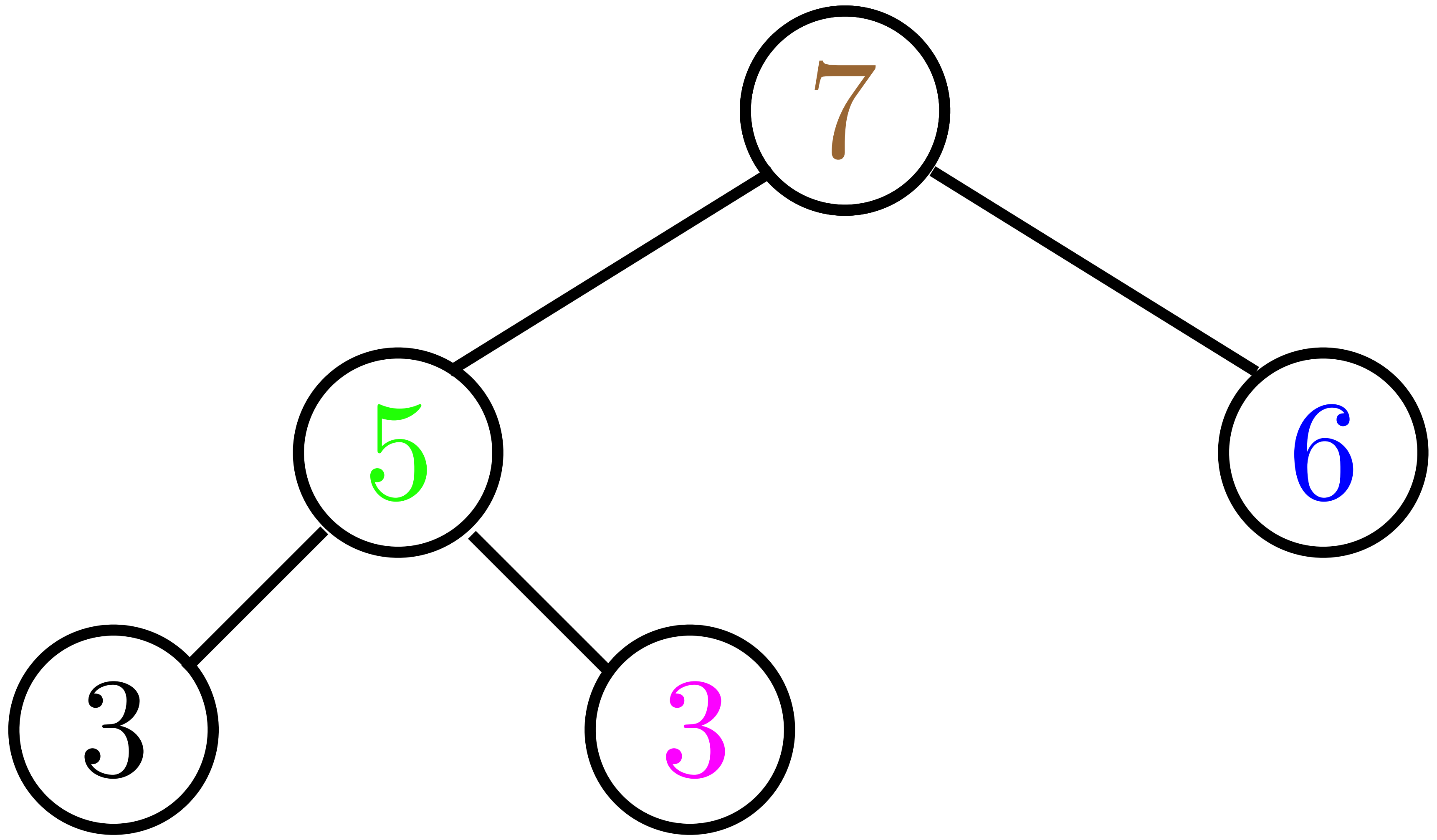
Next we insert 3 .



Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5



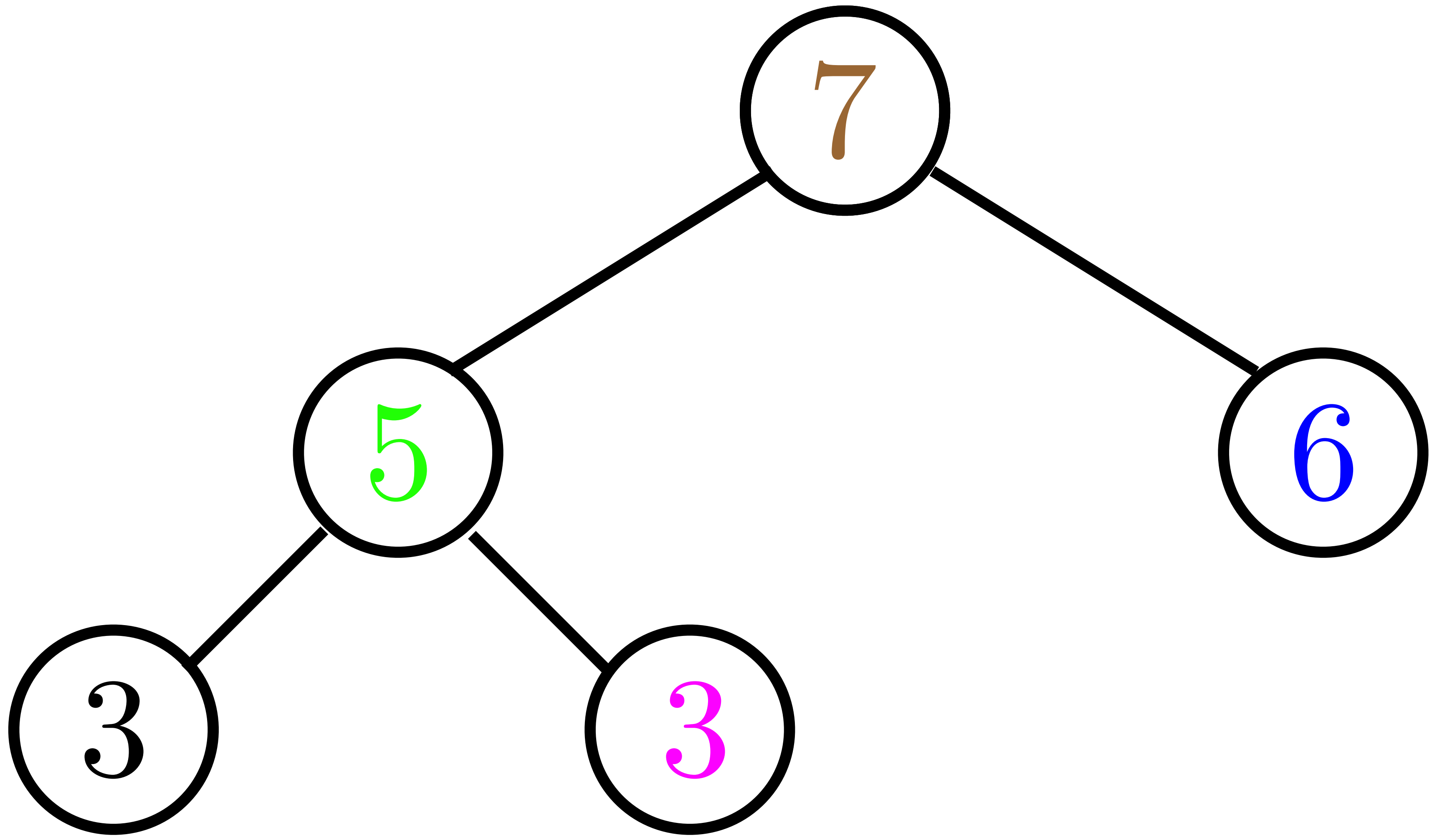
Next we insert 3 .

Now swim with 3 .

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5



Next we insert 3 .

Now swim with 3 .

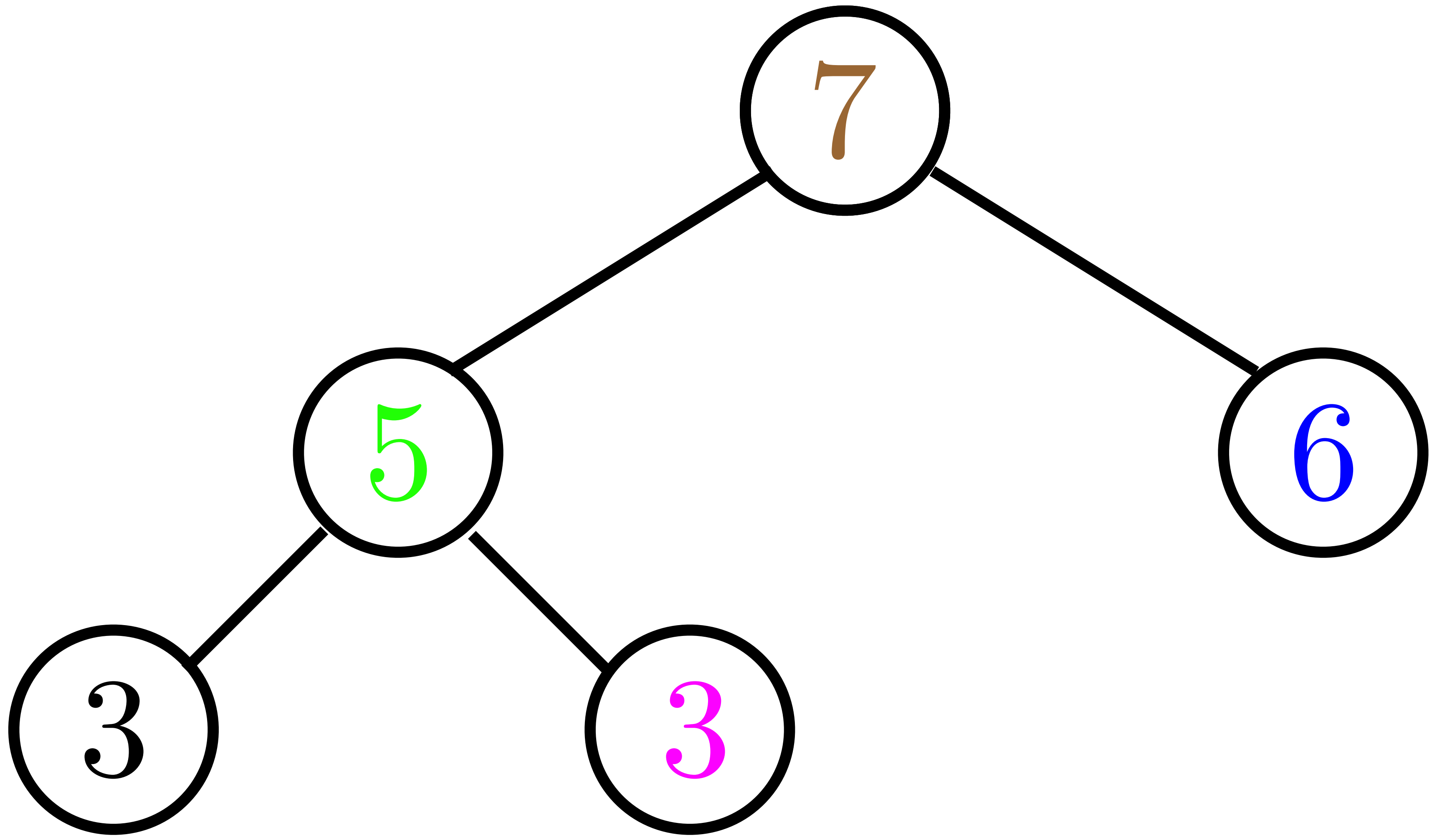
Is $5 < 3$? No, so we have a max heap on the first 5 elements.

Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

Finally, we insert 5 .

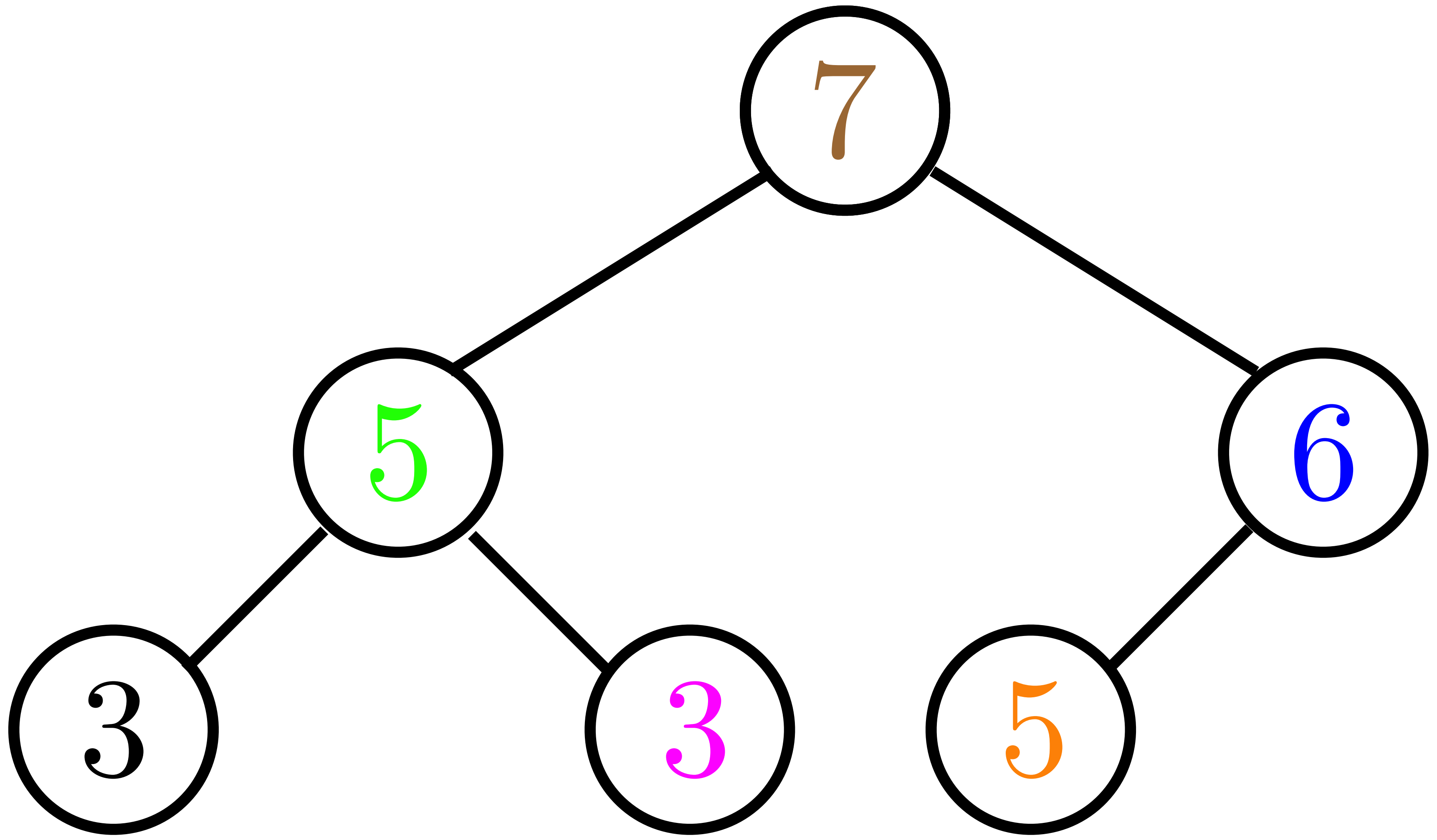


Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

Finally, we insert 5 .



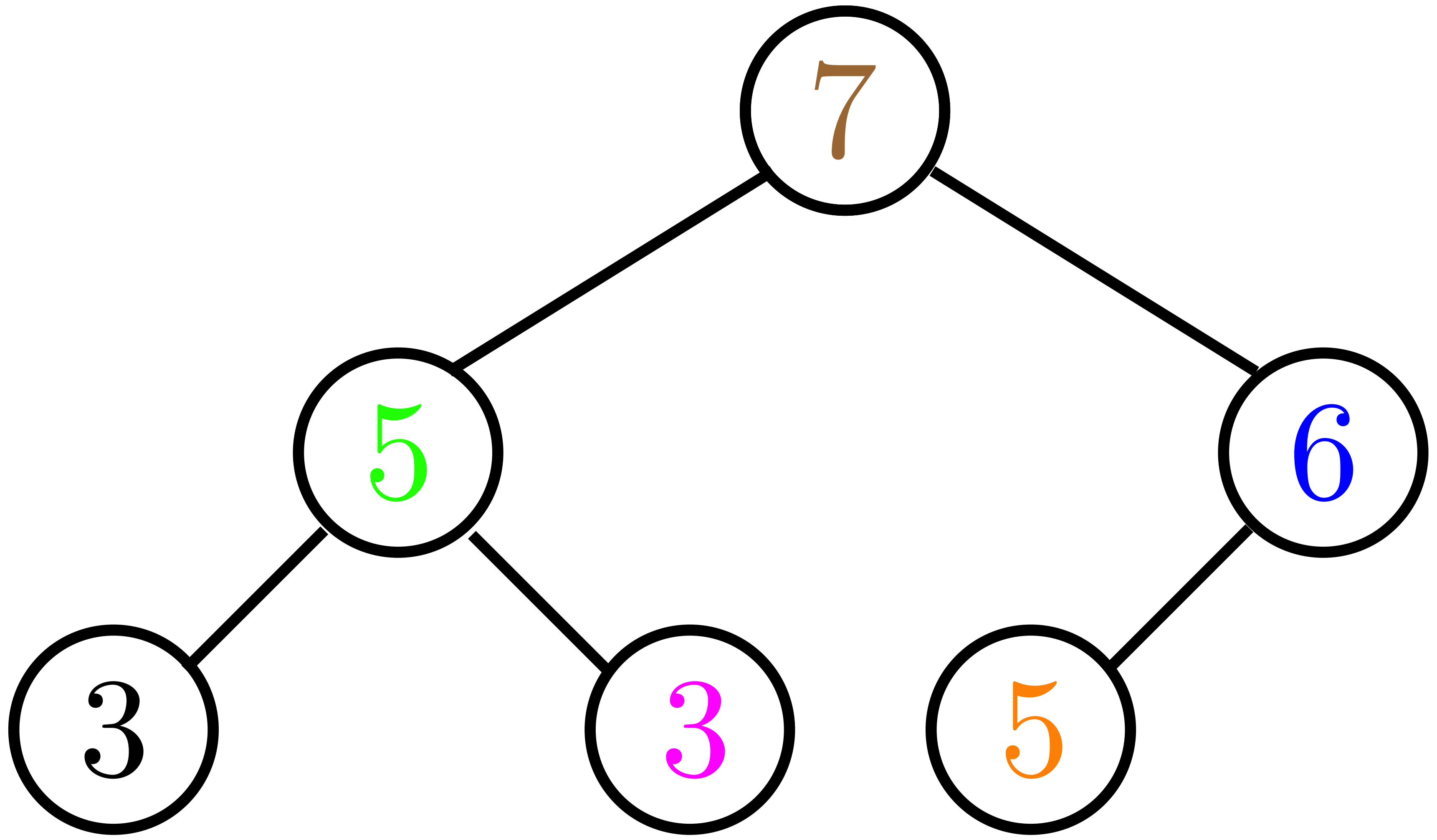
Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

Finally, we insert 5 .

Now swim with 5.



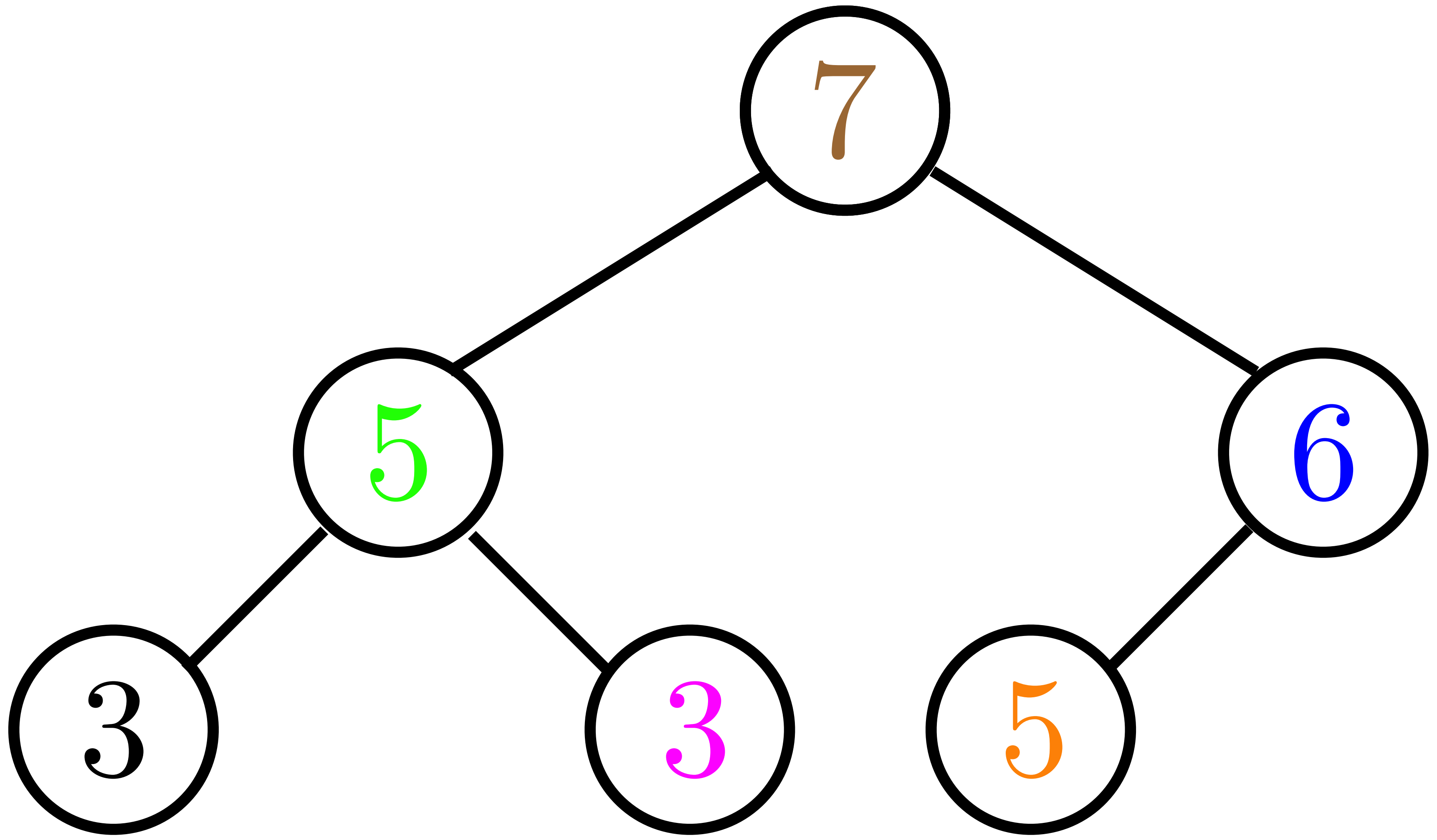
Heapsort

In the first phase we create a max heap with the elements of the vector.

	7	5	6	3	3	5

Finally, we insert 5 .

Now swim with 5.



Is $6 < 5$? No, so now we have a max heap with our initial vector.

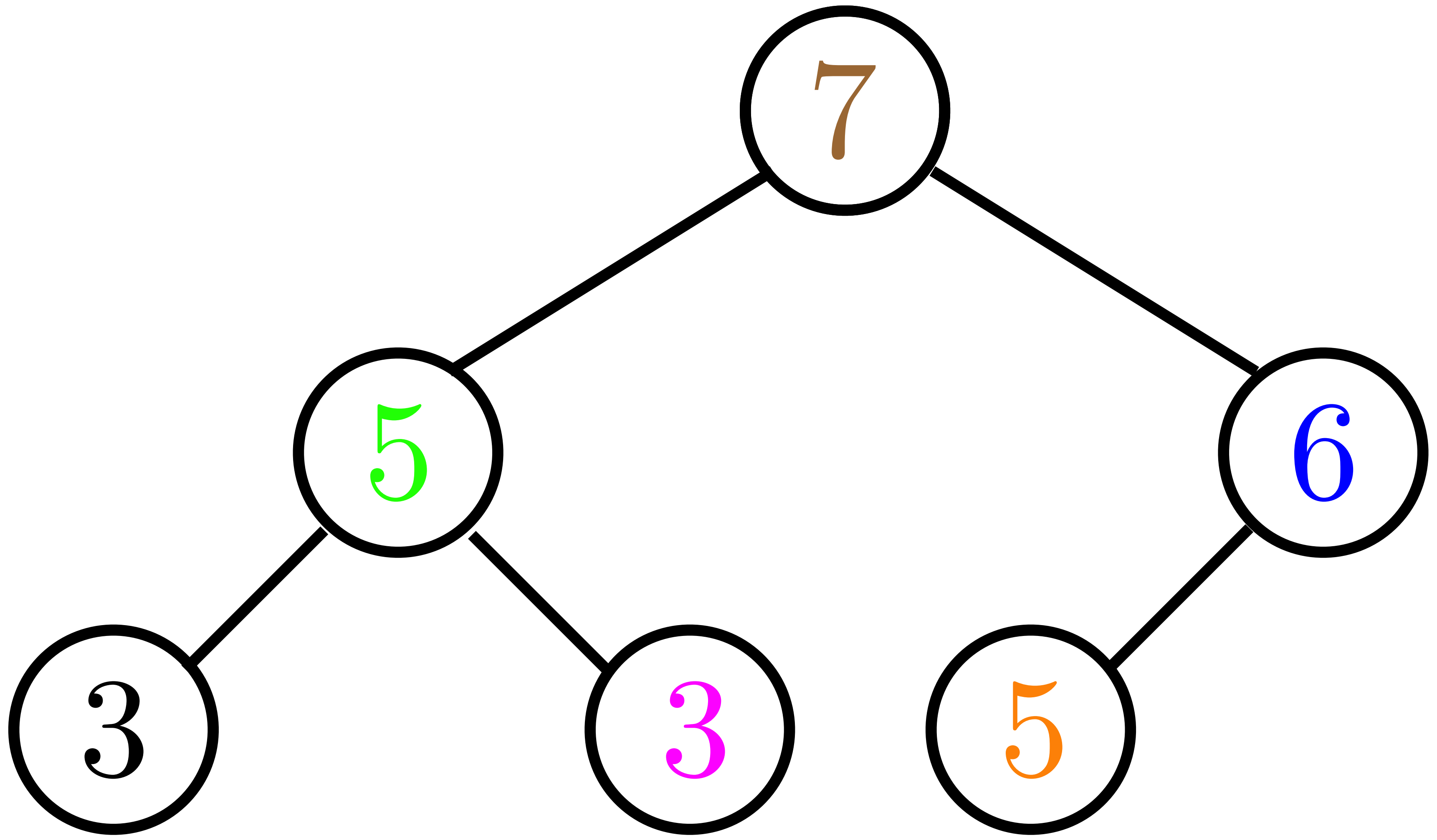
Heapsort

We have completed the first phase.

	7	5	6	3	3	5

We now pop the elements one by one.

After popping, we store the elements at the back of the vector.

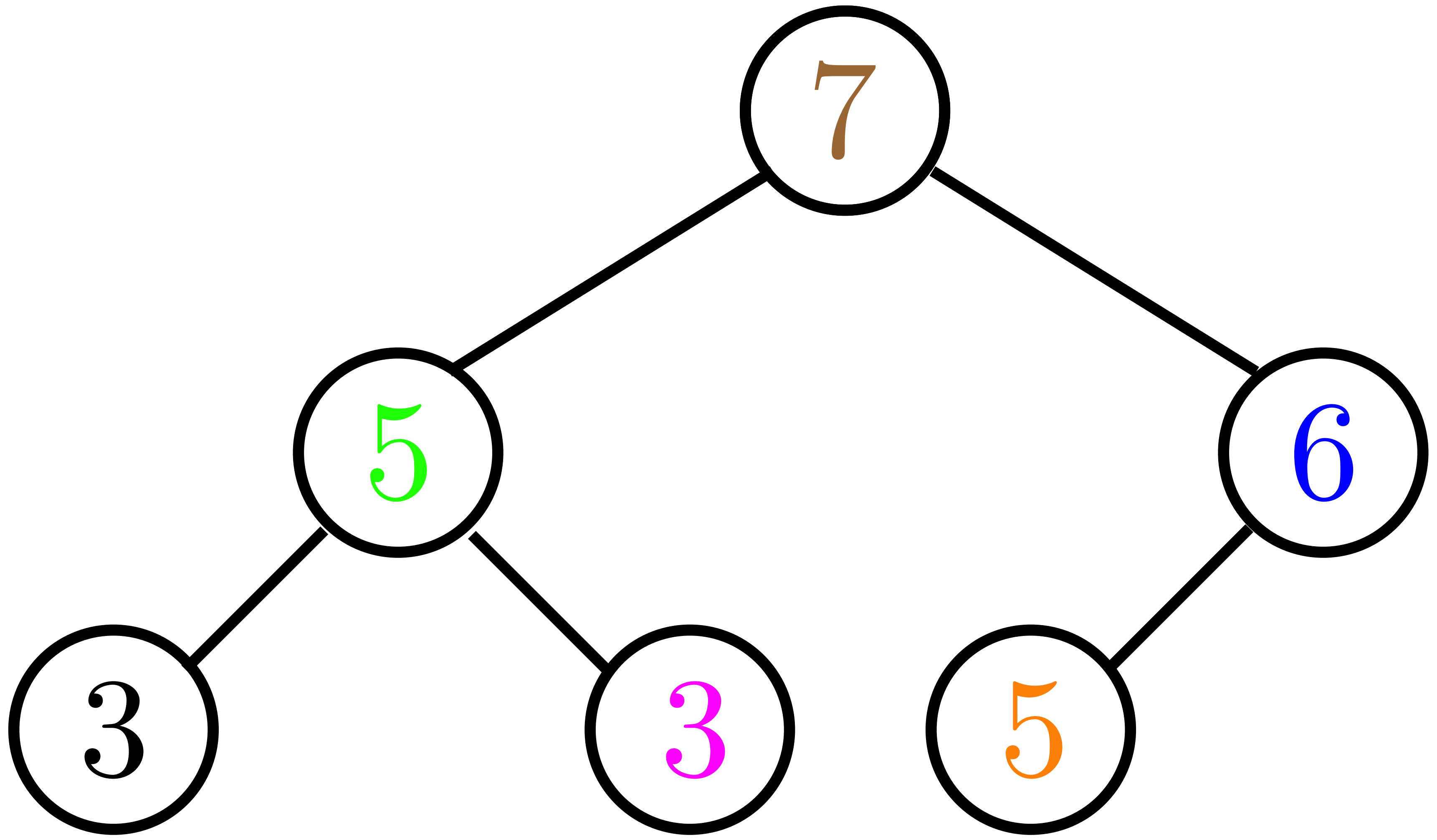


Heapsort

We have completed the first phase.

	7	5	6	3	3	5

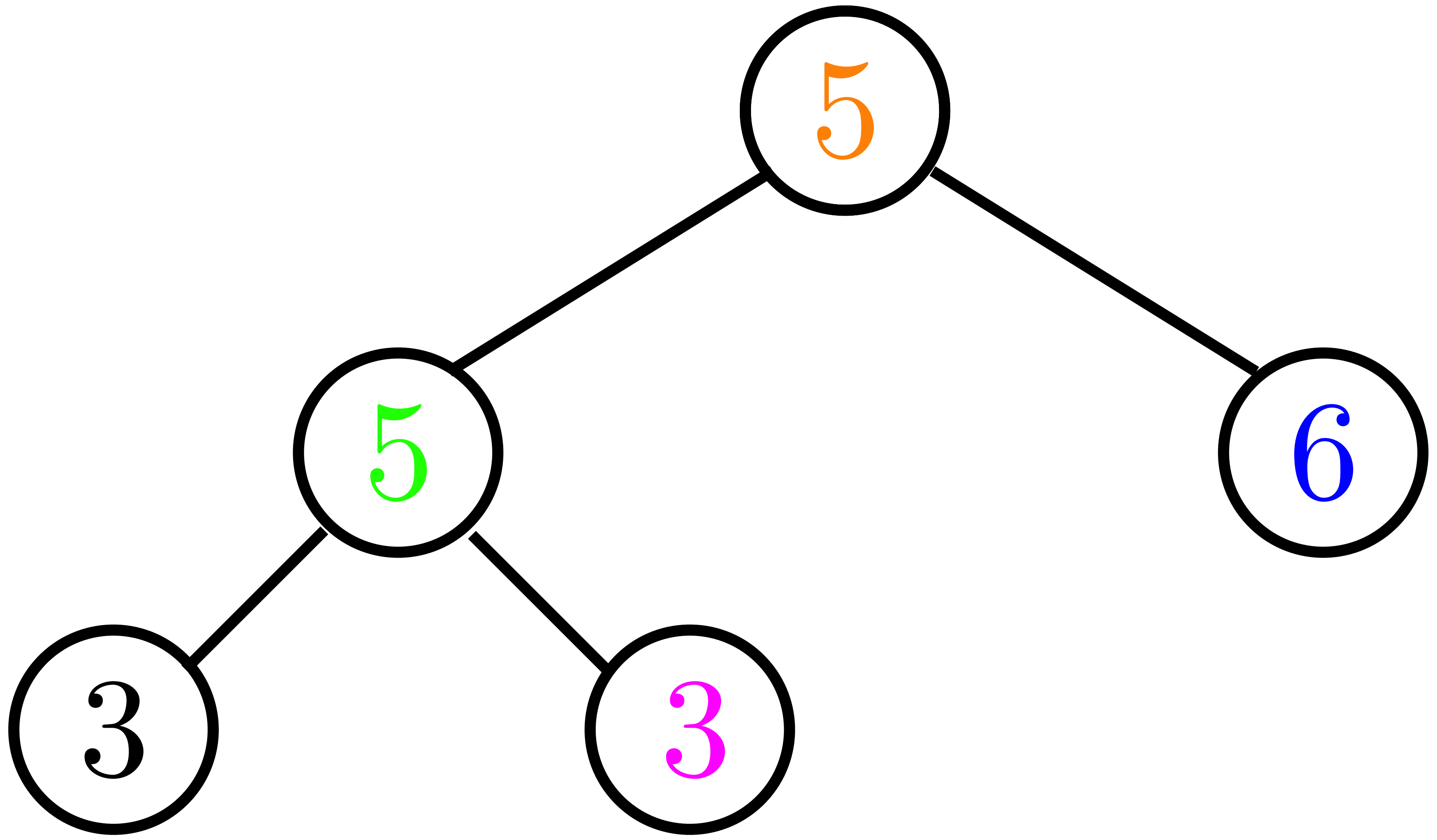
Pop 7. We replace 7 with 5.



Heapsort

We have completed the first phase.

	5	5	6	3	3	7



Pop 7. We replace 7 with 5.

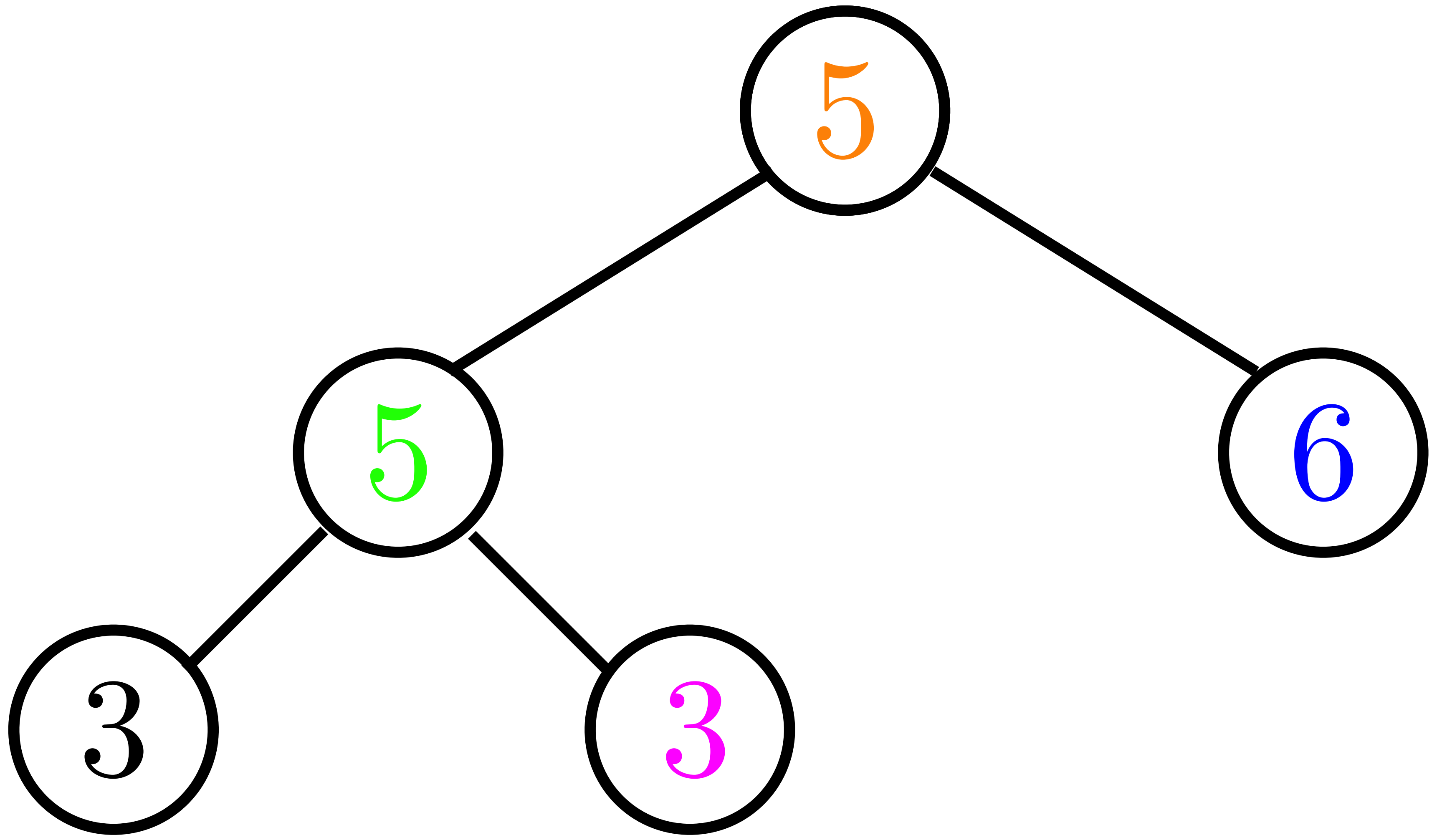
We store 7 at the old position of in the vector 5.

We no longer think of 7 as being in the heap. It will not move again.

Heapsort

We have completed the first phase.

	5	5	6	3	3	7



Pop 7. We replace 7 with 5.

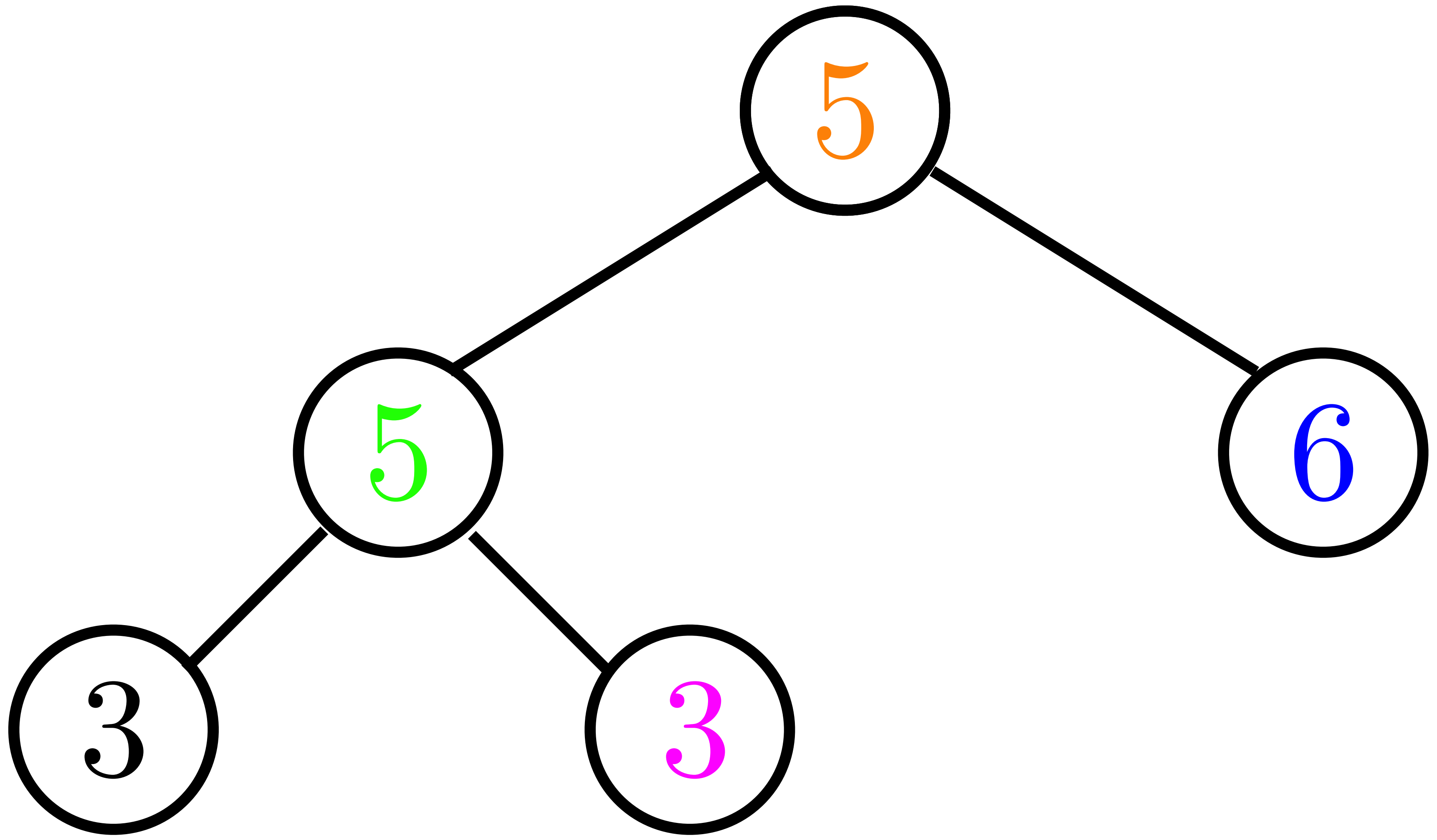
We store 7 at the old position of in the vector 5.

We no longer think of 7 as being in the heap. It will not move again.

Heapsort

We have completed the first phase.

	5	5	6	3	3	7



Now "sink" with 5 .

Is $5 < 6$? Yes, so we swap them.

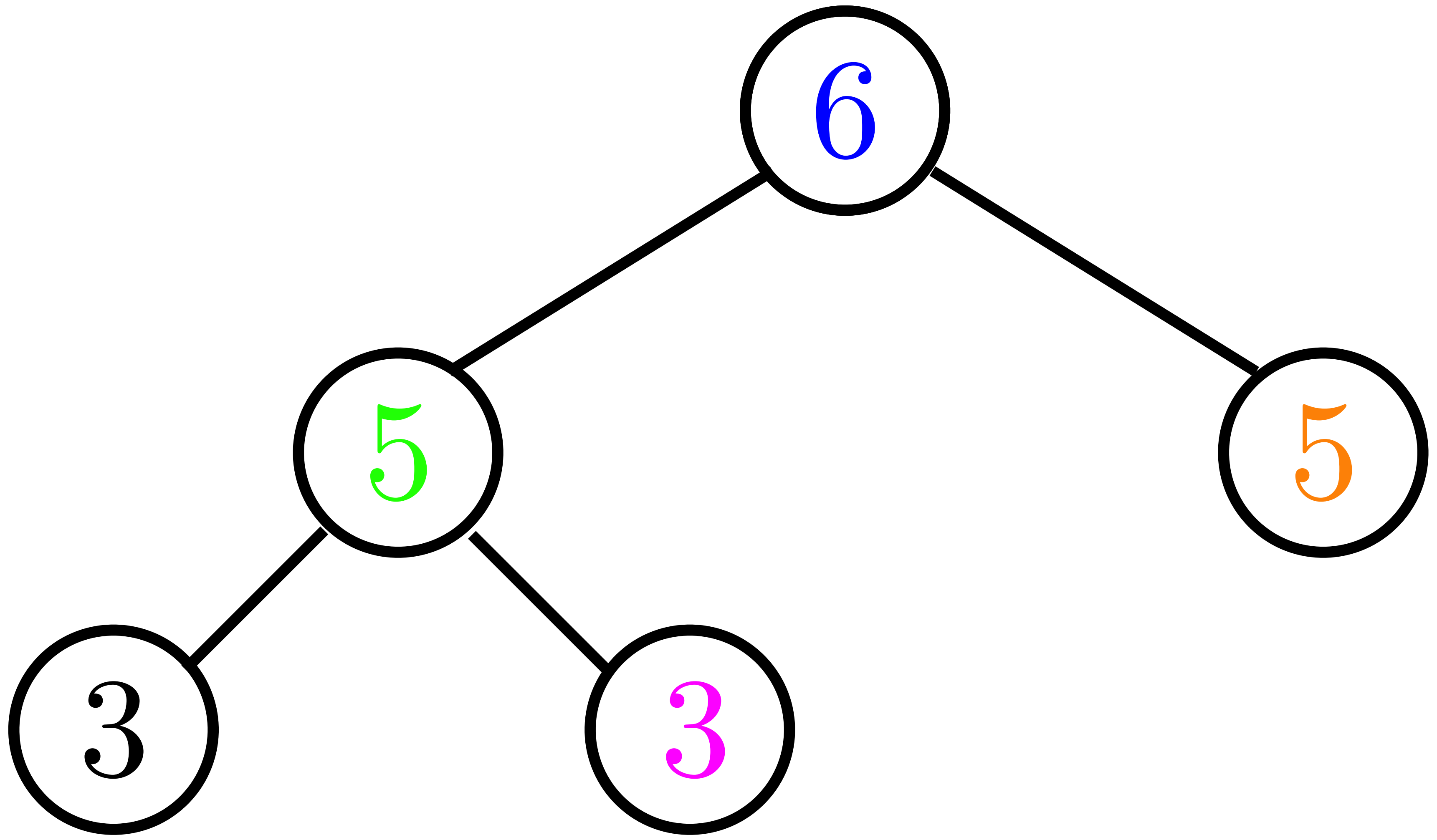
Heapsort

We have completed the first phase.

	6	5	5	3	3	7

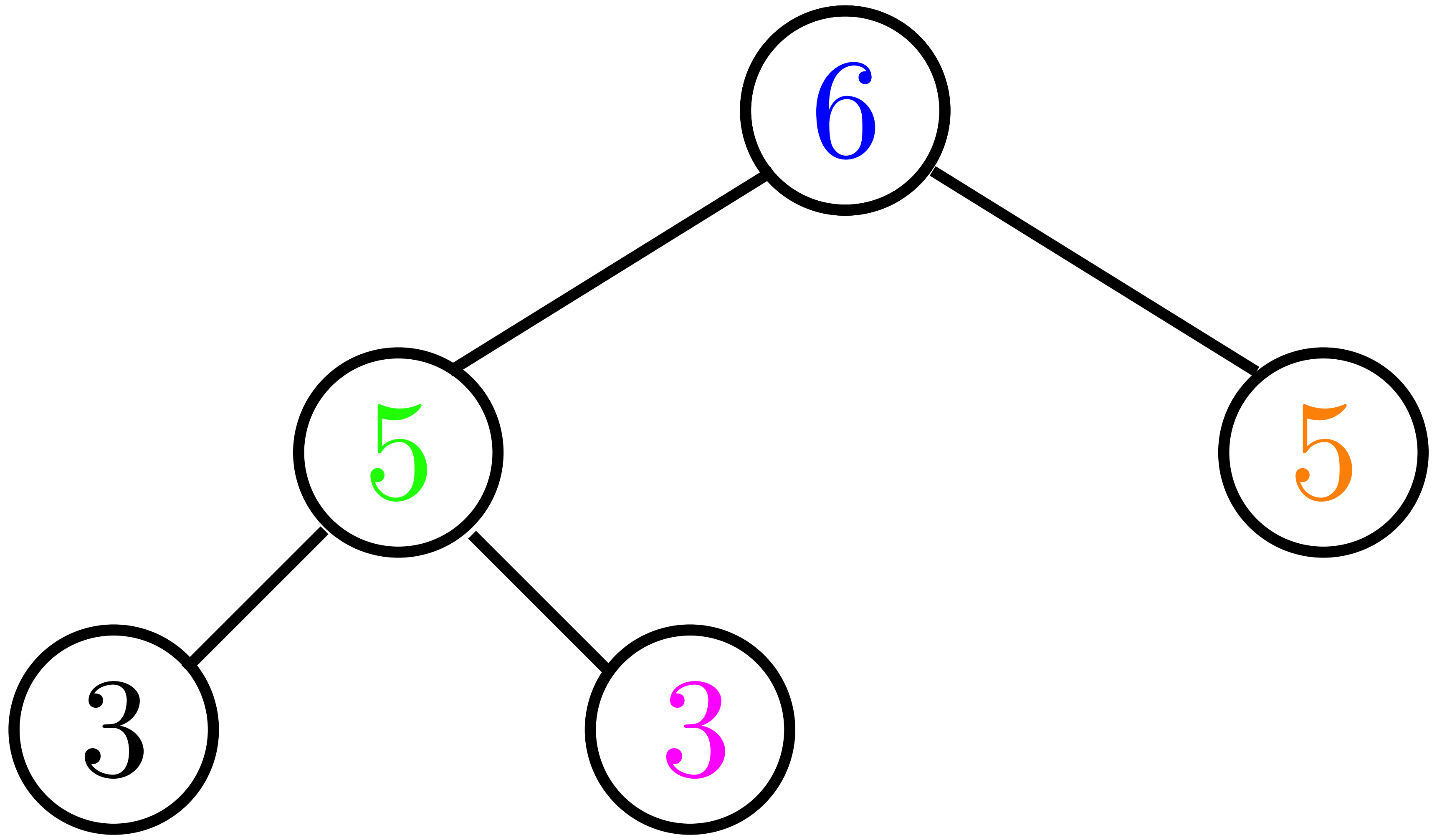
Now "sink" with 5 .

We have restored the max heap property.



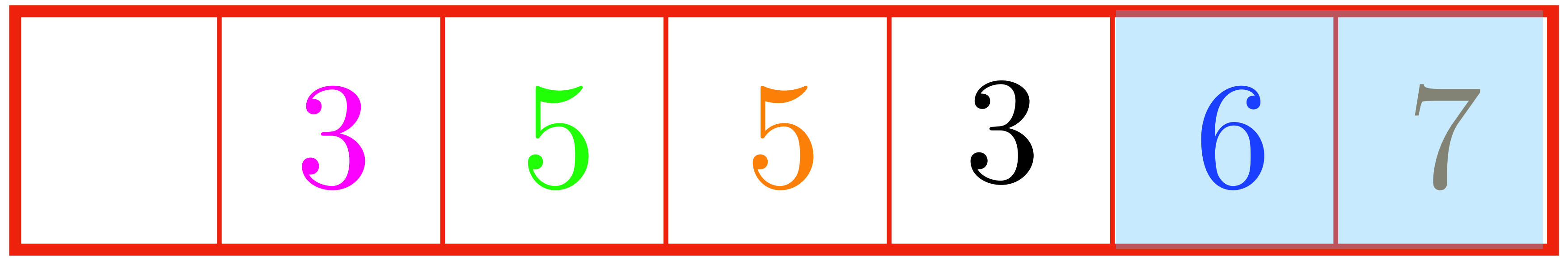
Heapsort

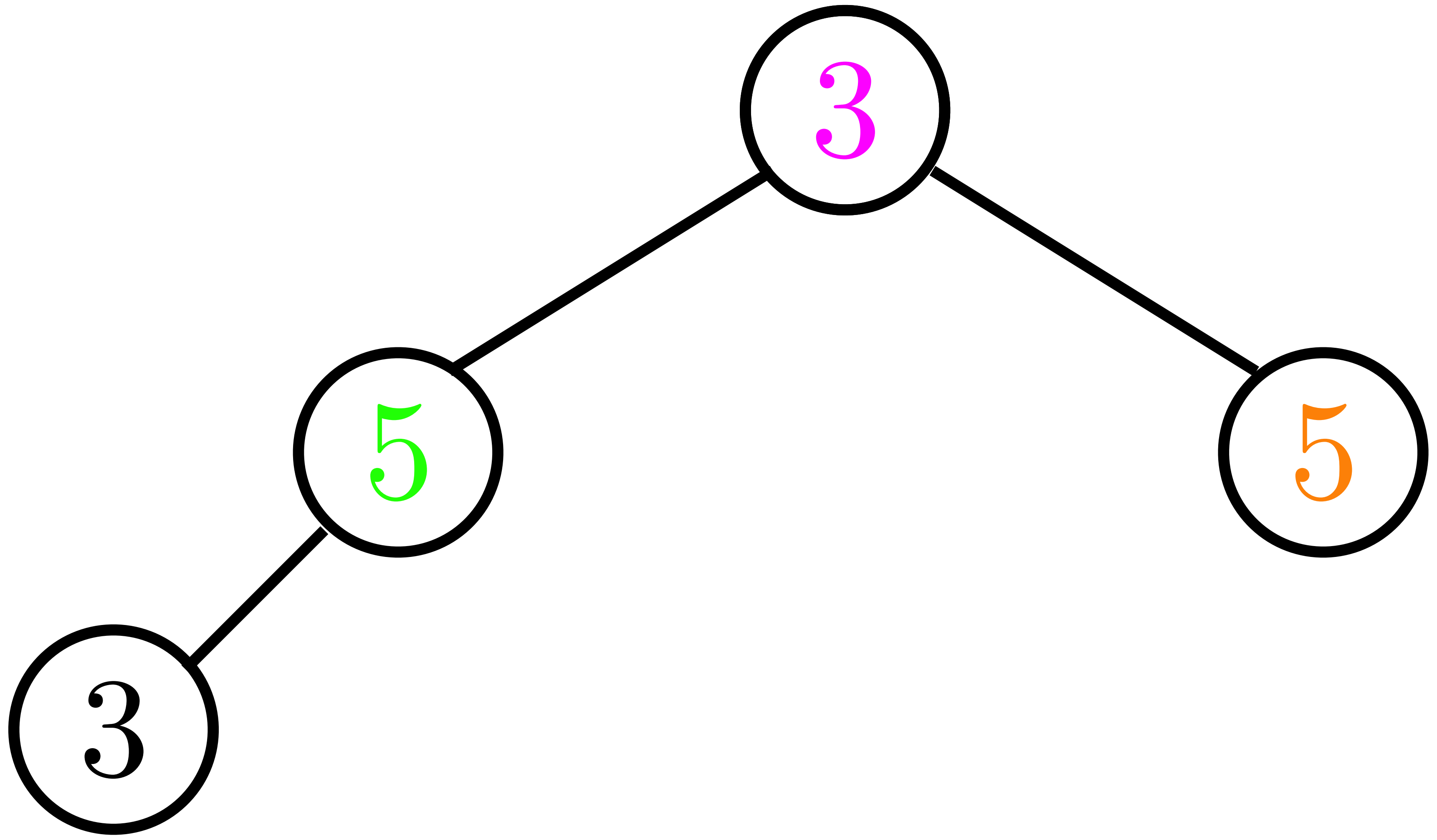
	6	5	5	3	3	7



Now we pop again. We swap 6 and 3 in the vector.

Heapsort



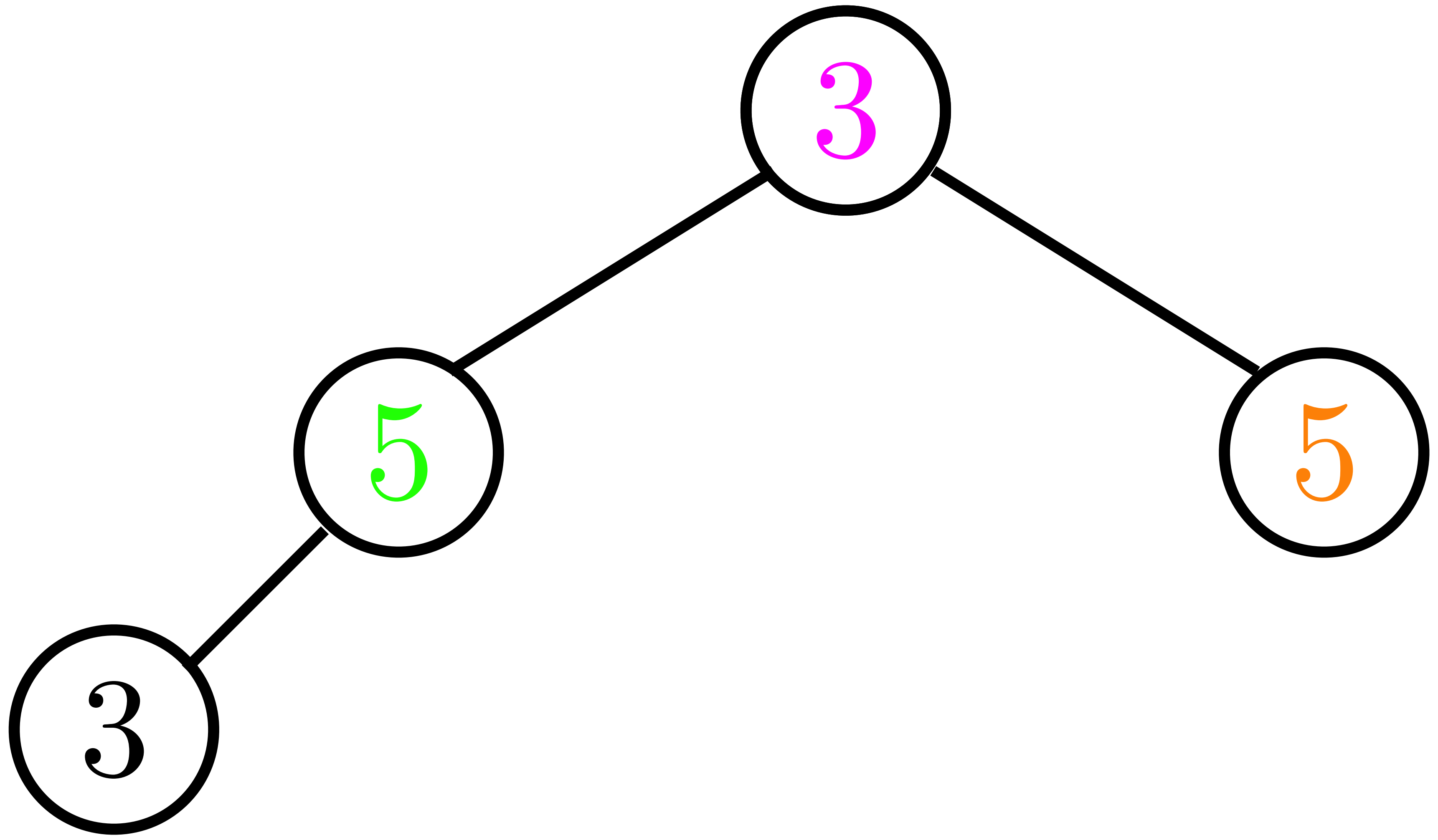


Now we pop again. We swap 6 and 3 in the vector.

We no longer consider 6 as being in the heap. It will not move again.

Heapsort

	3	5	5	3	6	7

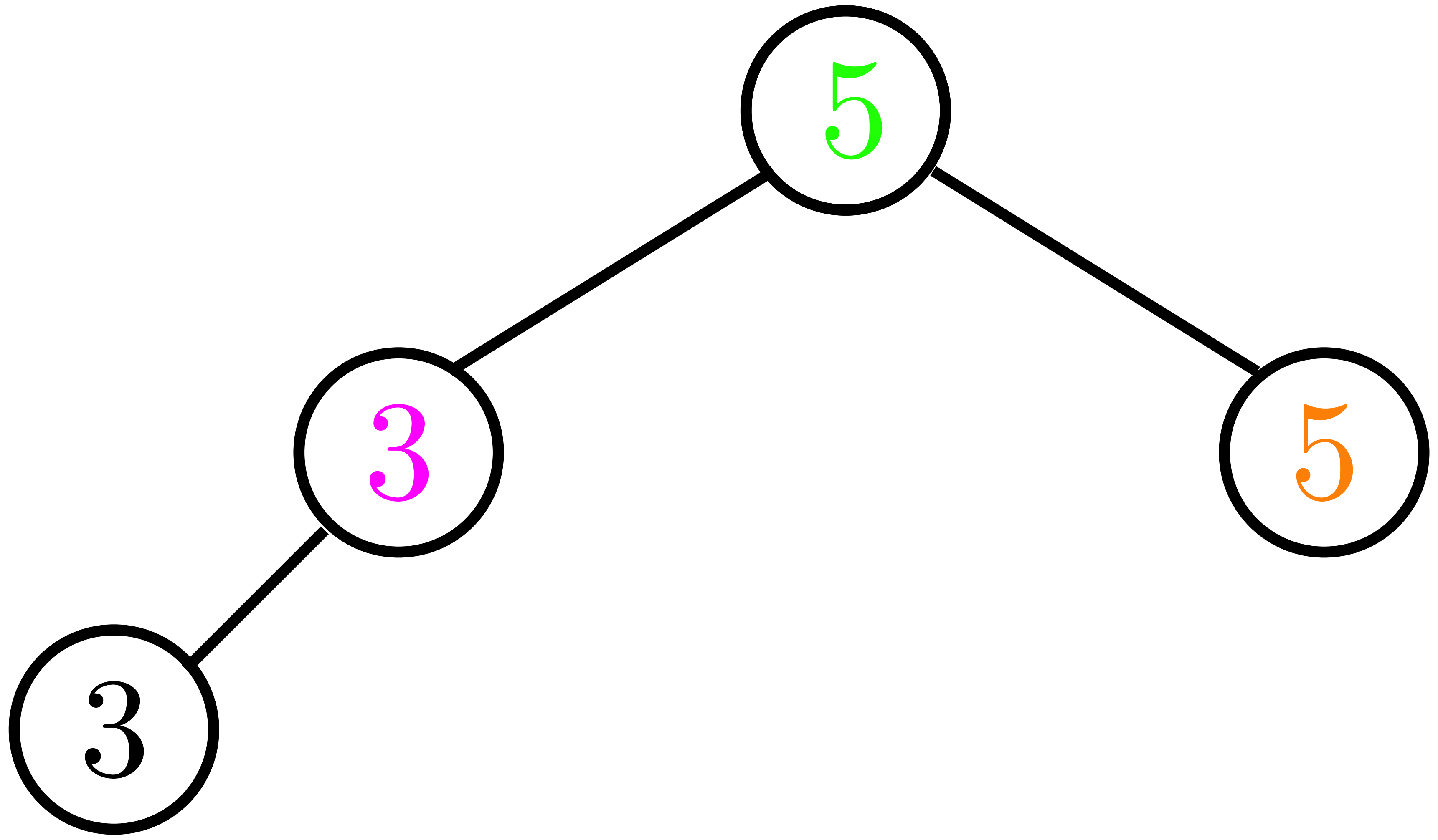


Now we sink with 3 .

Is $3 < 5$? Yes, so we swap them.

Heapsort

	5	3	5	3	6	7

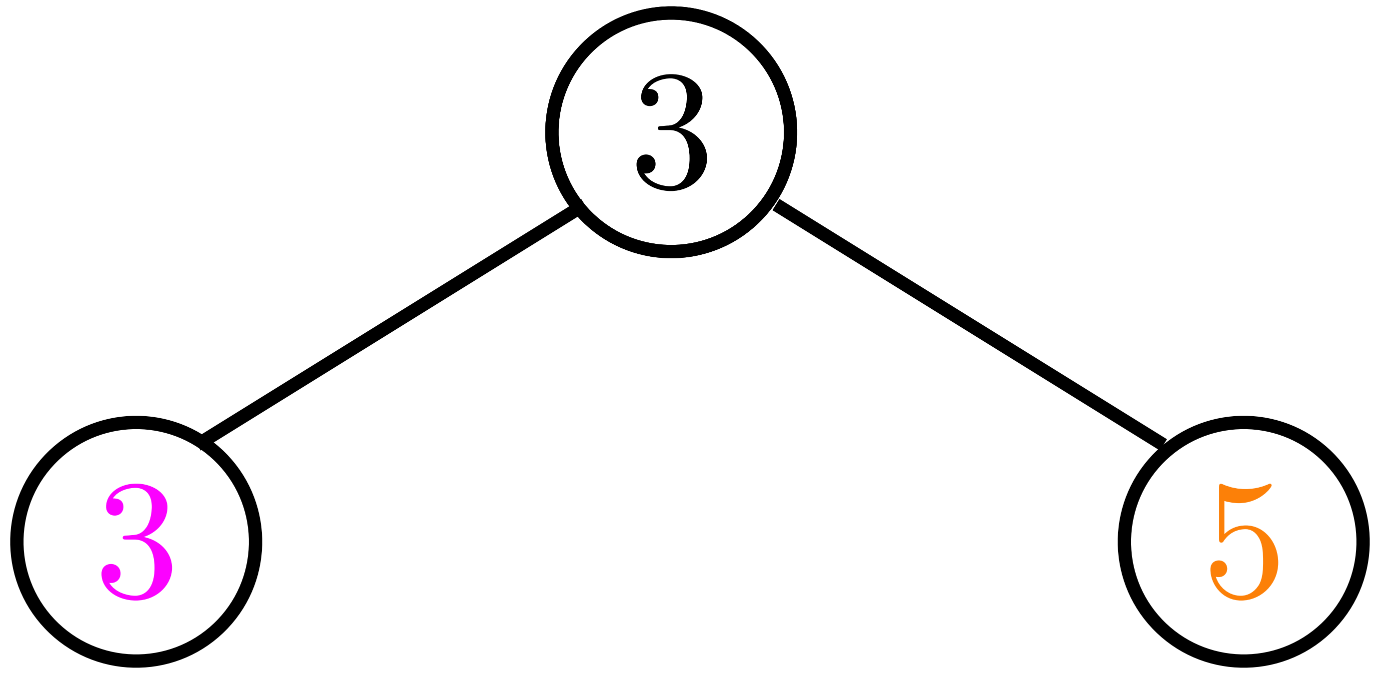


We have restored the heap property.

Next we pop 5 .

Heapsort

	3	3	5	5	6	7

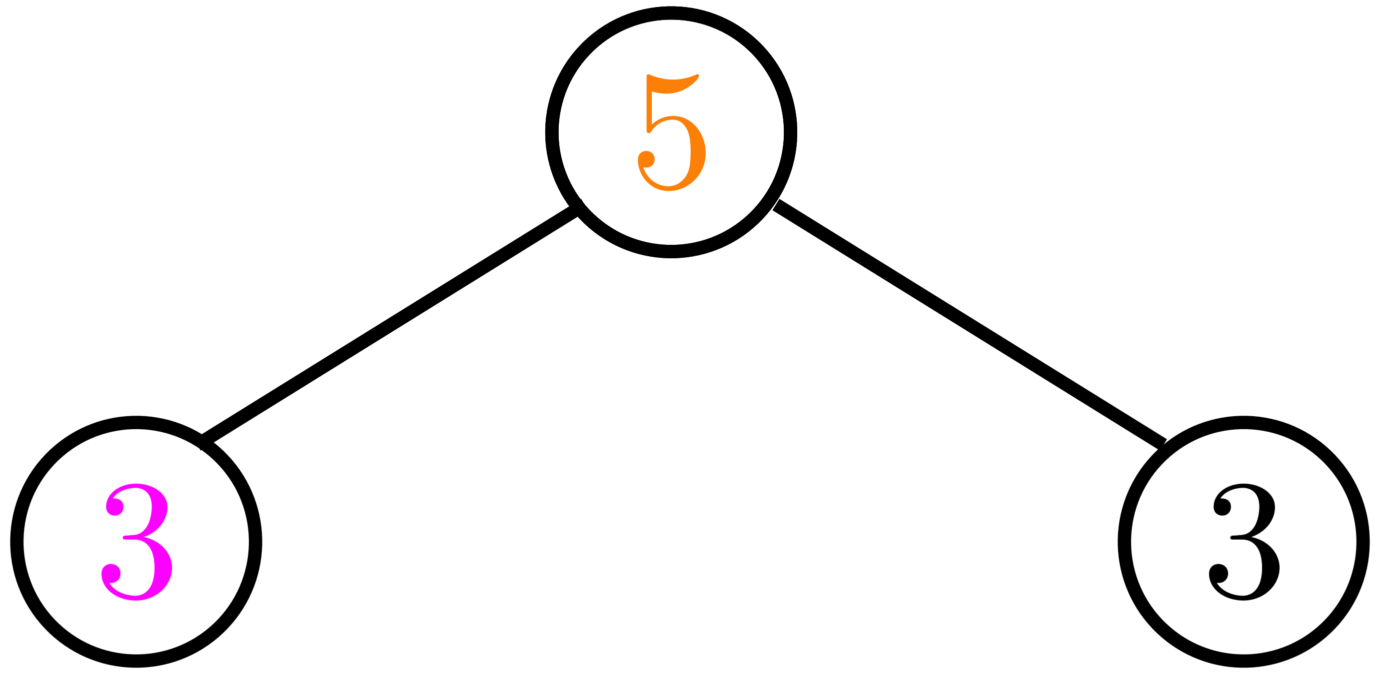


And sink with 3 .

We swap 3 and 5.

Heapsort

	5	3	3	5	6	7

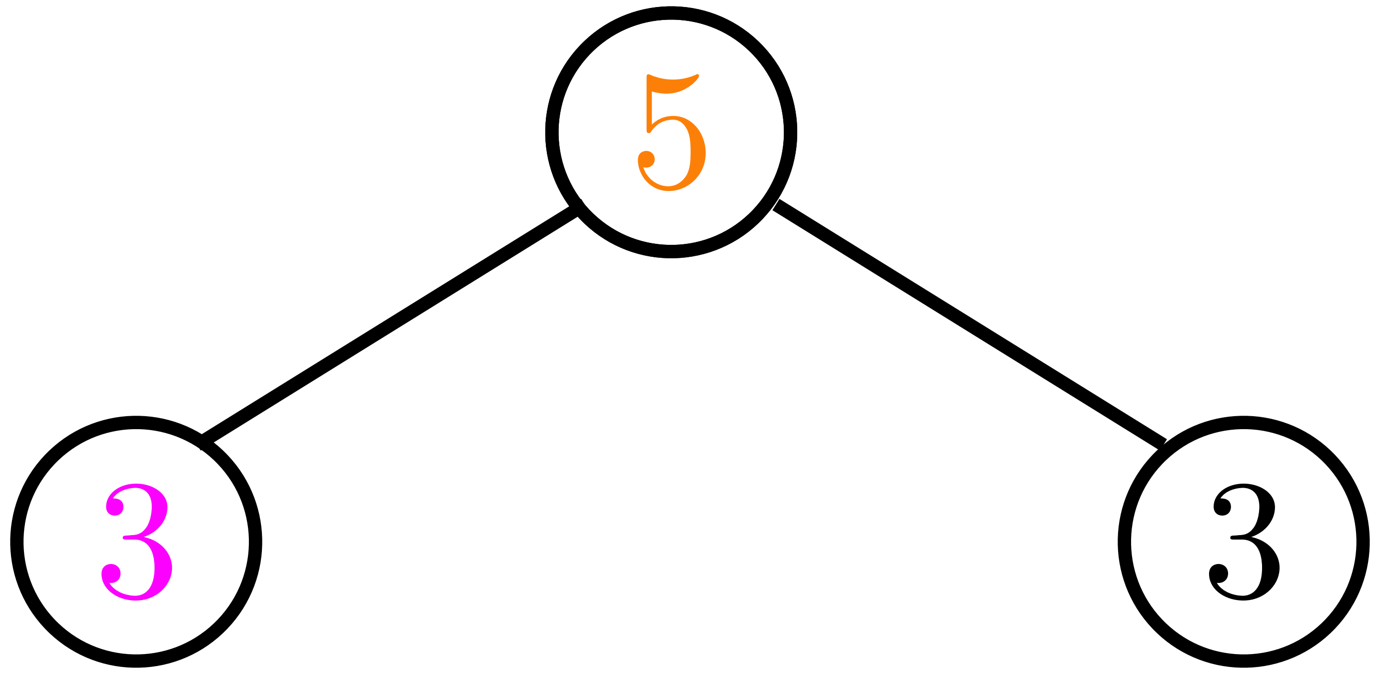


And sink with 3 .

We swap 3 and 5.

Heapsort

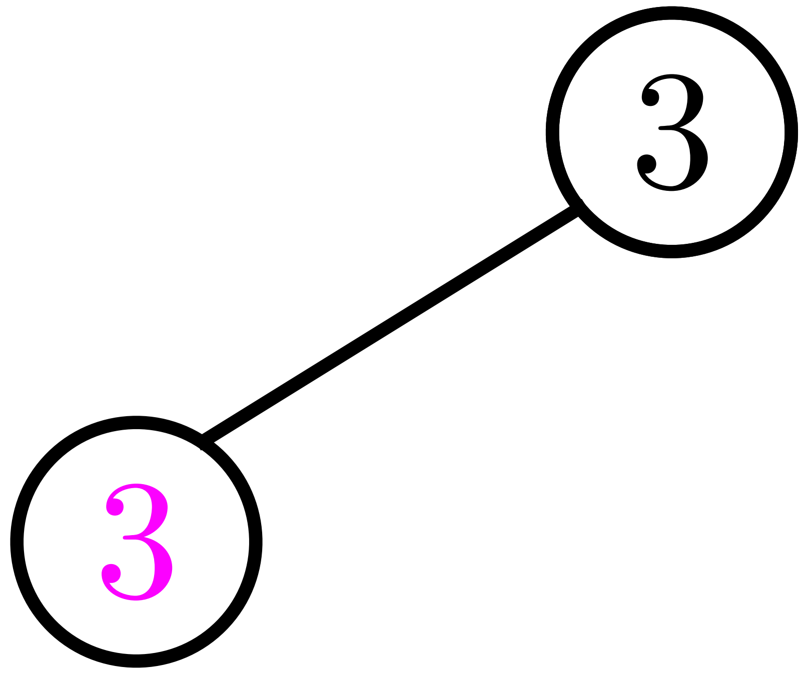
	5	3	3	5	6	7



Pop 5.

Heapsort

	3	3	5	5	6	7



Pop 3.

Heapsort

	3	3	5	5	6	7

Pop 3.

Heapsort

	3	3	5	5	6	7

Now our vector is in sorted order.

What is the time complexity of heap sort?

Properties

input vector

	3	7	6	5	3	5

sorted vector

	3	3	5	5	6	7

Heapsort is not a stable sorting algorithm. The order of keys with the same value is not necessarily the same in the output as in the input.

Properties

input vector		3	7	6	5	3	5
sorted vector		3	3	5	5	6	7

Heapsort is not a stable sorting algorithm. The order of keys with the same value is not necessarily the same in the output as in the input.

Is heapsort a comparison based sorting algorithm?

Comparison-based sort

Any comparison based sorting algorithm must make at least a constant times $n \log n$ comparisons in the worst case.

Heapsort is optimal with respect to number of comparisons.

We cannot expect to have worst case $O(1)$ insert and pop operations on a heap.

In-Place

Heapsort is also an in-place sorting algorithm.

In-Place

Heapsort is also an in-place sorting algorithm.

We just needed a constant number of helper variables, in addition to the original input array.

Intro Sort

Implementation of the standard library sorting algorithm `std::sort` typically uses an algorithm called introspective sort.

It starts out doing quicksort, but if this takes too long it switches to heapsort.

This allows it to have $O(n \log n)$ worst-case running time (which is required by the standard since C++11).

Representations of Graphs

Graphs

What are the main flavours of edge types in graphs?

Graph Operations

What operations might you want a graph data structure to support?

Graph Representation

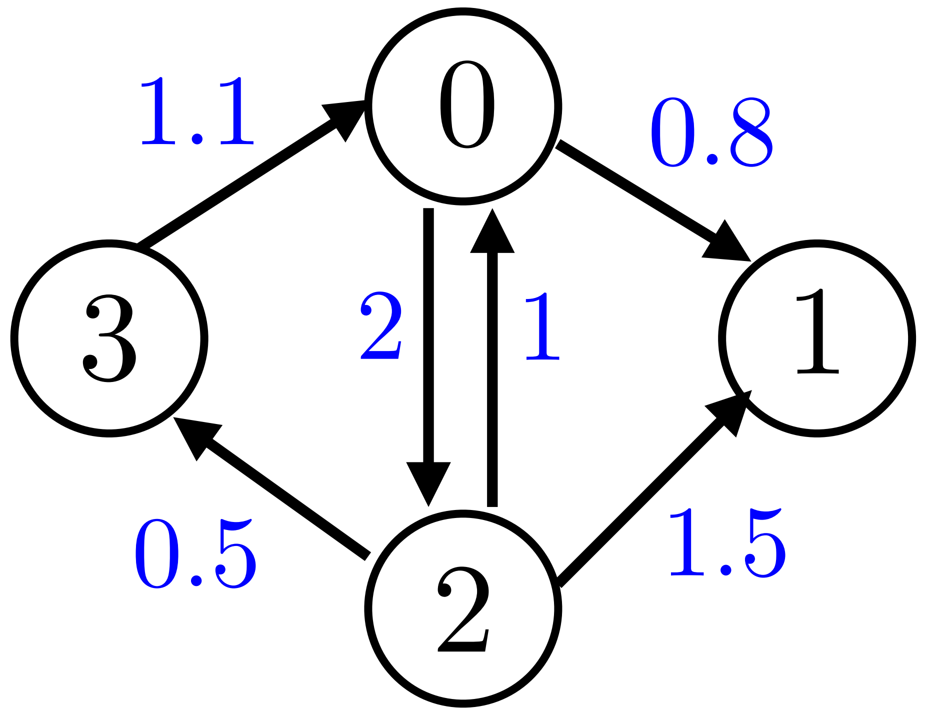
What are the two main data structures for representing a graph?

Graph Representation

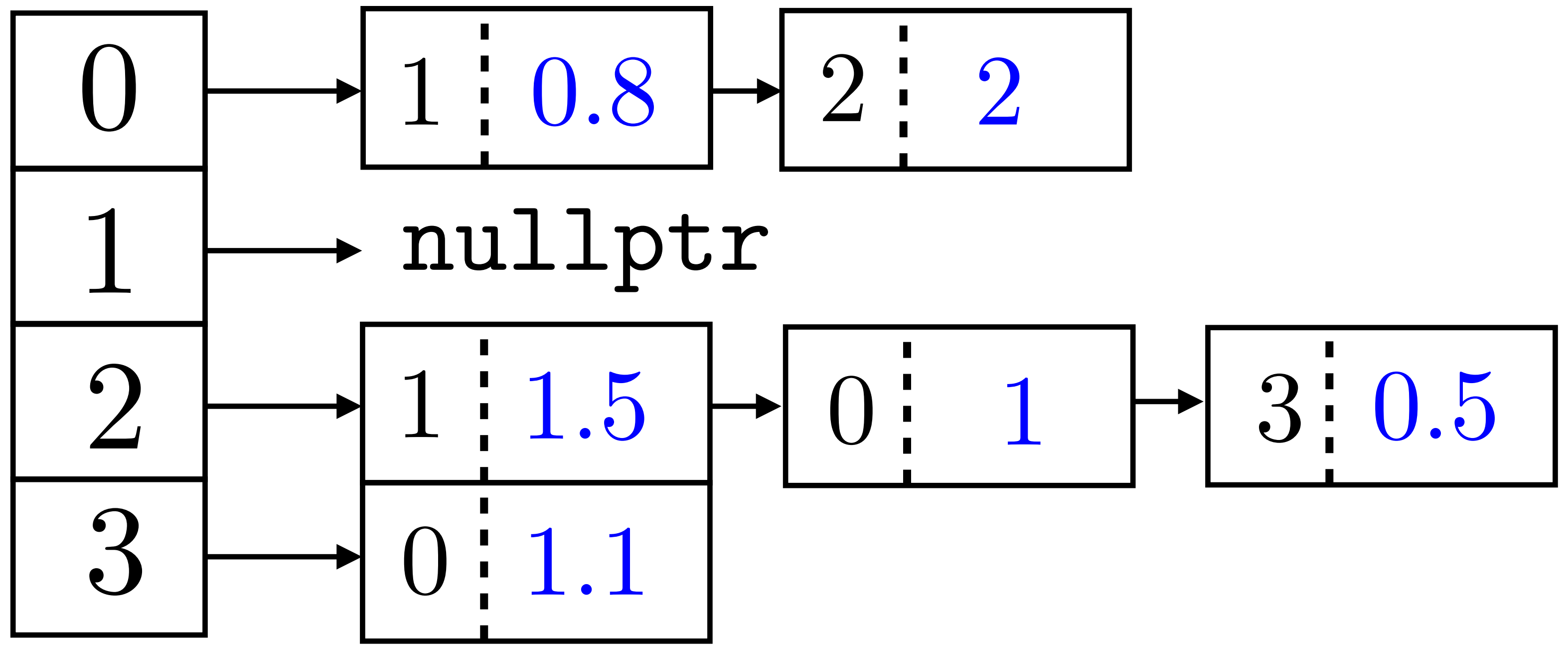
What are the two main data structures for representing a graph?

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$

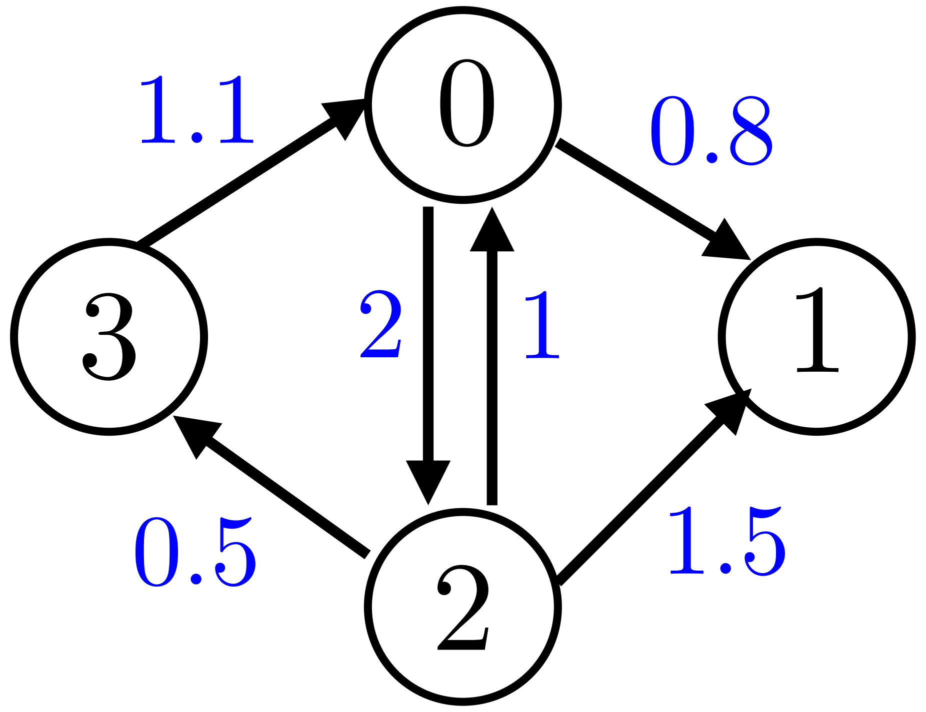


Adjacency list

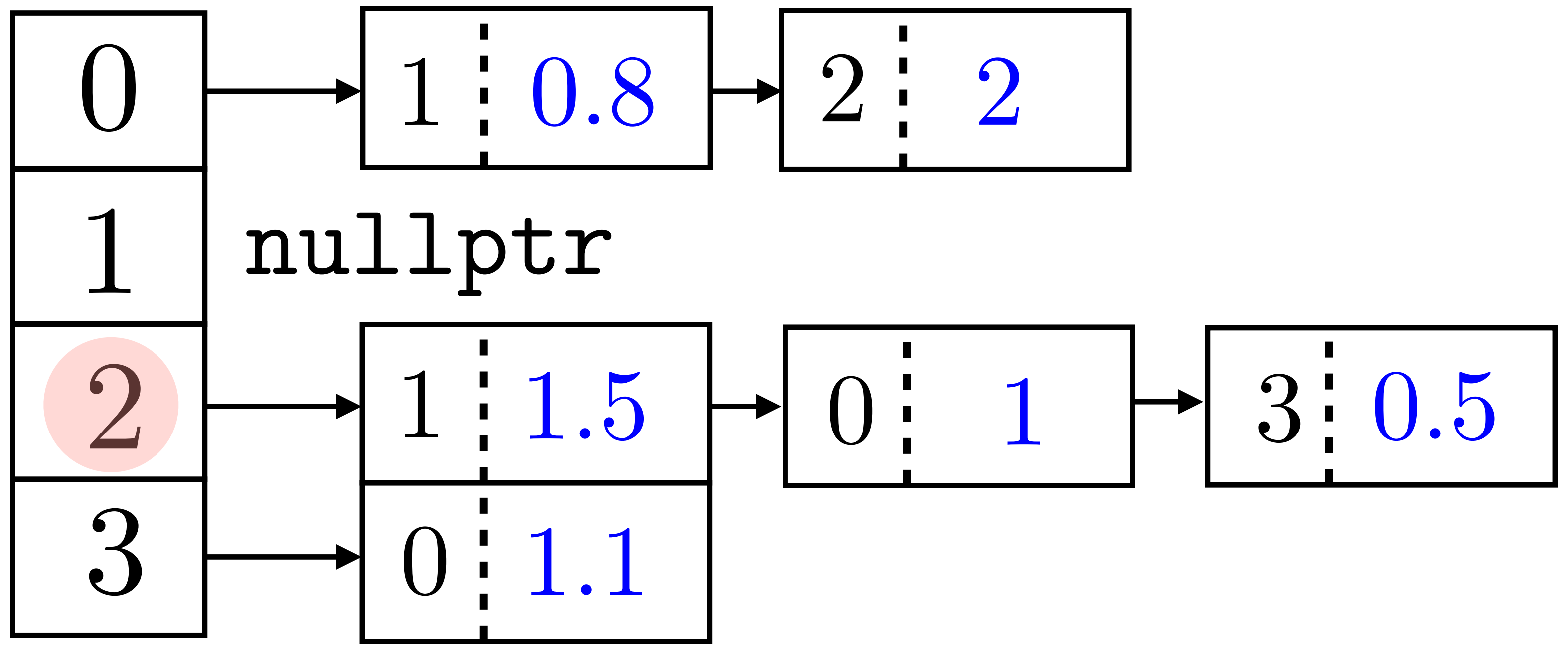


Adjacency matrix

$$A = \begin{bmatrix} 0 & 0.8 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1.5 & 0 & 0.5 \\ 1.1 & 0 & 0 & 0 \end{bmatrix}$$



Adjacency list



	Adj. matrix	Adj. list
size	$\Theta(n^2)$	$\Theta(n+m)$
add edge	$\Theta(1)$	$\Theta(1)$
edge (i,j) ?	$\Theta(1)$	$\Theta(\deg(i))$
list out-adjacent to v	$\Theta(n)$	$\Theta(\deg(v))$

Graph Traversals

Graph Traversals

A graph traversal is a way to visit every vertex in the graph.

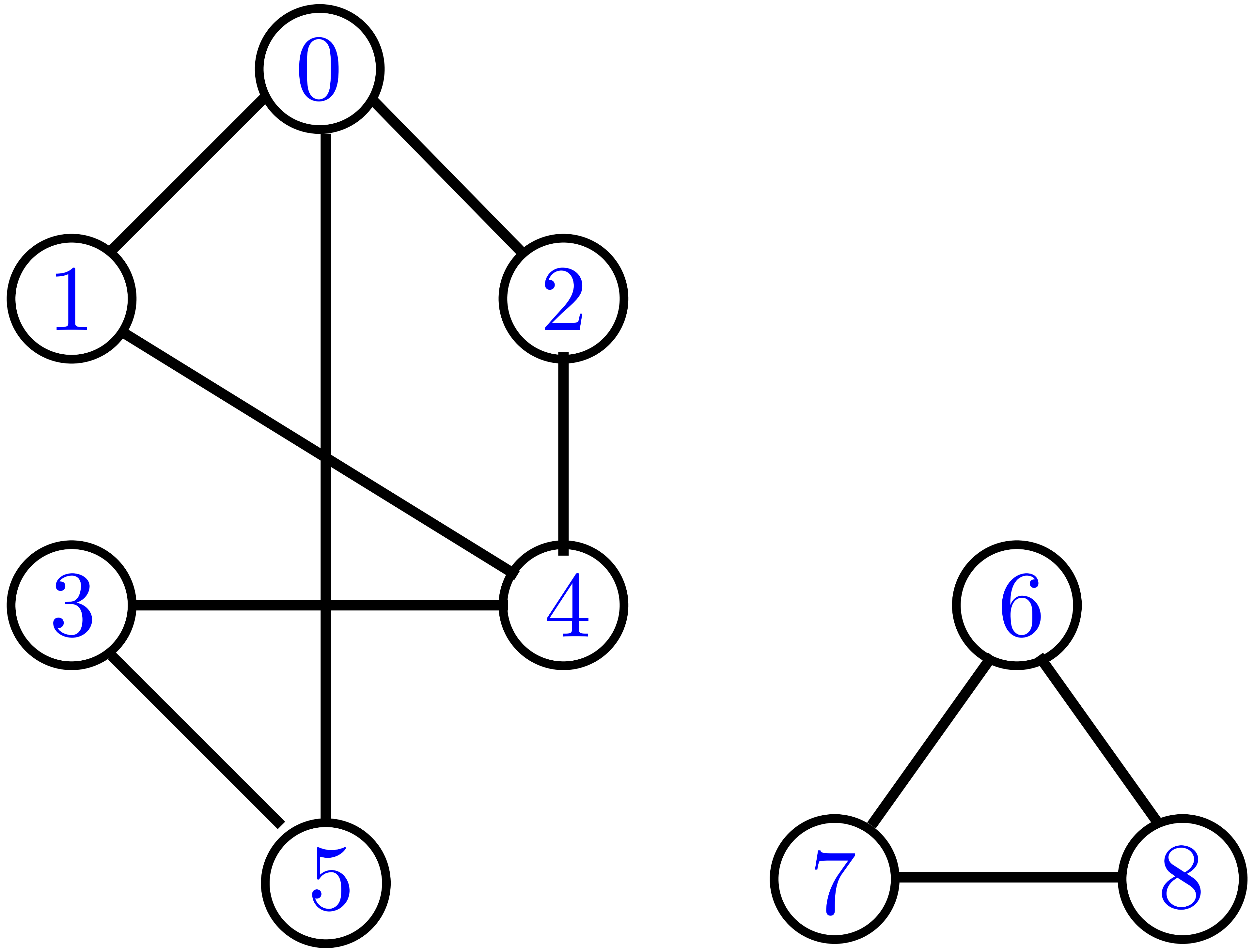
We start at one vertex and visit all of the other vertices reachable from that vertex.

Vertex u is reachable from v if there is a (directed) path from v to u .

This routine can be called from an outer loop to visit all the vertices in the graph.

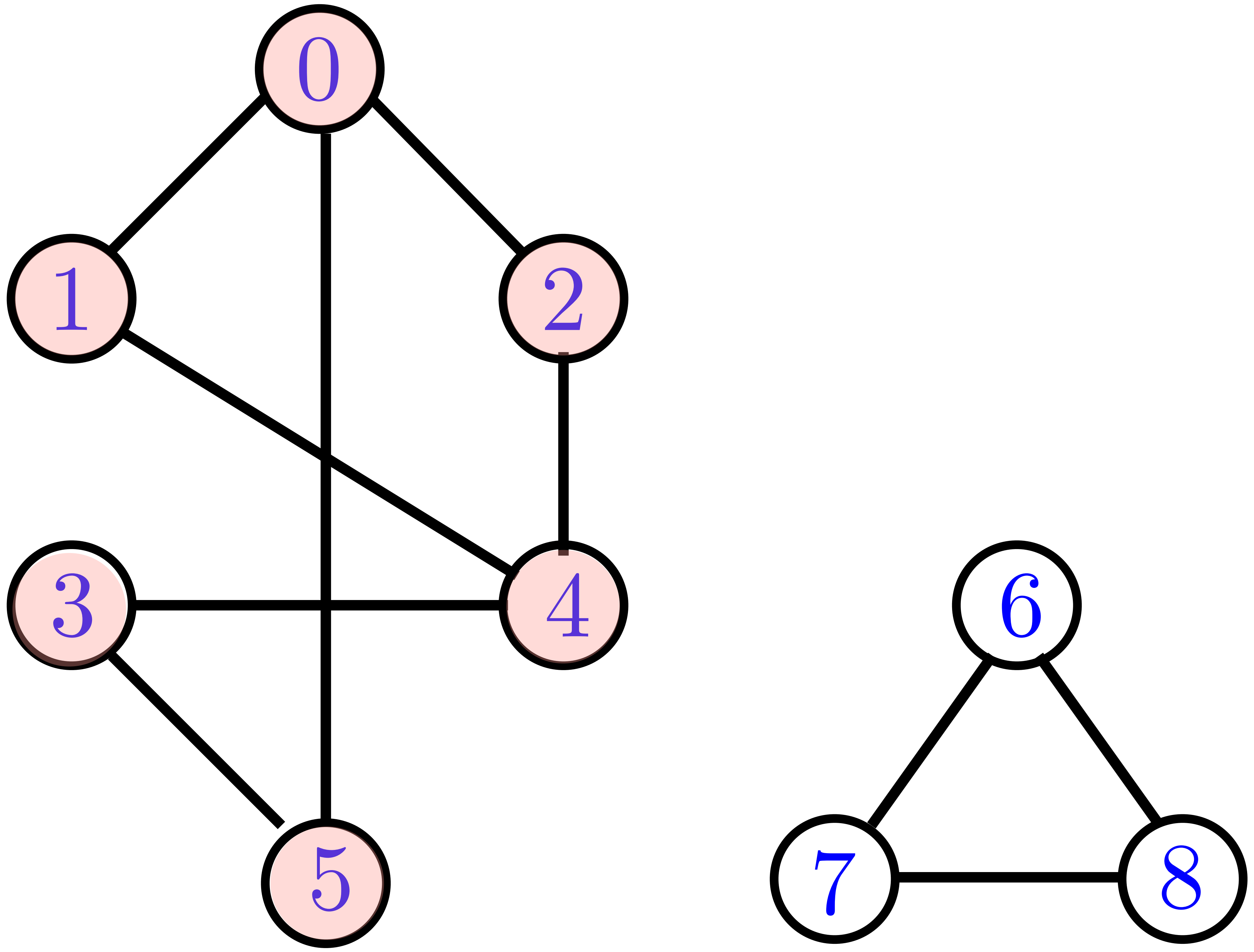
Undirected Graph

Which vertices are reachable from vertex 4?



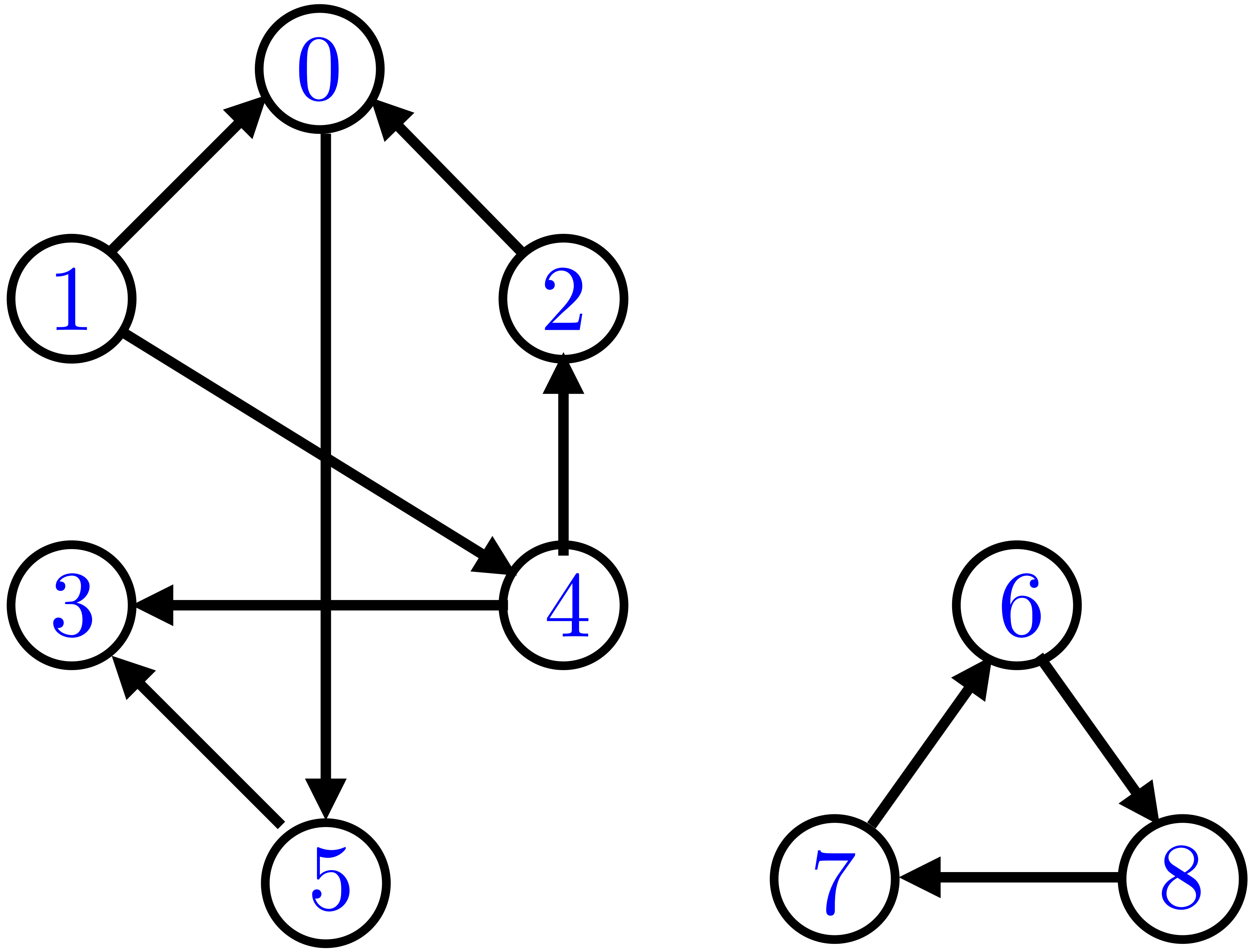
Undirected Graph

Which vertices are reachable from vertex 4?



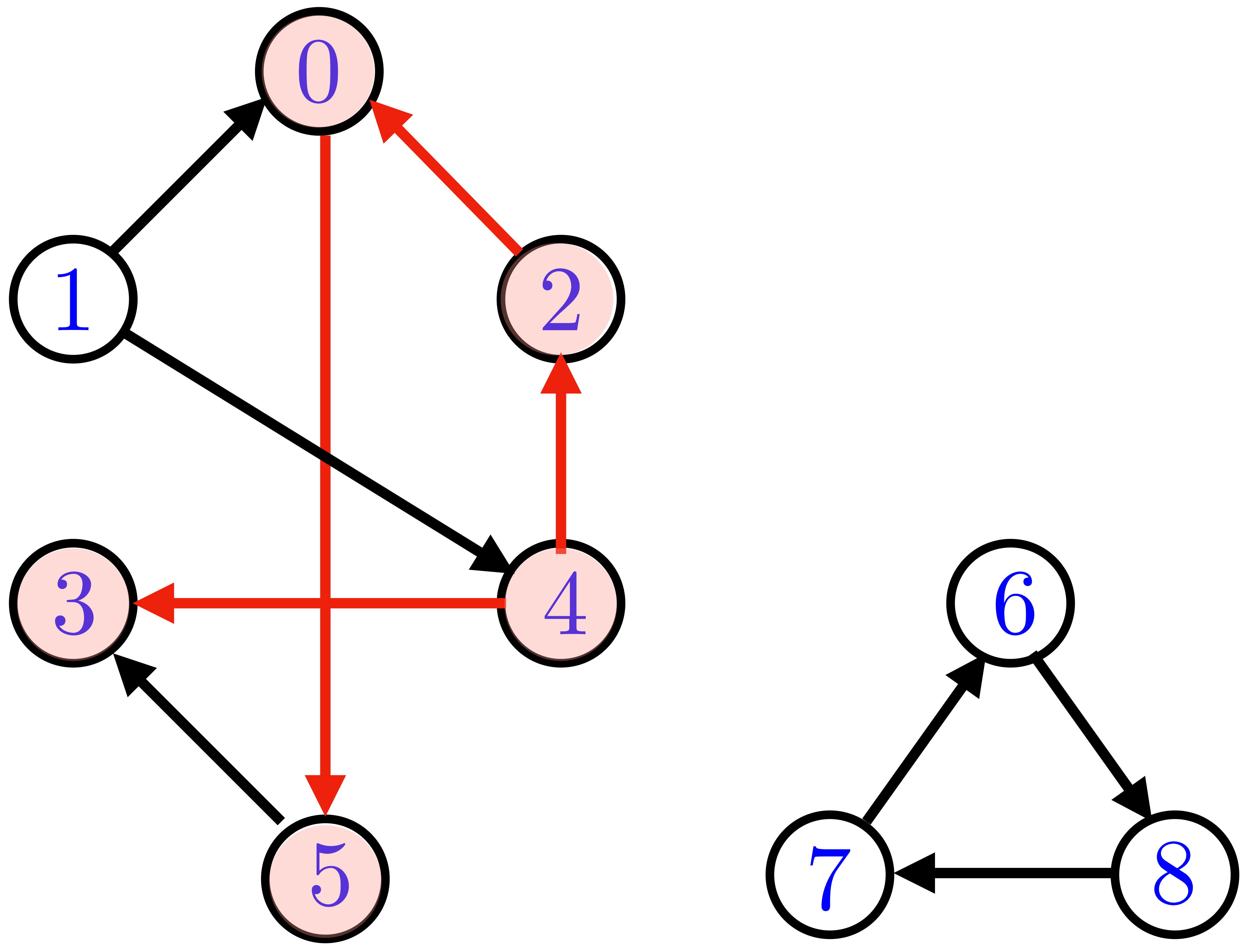
Directed Graph

Which vertices are reachable from vertex 4?



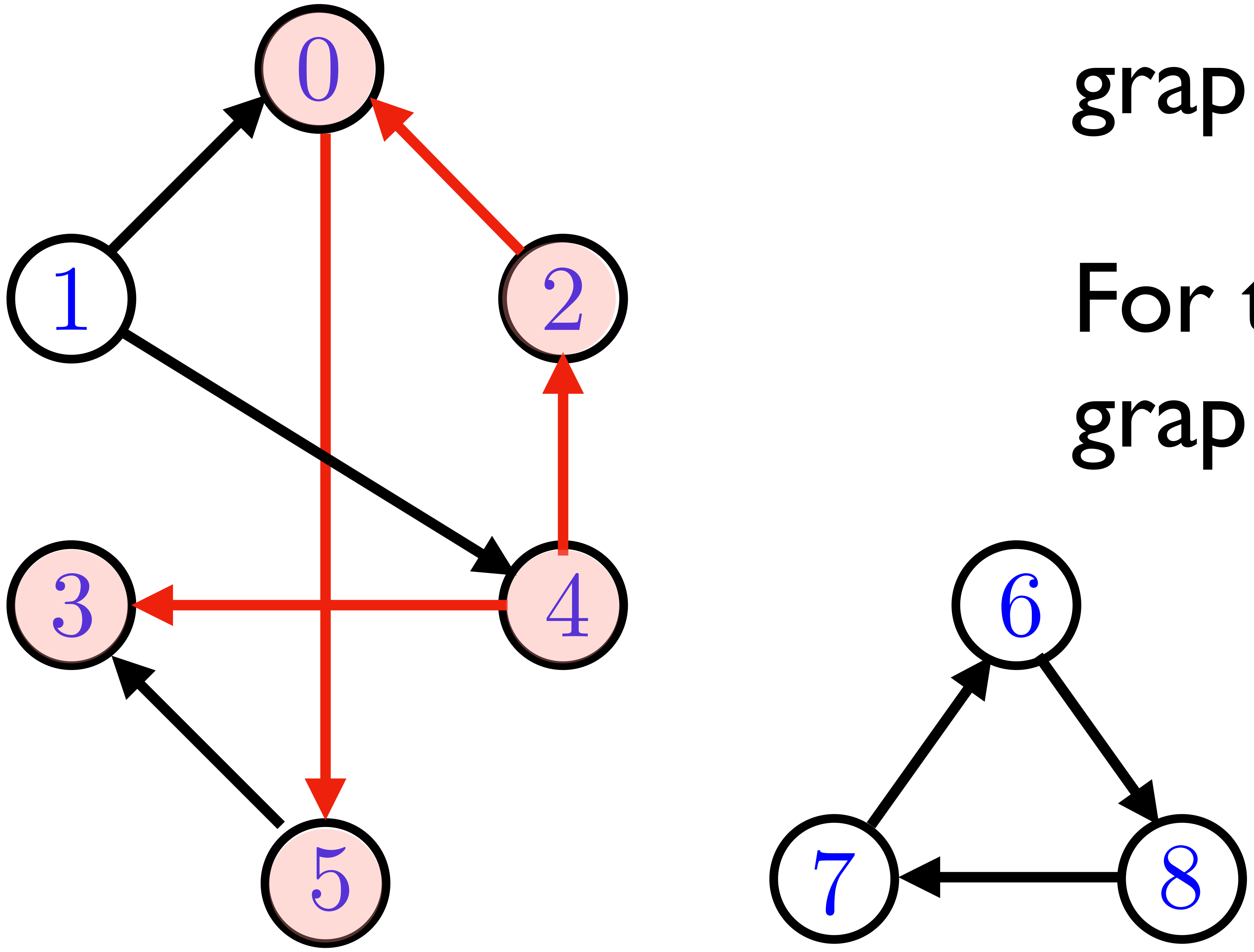
Directed Graph

Which vertices are reachable from vertex 4?



Directed Graph

Which vertices are reachable from vertex 4?



We will talk more about directed graphs next time.

For today we stick to undirected graphs.

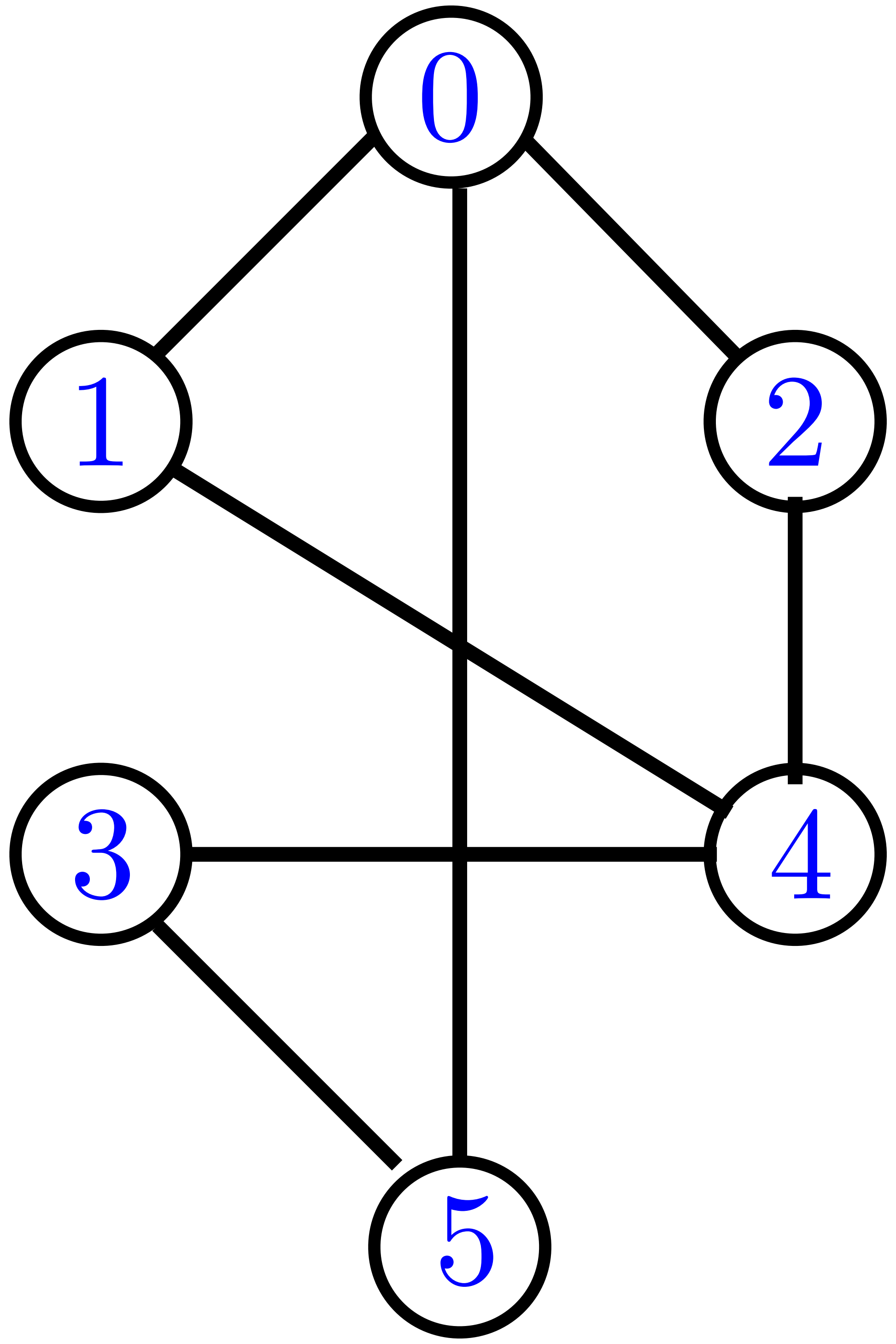
Graph Traversals

What kinds of problems can graph traversals solve in undirected graphs?

Connected Component

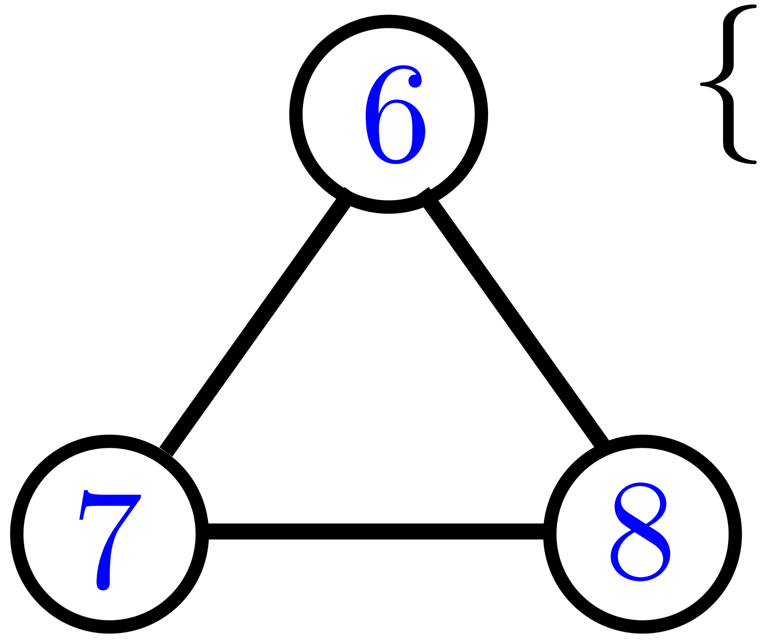
Let's say we have an undirected graph.

A connected component of a graph is a subset S of vertices that



1) is connected, i.e., there is a path between every $u, v \in S$.

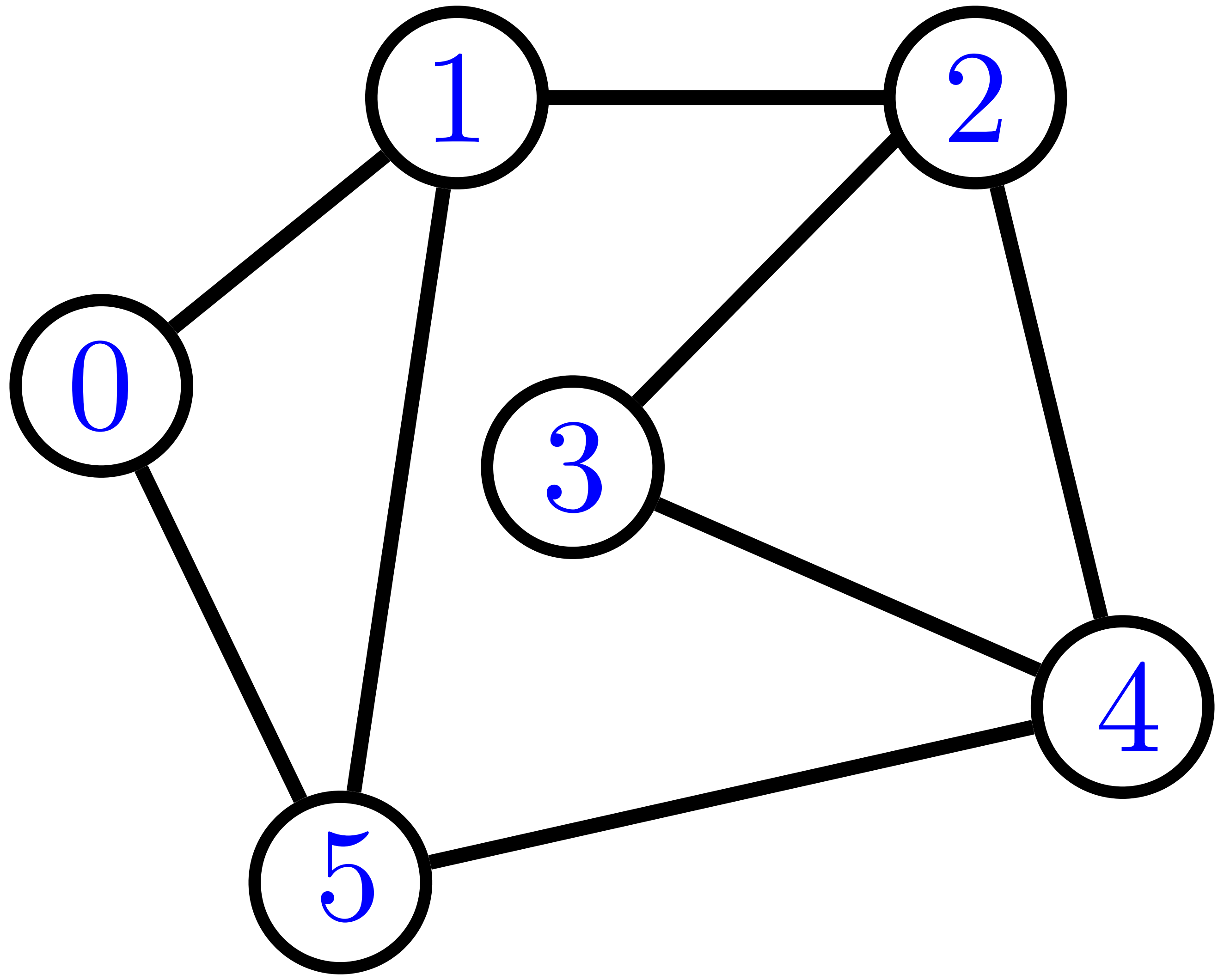
2) Any $u \in S$ is not connected to any $v \notin S$.



The connected components in this graph are $\{0, 1, 2, 3, 4, 5\}$ and $\{6, 7, 8\}$.

Depth-First Search

Depth-First Search

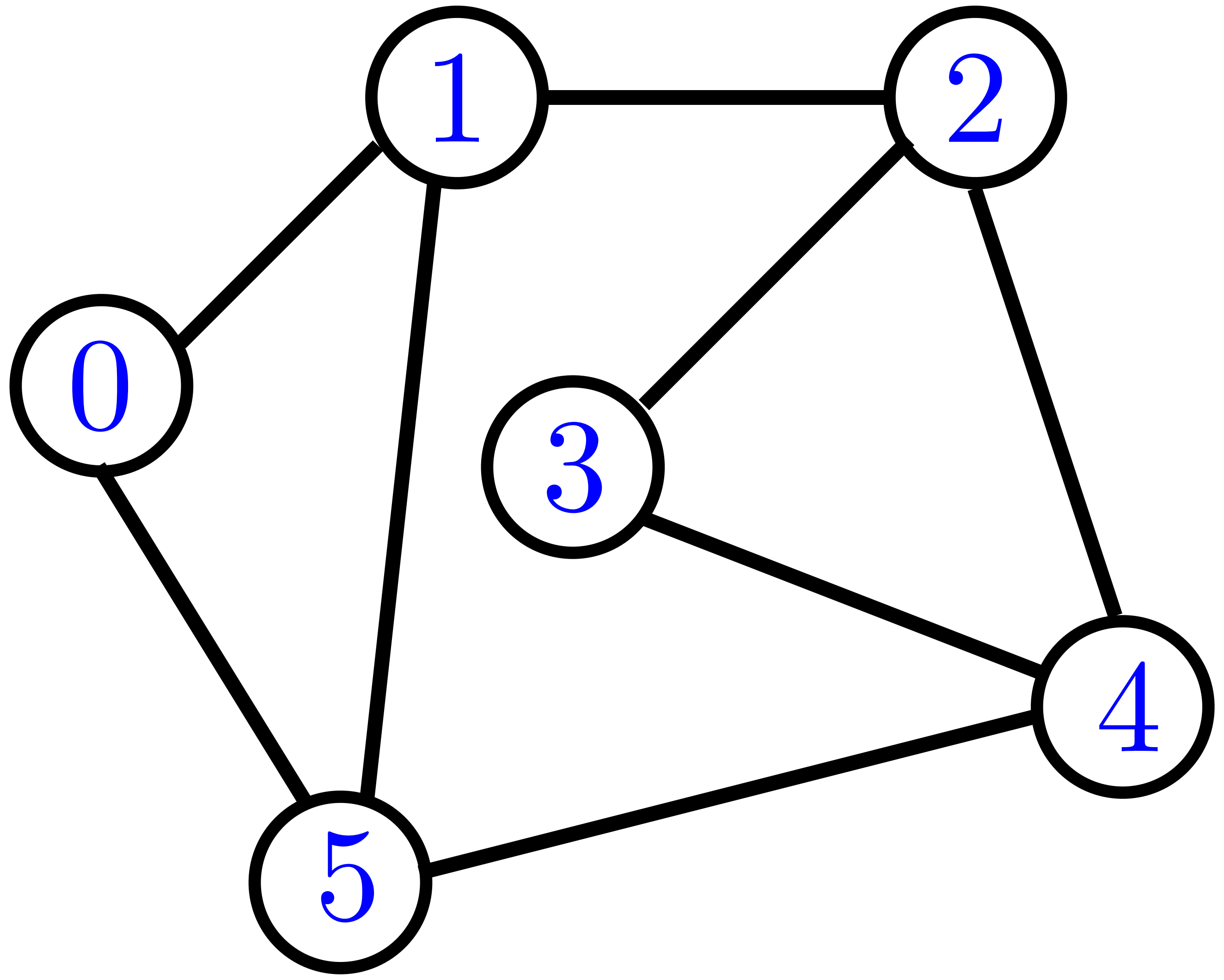


We start at one particular vertex. Let's say we start at vertex 4.

The goal is to visit all vertices connected to the starting vertex.

The main place where we have choice is the order in which we visit neighbours.

Depth-First Search



Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

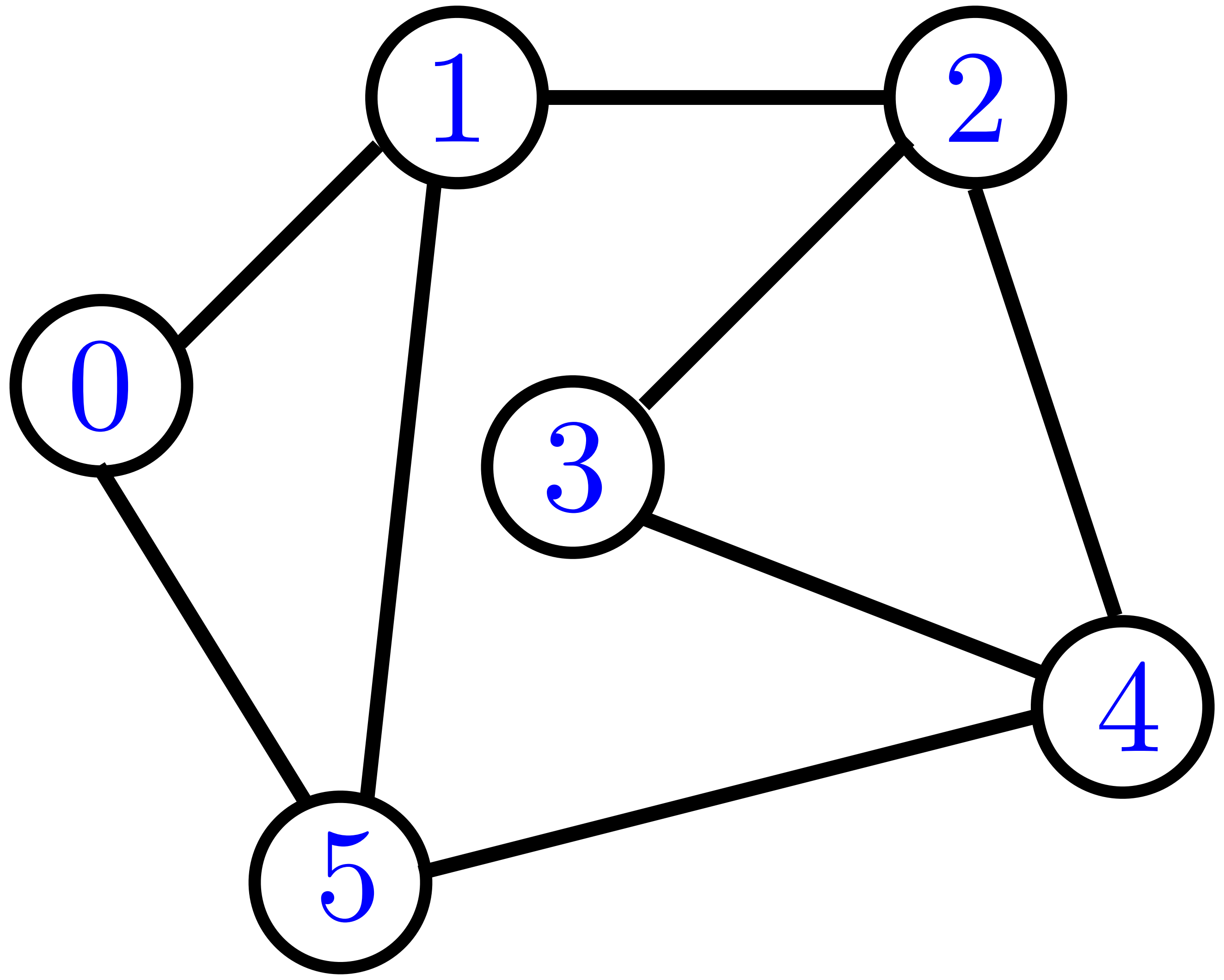
```
bool marked[N] {};

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs(u);
        }
    }
}
```

Here $\texttt{arr}[v]$ is the list of vertices adjacent to v.

The order of neighbors in the list affects the order in which we visit vertices.

Depth-First Search



Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

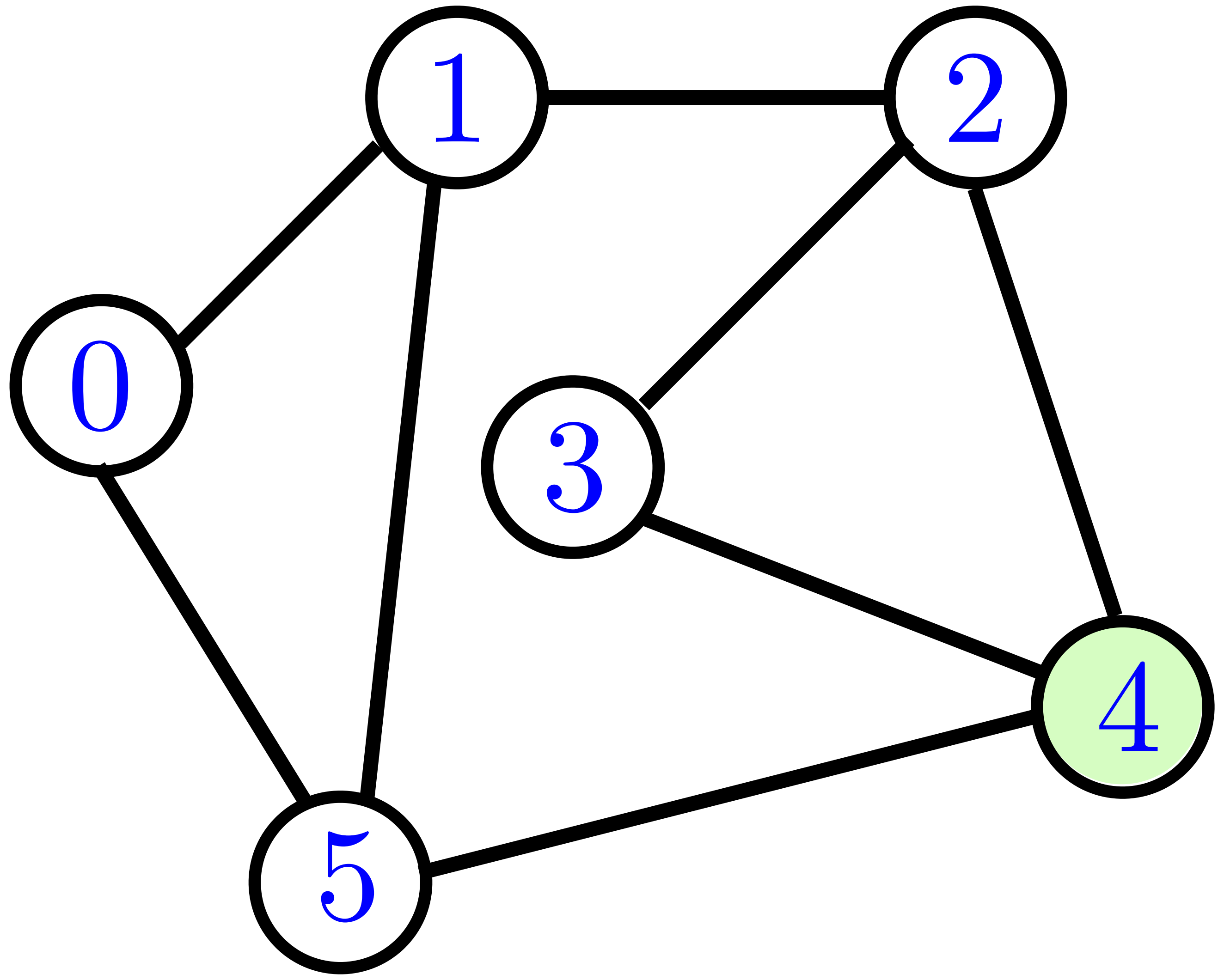
```
bool marked[N] {};
std::vector<int> edge_to(N,-1);

void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

We make one addition: we also have an array `edge_to` where `edge_to`$[u]$ is the vertex from which we visit u.

https://godbolt.org/z/aEcfzd5e1

Depth-First Search



Adjacency List

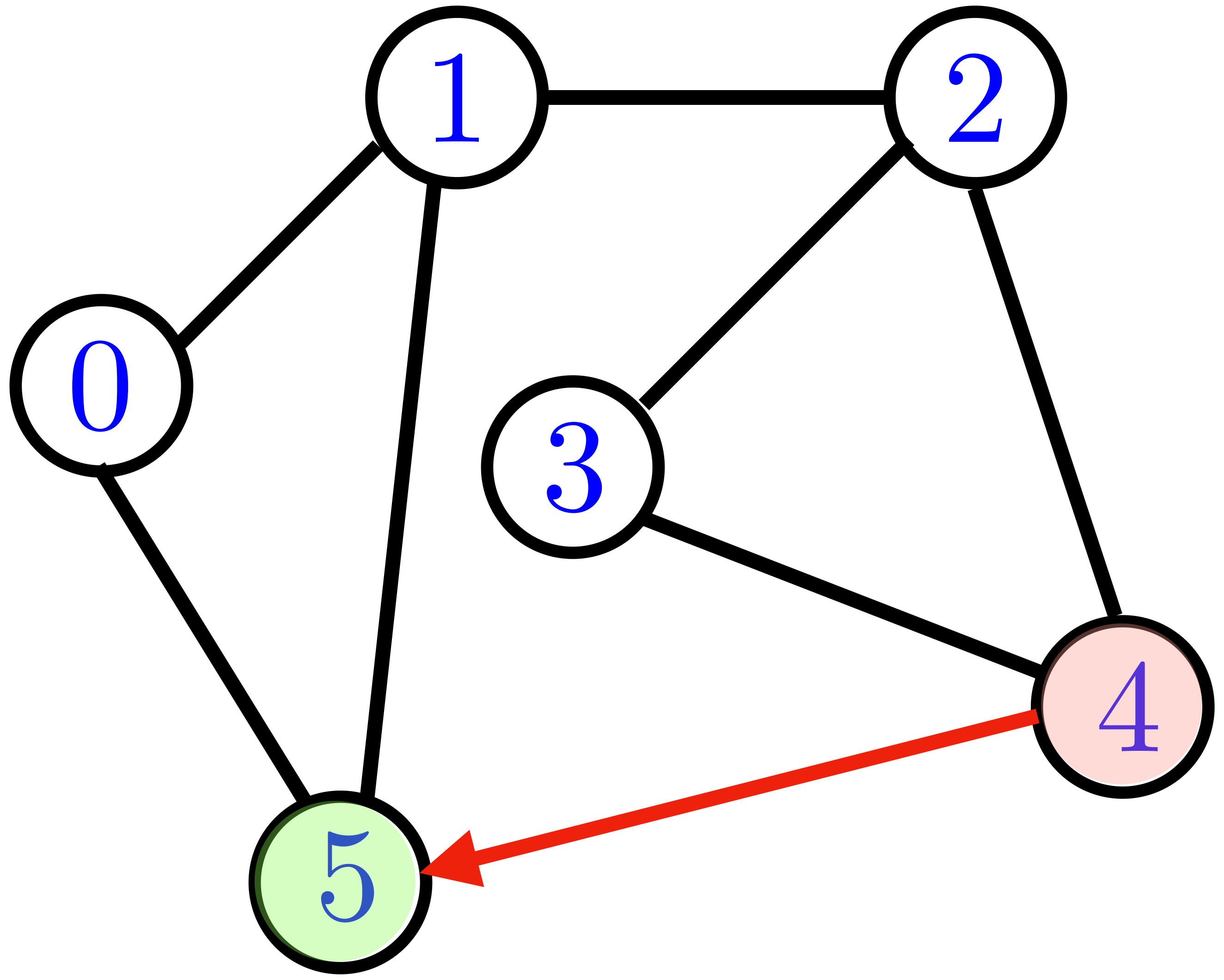
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

We start at vertex 4.

Where do we go next?

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



Vertex 5.

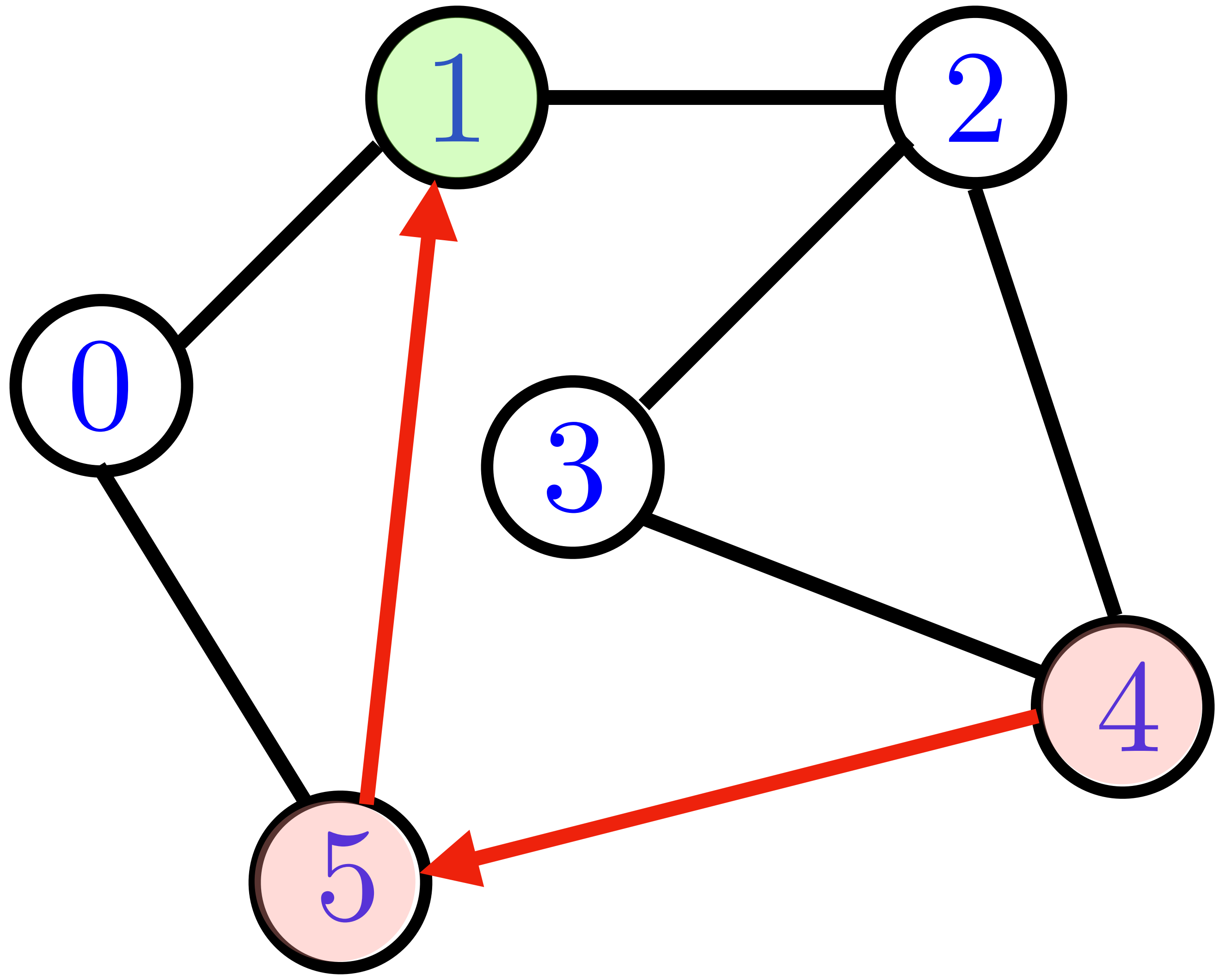
Now where?

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

Depth-First Search



Adjacency List

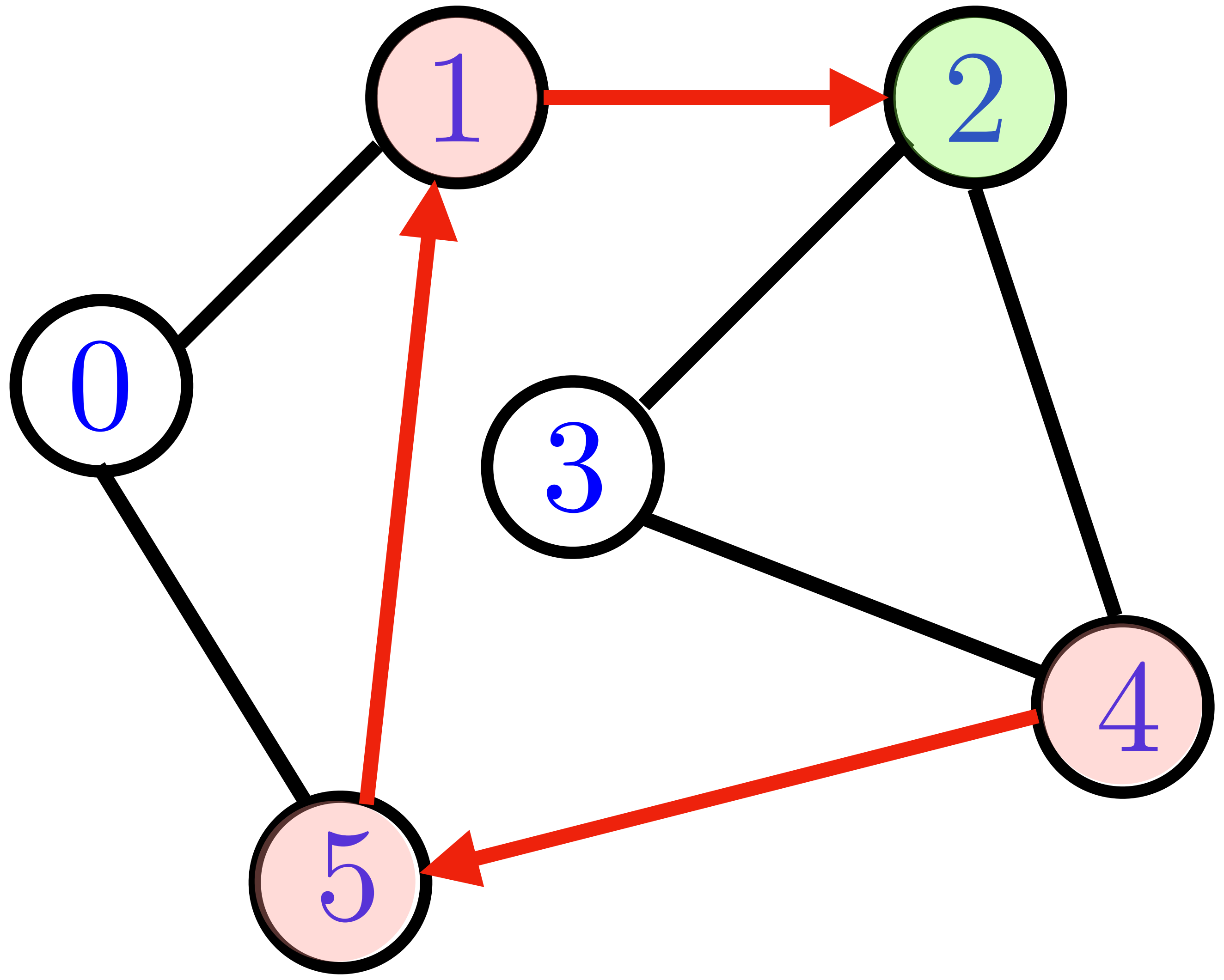
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

Vertex 1.

Next up?

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
           edge_to[u] = v;
           dfs(u);
        }
    }
}
```

Depth-First Search



Vertex 2.

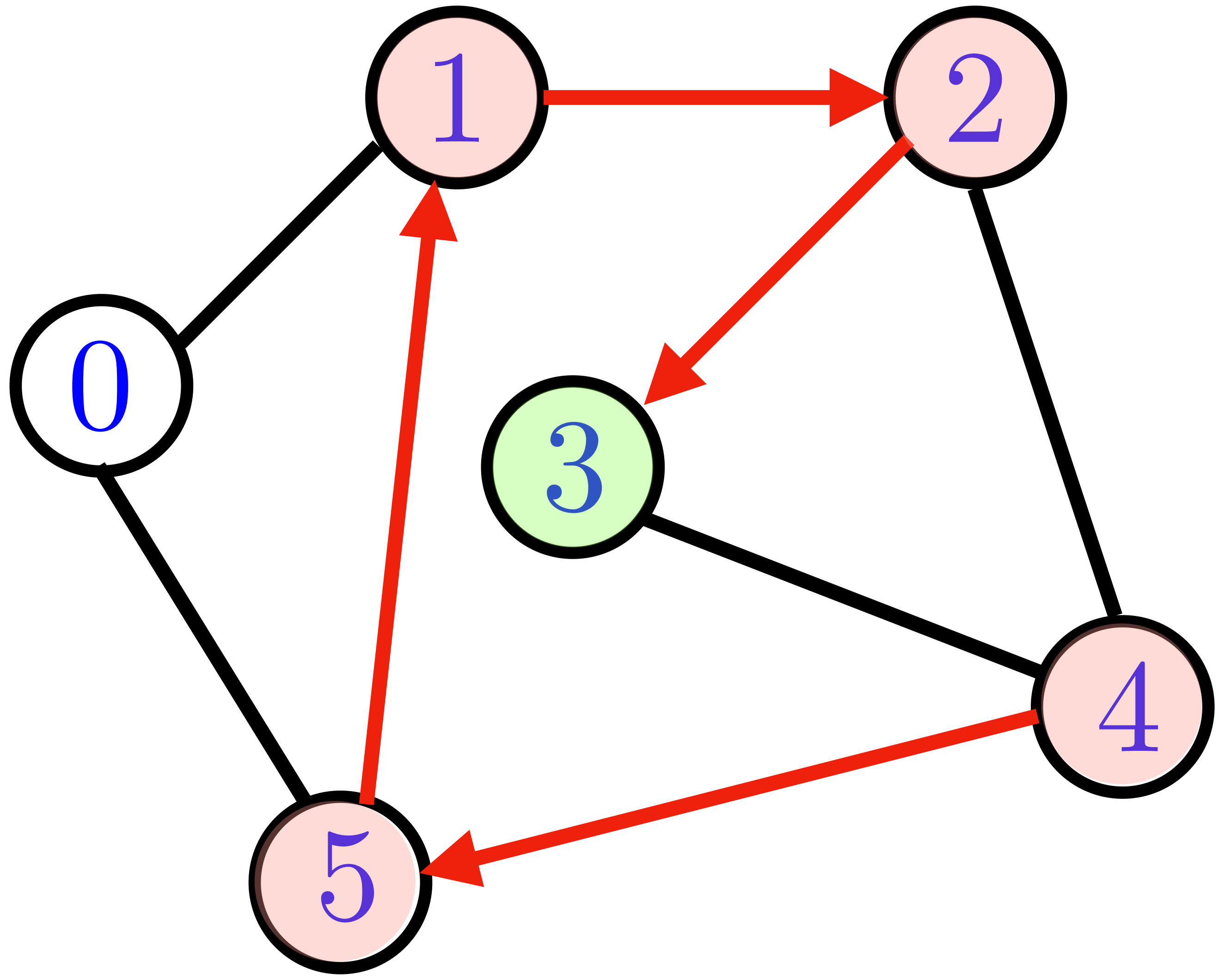
Next up?

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
          edge_to[u] = v;
          dfs(u);
        }
    }
}
```

Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

Depth-First Search



Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

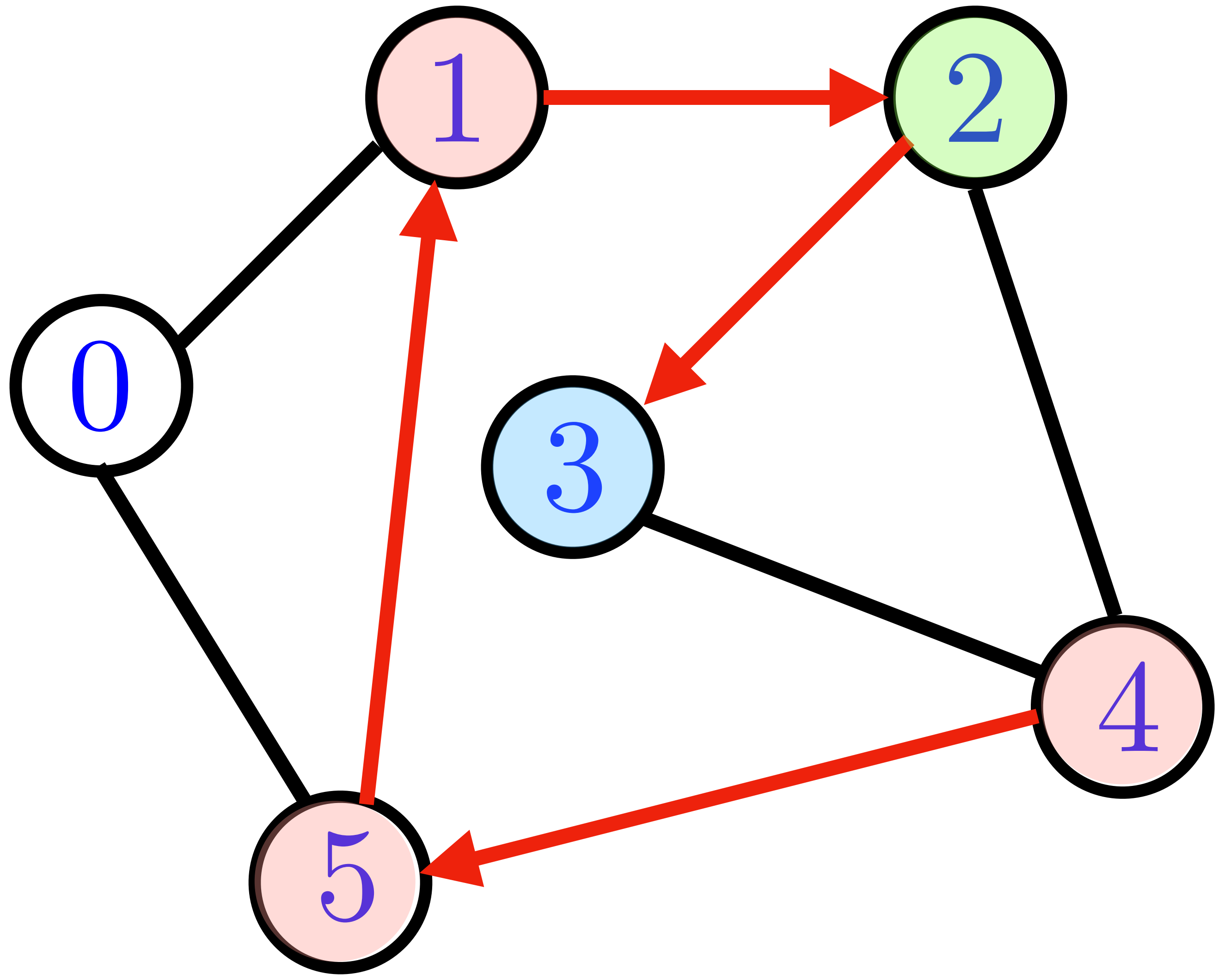
Vertex 3.

The call to `dfs` on vertex 3 terminates.

You can see from the red arrow that we return to `dfs` on vertex 2.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

We are back in $\mathtt{dfs}(2)$.

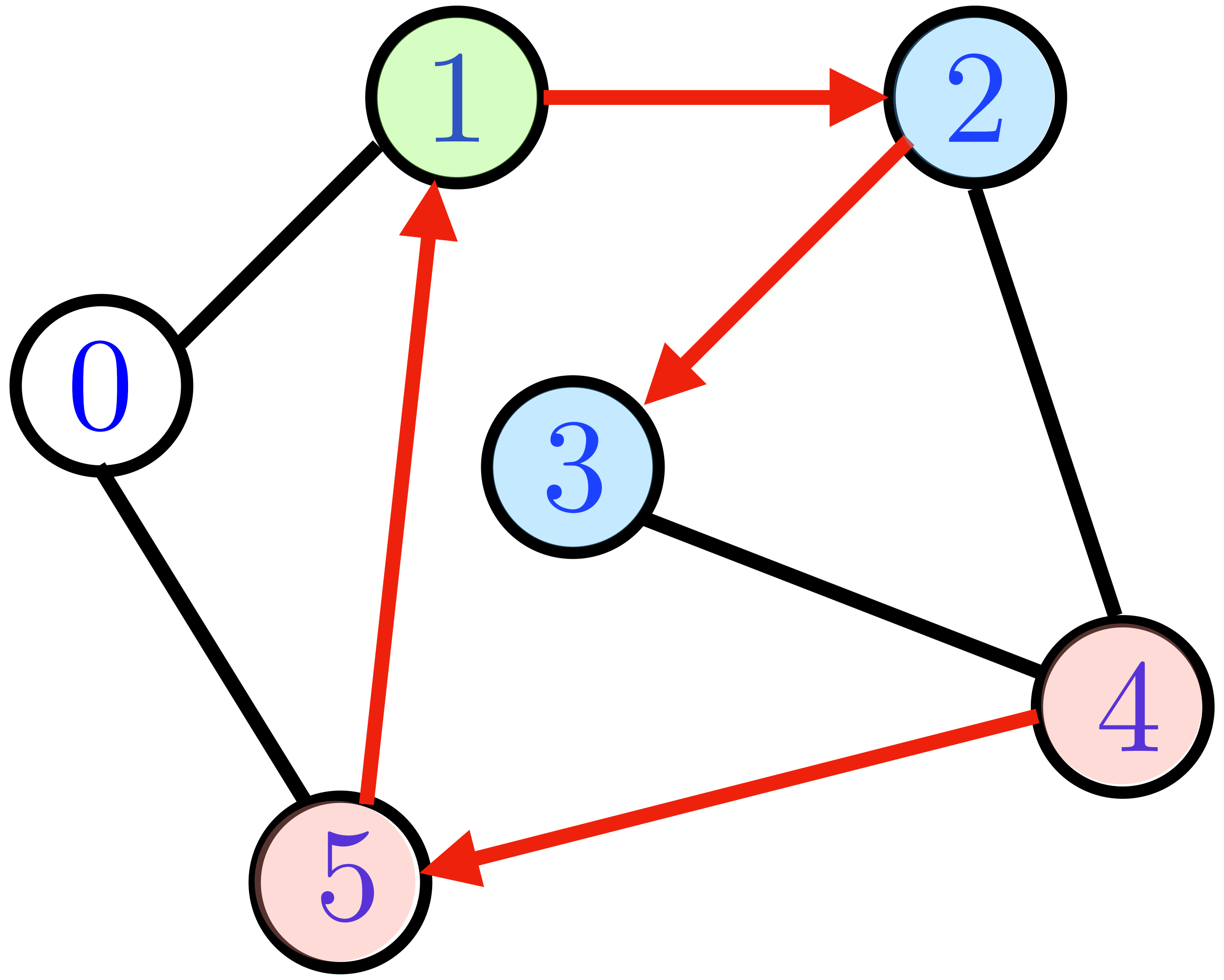
We continue in the for loop with vertex 1.

It is already marked, so the call terminates.

Where do we return to now?

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

We are back in $\texttt{dfs}(1)$.

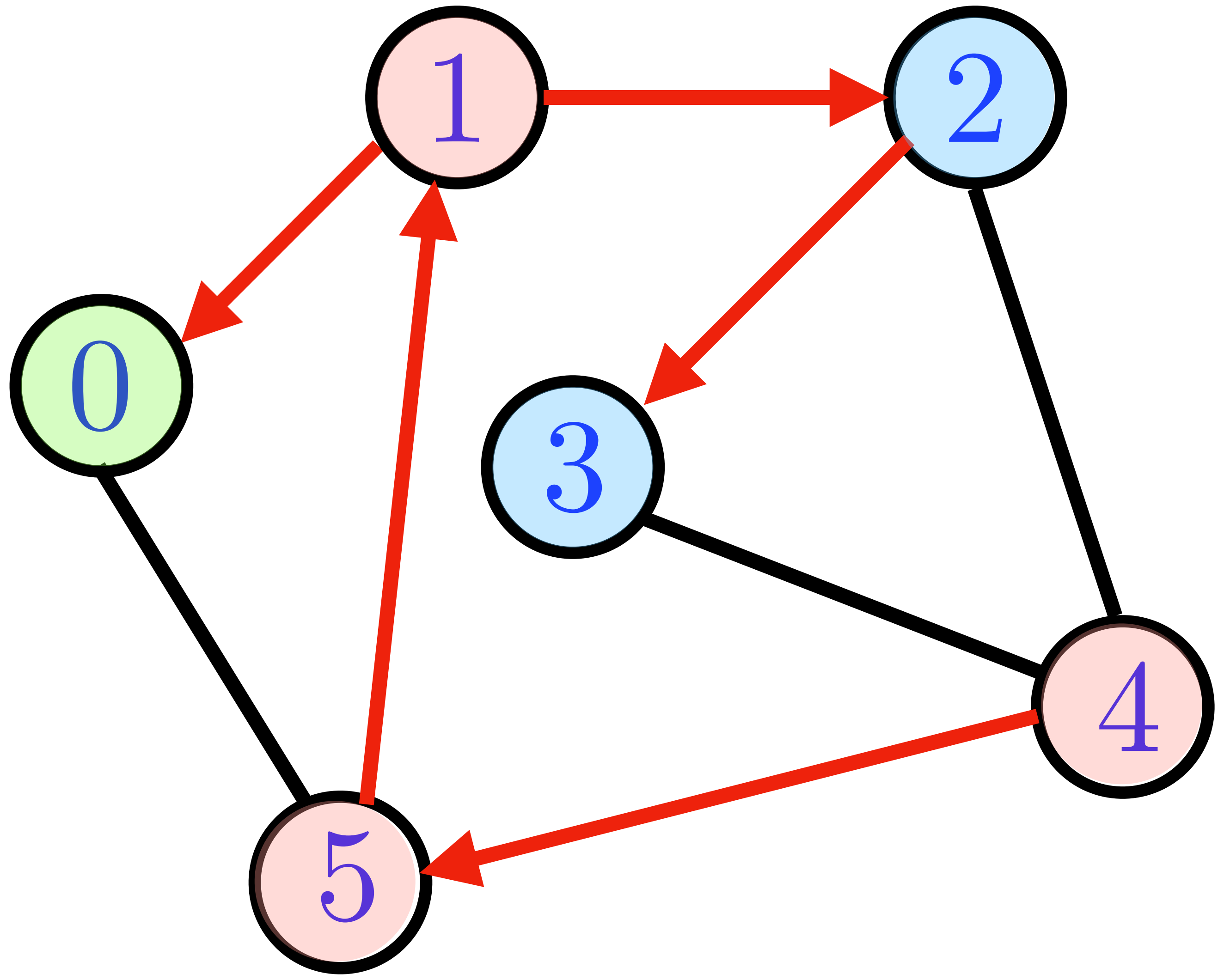
We continue in the for loop with vertex 0.

It is unmarked!

So we visit it.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



We enter $\texttt{dfs}(0)$.

All neighbors are already marked.

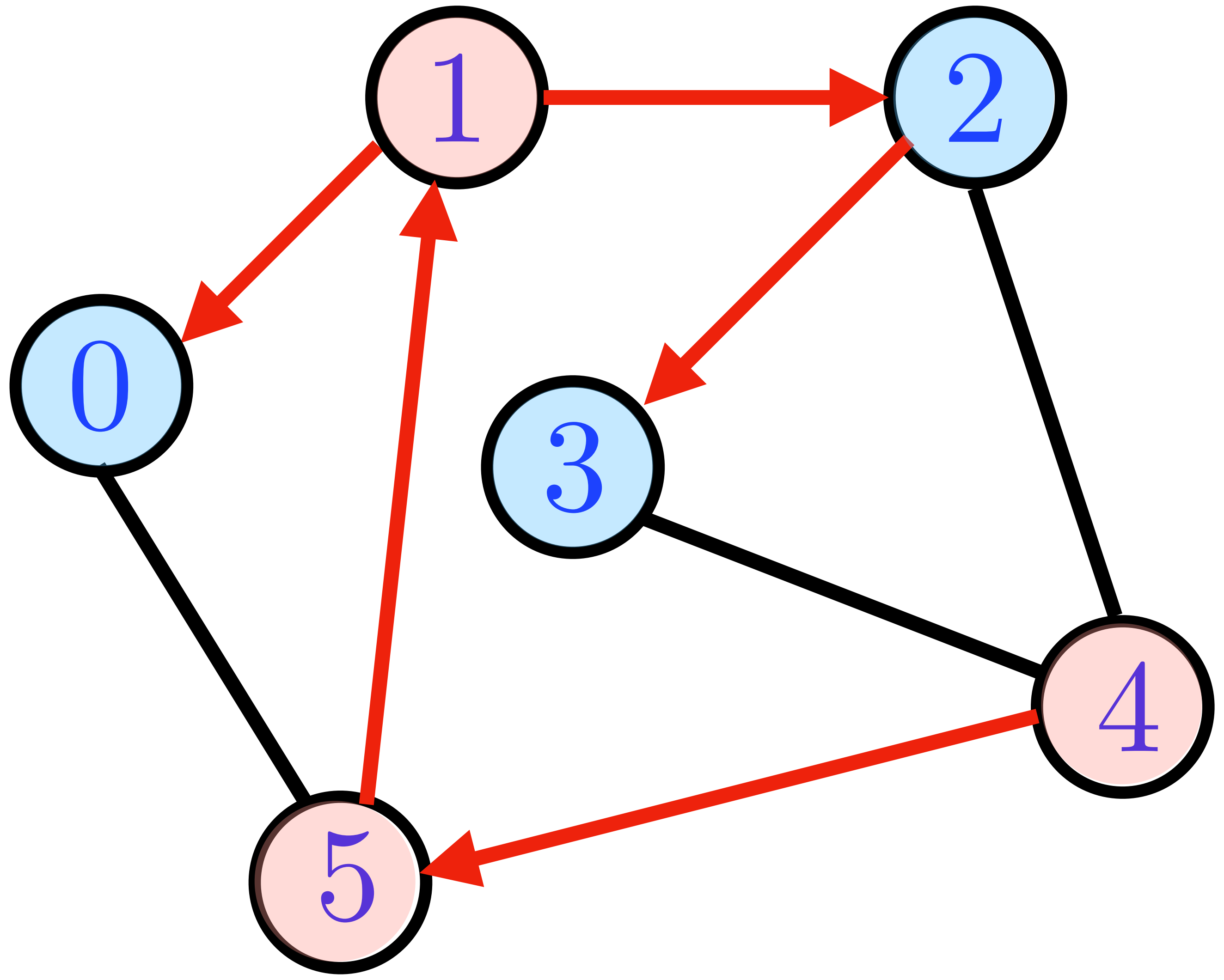
The call terminates.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Adjacency List

0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

Depth-First Search



Adjacency List

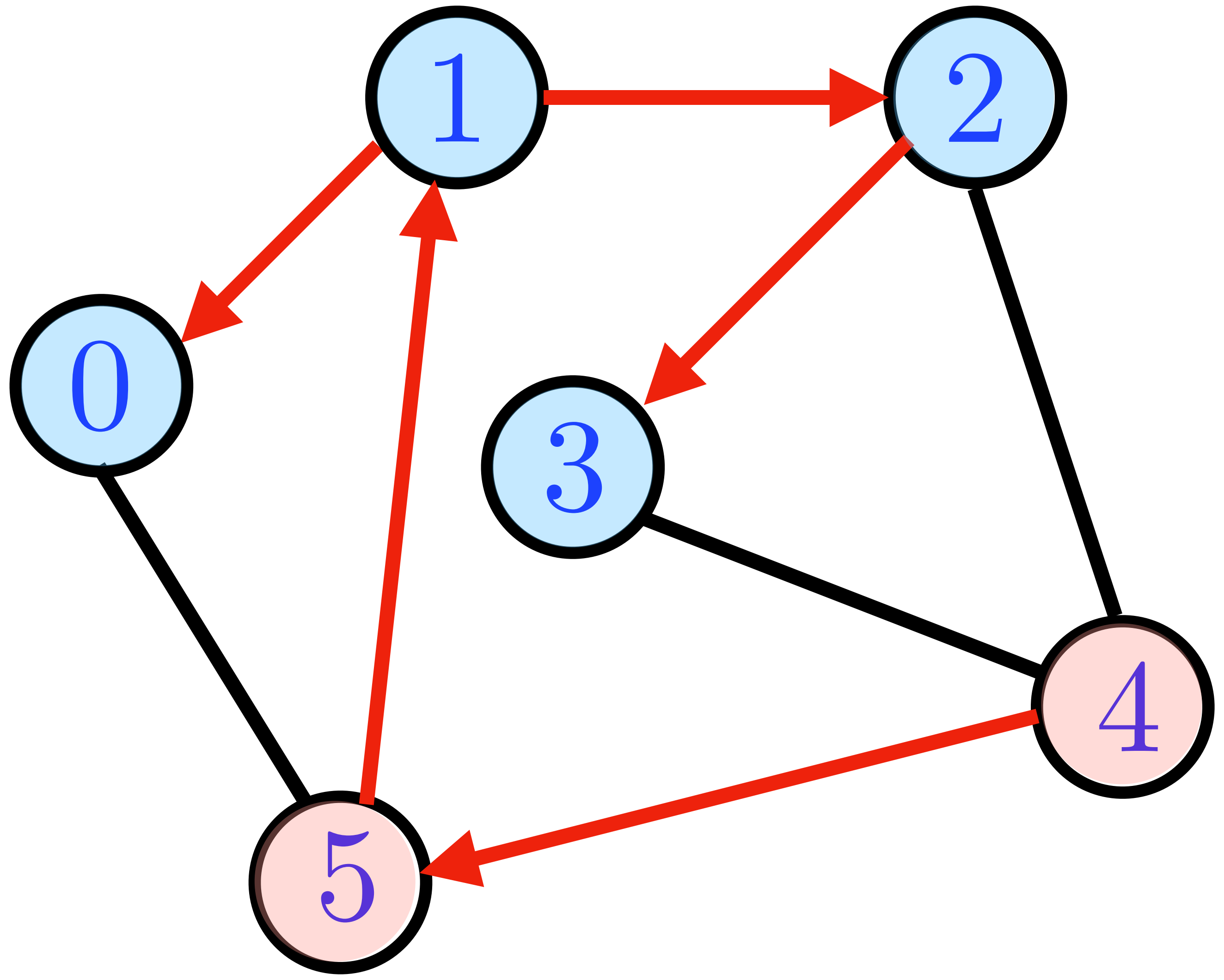
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

All vertices are now marked.

The recursive calls unwind without further action.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
          edge_to[u] = v;
          dfs(u);
        }
    }
}
```

Depth-First Search



Adjacency List

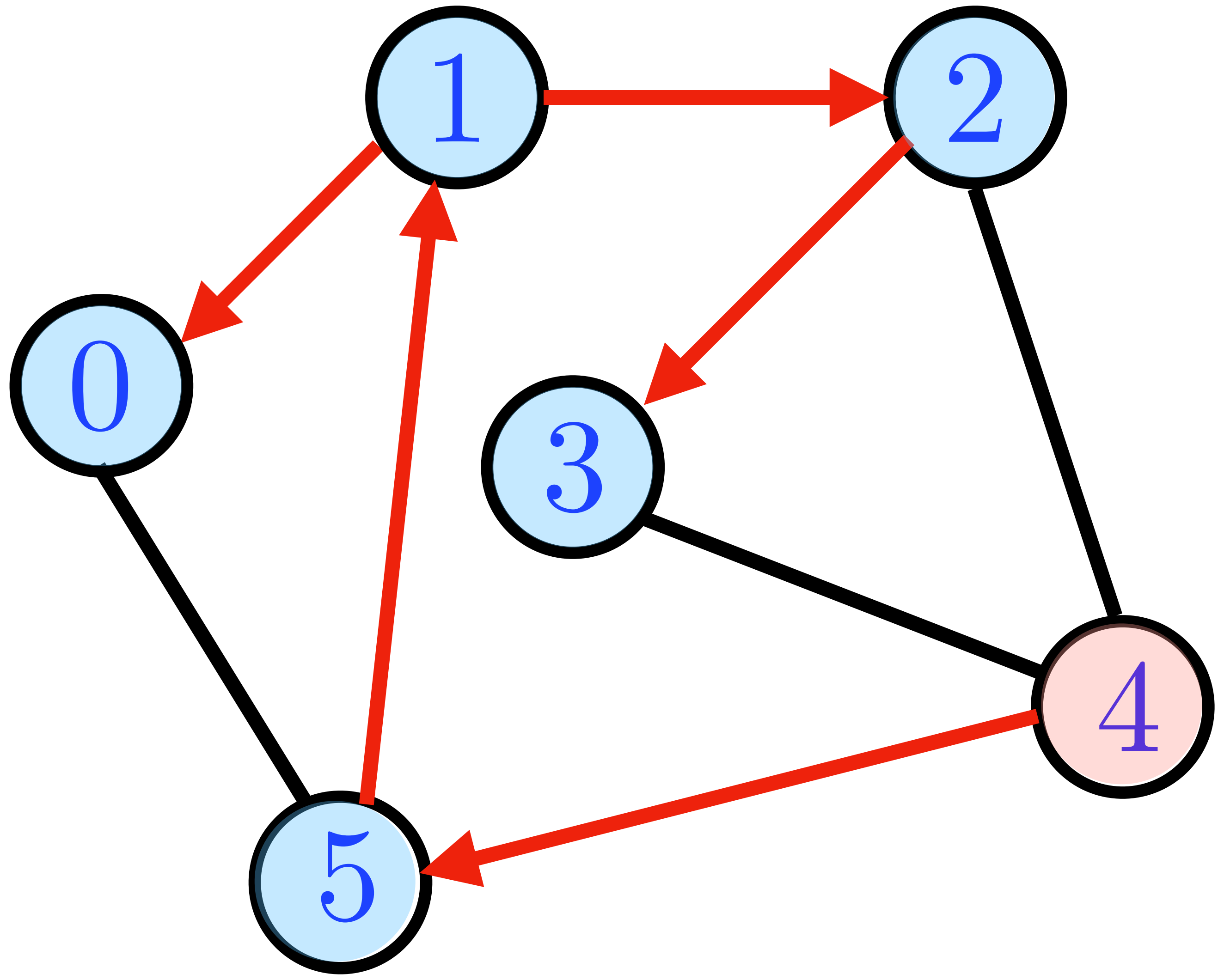
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

All vertices are now marked.

The recursive calls unwind without further action.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



Adjacency List

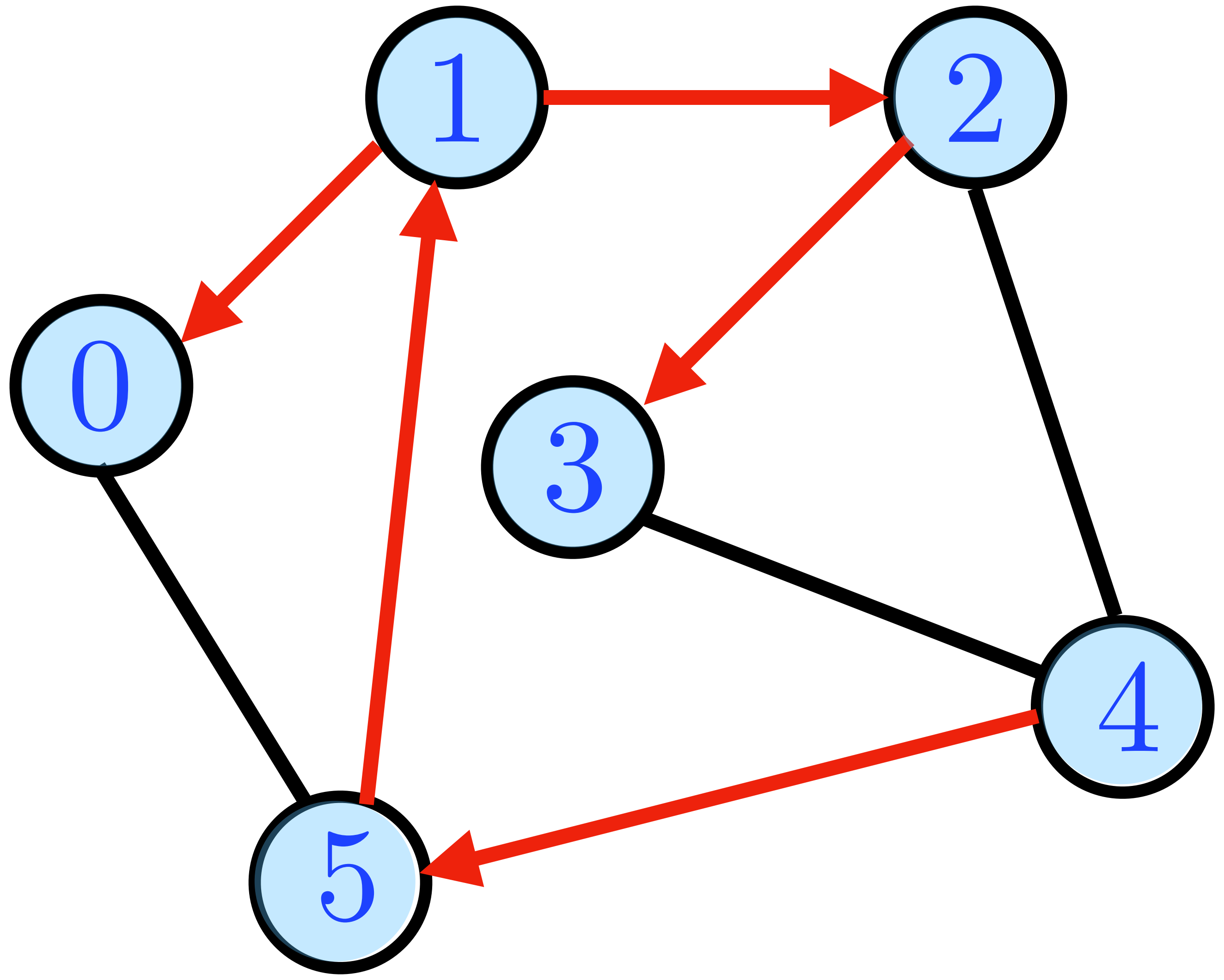
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

All vertices are now marked.

The recursive calls unwind without further action.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search



Adjacency List

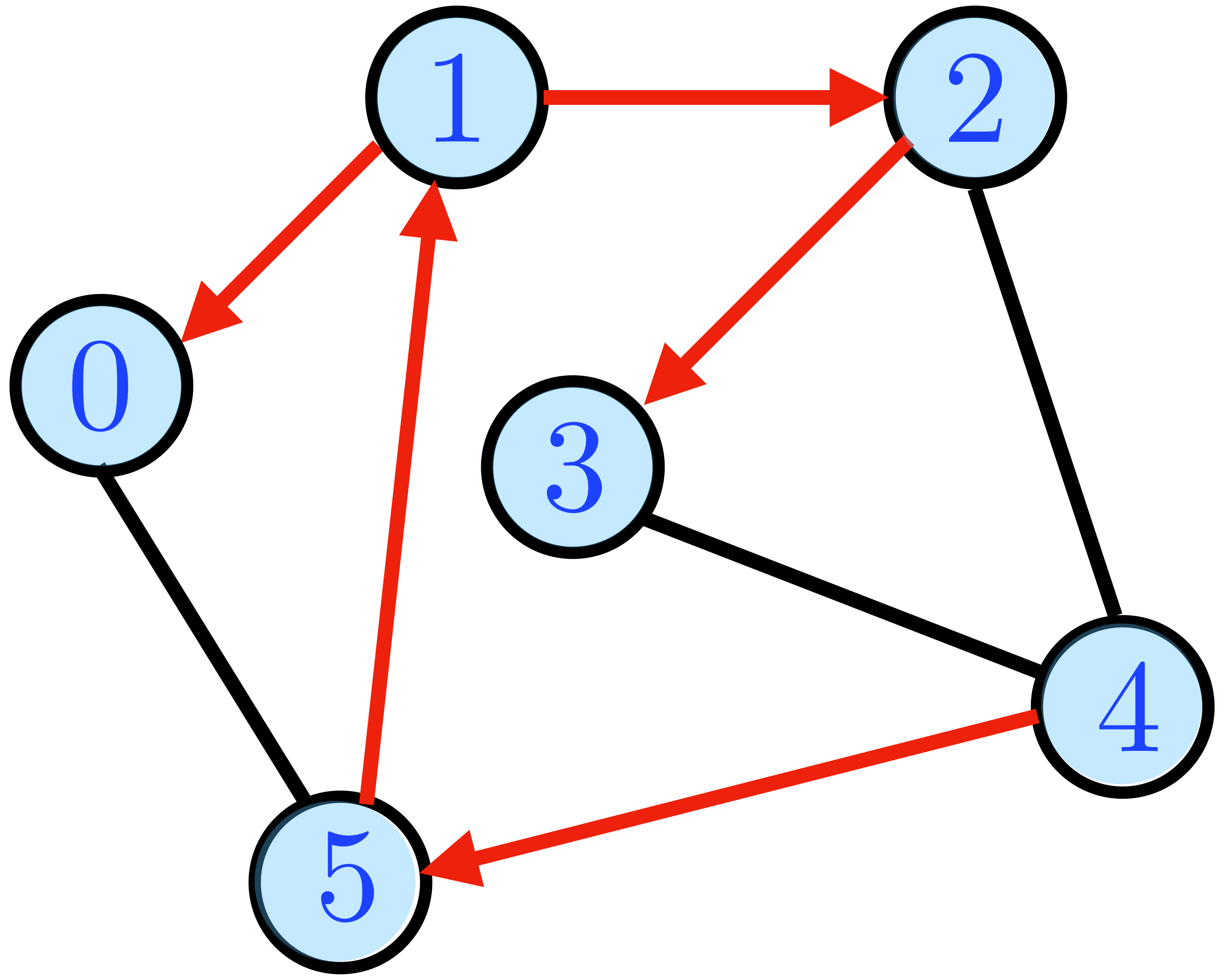
0: 1 5
1: 5 2 0
2: 4 3 1
3: 4 2
4: 5 3 2
5: 4 1 0

All vertices are now marked.

The recursive calls unwind without further action.

```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Depth-First Search




```
void dfs(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            edge_to[u] = v;
            dfs(u);
        }
    }
}
```

Summary:

- We mark exactly the vertices reachable from the starting vertex.
- We can use `edge_to` to find paths from the starting vertex to the other marked vertices.
- What is the running time?

Iterative DFS

We can also write DFS without recursion.

We add the neighbors of the vertex we are visiting to a stack.

This can simulate the order of calls of the recursive version.

```
void iterative_dfs(unsigned start)
{
    visit_stack.push(start);
    while(!visit_stack.empty())
    {
        unsigned x = visit_stack.top();
        visit_stack.pop();
        if(marked[x])
        {
            continue;
        }
        marked[x] = true;
        for(auto u : arr[x])
        {
            if(!marked[u])
            {
                visit_stack.push(u);
            }
        }
    }
}
```

Breadth-First Search

Breadth-First Search

How does breadth-first search differ from depth-first search?

Breadth-First Search

```
void bfs(unsigned start)
{
    visit_queue.push(start);
    marked[start] = true;
    while(!visit_queue.empty())
    {
        unsigned x = visit_queue.front();
        visit_queue.pop();
        for(auto u : arr[x])
        {
            if(!marked[u])
            {
                visit_queue.push(u);
                marked[u] = true;
                // we came to u from x
                edge_to[u] = x;
            }
        }
    }
}
```

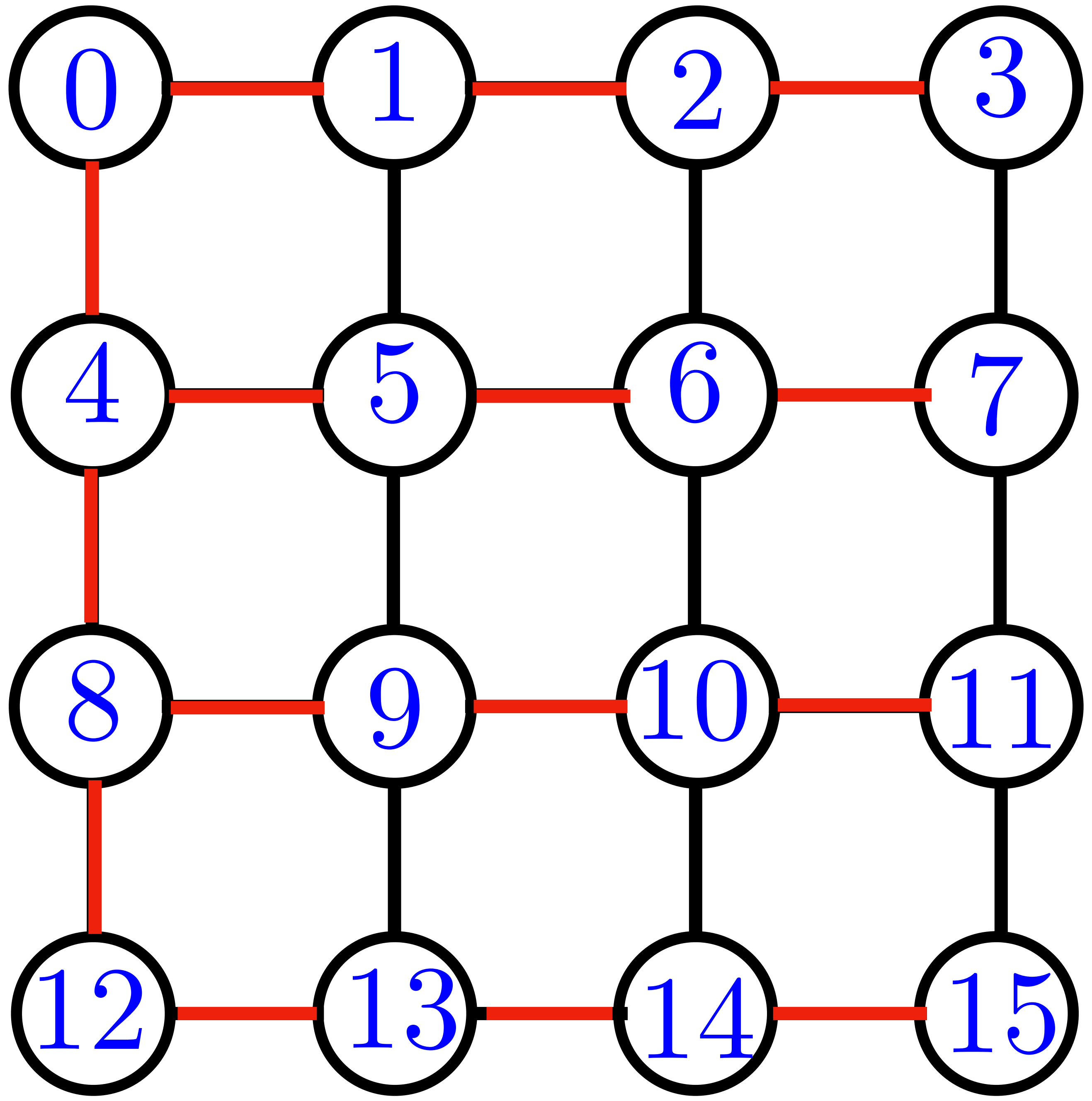
Basically we replace the stack data structure of iterative DFS with a queue.

We explore near neighbors before far ones.

Breadth-First Search

What is breadth-first search good at?

Shortest Path Tree



This is a picture of the `edge_to` relation from running breadth-first search starting at vertex 0.

The path between 0 and any other vertex v in this tree is a shortest path between them in the original graph.

Whatever-first search

Whatever-first search

Depth-first and breadth-first search follow the same outline, but they choose which vertex to visit next in a different way.

In DFS the visit order is determined by a stack and in BFS a queue.

Plugging in different data structures also gives interesting algorithms!

This leads to an idea Jeff Erickson calls "whatever-first search".

Section 5.5 in Algorithms by Jeff Erickson
https://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf

Whatever-first search

A bag stands for any data structure that has operations of push, front, and pop.

```
WHATEVERFIRSTSEARCH(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

Algorithms by Jeff Erickson, page 200.

Next time, we will see Dijkstra's algorithm, which follows this template where the bag is a priority queue.