Collection Concurrency

Robert Horvick
SOFTWARE ENGINEER

@bubbafat www.roberthorvick.com

Overview

Concurrency

- Overview
- Problems

Solutions

- Caller synchronization
- Monitor locking
- Read/write locking

.NET concurrent collections

Concurrency

Two or more operations executing at the same time (concurrently).

Concurrency

Multiple threads executing within a single process accessing a shared resource (e.g., a shared List<T>)

Multiple processes running on the same computer accessing a shared resource (e.g., a shared file)

Multiple processes running on different computers accessing a shared resource (e.g., a shared database table)

```
class Job {
    readonly int Priority;  // set in constructor
    public void Process() {
        ...
    }
}
```

Job Class

A basic job class that contains a priority and a process method.

```
var jobs = new PriorityQueue<Job>();


for(int i = 0; i < 100; i++) {
    jobs.Enqueue(new Job(i));
}


while(jobs.Count > 0) {
    var job = jobs.Dequeue();
    job.Process();
}
```

◂ The priority queue of jobs to execute

◂ Add 100 jobs to the queue (the constructor parameter is the job priority)

◂ Dequeue and process each of the job in priority order.

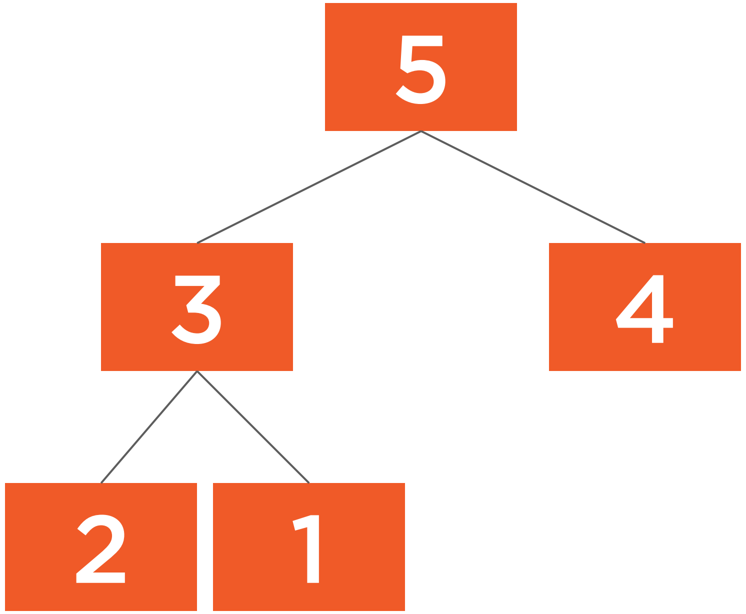
```
Thread[] adders = new Thread[4];

ThreadStart addJobs = delegate() {
    for(int i = 0; i < 25; i++) {
        jobs.Enqueue(new Job(i));
    }
}

for(int i=0; i < adders.Length; i++) {
    adders[i] = new Thread(addJobs);
    adders[i].Start();
}
```

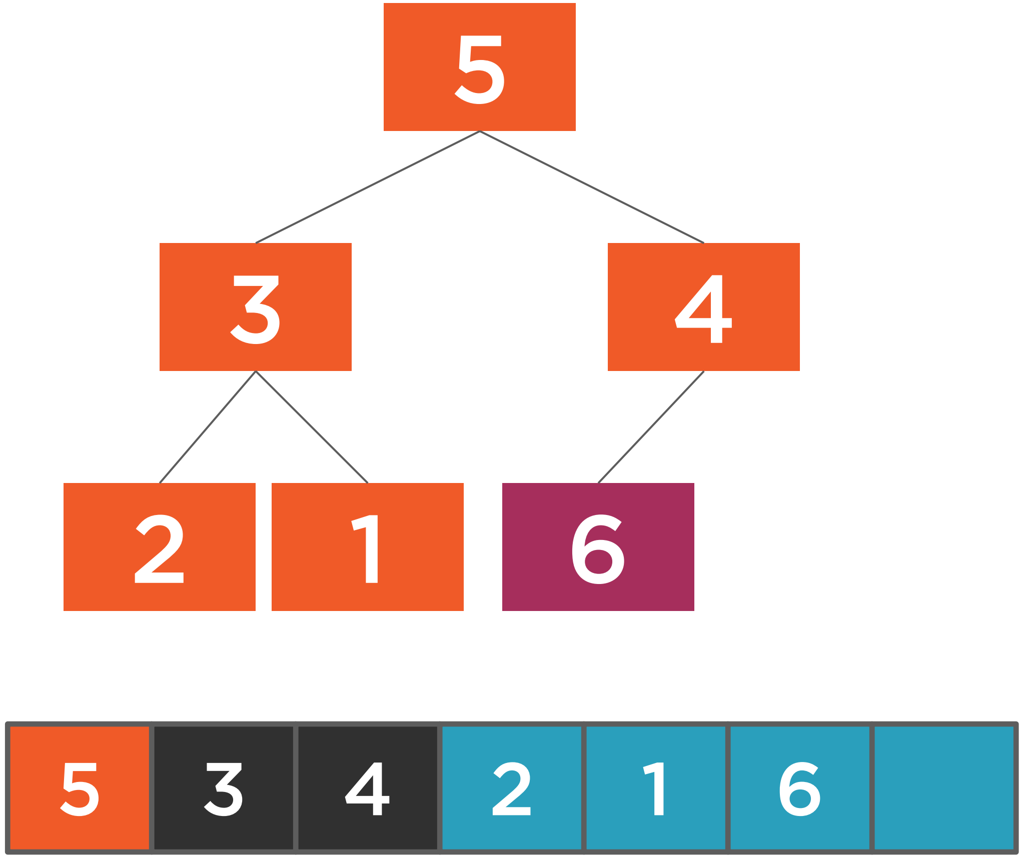
- Add jobs using 4 threads
- Each thread will add 25 jobs to the queue
- Create the 4 threads and start them to add the jobs to the queue concurrently

Add new value to end of heap



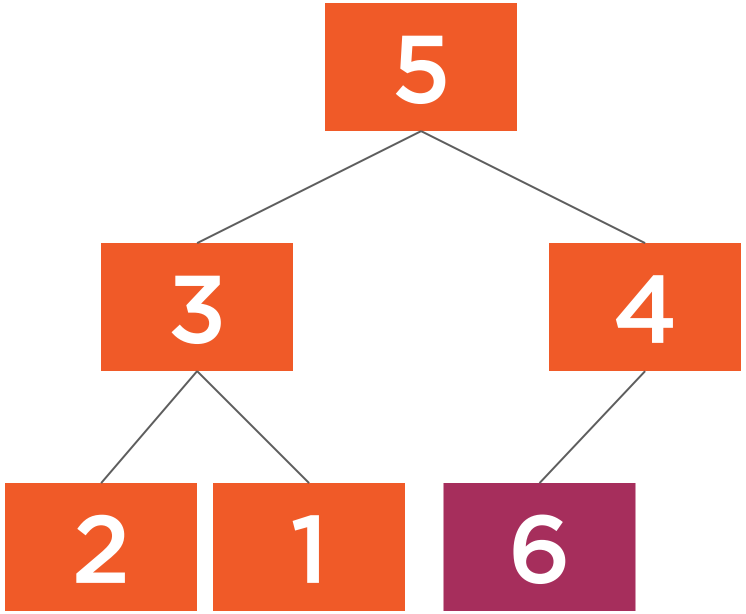
5	3	4	2	1		

Add new value to end of heap



Add new value to end of heap

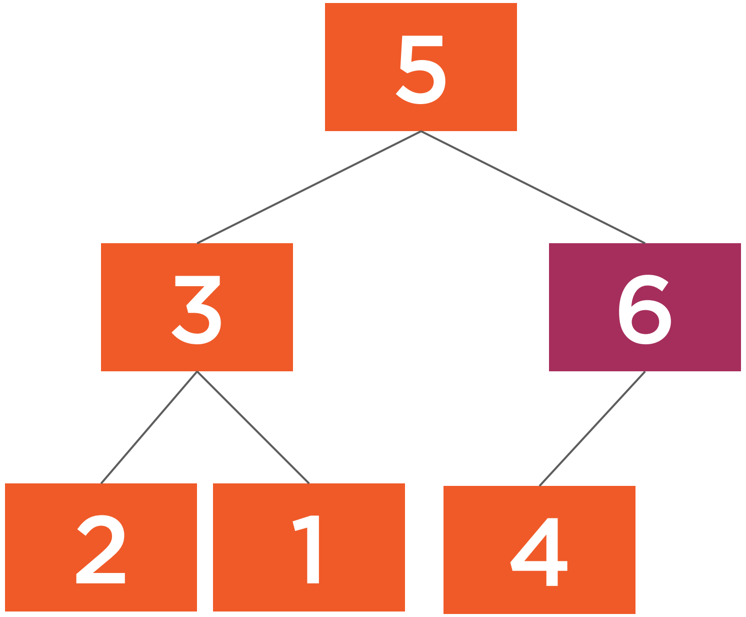
While the heap property is not satisfied, swap with its parent



5	3	4	2	1	6	

Add new value to end of heap

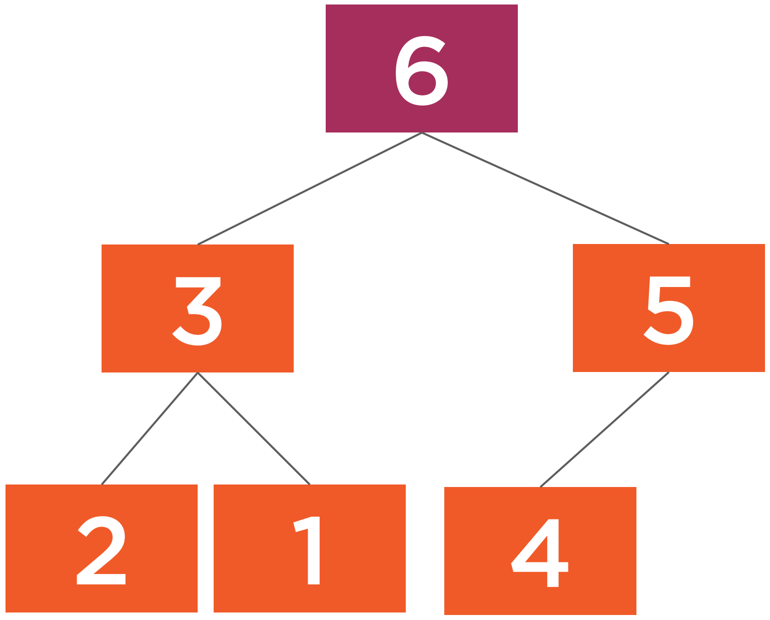
While the heap property is not satisfied, swap with its parent



5	3	6	2	1	4	

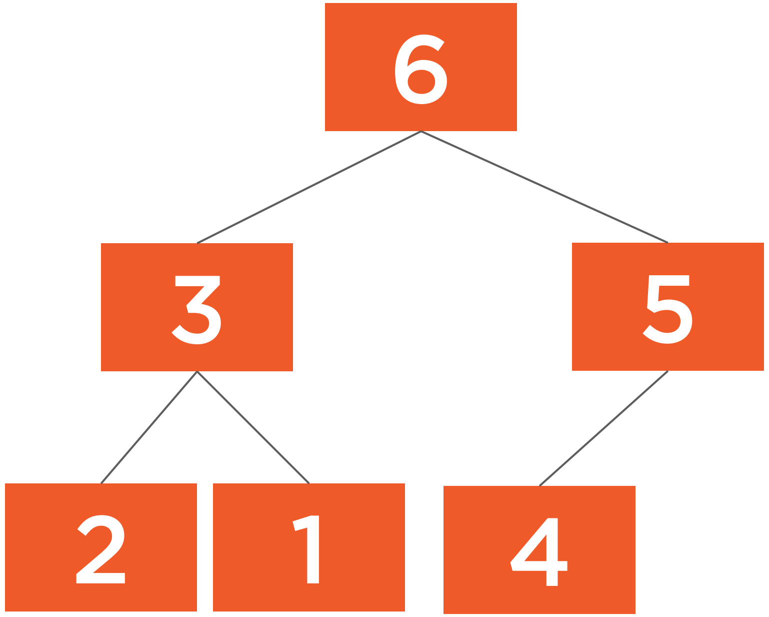
Add new value to end of heap

While the heap property is not satisfied, swap with its parent



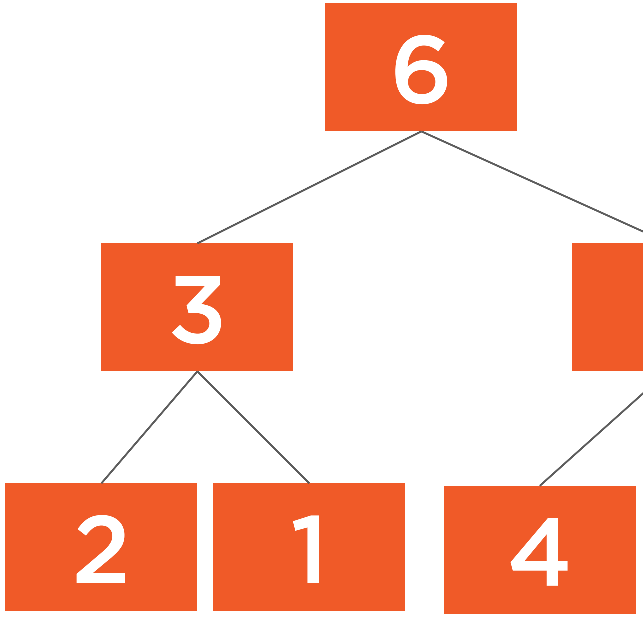
6	3	5	2	1	4	

Thread 1	Thread 2



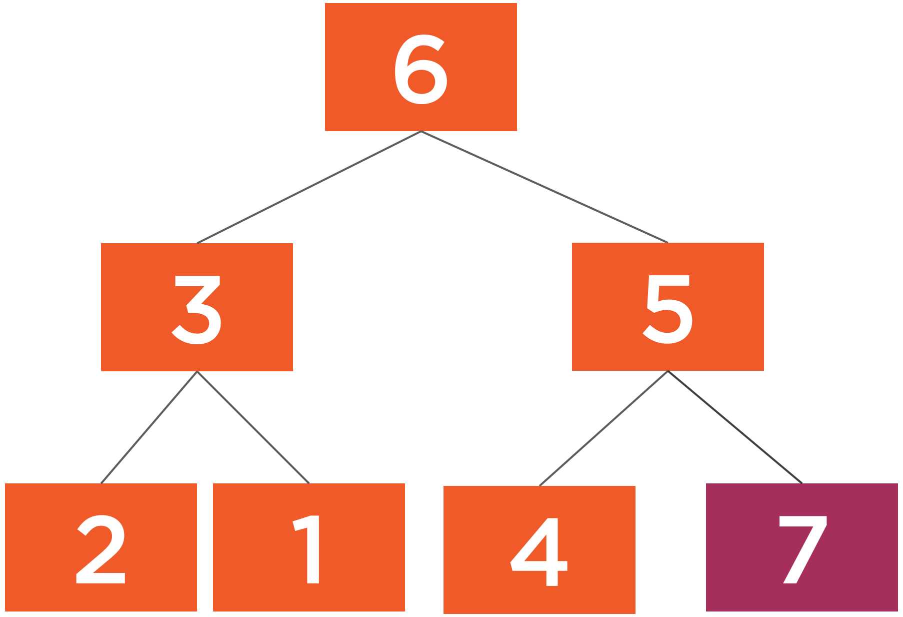
6	3	5	2	1	4	

Thread 1	Thread 2
7	



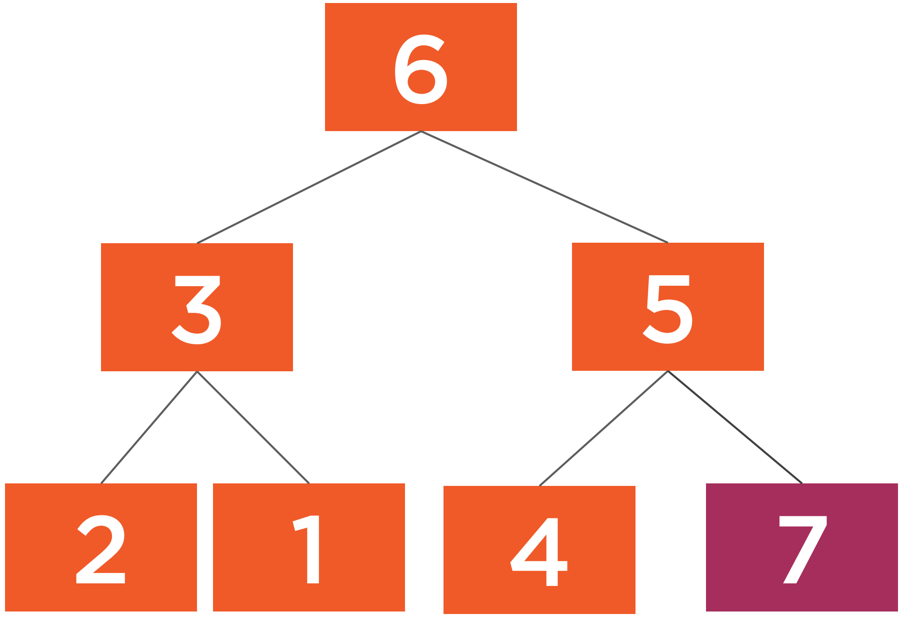
6	3	5	2	1	4	

Thread 1	Thread 2
7	



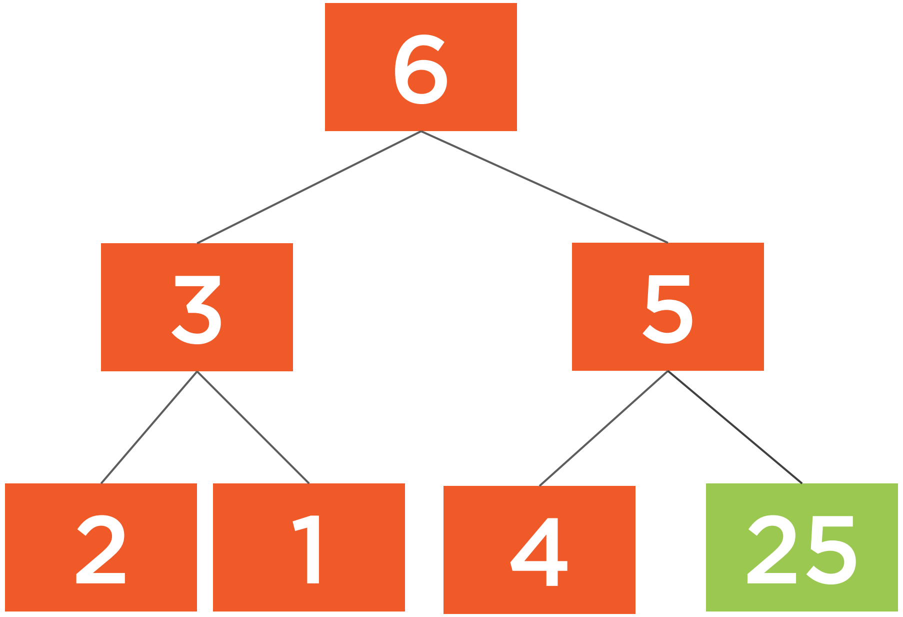
6	3	5	2	1	4	7

Thread 1	Thread 2
7	25



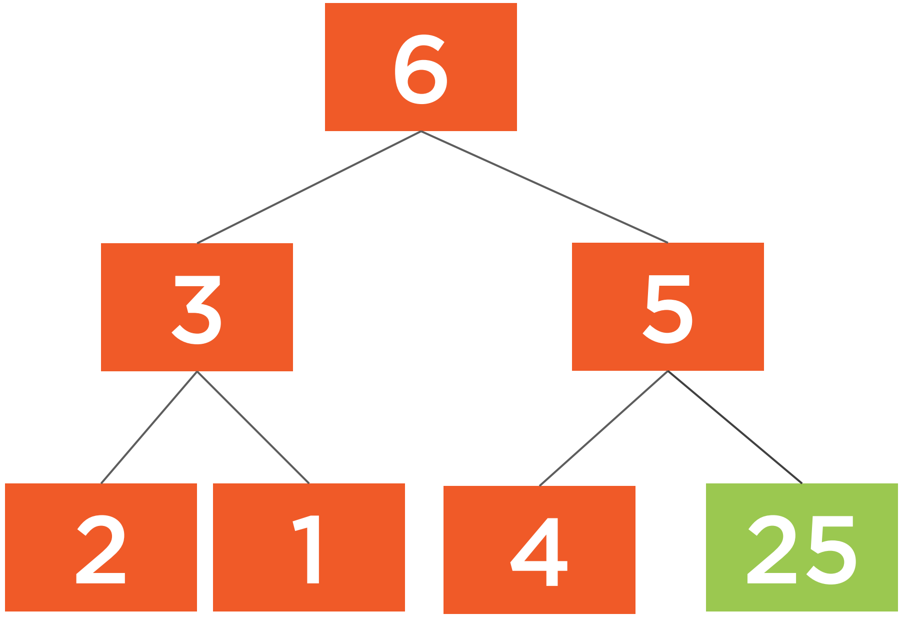
6	3	5	2	1	4	7

Thread 1	Thread 2
7	25



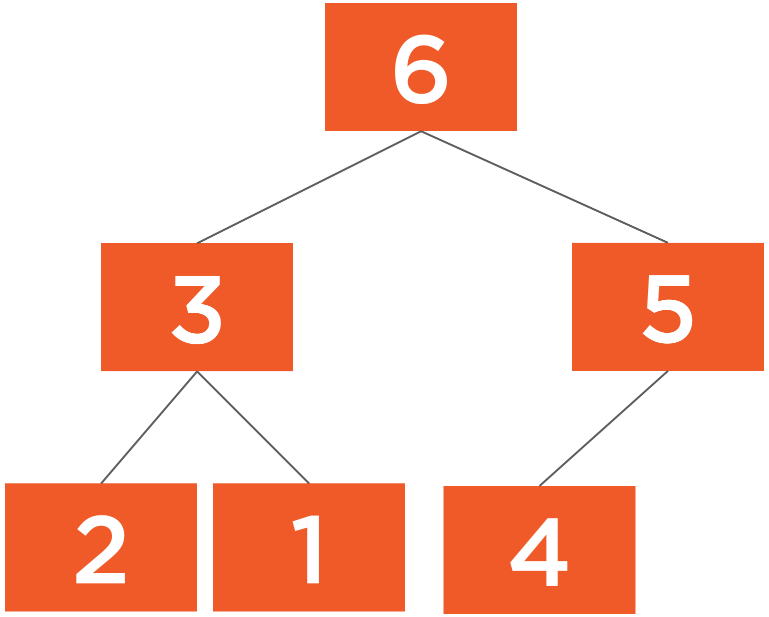
6	3	5	2	1	4	25

Thread 1	Thread 2
~~7~~	25



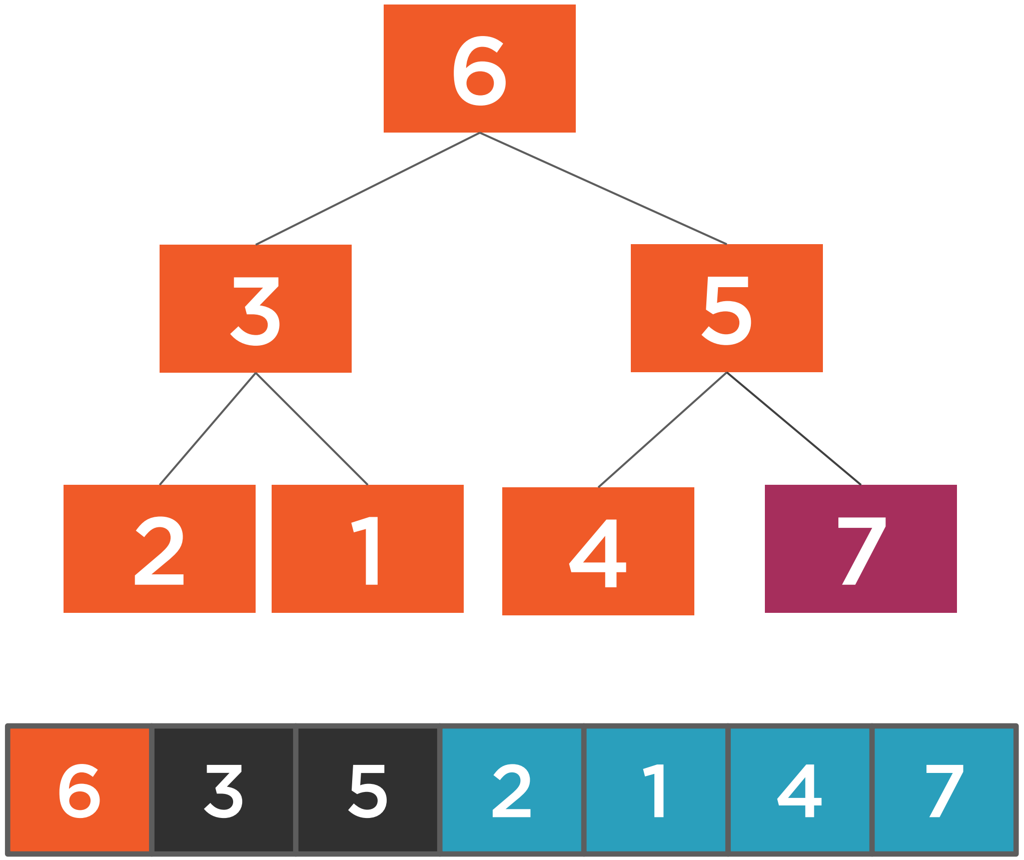
6	3	5	2	1	4	25

Thread 1	Thread 2
7	



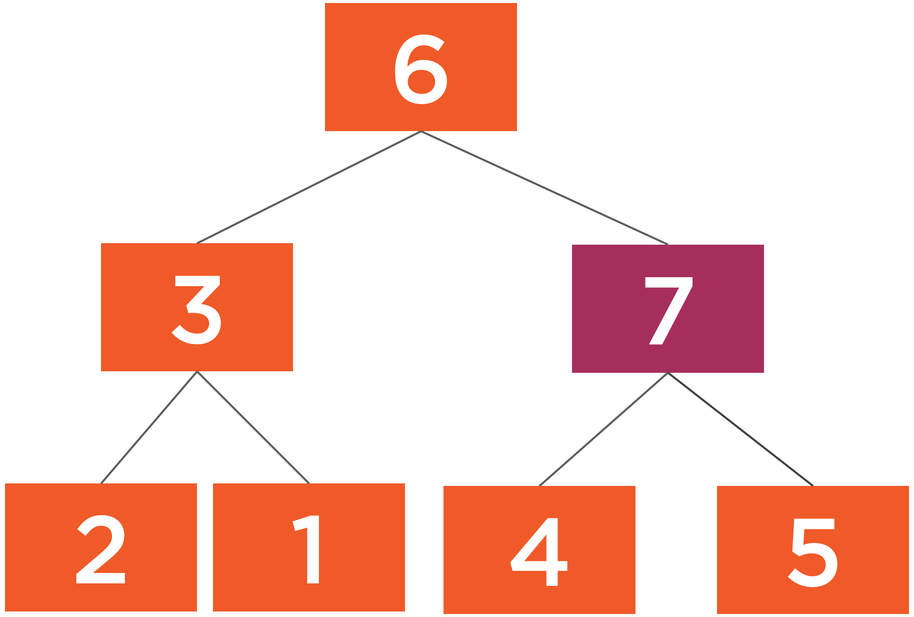
6	3	5	2	1	4	

Thread 1	Thread 2
7	



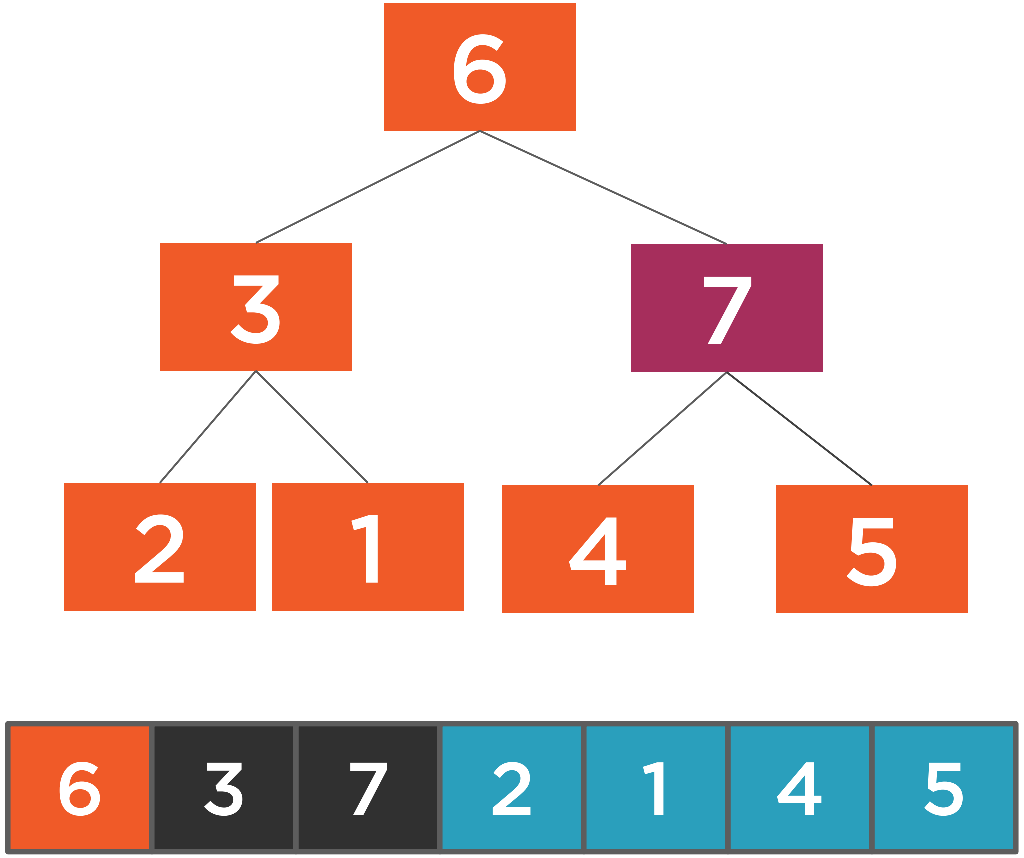
6	3	5	2	1	4	7

Thread 1	Thread 2
7	

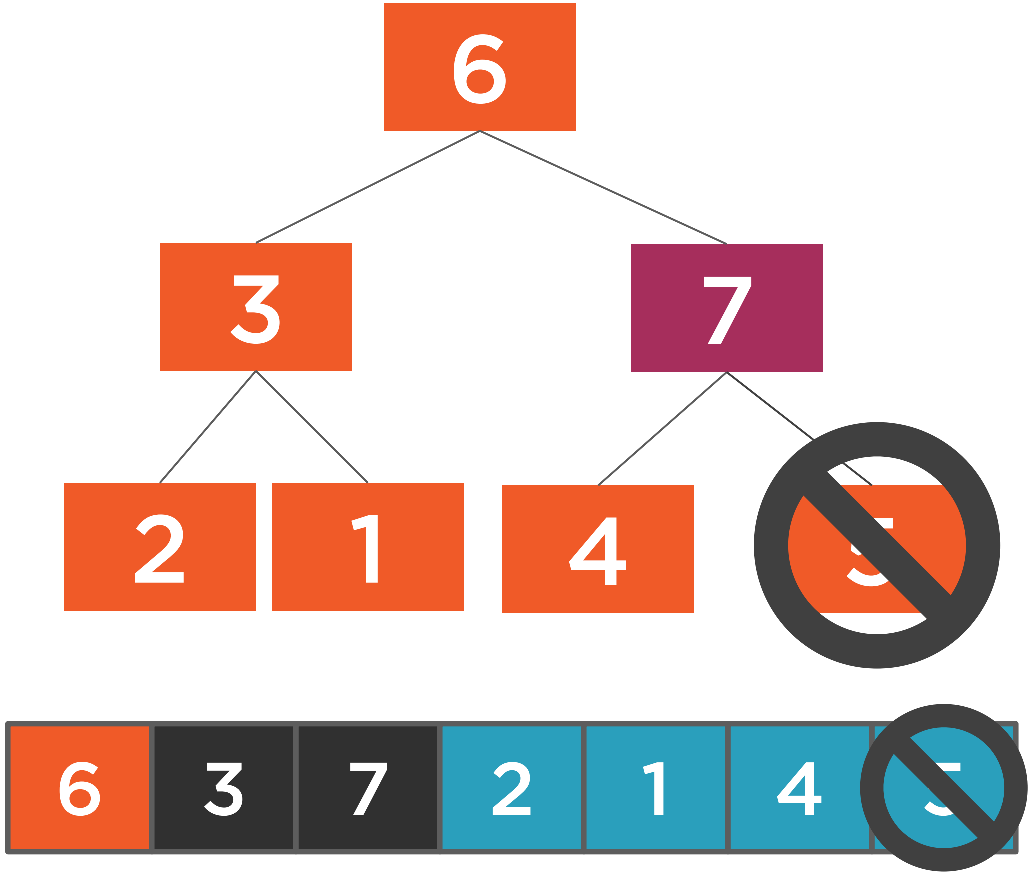


6	3	7	2	1	4	5

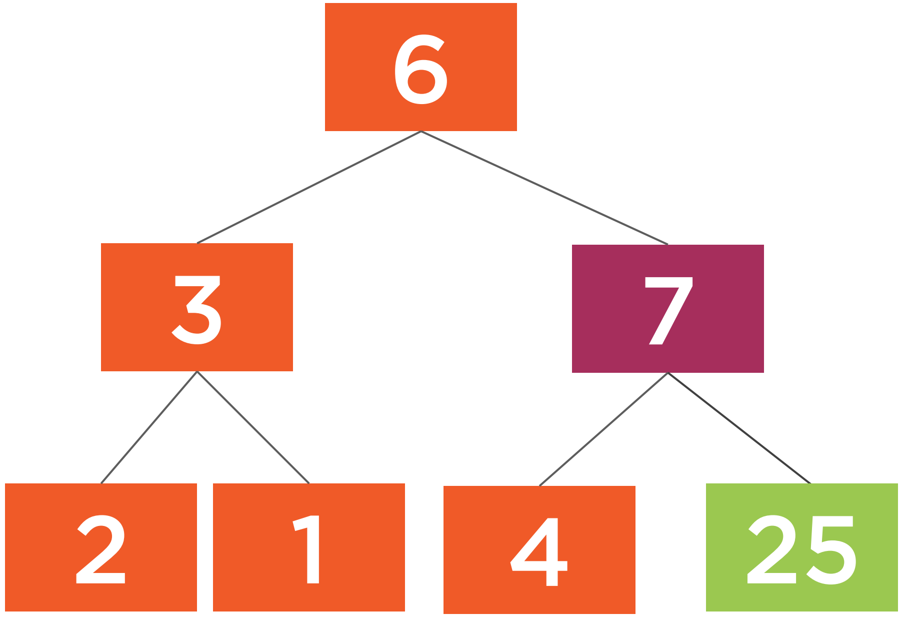
Thread 1	Thread 2
7	25



Thread 1	Thread 2
7	25



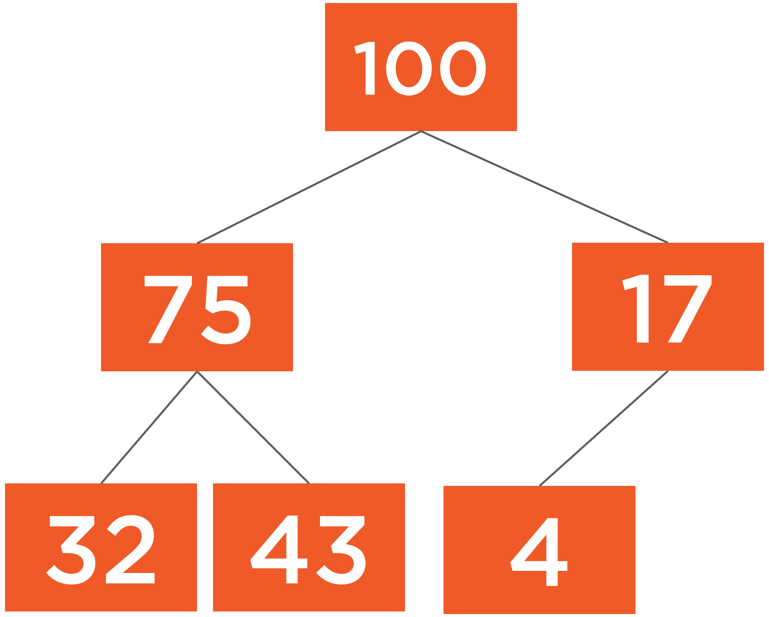
Thread 1	Thread 2
7	25



6	3	7	2	1	4	25

Concurrent updates to non-concurrency-safe collections can lead to unexpected behavior and data loss

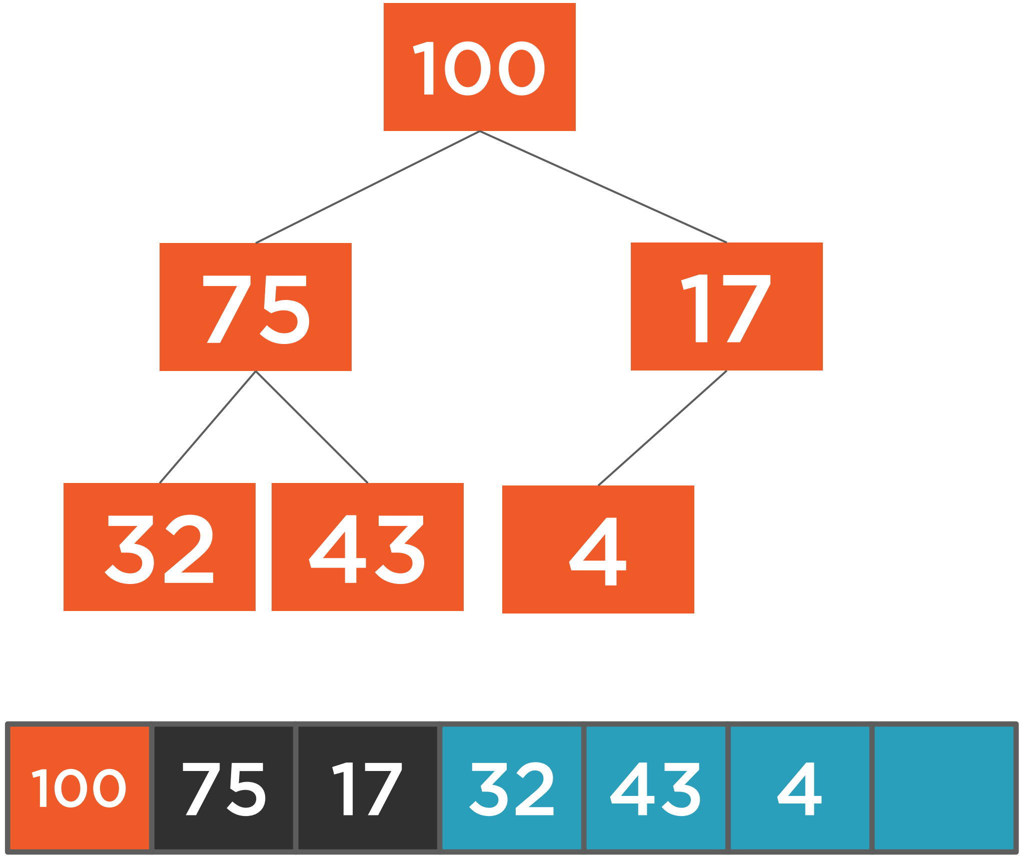
Thread 1	Thread 2



100	75	17	32	43	4	

Dequeue

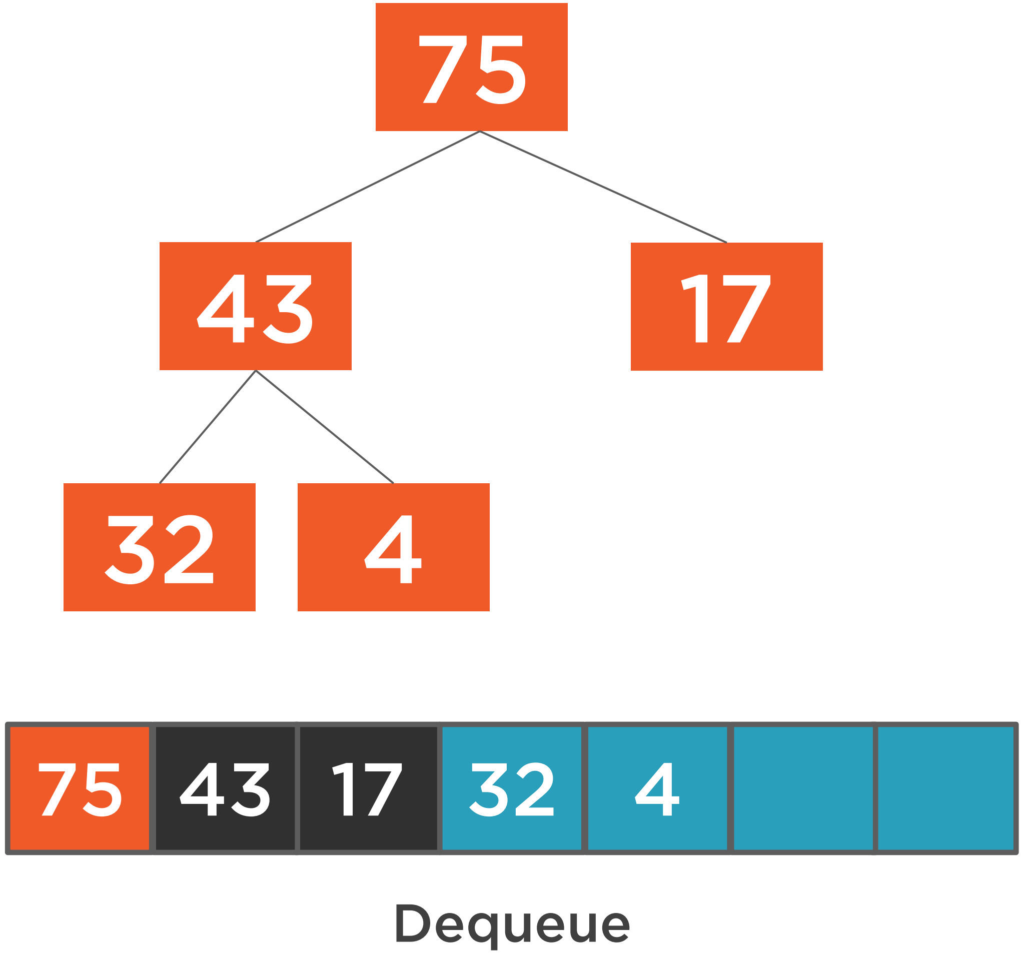
Thread 1	Thread 2
100	



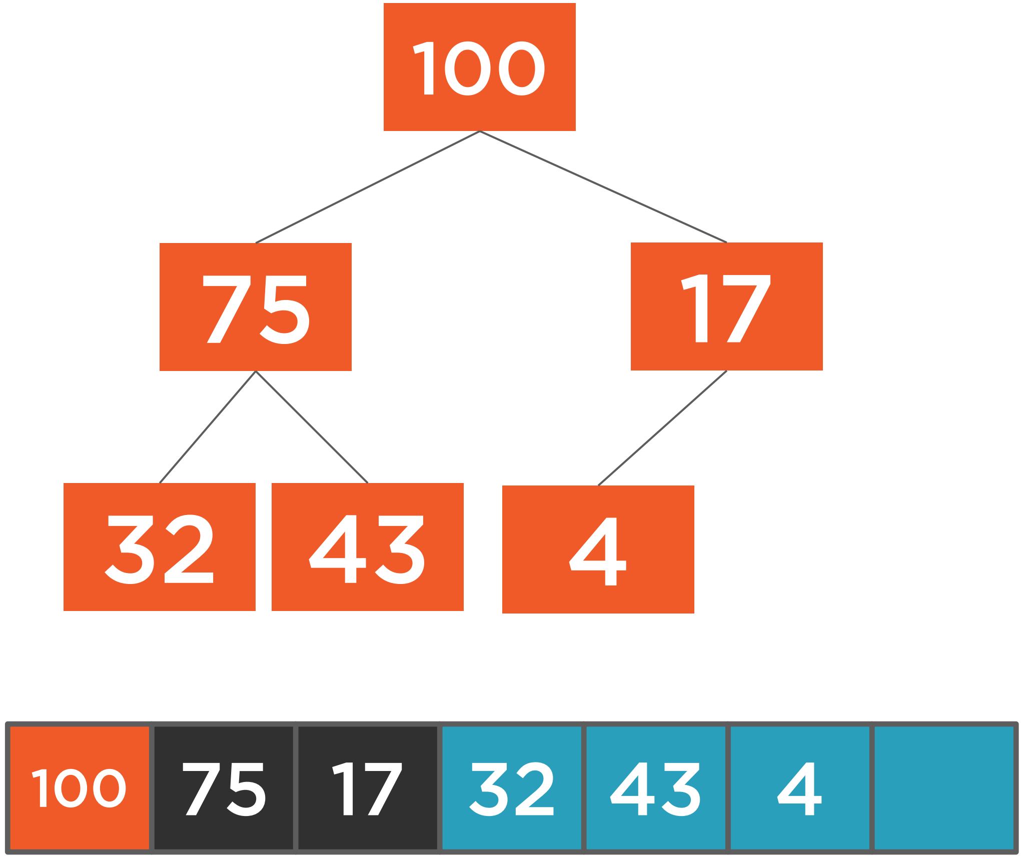
100	75	17	32	43	4	

Dequeue

Thread 1	Thread 2
100	

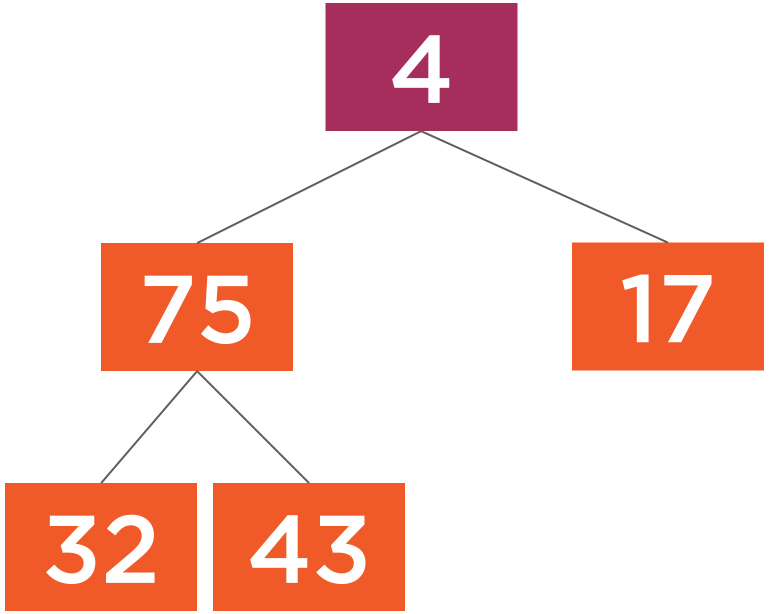


Thread 1	Thread 2
100	



Dequeue

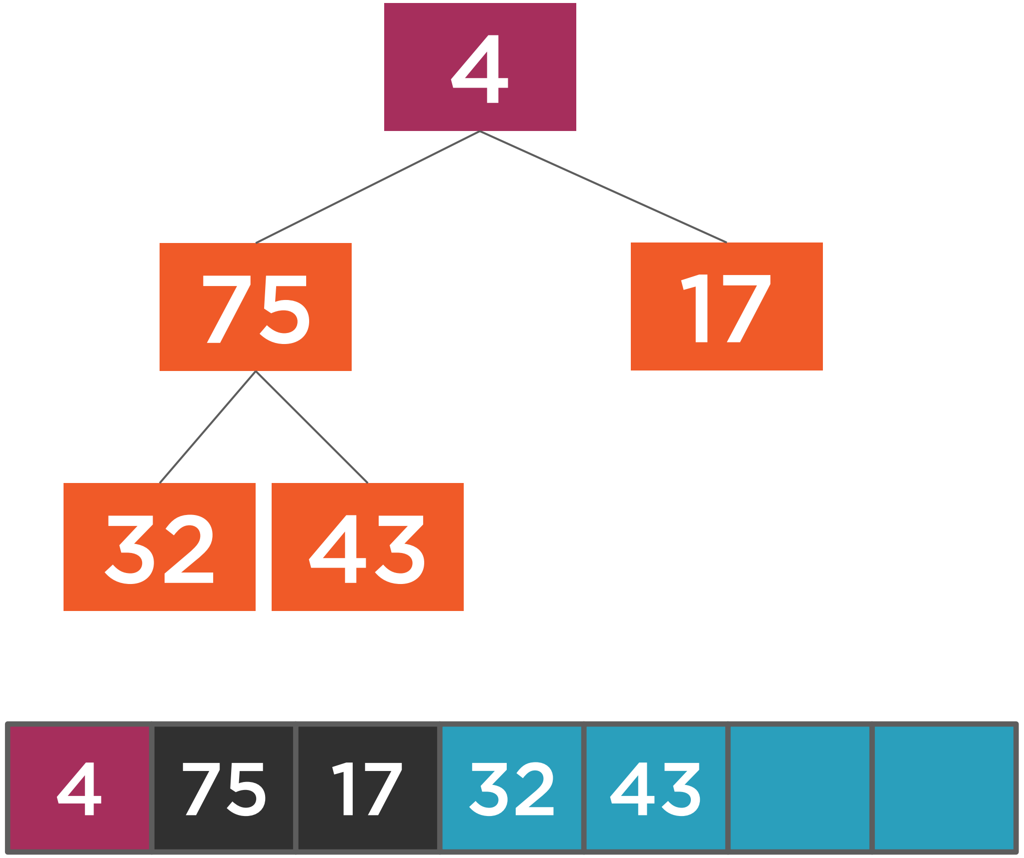
Thread 1	Thread 2
100	



4	75	17	32	43		

Dequeue

Thread 1	Thread 2
100	4



Dequeue

Caller Synchronization

The caller is responsible for ensuring all access to the collection is performed in a concurrency-safe manner

Caller Synchronization

Allows concurrent-safe access to non-concurrency-safe collections

No overhead when the collection is used non-concurrently

The caller can determine the optimal synchronization approach

```
object jobsLock = new object();

// ...

lock(jobsLock) {

    jobs.Enqueue(new Job(...));

}


lock(jobsLock) {

    Job nextJob = jobs.Dequeue();

    nextJob.Process();

}
```

◀ Create the object that will be used as the shared monitor lock object

◀ Take the lock before enqueuing jobs

◀ Lock before dequeuing the next job

◀ The job can be processed

```
object jobsLock = new object();

// ...

while(jobs.Count > 0) {


  lock(jobsLock) {

    Job nextJob = jobs.Dequeue()

    nextJob.Process();

  }

}
```

◀ Count needs to be called within the same lock scope as the call to Dequeue

◀ The Process method is holding the lock open.

```
while(jobs.Count > 0) {
  Job nextJob = null;

  lock(jobsLock) {
    if(jobs.Count > 0) {
      nextJob = jobs.Dequeue();
    }
  }
  if(nextJob != null) {
    nextJob.Process();
  }
}
```

- Check if the count is more than 0 while outside the lock
- Take the lock
- Check the count again while under the lock
- While holding the lock, Dequeue the next job
- If we dequeued a job, then process it

Caller Synchronization Using Monitor Lock

Pros

Non-concurrent-safe collections can be used in concurrent environments

Easy to implement

Cons

Caller is responsible for all thread synchronization

Readers block other readers

Easy to implement wrong

CODE – Caller Non-Locking Client and then locking client

Collection Synchronization

The caller is responsible for ensuring all access to the collection is performed in a concurrency-safe manner

Collection Synchronization

Monitor Locking

A single monitor lock is used to serialize access to the container

Monitor locks are very light-weight

Readers block other readers

Reader/Writer Locking

A single reader/writer lock is used to serialize write-access to the container

The reader/writer lock allows multiple readers concurrently while blocking writes

Concurrent reading can overcome performance costs versus monitors

Collection Synchronization

Monitor Locking

A single monitor lock is used to serialize access to the container

Monitor locks are very light-weight

Readers block other readers

Reader/Writer Locking

A single reader/writer lock is used to serialize write-access to the container

The reader/writer lock allows multiple readers concurrently while blocking writes

Concurrent reading can overcome performance costs versus monitors

The caller must still avoid non-concurrency-safe access patterns

```
object syncLock = new object();

// ...

public void Enqueue(T value) {

  lock(syncLock) {

    heap.Push(value);

  }

}
```

- A single synchronization object is used to serialize access to the priority queue
- The lock is taken during any operation that requires access to the heap

CODE – Locking Queue

Reader Writer Locks

The .NET ReaderWriterLockSlim class used to provide concurrent readers while serializing all writers.

```
var rwLock = new ReaderWriterLockSlim();

// ...

public void Enqueue(T value) {

  rwLock.EnterWriteLock();

  try

  {

    heap.Push(value);

  }

  finally

  {

    rwLock.ExitWriteLock();

  }

}
```

- A single ReaderWriterLockSlim instance serializes writes and blocks writes while allowing concurrent reads.

- The write lock is entered before a non-concurrency-safe operation. All reads and writes are blocked until this is exited.

- The non-concurrent-safe operation runs within a try-block

- In the finally-block the write lock is exited

```
var rwLock = new ReaderWriterLockSlim();

// ...

public T Peek() {

  rwLock.EnterReadLock();

  try

  {

    return heap.Top();

  }

  finally

  {

    rwLock.ExitReadLock();

  }

}
```

- In a read-only method a read lock is used to allow concurrent readers while blocking writes

- The read operation is performed within a try-block

- When the number of readers is zero then writes will be allowed again

CODE – RW Locking Queue

Concurrent .NET Collections

ConcurrentDictionary<TK,TV>

ConcurrentQueue<T>

ConcurrentStack<T>

ConcurrentBag<T>

.NET Concurrent Collections

Not drop-in replacements for existing collection types

Prefer these types with code requiring concurrency-safe collections

ConcurrentQueue and ConcurrentStack are lock-free collections

```
using System.Collections.Concurrent;

//...

var queue = new ConcurrentQueue<int>();


queue.Enqueue(1);


int value;

if(queue.TryPeek(out value)) {

  Console.WriteLine(value);

}


if(queue.TryDequeue(out value)) {

  Console.WriteLine(value);

}
```

- The concurrent collections are in the System.Collections.Concurrent namespace
- Allocate a concurrent queue of integers
- Enqueue works the same as Queue<T>
- Peeking requires the “Try” pattern which avoids having to fail if the queue is empty
- Dequeuing also uses the “Try” pattern to avoid failure when the queue is empty