Data Structures That Make Video Games Go Round

AL-AFIQ YEONG

Exposing the Games Industry:

~~Data Structures That Make Video Games Go Round~~

One Data Structure At A Time

AL-AFIQ YEONG

Cppcon
The C++ Conference

2024
September 15 - 20

Games are … **COMPLICATED**

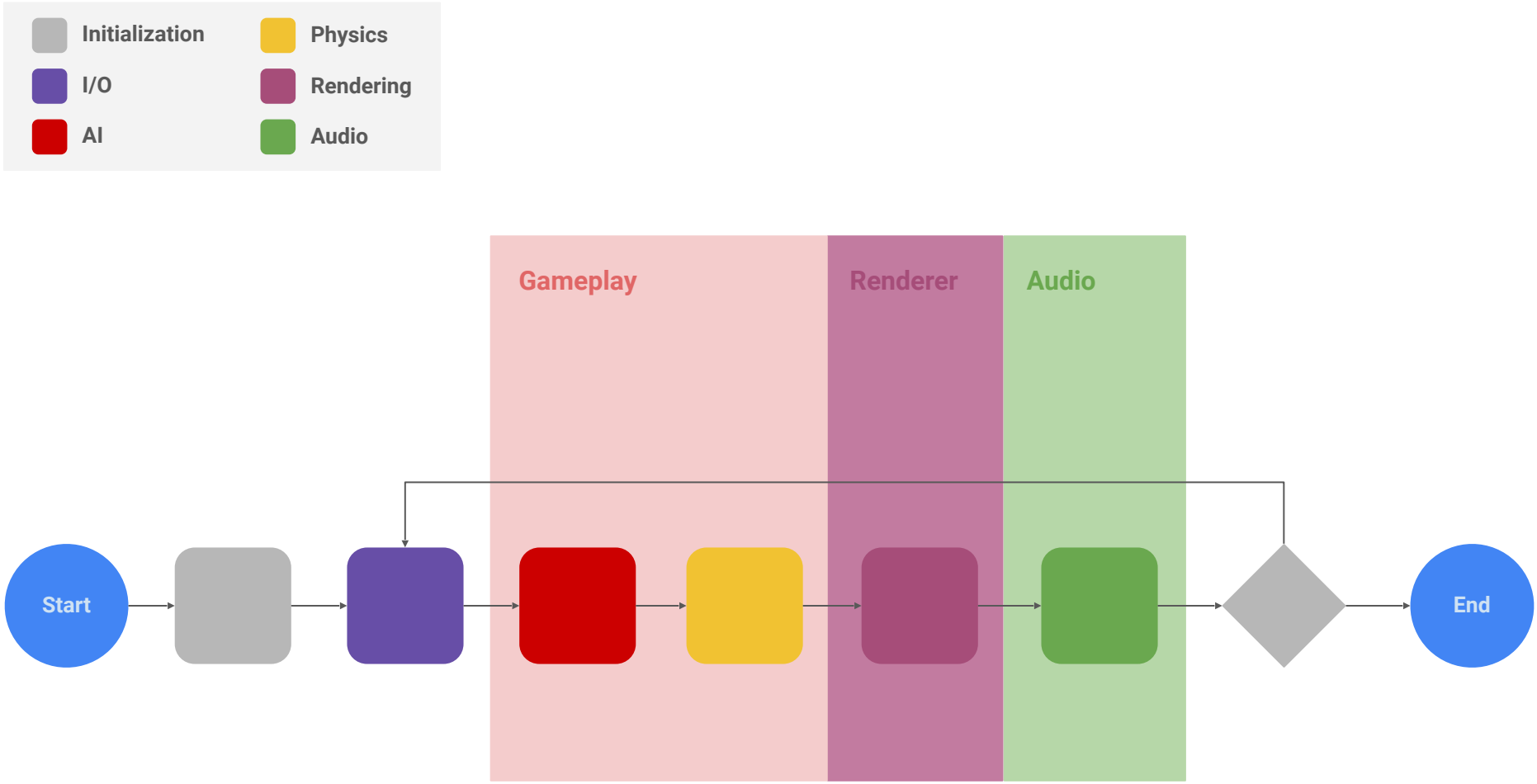
Back in the old days, games were:

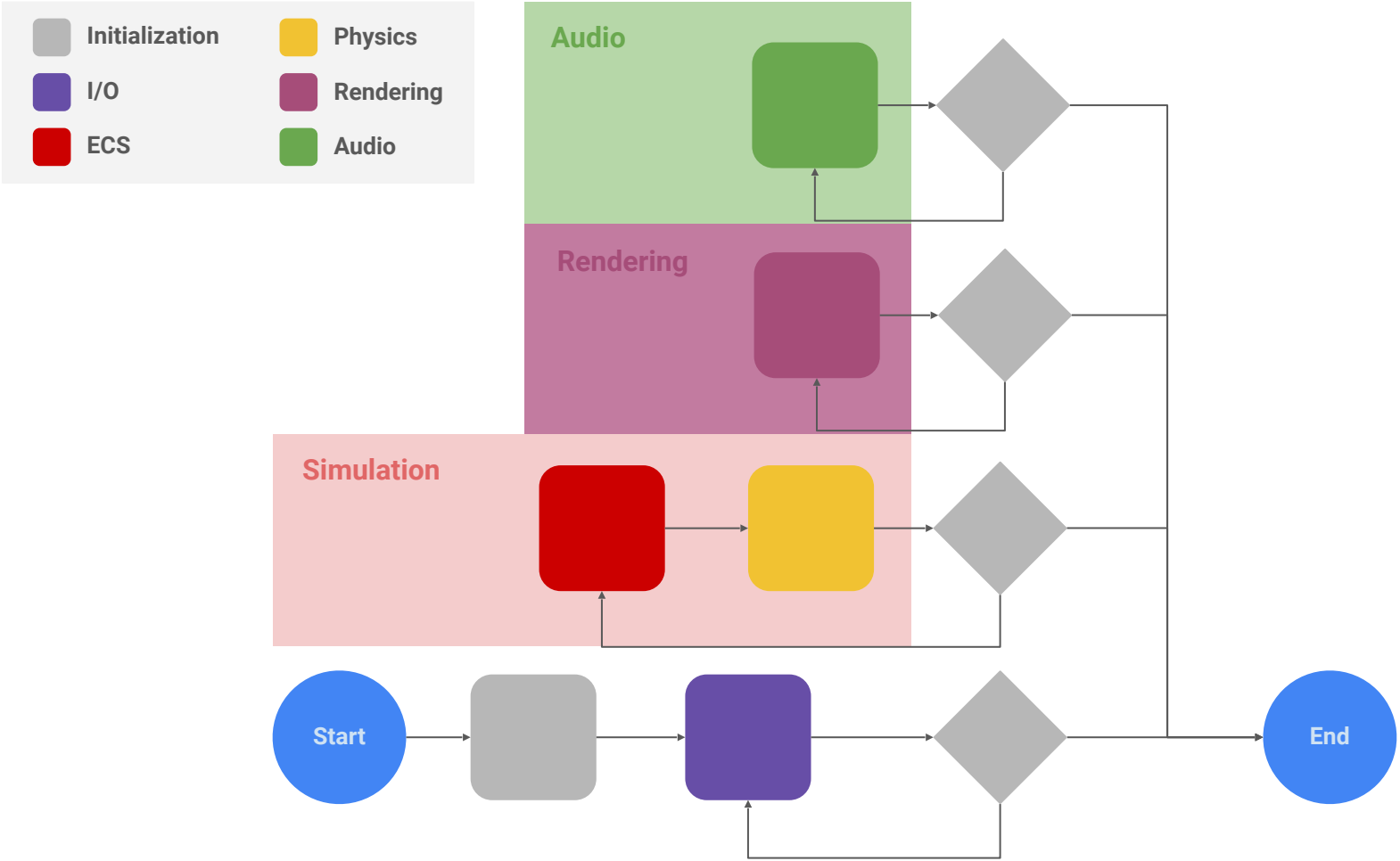
- Simple; 2D, 2.5D mostly.
- Single threaded.
- Made from scratch (no game engine).
- Short release cycle.
- Small, by modern standards.

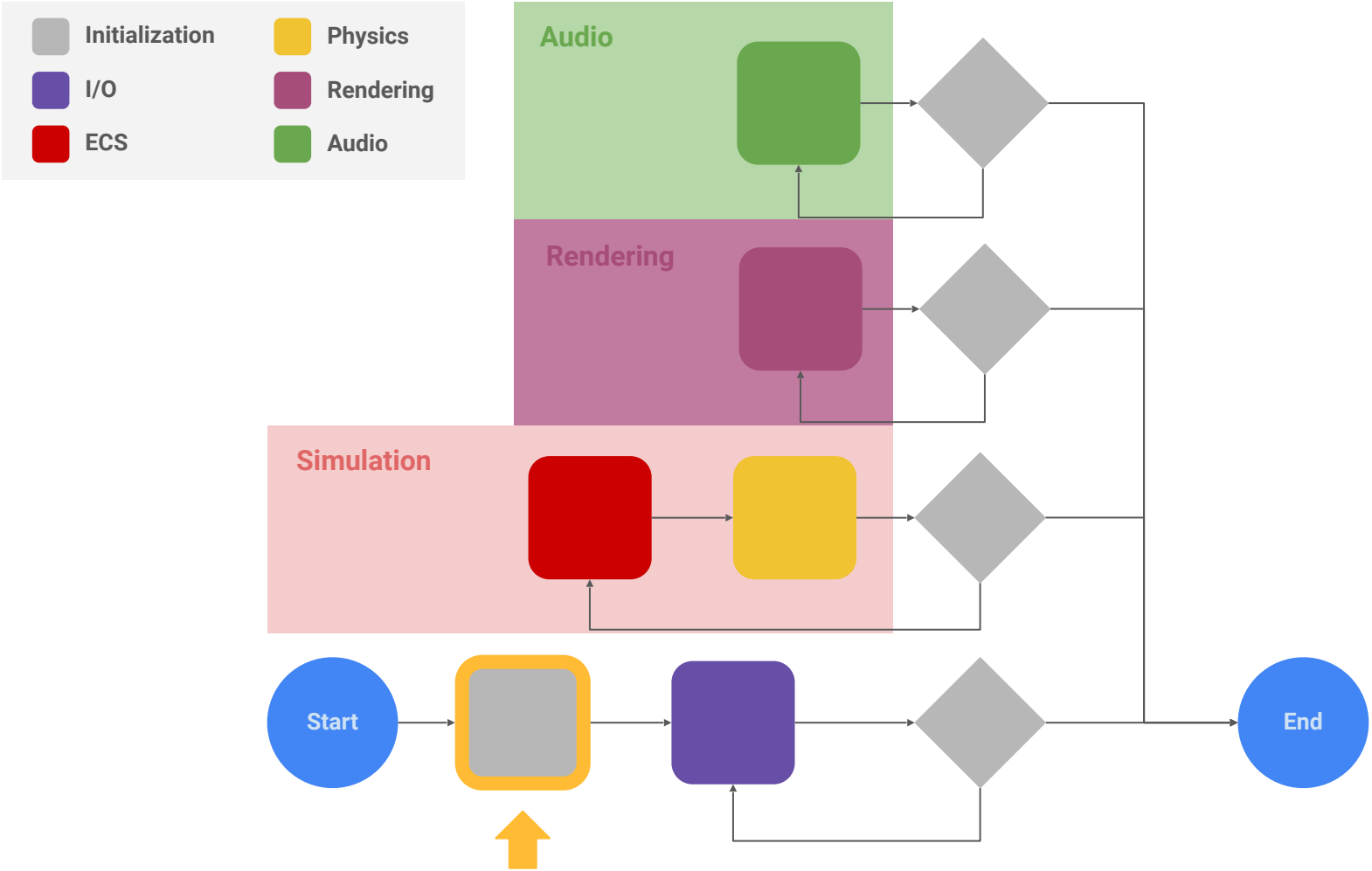
Today:

- 3D, Photorealistic, Physics based.
- Multi-threaded.
- Made with a game engine.
- Long development cycles.
- Large, complex codebases.

But the structure of a game is ***roughly the same***


Initialization
Physics
I/O
Rendering
AI
Audio
Gameplay
Renderer
Audio
Start
End


Initialization
Physics
I/O
Rendering
ECS
Audio
Audio
Rendering
Simulation
Start
End


Initialization
Physics
I/O
Rendering
ECS
Audio
Audio
Rendering
Simulation
Start
End

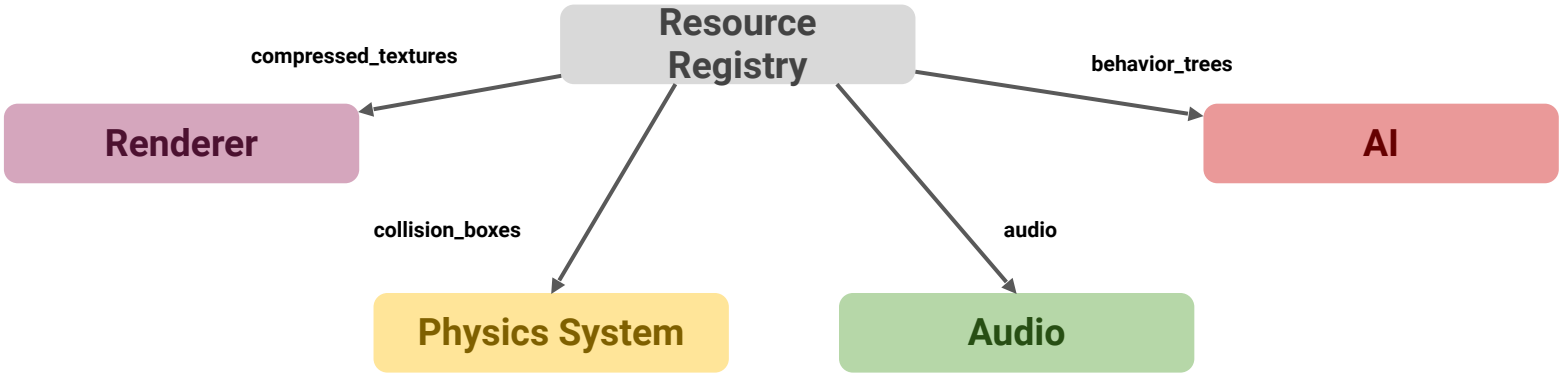
Initialization

- Environment Variables.
- Error Handler.
- Job system.
- Resource Registry.
- Reads boot data.
- Listens to kernel message pump.

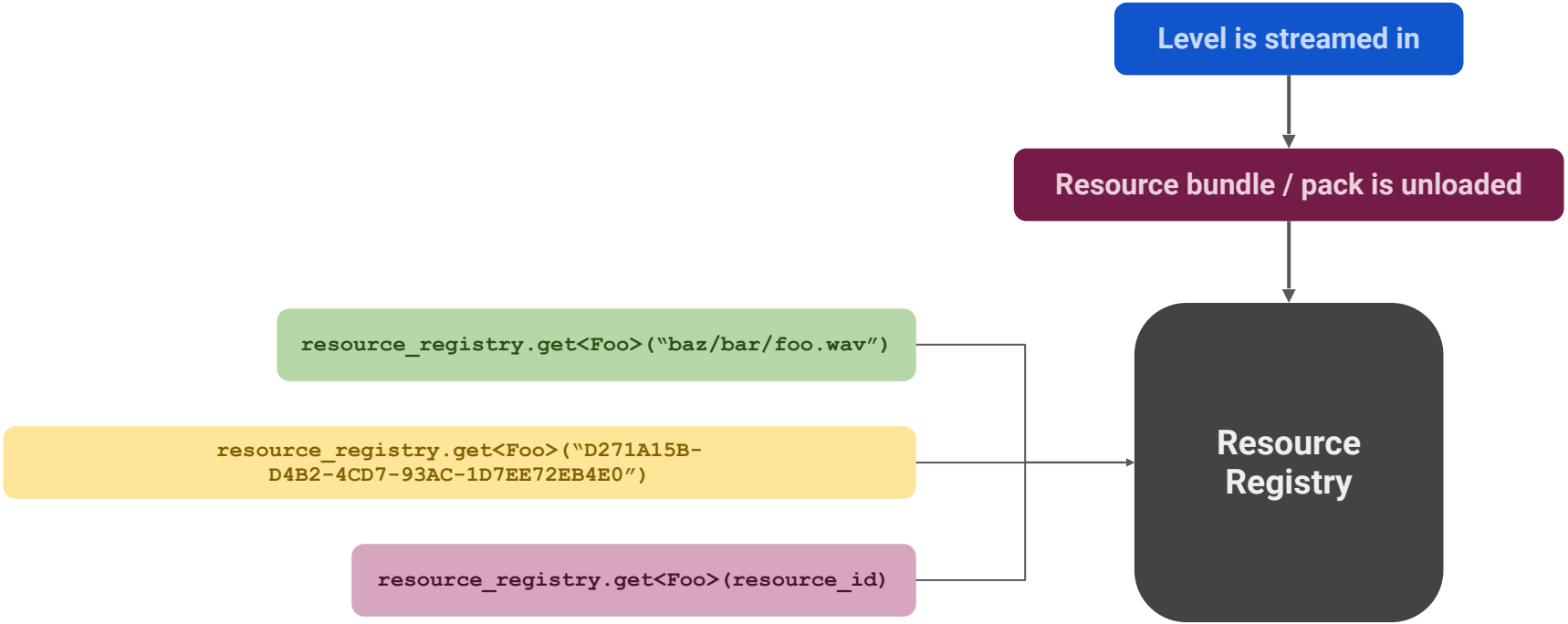
Resource Registry

Games are constructed with a wide variety of resources such as textures, meshes, materials, sounds, animations and many more.

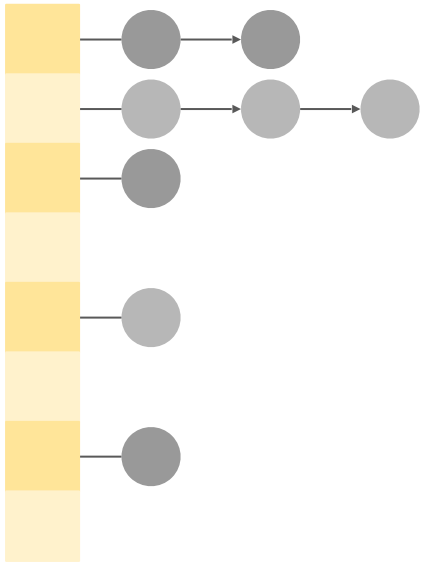
The resource registry acts as a place to store, manage and manipulate those resources.



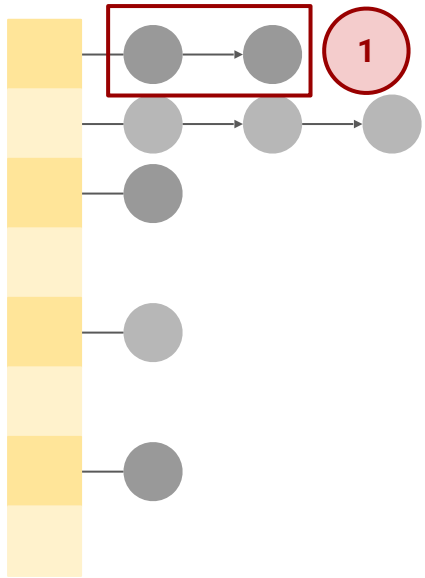
Resource Registry



Unordered Map

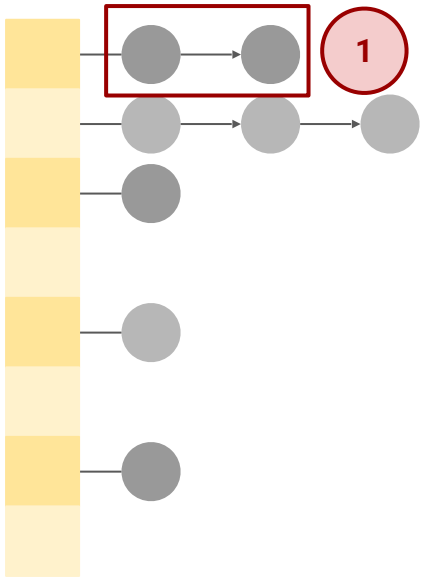


Unordered Map



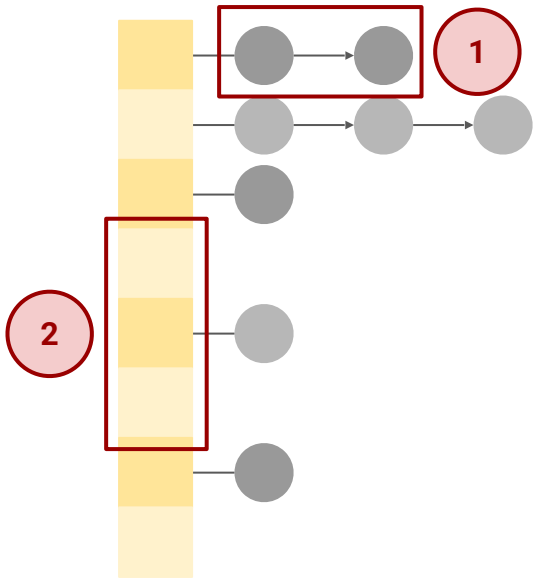
Unordered Map

1. Not cache friendly. A lot of pointer chasing especially when load factor is high.



Unordered Map

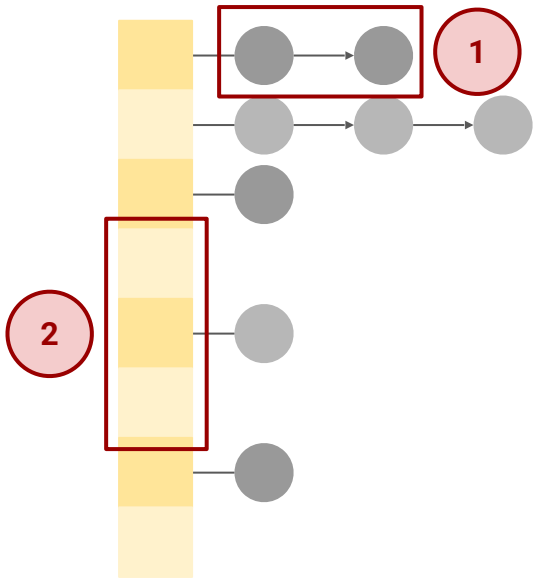
1. Not cache friendly. A lot of pointer chasing especially when load factor is high.



Unordered Map

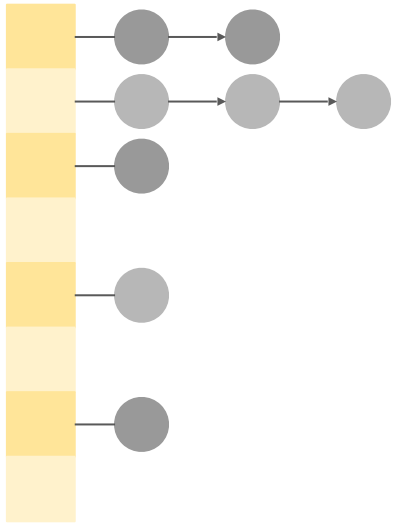
1. Not cache friendly. A lot of pointer chasing especially when load factor is high.

2. High variance. Elements are widely distributed between buckets.



Open Addressing Hash Maps

- Simplest implementation only require a contiguous block of memory.
- Elements are more densely packed.
- Better cache performance.
- Collisions are resolved via probing. Probing methods include:
  - Linear Probing.
  - Quadratic Probing.
  - Double Hashing.
  - **Robin Hood Hashing**.



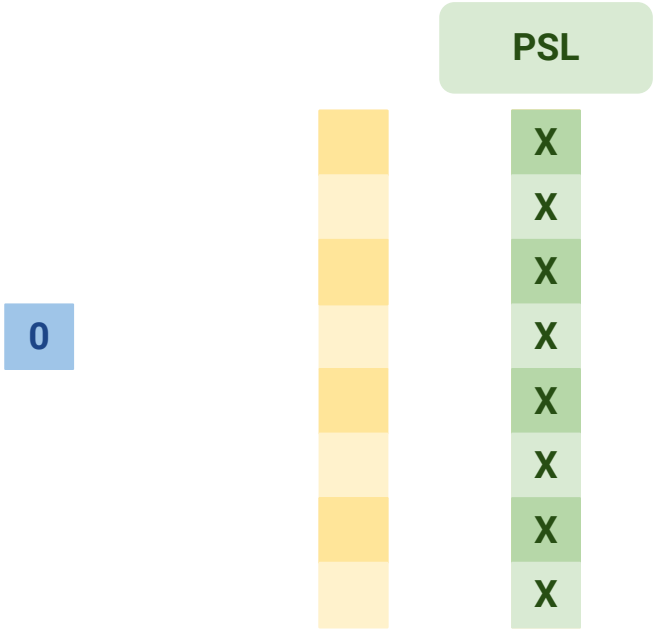
Open Addressing Hash Maps

- Simplest implementation only require a contiguous block of memory.
- Elements are more densely packed.
- Better cache performance.
- Collisions are resolved via probing. Probing methods include:
  - Linear Probing.
  - Quadratic Probing.
  - Double Hashing.
  - **Robin Hood Hashing**.

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

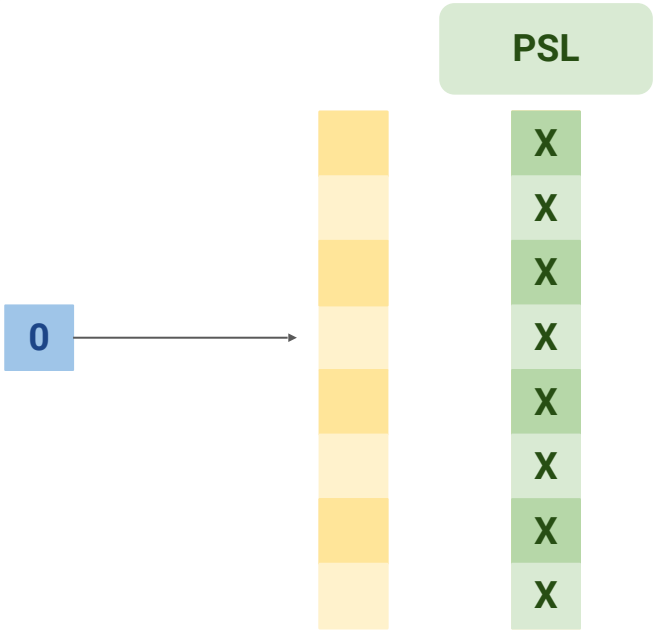


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

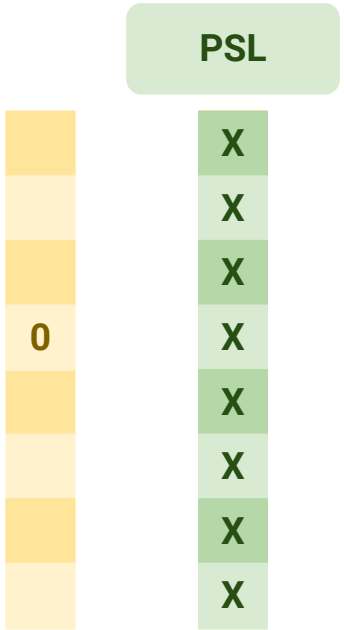


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

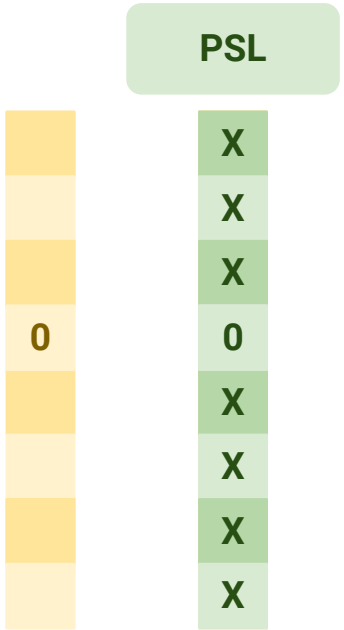


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

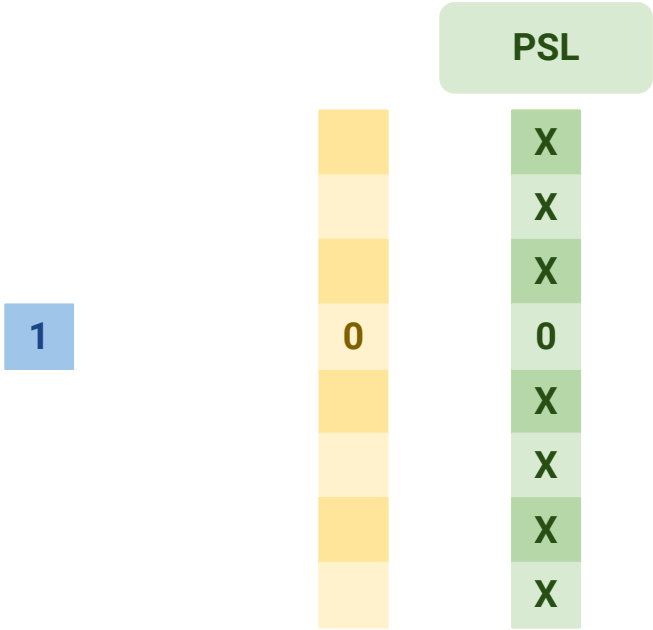


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

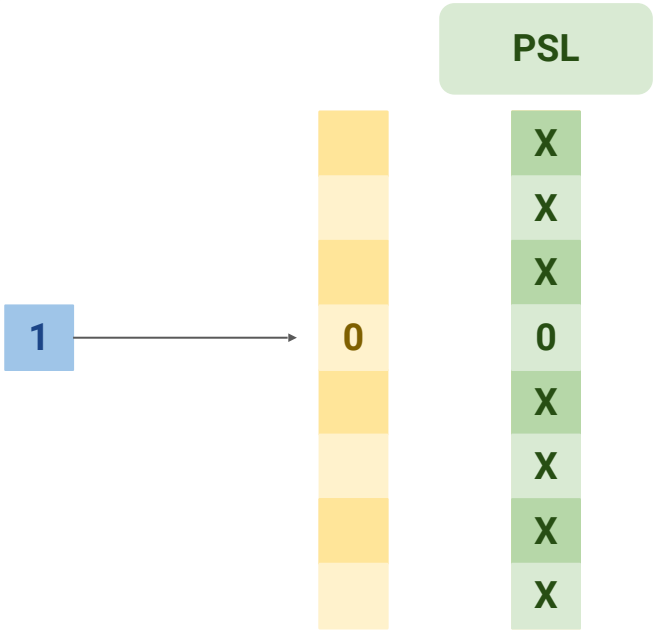


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

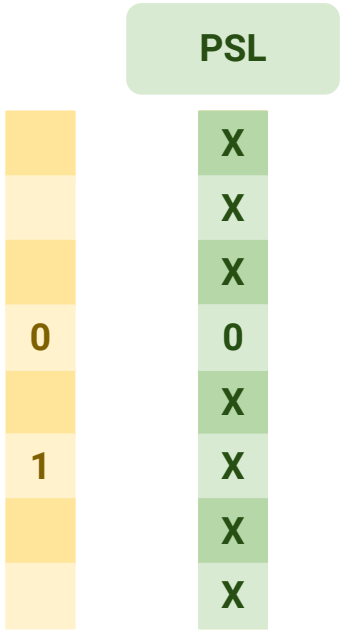


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

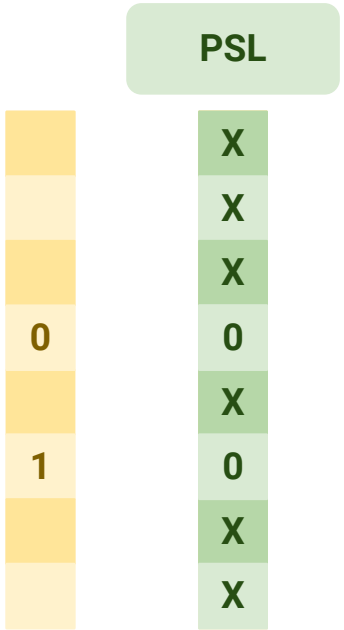


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

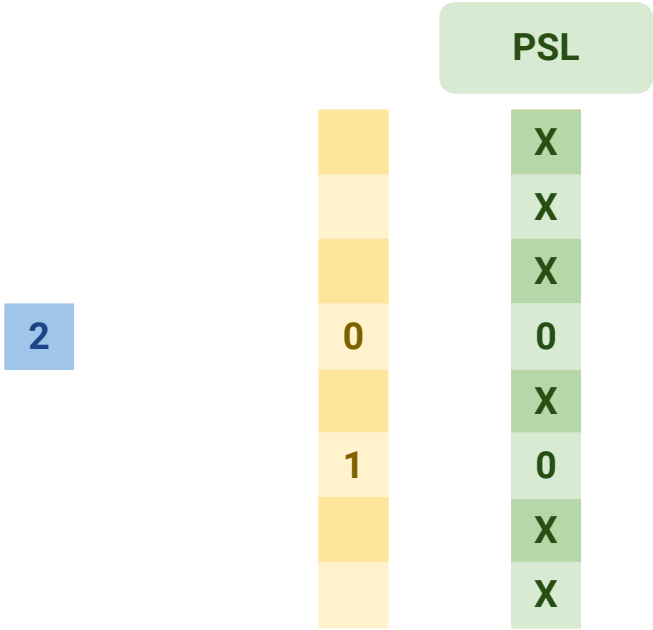


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

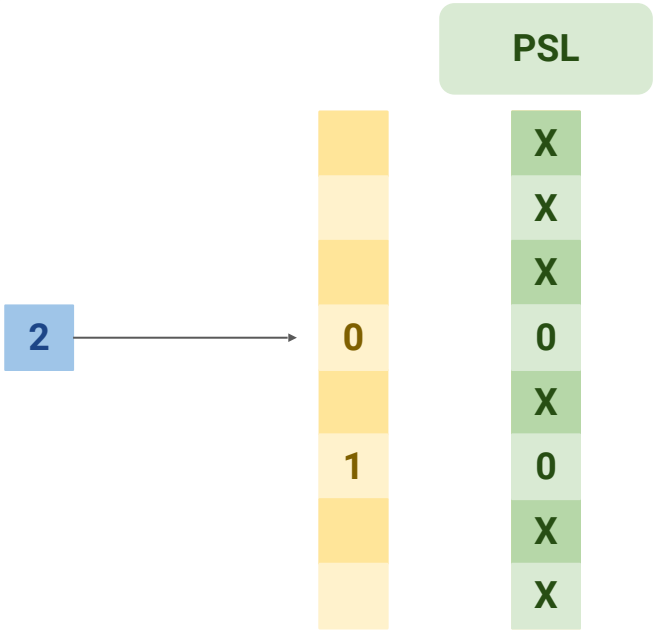


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

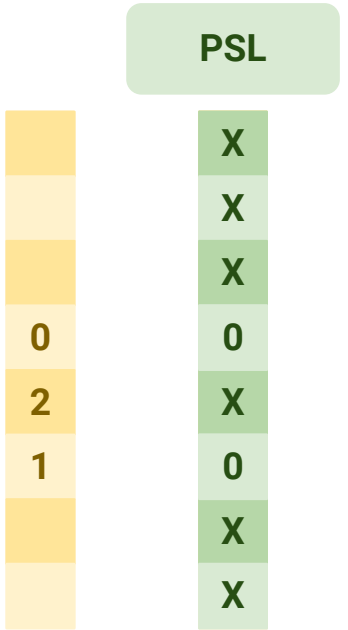


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

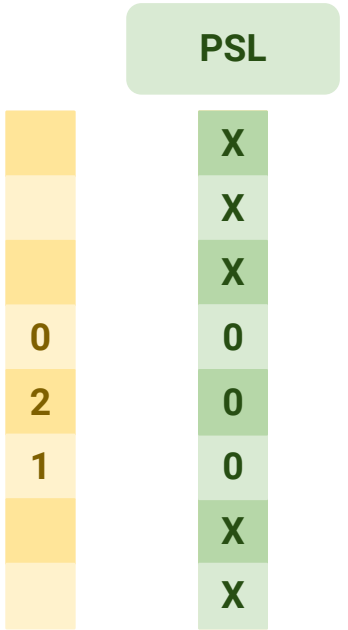


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

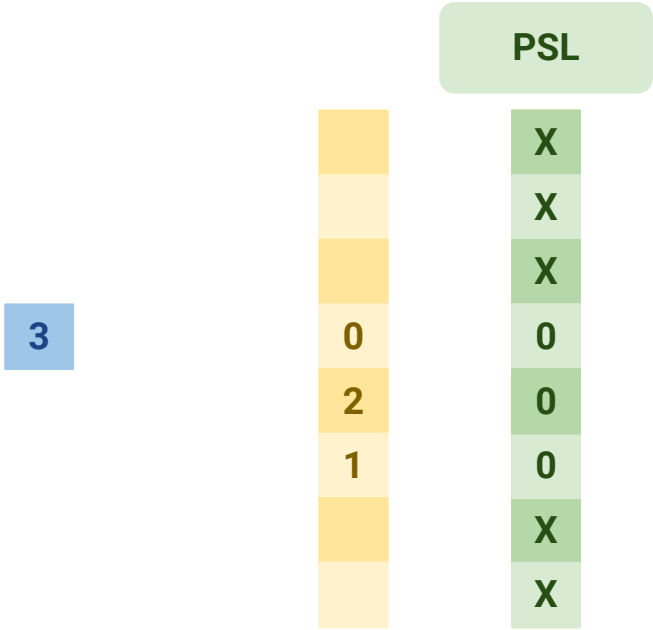


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

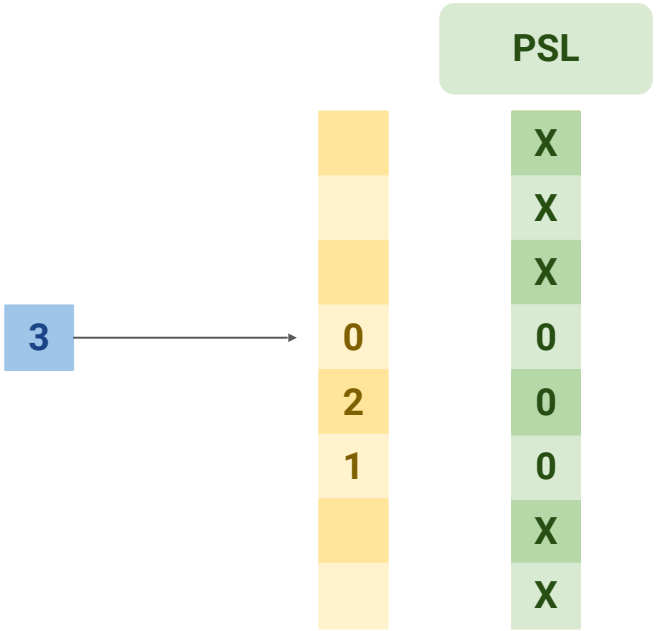


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

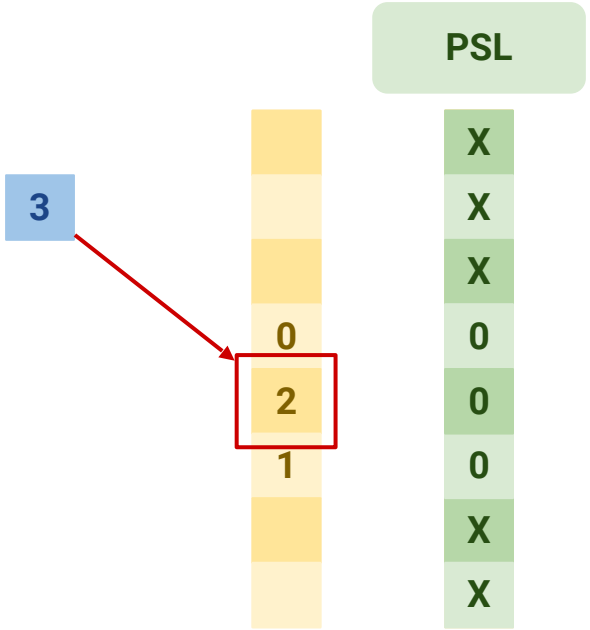


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

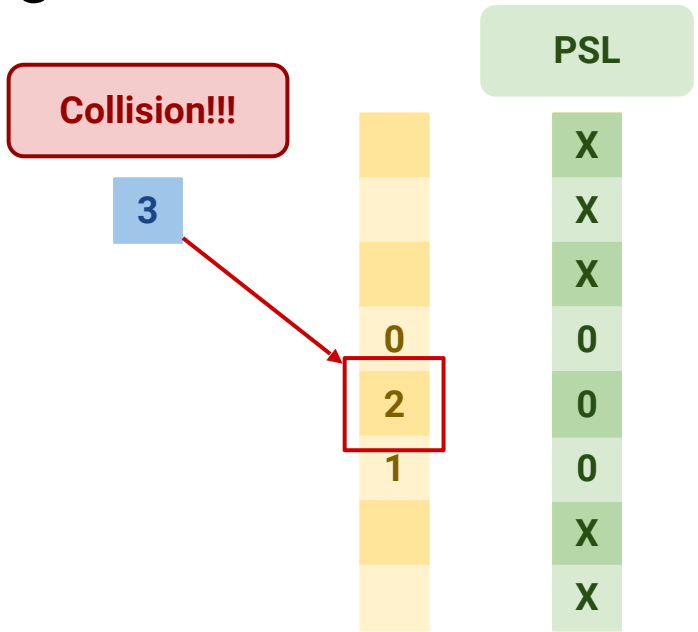


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

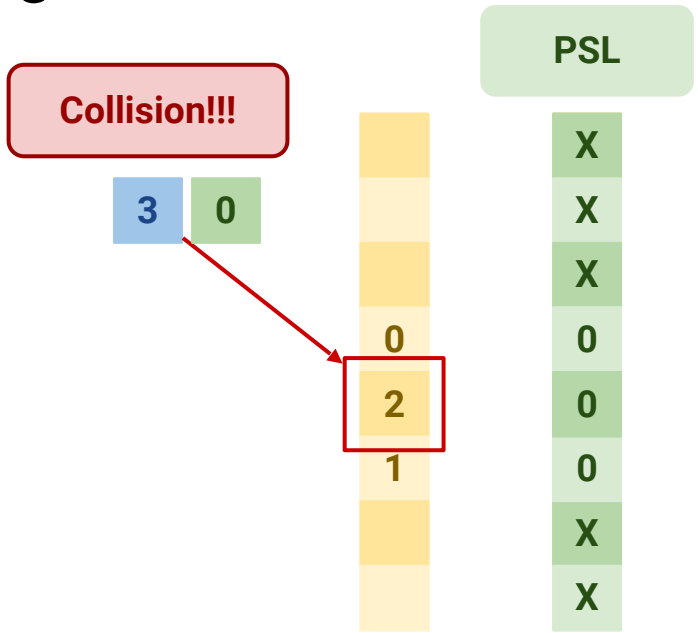


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

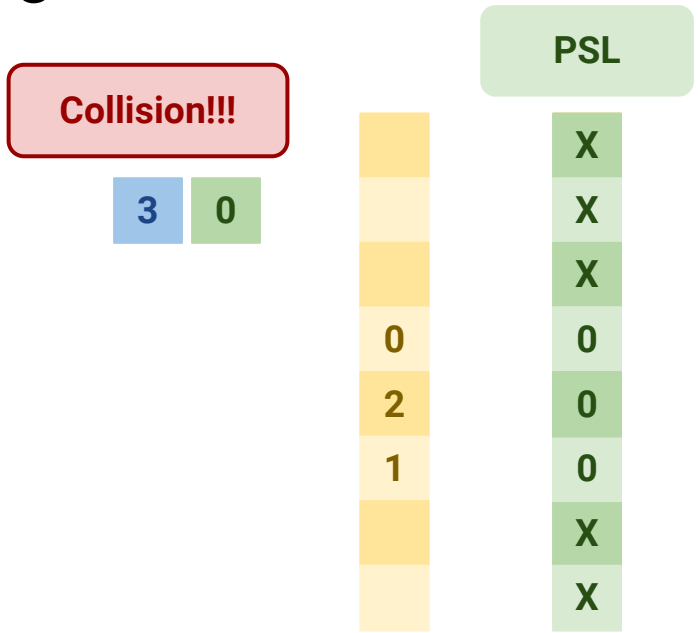


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

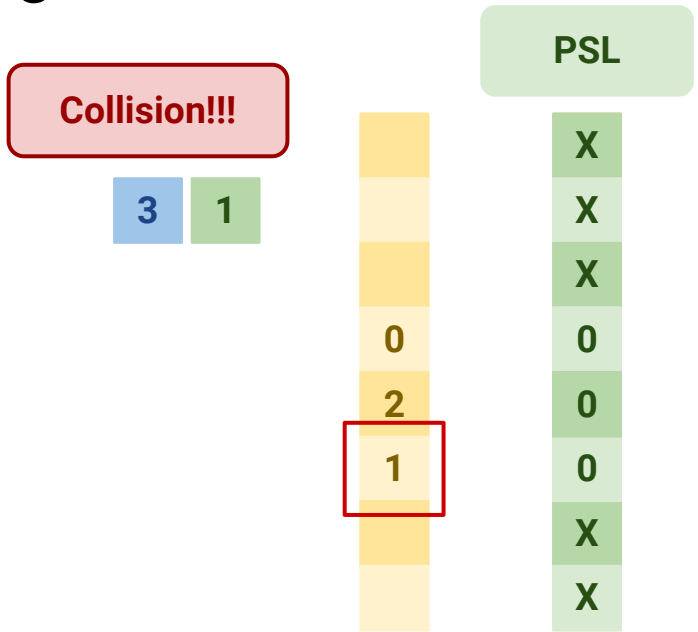


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

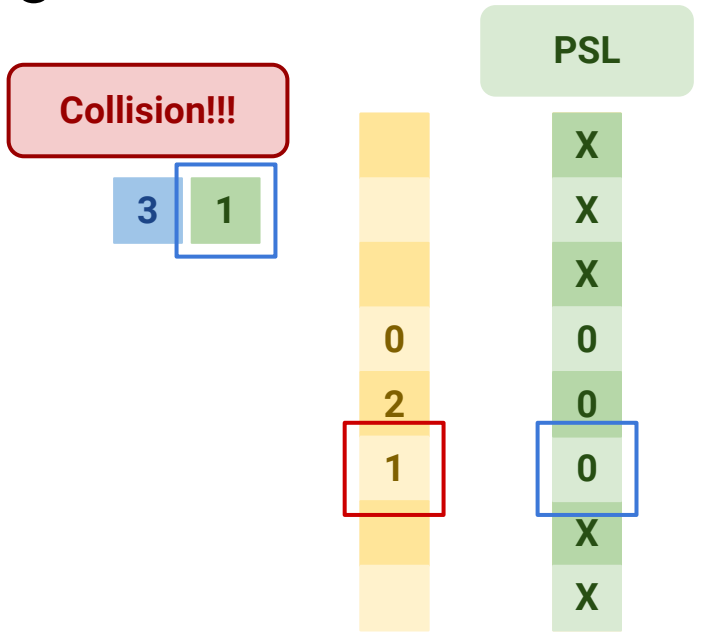


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

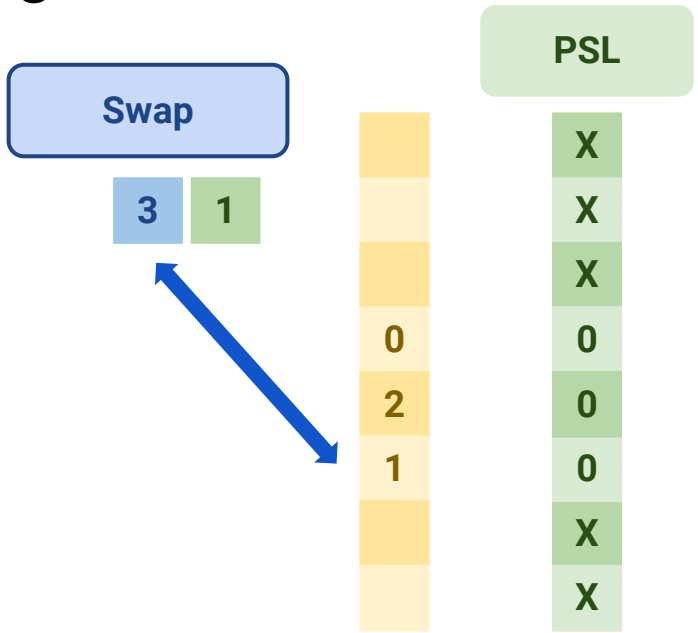


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

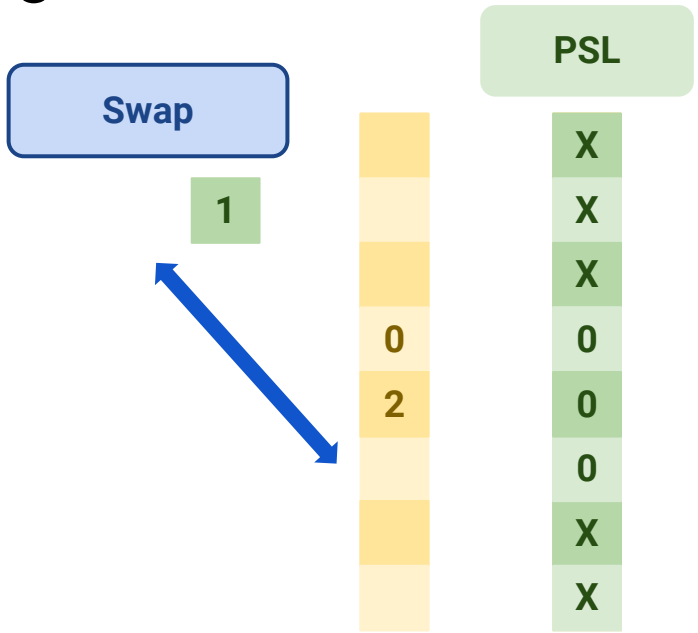


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

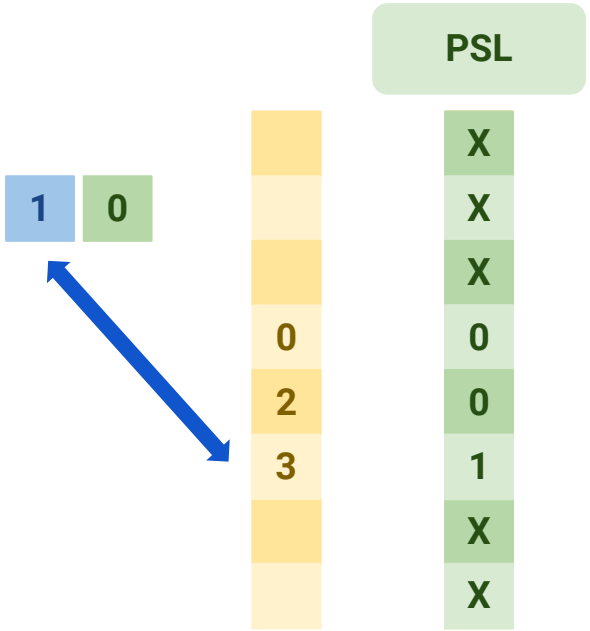


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

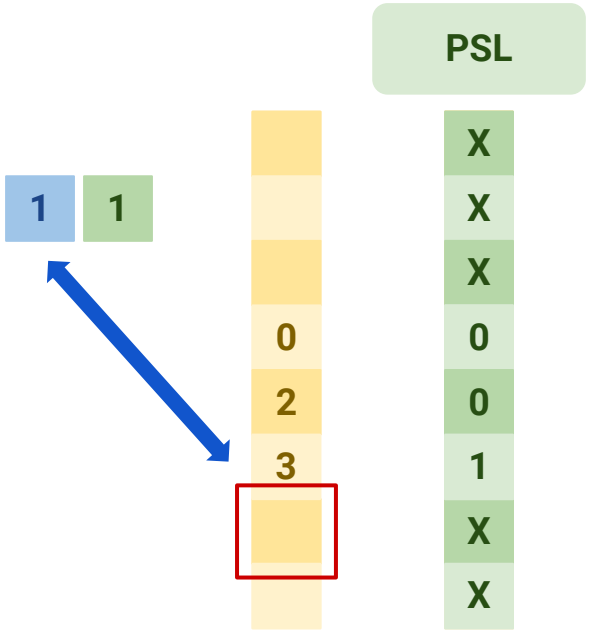


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

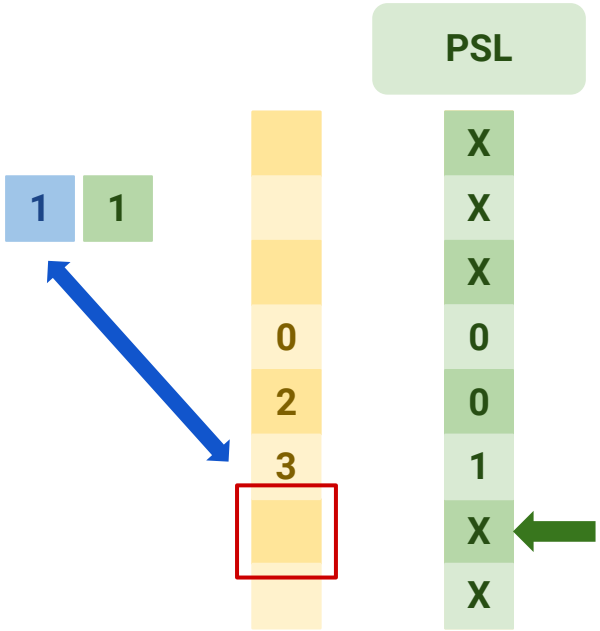


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

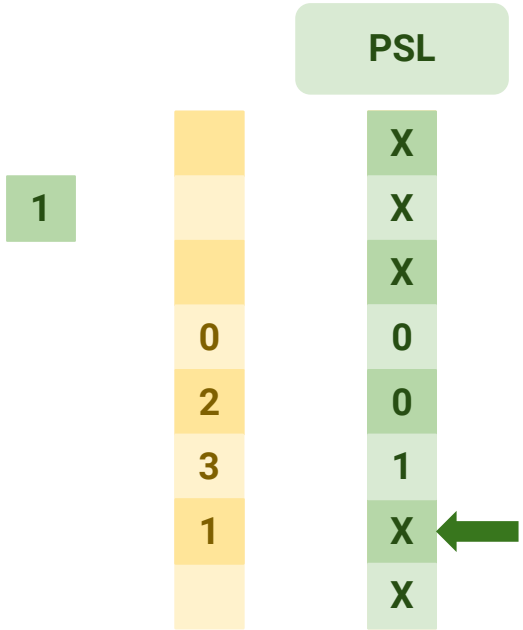


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.

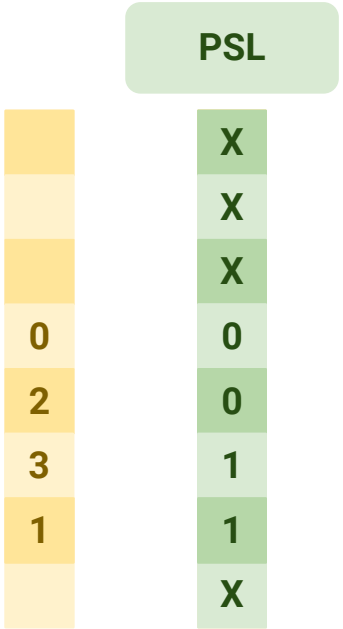


PSL - Probe Sequence Length

Linear Probed Robin Hood Hashing

Core algorithm,

1. Every bucket contains metadata about it's probe sequence length, PSL and is defaulted to some value.
2. On insert, elements start with a PSL value of 0.
3. PSL of the inserted element is compared to the PSL of the bucket.
4. If bucket's PSL value is the default, simply insert the element and store it's PSL.
5. If the bucket's PSL value is less than the element's, swap.
6. Probe to the next bucket and increment the PSL value of the swapped element.
7. Repeat steps 5 and 6 until an empty bucket(bucket with default PSL value) is found.



PSL - Probe Sequence Length

Better animation -> [Here!]

Linear Probed Robin Hood Hashing

Because the probe sequence length (PSL) keeps growing, inserted elements starts clustering around the mean of the container.

Ideally, you would keep the PSL for each element roughly the same.

- An element with a PSL of 1 isn't orders of magnitude faster than an element with a PSL of 3.
- They can exist within the same cache line.

References:

- Robin Hood Hashing should be your default Hash Table Implementation, Sebastian Sylvan.
- Robin Hood Hashing, Emmanuel Goossaert.
- Comprehensive C++ Hashmap Benchmarks 2022, Martin Leitner-Ankerl.
- martinus/unordered_dense.

What about std::vector?

std::vector is **GREAT**. 90% of the time that's all you really need.

- O(1) random access.
- O(1) insert and removal at the end.

But std::vector has a problem …

Iterator Invalidation

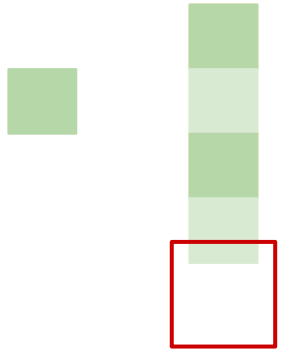
- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

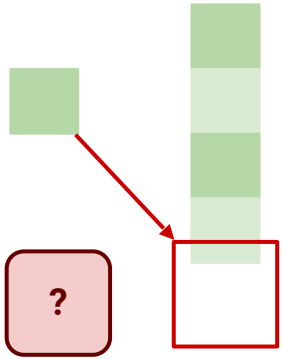
Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.



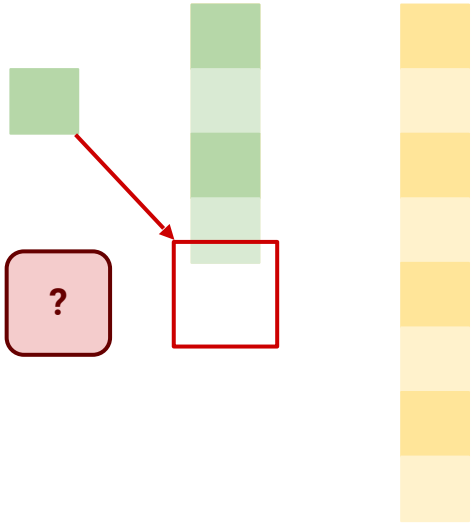
Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.



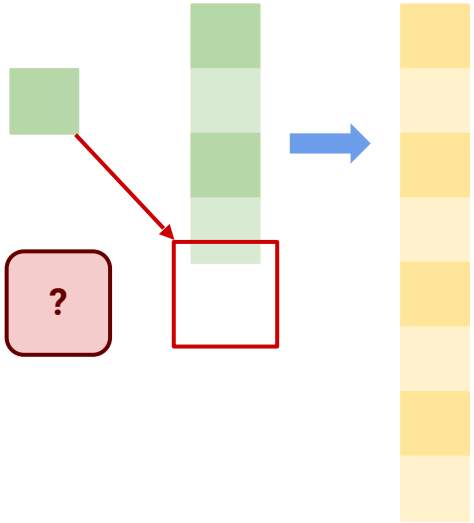
Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.



Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

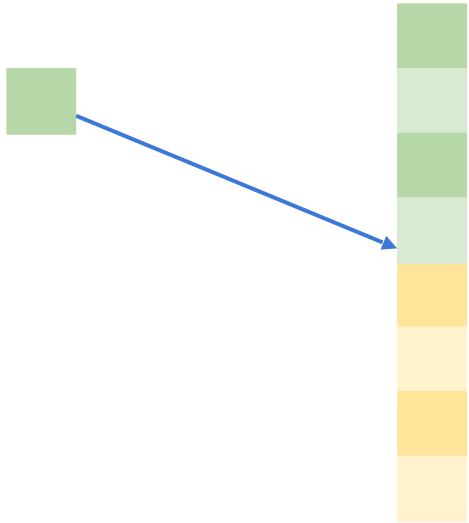


Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

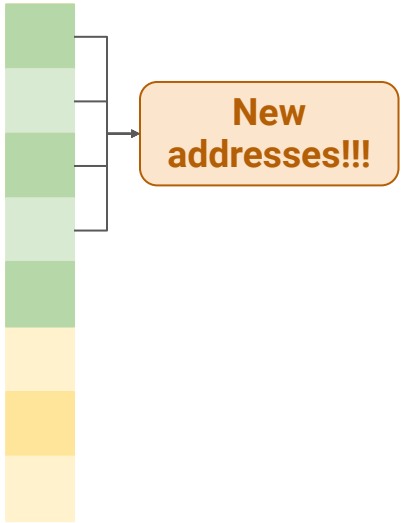


Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.

Iterator Invalidation

- Everytime when memory is reallocated, pointers are invalidated.
- We can reserve up front but what if the size to reserve is unknown?.
  - What happens if the amount of data exceeds the reserved amount?
- Removal of elements that is not the end, shifts elements down to remove fragmentation.
  - Iterators (including the `end()` iterator) and references to the elements at or after the point of the erase are invalidated.



Requirements:

- Stable pointers / references.
- Contiguous in memory.
- Grow on demand.
- No fragmentation.

Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.

Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.

Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.

Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.

Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.

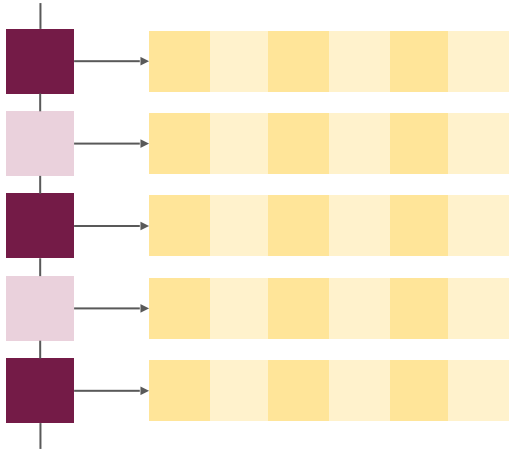
Hypothetical Data Structure

What about std::array?

- Stable pointers / references.
  - No reallocation.
  - Contents do not shift around in memory.
- Contiguous in memory. Cache friendly.

Grow on demand by making a list of std::array.

- Pool of fixed sized chunks.
- When a chunk runs out of space, allocate a new chunk and append it to the pool.



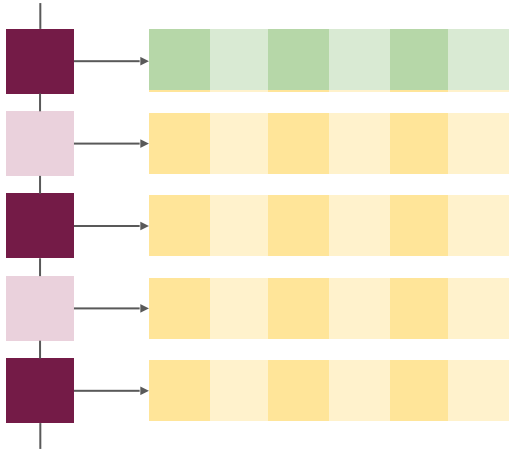
std::deque

With std::deque you get:

- O(1) random access.
- O(1) insert / remove at end / beginning.

With a few drawbacks:

- O(n) on insert and remove.
- No pointer / reference stability.



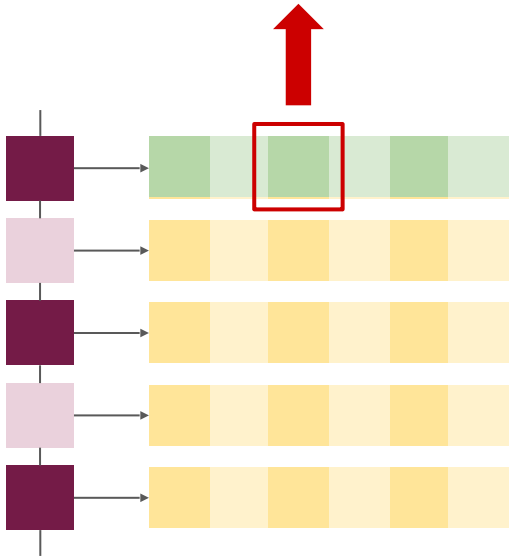
std::deque

With std::deque you get:

- O(1) random access.
- O(1) insert / remove at end / beginning.

With a few drawbacks:

- O(n) on insert and remove.
- No pointer / reference stability.



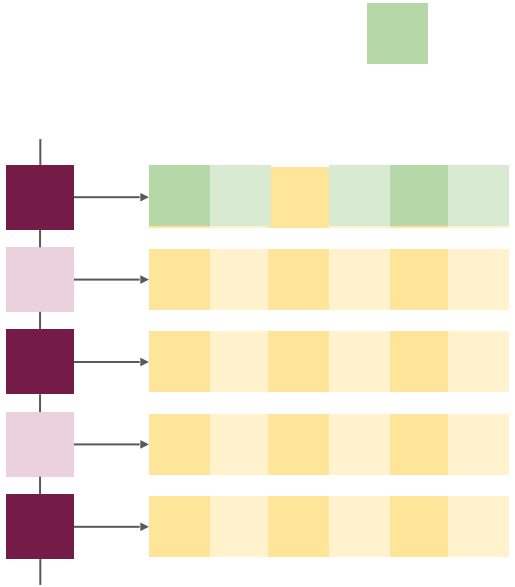
std::deque

With std::deque you get:

- O(1) random access.
- O(1) insert / remove at end / beginning.

With a few drawbacks:

- O(n) on insert and remove.
- No pointer / reference stability.



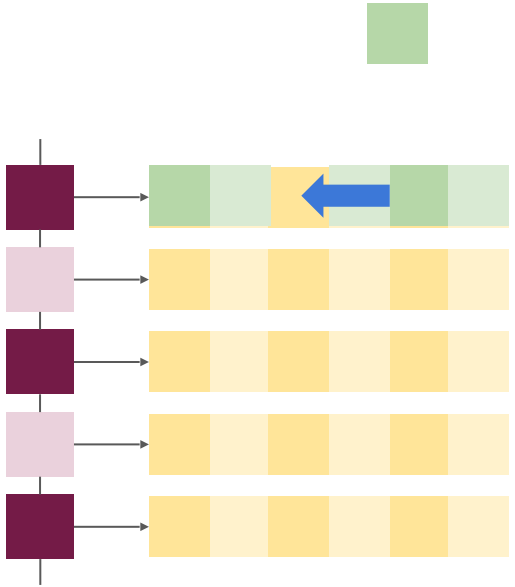
std::deque

With std::deque you get:

- O(1) random access.
- O(1) insert / remove at end / beginning.

With a few drawbacks:

- O(n) on insert and remove.
- No pointer / reference stability.



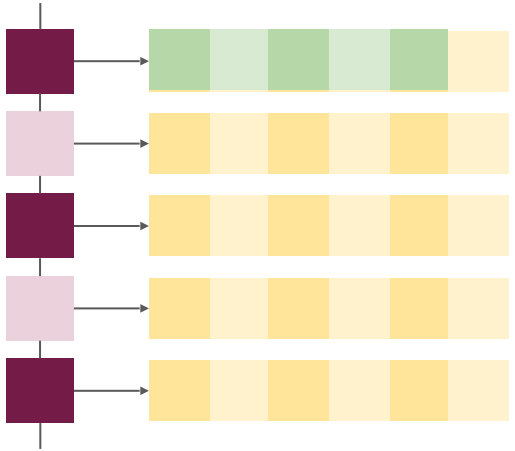
std::deque

With std::deque you get:

- O(1) random access.
- O(1) insert / remove at end / beginning.

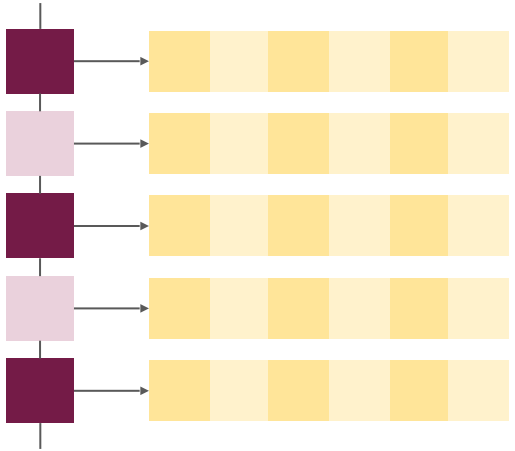
With a few drawbacks:

- O(n) on insert and remove.
- No pointer / reference stability.



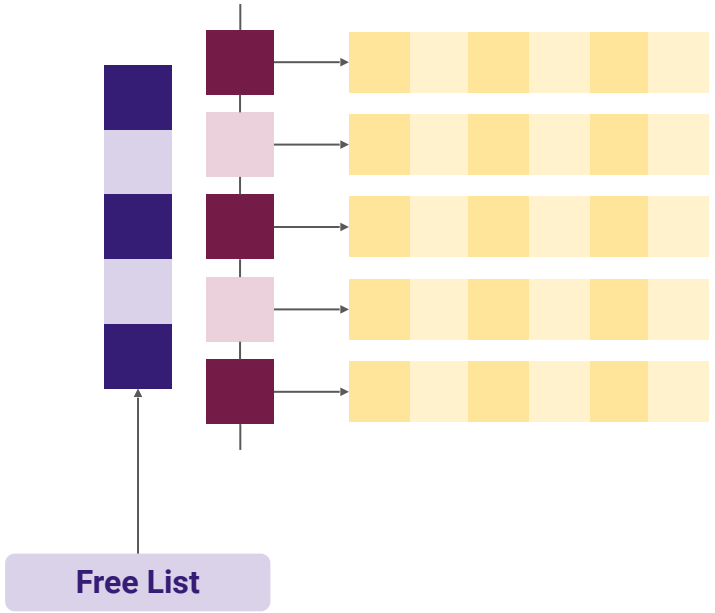
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



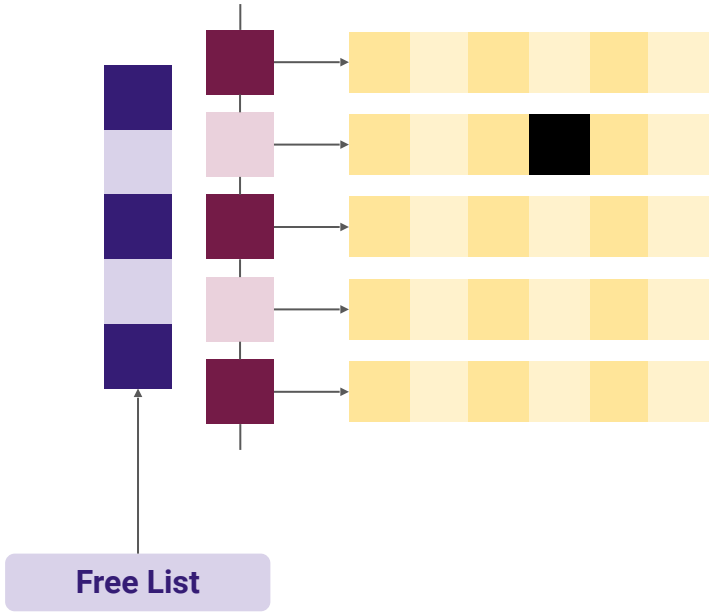
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



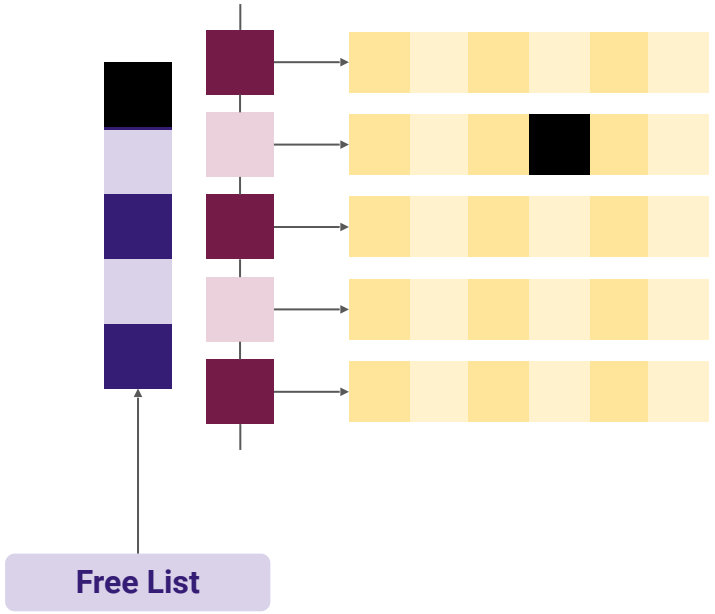
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



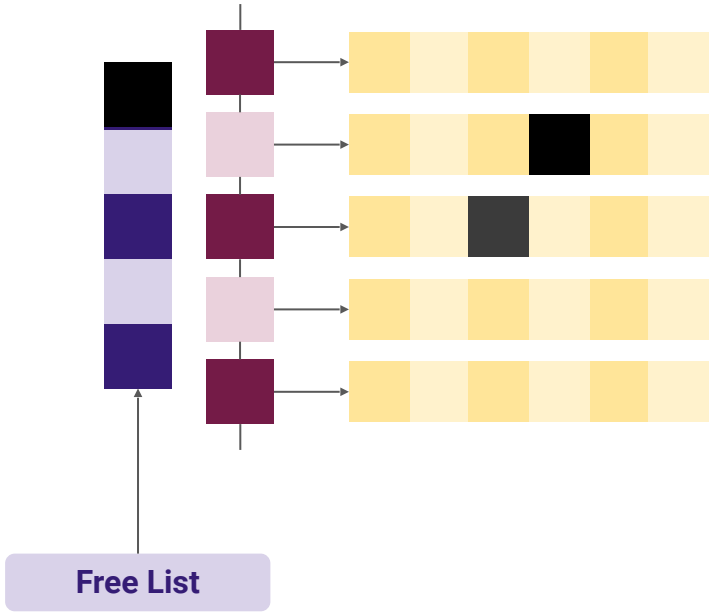
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



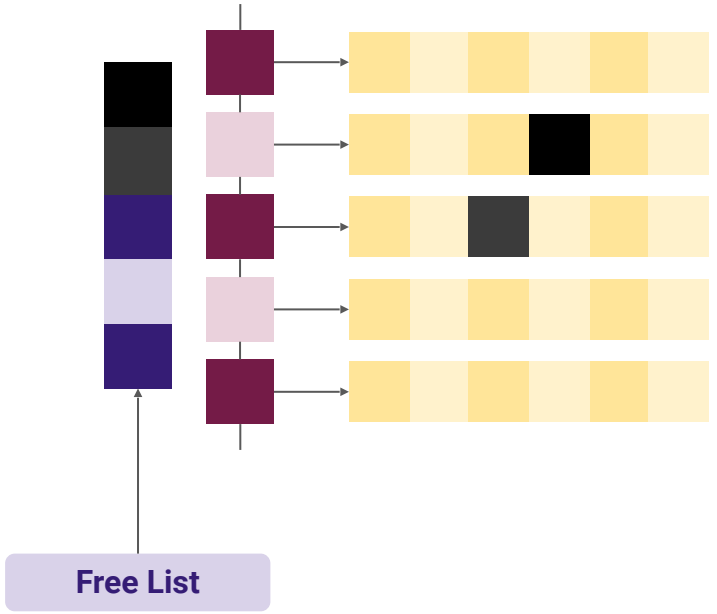
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



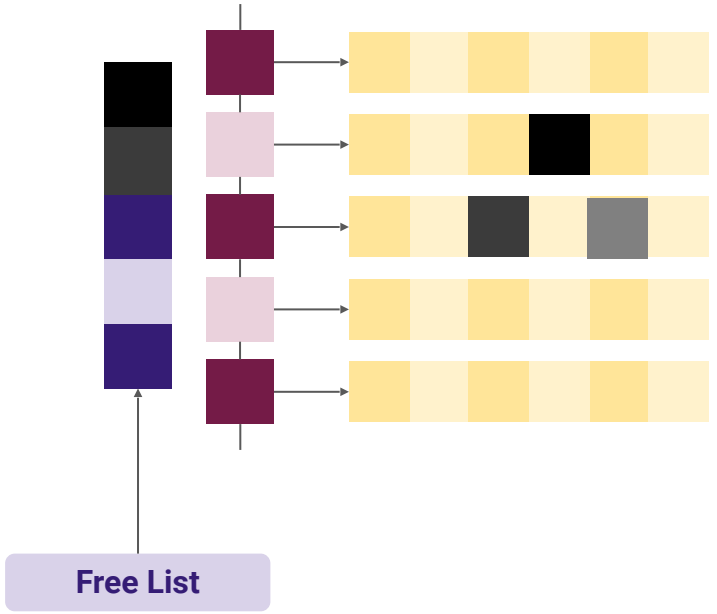
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



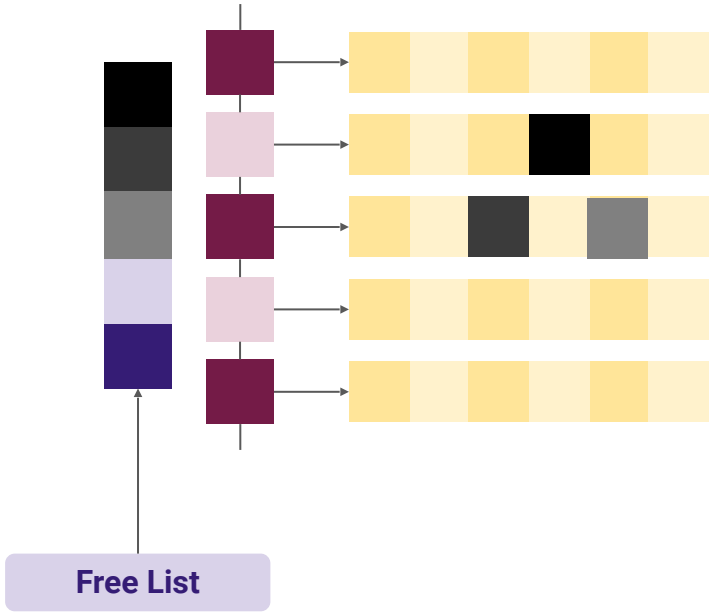
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



Trivial solutions to remaining problems:

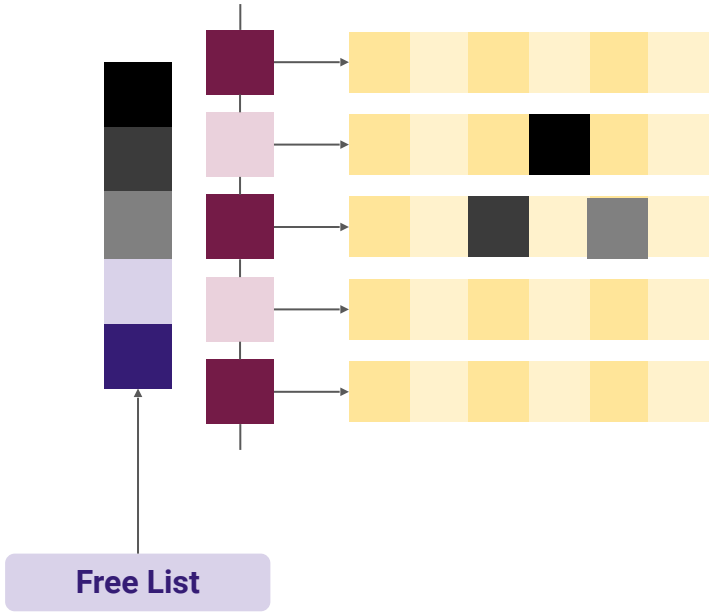
- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.



Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.

Object removal will always be O(1).

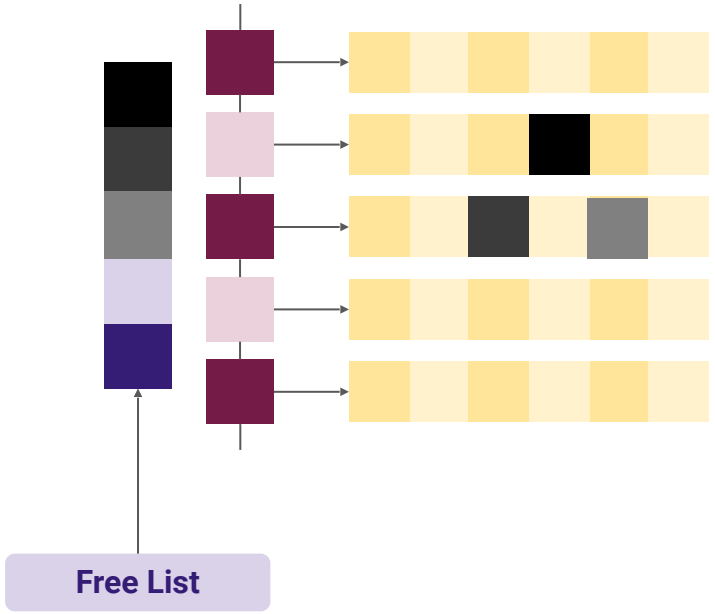


Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.

Object removal will always be O(1).

Stable pointer / references.



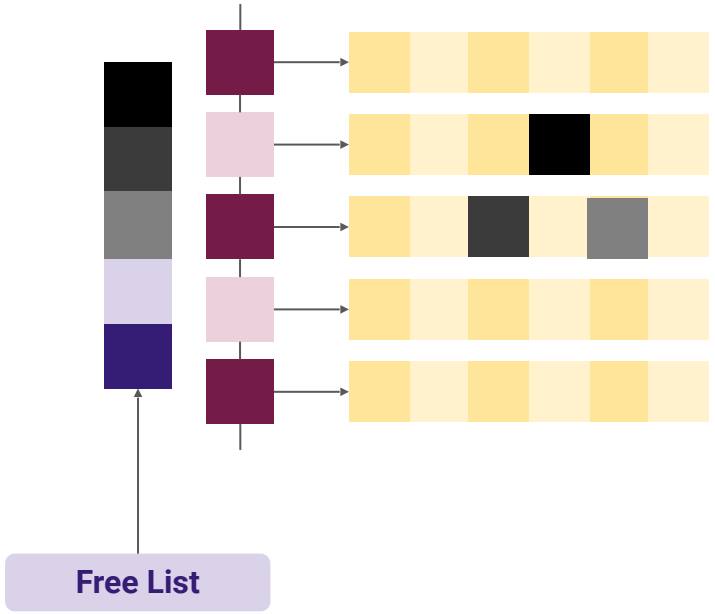
Trivial solutions to remaining problems:

- Introduce a free list into the container.
- Free list can be FIFO or LIFO.
- Don’t shift contents when an element that is not in the ends is removed.
- Add the index and it’s pool into the free list.

Object removal will always be O(1).

Stable pointer / references.

No fragmentation.



Use the ***INDEX***, Luke!

What is an Index?

An index is a structure that contains:

- Page / pool the object is in.
- Offset / index of the object in the chunk.

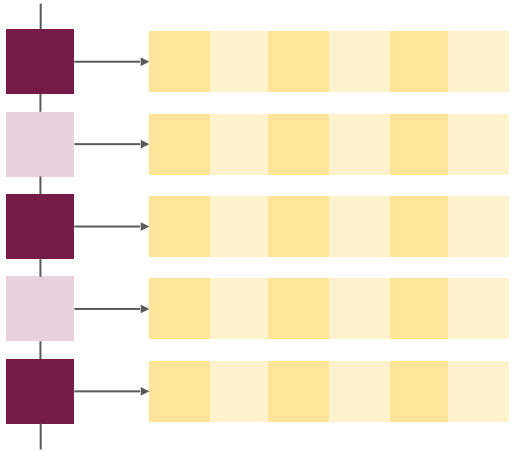
References to the object are done via an index.

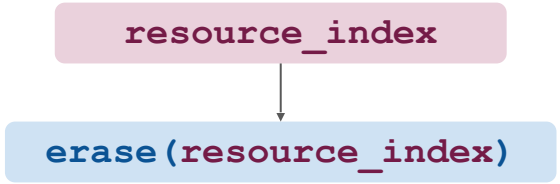
Use the index to access the object either via `operator[]` or the `at()` method.

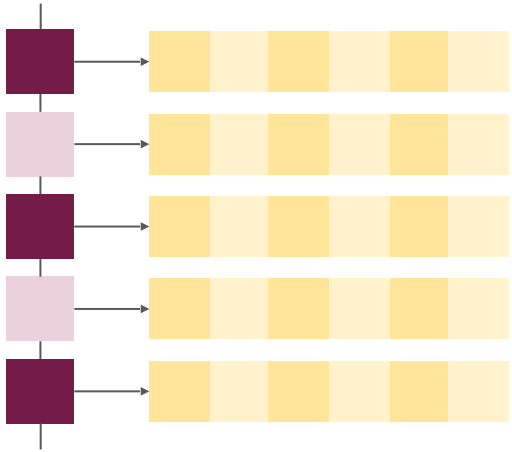
Ideally, encapsulated within an iterator.

Index	
size_t	*m_page_index;*
size_t	*m_offset_in_chunk;*
???	*???*

`resource_index`



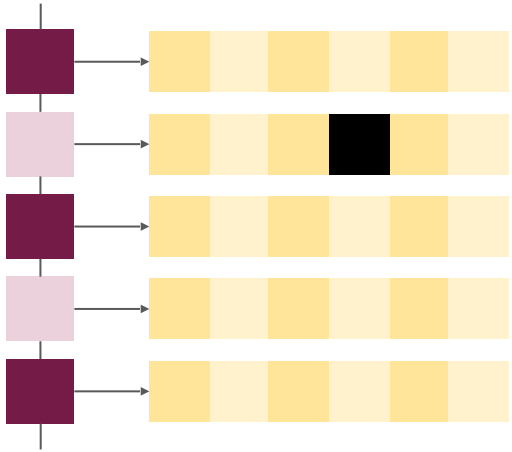

resource_index
erase(resource_index)

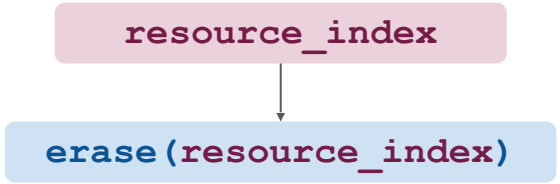


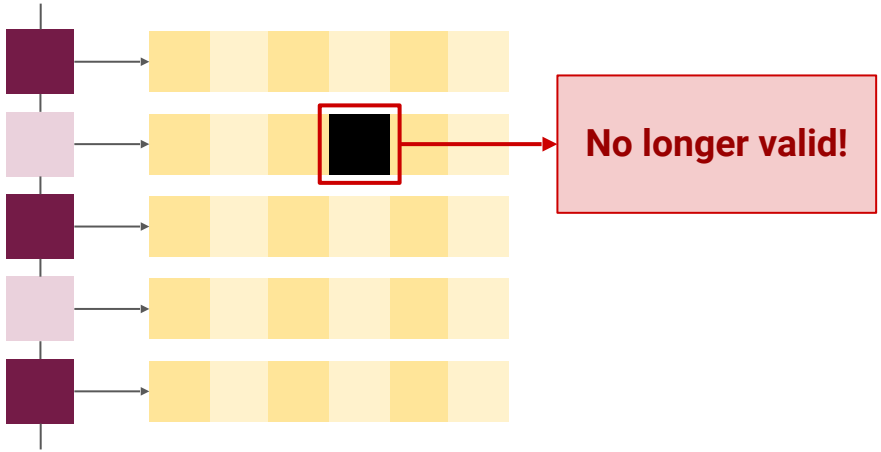
```
resource_index
```

↓

```
erase(resource_index)
```




resource_index
erase(resource_index)


No longer valid!

`resource_index`

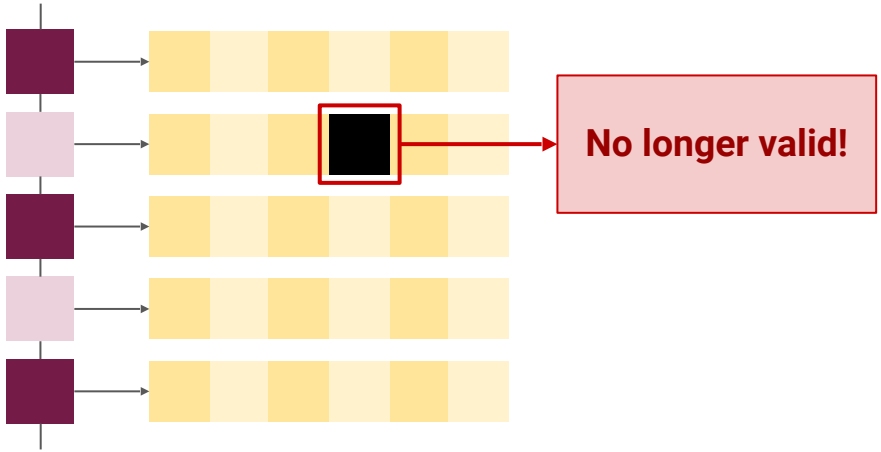
↓

`erase(resource_index)`

`resource_index`

↓

`at(resource_index) / operator[resource_index]`



`resource_index`

↓

`erase(resource_index)`

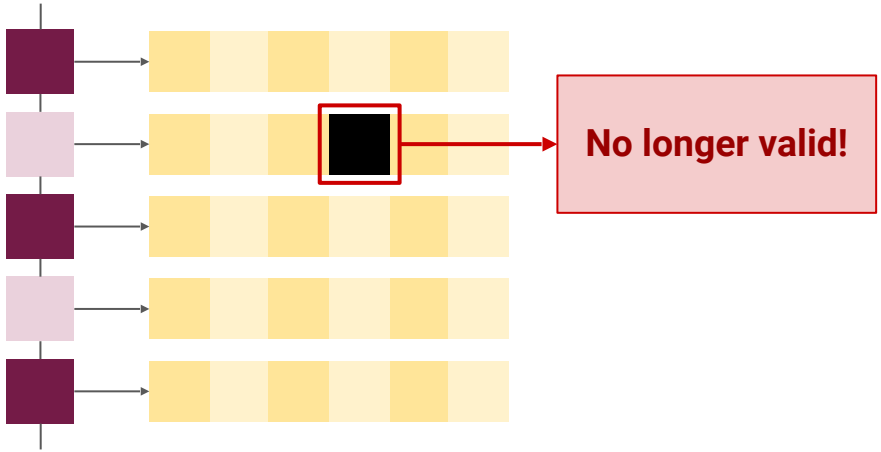
`resource_index`

↓

`at(resource_index) / operator[resource_index]`

↓

Invalid resource!!!



By versioning each slot in a chunk, we can prevent access to destroyed objects.

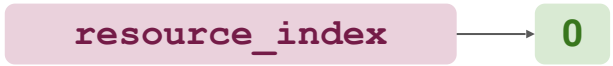
Block to store version metadata.

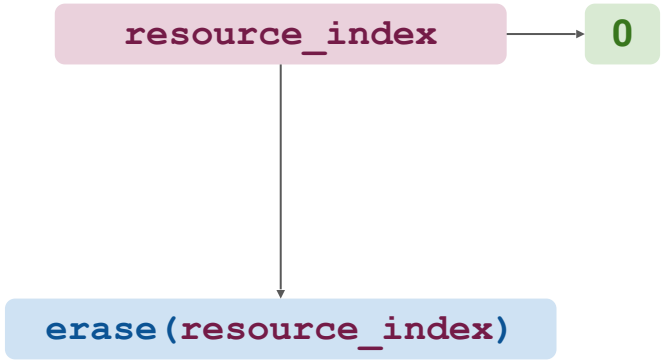
OR

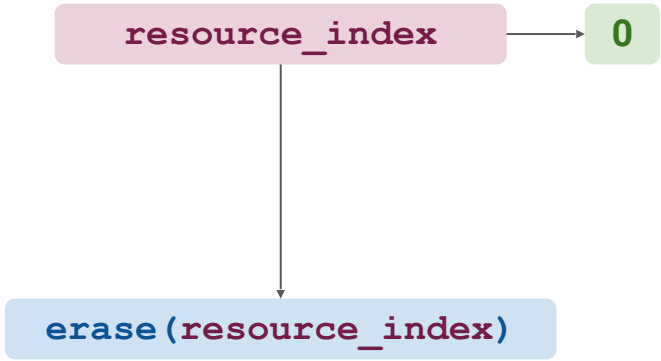
Included as part of the stored object.

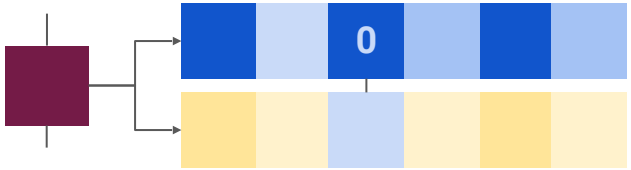
Returned Index should also contain the version of the object that is being referenced.

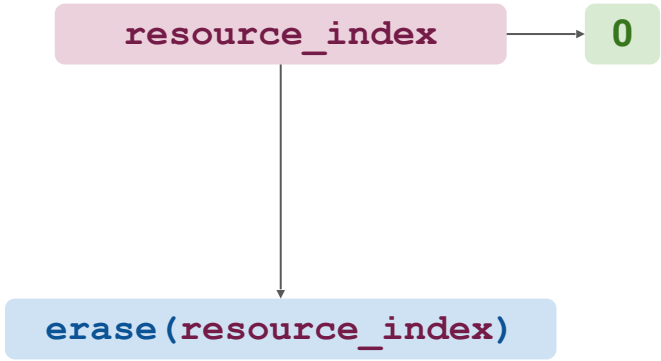
Index	
size_t	*m_page_index;*
size_t	*m_offset_in_chunk;*
size_t	*m_version;*

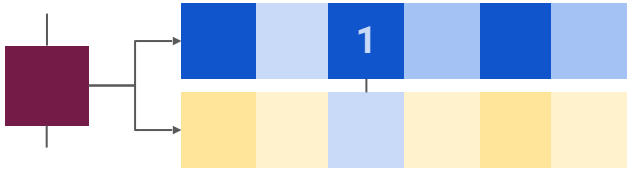

resource_index
0


resource_index
0
erase(resource_index)


resource_index
0
erase(resource_index)

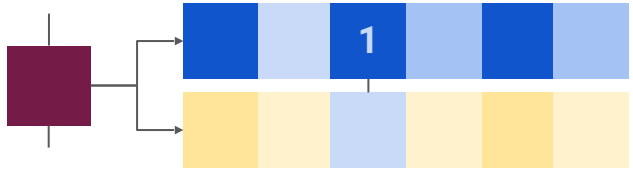

0


resource_index
0
erase(resource_index)


1

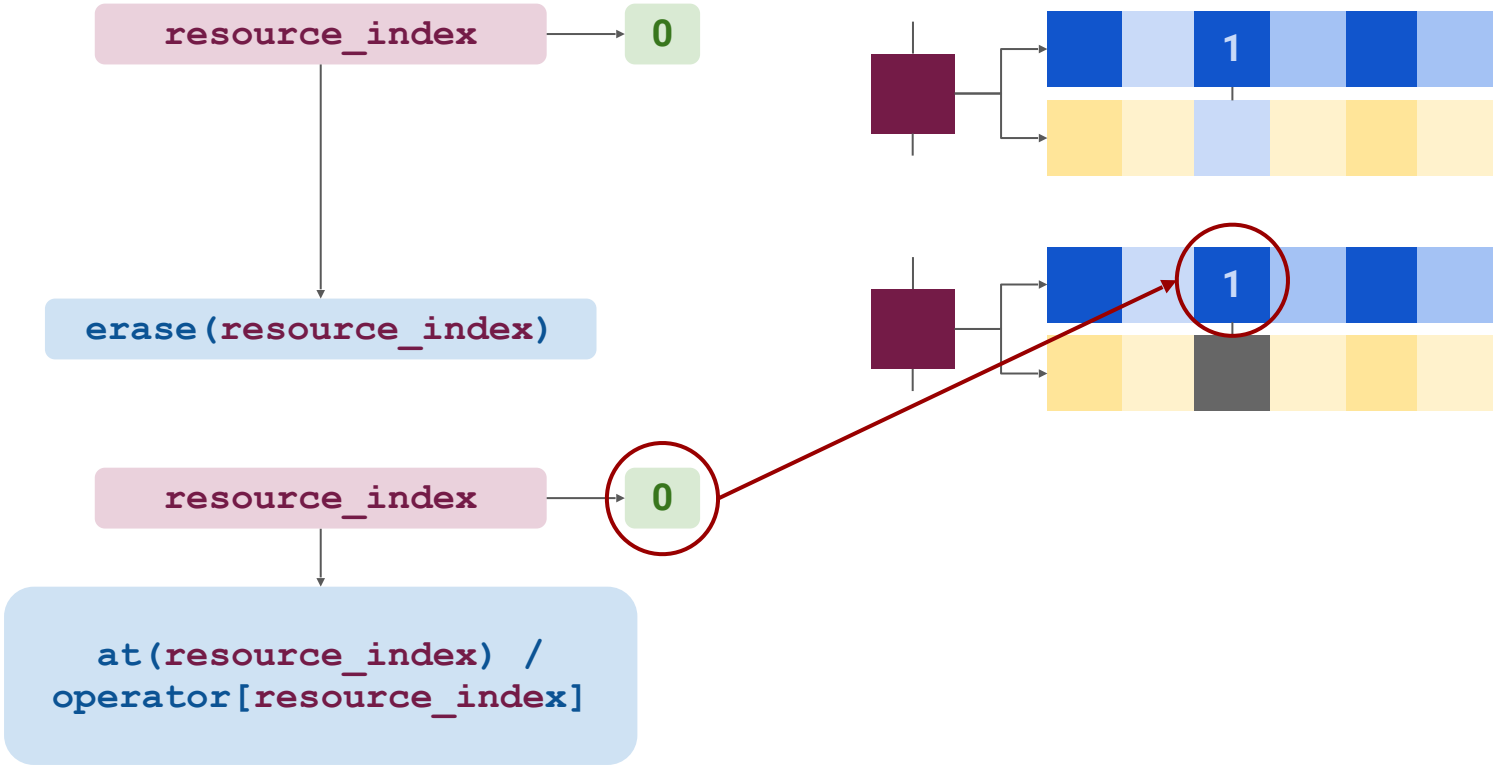
resource_index → 0

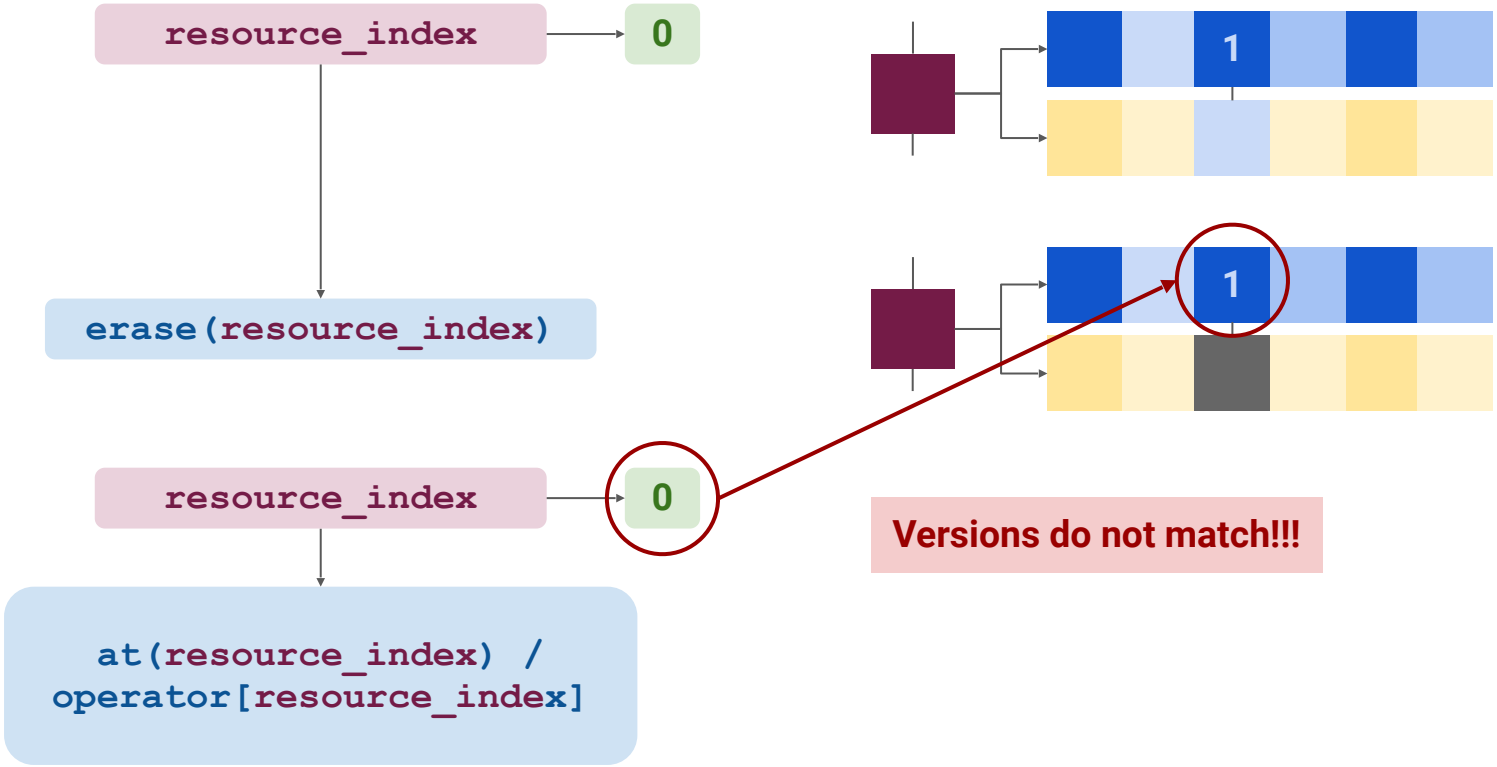
erase(resource_index)

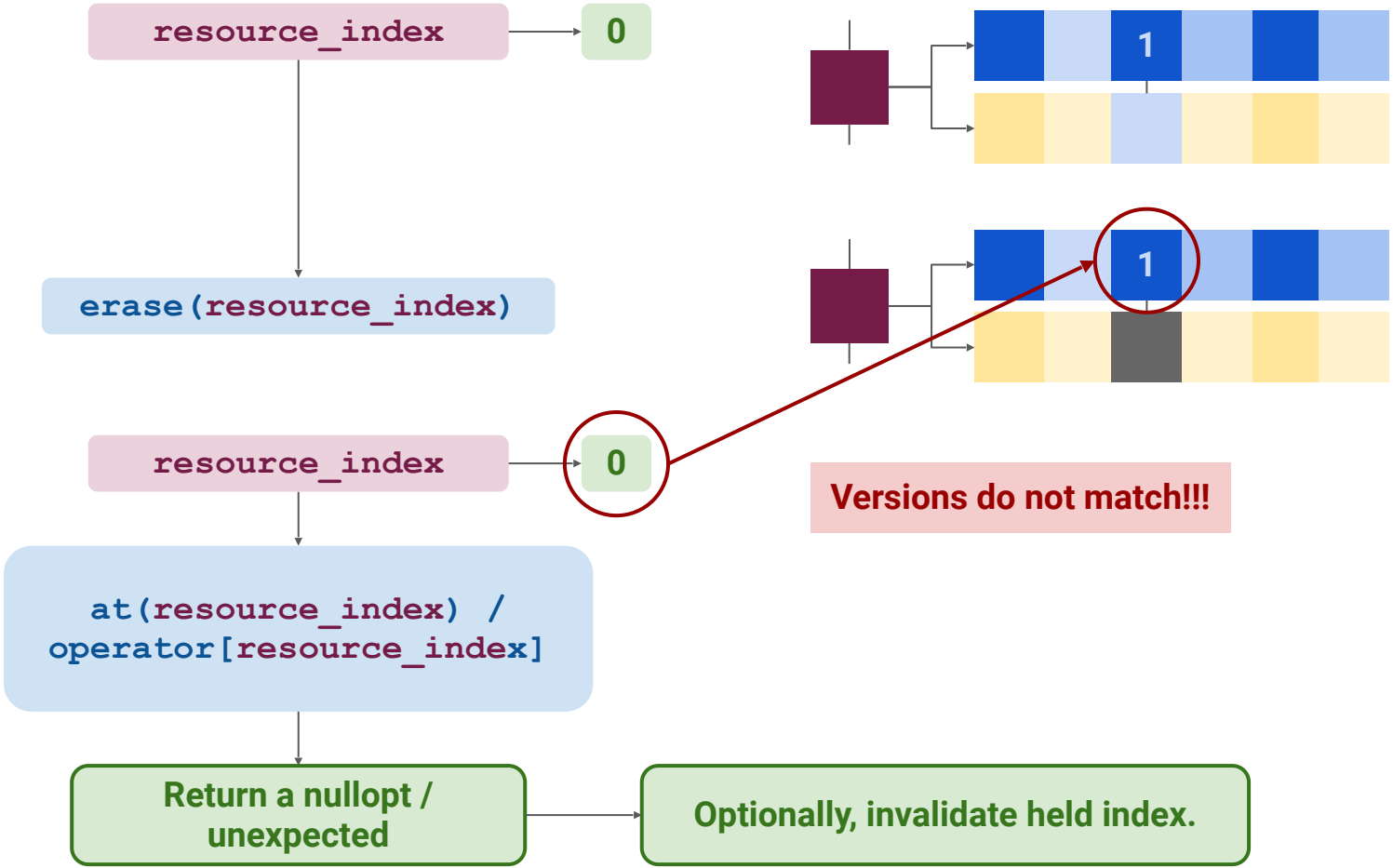


resource_index → 0

at(resource_index) /
operator[resource_index]


resource_index
0
erase(resource_index)
resource_index
0
at(resource_index) /
operator[resource_index]
1
1


resource_index
0
erase(resource_index)
resource_index
0
at(resource_index) /
operator[resource_index]
1
1
Versions do not match!!!


resource_index
0
erase(resource_index)
1
1
resource_index
0
Versions do not match!!!
at(resource_index) /
operator[resource_index]
Return a nullopt /
unexpected
Optionally, invalidate held index.

There is a variant of this data structure being proposed.

plf::colony

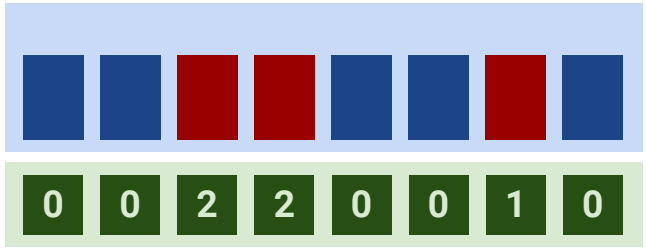
Authored by Matthew Bentley. The same idea with some differences:

- Newly allocated chunks is always twice as large.
- Uses a skipfield to skip iteration on empty elements.
- Freelist is embedded into freed slots in chunks.
- No versioning for each slots in a chunk.

Has all the benefits of the previously discussed container:

- Pointer / iterator stability.
- Memory block reuse.
- O(1) on insert amortized.
- O(1) on erase amortized (regardless of location).

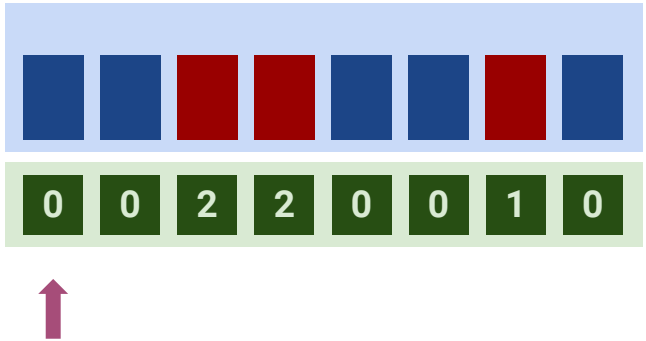
Skipfields are used during iteration to skip empty blocks without any branching.



When iterating through the container, the iterator queries into the skipfield metadata.

- If the skipfield metadata for that block is 0, it simply iterates to the next block.
- If the skipfield metadata has some value other than 0, it increments the iterator with the skip count.

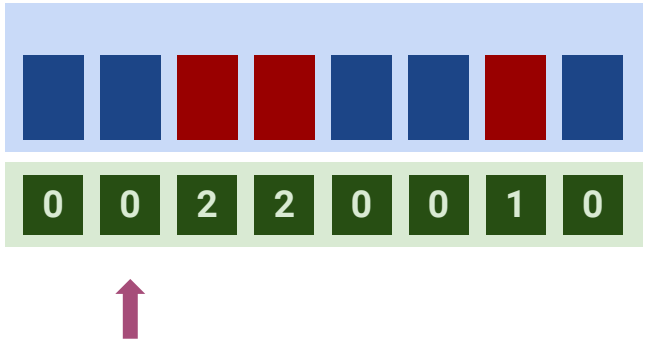
Skipfields are used during iteration to skip empty blocks without any branching.



When iterating through the container, the iterator queries into the skipfield metadata.

- If the skipfield metadata for that block is 0, it simply iterates to the next block.
- If the skipfield metadata has some value other than 0, it increments the iterator with the skip count.

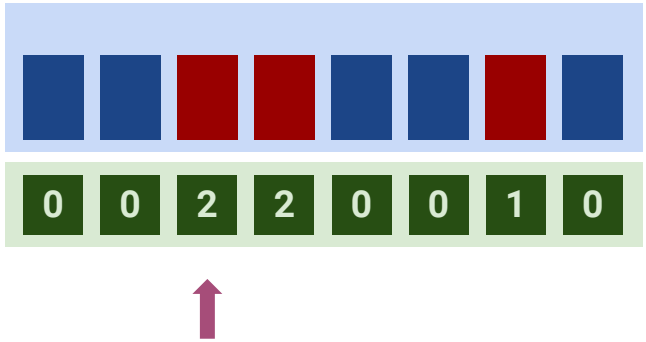
Skipfields are used during iteration to skip empty blocks without any branching.



When iterating through the container, the iterator queries into the skipfield metadata.

- If the skipfield metadata for that block is 0, it simply iterates to the next block.
- If the skipfield metadata has some value other than 0, it increments the iterator with the skip count.

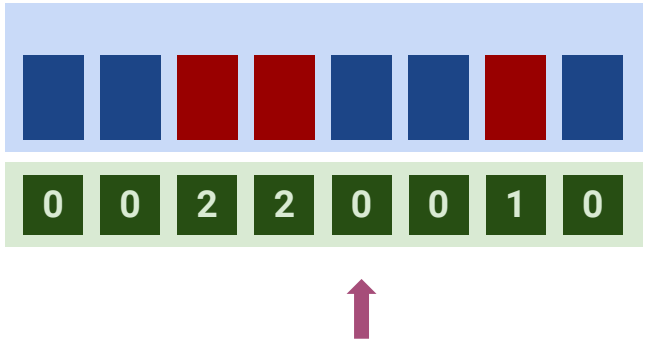
Skipfields are used during iteration to skip empty blocks without any branching.



When iterating through the container, the iterator queries into the skipfield metadata.

- If the skipfield metadata for that block is 0, it simply iterates to the next block.
- If the skipfield metadata has some value other than 0, it increments the iterator with the skip count.

Skipfields are used during iteration to skip empty blocks without any branching.

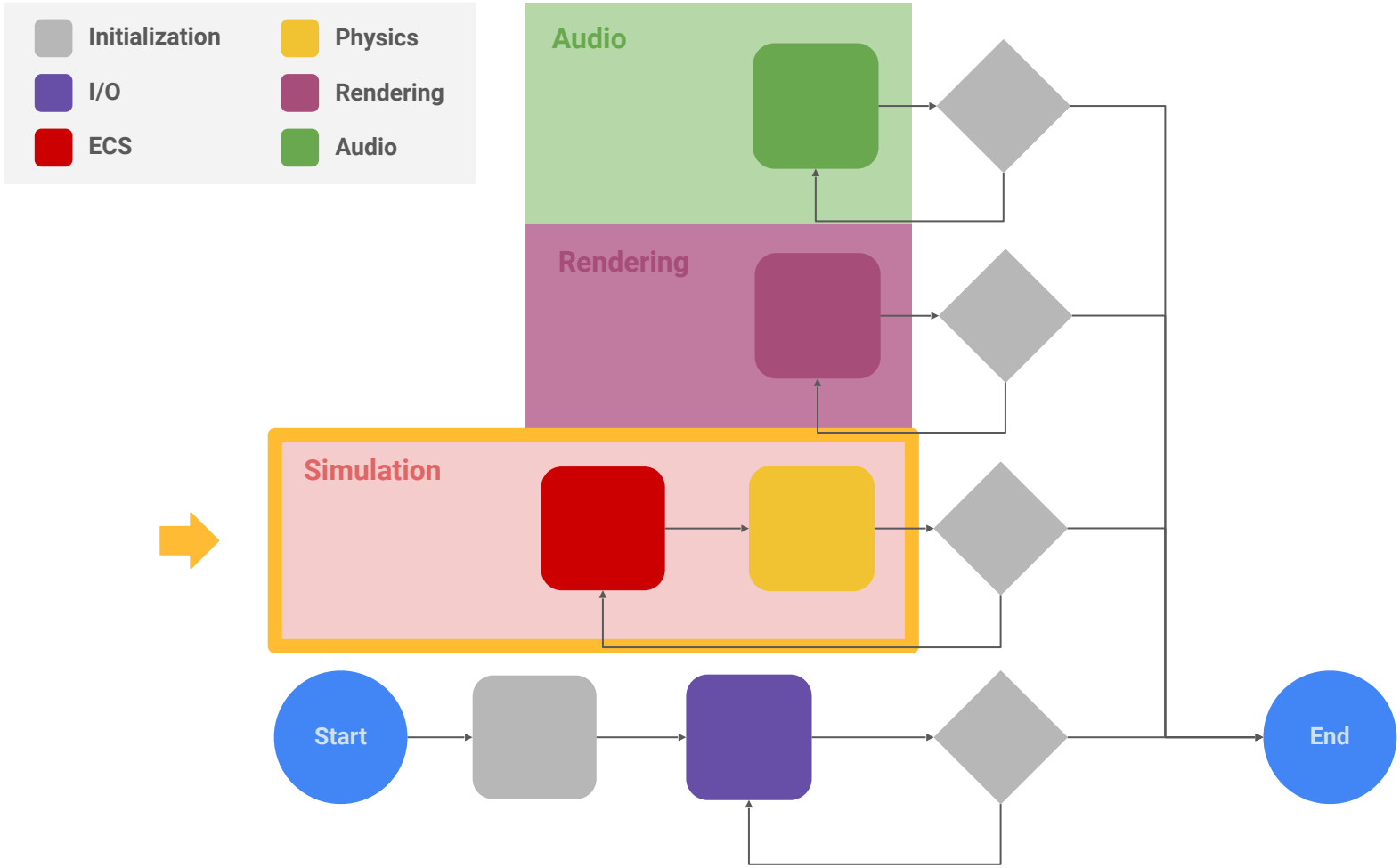


When iterating through the container, the iterator queries into the skipfield metadata.

- If the skipfield metadata for that block is 0, it simply iterates to the next block.
- If the skipfield metadata has some value other than 0, it increments the iterator with the skip count.

P0447R - std::hive

- Colonies, performance and why you should care - Matthew Bentley
- How to: Colony - Matthew Bentley


Initialization
Physics
I/O
Rendering
ECS
Audio
Audio
Rendering
Simulation
Start
End

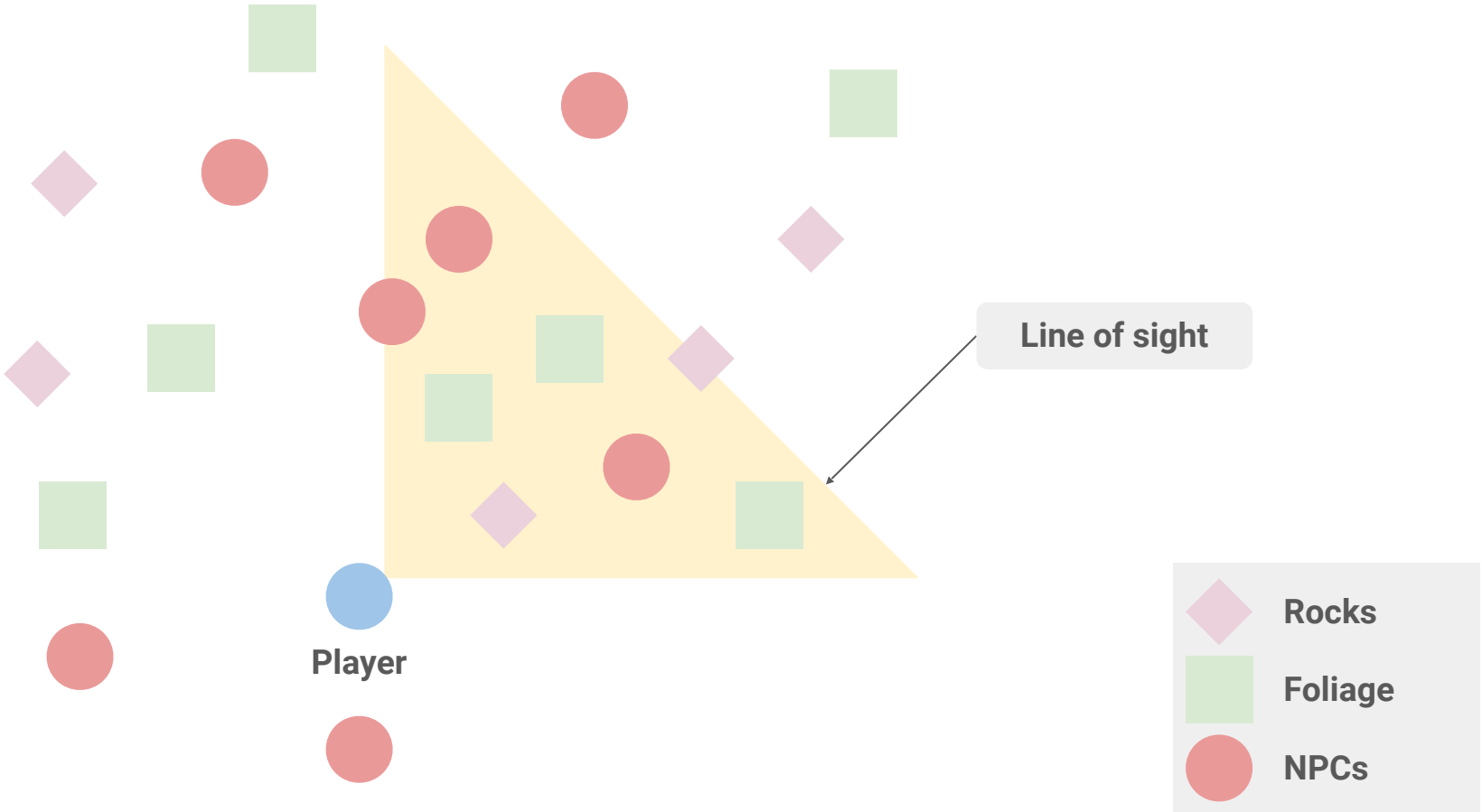
Simulation

Imagine our hypothetical game is 3D with a vast world.

Multiple systems ticking,

- Entity Component Systems (ECS)
- Artificial Intelligence
- Animation
- Physics
- Gameplay specific

We’ve simulated everything and all that’s left is to send them to the renderer.

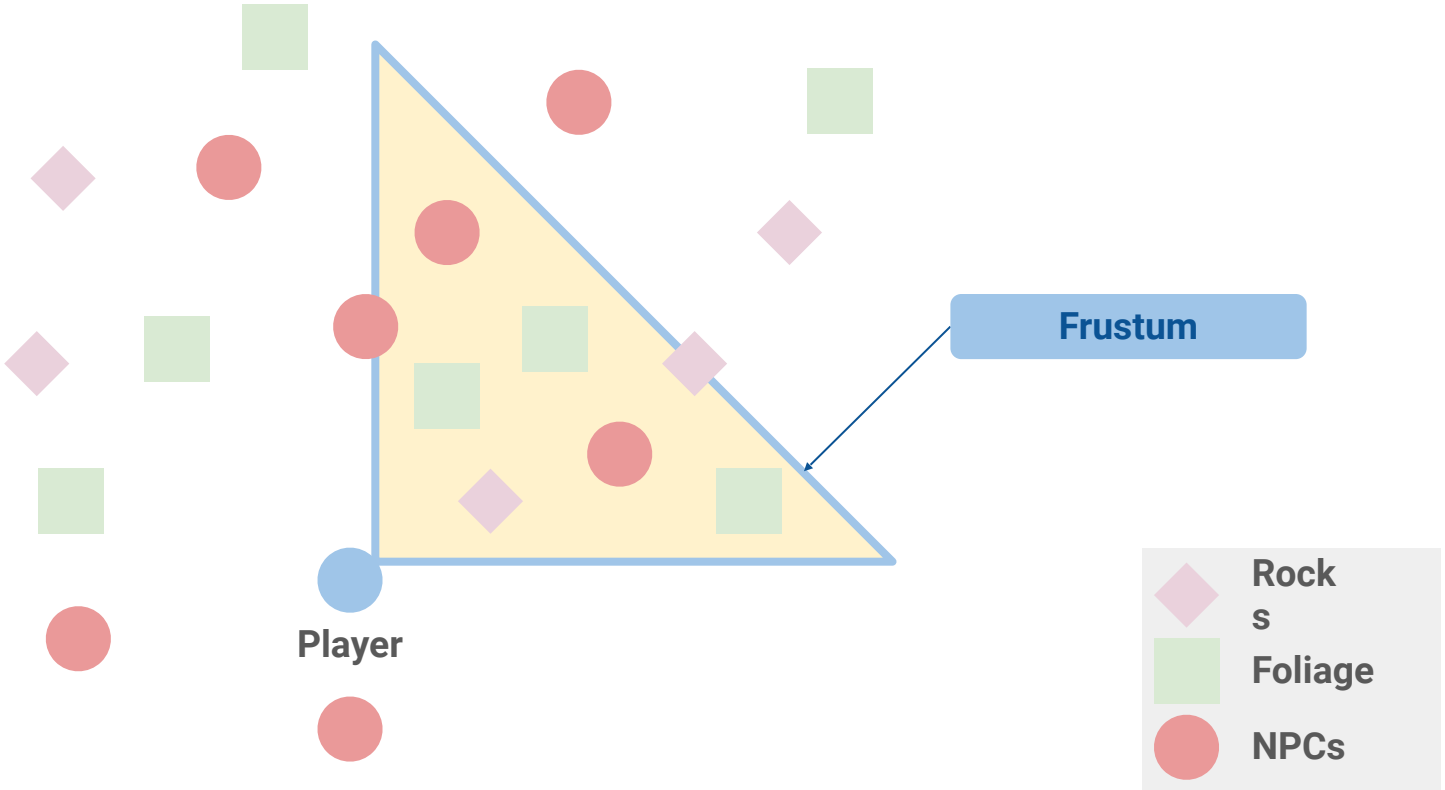

Line of sight
Player
Rocks
Foliage
NPCs

The Problem

We can’t send everything to the GPU.

- Memory on the GPU is limited.
  - RTX 4090 -> 24GB
  - RTX 2070 -> 8GB
  - Current gen consoles (PS5 & XBSX) -> 16GB Unified memory, shared between CPU and GPU.
- Models can have high vertex counts.
- Textures can be high resolution.

But the biggest problem, **we’re rendering things that we don’t see in-game.**



Render only things that ***exist in the frustum***.

This technique is called ***FRUSTUM CULLING*.**

Frustum Culling

Usually done by wrapping objects in a bounding volume (sphere, AABB) and fancy math is done to determine if it intersects the frustum.

Naive approach, linearly search through each game entity to determine if it exist inside the frustum.

Does not scale well, O(N) time complexity.

We need a data structure that represents the map in a way for us to search faster.

Quadtrees & Octrees

Used to partition space in the game world by recursively subdividing.

Quadtree	Octree
• 2D • Recursively subdivided into 4 spaces	• 3D • Recursively subdivided into 8 spaces.

Quadtrees & Octrees

Used to partition space in the game world by recursively subdividing.

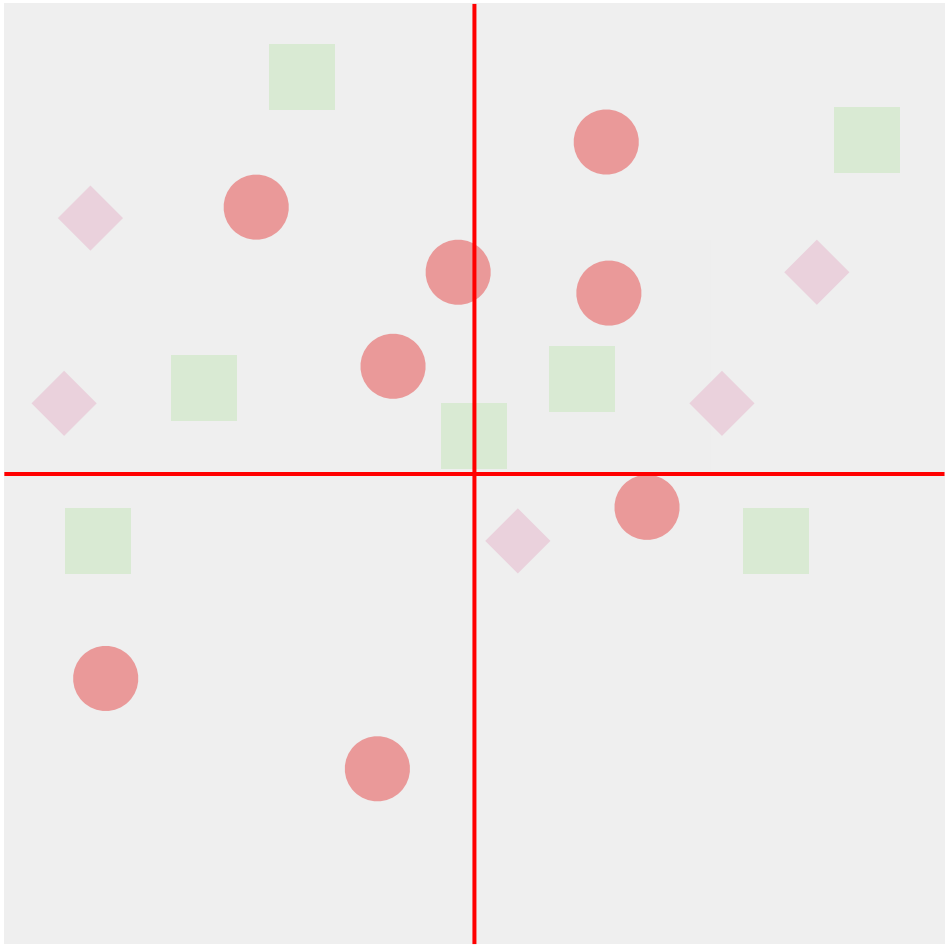
Quadtree	Octree
• 2D • Recursively subdivided into 4 spaces	• 3D • Recursively subdivided into 8 spaces.

How to construct a quadtree for a level?

- Our level starts of as the root node.
- A node in the tree can represent a game entity or a quadrant itself.
- Any time a quadrant has more than 4 entities, the node subdivides into 4 more nodes.
- If an entity intersects with multiple quadrants, include them as a child node in each quadrant.

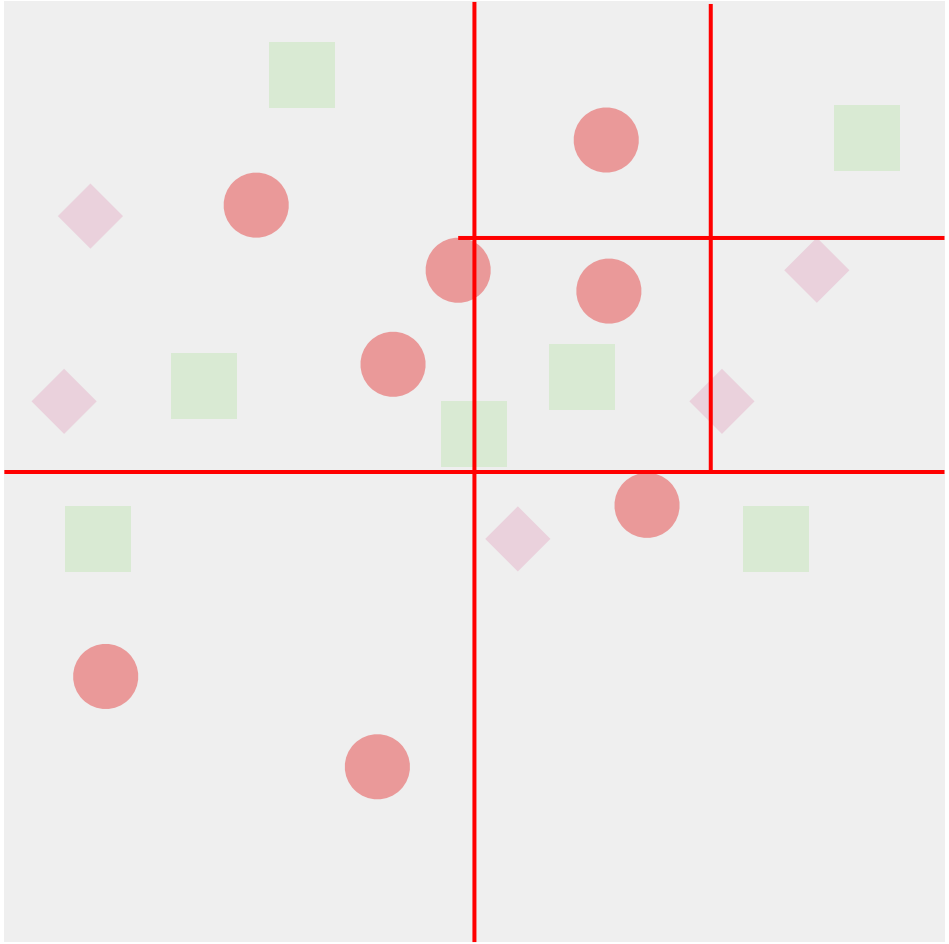
How to construct a quadtree for a level?

- Our level starts of as the root node.
- A node in the tree can represent a game entity or a quadrant itself.
- Any time a quadrant has more than 4 entities, the node subdivides into 4 more nodes.
- If an entity intersects with multiple quadrants, include them as a child node in each quadrant.



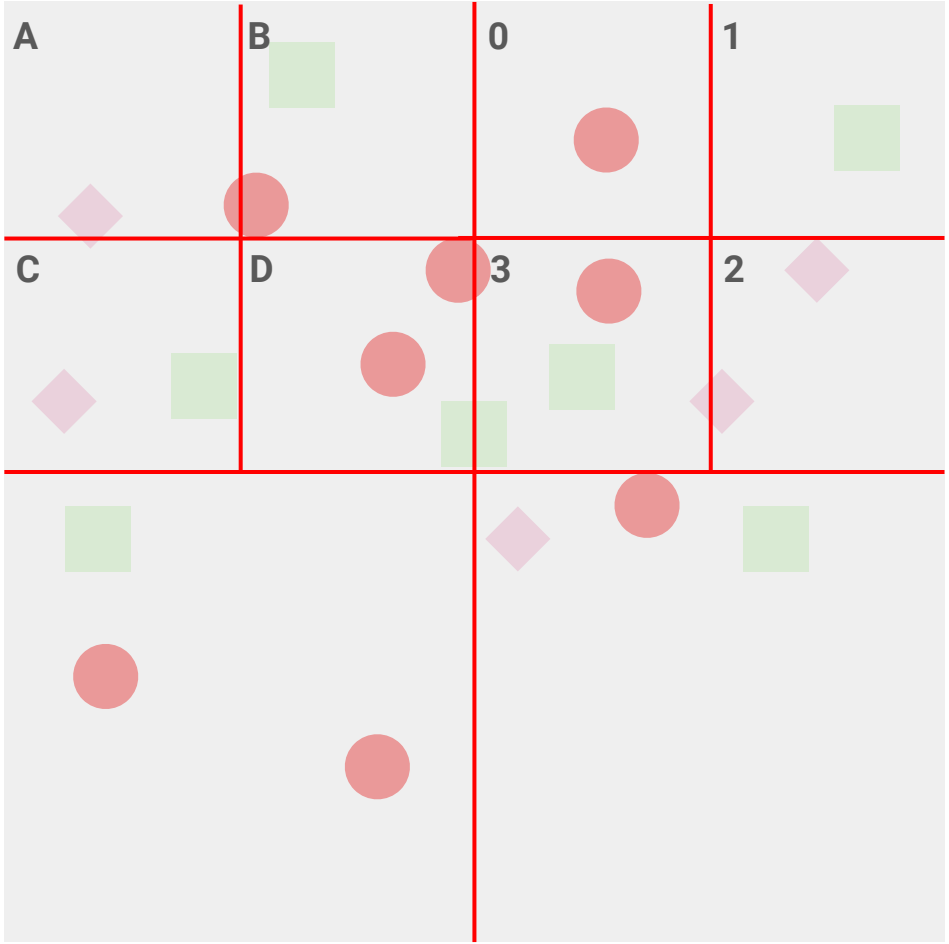
How to construct a quadtree for a level?

- Our level starts of as the root node.
- A node in the tree can represent a game entity or a quadrant itself.
- Any time a quadrant has more than 4 entities, the node subdivides into 4 more nodes.
- If an entity intersects with multiple quadrants, include them as a child node in each quadrant.



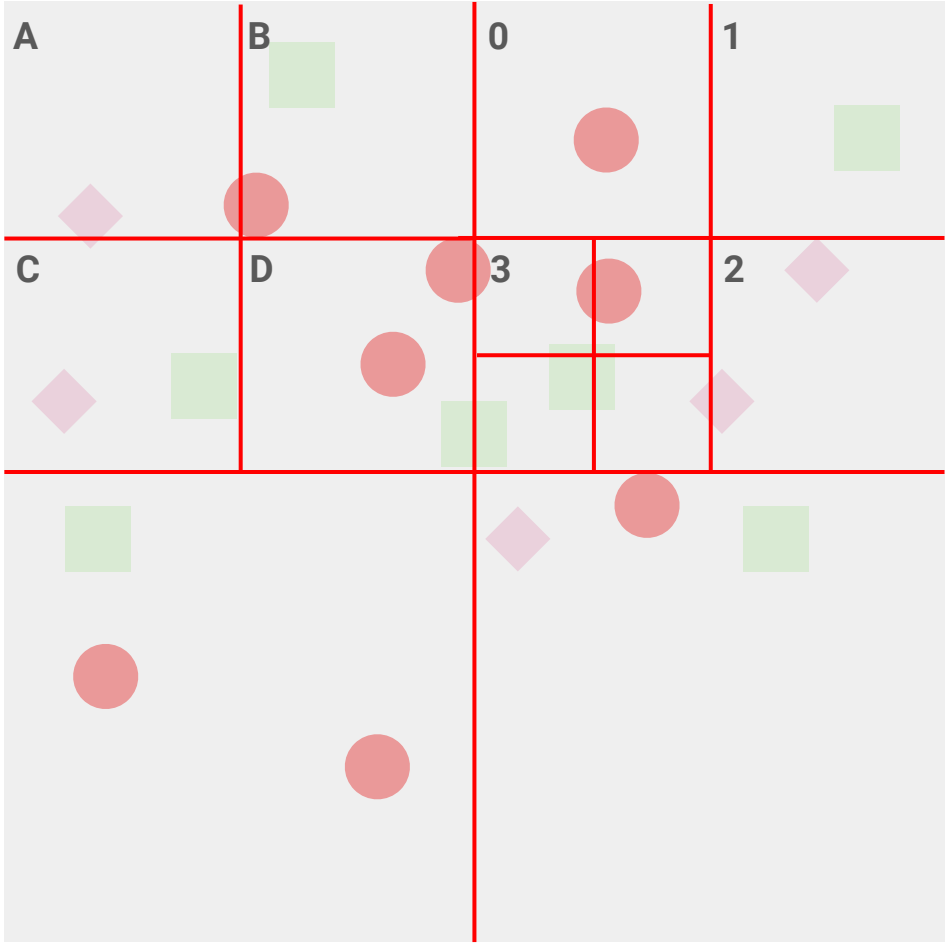
How to construct a quadtree for a level?

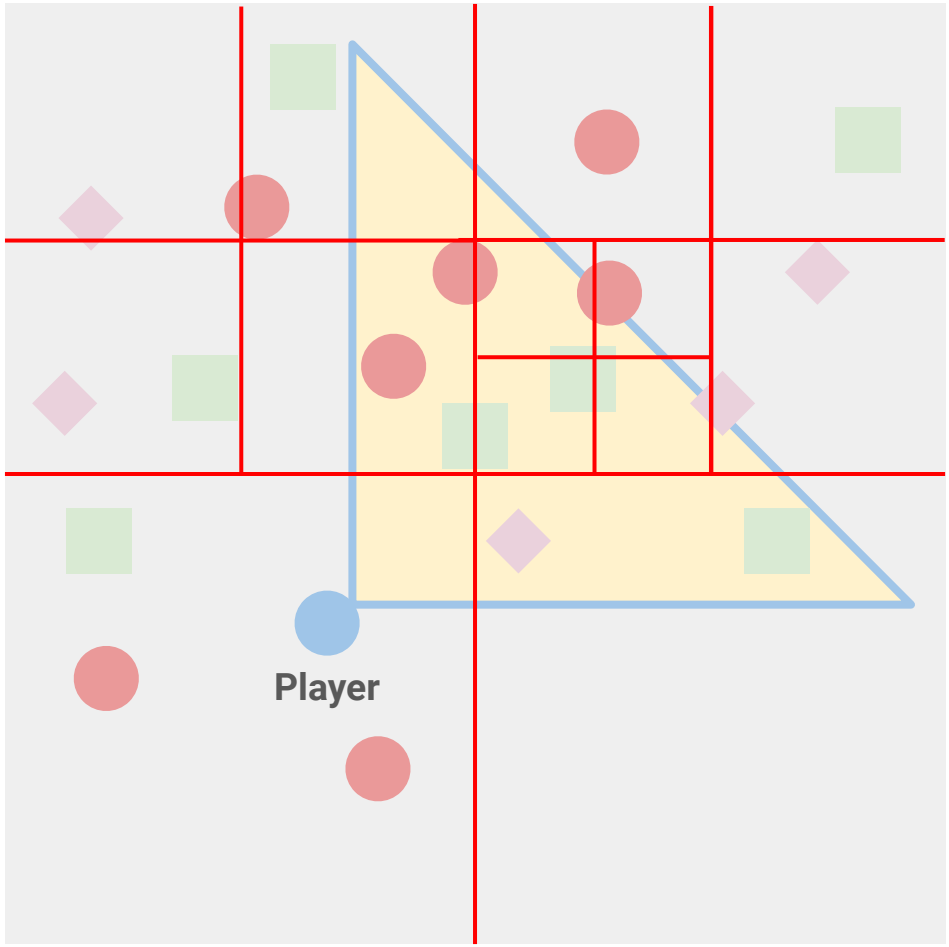
- Our level starts of as the root node.
- A node in the tree can represent a game entity or a quadrant itself.
- Any time a quadrant has more than 4 entities, the node subdivides into 4 more nodes.
- If an entity intersects with multiple quadrants, include them as a child node in each quadrant.

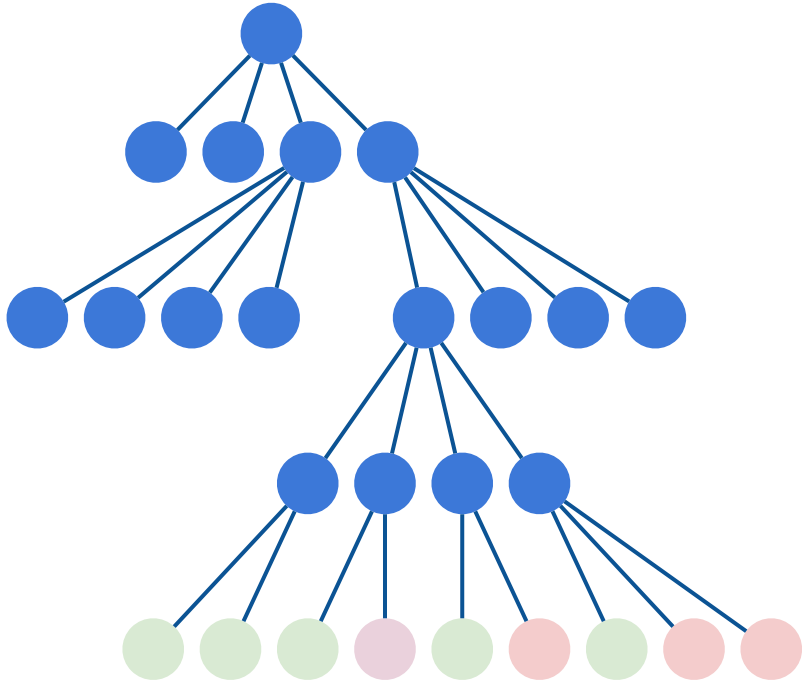


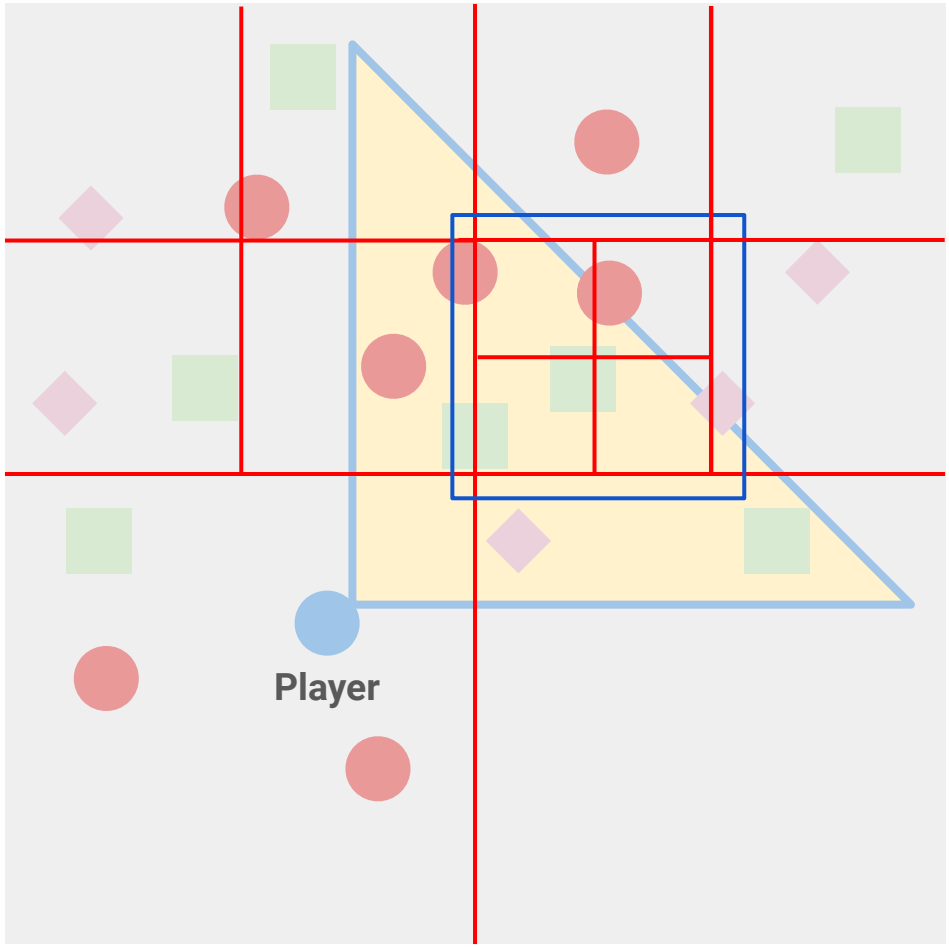
How to construct a quadtree for a level?

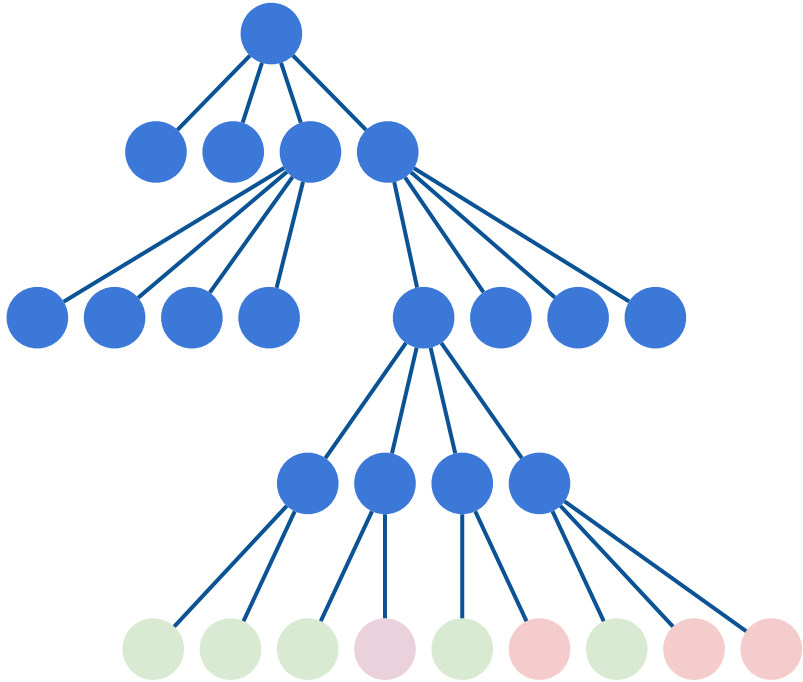
- Our level starts of as the root node.
- A node in the tree can represent a game entity or a quadrant itself.
- Any time a quadrant has more than 4 entities, the node subdivides into 4 more nodes.
- If an entity intersects with multiple quadrants, include them as a child node in each quadrant.

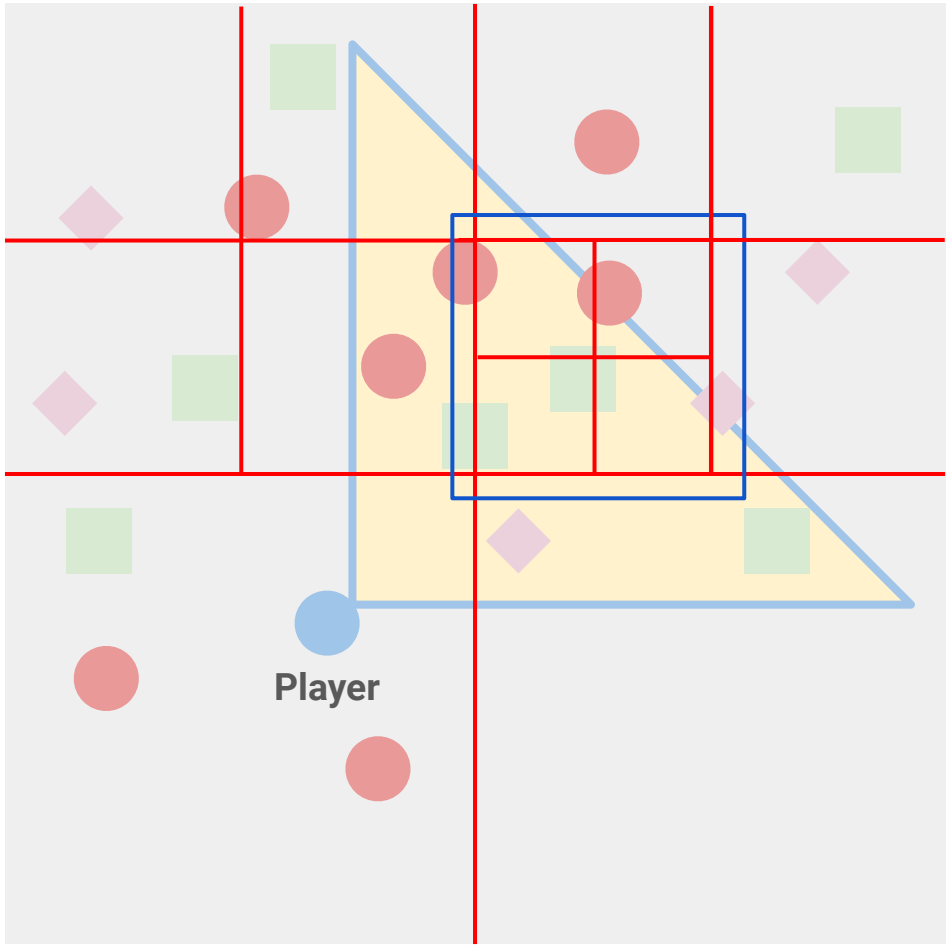


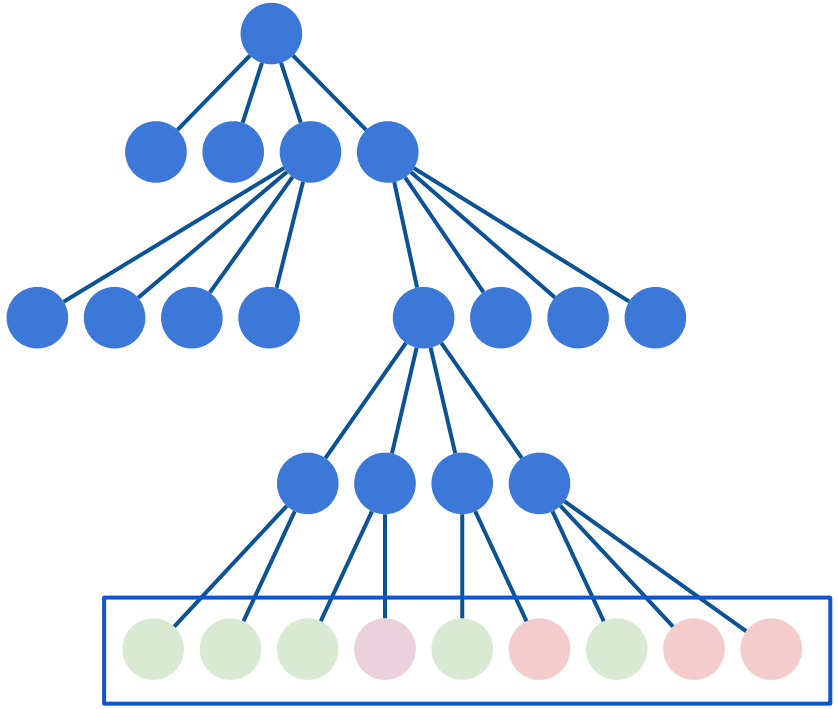

Player



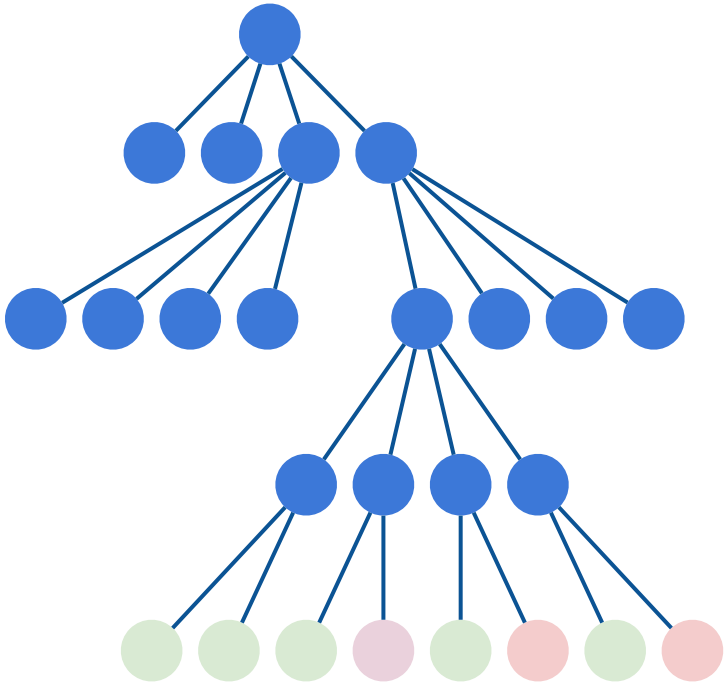

Player




Player



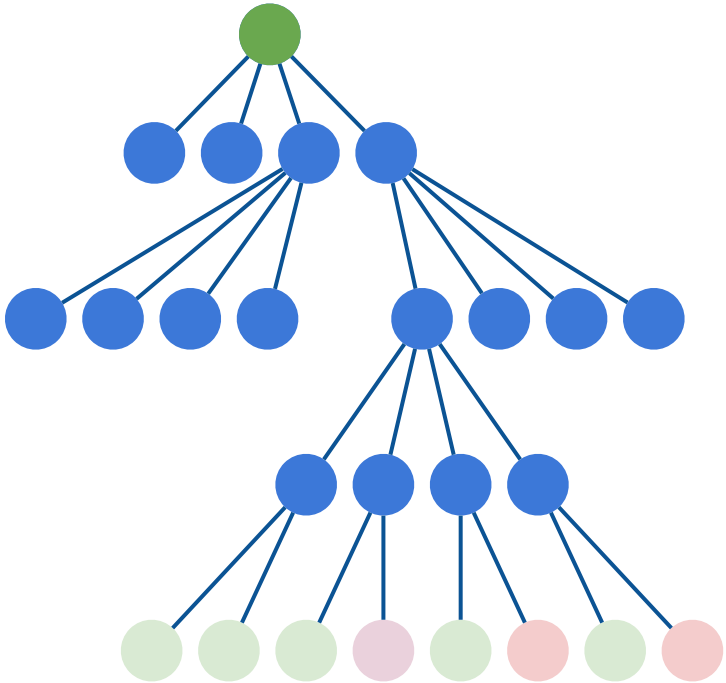
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

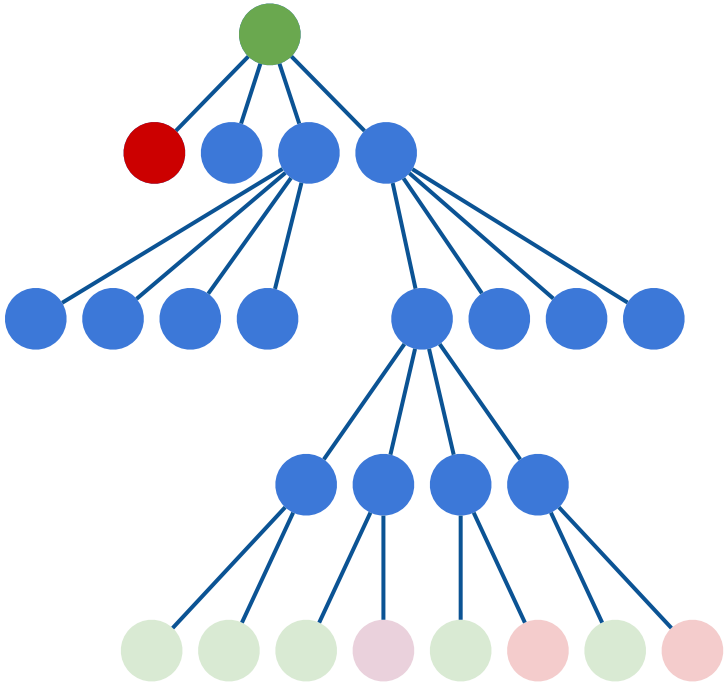
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

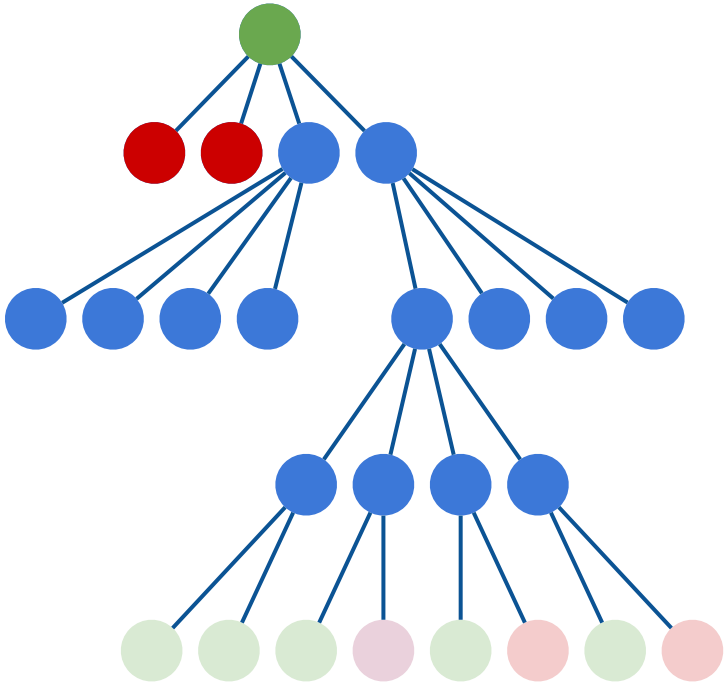
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

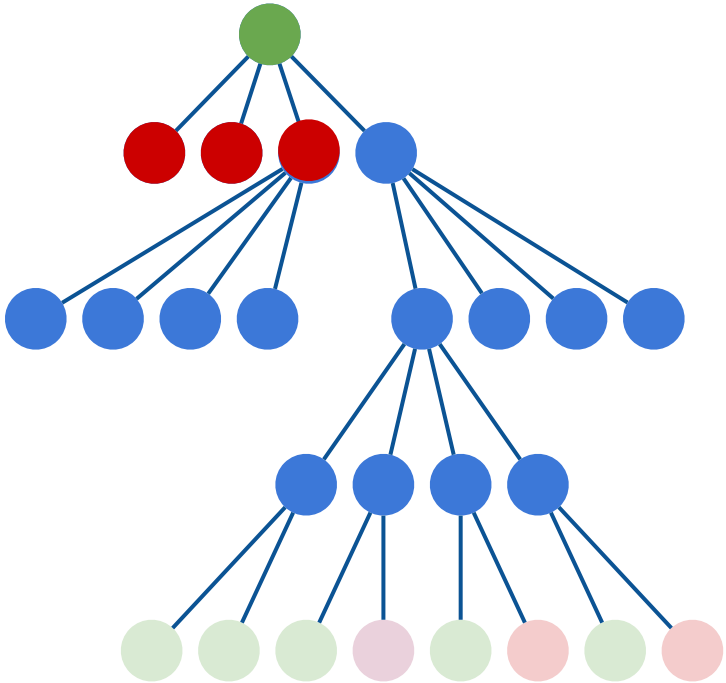
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

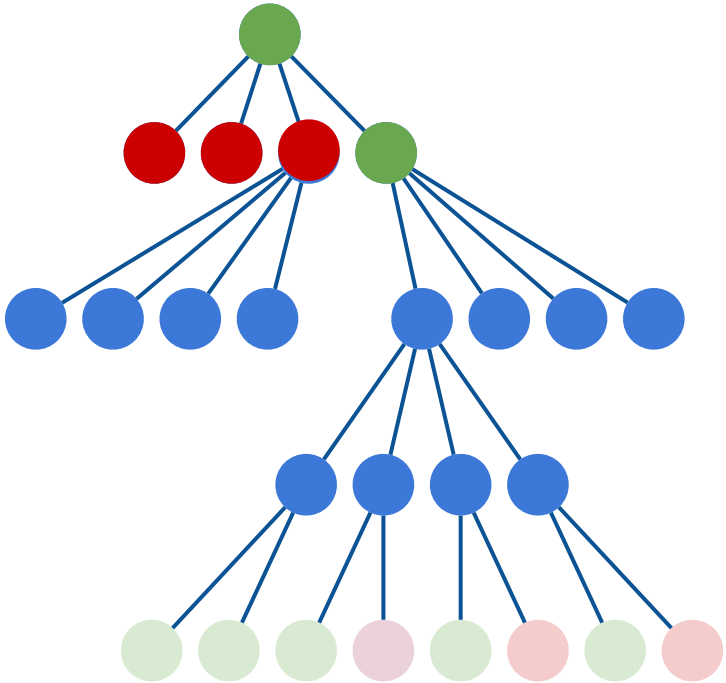
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

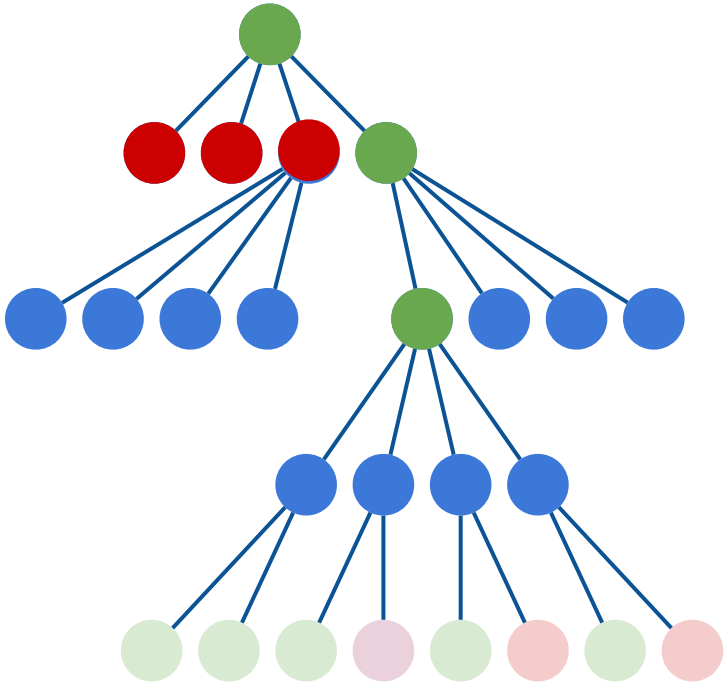
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

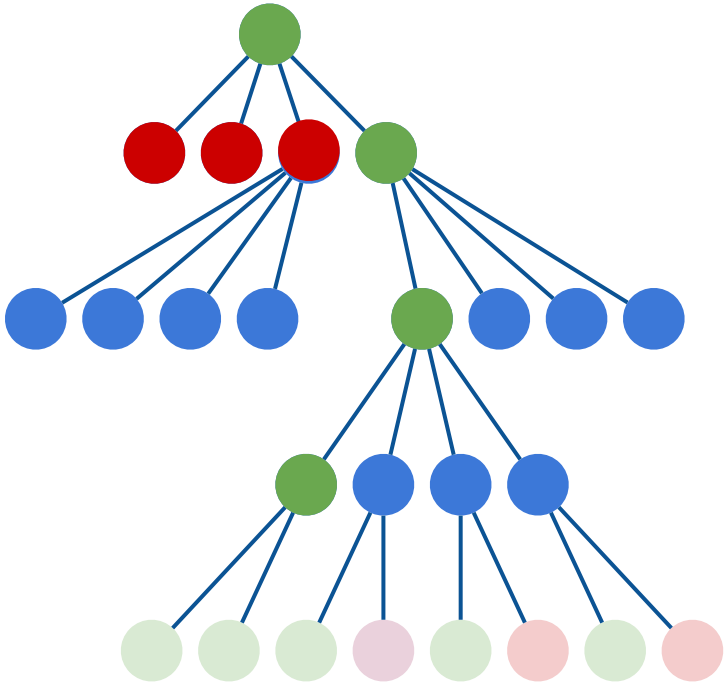
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

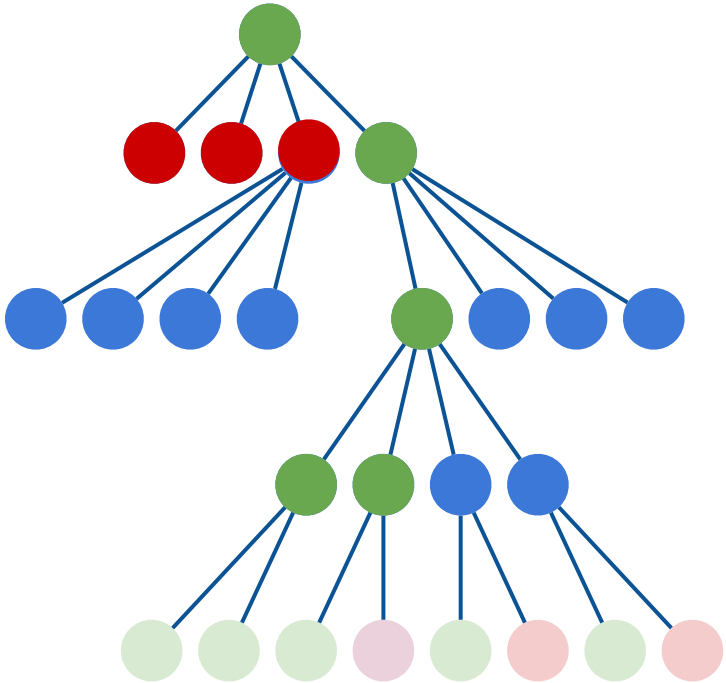
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

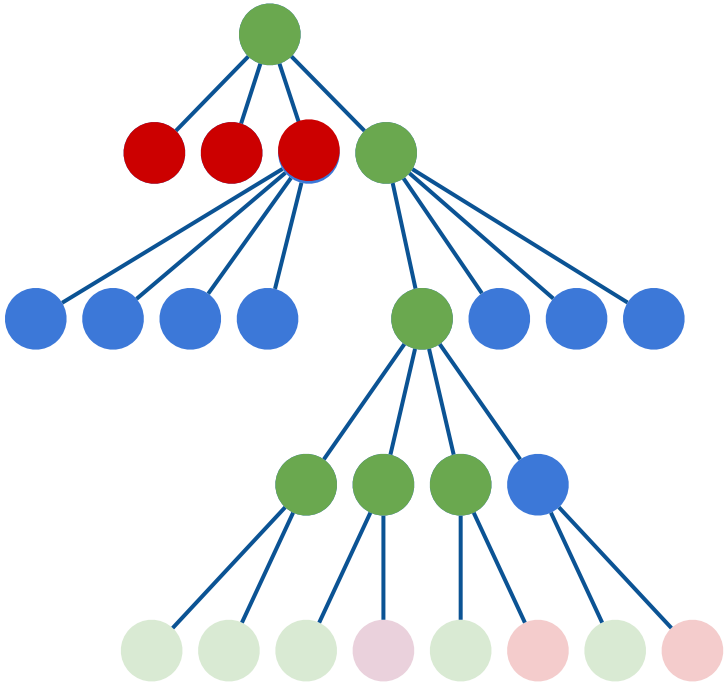
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

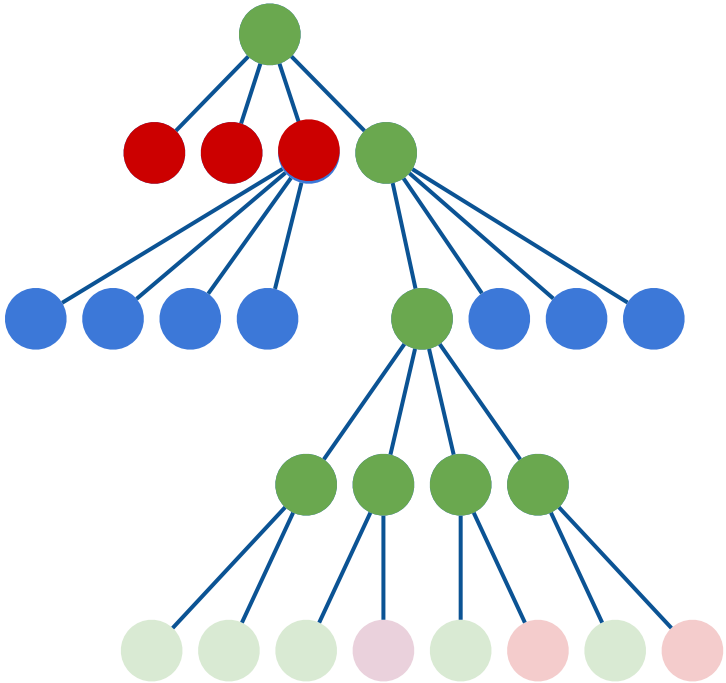
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

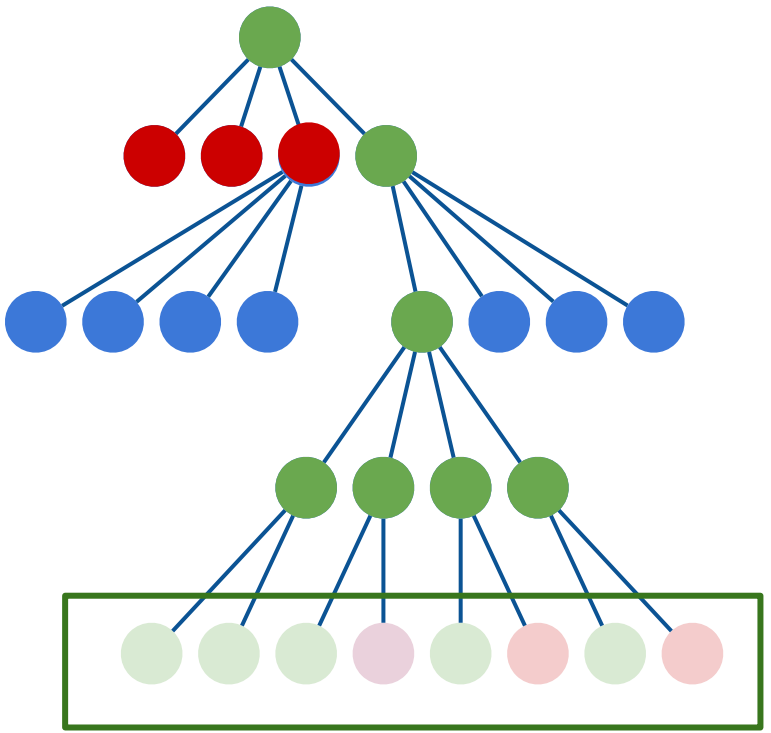
Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

Searching



Once the tree is constructed, we search the tree for game entities that exist in the frustum.

- Nodes can represent a quadrant or a game entity.
- For each node, visit each child and check if the frustum exist in that space.
- If exist, continue deeper into the tree else skip.
- When a node represents a game entity, do fancy math to determine frustum intersection.

Our search is now ***O (log N)!*** Awesome!!!

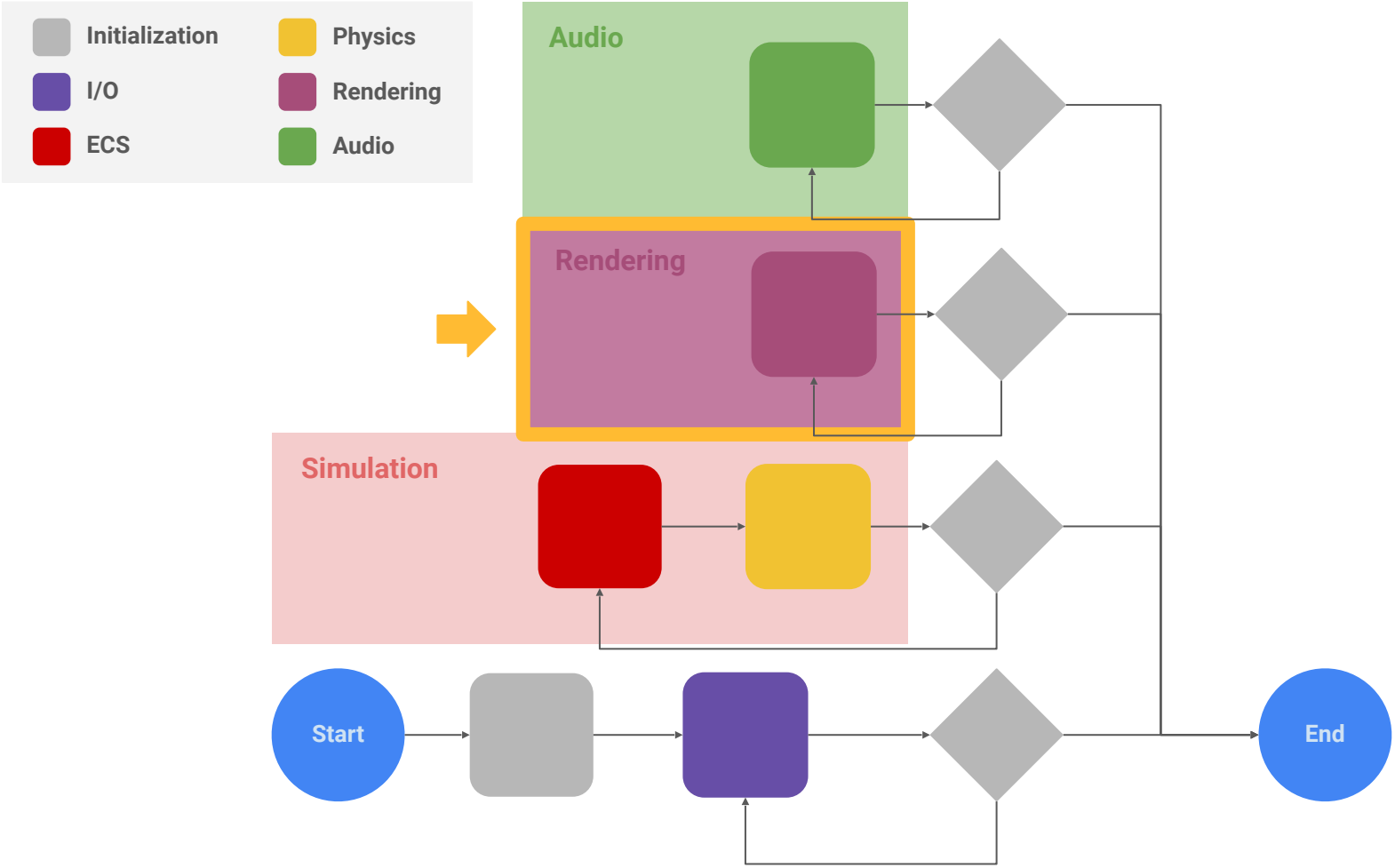
Problems (Not really, more like caveats …)

Like all tree data structures, not really cache friendly.

- Can always represent the tree as a heap but you still jump around each node.
- Can’t be SIMD-ed.
- Low occupancy on the GPU.

But when you’re iterating large data sets, time complexity has higher importance.

Regardless, we now have the list of entities that we can send to the GPU.


Initialization
Physics
I/O
Rendering
ECS
Audio
Audio
Rendering
Simulation
Start
End

Rendering

The renderer receives a list of items to process. Then what?

Responsibilities include:

- Descriptor management, buffers and textures.
- Swapchain management.
- Synchronization.
- Command list / buffer recording.
  - Ownership transfers.
  - Memory barriers, image layout transitions.
- Compute dispatches.

Hold on …

This all sounds alien. Let’s take a step back!

How is a frame structured?

A single frame consists of multiple “passes”.

Fundamentally, each pass computes outputs based on inputs.

How is a frame structured?

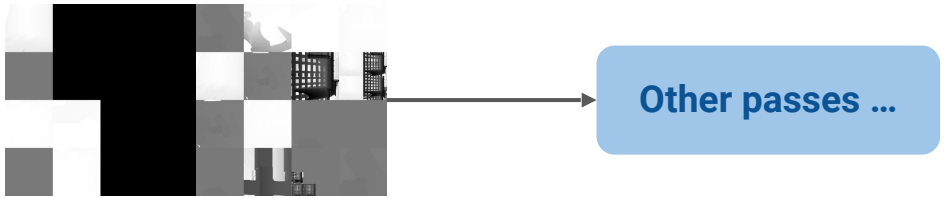
A single frame consists of multiple “passes”.

Fundamentally, each pass computes outputs based on inputs.

How is a frame structured?

A single frame consists of multiple “passes”.

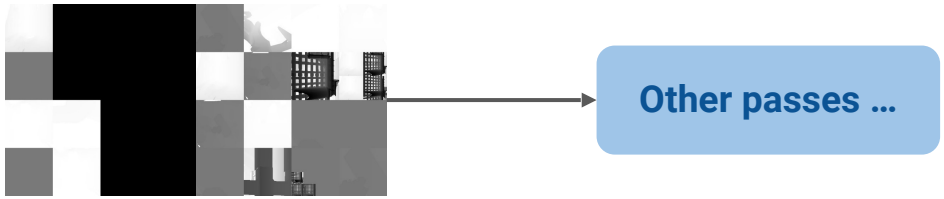
Fundamentally, each pass computes outputs based on inputs.



How is a frame structured?

A single frame consists of multiple “passes”.

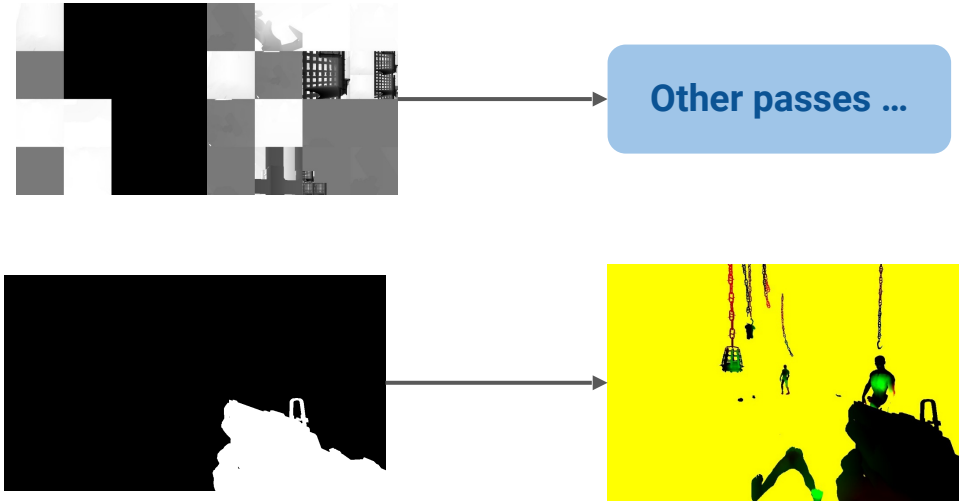
Fundamentally, each pass computes outputs based on inputs.



How is a frame structured?

A single frame consists of multiple “passes”.

Fundamentally, each pass computes outputs based on inputs.



How is a frame structured?

A single frame consists of multiple “passes”.

Fundamentally, each pass computes outputs based on inputs.

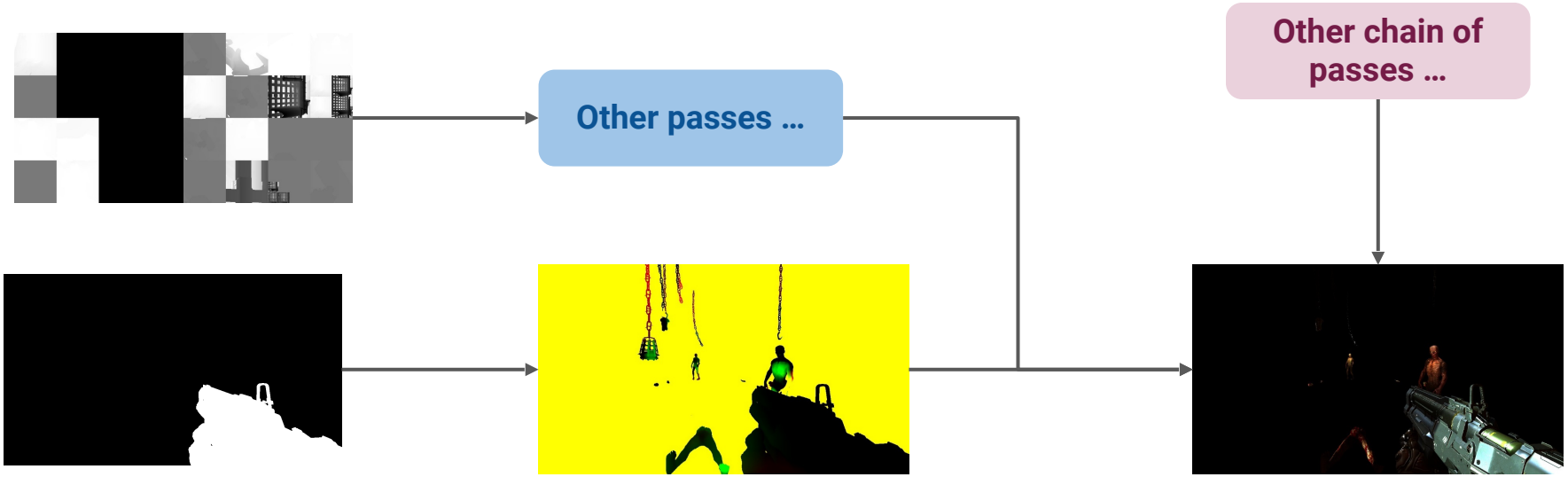
Other passes …

Other chain of passes …

How is a frame structured?

A single frame consists of multiple “passes”.

Fundamentally, each pass computes outputs based on inputs.



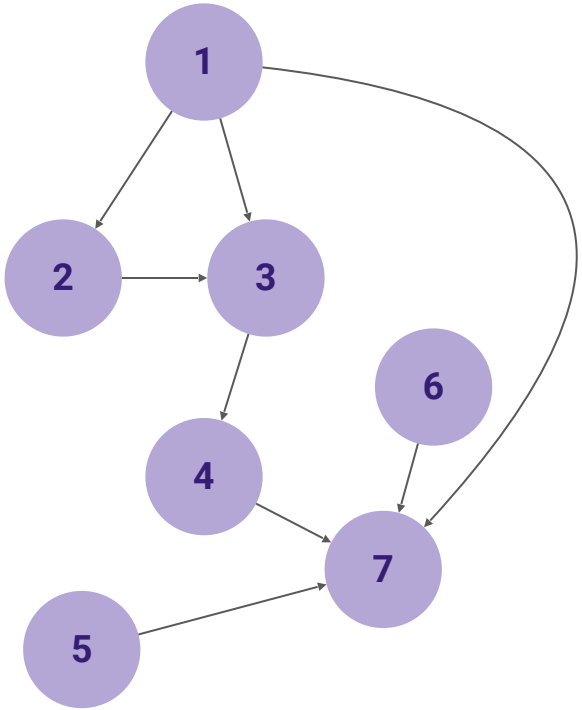
But how do we sort ***DEPENDENCIES*** between each pass?

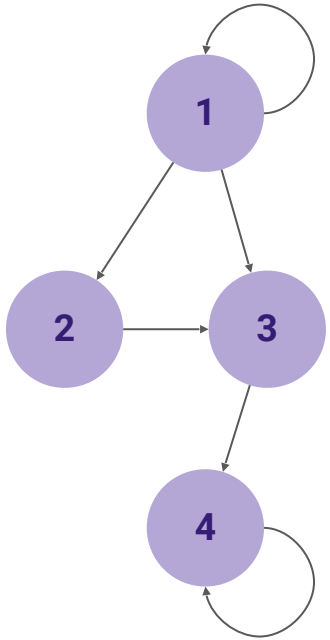
Directed Acyclic Graph (DAG)

Core idea:

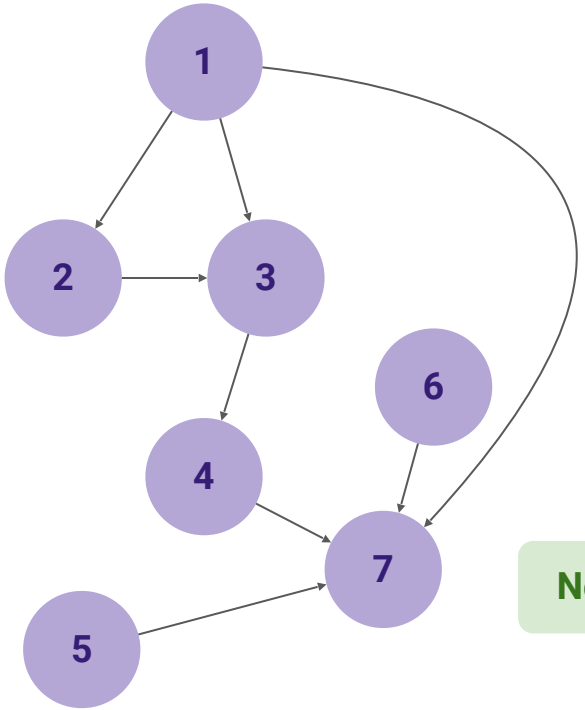
- Represent the frame as a ***Directed Acyclic Graph (DAG).***
- Each pass is represented as a vertex.
- Vertices that are dependent on the output of another are connected by a directed edge.
- A vertex cannot depend on itself (acyclic).
- Once the graph is built, topological sort to get the order of execution.

Directed Acyclic Graph (DAG)

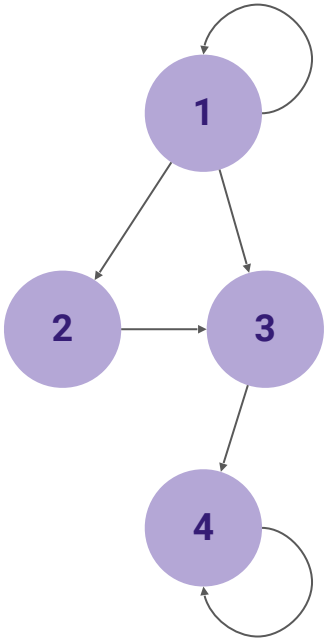




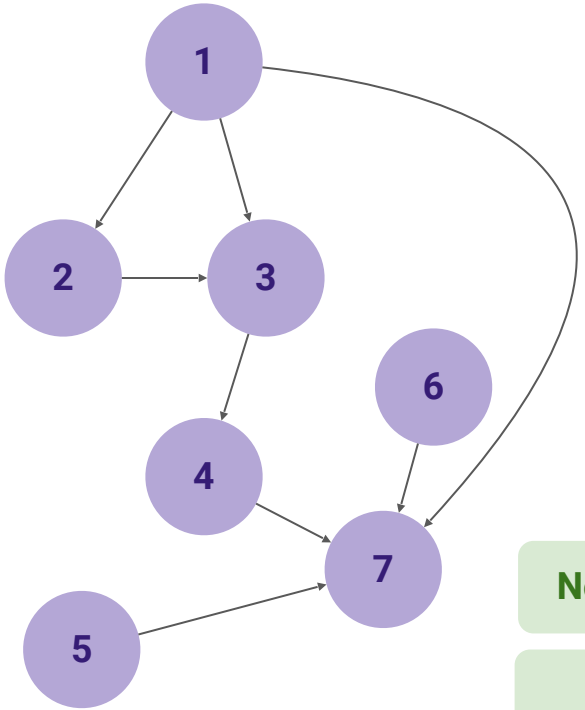
Directed Acyclic Graph (DAG)



No cycles!

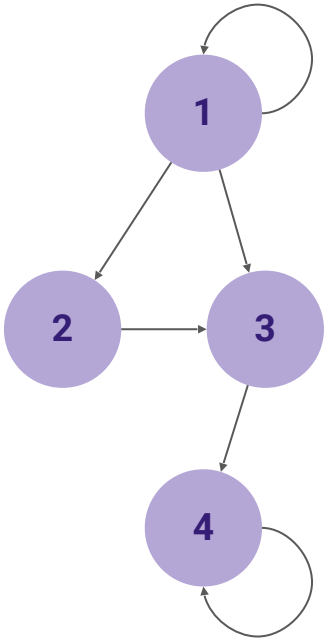


Directed Acyclic Graph (DAG)

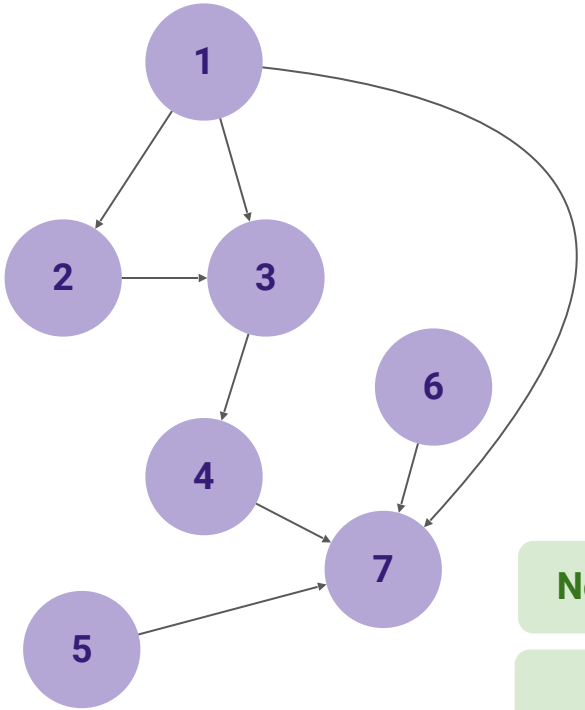


No cycles!

Valid

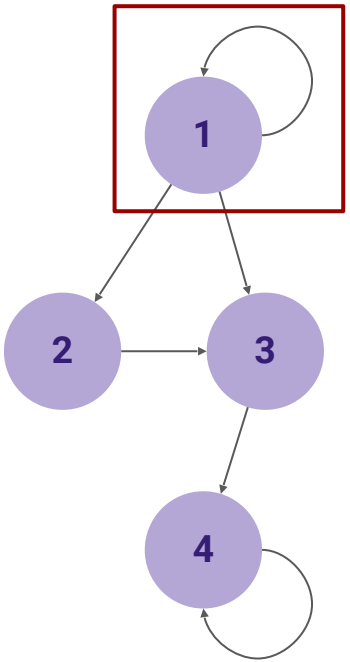


Directed Acyclic Graph (DAG)

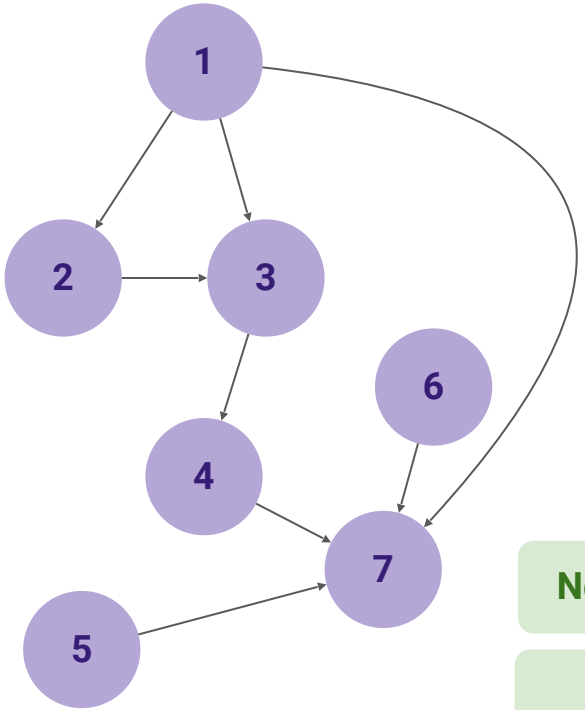


No cycles!

Valid

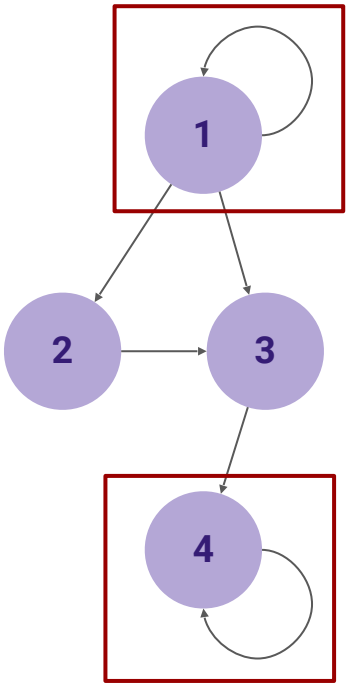


Directed Acyclic Graph (DAG)

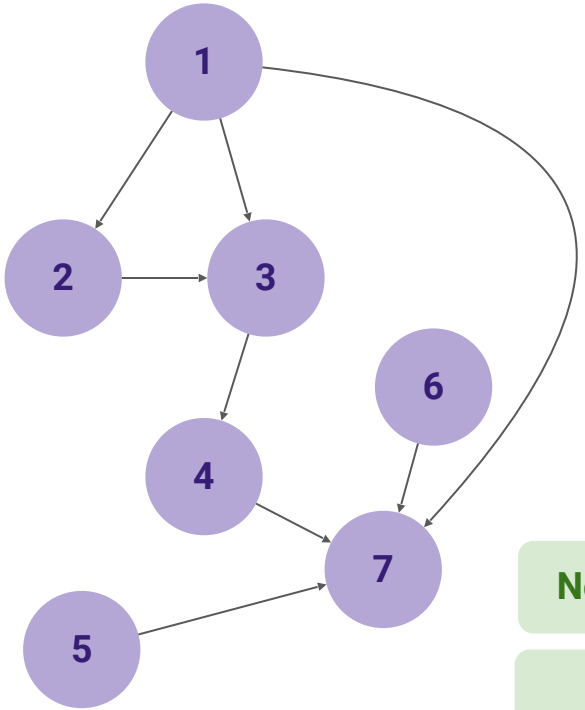


No cycles!

Valid

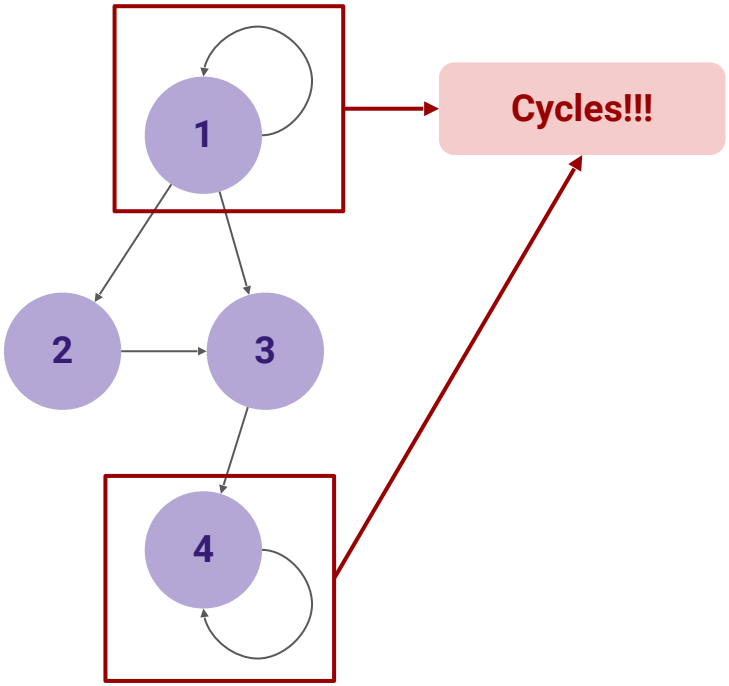


Directed Acyclic Graph (DAG)

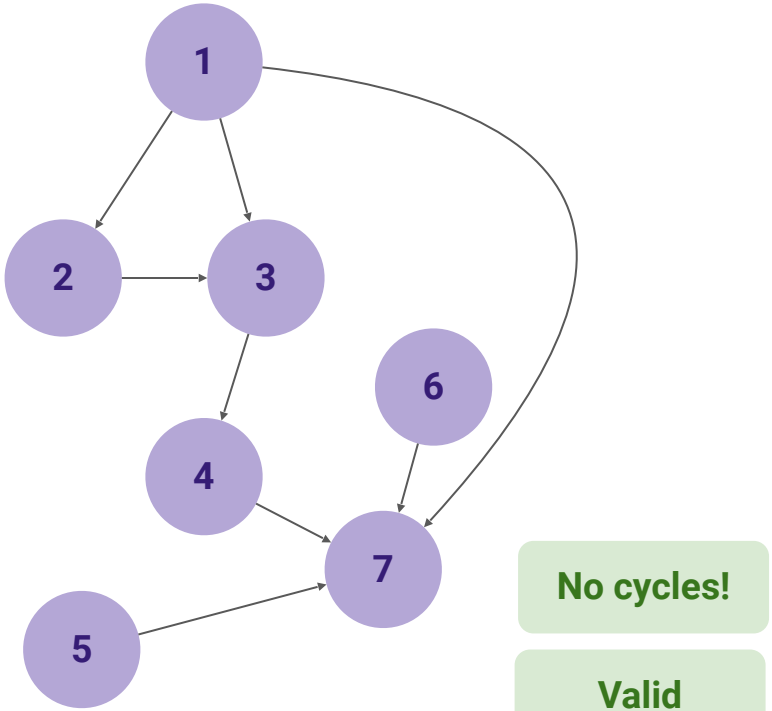


No cycles!

Valid

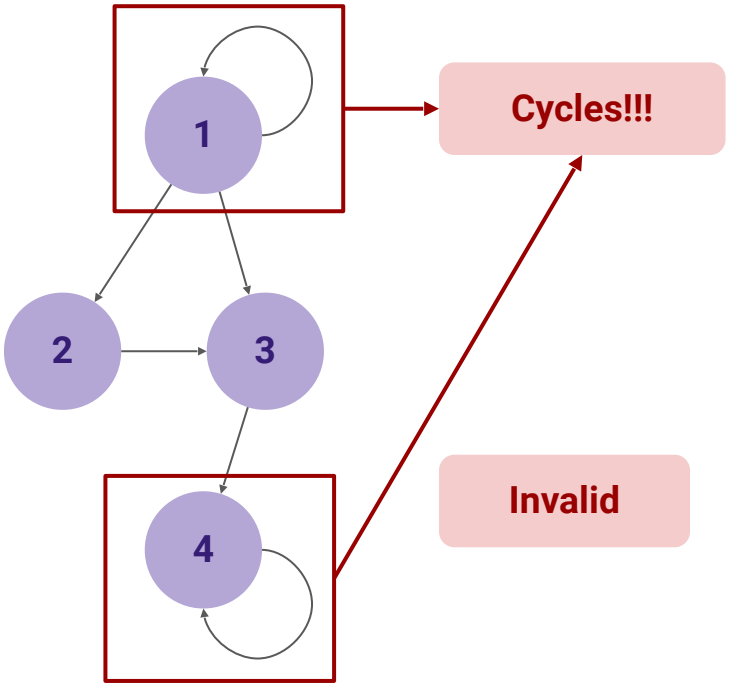


Directed Acyclic Graph (DAG)



No cycles!

Valid



Sorting the DAG

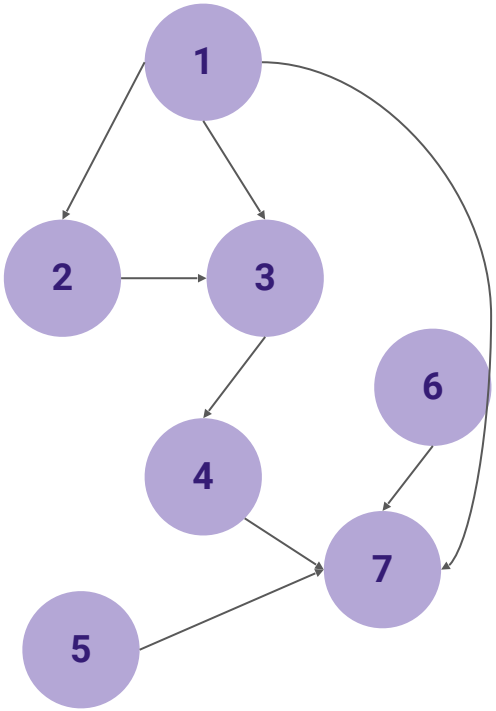
A DAG needs to be ***topologically sorted,*** into a linear list. Vertices that has more incoming edges needs to be lower in the list.

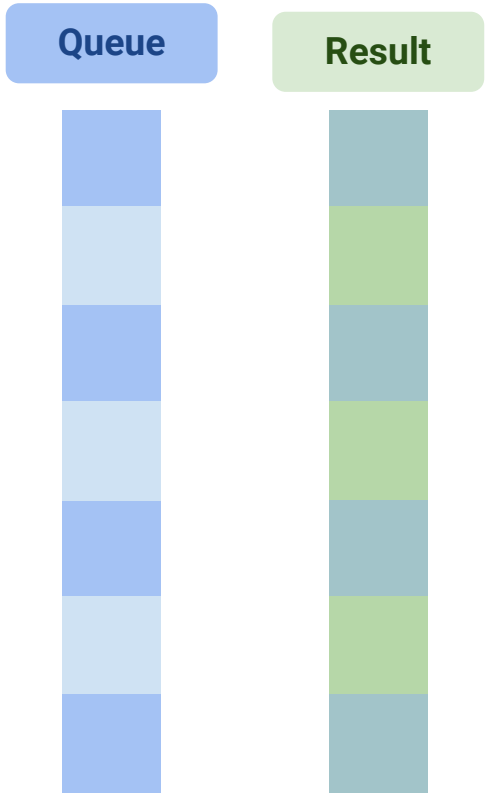
Two methods,

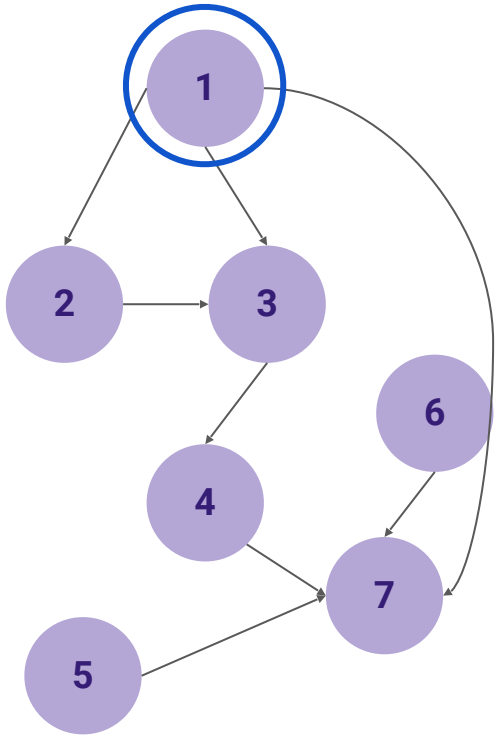
1. Depth-first search, O(N).
2. Kahn’s algorithm, O(N),

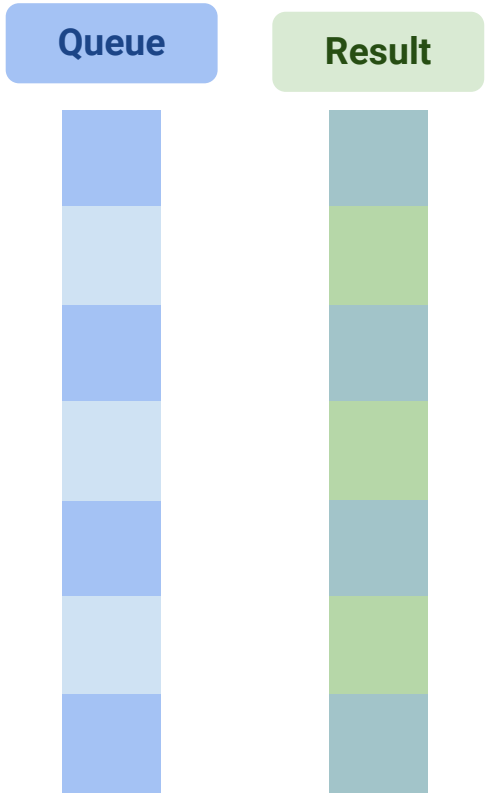
Kahn’s algorithm is preferred. DFS can’t detect cycles.

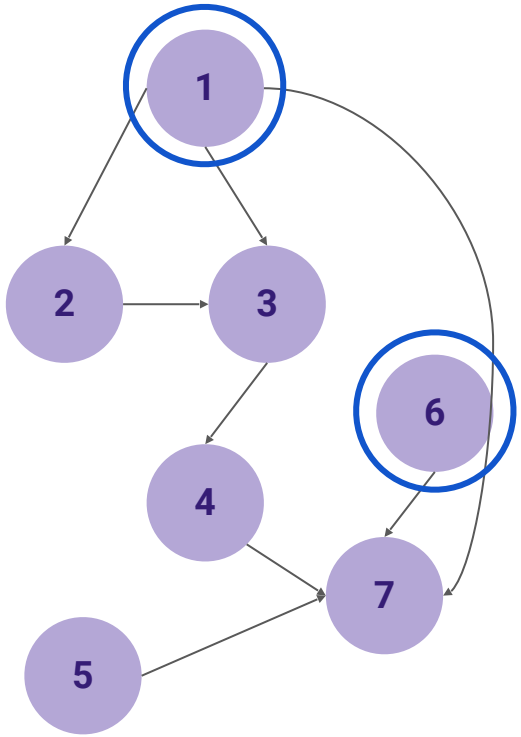
So how do we sort with Kahn’s algorithm?

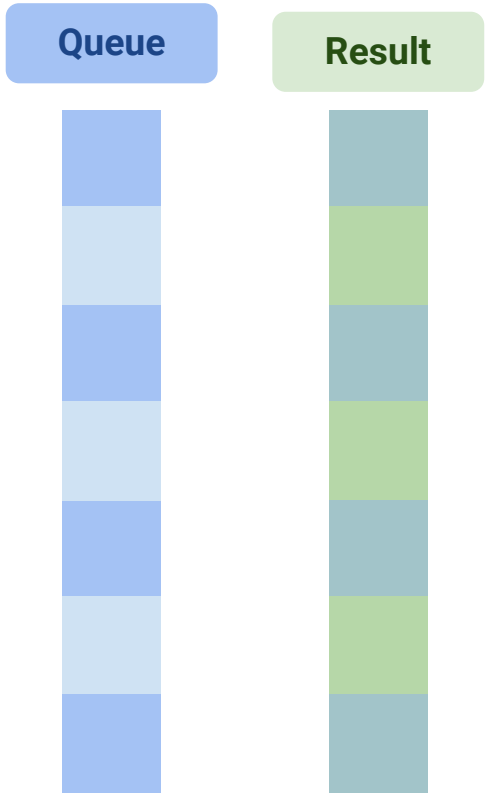

1
2
3
4
5
6
7

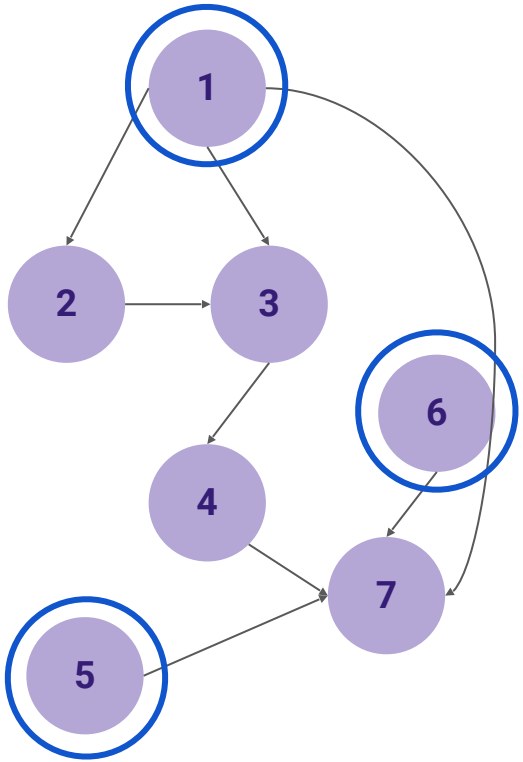

Queue
Result

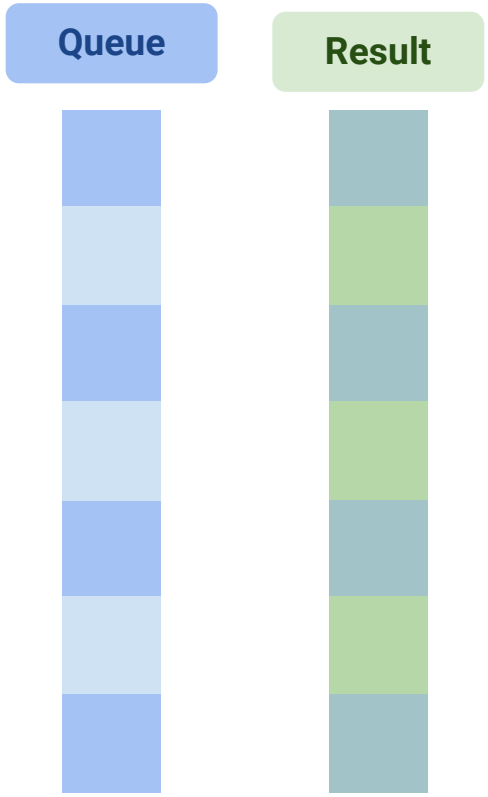

1
2
3
4
5
6
7

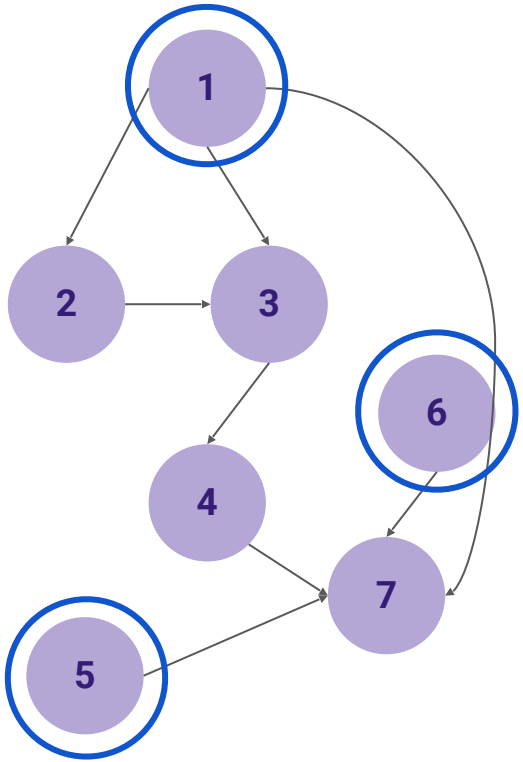

Queue
Result

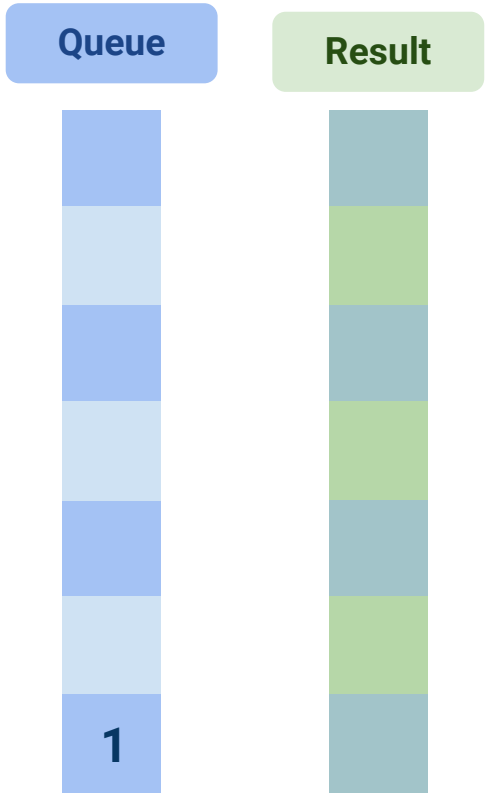

1
2
3
6
4
7
5

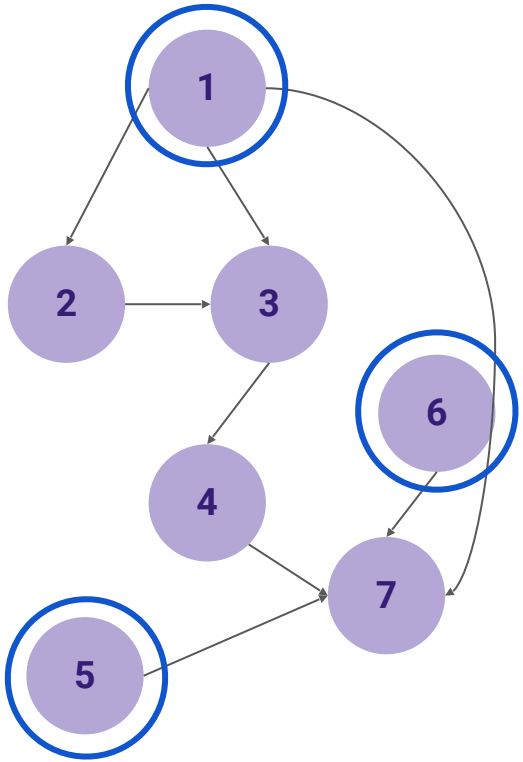

Queue
Result

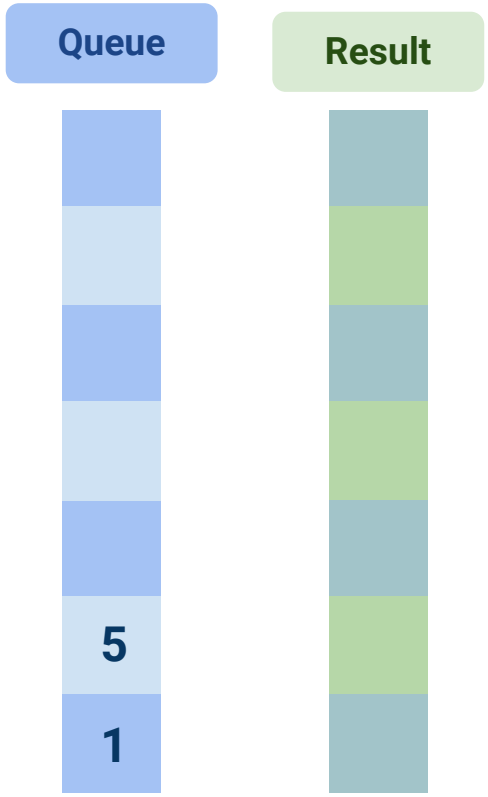

1
2
3
4
5
6
7

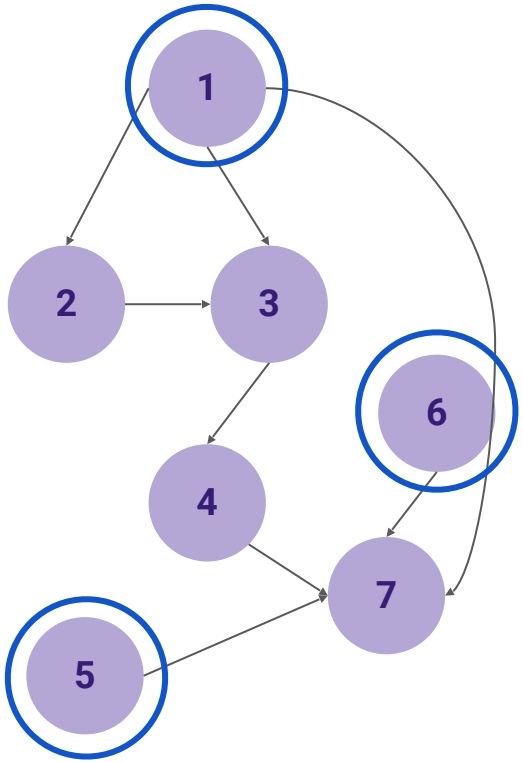

Queue
Result

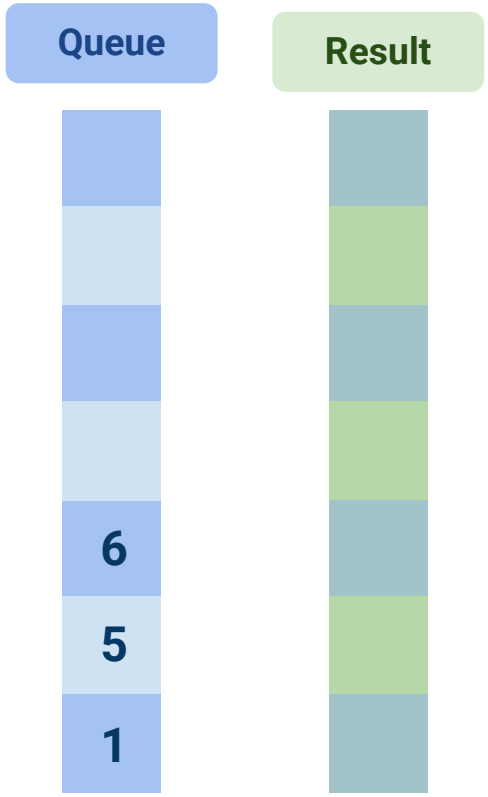

1
2
3
6
4
7
5

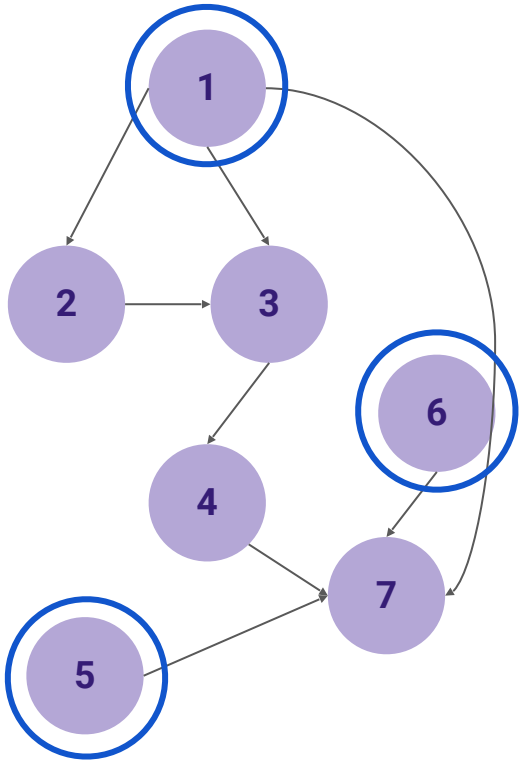

Queue
Result
1

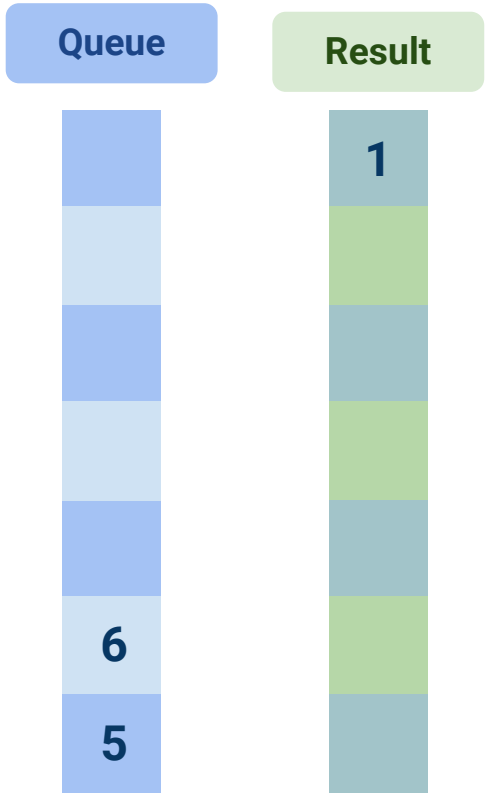

1
2
3
6
4
7
5

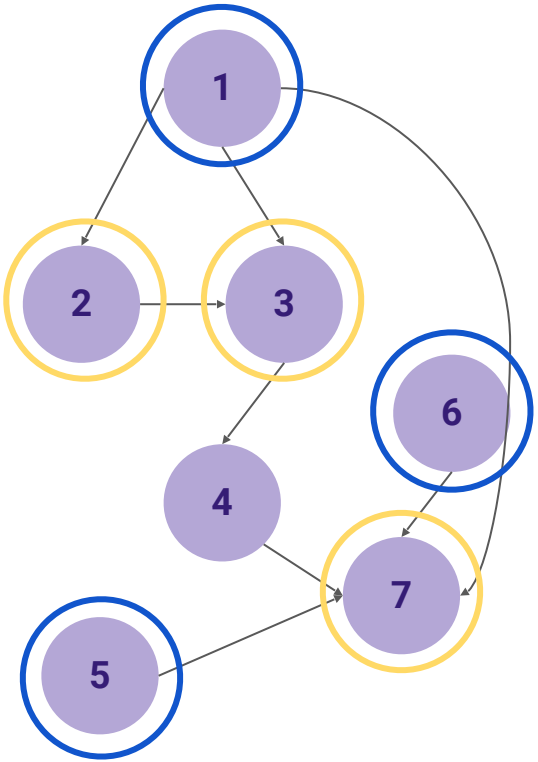

Queue
Result
5
1

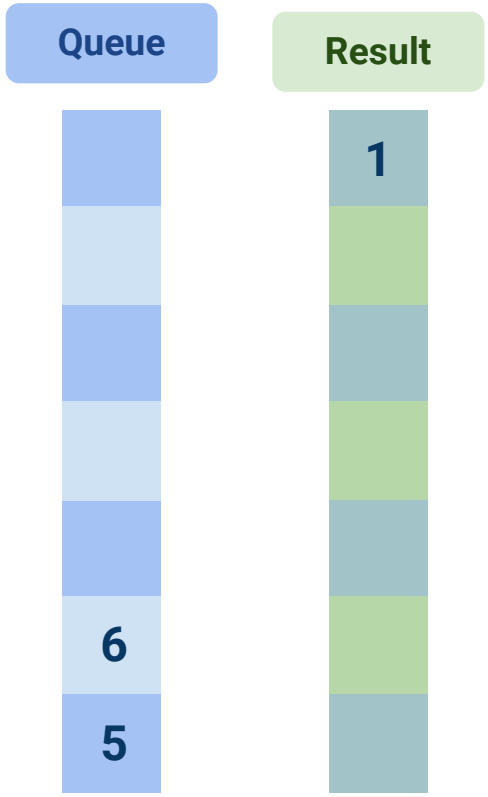

1
2
3
6
4
7
5

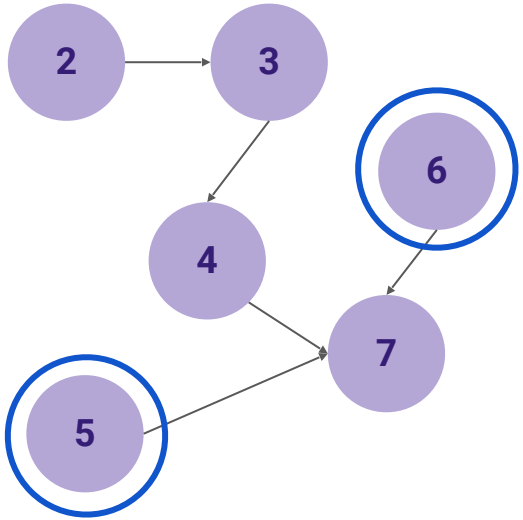

Queue
Result
6
5
1

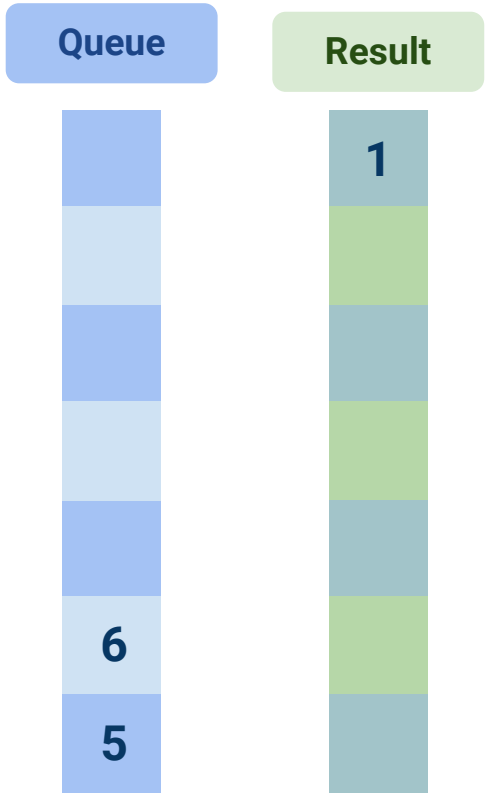

1
2
3
6
4
7
5

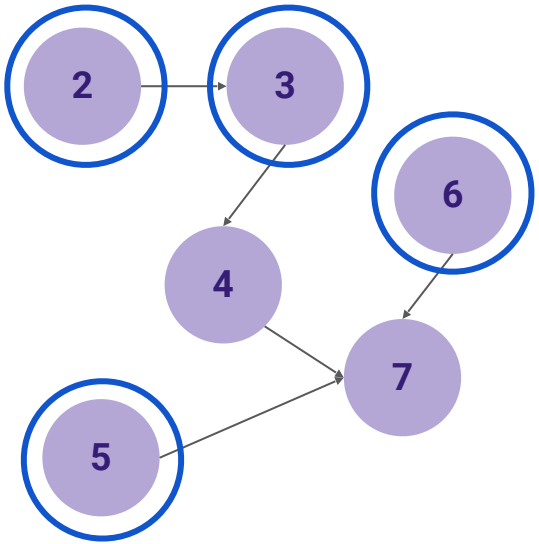

Queue
Result
1
6
5

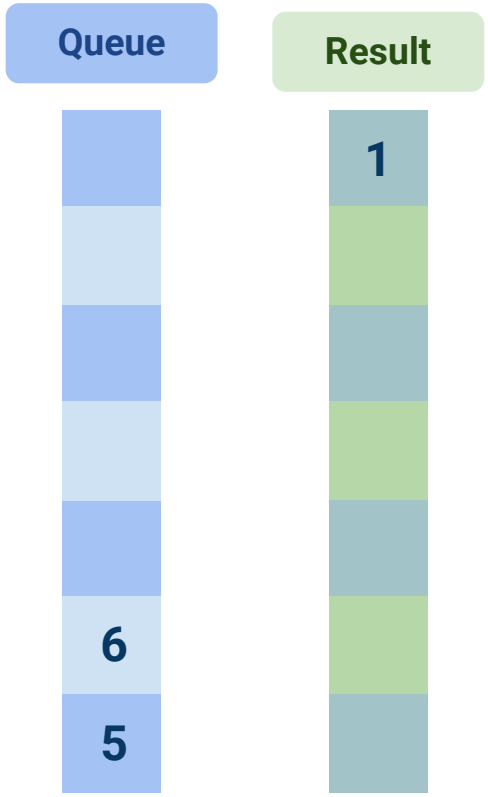

1
2
3
6
4
7
5

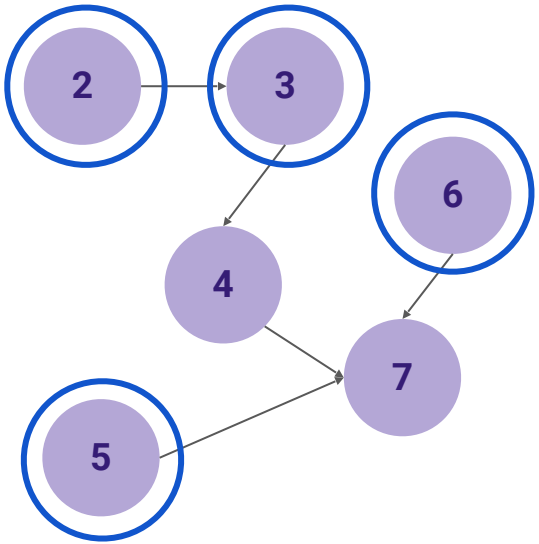

Queue
Result
1
6
5

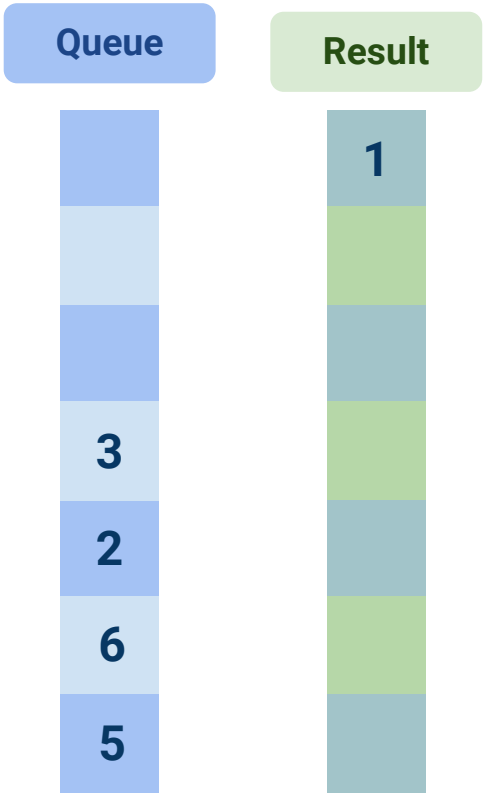

2
3
6
4
7
5

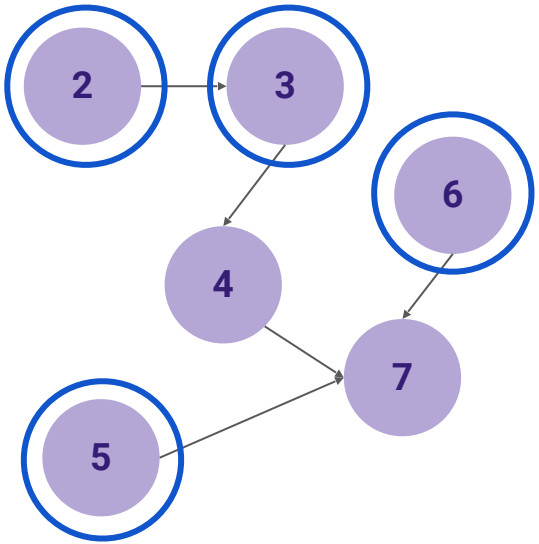

Queue
Result
1
6
5

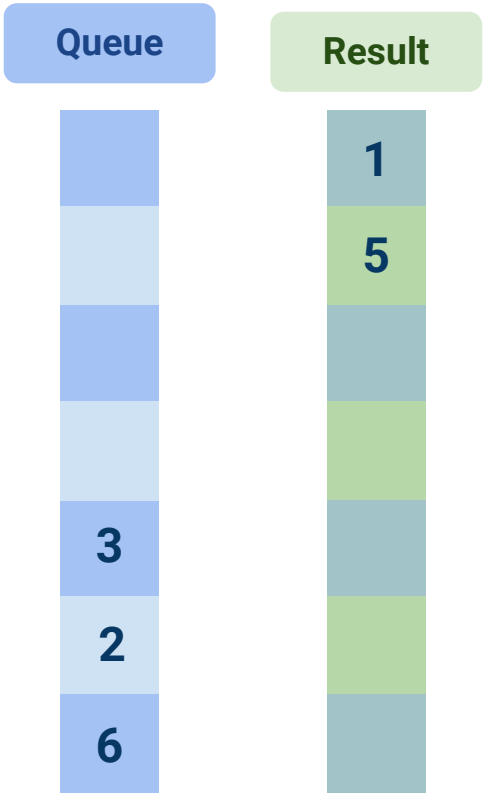

2
3
6
4
7
5

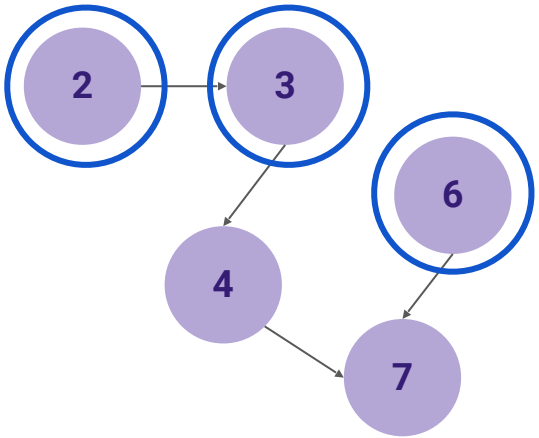

Queue
Result
1
6
5

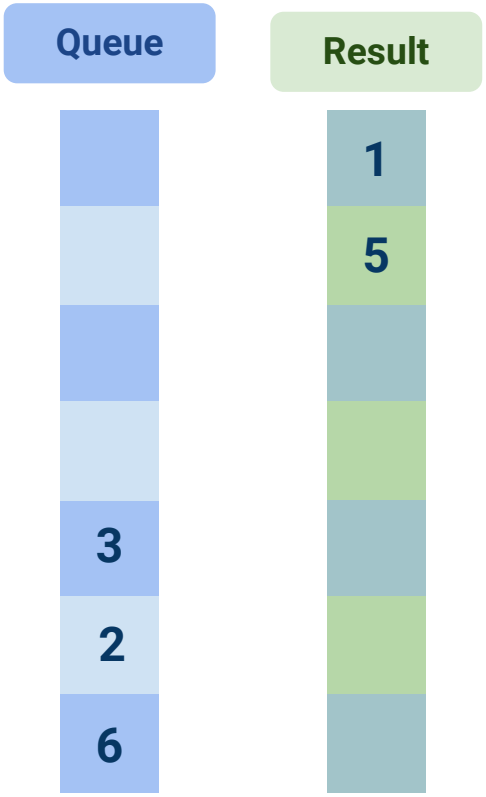

2
3
6
4
7
5

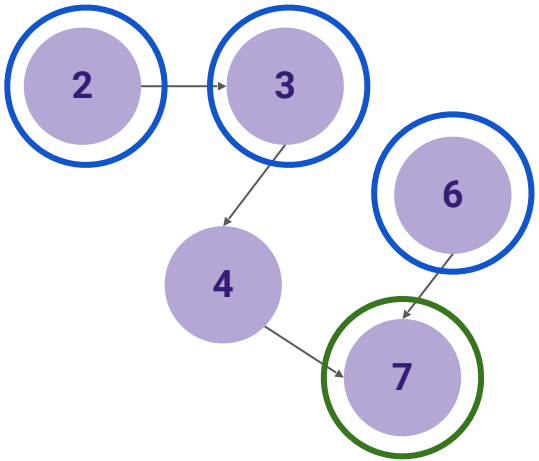

Queue
Result
1
3
2
6
5

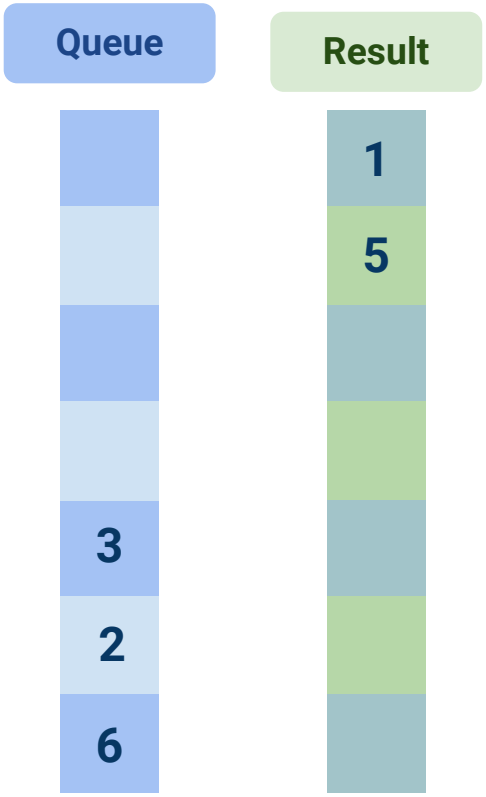

2
3
6
4
7
5

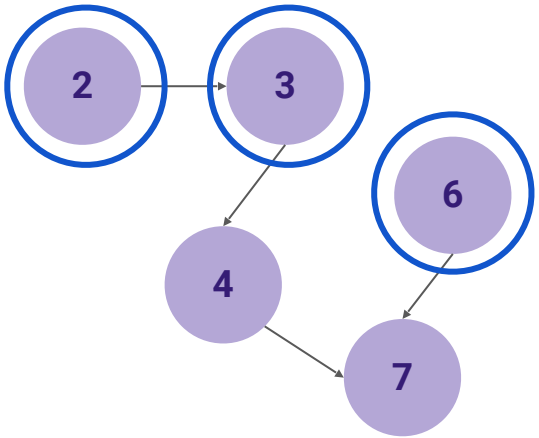

Queue
Result
3
2
6
1
5

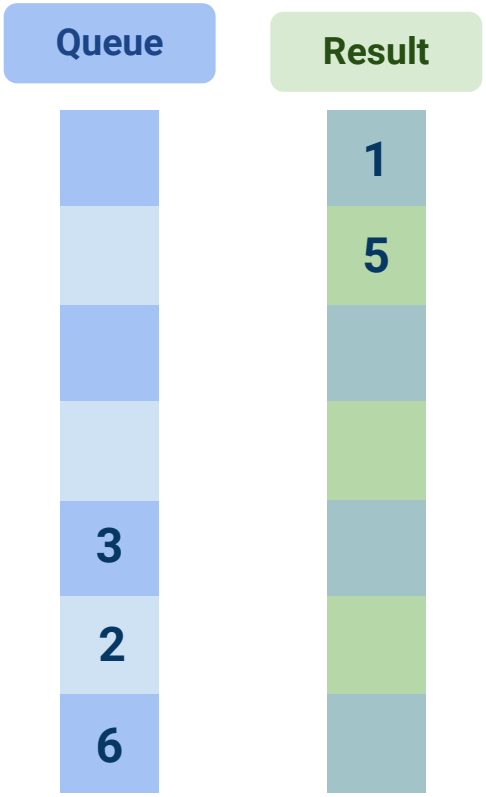

2
3
6
4
7

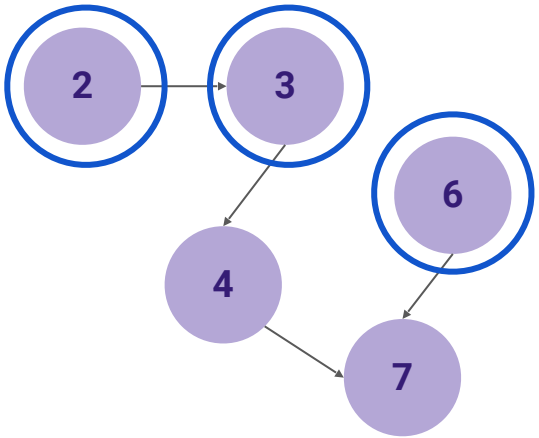

Queue
Result
3
2
6
1
5

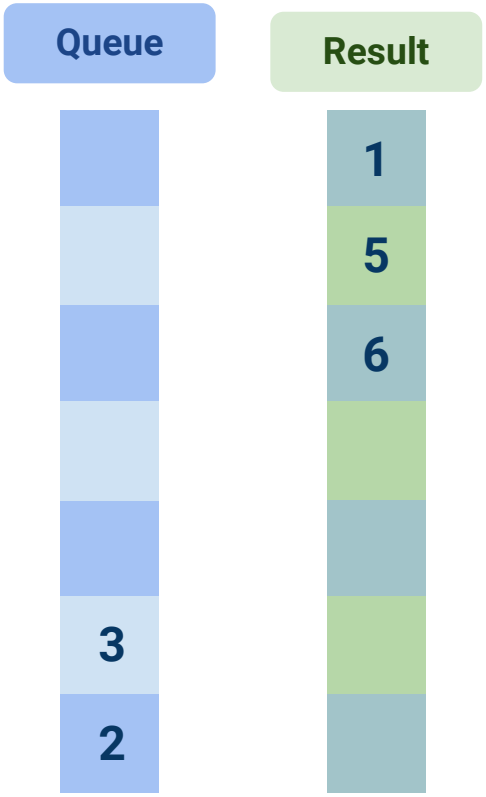

2
3
6
4
7

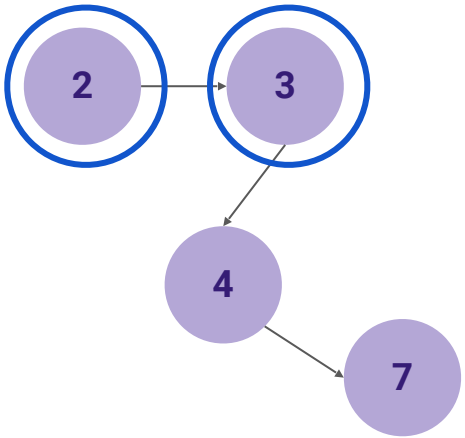

Queue
Result
3
2
6
1
5

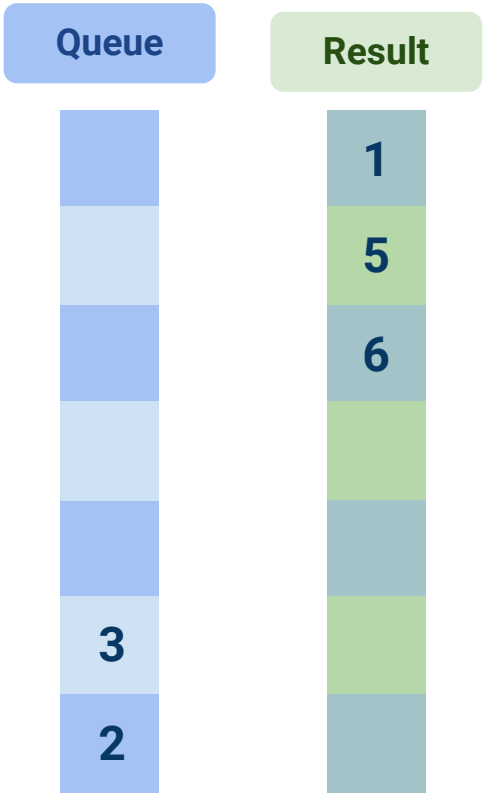

2
3
6
4
7

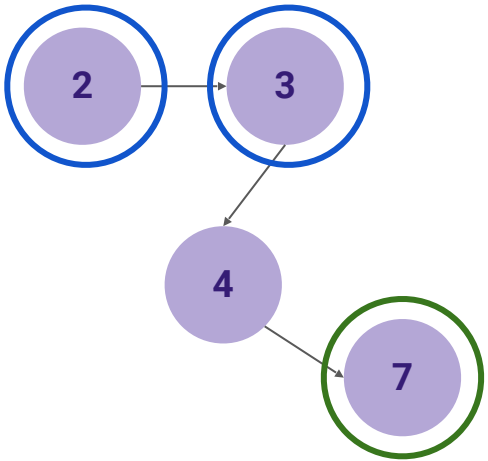

Queue
Result
3
2
6
1
5

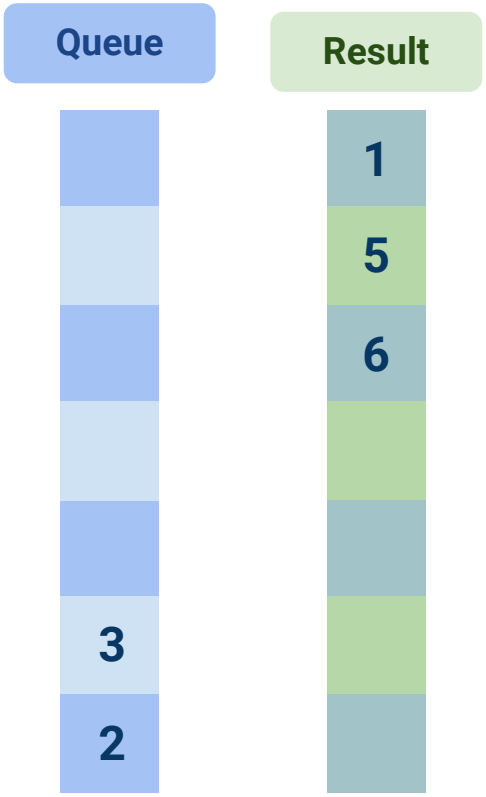

2
3
6
4
7

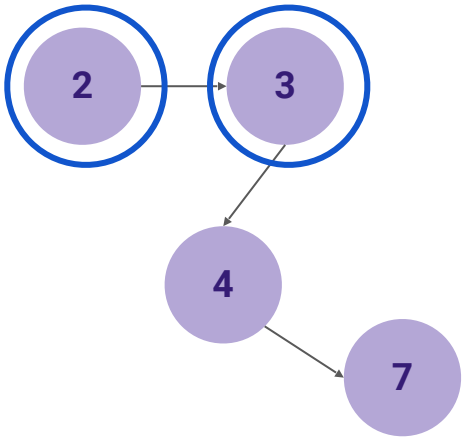

Queue
Result
3
2
1
5
6

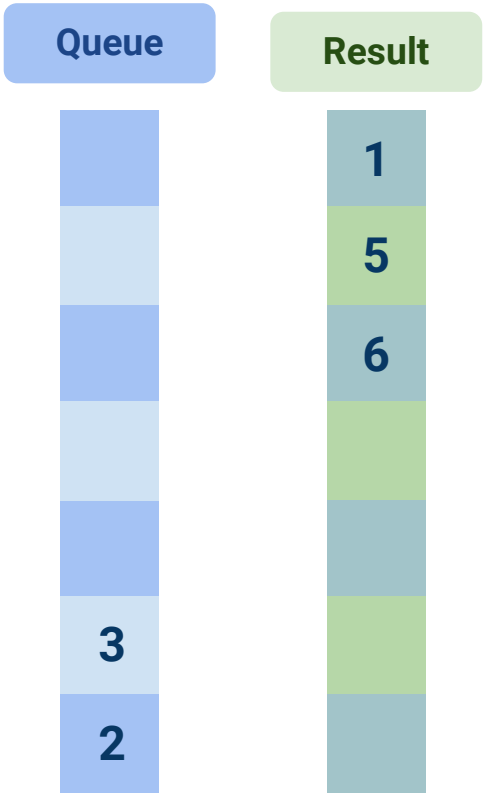

2
3
4
7

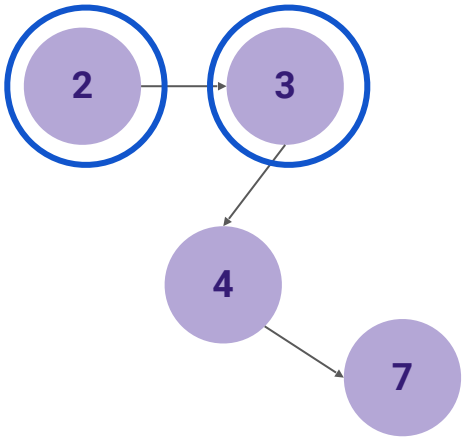

Queue
Result
3
2
1
5
6

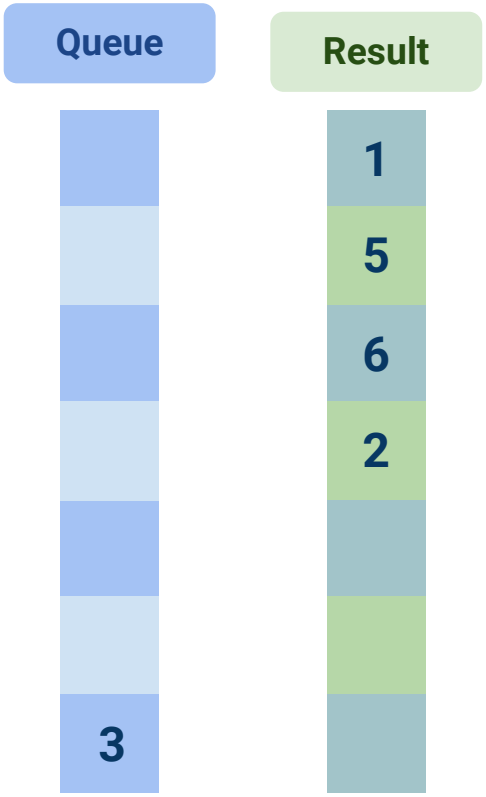

2
3
4
7

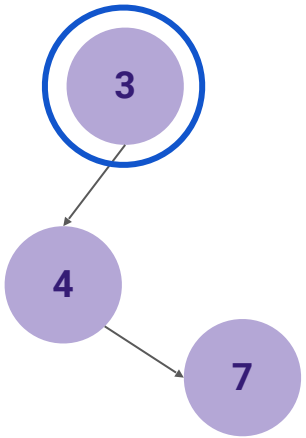

Queue
Result
3
2
1
5
6

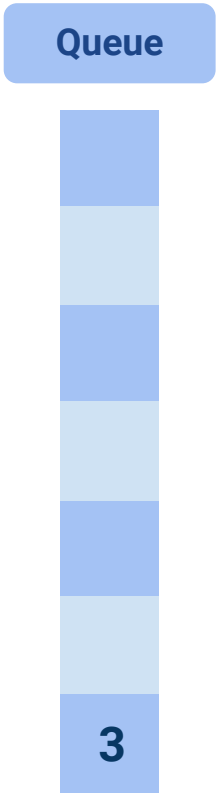

2
3
4
7

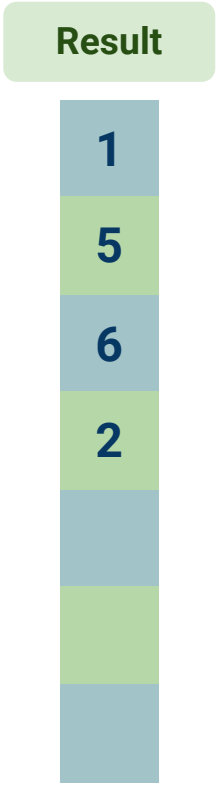

Queue
Result
3
2
1
5
6

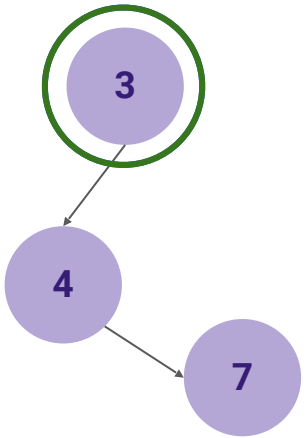

2
3
4
7

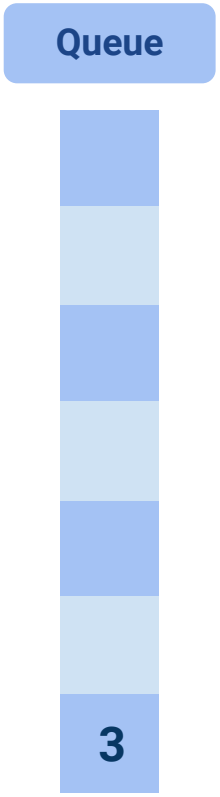

Queue
Result
1
5
6
2
3

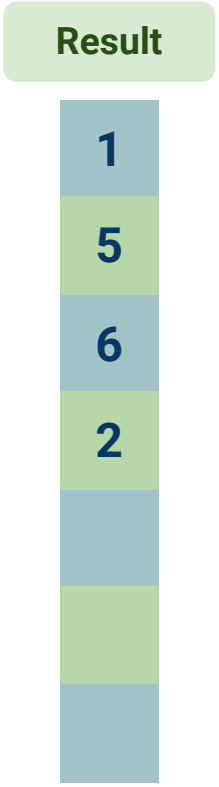

3
4
7

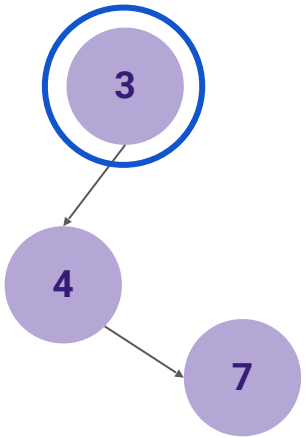

Queue
3

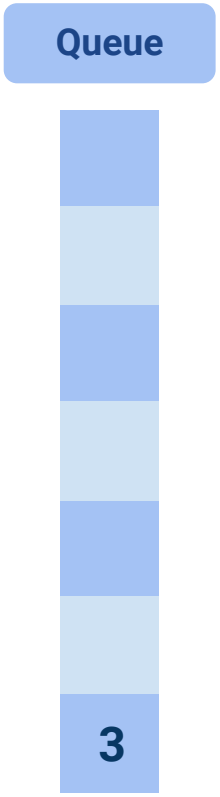

Result
1
5
6
2

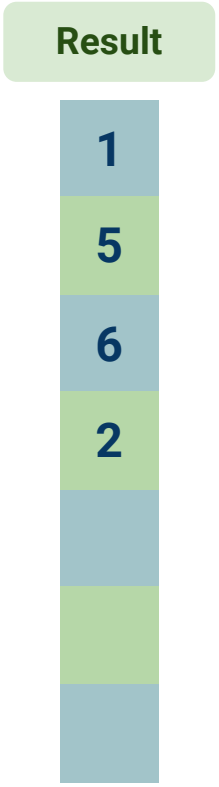

3
4
7

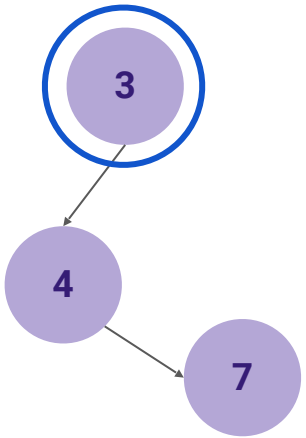

Queue
3

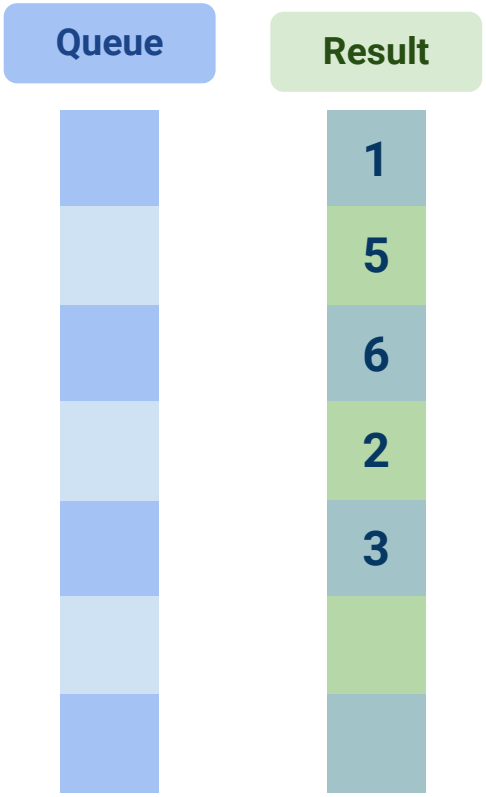

Result
1
5
6
2

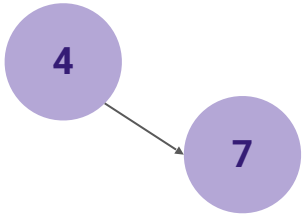

3
4
7

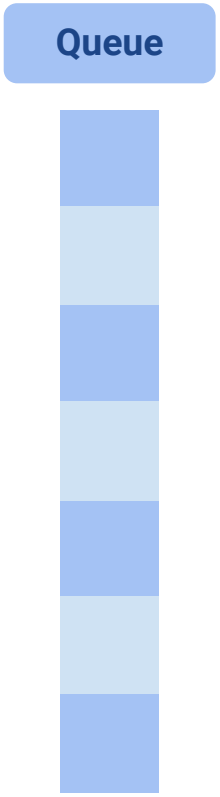

Queue
3

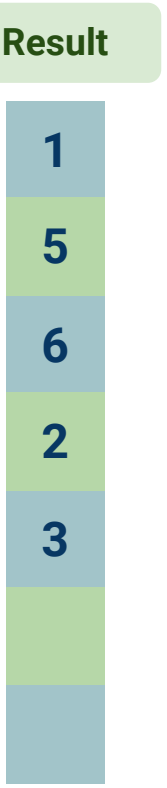

Result
1
5
6
2

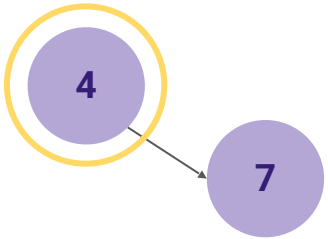

3
4
7

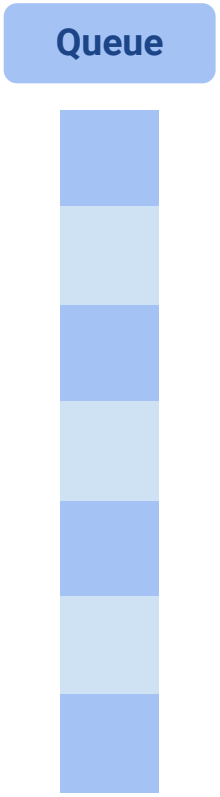

Queue
Result
1
5
6
2
3

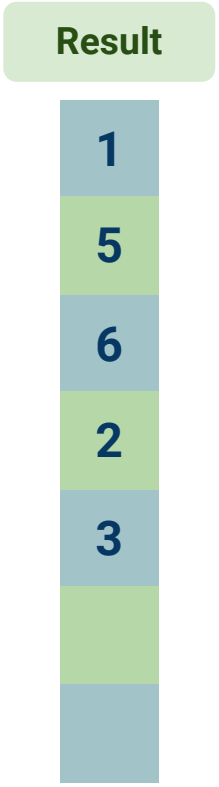

4
7

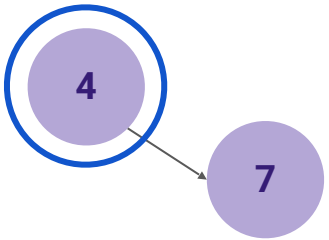

Queue

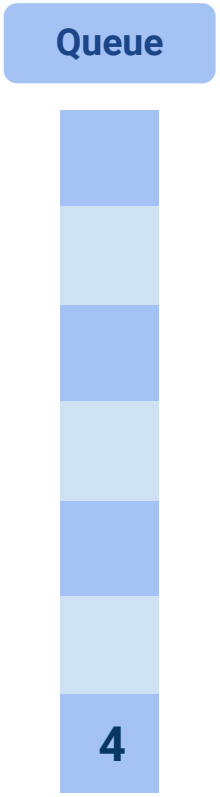

Result
1
5
6
2
3

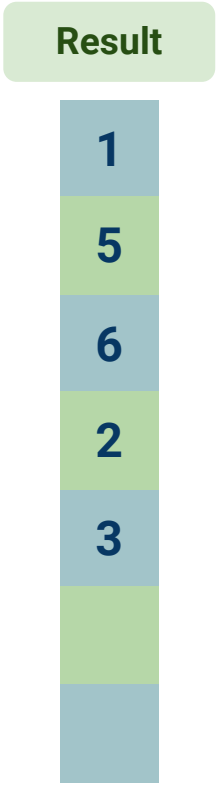

4
7

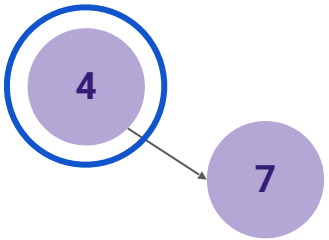

Queue


Result
1
5
6
2
3


4
7


Queue
4


Result
1
5
6
2
3



Queue

Result

1

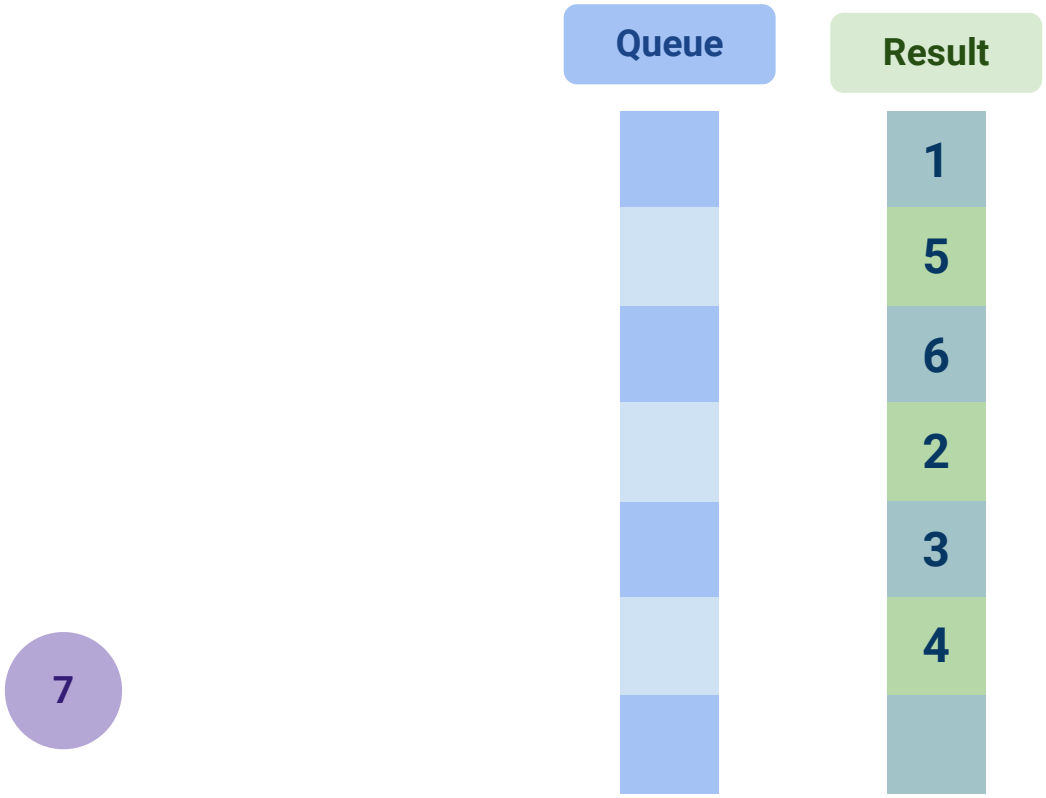
5

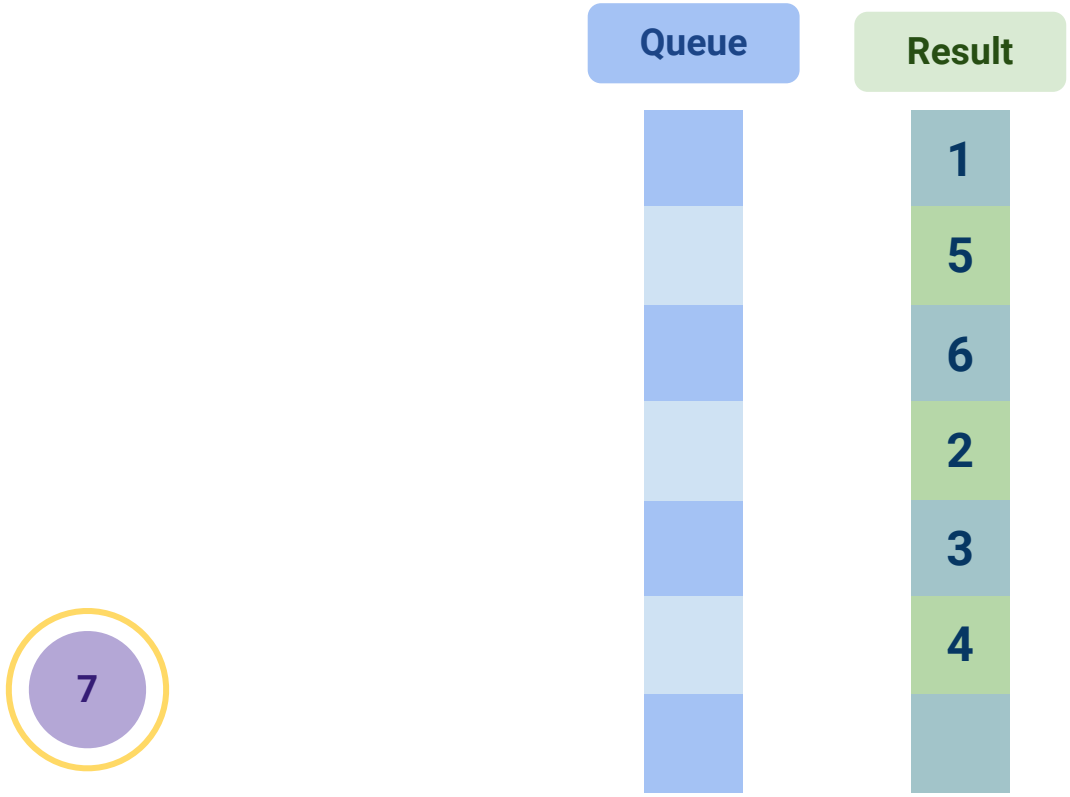
6

2

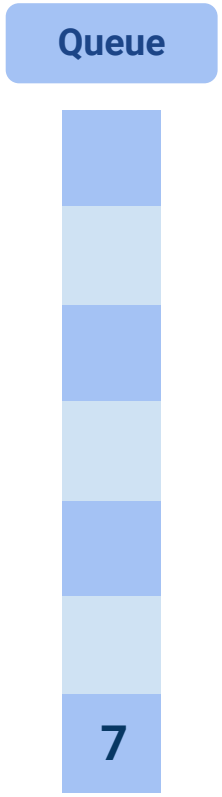
3

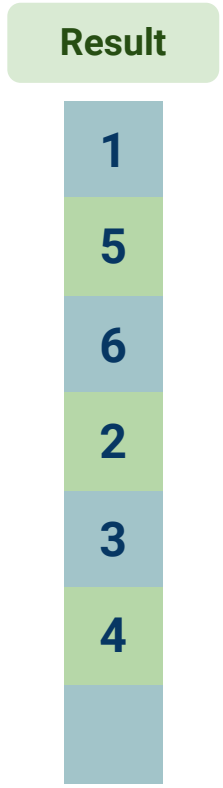
4

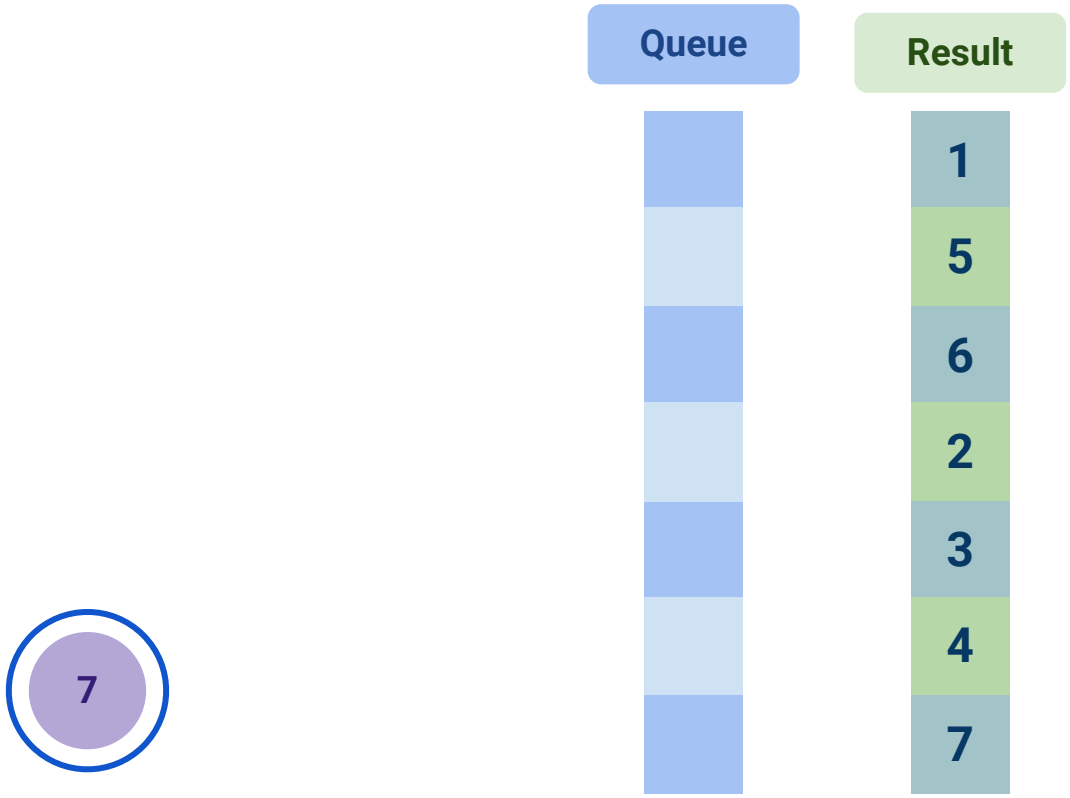

Queue
Result
1
5
6
2
3
4
7

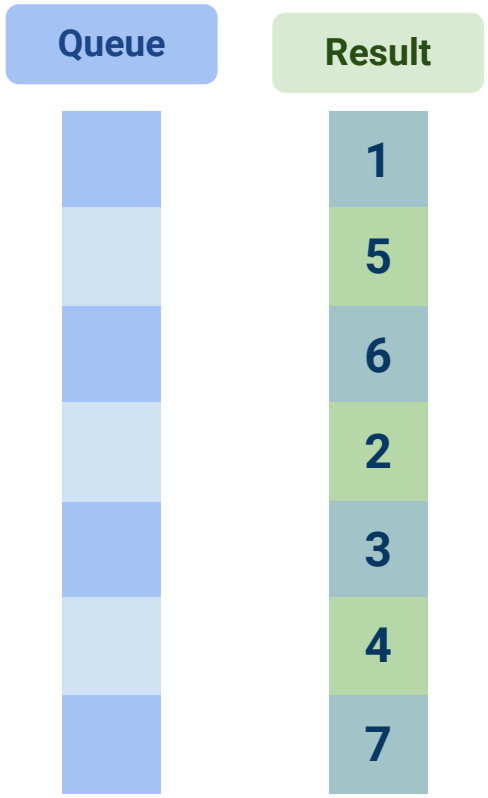

Queue
Result
1
5
6
2
3
4
7


7


Queue
7


Result
1
5
6
2
3
4


Queue
Result
1
5
6
2
3
4
7
7


Queue
Result
1
5
6
2
3
4
7

Opportunities from using a DAG

A frame does not need to be hardcoded into the renderer!

- During development, passes can be defined in a text file format (XML, JSON and etc).
- Renderer becomes more extensible across different projects.
- Optimized GPU resource usage.

Other applications of a DAG:

- Build systems, package dependency management
- CPU side task scheduling
  - Taskflow

That’s all folks … :)

Thank You!

afiqyeong@gmail.com

https://www.linkedin.com/in/afiqyeong/