

Data Structures and Algorithms

Lesson 2: *Declare your intentions (not your actions)*



ADTs, stacks, queues, priority queues, sets, maps

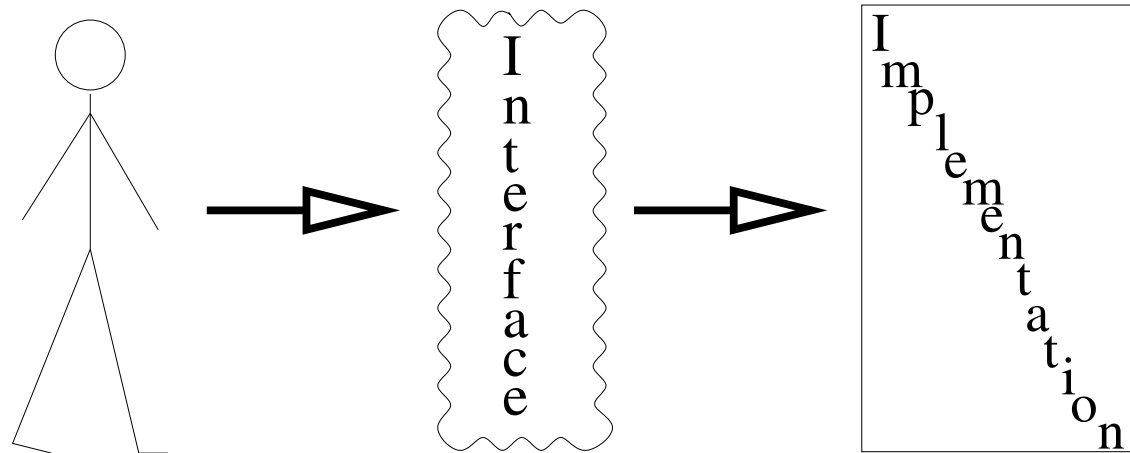
Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



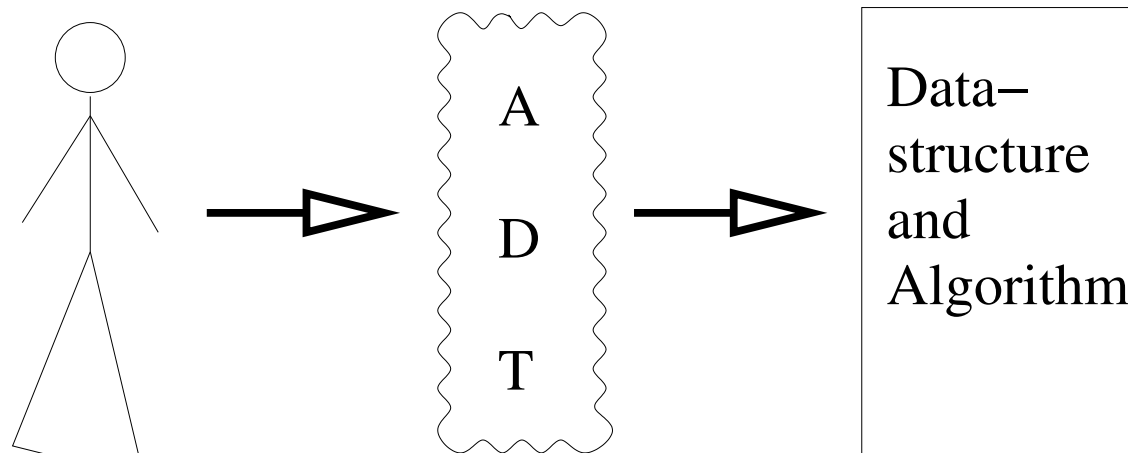
Encapsulation

- In the OO-methodology you separate the **interface** from the implementation
- The implementation is hidden (encapsulated) and may be changed without affecting how the class is used
- This is done in classes with the set of public methods being the interface



Abstract Data Types

- An **Abstract Data Type** (ADT) specifies a set of operations through which a certain data structure (e.g. arrays, lists, trees, graphs) can be accessed
- ADTs are mathematical abstractions (implementation-free)
- Their purpose is to allow you to declare your intentions
- You are entering into an agreement that you only intend to use the underlying data structure in the way specified by the ADT



Say it with an ADT

- Common ADTs include stacks, queues, priority queues, sets, multisets and maps
- There are many possible implementations of these ADTs (some far from obvious)
- Each ADT has a limited set of operations associated with it
- They are an abstraction away from implementation
- By declaring your intentions you are making your code easier to understand and maintain

Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



Stacks

- Last In First Out (LIFO) memory

Item 4
Item 3
Item 2
Item 1

Stacks

- Last In First Out (LIFO) memory
- Standard operations

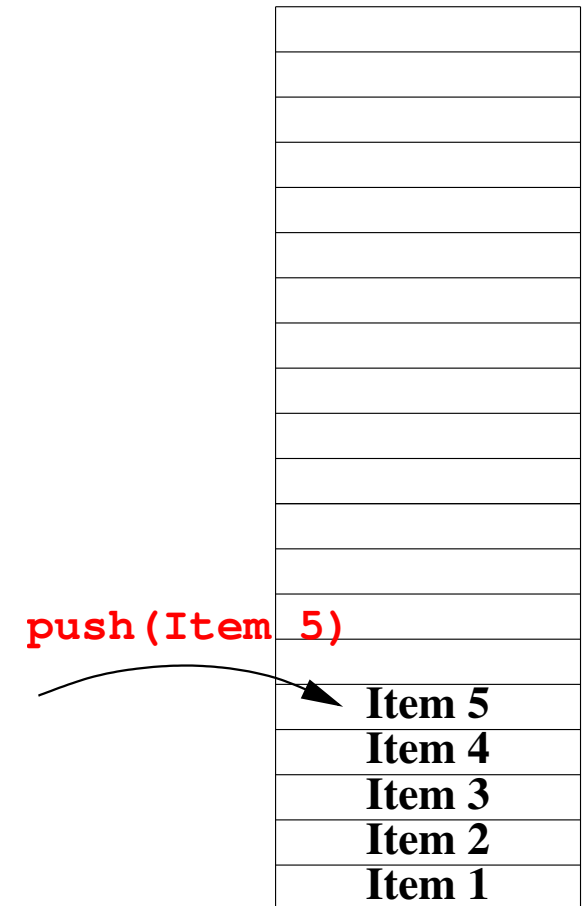
Item 4
Item 3
Item 2
Item 1

Stacks

- Last In First Out (LIFO) memory

- Standard operations

★ `push(item)`



Stacks

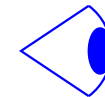
- Last In First Out (LIFO) memory

- Standard operations

- ★ push(item)

★ T peek ()

peek ()



Item 5
Item 4
Item 3
Item 2
Item 1

Stacks

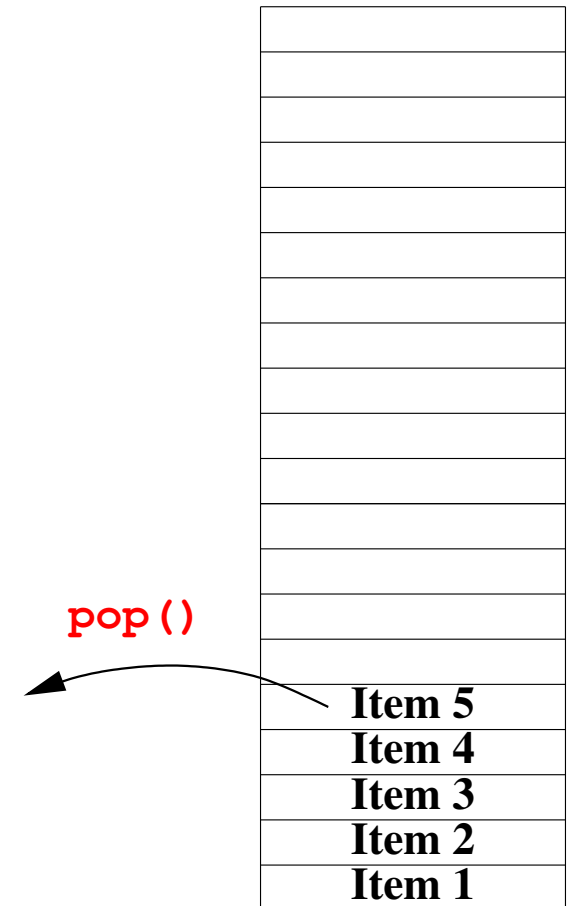
- Last In First Out (LIFO) memory

- Standard operations

★ `push(item)`

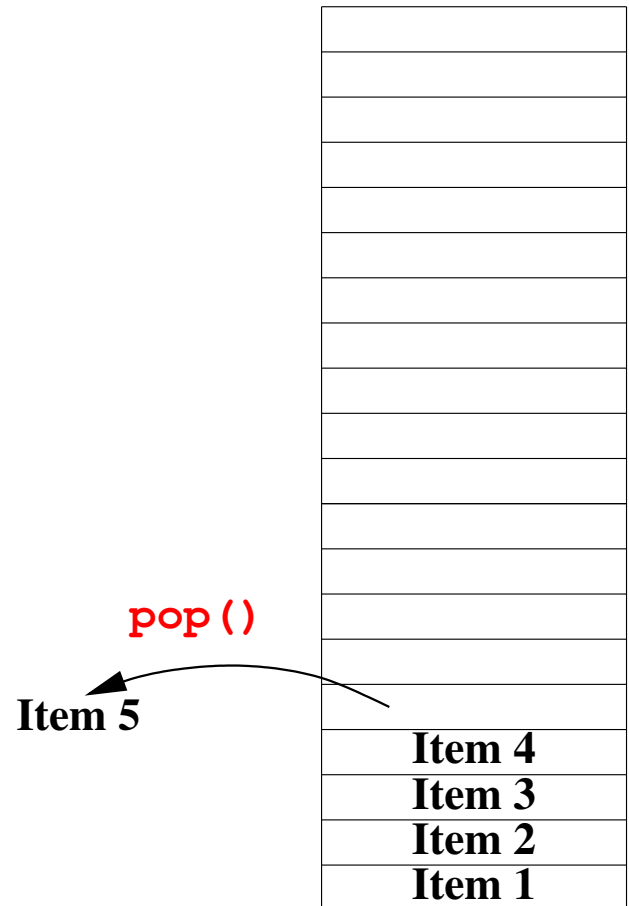
★ `T peek()`

★ `T pop()`



Stacks

- Last In First Out (LIFO) memory
- Standard operations
 - ★ `push(item)`
 - ★ `T peek()`
 - ★ `T pop()`



Stacks

- Last In First Out (LIFO) memory

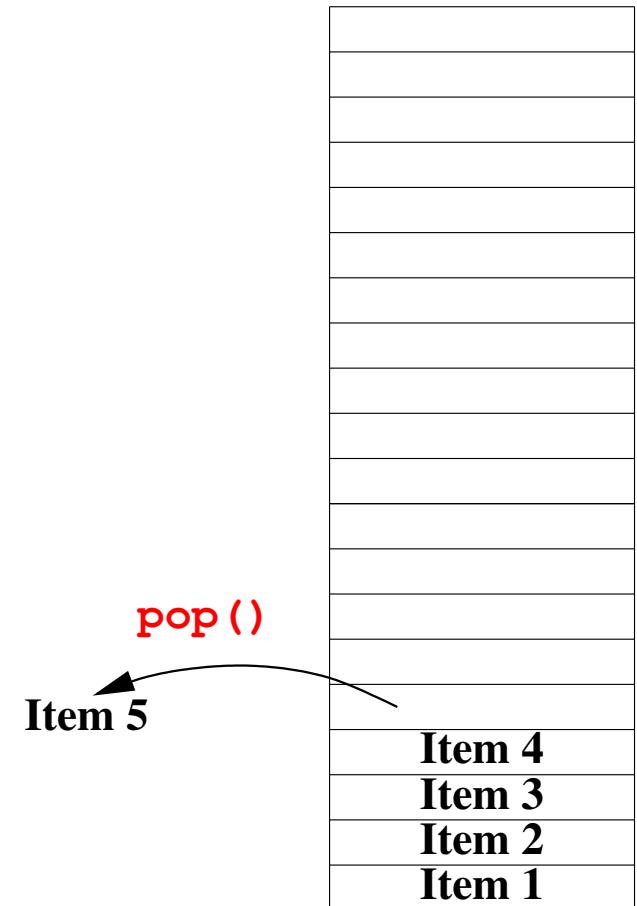
- Standard operations

★ `push(item)`

★ `T peek()`

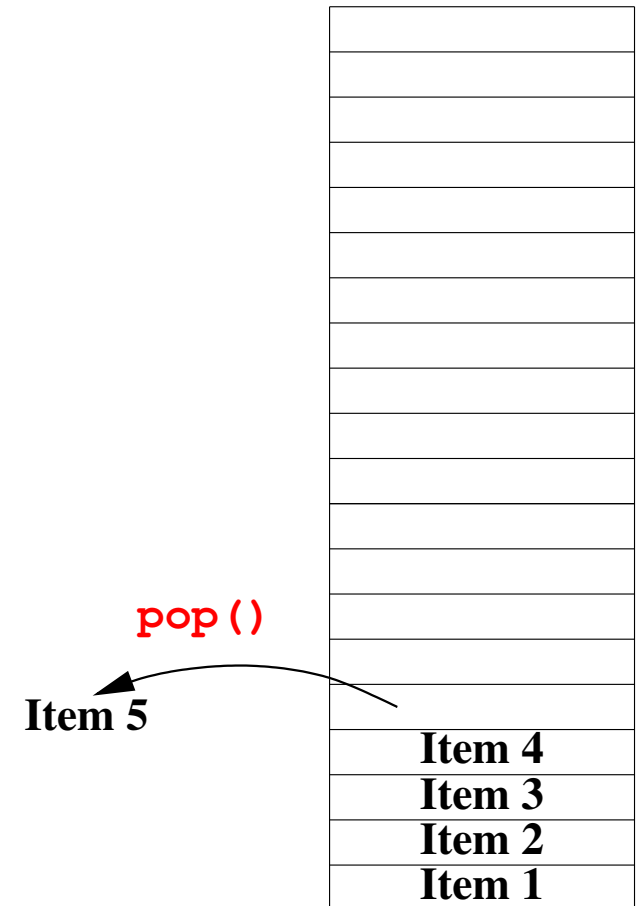
★ `T pop()`

★ **`boolean isEmpty()`**



Stacks

- Last In First Out (LIFO) memory
- Standard operations
 - ★ `push(item)`
 - ★ `T peek()`
 - ★ `T pop()`
 - ★ **boolean** `isEmpty()`
- Implemented using an array or a linked list
 - ★ **Complexity of above operations using the array implementation?**



Why Use a Stack?

- Gives you a very simple interface
- Reduces the access to memory – no longer random access !
 - ★ Seems counter intuitive to reduce what you can do . . .
 - ★ . . . but prevents another programmer from using memory in a way that will break existing code
- Sufficient for large number of algorithms

Uses of Stacks

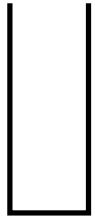
- Reversing an array
- Parsing expressions for compilers
 - ★ balancing parentheses
 - ★ matching XML tags
 - ★ evaluating arithmetic expressions
- Clustering algorithm

Evaluating Arithmetic Expressions

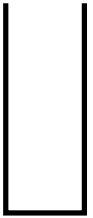
$((7+3)/5)*(7-5)$

Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$



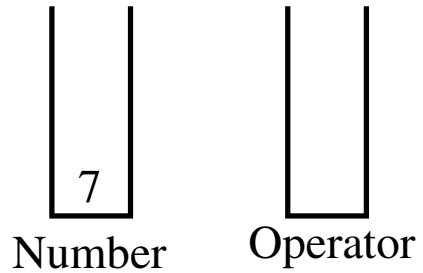
Number



Operator

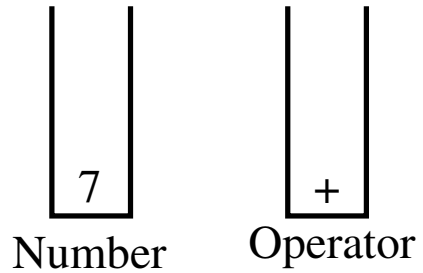
Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$



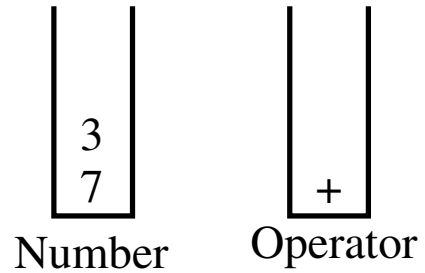
Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$



Evaluating Arithmetic Expressions

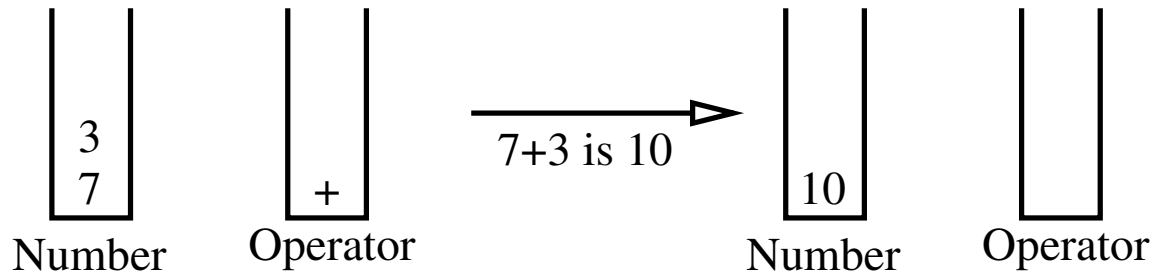
$((7+3)/5)*(7-5)$



Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$

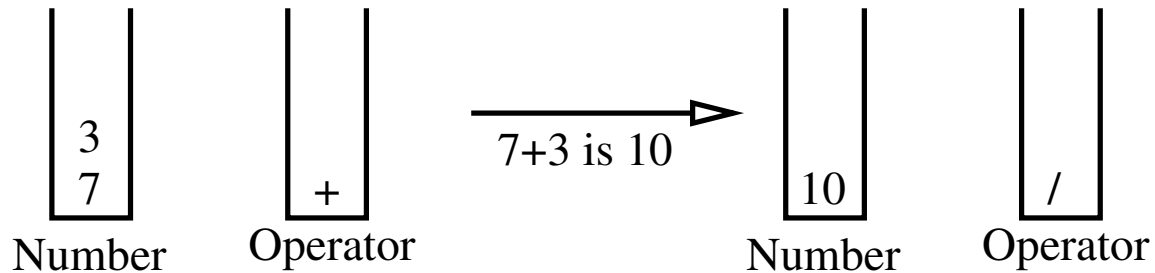
$X/5)*(7-5)$



Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$

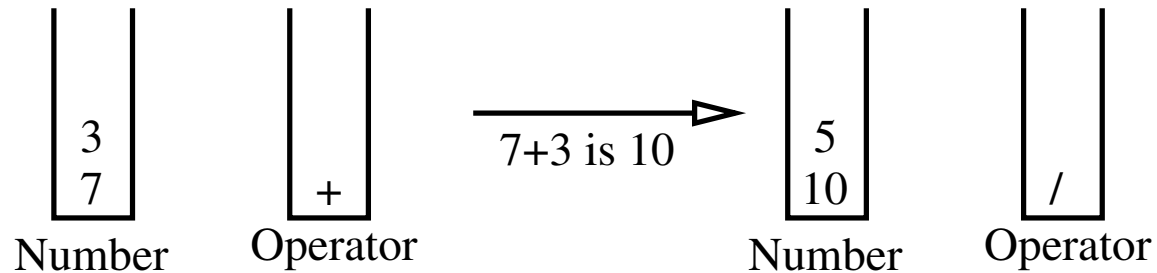
$X/5)*(7-5)$



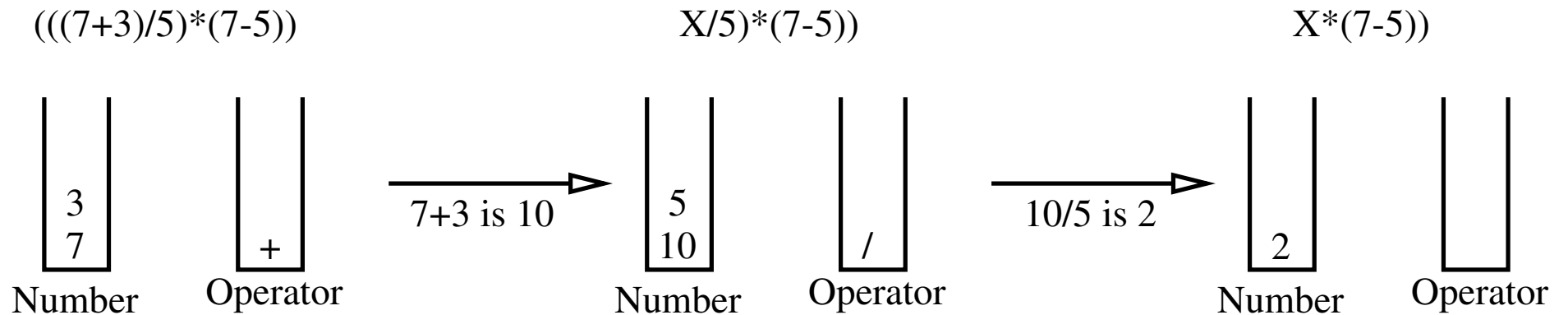
Evaluating Arithmetic Expressions

$((7+3)/5)*(7-5)$

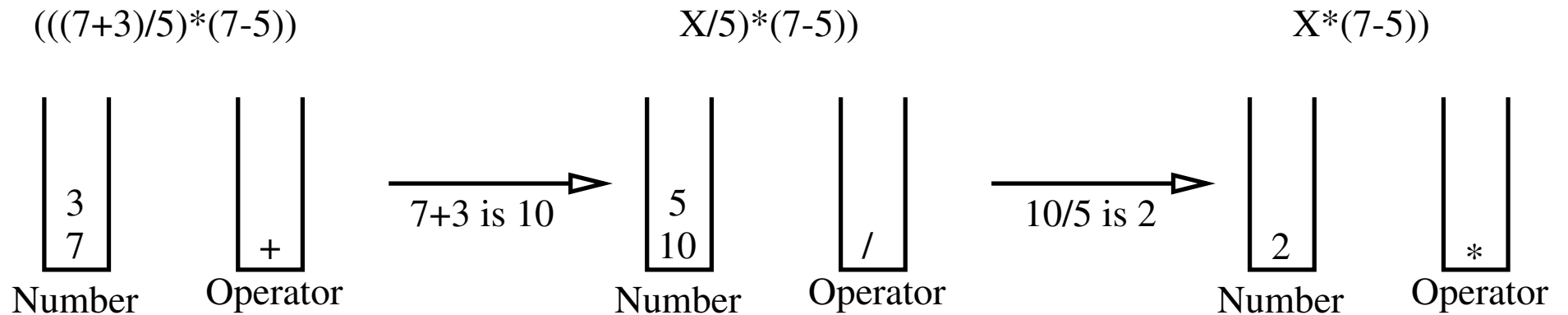
$X/5)*(7-5)$



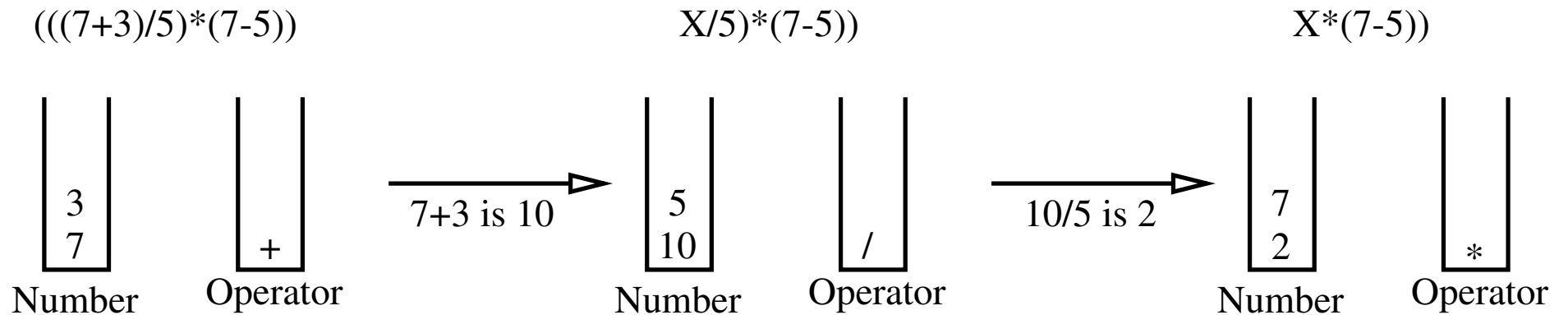
Evaluating Arithmetic Expressions



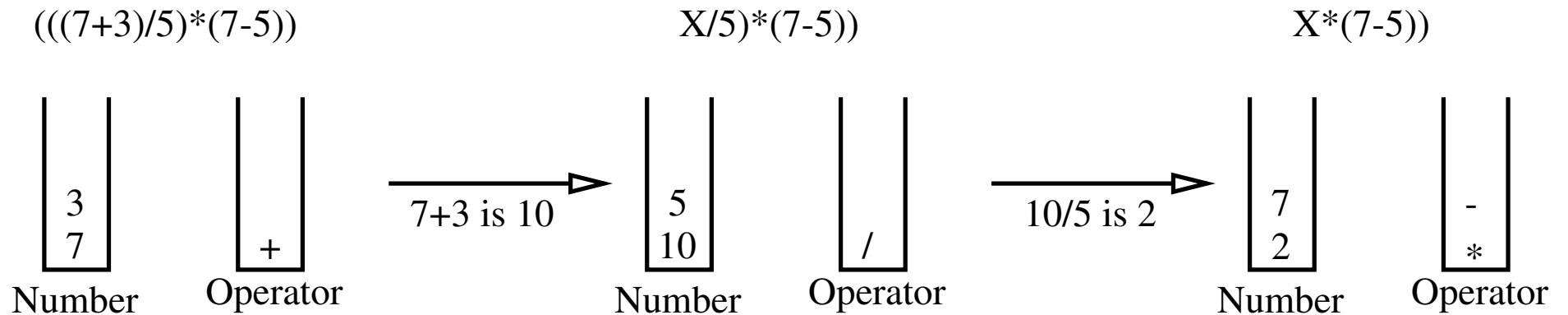
Evaluating Arithmetic Expressions



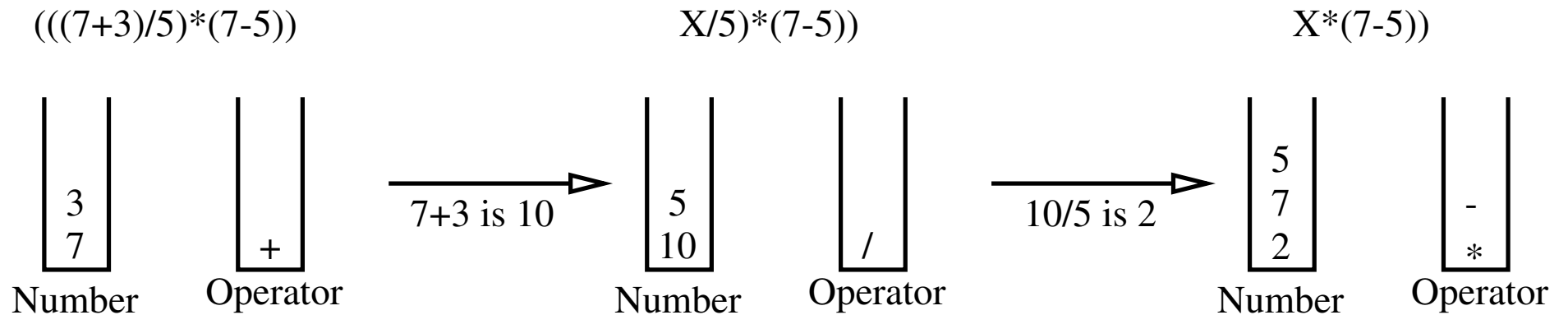
Evaluating Arithmetic Expressions



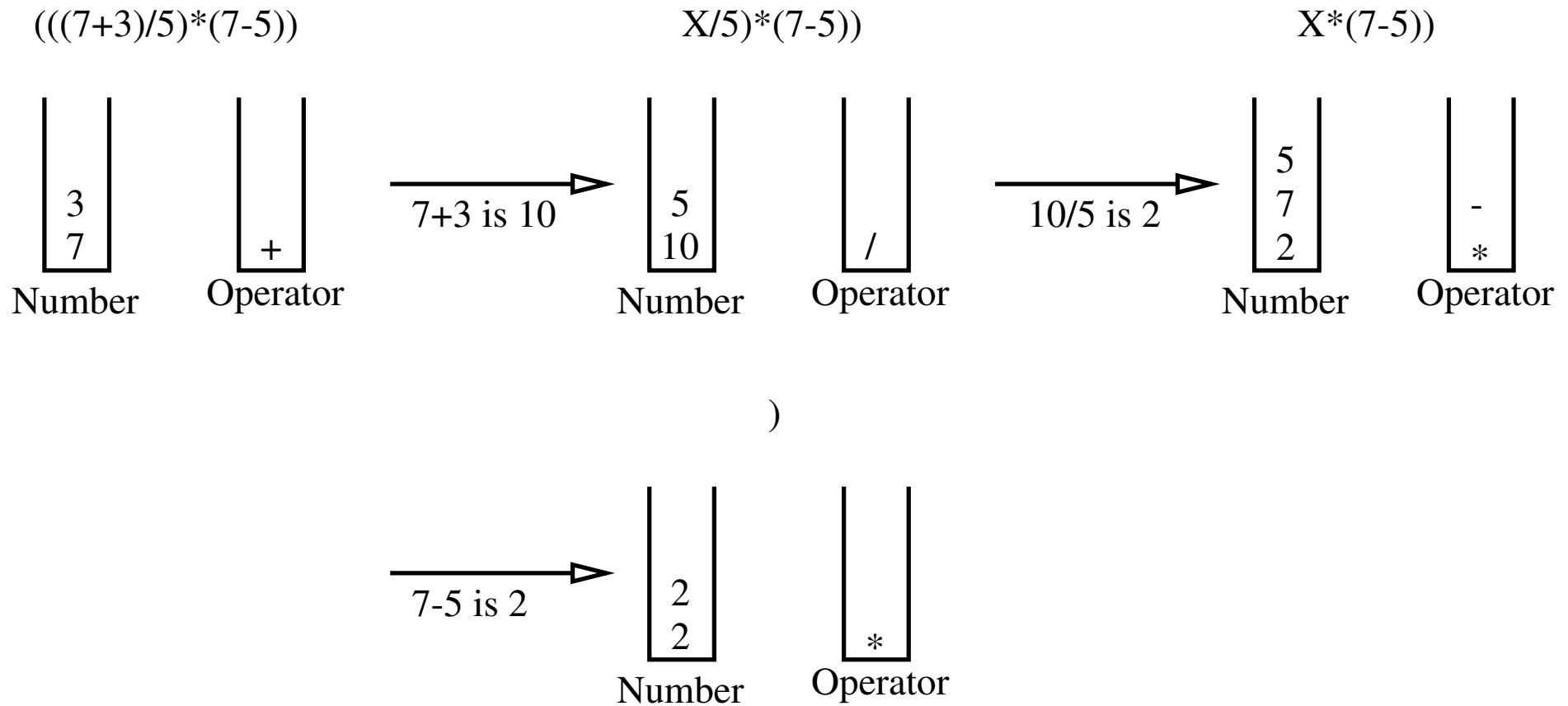
Evaluating Arithmetic Expressions



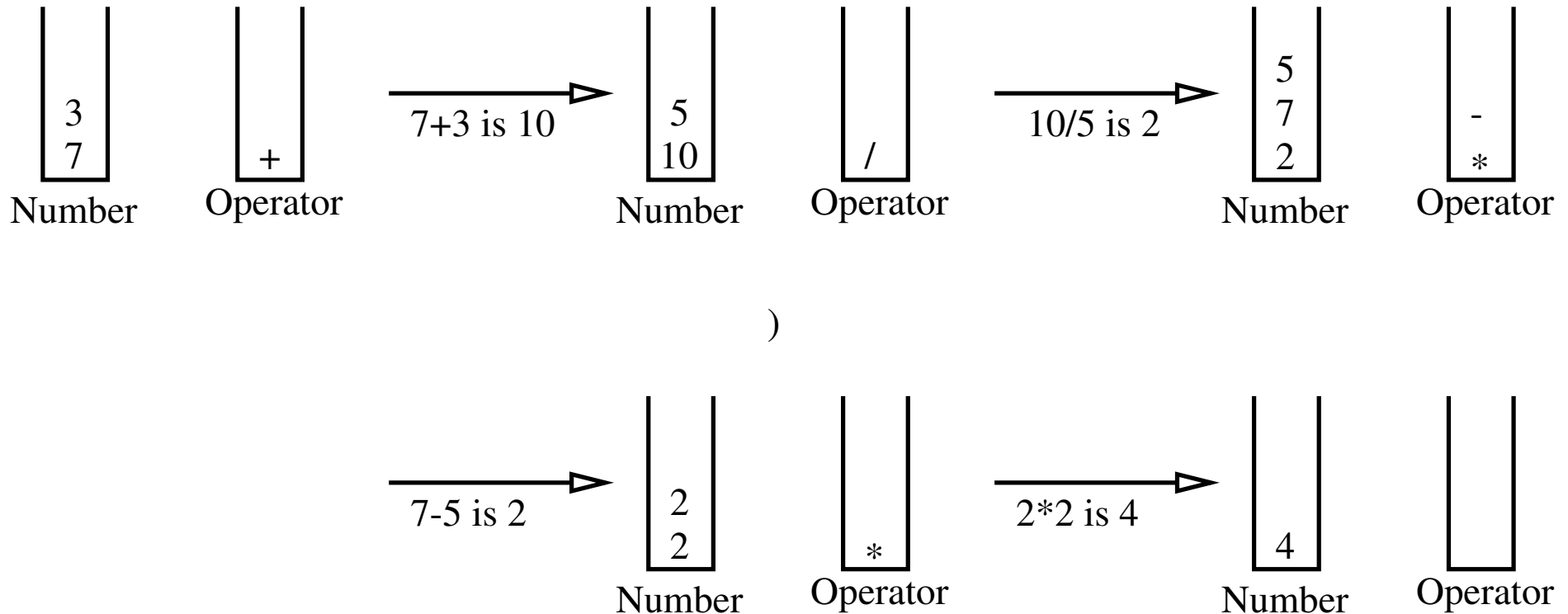
Evaluating Arithmetic Expressions



Evaluating Arithmetic Expressions



Evaluating Arithmetic Expressions

$$(((7+3)/5)*(7-5))$$
$$X/5)^*(7-5))$$
$$X^*(7-5))$$


Java Stacks

- In Java the class `Stack<T>` implements `push`, `pop`, `peek`, `empty` and `search`
- It extends the `Vector` class

Java Stacks

- In Java the class `Stack<T>` implements `push`, `pop`, `peek`, `empty` and `search`
- It extends the `Vector` class
- **Yes, it's a mess !**
- Why `search`?
- A `stack` isn't a type of `vector` – it could be implemented by a `Vector` but it should not extend `Vector` !

Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



Queues

- First-in-first-out (FIFO) memory model



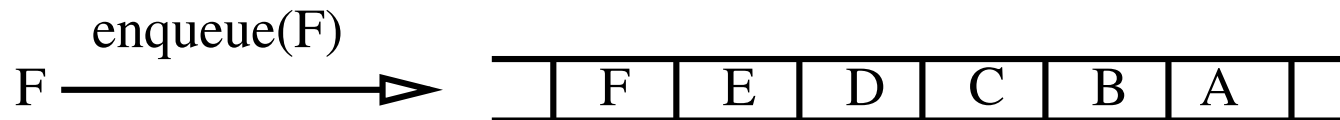
Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`
- `peek()`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`
- `peek()`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue (elem)`
- `peek ()`
- `dequeue ()`



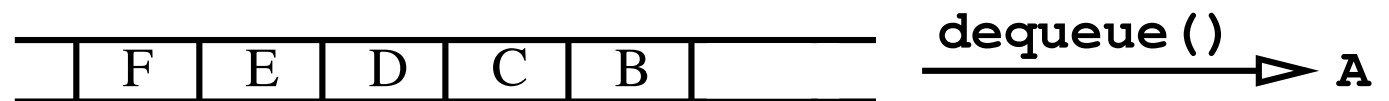
Queues

- First-in-first-out (FIFO) memory model
- `enqueue (elem)`
- `peek ()`
- `dequeue ()`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue (elem)`
- `peek ()`
- `dequeue ()`



Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`
- `peek()`
- `dequeue()`
- `isEmpty()`



Uses of Queues

- In operating systems
 - ★ print queues
 - ★ job queues
- Communication/Message passing

Implementation of Queues

- Either using arrays or using linked lists
- Java has a `Queue<T>` interface with all the wrong names
 - ★ `add(elem)` or `offer(elem)` instead of enqueue
 - ★ `remove()` or `poll()` instead of dequeue
 - ★ `element()` and `peek()` to peek
- Java 6 adds
 - ★ Double ended queues, `Deque` (allow add/remove at both ends)
 - ★ `BlockingQueue` and `BlockingDeque` that supports concurrency

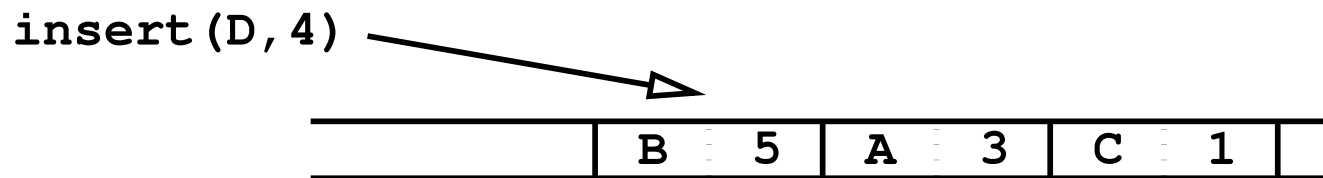
Priority Queues

- Queue with priorities

	B : 5	A : 3	C : 1	
--	---------------------	---------------------	---------------------	--

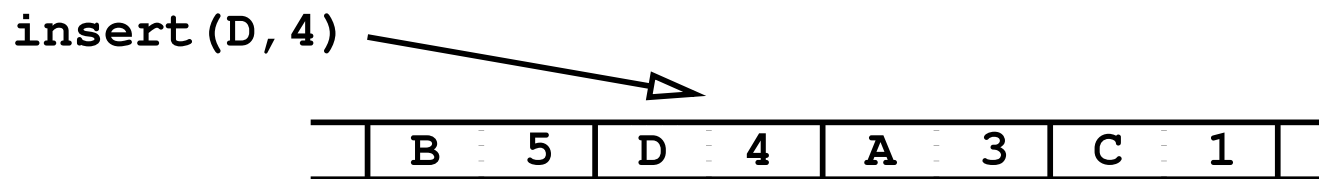
Priority Queues

- Queue with priorities
- `insert(elem, priority)`



Priority Queues

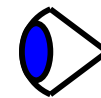
- Queue with priorities
- `insert(elem, priority)`



Priority Queues

- Queue with priorities
- `insert(elem, priority)`
- `findMin()`

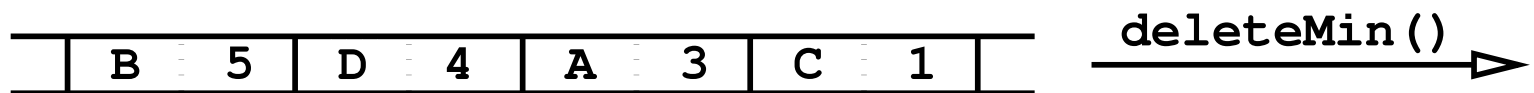
	B	5	D	4	A	3	C	1	
--	---	---	---	---	---	---	---	---	--



`findMin()`

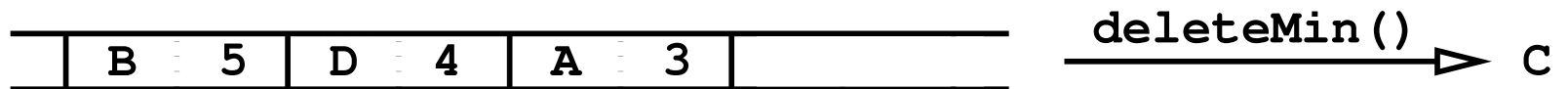
Priority Queues

- Queue with priorities
- `insert(elem, priority)`
- `findMin()`
- `deleteMin()`



Priority Queues

- Queue with priorities
- `insert(elem, priority)`
- `findMin()`
- `deleteMin()`



Uses of Priority Queues

- Queues with priorities (e.g. which threads should run)
- Often used in “greedy algorithms”
 - ★ Prim’s minimum spanning tree algorithm

Implementation of Priority Queues

- Could be implemented using a linked list or a binary tree
- Most efficient implementation uses a **heap** (binary tree implemented using an array)

Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



Lists

- A **list** is a collection where the order of elements is important
- Repetitions are allowed
- You can `add` (at a specified position), `remove` (from a specified position), `set` (at a specified position) and, `get` elements
- Java has a `List<T>` interface
- Java provides specific implementations as `ArrayList<T>` and `LinkedList<T>`

Sets

- Models mathematical sets
 - ★ no ordering or repetitions
- operations: `add`, `remove`, `contains`, `size`, `isEmpty`
- Java has a `Set<T>` interface
- Two common implementations of sets are
 - ★ `HashSet`s using hash tables (fast access)
 - ★ `TreeSet`s using binary trees
- Which is most efficient depends on the application

Maps

- A map provides a content addressable memory for pairs *key:data*
- It provides fast access to the data through the key
 - ★ no duplicate keys!
- Implemented using trees or hash tables
- Multimaps allow each key to be associated with multiple values

Program to Interfaces not Implementations

- Use data structures through their interfaces (ADT)

```
Set<String> myset = new HashSet<String>();  
List<Integer> mylist = new ArrayList<Integer>();%
```

- Much better than

```
HashSet<String> myset = new HashSet<String>();  
ArrayList<Integer> mylist = new ArrayList<Integer>();
```

which is implementation dependent and can make modification and maintenance hard

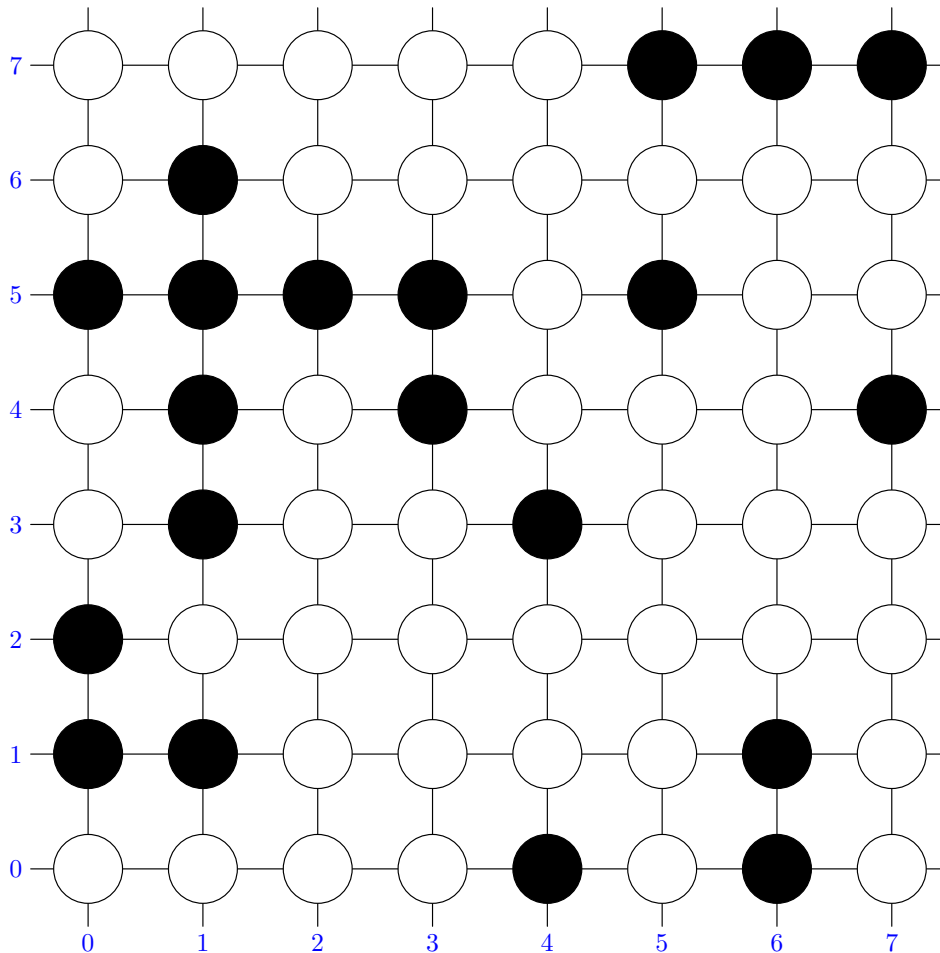
- Learn to program to interfaces not implementations
- Declare your intentions not your actions

Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



Clustering



- A frequent problem is to find clusters of connected cells
- Applications in computer vision, computer go, graph connectedness, . . .

Assume a `Graph` interface with methods `getNeighbours(Node n)`, `isOccupied(Node n)`.

Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph)
```

Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
}
```

Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
  
    uncheckedNodes.push(startNode);  
    clusterNodes.add(startNode);
```

Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
  
    uncheckedNodes.push(startNode);  
    clusterNodes.add(startNode);  
  
    while (!uncheckedNodes.isEmpty()) {
```


Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
  
    uncheckedNodes.push(startNode);  
    clusterNodes.add(startNode);  
  
    while (!uncheckedNodes.isEmpty()) {  
        Node next = uncheckedNodes.pop();  
        List<Node> neighbours = graph.getNeighbours(next);
```

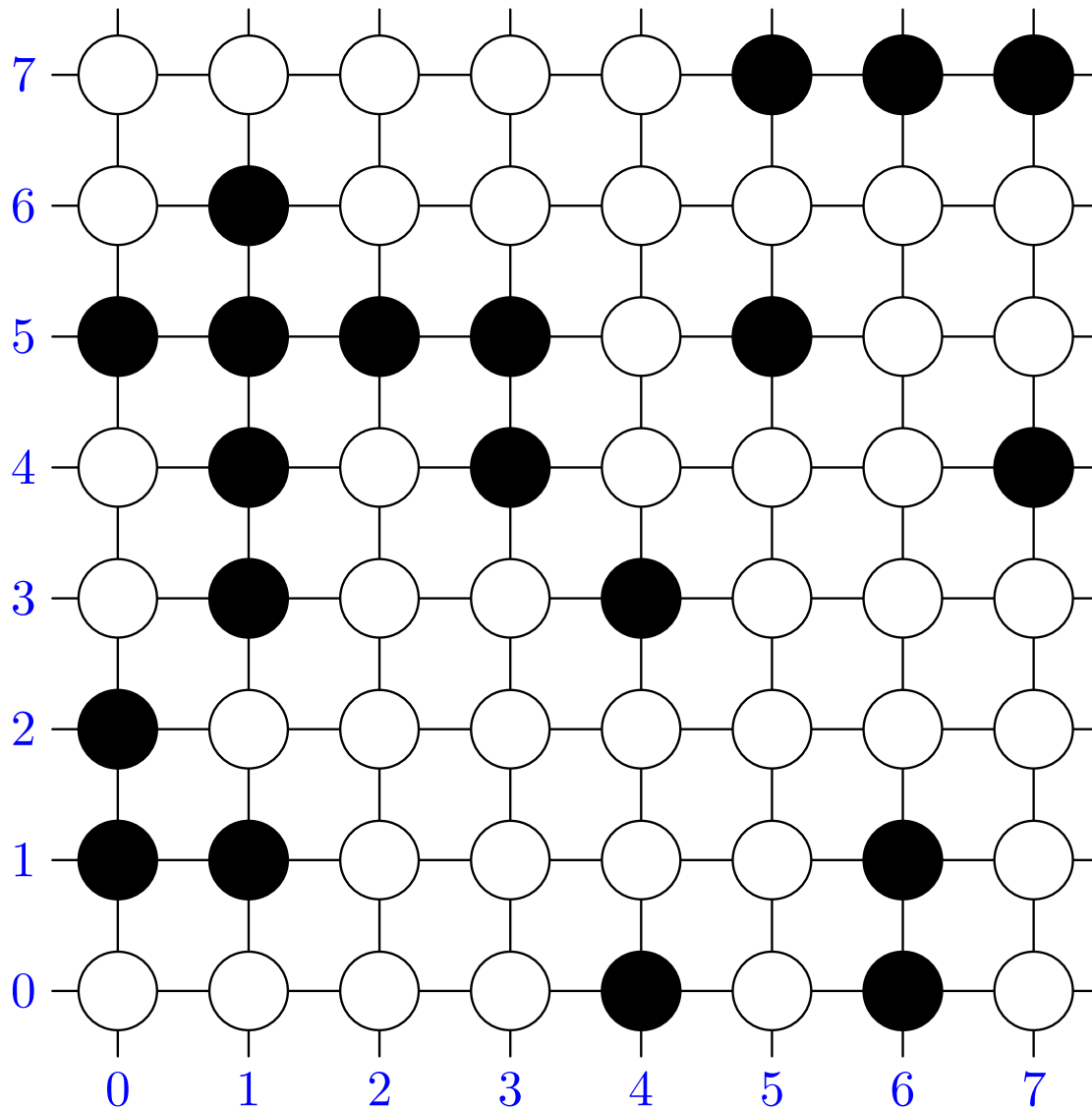
Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
  
    uncheckedNodes.push(startNode);  
    clusterNodes.add(startNode);  
  
    while (!uncheckedNodes.isEmpty()) {  
        Node next = uncheckedNodes.pop();  
        List<Node> neighbours = graph.getNeighbours(next);  
        for (Node neigh: neighbours) {  
            if (graph.isOccupied(neigh) && !clusterNodes.contains(neigh) ) {  
                uncheckedNodes.push(neigh);  
                clusterNodes.add(neigh);  
            }  
        }  
    }  
}
```

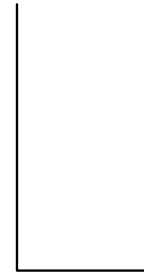
Clustering Algorithm

```
public Set<Node> findCluster(Node startNode, Graph graph) {  
    Stack<Node> uncheckedNodes = new Stack<Node>();  
    Set<Node> clusterNodes = new HashSet<Node>();  
  
    uncheckedNodes.push(startNode);  
    clusterNodes.add(startNode);  
  
    while (!uncheckedNodes.isEmpty()) {  
        Node next = uncheckedNodes.pop();  
        List<Node> neighbours = graph.getNeighbours(next);  
        for (Node neigh: neighbours) {  
            if (graph.isOccupied(neigh) && !clusterNodes.contains(neigh) ) {  
                uncheckedNodes.push(neigh);  
                clusterNodes.add(neigh);  
            }  
        }  
    }  
    return clusterNodes;  
}
```

Clustering

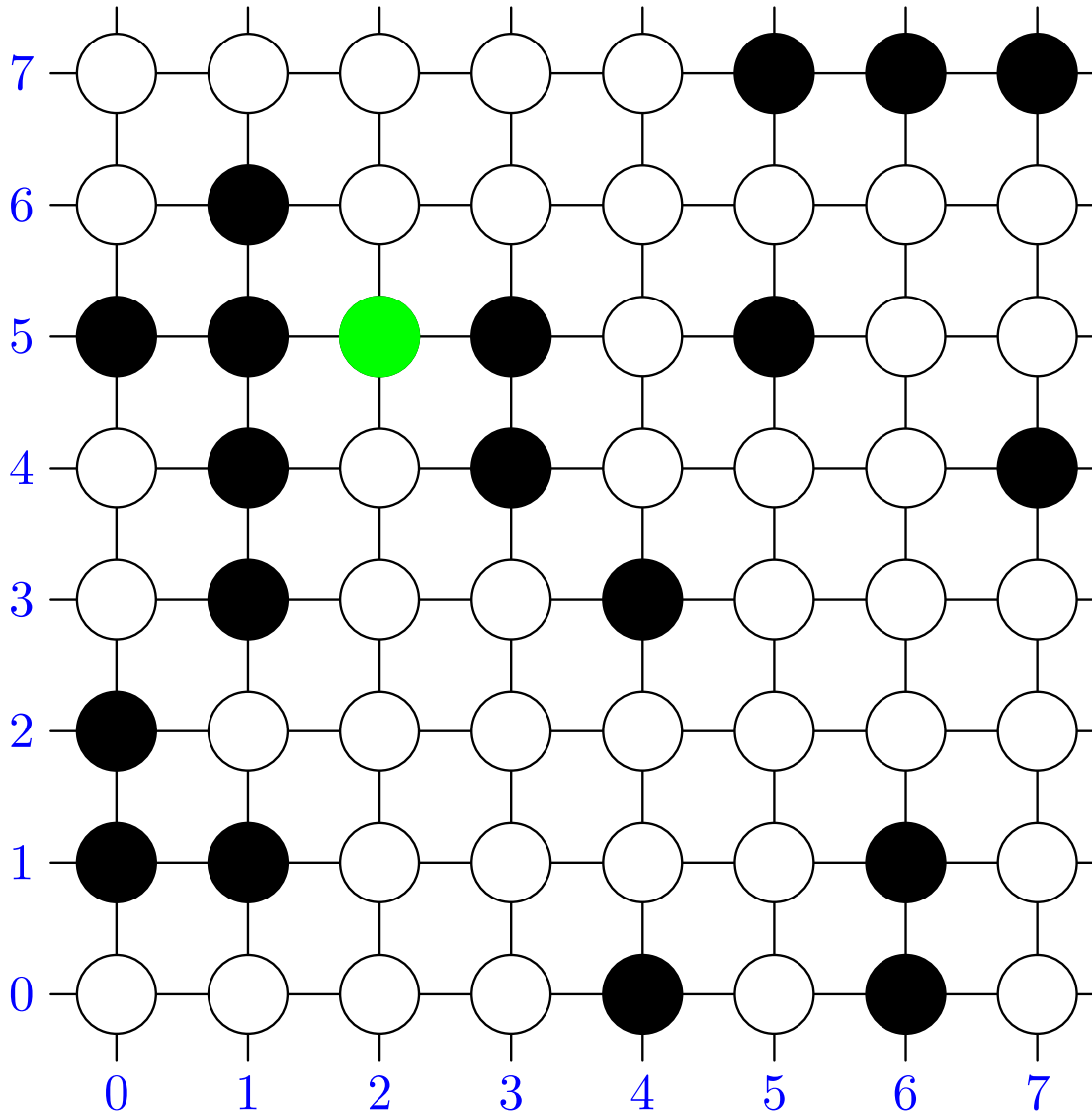


uncheckedNodes =



clusterNodes =
{ }

Clustering



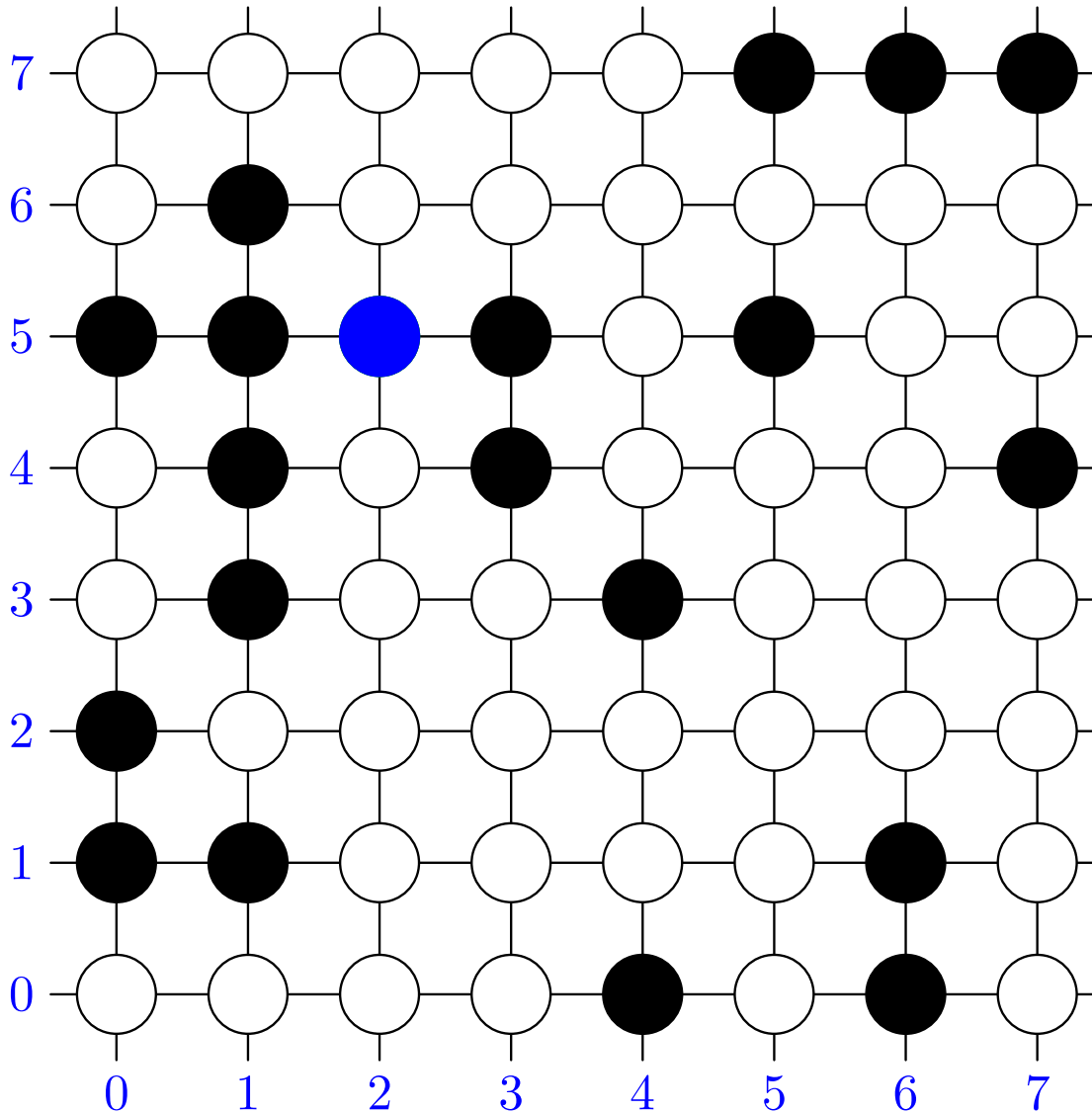
startNode = (2,5)

uncheckedNodes =

(2,5)

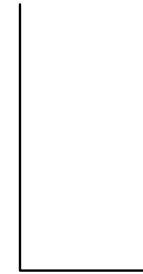
clusterNodes =
 $\{ (2,5) \}$

Clustering



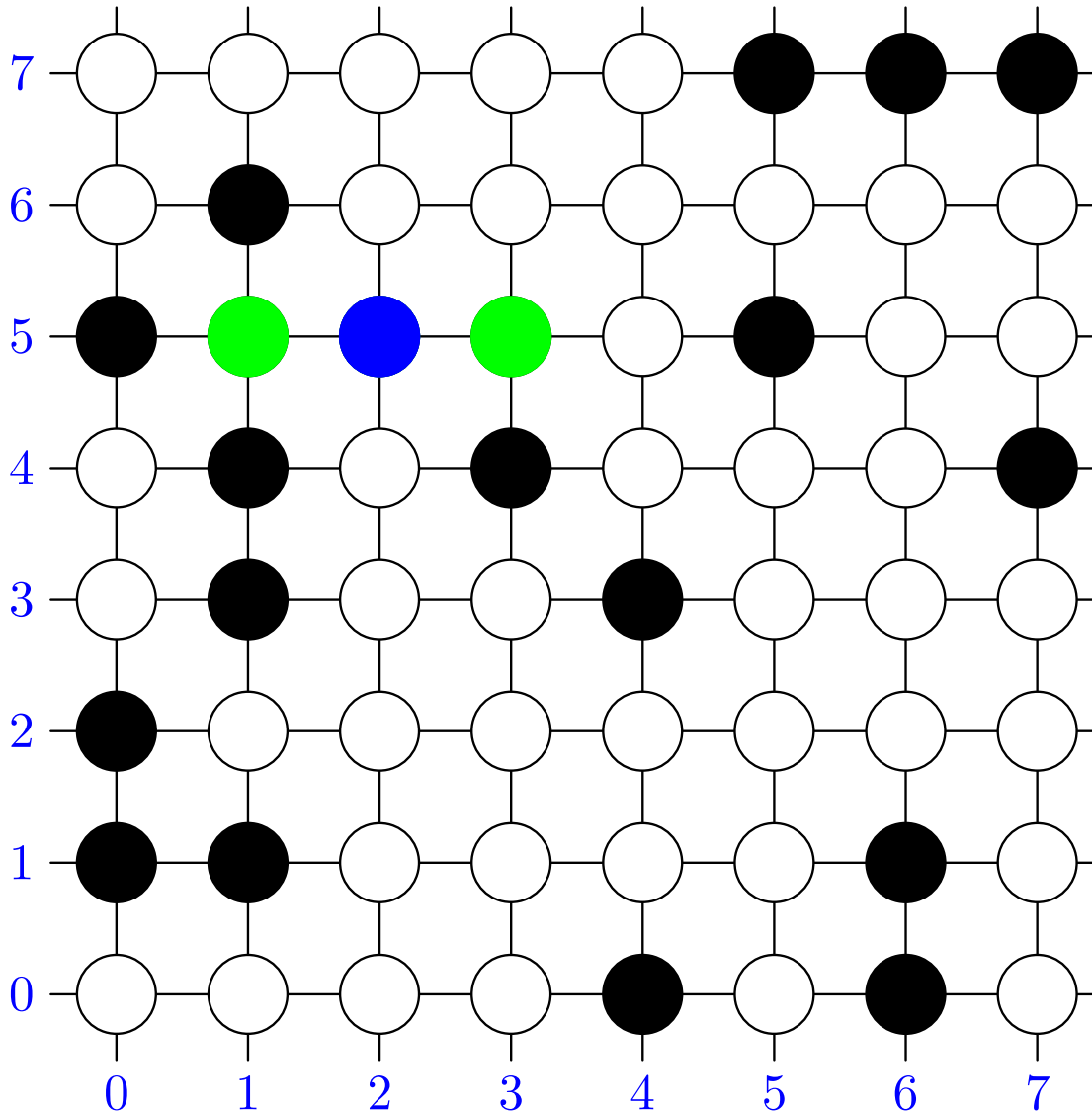
$\text{next} = (2, 5)$

`uncheckedNodes =`



`clusterNodes =`
 $\{ (2, 5) \}$

Clustering



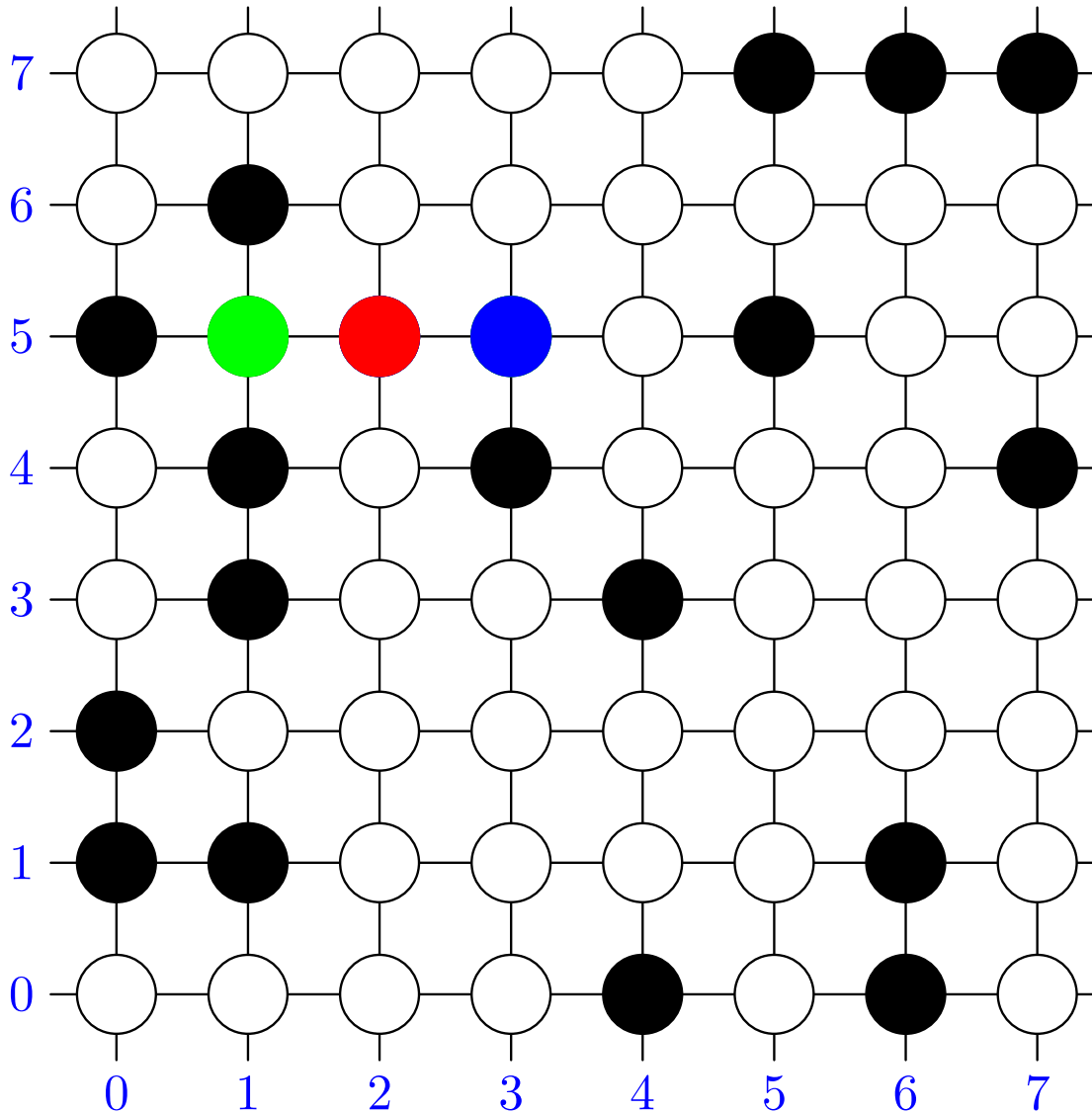
$\text{next} = (2, 5)$

$\text{uncheckedNodes} =$

$\begin{matrix} (3, 5) \\ (1, 5) \end{matrix}$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5) \}$

Clustering



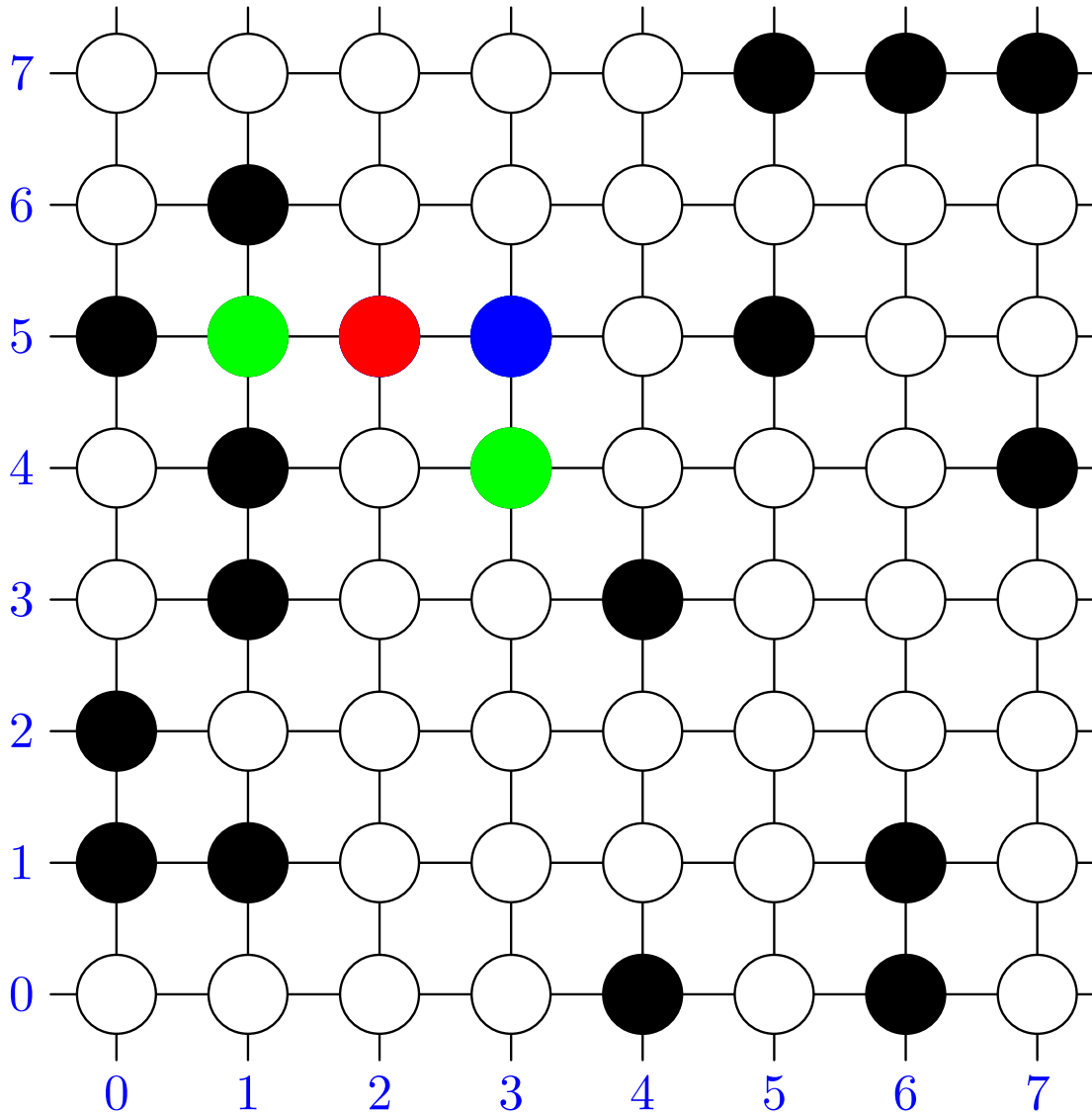
$\text{next} = (3, 5)$

$\text{uncheckedNodes} =$

$(1, 5)$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5) \}$

Clustering



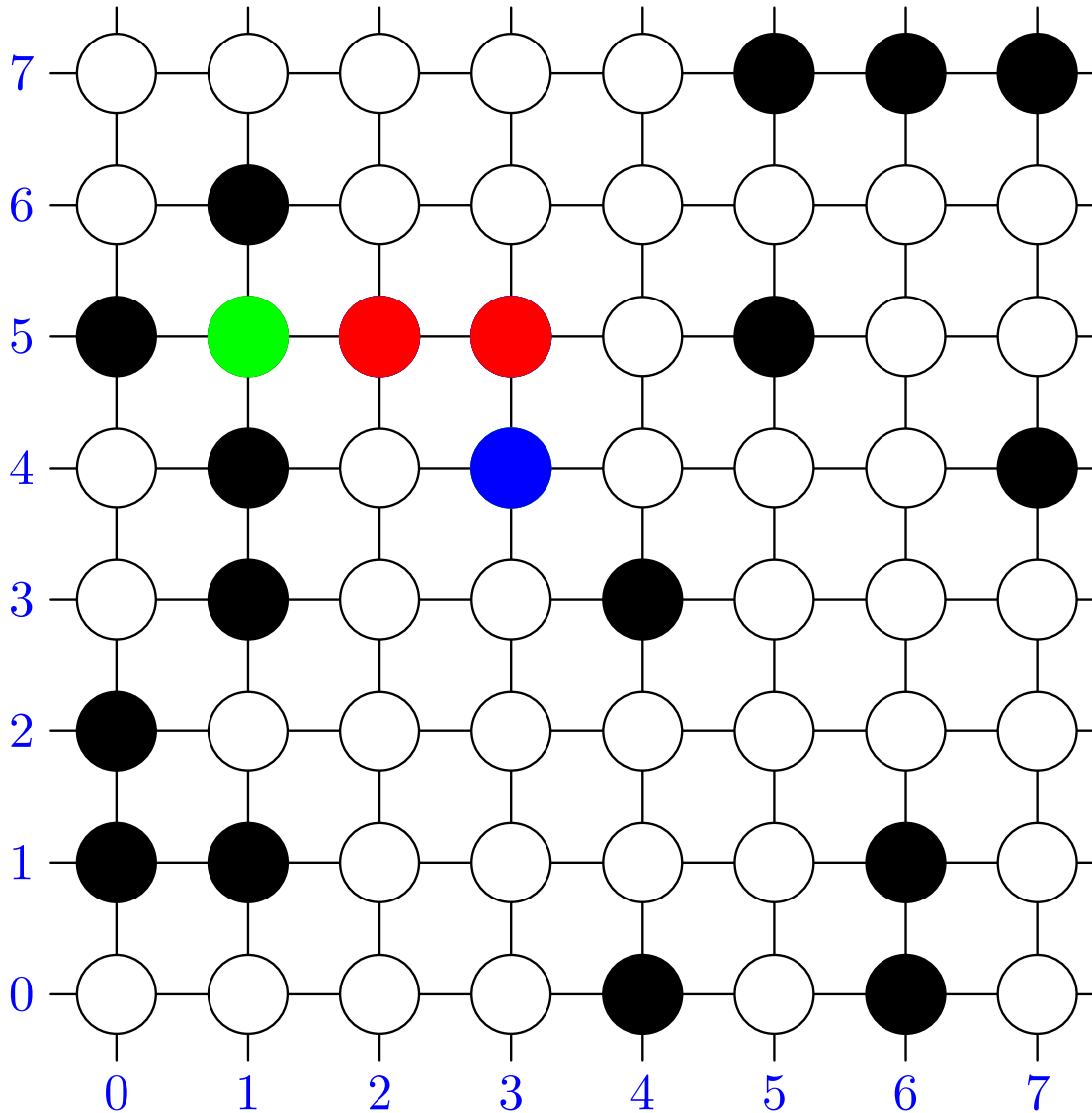
next = (3,5)

uncheckedNodes =

(3,4)
(1,5)

clusterNodes =
 $\{ (2,5), (1,5), (3,5), (3,4) \}$

Clustering



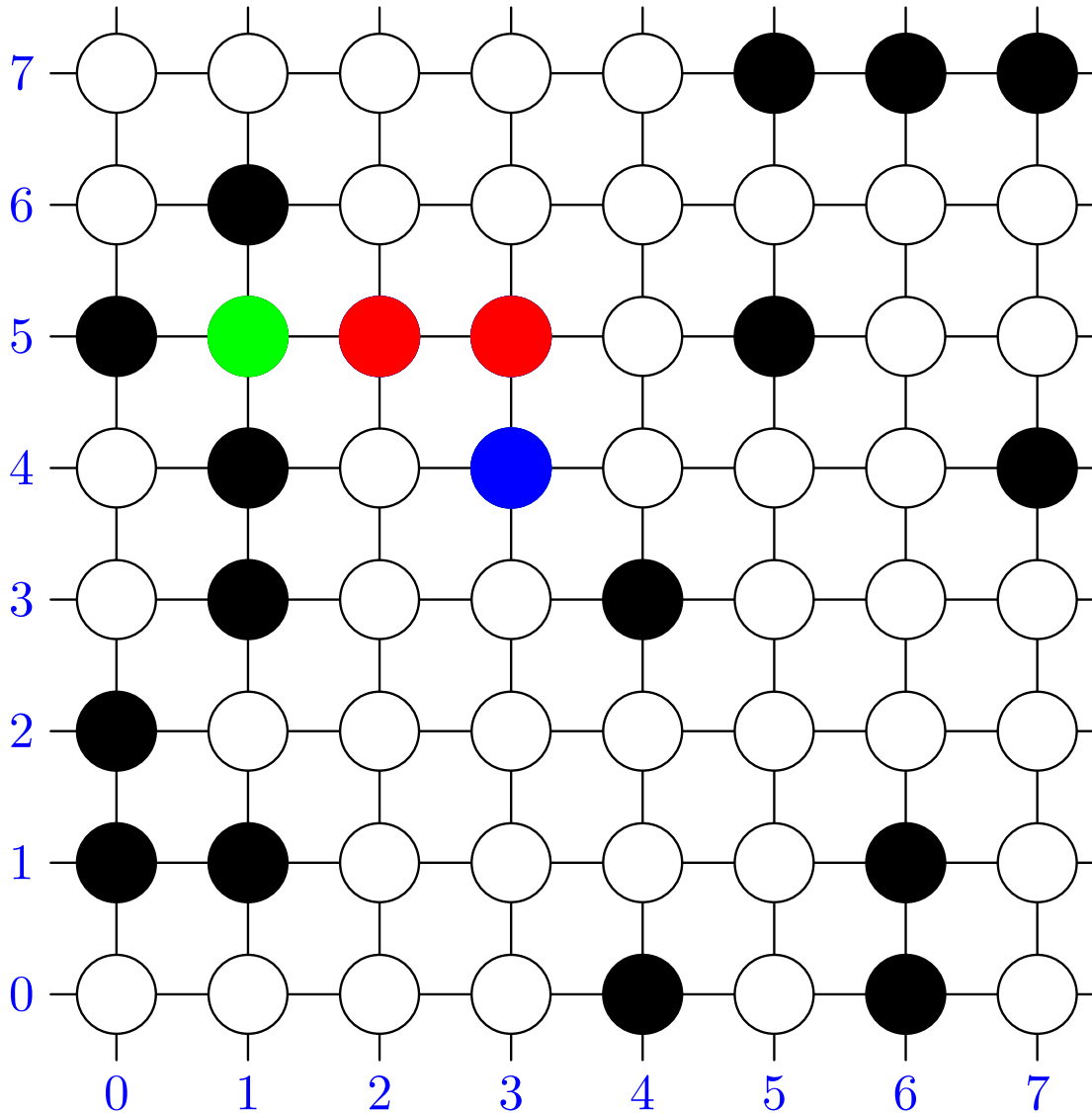
$\text{next} = (3, 4)$

$\text{uncheckedNodes} =$

$(1, 5)$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4) \}$

Clustering



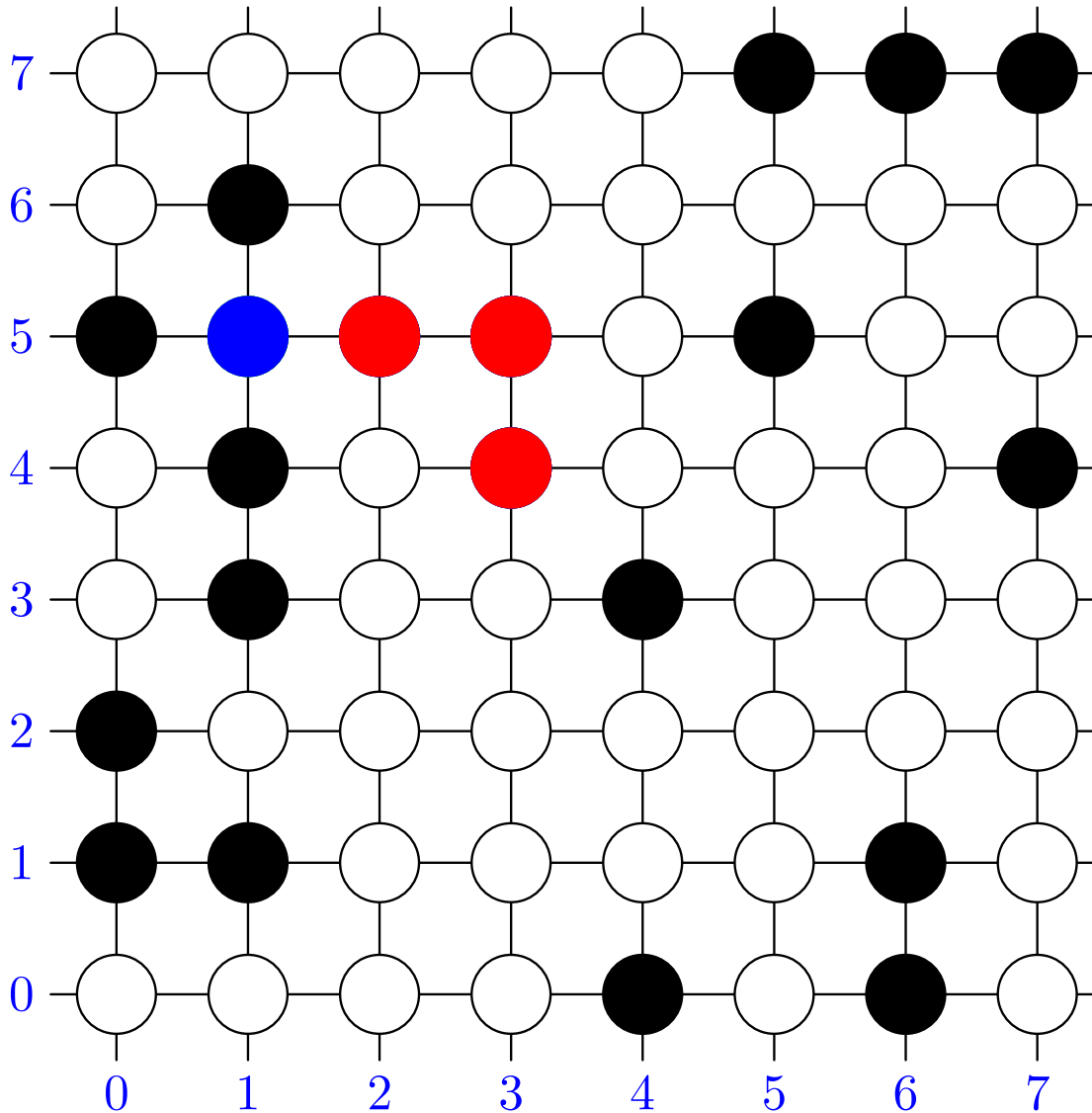
$\text{next} = (3, 4)$

$\text{uncheckedNodes} =$

$(1, 5)$

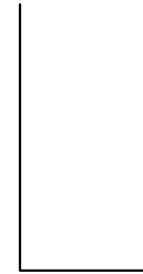
$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5), (3, 4) \}$

Clustering



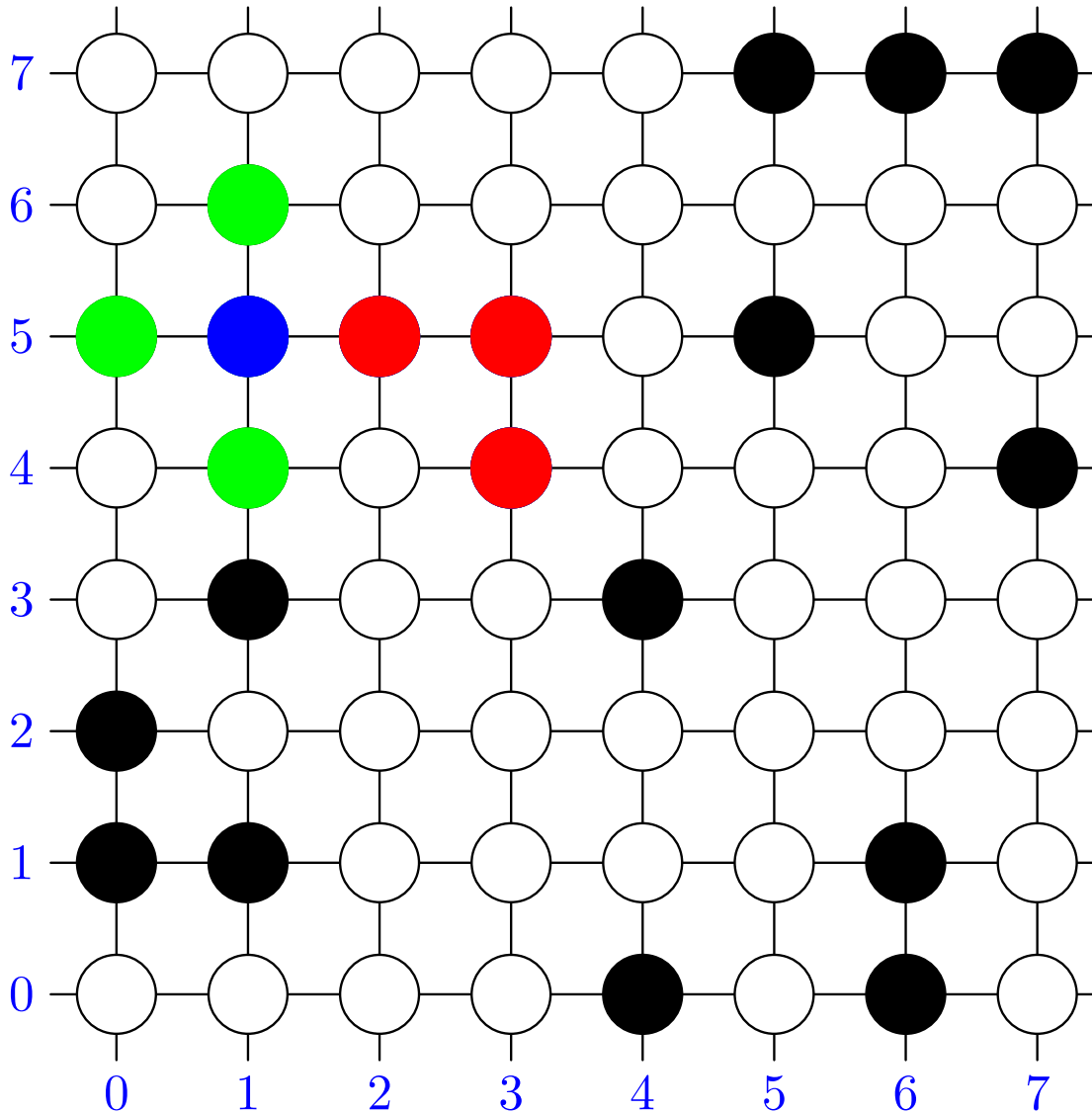
$\text{next} = (1, 5)$

$\text{uncheckedNodes} =$



$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5), (3, 4) \}$

Clustering



$\text{next} = (1, 5)$

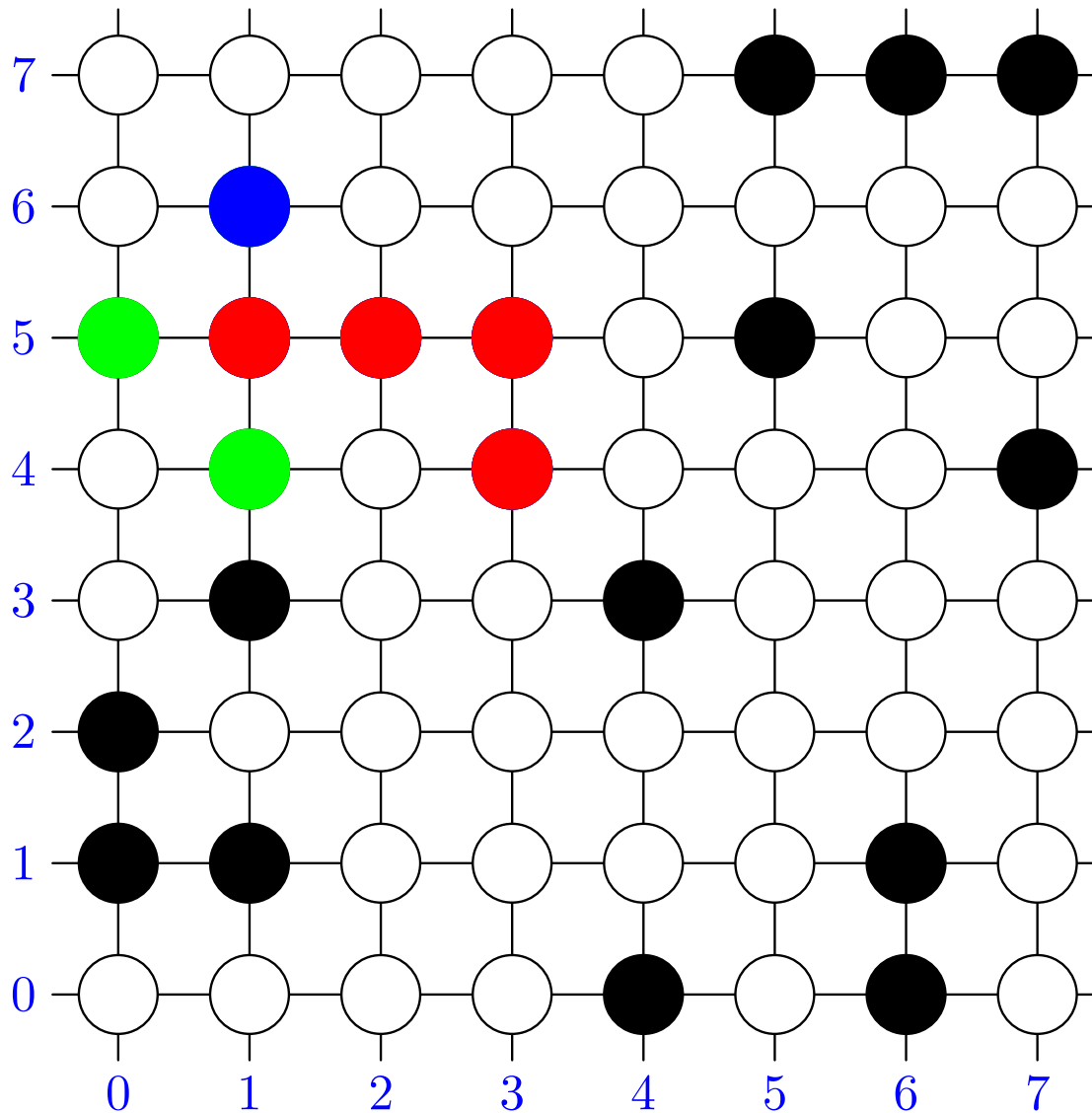
$\text{uncheckedNodes} =$

$\begin{bmatrix} (1, 6) \\ (1, 4) \\ (0, 5) \end{bmatrix}$

$\text{clusterNodes} =$

$\{ (2, 5), (1, 5), (3, 5), \\ (3, 4), (0, 5), (1, 4), \\ (1, 6) \}$

Clustering



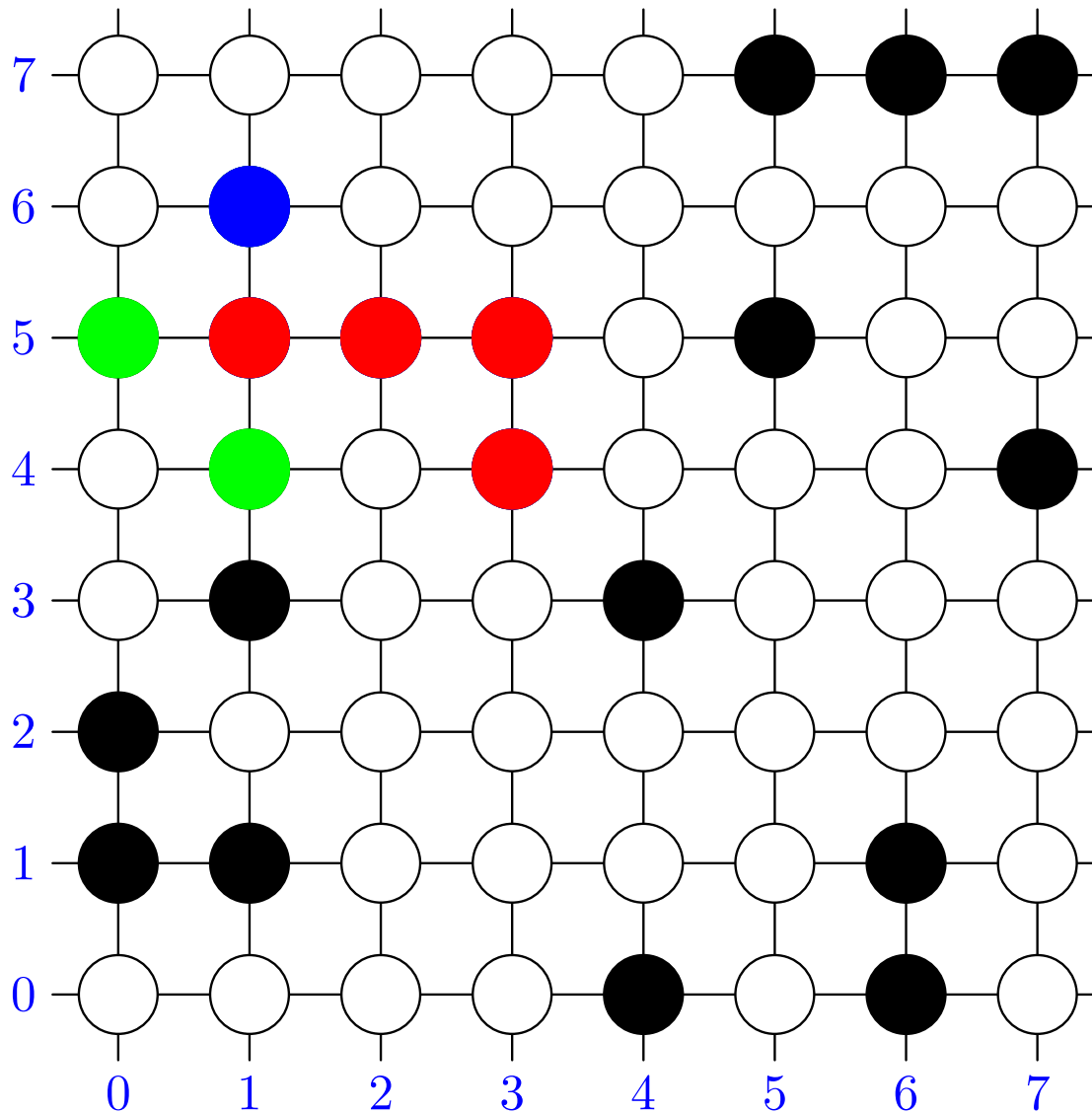
$\text{next} = (1, 6)$

$\text{uncheckedNodes} =$

$\begin{matrix} (1, 4) \\ (0, 5) \end{matrix}$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6) \}$

Clustering



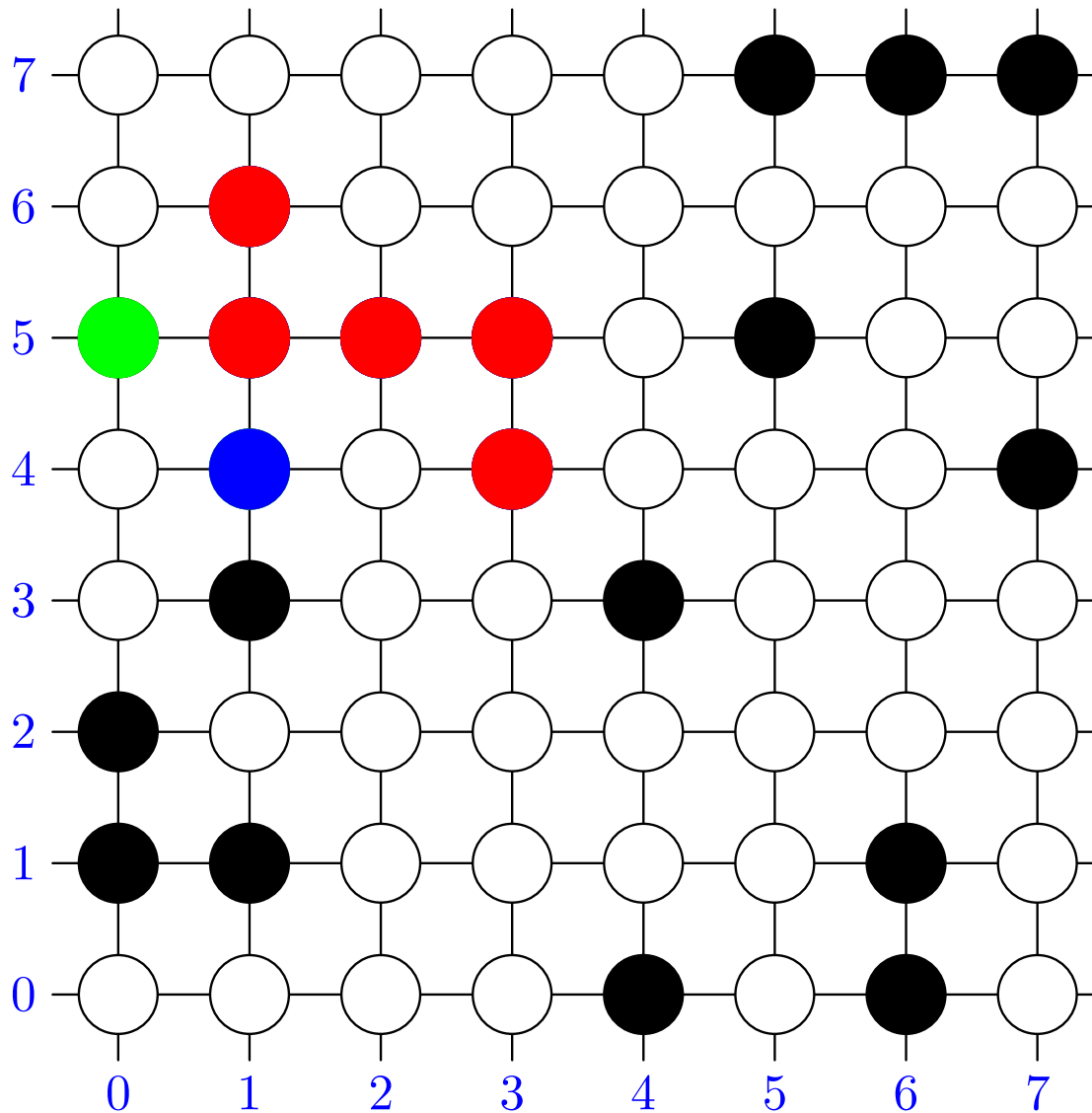
$\text{next} = (1, 6)$

$\text{uncheckedNodes} =$

$\begin{matrix} (1, 4) \\ (0, 5) \end{matrix}$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6) \}$

Clustering



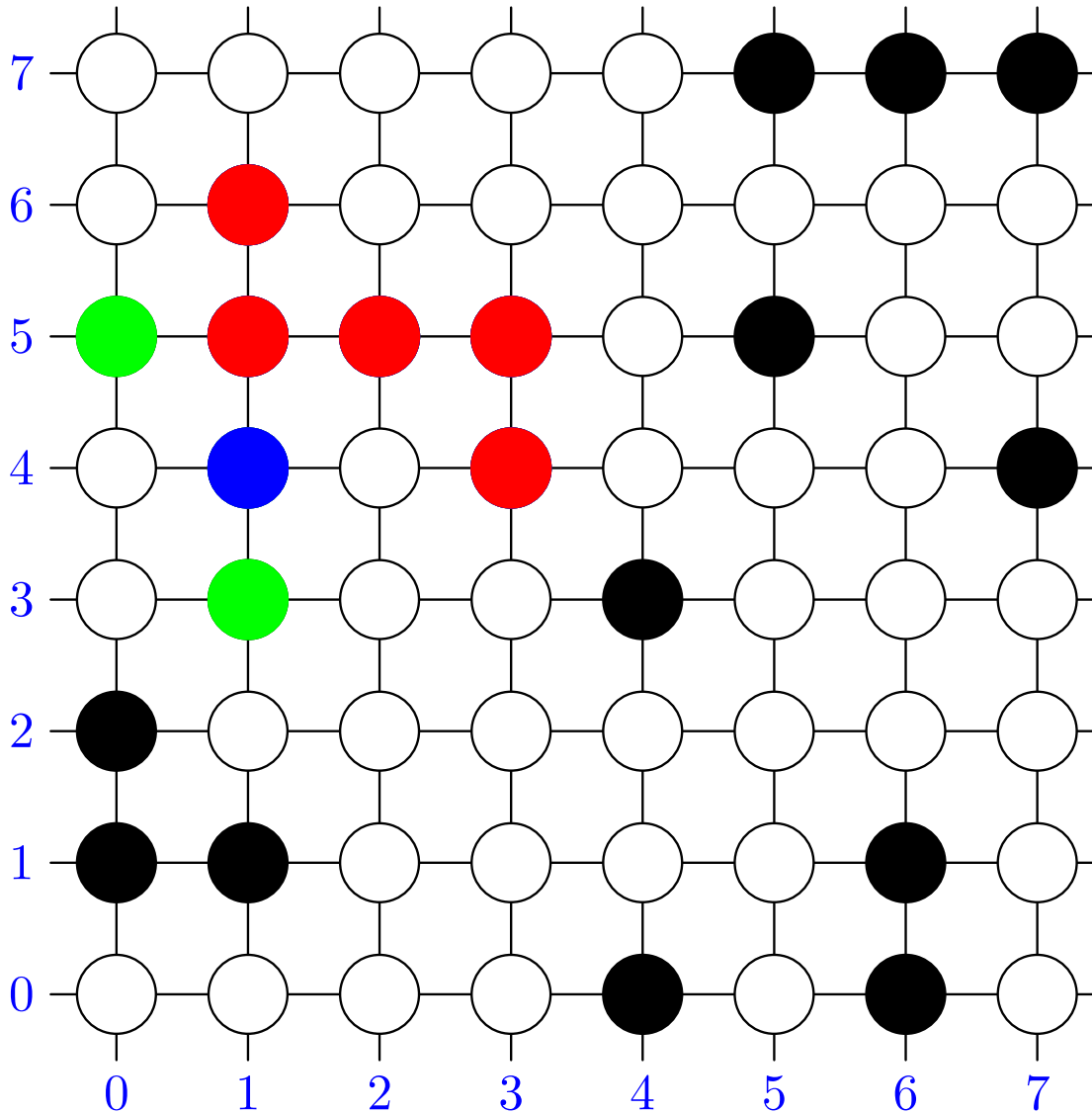
$\text{next} = (1, 4)$

$\text{uncheckedNodes} =$

$(0, 5)$

$\text{clusterNodes} =$
 $\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6) \}$

Clustering



$\text{next} = (1, 4)$

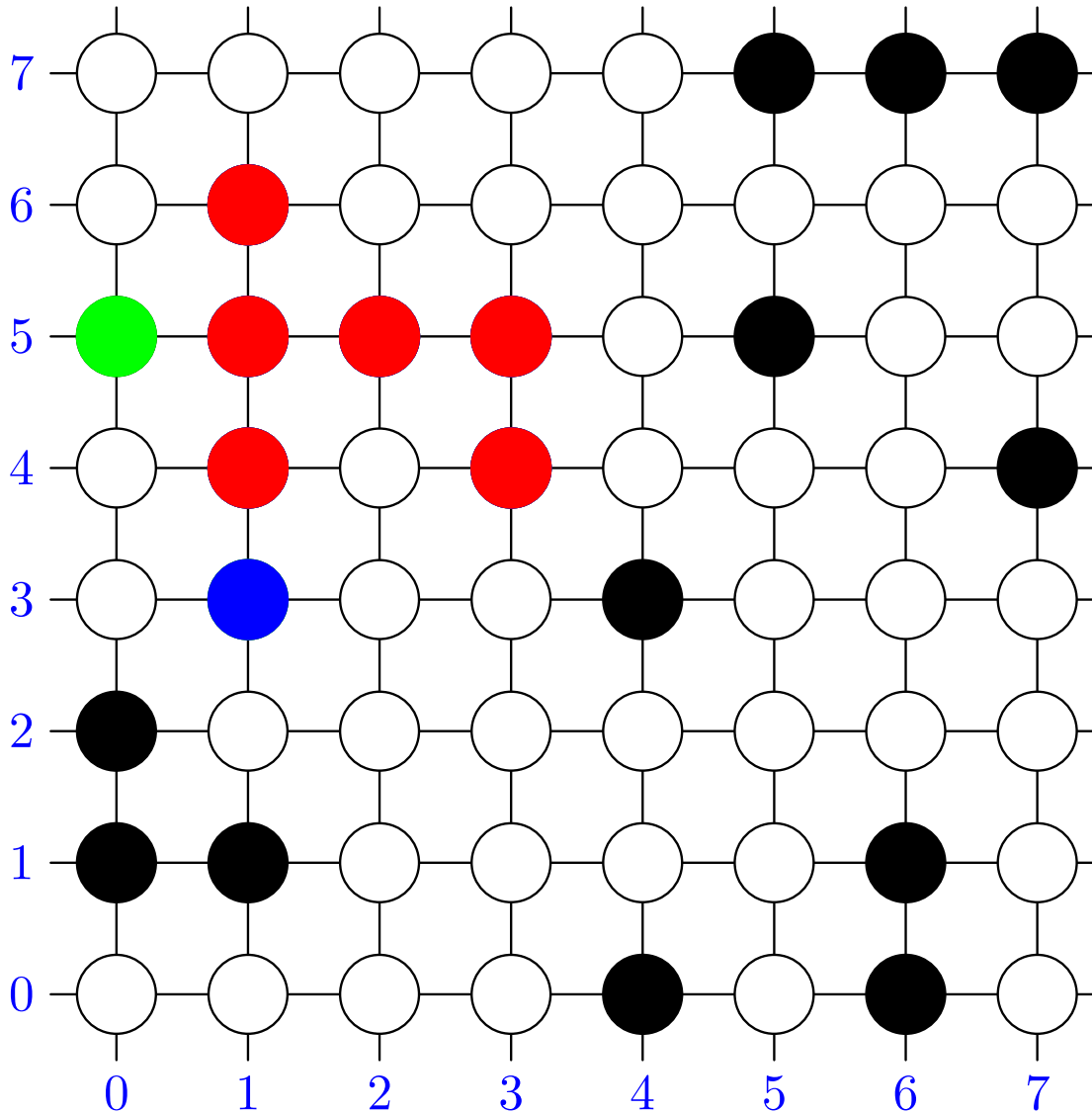
$\text{uncheckedNodes} =$

$\begin{matrix} (1, 3) \\ (0, 5) \end{matrix}$

$\text{clusterNodes} =$

$\{ (2, 5), (1, 5), (3, 5), \\ (3, 4), (0, 5), (1, 4), \\ (1, 6), (1, 3) \}$

Clustering



$\text{next} = (1, 3)$

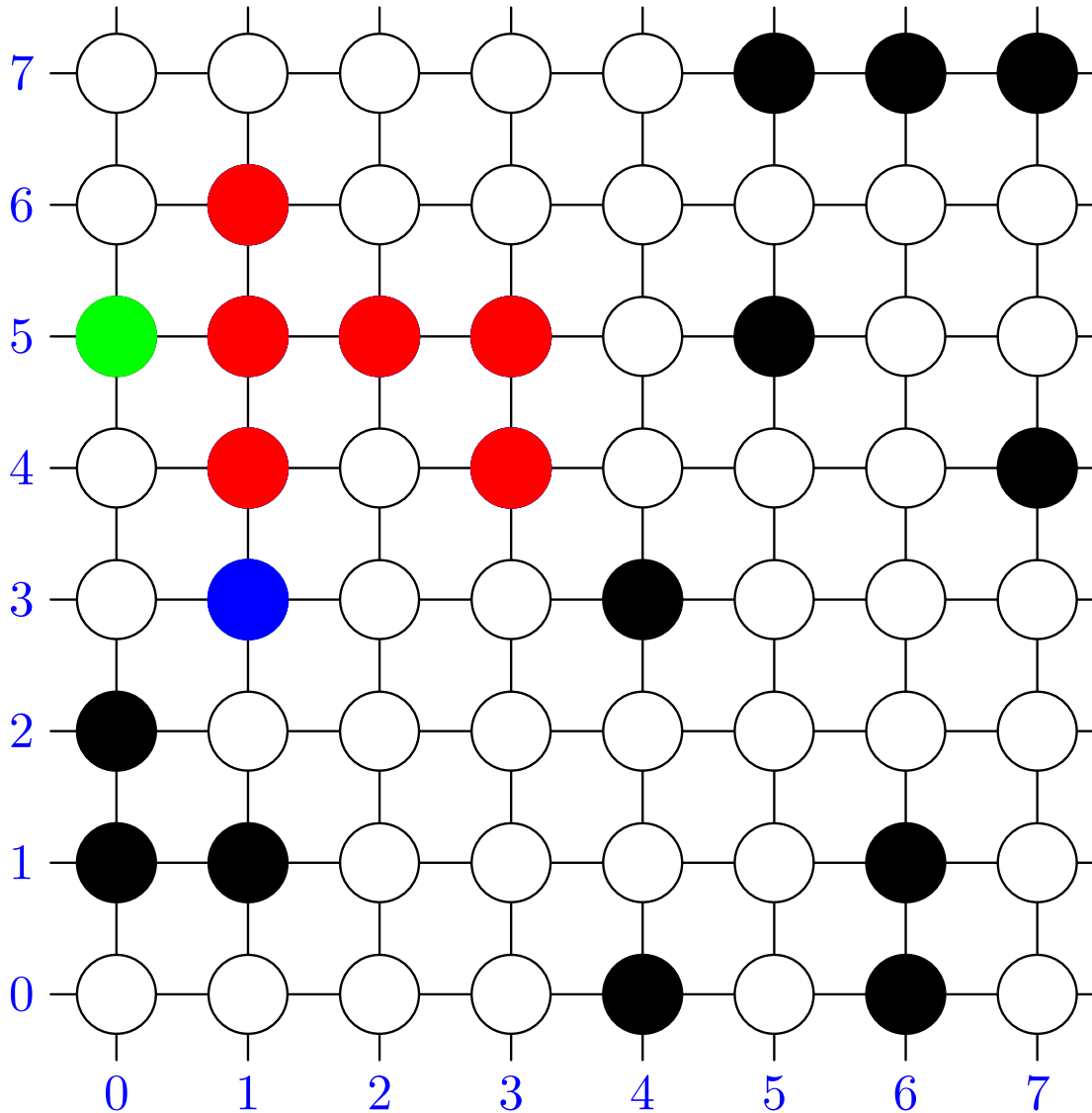
$\text{uncheckedNodes} =$

$(0, 5)$

$\text{clusterNodes} =$

$\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6), (1, 3) \}$

Clustering



$\text{next} = (1, 3)$

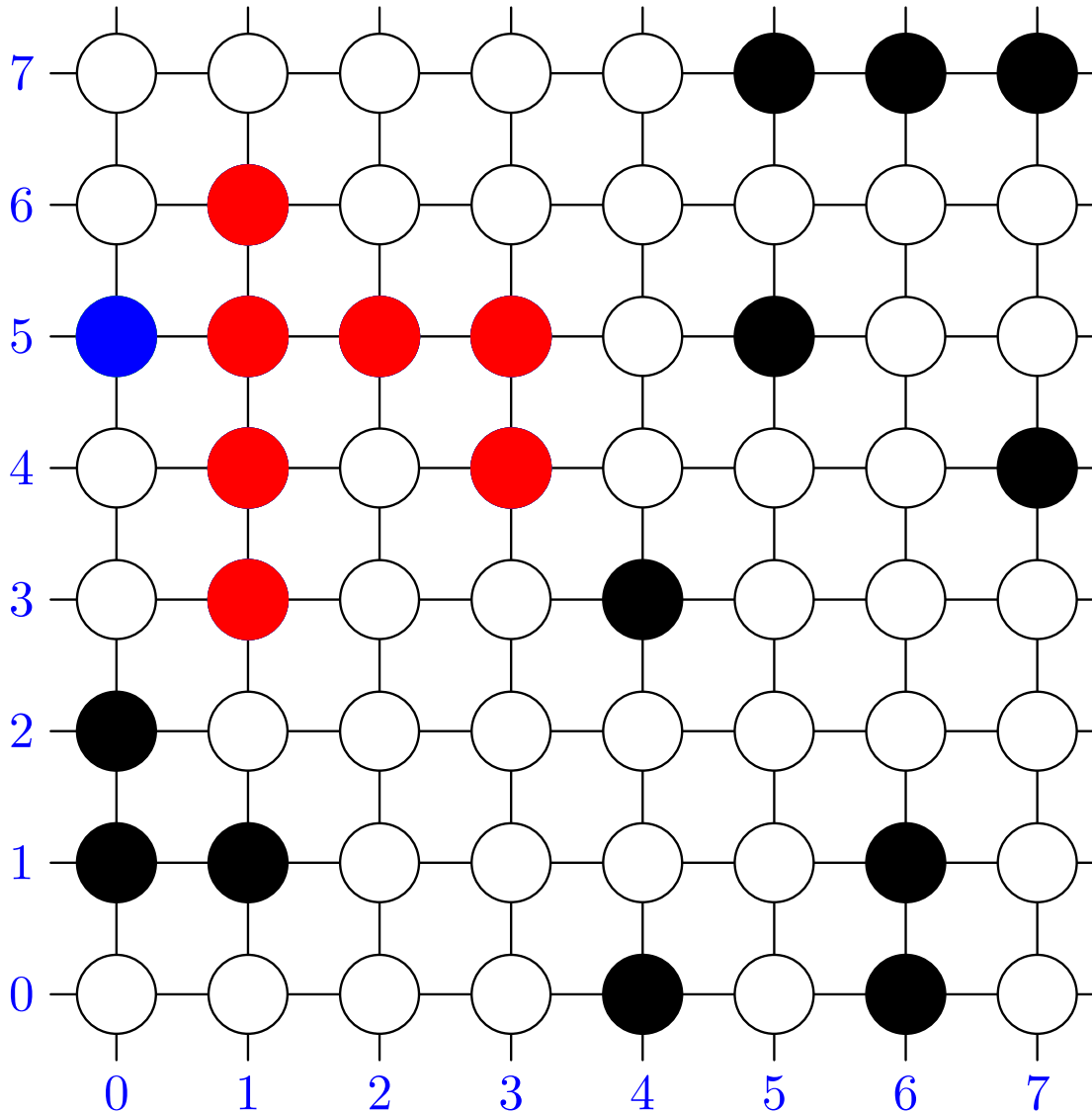
$\text{uncheckedNodes} =$

$(0, 5)$

$\text{clusterNodes} =$

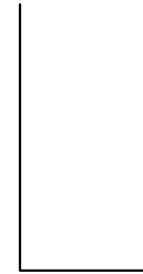
$\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6), (1, 3) \}$

Clustering



$\text{next} = (0, 5)$

$\text{uncheckedNodes} =$



$\text{clusterNodes} =$

$\{ (2, 5), (1, 5), (3, 5),$
 $(3, 4), (0, 5), (1, 4),$
 $(1, 6), (1, 3) \}$

Lessons

- Abstract Data Types (ADT) are interfaces for data structures
- Their purpose is to allow the programmer to declare their intentions
- They often have different implementations with different properties
- The most efficient implementation is not always obvious – we will see many of these implementations as we go through this course
- You need to know the common ADTs (e.g. Stack, Queue, List, Set, Map) and how and when to use them