# Search

## EECS 4315

www.eecs.yorku.ca/course/4315/

Source: weknowyourdreams.com

# Breadth first search

```
import gov.nasa.jpf.search.Search;

public class BFSearch extends Search {
  ...
}
```

# Constructor of BFSearch

```
public Search(Config config, VM vm)
```

- The `Config` object contains the JPF properties.
- The `VM` object refers to JPF's virtual machine.

**Question**

Implement the constructor of the `BFSearch`.

# Constructor of BFSearch

```
public Search(Config config, VM vm)
```

- The `Config` object contains the JPF properties.
- The `VM` object refers to JPF's virtual machine.

**Question**

Implement the constructor of the `BFSearch`.

**Answer**

```
public BFSearch(Config config, VM vm) {
  super(config, vm);
}
```

# Breadth first search

**Question**

Which data structure is usually used to implement breadth first search?

# Breadth first search

**Question**

Which data structure is usually used to implement breadth first search?

**Answer**

Queue.

# Breadth first search

**Question**

Which data structure is usually used to implement breadth first search?

**Answer**

Queue.

In Java, the class `java.util.LinkedList` implements the interface `java.util.Queue`.

| | |
|---|---|
| enqueue | offer |
| dequeue | poll |
| is empty? | isEmpty |

# The search method

The method

```
public void search()
```

drives the search.

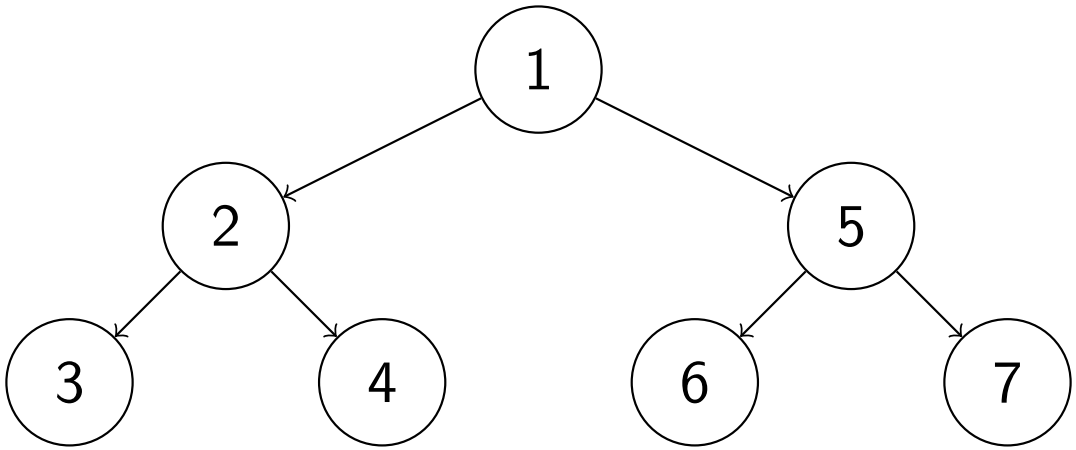
```
public boolean forward()
```

tries to move forward along an unexplored transition and returns whether the move is successful.

```
public boolean backtrack()
```

tries to backtrack and returns whether the backtrack is successful.

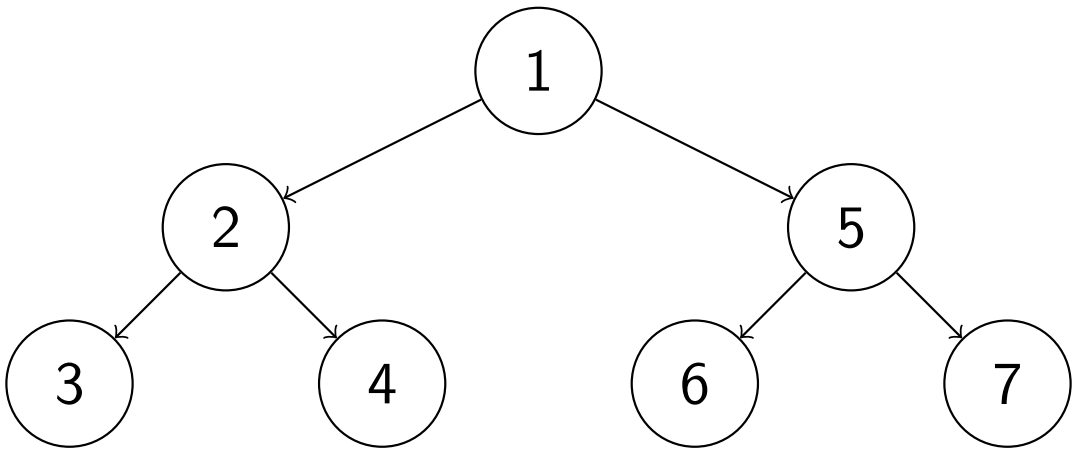
# The search method




## Question

For the above state space, provide the content of the queue and the sequence of calls to `forward`, `backtrack`, `enqueue` and `dequeue`, and the value returned by the first two, corresponding to breadth first search started in the top most state. Assume that initially the queue contains the top most state.

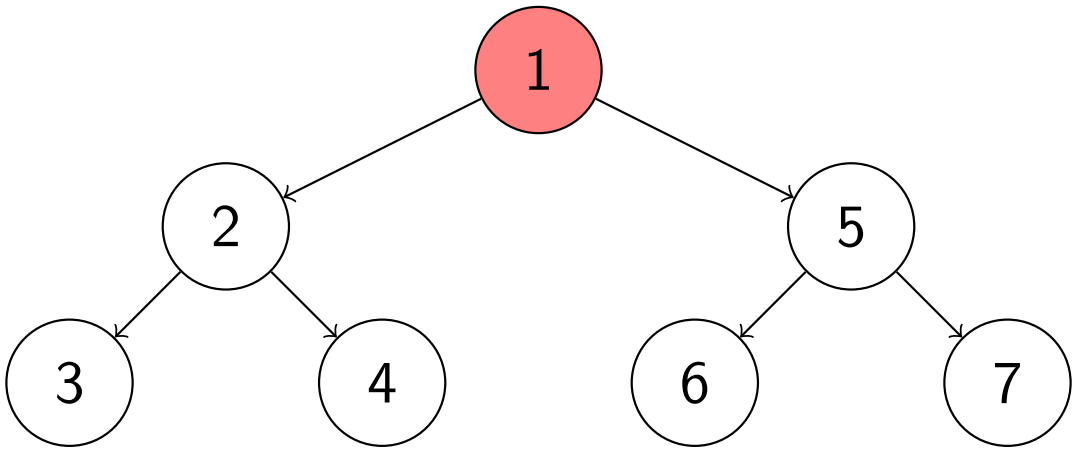
# The search method




**Answer**

queue: 1

# The search method

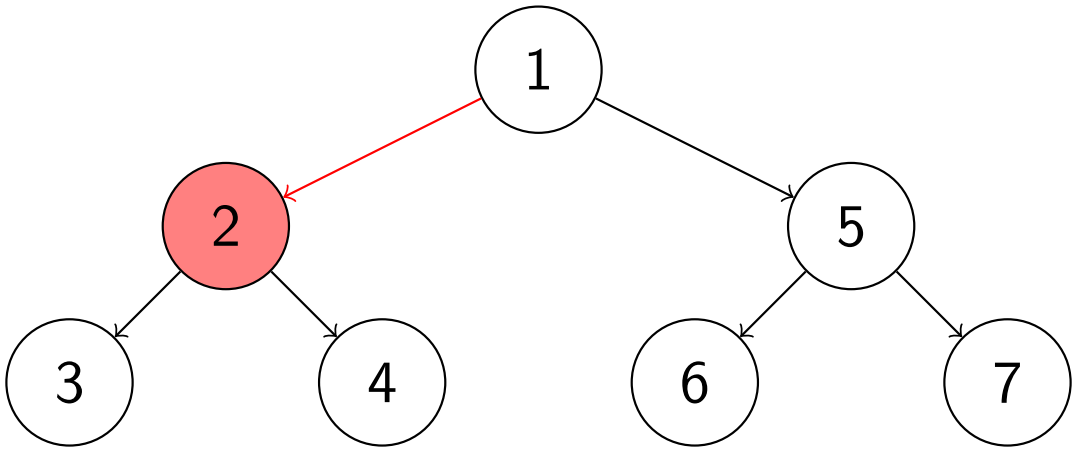



**Answer**

queue: empty

dequeue

# The search method

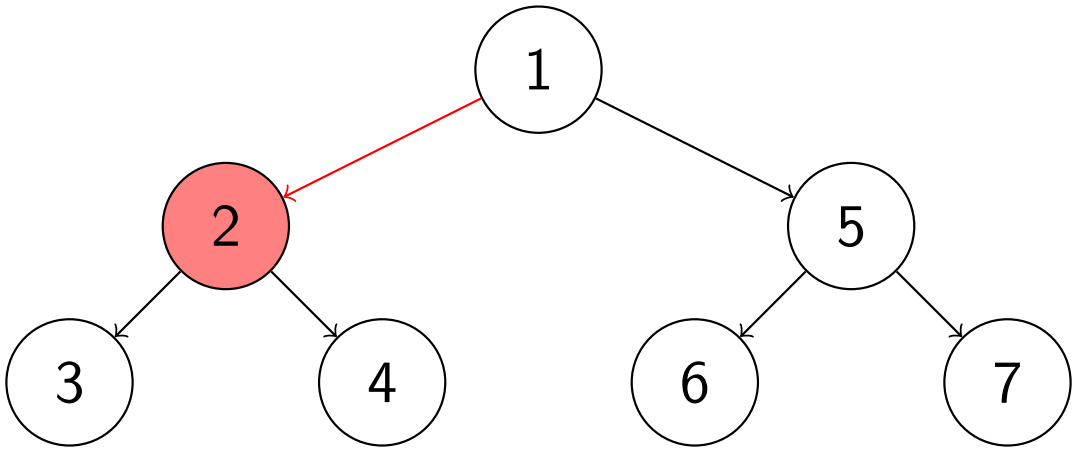



**Answer**

queue: empty

dequeue; forward(true)

# The search method

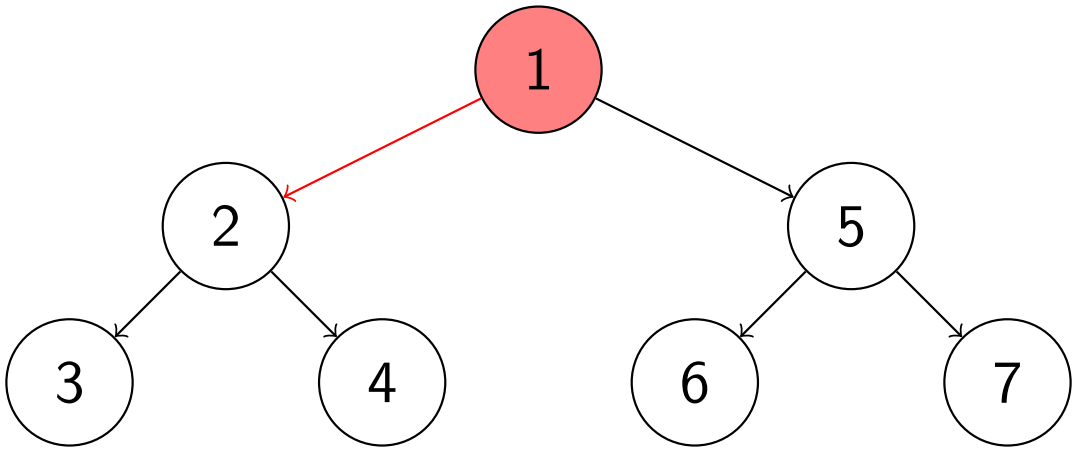



**Answer**

queue: 2

dequeue; forward(true); enqueue

# The search method

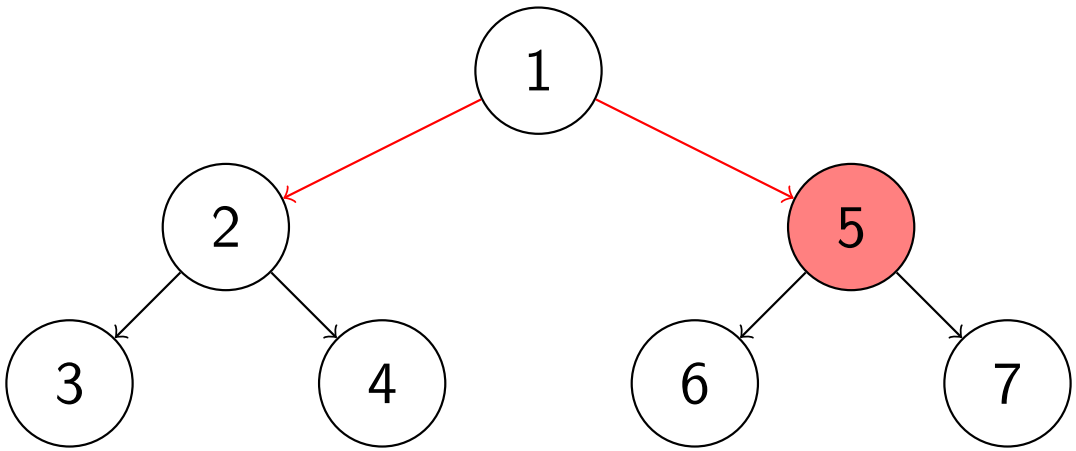



**Answer**

queue: 2

dequeue; forward(true); enqueue; backtrack(true)

# The search method

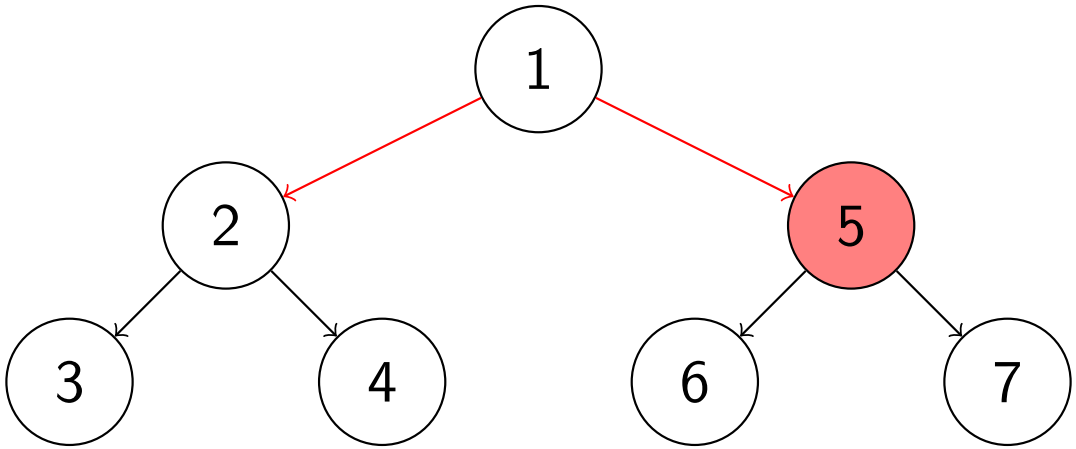



**Answer**

queue: 2

dequeue; forward(true); enqueue; backtrack(true); forward(true)

# The search method

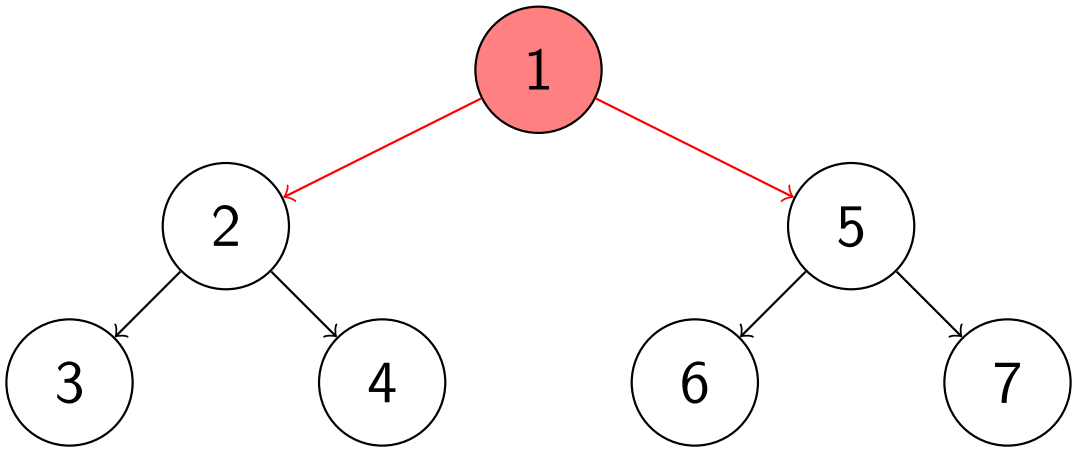



**Answer**

queue: 2, 5

dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue

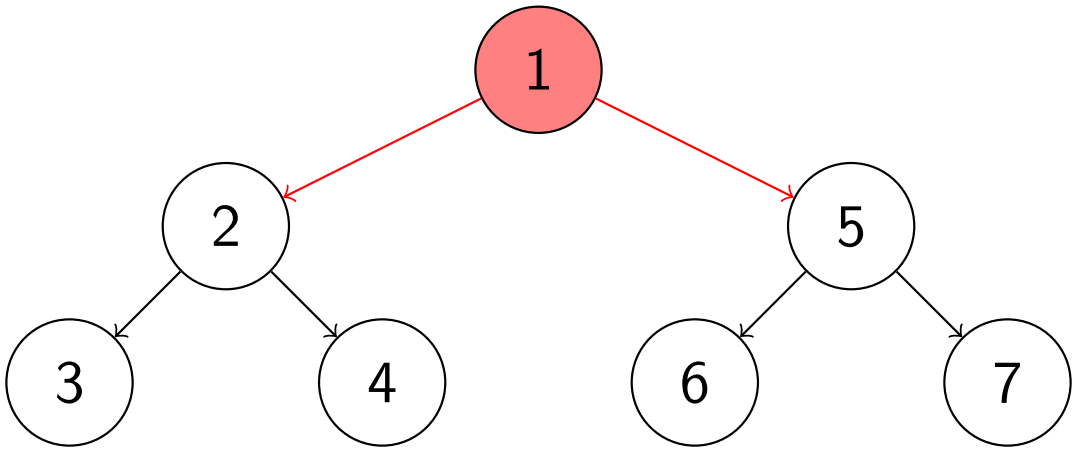
# The search method




**Answer**

queue: 2, 5
dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true)

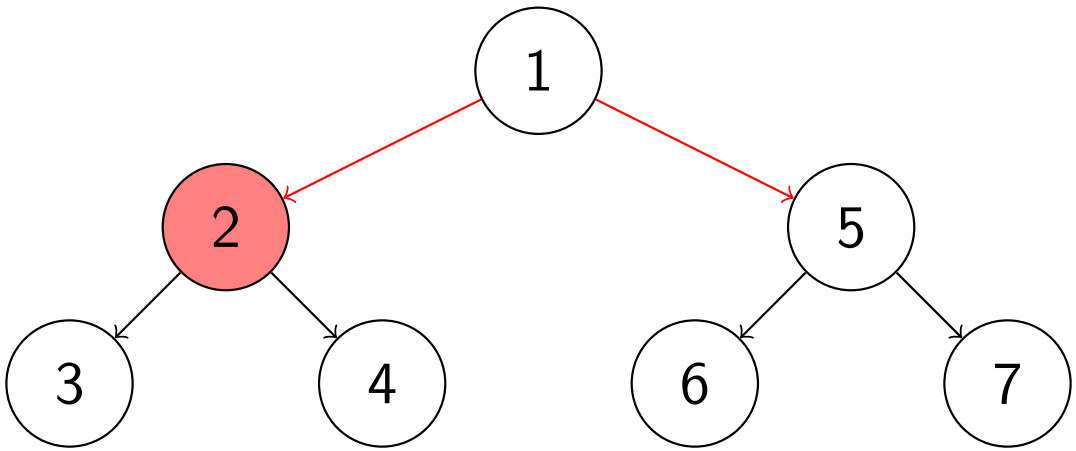
# The search method




**Answer**

queue: 2, 5
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false)

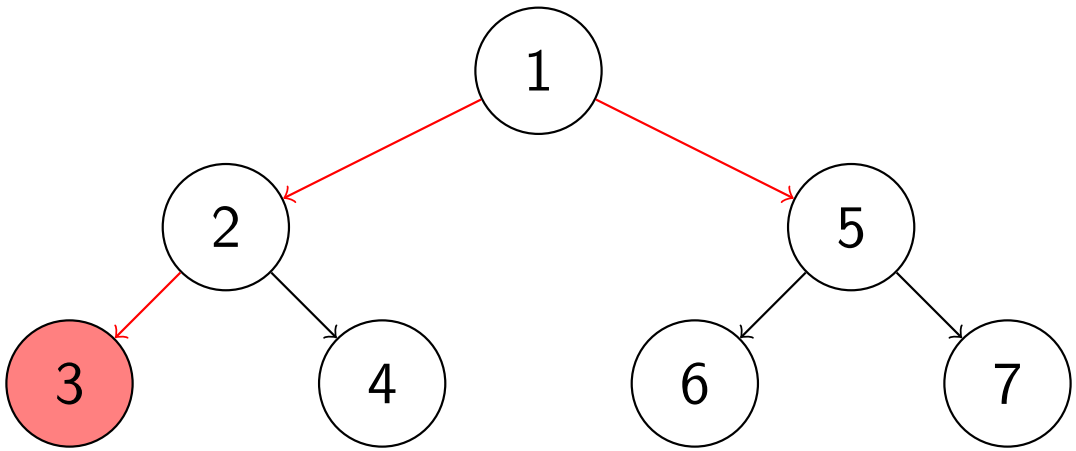
# The search method




**Answer**

queue: 5
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue

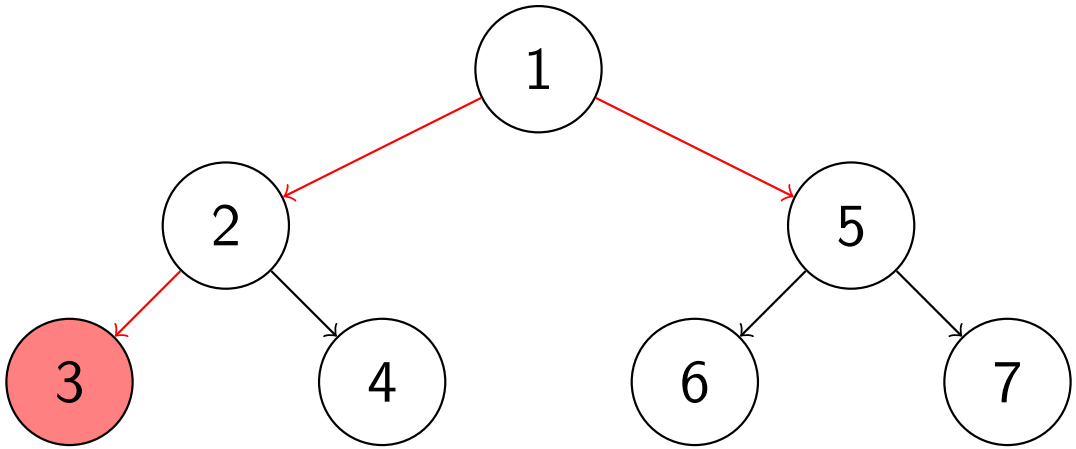
# The search method




**Answer**

queue: 5
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true)

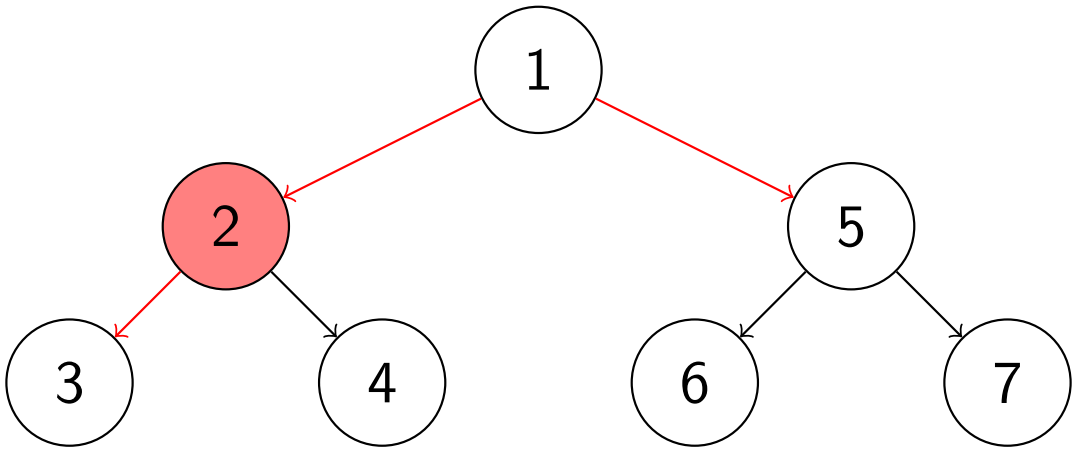
# The search method




**Answer**

queue: 5, 3
dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false); dequeue; forward(true); enqueue

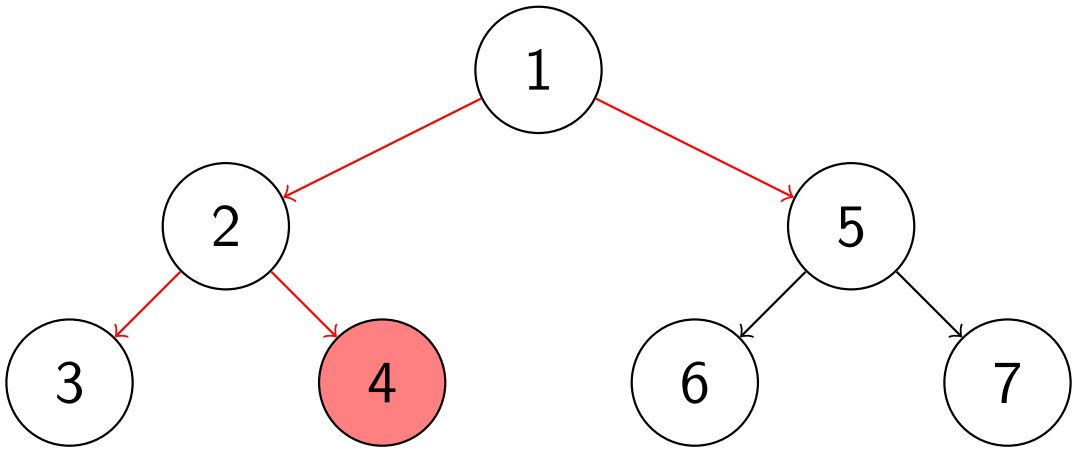
# The search method




**Answer**

queue: 5, 3
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true)

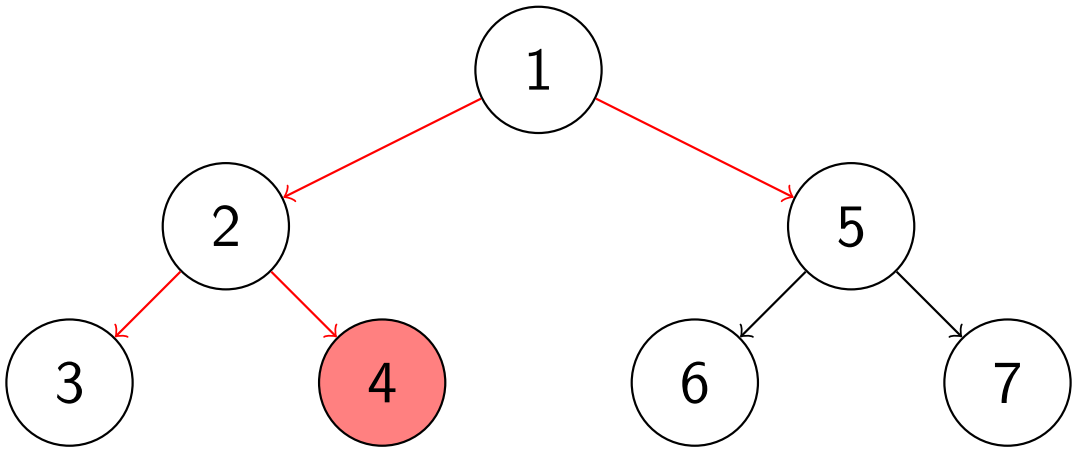
# The search method




**Answer**

queue: 5, 3
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true); forward(true)

# The search method

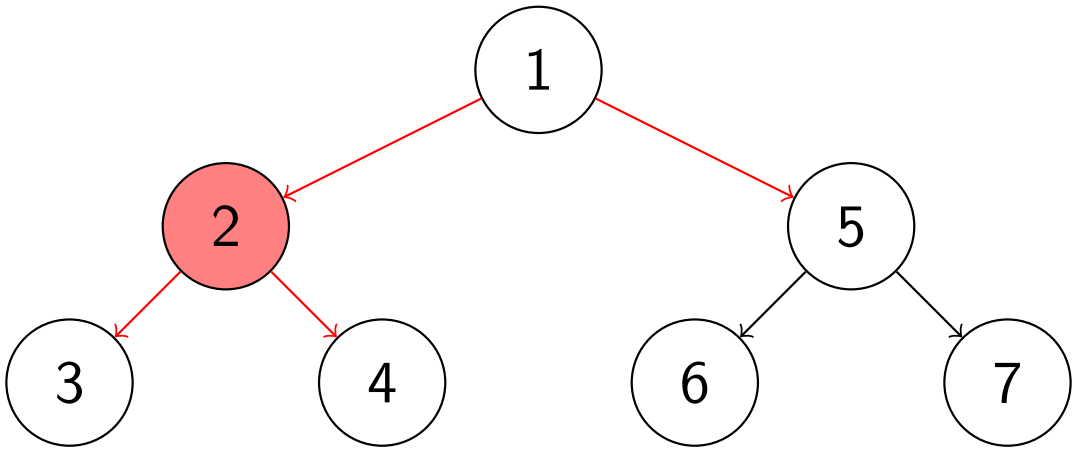



**Answer**

queue: 5, 3, 4

dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false); dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue

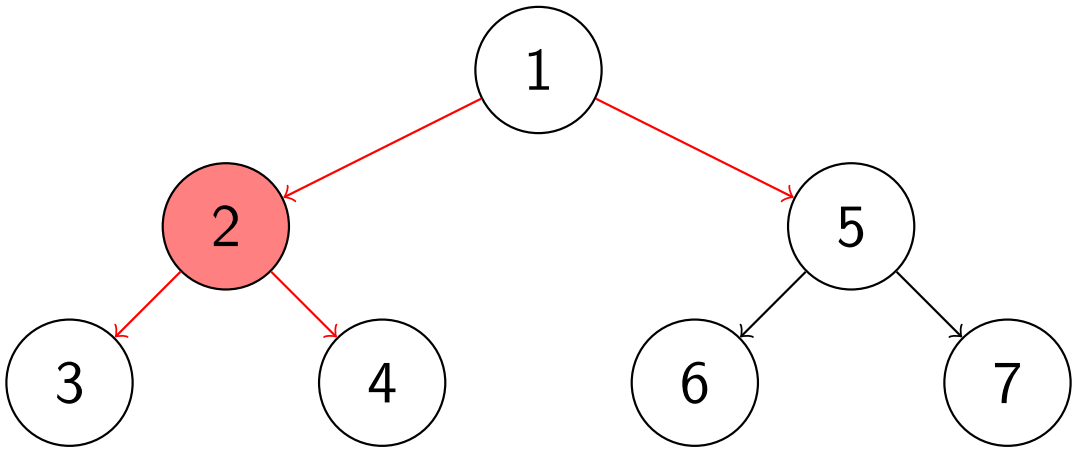
# The search method




**Answer**

queue: 5, 3, 4
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true); forward(true); enqueue; backtrack(true)

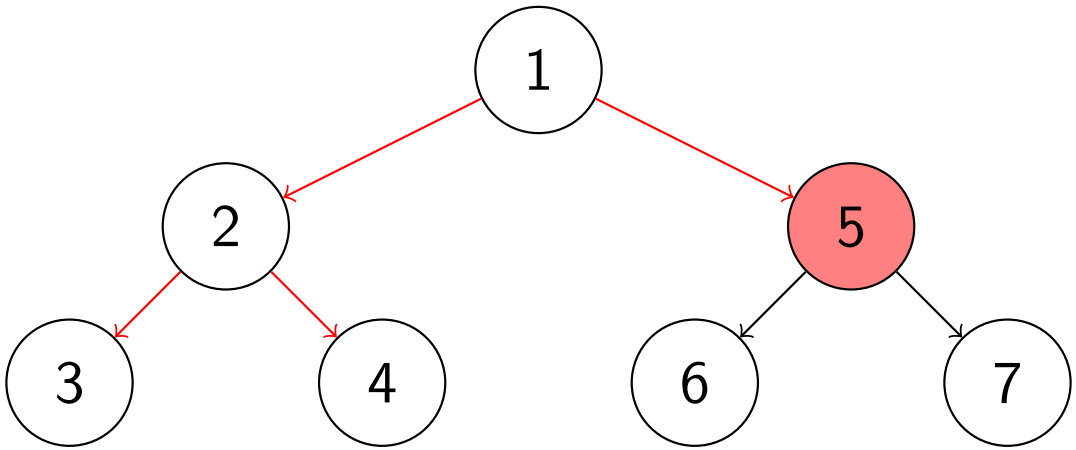
# The search method




**Answer**

queue: 5, 3, 4
dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false); dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false)

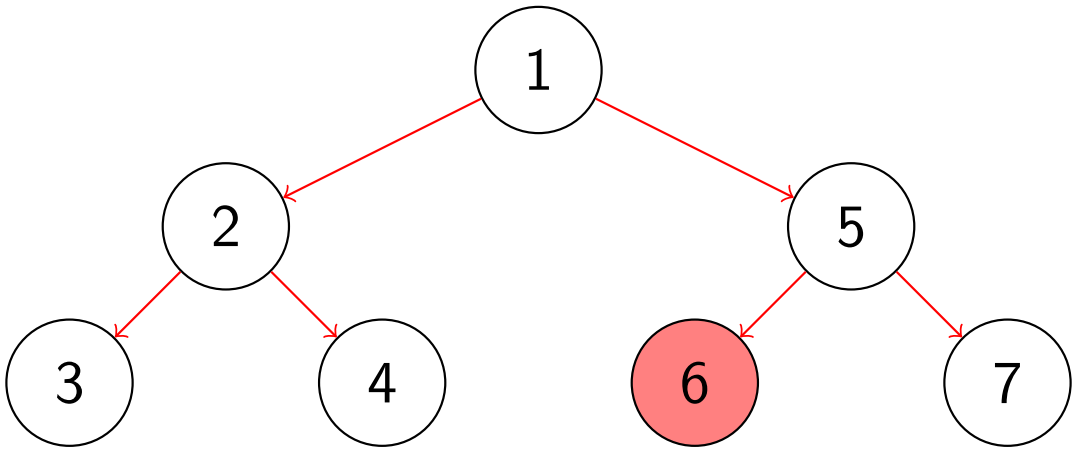
# The search method




**Answer**

queue: 3, 4
dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false); dequeue; forward(true); enqueue; backtrack(true); forward(true); enqueue; backtrack(true); forward(false); dequeue

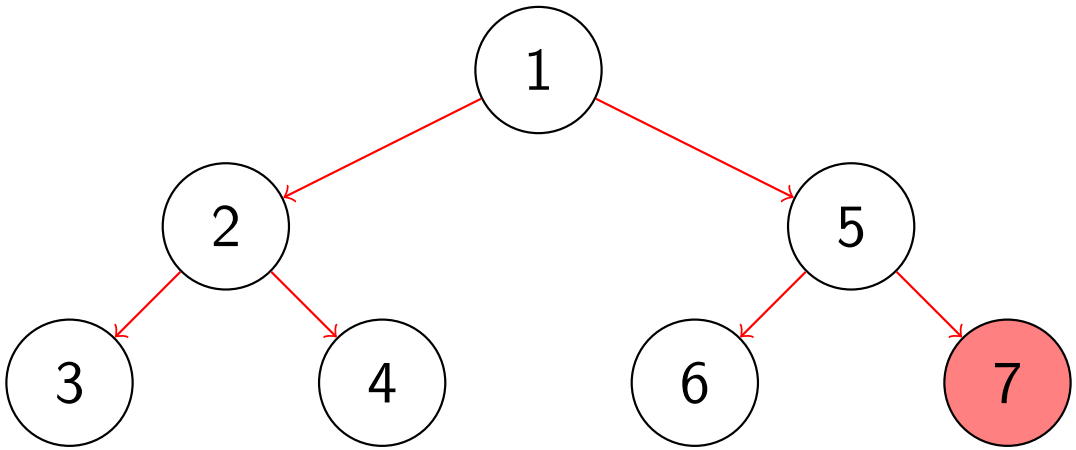
# The search method




**Answer**

queue: 6
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true); forward(true); enqueue; backtrack(true);
forward(false); dequeue; $\cdots$ ; forward(false)

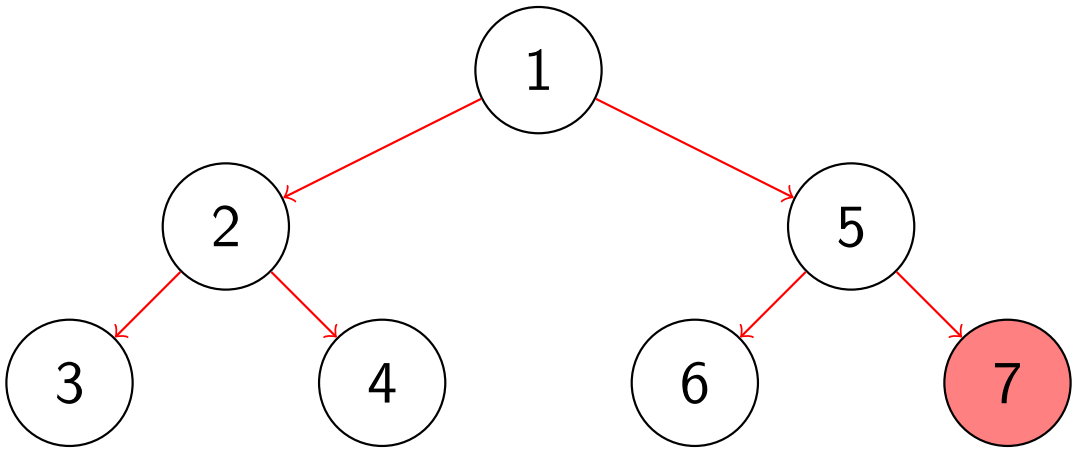
# The search method




**Answer**

queue: empty
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true); forward(true); enqueue; backtrack(true);
forward(false); dequeue; $\cdots$ ; forward(false); dequeue

# The search method




## Answer

queue: empty
dequeue; forward(true); enqueue; backtrack(true); forward(true);
enqueue; backtrack(true); forward(false); dequeue; forward(true);
enqueue; backtrack(true); forward(true); enqueue; backtrack(true);
forward(false); dequeue; $\cdots$ ; forward(false); dequeue;
forward(false)

# The search method

**Question**

Write some code consisting only of calls to `forward`, `backtrack`, `enqueue`, `dequeue` and `isEmpty` and loops that gives rise to the sequence on the previous slide.

# The search method

## Question

Write some code consisting only of calls to `forward`, `backtrack`, `enqueue`, `dequeue` and `isEmpty` and loops that gives rise to the sequence on the previous slide.

## Answer

```
enqueue();
while (!isEmpty()) {
  dequeue();
  while (forward()) {
    enqueue();
    backtrack();
  }
}
```

# Restoring states

We introduce the following methods.

```
/**
 * Returns the current state so that it is restorable.
 * @return the current state.
 */
private RestorableVMState getRestorableState() {
  return this.getVM().getRestorableState();
}

/**
 * Restores the given state.
 * @param state a state that is restorable.
 */
private void restoreState(RestorableVMState state) {
  this.getVM().restoreState(state);
}
```

# The search method

**Question**

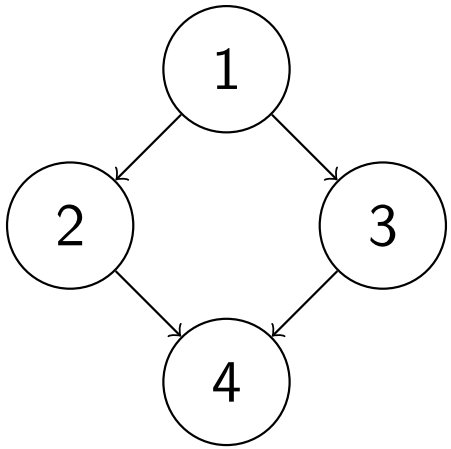
Implement the `search` method using a `Queue` of `RestorableVMState`s.

# The search method

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      queue.offer(this.getRestorableState());
      this.backtrack();
    }
  }
}
```

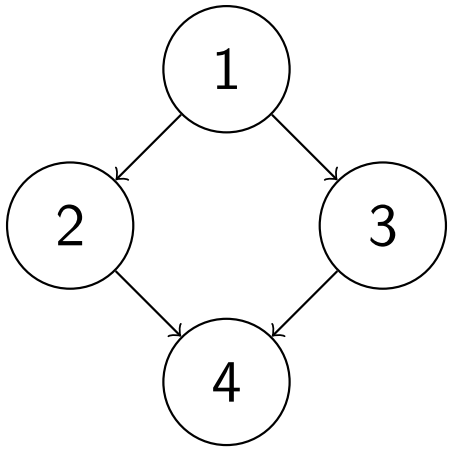
# The search method




**Question**

How often is state 4 enqueued?

# The search method




**Question**

How often is state 4 enqueued?

**Answer**

Twice.

# States

`public boolean isNewState()`

tests whether the current state has not been visited before.

**Question**

Modify the `search` method so that each state is enqueued at most once.

# States

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      if (this.isNewState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

# States

`public boolean isEndState()`

tests whether the current state is a final state.

**Question**

Modify the `search` method so that end states are not enqueued.

# States

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      if (this.isNewState() && !this.isEndState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

# States

`public boolean isIgnoredState()`

tests whether the current state can be ignored in the search.

States can, for example, be ignored by using in the system under test the method `ignoreIf(boolean)` of JPF's class `Verify` which is part of the package `gov.nasa.jpf.vm`.

**Question**

Incorporate the `isIgnoredState` method into the `search` method.

# The search method

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      if (this.isNewState()
          && !this.isEndState()
          && !this.isIgnoredState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

# The done attribute

Other components of JPF can end a search by setting the attribute `done` of the class `Search` to true.

**Question**

Modify the `search` method to incorporate the `done` attribute.

# The search method

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty() && !this.done) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward() && !this.done) {
      if (this.isNewState()
          && !this.isEndState()
          && !this.isIgnoredState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

# Request backtrack

The class `Search` contains the method `supportBacktrack` which tests whether a search supports backtrack requests.

**Question**

Modify the `BFSearch` class so that it does not support backtrack requests.

# Request backtrack

The class `Search` contains the method `supportBacktrack` which tests whether a search supports backtrack requests.

**Question**

Modify the `BFSearch` class so that it does not support backtrack requests.

**Answer**

```
public boolean supportBacktrack() {
  return false;
}
```

# Depth of search

The `Search` class contains the attribute `depth` that can be used to keep track of the depth of the search. It is initialized to zero.

**Question**

Modify the `search` method to keep track of the depth.

# Depth of search

## Answer

```
public void search() {
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  queue.offer(null);
  while (queue.size() > 1 && !this.done) {
    RestorableVMState state = queue.poll();
    if (state == null) {
      this.depth++;
      queue.offer(null);
    } else {
      this.restoreState(state);
      while (this.forward() && !this.done) {
        if (this.isNewState()
            && !this.isEndState()
            && !this.isIgnoredState()) {
          queue.offer(this.getRestorableState());
        }
        this.backtrack();
```

# Limit depth of search

JPF can be configured to limit the depth of the search by setting the JPF property `search.depth_limit`. The default value of `search.depth_limit` is `Integer.MAX_VALUE`. The `Search` class provides the method `getDepthLimit` which returns the maximal allowed depth of the search.

**Question**

Add the method `checkDepthLimit` that tests whether the current depth is smaller than the limit. Also modify the `search` method to incorporate this method.

# Limit depth of search

## Answer

```
private boolean checkDepthLimit() {
  return this.depth < this.getDepthLimit();
}

public void search() {
  ...
  while (queue.size() > 1
         && !this.done
         && this.checkDepthLimit()) {
    ...
  }
}
```

# Limit memory usage

The JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The default value is $1024 \ll 10 = 1024^2 = 1,048,576B \approx 1MB$. By leaving some memory free, JPF can report that it ran out of memory and provide some useful statistics instead of simply throwing an `OutOfMemoryError`. The method `checkStateSpaceLimit` of the class `Search` checks whether the minimal amount of memory that should be left free is still available.

**Question**

Modify the `search` method to limit the memory usage.

# Limit memory usage

## Answer

```
public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
  if (!available) {
    this.done = true;
  }
  return available;
}

public void search() {
  ...
    while (this.forward()
           && !this.done) {
           && this.checkStateSpaceLimit()) {
  ...
}
```

# Multiple errors?

The JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The `forward` method also checks whether any property is violated after the unexplored transition has been traversed. If a violation has been detected then the attribute `done` is set to true if and only if JPF has been configured to report at most one error.

The method `hasPropertyTermination` of the class `Search` checks whether a violation was encountered during the last transition. The method returns true if and only if a violation was encountered and the attribute `done` is set to true.

**Question**

Modify the `search` method to take `search.multiple_errors` into account.

# Multiple errors

## Answer

```
public void search() {
  ...
    while (this.forward()
           && !this.done
           && this.checkStateSpaceLimit()
           && !this.hasPropertyTermination()) {
      if (!this.isErrorState()
          && this.isNewState()
          && ...) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
  ...
}
```

# Notifications

A search should also notify listeners of particular events by invoking to the methods of the interface `SearchListener`, which can be found in the package `gov.nasa.jpf.search`. The `Search` class contains a number of `notify` methods.

**Question**

Modify the `search` method of the `BFSearch` class to incorporate following notifications.

- `notifySearchStarted`
- `notifySearchFinished`

# Notifications

## Answer

```
public void search() {
  this.notifySearchStarted();
  Queue<RestorableVMState> queue =
    new LinkedList<RestorableVMState>();
  ...
  this.notifySearchFinished();
}
```

# Notifications

## Question

Override the `forward` method and the `backtrack` method of the `Search` class to incorporate following notifications.

- `notifyStateAdvanced`
- `notifyStateBacktracked`
- `notifyStateProcessed`

# Notifications

## Answer

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}

protected boolean backtrack() {
  boolean successful = super.backtrack();
  if (successful) {
    this.notifyStateBacktracked();
  }
  return successful;
}
```

# Notifications

## Question

Override the `checkStateSpaceLimit` method and modify the `checkDepthLimit` method to incorporate `notifySearchConstraintHit(String)` to notify the following.

- "memory limit reached"
- "depth limit reached"

# Notifications

## Answer

```
public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
  if (!available) {
    this.done = true;
    this.notifySearchConstraintHit("memory limit reached");
  }
  return available;
}

private boolean checkDepthLimit() {
  boolean below = this.depth <= this.getDepthLimit();
  if (!below) {
    this.notifySearchConstraintHit("depth limit reached");
  }
  return below;
}
```

# Notifications

Immediately after an invocation of the `forward` method of the `Search` class, an invocation of the `getCurrentError` method of the `Search` class returns `null` if and only if no property violation has been detected.

**Question**

Modify the overridden `forward` method of the `BFSearch` class to include an invocation of the `notifyPropertyViolated` method.

# Notifications

## Answer

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
    if (this.getCurrentError() != null) {
      this.notifyPropertyViolated();
    }
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}
```

# Testing our searches

**Question**

How do we test our `DFSearch` and `BFSearch`?

# Testing our searches

**Question**

How do we test our `DFSearch` and `BFSearch`?

**Answer**

Compare them with the corresponding JPF search strategies.

# Testing our searches

**Question**

What can be observed about a search?

# Testing our searches

**Question**

What can be observed about a search?

**Answer**

Its notifications.

# Testing our searches

**Question**

What can be observed about a search?

**Answer**

Its notifications.

**Question**

How can those be recorded?

# Testing our searches

**Question**

What can be observed about a search?

**Answer**

Its notifications.

**Question**

How can those be recorded?

**Answer**

Implement a search listener.

# Testing our searches

**Question**

What to do with the recording of a search?

# Testing our searches

**Question**

What to do with the recording of a search?

**Answer**

Compare it to the recording of another search.

# Testing our searches

**Question**

What to do with the recording of a search?

**Answer**

Compare it to the recording of another search.

**Question**

In which way can the recordings be stored so that they can easily be compared?

# Testing our searches

**Question**

What to do with the recording of a search?

**Answer**

Compare it to the recording of another search.

**Question**

In which way can the recordings be stored so that they can easily be compared?

**Answer**

A serialized list of strings.