人工知能入門
第9回

藤田 悟

黄　潤和

前回学んだこと

- 探索とは
  - 探索問題
  - 探索解の性質
- 探索空間の構造
  - 探索木
  - 探索グラフ
- 探索順序
  - 深さ優先探索
  - 幅優先探索

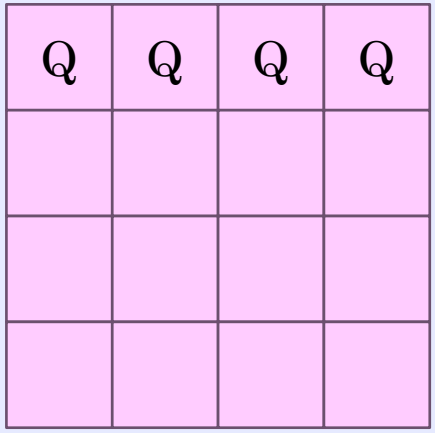
今回学ぶこと

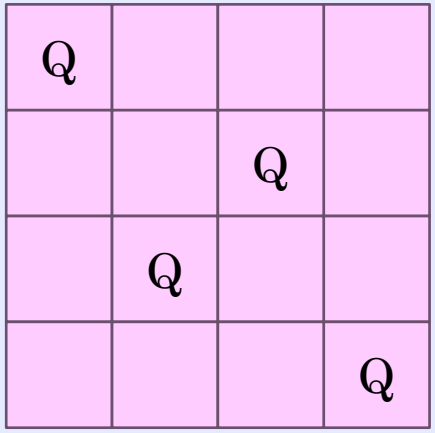
- 探索プログラムの作成
  - バックトラック
  - 深さ優先探索
  - 幅優先探索

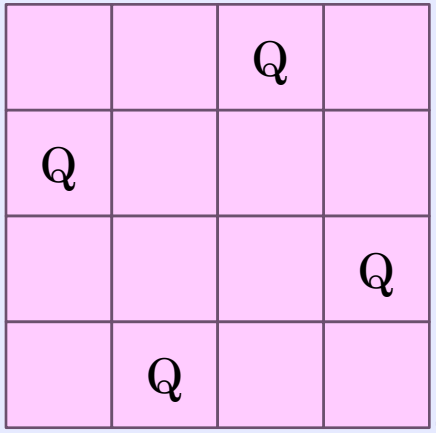
探索プログラム

n-Queen の探索プログラム

- n個のQueen をn×nのマスの中に、縦横斜めに重ならないように配置する。
  - 簡単化のために　4-Queen を考える







◎正解

探索プログラム

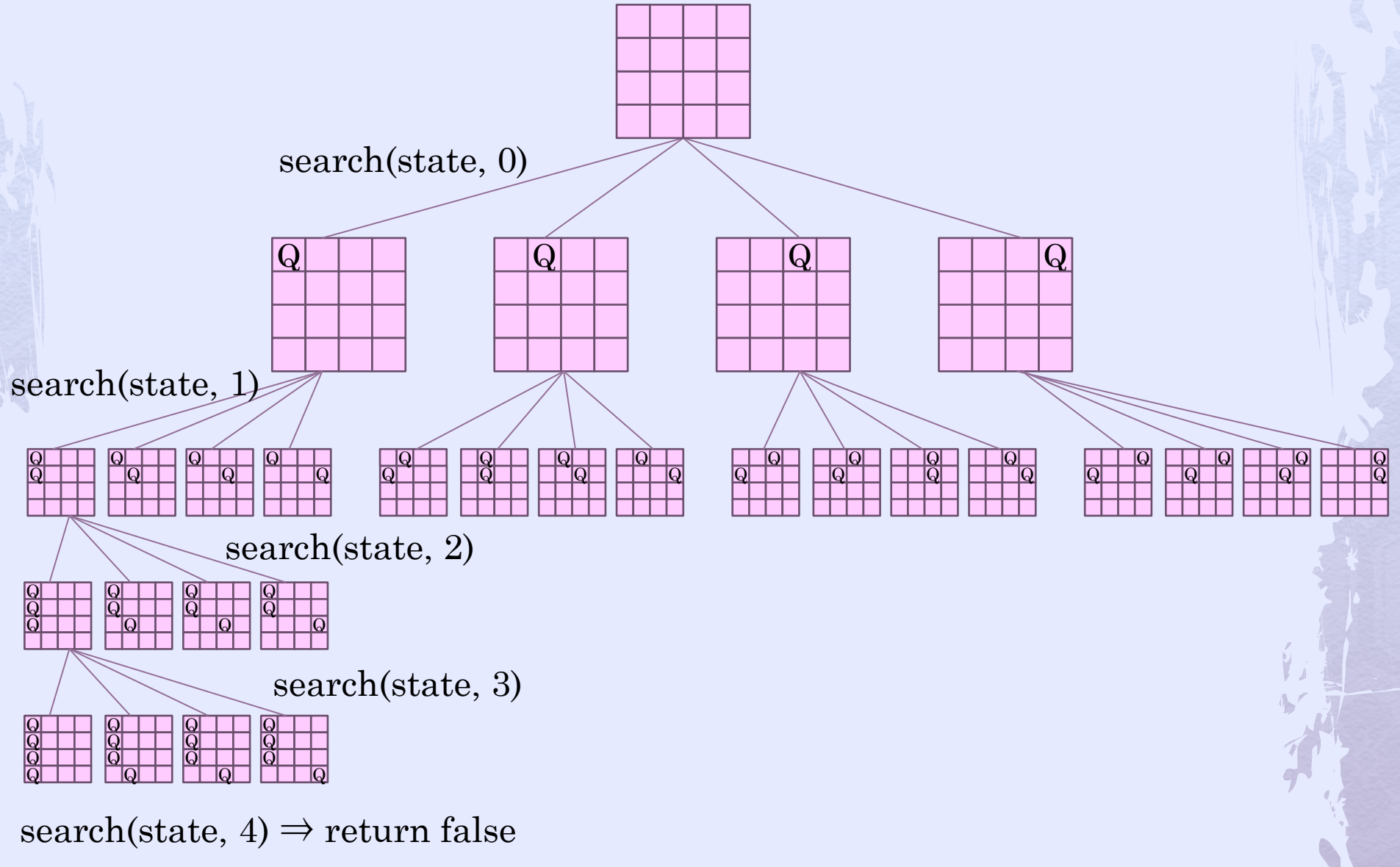
- 全状態の探索プログラム
  - 全ての最終状態を生成した後に、最終状態が解であるかどうかを判定する
- 深さ優先探索プログラム
  - 中間状態についても判定を行い、枝刈りを実施。
  - 解の条件を満たす限り、探索を深堀りする
- 幅優先探索プログラム
  - 深さが同じ中間状態をすべて探索した後に、次の深さの中間状態を生成する。

探索プログラムの擬似コード

```
void start() {
    初期化();
    search(状態, 0);
}
boolean search(状態, depth) {
    if(depth == n) { // 最終状態の深さに到達
        if(最終状態判断(状態)) return true; // 解が見つかった
        else return false; // 解ではなかったので、バックトラック
    }
    // 状態の下に存在する新たな状態を生成する
    for(int i = 0; i < width; i++) {
        新状態 = update(状態, i);
        //  新状態に対して、探索を再帰的に呼び出す
        if (search(新状態, depth+1)) {
            return true;
        }
    }
    return false;
}
```

再帰呼び出しがポイント

全状態を生成してから判定

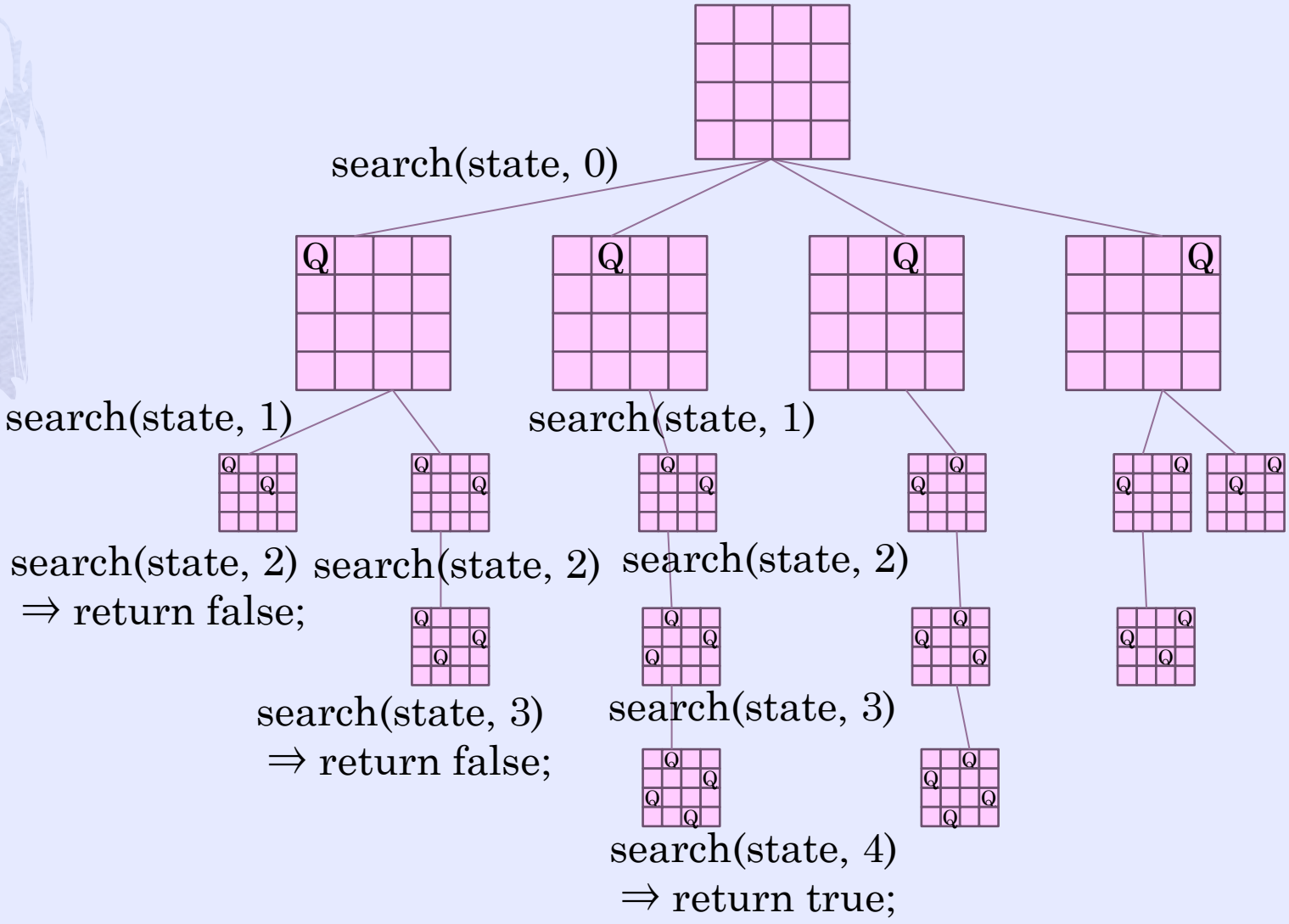


深さ優先探索プログラムの擬似コード

```
void start() {
        初期化();
        search(状態, 0);
}
boolean search(状態, depth) {
        // 状態の下に存在する新たな状態を生成する
        for(int i = 0; i < width; i++) {
                新状態 = update(状態, i);
                if(中間状態判定(新状態)) { // 中間状態が矛盾ない場合だけ、探索を深く行う
                        if(depth == n) { // 最終状態の深さに到達
                                return true; // 深さnまで矛盾のない状態が作れたので、探索成功
                        } else {
                                //  新状態に対して、探索を再帰的に呼び出す
                                if(search(新状態, depth+1)) {
                                        return true;
                                }
                        }
                }
        }
        return false;
}
```

可能性のなくなった枝は false を返して
バックトラック(一つ上の search に戻る)する

深さ優先探索


search(state, 0)
search(state, 1)
search(state, 1)
search(state, 2)
⇒ return false;
search(state, 2)
search(state, 2)
search(state, 3)
⇒ return false;
search(state, 3)
search(state, 4)
⇒ return true;

深さ優先探索プログラム: main

```
public class QueenDepth {
        // nQueen のサイズ
        int n = 4;

        //  全解探索の時には初期値を0にして、解の数を計算
        //  -1の時は、単独解の探索
        int count = 0;

        public static void main(String[] args) {
                new QueenDepth().start();
        }

        void start() {
                int[] board = init(); // 初期化

                search(board, 0); // 探索

                if(count >= 0) {
                        System.out.println(count);
                }
        }
```

In your Eclipse, please make

Java Project Name: AI-0
Package Name: nQueen
Java Class Name: QueenDepth

深さ優先探索プログラム: 初期化

```
// 初期化
int[] init() {
        int[] board = new int[n];
        for(int i = 0; i < n; i++) {
                board[i] = -1;
        }
        return board;
}
// 結果のコンソール出力
void print(int[] board) {
        System.out.println();
        for(int i = 0; i < n; i++) {
                String line = "";
                for(int j = 0; j < n; j++) {
                        if(board[i] == j) {
                                line += "*";
                        } else {
                                line += ".";
                        }
                }
                System.out.println(line);
        }
}
```

board[0] = -1
board[1] = -1
board[2] = -1
board[3] = -1

. *. .
. . . *
*. . .
. . *.

. . *.
*. . .
. . . *
. *. .

board[0] = 1
board[1] = 3
....

2 solutions

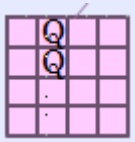
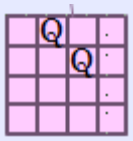
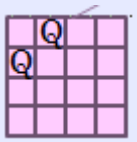
深さ優先探索プログラム: 探索

```
// 探索
boolean search(int[] board, int depth) {
        // ブランチの数だけ探索を行う
        for(int i = 0; i < n; i++) {
                // 中間状態の生成
                board[depth] = i;
                // 中間状態を評価し、真なら次の深さを探索
                if(check(board, depth)) {
                        if (depth + 1 == n) { // 正解に到達
                                print(board);
                                if(count >= 0) {
                                        count++;
                                } else {
                                        return true;
                                }
                        } else {
                                if(search(board, depth + 1) && count < 0) {
                                        return true;
                                }
                        }
                }
        }
        return false;
}
```

n=4

可能性のなくなった枝は false を返して

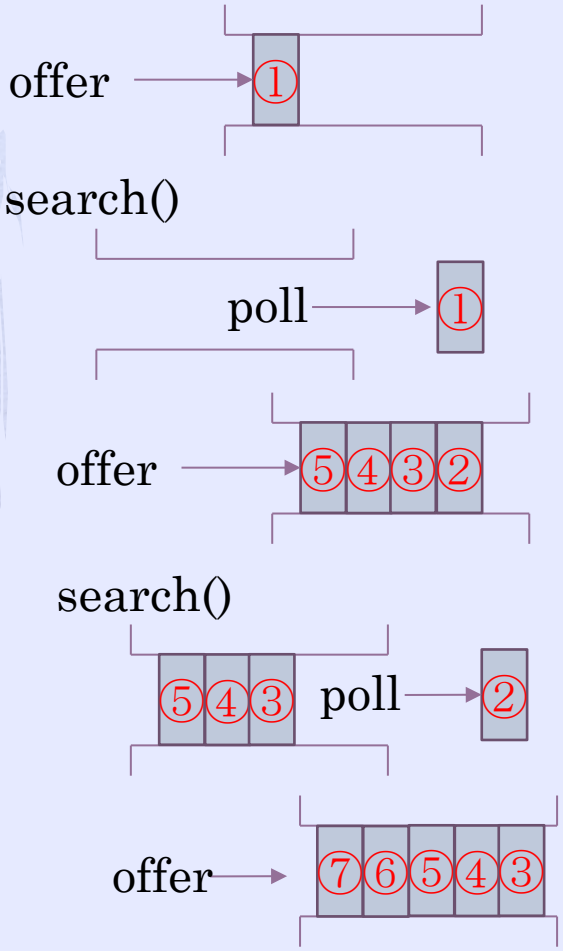
深さ優先探索プログラム: 判定

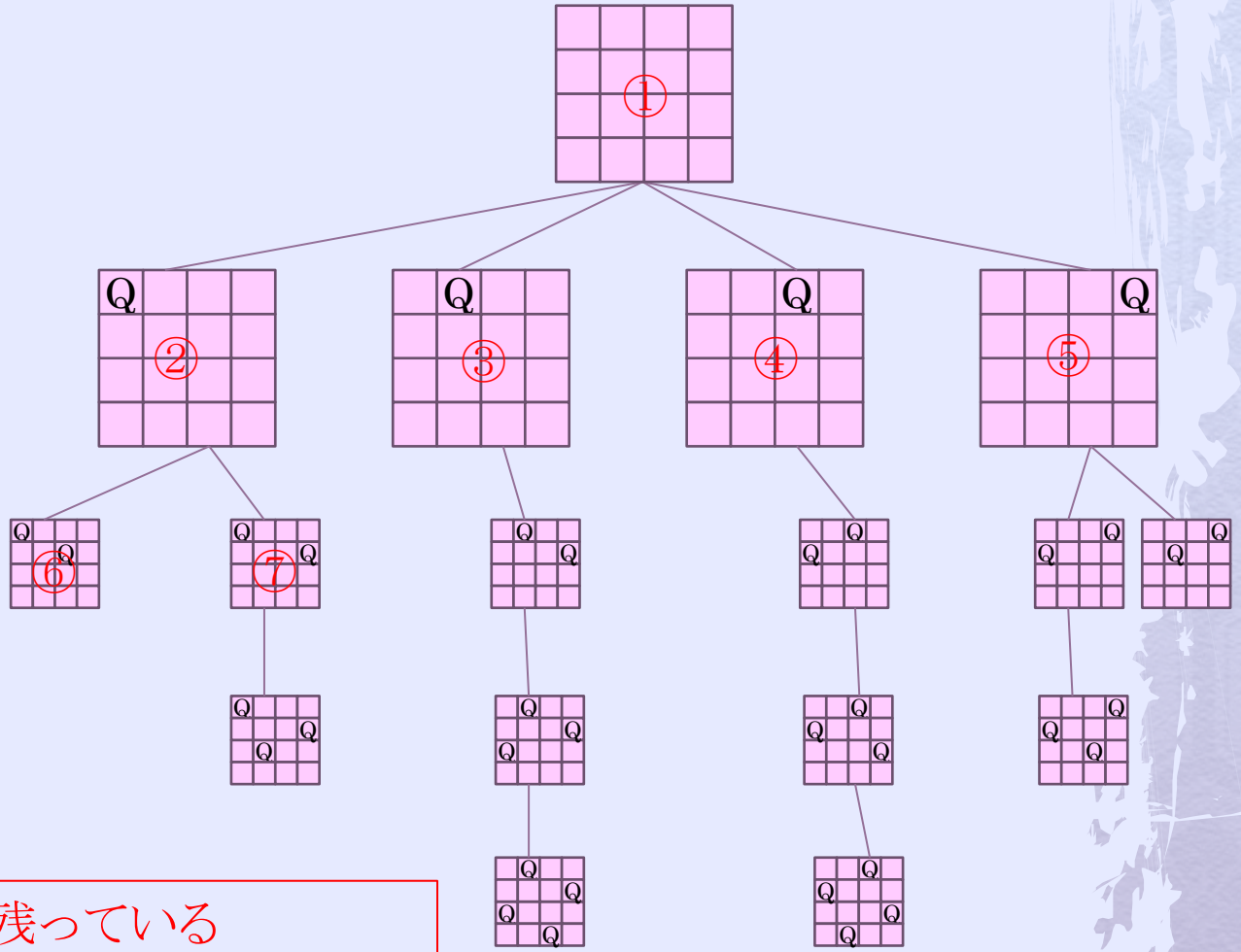
```
// 条件判定
boolean check(int[] board, int depth) {
        for(int i = 0; i < depth; i++) {
                
                // 縦チェック
                if(board[i] == board[depth]) {
                        return false;
                }

                
                // 斜め左上チェック
                if(board[depth] - (depth - i) == board[i]) {
                        return false;
                }

                
                // 斜め右上チェック
                if(board[depth] + (depth - i) == board[i]) {
                        return false;
                }
        }

        return true;
}
}
```

幅優先探索プログラムの擬似コード

```
void start() {
        初期化();
        search();
}
boolean search() {
        state = poll(); // Queue から状態を1個取り出す
        if(state == null) {
                return false; // Queue が空になったので終了
        }
        // 状態の下に存在する新たな状態を生成する
        for(int i = 0; i < width; i++) {
                新状態 = update(状態, i);
                if(中間状態判定(新状態)) { // 中間状態が矛盾ない場合だけ、探索を深く行う
                        if(state.depth == n) {
                                return true; // 深さnの解が見つかったので、終了
                        } else {
                                offer(新状態); // 新状態を Queue に詰め込む
                        }
                }
        }
        return search(); // 下位のノードを全て作り終わったところで、search()を再帰呼び出しする。
}
```

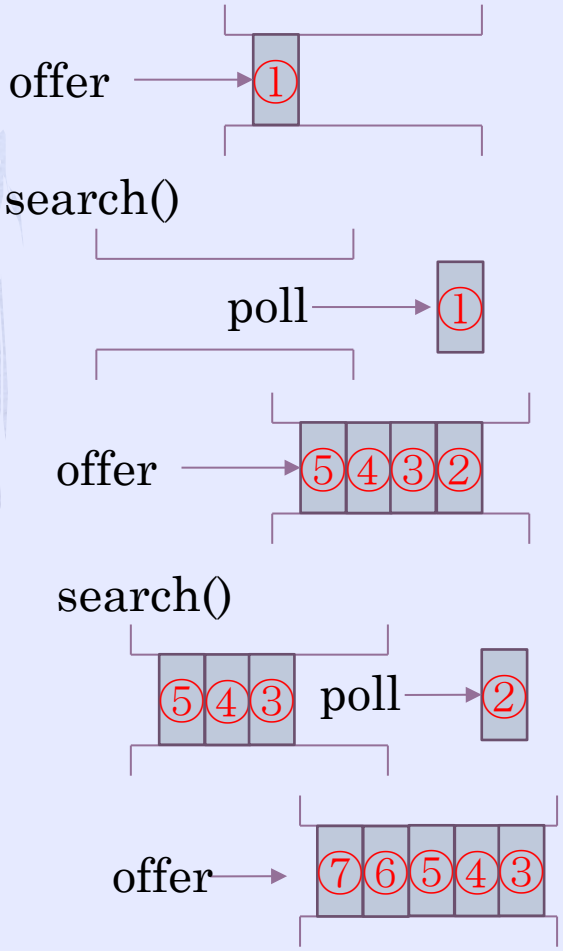
幅優先探索

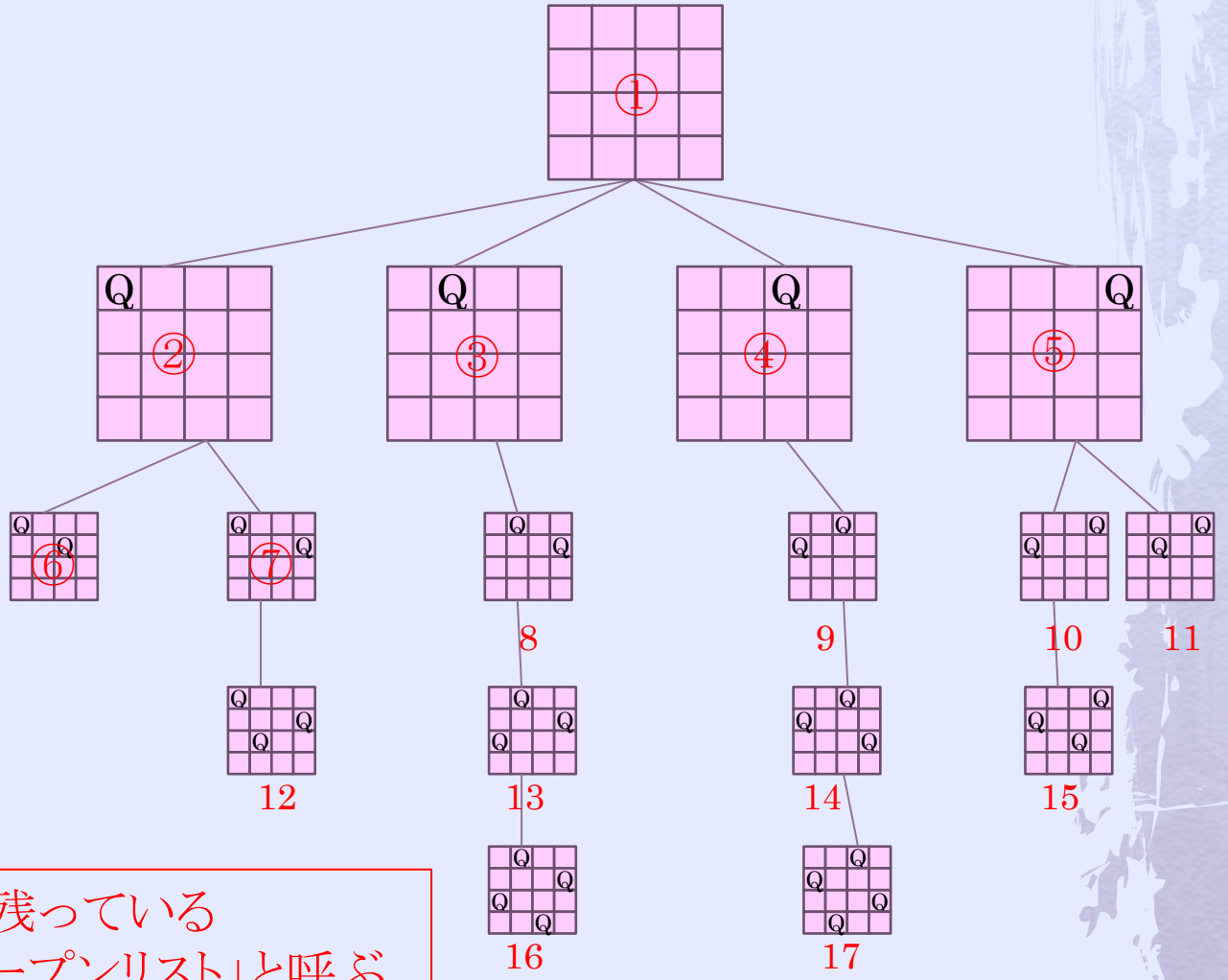




探索の途中で、Queueに残っている
ノード(状態)の集合を「オープンリスト」と呼ぶ

幅優先探索





探索の途中で、Queueに残っている
ノード(状態)の集合を「オープンリスト」と呼ぶ

Queue の状態

poll → 3
offer → [8 7 6 5 4]
poll → 4
offer → [9 8 7 6 5]

???

Queue の状態

poll → 3
offer → [8 7 6 5 4]
poll → 4
offer → [9 8 7 6 5]
poll → 5
offer → [11 10 9 8 7 6]
poll → 6
offer → [11 10 9 8 7]
poll → 7
offer → [12 11 10 9 8]
poll → 8
offer → [13 12 11 10 9]
poll → 9
offer → [14 13 12 11 10]
poll → 10
offer → [15 14 13 12 11]
poll →11
offer → [15 14 13 12]
poll →12
offer → [15 14 13]
poll →13
offer → [16 15 14]
poll →14
offer → [17 16 15]
poll →15
offer → [17 16]
poll → 16　　解です！ If（count = -1） 終了！Else（count>= 0）続き…
offer → [17]
poll → 17　　解です！
offer → [ ]　　終了！

幅優先探索プログラム: main

```
public class QueenWidth {
        // nQueen のサイズ
        int n = 4;

        //  全解探索の時には初期値を0にして、解の数を計算
        //  -1の時は、単独解の探索
        int count = 0;

        public static void main(String[] args) {
                new QueenWidth().start();
        }

        void start() {
                init(); // 初期化

                search(); // 探索

                if(count >= 0) {
                        System.out.println(count);
                }
        }
```

search()の引数がなくなっている以外は、QueenDepth と同じコード

幅優先探索プログラム
状態クラス定義と初期化

```
class State {
        int[] board;
        int depth;
}

/**
 * 初期化
 * @return 初期化された盤面
 */
void init() {
        State state = new State();
        state.depth = 0;
        state.board = new int[n];

        for(int i = 0; i < n; i++) {
                state.board[i] = -1;
        }

        offer(state);
}
```

内部クラス
盤面の配列と、探索の深さを格納

Queueに探索のルートノードを入れて、初期化完了

幅優先探索プログラム: 探索

```
boolean search() {
        State state = poll();
        // Queueにデータがなくなったら終了
        if(state == null) return false;
        // ブランチの数だけ探索を行う
        for(int i = 0; i < n; i++) {
                State state0 = copy(state); // 中間状態の生成
                state0.board[state.depth] = i;
                // 中間状態を評価し、真なら次の深さを探索
                if(check(state0)) {
                        state0.depth++; // depth を1増加させる
                        if(state0.depth == n) {// 正解に到達
                                print(state0);
                                if(count >= 0) {
                                        count++;
                                } else {
                                        // 単独解が見つかったら終了
                                        return true;
                                }
                        } else {
                                offer(state0);
                        }
                }
        }
        return search();  // 可能な中間状態をQueueに入れた後で、次のsearch()を行う
}
```

Queue からデータを取り出す

可能な下位ノードを展開し、Queue に格納

中間状態- OK
でも正解に到達ではない

探索を継続

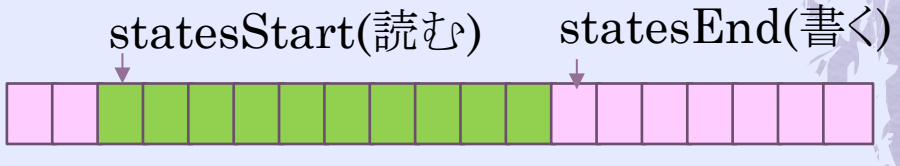
幅優先探索プログラム: Queueの例

データ構造とアルゴリズムで学ぶ可変長配列、リストを使うコードの方が良いが、今回は、簡単化のため、固定長配列で実装。
配列の大きさに注意が必要。

後ろのstateを入れる

先頭のstateを取る

```
// 中間状態を保存する配列
State[] states = new State[1000];
int statesStart = 0;
int statesEnd = 0;
void offer(State state) {
        states[statesEnd] = state;
        statesEnd++;
        if(statesEnd == states.length) statesEnd = 0;
        if(statesEnd == statesStart) {
                System.out.println("配列の大きさが不足しています");
                RuntimeException e
                        = new RuntimeException("Array Overflow");
                throw(e);
        }
}
State poll() {
        if(statesStart == statesEnd) return null;
        State state = states[statesStart];
        statesStart++;
        if(statesStart == states.length) statesStart = 0;
        return state;
}
```



幅優先探索プログラム: その他

```
    boolean check(State s) {
            // 深さ優先探索のコードと同等なので省略
    }

    void print(State s) {
            // 深さ優先探索のコードと同等なので省略
    }

    // 中間状態のコピー
    State copy(State b) {
            State state = new State();
            state.board = new int[n];
            for(int j = 0; j < n; j++) {
                    state.board[j] = b.board[j];
            }
            state.depth = b.depth;

            return state;
    }
}
```

状態をコピーした新しい状態を生成するメソッド

(演習，課題)

1. 配布プログラムを使ってし、深さ優先探索プログラムの search() メソッドを入力してコードを完成させ、動作させよ。
2. 深さ優先探索プログラムを改造して、幅優先探索プログラムを作成して、動作させよ。
    1. 余裕があれば、Queueの実装を、List を使うなど、独自の実装に変えてみよ。
3. start() メソッドの最初と最後で、System.nanoTime() を実行し、その差分を計算することで、深さ優先探索と幅優先探索の実行時間を計測し、考察せよ。
    1. Queen の数を変化させる
    2. 単解探索か全解探索かの条件を変える

(オプション課題)

- 幅優先探索プログラムは、再帰を利用して書かれているが、これを while 文に書き直して動作させよ。
- 幅優先探索プログラムの Queue を改造すると、深さ優先探索プログラムに変化する。どのように改造すればよいか考え、実装してみよ。