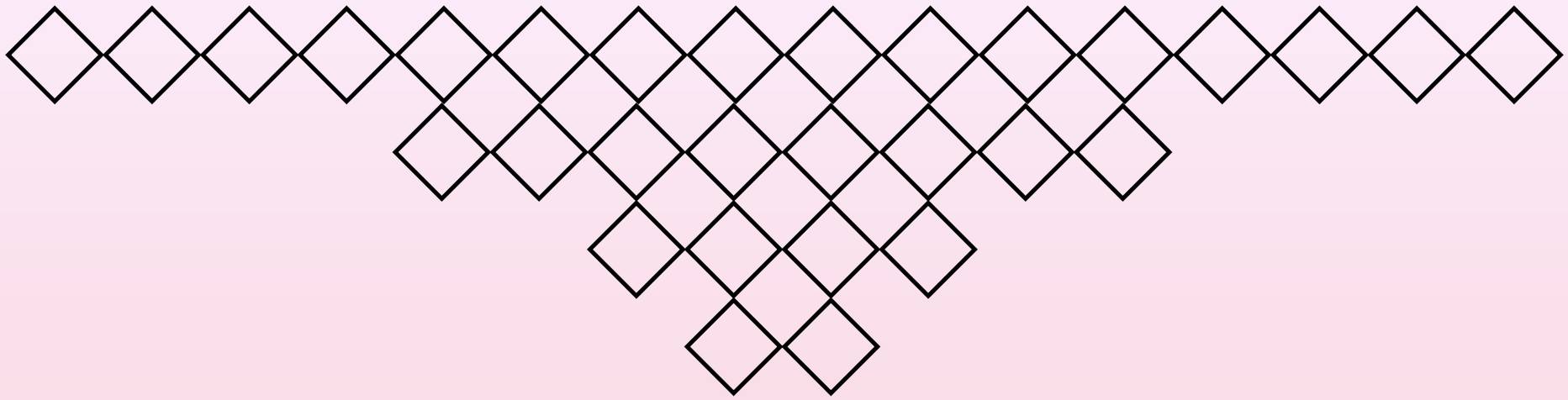
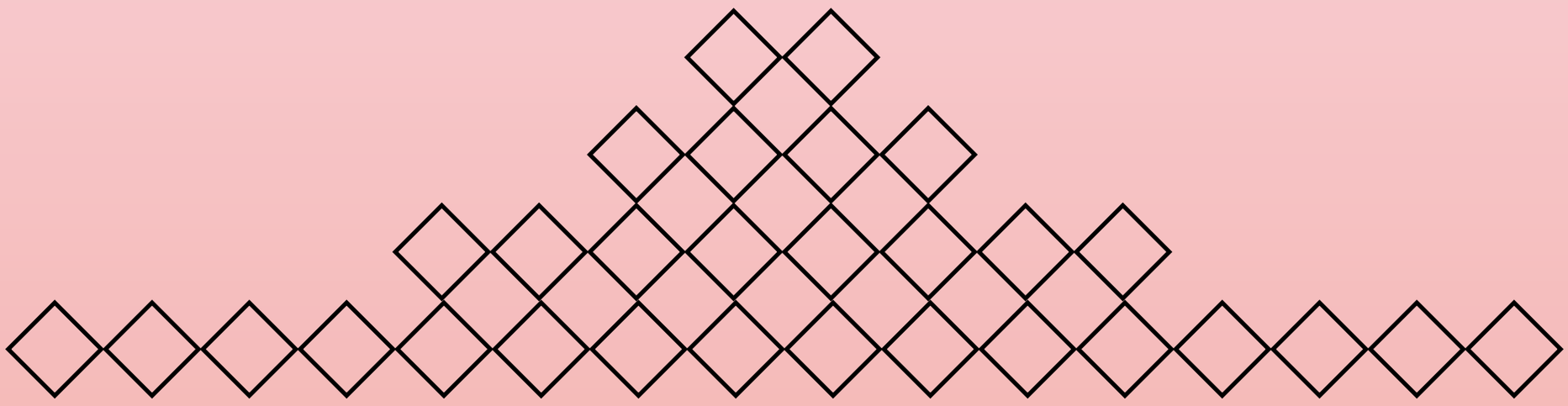


アルゴリズム・データ構造 I 第7回

木構造と深さ優先探索，幅優先探索

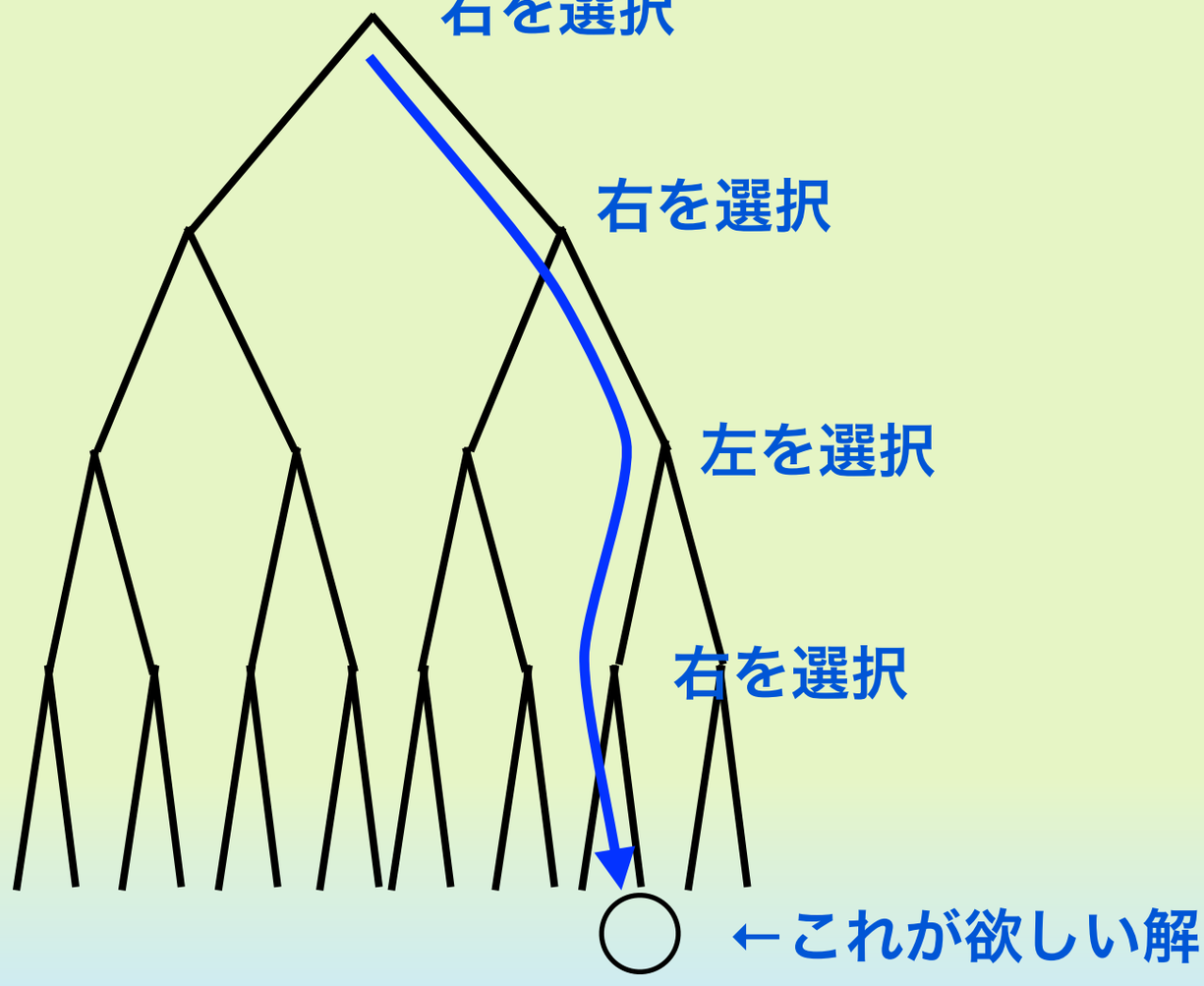
名城大学理工学部情報工学科

山本修身



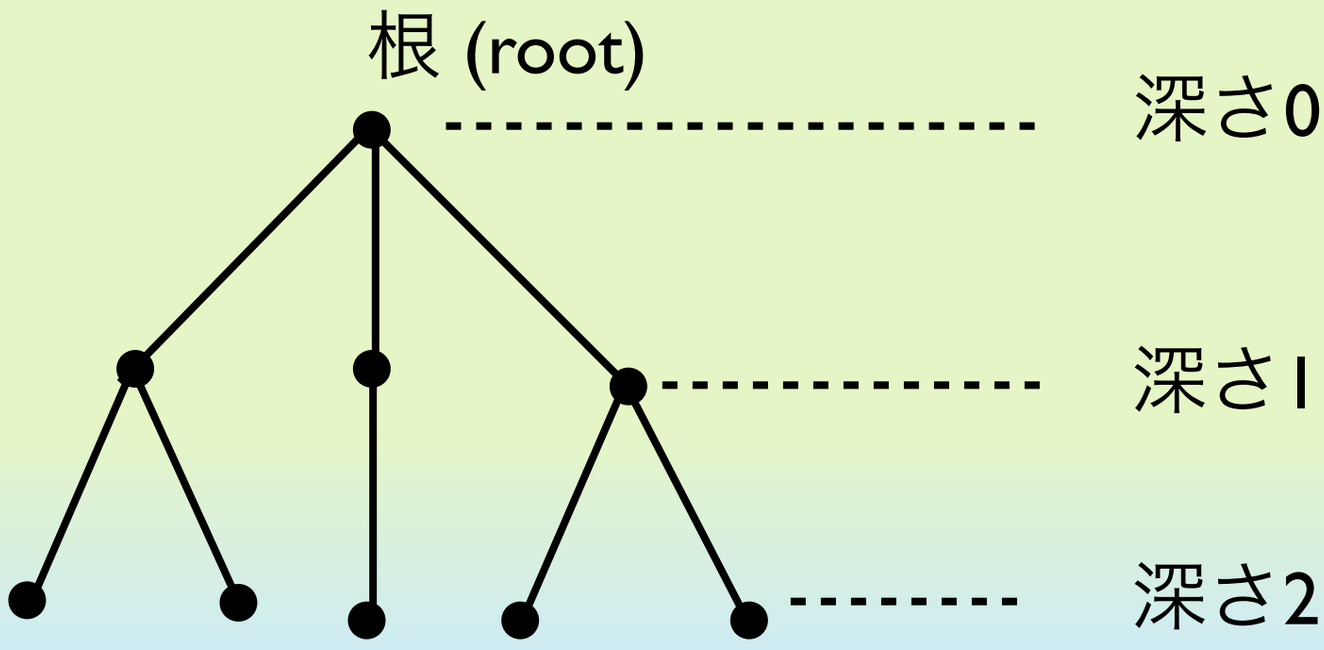
２分探索法とは（復習）

- 以下のような木構造（これを2分木という）があるとする．この木構造において，それぞれのレベルで2つうちのどちらかを選択すると，最終的に欲しい解に辿り着くことができる場合，この探索方法を**2分探索法（バイナリーサーチ：binary search）**と呼ぶ．



木構造の定義

- 木には唯一の根 (root) がある．任意のノードから根への経路はただ1通りである.
- あるノードの深さは，そのノードから根への経路上に存在する辺の個数である．根の深さは0となる.
- あるノードから深い方向に接続しているノードを子という．また，浅い方向に接続している唯一のノードを親と呼ぶ．子を持たないノードを葉と呼ぶ.



木構造の表現方法 (1)

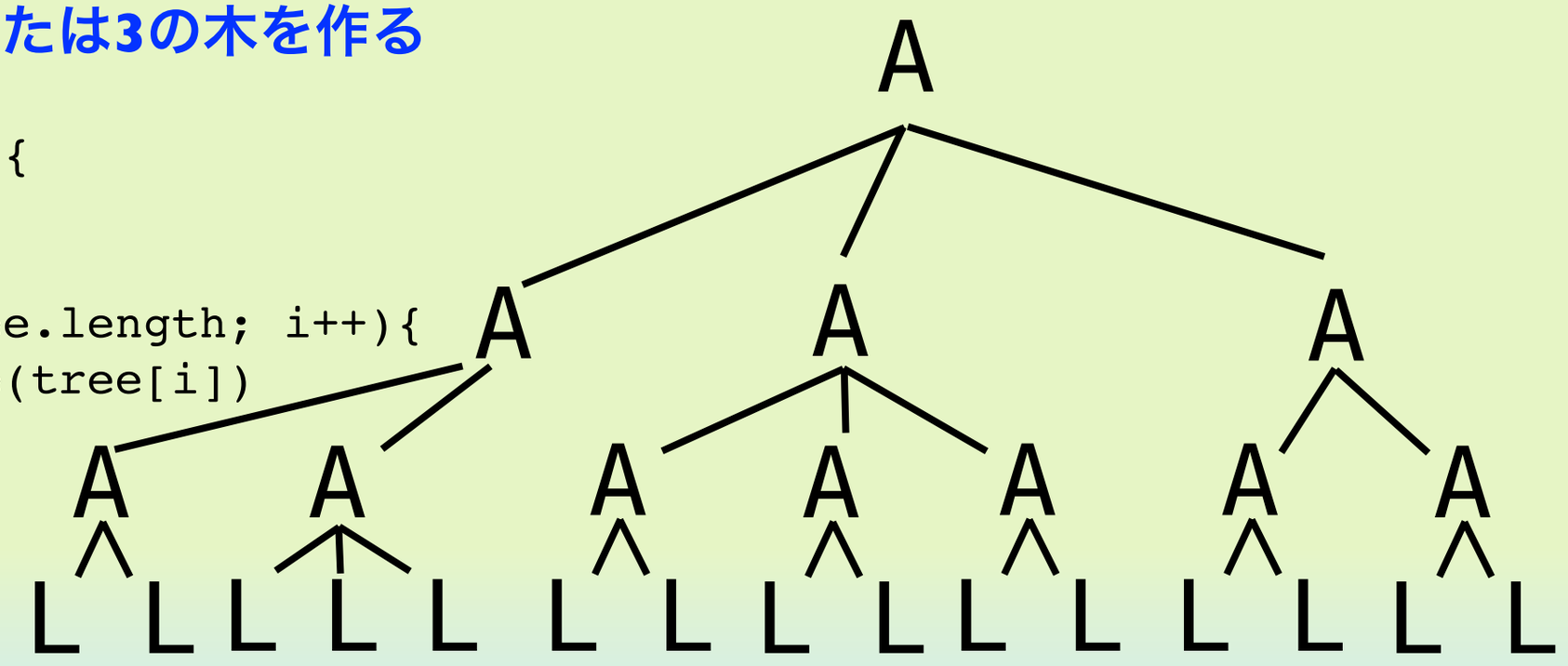
木構造を表現するには色々方法がある．あるノードを表現するのに，その子供のノードが根となる部分木を要素とする配列で表現することができる．**(ノードの記号，子，子）または（ノードの記号，子，子，子)**

```
function make_a_tree(n){
  if (n == 0) return ['L']
  var fun_out = 2
  if (Math.random()< 0.5) fun_out = 3
  var s = ['A']
  for (var i = 0; i < fun_out; i++){
    s.push(make_a_tree(n - 1))
  }
  return s
}

function print_tree(tree){
  var m = tree.shift()
  var s = "(" + m + " "
  for (var i = 0; i < tree.length; i++){
    s += "," + print_tree(tree[i])
  }
  s += ")"
  return s
}

puts(print_tree(make_a_tree(3)))
```

出力例

```
(A ,(A ,(A ,(L ),(L )),(A ,
(L ),(L ),(L ))),(A ,(A ,
(L ),(L )),(A ,(L ),(L )),
(A ,(L ),(L ))),(A ,(A ,
(L ),(L )),(A ,(L ),(L ))))
```

出次数が2または3の木を作る



木構造の表現方法 (2)

- もっと見やすく別の出力方法を考える.

```
function make_a_tree(n){
  if (n === 0) return ['L']
  var fun_out = 2
  if (Math.random()< 0.5) fun_out = 3
  var s = ['A']
  for (var i = 0; i < fun_out; i++){
    s.push(make_a_tree(n - 1))
  }
  return s
}

function spaces(n){
    var s = ""
    for (var i = 0; i < n; i++) s += " "
    return s
}
```

```
function print_tree2(tree, level){
    var len = tree.length
    puts( spaces(level * 2) + tree[0] )
    for (var i = 1; i < len; i++){
        print_tree2(tree[i], level + 1)
    }
}

print_tree2(make_a_tree(3), 0)
```

```
A
  A
    A
      L
      L
      L
    A
      L
      L
      L
  A
    A
      L
      L
    A
      L
      L
```

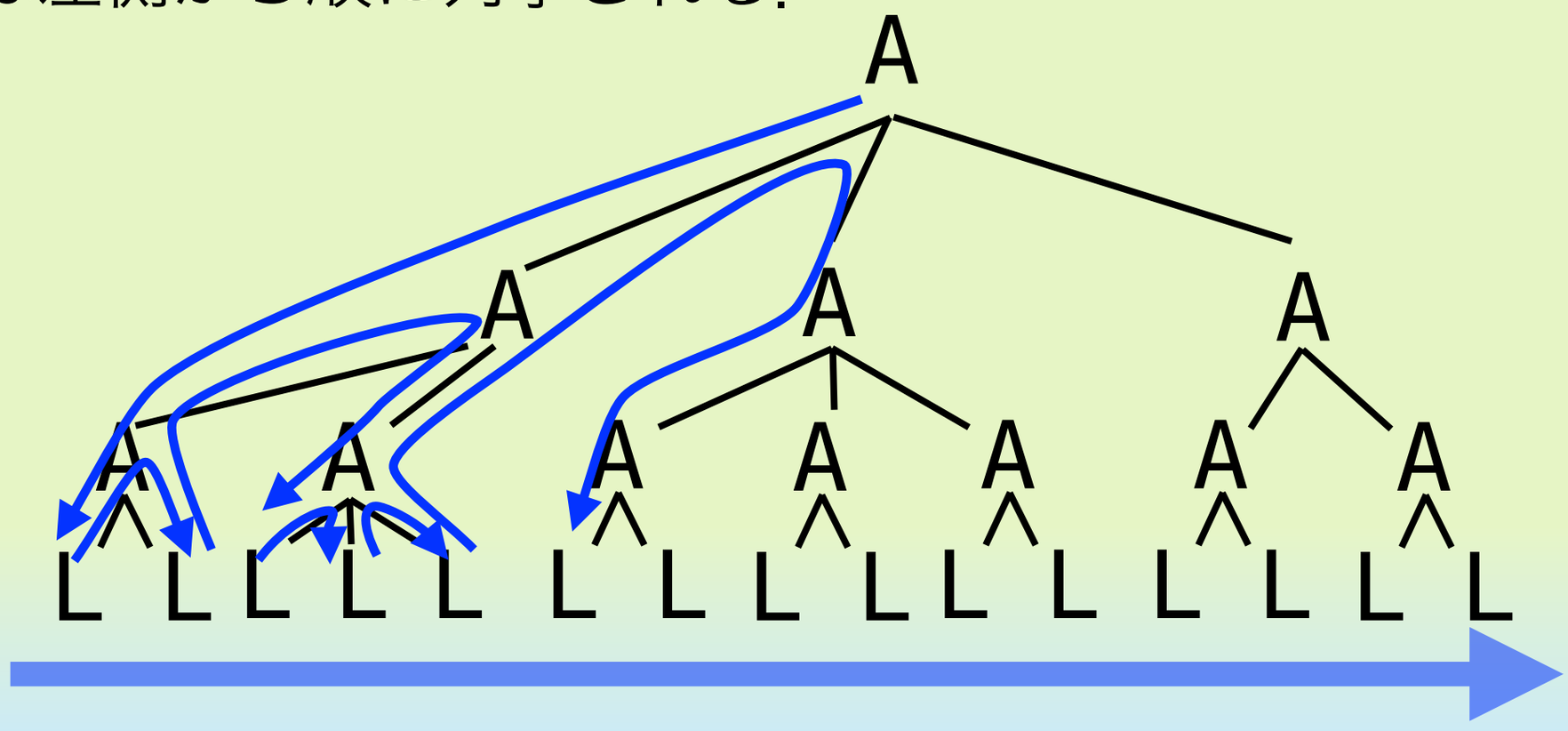
ランダムに生成された木はprint_tree2によって左のように出力される. 右にプリントされるほど深い位置の要素を表す.

与えられた木のノードを順に走査する方法：深さ優先探索 (1)

木構造を陽に作って計算するか否かは別にして，木構造を頭の中に描いてアルゴリズムを設計することは非常に多い．その際，木に対して必要な操作として，ノードを順に走査 (scan) することが重要になる．ノードをどの順に走査するかは任意性があり，よく使われる方法として，**深さ優先探索（depth-first search)** と**幅優先探索 (breadth-first search)** がある．ここではプログラミングが簡単な深さ優先探索から見て行く．

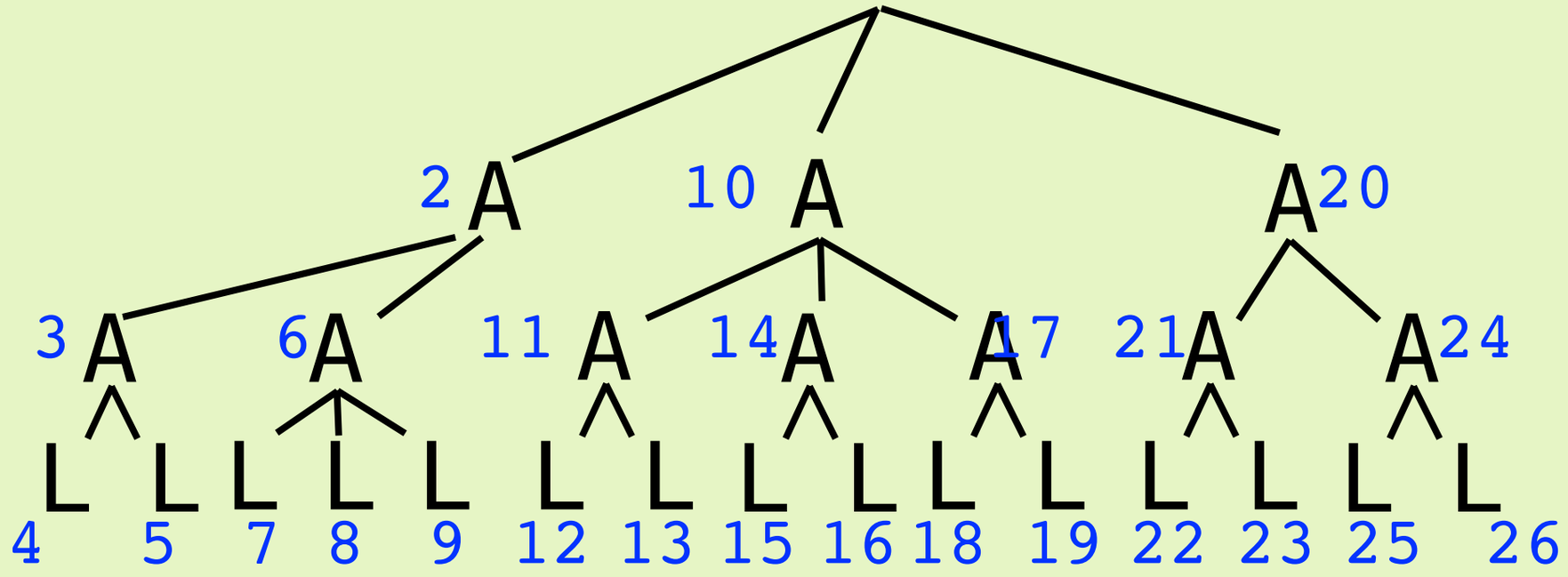
与えられた木のノードを順に走査する方法：深さ優先探索 (2)

- 深さ優先探索は葉に出会うまで，深く進み，深く進めなくなったら，一段階もどって，別の方向に深く進む．深く進むことを優先する探索法である．
- もし，木が無限に深い場合には止まらなくなる．したがって，この方法は使えない．
- 葉は左側から順に列挙される．



与えられた木のノードを順に走査する方法：深さ優先探索 (3)

- 葉ではないノードの順位も含めて以下のような順番で探索が行われる.



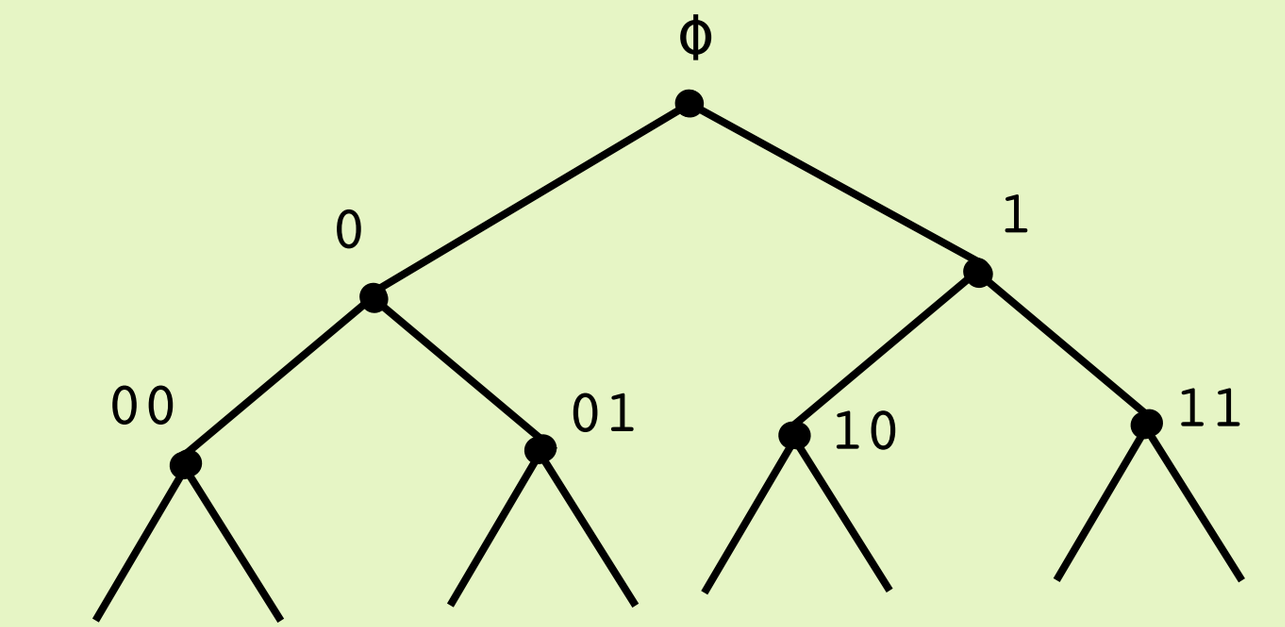
与えられた木のノードを順に走査する方法：深さ優先探索 (4)

- 深さ優先探索を行うためのプログラム

```
function DFS(node){
    nodeについての処理;
    for (nodeのすべての子供childについて) {
       DFS(child)
    }
}
```

◇◇深さ優先探索の例：すべての2進数を表示する◇◇

- n桁の2進数をすべて表示する.



出力

```
0000000000
0000000001
0000000010
..........
..........
1111111101
1111111110
1111111111
```

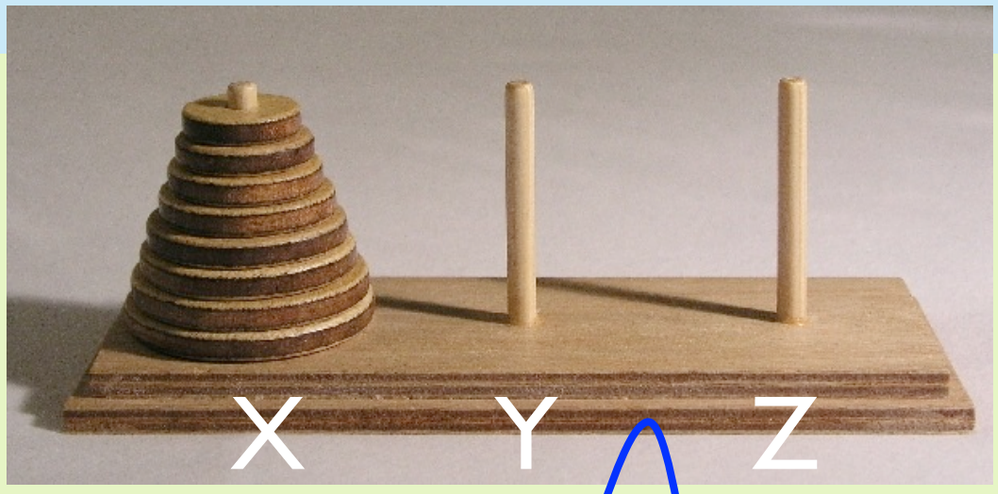
```
var b = []
var n = 10
function print_bin(){
  var s = ""
  for (var i = 0; i < n; i++){
    s += b[i]
  }
  puts(s)
}
```

nビットを表示する

k番目のビットを決定する

```
function binary(k){
  if (k >= n){
    print_bin();
  } else {
    b[k] = 0
    binary(k + 1)
    b[k] = 1
    binary(k + 1)
  }
}

binary(0)
```

深さ優先探索の例：ハノイの塔 (1)



- ルール１：小さな円盤の上に大きな円盤を乗せてはいけない
- ルール２：１度に２枚以上移動させてはいけない.

n>1のとき

上のn枚をXからYを経由してZへ移動させる

≡ 上の n - l 枚をXからZを経由してYへ移動させる

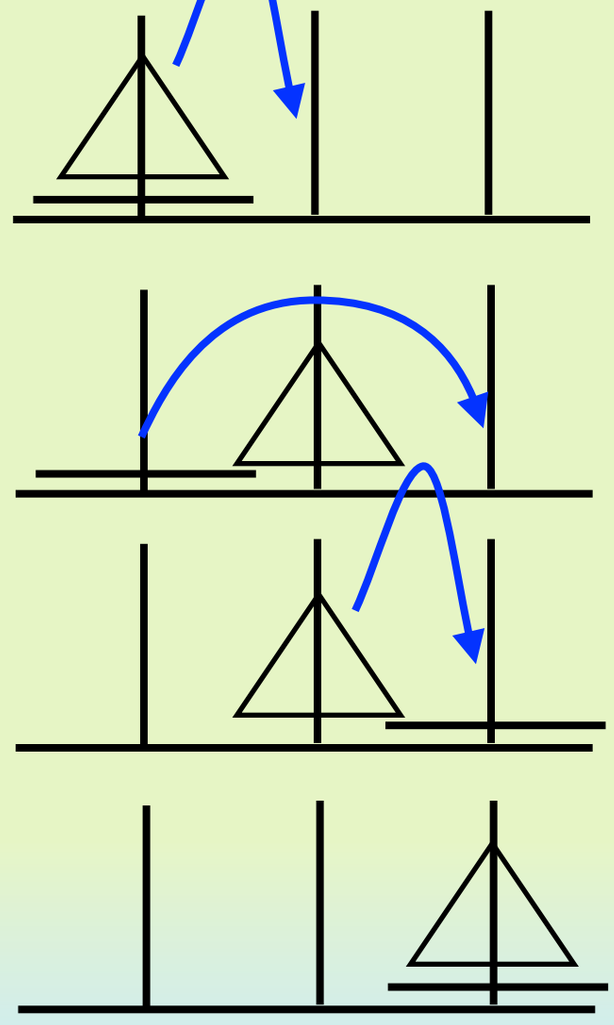
上からn枚目のディスクをXからZへ移動させる

上の n - l 枚をYからXを経由してZへ移動させる

n=1のとき

上のn枚をXからYを経由してZへ移動させる

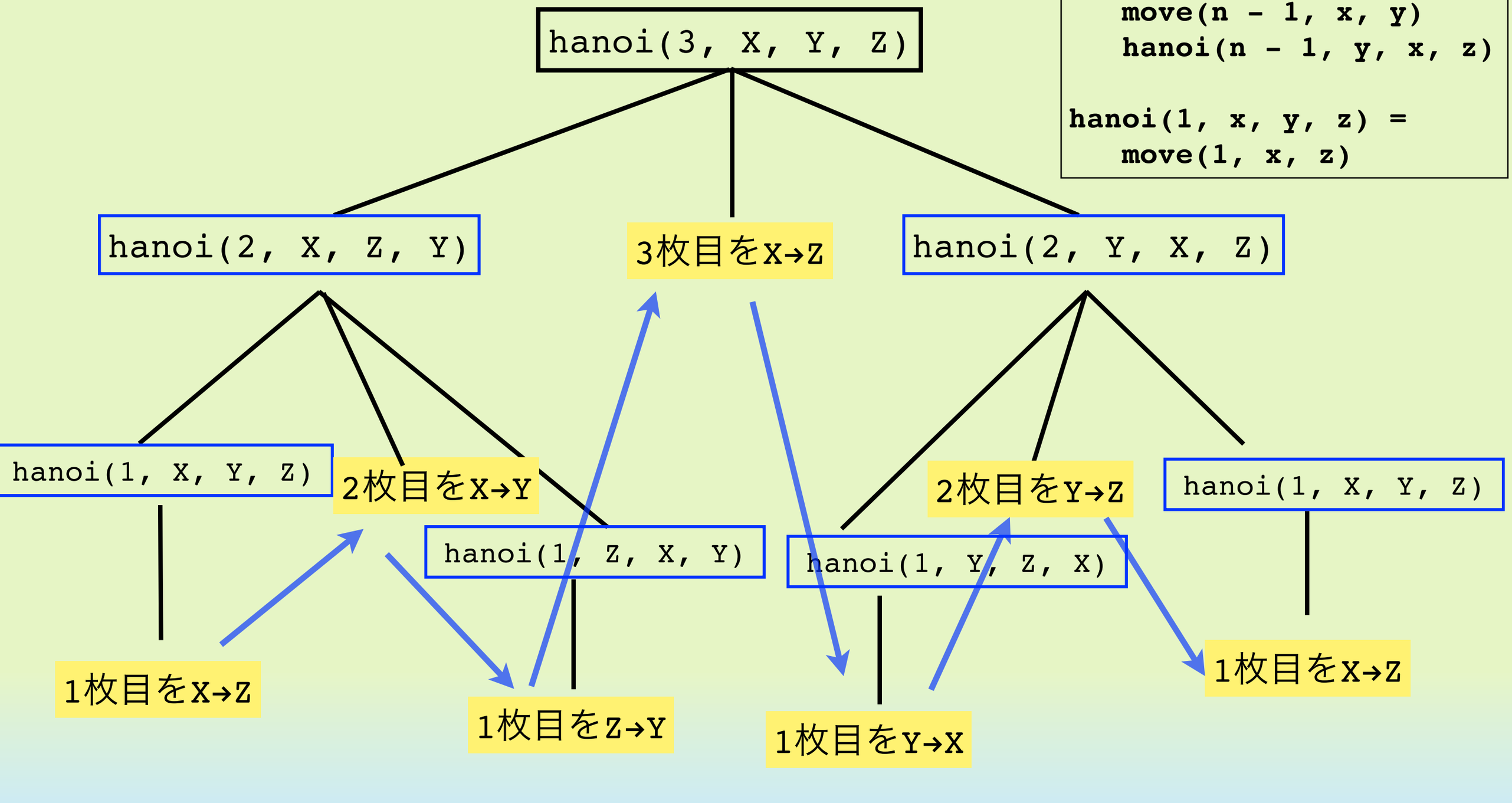
≡ 上からl枚目のディスクをXからZへ移動させる



深さ優先探索の例：ハノイの塔 (2)

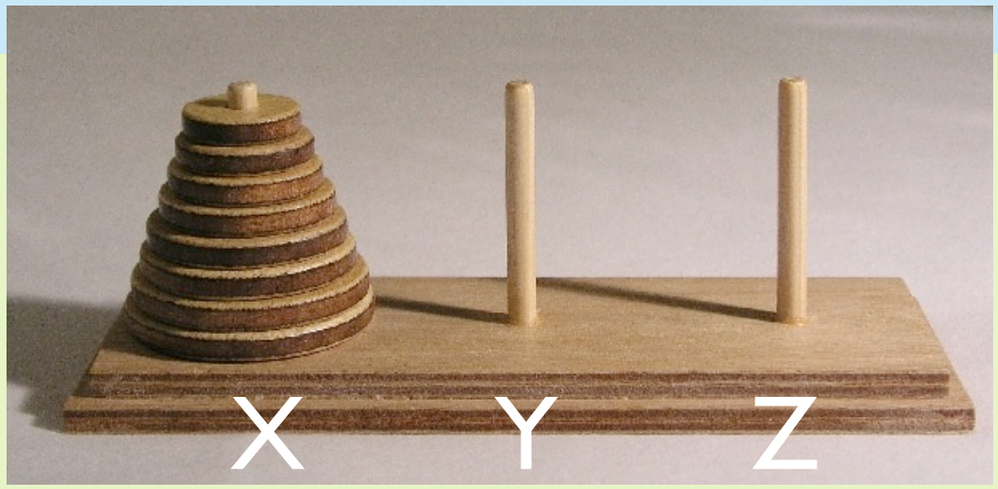
- ハノイの塔を解く仮定は木構造として書けば以下のようになる

```
hanoi(n, x, y, z) =
    hanoi(n - 1, x, z, y)
    move(n - 1, x, y)
    hanoi(n - 1, y, x, z)

hanoi(1, x, y, z) =
    move(1, x, z)
```



深さ優先探索の例：ハノイの塔 (3)

- ハノイの塔のプログラム




```
function hanoi(n, x, y, z){
  if (n == 0) return
  else {
    hanoi(n - 1, x, z, y);
    puts("ディスク " + n + " を " + x + " から " +
      z + " へ移動" )
    hanoi(n - 1, y, x, z);
  }
}

hanoi(6, 'X', 'Y', 'Z')
```

ディスク 1 を X から Y へ移動
ディスク 2 を X から Z へ移動
ディスク 1 を Y から Z へ移動
ディスク 3 を X から Y へ移動
ディスク 1 を Z から X へ移動
.....
.....
.....
ディスク 1 を Z から X へ移動
ディスク 3 を Y から Z へ移動
ディスク 1 を X から Y へ移動
ディスク 2 を X から Z へ移動
ディスク 1 を Y から Z へ移動

クイックソート：深さ優先探索の例 (1)

ソーティングはアルゴリズムの世界では古典的な話題である．非常に多くのソーティングアルゴリズムが提案されている．ソーティングは色々なアルゴリズムの基礎となるので非常に重要である．

```
[3, 5, 4, 8, 2, 3, 1]
```

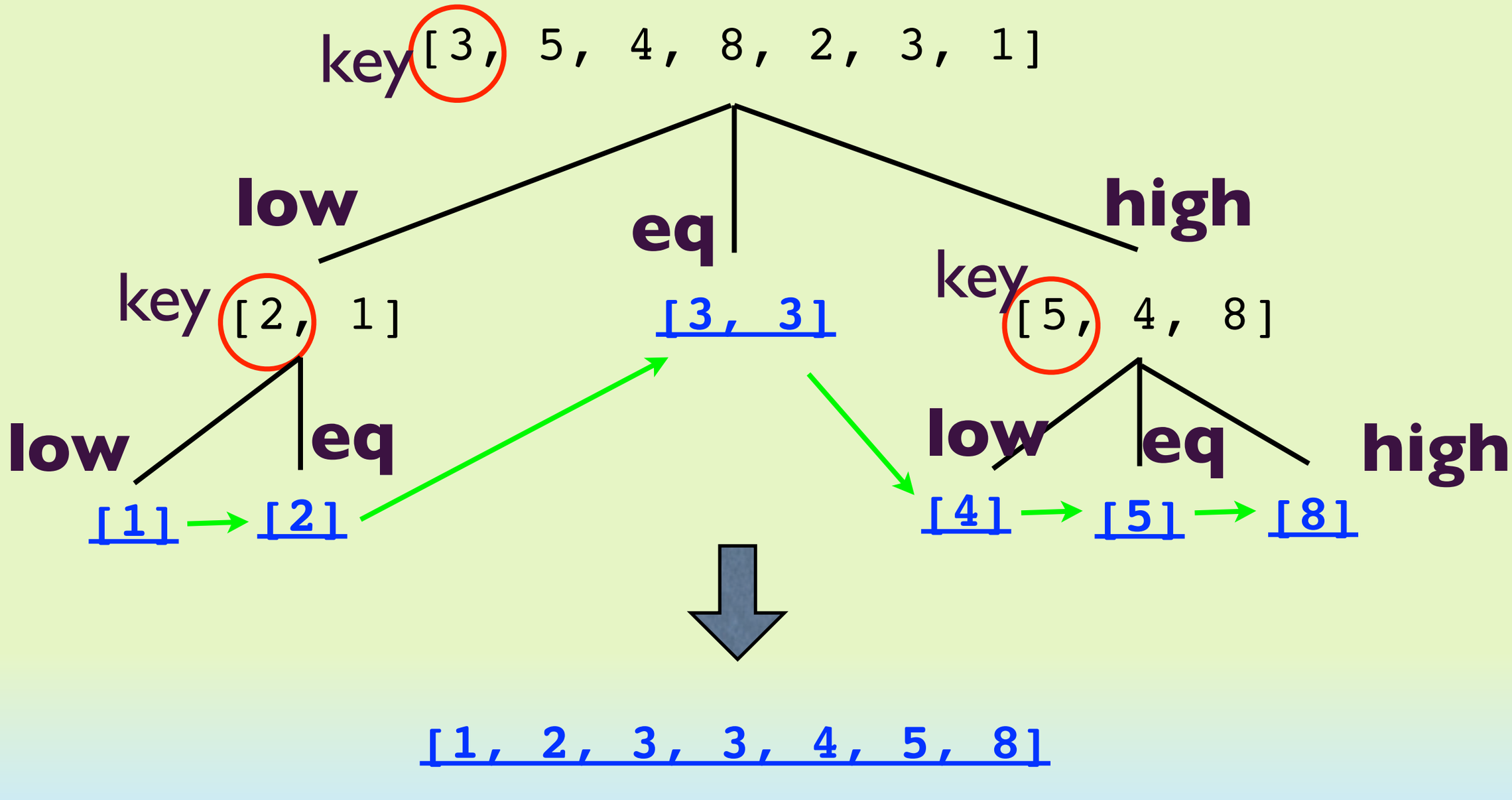
↓ **小さい順に並べ替える**

```
[1, 2, 3, 3, 4, 5, 8]
```

クイックソートはC.A. R Hoareによって見つけられた有名なアルゴリズムである．考え方は単純で簡単にプログラミングすることができる．実は，このソートアルゴリズムは前述のハノイの塔のアルゴリズムと構造が同じである．深さ優先探索の一例となっている．

クイックソート：深さ優先探索の例 (2)

長いリストの最初の要素をkeyとして，< key, ==key, > keyの３種類に分けてさらに<key, >keyのグループは再帰的に調べる．葉の出現した順に集めるとソートされている．



クイックソート：深さ優先探索の例 (3)

- プログラミングの例

```
ans = []
function qsort(lst){
  if (lst.length <= 1) ans = ans.concat(lst)
  else{
    var key = lst[0]
    var low = []
    var eq = [key]
    var high = []
    for (var i = 1; i < lst.length; i++){
      var m = lst[i]
      if (key > m) low.push(m)
      else if (key < m) high.push(m)
      else eq.push(m)
    }
    qsort(low)
    ans = ans.concat(eq)
    qsort(high)
  }
}
qsort([3, 5, 4, 8, 2, 3, 1])
puts(ans)
```

→ 1,2,3,3,4,5,8

[a, b, c].concat([d, e, f]) = [a, b, c, d, e, f]

◇◇ クイックソート：深さ優先探索の例 (4) ◇◇

クイックソートはランダムに並んだデータに対して，非常に高速にソートすることが知られている．名前のとおり「クイック（高速）」である．以下の例は，同じプログラムを用いて1000個の実数をソートしたところ．１秒以内に並び替えが完了している．

計算量については別の回に説明する

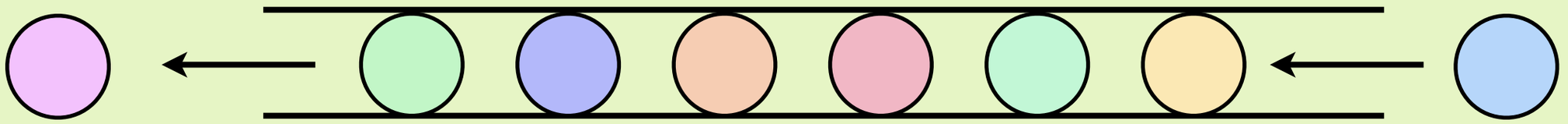
```
var ans = []
prob = []
for (var i = 0; i < 10000; i++){
   prob.push(Math.random())
}
t1 = new Date()
qsort(prob)
t2 = new Date()
puts(ans[0] + " " + ans[5000] + " " + ans[9999])
puts((t2 - t1) + " ms")
```

↓

```
0.0001643446251519131 7 0.5021072253070673 0.9998156603363418
87 ms
```

◇◇もう一つの探索方法：幅優先探索 (1)◇◇

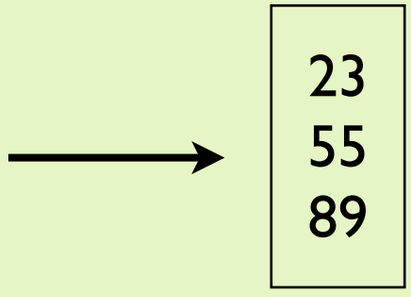
- データ構造：キュー（queue; FIFO: **F**irst **I**n **F**irst **O**ut)



取り出すときは左から

入れるときは右から

```
a = []
a.push(23)
a.push(55)
puts(a.shift())
a.push(89)
puts(a.shift())
puts(a.shift())
```

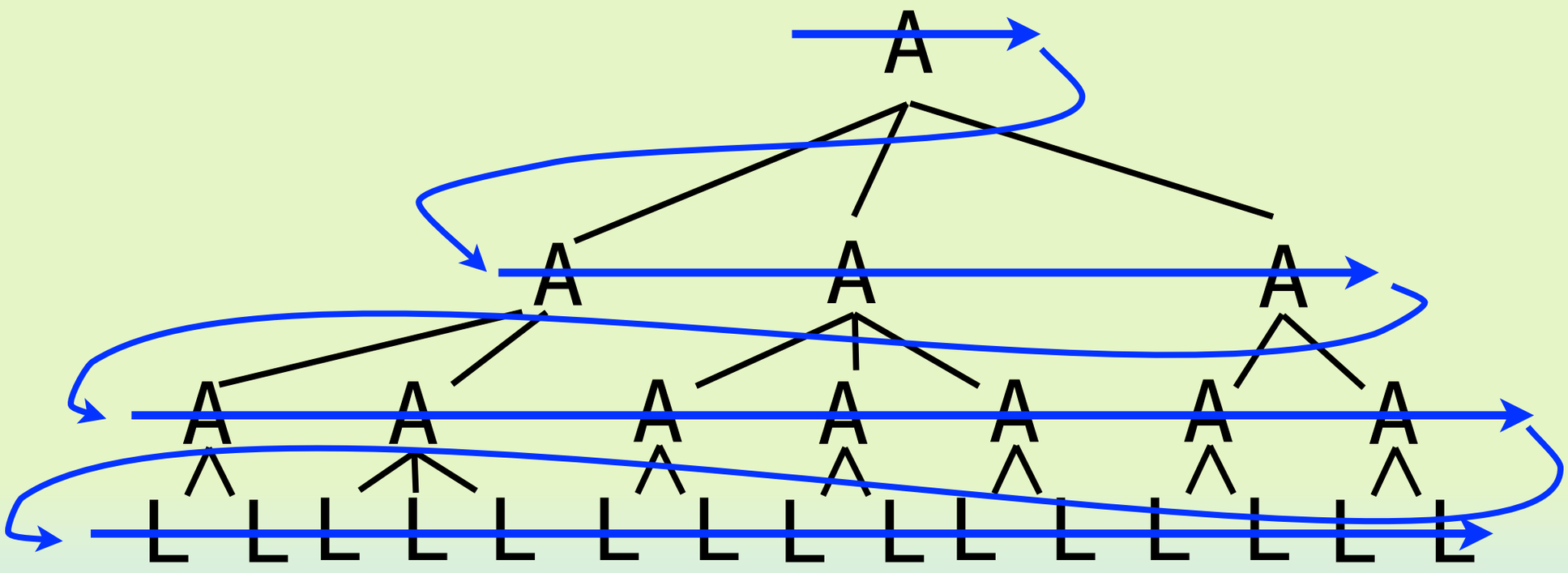


JavaScriptでは，配列にキューの機能がついている．データを入れる場合にはpush(), データを取り出す場合にshift()を用いる

◇◇もう一つの探索方法：幅優先探索 (2)◇◇

もう一つの木の辿り方：深さ優先探索では深いところまでまず，降りて行き，降りられなくなったら戻るという方法をとった．もう一つの方法は，根 (root) に近いところから順に見て行く方法である．木の深さが無限大でも探索することができる．

幅優先探索：BFS (breadth first search)



◇◇もう一つの探索方法：幅優先探索 (3)◇◇

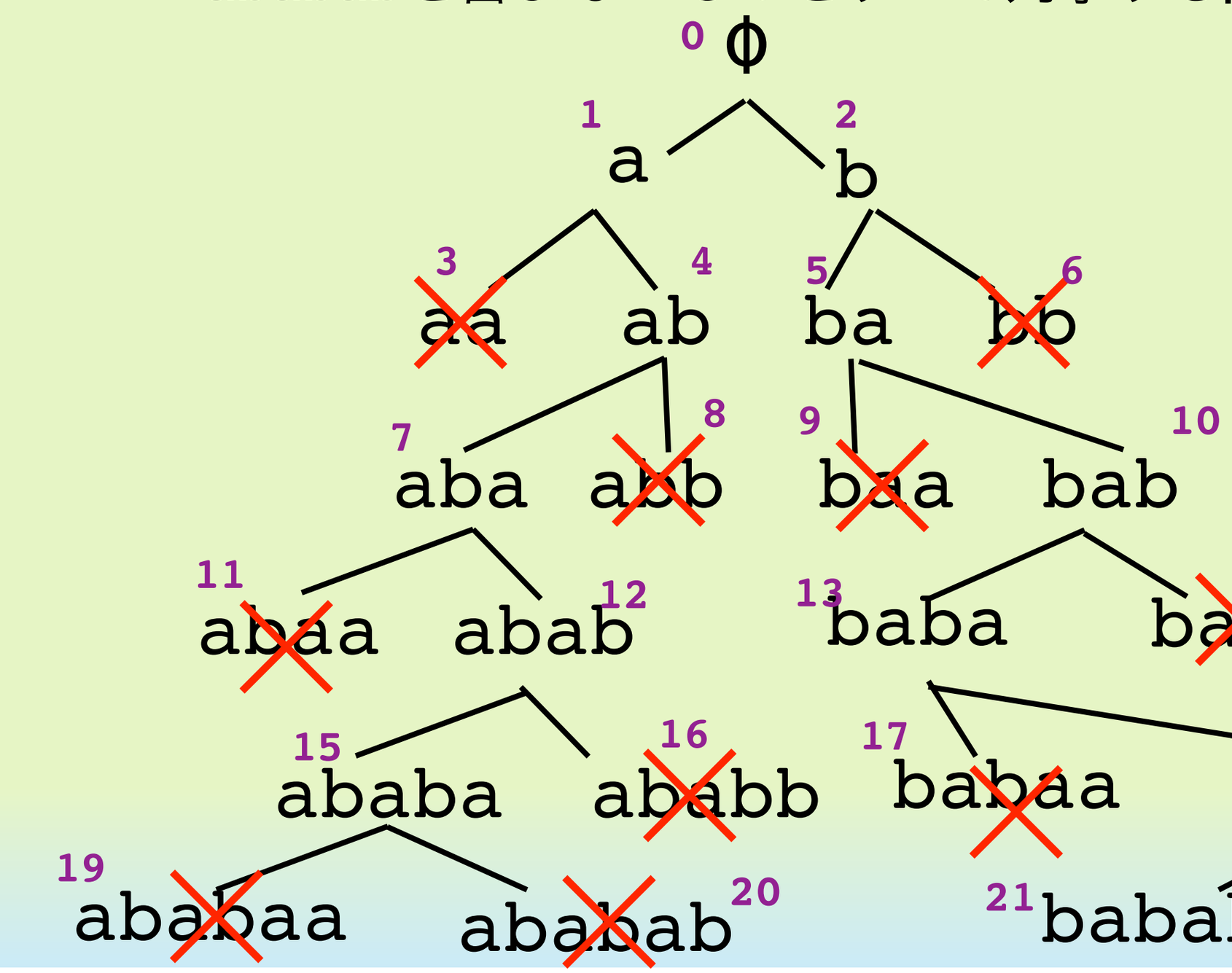
- 深さ優先探索が再帰的なプログラミングで実現できるのに対して，幅優先探索は繰り返しを用いて実現できる.

```
queue = [根のノード]
while (queue.length > 0){
    node = queue.shift()
    node 個別の処理

    nodeの子供をqueueに付け加える

}
```

この方法ではqueueの長さが爆発する可能性がある

幅優先探索を利用した例 (1)

- 問題： a, b 2文字を用いて作られる文字列のうち，aa, bb, abababを含まないものをすべて列挙する．



Φ←[Φ]

a←[a, b]

b←[b, aa, ab]

aa←[aa, ab, ba, bb]

ab←[ab, ba, bb]

ba←[ba, bb, aba, abb]

⋮

幅優先探索を利用した例 (2)

- この問題は幅優先探索でないと解けないわけではないが，この探索法を用いれば，文字列を短いものから順に列挙することができる.

```
> "abcdef".search("de")
3
> "abcdef".search("x")
-1
>
```

```
function judge(c){
   var m = ["aa", "bb", "ababab"]
   for (var i = 0; i < m.length; i++){
      if (c.search(m[i])>= 0) return true
   }
   return false
}
```

[],[a],[b],[ab],[ba],[aba],
[bab],[abab],[baba],
[ababa],[babab],[bababa]

```
function bfs(){
   var queue = [""]
   var ans = []
   while (queue.length >= 1){
      var c = queue.shift()
      if (judge(c)){}
      else {
         ans.push('[' + c + ']')
         queue.push(c + "a")
         queue.push(c + "b")
      }
   }
   return ans
}

puts(bfs())
```

幅優先探索を利用した例 (3)

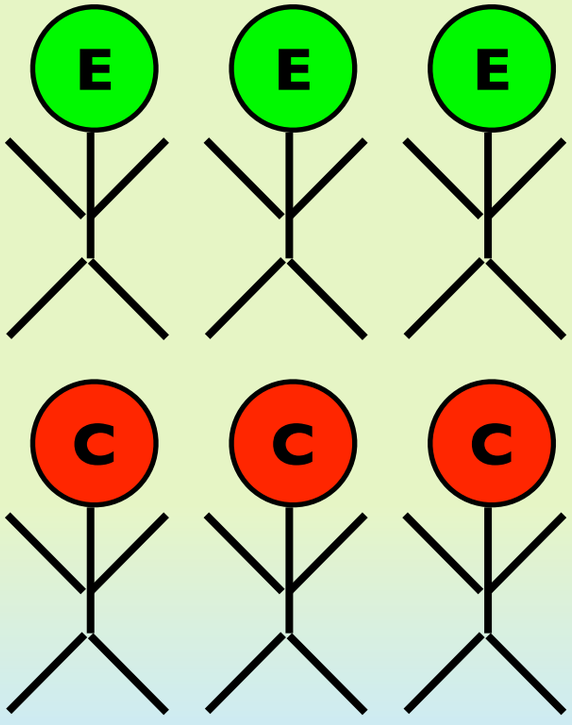
```
function judge(c){
    var m = ["aa", "bb", "ababab"]
    for (var i = 0; i < m.length; i++){
   if (c.search(m[i])>= 0) return true
    }
    return false
}

var ans = []
function dfs(s){
    if (judge(s)) return
    else {
        ans.push('[' + s + ']')
   dfs(s + "a")
   dfs(s + "b")
    }
}

dfs("")
puts(ans)
```

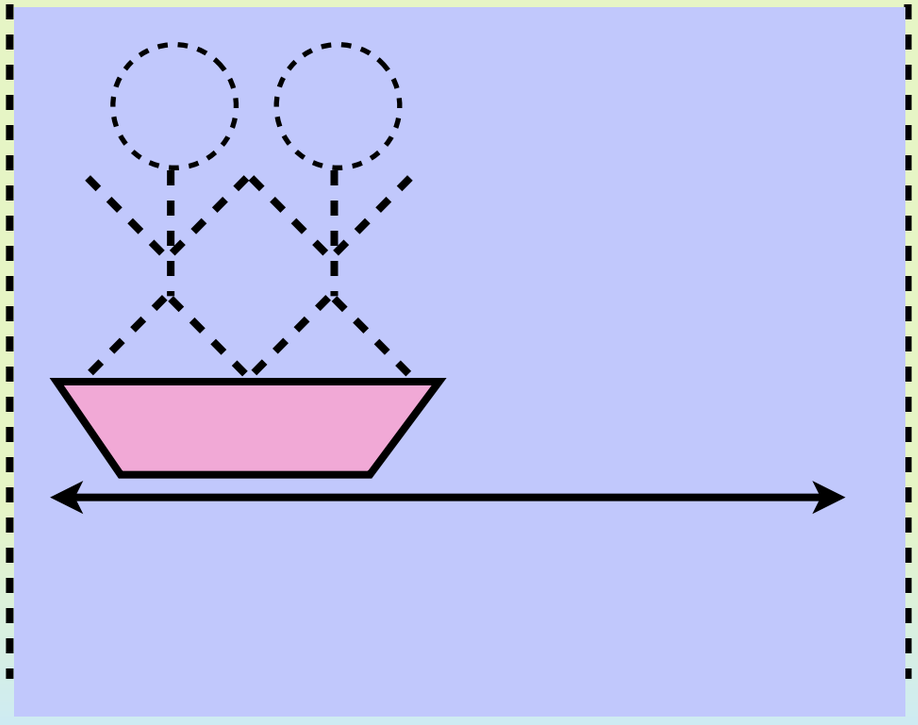
- 同じ問題を深さ優先探索で解けば左のようになる.

[],[a],[ab],[aba],[abab],
[ababa],[b],[ba],[bab],
[baba],[babab],[bababa]

◇◇ 宣教師と人食い人種の問題 (1) ◇◇

3人の宣教師と3人の人食い人種が川を渡ろうとしている．ボートは1つしかなく一度に2人までしか乗れない．また，各岸で宣教師の人数よりも人食い人種の人数が多くなると宣教師は食べられてしまう．合計6人が左から右の岸へ無事渡るためにはどうすればよいか？ボードに乗る回数はなるべく少なくしたい．





◇◇ 宣教師と人食い人種の問題 (2) ◇◇

宣教師 (E)，人食い人種 (C)

状態の表現方法： リストによる表現.

[左岸のCの人数，左岸のEの人数,
ボートの位置 (左岸:0，右岸: 1)
右岸のCの人数，右岸のEの人数]

初期状態： [3, 3, 0, 0, 0]

↓

終了状態： [0, 0, 1, 3, 3]

◇◇ 宣教師と人食い人種の問題 (3) ◇◇

- それぞれの岸における移動の方法．船には最大2名しか乗れないので宣教師，人食い人種それぞれの人数は以下のとおり．

```
var move_list = [[1, 0], [0, 1], [1, 1], [2, 0], [0, 2]]
```

mvはmove_listの要素の一つ

```
function move_state(st, mv){
    var [c0, e0, boat, c1, e1] = st
    var [mc, me] = mv
    if (boat == 0){
        if (c0 >= mc && e0 >= me)
            return [c0 - mc, e0 - me, 1, c1 + mc, e1 + me]
        else return null
    } else {
        if (c1 >= mc && e1 >= me)
            return [c0 + mc, e0 + me, 0, c1 - mc, e1 - me]
        else return null
    }
}
```

移動不可の場合にはnullを返す

◇◇ 宣教師と人食い人種の問題 (4) ◇◇

- ある状態で宣教師が食べられてしまうか否かの判定.

宣教師が1名以上いて，かつ人食い人種の方が多い場合，食べられてしまう．各岸についてチェックする必要がある.

```
function check_cond(st){
    if ((st[0] > st[1] && st[1] > 0) ||
        (st[3] > st[4] && st[4] > 0)) return false
    else return true
}
```

◇◇ 宣教師と人食い人種の問題 (5) ◇◇

タブーリストを利用する．両岸を含めた状態がすでにどこかで出た状態であれば，改めてその状態について考えない．その状態は別状態からすでに到達可能であることを意味するから．

```
var tabu_list = []
```

eq_stateで同じ状態か否かをチェック

```
function eq_state(st1, st2){
    for (var i = 0; i < 5; i++)
        if (st1[i] !== st2[i]) return false
    return true
}
```

すでに出たものでなければタブーリストに登録する

```
function put_tabu_list(st){
    for (var i = 0; i < tabu_list.length; i++){
        if (eq_state(tabu_list[i], st)) return false
    }
    tabu_list.push(st)
    return true
}
```

◇◇ 宣教師と人食い人種の問題 (6) ◇◇

```
var queue = []

var init_state = [3, 3, 0, 0, 0]
var final_state = [0, 0, 1, 3, 3]
put_tabu_list(init_state)
queue.push([init_state, null])

while(true){
    if (queue.length == 0){
        puts("no solutions.")
        return
    }
    var ele = queue.shift()
    var [st, parent] = ele
    if (eq_state(st, final_state)) break;
    else if (check_cond(st)){
        for (var i = 0; i < move_list.length; i++){
            var m = move_state(st, move_list[i])
            if (m == null) continue
            if (put_tabu_list(m) == false) continue
            queue.push([m, ele])
        }
    }
}
```

幅優先探索を実行.

キューには状態とその状態を作った一つ前の状態をリストにしたものが入っている.

終了状態が見つかったら止まる.

◇◇ 宣教師と人食い人種の問題 (7) ◇◇

最後に終了状態から初期状態までの移動過程を取り出す．それを表示する．

```
var answer = []
while (ele != null){
    var [st, parent] = ele
    answer.unshift(st)
    ele = parent
}
for (var i = 0;  i < answer.length; i++)
    puts(i + " : " + answer[i])
```

実行結果：

```
 0 : 3,3,0,0,0
 1 : 2,2,1,1,1
 2 : 2,3,0,1,0
 3 : 0,3,1,3,0
 4 : 1,3,0,2,0
 5 : 1,1,1,2,2
 6 : 2,2,0,1,1
 7 : 2,0,1,1,3
 8 : 3,0,0,0,3
 9 : 1,0,1,2,3
10 : 2,0,0,1,3
11 : 0,0,1,3,3
```

宣教師と人食い人種の問題 (8)

- 結果の解釈　　● 人食い人種　●宣教師

左岸		右岸
● ● ● ● ●	ボ	
● ● ● ●	ボ	● ●
● ● ● ● ●	ボ	●
● ● ●	ボ	● ● ●
● ● ● ●	ボ	● ●
● ●	ボ	● ● ● ●
● ● ● ●	ボ	● ●
● ●	ボ	● ● ● ●
● ● ●	ボ	● ● ●
●	ボ	● ● ● ● ●
● ●	ボ	● ● ● ●
	ボ	● ● ● ● ● ●

実行結果：

```
 0 : 3,3,0,0,0
 1 : 2,2,1,1,1
 2 : 2,3,0,1,0
 3 : 0,3,1,3,0
 4 : 1,3,0,2,0
 5 : 1,1,1,2,2
 6 : 2,2,0,1,1
 7 : 2,0,1,1,3
 8 : 3,0,0,0,3
 9 : 1,0,1,2,3
10 : 2,0,0,1,3
11 : 0,0,1,3,3
```