

Part. 4 データ構造と再帰

データを格納するときの論理的構造をデータ構造という。
迷路を作りながら、「グラフ」「木」というデータ構造と、「最小全域木」を求める「クラスカル法」というアルゴリズムを学ぶ。

目次

1. 基本的なデータ構造
2. 迷路を作る
 1. グラフ
 2. 迷路と木構造
 3. class
 4. 最小全域木の構成（クラスカル法）
 5. ループの判定
3. 迷路を解く（最短経路問題）
 1. 幅優先探索
 2. ダイクストラ法

4-1. 基本的なデータ構造（お話）

（１）配列

最も基本的なデータ構造。一行に並んだデータを記録する。データの挿入や削除には弱い。

実装

```
int[] a;  
int n;      (データ数)  
a[k]の後ろにデータ x を追加  
for(int i=n-1; i>k; i--) a[i+1] = a[i];    (後ろにずらす)  
a[k] = x;  
n++;
```

特徴

- ・ランダムアクセス（i 番目の要素はなにか調べる）が $O(1)$ でできる
- ・中途へのデータの挿入、削除は、それより後のデータを 1 つずつずらさないといけないので $O(N)$

（２）リスト

一行に並んだデータを記録する。 $a[0] \rightarrow a[3] \rightarrow a[7] \rightarrow a[2] \rightarrow \dots$ のように変則的に記録できる。next と bck には次と前を記録するので、この例では $\text{next}[3]=7$ で、 $\text{bck}[3]=0$ となる。データの挿入と削除には強いが、ランダムアクセス（i 番目の要素にいきなりアクセスする）はできず、先頭から順にたどらないといけない。

実装

```
int[] a;  
int n;      (データ数)  
int[] next;  next[i]には、i 番目のデータの次のデータが何番目かを記録する  
int[] bck;   bck[i]には、i 番目のデータの前のデータが何番目かを記録する  
a[k]の後ろにデータ x を追加  
a[n] = x;  
bck[n] = k;    bck[next[k]] = n;    もともと k の次だったデータの前を n に  
next[n] = next[k];  next[k] = n;    n の次を、もともとの k の次に  
n++;
```

特徴

- ・ランダムアクセスはできず、先頭か末尾からたどっていくしかない ($O(N)$)
- ・データの挿入は、a の末尾にデータを追加し、next, bck を適切に繋ぎ変えればいいので $O(1)$ でできる。
- ・データの削除も、next, bck のつなぎ変えで $O(1)$ でできる。


配列

	0	1	2	3	4	5	6	7	8	9	10
配列 a	3	1	4	1	5	9	2	6			

7をa[2]の後ろに挿入

	0	1	2	3	4	5	6	7	8	9	10
配列 a	3	1	4	7	1	5	9	2	6		

リスト



	0	1	2	3	4	5	6	7	8	9	10
配列 a	0	0	1	1	6	3	9	4	2	5	
nxt	5	-1	9	7	1	3	8	2	4	6	
bck	-1	4	7	5	8	0	9	3	6	2	

3→1→4→1→5→9→2→6
をリストに記憶した例
a[0], a[1]は先頭と末尾を表している

nxtは次のデータが何番目にあるか
bckは前のデータが何番目にあるか
を表している

	0	1	2	3	4	5	6	7	8	9	10
配列 a	0	0	1	1	6	3	9	4	2	5	7
nxt	5	-1	9	7	1	3	8	10	4	6	2
bck	-1	4	10	5	8	0	9	3	6	2	7

7を4→1のところに挿入した

4→7→1となるので
4のnxtと、1のbckを書きかえる

(3) スタック

リストの機能を限定したもの。最初に入れたものが最初に出てくるシステム。

実装

```
int[] a;  
int b;    ここにはデータの末尾を記録する。  
データの追加  
    a[b] = x;  
    b++;  
データの取り出し  
    b--;  
    return a[b];
```

特徴

- ・ 末尾へのデータの追加、末尾からのデータ取り出しのみを許す。
- ・ どちらも $O(1)$

例

- ・ 本の山。上においた本から先に取り出せる。

(4) キュー

リストの機能を限定したもの。最初に入れたものが最後に出てくるシステム。

実装

```
int[] a;  
int f;    データの先頭(front)  
int b;    データの末尾(bottom)  
データの追加  
    a[b++] = x;    (a[b]に x を代入した後で x に 1 を足す)  
データの取り出し  
    return a[f++];    (a[f]を return して f に 1 を足す)
```

特徴

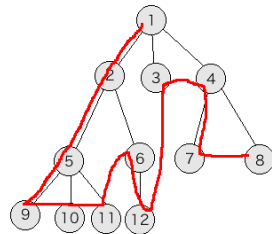
- ・ 先頭へのデータ追加、末尾からのデータ取り出しのみを許す。
- ・ どちらも $O(1)$

例

- ・ 順番待ちの行列。先に並んだ人が先に出ていく。

・ スタックと深さ優先探索

行きにデータ取り出して、子をスタックに追加



	0	1	2	3	4	5	6	7	8	9	10
スタック	1										

	0	1	2	3	4	5	6	7	8	9	10
スタック	4	3	2								

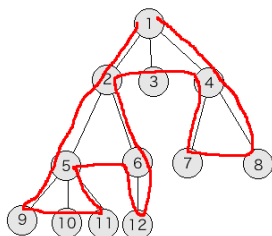
	0	1	2	3	4	5	6	7	8	9	10
スタック	4	3	6	5							

	0	1	2	3	4	5	6	7	8	9	10
スタック	4	3	6	11	10	9					

	0	1	2	3	4	5	6	7	8	9	10
スタック	4	3	6	11	10						

	0	1	2	3	4	5	6	7	8	9	10
スタック	4	3	6	11							

行きに子をスタックに追加し、帰りにデータを取り出す



	0	1	2	3	4	5	6	7	8	9	10
スタック	1										

	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2							

	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2	6	5					

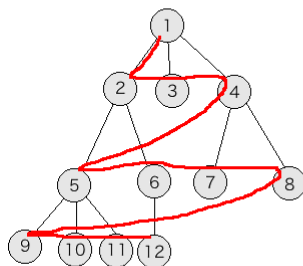
	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2	6	5	11	10	9		

	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2	6	5					

	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2	6						

	0	1	2	3	4	5	6	7	8	9	10
スタック	1	4	3	2	6	12					

・ キューと幅優先探索



	0	1	2	3	4	5	6	7	8	9	10
キュー	1										

	0	1	2	3	4	5	6	7	8	9	10
キュー		2	3	4							

	0	1	2	3	4	5	6	7	8	9	10
キュー			3	4	5	6					

	0	1	2	3	4	5	6	7	8	9	10
キュー				4	5	6					

	0	1	2	3	4	5	6	7	8	9	10
キュー					5	6	7	8			

	0	1	2	3	4	5	6	7	8	9	10
キュー						6	7	8	9	10	11

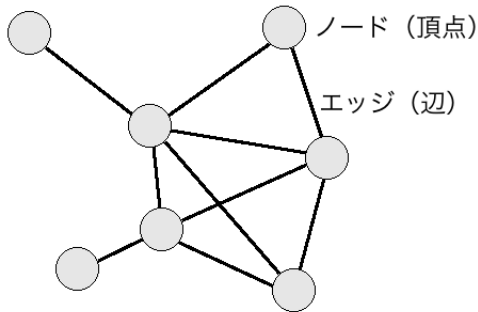
4-2. 迷路を作る

(1) グラフ

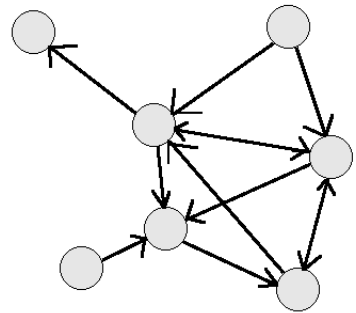
グラフ構造は、都市の地理的關係、人間關係、路線図、ネット、水道管など、複数の要素が互いにネットワークで結ばれているような關係を表すのに使われる。

・有向グラフと無向グラフ

無向グラフ



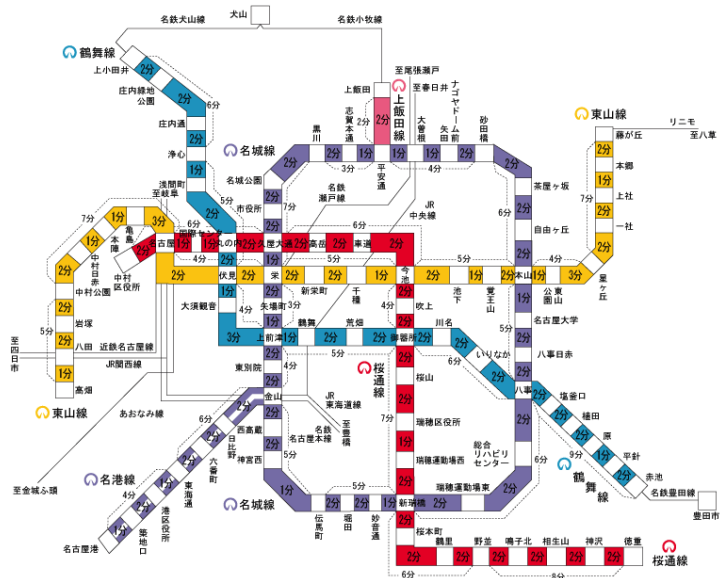
有向グラフ



辺に向きがあるかないかで、無向グラフ、有向グラフという。無向グラフは、全ての辺が双方向の辺になった有向グラフともみなせる。

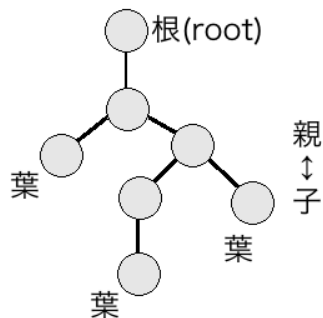
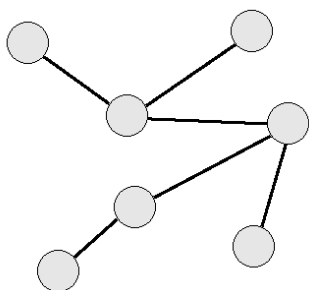
・辺の重み

距離や所要時間、値段、水量、好感度など、辺に数値データが附属しているとき、それを「辺の重み」という。右は所要時間のデータが辺の重みとして記された地下鉄路線図の例（名古屋市交通局）。

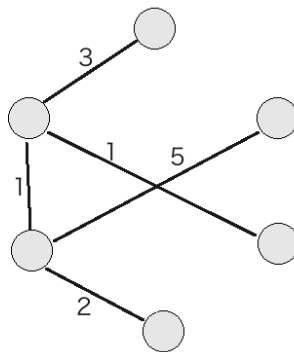
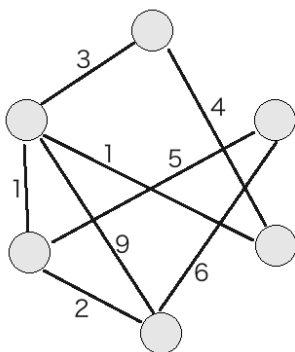


・ 木

すべてのノードが結ばれていて、閉路（ループ）を含まないようなグラフを木という。木はうまく配置を整えてやると、右図のように階層構造で表すことができる。



・ 全域木

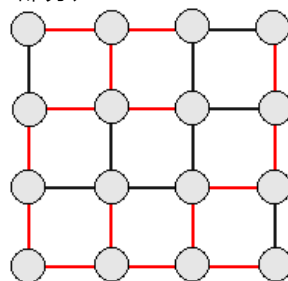
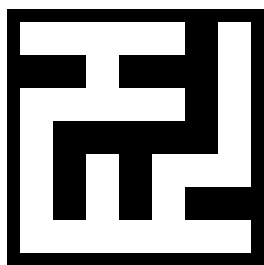


グラフからうまく辺を除去していくと木を作ることができる。このようにして作られた、もとのグラフに含まれるノードを全て含むような木を「全域木」という。重みの付けられた無向グラフから作られた全域木で、重みの和が最小になるようなものを「最小全域木」という。迷路の生成ではこの最小全域木を利用する。

上の図では、右が左のグラフの最小全域木となっている。

(2) 迷路と木構造

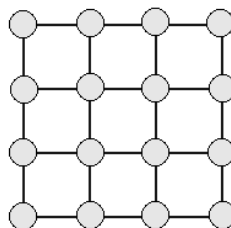
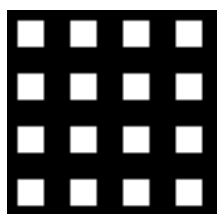
迷路の「通路」を辺、「分岐点」を頂点とみたとすると、迷路をグラフとして考えることができる。(下の右図では赤線が通路の部分)



迷路には複数の解答があってはならない。これはグラフ上でループがあってはならないことと対応する(スタートからゴールまで二通りの解答があったら、両者を結合した道にはループが含まれる)。また、迷路を画面全体に広げるなら、使われないノードは無い方がいい。従って、迷路を作るには、マス目状に並んだノードたちからなるグラフの全域木をランダムに生成できればよさそうである。

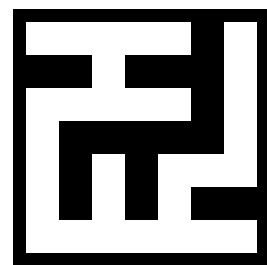
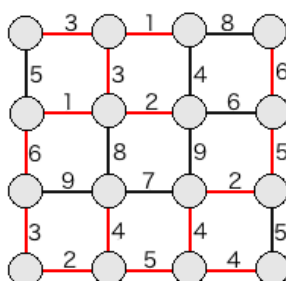
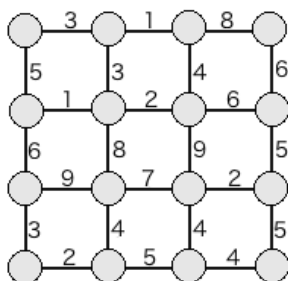
・迷路作成の手順

(i) 分岐点だけをつくり、マス目状に辺をはって右のようなグラフにする。



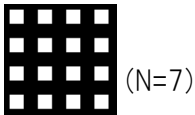
(ii) 辺にランダムに重みをつける

(iii) 最小全域木を構成する



(3) プログラミング開始

- (i) グローバル変数として、迷路の一边の長さを表す整数 N を用意しよう。
- (ii) 迷路を表す二次元配列を用意する。int 型の二次元配列を用意しよう。
0 が壁、1 が通路とする。まずは偶数番目のマスだけ通路にする。
壁を黒、通路を白で描画してやると、次のようになる。



- (iii) 次に辺を作るのだが、クラスという手法を使う。

・ class とは

クラスというのはその名の通り「組」のことで、「いくつかの変数を組にした新しい型」と考えると良い。本当は命令も組にできるが、ここでは変数だけを考える。

例えば int 型の変数 x, y, z を組にしたクラス Point を作ったとしよう。

```
class Point{
    int x, y, z;
}
```

これは新しい型として使うことができ、Point 型の変数（本当はインスタンスという）p を作ってみる。

```
Point p;
```

インスタンスは基本型でないので、配列と同じように new をする必要がある。

```
p = new Point();
```

p に属している変数 x, y, z はドット(.)記号を介して使うことができる。

```
p.x = 3;
p.y = p.x;
```

クラスを使うことで、次のようにプログラムを簡単にすることができる。

```
int[] x = new int[N];
int[] y = new int[N];
int[] z = new int[N];
```

```
int tmp = x[i];
x[i] = x[j];
x[j] = tmp;
tmp = y[i];
y[i] = y[j];
y[j] = tmp;
tmp = z[i];
z[i] = z[j];
z[j] = tmp;
```



```
Point[] ps = new Point[N];
for(int i=0; i<N; i++) ps[i] = new Point( );
```

```
Point tmp = ps[i];
ps[i] = ps[j];
ps[j] = tmp;
```

配列のnewとクラスのnewの2つのnewが必要

i番目とj番目を入れ替えている

・ class の使い方まとめ

構文 class

- ・ クラスを定義する


```
class Hoge{
    int a, b;
    int[] x;
    void hoge hoge(){
        ...
    }
}
```

`class`と書いた後にクラス名を書く。通例大文字ではじめる。

変数たち。配列や他のクラスが入ってても良い。

命令たち。クラスに含まれる変数进行操作できる。
- ・ 作ったクラス型の変数（インスタンス）を作る


```
Hoge h = new Hoge();
```
- ・ インスタンスの持ってる変数、命令にアクセスする


```
h.a
h.x[3]
h.hoge hoge()
```

さて、辺（エッジ、Edge）を表すクラスを作る。

```
class Edge{
    int fx, fy;
    int tx, ty;
    float w;
    int f;
}
```

これに含まれる変数たちには次のような意味を持たせる。

fx, fy, tx, ty ... 分岐点 a[fx][fy] から a[tx][ty] に伸びる通路を表す。
 w ... 最小全域木を作るときに使う辺の重み。乱数を代入したい。
 f ... 壁にする辺は 0, 通路にする辺は 1 とする。

次に、すべての辺を格納する配列を作る。辺の数は、迷路の一辺が N だったから、分岐点が $(N/2) \times (N/2)$ で、右端と下端以外の各頂点から右と下に 1 本ずつ辺が出てくるから、 $(N/2) \times (N/2) \times 2 - (N/2) - (N/2) + 1$ 本あるはずだが、細かいことは気にしないで $N \times N$ 分確保しておけば OK だろう。その代わりに `int` 型の変数 `ne` で辺の本数を記録することにする。

```
Edge[] edges;
int ne;
```

グローバルに

```
edges = new Edge[N*N];
```

こっちはsetupブロックの中

次に辺を作る。辺は、全ての分岐点から右方向と下方向に作ればいいが、右端と下端に関しては例外処理が必要になる。

・ $[i][j]$ から $[i+2][j]$ に伸びる辺を作る処理の例 (for 文と if 文の内側に)

```
edges[ne] = new Edge();
edges[ne].fx = i;
edges[ne].fy = j;
edges[ne].tx = i+2;
edges[ne].ty = j;
edges[ne].w = random(0, 1);
edges[ne].f = 0;
ne++;
```

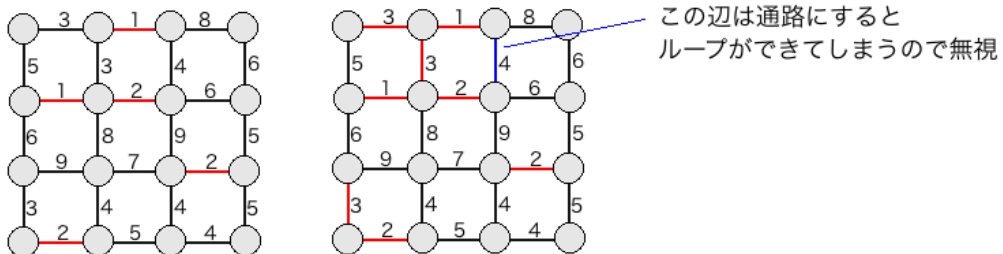
重み w は乱数で与える。最初は全て壁なので f は 0 にする。

あとは最小全域木を構成するだけである (これが一番難しいが)。

(4) 最小全域木の構成 (クラスカル法)

最小全域木の構成にはクラスカル法と呼ばれるアルゴリズムを用いる。

- (i) 辺を重みが小さい順にソートする。
- (ii) 重みが小さい辺から順番にみていく。
通路にすると閉路 (ループ) ができてしまうような辺は壁のまま、
そうでない辺は通路にする。
- (iii) 一番重みが大きな辺までこの処理を行うと、最小全域木ができあがっている。



重みの小さい辺から通路にしていく

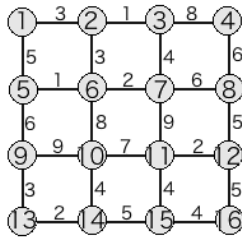
(i) のソートのところは Part3 で作ったクイックソートを少し書きかえるだけで良い。
配列 `edges` が重みが小さい順に並ぶようにしよう。

問題は (ii) のループができるかどうか調べるところで、これは一筋縄ではいかない。
1 つの方針として、次のようなアルゴリズムが考えられる。

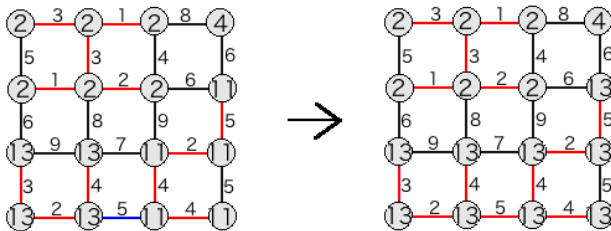
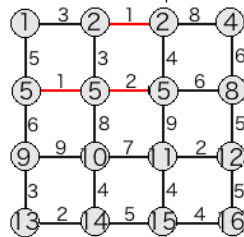
(5) ループの判定

- ・ ループ判定のアルゴリズム

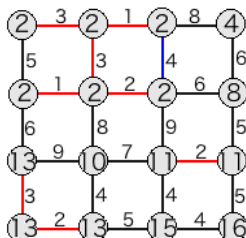
全ての頂点に違う番号をふる



連結された頂点を同じ番号にする



島同士がつながったときも、全て同じ番号にする
ここでは11が全て13に併合された



その辺を通路にするとループができるときは、
必ず同じ番号の頂点をつなごうとしている

だからそのようなときは
その辺は壁のままにする

K 会 2012 年度夏期講習 情報講座

左ページ中段の、11 と 13 をつなげるときに 11 をすべて 13 にするという処理を最も簡単に実現するには、すべての頂点を for 文でみて、11 のものを見つけたら 13 に変えれば良い。

この併合アルゴリズムでは、頂点数を V としたときに、一回の併合に $O(V)$ かかり、最終的な併合回数が $O(V)$ なので、合計で $O(V^2)$ の計算量がかかる。一辺が 21 の迷路なら頂点数は 100 なので、 $100^2 = 1$ 万であり一瞬で計算は終わる。頂点数が数万程度まで大きくなると数秒から数分かかるだろう。

さて、この方針でループ判定を実装し、迷路を作ってみよう。
[i][j] と [i+2][j] を結ぶ辺について f が 1 になっていたら、[i+1][j] の位置に通路を作ればよいことに注意しよう。

もしもっと高速化して大きな迷路を作りたかったら、次のような併合アルゴリズムがある。

- (i) 各島の番号について、島の面積の大きさを記録する配列を作る
- (ii) 併合時には、面積の少ない方を、大きい方の番号に合わせる。
このとき、Part2 で書いた連鎖判定のアルゴリズムを使う。
- (iii) 併合したときに、島の面積を表す配列を更新する。

このアルゴリズムの計算量の見積りは少し難しいが、

面積の大きい島と小さい島を併合したときに効率がいい
という性質を考えると、最も効率が悪くなる場合は面積の同じ島を併合し続けた時である。すると、次のように併合が行われた時に一番効率が悪い。

```
1 1 1 1 1 1 1      二つずつ併合する
2 0 1 1 1 1 1
...
2 0 2 0 2 0 2      また 2 同士を併合する
4 0 0 0 2 0 2
4 0 0 0 4 0 0      4 同士を併合する
8 0 0 0 0 0 0
```

全部 1 だったのが全部 2 になるまでに $O(V)$ の計算量がかかる。これが $O(\log V)$ 段階あるから、これは最悪の場合でも $O(V \log V)$ で終わるアルゴリズムである。

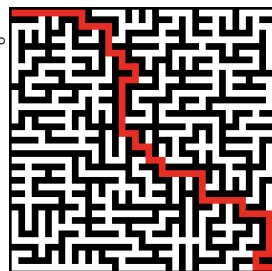
「ユニオンファインド」というアルゴリズムを使うと、更に高速化することができるので余裕があったら勉強してみよう。

4-3. 迷路を解く（最短経路問題）

（１）深さ優先探索

スタートを左上のマス、ゴールを右下のマスとしよう。
Part2 で学んだ深さ優先探索を使って、この迷路を解くことができる。

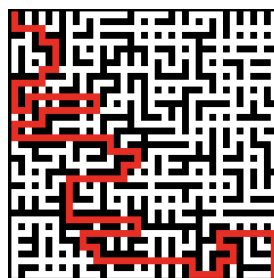
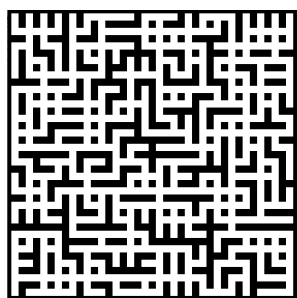
方針としては、連鎖判定のときのプログラムは連結されているすべてのマスを回ってくれるので、ゴールに到着した時点で停止させて、それまでの経路を出力すれば良い。



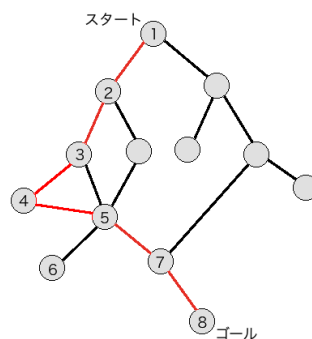
（２）幅優先探索

今までの迷路に辺をさらに追加すると、ループを含んだ迷路が出来上がる。今度はこのような迷路に対して、スタートからゴールまでの最短経路を出力する問題を考えよう。

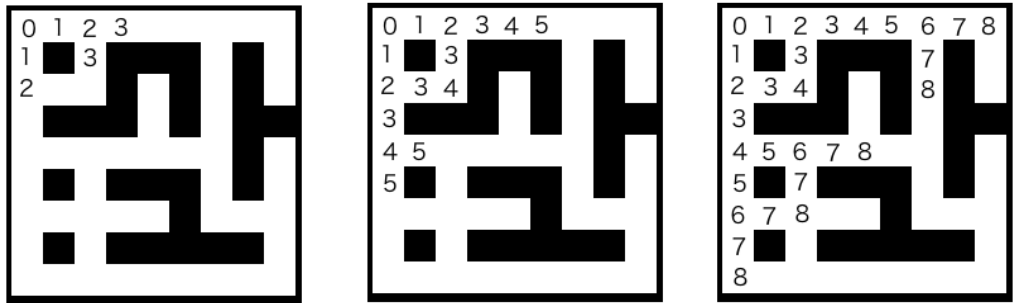
これを深さ優先探索で解こうとすると、次のように全く最短ではないルートが提示されてしまう。深さ優先探索が、行き止まりにあたるまでとりあえず進み続けるという方針だからである。



右図は深さ優先探索での探索結果の例。数字は探索の順番（再帰呼び出しされる順番）を表す。左の子から順番に再帰していくので、本当は右側を通るルートのほうが最短なのに、先に見つかった左側からのルートが出力されてしまう。

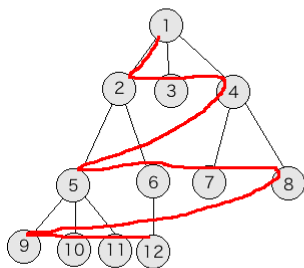


このような問題では、「幅優先探索」を使う。幅優先探索は、次の図のように、スタートから近い通路から順番に調べていく。下図の数字はスタートからの距離を表している。近い方から調べていくので、最初にゴールに辿り着いた時のルートが最短経路となるという戦略である。



5 ページ目に書いたように、幅優先探索はキューというデータ構造を使う。キューとは先に入れたものから順番に取り出せるようなデータ構造で、幅優先探索のアルゴリズムは次のようになる。

- キュー：マスの位置(x,y)とそこまでの距離 l を持つ class を格納できるとする
二次元配列 $c[i][j]$ ：上の図の数字に対応する。まだ訪れてないマスは-1 とする
- (i) スタートのマスをキューに入れる
 - (ii) キューの先頭のマスをキューから取り出す。キューが空なら(viii)へ。
 - (iii) そのマスの $c[x][y]$ が-1 でなければ（すでに見てたら）、(ii)にもどる。
 - (iv) $c[x][y]$ に l を代入する。
 - (v) $c[x][y]$ がゴールだったら(viii)へ。
 - (vi) そのマスに接続しているマスを、距離を $l+1$ にしてキューに入れる。
 - (vii) (ii)に戻る
 - (viii) 幅優先探索の終了



	0	1	2	3	4	5	6	7	8	9	10
キュー	1										

	0	1	2	3	4	5	6	7	8	9	10
キュー		2	3	4							

	0	1	2	3	4	5	6	7	8	9	10
キュー			3	4	5	6					

	0	1	2	3	4	5	6	7	8	9	10
キュー				4	5	6					

	0	1	2	3	4	5	6	7	8	9	10
キュー					5	6	7	8			

	0	1	2	3	4	5	6	7	8	9	10
キュー						6	7	8	9	10	11

・ 幅優先探索の実装

次のものを用意しよう。

(1) Cell クラス

```
class Cell{
    int fx, fy;    (一つ前のマスの位置)
    int tx, ty;    (そのマス)
    int l;         (スタートからの距離)
}
```

(2) グローバル二次元配列

```
int[][] memo;    (そのマスをすでに調べたかを記録する)
int[][] px, py;  (そのマスの一つ前のマスを記録する)
```

(3) キュー

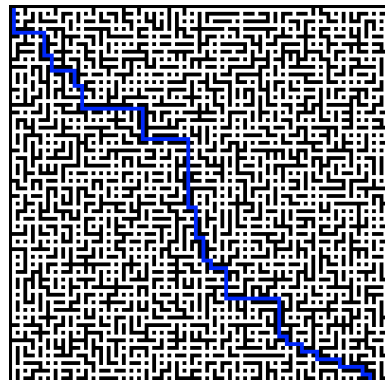
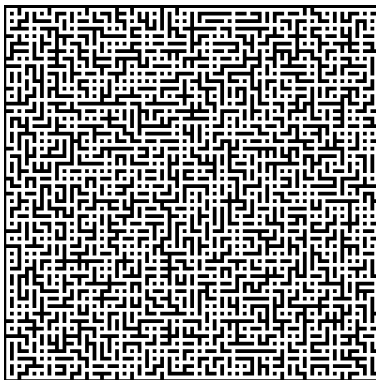
```
Cell[] q;        (Cell 型のインスタンスを格納するキュー。
                  配列の長さは十分長くしておく)
int fq, bq;      (fq はキューの先頭、bq は末尾の添字番号を記録)
void q_push(Cell c) (キューの末尾に c を追加する命令)
Cell q_pop()      (キューの先頭を取り出す命令。
                  キューが空なら null (空) を返す (return null;))
```

(4) 幅優先探索

```
void bfs()        (前のページのアルゴリズムを記述する)
```

(5) 最後に経路を表示する

```
void disp(int x, int y) (配列 px, py に直前のマスが記録されているので、
                        ゴールから順に再帰でたどっていけばいい)
```



(3) ダイクストラ法

・ 問題

迷路の問題に、「壁の中も $1/3$ の速度で移動することができる」という条件を付け足した問題を考える。このとき最短経路はどうやって求めればいいだろうか？

・ 発想

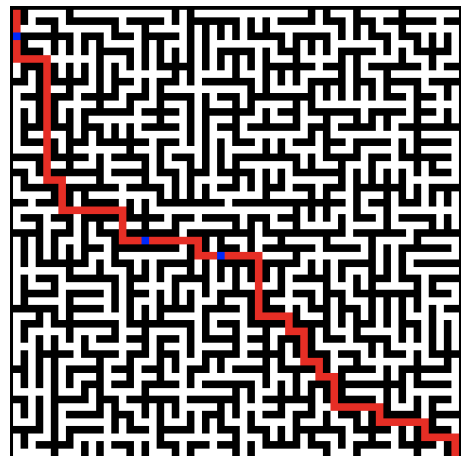
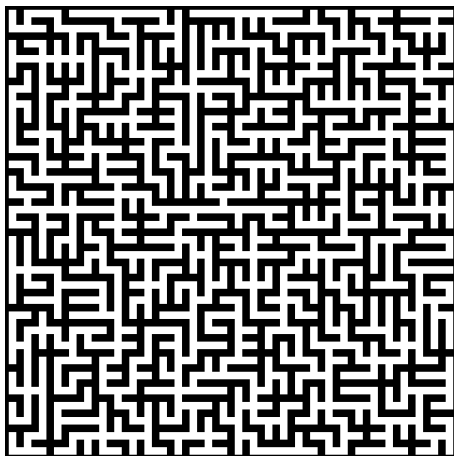
幅優先探索と同じ発想で、「近いところから見ていけばいい」ということになる。前の問題の時は通路は全てコスト 1 だったので、行けるところからキューに入れていけば、キューの先頭は必ず、キューの中で最も近い地点を指していた。しかし、今度は壁は通路 3 個分のコストがかかるので、例えばあるマスの次のマスが壁と通路のとき、壁を先にキューに入れると、本当は通路のほうが近いのに壁のほうが先にキューから出てきてしまう。

この問題を解決するためには、「今キューに入っているマスのなかで、最もスタート地点から早く行けるマスを探して、それを取り出す」ということをすればよさそうである。

・ 素直な実装

この発想でプログラムを書いてみよう。q_pop 命令を、キューの中で最も速く行けるマスを返すように書きかえる必要がある。キューの要素を全て眺めて、最小のものを探せば良い。また、キューに隣接マスを追加する処理を、隣が壁の時に距離を +3 して追加するようにしよう。

結果は次のようになるはずだ。青マスが壁を通過した地点である。



・ダイクストラ法とプライオリティキュー

このような、最も速く行けるノードから順番に調べていく方法をダイクストラ法という。また、前のページの下線を引いた部分のように、格納されている要素の中で最小の要素を返すようなキューのことを優先度付きキュー、あるいはプライオリティキューという。

ところで、前のページでのプライオリティキューの実装は、キューの要素を全て調べてその中の最小を返したので、一回の処理に $O(N)$ かかっている (N はキューに入っている要素数)。この N がどれくらいかは見積もりが難しいが、だいたい迷路の大きさくらいのオーダーだろう。キューから要素を取り出す、という処理も N 回行われるので、合計で $O(N^2)$ がかかることがわかる。ということは迷路が大きくなると時間がかかるようになる。迷路の一辺の長さを 500 くらいにすると、数秒かかるだろう。

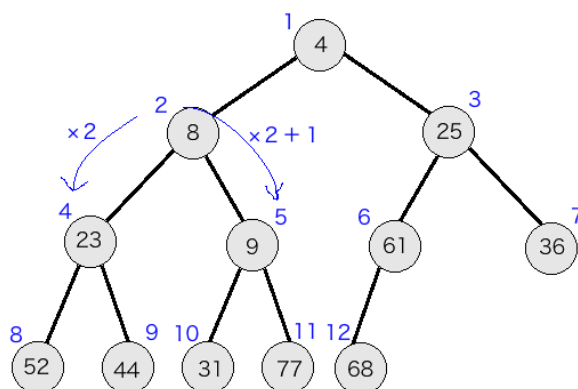
さらに、壁のコストはいままで 3 だったが、これを 100 くらいにするともっと時間がかかるようになる。これは、探索の範囲が広がるからである。

ここで、この $O(N^2)$ を $O(N \log N)$ まで落とすデータ構造とアルゴリズムが存在する。それはヒープというものだ。

・ヒープ

ヒープでは「 $a[i]$ は $a[2*i]$ と $a[2*i+1]$ より小さい」という条件をみたすように、配列にデータを記録したものである。この条件は、配列を下の木目の図のように表した時に、「親が子よりも小さい」という条件である。

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
q[i]	-	4	8	25	23	9	61	36	52	44	31	77	68			



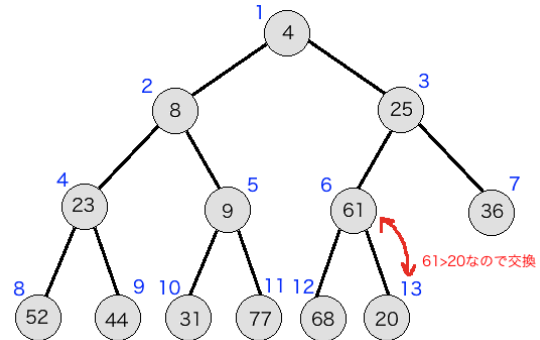
このように枝が2倍2倍に分かれていく木のことを2分木という。

親が子よりも小さいということは、 $q[i]$ の中で最小の要素を取り出したかったら、 $q[1]$ を取り出せば良い。では、 $q[1]$ を取り出した後、木をどのように変形すればいいだろうか？また、新たな要素を追加するときにはどこに追加すればいいのだろうか？木の変形をした後でも、親が子よりも小さいという条件が保たれる必要がある。

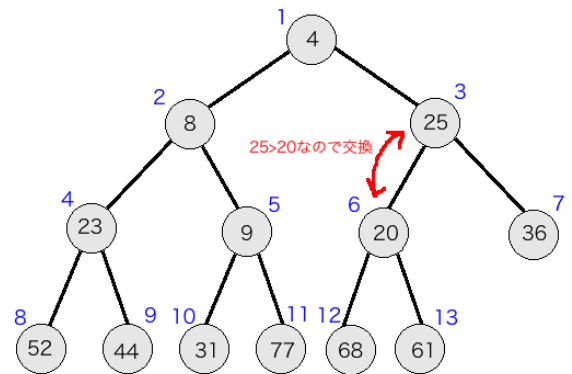
・要素の追加

まずは配列の末尾に新しい要素を追加する。ここでは20を入れてみよう。

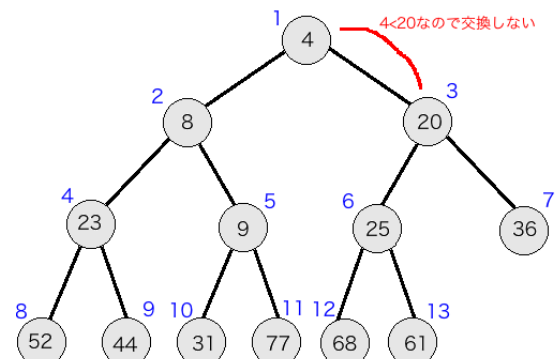
その要素と、その親を比較して、親のほうが大きかったら交換する。
親が配列の何番目にいるかは、子の番号を2で割ればわかる。
右の例だと $13/2 = 6$ が親だ。



さらに、その親と比較して、親のほうが大きかったら交換する。



親のほうが小さくなったら終わる。
これで要素20の追加が完了した。



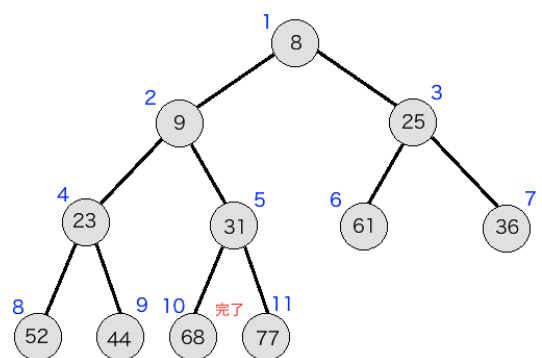
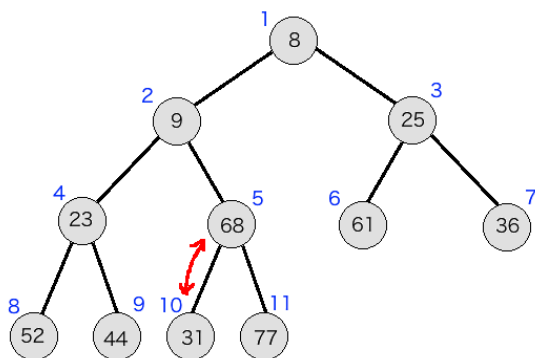
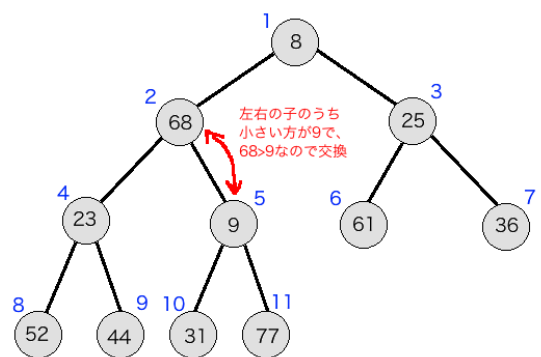
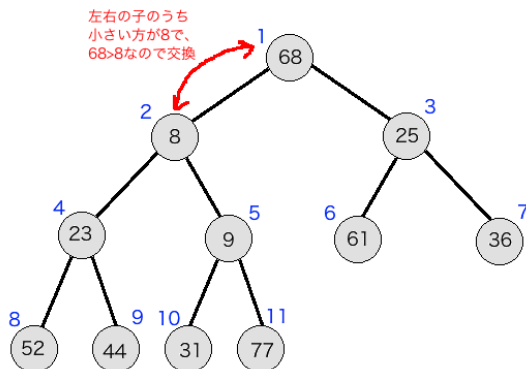
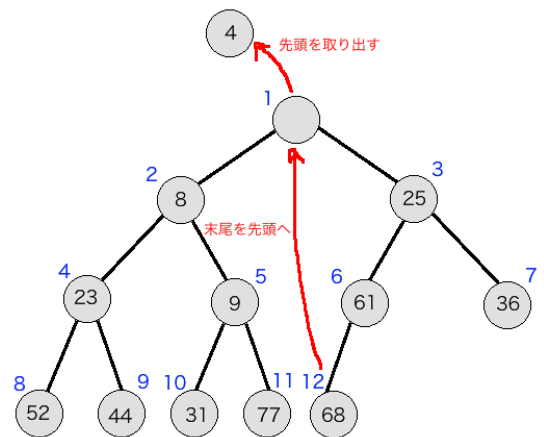
・要素の取り出し

要素を取り出すときは、先頭を取り出す。

次に末尾を先頭に移動し、次の操作を繰り返す。

「左右の子のうち小さい方について、親のほうが大きかったら、交換する」

交換する相手がなくなるか、親のほうが子より小さくなれば終了である。



この二分木の、親から一番下の子までの深さは $\log(N)$ である。従って、要素の追加と取り出しにはどちらも $O(\log(N))$ の計算量がかかる。したがって、ダイクストラ法の計算量を $O(N\log N)$ まで落とすことができるようになる。

・手元で計算してみた結果

盤面が 500×500 , 壁のコストが 3 のとき

$O(N^2)$ アルゴリズム 5 秒

ヒープ 1.5 秒

盤面が 500×500 , 壁のコストが 200 のとき

$O(N^2)$ アルゴリズム 15 分

ヒープ 1.5 秒

