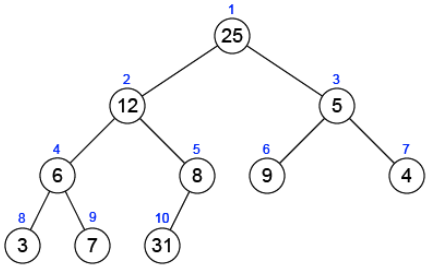
Part3.　グラフと木のアルゴリズム

1. ヒープ

2. ダイクストラ法

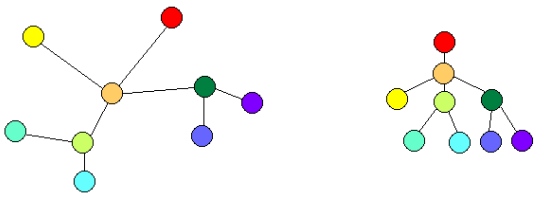
3-1.　ヒープ

・AOJ Lesson Algorithms and Data Structuresの 9-A　“Heap”を開こう。次のような図が書かれているはずだ。



1	2	3	4	5	6	7	8	9	10
25	12	5	6	8	9	4	3	7	31

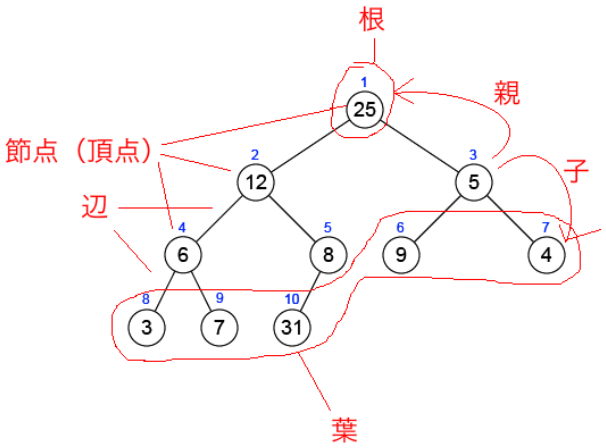
配列というと横一列に並んでいるイメージだが、「実はこの上のように二つずつに分かれている木なんです」と解釈することを考える。
ところで木といったが、プログラミングの世界で木というのは次の図のような、N 個の点が N-1 本の辺に結ばれていて、どこにもループがないような構造のことを言う。



木は、どれかの頂点を上に引っ張って持ち上げると、右のような樹形図の形にすることができる。このとき、ある点にとってすぐ上にいる点のことを親、すぐ下にいる点のことを子という。植物の木を上下ひっくり返した状態を考えて、一番上にいるものを根、一番下にいて子がいない点を葉という。

人間の上下関係とか、国-県-市の関係とか、このような構造を持つものは多い。

さて、そのような木のなかでも、二つずつに枝分かれしていくような（各頂点が子を最大二人までしか持たない）木を二分木という。そのなかでもこの問題のように配列と一対一に対応できる二分木を完全二分木という。



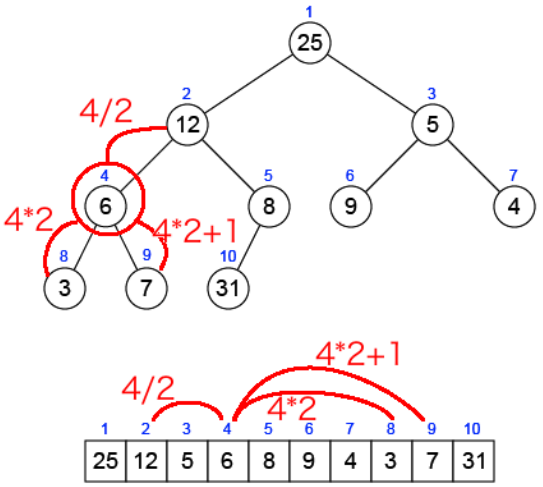
配列上での親子関係は、問題文の通り、次のようになる。

```
parent(i)
    return ⌊i/2⌋
```
2で割って切り捨てると親の場所

```
left(i)
    return 2i
```
2倍すると左の子の場所

```
right(i)
    return 2i+1
```
2倍して1足すと右の子の場所

図の上ではこんな感じ



それではこの問題を解こう。

出力部分の見本

```
for(int i=1; i<=n; i++){
    printf("node %d: ", i);
    printf("key = %d, ", A[i]);
    if(inside(parent(i))) printf("parent key = %d, ", A[parent(i)]);
    if(inside(left(i))) printf("left key = %d, ", A[left(i)]);
    if(inside(right(i))) printf("right key = %d, ", A[right(i)]);
    printf("\n");
}
```

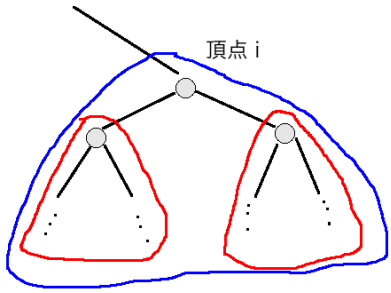
printf(“出力するもの”, 変数たち)
出力するものの中に%d があると、その部分が、後ろに書いた変数になって出力される。ちなみに文字列なら%s, 文字なら%c, 小数なら%lf とかである。

問題文に書かれている３つの命令と、inside(x)という命令（A[x]がヒープの内側にあれば 1, 外側にはみ出していたら 0 を返す関数）を実装した。inside(x)が 0 ということはその接点が存在しないということになる。

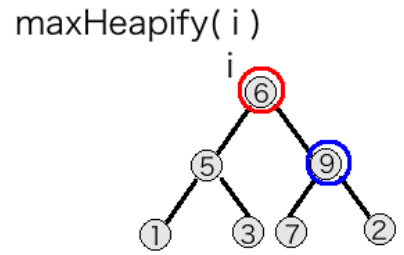
・次に、同じ Heap のところの B “Maximum Heap” を開く。A のコードを利用して解いていこう。

max-Heap とは、ヒープのどの頂点についても、親の値のほうが二人の子の値よりも大きいようなヒープのことをいう。この問題では max ヒープになっていないヒープから、max ヒープを構成するのが目標である。

次のような命令 maxHeapify(i)を作る。
　頂点 i の二人の子を根とする部分木（下の図の赤の部分）が max ヒープとなっているとき、頂点 i を根とする部分木（青の部分）を max ヒープにする



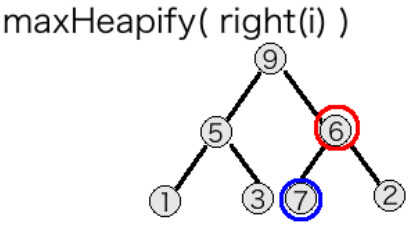
maxHeapify(i)のアルゴリズム（問題文中の擬似コードの内容）



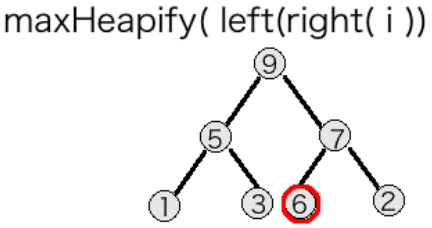
maxHeapify(i)

頂点 i の二人の子のうち
大きいほう(頂点 j とする)が
親より大きかったら親と交換し、
maxHeapify(j)を再帰呼び出し

左の図では赤と青を交換している



i が葉であるか、
二人の子よりも大きいなら終了



maxHeapify 命令は i の左右の子を根とする部分木が max ヒープとなっているとき、i を根とする部分木を max ヒープとするものであるから、これを葉のほうから順番に(配列の後ろの方から順に)実施していけば、ヒープ全体を max ヒープにすることができる。

maxHeapify は、再帰呼び出しの過程では一段ずつ下に降りていくので、O(log(n))の計算量を要する。これを葉の方から順にすべての頂点について実行するので、全体として O(n log(n))の計算量がかかることになる。

max ヒープは根がヒープ内で最大の要素になっているというところが強力である。それは次の問題 C を解くと分かる。

・B が解けたら、C の Priority Queue を開こう。

B の max ヒープに対し、さらに次の命令を実装する。

void insert(k)
　ヒープに値 k をもつ要素を追加し、max ヒープになるように整形する。

int extract()
　ヒープの先頭の要素(最大の要素でもある)を返し、これを削除する。Max ヒープとなるように整形する。

このような命令が実装されたヒープを「priority queue(優先度付きキュー)」という。このデータ構造は、次のようなケースを考えるとわかりやすい。

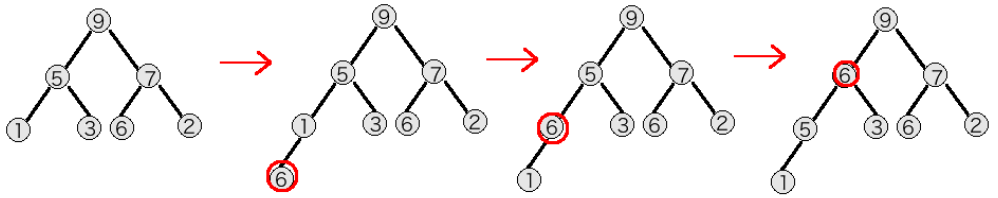
病院の外来に次々と患者がやってくる。症状は様々で、重い人も軽い人もいる。
彼らは治療まで待合室で待機する。
症状の重さ k の患者がやってくる(insert(k))。
待合室にいる中で最も症状の重い患者から順に治療される(extract())。

最初の方に来ても症状が軽ければなかなか治療してもらえず、症状が重ければ来た途端に治療されることもあるという、ありがちなシステムである。

問題文の擬似コードとは若干違うが、次のように実装すると良いだろう。

insert(k)

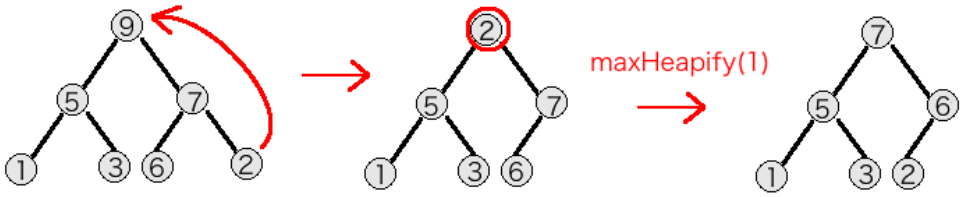
1. ヒープの最後(A[n+1]番目)に k を追加して、n++。
2. x を、その追加した頂点とする
3. x が根でない（親を持つ）限り、次の処理を繰り返す
    1. x が親より大きければ、親と交換し、x を親とする
    2. そうでなければ、ループを抜ける



これは maxHeapify とは逆に、追加した頂点が、葉の方から順に適切な位置まで根の方に浮上していくことになる。完全二分木なので、浮上回数は最大でも O(log(n))であり、この計算量も O(log(n))となる。

extract()

1. ヒープの先頭の要素を変数 ret に覚えておく
2. 末尾の要素(A[n])を先頭に上書きし、n--。（先頭を削除して、そこに代わりに末尾だった要素を持ってくるということ）
3. 根に対し maxHeapify を実行(maxHeapify(1))
4. ret を返す



先頭に対して maxHeapify が一回実行されるので、計算量は O(log(n))である。

n 個のデータをまず insert し、次に n 回連続して extract すると、元の n 個のデータが大きい順に並んで（ソートされて）extract されていくことがわかるだろうか。各 insert, extract にかかる計算量は O(log(n))だったので、この操作には O(n log(n))かかる。これはヒープソートというソートアルゴリズムである。

それではこの問題を実装しよう。

入力が 2,000,000 もあるので、入出力に cin, cout を使うと間に合わない。そこで、scanf, printf を使って次のように入出力を行うと良い。
#include <cstdio>をつける必要がある。

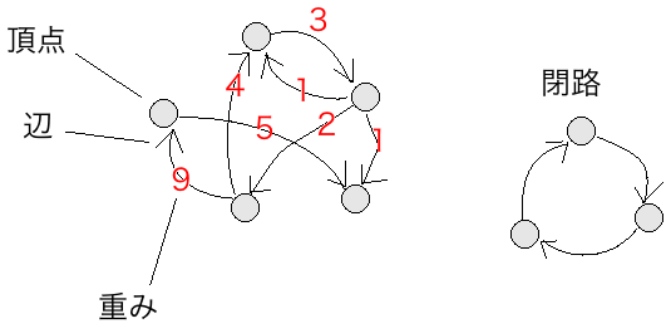
```
66  int main(){
67      n = 0;
68      for(;;){
69          char s_[100];
70          scanf("%s", s_);    // charの配列s_に各入力行の単語を読み込み
71          string s = s_;      // C++のstringに変換する(==演算子を使えるように)
72          if(s=="end"){
73              break;
74          }else if(s=="insert"){
75              int k;
76              scanf("%d", &k);
77              insert(k);
78          }else if(s=="extract"){
79              printf("%d\n", extract());
80          }
81      }
82      return 0;
83  }
```

scanf は C 言語由来の関数なので、C++で出来た string 型への入力ができない。そこで C 言語での文字列(char の配列)に一旦入力させて、それを string 型に代入することで自動的に string 型に変換される。

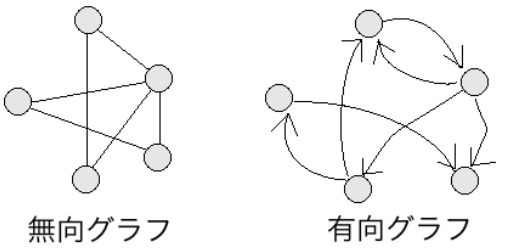
3-2.　ダイクストラ法

（１）グラフ

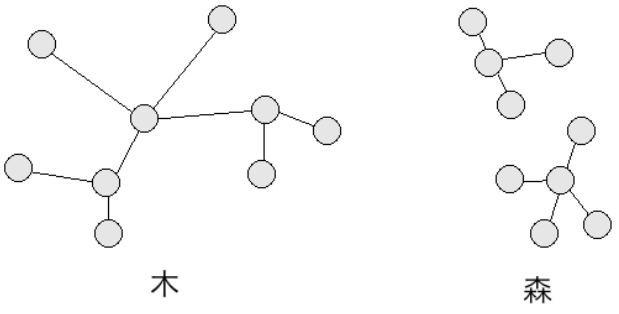
3-1 では木という構造を使った。木はグラフという構造の特別なものである。



一般に頂点が辺で結ばれている構造を「グラフ」という。次のように、辺に向きがないものを無向グラフ、向きがあるものを有向グラフという。無向グラフは、どの辺についても双方向に同じ重みの辺が張られている有向グラフと見ることも出来る。



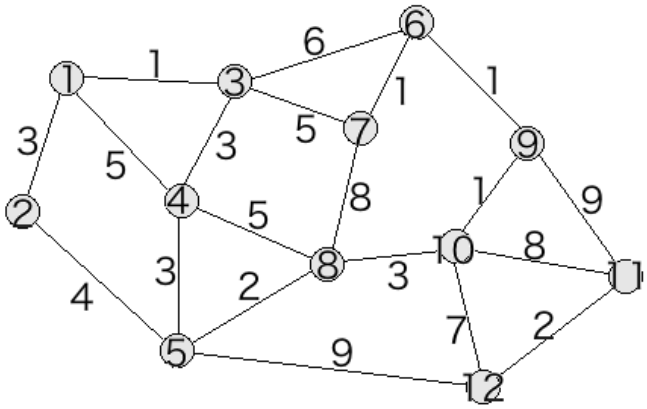
グラフの中でも、閉路（ループ）を含まず、全てがつながっている無向グラフを木という。閉路がなくて、全てが繋がっていない無向グラフは森という。



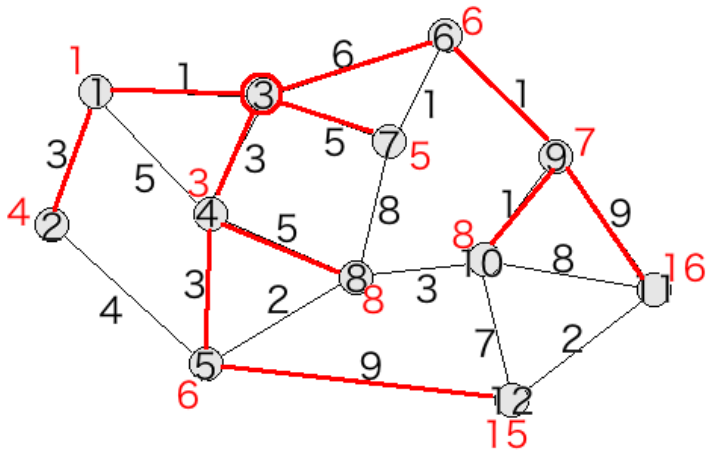
（２）ダイクストラ法

Part2-6 でワーシャルフロイド法を扱ったが、これはグラフ上の頂点から頂点への最短経路を求めるものであった。ワーシャルフロイド法を実行すると、出来上がった二次元配列には全ての頂点対について最短経路が格納されており、頂点数を N とするとその計算量は O(N^3)である。

ところで、問題によっては全部の頂点対について最短経路が求まってなくても、ある始点 A から、他の頂点たちへの最短路が分かればよいことも多い。



③ を始点とした時の結果



各辺の重みのことを通行料だと考えて、料金を円であらわすことにする。
次の頂点の集合を考える

V0:未調査の頂点
V1:調査中の頂点(暫定金額 C)
V2:調査済みの頂点(確定金額 C)

・アルゴリズム

1. すべての頂点を未調査にする
2. 始点を暫定金額 0 で調査中の頂点に追加
3. 調査中の頂点数が 0 になるまで繰り返す
    1. 調査中の頂点の中で、一番暫定金額の安い頂点を X, その暫定金額を C とする。
    2. X を V2 に移動し金額は C で確定、X から伸びている各辺について
        1. 伸びている先を Y, 辺の重みを W とする
        2. Y が V0 にあるなら、暫定金額 C+W で Y を V1 に移動
        3. else if Y が V1 にあるなら、C+W が Y の暫定金額より安ければ、C+W に更新する
        4. else Y は調査済みなので、何もしない

アルゴリズムが終了すると、始点から到達できなかった点は V0 に、到達できた点は V2 にあり、最安値が記録されている。

各頂点は最初 V0 にあり、V1 に移動し、V2 に行く。V1 から V0 に戻ったりはしないので、アルゴリズムの 3 の繰り返し回数は、O(N)である。アルゴリズムの 3-1 については、V1 に含まれる頂点すべての最小値を 1 個ずつ調べたら、O(N)かかる。ここまでで O(N^2)である。3-2 の繰り返しはある頂点から出ている辺全てについて繰り返すが、3 の繰り返しは各頂点について繰り返すので、つまり 3-2 は全体で合計すると、すべての辺につき一回ずつまわる。辺の数を M としたら、O(M)である。

つまり、M は N^2 より小さいことを鑑みるとダイクストラ法の計算量は O(N^2)である…といいたいところだが、実はもっと速くできて、調査中の頂点をヒープを使って記録すると、検索を一瞬 (O(1))で、追加削除を O(log N)で行えるようになるため、3-1 は一瞬でできて、3-2 はすべての辺について、追加や削除が行われるので O(M log(N))で終わるようになる。つまりダイクストラ法の計算量は O(M log(N))である（削除が合計で N 回行われることも考慮して O((M+N)log(N))と書く場合もある）。

10 ページの図のように金額だけでなく経路も一緒に求めたいときは、V1 の暫定金額と同時に、どの頂点から来たか、を記録しておく。すると V1 から V2 に移す時、やってきた頂点とその頂点を結ぶ辺が、使用した経路となる。

・フロイドワーシャルの問題 0117 をダイクストラ法でも解いてみよ

・問題 0150 を解いてみよ。壁を通過するコストを 1、その他を 0 とすると…

・フロイドワーシャルでは間に合わない問題 1150 を解け（ちょっとむずい）

・問題 0530（むずい）

・最小全域木を求める問題は、ダイクストラ法と類似のアルゴリズムで解くことが出来る（プリム法）
問題 0180 を解くアルゴリズムを考え、実装せよ