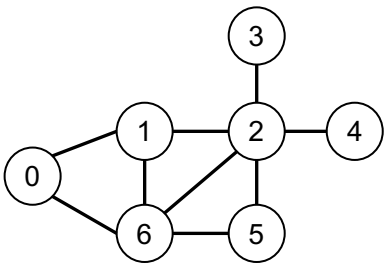
1. **グラフ構造**とは，右図のようにノード（または頂点）がリンク（または辺，エッジ）によって連結された構造である。木もグラフの一種である。数学ではグラフの表現として，**隣接行列**が用いられる。グラフ$G$を表す隣接行列$A$は，成分$a_{ij}$の値を$G$のノード$v_i$と$v_j$を直接つなぐリンクの本数とする。これはプログラム言語では 2 次元配列によって表現できる。




　グラフ構造の表現としては，他にも連結リストや木構造のようにクラス（構造体）と参照（ポインタ）を用いる方法（**隣接リスト**）などがある。

　下記のプログラムは，右図のグラフを隣接行列で表し，指定のノードから指定のホップ数以内で到達できるノードを，**深さ優先探索**（リンクがあればどんどんたどっていく）によって列挙する。空欄を適切に埋めてプログラムを入力して実行せよ。

```
/* Graph.java */
public class Graph {
   public final int[][] adj;  // 隣接行列
   public final int NUM;      // ノード数

   public Graph(int[][] adj) {
      this.adj = adj;
      NUM = adj.length;
   }

   // 始点ノードからの探索開始
   public void traverse(int start, int max) {
      boolean[] visited = new boolean[NUM];
      traverse(start, max, 0, visited);
   }

   // 再帰による深さ優先探索
   void traverse(int n, int max, int d,
                 boolean[] visited) {
      // 訪問済みにする
      visited[n] = true;

      // 距離が上限に達したらたどるのをやめる
      if (d >= max) return;

      for (int i = 0; i < NUM; i++) {
         // リンクがつながっていない場合は次へ
         if (adj[n][i] == 0) continue;

         for (int j = 0; j < d; j++)
            System.out.print("   ");
         System.out.println(n + "->" + i +
                    (visited[i] ? "*" : ""));
         // 再帰的に訪問
         traverse(i, max, d + 1, visited);
      }
   }
}
```

```
/* Program.java */
import java.util.*;

public class Program {

   // 隣接行列：ノード i と j が連結していれば
   // adj[i][j]==1，そうでなければ 0 を設定
   private static final int[][] adj = {

      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   },
      {   ,   ,   ,   ,   ,   ,   }};

   public static void main(String[] args) {

      Graph graph = new Graph(adj);

      Scanner sc = new Scanner(System.in);
      for (;;) {
         System.out.printf("始点 (0-%d): ",
                           graph.NUM - 1);
         int start = sc.nextInt();
         if (start < 0) break;

         System.out.print("最大距離: ");
         int max = sc.nextInt();

         // 再帰的な探索を開始する
         graph.traverse(start, max);
      }
   }
}
```

2. 本科目で説明してきた基礎的なデータ構造は，Java では**コレクションフレームワーク**よって提供されている。裏面の表に代表的なクラスを示しておくが，詳しくは JDK のリファレンスなどを参照してほしい。

　なかでも Set と Map はハッシュテーブルと 2 分探索木でほぼ同じ機能が提供されている。このような場合，`Map<String, String> map = new HashMap<String, String>();` のように親インタフェースの変数に代入して使うようにしておけば，後から実装クラスを変更してもコードの修正が最小限ですむ。

　裏面のプログラムは，標準入力から単語を 1 つずつ読み込んで出現回数を数えるものだが，最初に HashMap を使うか TreeMap を使うか選べるようになっている。空欄を適切に埋めて実行すると，入力が全く同じでも HashMap の場合と TreeMap の場合では少し表示が異なることが分かる。それはどうしてか考察せよ。

| クラス | 内部構造 | 親インタフェース | インタフェースの説明と代表的なメソッド |
|---|---|---|---|
| ArrayList<E> | 配列 | List<E> | データを一列に並べて格納する（第 6 回，第 10 回で説明） |
| LinkedList<E> | 連結リストの一種 | | |
| HashSet<E> | ハッシュテーブル | Set<E> | 要素が重複しないようにデータを格納する（集合）<br>add(data), contains(data), remove(data), size(), isEmpty() |
| TreeSet<E> | 2 分探索木の一種 | | |
| HashMap<K, V> | ハッシュテーブル | Map<K, V> | キーと値のペアでデータを格納する（写像，辞書，連想配列）<br>put(key, value), get(key), remove(key), isEmpty(), size(), containsKey(key), entrySet(), keySet(), values() |
| TreeMap<K, V> | 2 分探索木の一種 | | |
| Map.Entry<K, V> | | | Map<K, V>の各要素を表す内部クラス　getKey(), getValue() |

```
import java.util.*;

public class Program {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      Map<String, Integer> map = null;
      do {
         System.out.print(
            "1. HashMap 2. TreeMap ? ");
         int ans = sc.nextInt();
         switch (ans) {
         case 1:  // HashMap でインスタンス生成
            map = new
            break;
         case 2:  // TreeMap でインスタンス生成
            map = new
            break;
         default:
            break;
         }
      } while (map == null);

      System.out.println("Type any words.");
      for (;;) {
         String word = sc.next();
         if (word.equals("QUIT")) break;

         // 単語が登録済みか調べる
         if (map.containsKey(word)) {
            // データの取得は get，登録・変更は put

            map.     (word, map.     (word)+1);
         } else {
            map.     (word, 1);
         }
      }

      // 全要素の処理
      for (Map.Entry<         ,         > entry:
           map.entrySet()) {
         System.out.printf("%-10s : %3d%n",
            entry.getKey(), entry.getValue());
      }
   }
}
```

3. 第 6 回の 4.および 6.では，整列や探索のおける比較方法を変更するために，以下のように Arrays.sort 等の最後の引数として比較用クラスのインスタンスを渡す方法を学んだ。

```
Arrays.sort(items, new ItemComparator());  // 親インタフェースは Comparator<Item>
```

このような使い捨てのクラスは，**匿名クラス**という手法で class 宣言を行わずにインスタンスが生成できる。

```
Arrays.sort(items, new Comparetor<Item> {
    public int compare(Item item1, Item item2) { return item2.code - item1.code; }
});
```

さらに，Java 8 以降では抽象メソッドが 1 つしかないインタフェース（関数型インタフェース）の場合には，以下のように親インタフェース名もメソッド名も省略可能になった。この記法を**ラムダ式**と呼ぶ。

```
Arrays.sort(items, (item1, item2) -> { return item2.code - item1.code; });
```

ラムダ式を用いると「処理のかたまり」をメソッドの引数として渡すことが容易になる。コレクションクラスでは for 文の代わりに forEach メソッドを使うことができるようになった。下記の 2 行は同じ処理を行う。

```
for (String s : strlist) System.out.println(s);
strlist.forEach((s) -> { System.out.println(s); });
```

以上の説明を踏まえた上で，第 6 回の 6.を参考にして，第 6 回で用いた Item クラスの ArrayList を定義し，キーボードから商品データ（品番と品名）をいくつか読み込んでから，ラムダ式を用いてそれらを品名で並び替え，最後にすべての商品を forEach メソッドで表示するプログラムを作成せよ。

4. 【発展】下記のプログラムは，あるノードから他のノードへの最短距離とその経路を**ダイクストラ法**というアルゴリズムによって求めるものである。コメント文を手がかりに適切に空欄を埋めてからプログラムを実行し，説明を読んでアルゴリズムについて理解せよ。

```
/* Graph.java */
import java.util.*;

public class Graph {
   public final int[][] adj;  // 隣接行列
   public final int N;       // ノード数

   public Graph(int[][] adj) {
     this.adj = adj;
     N = adj.length;
   }

   // 距離無限大を表す十分に大きな数
   public final static int INF = 9999;

   // ダイクストラ法の結果
   public class DResult {
     public int[] dist;  // ノードまでの最短距離
     public int[] back;  // 逆順の経由ノード

     DResult(int[] dist, int[] back) {
       this.dist = dist;
       this.back = back;
     }
   }

   public DResult dijkstra(int start) {

     // 始点ノードから各ノードまでの最短距離
     int[] dist = new int[N];
     // 各ノードへの経路の 1 つ前の経由ノード
     int[] back = new int[N];

     // 始点からの距離が未確定のノードの集合
     //（コレクションクラスの集合を利用した）
     Set<Integer> Q = new HashSet<Integer>();

     // 始点から全ノードへの距離を無限大に初期化
     for (int i = 0; i < N; i++) {
       dist[i] = INF;
       Q.add(i);  // 最初は全ノードが未確定
     }

     dist[start] = 0;   // 始点→始点は距離 0
     back[start] = -1;  // 始点の 1 つ前はない

     // 距離が未確定なノードがある間，繰り返す
     while (Q.size() > 0) {

       // 距離が未確定なノードたち Q の中で
       // 始点に一番近い nearest を選び出す
       //（初期値は適当に 1 つピックアップ）
       int nearest =  Q.iterator().next();
       for (int node : Q) {
         if (dist[node] <              ) {

           nearest =
         }
       }

       // Q が到達できないノードだけなら終了
       if (dist[nearest] == INF) break;

       // 距離が最短だったノード（nearest）へは，
       // もうこれ以上短い経路は存在し得ない
       // よって，Q から取り除き，距離を確定する
       Q.remove(nearest);

       // 距離未確定の全ノードについて，nearest を
       // 経由した場合の距離を計算し，今までよりも
       // 短くなるなら，距離と経由ノードを更新する
       for (int node : Q) {

         // nearest→node の接続があるか？
         if (                    > 0) {

           // 現在既知の node への最短距離よりも
           // nearest への距離+1 のほうが短いか？
           if (dist[node] > dist[nearest] + 1) {
                          = dist[nearest] + 1;

             back[node] =
           }
         }
       }
     }
     // 配列を 2 つ返したいのでクラスにまとめた
     return new DResult(dist, back);
   }
}


/* Program.java */
import java.util.*;

public class Program {
  static final int[][] adj = { /* 省略 */ };

  public static void main(String[] args) {

     Graph graph = new Graph(adj);

     Scanner sc = new Scanner(System.in);
     System.out.printf("始点 (0-%d): ",
                       graph.N - 1);
     int start = sc.nextInt();

     Graph.DResult d = graph.dijkstra(start);

     // 最後に始点から全ノードへの距離を表示
     for (int i = 0; i < graph.N; i++) {
        System.out.printf("%d->%d 距離:%d 経路:",
                          start, i, d.dist[i]);
        // 経路を逆にたどって表示
        for (int x = i; x != -1; x = d.back[x])
           System.out.print(x + " ");
        System.out.println();
     }
  }
}
```