Prim Lazy

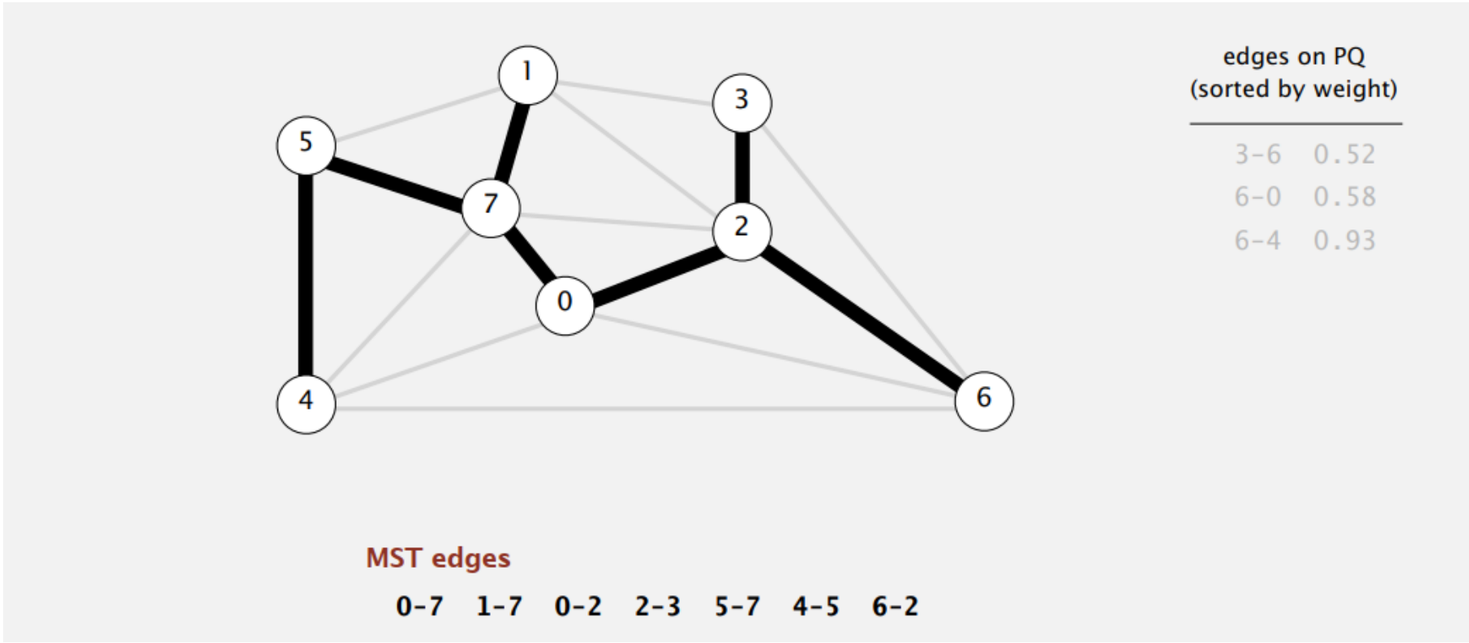
```
public LazyPrimMST(WeightedGraph G)
{
     MinPQ<Edge> pq = new MinPQ<Edge>();
     Queue<Edge> mst = new Queue<Edge>();
     boolean[] marked = new boolean[G.V()];
     visit(G, 0);
     while (!pq.isEmpty() && mst.size() < G.V() - 1)
     {
          Edge e = pq.delMin();
          int v = e.either(), w = e.other(v);
          if (marked[v] && marked[w]) continue;
          mst.enqueue(e);
          if (!marked[v]) visit(G, v);
          if (!marked[w]) visit(G, w);
     }
}

     private void visit(WeightedGraph G, int v)
     {
          marked[v] = true;
          for (Edge e : G.adj(v))
          if (!marked[e.other(v)])
          pq.insert(e);
     }
}
```

Prim Eager

```
public PrimMST(EdgeWeightedGraph G)
{
    Edge[] edgeTo = new Edge[G.V()];
    double[] distTo = new double[G.V()];
    boolean[] marked = new boolean[G.V()];
    IndexMinPQ<Double> pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;

    for (int v = 0; v < G.V(); v++)
        if (!marked[v]) prim(G, v);
}

private void prim(EdgeWeightedGraph G, int s) {
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        scan(G, v);
    }
}

private void scan(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (marked[w]) continue;
        if (e.weight() < distTo[w]) {
            distTo[w] = e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
```

	} } }

- Thuật toán Prim Lazy sẽ duyệt từ đỉnh 0, bỏ tất cả các cạnh có một đầu là đỉnh 0 vào trong 1 PQ chứa các Edge (mục đích để sắp xếp chúng) và lấy ra phần tử nhỏ nhất trong PQ. Và ta chỉ lấy đến khi có đủ V-1 cạnh để hình thành 1 cây khung. Do hệ quả của thuật toán tham lam nên sau khi ta thực hiện Prim, cây khung sẽ là cây khung nhỏ nhất.
- Nhưng thuật toán Prim có một điều là do ta cứ bỏ tất cả các cạnh kề với đỉnh đang xét vào PQ nên sau khi thuật toán dừng, PQ của ta vẫn còn cạnh, tức là ta lấy thừa cạnh bỏ vào PQ, gây lãng phí thời gian chạy.



- Một cách hiệu quả hơn là ta sẽ duyệt các đỉnh thay vì duyệt các cạnh (từ đỉnh đang xét (v), đỉnh nào mà đường đi tới nó là nhỏ nhất thì mình chọn đỉnh đó). Và ta sẽ tạo ra một CTDL IndexPQ thay cho PQ, IndexPQ của ta sẽ chứa các đỉnh với thứ tự ưu tiên các đỉnh trong IndexPQ là trọng số cạnh tới các đỉnh đó. Và do đó cách này IndexPQ của ta chỉ cần chứa đúng V đỉnh cần xét, không bị dư thừa phần tử khi thuật toán kết thúc.
- Trong đoạn code ở Prim Eager, IndexPQ có 2 phương thức là `pq.decreaseKey(w, distTo[w]);` và `pq.insert(w, distTo[w]);`. Khác với PQ, khi ta insert 1 phần tử vào PQ thì PQ sẽ lấy chính phần tử đó để so sánh và để đưa ra thứ tự các phần tử trong PQ, thì IndexPQ sẽ chứa các phần tử và các key tương ứng với các phần tử đó, và khi ta insert vào pq, ta sẽ phải insert cả key của phần tử đó (trong đoạn code kia thì w chính là phần tử còn distTo[w] là key).

- Và do tính chất như vậy nên IndexPQ có một cái hay là ta có thể thay đổi thứ tự ưu tiên của phần tử trong IndexPQ bằng cách thay đổi key tương ứng của nó. Và môt jtrong hai hàm của IndexPQ để thay đổi key tương ứng của nó là decreaseKey và increaseKey (nhưng trong các bài ta học gồm Dijkstra và Prim này thì ta chỉ dùng đến decreaseKey). Phương thức decreaseKey là ta sẽ có các tham số đầu vào là phần từ mà muốn đổi Key và trọng số tương ứng. Trọng số này phải nhỏ hơn trọng số ban đầu của Key (vì ta muốn decreaseKey).
- Code của IndexMinPQ có thể tham khảo thêm trong algs4.jar (tại nó dài quá :))) nhưng chỉ cần tham khảo các method là constructor, insert, decreaseKey, elMin, các general helper functions và heap helper functions cho bài này)
- **Độ phức tạp:**
    - **Prim Lazy:** Ta sẽ có 2 thao tác chính trong PrimLazy là delete min và insert. Trong trường hợp xấu nhất, ta phải insert tất cả các cạnh trong đồ thị vào PQ nên mỗi một thao tác sẽ có độ phức tạp là logE (trong đó E là số cạnh trong đồ thị) và thực hiện mỗi động tác E lần => Độ phức tạp là O(ElogE)
    - **Prim Eager:** Tương tự như PrimLazy thì ta cũng có 2 thao tác chính là delete min và insert. Trong mọi trường hợp, ta sẽ phải insert vào IndexPQ là V đỉnh để tạo ra MST nên độ phức tạp mỗi thao tác sẽ là logV, và trong trường hợp xấu nhất, ta phải duyệt E cạnh (`for (Edge e : G.adj(v))` nên độ phức tạp thuật toán sẽ là O(ElogV) (tốt hơn khi trong đồ thị có càng nhiều cạnh)