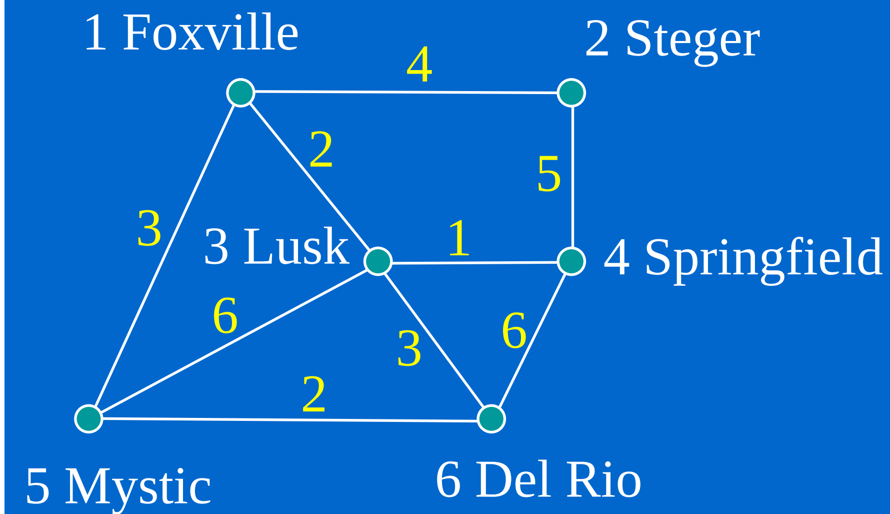
Minimal Spanning Tree

✧ JohnsonBaugh's *Algorithms*, Section 7.3 (page 284) find Minimal Spanning Tree (MST) with **Prim's algorithm**:

Six cities

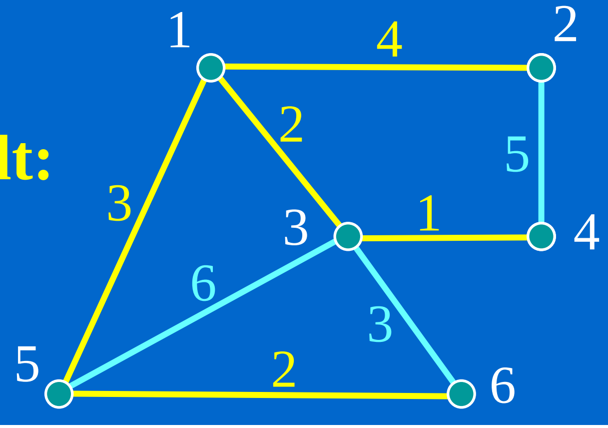


We want to construct a set of interconnecting roads such that one can reach any city from any starting city and the **total construction costs are minimized**.

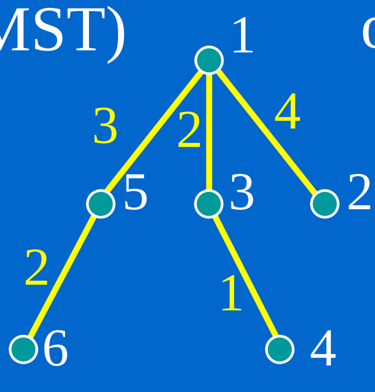
The estimated costs for some pairs of cities are as labeled.

Result:

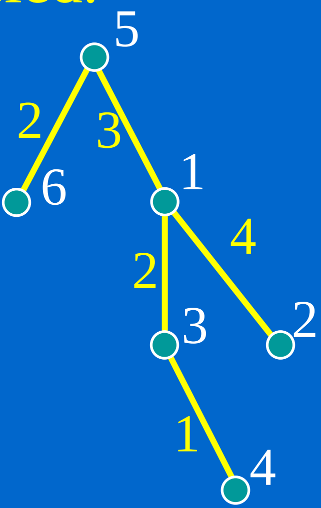
Best



A tree (MST)

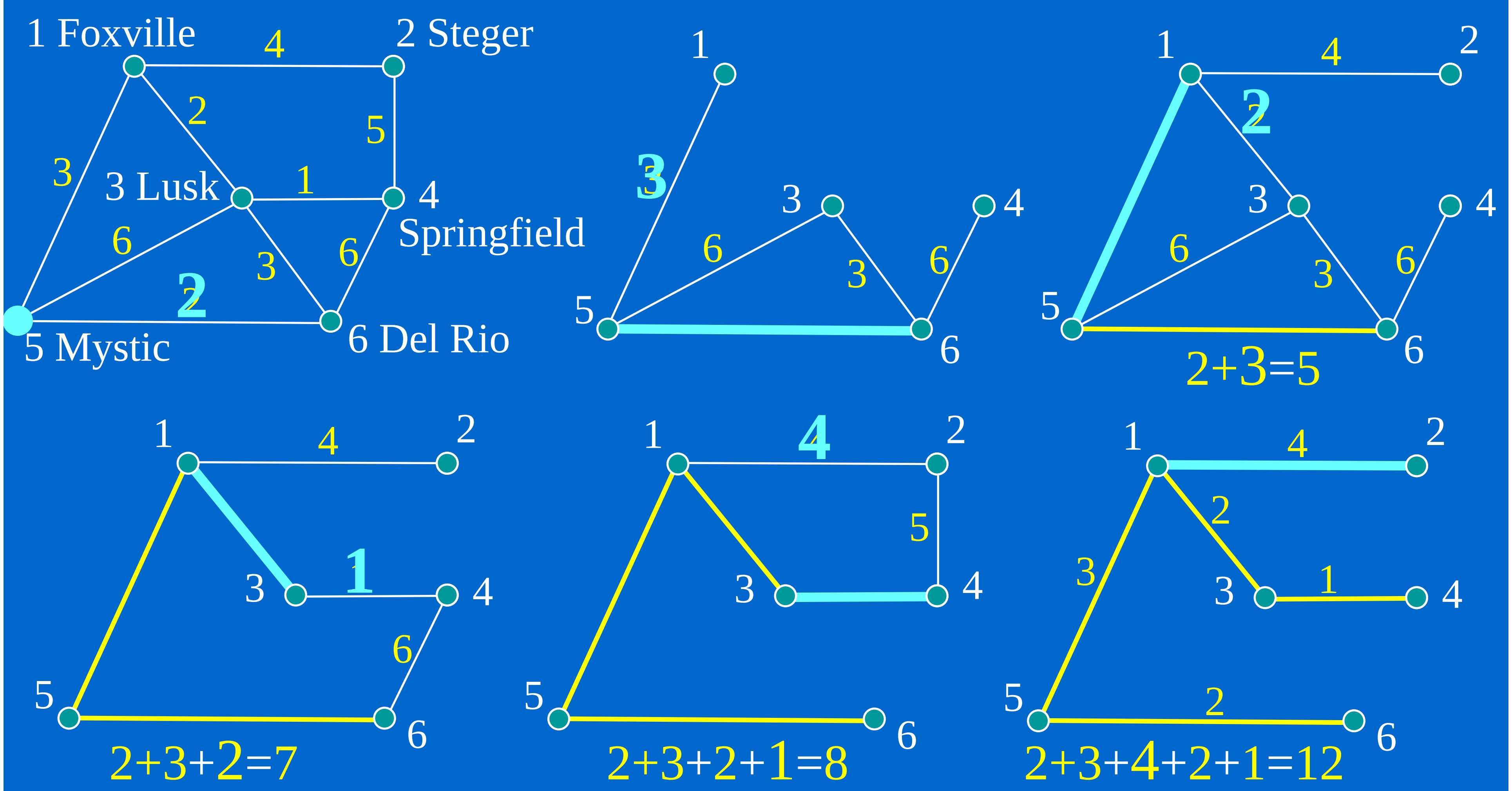


or



Prim's MST (1/7)

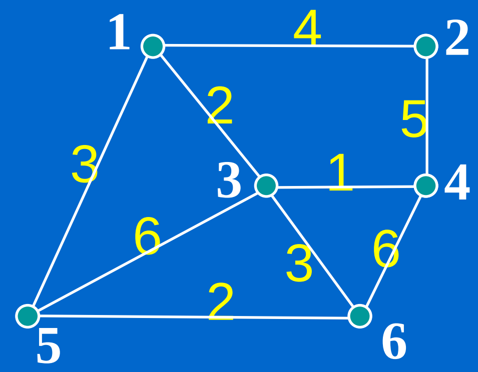
✧ **Prim's algorithm**: starting with vertex **5** (Mystic)



Prim's MST (2/7)

Adjacency matrix:

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 4 & 2 & 0 & 3 & 0 \\
2 & 4 & 0 & 0 & 5 & 0 & 0 \\
3 & 2 & 0 & 0 & 1 & 6 & 3 \\
4 & 0 & 5 & 1 & 0 & 0 & 6 \\
5 & 3 & 0 & 6 & 0 & 0 & 2 \\
6 & 0 & 0 & 3 & 6 & 2 & 0
\end{array}
$$

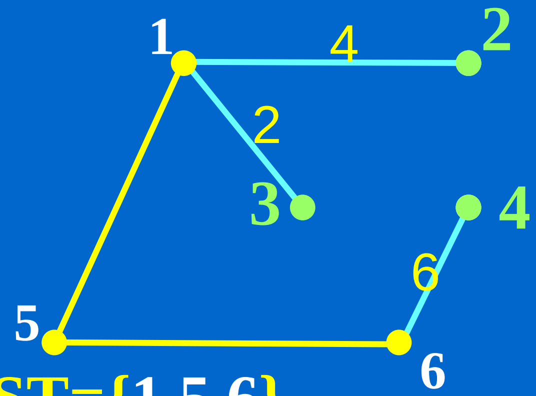


h: a list of vertices ***v*** not in the MST and its minimum weight to MST (weight of the edge from ***v*** to the vertex ***parent*[*v*]**)

***parent*[*v*]**: (*v*, *parent*[*v*]) is an edge of the minimal spanning tree

h

v	minimum weight from *v* to **MST**	***parent*[*v*]**
2	**4**	**1**
3	**2**	**1**
4	**6**	**6**

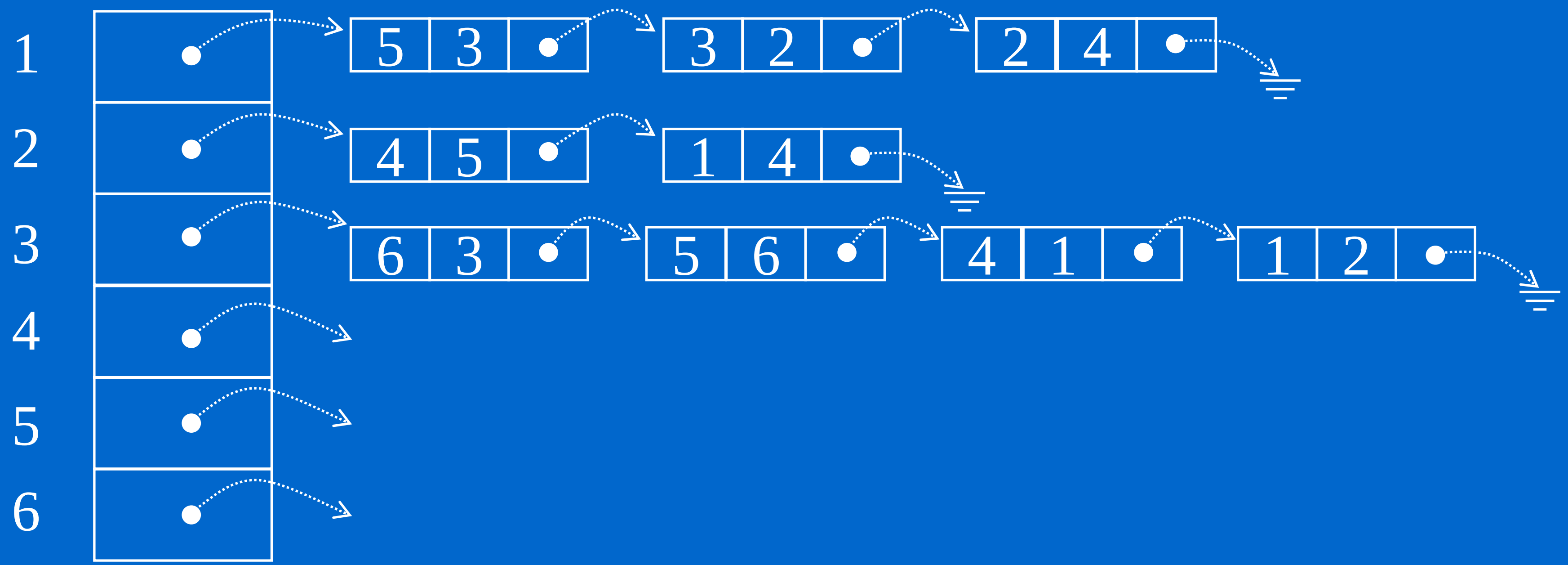


MST={1,5,6}

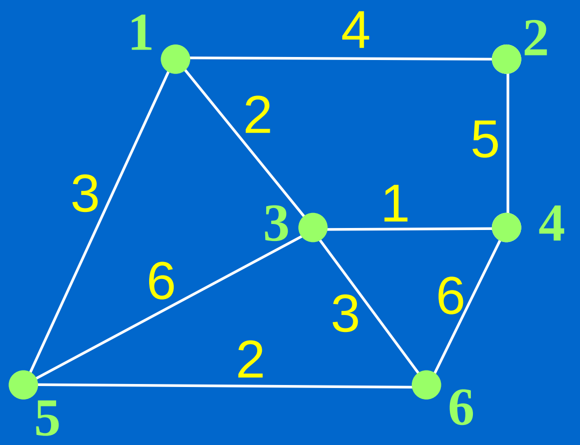
Prim's MST (3/7)

	1	2	3	4	5	6
1	0	4	2	0	3	0
2	4	0	0	5	0	0
3	2	0	0	1	6	3
4	0	5	1	0	0	6
5	3	0	6	0	0	2
6	0	0	3	6	2	0

- ✧ Adjacency list **adj**:



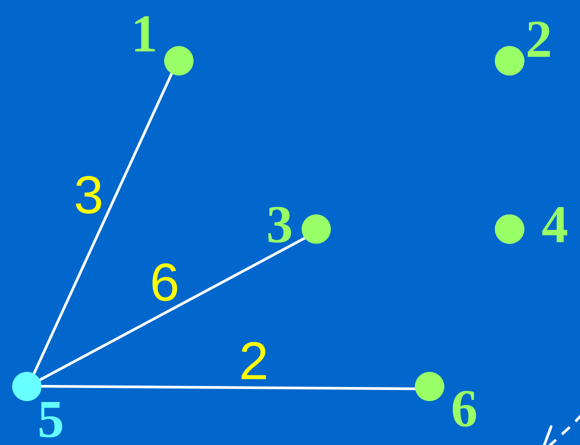
Prim's MST (4/7)



MST={ }

h

v	minimum weight from *v* to MST	*parent*[*v*]
1	∞	–
2	∞	–
3	∞	–
4	∞	–
5	**0**	0
6	∞	–



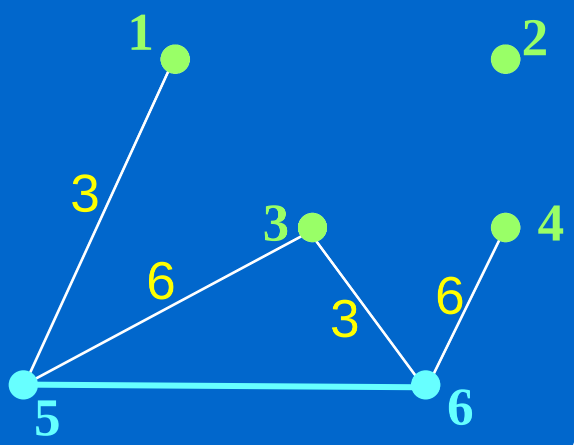
MST={5}

h

❶ ❷

v	minimum weight from *v* to MST	*parent*[*v*]
1	~~∞~~ **3**	~~–~~ 5
2	∞	–
3	~~∞~~ **6**	~~–~~ 5
4	∞	–
6	~~∞~~ **2**	~~–~~ 5

Prim's MST (5/7)



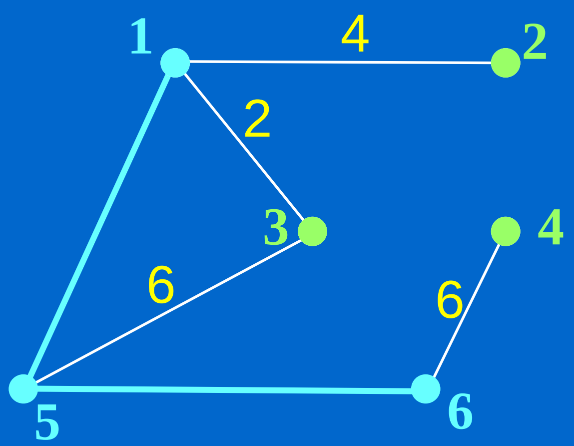
MST={5,6}

h

v	minimum weight from *v* to MST	*parent*[*v*]
1	3	5
2	∞	–
3	~~6~~ 3	~~5~~ 6
4	~~∞~~ 6	~~–~~ 6

parent

1	2	3	4	5	6
5				0	5



MST={5,6,1}

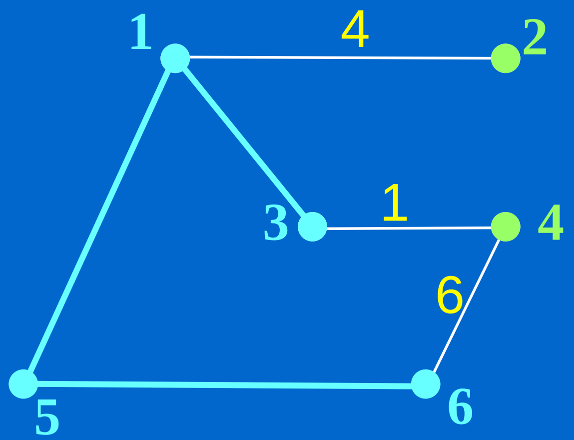
h

v	minimum weight from *v* to MST	*parent*[*v*]
2	~~∞~~ 4	~~–~~ 1
3	~~3~~ 2	~~6~~ 1
4	6	6

parent

1	2	3	4	5	6
5		1		0	5

Prim's MST (6/7)



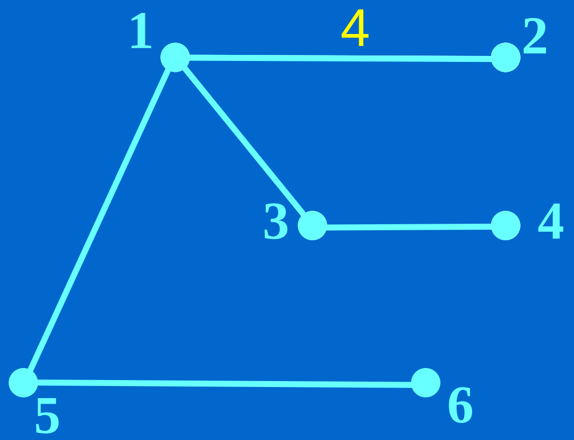
MST={5,6,1,3}

h

v	minimum weight from *v* to MST	*parent*[*v*]
2	4	1
4	~~6~~ 1	~~6~~ 3

parent

1	2	3	4	5	6
5		1	3	0	5



MST={5,6,1,3,4}

h

v	minimum weight from *v* to MST	*parent*[*v*]
2	4	1

parent

1	2	3	4	5	6
5	1	1	3	0	5

MST={5,6,1,3,4,2}

Prim's MST (7/7)

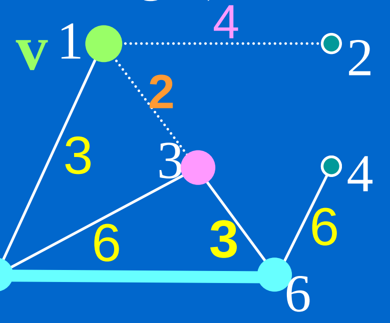
```
prim(adj, start, parent) {
    n = adj.last
    for i = 1 to n
        key[i] = ∞
    key[start] = 0
    parent[start] = 0
    h.init(key, n)
    for i = 1 to n {
        v = h.del()
        ref = adj[v]
        while (ref != null) {
            w = ref.ver
            if (h.isin(w) &&
                ref.weight < h.keyval(w)) {
                parent[w] = v
                h.decrease(w, ref.weight)
            }
            ref = ref.next
        }
    }
}
```

❶ ❷

v=1

ref → (5,3) → (3,2) → (2,4) → null

w=3, w∉ MST
ref.weight=2
h.keyval(w)=3



h is an **abstract data type** that supports the following operations

h.**init**(key, n): initializes h to the values in key

h.**del**(): deletes the item in h with the smallest weight and returns the vertex

h.**isin**(w): returns true if vertex w is in h

h.**keyval**(w): returns the weight corresponding to vertex w

h.**decrease**(w, new_weight): changes the weight of w to new_weight (smaller)

Implementation Hints

1. Write a function to read the file to an adjacency matrix
2. Write a function to convert the matrix to an adjacency list
    a. Define the list node structure **(vertex, weight, next)**
    b. Define a pointer array **adj[]** for list heads
    c. Write an **insert()** function to insert a node to a specified list
    d. Write a **freeList()** function free all lists
3. Define the structure of container h to store all nodes currently not in MST
    a. An array **vertices[]** to store nodes
    b. An array **keys[]** to store the minimal distance of vertices[] to the MST
4. Define the array **parent[]** to store the MST
5. Write a C function for the Prim algorithm of previous page
6. Write an **init()** function to initialize the container h from key[]
7. Write a **del()** function to find the node with minimal keyvalue in h and delete that node/key
8. Write an **isin()** function to test if a node is currently in MST
9. Write a **keyvalue()** function to return the key value of specified node in h
10. Write a **decrease()** function to modify the keyvalue fields for all neighboring nodes of the node being deleted from h