Algorithms and Data Structures (II)

Gabriel Istrate

May 20, 2020

Where are we:

Graphs: Representations and Algorithms

- Continue with Minimum Spanning Tree.
- Data Structures for external memory: B-Trees.

Minimum Spanning Tree: Algorithms

Minimum Spanning Tree: Algorithms

- Kruskal's algorithm (1956)
  - based on the generic minimum-spanning-tree algorithm (last time)
  - incrementally builds a ***forest*** A

Minimum Spanning Tree: Algorithms

- Kruskal's algorithm (1956)
  - based on the generic minimum-spanning-tree algorithm (last time)
  - incrementally builds a ***forest*** A
- Prim's algorithm (1957)
  - based on the generic minimum-spanning-tree algorithm
  - incrementally builds a ***single tree*** A

Disjoint-Set Data Structure

- *Make-Set*(x) creates a set containing x

- *Make-Set*(x) creates a set containing x
- *Find-Set*(x) returns the *representative* of the set containing x
  - $x, y \in S \Rightarrow \textit{Find-Set}(x) = \textit{Find-Set}(y)$
  - $x \in S_1 \wedge y \in S_2 \wedge S_1 \neq S_2 \Rightarrow \textit{Find-Set}(x) \neq \textit{Find-Set}(y)$

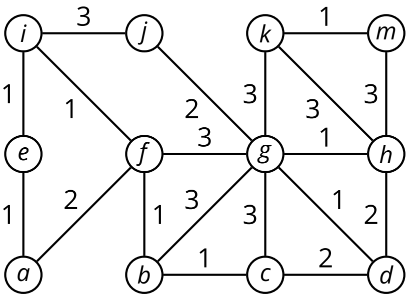
- *Make-Set*(x) creates a set containing x
- *Find-Set*(x) returns the *representative* of the set containing x
  - $x, y \in S \Longrightarrow \textit{Find-Set}(x) = \textit{Find-Set}(y)$
  - $x \in S_1 \land y \in S_2 \land S_1 \neq S_2 \Longrightarrow \textit{Find-Set}(x) \neq \textit{Find-Set}(y)$
- *Union*(x, y) joins the sets containing x and y

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                     // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

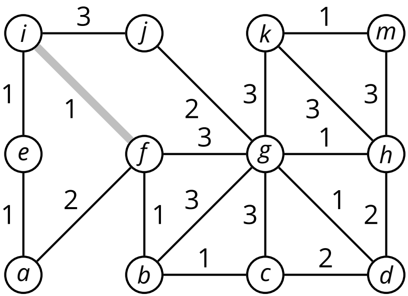
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



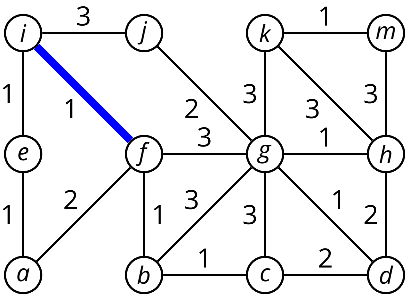
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



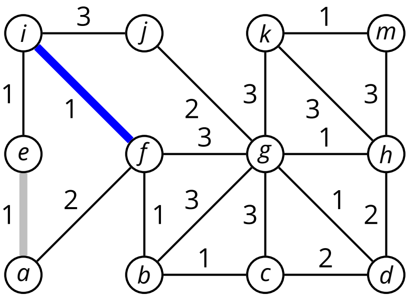
MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```

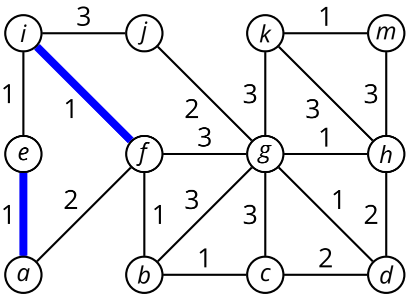


MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```

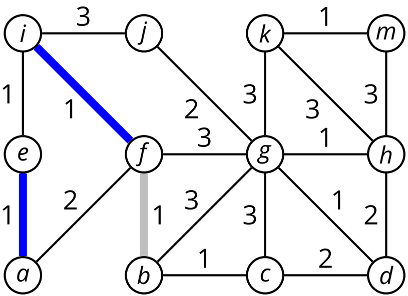


```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



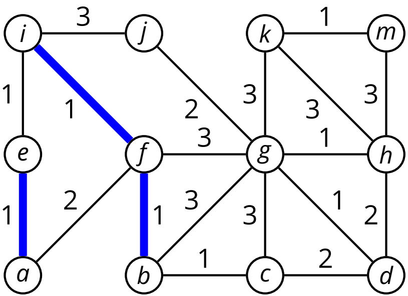
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



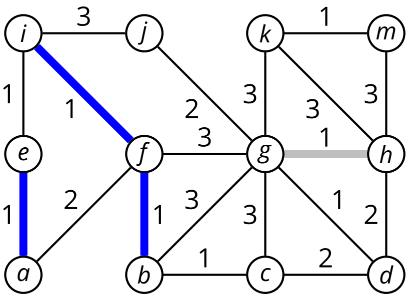
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```

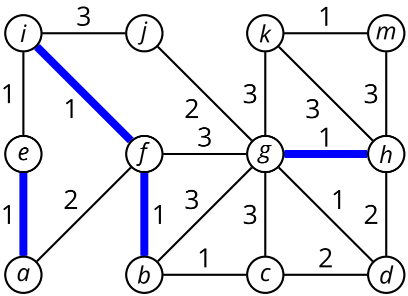


Kruskal's Algorithm

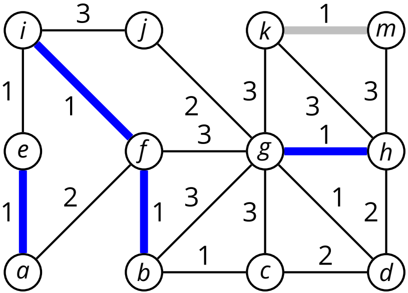
```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

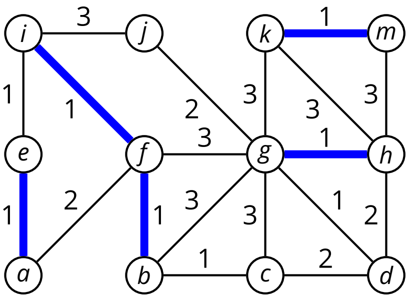


```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



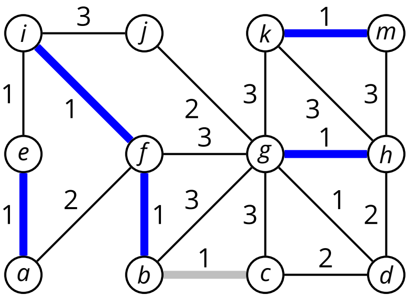
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



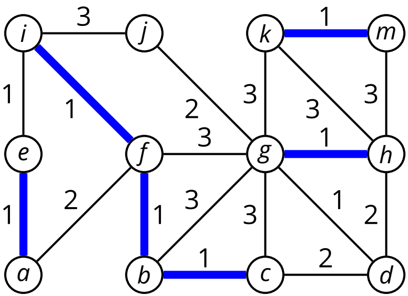
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



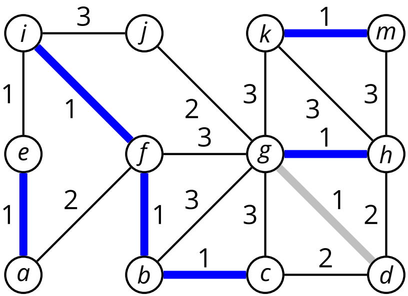
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



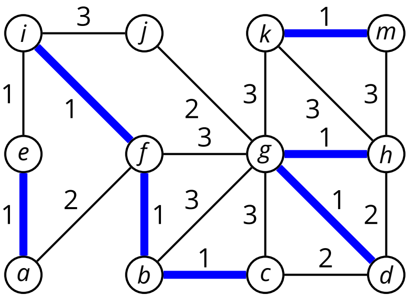
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                       // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



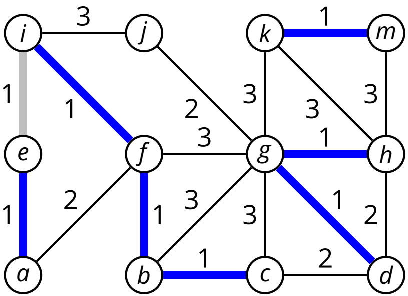
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



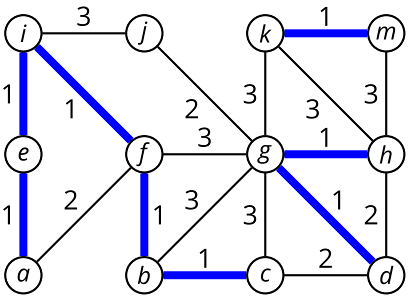
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                       // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



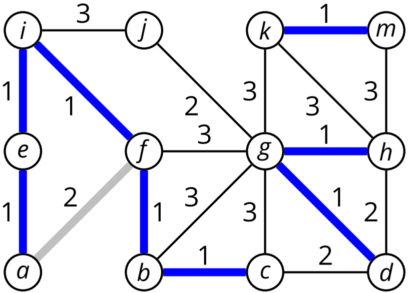
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
              A = A ∪ {(u, v)}
              UNION(u, v)
```

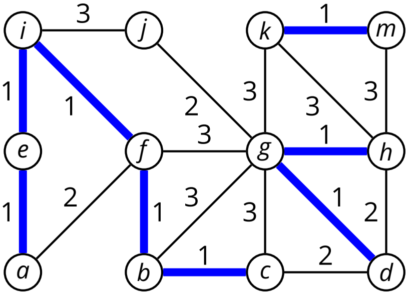


Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```

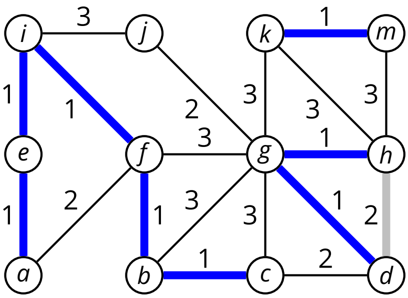


```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



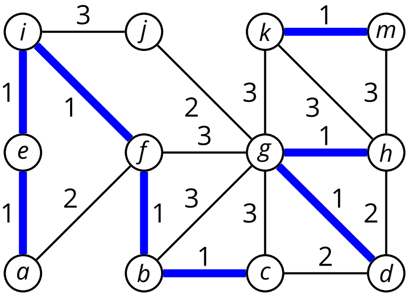
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



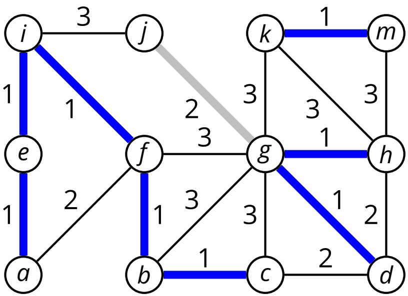
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



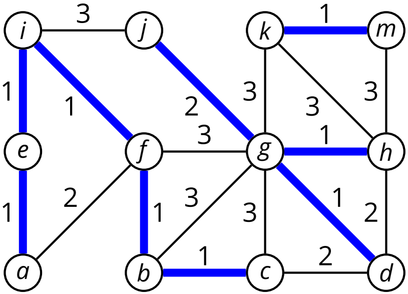
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



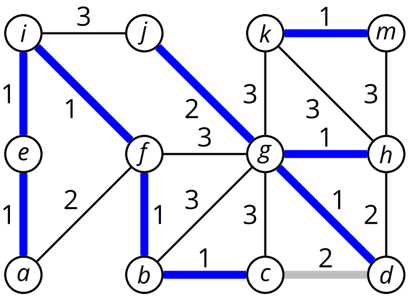
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



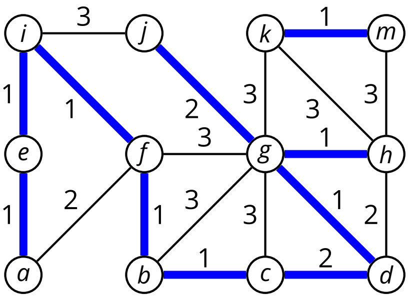
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



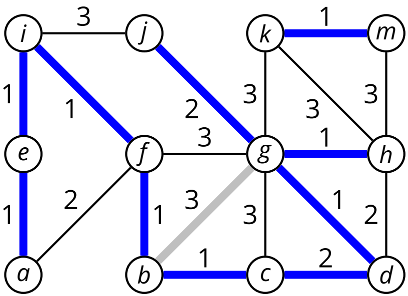
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



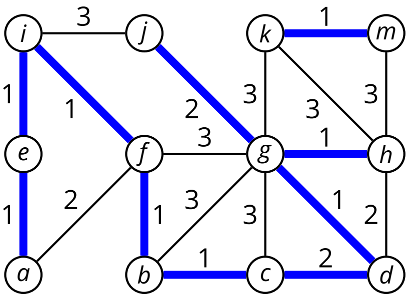
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



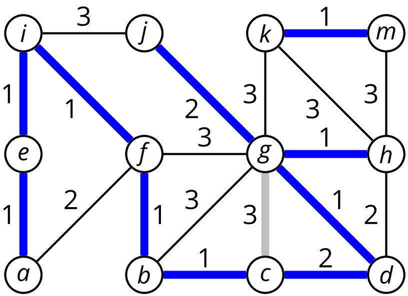
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



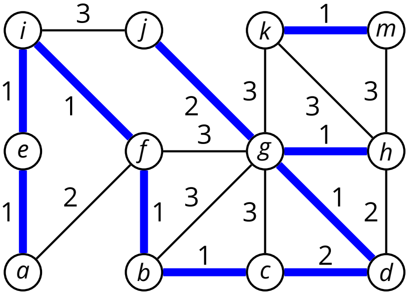
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



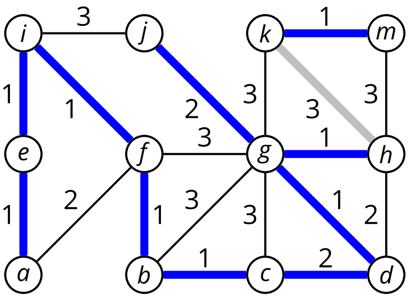
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



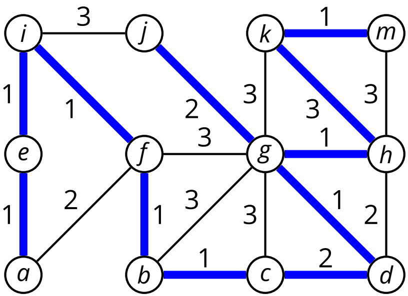
MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



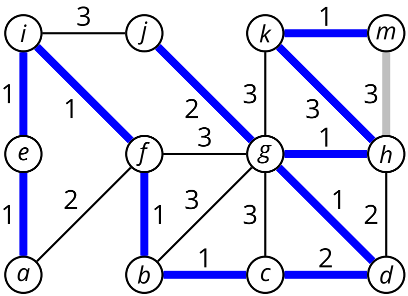
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



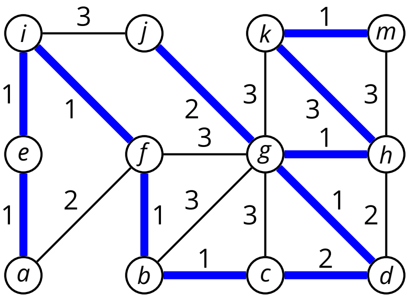
MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



Kruskal's Algorithm

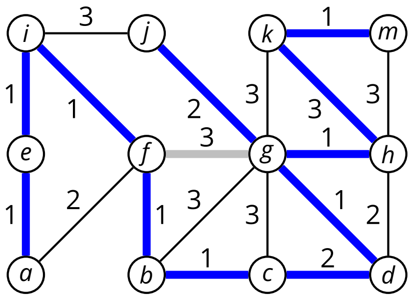
```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



Kruskal's Algorithm

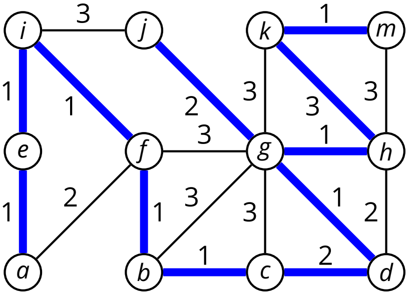
MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



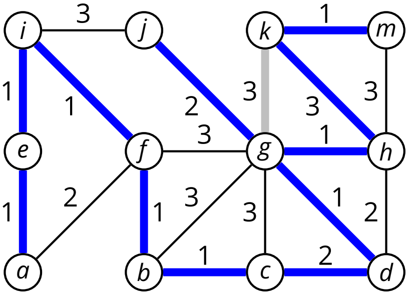
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                        // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



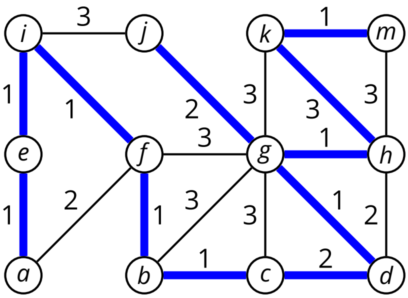
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



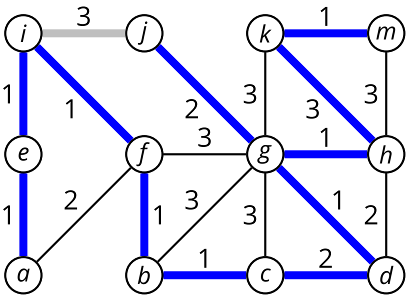
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



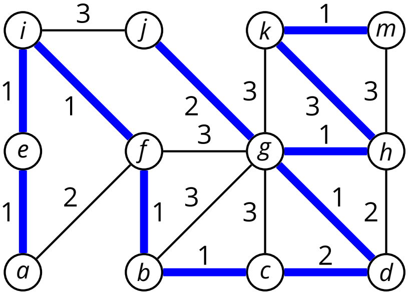
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3        MAKE-SET(v)                         // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6        if FIND-SET(u) ≠ FIND-SET(v)
7              A = A ∪ {(u, v)}
8              UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3        MAKE-SET(v)                    // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6        if FIND-SET(u) ≠ FIND-SET(v)
7              A = A ∪ {(u, v)}
8              UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)

Complexity of MST-KRUSKAL

MST-KRUSKAL(G, w)

1 $A = \emptyset$
2 **for** each vertex $v \in V(G)$
3   **MAKE-SET**(v) // *disjoint-set* data structure
4 sort E in non-decreasing order by weight w
5 **for** each edge $(u, v) \in E$, taken in non-decreasing order by w
6   **if FIND-SET**$(u) \neq$ **FIND-SET**(v)
7         $A = A \cup \{(u, v)\}$
8   **UNION**(u, v)

- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)
- $2|E|$ times **FIND-SET**

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3      MAKE-SET(v)                    // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)
- $2|E|$ times **FIND-SET**
- $O(|E|)$ times **UNION**

So What's the Complexity of MST-KRUSKAL ?

- depends on the complexity of implementing DISJOINT-SET operations.

So What's the Complexity of MST-KRUSKAL ?

- depends on the complexity of implementing DISJOINT-SET operations.
- Can be done (complicated way) so that complexity of Kruskal: $O(Elog(V))$.

So What's the Complexity of MST-KRUSKAL ?

- depends on the complexity of implementing DISJOINT-SET operations.
- Can be done (complicated way) so that complexity of Kruskal: $O(E\log(V))$.
- "Union-by-rank" and "path compression" heuristics.

So What's the Complexity of MST-KRUSKAL ?

- depends on the complexity of implementing DISJOINT-SET operations.
- Can be done (complicated way) so that complexity of Kruskal: $O(E\log(V))$.
- "Union-by-rank" and "path compression" heuristics.
- Cormen: Chapter 21.

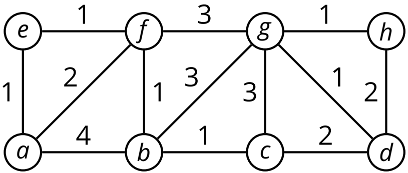
Remember

- In Kruskal's algorithm: grow a forest.
- In Prim's algorithm: grow a tree.
- Use: min-priority queue.
- Remember ? First semester. Implemented using heaps.
- Elements have "priorities". An element with high priority is served before an element with low priority.
- Operations:
- IS_EMPTY: $O(1)$.
- INSERT_WITH_PRIORITY: $O(\log n)$.
- POP: $O(\log n)$.
- PEEK: $O(1)$.
- (BUILD): $O(n)$.
- ADVANCED: Can improve over the heap implementation !

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```

Prim's Algorithm

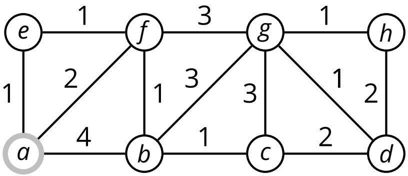
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$$Q = \{(a, 0, \cdot), (b, \infty, \cdot), (c, \infty, \cdot), (d, \infty, \cdot), (e, \infty, \cdot), (f, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

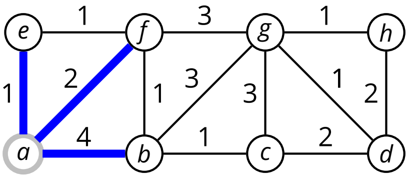
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$Q = \{(b, \infty, \cdot), (c, \infty, \cdot), (d, \infty, \cdot), (e, \infty, \cdot), (f, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

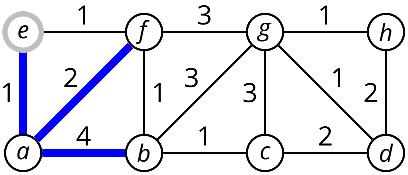
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(e, 1, a), (f, 2, a), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim’s Algorithm

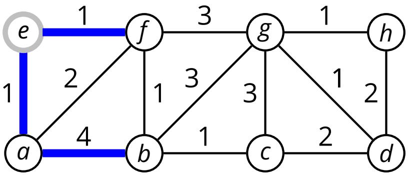
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(f, 2, a), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

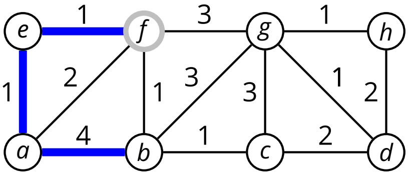
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$$Q = \{(f, 1, e), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

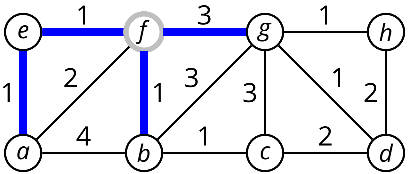
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

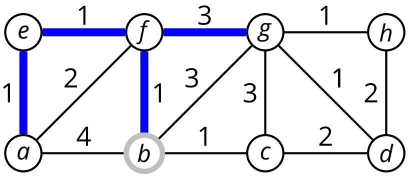
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$$Q = \{(b, 1, f), (g, 3, f), (c, \infty, \cdot), (d, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

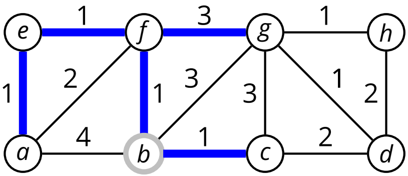
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$$Q = \{(g, 3, f), (c, \infty, \cdot), (d, \infty, \cdot), (h, \infty, \cdot)\}$$

MST-PRIM(G, w, r)

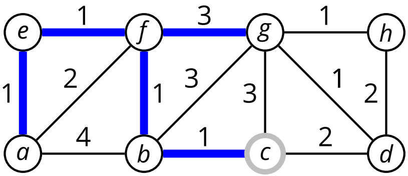
```
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(c, 1, b), (g, 3, f), (d, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

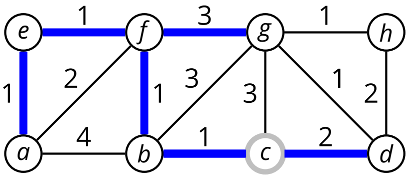
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$$Q = \{(g, 3, f), (d, \infty, \cdot), (h, \infty, \cdot)\}$$

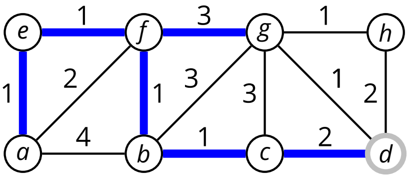
Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



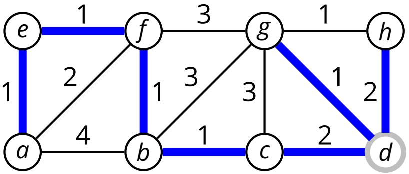
$Q = \{(d, 2, c), (g, 3, f), (h, \infty, \cdot)\}$

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$$Q = \{(g, 3, f), (h, \infty, \cdot)\}$$

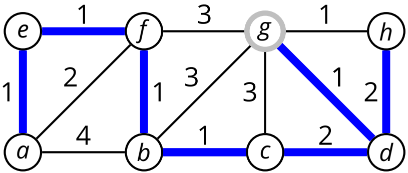
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$Q = \{(g, 1, d), (h, 2, d)\}$

Prim's Algorithm

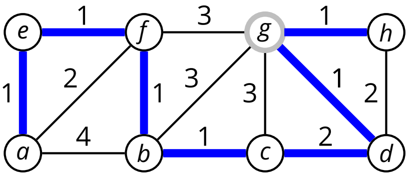
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(h, 2, d)\}$

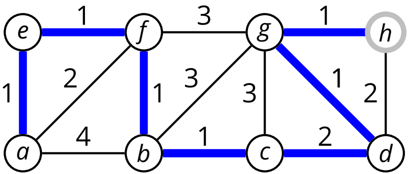
Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(h, 1, g)\}$

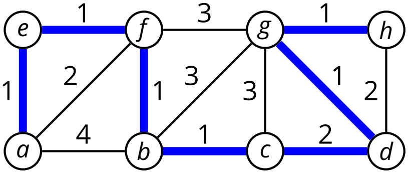
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{\}$

Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$Q = \{\}$

Complexity of Prim's Algorithm.

Come on, not again !

- depends on the complexity of implementing PRIORITY QUEUE.

Complexity of Prim's Algorithm.

Come on, not again !

- depends on the complexity of implementing PRIORITY QUEUE.
- With binary min-heap: $O(Elog(V))$.

Complexity of Prim's Algorithm.

Come on, not again !

- depends on the complexity of implementing PRIORITY QUEUE.
- With binary min-heap: $O(Elog(V))$.
- Can be done (more complicated way) so that complexity of Prim: $O(E + Vlog(V))$.

Complexity of Prim's Algorithm.

Come on, not again !

- depends on the complexity of implementing PRIORITY QUEUE.
- With binary min-heap: $O(Elog(V))$.
- Can be done (more complicated way) so that complexity of Prim: $O(E + Vlog(V))$.
- Cormen: Chapter 20 (Fibonacci heaps).

Why do we care about MST ?

- One answer: neat greedy algorithms.

Why do we care about MST ?

- One answer: neat greedy algorithms.
- MST: many applications ! (see Wikipedia). Subroutine/heuristic in many scenarios. E.g. TSP.

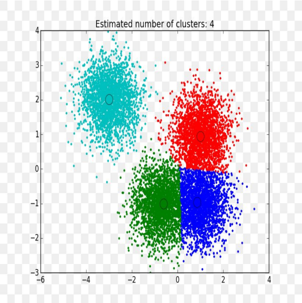
Why do we care about MST ?

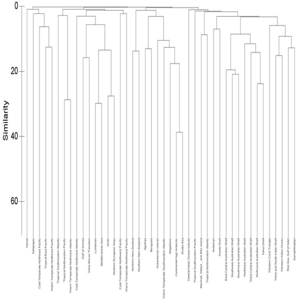
- One answer: neat greedy algorithms.
- MST: many applications ! (see Wikipedia). Subroutine/heuristic in many scenarios. E.g. TSP.
- One particular application: clustering.

Why do we care about MST ?

- One answer: neat greedy algorithms.
- MST: many applications ! (see Wikipedia). Subroutine/heuristic in many scenarios. E.g. TSP.
- One particular application: clustering.
- Where do we go from here ? Matroids. “Abstract” formalism that generalizes vector spaces and spanning trees (!). Cormen, 16.4.

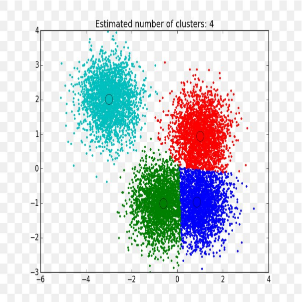
Clustering using MST algorithms.

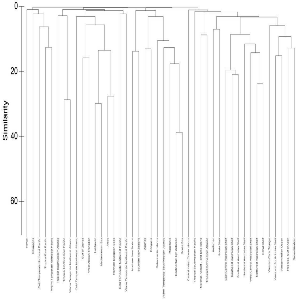




- Clustering: *n* points into some high-dimensional space. They may be measurements, features, etc.

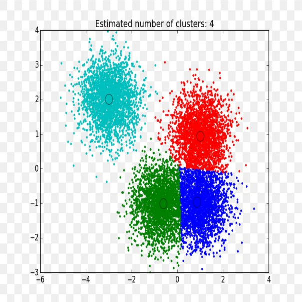
Clustering using MST algorithms.

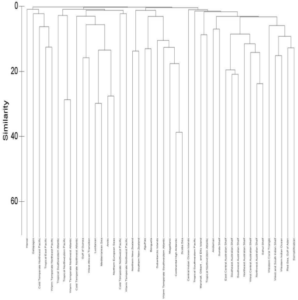




- Clustering: *n* points into some high-dimensional space. They may be measurements, features, etc.
- to group the points into *k* “clusters” so that points in the same clusters are “close” and points in different clusters are “far apart”

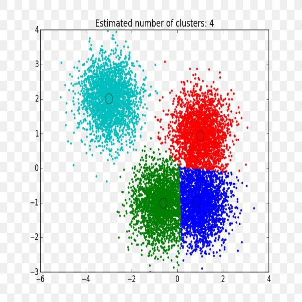
Clustering using MST algorithms.

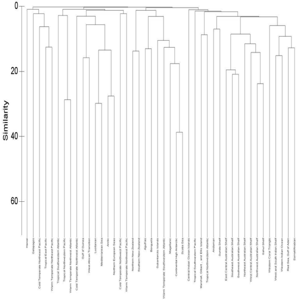




- Clustering: *n* points into some high-dimensional space. They may be measurements, features, etc.
- to group the points into *k* “clusters” so that points in the same clusters are “close” and points in different clusters are “far apart”
- Kruskal’s algorithm: points start into *n* clusters, eventually coalesce into one.

Clustering using MST algorithms.





- Clustering: *n* points into some high-dimensional space. They may be measurements, features, etc.
- to group the points into *k* "clusters" so that points in the same clusters are "close" and points in different clusters are "far apart"
- Kruskal's algorithm: points start into *n* clusters, eventually coalesce into one.
- Why not stop when there are *k* clusters ? (or compute MST and remove heaviest edges) ?

Outline:

- Search in secondary storage
- B-Trees
  - properties
  - search
  - insertion

Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all “flat”

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all “flat”

- However, some applications require more storage than what fits in main memory
  - we must use data structures that reside in *secondary storage* (i.e., disk)

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all “flat”
- However, some applications require more storage than what fits in main memory
  - we must use data structures that reside in *secondary storage* (i.e., disk)

$\boxed{\textit{Disk is 10,000–100,000 times slower than RAM}}$

Memory access/transfer	**CPU cycles (≈ 1ns)**
Register	***1***

Memory access/transfer	**CPU cycles (≈ 1ns)**
Register	***1***
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
Remote DRAM (main memory)	***100***

Memory access/transfer	CPU cycles (≈ 1ns)
Register	***1***
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
Remote DRAM (main memory)	***100***
SSD seek	***20,000***

Memory access/transfer	CPU cycles ($\approx$ 1ns)
Register	***1***
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
Remote DRAM (main memory)	***100***
SSD seek	***20,000***
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within a datacenter	500,000

Memory access/transfer	**CPU cycles (≈ 1ns)**
Register	***1***
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
Remote DRAM (main memory)	***100***
SSD seek	***20,000***
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within a datacenter	500,000
HDD seek	***10,000,000***
Read 1 MB sequentially from network	10,000,000
Read 1 MB sequentially from disk	30,000,000
Round-trip time USA–Europe	150,000,000

Modeling Disk Access

Modeling Disk Access

- Let x be a pointer to some (possibly complex) object

- Let x be a pointer to some (possibly complex) object
- When the object is in memory, x can be used directly as a reference to the object
  - e.g., $\ell = x.size$ or $x.root = y$

- Let x be a pointer to some (possibly complex) object
- When the object is in memory, x can be used directly as a reference to the object
  - e.g., $\ell = x.size$ or $x.root = y$
- When the object is on disk, we must first perform a disk-read operation

  Disk-Read(x) reads the object into memory, allowing us to refer to it (and modify it) through x

- Let x be a pointer to some (possibly complex) object
- When the object is in memory, x can be used directly as a reference to the object
  - e.g., $\ell = x.size$ or $x.root = y$
- When the object is on disk, we must first perform a disk-read operation

  DISK-READ(x) reads the object into memory, allowing us to refer to it (and modify it) through x
- Any changes to the object in memory must be eventually saved onto the disk

  DISK-WRITE(x) writes the object onto the disk (if the object was modified)

Binary Trees on Disk

- Assume each node x is stored on disk

Binary Trees on Disk

- Assume each node x is stored on disk

```
ITERATIVE-TREE-SEARCH(T, k)
x = T.root
while x ≠ NIL
        DISK-READ(X)
        if k == x.key
                return x
        elseif k < x.key
                x = x.left
        else x = x.right
return x
```

- Assume each node x is stored on disk

```
ITERATIVE-TREE-SEARCH(T, k)
x = T.root
while x ≠ NIL
        DISK-READ(X)
        if k == x.key
                return x
        elseif k < x.key
                x = x.left
        else x = x.right
return x
```

cost

- Assume each node x is stored on disk

ITERATIVE-TREE-SEARCH(T, k)	*cost*
$x = T.root$	c
while $x \neq$ NIL	c
DISK-READ(X)	$100000c$
if $k == x.key$	c
return x	c
elseif $k < x.key$	c
$x = x.left$	c
else $x = x.right$	c
return x	c

Basic Intuition

Basic Intuition

- Assume we store the nodes of a search tree on disk

  1. node accesses should be reduced to a minimum
  2. spending more than a few basic operations for each node is not a problem

- Assume we store the nodes of a search tree on disk
  1. node accesses should be reduced to a minimum
  2. spending more than a few basic operations for each node is not a problem

- Rationale
  - basic in-memory operations are much cheaper
  - the bottleneck is with node accesses, which involve **DISK-READ** and **DISK-WRITE** operations

Idea

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

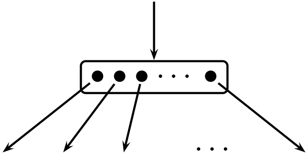
- **Idea:** store several keys and pointers to children nodes in a single node

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

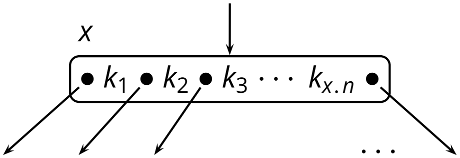
- **Idea:** store several keys and pointers to children nodes in a single node
  - in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - in practice d can be as high as a few thousands

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

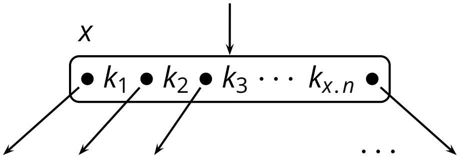
- **Idea:** store several keys and pointers to children nodes in a single node
  - in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - in practice d can be as high as a few thousands



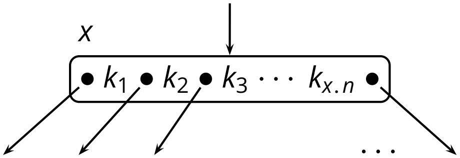
E.g., if $d = 1000$, then
only three accesses ($h = 2$)
cover ***up to one billion keys***



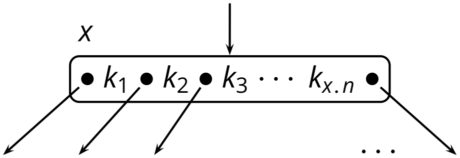
x
k_1 k_2 k_3 $\cdots$ $k_{x.n}$
$\cdots$



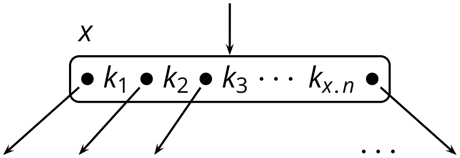
- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node



- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order

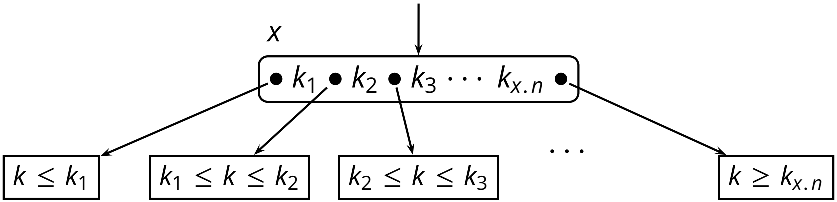


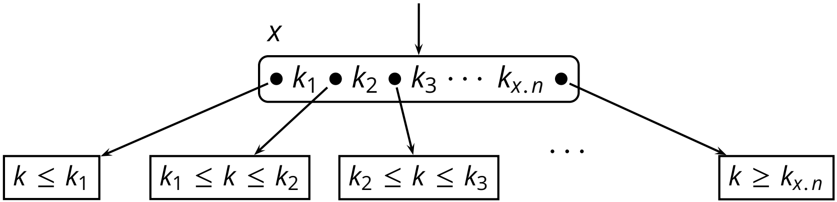
- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*



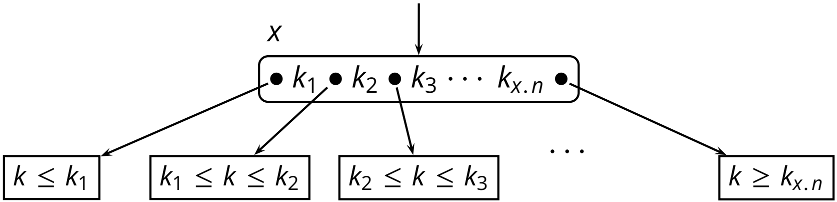
- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*
  - $x.c[1], x.c[2], \ldots, x.c[x.n+1]$ are the $x.n+1$ pointers to its children, if x is an *internal node*

Definition of a B-Tree (2)





- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree



- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree

  $x.c[1] \longrightarrow$ subtree containing keys $k \leq x.key[1]$
  $x.c[2] \longrightarrow$ subtree containing keys k, $x.key[1] \leq k \leq x.key[2]$
  $x.c[3] \longrightarrow$ subtree containing keys k, $x.key[2] \leq k \leq x.key[3]$
  $\ldots$
  $x.c[x.n+1] \longrightarrow$ subtree containing keys k, $k \geq x.key[x.n]$

Definition of a B-Tree (3)

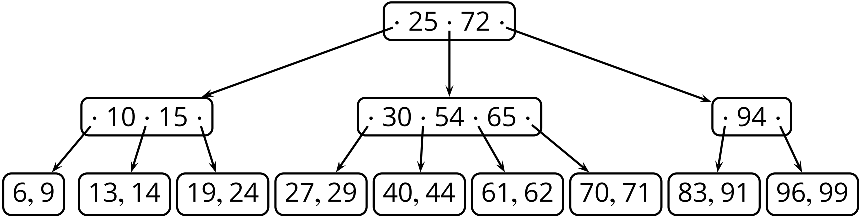
Definition of a B-Tree (3)

- ***All leaves have the same depth***

Definition of a B-Tree (3)

- ***All leaves have the same depth***
- Let $t \geq 2$ be the ***minimum degree*** of the B-tree
  - every node other than the root must have ***at least*** $t - 1$ ***keys***
  - every node must contain ***at most*** $2t - 1$ ***keys***
    - a node is *full* when it contains exactly $2t - 1$ keys
    - a full node has $2t$ children

Example


· 25 · 72 ·
· 10 · 15 ·
· 30 · 54 · 65 ·
· 94 ·
6, 9
13, 14
19, 24
27, 29
40, 44
61, 62
70, 71
83, 91
96, 99

Search in B-Trees

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key[i]
        i = i + 1
if i ≤ x.n and k == x.key[i]
        return (x, i)
if x.leaf
        return NIL
else DISK-READ(x.c[i])
        return B-TREE-SEARCH(x.c[i], k)
```

Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

  - $n \geq 1$, so the root has at least one key (and therefore two children)

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least t children

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

  - $n \geq 1$, so the root has at least one key (and therefore two children)
  - every other node has at least t children
  - in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t-1$ keys

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least t children
- in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t - 1$ keys
- each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t - 1$ keys

Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

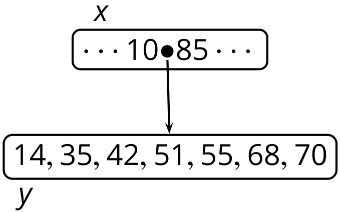
Proof:

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least t children
- in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t-1$ keys
- each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t-1$ keys, so

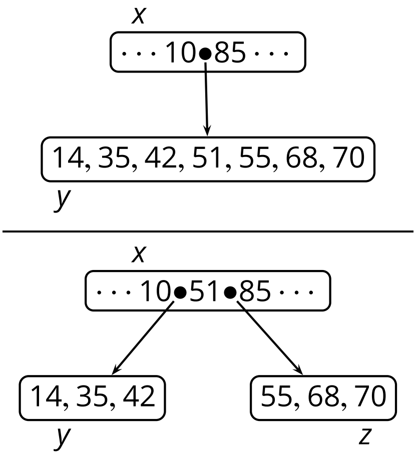
$$n \geq 1 + 2(t^h - 1)$$

Splitting

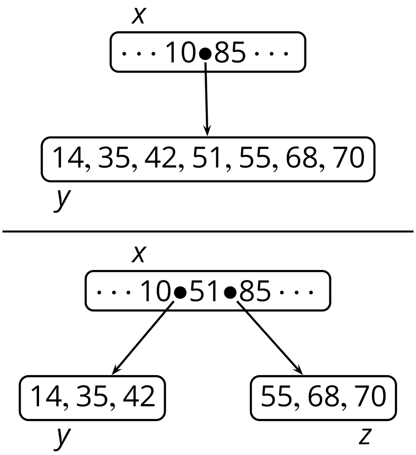
Splitting



Splitting



Splitting




```
B-TREE-SPLIT-CHILD(x, i, y)
z = ALLOCATE-NODE()
z.leaf = y.leaf
z.n = t − 1
for j = 1 to t − 1
    z.key[j] = y.key[j + t]
if not y.leaf
    for j = 1 to t
        z.c[j] = y.c[j + t]
y.n = t − 1
for j = x.n + 1 downto i + 1
    x.c[j + 1] = x.c[j]
for j = x.n downto i
    x.key[j + 1] = x.key[j]
x.key[i] = y.key[t]
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

Complexity of B-Tree-Split-Child

- What is the complexity of **B-Tree-Split-Child**?

Complexity of B-TREE-SPLIT-CHILD

- What is the complexity of **B-TREE-SPLIT-CHILD**?
- $\Theta(t)$ basic CPU operations

Complexity of B-TREE-SPLIT-CHILD

- What is the complexity of **B-TREE-SPLIT-CHILD**?
- $\Theta(t)$ basic CPU operations
- 3 **DISK-WRITE** operations

```
B-TREE-SPLIT-CHILD(x, i, y)
    z = ALLOCATE-NODE()
    z.leaf = y.leaf
    z.n = t − 1
    for j = 1 to t − 1
        x.key[j] = x.key[j + t]
    if not x.leaf
        for j = 1 to t
            z.c[j] = y.c[j + t]
    y.n = t − 1
    for j = x.n + 1 downto i + 1
        x.c[j + 1] = x.c[j]
    for j = x.n downto i
        x.key[j + 1] = x.key[j]
    x.key[i] = y.key[t]
    x.n = x.n + 1
    DISK-WRITE(y)
    DISK-WRITE(z)
    DISK-WRITE(x)
```

Insertion Under Non-Full Node

Insertion Under Non-Full Node

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n                                   // assume x is not full
if x.leaf
    while i ≥ 1 and k < x.key[i]
        x.key[i + 1] = x.key[i]
        i = i − 1
    x.key[i + 1] = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key[i]
        i = i − 1
    i = i + 1
    DISK-READ(x.c[i])
    if x.c[i].n == 2t − 1                 // child x.c[i] is full
        B-TREE-SPLIT-CHILD(x, i, x.c[i])
        if k > x.key[i]
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c[i], k)
```

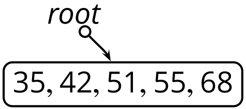
Insertion Procedure

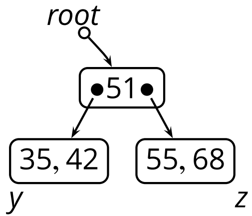
Insertion Procedure

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
        s = ALLOCATE-NODE()
        T.root = s
        s.leaf = FALSE
        s.n = 0
        s.c[1] = r
        B-TREE-SPLIT-CHILD(s, 1, r)
        B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Insertion Procedure

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
      s = ALLOCATE-NODE()
      T.root = s
      s.leaf = FALSE
      s.n = 0
      s.c[1] = r
      B-TREE-SPLIT-CHILD(s, 1, r)
      B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```





Complexity of Insertion

- What is the complexity of **B-TREE-INSERT**?

Complexity of Insertion

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations

Complexity of Insertion

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations
- $O(h) = O(\log_t n)$ disk-access operations

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations
- $O(h) = O(\log_t n)$ disk-access operations
- The best value for t can be determined according to
  - the ratio between CPU (RAM) speed and disk-access time
  - the *block-size* of the disk, which determines the maximum size of an object that can be accessed (read/write) in one shot