Northern Illinois University

Review

Dr. Maoyuan Sun – smaoyuan@niu.edu

Final Exam Schedule

Date: **December 11, 2024**

Time: 10:00 - 11:15 am (75 mins)

Location: PM 110

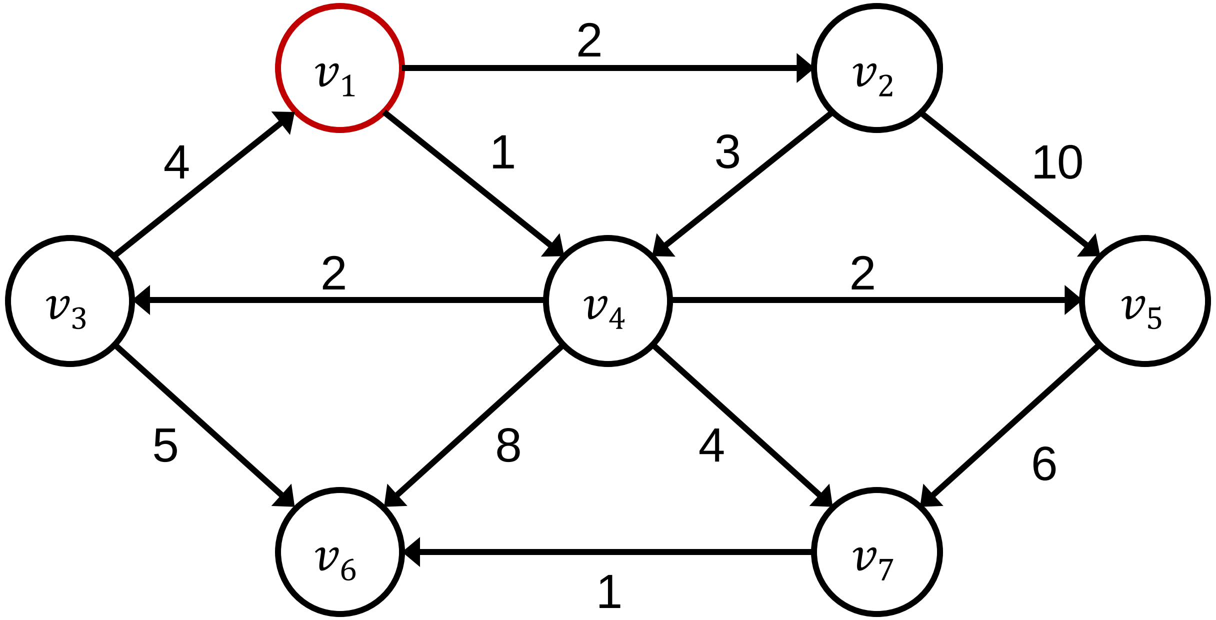
Topics: B-tree, Hash, Graph

Dijkstra's Algorithm

- We choose a v that has the smallest d_v from all unknown vertices and is adjacent to s
- This path is declared the shortest path from s to v and marked known
- The remaining step is updating d_w (we didn't track d_w before, as we just were thinking $d_w = dv + 1$ if $d_w = \infty$) and $d_w = dv + c_{v,w}$ if this new value for d_w would be an improvement
- The algorithm decides if it's a good idea or not to use v on path to w given known cost and new cost

Dijkstra's Algorithm

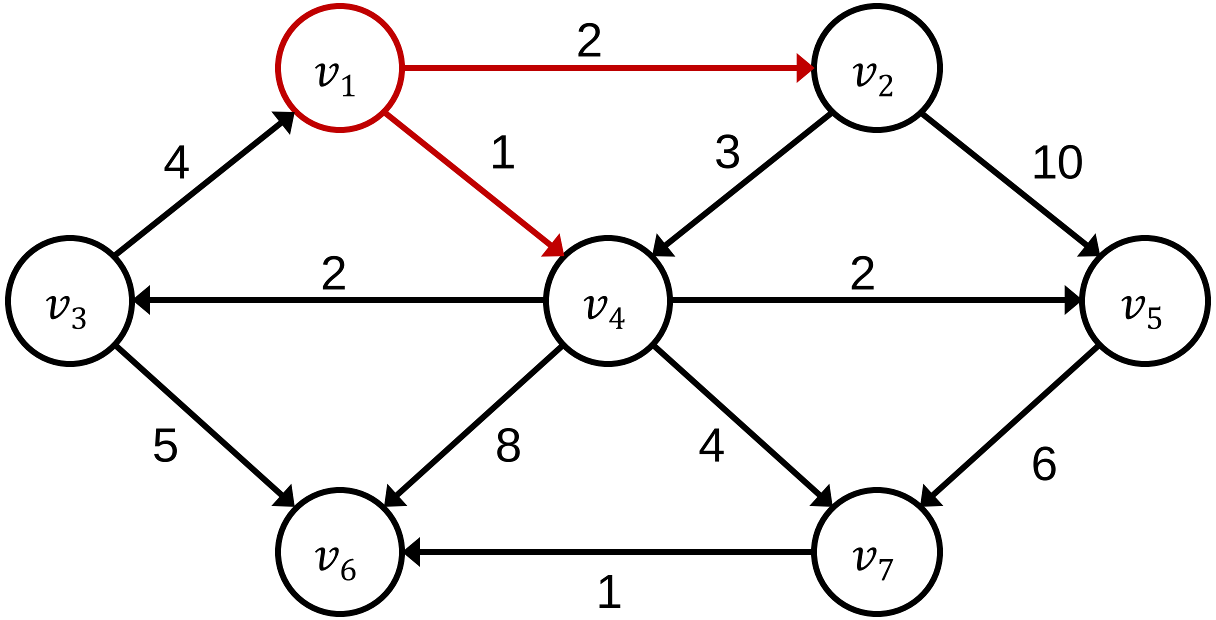
V	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



Pick s to be v_1, the path to v_1 is 0

Dijkstra's Algorithm

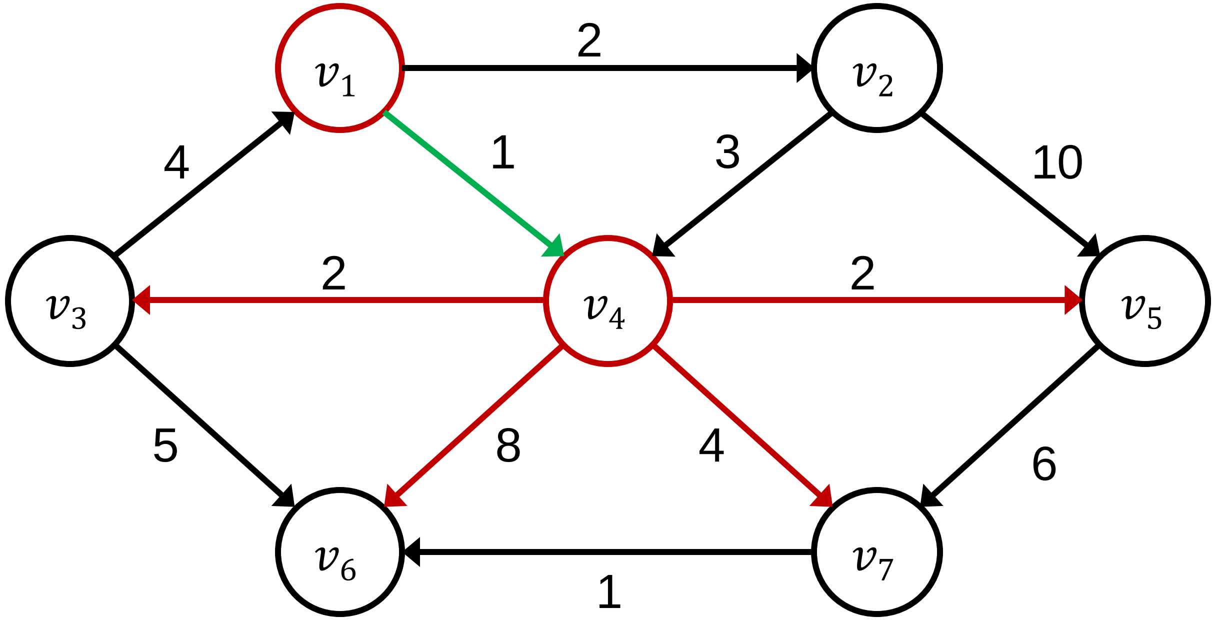
V	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



From v_1 we have path to v_2 and v_4, we choose v_4 (why?)

Dijkstra's Algorithm

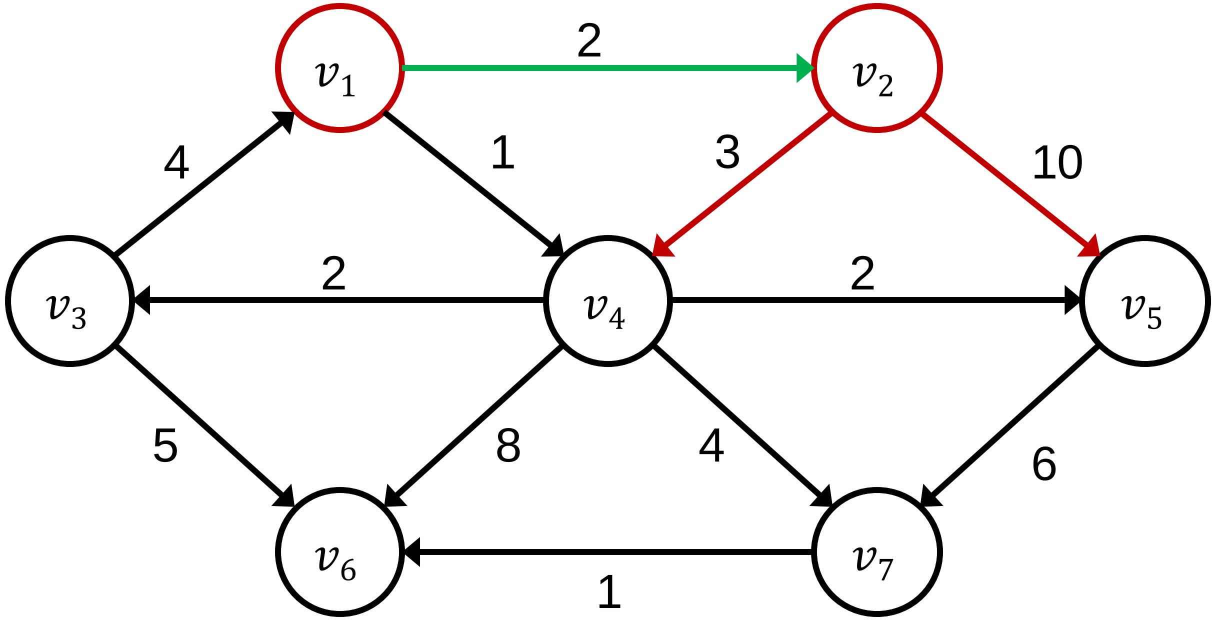
V	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	F	$3\ (1+2)$	v_4
v_6	F	$9\ (1+8)$	v_4
v_7	F	$5\ (1+4)$	v_4



From v_4 we have path to v_3 , v_5, v_6 , v_7, we choose v_2 (new cheapest)

Dijkstra's Algorithm

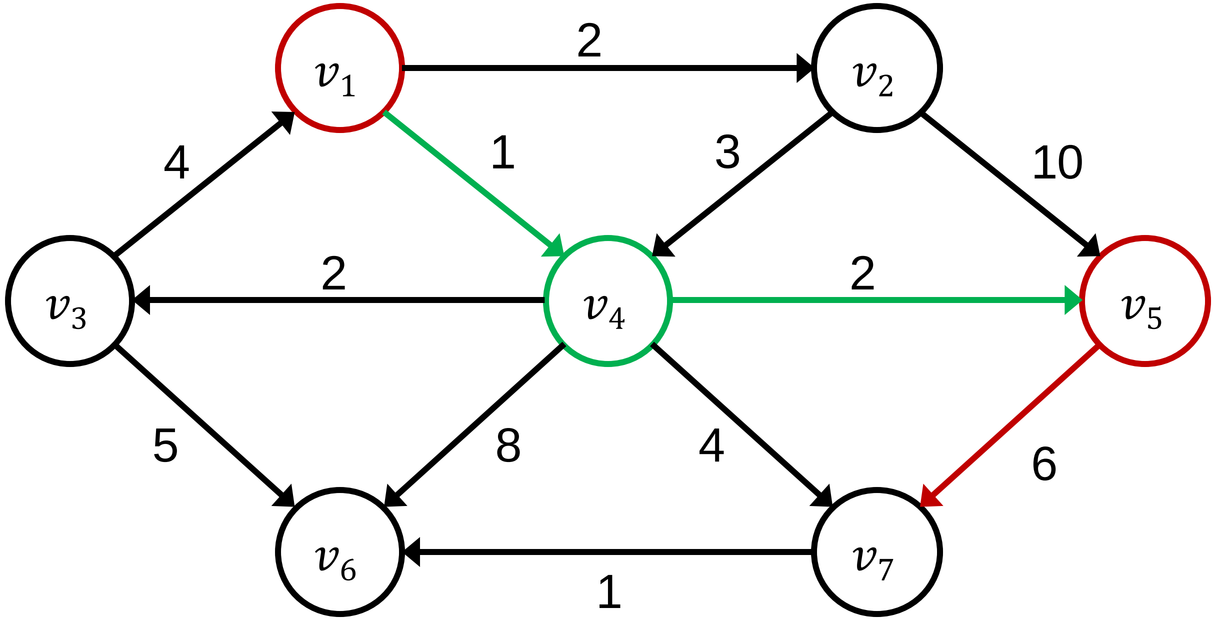
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	F	$3\ (1+2)$	v_4
v_6	F	$9\ (1+8)$	v_4
v_7	F	$5\ (1+4)$	v_4



From v_2 we have path to v_4 , v_5 we look at v_5 (since v_4 is already known) none of the paths are better, v_1 to v_2 to v_5 costs 2 + 10 = 12 > 3

Dijkstra's Algorithm

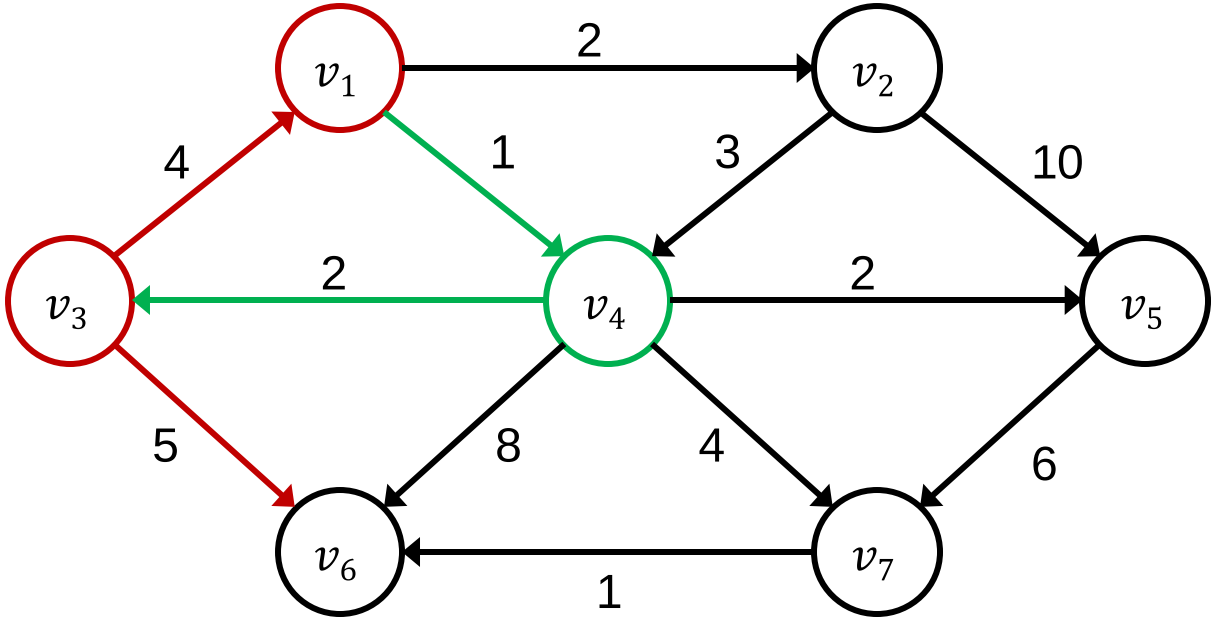
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	T	$3\ (1+2)$	v_4
v_6	F	$9\ (1+8)$	v_4
v_7	F	$5\ (1+4)$	v_4



From v_5 we have path to v_7 none of the paths are better v_1 to v_4 to v_5, 1 + 2 + 6 = 9 > 5 back to selecting the smallest unvisited node which is v_3

Dijkstra's Algorithm

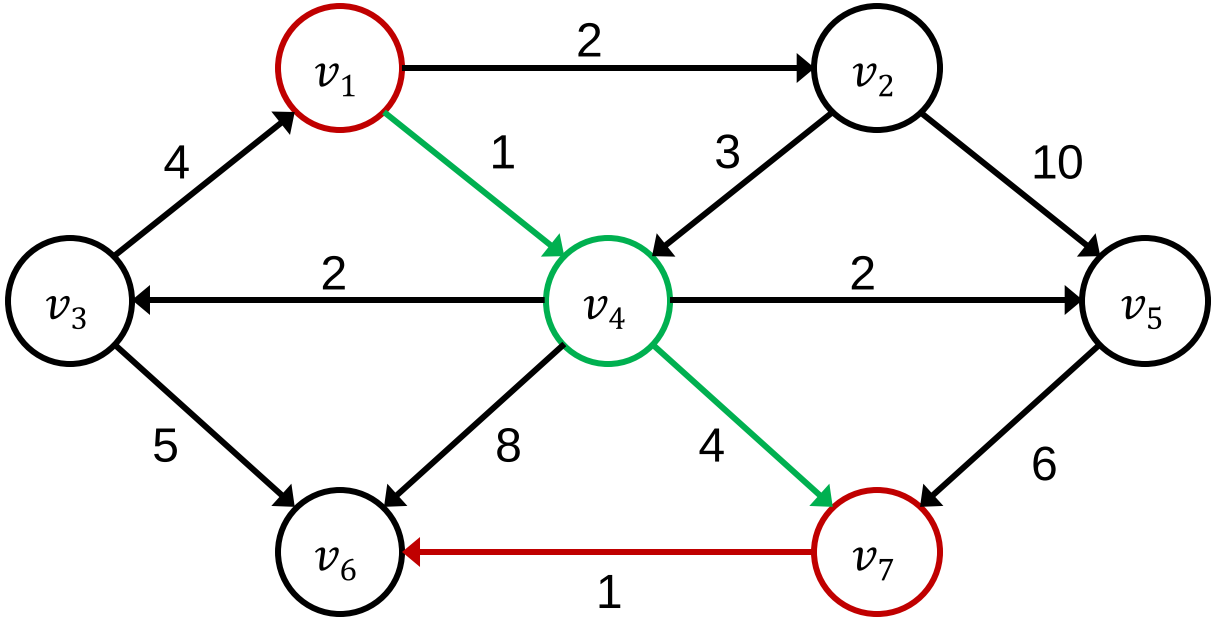
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	T	$3\ (1+2)$	v_4
v_6	F	$8\ (1+2+5)$	v_3
v_7	F	$5\ (1+4)$	v_4



From v_3 we have path to v_1 , v_6 with v_1 cost is $1 + 2 + 4 = 7 > 0$ but v_6 is $1 + 2 + 5 = 8 < 9$, so we update, v_7 is now selected as the smallest

Dijkstra's Algorithm

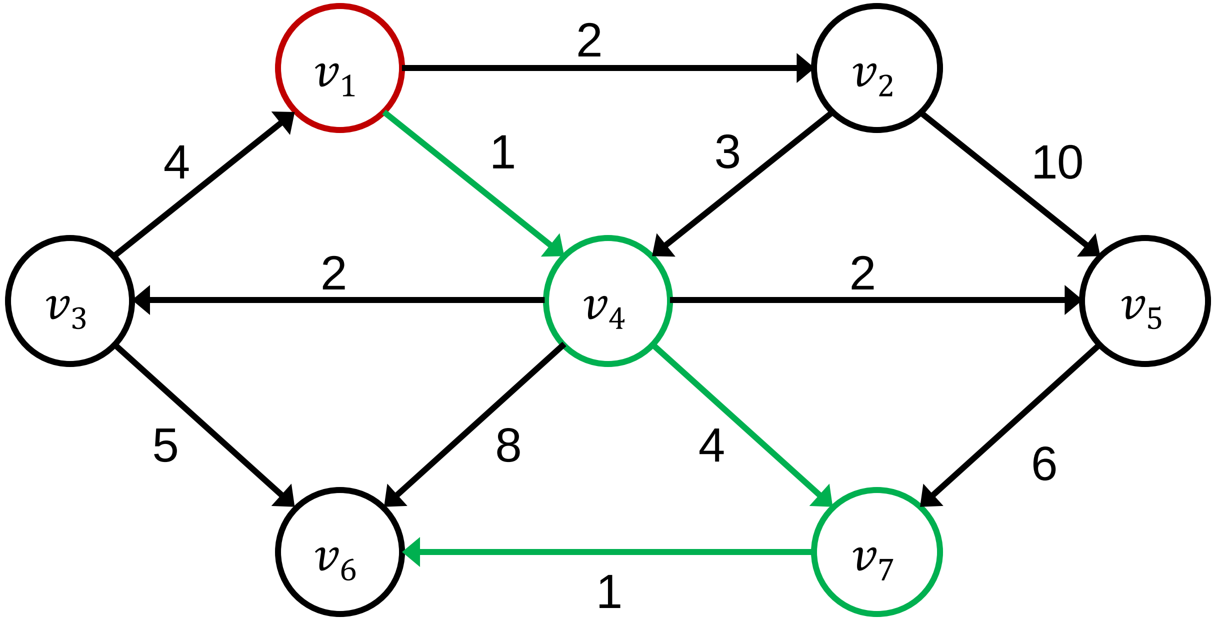
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	T	$3\ (1+2)$	v_4
v_6	F	$6\ (1+4+1)$	v_7
v_7	T	$5\ (1+4)$	v_4



From v_7 we have path to v_6 with cost is 1 + 4+ 1 = 6 < 8, so we update and v_6 is the last one for us to visit

Dijkstra's Algorithm

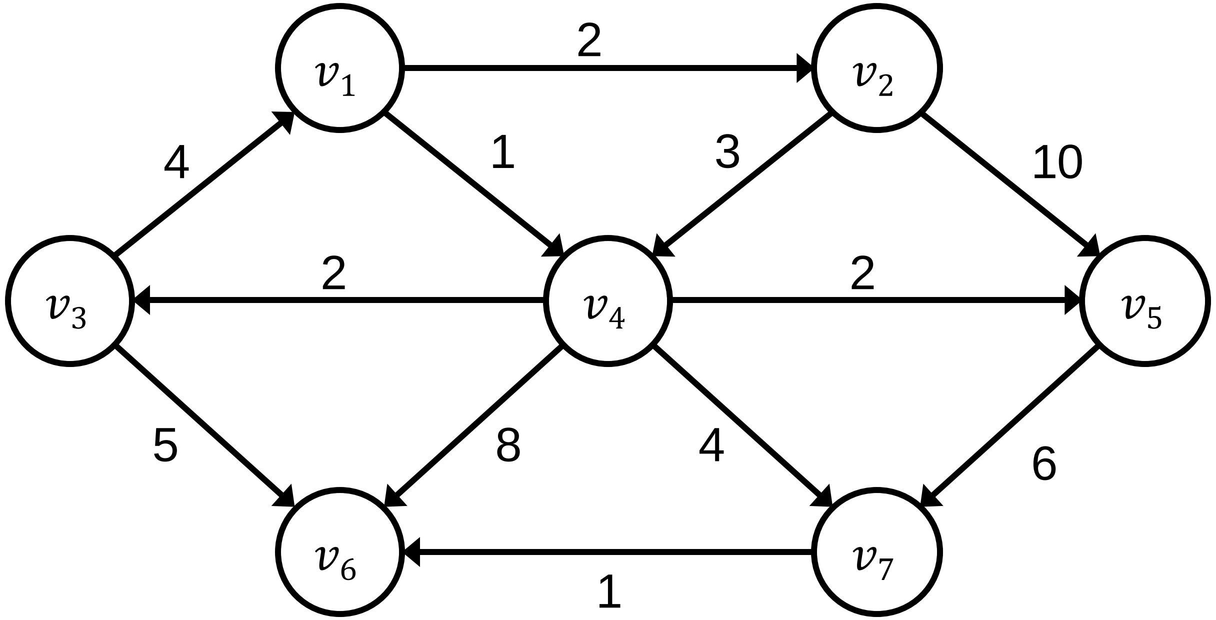
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	$3\ (1+2)$	v_4
v_4	T	1	v_1
v_5	T	$3\ (1+2)$	v_4
v_6	T	$6\ (1+4+1)$	v_7
v_7	T	$5\ (1+4)$	v_4



v_6 no where to visit

Dijkstra's Algorithm

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4



What is the shortest path from:

v_1 to $v_1 \rightarrow v_1$

v_1 to $v_2 \rightarrow v_1 - v_2$

v_1 to $v_3 \rightarrow v_1 - v_4 - v_3$

v_1 to $v_4 \rightarrow v_1 - v_4$

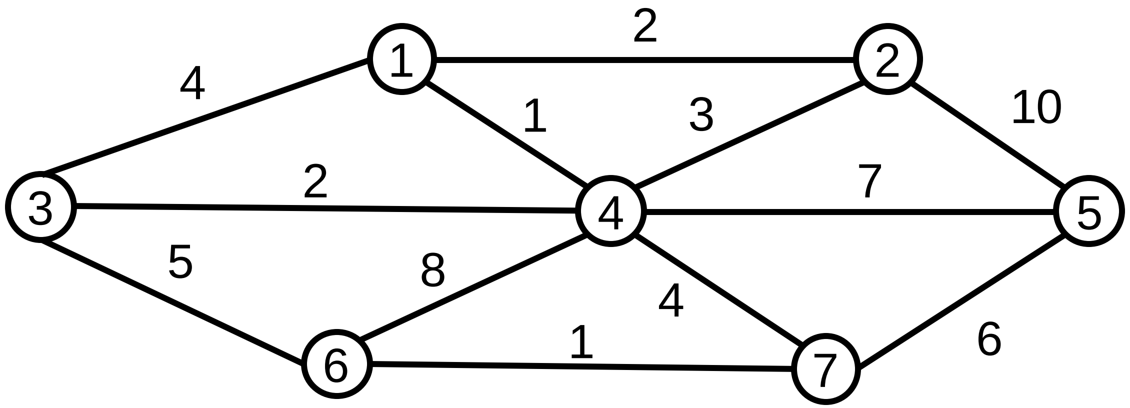
v_1 to $v_5 \rightarrow v_1 - v_4 - v_5$

v_1 to $v_6 \rightarrow v_1 - v_4 - v_7 - v_6$

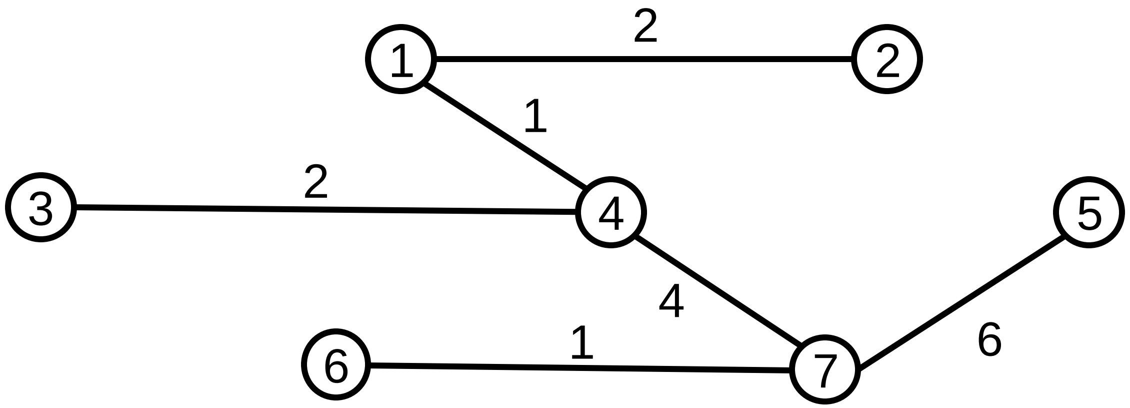
v_1 to $v_7 \rightarrow v_1 - v_4 - v_7$

Minimum Spanning Tree

- A **minimum spanning tree** of an undirected graph G is a formed from the graph edges that connects all the vertices of G at the lowest total cost.
- A minimum spanning tree exists <u>if and only if</u> G is **connected**
- **Reminder**: an undirected graph is **connected** if there is a path from every vertex to every other vertex

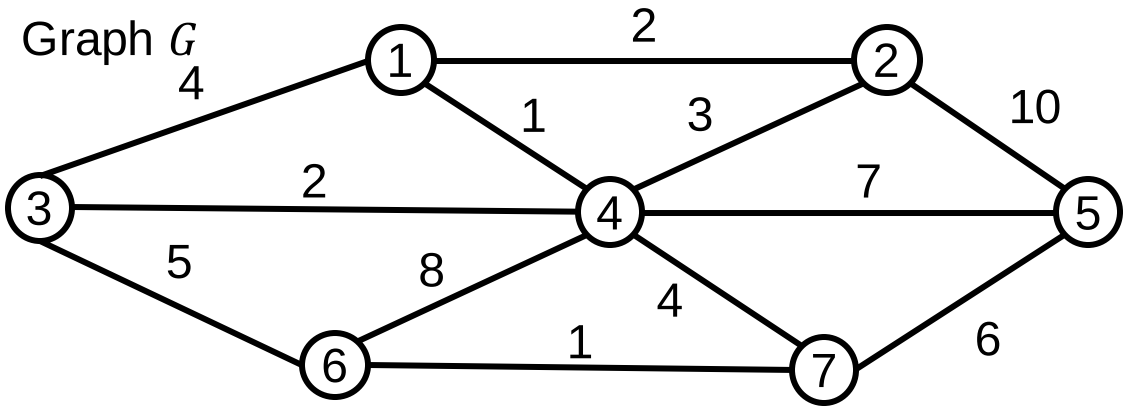


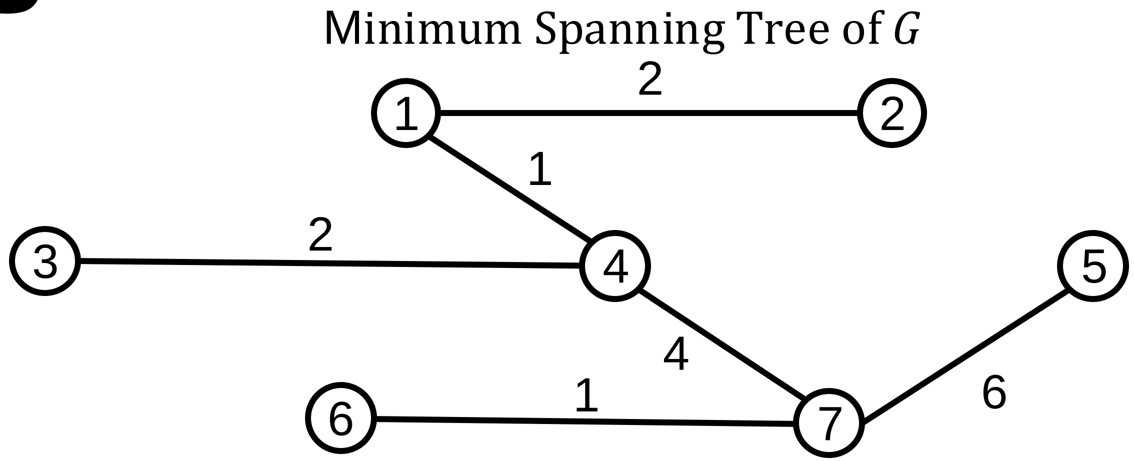
Graph G



Minimum Spanning Tree of G

Minimum Spanning Tree





- The number of edges in the minimum spanning tree is the number of **vertices – 1** (|V| - 1)
- It is a **tree** because the graph has become acyclic, **spanning** because it covers all vertices and **minimum** by the defined goal

Minimum Spanning Tree

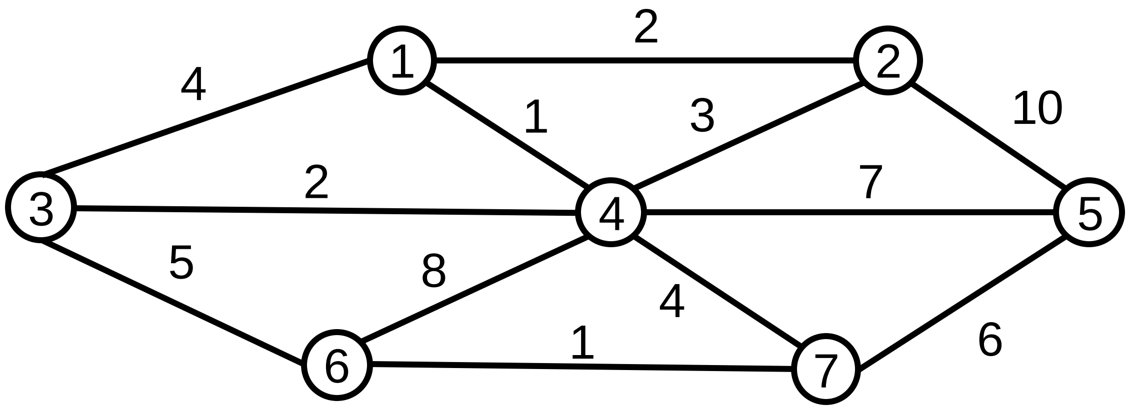
- Programmatically we can leverage the table like approach we saw with *Dijkstra's algorithm* for shortest path
- We will keep track of d_v and p_v for each vertex, will also keep track if the vertex is known or not
- d_v is the weigh of the shortest path connecting v to a known vertex
- p_v is the last vertex that caused a change in d_v
- Algorithm proceeds as it did in the case of the shortest path with an exception of the update (its simpler); a vertex v is selected for each unknown w adjacent to v such that the $d_w = \min(dw, c_{w,v})$

Minimum Spanning Tree

- In *Dijkstra's algorithm* d_v represented the tentative distance, the shortest distance from s (starting point) to v using only known vertices as intermediates
- Here we are only looking at the single edge (not path)

V	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

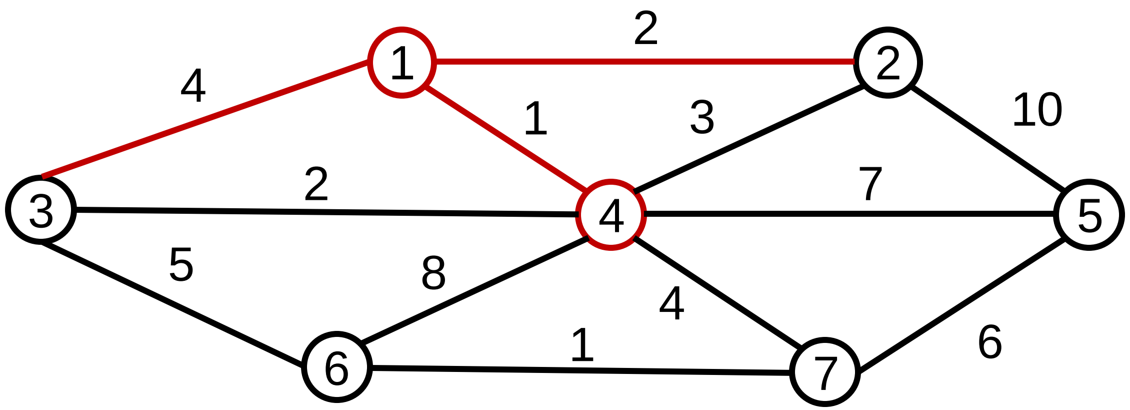
v_1 is the starting point



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

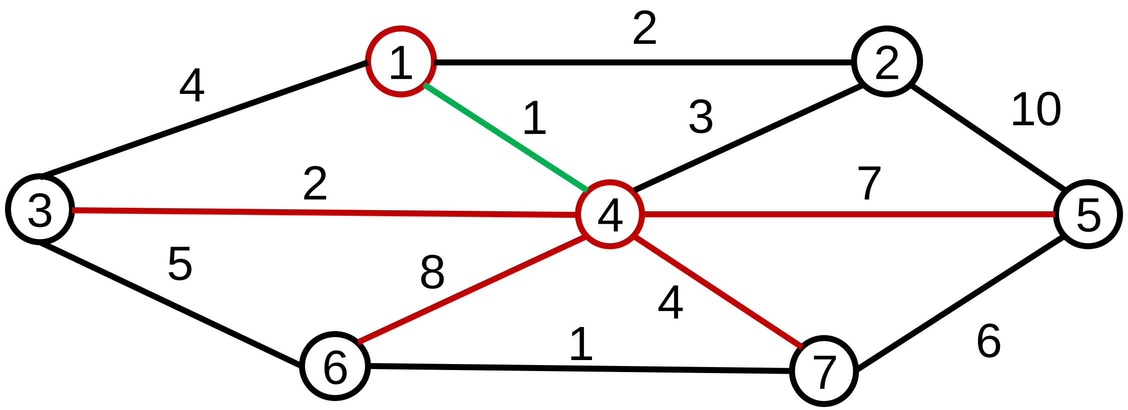
update v_2, v_3, $v_{4,}$ select v_4, lowest cost



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

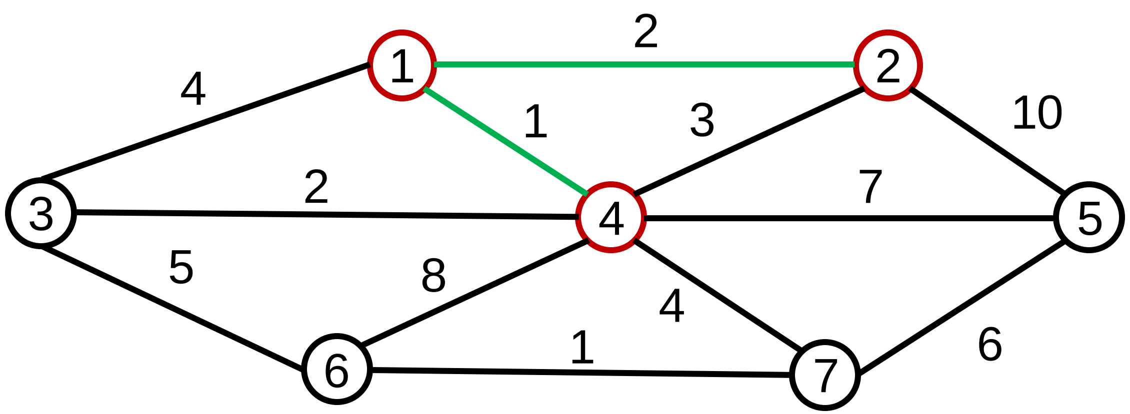
given v_4, lowest cost can be updated to v_3 and fill out the rest
we select v_2 next, could have selected v_2 or v_3 given same cost



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

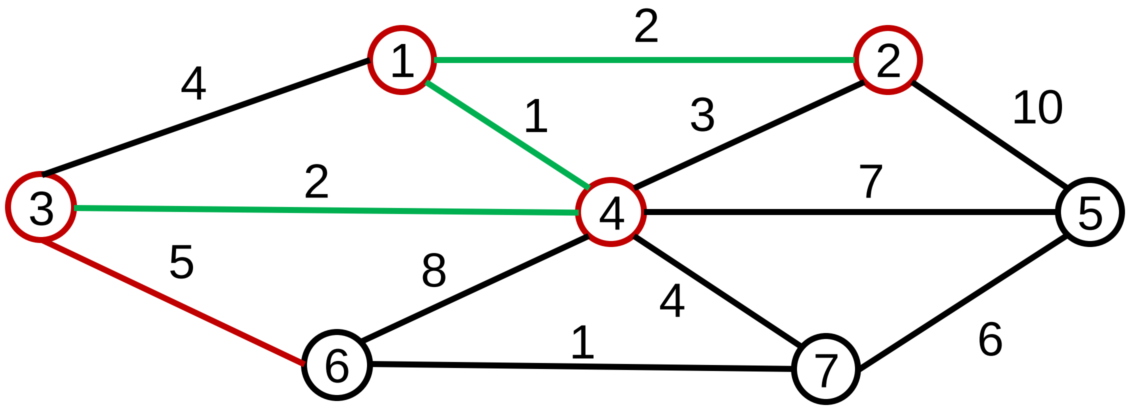
v_2 the only unknown v_2 can reach is v_5 with a cost of 10
now we select v_3



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

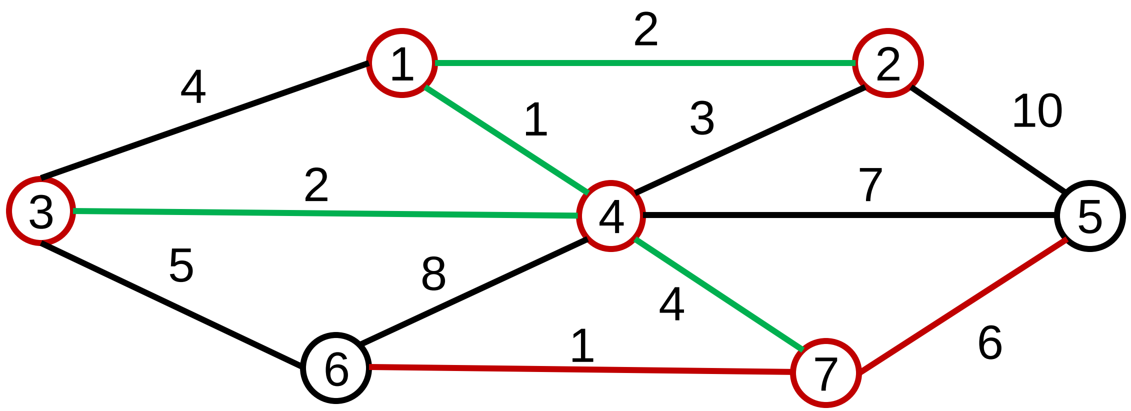
v_3 the only unknown v_3 can reach is v_6 with a better cost, we select v_7 next as lowest cost



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

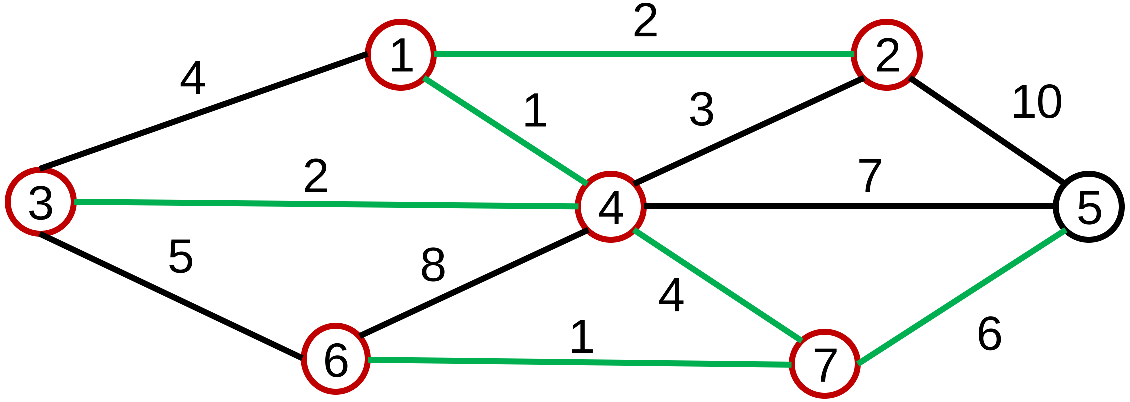
v_7 which can get to v_6 at lower cost and v_5 at lower cost
choose v_6 as next unknown to visit



Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

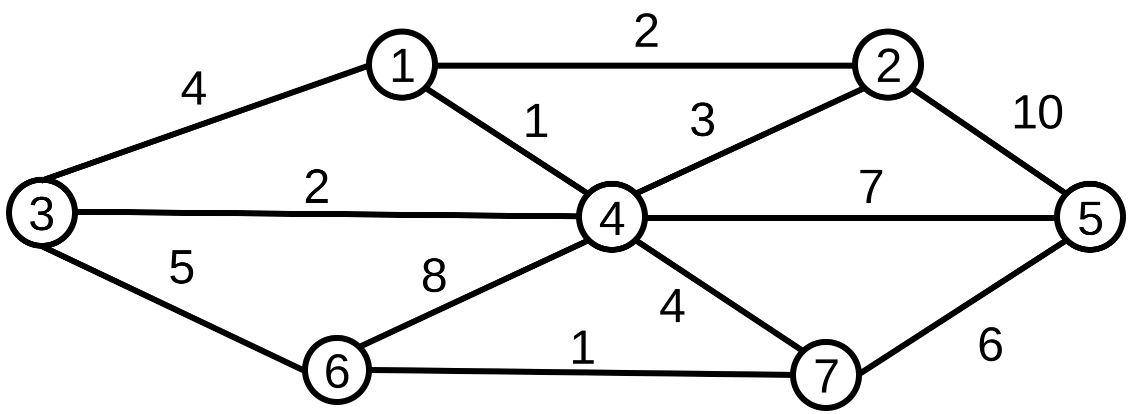
v_6 can not reach any unknowns, v_5 left



Minimum Spanning Tree

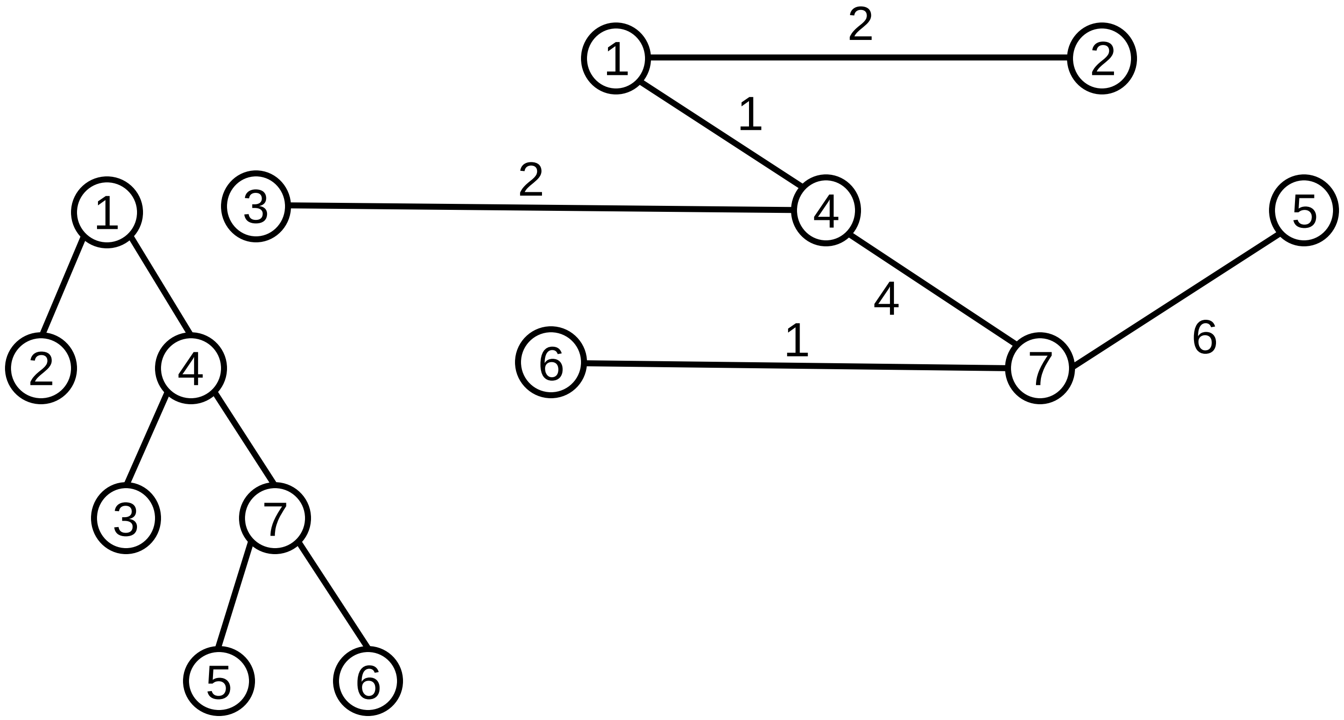
V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

v_5 can get no where new either



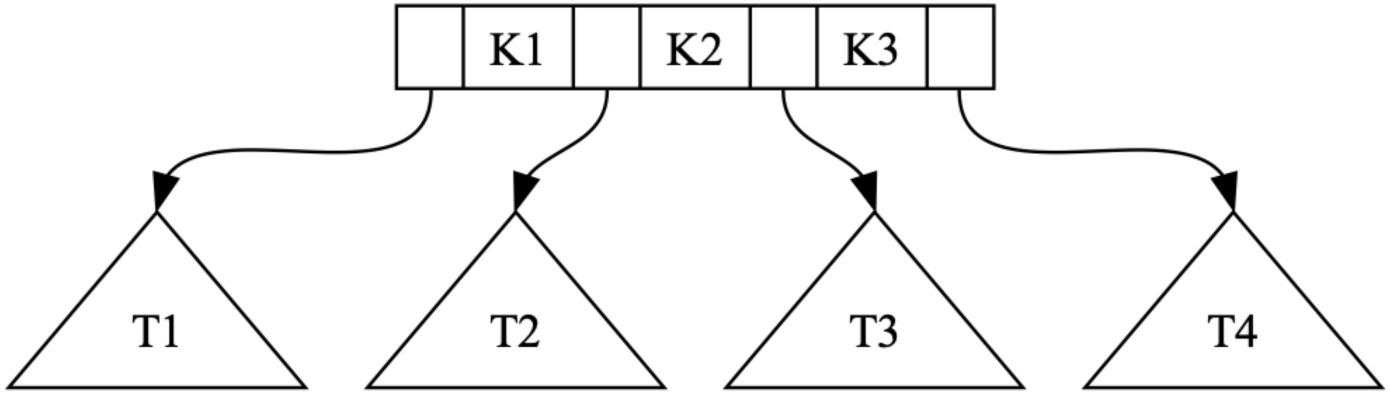
Minimum Spanning Tree

V	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4



m-ary Tree

- m-ary search tree allows m-way branching (**m children**)
- A node in a m-ary tree stores **m-1 keys** (K1, K2, K3, …) **in order**
- Each piece of data stored is called a **key** (unique, only in one location)
- The keys in a node serve as **dividing points**
- Each node also has **m pointers**

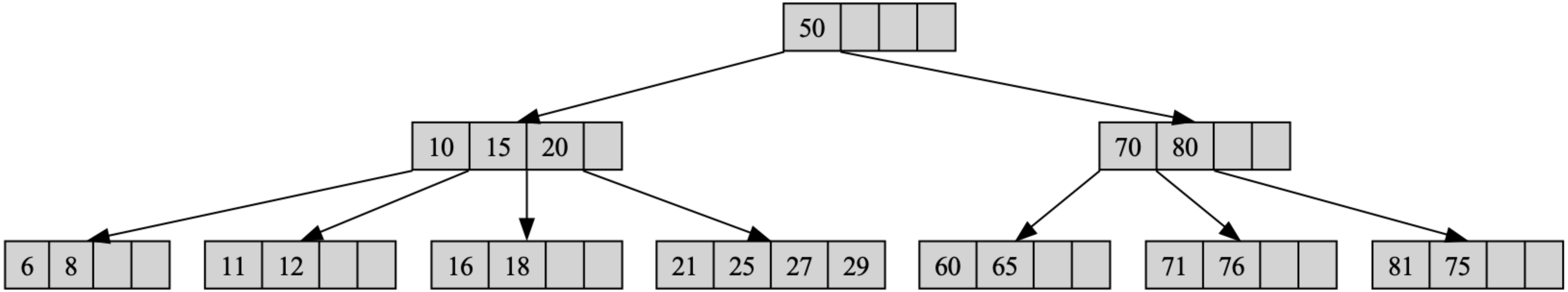


B-tree of Order m Properties

- Developed by Bayer and McCreight in 1972
- Properties of a B-tree:
  1. The root has at least two subtrees unless it is a leaf.
  2. Each nonroot and each nonleaf node holds $k - 1$ keys and k pointers to subtrees where $\left\lceil \frac{m}{2} \right\rceil \leq k \leq m$.
  3. Each leaf node holds $k - 1$ keys where $\left\lceil \frac{m}{2} \right\rceil \leq k \leq m$.
  4. All leaves are on the same level.
- According to these conditions, a B-tree is always at least half full, has a few levels, and is perfectly balanced.

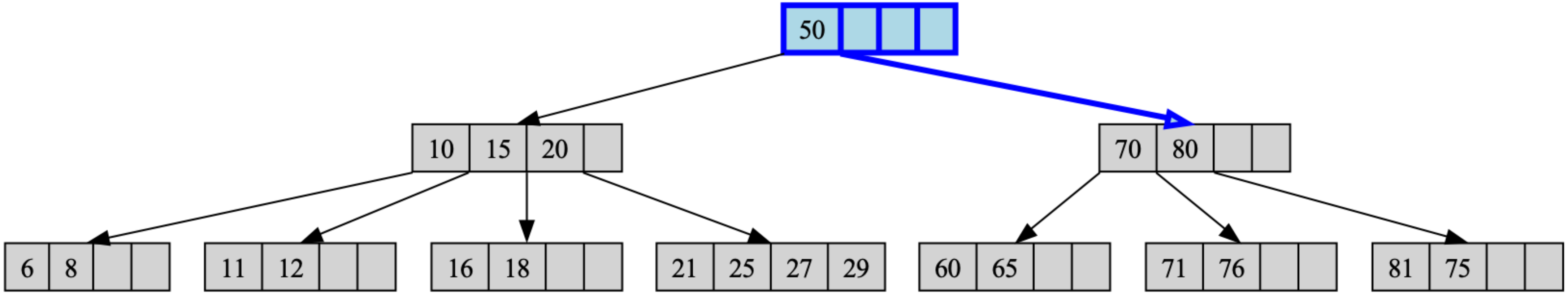
Searching B-tree

- Search for 71
- Almost the same process as binary tree



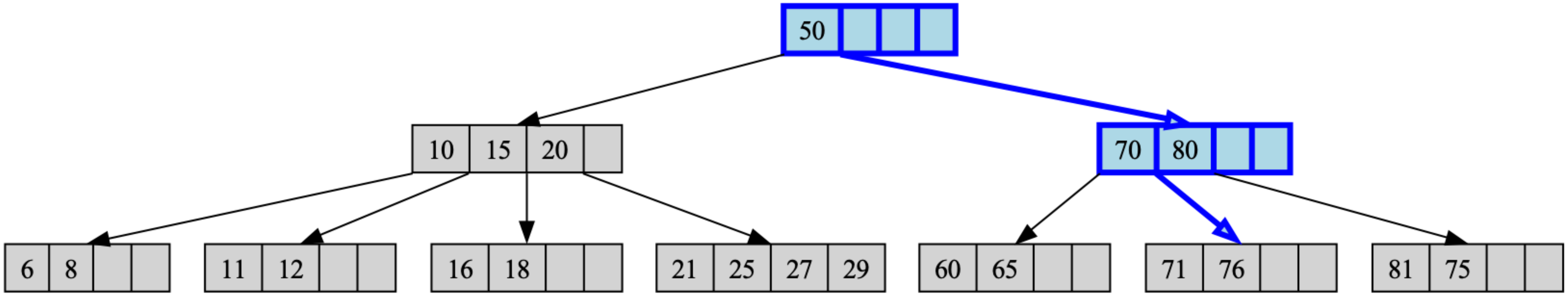
Searching B-tree

- Search for 71
- Almost the same process as binary tree



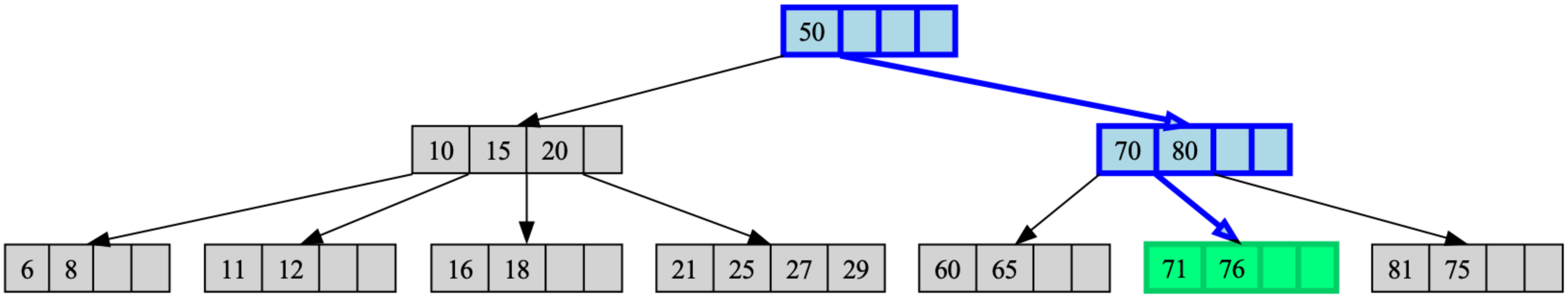
Searching B-tree

- Search for 71
- Almost the same process as binary tree



Searching B-tree

- Search for 71
- Almost the same process as binary tree

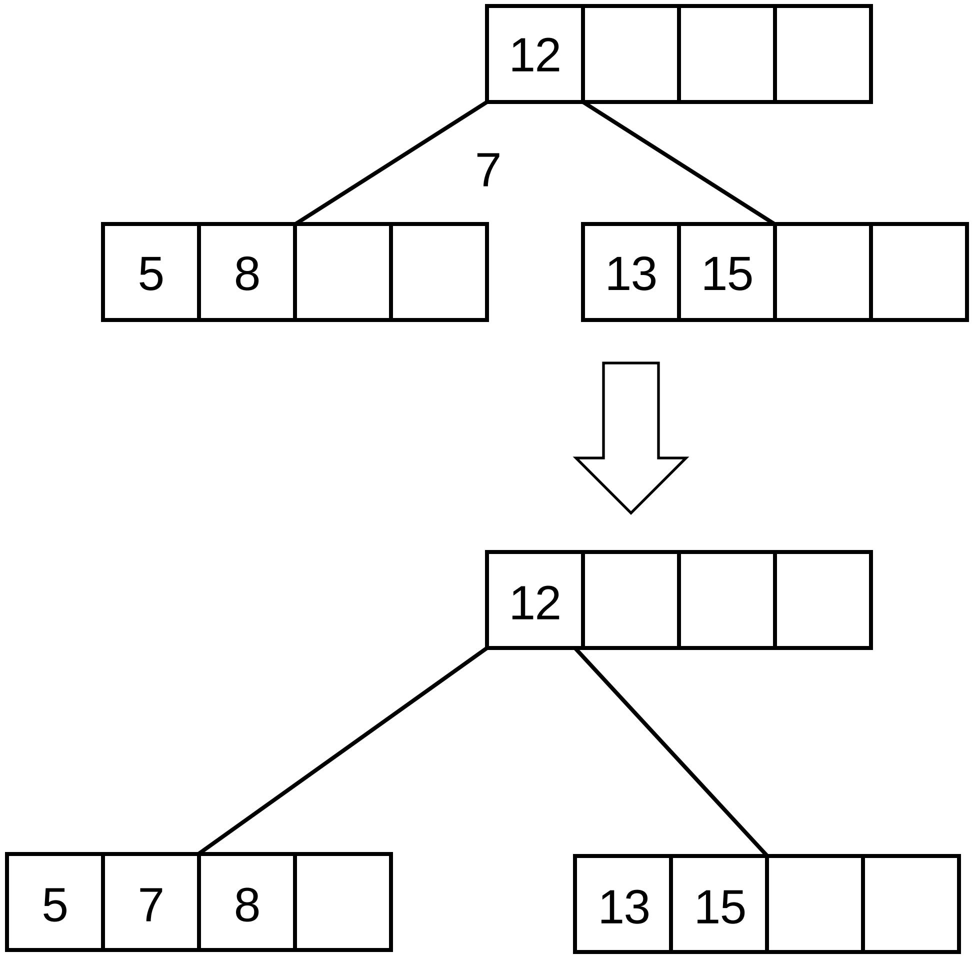


Insertion into a B-tree

- Incoming keys are added directly to a leaf if there is space available.
- When a leaf is full, the keys are divided between the leaves and one key is promoted to the parent.
- If the parent is full the process is repeated until the root is reached and a new root created.

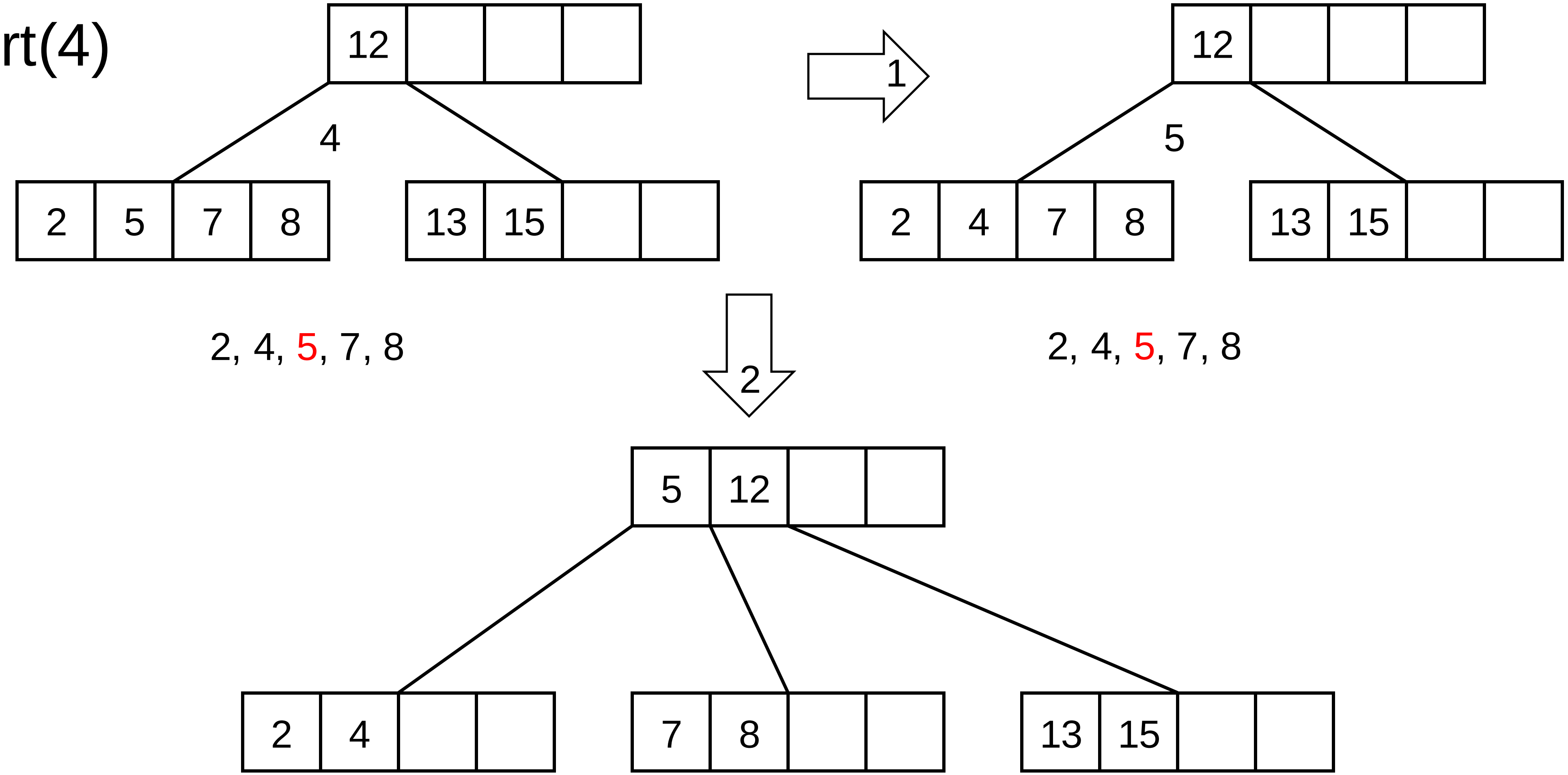
Insertion Example B-tree Order 5

- Insert(7)

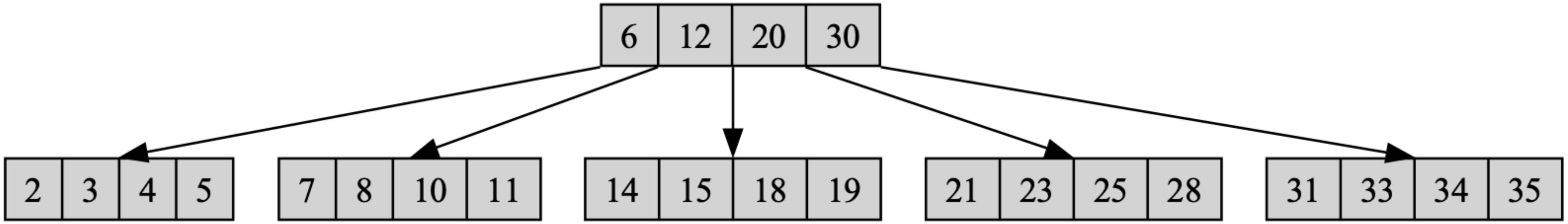


Insertion Example B-tree Order 5

- Insert(4)

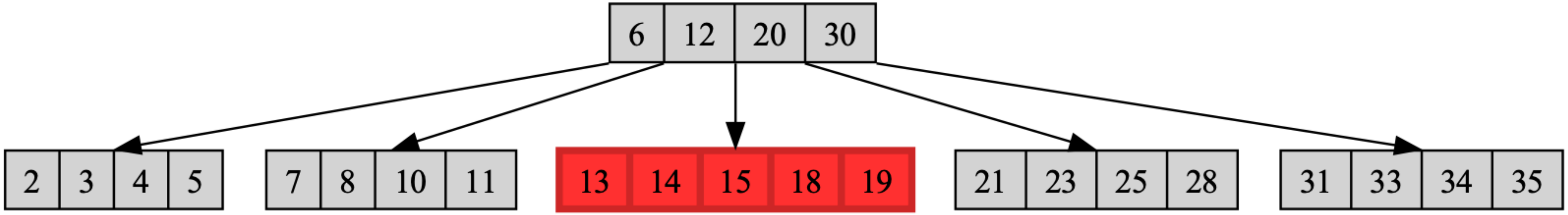


Insertion Example B-tree Order 5



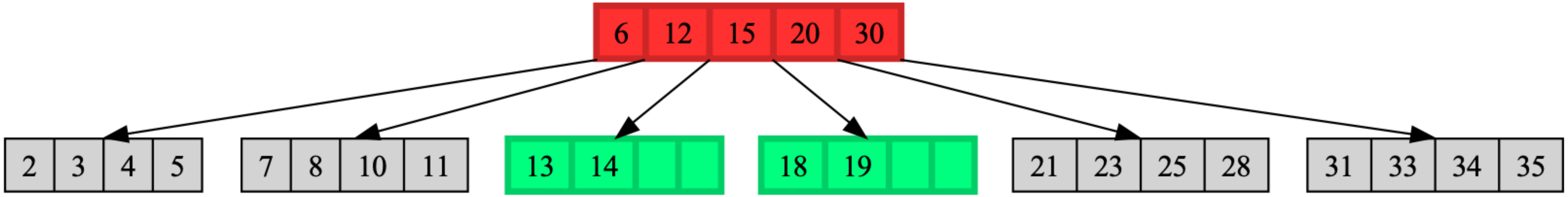
Insertion Example B-tree Order 5

- Insert 13



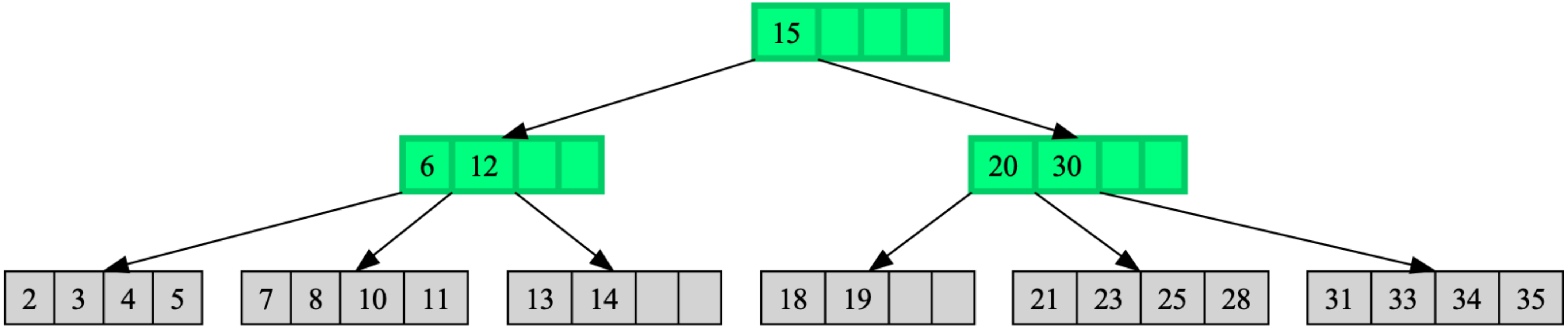
Insertion Example B-tree Order 5

- Insert 13



Insertion Example B-tree Order 5

- Insert 13

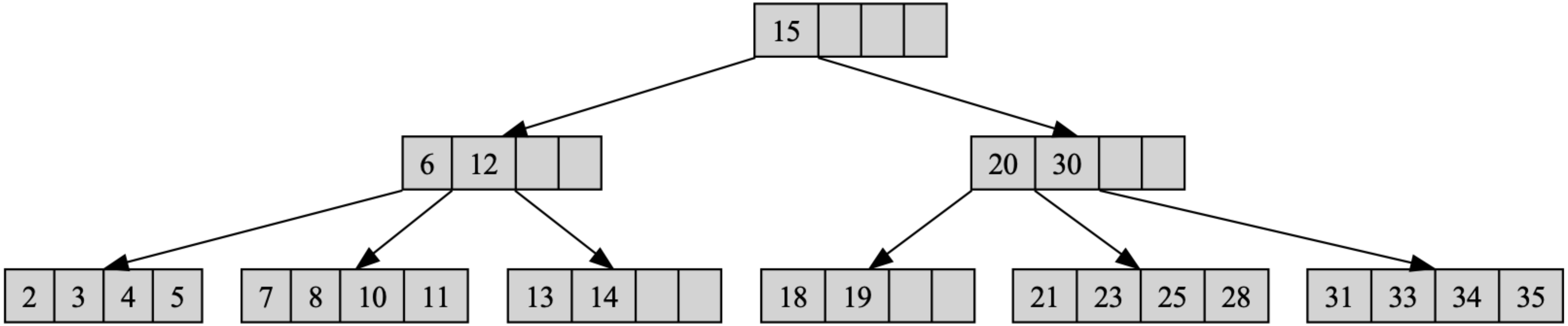


B-tree Deletion

- Deletion is basically the **reverse** of insertion.
- Nodes can not become **less than half full** after a deletion
- If less than half full, nodes need to be **merged**.
- Two cases to look at:
  - Deleting from a leaf: If merging, include the splitting key as it may resplit
  - Deleting from a non-leaf:
    - If either child has more than minimum keys, promote the predecessor/ successor
    - If neither child has more, merge the nodes

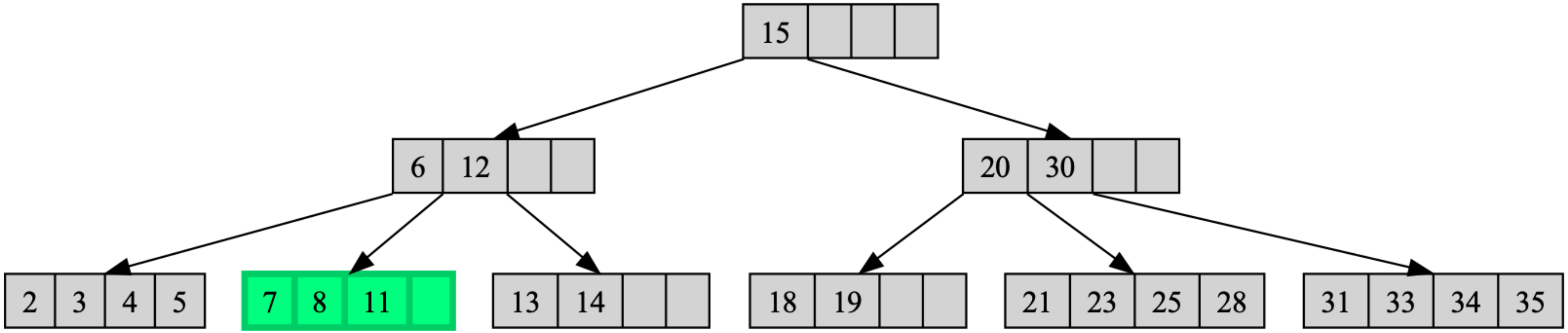
B-tree Delete Example

- Delete 10



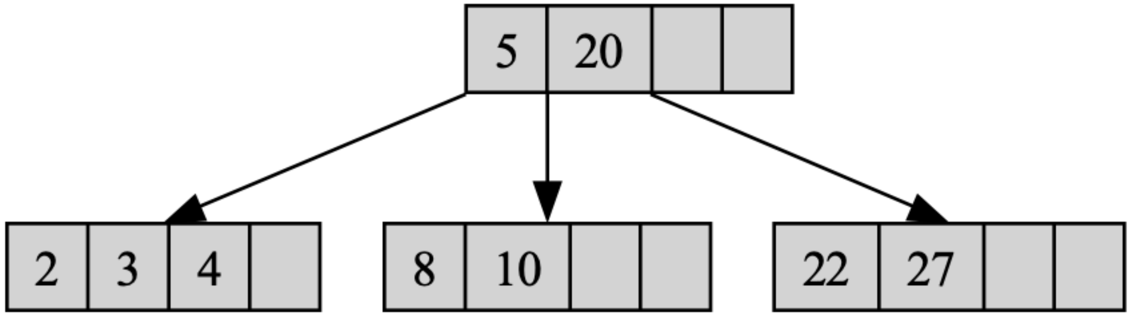
B-tree Delete Example

- Delete 10



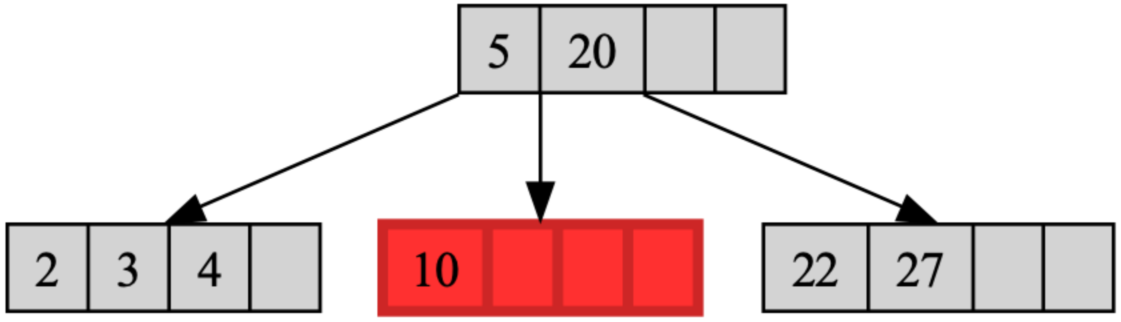
B-tree Delete Example

- Delete 8



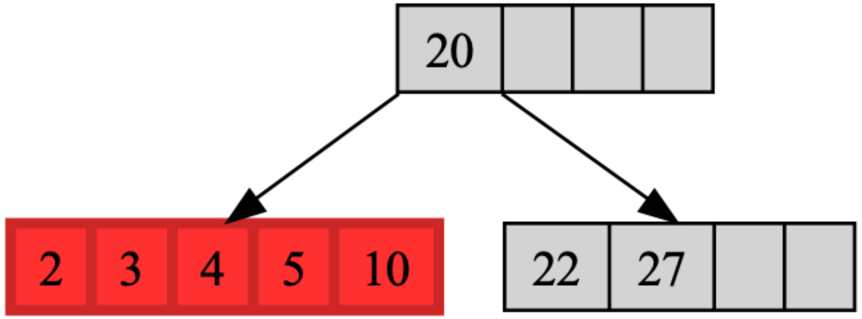
B-tree Delete Example

- Delete 8



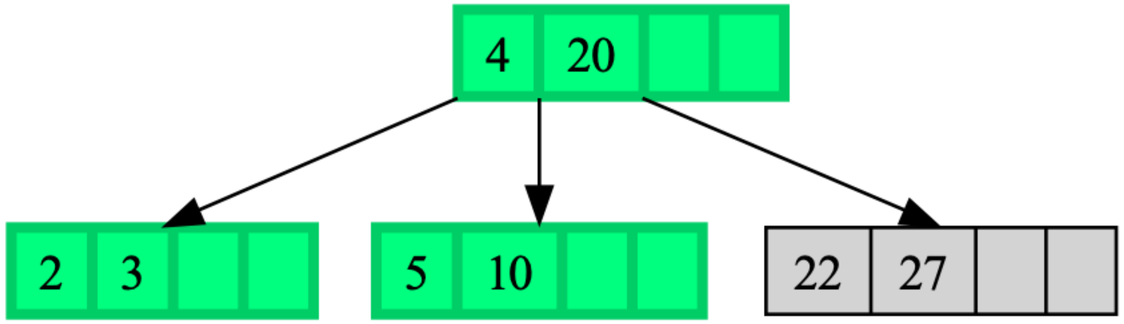
B-tree Delete Example

- Delete 8



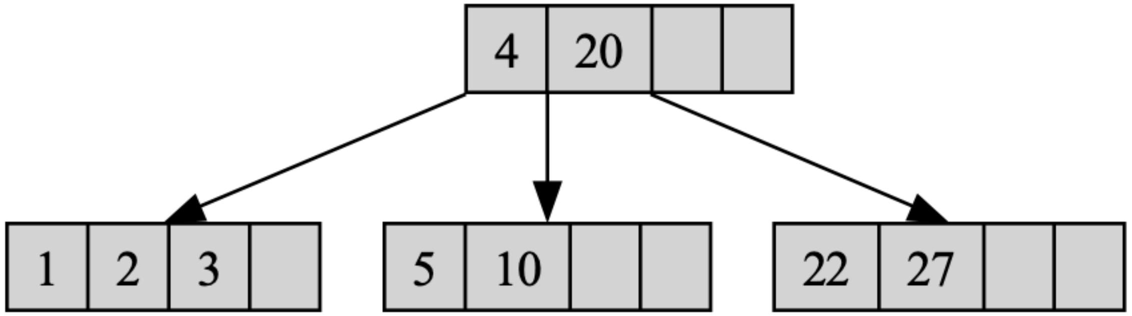
B-tree Delete Example

- Delete 8



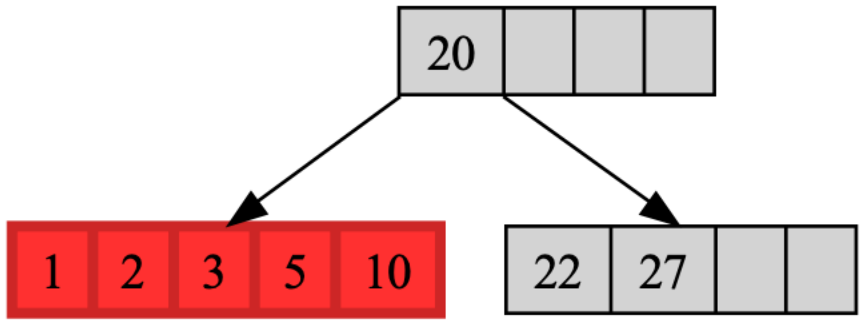
B-tree Delete Example

- Delete 4



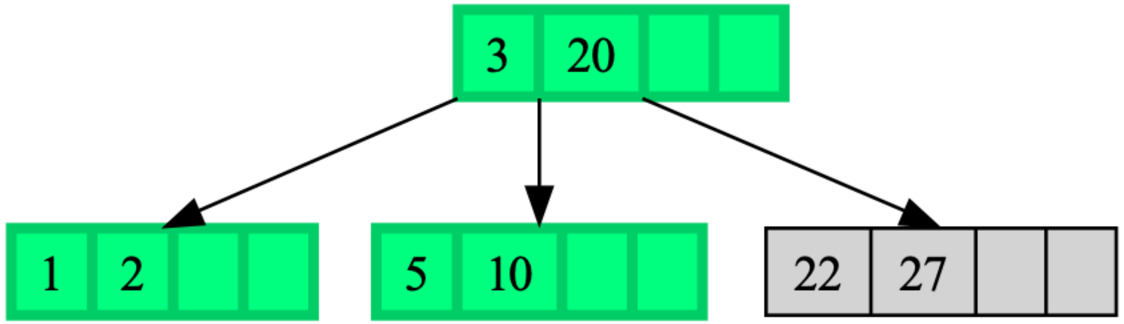
B-tree Delete Example

- Delete 4



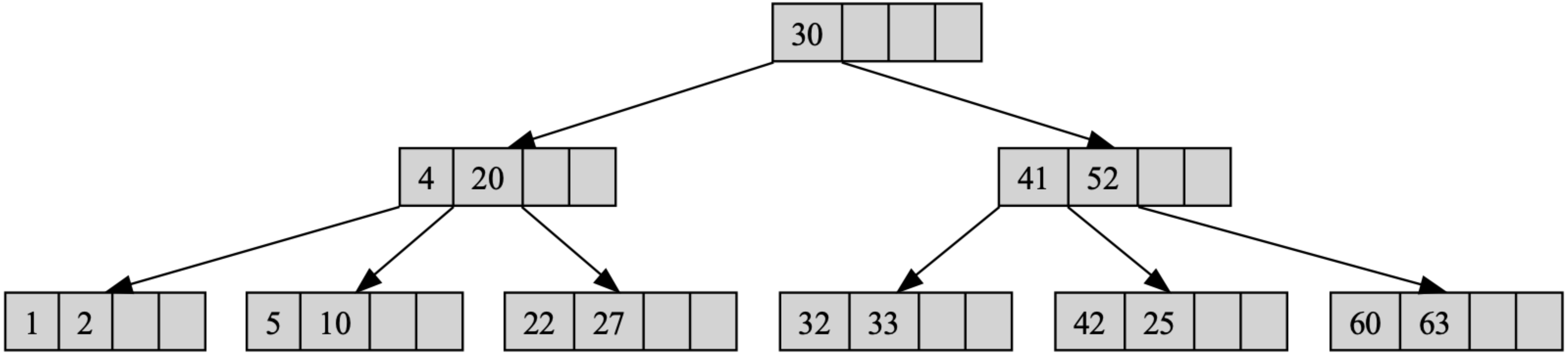
B-tree Delete Example

- Delete 4



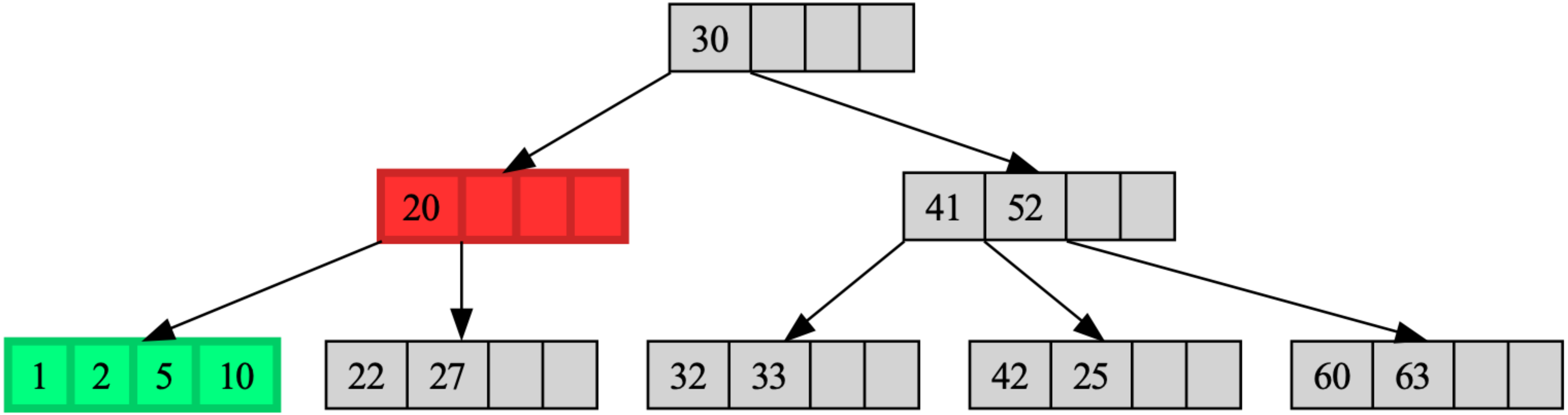
B-tree Delete Example

- Delete 4



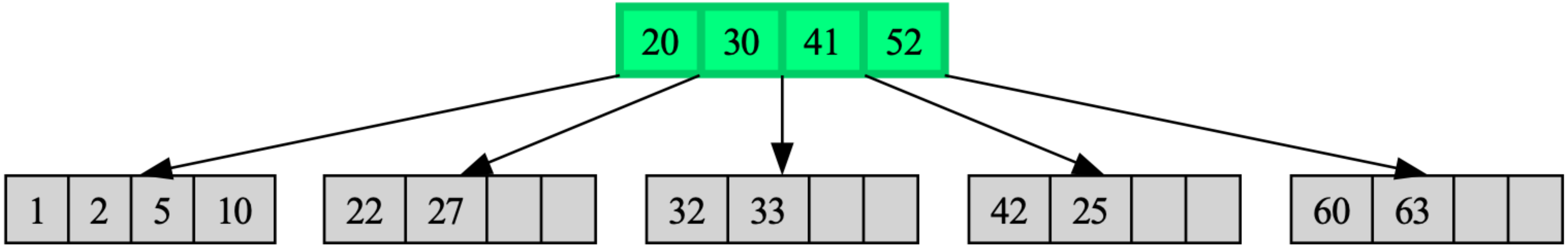
B-tree Delete Example

- Delete 4



B-tree Delete Example

- Delete 4



Hash Table

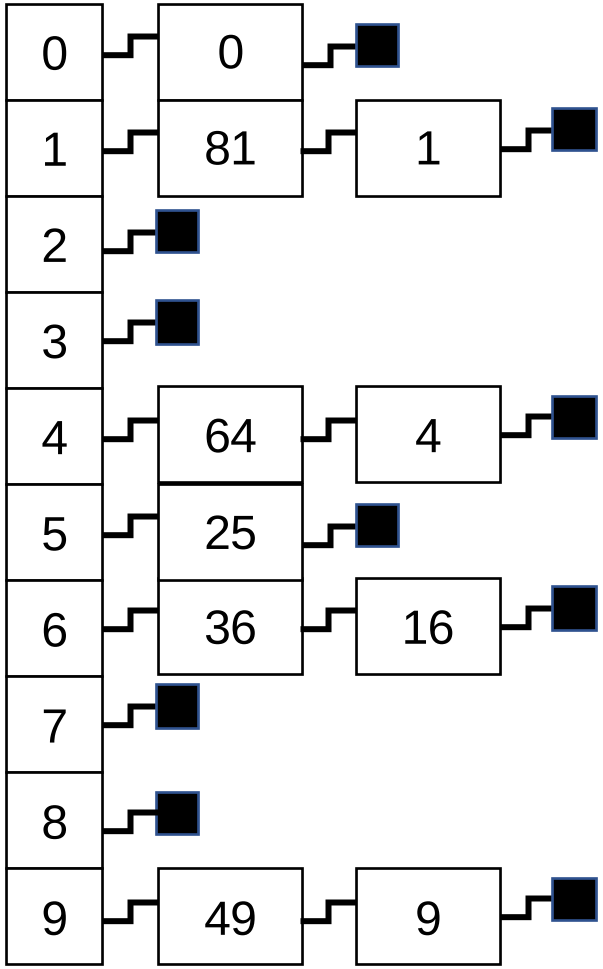
- The ideal hash table data structure is an array of some fixed size containing items
- A **key** is used for the lookup
- The size of the hash table is known as the *tablesize* and common convention has the table run from 0 to (*tablesize* - 1)
- Each key is mapped into some number in the range 0 to (*tablesize* - 1) which corresponds to a cell in the array
- The mapping is called a hash function and has the properties:
  - it is easy to compute
  - ensures that any two keys result in a different cell location

Collisions

- If/when an element is inserted and it hashes to an index that is already in use, it is known as a collision.

Collisions (Separate Chaining)

- Keeps a list of all the elements that hash to the same value
- Example:
  - Let's assume the keys are the first 10 perfect squares
  - Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  - Hashing Function: hash(x) = x%10, where 10 is tablesize
  - Then index looks like 0, 1, 4, 9, 6, 5, 6, 9, 4, 1



Collisions (Probing)

- In **probing**, you look for alternative cells until an empty cell is found
- Given cells:

$h_0(x), h_1(x), h_2(x), \ldots$ are tried in succession where:

$$h_i(x) = (hash(x) + f(i))\%tablesize \text{ with } f(0) = 0$$

The function f is the *collision resolution strategy*

- When using probing all elements go into the table so the table is generally larger
- Types of probing
  - Linear
  - Quadratic
  - Double hashing

Collisions (Linear Probing)

- In linear probing, f is a linear function of i

$$f(i) = i$$

- Cells are tried sequentially (with wrap around) in search of empty cell
- Example:
  - Keys: 89, 18, 49, 58, 69
  - Hashing Function: hash(x) = x%10, where 10 is tablesize

Collisions (Quadratic Probing)

- In quadratic probing, f is a quadratic function of i

$$f(i) = i^2$$

- Cells are tried sequentially (with wrap around) in search of empty cell
- Example:
  - Keys: 89, 18, 49, 58, 69
  - Hashing Function: hash(x) = x%10, where 10 is tablesize

Collisions (Double Hashing)

- Double hashing uses a second hash function when a collision occurs.

$$f(i) = i * hash_2(x)$$

which mean, we apply a second hash function to x probe at a distance $hash_2(x)$, then $2*hash_2(x)$, then $3*hash_2(x)$, …

- NOTE: Poor choice of $hash_2$ can be disastrous!
- Example: $hash_2(x) = x\%9$

$$f(i) = i * hash_2(x)$$

if building on our last example we inserted 99, it would conflict with 89 which when executed the second has would be $i * 0$ for all i's.

Take Away: $hash_2$ can **NEVER** return 0!!!!!

Rehashing

- What do you do if the table gets full?
- Build a new table that is twice as big (with a new hash function).
  - Scan the original table computing new key
  - Place element at new location based on key

0	6	6
1	15	15
2		23
3	24	24
4		
5		
6	13	13

$h(x) = x\%7$

insert 23, now the table is 70% full, time to create new table twice as big

Load Factor

- **Load factor** is the ratio of the number of elements in the hash table to the table size
- Ideal case is if **load factor** is less than 1
- As **load factor** approaches or is greater than 1 you might want to rehash