Data Structures and Algorithms
Graph

2022-09-16

Ping-Han Hsieh

Overview

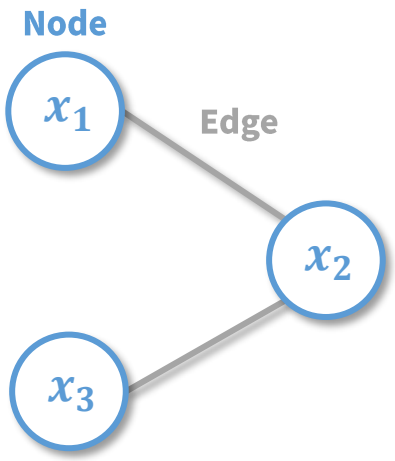
- Data Structures:
  - Linked-List, Array
  - Stack, Queue
  - Union Find, Hash Table
  - Binary Search Tree, Heap
  - Graph

- Algorithms
  - Big-O Notation
  - Sorting
  - Graph Algorithms
  - Dynamic Programming

Graph

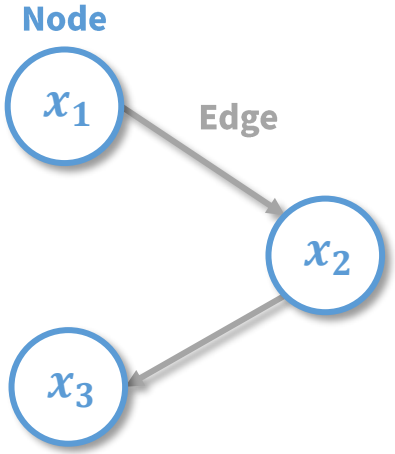
Graph Data Structure (1)

G = (N, E)

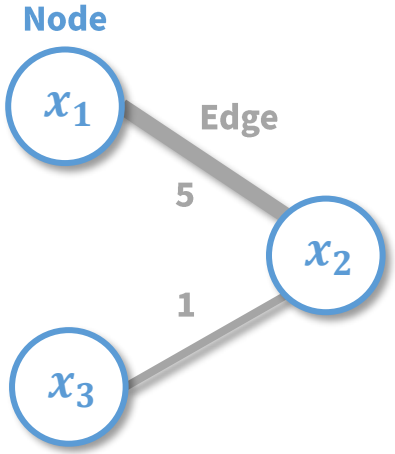


both node and edge can
have their own properties

Directed Graph

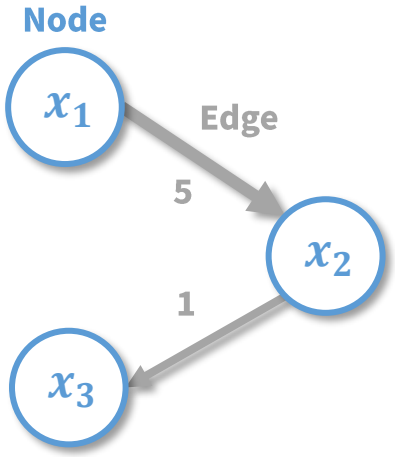


Weighted Graph



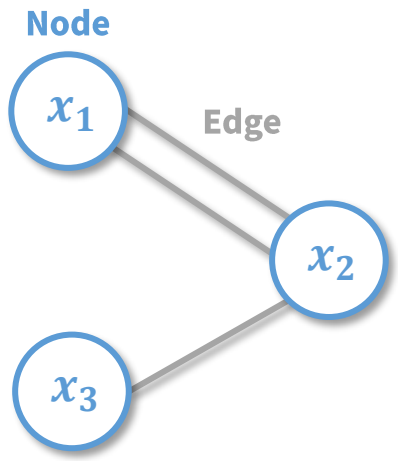
based on properties of edges

Directed Weighted Graph



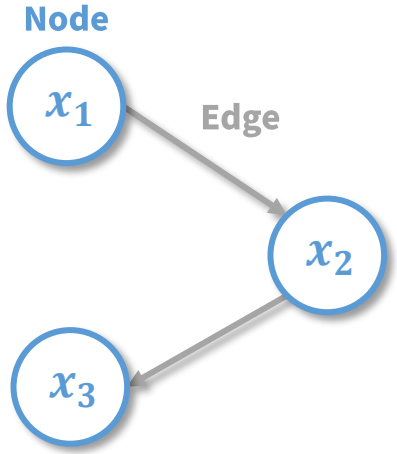
Graph Data Structure (2)

Multi-edges Graph

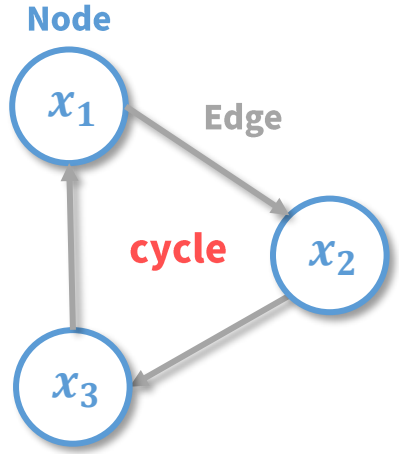


based on properties of edges

Directed Acyclic Graph (Tree)



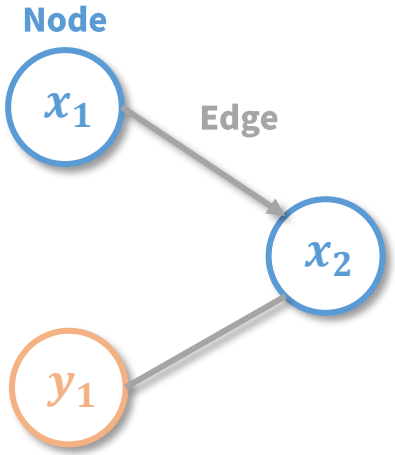
Directed Cyclic Graph



start and end with the same node

based on topology

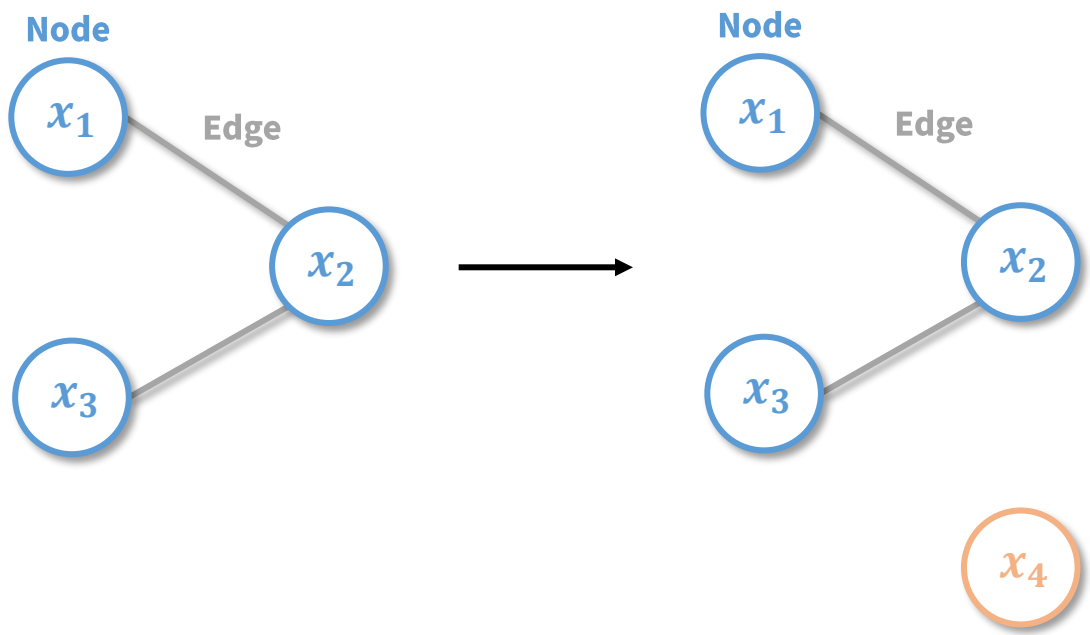
Heterogeneous Graph



based on properties of edges and nodes

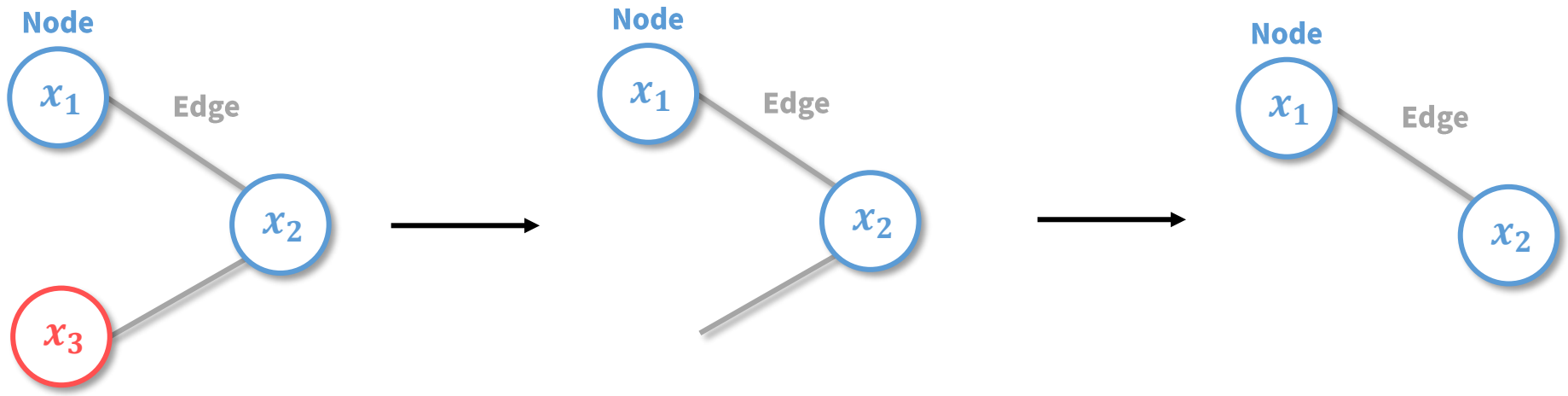
Graph Abstract Data Types (1)

Insert Node



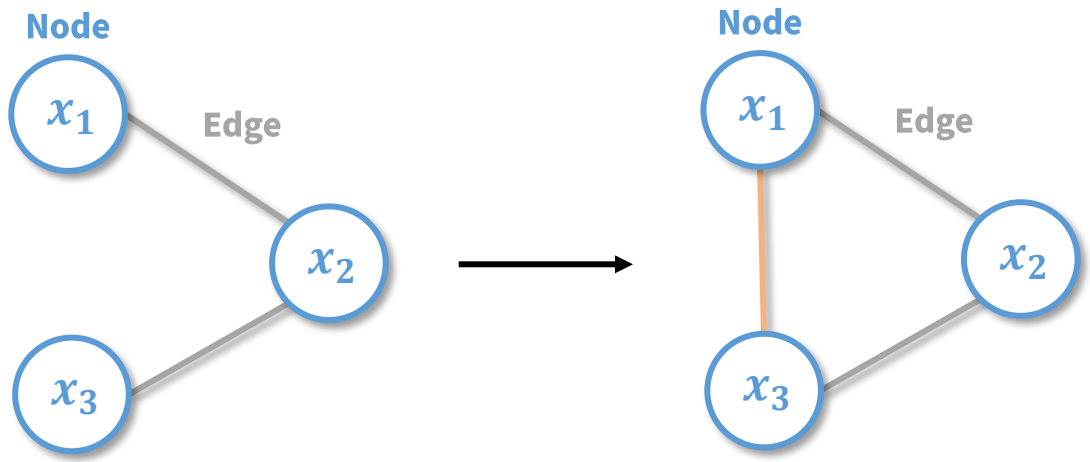
Graph Abstract Data Types (2)

Delete Node



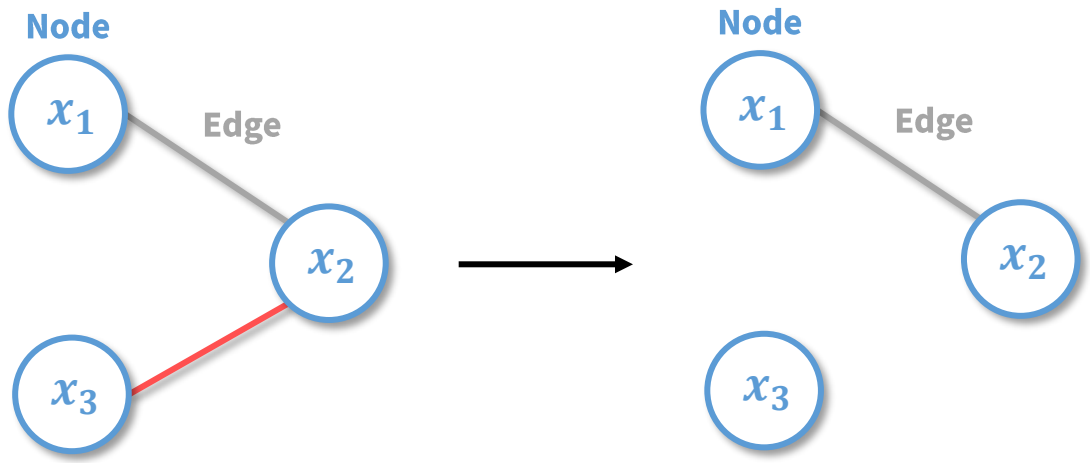
Graph Abstract Data Types (3)

Insert Edge



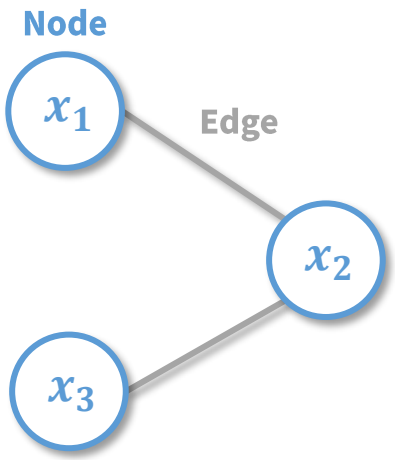
Graph Abstract Data Types (4)

Delete Edge



Graph Abstract Data Types (5)

Adjacent

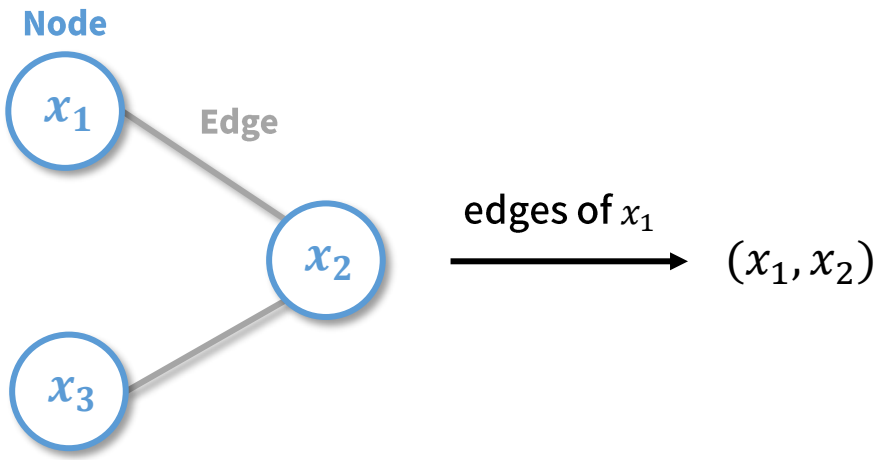


are x_1 and x_3 adjacent to each other

False

Graph Abstract Data Types (6)

Edges

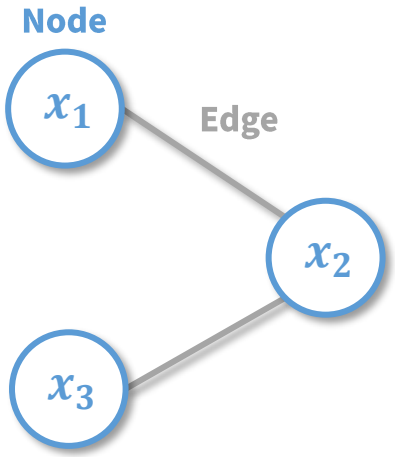


edges of x_1 → (x_1, x_2)

Graph Implementation (1)

Node and Edge List

Graph Representation



Implementation

Node List

x_1
x_2
x_3

Edge List

(x_1, x_2)
(x_2, x_3)

both can be implemented using either array or hash table

Memory Complexity

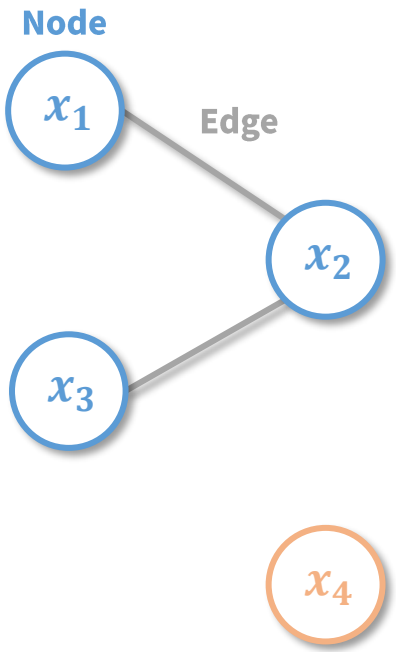
$O(n + m)$

$n = |N|, m = |E|$

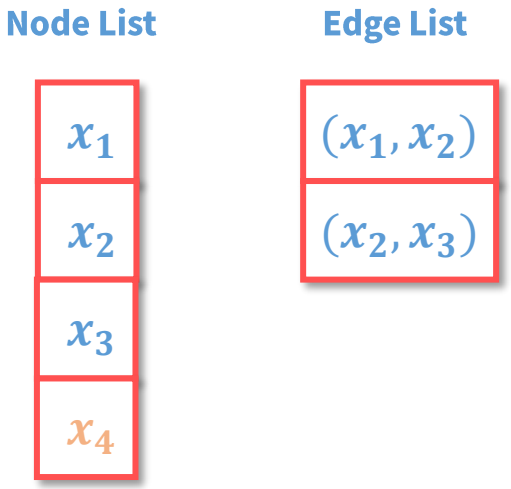
Node and Edge List (1)

Insert Node

Graph Representation



Implementation



Time Complexity (Array)

worst case $O(n)$

amortized $O(1)$

$n = |N|$

Time Complexity (Hash Table)

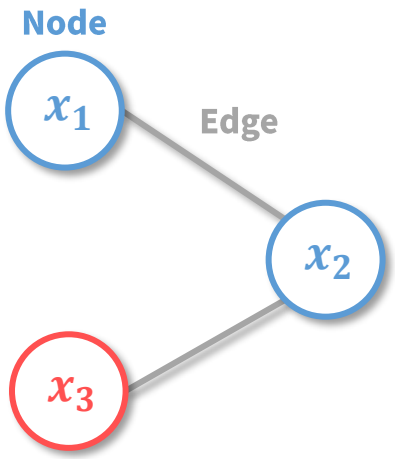
worst case $O(1)$

$n = |N|$

Node and Edge List (2)

Delete Node (1)

Graph Representation



Implementation

Node List

x_1
x_2
x_3

Edge List

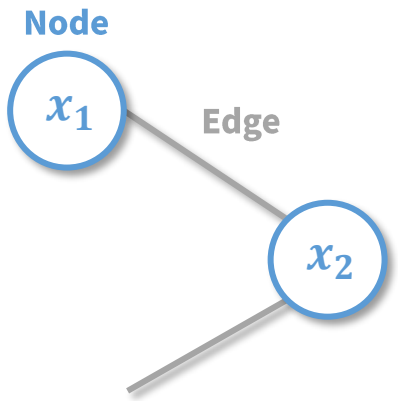
(x_1, x_2)
(x_2, x_3)

Time Complexity (Array)

Node and Edge List (3)

Delete Node (2)

Graph Representation



Implementation

Node List

x_1
x_2

Edge List

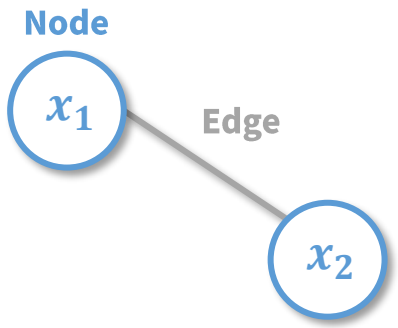
(x_1, x_2)
(x_2, x_3)

Time Complexity (Array)

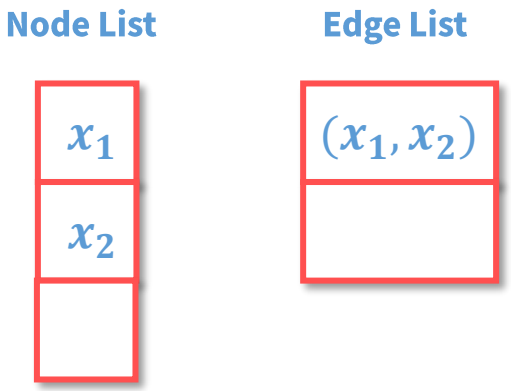
Node and Edge List (4)

Delete Node (3)

Graph Representation



Implementation



Time Complexity (Array)

worst case $O(n + m)$

$n = |N|, m = |E|$

Time Complexity (Hash Table)

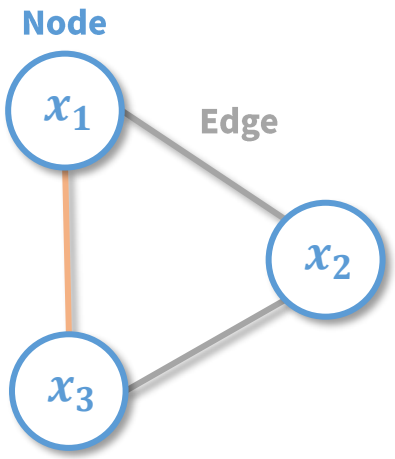
worst case $O(n)$ *or* $O(m)$

$n = |N|, m = |E|$

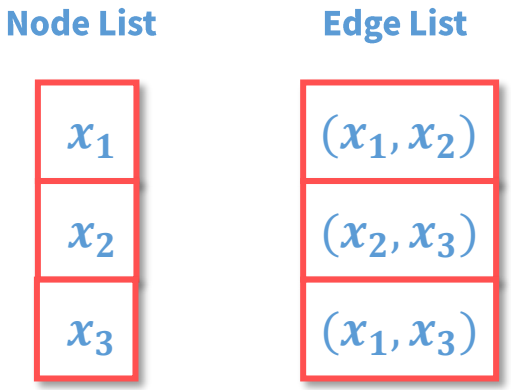
Node and Edge List (5)

Insert Edge

Graph Representation



Implementation



Time Complexity (Array)

worst case $O(m)$

amortized $O(1)$

$m = |E|$

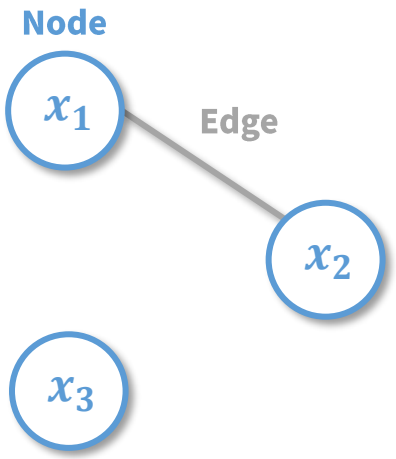
Time Complexity (Hash Table)

worst case $O(1)$

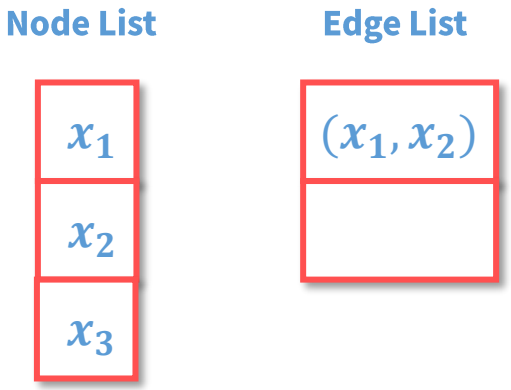
Node and Edge List (6)

Delete Edge

Graph Representation



Implementation



Time Complexity (Array)

worst case $O(m)$

$m = |E|$

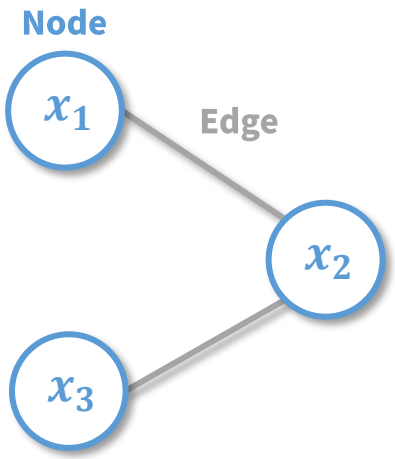
Time Complexity (Hash Table)

worst case $O(1)$

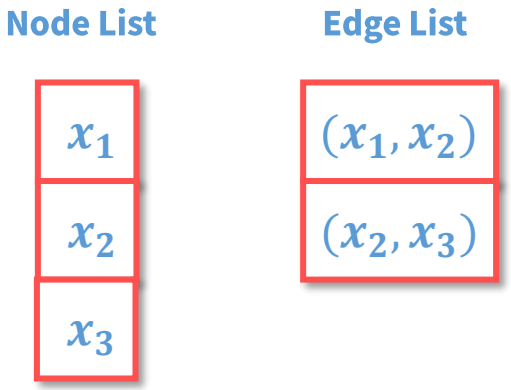
Node and Edge List (7)

Adjacent

Graph Representation



Implementation



Time Complexity (Array)

worst case $O(m)$

$m = |E|$

Time Complexity (Hash Table)

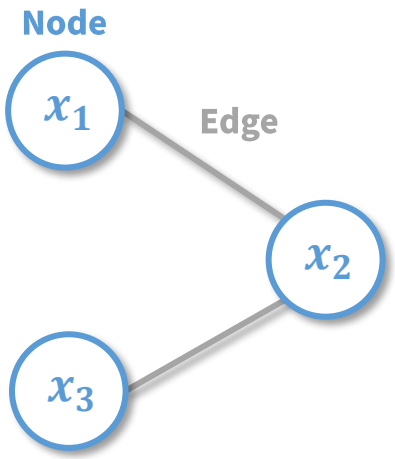
worst case $O(1)$

are x_1 and x_3 adjacent to each other

Node and Edge List (8)

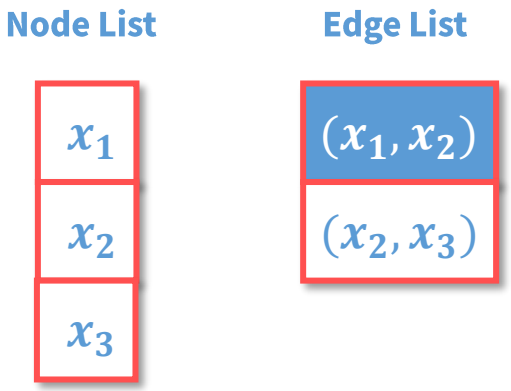
Edges

Graph Representation



edges of x_1

Implementation



Time Complexity (Array)

worst case $O(m)$

$m = |E|$

Time Complexity (Hash Table)

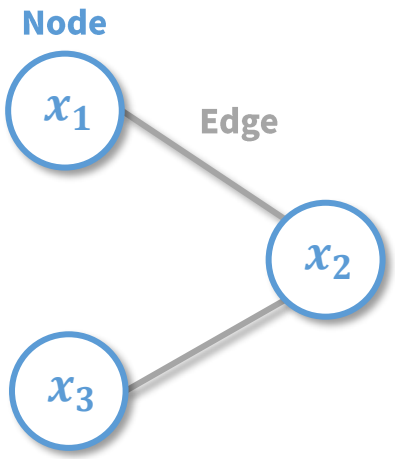
worst case $O(n)$ *or* $O(m)$

$n = |N|, m = |E|$

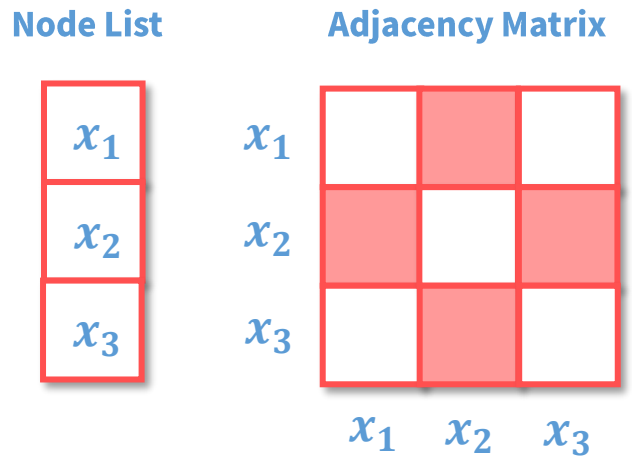
Graph Implementation (2)

Node List and Adjacency Matrix

Graph Representation



Implementation



node list and index of adjacency matrix
can be implemented using hash table

Memory Complexity

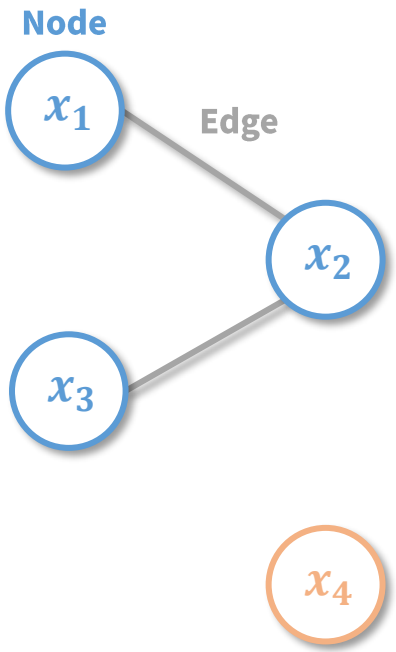
$O(n^2)$

$n = |N|$

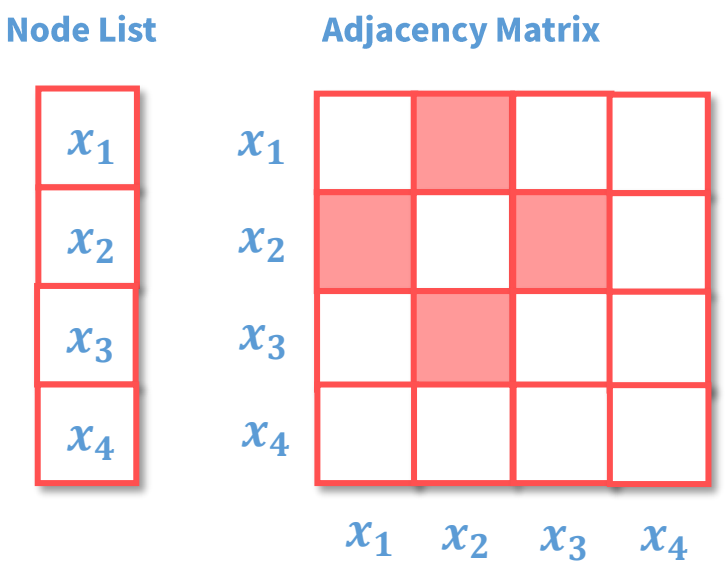
Node List and Adjacency Matrix (1)

Insert Node

Graph Representation



Implementation



Time Complexity

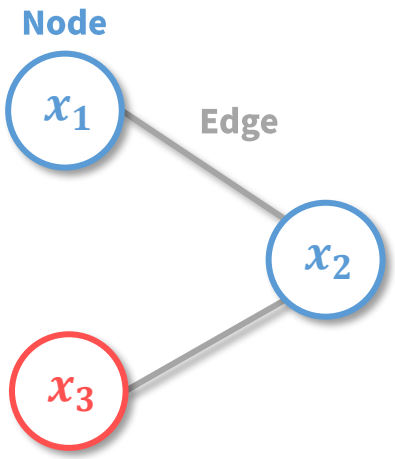
worst case $O(n)$

$n = |N|$

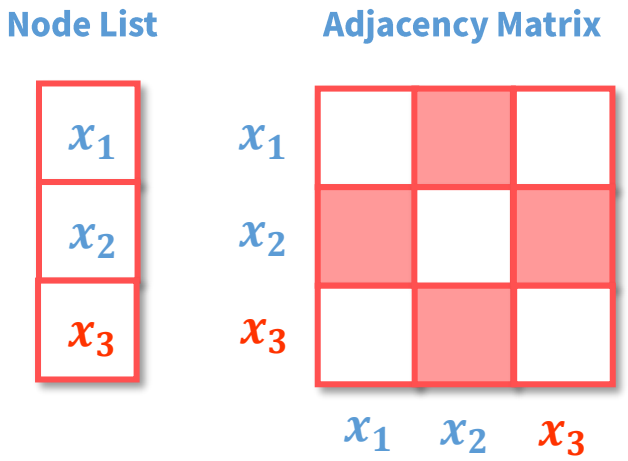
Node List and Adjacency Matrix (2)

Delete Node (1)

Graph Representation



Implementation

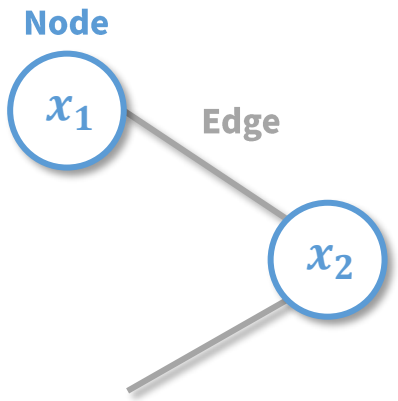


Time Complexity

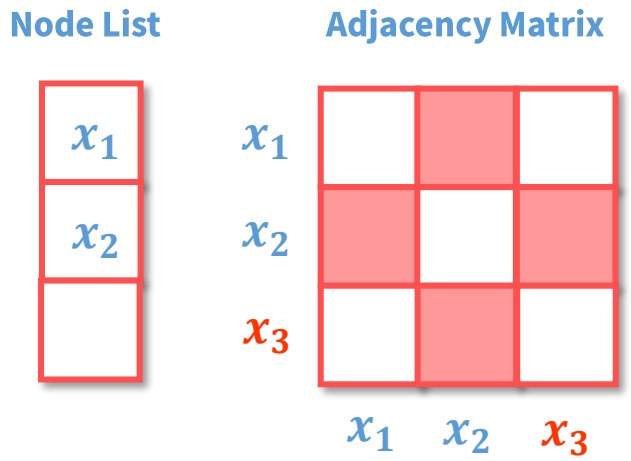
Node List and Adjacency Matrix (3)

Delete Node (2)

Graph Representation



Implementation

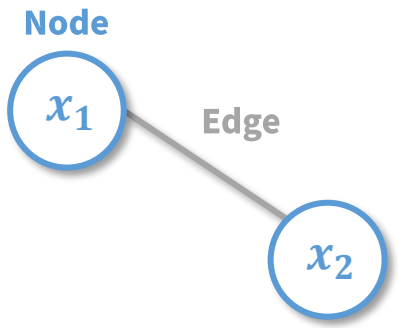


Time Complexity

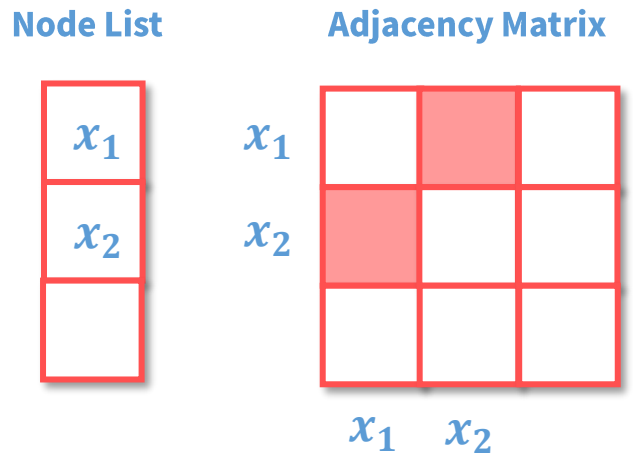
Node List and Adjacency Matrix (4)

Delete Node (3)

Graph Representation



Implementation



Time Complexity

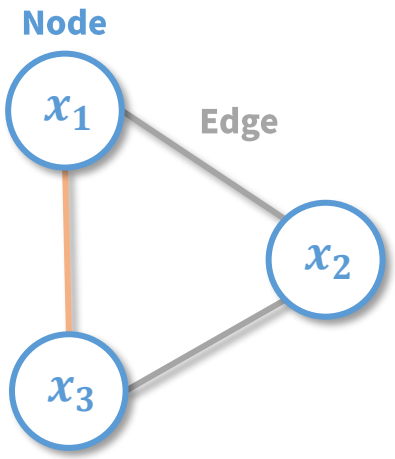
worst case $O(n)$

$n = |N|$

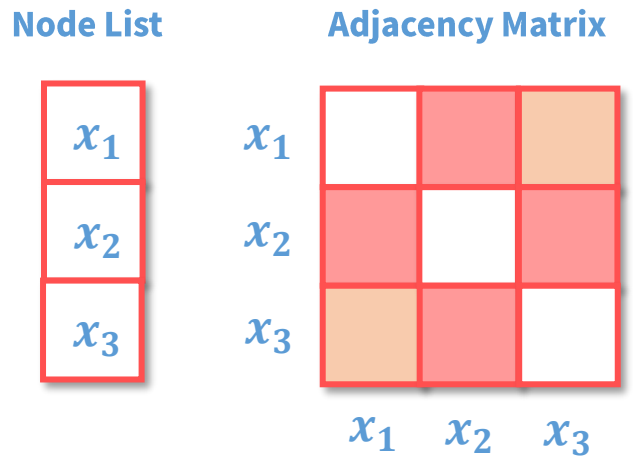
Node List and Adjacency Matrix (5)

Insert Edge

Graph Representation



Implementation



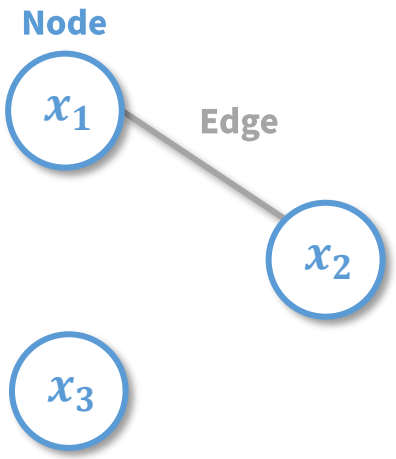
Time Complexity

worst case $O(1)$

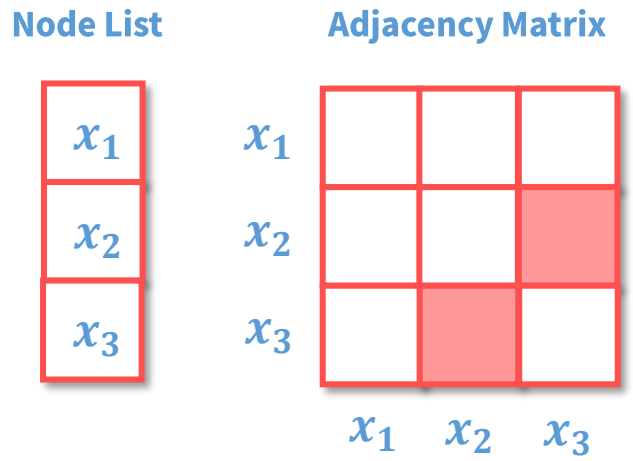
Node List and Adjacency Matrix (6)

Delete Edge

Graph Representation



Implementation



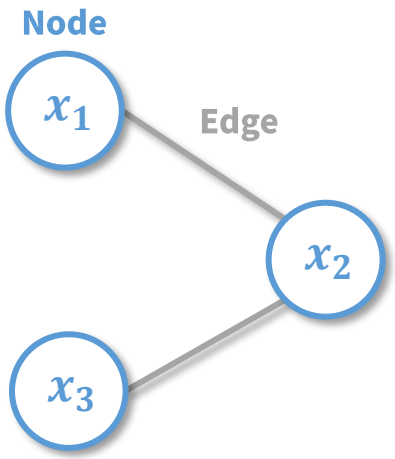
Time Complexity

worst case $O(1)$

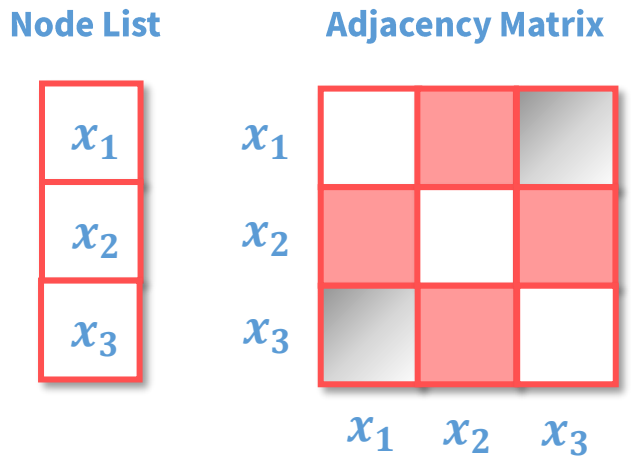
Node List and Adjacency Matrix (7)

Adjacent

Graph Representation



Implementation



Time Complexity

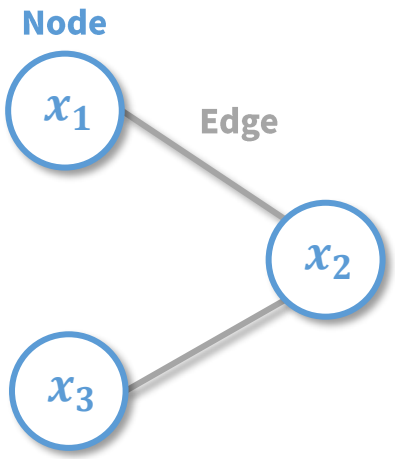
worst case $O(1)$

are x_1 and x_3 adjacent to each other

Node List and Adjacency Matrix (8)

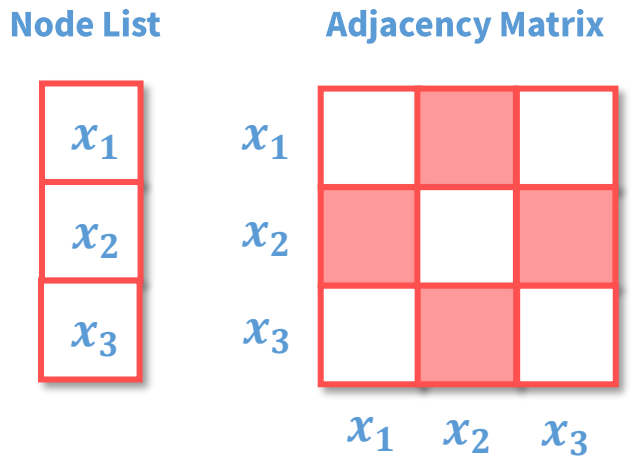
Edges

Graph Representation



edges of x_1

Implementation



Time Complexity

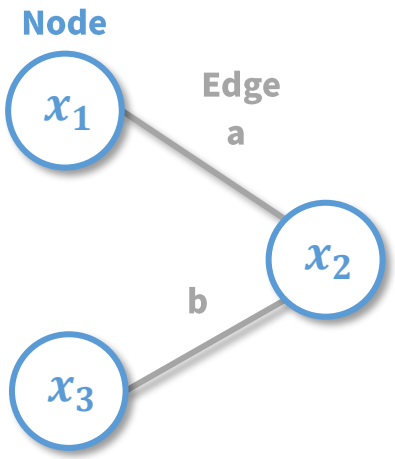
worst case $O(n)$ *or* $O(m)$

$n = |N|, m = |E|$

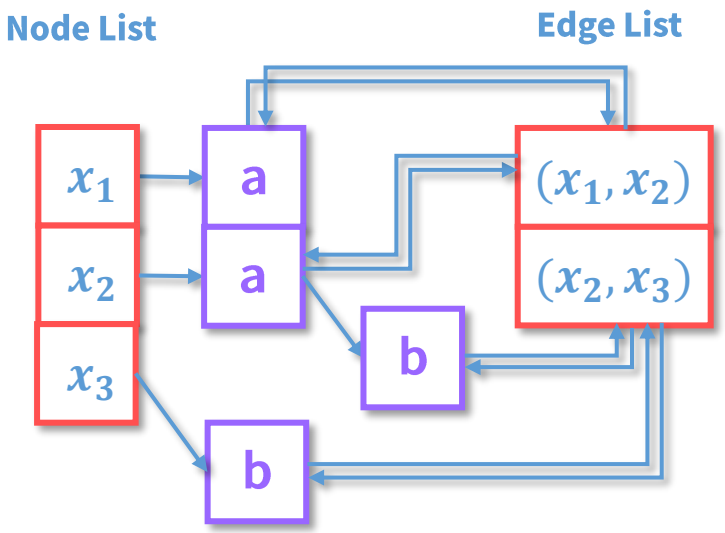
Graph Implementation (3)

Adjacency List

Graph Representation



Implementation



Memory Complexity

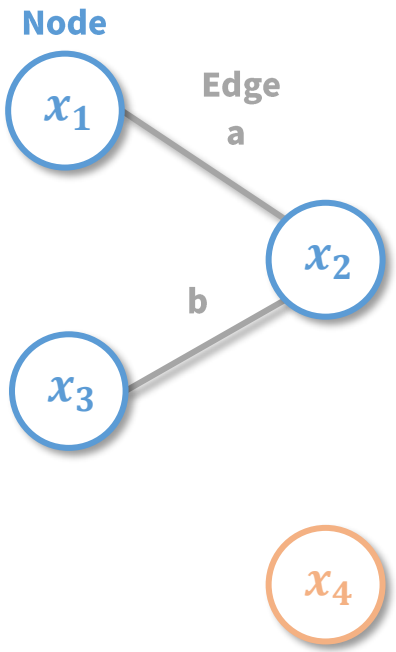
$O(n + m)$

$n = |N|, m = |E|$

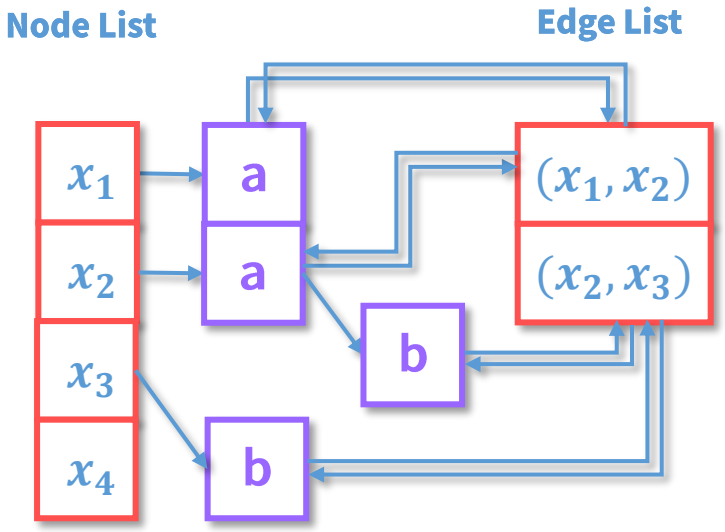
Adjacency List (1)

Insert Node

Graph Representation



Implementation



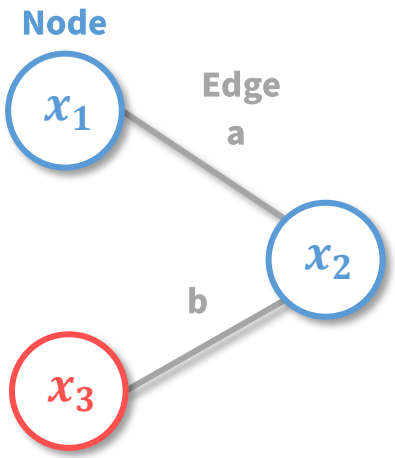
Time Complexity

worst case $O(1)$

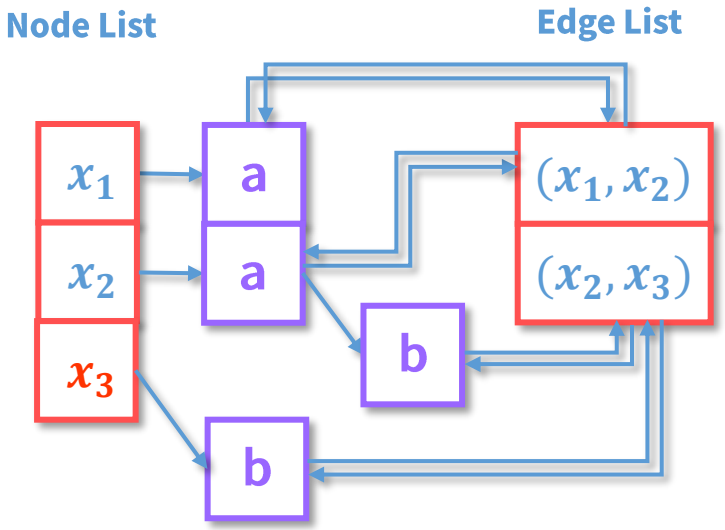
Adjacency List (2)

Delete Node (1)

Graph Representation



Implementation

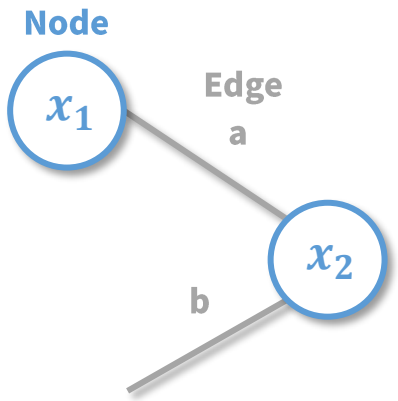


Time Complexity

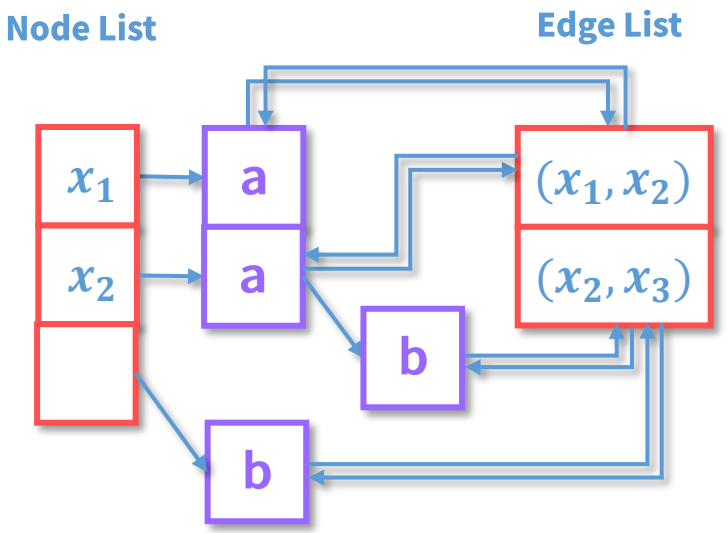
Adjacency List (3)

Delete Node (2)

Graph Representation



Implementation

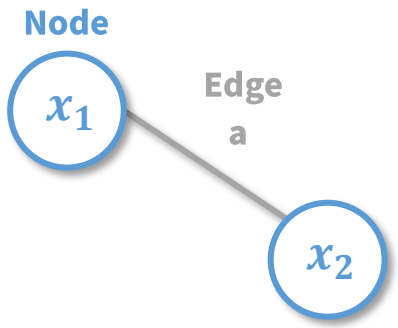


Time Complexity

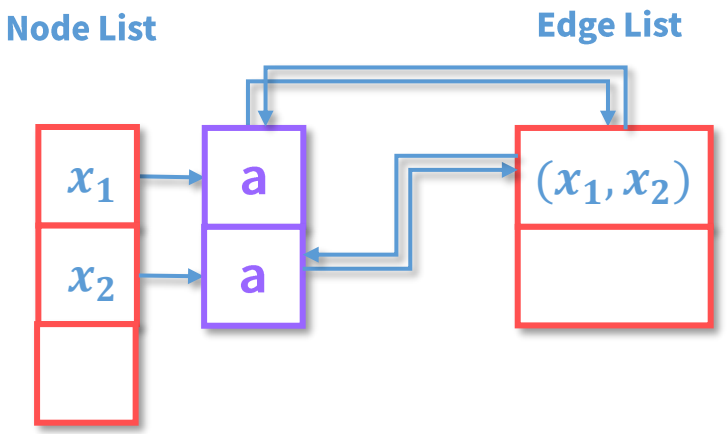
Adjacency List (4)

Delete Node (3)

Graph Representation



Implementation



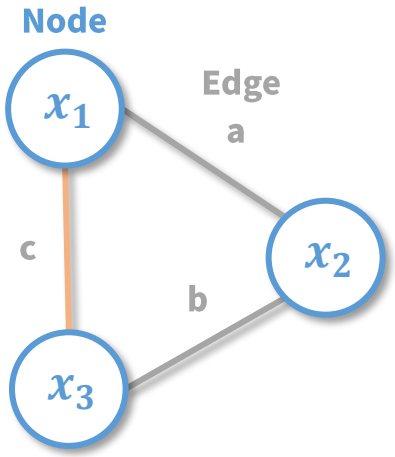
Time Complexity

worst case $O(\deg(x))$

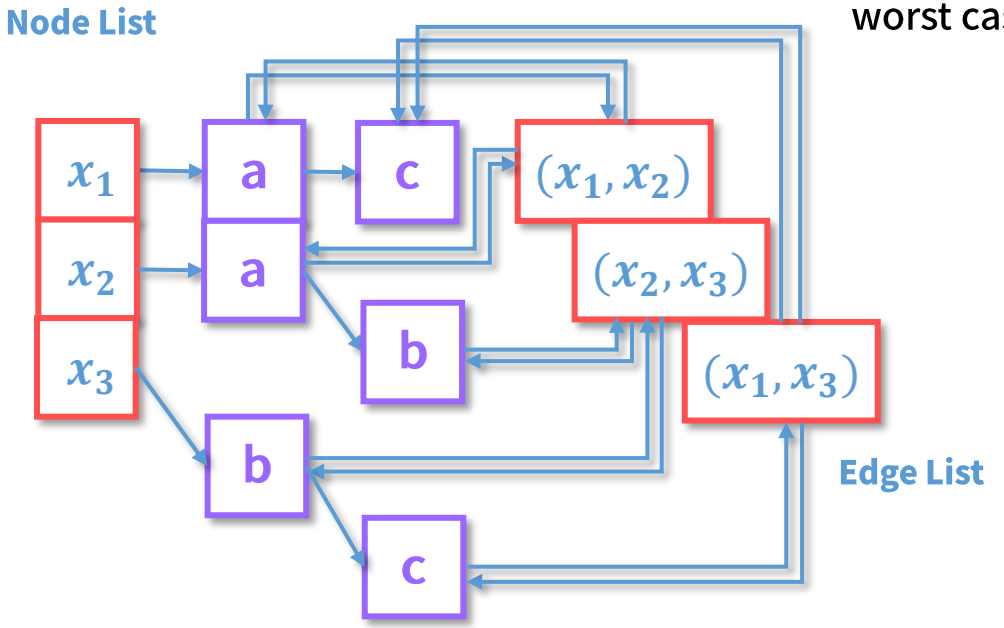
Adjacency List (5)

Insert Edge

Graph Representation



Implementation



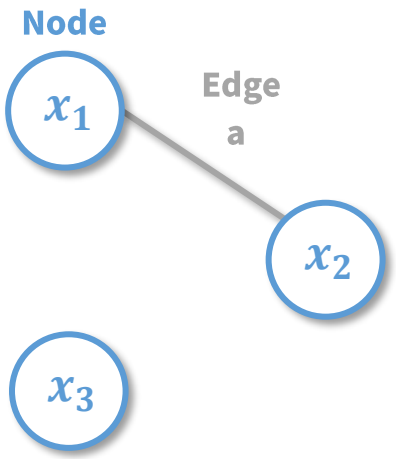
Time Complexity

worst case $O(1)$

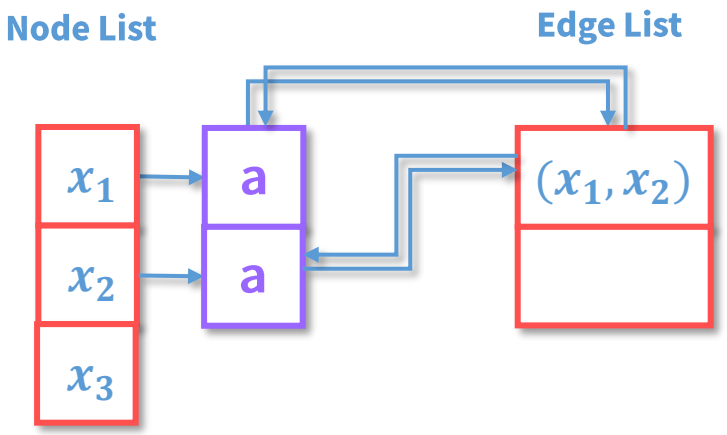
Adjacency List (6)

Delete Edge

Graph Representation



Implementation



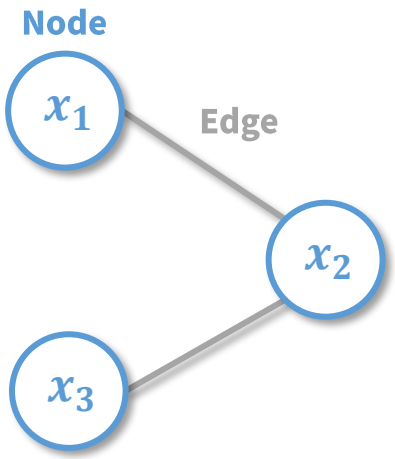
Time Complexity

worst case $O(1)$

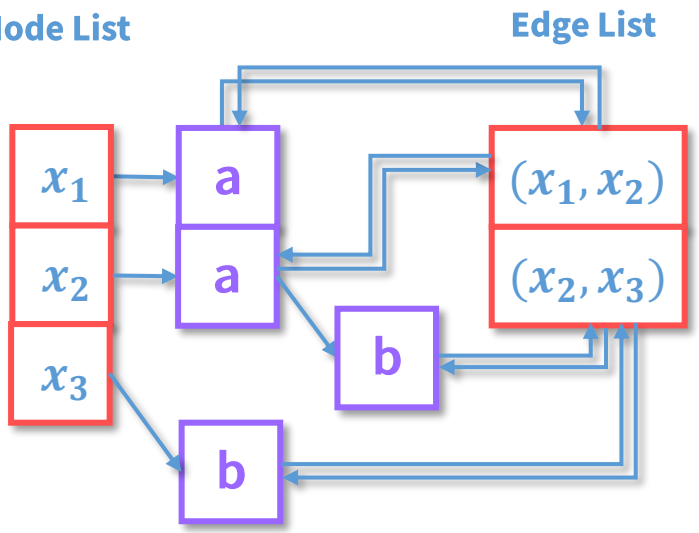
Adjacency List (7)

Adjacent

Graph Representation



Implementation



Time Complexity

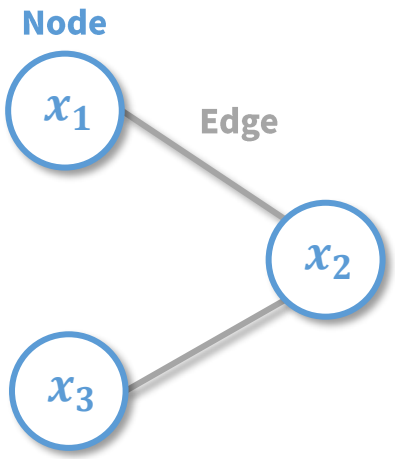
worst case $O(1)$

are x_1 and x_3 adjacent to each other

Adjacency List (8)

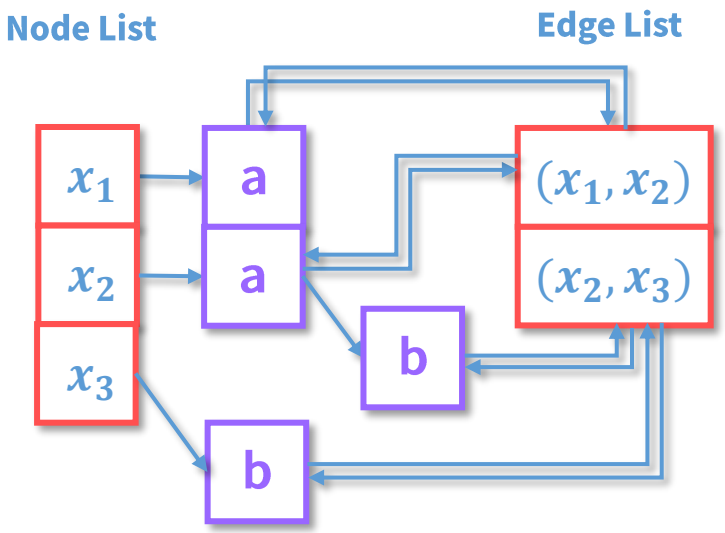
Edges

Graph Representation



edges of x_1

Implementation



Time Complexity

worst case $O(\deg(x))$

Graph Algorithms - Traversal

Depth First Search Traversal (1)

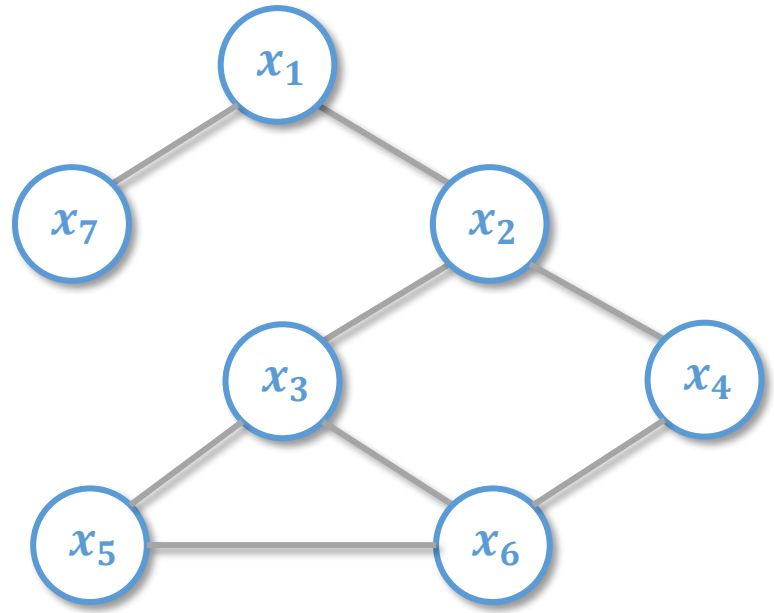
- Applications
  - Path discovery.
  - Detect cycle in graph.
  - Find connected components.
  - Check if graph is bipartite.
  - Topological sorting.
  - Solving mazes.
  - Find maximum flow in graph.
  - Phylogenetic analysis.

Depth First Search Traversal (2)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



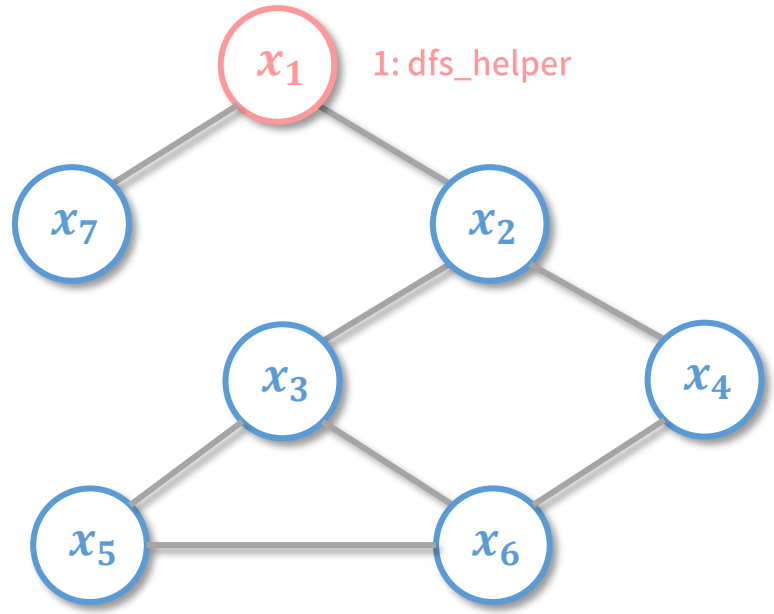
Result

Depth First Search Traversal (3)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



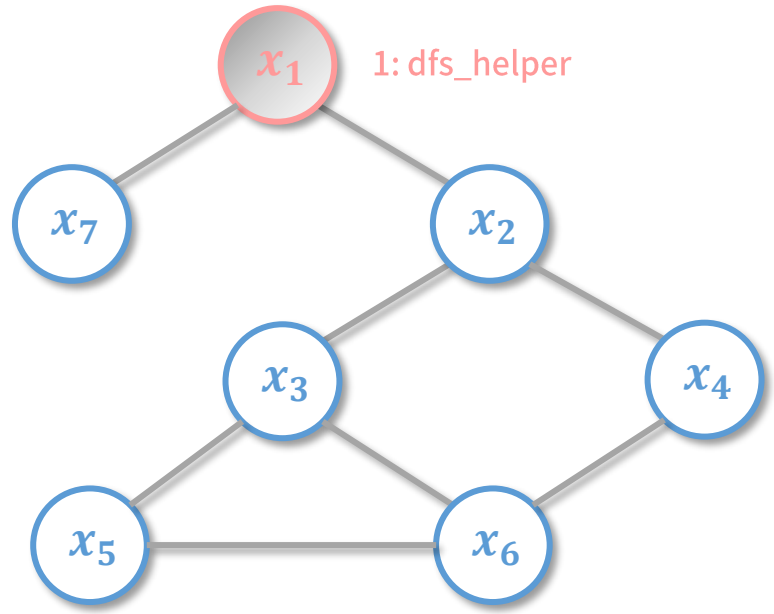
Result

Depth First Search Traversal (4)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



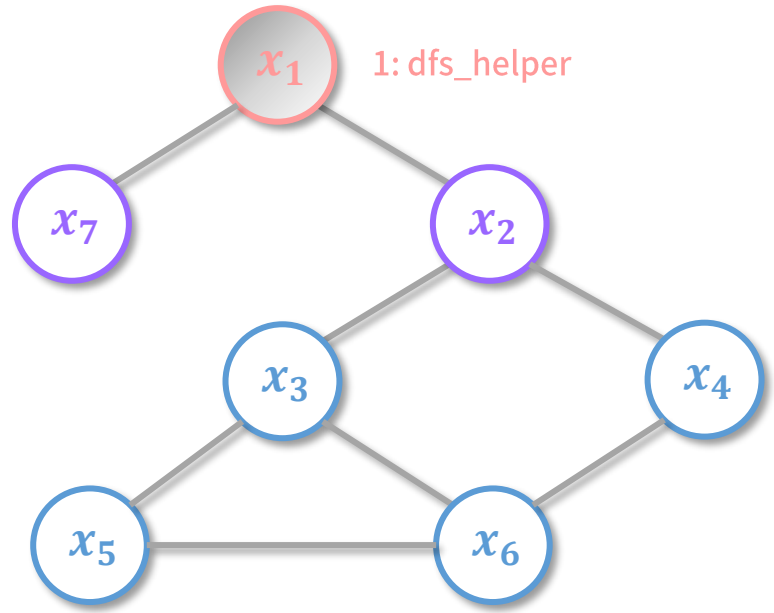
Result

Depth First Search Traversal (5)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



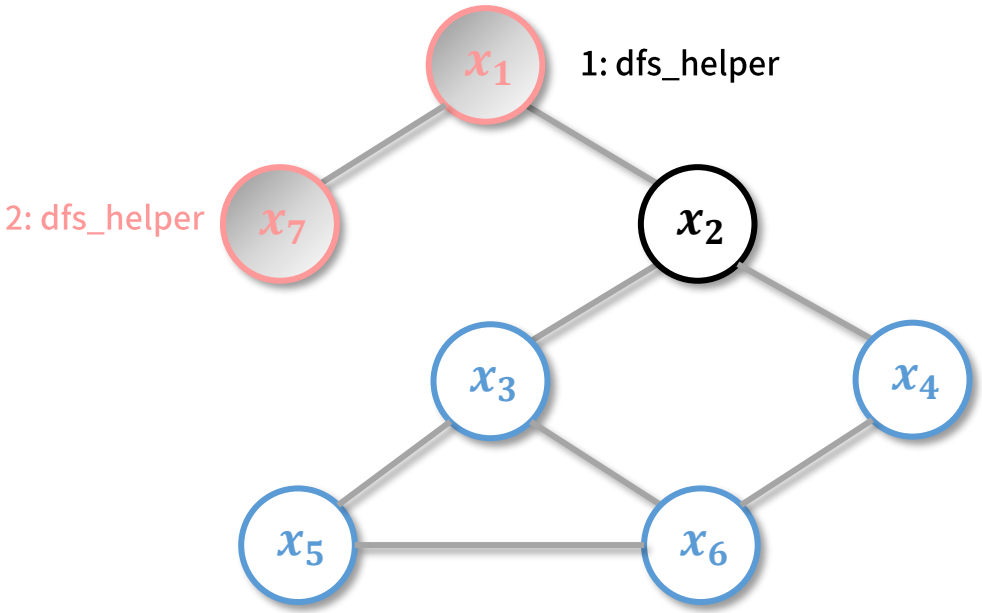
Result

Depth First Search Traversal (6)

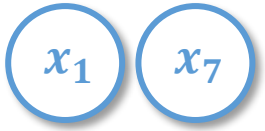
Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

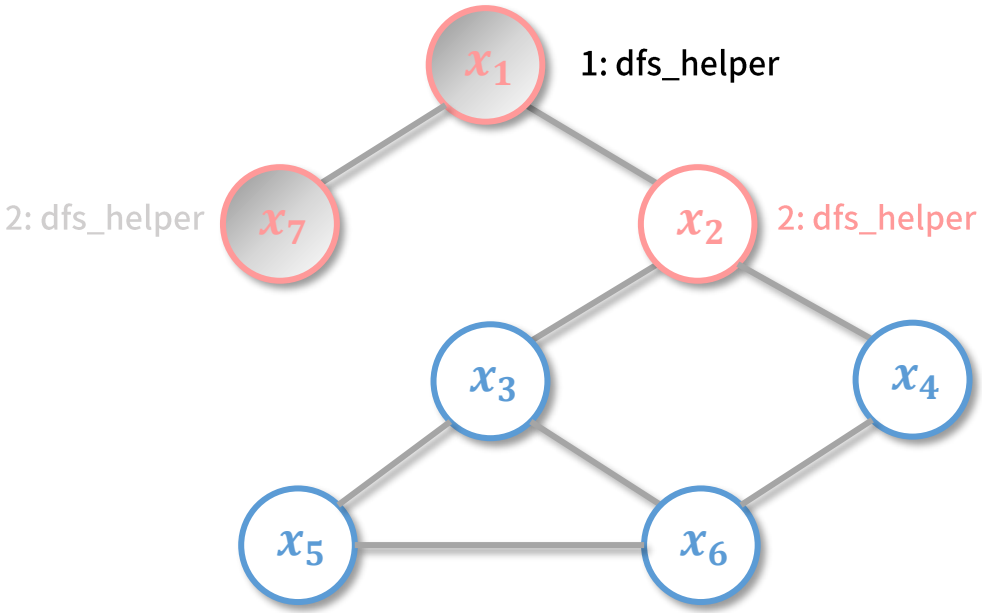


Depth First Search Traversal (7)

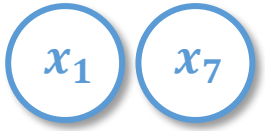
Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

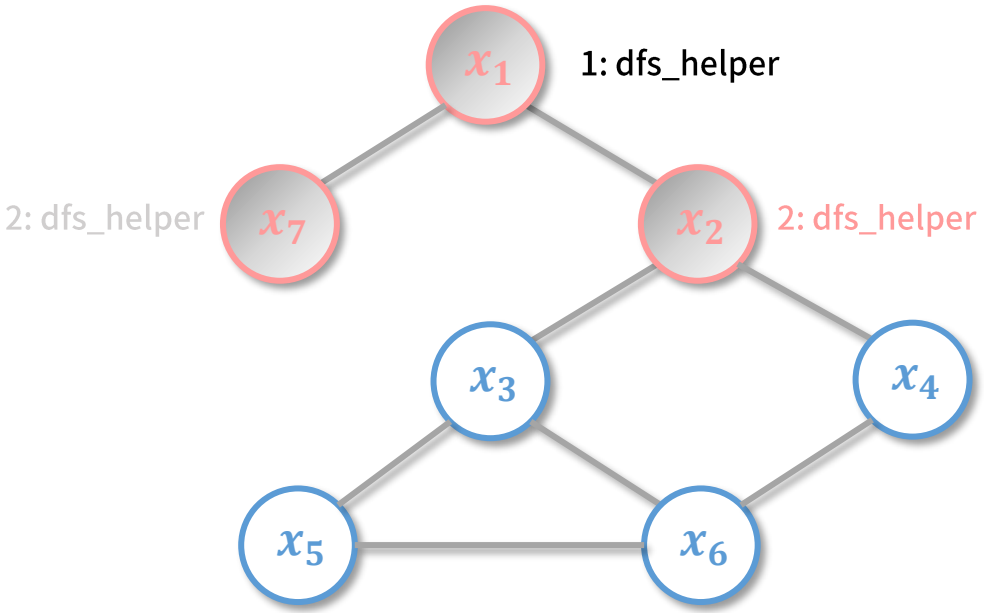


Depth First Search Traversal (8)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



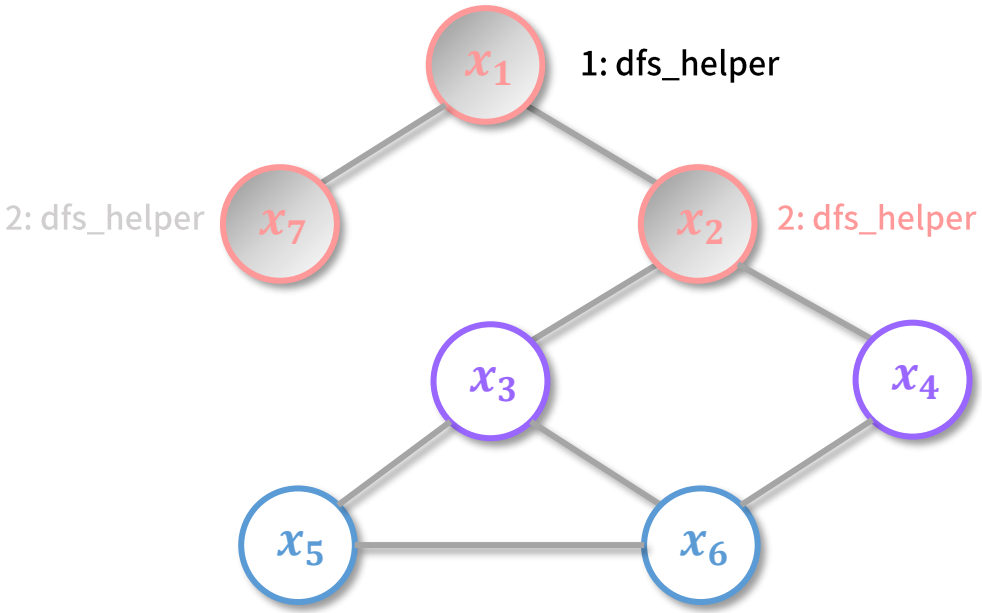
Result

Depth First Search Traversal (9)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



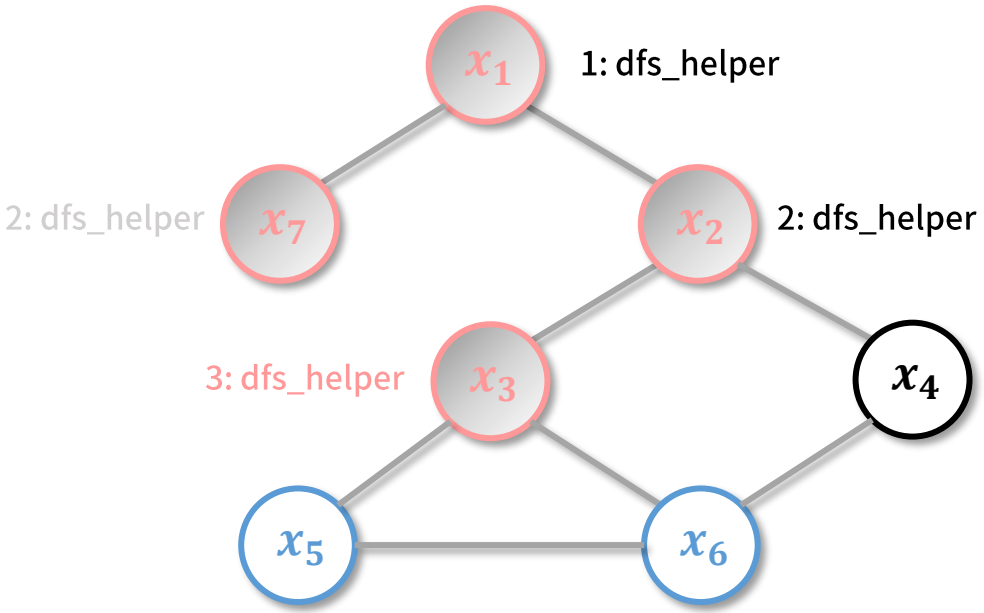
Result

Depth First Search Traversal (10)

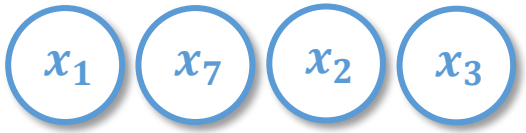
Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

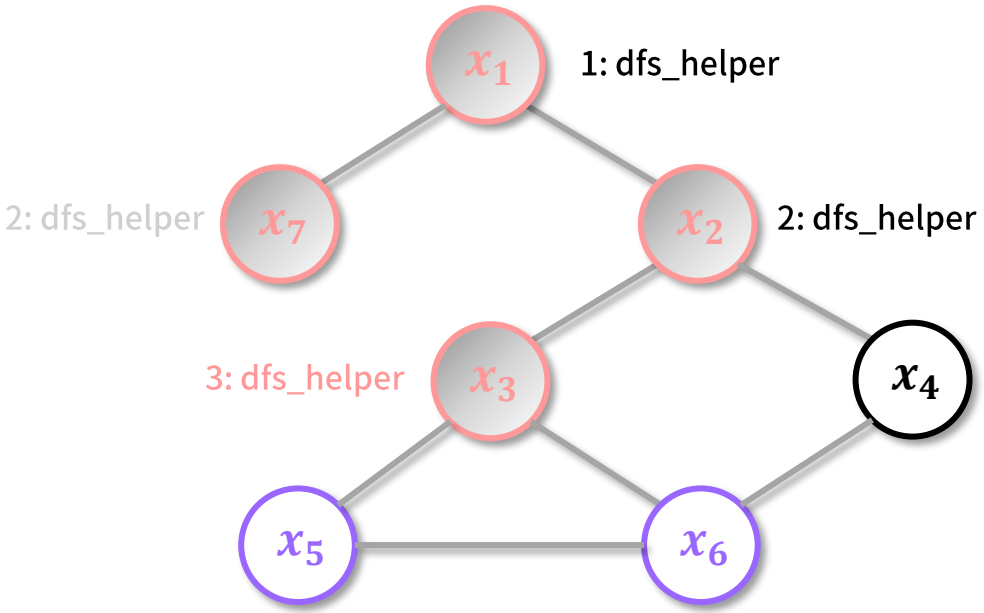


Depth First Search Traversal (11)

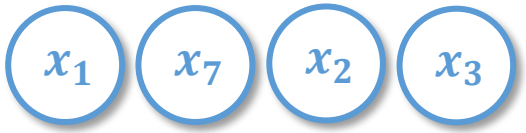
Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

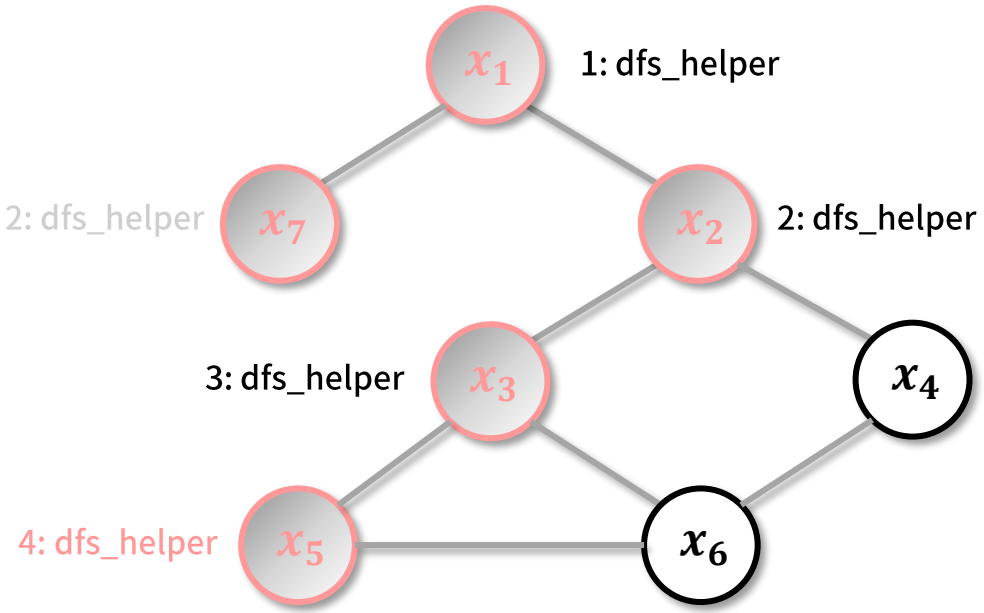


Depth First Search Traversal (12)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



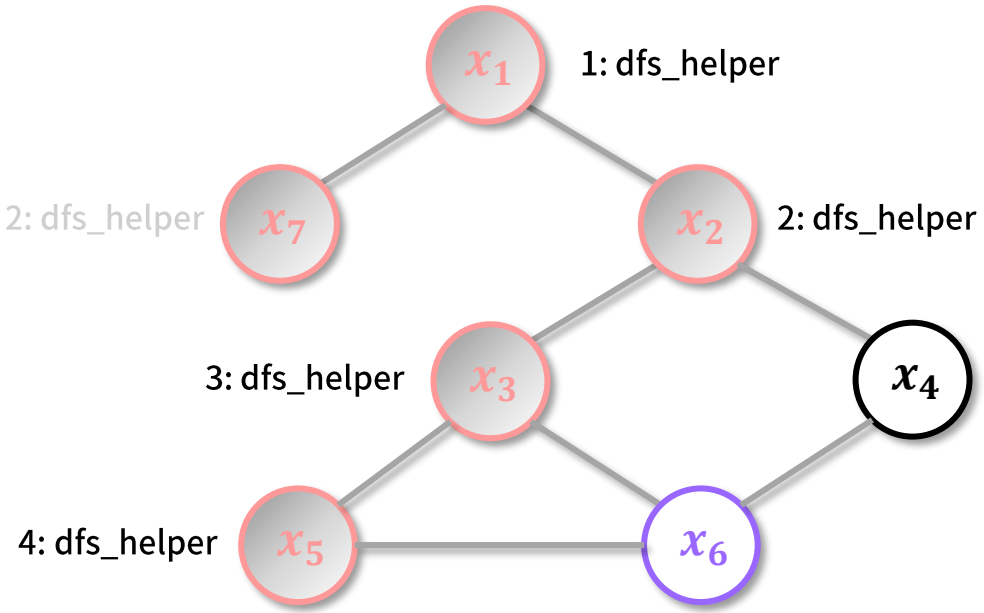
Result

Depth First Search Traversal (13)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



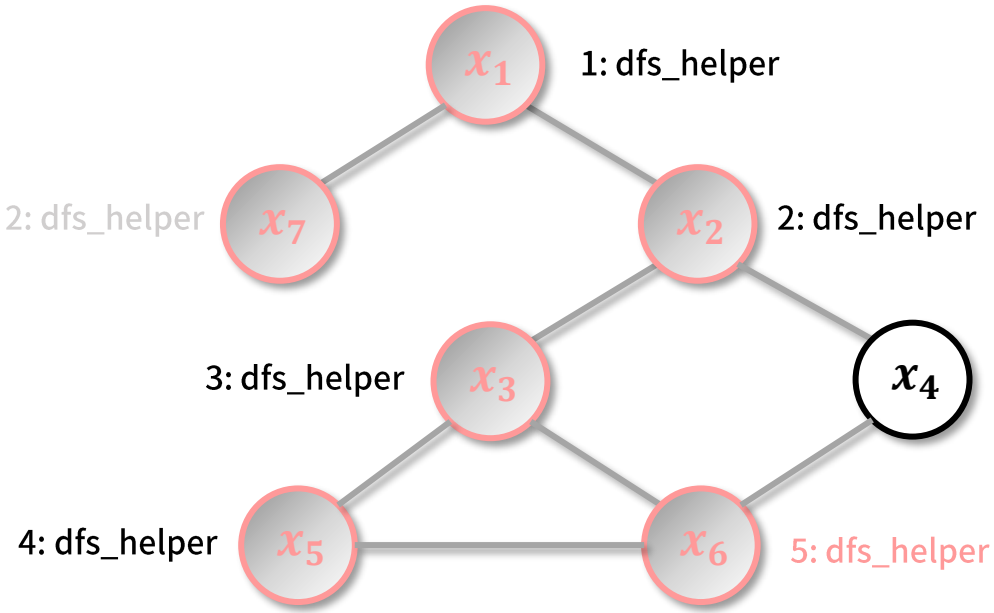
Result

Depth First Search Traversal (14)

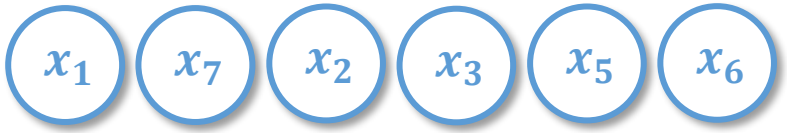
Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

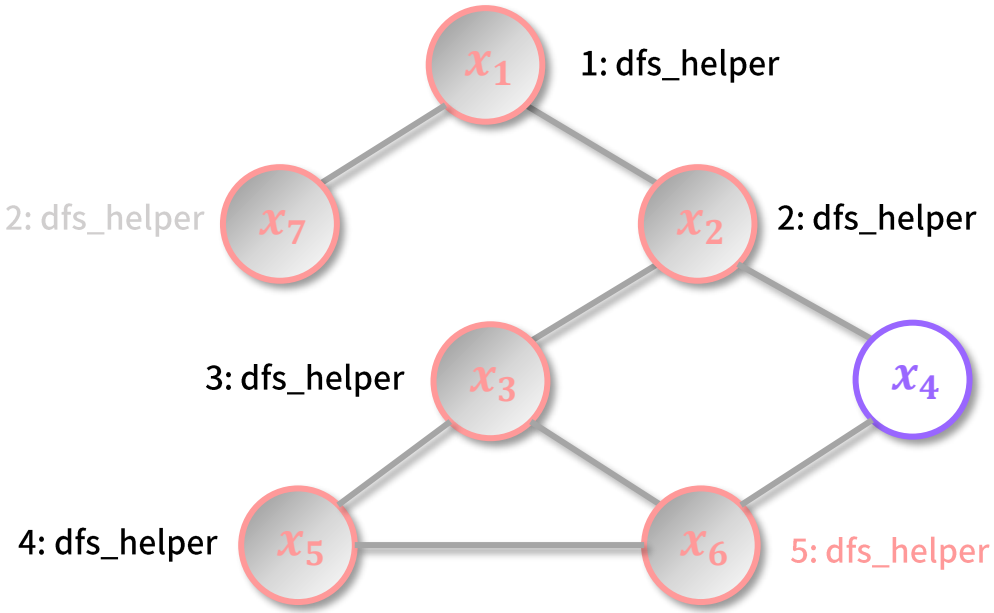


Depth First Search Traversal (15)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



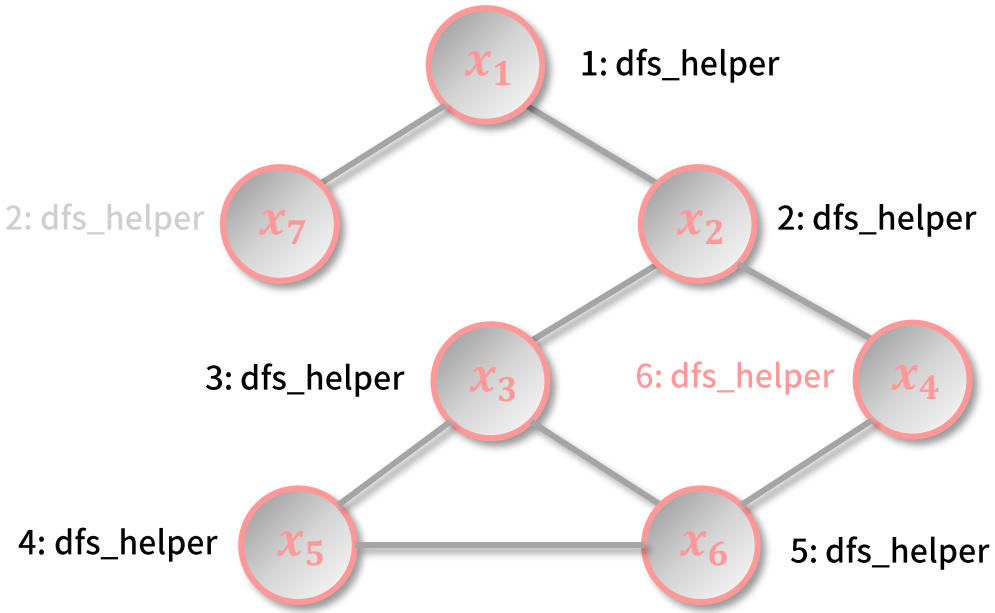
Result

Depth First Search Traversal (16)

Implementation

```python
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



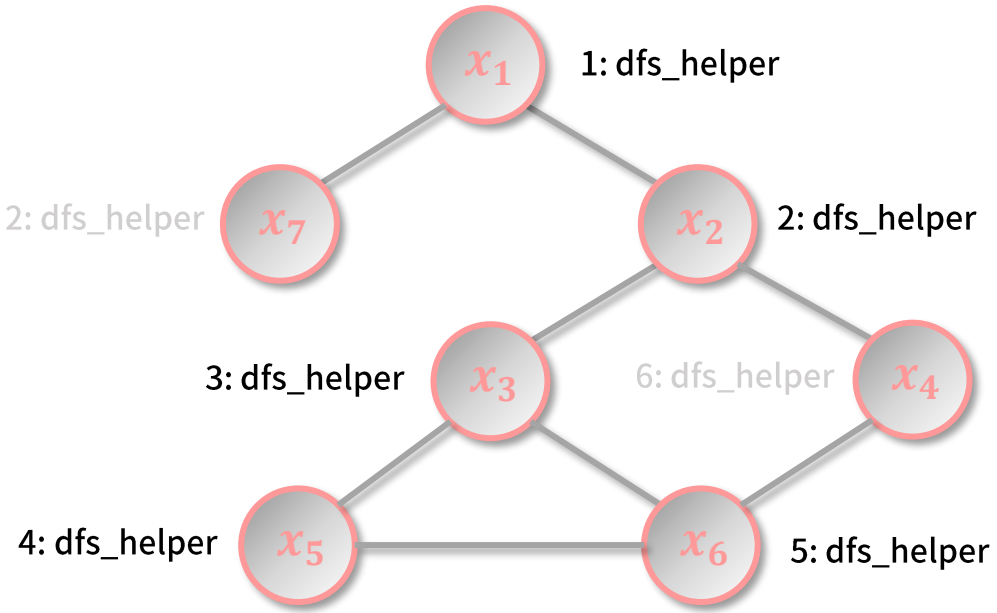
Result

Depth First Search Traversal (17)

Implementation

```
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



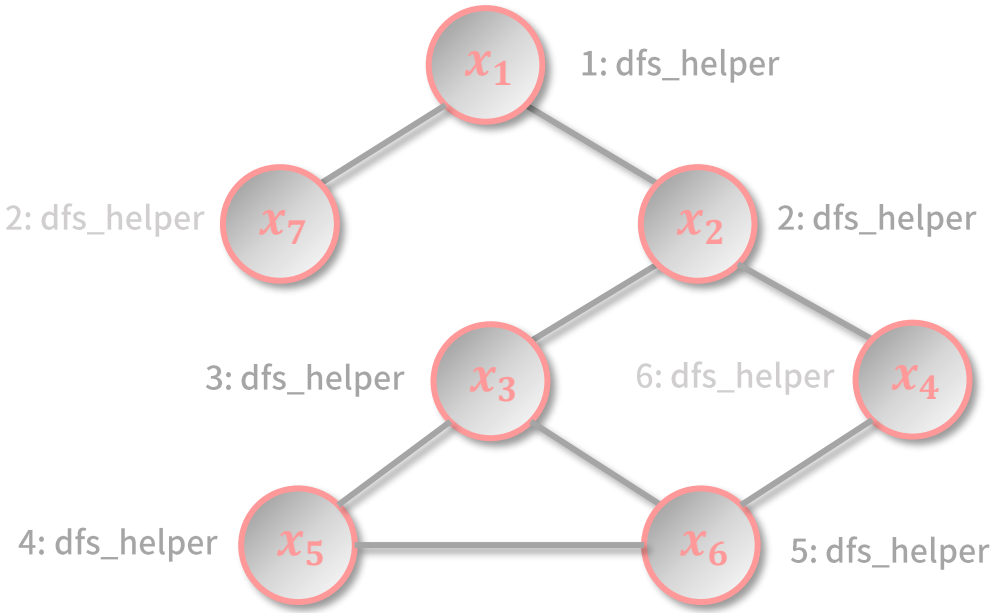
Result

Depth First Search Traversal (18)

Implementation

```
def dfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    dfs_helper(graph, node, visited, result)

  return result

def dfs_helper(
    graph: Graph,
    node: Node,
    visited: Set,
    result: List[Node]) -> None:

  if node not in visited:
    result.append(node)
    visited.add(node)
    for neighbour in graph.edges(node):
      dfs_helper(visited, graph, neighbour, visited, result)

  return
```

Graph Representation



Result

Breadth First Search Traversal (1)

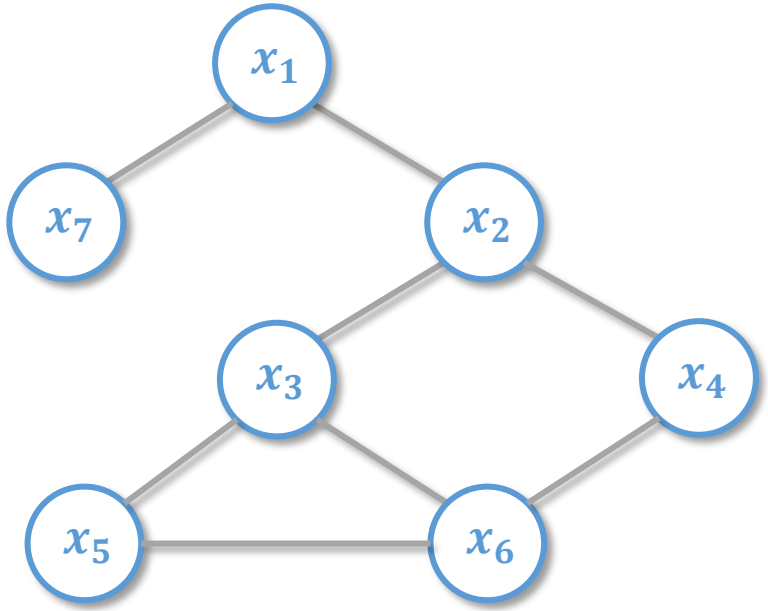
- Applications
  - Path discovery.
  - Detect cycle in graph.
  - Check if graph is bipartite.
  - Find shortest path.
  - Find minimum spanning tree.
  - Signal broadcasting.
  - Find maximum flow in graph.

Breadth First Search Traversal (2)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

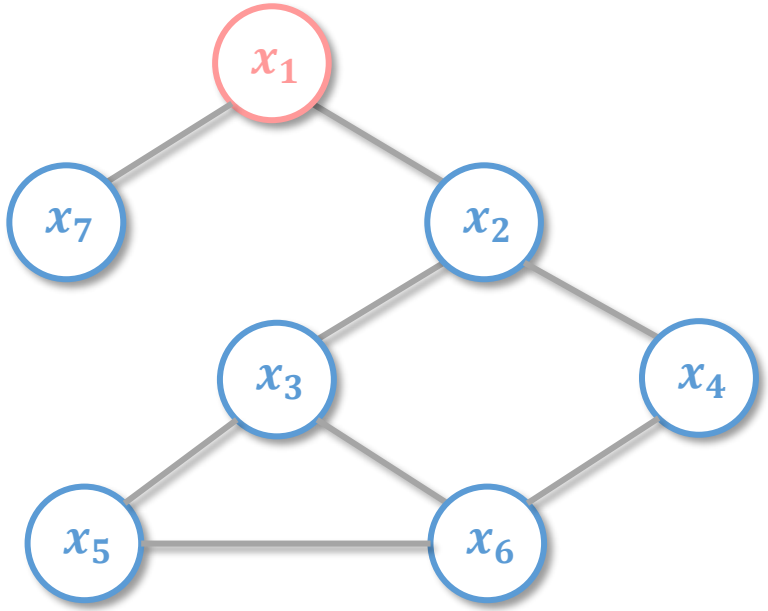
Result

Breadth First Search Traversal (3)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

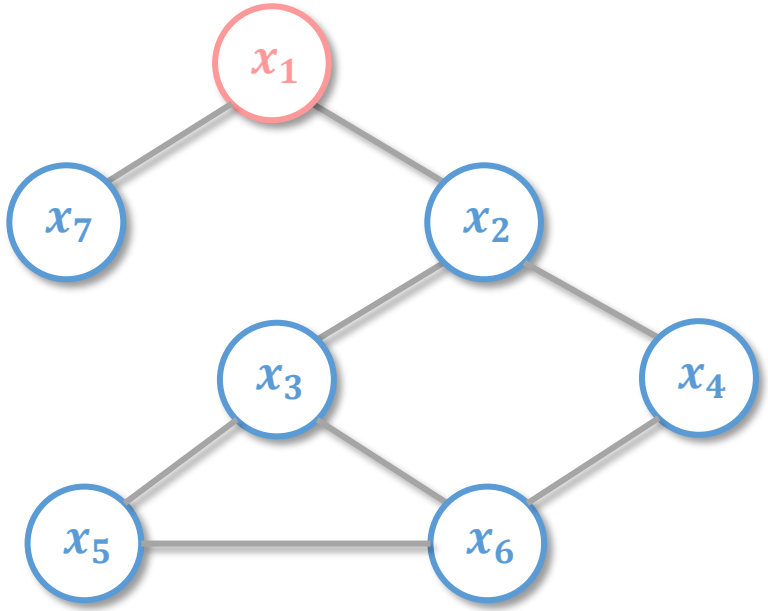
Result

Breadth First Search Traversal (4)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

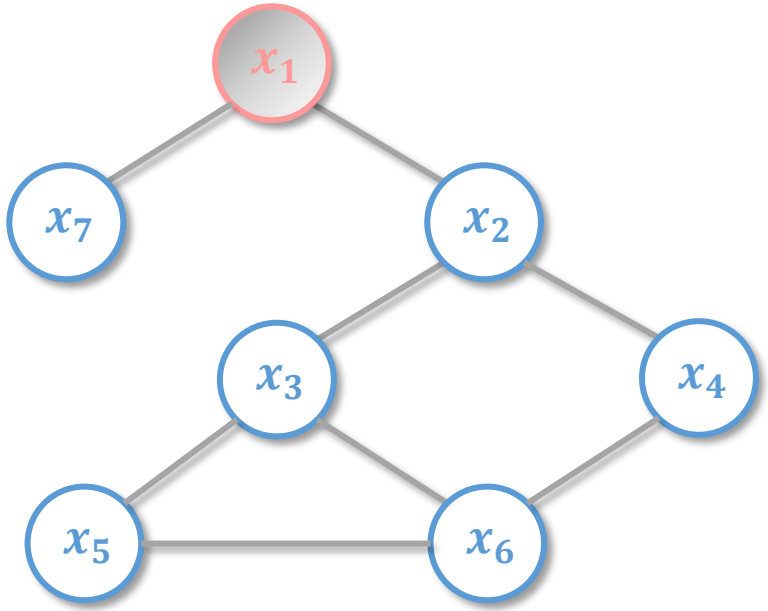
Result

Breadth First Search Traversal (5)

Implementation

```python
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

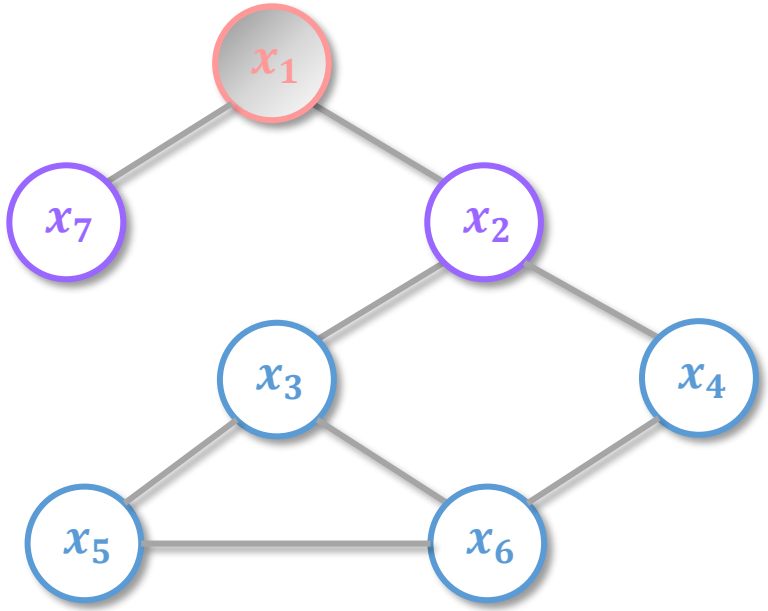
Result

Breadth First Search Traversal (6)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

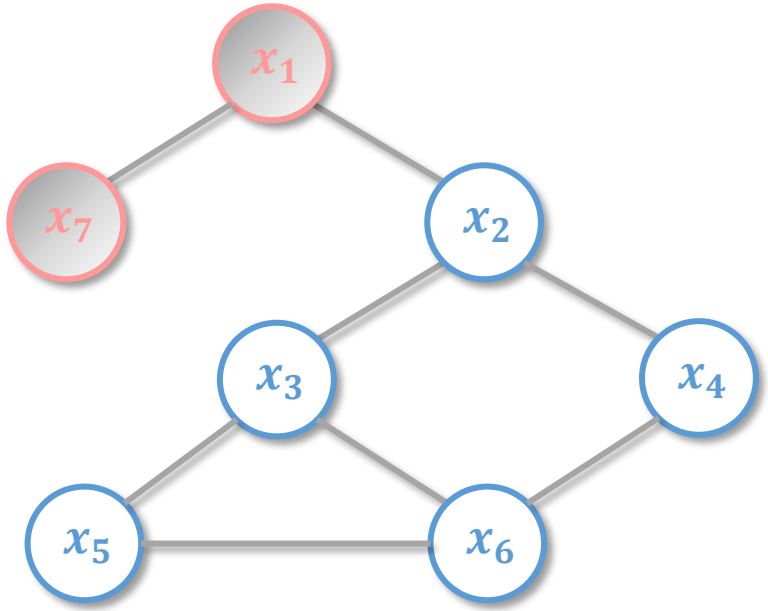
Result

Breadth First Search Traversal (7)

Implementation

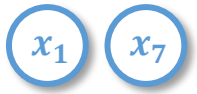
```python
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

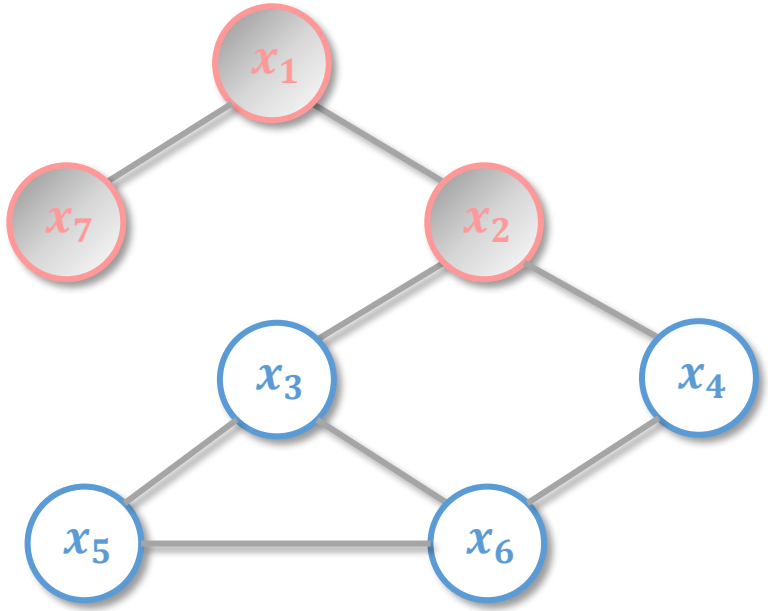


Breadth First Search Traversal (8)

Implementation

```python
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

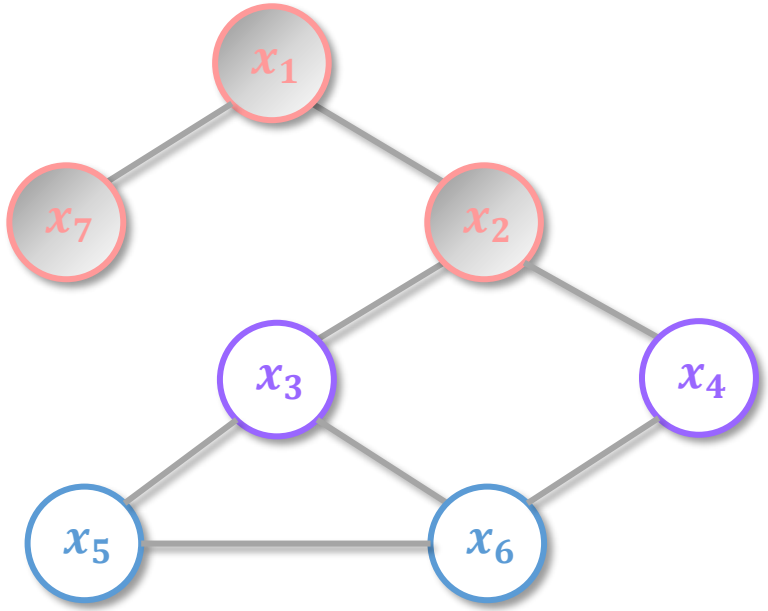
Result

Breadth First Search Traversal (9)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

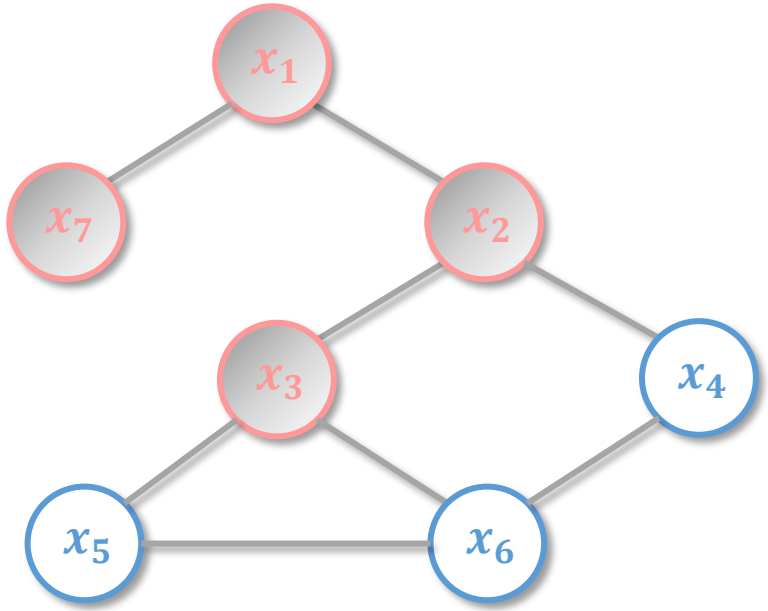
Result

Breadth First Search Traversal (10)

Implementation

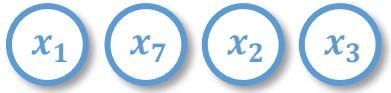
```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

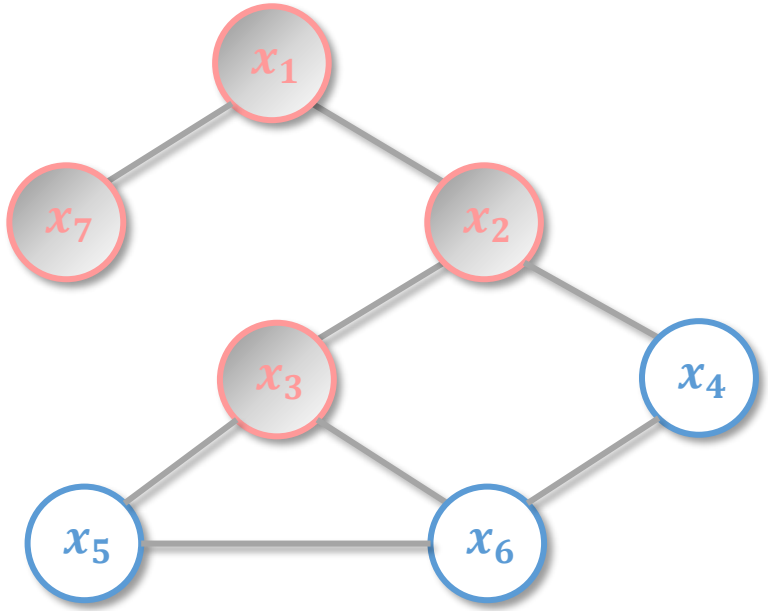


Breadth First Search Traversal (11)

Implementation

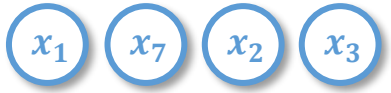
```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

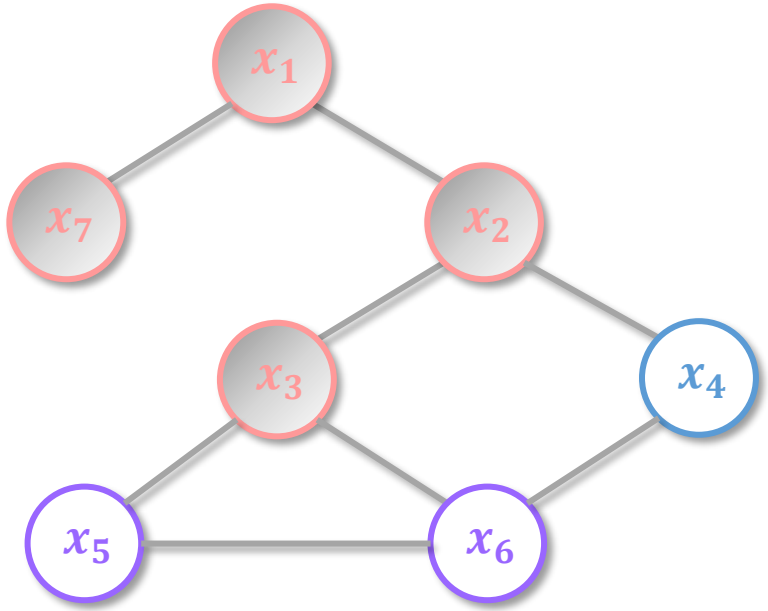


Breadth First Search Traversal (12)

Implementation

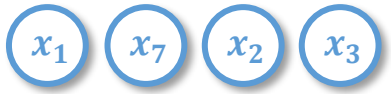
```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

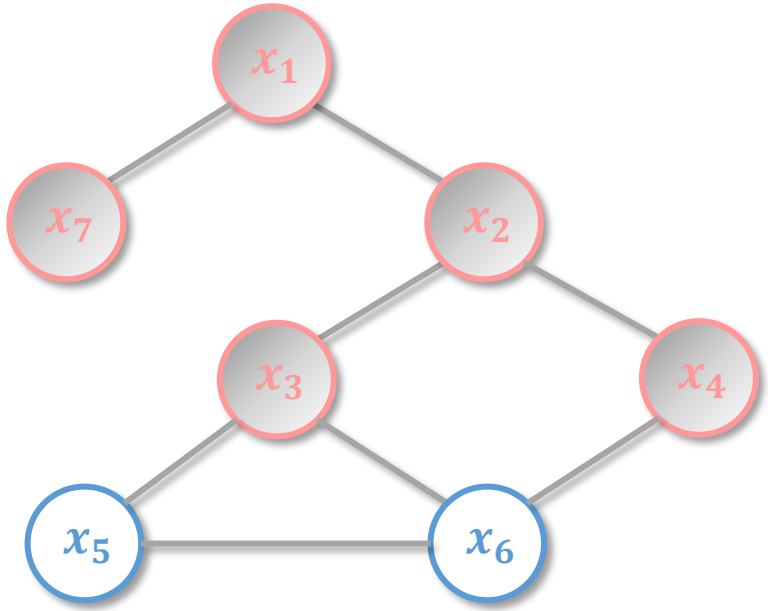


Breadth First Search Traversal (13)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

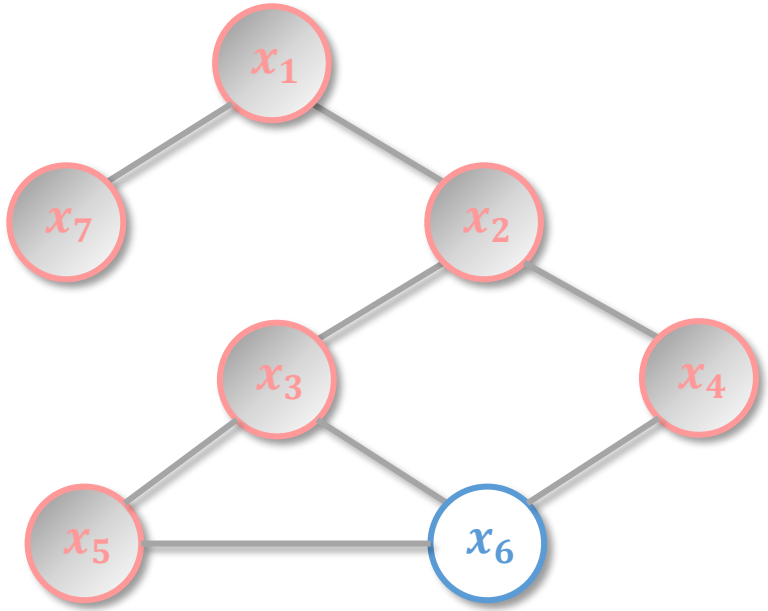
Result

Breadth First Search Traversal (14)

Implementation

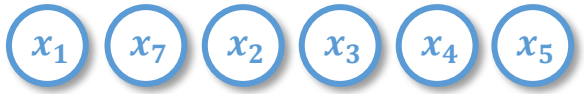
```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

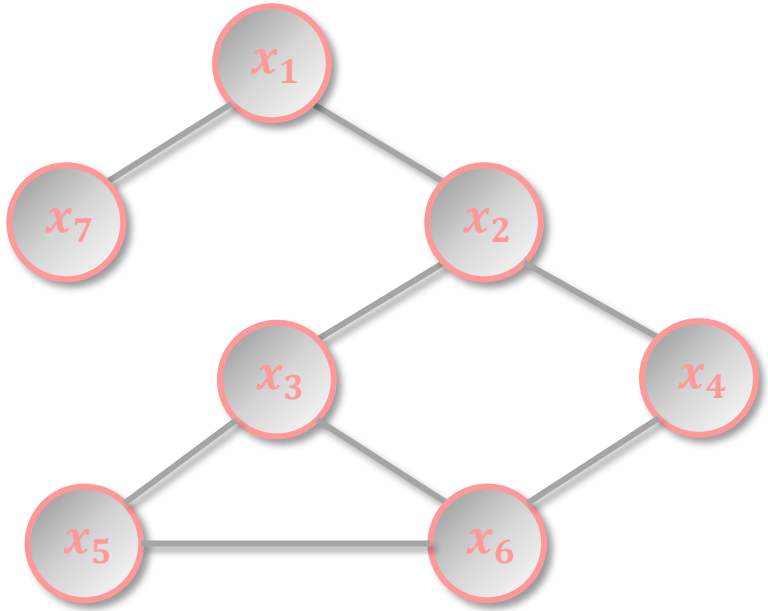


Breadth First Search Traversal (15)

Implementation

```
def bfs(graph: Graph) -> List[Node]:
  visited: Set[Node] = set()
  result: List[Node] = list()
  for node in graph.nodes:
    bfs_helper(graph, node, visited, result)

  return result

def bfs_helper(
      graph: Graph,
      node: Node,
      visited: Set[Node],
      result: List[Node]) -> None:

    queue: Queue[Node] = Queue()
    queue.enqueue(node)

    while len(queue) != 0:
      node = queue.dequeue()
      result.append(node)
      visited.add(node)
      for neighbor in graph.edges(node):
        if node not in visited:
          queue.enqueue(neighbor)

    return
```

Graph Representation



Queue

Result

Summary

Worst Time Complexity

Depth First Search	Breadth First Search
$O(n+m)$	$O(n+\mathrm{m})$

$* \ n = |N|, m = |E|$

Graph Algorithms – Minimum Spanning Tree

Minimum Spanning Tree (1)

- Goal
  - Find a set of edges that connect all the nodes in a weighted undirected graph with the minimum sum of edge weight.
- Applications
  - Hierarchical classification.
  - Clustering.
  - Find maximum flow in graph.
  - Find lowest cost connecting the graph.
  - Network planning.
  - Circuit design.

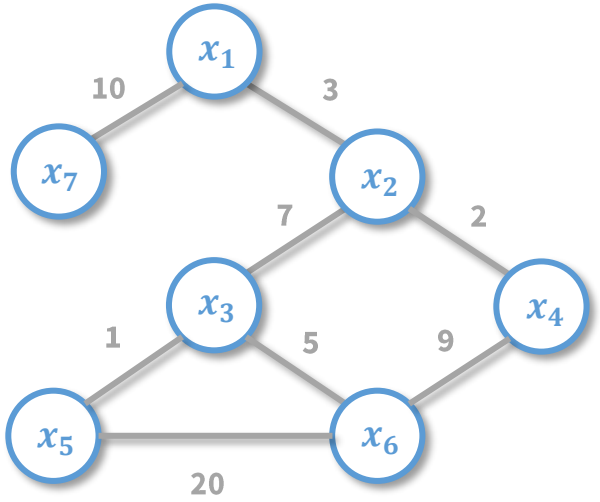
Kruskal's Algorithm (1)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

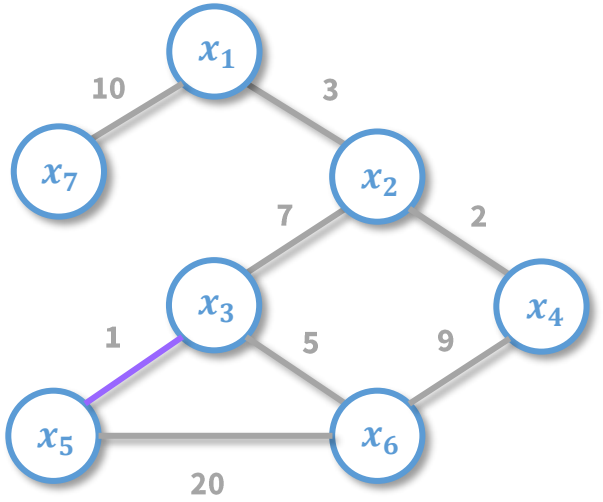
Result

Kruskal's Algorithm (2)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

UnionFind

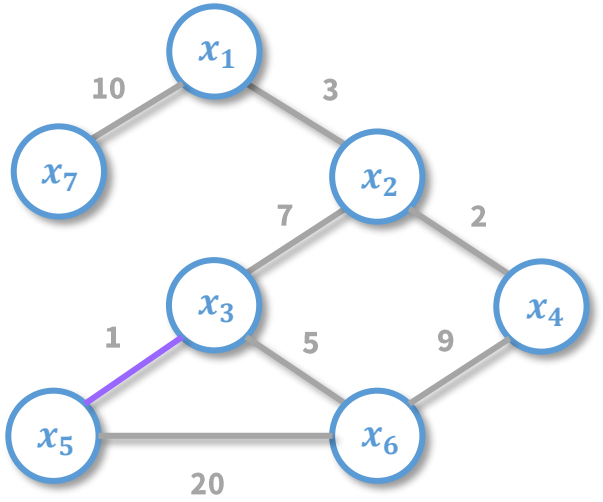
Result

Kruskal's Algorithm (3)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

UnionFind

Result

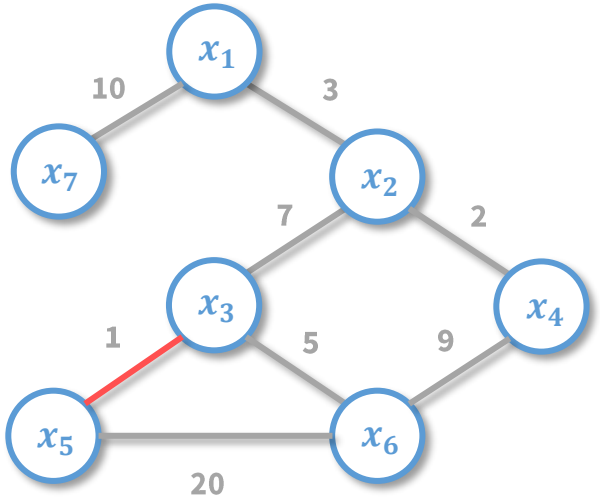
Kruskal's Algorithm (4)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5)

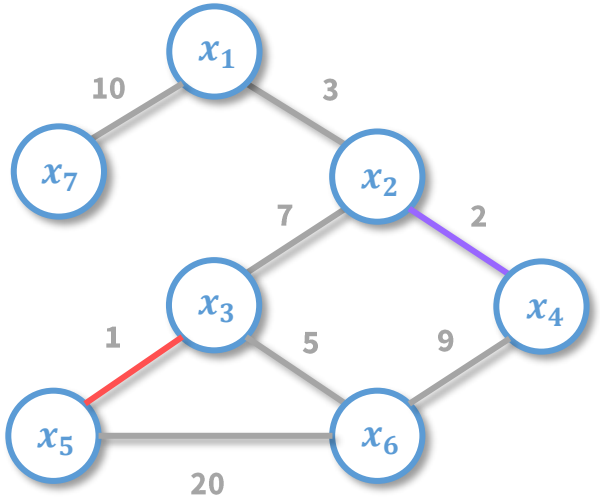
Kruskal's Algorithm (5)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5)

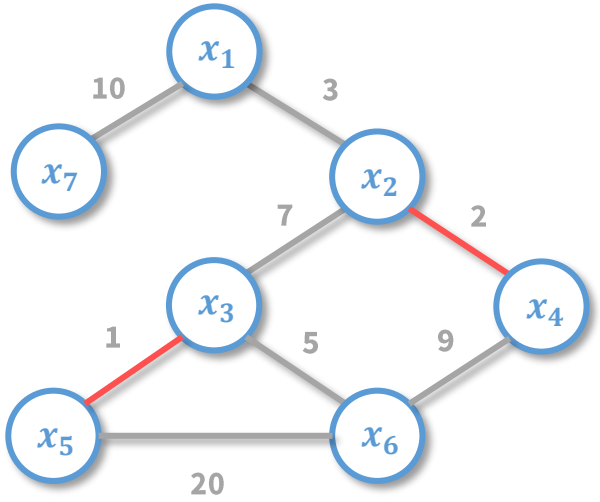
Kruskal's Algorithm (6)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4)

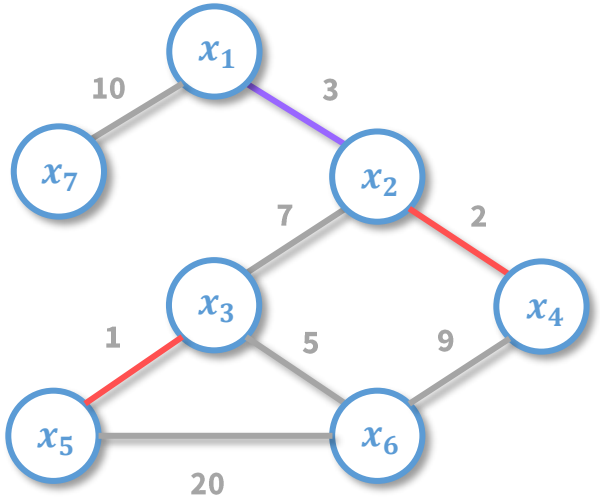
Kruskal's Algorithm (7)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4)

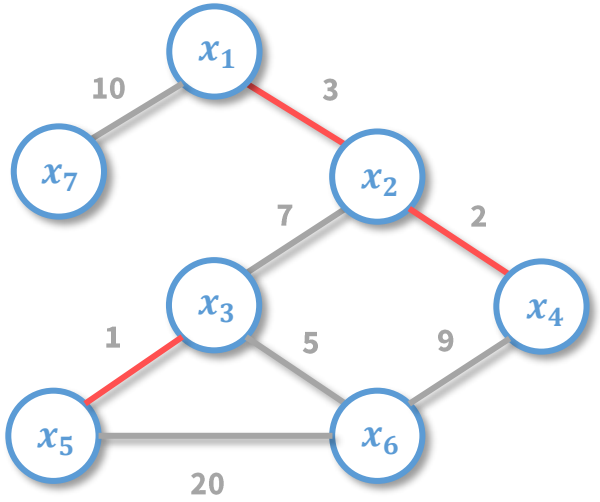
Kruskal's Algorithm (8)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

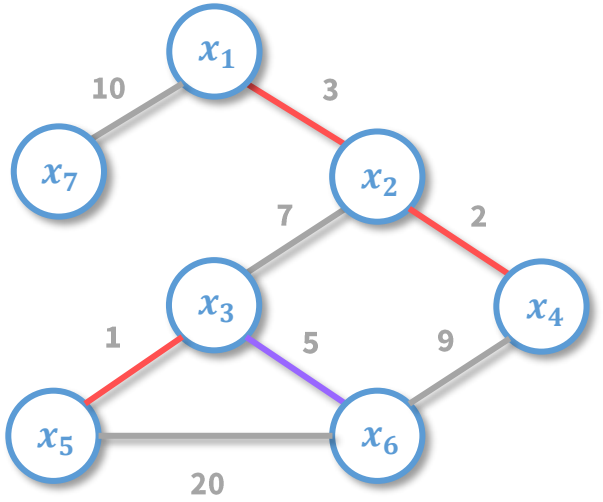
(x_3, x_5) (x_2, x_4) (x_1, x_2)

Kruskal's Algorithm (9)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

UnionFind

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2)

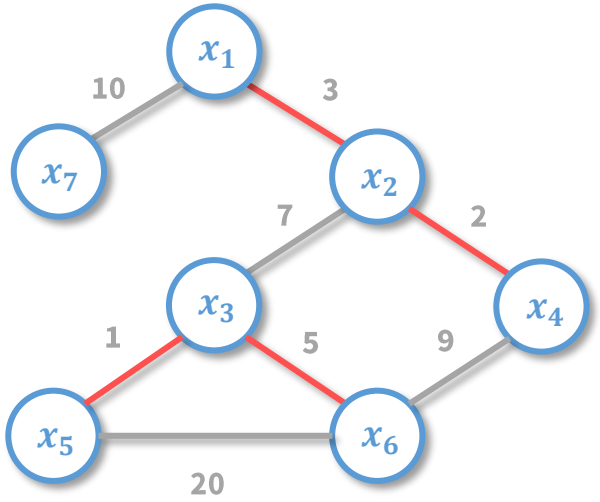
Kruskal's Algorithm (10)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6)

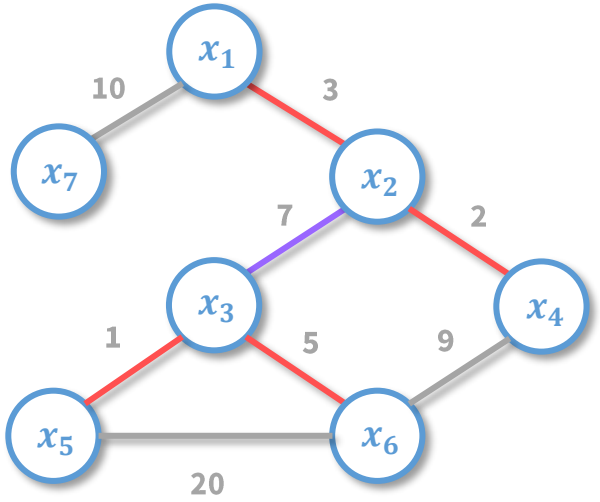
Kruskal's Algorithm (11)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6)

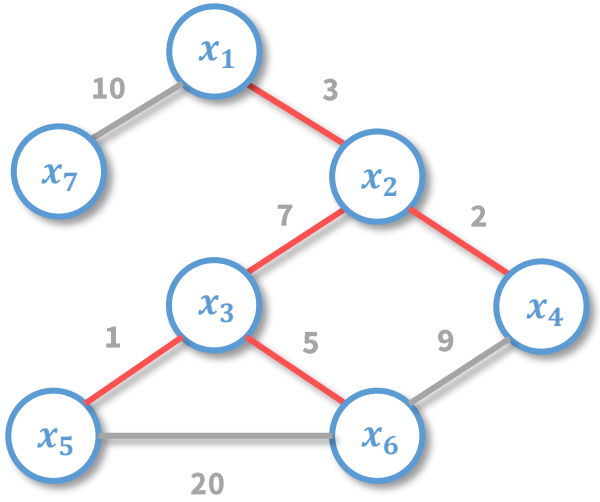
Kruskal's Algorithm (12)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3)

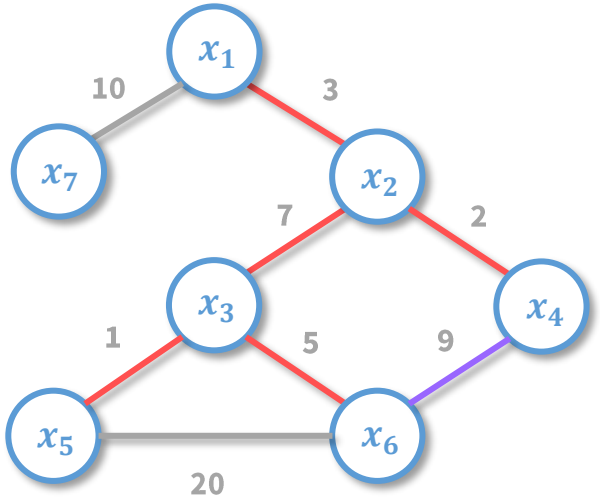
Kruskal's Algorithm (13)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3)

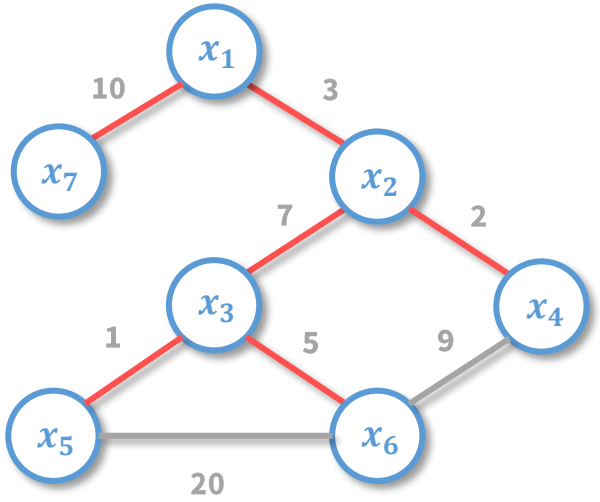
Kruskal's Algorithm (14)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7) (x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3)

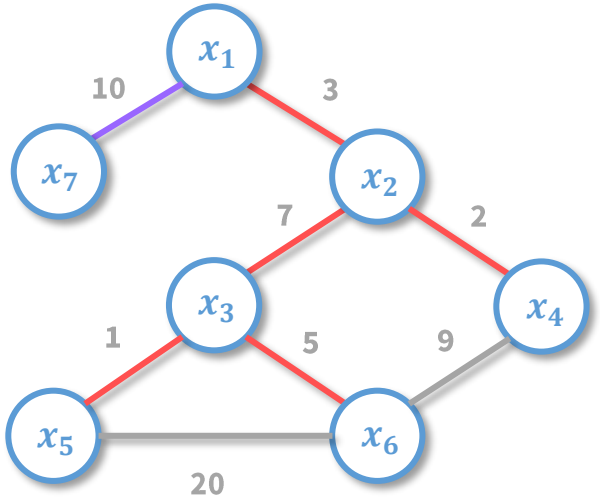
Kruskal's Algorithm (15)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
        edge = edges.pop()
        node_a = edge.node_a
        node_b = edge.node_b
        weight = edge.weight
        if not union_find.is_connected(node_a, node_b):
            result.append(edge)
            union_find.union(node_a, node_b)
            total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7)

(x_5, x_6)

Result

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3)

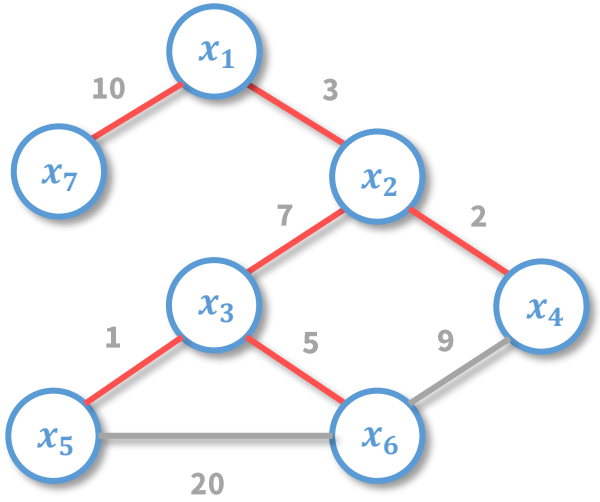
Kruskal's Algorithm (16)

Implementation

```python
def kruskal_mst(graph: Graph) -> List[Edge]:
    n_nodes = len(graph.nodes)
    edges = MinHeap.build_heap(graph.edges())
    union_find = UnionFind(graph.nodes)
    result = list()
    total_weight = 0
    while len(result) < n_nodes - 1:
      edge = edges.pop()
      node_a = edge.node_a
      node_b = edge.node_b
      weight = edge.weight
      if not union_find.is_connected(node_a, node_b):
        result.append(edge)
        union_find.union(node_a, node_b)
        total_weight += weight

    return result
```

UnionFind

Graph Representation



MinHeap

(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_4, x_6) (x_1, x_7)

(x_5, x_6)

Result

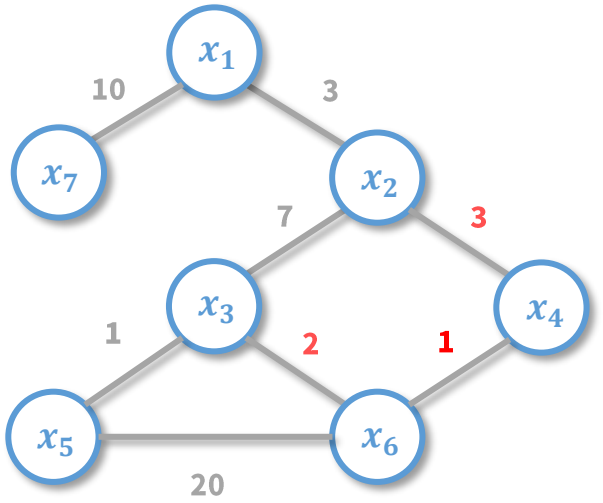
(x_3, x_5) (x_2, x_4) (x_1, x_2) (x_3, x_6) (x_2, x_3) (x_1, x_7)

Prim's Algorithm (1)

Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



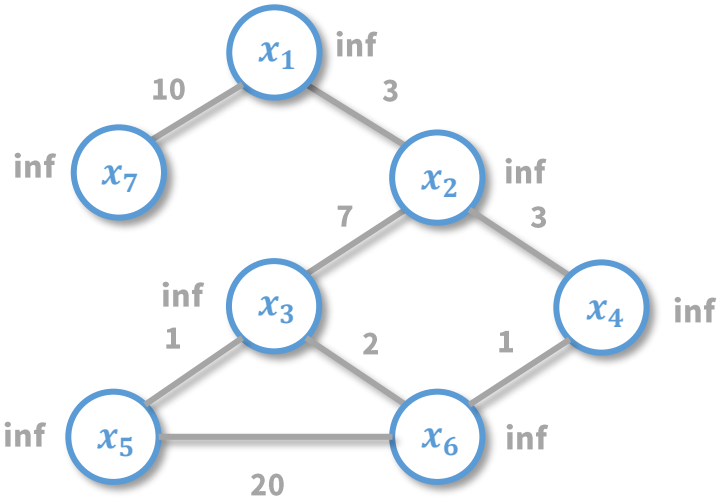
Note that the weight here is different from the one used in Kruskals algorithm to showcase some properties of Prims algorithm

Prim's Algorithm (2)

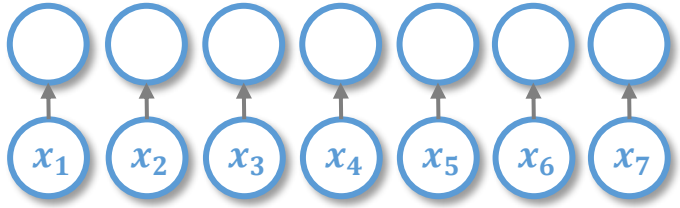
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent

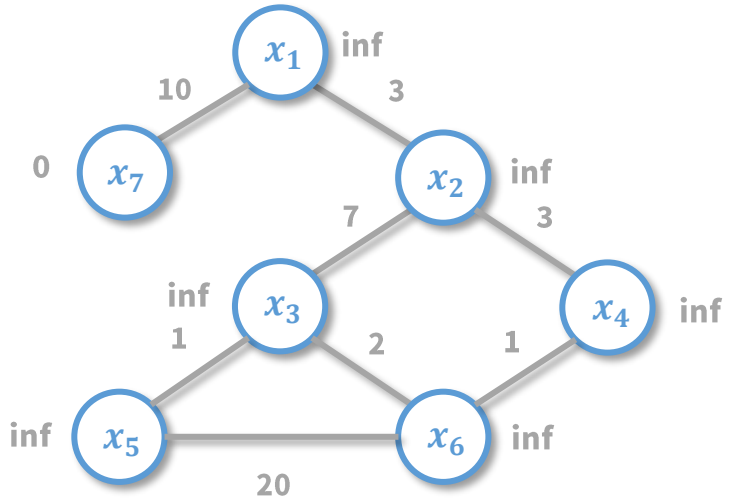


Prim's Algorithm (3)

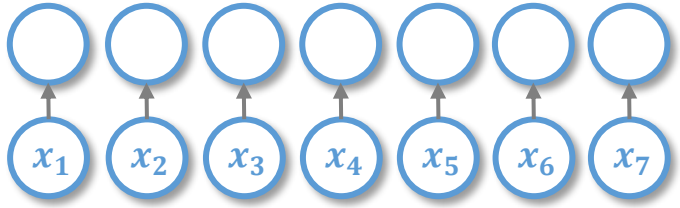
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

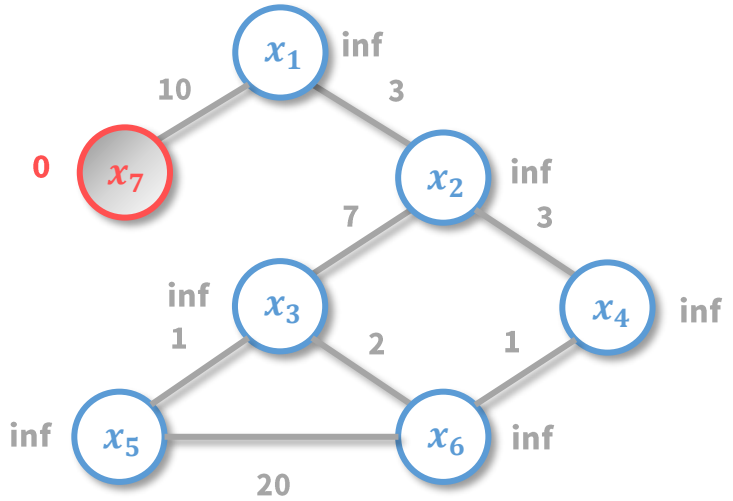
$(x_7: \mathbf{0})\ (x_1: inf)\ (x_2: inf)(x_3: inf)\ (x_4: inf)\ (x_5: inf)(x_6: inf)$

Prim's Algorithm (4)

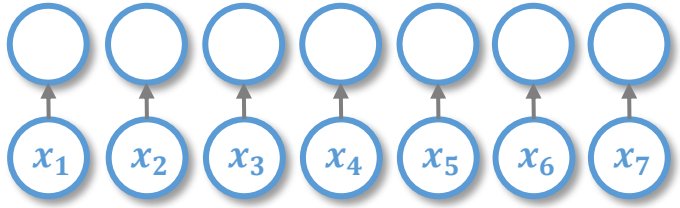
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

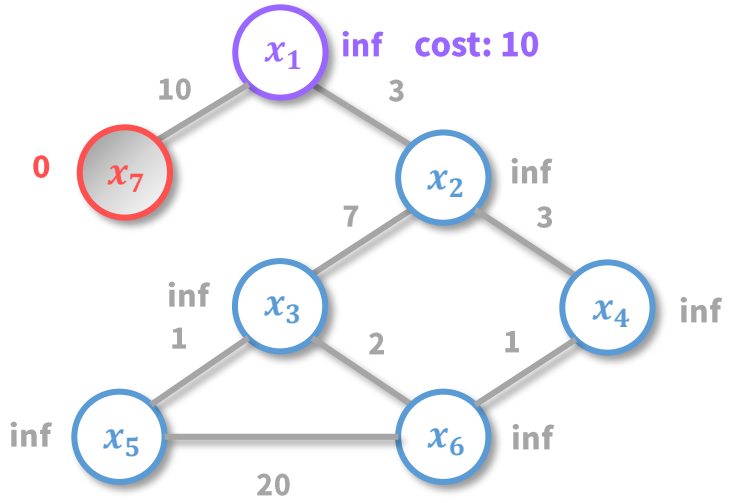
$(x_7: 0)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (5)

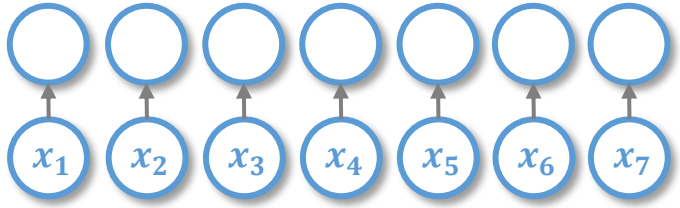
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

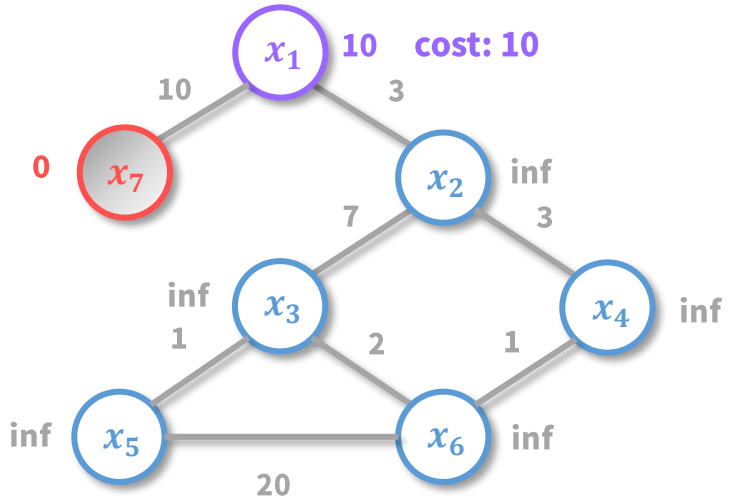
$(x_7: 0)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (6)

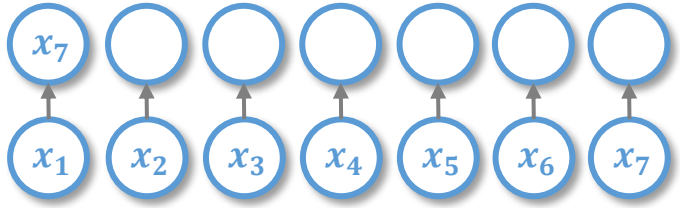
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

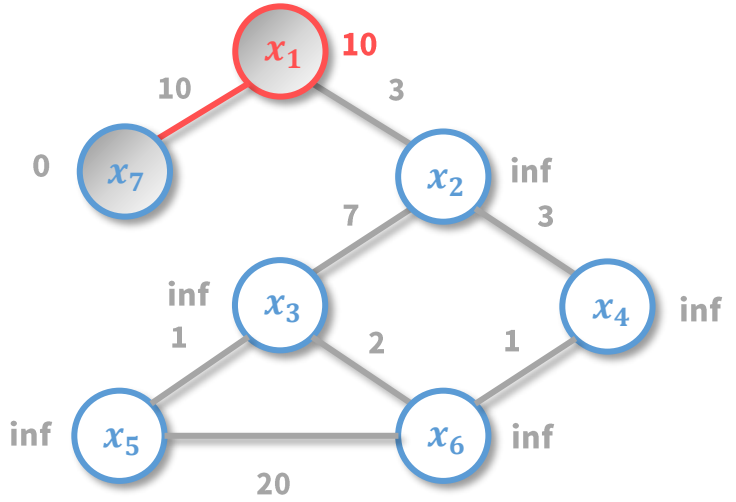
$(x_7: 0)$ $(x_1: 10)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (7)

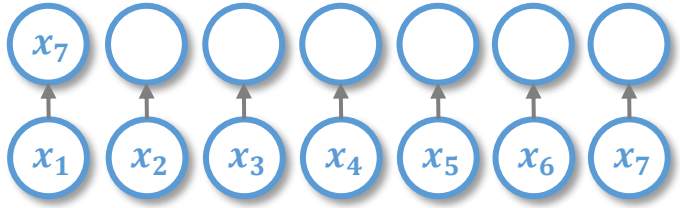
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

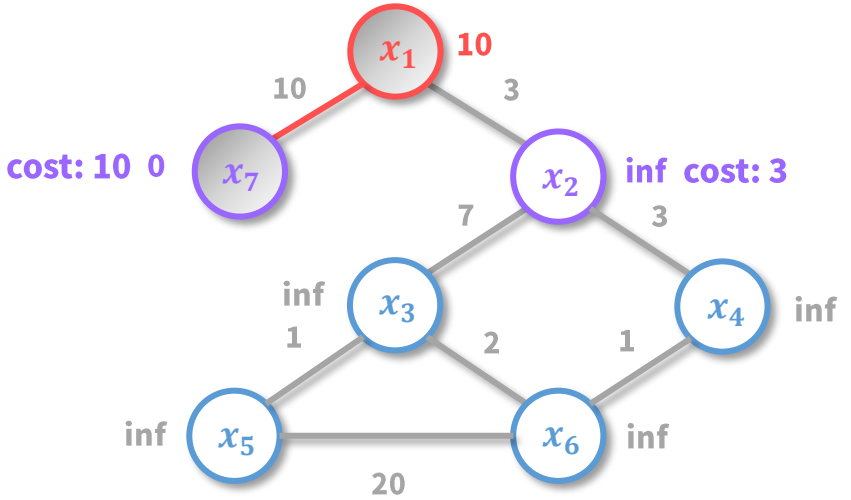
$(x_7: 0)$ $(x_1: 10)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (8)

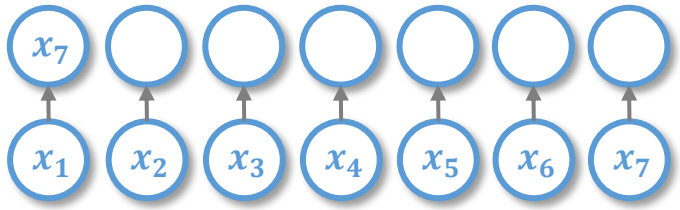
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

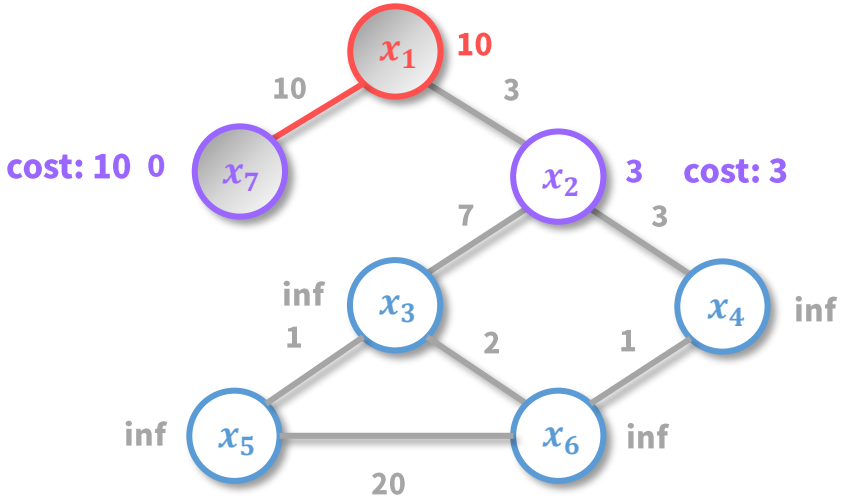
$(x_7: 0)$ $(x_1: 10)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (9)

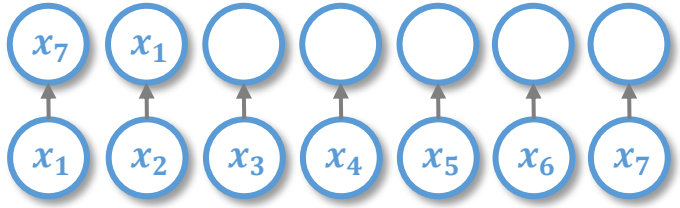
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

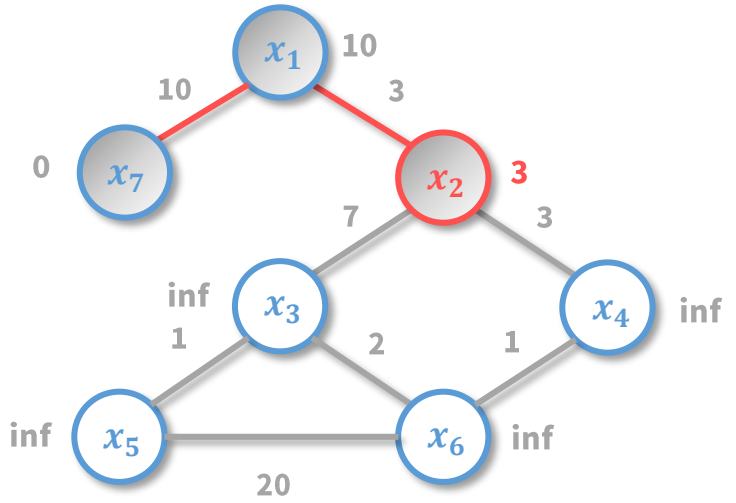
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (10)

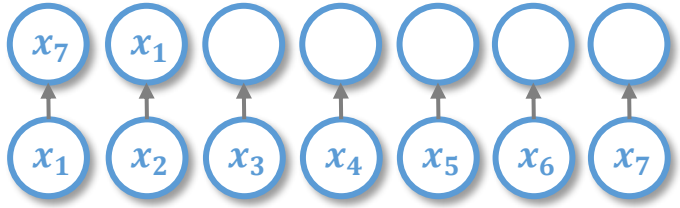
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

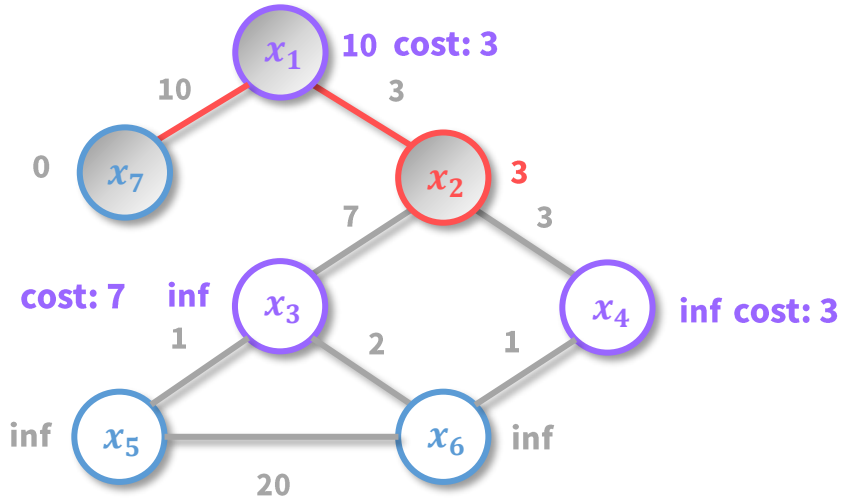
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (11)

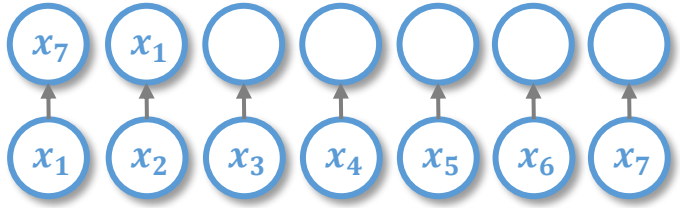
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

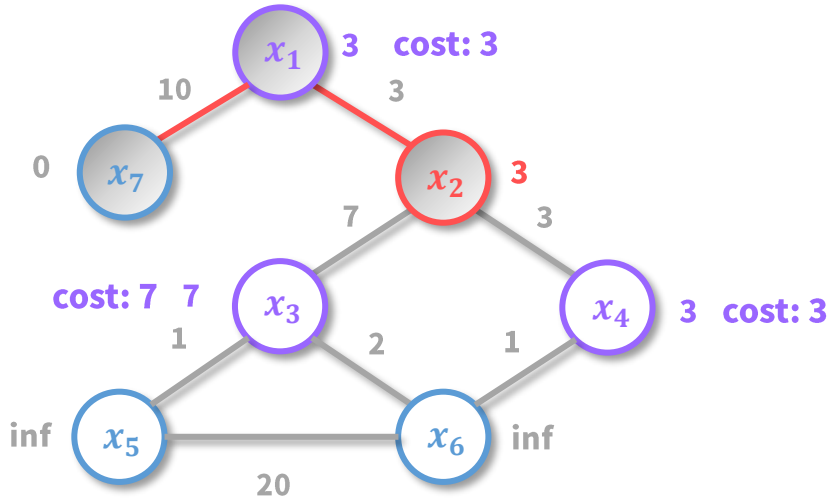
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (12)

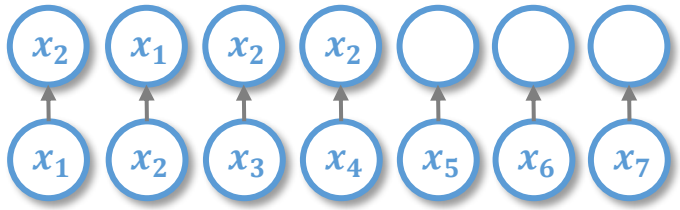
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

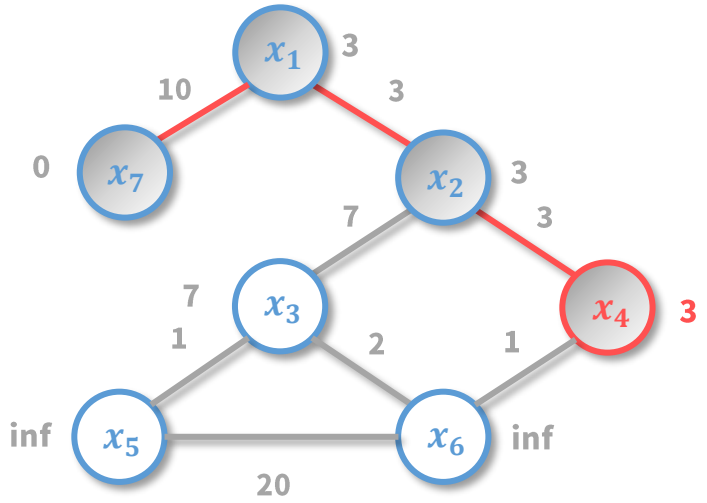
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$ $(x_2: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (14)

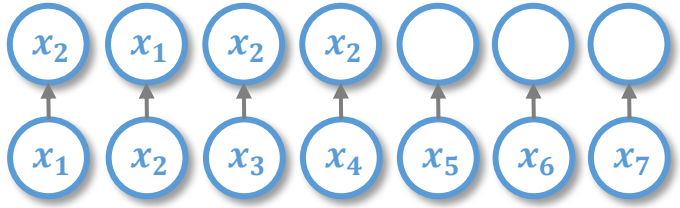
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

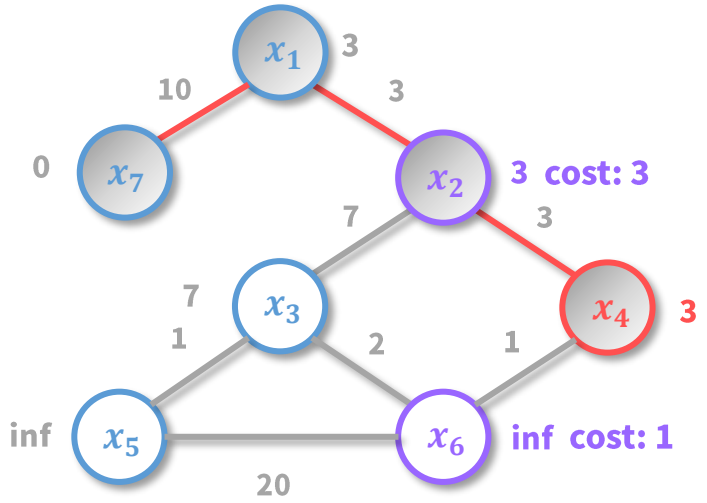
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$ $(x_2: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (13)

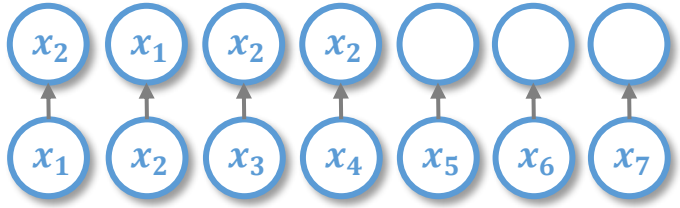
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

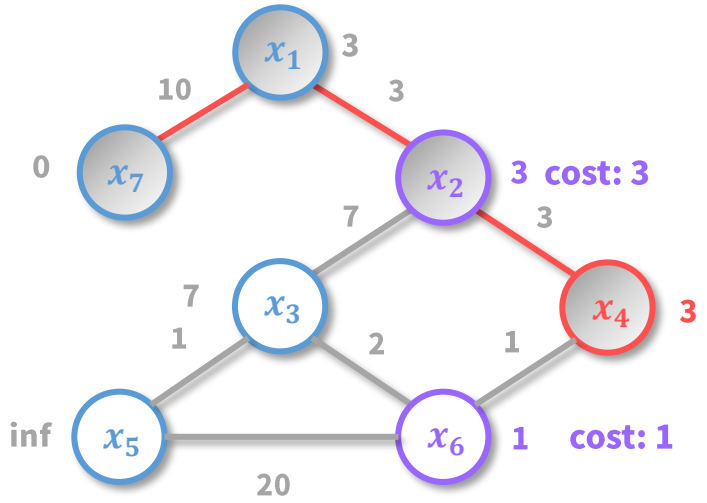
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$ $(x_2: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (14)

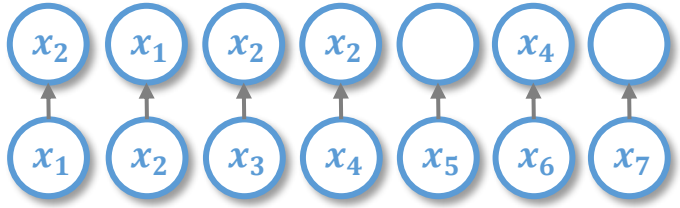
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

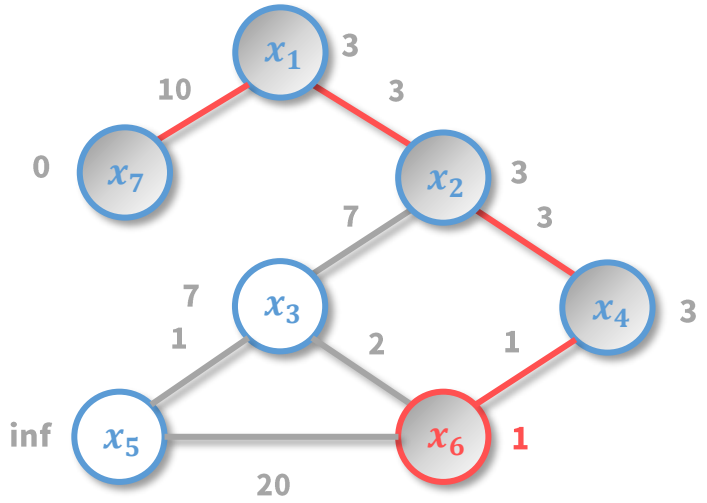
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$
$(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (15)

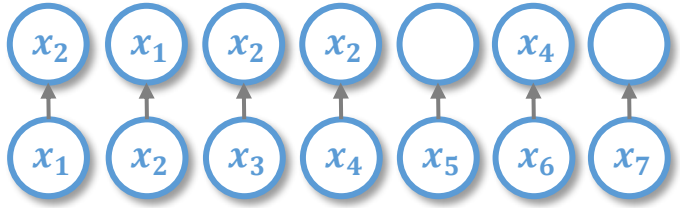
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

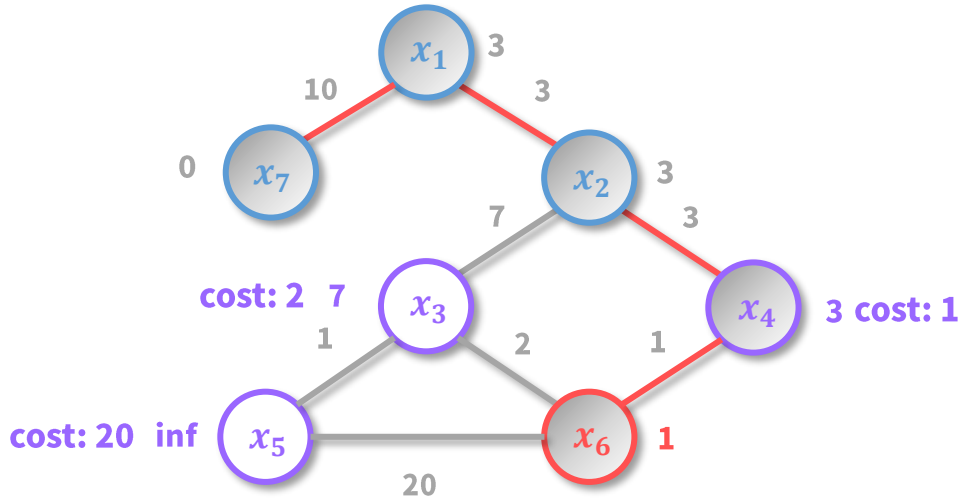
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$
$(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (16)

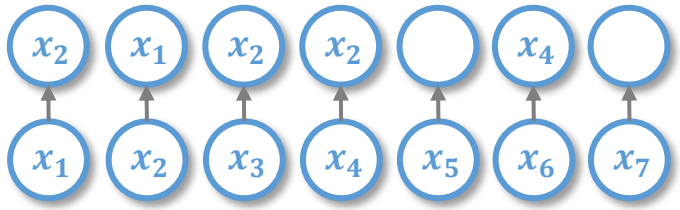
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

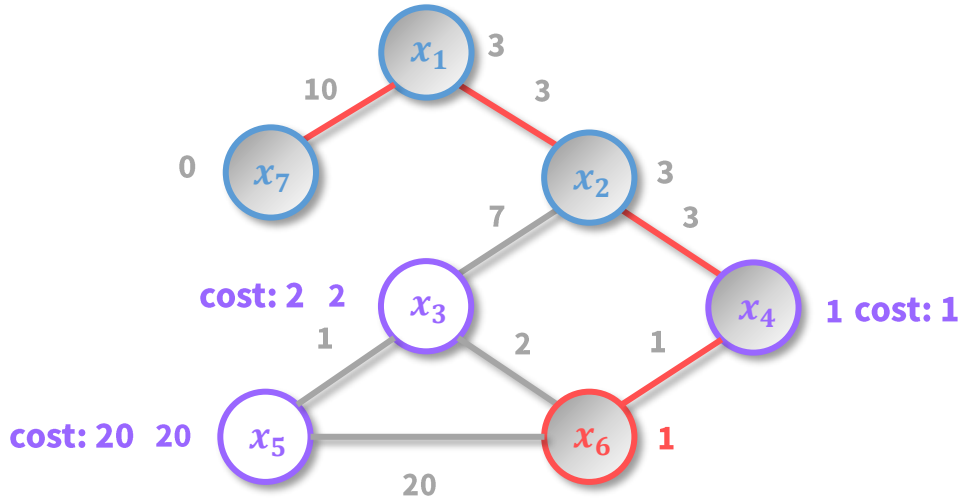
$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_1: 3)$ $(x_3: 7)$ $(x_1: inf)$
$(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (17)

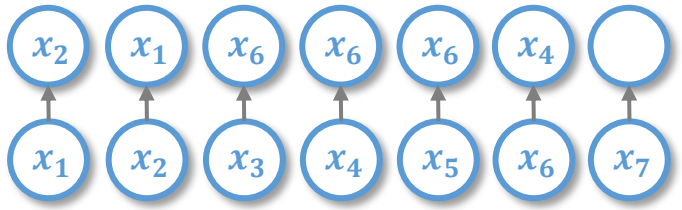
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_4: 1)$ $(x_3: 2)$ $(x_1: 3)$ $(x_3: 7)$

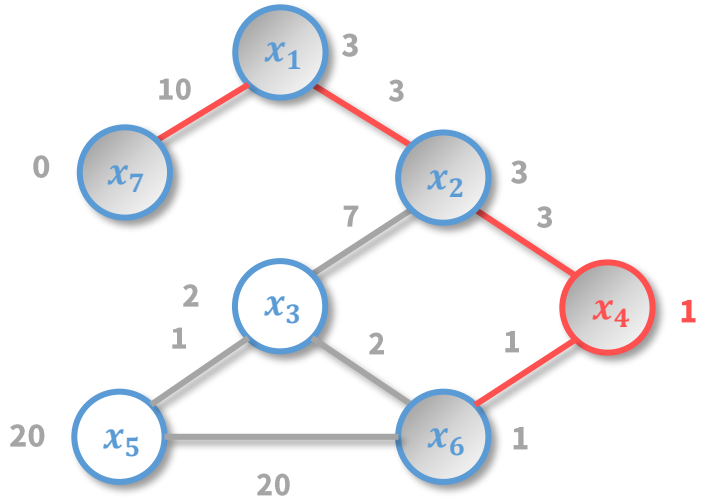
$(x_5: 20)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (18)

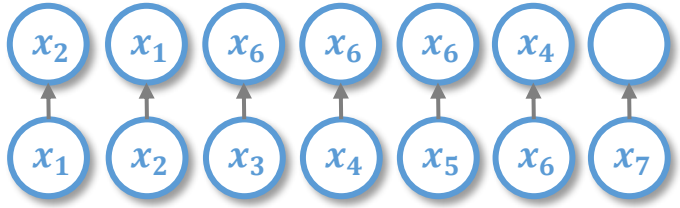
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_4: 1)$ $(x_3: 2)$ $(x_1: 3)$ $(x_3: 7)$

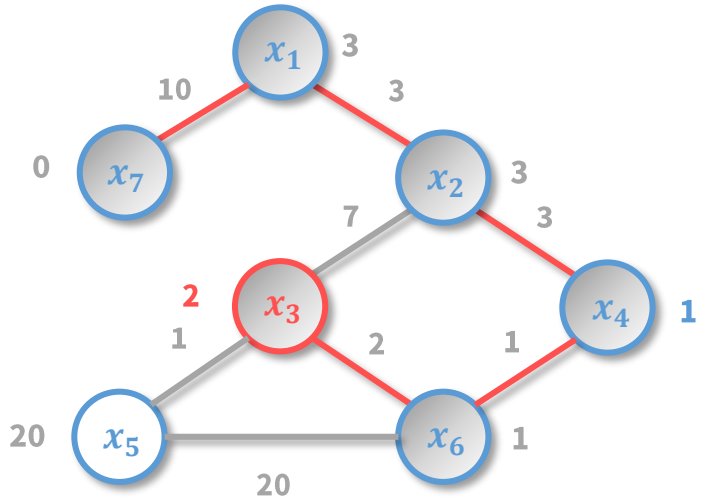
$(x_5: 20)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (19)

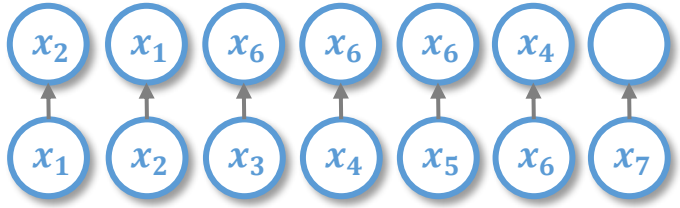
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_4: 1)$ $(x_3: 2)$ $(x_1: 3)$ $(x_3: 7)$

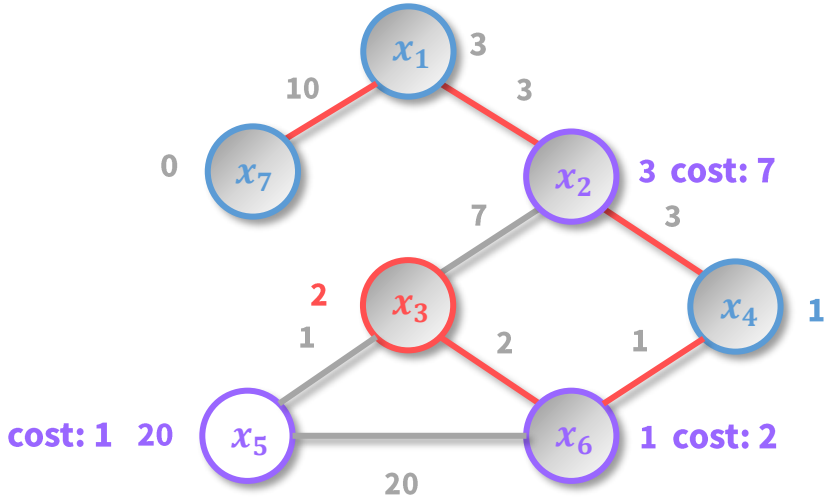
$(x_5: 20)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (20)

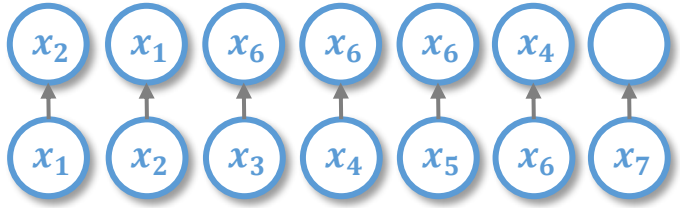
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_7: 0)$ $(x_1: 10)$ $(x_2: 3)$ $(x_4: 3)$ $(x_6: 1)$ $(x_4: 1)$ $(x_3: 2)$ $(x_1: 3)$ $(x_3: 7)$

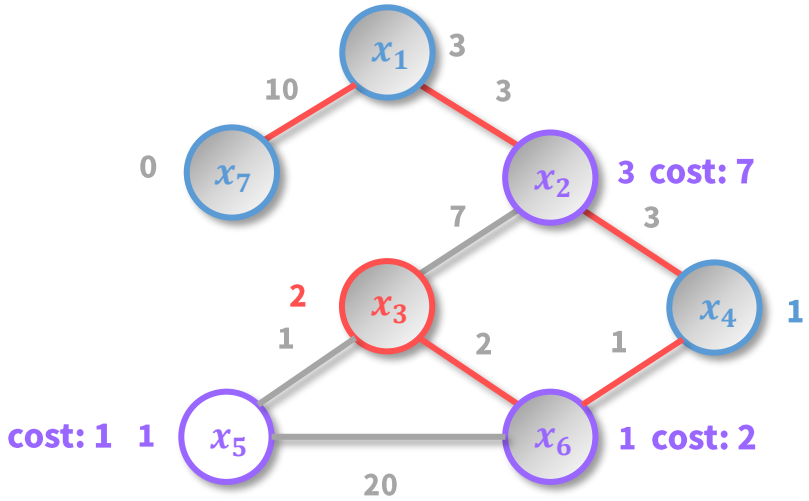
$(x_5: 20)$ $(x_1: inf)$ $(x_2: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$

Prim's Algorithm (20)

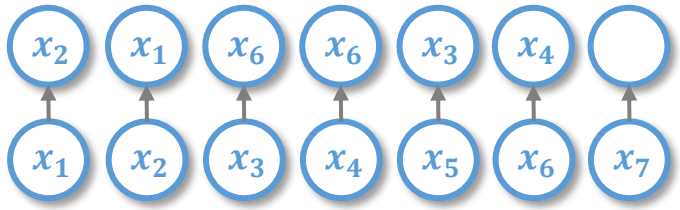
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

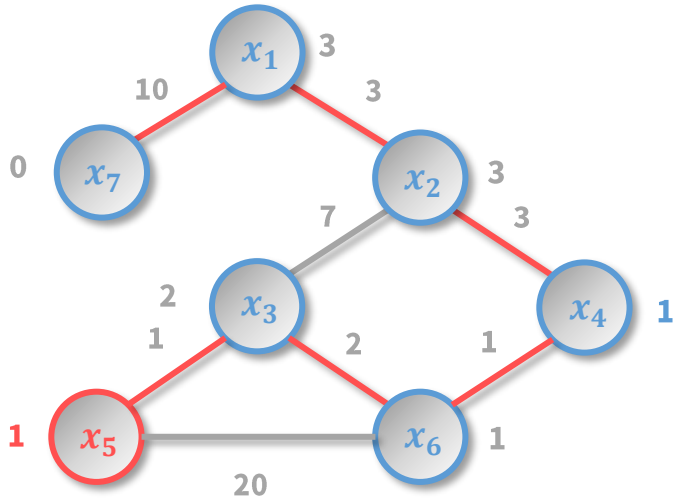
$(x_5: 1)\ (x_1: 3)\ (x_3: 7)\ (x_5: 20)\ (x_1: inf)\ (x_2: inf)\ (x_3: inf)$
$(x_4: inf)\ (x_5: inf)\ (x_6: inf)$

Prim's Algorithm (21)

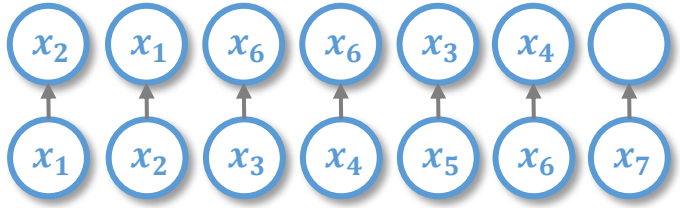
Pseudo code

```
def prim_mst(graph: Graph) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[node] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_5: 1)\ (x_1: 3)\ (x_3: 7)\ (x_5: 20)\ (x_1: inf)\ (x_2: inf)\ (x_3: inf)$
$(x_4: inf)\ (x_5: inf)\ (x_6: inf)$

Summary

Worst Time Complexity

$* \ n = |N|, m = |E|$

Kruskals Algorithm	Prims Algorithm
$O(m\log m)$	$O((n+m)\log n)$
1. build heap: $O(m)$	1. build heap, assign weights, parents: $O(n)$
2. pop edge: $O(\log m)$	2. pop node: $O(\log n)$
3. check connected component: $O(\log^* n)$	3. number of pop node called: $O(n)$
4. union connected component: $O(\log^* n)$	4. check neighbors: $O(2m) = O(m)$
5. number of pop edge called: $O(m)$	5. insert neighbors into heap: $O(\log n)$
6. time complexity: $O(m + m\log m + m\log^* n) = O(m\log m)$	6. time complexity: $O(n + n\log n + m\log n) = O((n+m)\log n)$

Prims algorithm can be further optimized with Fibonacci heap

Graph Algorithms – Shortest Path

Shortest Path (1)

- Goal
  - Given a node, find a set of edges such that all the paths to other nodes are the shortest path.

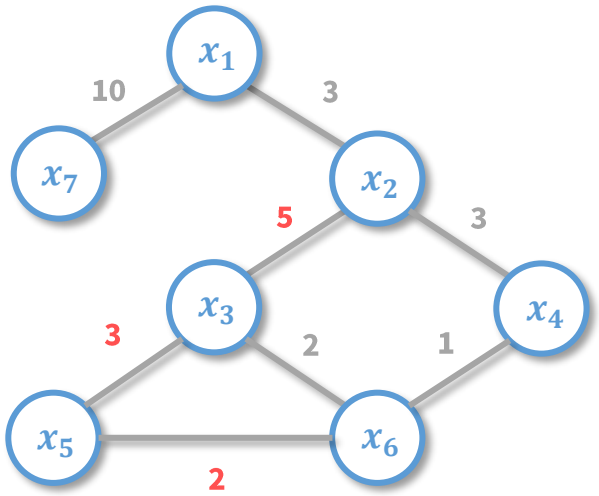
- Applications
  - Centrality of graph.
  - Schedule planning.
  - Route planning.
  - Social network analysis.
  - Robotics.

Dijkstra's Algorithm (1)

Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



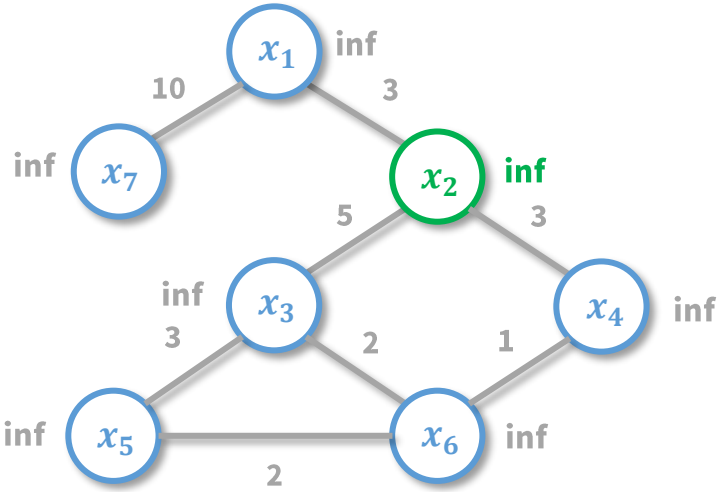
Note that the weight here is different from the one used in Prims algorithm to showcase some properties of Dijkstras algorithm

Dijkstra's Algorithm (2)

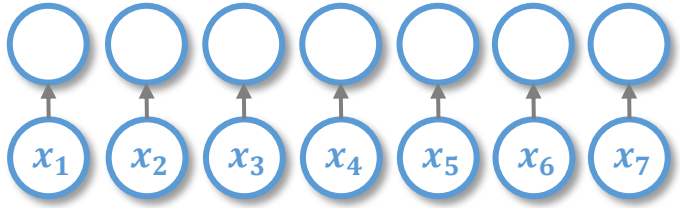
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent

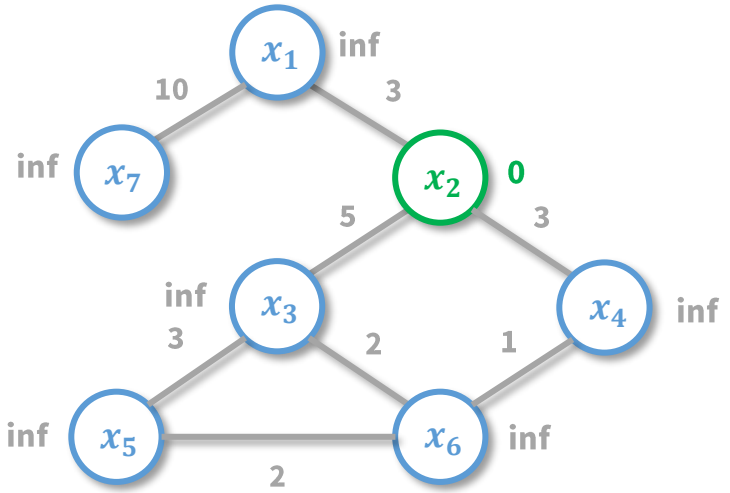


Dijkstra's Algorithm (3)

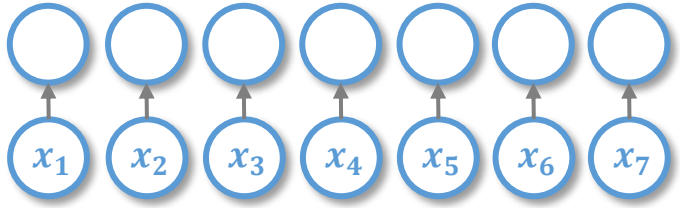
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

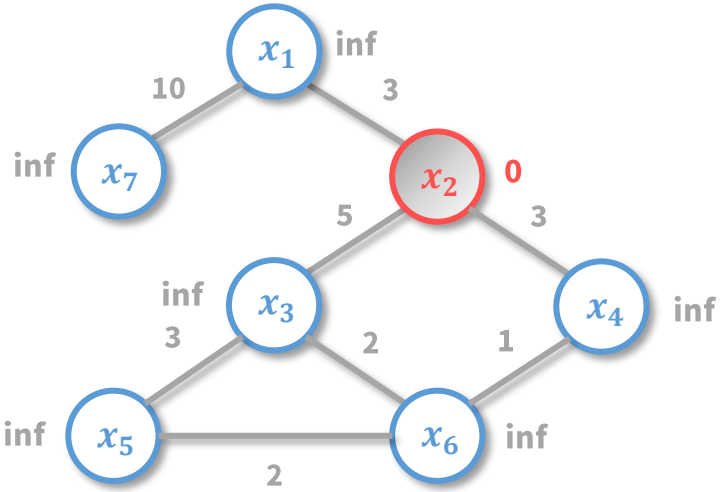
$(x_2: 0)\ (x_1: inf)\ (x_3: inf)\ (x_4: inf)\ (x_5: inf)\ (x_6: inf)\ (x_7: inf)$

Dijkstra's Algorithm (4)

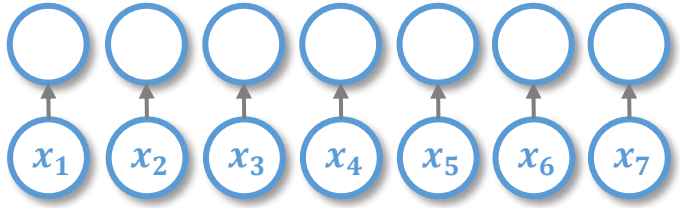
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

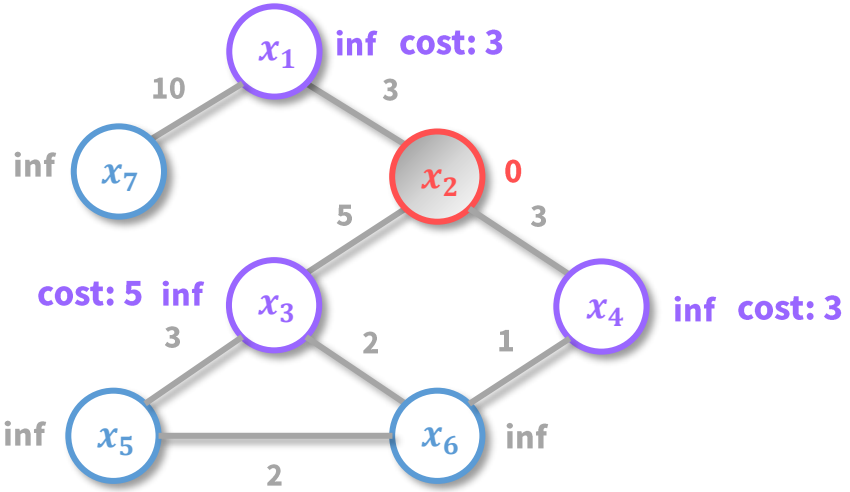
$(x_2: 0)$ $(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (5)

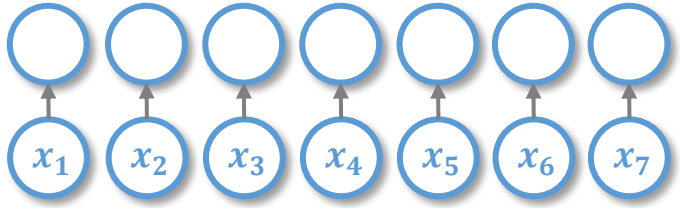
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

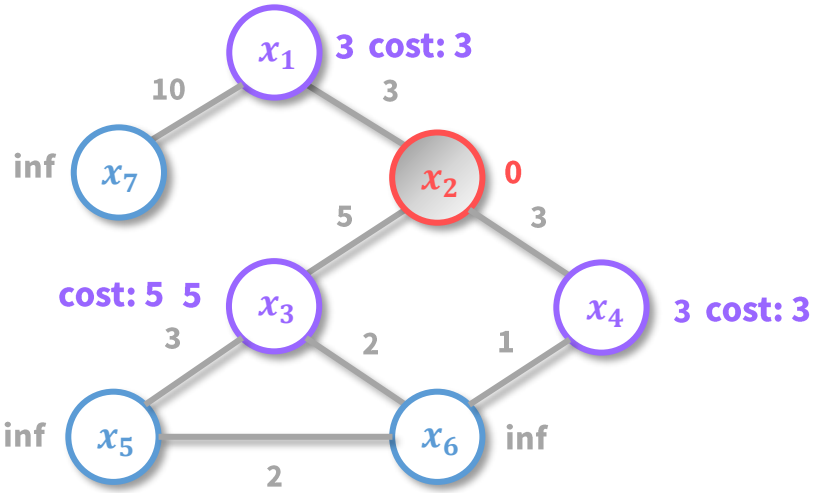
$(x_2: 0)$ $(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (6)

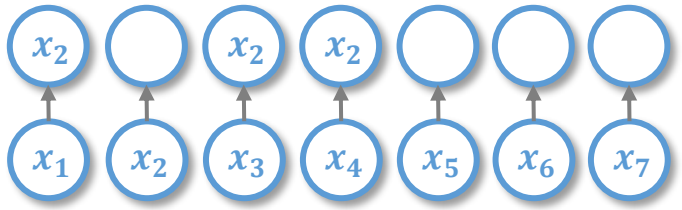
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

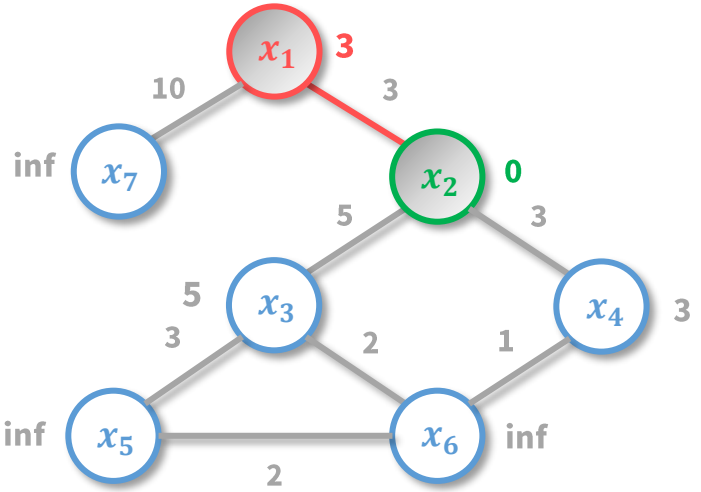
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (7)

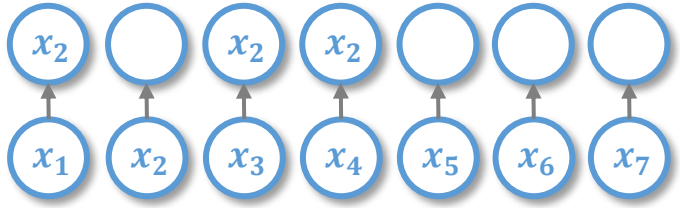
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

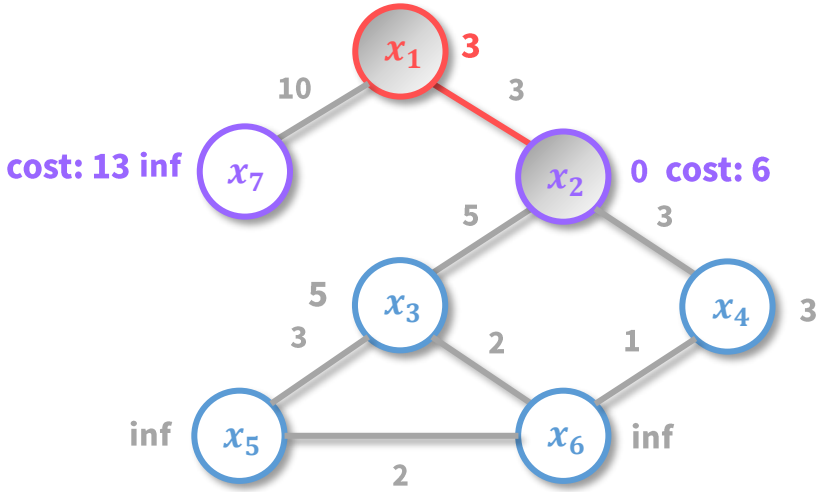
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (8)

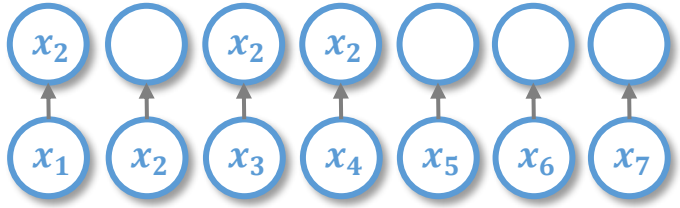
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

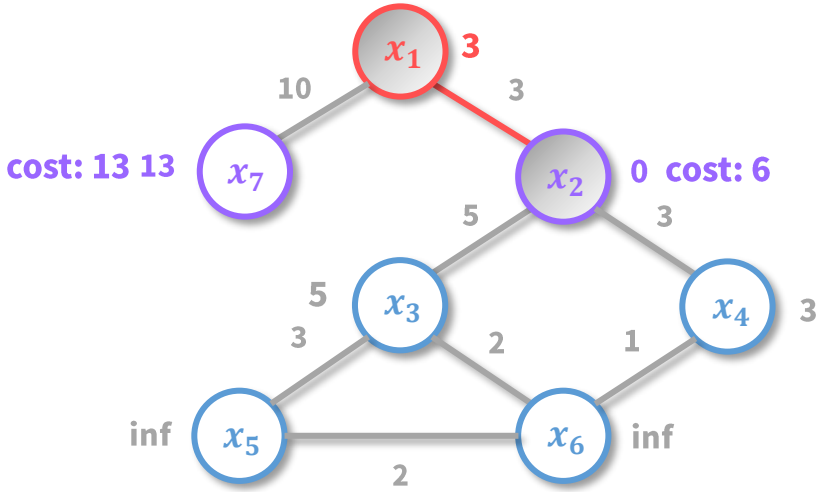
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$
$(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (9)

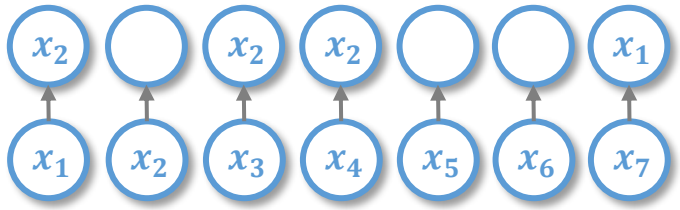
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

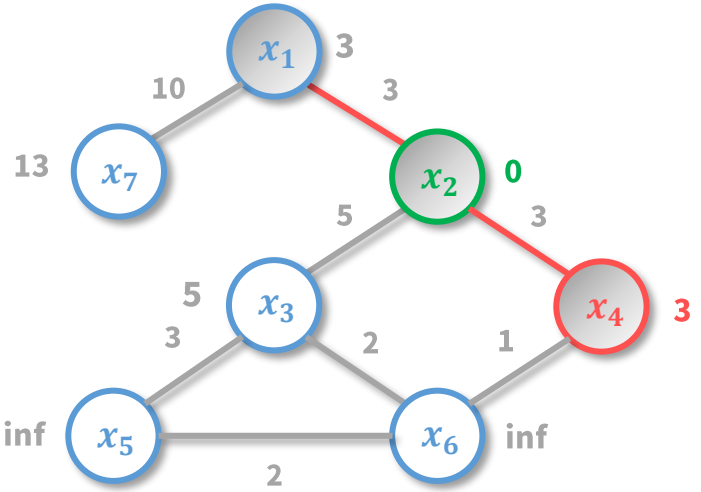
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$ $(x_3: inf)$
$(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (11)

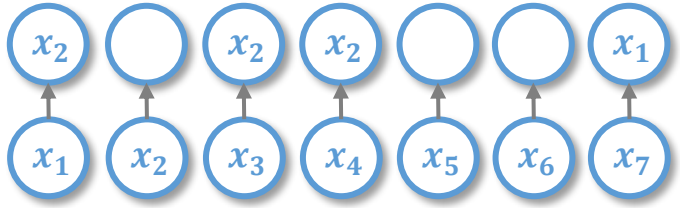
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

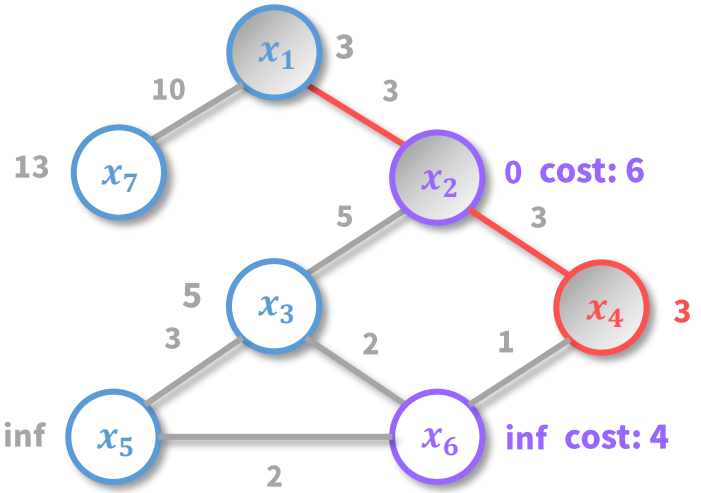
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$ $(x_3: inf)$
$(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (12)

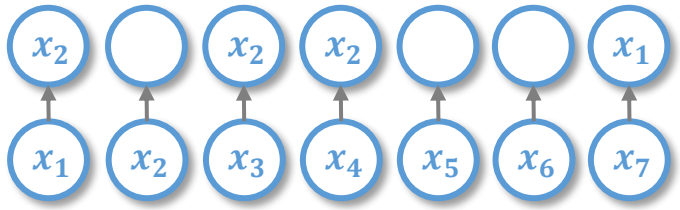
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

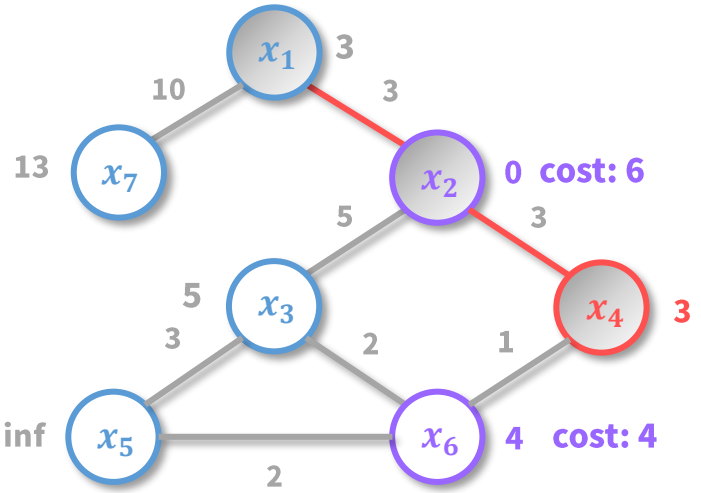
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$ $(x_3: inf)$
$(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (13)

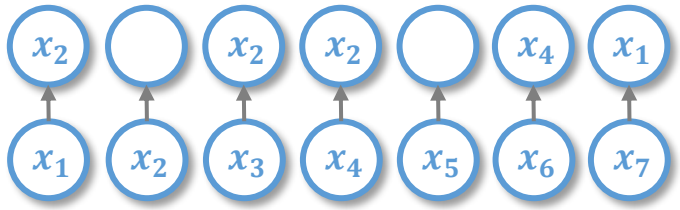
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

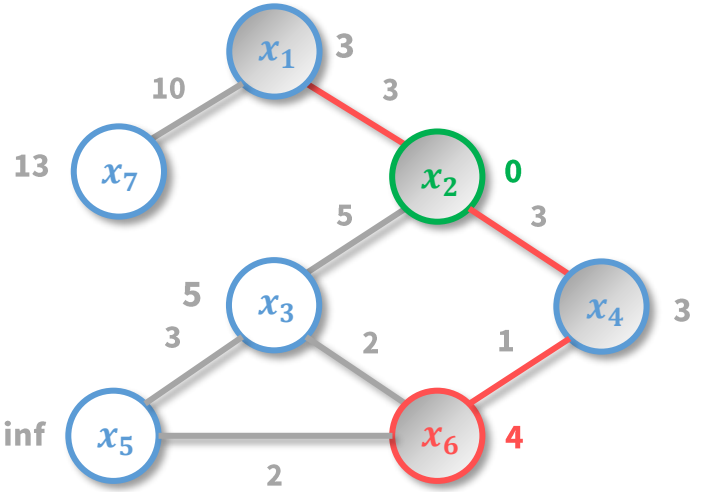
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (14)

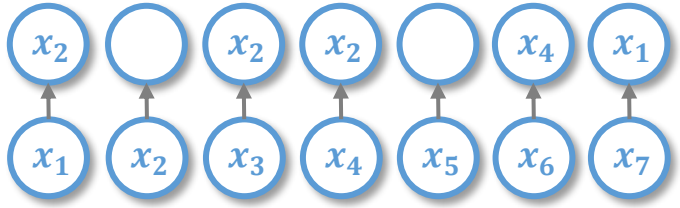
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

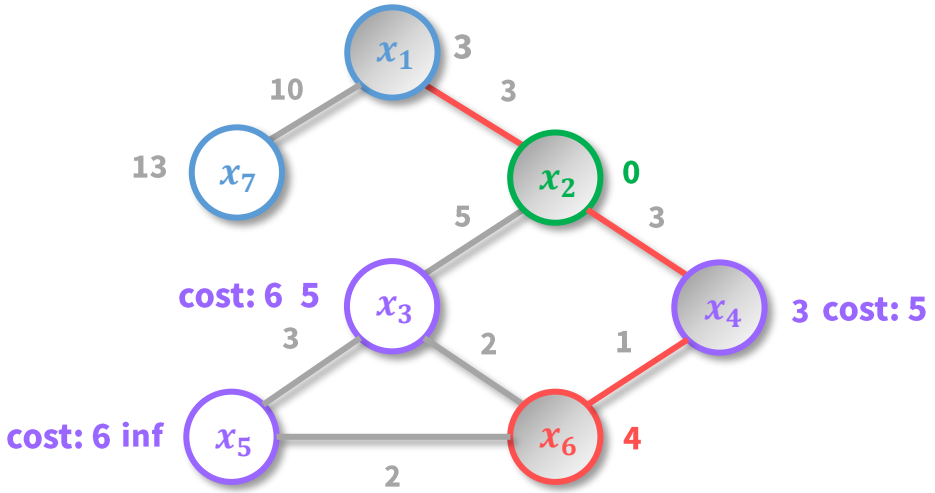
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (15)

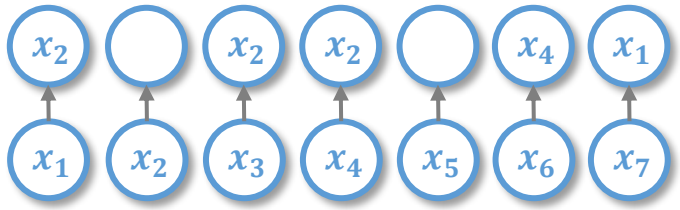
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

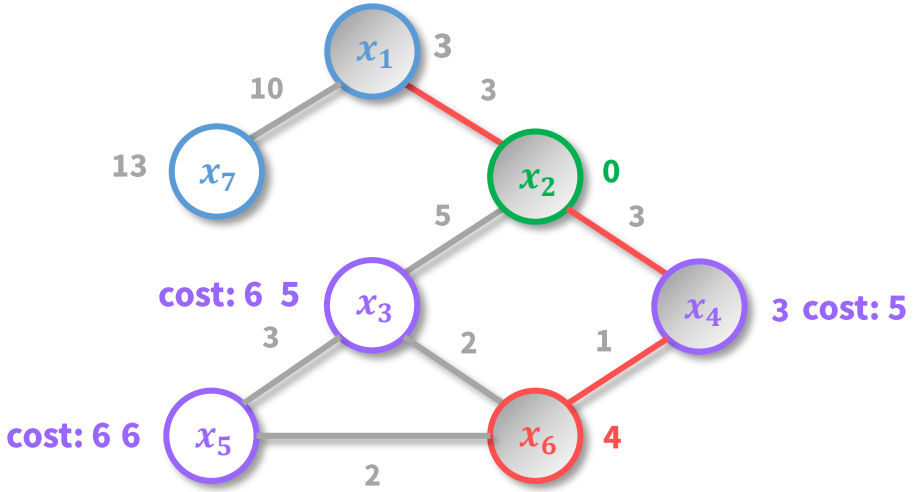
$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_7: 13)$ $(x_1: inf)$
$(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (16)

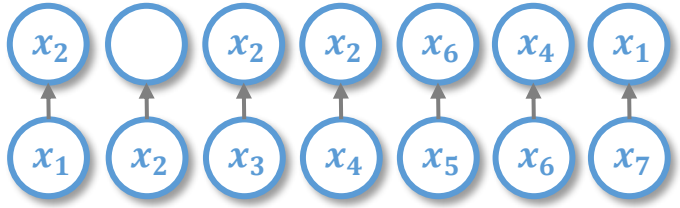
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_5: 6)$ $(x_7: 13)$

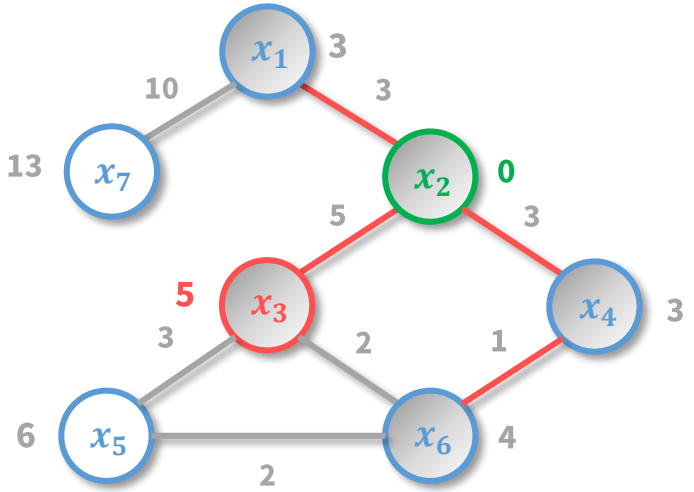
$(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (17)

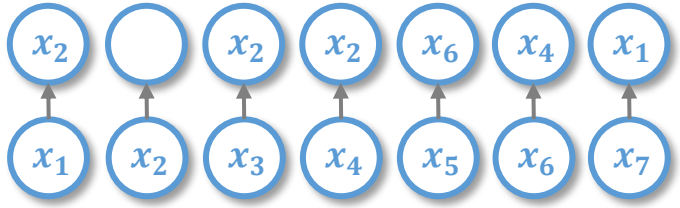
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_5: 6)$ $(x_7: 13)$

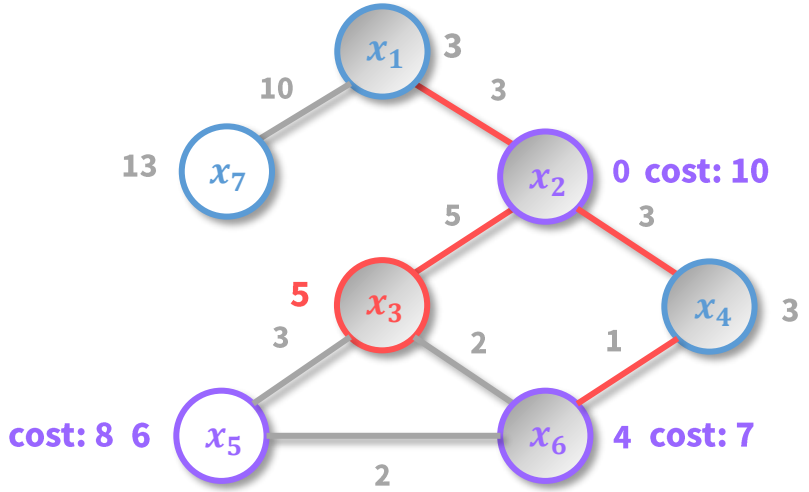
$(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (18)

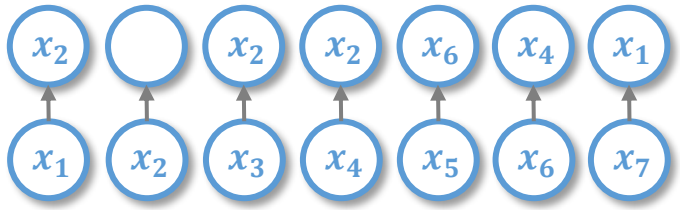
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_5: 6)$ $(x_7: 13)$

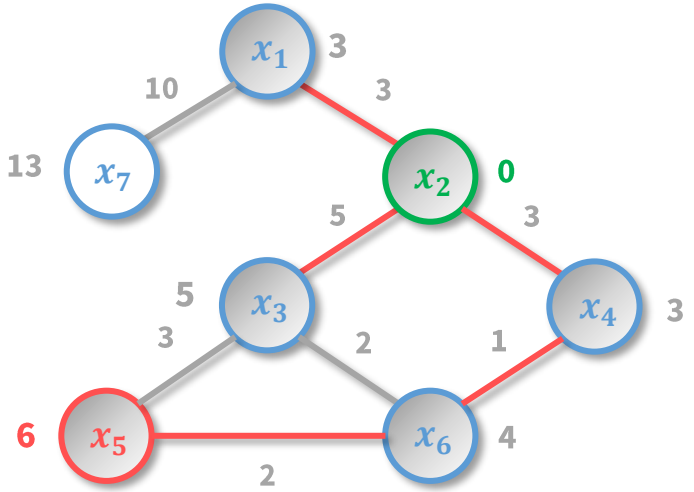
$(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (19)

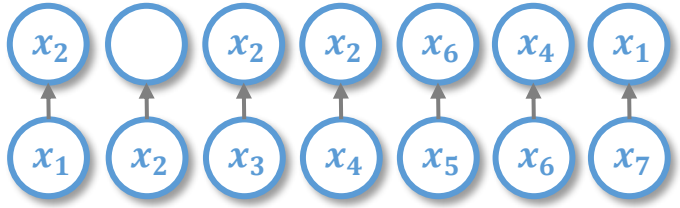
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_5: 6)$ $(x_7: 13)$

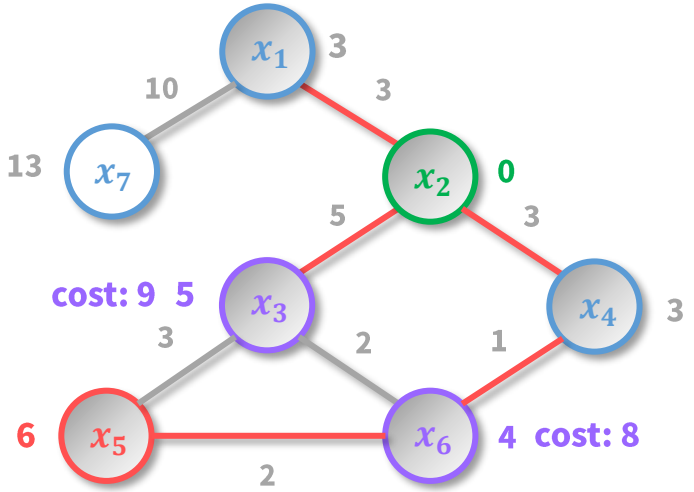
$(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (20)

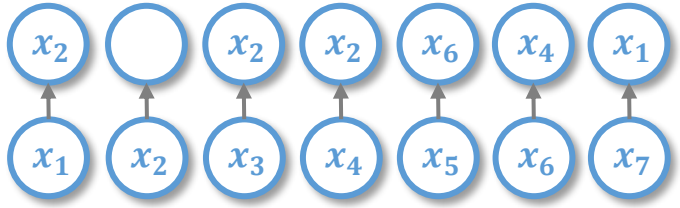
Pseudo code

```python
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)$ $(x_1: 3)$ $(x_4: 3)$ $(x_6: 4)$ $(x_3: 5)$ $(x_5: 6)$ $(x_7: 13)$

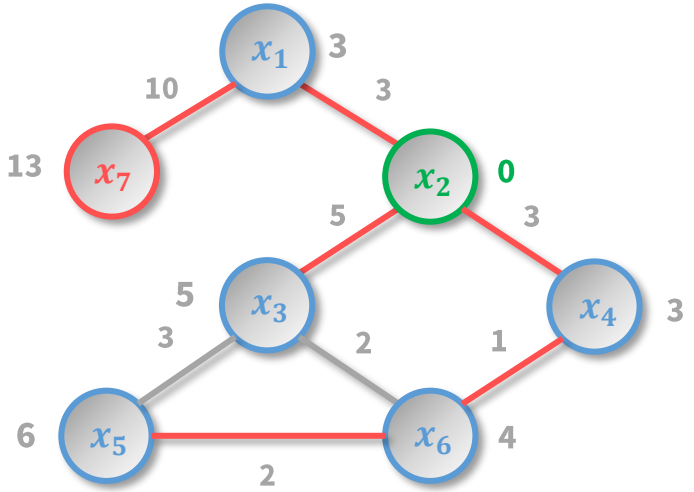
$(x_1: inf)$ $(x_3: inf)$ $(x_4: inf)$ $(x_5: inf)$ $(x_6: inf)$ $(x_7: inf)$

Dijkstra's Algorithm (21)

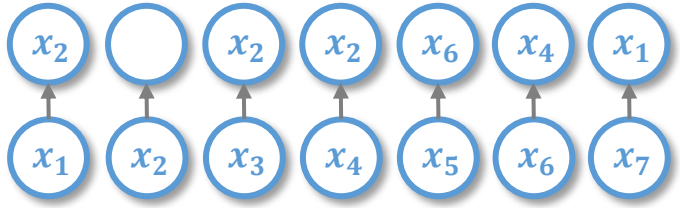
Pseudo code

```
def dijkstra_shortest_path(graph: Graph, origin: Node) -> List[Edge]:
  nodes = graph.nodes
  n_nodes = len(nodes)
  inf = sys.maxsize
  node_weights = dict()
  node_parent = dict()
  for node in graph.nodes:
    node_weights[node] = inf
    node_parent[node] = None
  node_weights[origin] = 0
  weight_heap = MinHeap.build_heap(node_weights)
  visited_nodes = set()
  result = []
  while len(visited_nodes) != n_nodes:
    node = weight_heap.pop()
    if node in visited_nodes:
      continue
    visited_nodes.add(node)
    weight = node_weights[node]
    result.append(Edge(node, node_parent[node]))
    for edge in graph.edges(node):
      node_other = edge.node_b if edge.node_a == node else edge.node_a
      cost = edge.weight + weight
      if cost < node_weights[node_other]:
        node_weights[node_other] = cost
        node_parent[node_other] = node
        weight_heap.insert(node_other)

  return result
```

Graph Representation



Node Parent



MinHeap

$(x_2: 0)\ (x_1: 3)\ (x_4: 3)\ (x_6: 4)\ (x_3: 5)\ (x_5: 6)\ (x_7: 13)$

$(x_1: inf)\ (x_3: inf)\ (x_4: inf)\ (x_5: inf)\ (x_6: inf)\ (x_7: inf)$