Data Structures and Algorithms
Union Find and Hash Table

2022-07-15

Ping-Han Hsieh

Overview

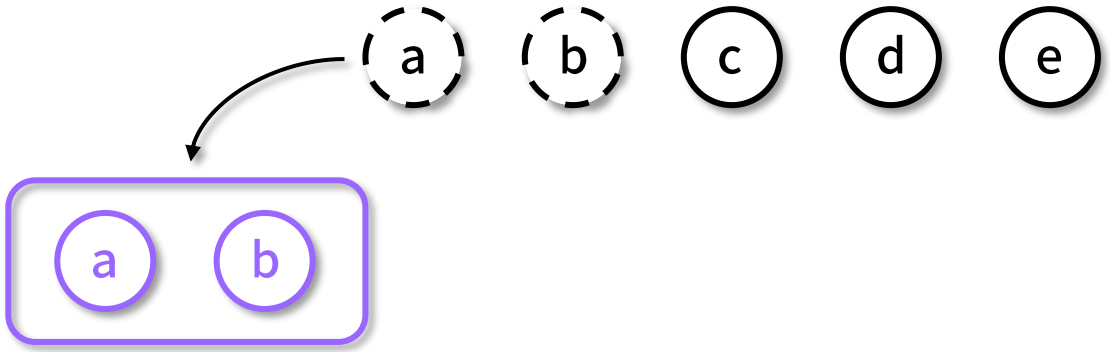
- Data Structures:
  - Linked-List, Array
  - Stack, Queue
  - Union Find, Hash Table
  - Binary Search Tree, Heap
  - Graph

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

Union Find (Disjoint Set)

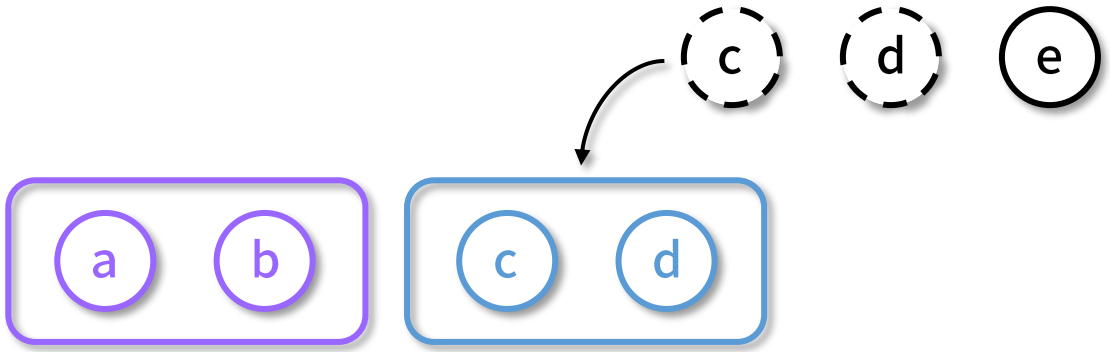
Union Find (1)

Union Find (2)



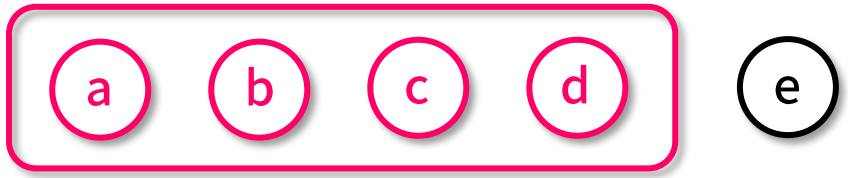
1. Put a, b in the same disjoint set

Union Find (3)



2. Put c, d in the same disjoint set

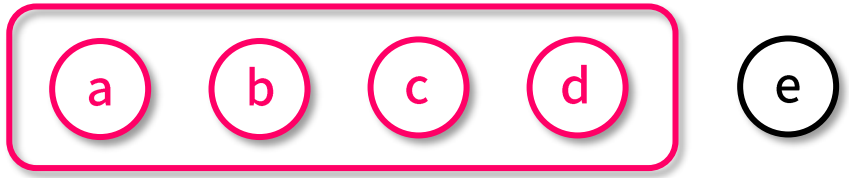
Union Find (4)



3. Union disjoint set with a and disjoint set with c

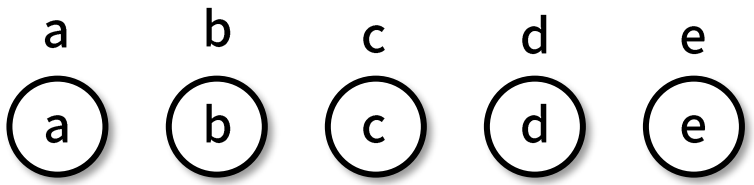
Union Find (5)

- Union find define disjoint set of fixed number of objects.
- One can use union to merge two disjoint sets.
- One can use find to see which disjoint set one object belongs to.
- Useful if we want to understand connected components in a more complicated data structure (e.g. disjoint graph in a graph).



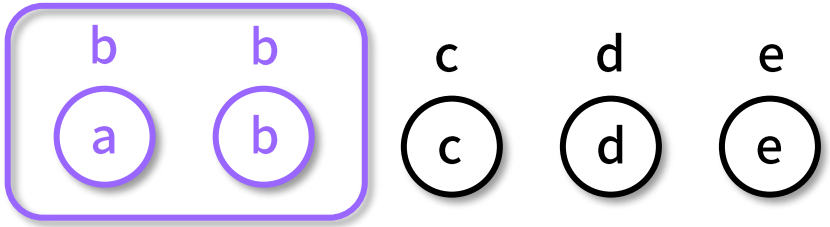
Quick Find UF (1)

- Store the label for the connected component for each object:



Quick Find UF (2)

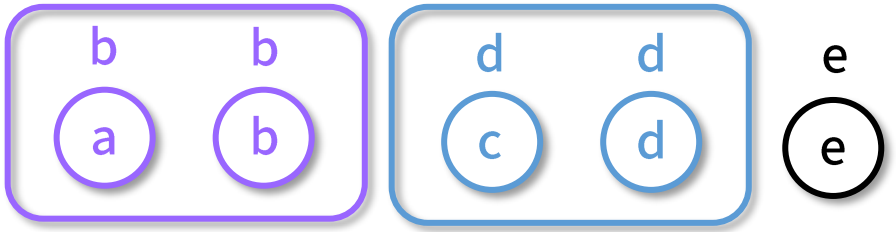
- Store the label for the connected component for each object
- Update the label when union two disjoint set



1. Put a, b in the same disjoint set

Quick Find UF (3)

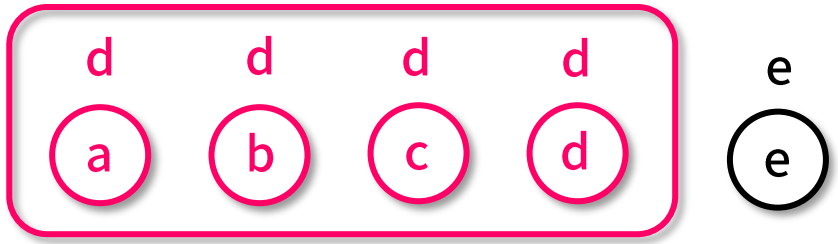
- Store the label for the connected component for each object
- Update the label when union two disjoint set



2. Put c, d in the same disjoint set

Quick Find UF (4)

- Store the label for the connected component for each object:



3. Union disjoint set with a and disjoint set with c

To know if two objects belong to the same connected component, simply check their label ID.

Quick Find UF (5)

Initialization

```
class QuickFindUF:
  def __init__(self, n_nodes):
    self.n_components: int = n_nodes
    self.n_nodes: int = n_nodes
    self.component_id: Array = Array(n_nodes)
    for i in range(n_nodes):
      self.component_id[i] = i

  def __str__(self) -> str: ···

  def __len__(self) -> int: ···

  def validate_index(self, index) -> None: ···
      check if the index is valid
  def is_connected(self, index_a, index_b) -> bool: ···
      check if two indices is in the same disjoint set
  def union(self, index_a, index_b) -> None: ···
      merge the disjoint set where a and b belong to
  def find(self, index) -> int: ···
      find the disjoint label of the given index
```

Command

```
union_find = QuickFindUF(5)
```

Visualization

n_components = 5

n_nodes = 5

component_id

0	1	2	3	4
0	1	2	3	4

Quick Find UF

Union (1)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

component_id

0	1	2	3	4
0	1	2	3	4

index_a = 0

index_b = 1

component_id_a = 0

component_id_b = 1

Quick Find UF

Union (2)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

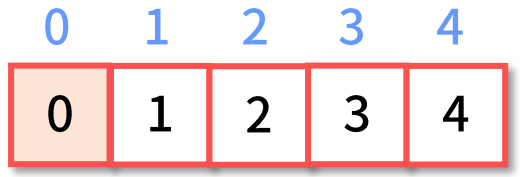
```
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

component_id



index_a = 0

index_b = 1

component_id_a = 0

component_id_b = 1

Quick Find UF

Union (3)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

component_id

0	1	2	3	4
1	1	2	3	4

index_a = 0

index_b = 1

component_id_a = 0

component_id_b = 1

Quick Find UF

Union (4)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 1)
```

Visualization

n_components = 4

n_nodes = 5

component_id

0	1	2	3	4
1	1	2	3	4

index_a = 0

index_b = 1

component_id_a = 0

component_id_b = 1

Quick Find UF

Union (5)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

n_components = 4

n_nodes = 5

component_id

0	1	2	3	4
1	1	2	3	4

index_a = 2

index_b = 3

component_id_a = 2

component_id_b = 3

Quick Find UF

Union (6)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

n_components = 4

n_nodes = 5

component_id

0	1	2	3	4
1	1	2	3	4

index_a = 2

index_b = 3

component_id_a = 2

component_id_b = 3

Quick Find UF

Union (7)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

n_components = 4

n_nodes = 5

component_id

0	1	2	3	4
1	1	3	3	4

index_a = 2

index_b = 3

component_id_a = 2

component_id_b = 3

Quick Find UF

Union (8)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

n_components = 3

n_nodes = 5

component_id

0	1	2	3	4
1	1	3	3	4

index_a = 2

index_b = 3

component_id_a = 2

component_id_b = 3

Quick Find UF

Union (9)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    component_id_a = self.component_id[index_a]
    component_id_b = self.component_id[index_b]
    if component_id_a == component_id_b:
        return
    for i in range(self.n_nodes):
        if self.component_id[i] == component_id_a:
            self.component_id[i] = component_id_b
    self.n_components -= 1
    return
```

Command

```
union_find.union(0, 2)
```

Visualization

n_components = 3

n_nodes = 5

component_id

0	1	2	3	4
1	1	3	3	4

index_a = 0

index_b = 2

component_id_a = 1

component_id_b = 3

Quick Find UF

Union (10)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 2)
```

Visualization

n_components = 3

n_nodes = 5

component_id

0	1	2	3	4
1	1	3	3	4

index_a = 0

index_b = 2

component_id_a = 1

component_id_b = 3

Quick Find UF

Union (11)

Union

```python
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 2)
```

Visualization

n_components = 3

n_nodes = 5

component_id

0	1	2	3	4
3	3	3	3	4

index_a = 0

index_b = 2

component_id_a = 1

component_id_b = 3

Quick Find UF

Union (12)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  component_id_a = self.component_id[index_a]
  component_id_b = self.component_id[index_b]
  if component_id_a == component_id_b:
    return
  for i in range(self.n_nodes):
    if self.component_id[i] == component_id_a:
      self.component_id[i] = component_id_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 2)
```

Visualization

n_components = 2

n_nodes = 5

component_id

0	1	2	3	4
3	3	3	3	4

index_a = 0

index_b = 2

component_id_a = 1

component_id_b = 3

Quick Find UF

Find

Find

```python
def find(self, index) -> int:
    self.validate_index(index)
    return self.component_id[index]
```

Command

```
union_find.find(0)
```

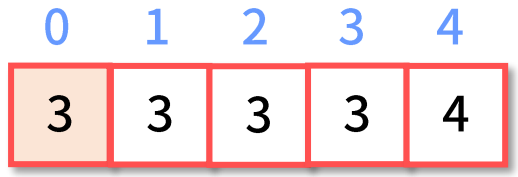
Visualization

n_components = 2

n_nodes = 5

component_id

index = 0



Quick Find UF

Connected

Check Connectivity

```python
def is_connected(self, index_a, index_b) -> bool:
    self.validate_index(index_a)
    self.validate_index(index_b)
    component_id_a = self.component_id[index_a]
    component_id_b = self.component_id[index_b]
    return component_id_a == component_id_b
```

Command

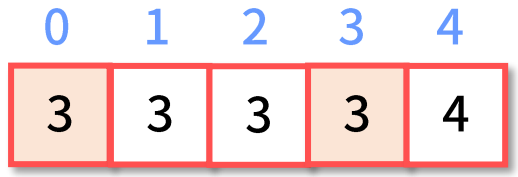
```python
union.is_connected(0, 3)
```

Visualization

n_components = 2

n_nodes = 5

component_id



index_a = 0

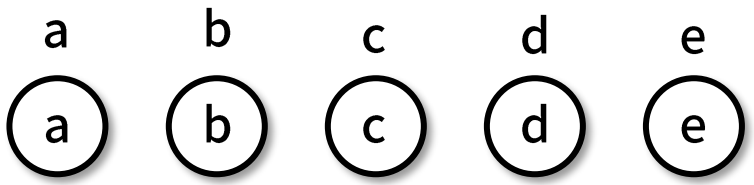
index_b = 3

component_id_a = 3

component_id_b = 3

Quick Union UF (1)

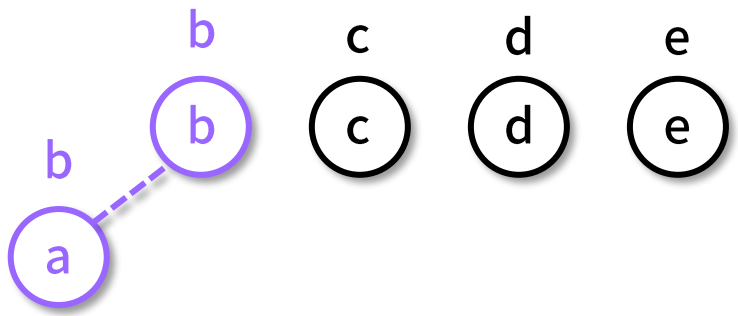
- Store the label for the connected component for each object:



An object is the root of a disjoint set if the label equals to the object value.

Quick Union UF (2)

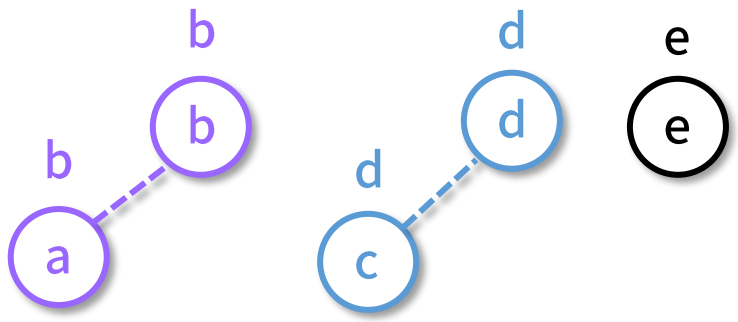
- Store the label for the connected component for each object
- Update the label when union two disjoint set (only update the root of disjoint set)



1. Put a, b in the same disjoint set

Quick Union UF (3)

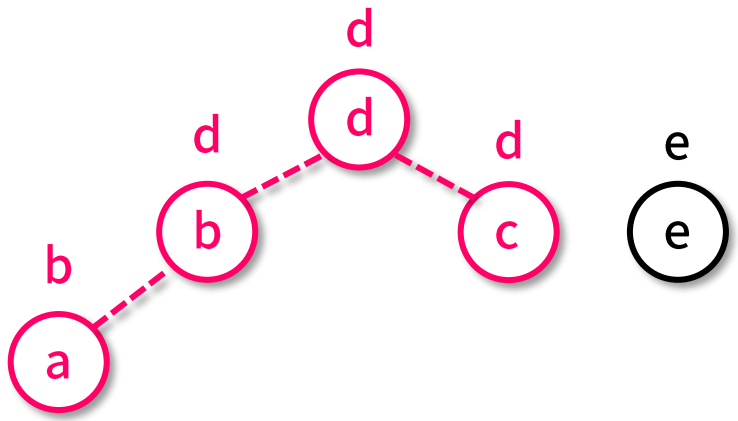
- Store the label for the connected component for each object
- Update the label when union two disjoint set (only update the root of disjoint set)



1. Put c, d in the same disjoint set

Quick Union UF (4)

- Store the label for the connected component for each object
- Update the label when union two disjoint set (only update the root of disjoint set)



1. Put a, c in the same disjoint set

To know if two objects belong to the same connected component, simply check their root label ID.

Quick Union UF (5)

Initialization

```
class QuickUnionUF:
  def __init__(self, n_nodes):
    self.n_components: int = n_nodes
    self.n_nodes: int = n_nodes
    self.parent_id: Array = Array(n_nodes)
    for i in range(n_nodes):
      self.parent_id[i] = i

  def __str__(self) -> str: ···

  def __len__(self) -> int: ···

  def validate_index(self, index) -> None: ···
      check if the index is valid
  def is_connected(self, index_a, index_b) -> bool: ···
      check if two indices is in the same disjoint set
  def union(self, index_a, index_b) -> None: ···
      merge the disjoint set where a and b belong to
  def find(self, index) -> int: ···
      find the disjoint label of the given index
```

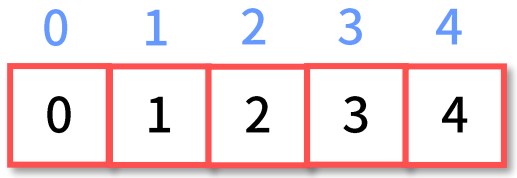
Command

```
union_find = QuickUnionUF(5)
```

Visualization

n_components = 5

n_nodes = 5



Quick Find UF

Union (1)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    self.parent_id[root_a] = root_b
    self.n_components -= 1
    return
```

Command

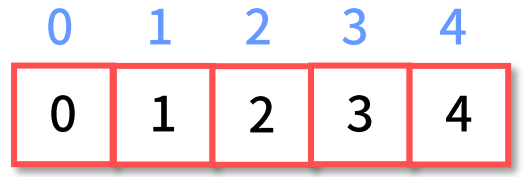
```python
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

parent_id



index_a = 0

index_b = 1

root_a = 0

root_b = 1

Quick Find UF

Union (2)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    self.parent_id[root_a] = root_b
    self.n_components -= 1
    return
```

Command

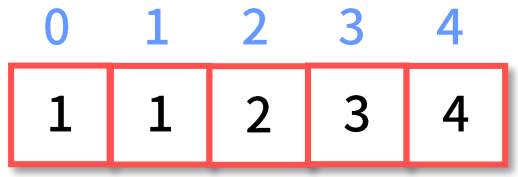
```
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

parent_id



index_a = 0

index_b = 1

root_a = 0

root_b = 1

Quick Find UF

Union (3)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    self.parent_id[root_a] = root_b
    self.n_components -= 1
    return
```

Command

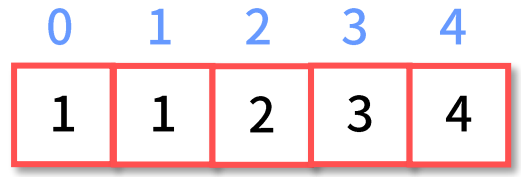
```
union_find.union(0, 1)
```

Visualization

n_components = 4

n_nodes = 5

parent_id



index_a = 0

index_b = 1

root_a = 0

root_b = 1

Quick Find UF

Union (4)

Union

```
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)                    T
    self.validate_index(index_b)                    T
    root_a = self.find(index_a)                     ?
    root_b = self.find(index_b)                     ?
    if root_a == root_b:                            T
        return                                      T
    self.parent_id[root_a] = root_b                 T
    self.n_components -= 1                          T
    return                                          T
```

Command

```
union_find.union(0, 1)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

0	1	2	3	4
1	1	2	3	4

index_a = 0

index_b = 1

root_a = 0

root_b = 1

Quick Find UF

Union (4)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  self.parent_id[root_a] = root_b
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

n_components = 3

n_nodes = 5

parent_id

0	1	2	3	4
1	1	3	3	4

index_a = 2

index_b = 3

root_a = 2

root_b = 3

Quick Union UF

Find (1)

Find

```python
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

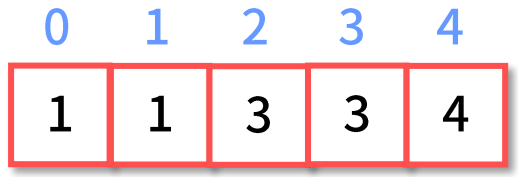
Visualization

n_components = 4

n_nodes = 5

parent_id

index = 0



Quick Union UF

Find (2)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

index = 0

0	1	2	3	4
1	1	3	3	4

Quick Union UF

Find (3)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

index = 1

0	1	2	3	4
1	1	3	3	4

Quick Union UF

Find (4)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

index = 1

0	1	2	3	4
1	1	3	3	4

Quick Union UF

Find (5)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

index = 1

0	1	2	3	4
1	1	3	3	4

Quick Union UF

Find (6)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

Visualization

n_components = 4

n_nodes = 5

parent_id

index = 1

0	1	2	3	4
1	1	3	3	4

What is the time complexity of find in QuickUnionUF?

Quick Find UF

Union (5)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  self.parent_id[root_a] = root_b
  self.n_components -= 1
  return
```

Command

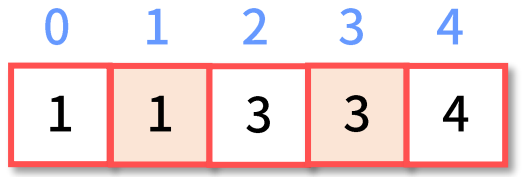
```
union_find.union(0, 2)
```

Visualization

n_components = 3

n_nodes = 5

parent_id



index_a = 2

index_b = 3

root_a = 1

root_b = 3

Quick Find UF

Union (6)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    self.parent_id[root_a] = root_b
    self.n_components -= 1
    return
```

Command

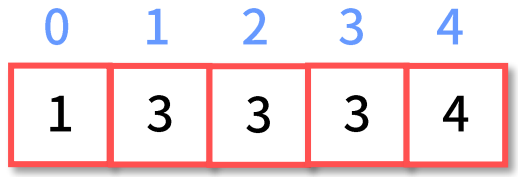
```
union_find.union(0, 2)
```

Visualization

n_components = 3

n_nodes = 5

parent_id



index_a = 2

index_b = 3

root_a = 1

root_b = 3

Quick Find UF

Union (7)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    self.parent_id[root_a] = root_b
    self.n_components -= 1
    return
```

Command

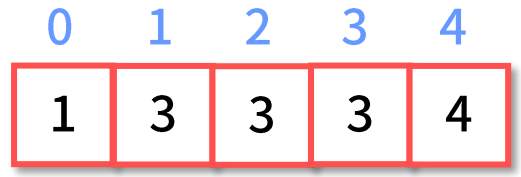
```
union_find.union(0, 2)
```

Visualization

n_components = 2

n_nodes = 5

parent_id



index_a = 2

index_b = 3

root_a = 1

root_b = 3

Quick Find UF

Connected (1)

Check Connectivity

```python
def is_connected(self, index_a, index_b) -> bool:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    return root_a == root_b
```

Command

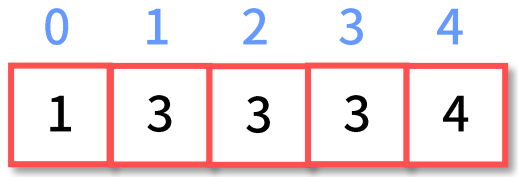
```
union.is_connected(0, 3)
```

Visualization

n_components = 2

n_nodes = 5

parent_id



index_a = 2

index_b = 3

root_a = 3

root_b = 3

Quick Find UF

Connected (2)

Check Connectivity

```python
def is_connected(self, index_a, index_b) -> bool:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    return root_a == root_b
```

Command

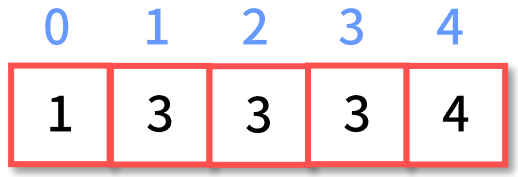
```
union.is_connected(0, 3)
```

Visualization

n_components = 2

n_nodes = 5

parent_id



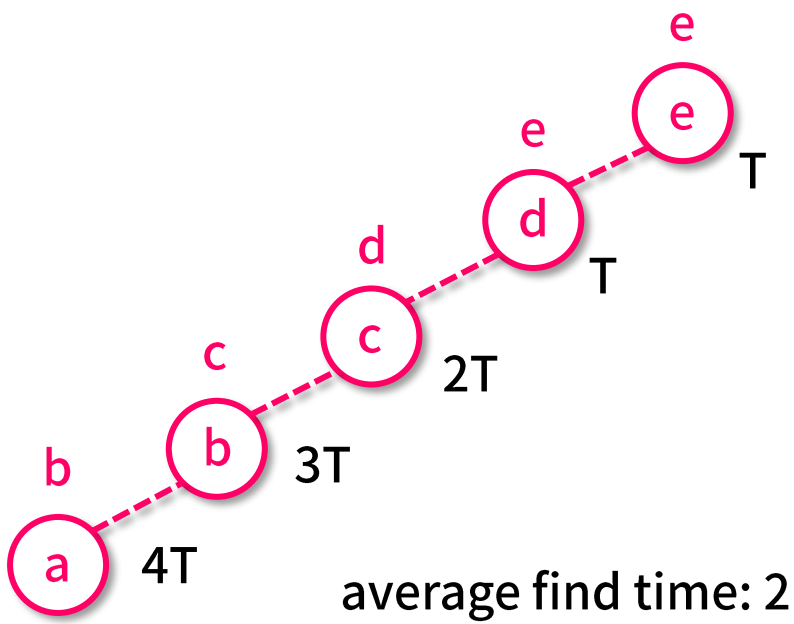
index_a = 2

index_b = 3

root_a = 3

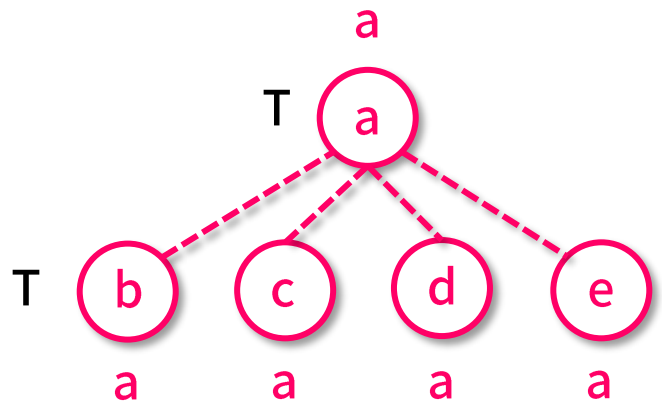
root_b = 3

Ranked Quick Union UF (1)



average find time: 2.2T

1. Put a, b in the same disjoint set
2. Put a, c in the same disjoint set
3. Put a, d in the same disjoint set
4. Put a, e in the same disjoint set

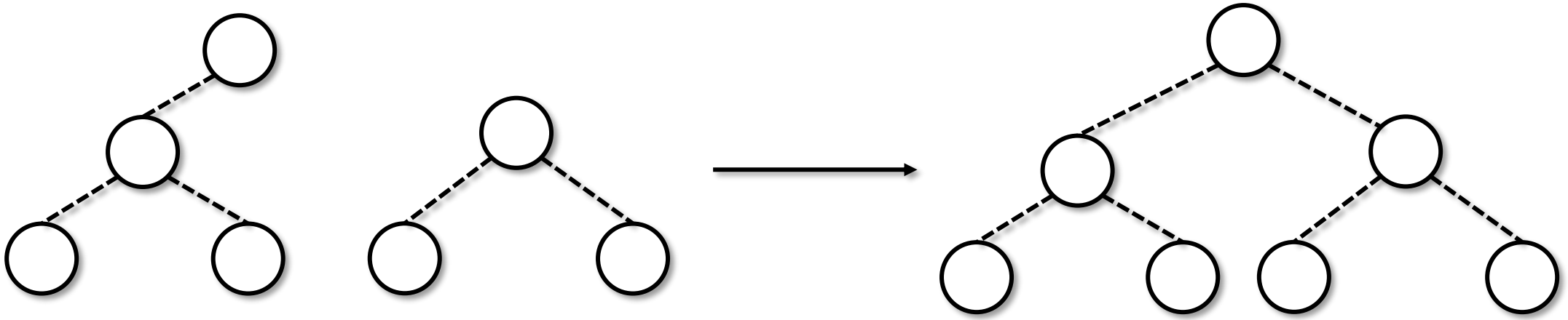


average find time: T

1. Put b, a in the same disjoint set
2. Put c, a in the same disjoint set
3. Put d, a in the same disjoint set
4. Put e, a in the same disjoint set

Ranked Quick Union UF (2)

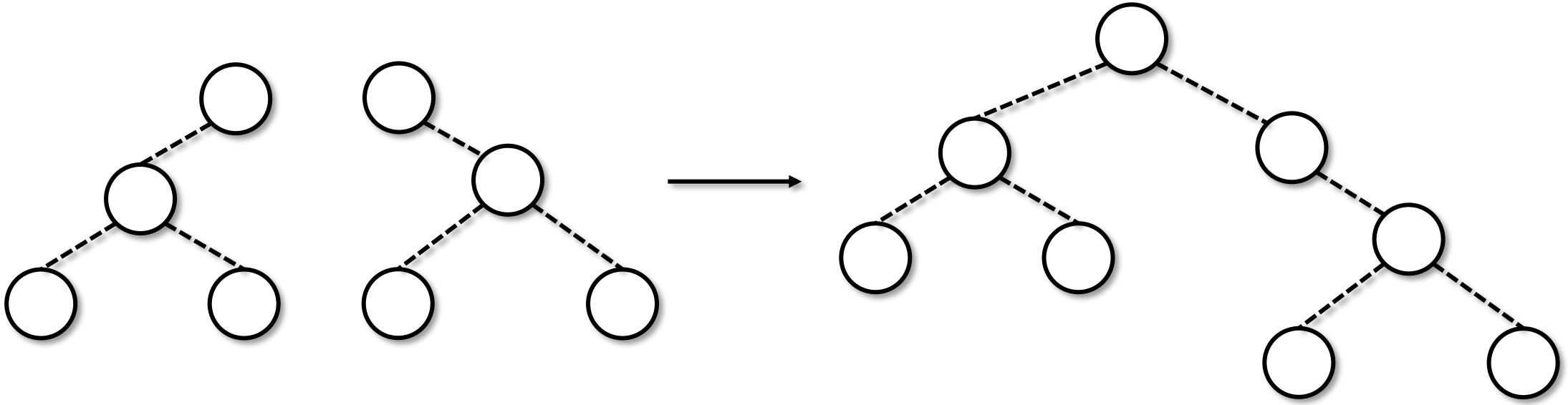
- The disjoint set with smaller rank needs to be merged to the disjoint set with larger rank.



- The highest rank of the disjoint sets either **does not change**.

Ranked Quick Union UF (3)

- The disjoint set with smaller rank needs to be merged to the disjoint set with larger rank.



- The highest rank of the disjoint sets either **does not change** or **increase by one** after each union operation.

Ranked Quick Union UF (4)

Initialization

```
class RankedQuickUnionUF:
  def __init__(self, n_nodes):
    self.n_components: int = n_nodes
    self.n_nodes: int = n_nodes
    self.parent_id: Array = Array(n_nodes)
    self.rank: Array = Array(n_nodes)
    for i in range(n_nodes):
      self.parent_id[i] = i
      self.rank[i] = 1

  def __str__(self) -> str: ···

  def __len__(self) -> int: ···

  def validate_index(self, index) -> None: ···
      check if the index is valid
  def is_connected(self, index_a, index_b) -> bool: ···
      check if two indices is in the same disjoint set
  def union(self, index_a, index_b) -> None: ···
      merge the disjoint set where a and b belong to
  def find(self, index) -> int: ···
      find the disjoint label of the given index
```

Command

```
union_find = RankedQuickUnionUF(5)
```

Visualization

n_components = 5

n_nodes = 5

	0	1	2	3	4
parent_id	0	1	2	3	4
rank	1	1	1	1	1

Ranked Quick Union UF

Union (1)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

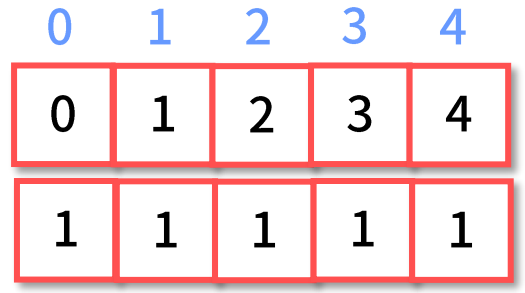
Command

```
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5



index_a = 0

index_b = 1

root_a = 0

root_b = 1

Ranked Quick Union UF

Union (2)

Union

```python
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

Command

```python
union_find.union(0, 1)
```

Visualization

n_components = 5

n_nodes = 5

	0	1	2	3	4
parent_id	1	1	2	3	4
rank	1	2	1	1	1

index_a = 0

index_b = 1

root_a = 0

root_b = 1

Ranked Quick Union UF

Union (3)

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

Command

```
union_find.union(0, 1)
```

Visualization

n_components = 4

n_nodes = 5

	0	1	2	3	4
parent_id	1	1	2	3	4
rank	1	2	1	1	1

index_a = 0

index_b = 1

root_a = 0

root_b = 1

Ranked Quick Union UF

Union (4)

Union

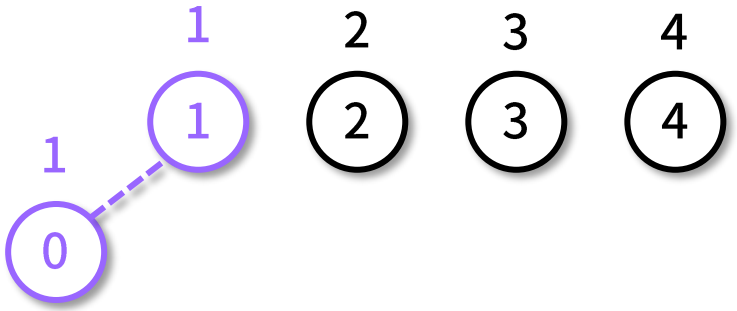
```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    if self.rank[root_a] < self.rank[root_b]:
        self.parent_id[root_a] = root_b
    elif self.rank[root_a] > self.rank[root_b]:
        self.parent_id[root_b] = root_a
    else:
        self.parent_id[root_a] = root_b
        self.rank[root_b] += 1
    self.n_components -= 1
    return
```

Command

```python
union_find.union(0, 1)
```

Visualization

	0	1	2	3	4
parent_id	1	1	2	3	4
rank	1	2	1	1	1



Ranked Quick Union UF

Union (5)

Union

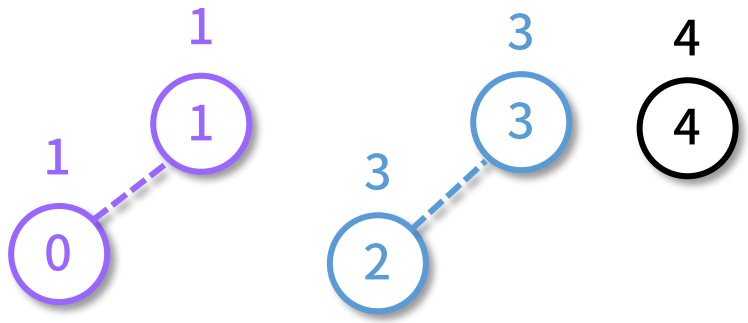
```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

Command

```
union_find.union(2, 3)
```

Visualization

	0	1	2	3	4
parent_id	1	1	3	3	4
rank	1	2	1	2	1



Ranked Quick Union UF

Union (6)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    if self.rank[root_a] < self.rank[root_b]:
        self.parent_id[root_a] = root_b
    elif self.rank[root_a] > self.rank[root_b]:
        self.parent_id[root_b] = root_a
    else:
        self.parent_id[root_a] = root_b
        self.rank[root_b] += 1
    self.n_components -= 1
    return
```

Command

```python
union_find.union(0, 4)
```

Visualization

n_components = 3

n_nodes = 5

	0	1	2	3	4
parent_id	1	1	3	3	4
rank	1	2	1	2	1

index_a = 0

index_b = 4

root_a = 1

root_b = 4

Ranked Quick Union UF

Union (7)

Union

```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    if self.rank[root_a] < self.rank[root_b]:
        self.parent_id[root_a] = root_b
    elif self.rank[root_a] > self.rank[root_b]:
        self.parent_id[root_b] = root_a
    else:
        self.parent_id[root_a] = root_b
        self.rank[root_b] += 1
    self.n_components -= 1
    return
```

Command

```python
union_find.union(0, 4)
```

Visualization

n_components = 3

n_nodes = 5

	0	1	2	3	4
parent_id	1	1	3	3	1
rank	1	2	1	2	1

index_a = 0

index_b = 4

root_a = 1

root_b = 4

Ranked Quick Union UF

Union (8)

Union

```python
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

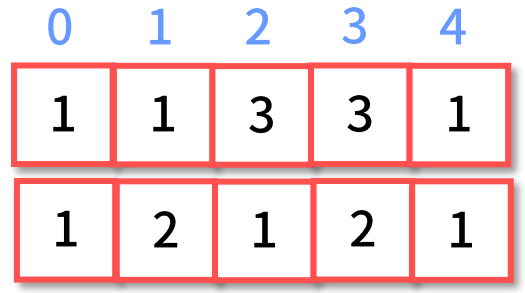
Command

```python
union_find.union(0, 4)
```

Visualization

n_components = 2

n_nodes = 5



index_a = 0

index_b = 4

root_a = 1

root_b = 4

Ranked Quick Union UF

Union (9)

Union

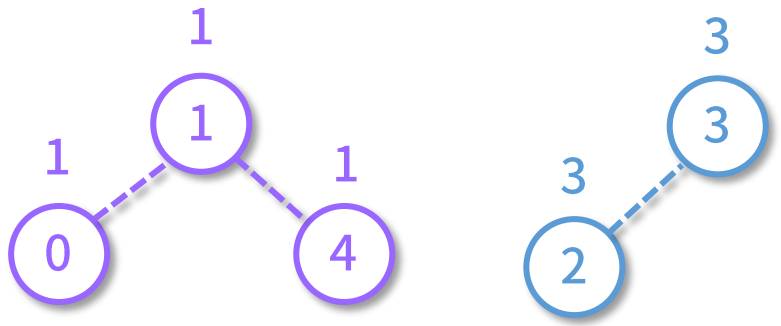
```python
def union(self, index_a, index_b) -> None:
    self.validate_index(index_a)
    self.validate_index(index_b)
    root_a = self.find(index_a)
    root_b = self.find(index_b)
    if root_a == root_b:
        return
    if self.rank[root_a] < self.rank[root_b]:
        self.parent_id[root_a] = root_b
    elif self.rank[root_a] > self.rank[root_b]:
        self.parent_id[root_b] = root_a
    else:
        self.parent_id[root_a] = root_b
        self.rank[root_b] += 1
    self.n_components -= 1
    return
```

Command

```python
union_find.union(0, 4)
```

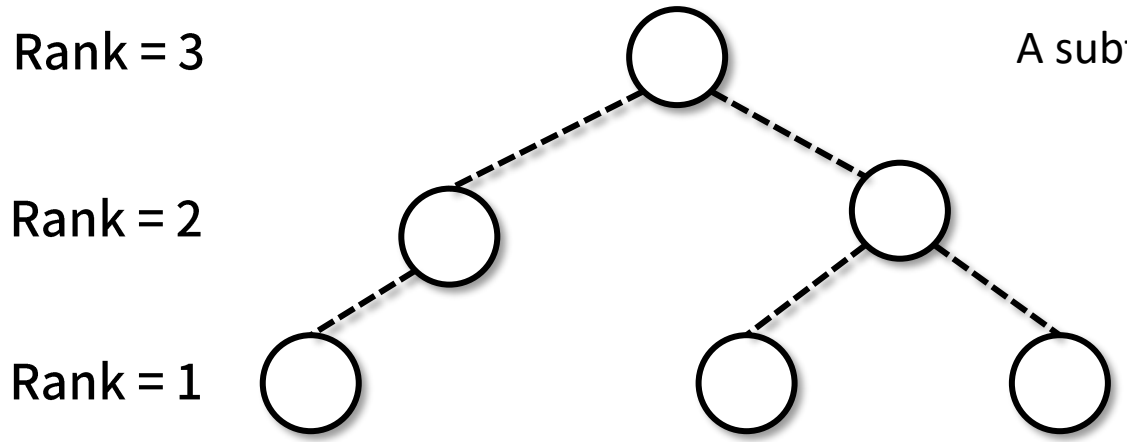
Visualization

	0	1	2	3	4
parent_id	1	1	3	3	1
rank	1	2	1	2	1



Ranked Quick Union UF

Time Complexity - Find



A subtree of rank r has at least 2^{r-1} nodes

The largest rank for tree with maximum height:

$n \geq 2^{1-1} + 2^{2-1} + \ldots + 2^{rank-1}$

$n \geq 2^{rank} - 1$

$\log_2(n+1) \geq rank$

Worst case time complexity: $\log(n)$

Path Compression Ranked Quick Union UF (1)

Ranked Quick Union UF

Union

```
def union(self, index_a, index_b) -> None:
  self.validate_index(index_a)
  self.validate_index(index_b)
  root_a = self.find(index_a)
  root_b = self.find(index_b)
  if root_a == root_b:
    return
  if self.rank[root_a] < self.rank[root_b]:
    self.parent_id[root_a] = root_b
  elif self.rank[root_a] > self.rank[root_b]:
    self.parent_id[root_b] = root_a
  else:
    self.parent_id[root_a] = root_b
    self.rank[root_b] += 1
  self.n_components -= 1
  return
```

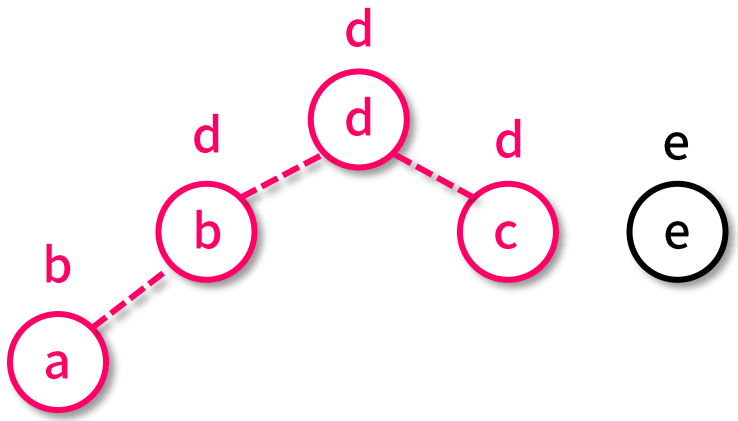
Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

every time we need to find the root, we traverse lots of nodes in the same disjoint set.

Path Compression Ranked Quick Union UF (2)

Ranked Quick Union UF



when we want to find the root of a, we go through a and b (which are all in the same disjoint set).

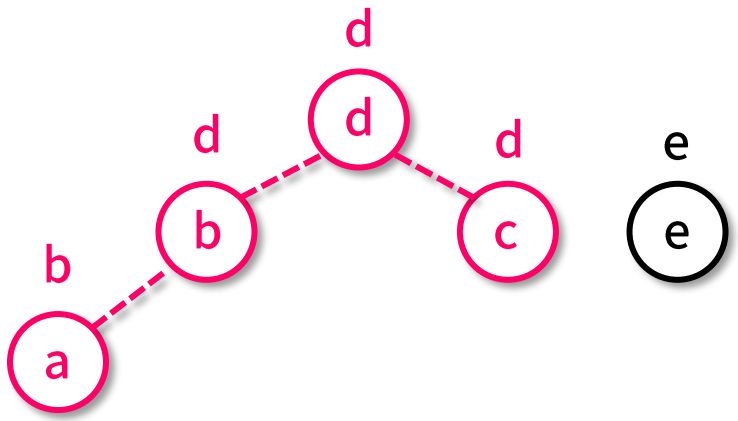
Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

every time we need to find the root, we traverse lots of nodes in the same disjoint set.

Path Compression Ranked Quick Union UF (3)

Ranked Quick Union UF



when we want to find the root of a, we go through a and b (which are all in the same disjoint set).

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    index = self.parent_id[index]
  return index
```

every time we need to find the root, we traverse lots of nodes in the same disjoint set.

since they belong to the same root, we can just flatten our node structure when we call find.

Path Compression Ranked Quick Union UF

Find (1)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

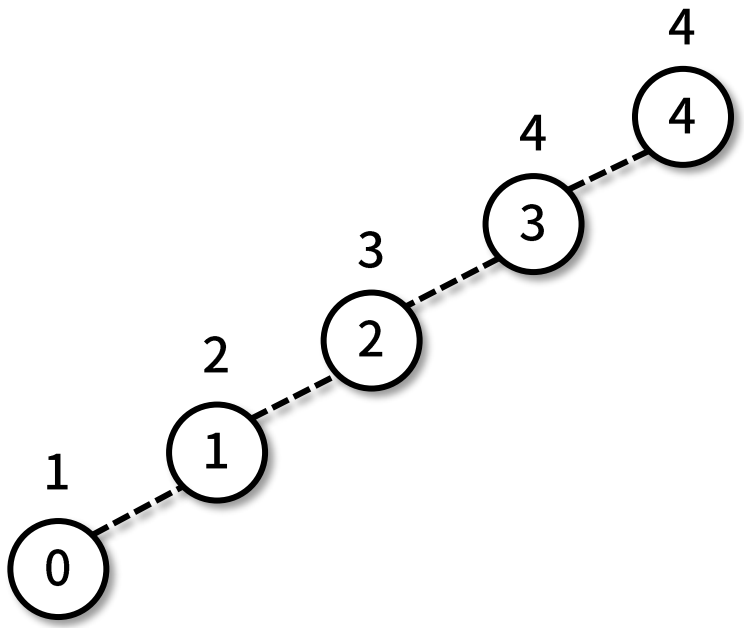
Command

```
union_find.find(0)
```

Visualization

	0	1	2	3	4
parent_id	1	2	3	4	4
rank	1	2	3	4	5

index = 0



Path Compression Ranked Quick Union UF

Find (2)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

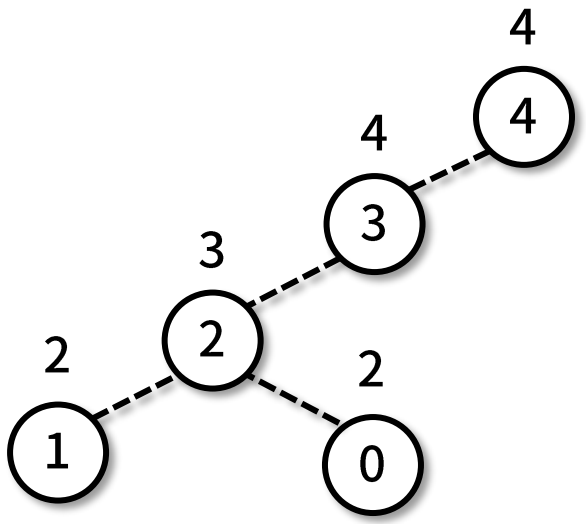
Command

```
union_find.find(0)
```

Visualization

	0	1	2	3	4
parent_id	2	2	3	4	4
rank	1	2	3	4	5

index = 0



Path Compression Ranked Quick Union UF

Find (3)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

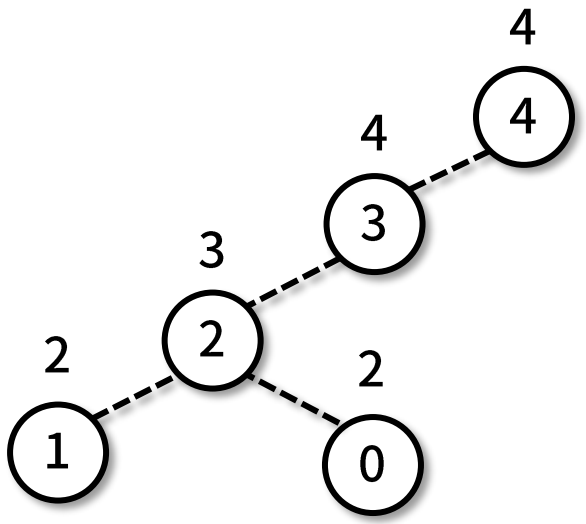
Command

```
union_find.find(0)
```

Visualization

	0	1	2	3	4
parent_id	2	2	3	4	4
rank	1	2	3	4	5

index = 2



Path Compression Ranked Quick Union UF

Find (4)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

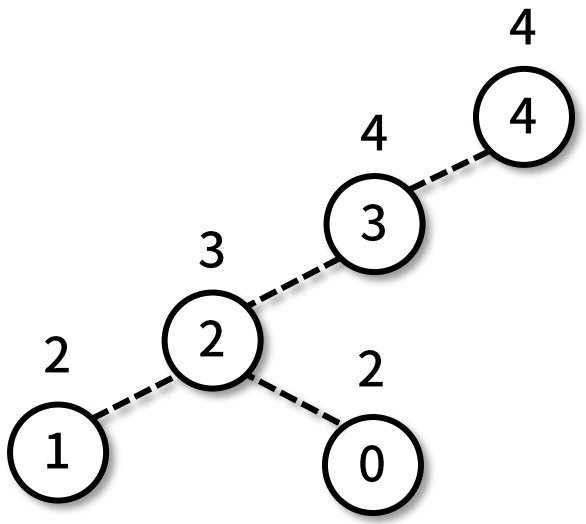
Command

```
union_find.find(0)
```

Visualization

	0	1	2	3	4
parent_id	2	2	3	4	4
rank	1	2	3	4	5

index = 2



Path Compression Ranked Quick Union UF

Find (4)

Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

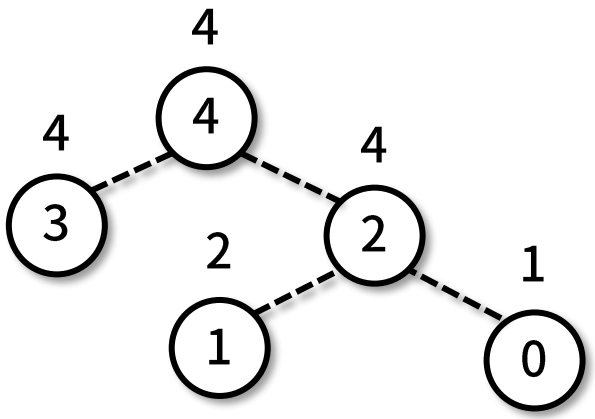
Command

```
union_find.find(0)
```

Visualization

	0	1	2	3	4
parent_id	2	2	4	4	4
rank	1	2	3	4	5

index = 2



Path Compression Ranked Quick Union UF

Find (4)

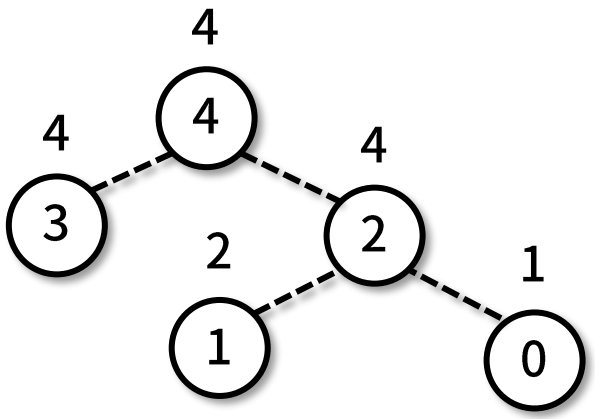
Find

```
def find(self, index) -> int:
  self.validate_index(index)
  while index != self.parent_id[index]:
    self.parent_id[index] = self.parent_id[self.parent_id[index]]
    index = self.parent_id[index]
  return index
```

Command

```
union_find.find(0)
```

index = 2



Visualization

	0	1	2	3	4
parent_id	2	2	4	4	4
rank	1	2	3	4	5

Do we need to change the rank?

Union Find

Summary

Time Complexity

	Quick Find	Quick Union	Ranked Quick Union	Path Compression Ranked Quick Union
Initialization	O( )	O( )	O( )	O( )
Union	O( )	O( )	O( )	O( )
Find	O( )	O( )	O( )	O( )
Connected	O( )	O( )	O( )	O( )

Union Find

Practices

- Longest Consecutive Sequence (Leetcode Problem 128)
- Is Graph Bipartite (Leetcode Problem 785)

Hash Table

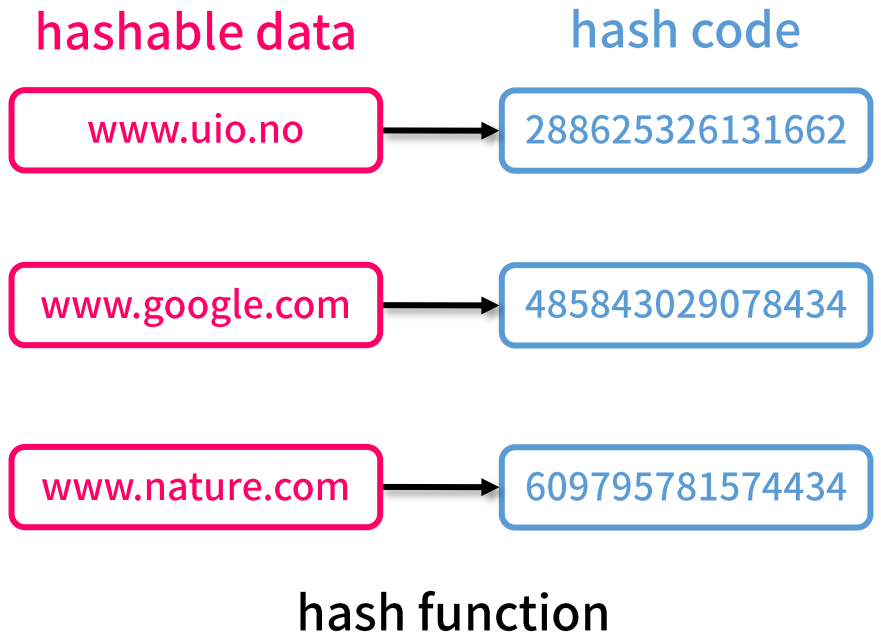
Hash Function

- A function that map data of arbitrary size to fix-sized integers.

- Good property of a hash function:
  - (Near) **constant time complexity**

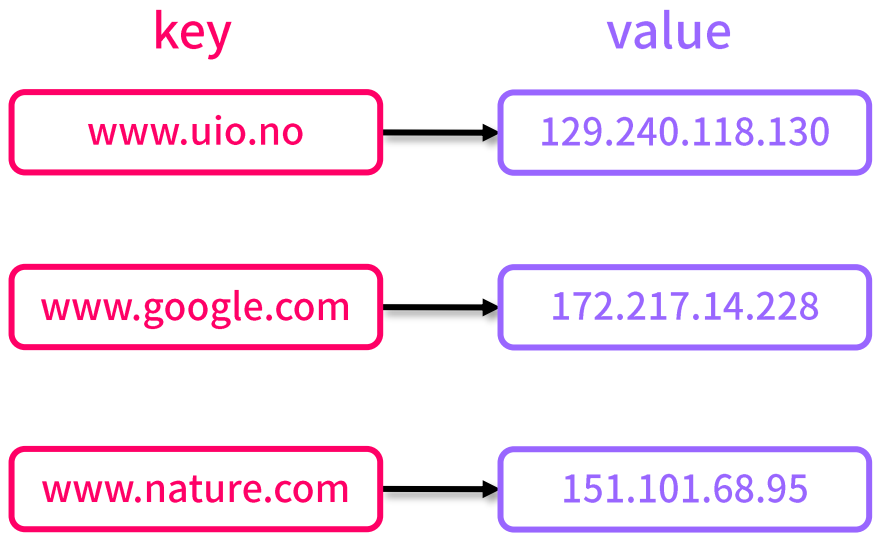
  - **Deterministic**
    - Hash code of the same input should be the same.

  - **Simple Uniform Hashing Assumption (SUHA)**
    - The output of the hash function for different inputs should be uniformly distributed in the array.



Hash Table (1)

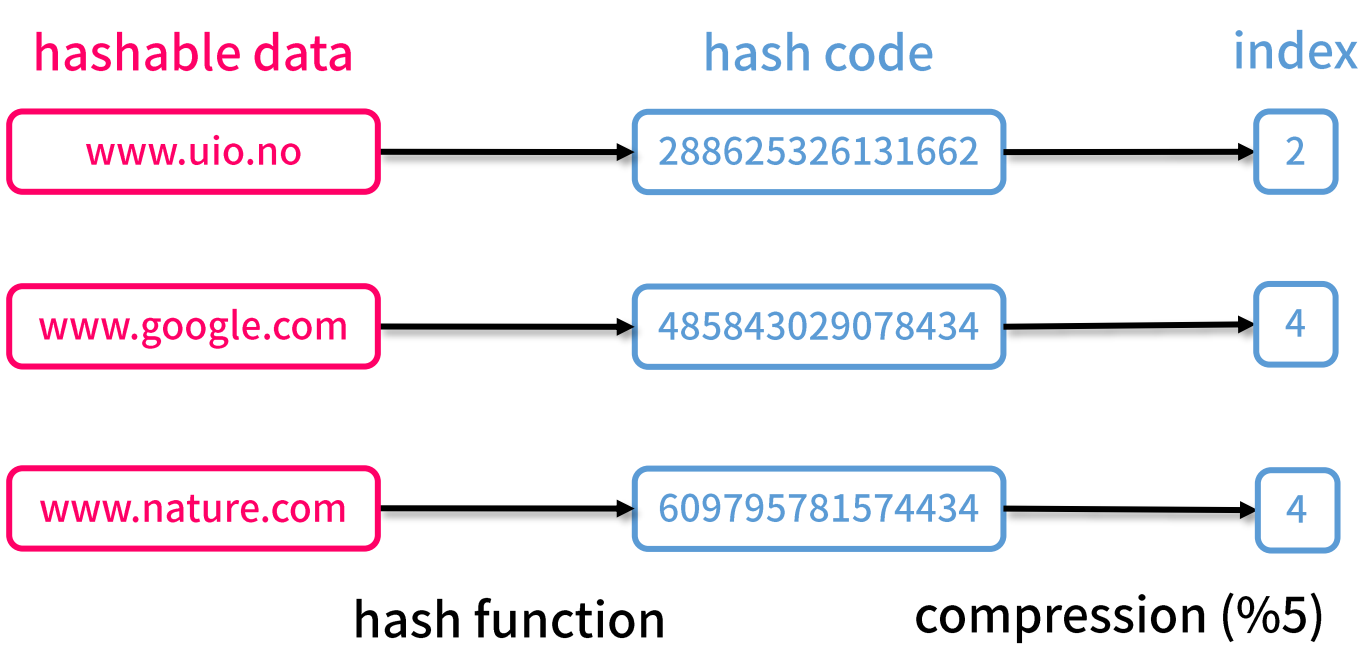
- Data structure which use **hash function** and **compression** to build a mapping between objects.
- Allow near constant search to find the corresponding value.
- Useful when we want to use non-integer index to store our data.
  - e.g. name to office

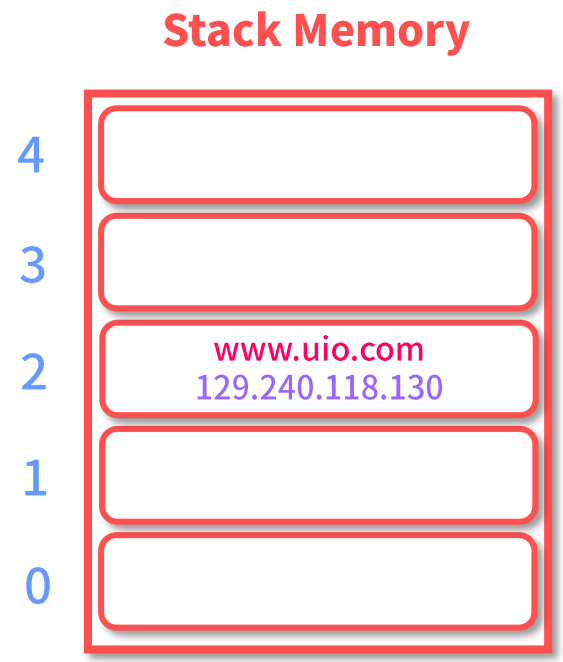


Hash Table (2)

Compression

- Can be achieve using mod (%) operator

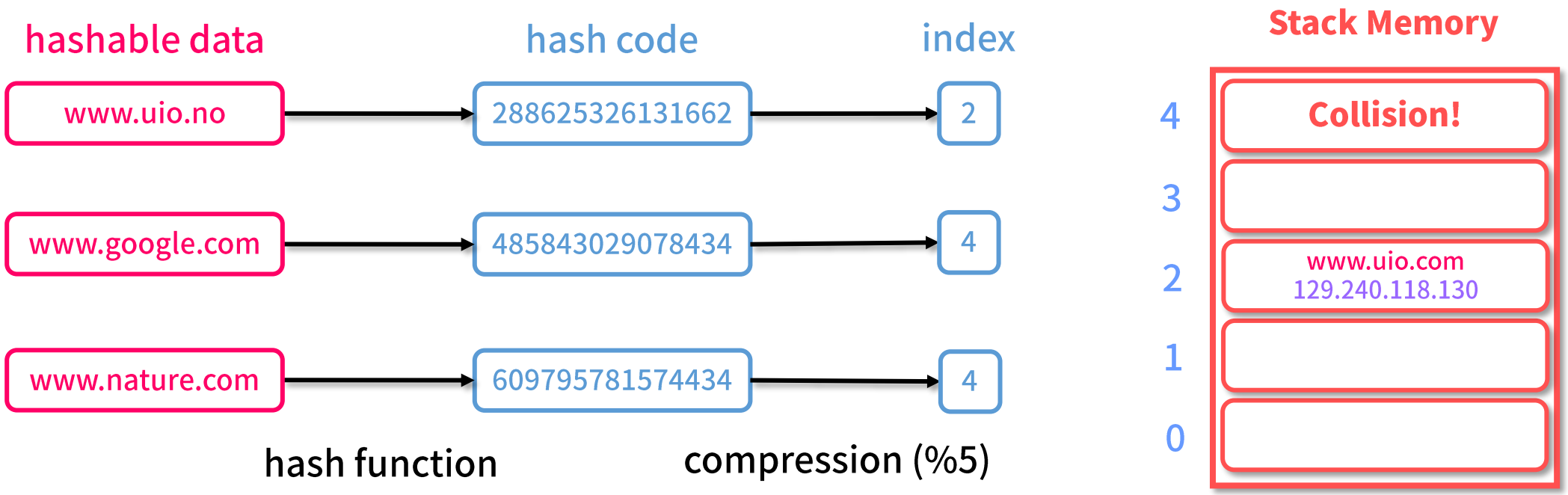




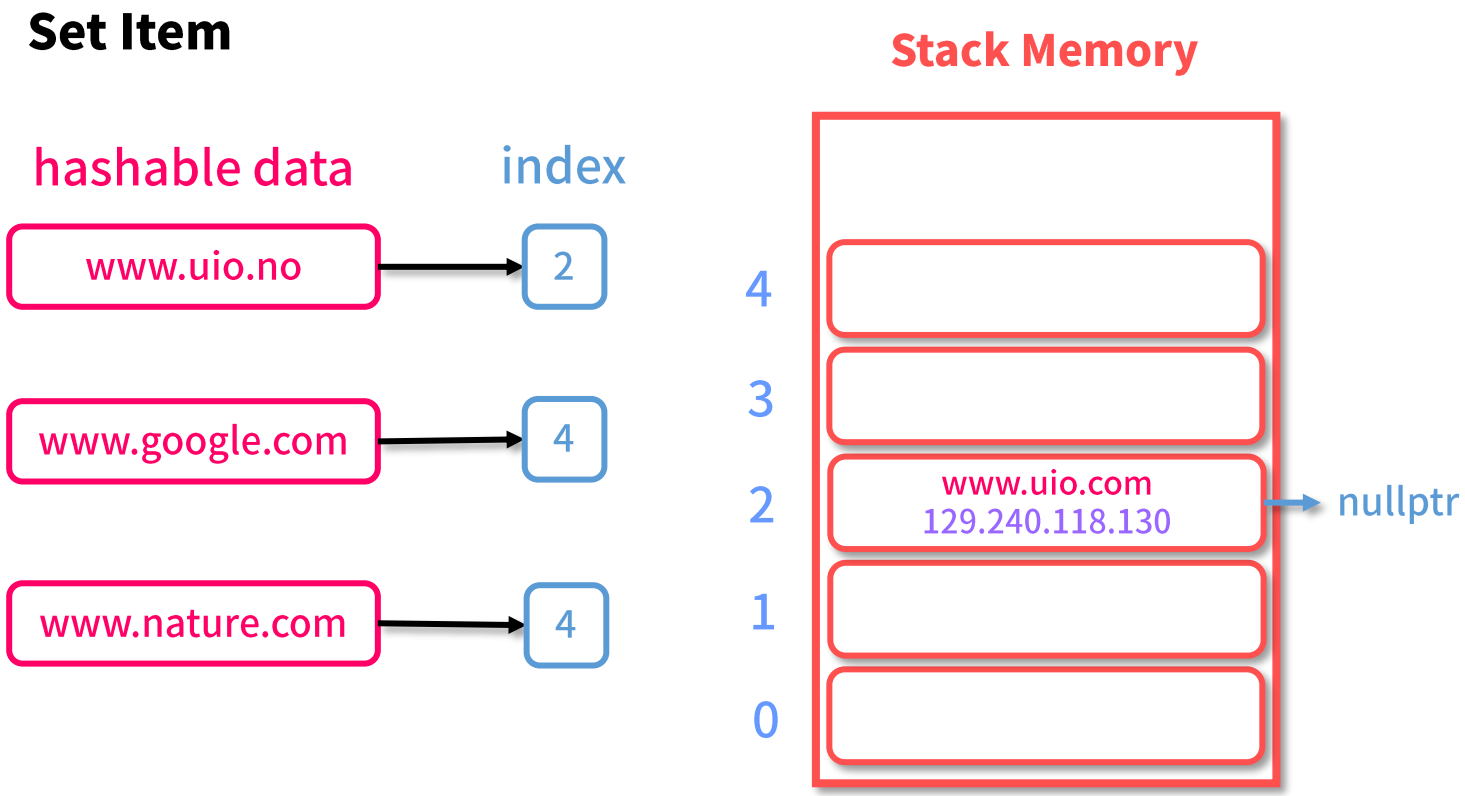
Hash Table (3)

Collision

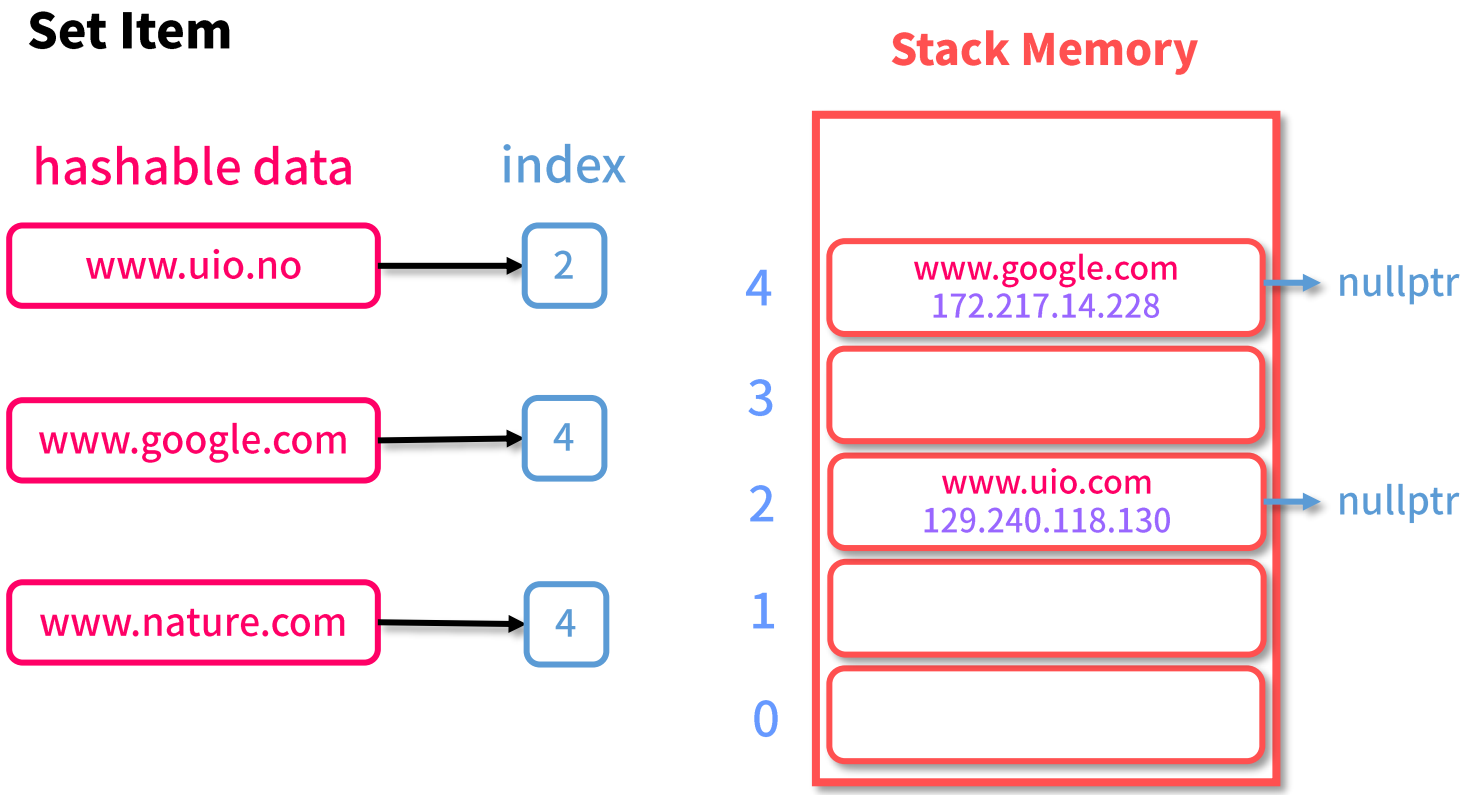
- Hash Table needs to deal with collision to store the data.



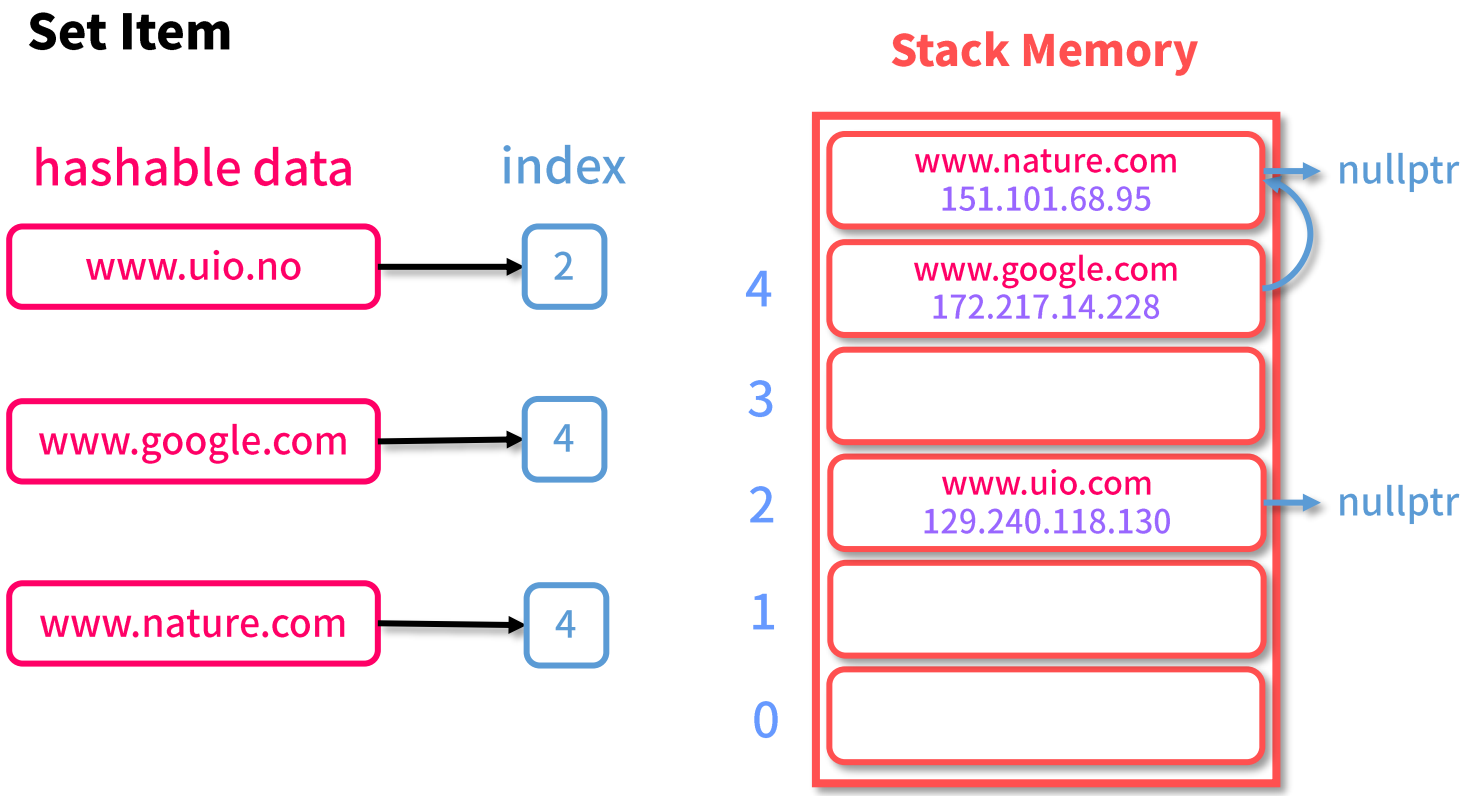
Separate Chaining Hash Table (1)



Separate Chaining Hash Table (2)

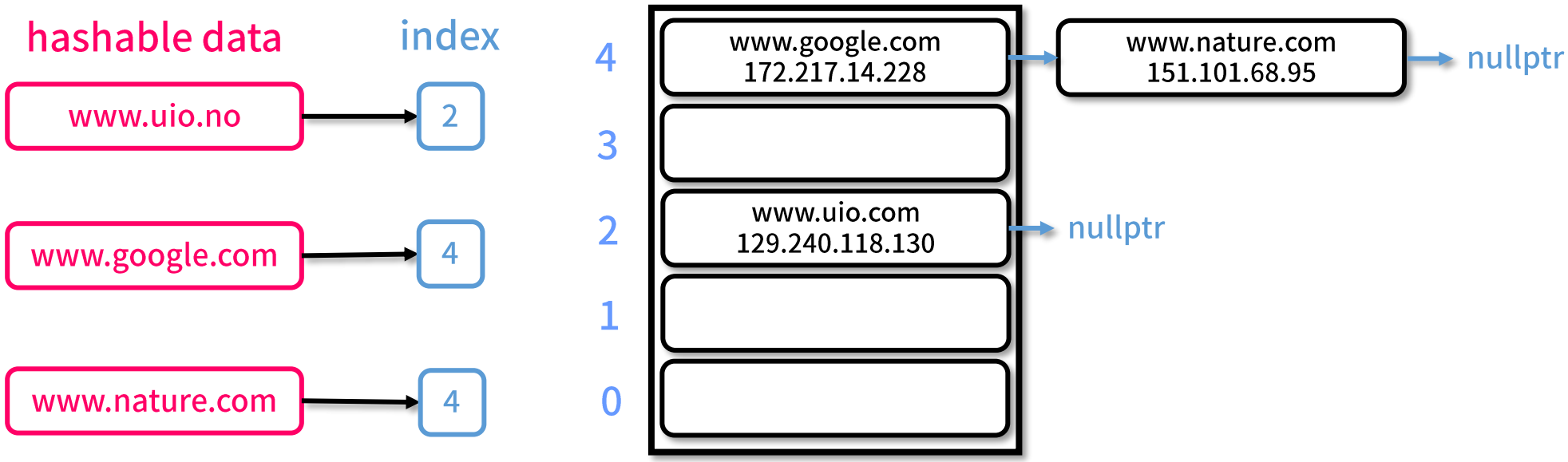


Separate Chaining Hash Table (3)



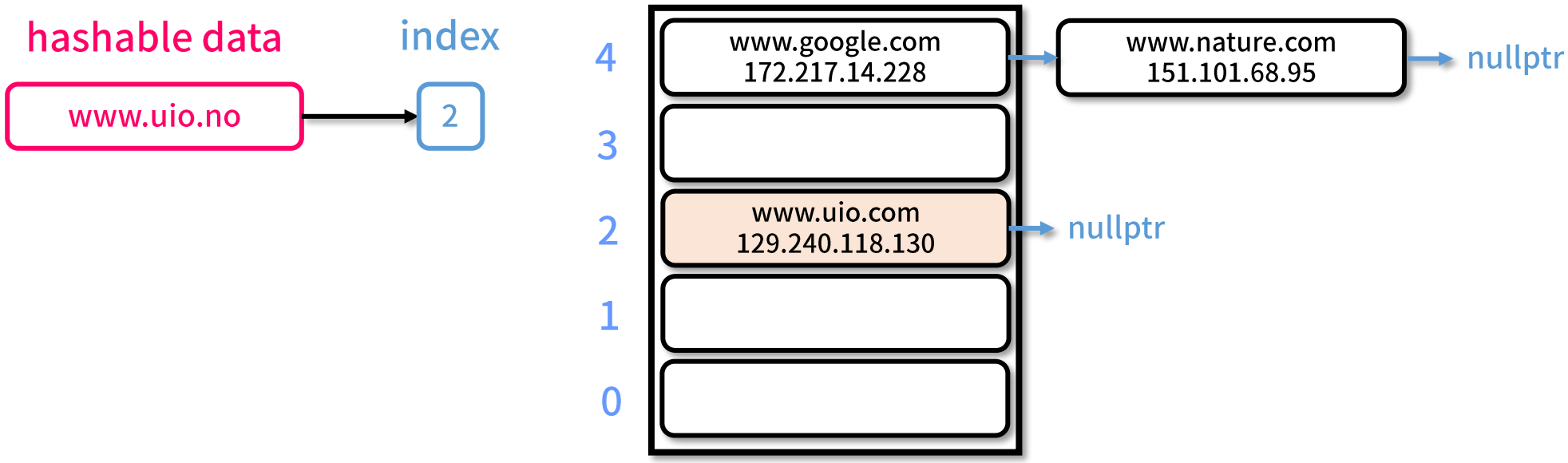
Separate Chaining Hash Table (4)

Set Item



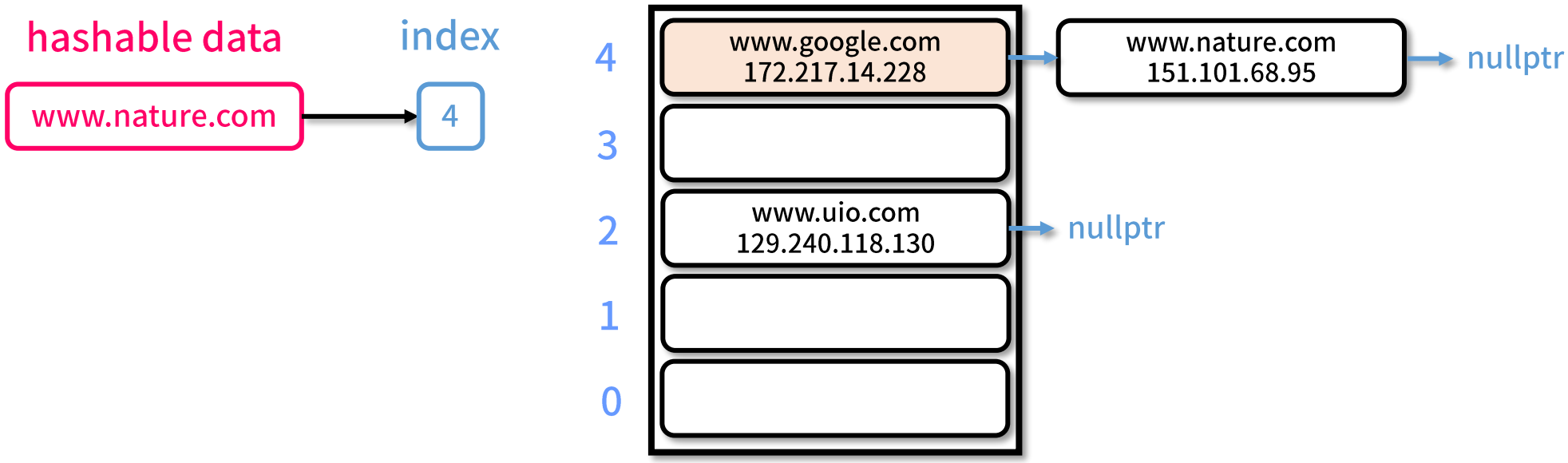
Separate Chaining Hash Table (5)

Search



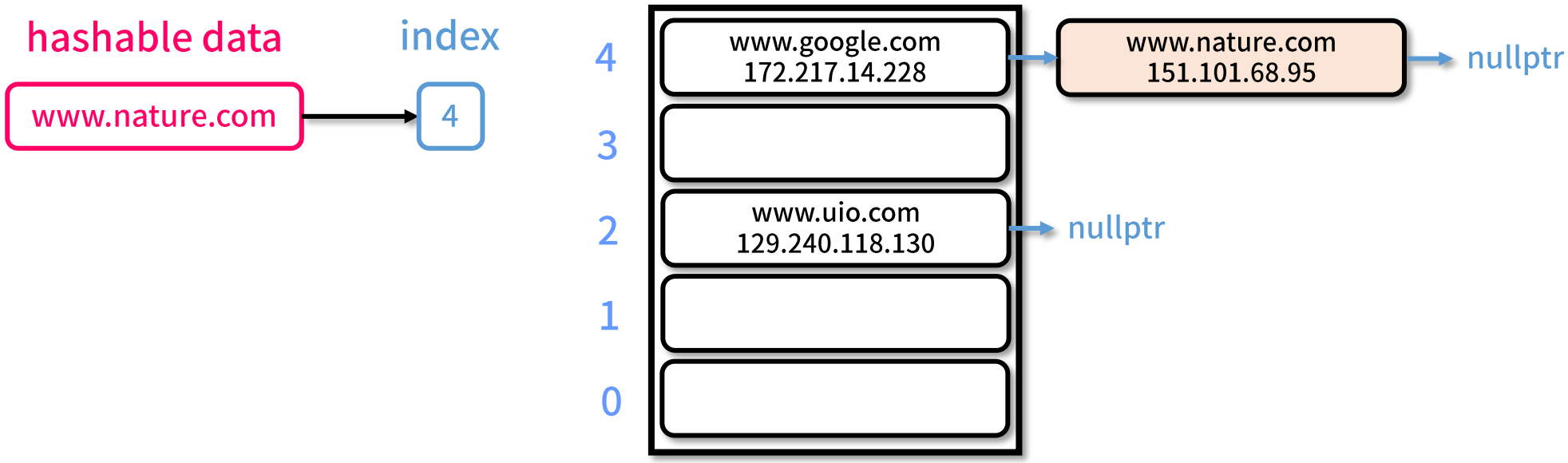
Separate Chaining Hash Table (6)

Search (collision)



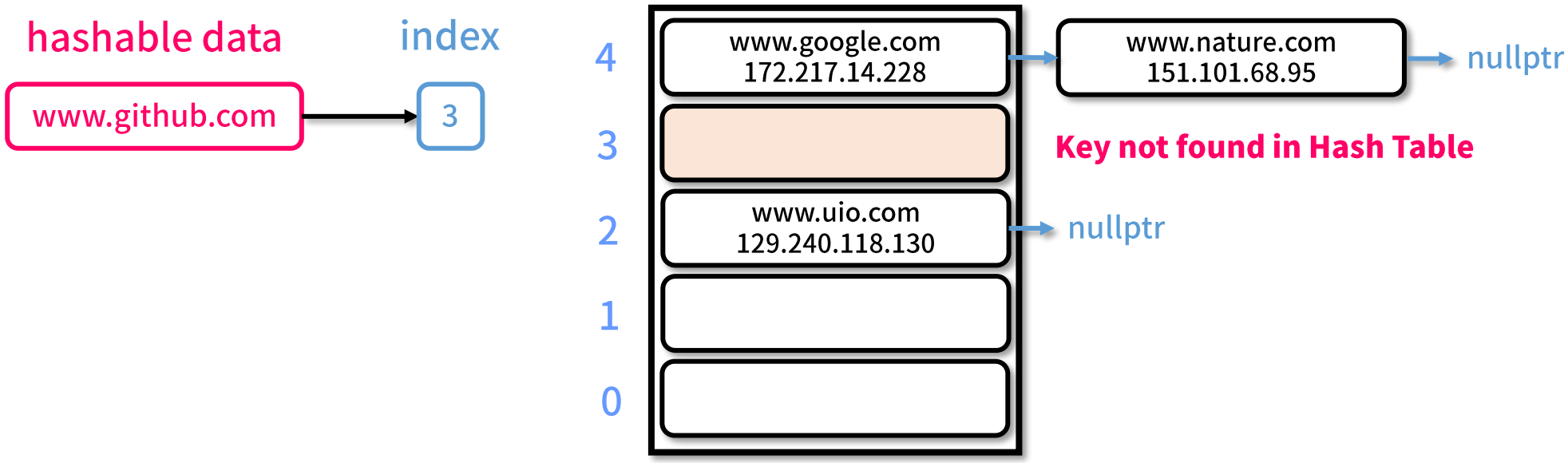
Separate Chaining Hash Table (7)

Search (collision)



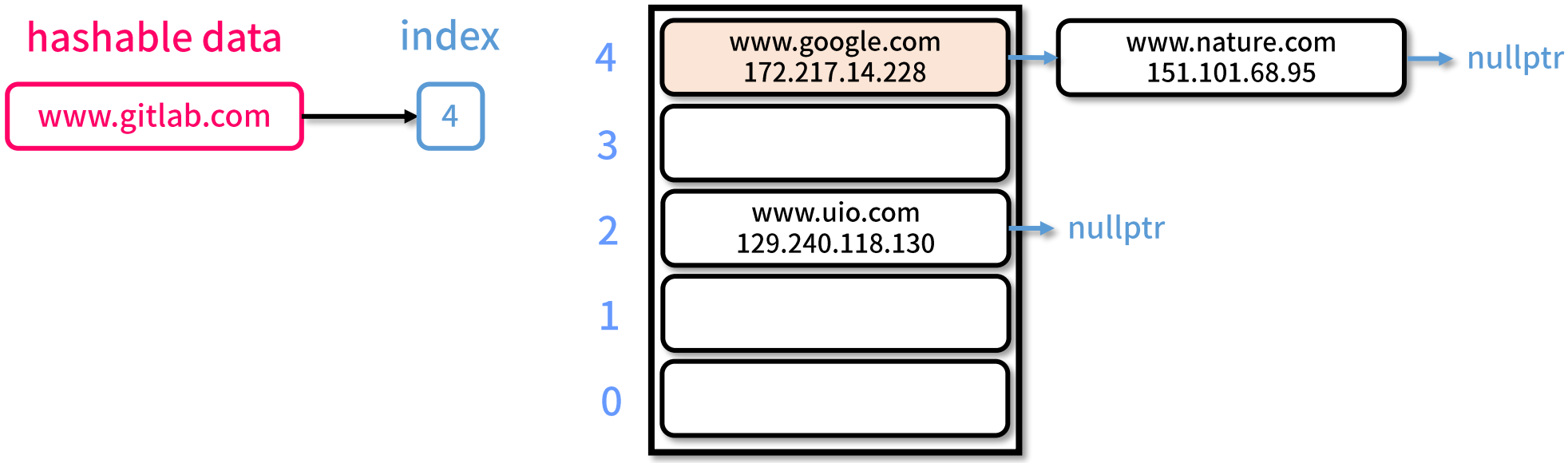
Separate Chaining Hash Table (8)

Search (not existing key)



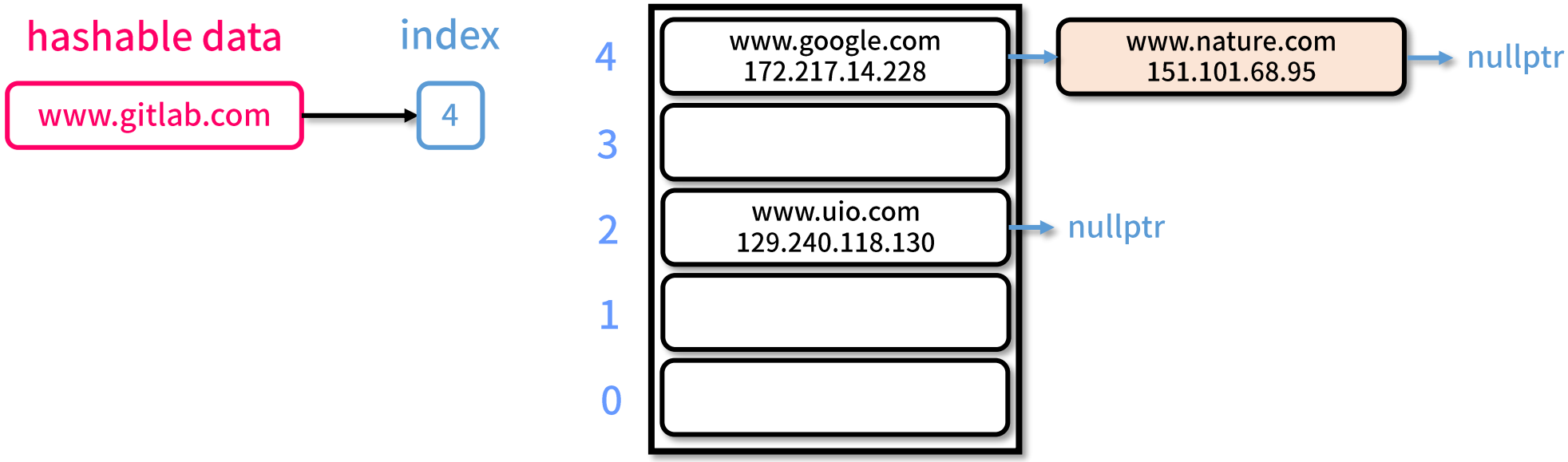
Separate Chaining Hash Table (9)

Search (not existing key)



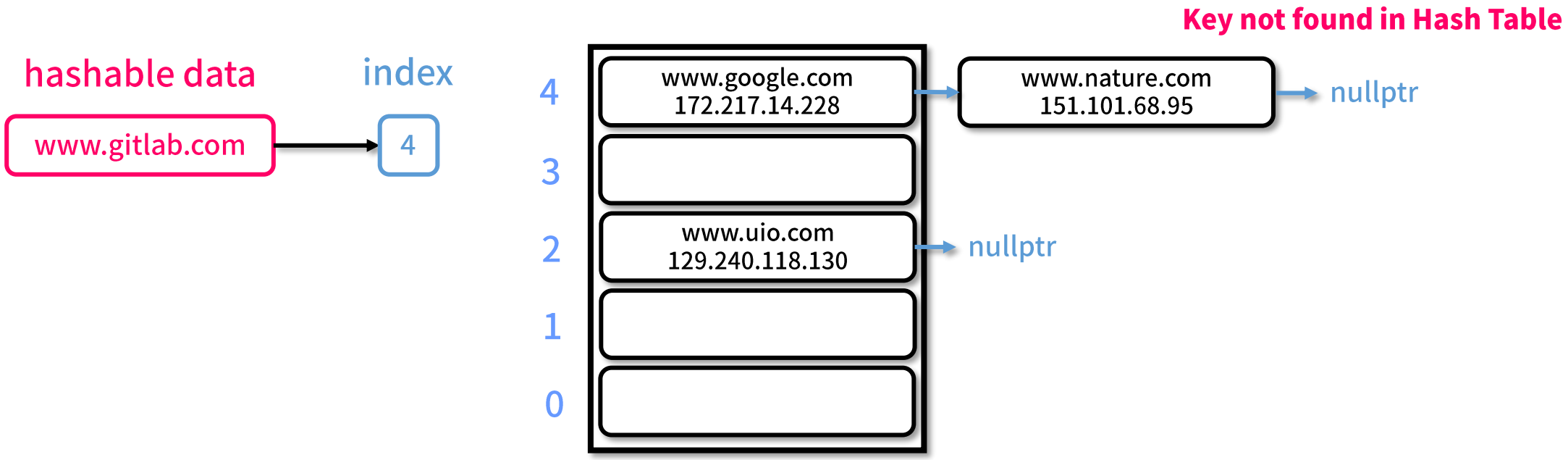
Separate Chaining Hash Table (9)

Search (not existing key)



Separate Chaining Hash Table (10)

Search (not existing key)



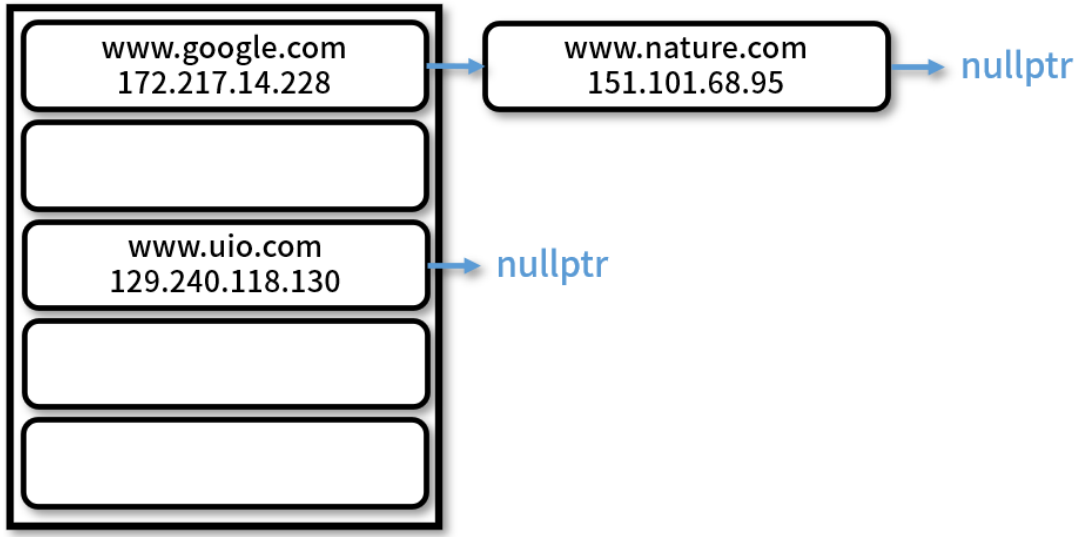
Separate Chaining Hash Table (11)

- Time complexity depends on how full the array is.
- Define a loading factor:

$$\text{let size}(\text{Array}) = m$$

$$\alpha = \frac{n}{m}$$

- Resize if the loading factor exceeds certain value.
- Under SUHA, what is the average time complexity?



Separate Chaining Hash Table (12)

Under SUHA, on average:

$$P(hash(a) = i) = P(hash(b) = i) = \frac{1}{m}$$

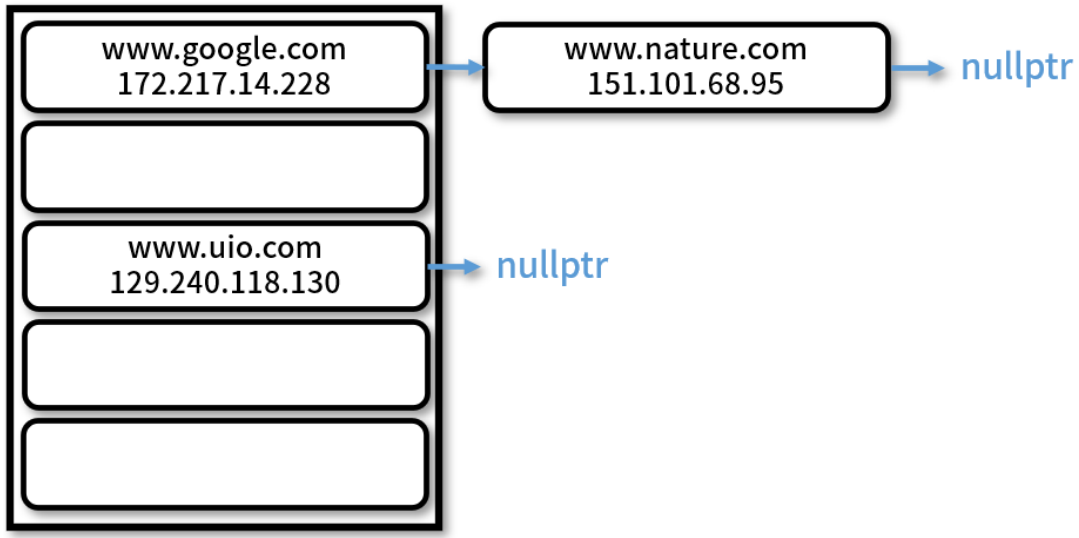
The average length of a link list:

$$\frac{n}{m} = \alpha$$

Average time complexity:

$$O(1 + \frac{n}{m}) = O(\alpha)$$

By controlling the loading factor (through resizing), we ensure the average time complexity of the hash table is constant time



Separate Chaining Hash Table (13)

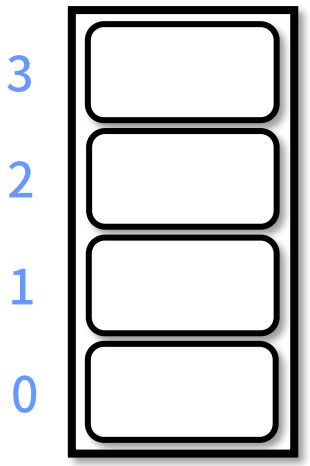
Initialization

```
class SeparateChainingHashTable(MutableMapping):
  def __init__(self, size: int = 4):
    self.n = 0
    self.size = size
    self.table = Array(size)

  def __setitem__(self, key: Hashable, value: Any) -> None: ···
      insert or set item in hash table with key, value pair
  def __getitem__(self, key: Hashable): ···
      get the value from provided key
  def __delitem__(self, key: Hashable): ···
      delete key, value pair in the hash table
  def __iter__(self) -> Iterator: ···
      iterator interface
  def __len__(self) -> int: ···

  def __str__(self) -> str: ···

  def resize(self, size: int) -> None: ···
      resize the hash table
  @property
  def alpha(self) -> float: ···
      get loading factor
  def clear(self) -> None: ···
      reset hash table
  def update(self, iterable: Iterable) -> None: ···
      merge the hash table with provided iterable map
```

Command

```
hash_table = SeparateChainingHashTable()
```

Visualization



Separate Chaining Hash Table

Set item (1)

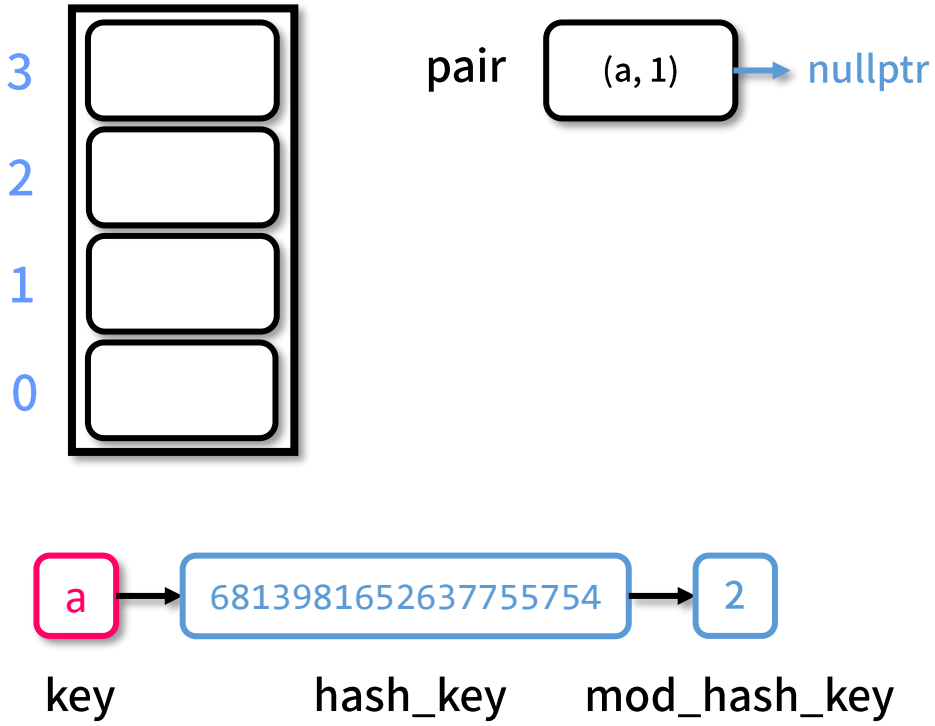
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```python
hash_table['a'] = 1
```

Visualization



Separate Chaining Hash Table

Set item (2)

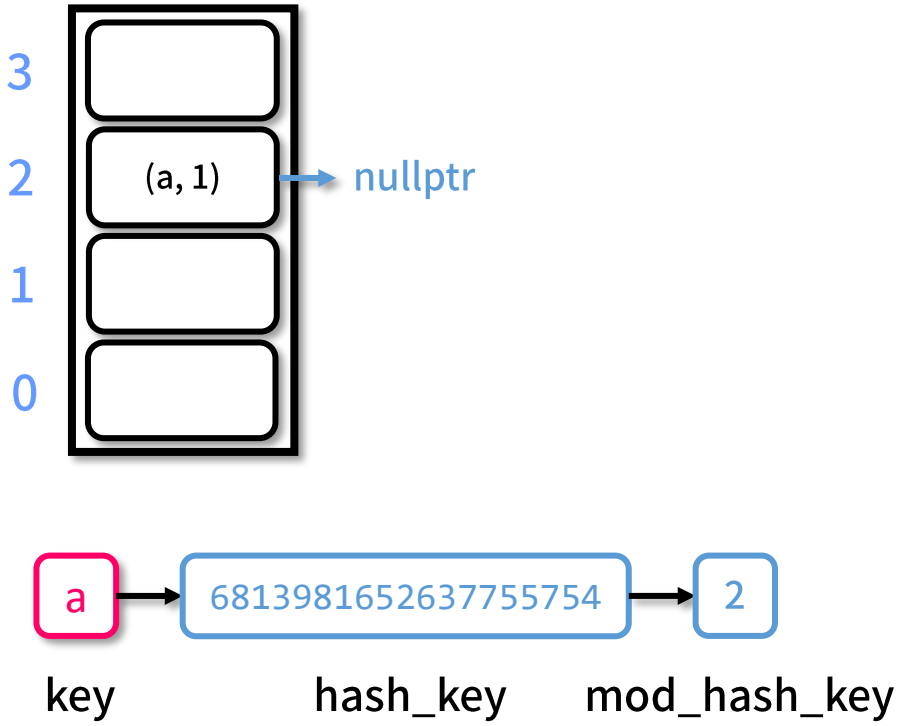
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```
hash_table['a'] = 1
```

Visualization



Separate Chaining Hash Table

Set item (3)

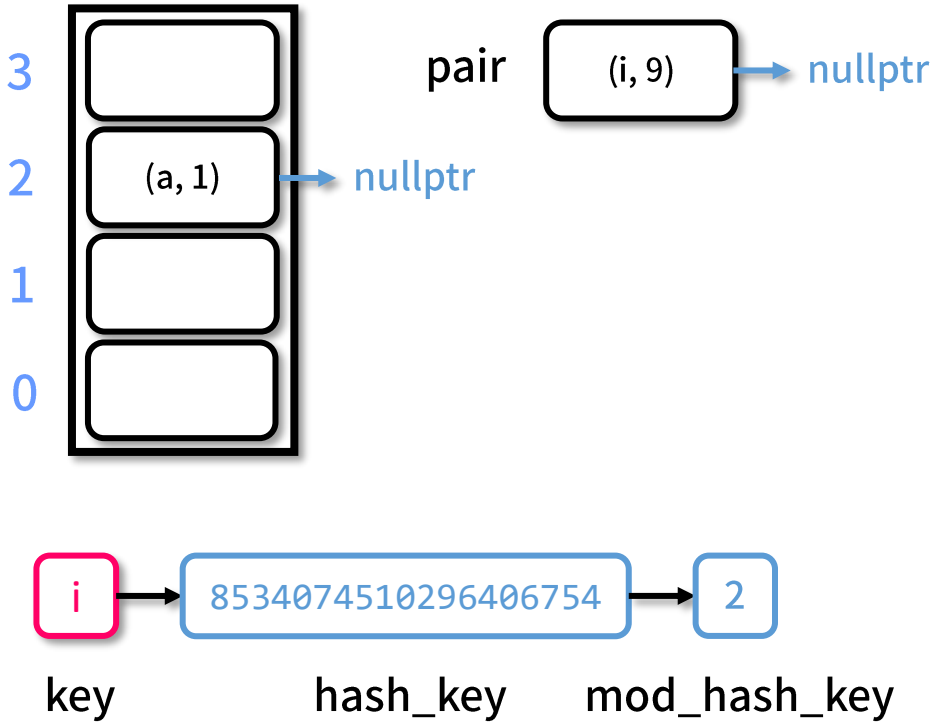
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```python
hash_table['i'] = 9
```

Visualization



Separate Chaining Hash Table

Set item (4)

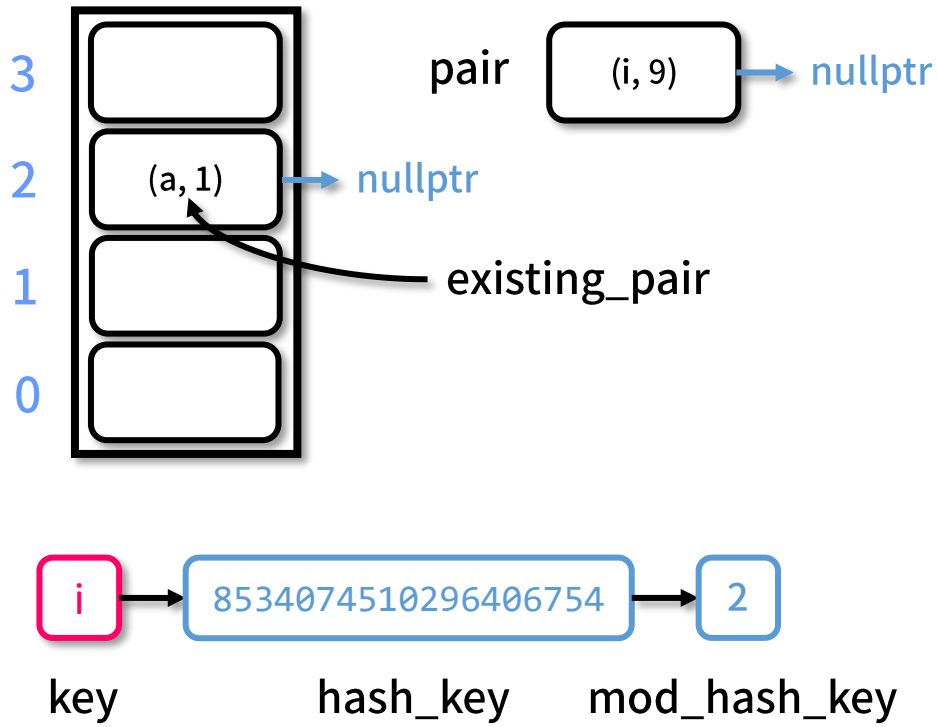
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```
hash_table['i'] = 9
```

Visualization



Separate Chaining Hash Table

Set item (5)

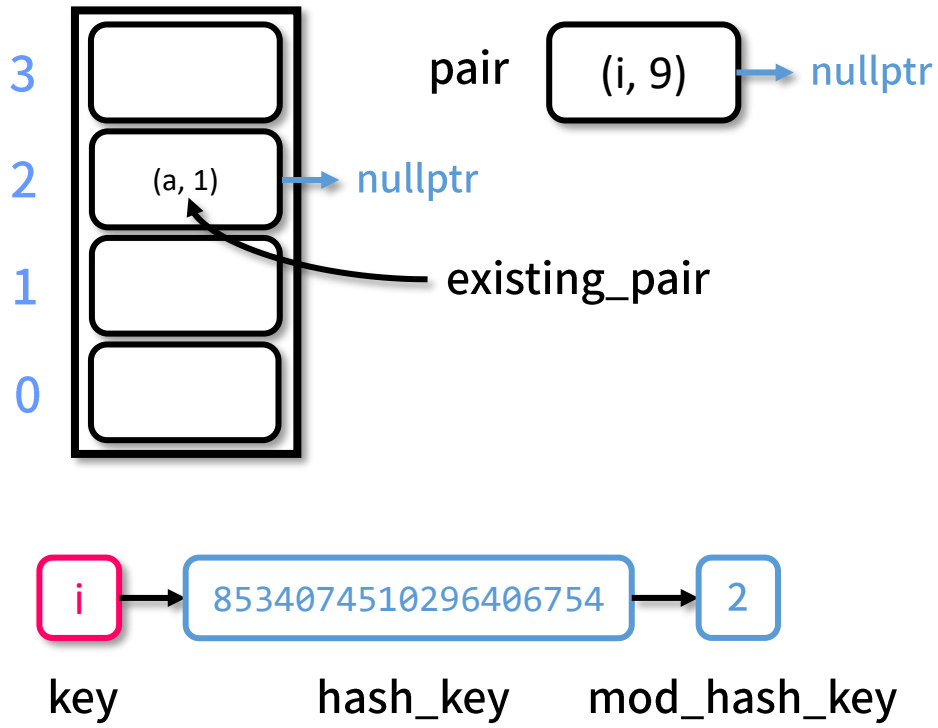
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```python
hash_table['i'] = 9
```

Visualization



Separate Chaining Hash Table

Set item (6)

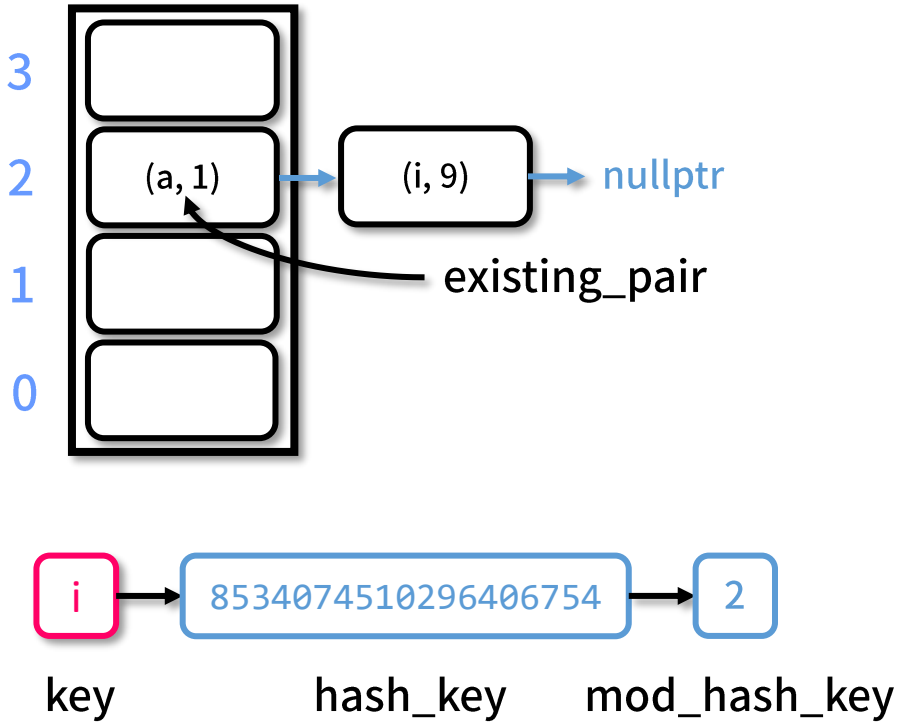
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```python
hash_table['i'] = 9
```

Visualization



Separate Chaining Hash Table

Set item (7)

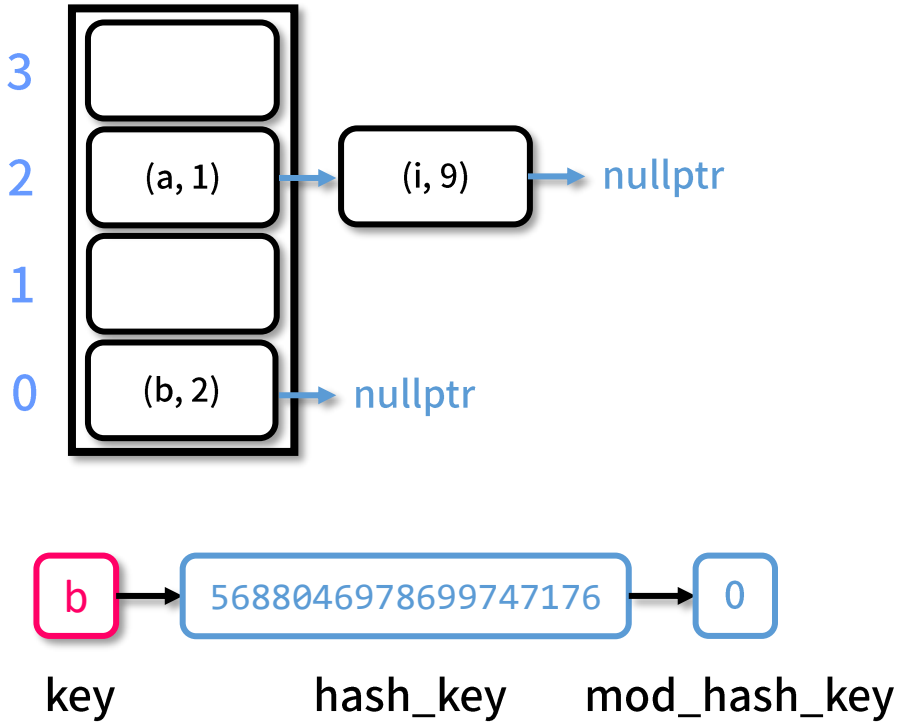
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    pair = Pair(key, value)
    if self.table[mod_hash_key] is None:
        self.table[mod_hash_key] = pair
        self.n += 1
    else:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                existing_pair.value = value
                return
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
        existing_pair.next_pair = pair
        self.n += 1
        if self.alpha >= 10:
            self.resize(self.size * 2)
```

Command

```python
hash_table['b'] = 2
```

Visualization



Separate Chaining Hash Table

Get item (1)

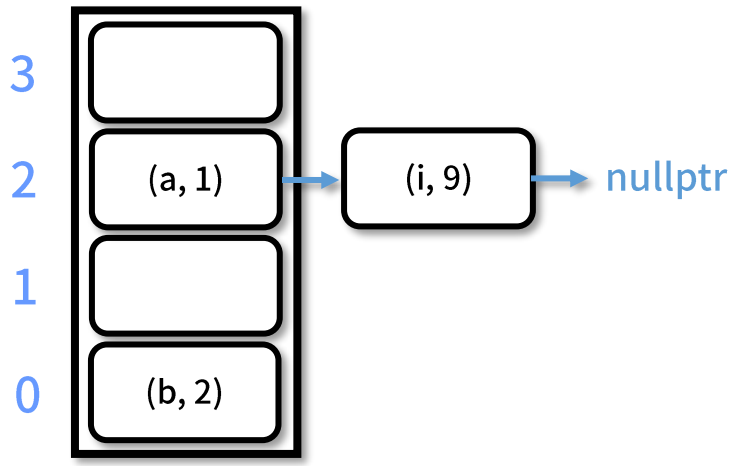
Get Item

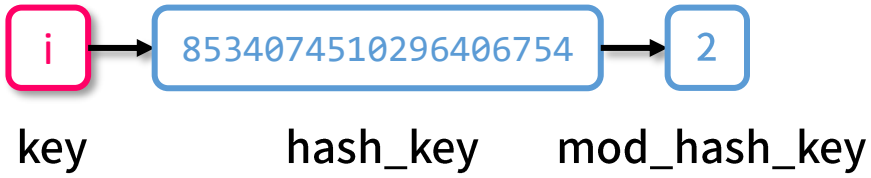
```python
def __getitem__(self, key: Hashable):
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    if self.table[mod_hash_key] is not None:
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                return existing_pair.value
            elif existing_pair.next_pair is not None:
                existing_pair = existing_pair.next_pair
            else:
                break
    raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization





Separate Chaining Hash Table

Get item (2)

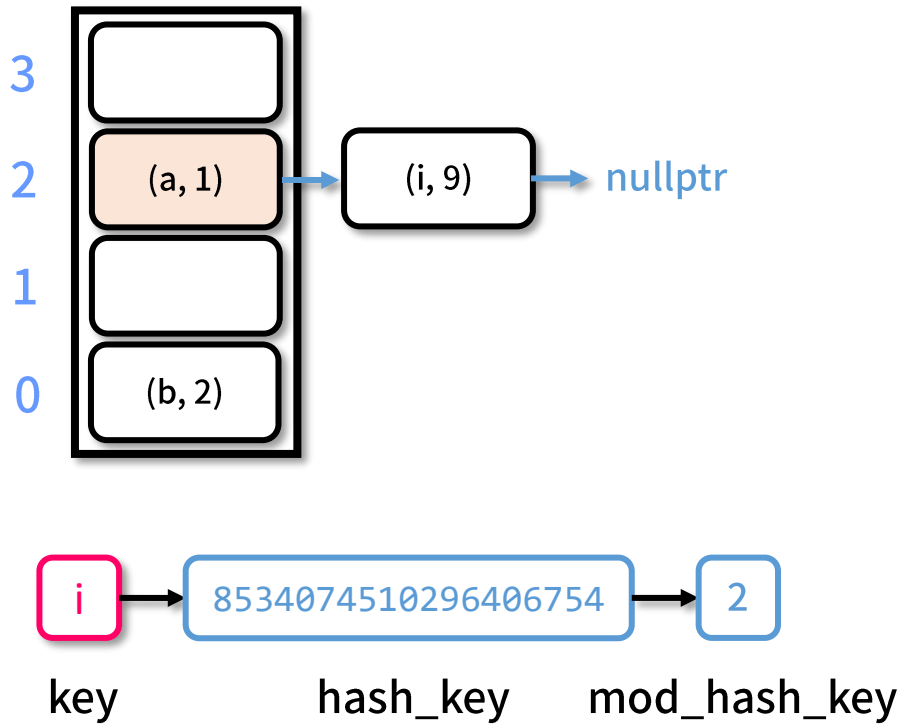
Get Item

```
def __getitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization



Separate Chaining Hash Table

Get item (3)

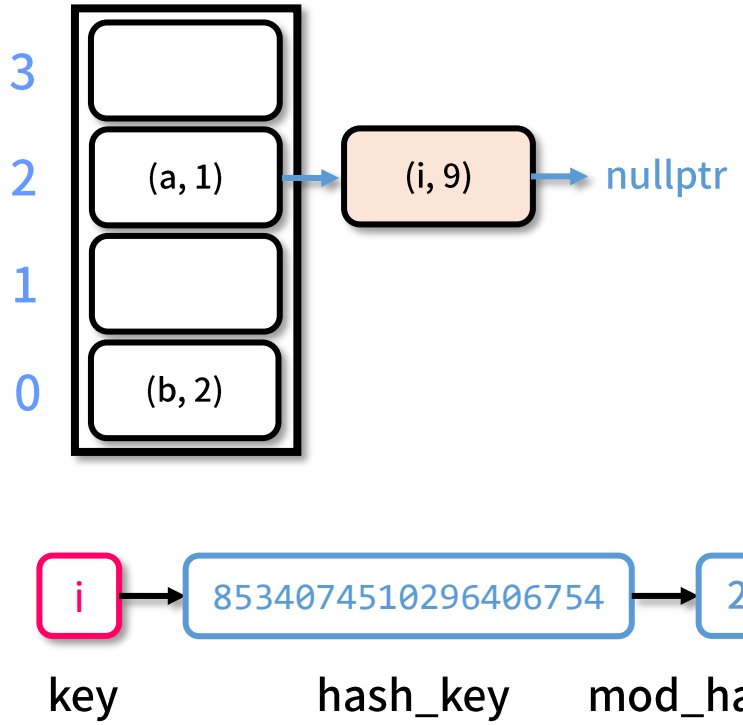
Get Item

```
def __getitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization



Separate Chaining Hash Table

Get item (4)

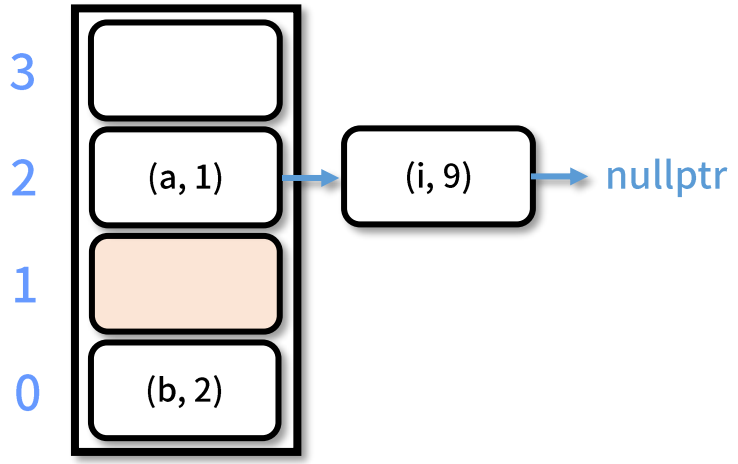
Get Item

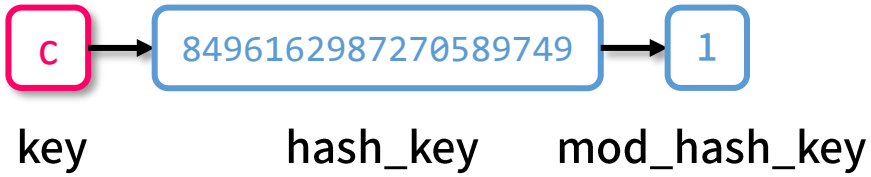
```
def __getitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

```
hash_table['c']
```

Visualization





Separate Chaining Hash Table

Delete item (1)

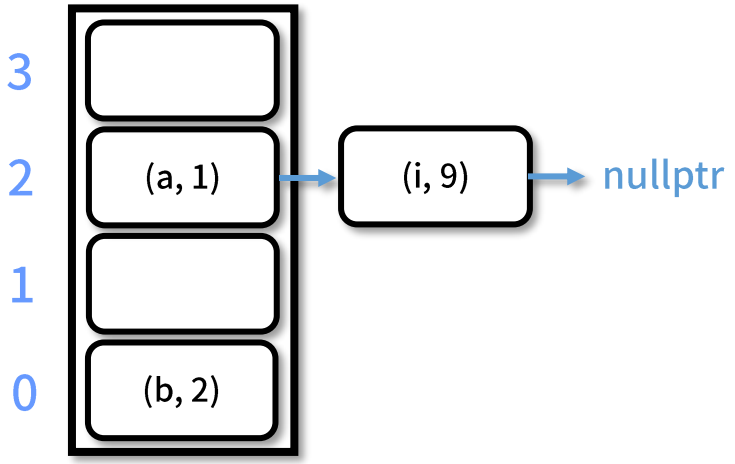
Delete Item

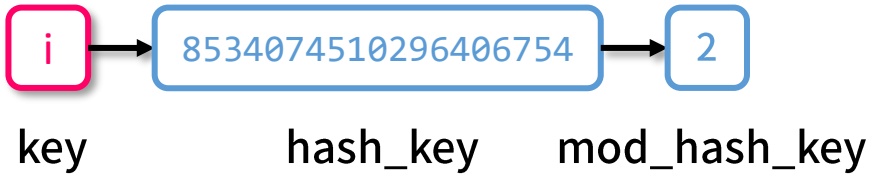
```python
def __delitem__(self, key: Hashable):
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    if self.table[mod_hash_key] is not None:
        previous_pair = None
        existing_pair = self.table[mod_hash_key]
        while True:
            if hash(existing_pair.key) == hash_key:
                if previous_pair is None:
                    self.table[mod_hash_key] = existing_pair.next_pair
                else:
                    previous_pair.next_pair = existing_pair.next_pair
                self.n -= 1
                if self.alpha <= 0.5 and self.size > 4:
                    self.resize(self.size // 2)
                return existing_pair.value
            elif existing_pair.next_pair is not None:
                previous_pair = existing_pair
                existing_pair = existing_pair.next_pair
            else:
                break
    raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization





Separate Chaining Hash Table

Delete item (2)

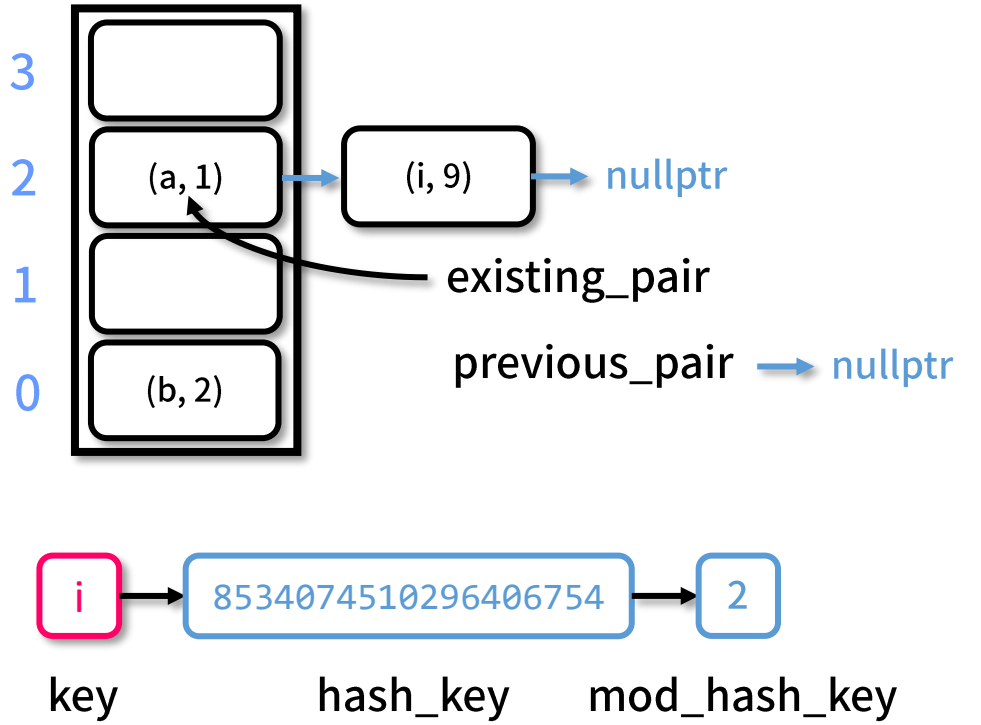
Delete Item

```python
def __delitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    previous_pair = None
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        if previous_pair is None:
          self.table[mod_hash_key] = existing_pair.next_pair
        else:
          previous_pair.next_pair = existing_pair.next_pair
        self.n -= 1
        if self.alpha <= 0.5 and self.size > 4:
          self.resize(self.size // 2)
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        previous_pair = existing_pair
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization



Separate Chaining Hash Table

Delete item (3)

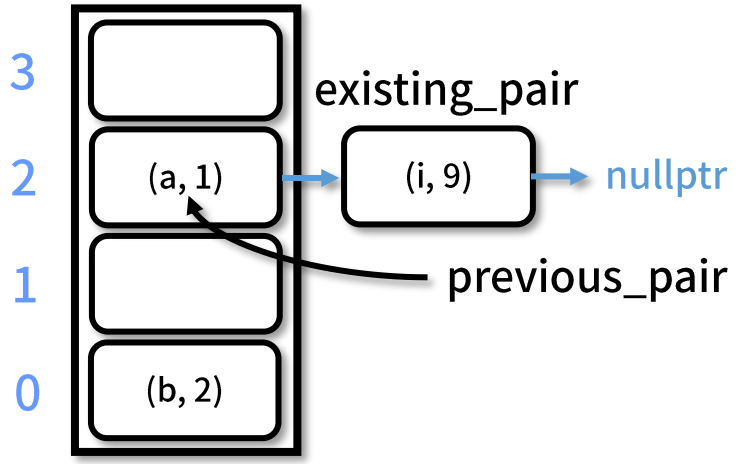
Delete Item

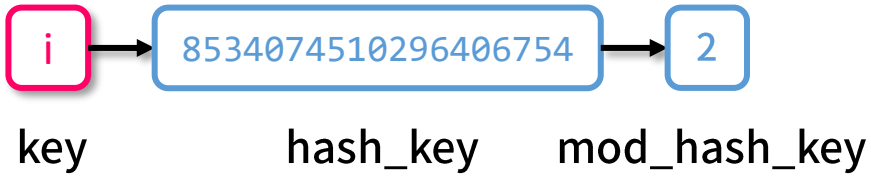
```python
def __delitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    previous_pair = None
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        if previous_pair is None:
          self.table[mod_hash_key] = existing_pair.next_pair
        else:
          previous_pair.next_pair = existing_pair.next_pair
        self.n -= 1
        if self.alpha <= 0.5 and self.size > 4:
          self.resize(self.size // 2)
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        previous_pair = existing_pair
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization





Separate Chaining Hash Table

Delete item (4)

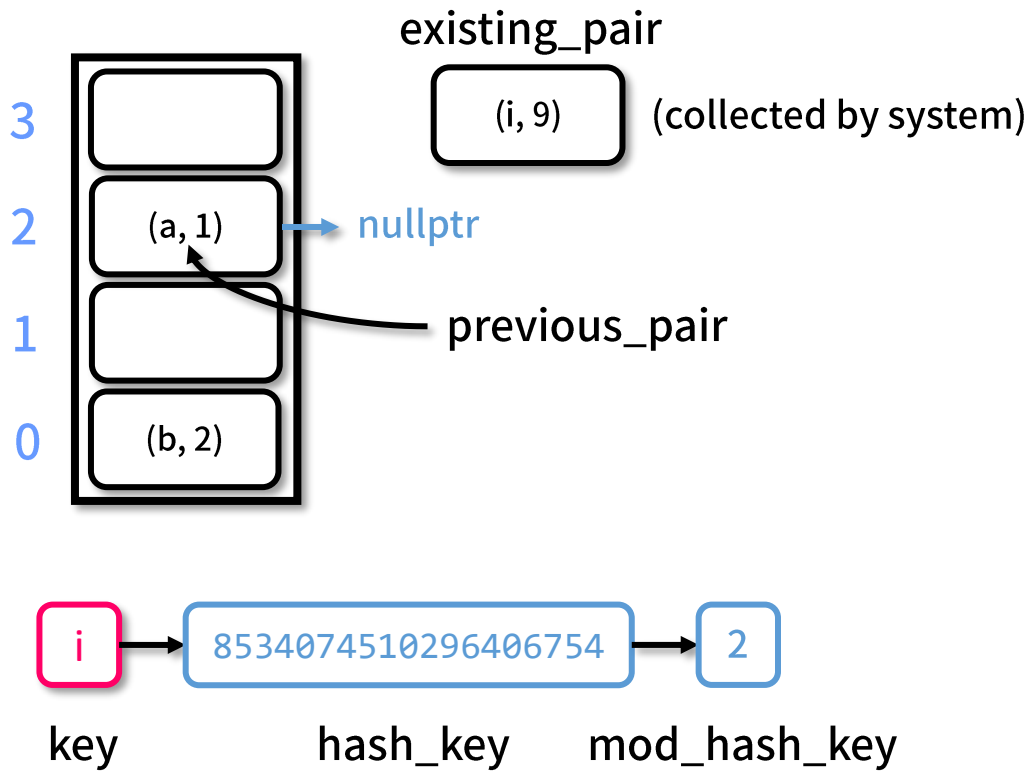
Delete Item

```python
def __delitem__(self, key: Hashable):
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  if self.table[mod_hash_key] is not None:
    previous_pair = None
    existing_pair = self.table[mod_hash_key]
    while True:
      if hash(existing_pair.key) == hash_key:
        if previous_pair is None:
          self.table[mod_hash_key] = existing_pair.next_pair
        else:
          previous_pair.next_pair = existing_pair.next_pair
        self.n -= 1
        if self.alpha <= 0.5 and self.size > 4:
          self.resize(self.size // 2)
        return existing_pair.value
      elif existing_pair.next_pair is not None:
        previous_pair = existing_pair
        existing_pair = existing_pair.next_pair
      else:
        break
  raise KeyError(f'{key}')
```

Command

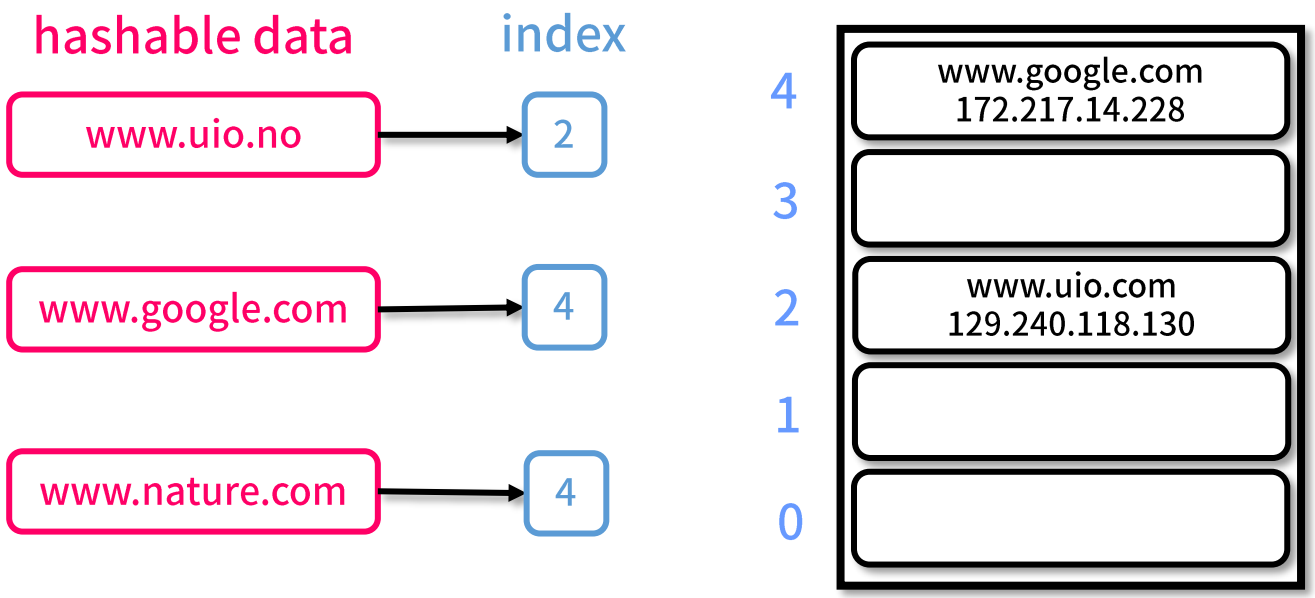
```
del hash_table['i']
```

Visualization



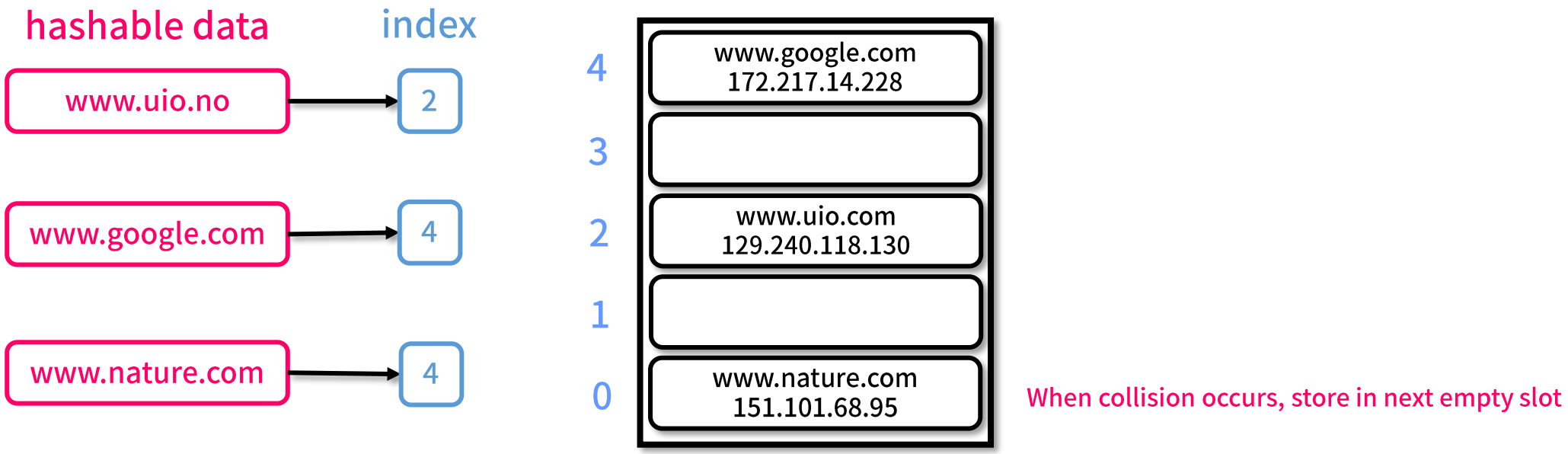
Linear Probing Hash Table (1)

Set Item



Linear Probing Hash Table (2)

Set Item



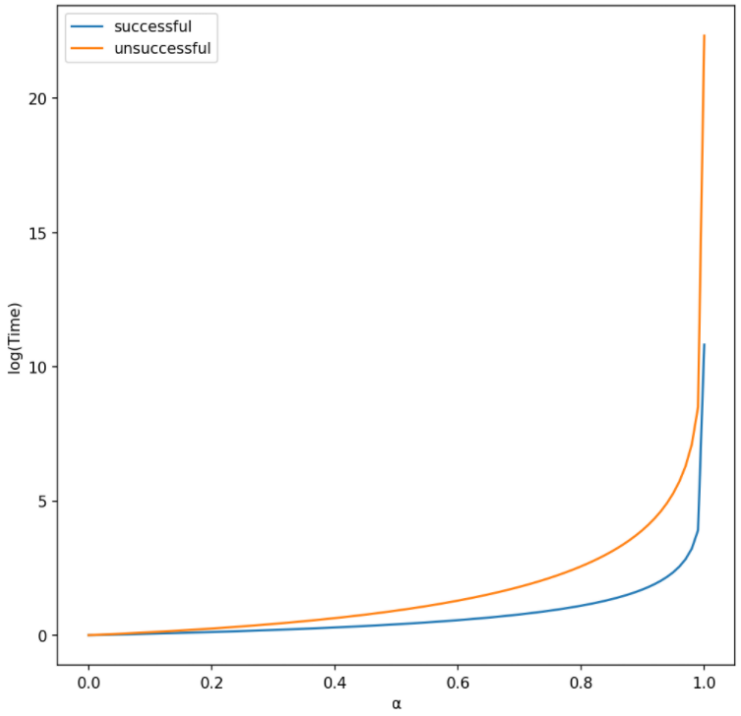
Linear Probing Hash Table (3)

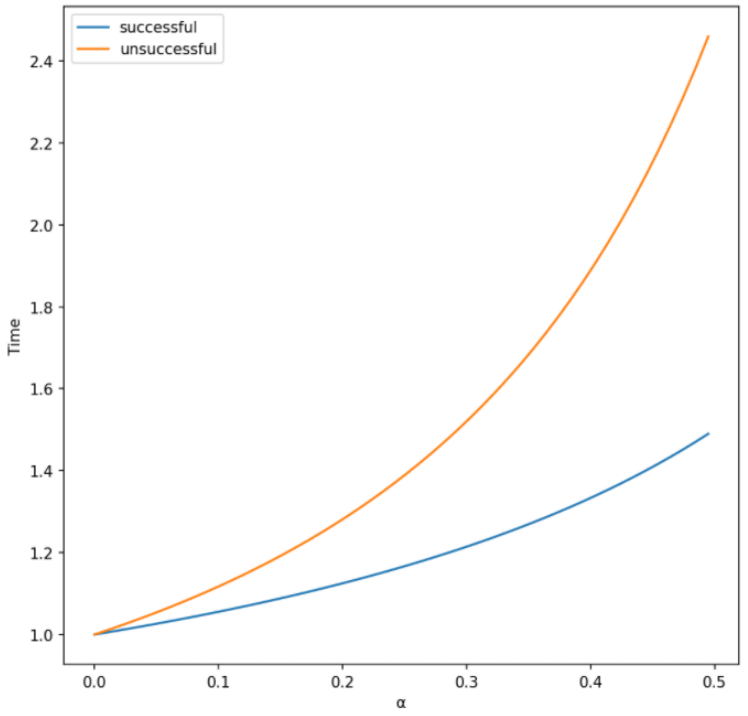
- Computing the time complexity is non-trivial (Knuth 1963).
- Successful

$$\frac{1}{2}(1+(\frac{1}{1-\alpha}))$$

- Unsuccessful

$$\frac{1}{2}(1+(\frac{1}{1-\alpha})^2)$$





Linear Probing Hash Table (4)

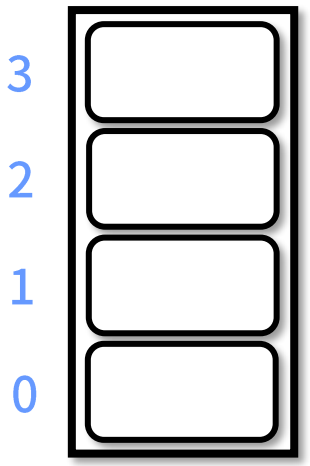
Initialization

```
class LinearProbingHashTable(MutableMapping):
  def __init__(self, size: int = 4):
    self.n = 0
    self.n_tombstone = 0
    self.size = size
    self.table = Array(size)

  def __setitem__(self, key: Hashable, value: Any) -> None: ···
      insert or set item in hash table with key, value pair
  def __getitem__(self, key: Hashable) -> Any: ···
      get the value from provided key
  def __delitem__(self, key: Hashable) -> None: ···
      delete key, value pair in the hash table
  def __iter__(self) -> Iterator: ···
      iterator interface
  def __len__(self) -> int: ···

  def __str__(self) -> str: ···

  def resize(self, size: int) -> None: ···
      resize the hash table
  @property
  def alpha(self) -> float: ···
      get loading factor
  def clear(self) -> None: ···
      reset hash table
  def update(self, iterable: Iterable) -> None: ···
      merge the hash table with provided iterable map
```

Command

```
hash_table = LinearProbingHashTable()
```

Visualization



Linear Probing Hash Table

Set item (1)

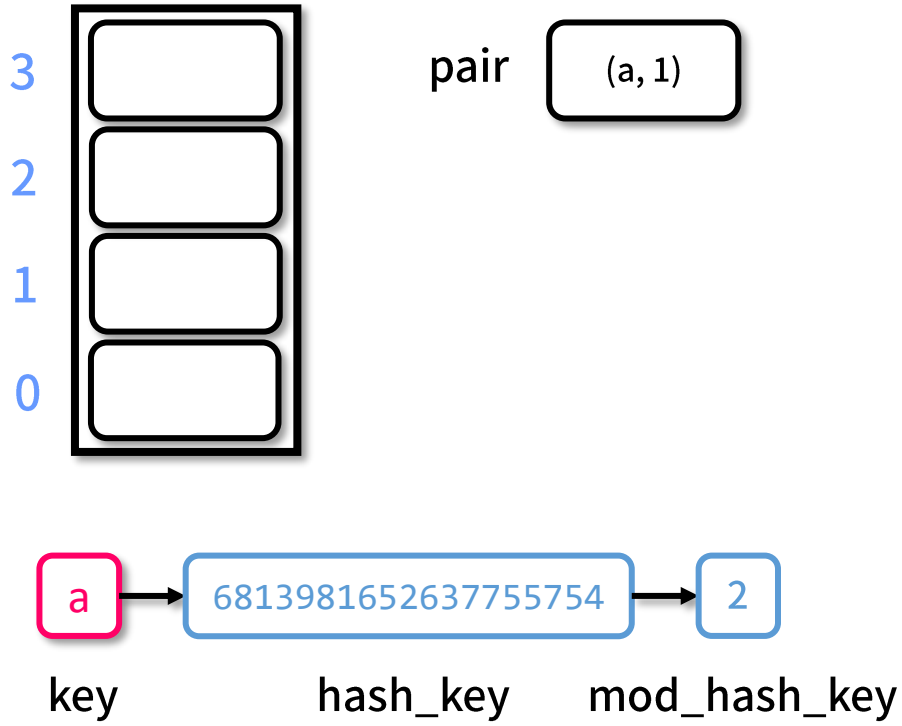
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    candidate_index = mod_hash_key
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.size * 2)
            break
        else:
            candidate_index += 1
        if candidate_index == self.size:
            candidate_index = 0
        if candidate_index == mod_hash_key:
            break
```

Command

```python
hash_table['a'] = 1
```

Visualization



Linear Probing Hash Table

Set item (2)

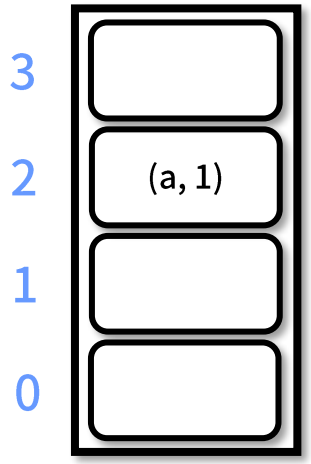
Set Item

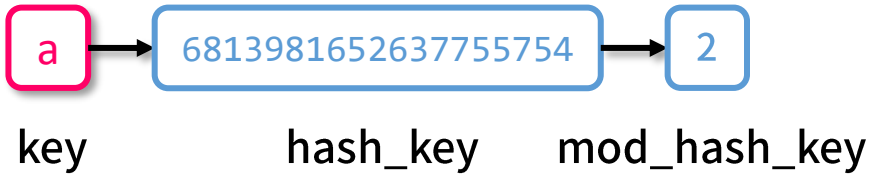
```python
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.size * 2)
      break
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if candidate_index == mod_hash_key:
      break
```

Command

```
hash_table['a'] = 1
```

Visualization





Linear Probing Hash Table

Set item (3)

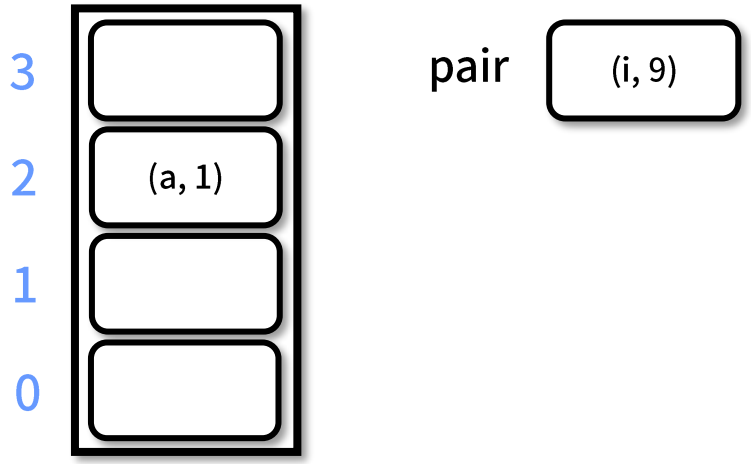
Set Item

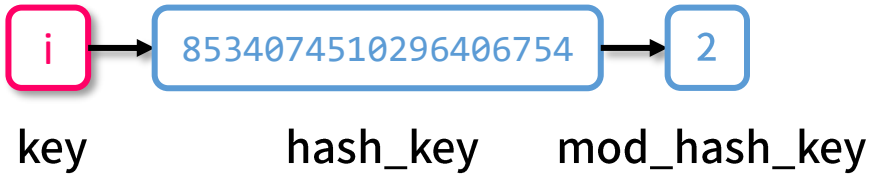
```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    candidate_index = mod_hash_key
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.size * 2)
            break
        else:
            candidate_index += 1
        if candidate_index == self.size:
            candidate_index = 0
        if candidate_index == mod_hash_key:
            break
```

Command

```
hash_table['i'] = 9
```

Visualization





Linear Probing Hash Table

Set item (4)

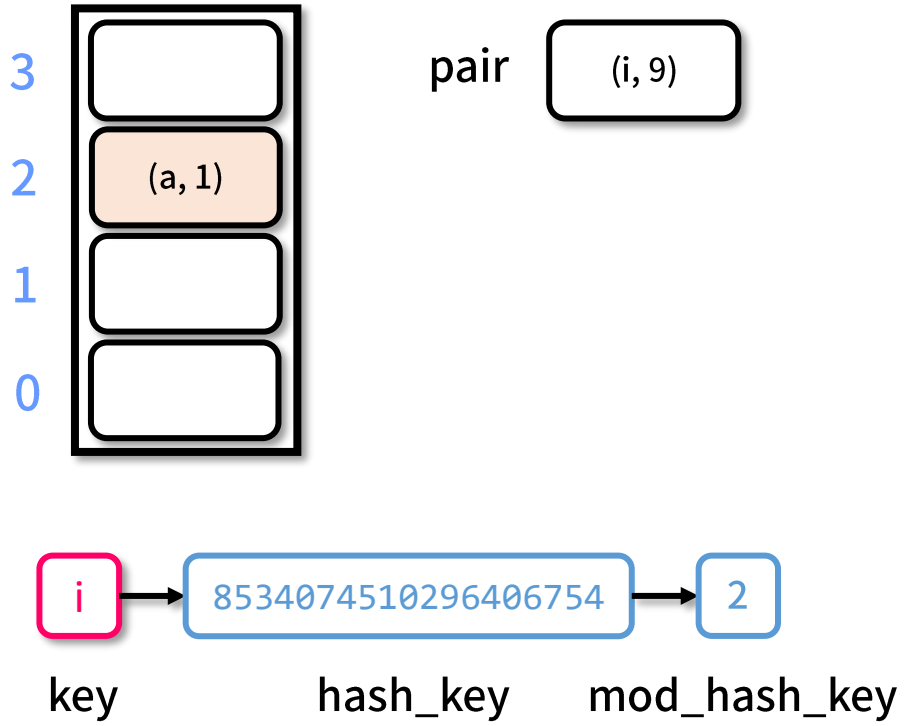
Set Item

```
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.size * 2)
      break
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if candidate_index == mod_hash_key:
      break
```

Command

```
hash_table['i'] = 9
```

Visualization



Linear Probing Hash Table

Set item (5)

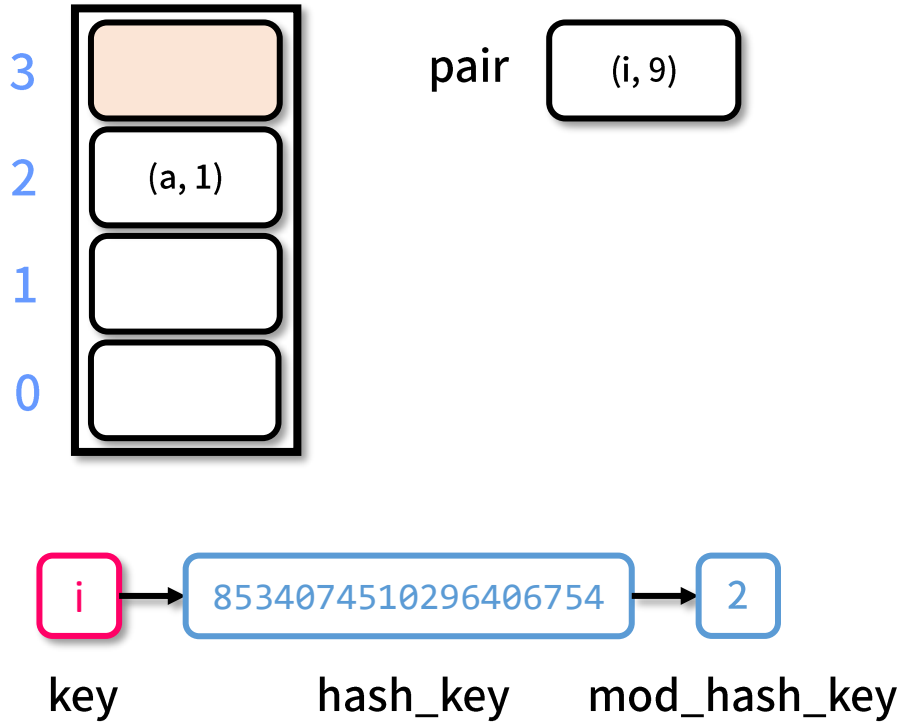
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.size * 2)
      break
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if candidate_index == mod_hash_key:
      break
```

Command

```
hash_table['i'] = 9
```

Visualization



Linear Probing Hash Table

Set item (6)

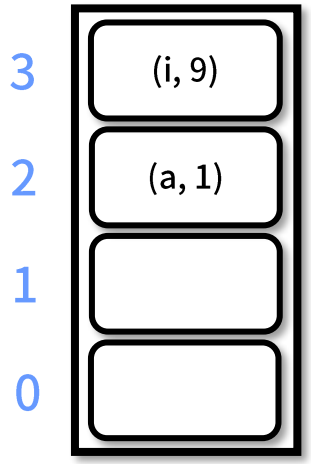
Set Item

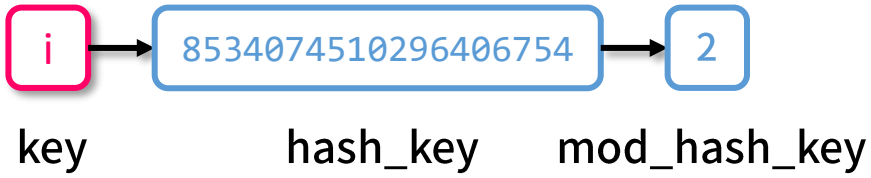
```
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.size * 2)
      break
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if candidate_index == mod_hash_key:
      break
```

Command

```
hash_table['i'] = 9
```

Visualization





Linear Probing Hash Table

Get item (1)

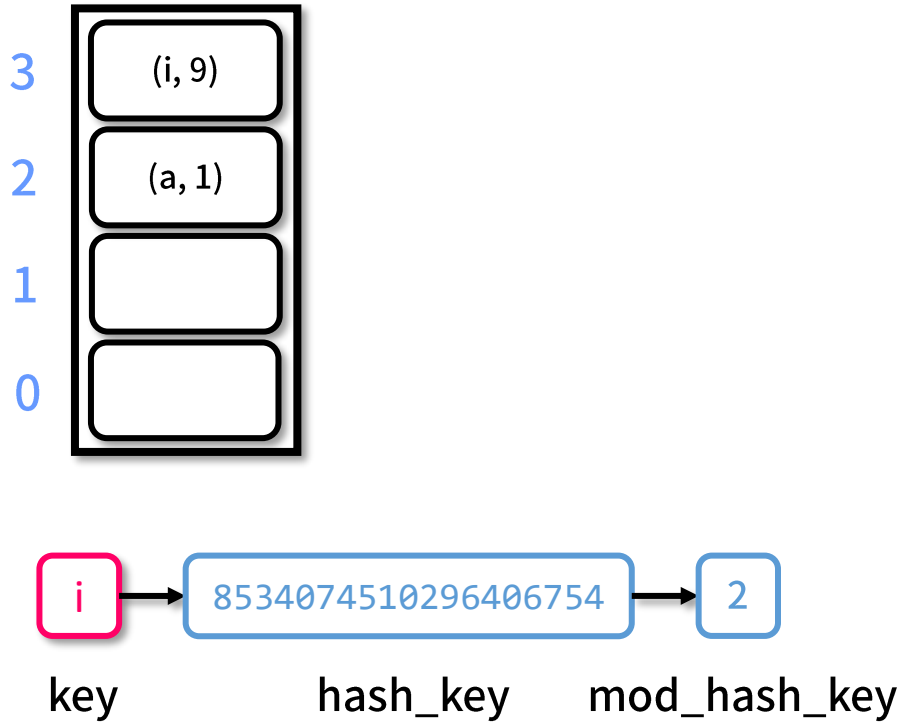
Get Item

```
def __getitem__(self, key: Hashable) -> Any:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      return self.table[candidate_index].value
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization



Linear Probing Hash Table

Get item (2)

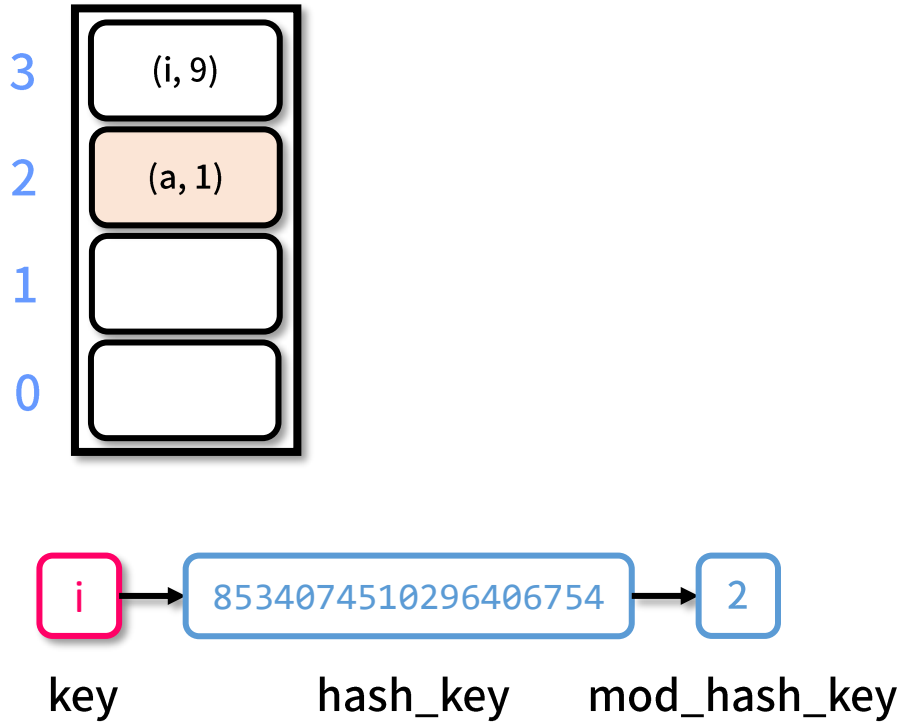
Get Item

```
def __getitem__(self, key: Hashable) -> Any:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      return self.table[candidate_index].value
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization



Linear Probing Hash Table

Get item (3)

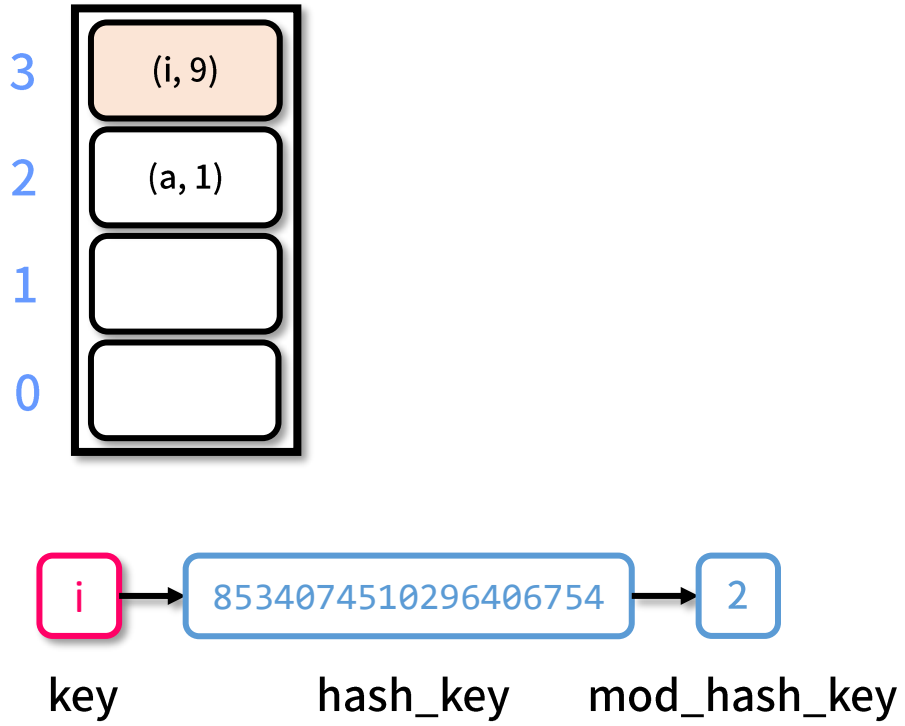
Get Item

```python
def __getitem__(self, key: Hashable) -> Any:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      return self.table[candidate_index].value
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

```
hash_table['i']
```

Visualization



Linear Probing Hash Table

Get item (4)

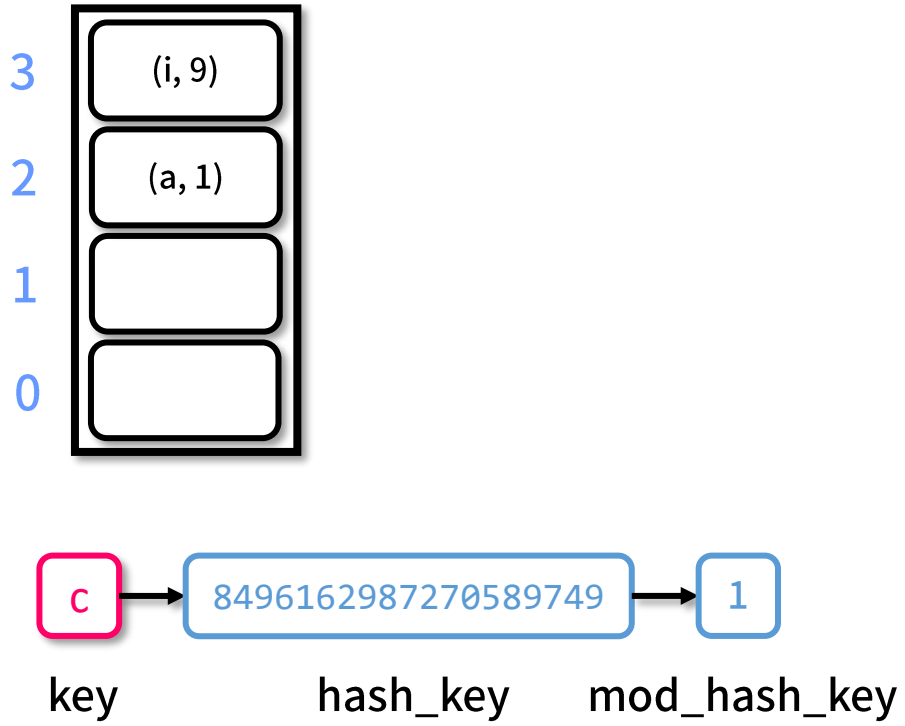
Get Item

```
def __getitem__(self, key: Hashable) -> Any:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    candidate_index = mod_hash_key

    while True:
        if (self.table[candidate_index] is not None and
            not isinstance(self.table[candidate_index], Tombstone) and
            hash(self.table[candidate_index].key) == hash_key):
            return self.table[candidate_index].value
        else:
            candidate_index += 1
        if candidate_index == self.size:
            candidate_index = 0
        if self.table[candidate_index] is None:
            break
    raise KeyError(f'{key}')
```

Command

```
hash_table['c']
```

Visualization



Linear Probing Hash Table

Get item (5)

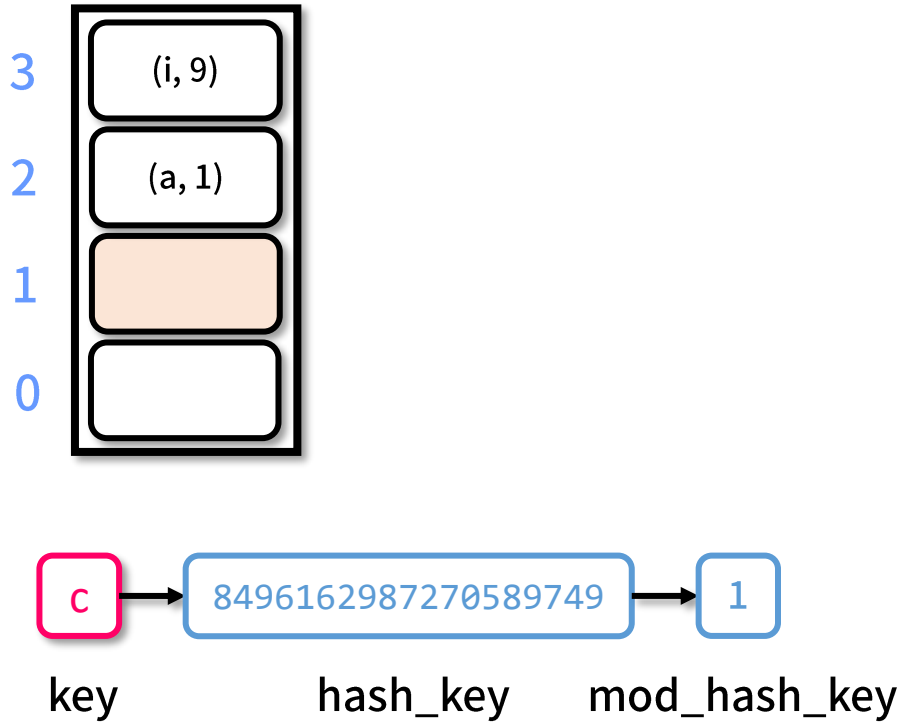
Get Item

```
def __getitem__(self, key: Hashable) -> Any:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      return self.table[candidate_index].value
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

```
hash_table['c']
```

Visualization



Linear Probing Hash Table

Delete item (1)

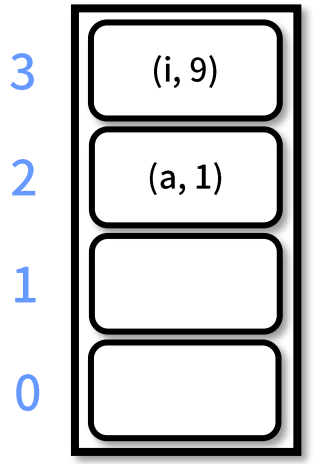
Delete Item

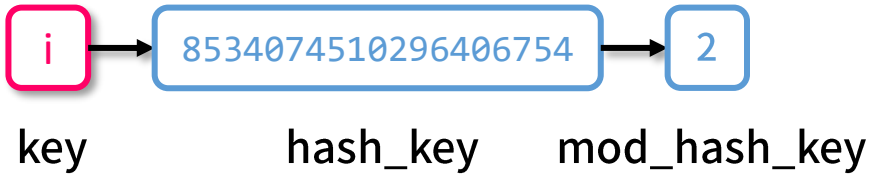
```python
def __delitem__(self, key: Hashable) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    candidate_index = mod_hash_key

    while True:
        if (self.table[candidate_index] is not None and
                not isinstance(self.table[candidate_index], Tombstone) and
                hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = Tombstone()
            self.n -= 1
            self.n_tombstone += 1
            if (self.alpha - self.n_tombstone / self.size < 0.25 and
                    self.size > 4):
                self.resize(self.size // 2)
            return
        else:
            candidate_index += 1
        if candidate_index == self.size:
            candidate_index = 0
        if self.table[candidate_index] is None:
            break
    raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization





Linear Probing Hash Table

Delete item (2)

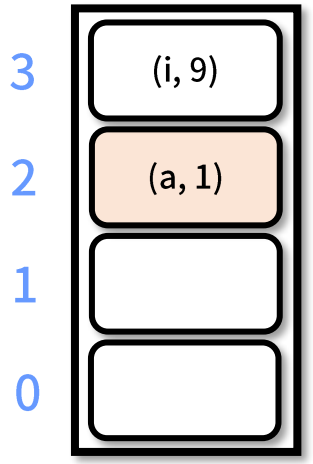
Delete Item

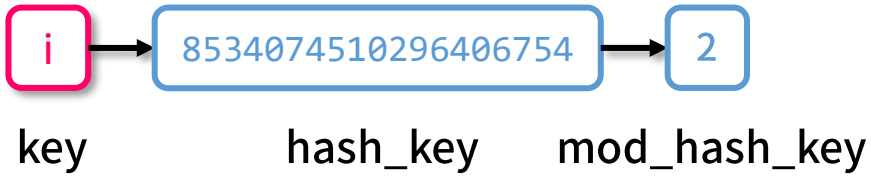
```
def __delitem__(self, key: Hashable) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = Tombstone()
      self.n -= 1
      self.n_tombstone += 1
      if (self.alpha - self.n_tombstone / self.size < 0.25 and
          self.size > 4):
        self.resize(self.size // 2)
      return
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization





Linear Probing Hash Table

Delete item (3)

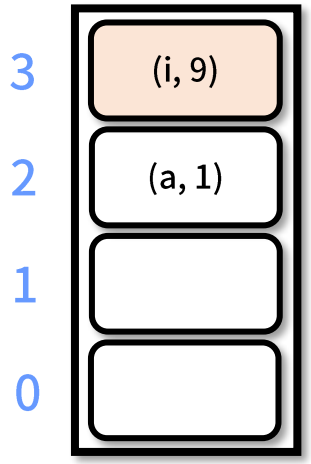
Delete Item

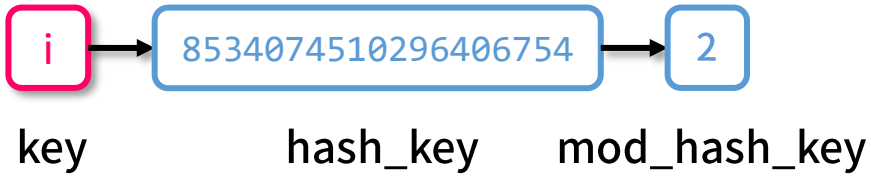
```python
def __delitem__(self, key: Hashable) -> None:
    hash_key = hash(key)
    mod_hash_key = hash_key % self.size
    candidate_index = mod_hash_key

    while True:
        if (self.table[candidate_index] is not None and
                not isinstance(self.table[candidate_index], Tombstone) and
                hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = Tombstone()
            self.n -= 1
            self.n_tombstone += 1
            if (self.alpha - self.n_tombstone / self.size < 0.25 and
                    self.size > 4):
                self.resize(self.size // 2)
            return
        else:
            candidate_index += 1
        if candidate_index == self.size:
            candidate_index = 0
        if self.table[candidate_index] is None:
            break
    raise KeyError(f'{key}')
```

Command

```
del hash_table['i']
```

Visualization





Linear Probing Hash Table

Delete item (4)

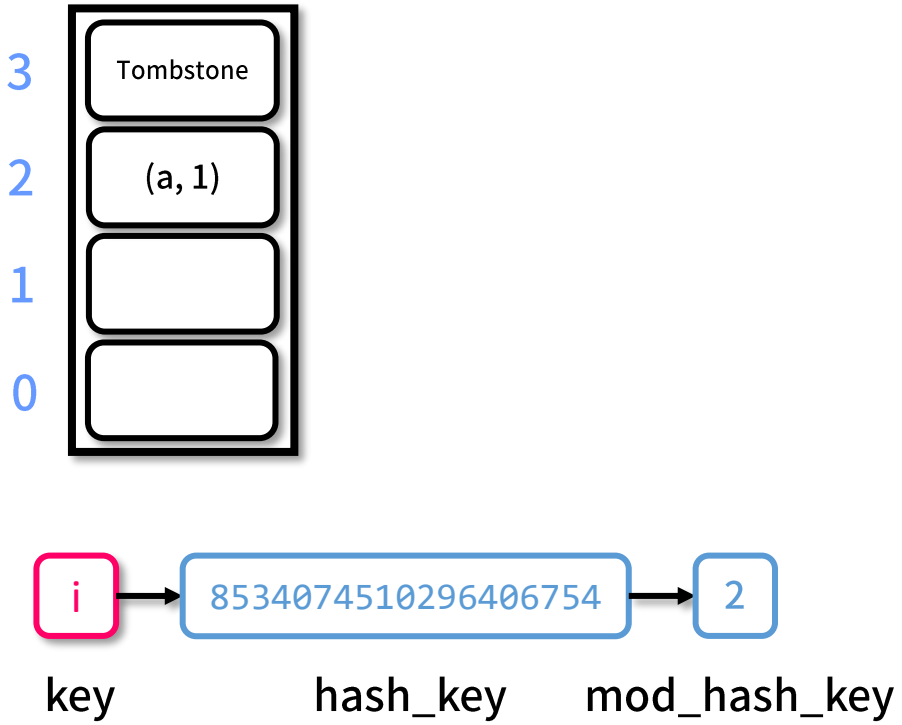
Delete Item

```
def __delitem__(self, key: Hashable) -> None:
  hash_key = hash(key)
  mod_hash_key = hash_key % self.size
  candidate_index = mod_hash_key

  while True:
    if (self.table[candidate_index] is not None and
        not isinstance(self.table[candidate_index], Tombstone) and
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = Tombstone()
      self.n -= 1
      self.n_tombstone += 1
      if (self.alpha - self.n_tombstone / self.size < 0.25 and
          self.size > 4):
        self.resize(self.size // 2)
      return
    else:
      candidate_index += 1
    if candidate_index == self.size:
      candidate_index = 0
    if self.table[candidate_index] is None:
      break
  raise KeyError(f'{key}')
```

Command

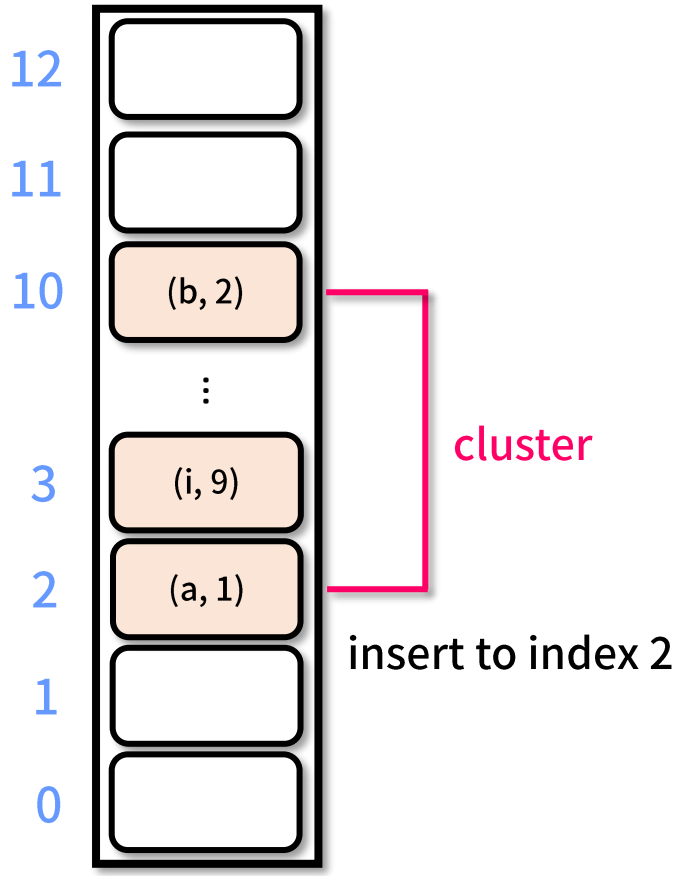
```
del hash_table['i']
```

Visualization



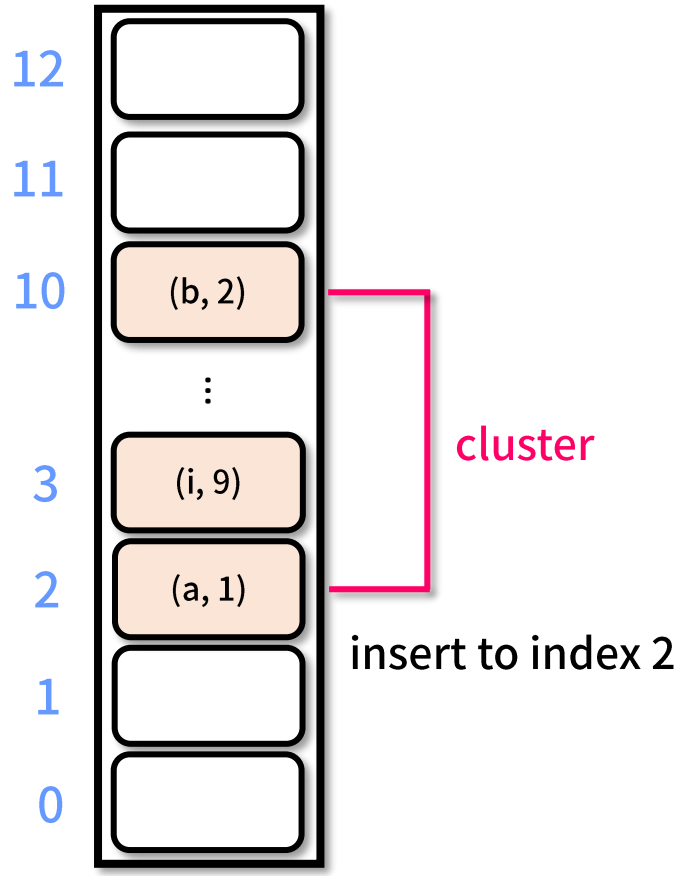
Problem for Linear Probing Hash Table (1)

- The index increase by one every time a collision occurs.
- Could form clusters of objects in the array.
- How to prevent the clusters



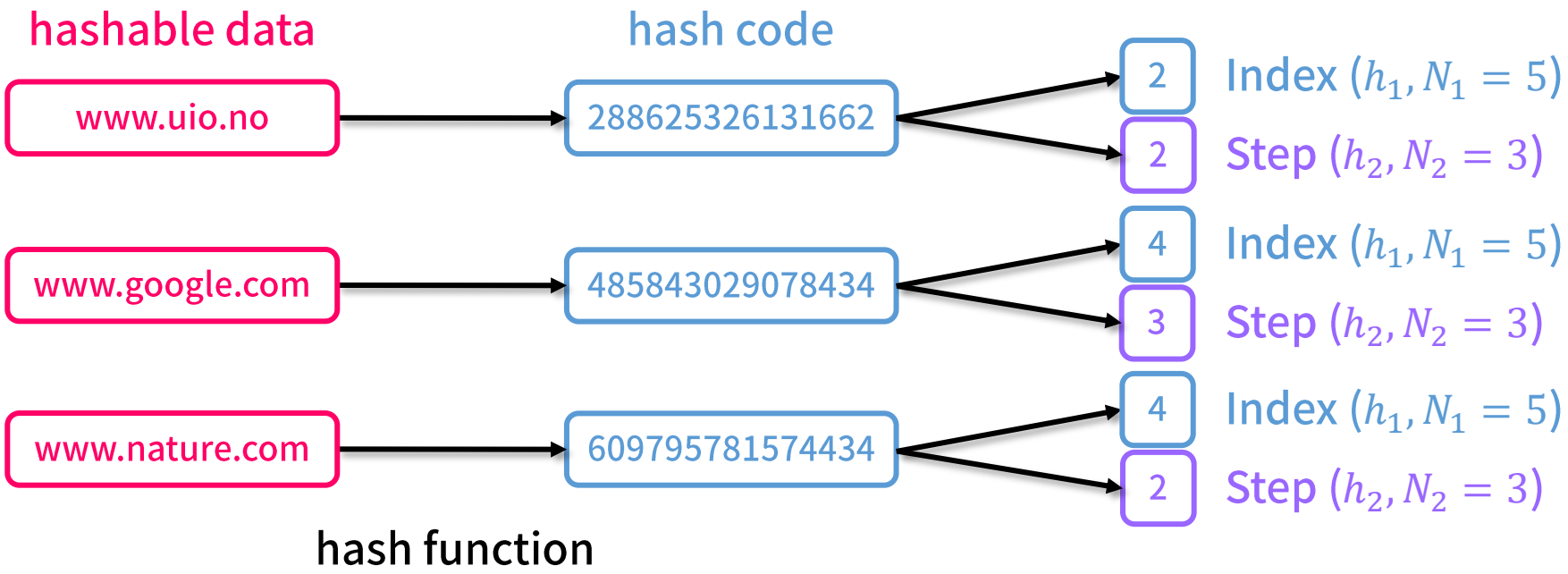
Problem for Linear Probing Hash Table (2)

- The index increase by one every time a collision occurs.
- Could form clusters of objects in the array.
- How to prevent the clusters
- **Randomize the step based on a specific rule.**



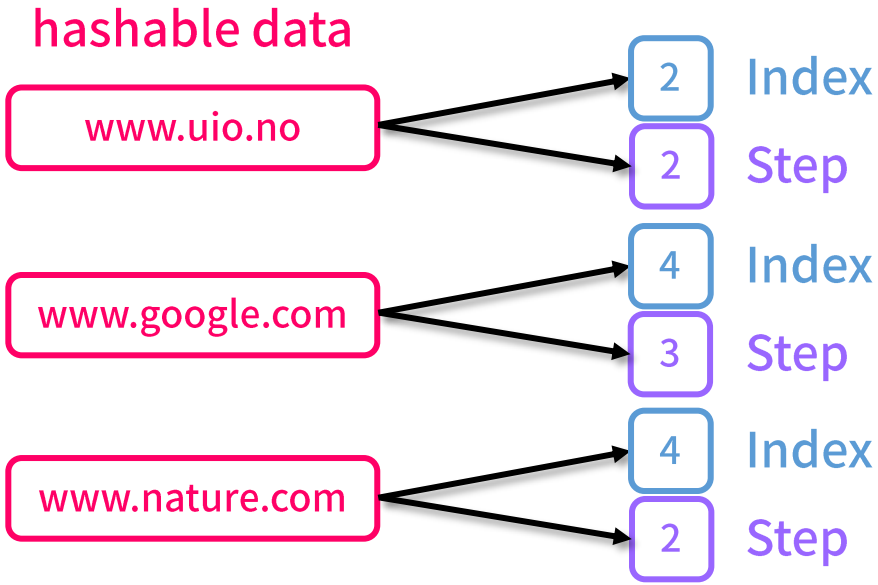
Double Hashing Hash Table (1)

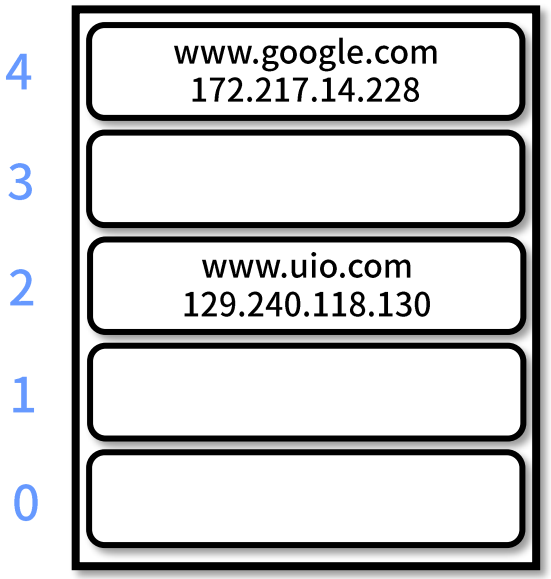
$h_1(x) = x \bmod$
$h_2(x) = N_2 - (x \bmod N_2)$ where $N_2 < N_1$



Double Hashing Hash Table (2)

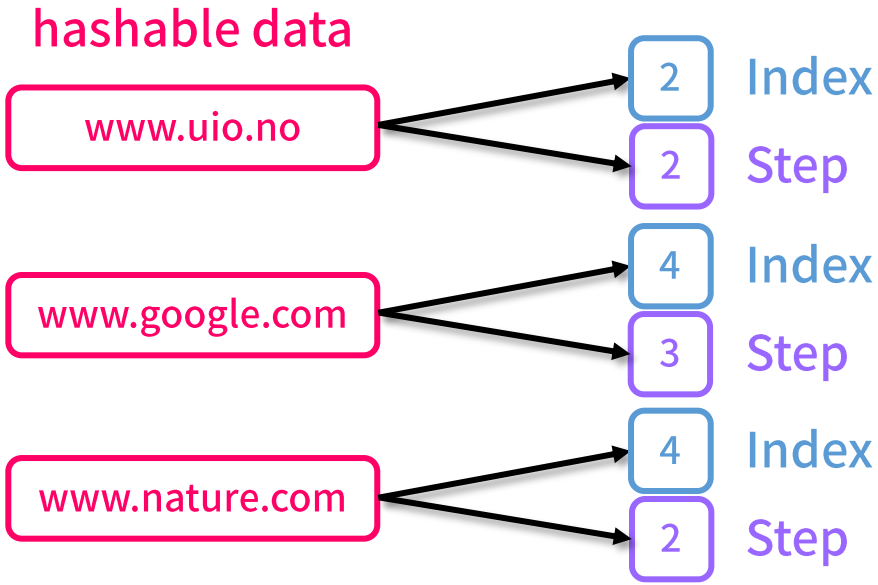
$$h_1(x) = x \bmod$$
$$h_2(x) = N_2 - (x \bmod N_2) \text{ where } N_2 < N_1$$

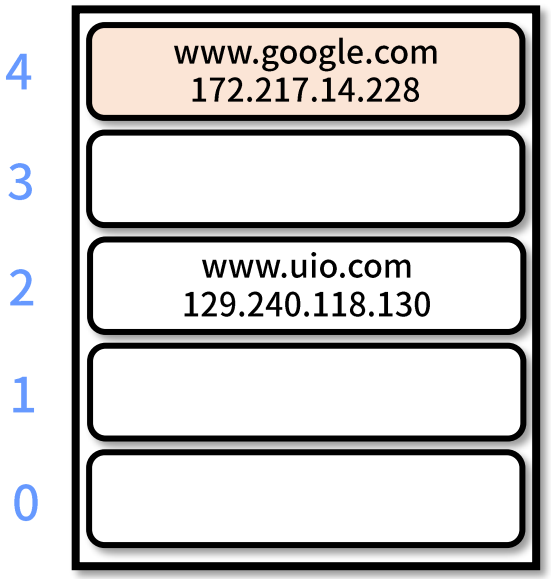




Double Hashing Hash Table (3)

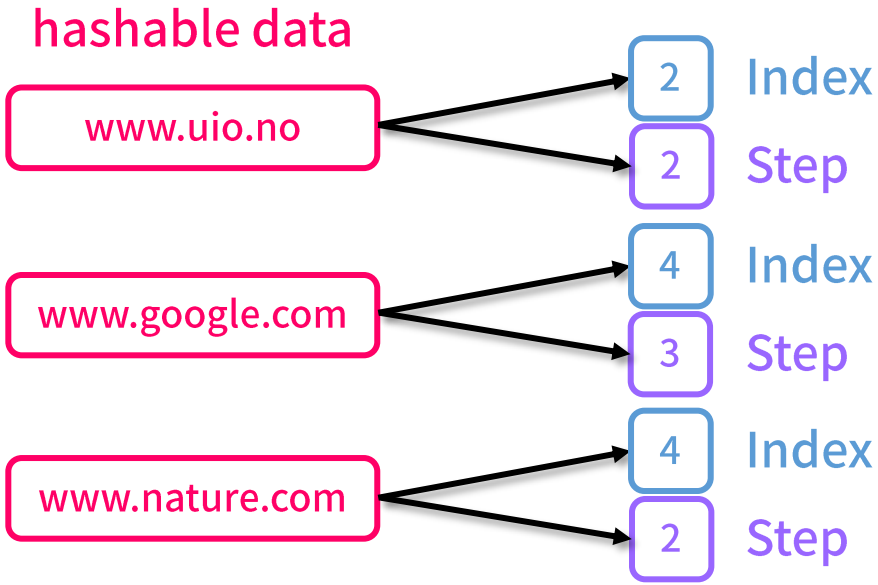
$$h_1(x) = x \bmod$$
$$h_2(x) = N_2 - (x \bmod N_2) \text{ where } N_2 < N_1$$

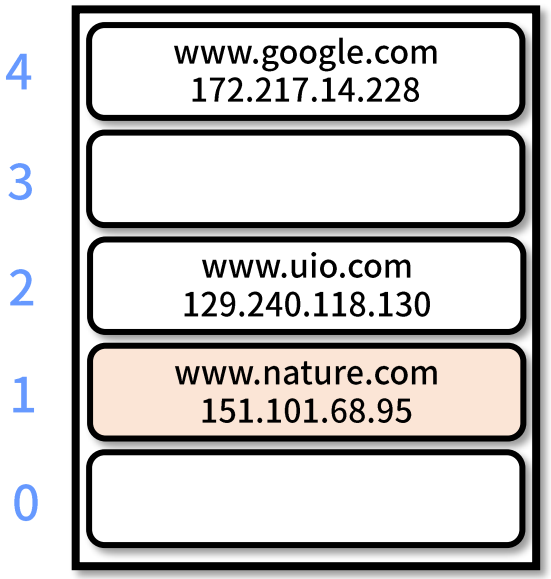




Double Hashing Hash Table (4)

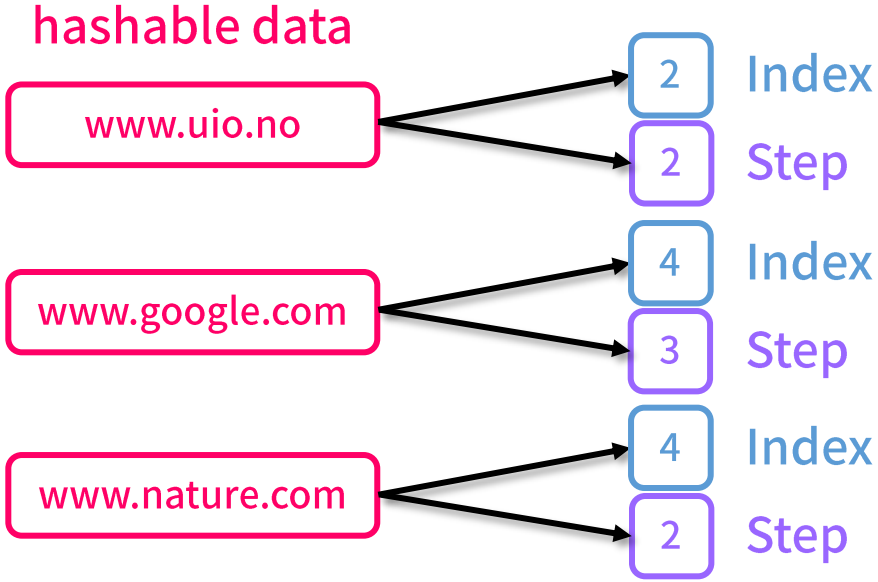
$$h_1(x) = x \bmod$$
$$h_2(x) = N_2 - (x \bmod N_2) \text{ where } N_2 < N_1$$

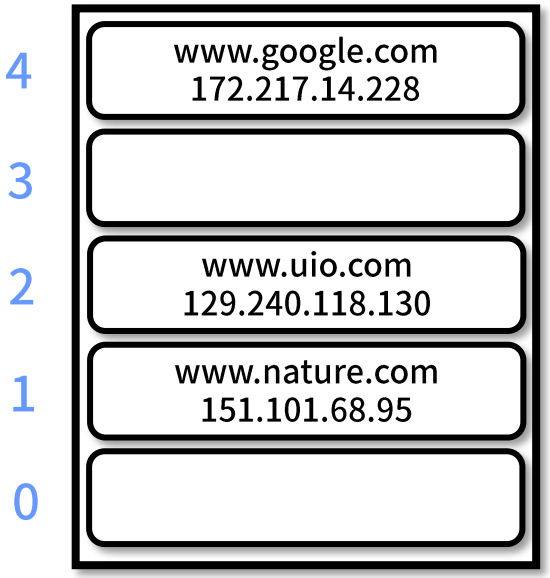




Double Hashing Hash Table (5)

$$h_1(x) = x \bmod$$
$$h_2(x) = N_2 - (x \bmod N_2) \text{ where } N_2 < N_1$$





array size needs to be a prime number

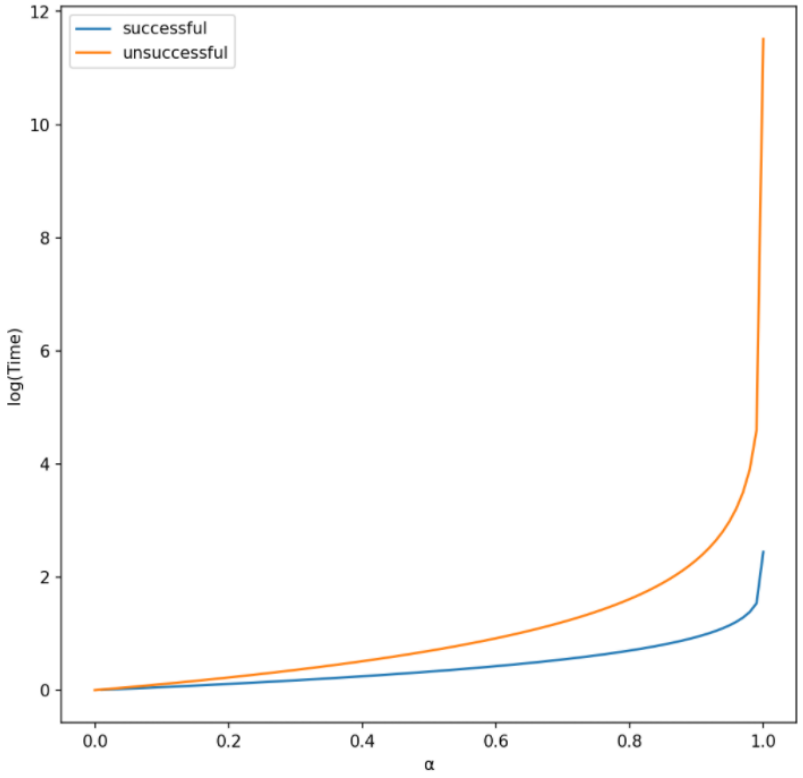
Double Hashing Hash Table (6)

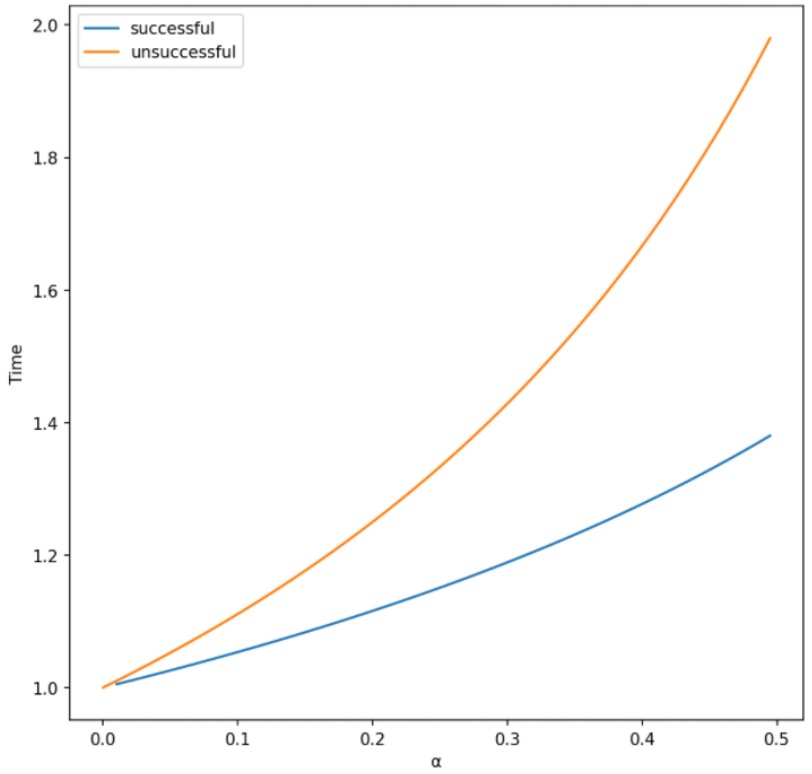
- Computing the time complexity is non-trivial (Guibas and Szemeredi 1978).
- Successful

$$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$$

- Unsuccessful

$$\frac{1}{1-\alpha}$$





Double Hashing Hash Table (7)

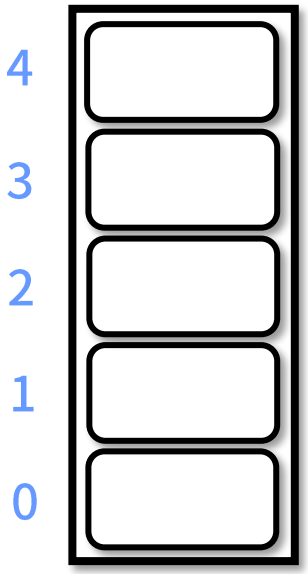
Initialization

```
class DoubleHashingHashTable(MutableMapping):
  def __init__(self, size: int = 5):
    self.n = 0
    self.n_tombstone = 0
    self.size = size
    self.table = Array(size)

  def __setitem__(self, key: Hashable, value: Any) -> None: ···
      insert or set item in hash table with key, value pair
  def __getitem__(self, key: Hashable) -> Any: ···
      get the value from provided key
  def __delitem__(self, key: Hashable) -> None: ···
      delete key, value pair in the hash table
  def __iter__(self) -> Iterator: ···
      iterator interface
  def __len__(self) -> int: ···

  def __str__(self) -> str: ···

  def resize(self, size: int) -> None: ···

  @property
  def alpha(self) -> float: ···
      resize the hash table
  @property
  def alpha_tombstone(self) -> float: ···
      get loading factor
  def clear(self) -> None: ···
      reset hash table
  def update(self, iterable: Iterable) -> None: ···
      merge the hash table with provided iterable map
  def get_next_prime_number_size(self, expand: bool = True) -> int: ···
      compute prime number for array size
```

Command

```
hash_table = DoubleHashingHashTable()
```

Visualization



Linear Probing Hash Table

Set item (1)

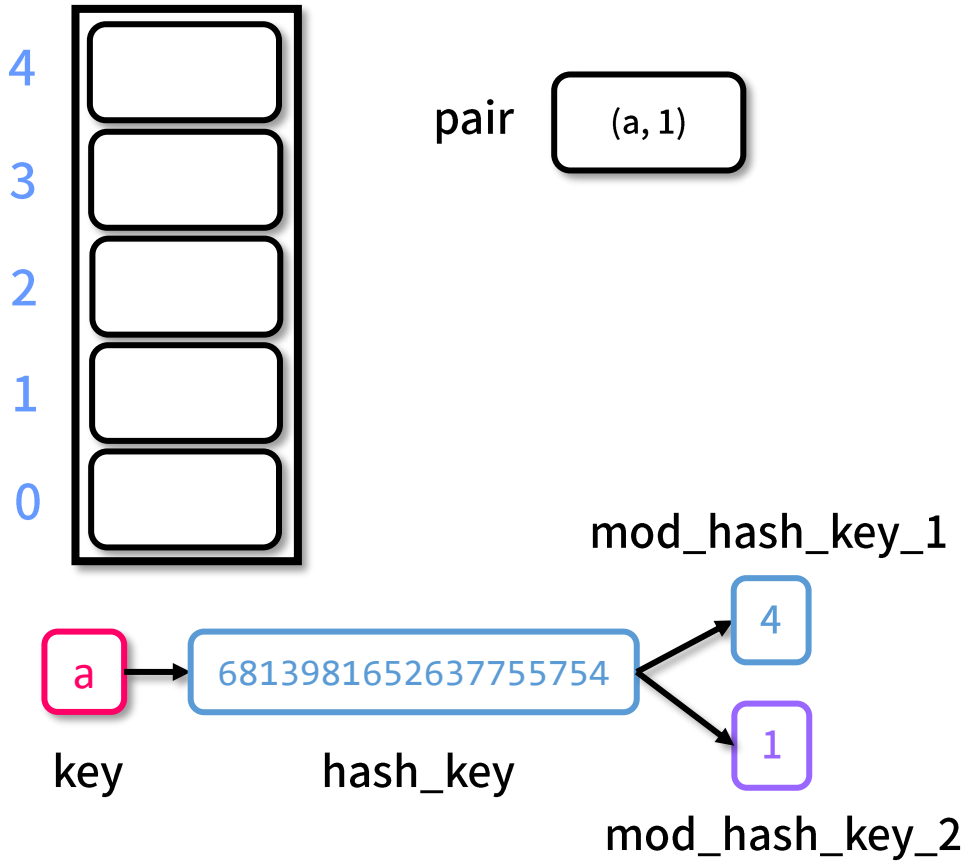
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_base_1 = self.size
    mod_base_2 = (self.size // 2 + 1)
    mod_hash_key_1 = hash_key % mod_base_1
    mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
    candidate_index = mod_hash_key_1
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.get_next_prime_number_size(expand=True))
            break
        else:
            candidate_index += mod_hash_key_2
        if candidate_index >= self.size:
            candidate_index -= self.size
```

Command

```python
hash_table['a'] = 1
```

Visualization



Linear Probing Hash Table

Set item (2)

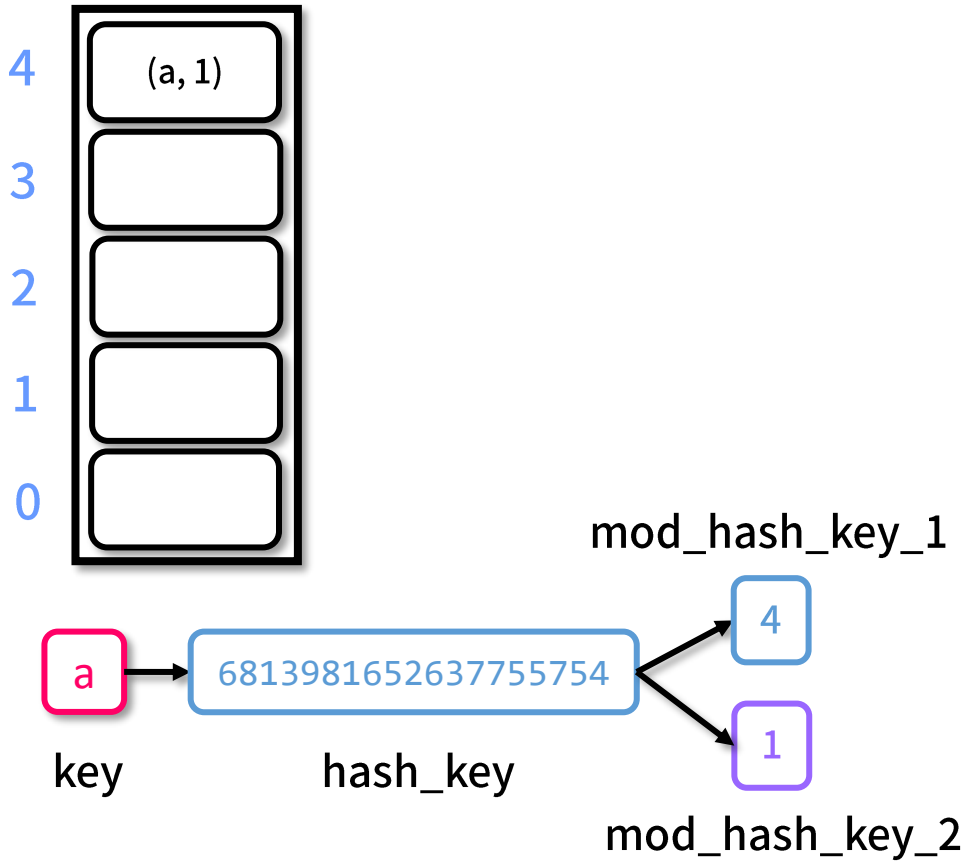
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_base_1 = self.size
  mod_base_2 = (self.size // 2 + 1)
  mod_hash_key_1 = hash_key % mod_base_1
  mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
  candidate_index = mod_hash_key_1
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.get_next_prime_number_size(expand=True))
      break
    else:
      candidate_index += mod_hash_key_2
    if candidate_index >= self.size:
      candidate_index -= self.size
```

Command

```
hash_table['a'] = 1
```

Visualization



Linear Probing Hash Table

Set item (3)

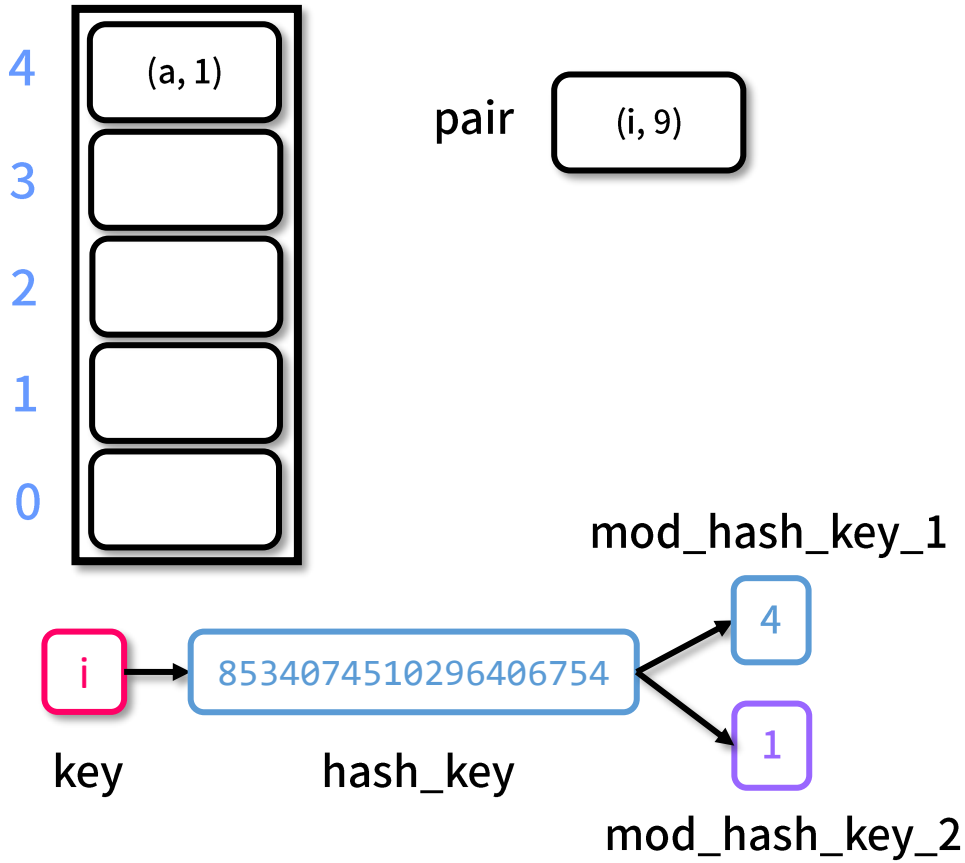
Set Item

```
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_base_1 = self.size
    mod_base_2 = (self.size // 2 + 1)
    mod_hash_key_1 = hash_key % mod_base_1
    mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
    candidate_index = mod_hash_key_1
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.get_next_prime_number_size(expand=True))
            break
        else:
            candidate_index += mod_hash_key_2
        if candidate_index >= self.size:
            candidate_index -= self.size
```

Command

```
hash_table['i'] = 9
```

Visualization



Linear Probing Hash Table

Set item (4)

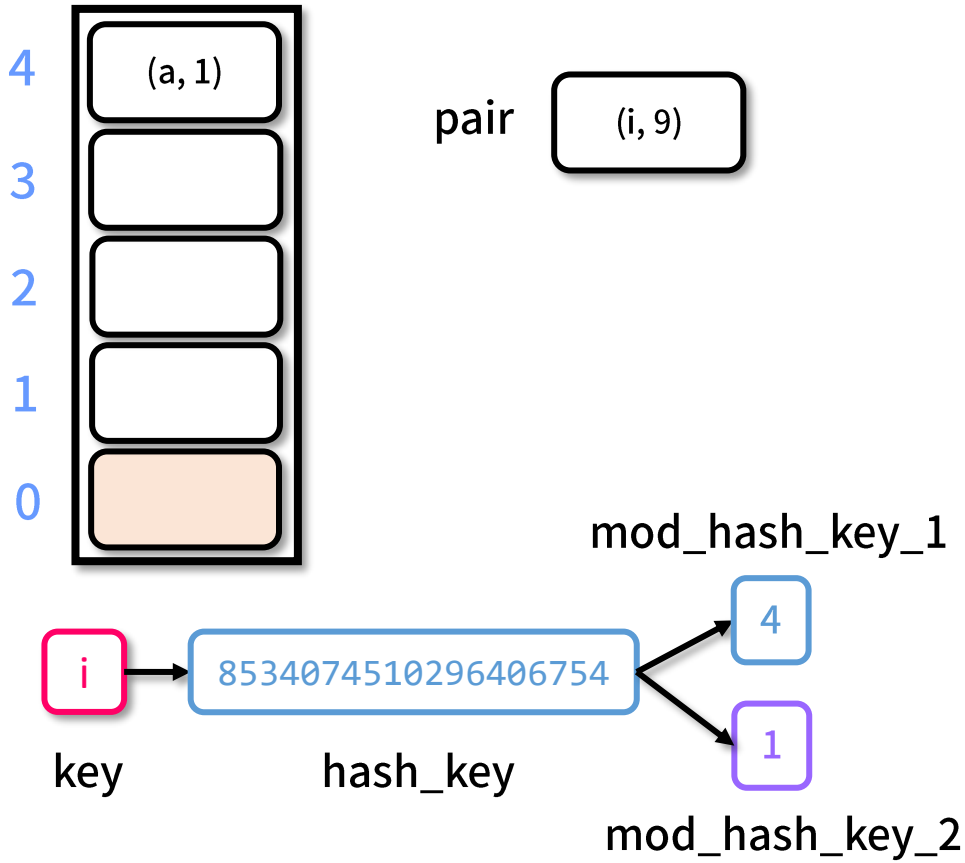
Set Item

```
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_base_1 = self.size
  mod_base_2 = (self.size // 2 + 1)
  mod_hash_key_1 = hash_key % mod_base_1
  mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
  candidate_index = mod_hash_key_1
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.get_next_prime_number_size(expand=True))
      break
    else:
      candidate_index += mod_hash_key_2
    if candidate_index >= self.size:
      candidate_index -= self.size
```

Command

```
hash_table['i'] = 9
```

Visualization



Linear Probing Hash Table

Set item (5)

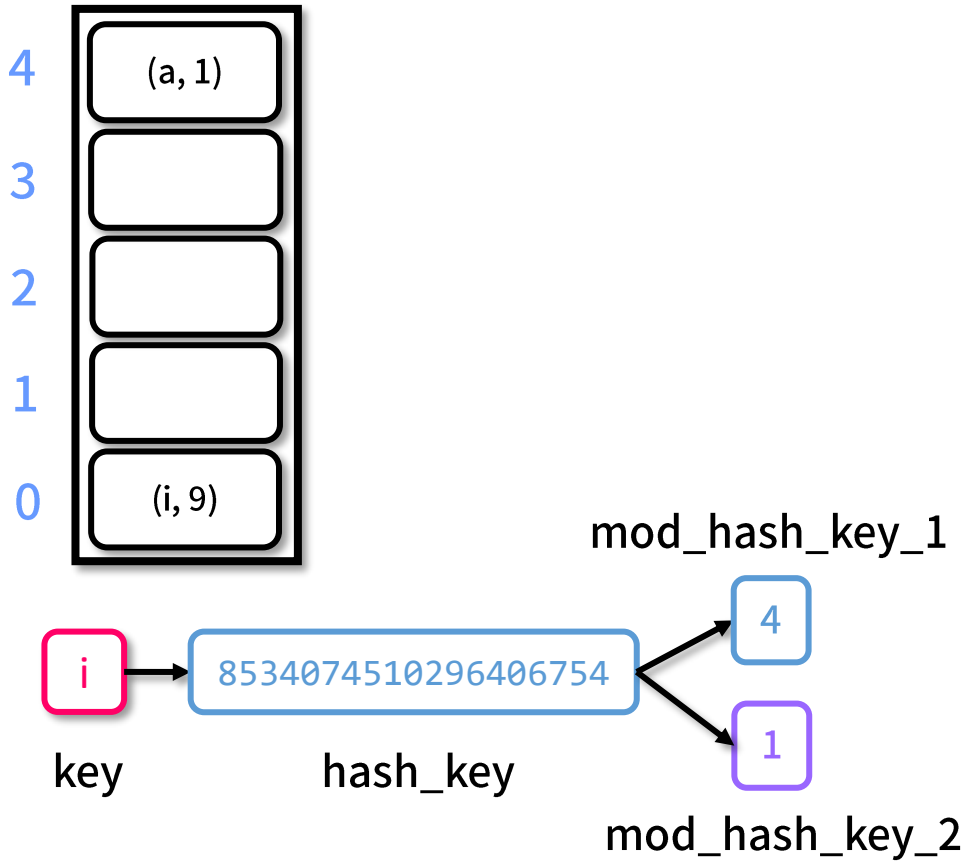
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_base_1 = self.size
    mod_base_2 = (self.size // 2 + 1)
    mod_hash_key_1 = hash_key % mod_base_1
    mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
    candidate_index = mod_hash_key_1
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.get_next_prime_number_size(expand=True))
            break
        else:
            candidate_index += mod_hash_key_2
        if candidate_index >= self.size:
            candidate_index -= self.size
```

Command

```python
hash_table['i'] = 9
```

Visualization



Linear Probing Hash Table

Set item (6)

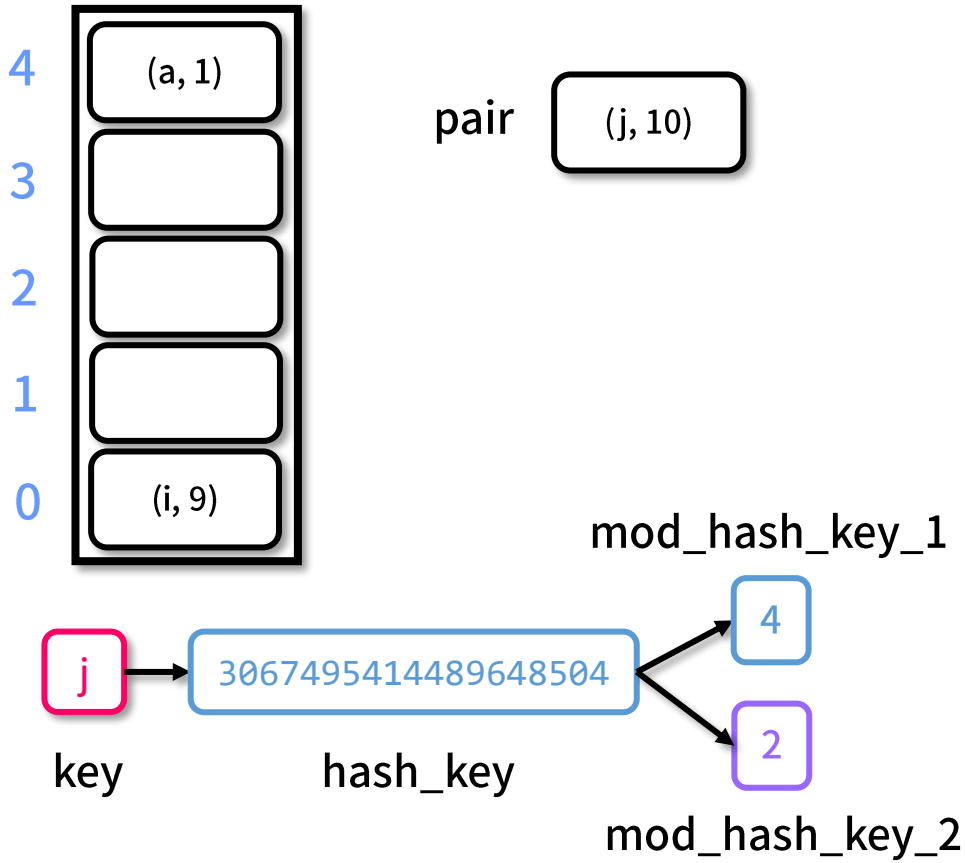
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_base_1 = self.size
    mod_base_2 = (self.size // 2 + 1)
    mod_hash_key_1 = hash_key % mod_base_1
    mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
    candidate_index = mod_hash_key_1
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.size * 2)
            break
        else:
            candidate_index += mod_hash_key_2
        if candidate_index >= self.size:
            candidate_index -= self.size
```

Command

```python
hash_table['j'] = 10
```

Visualization



Linear Probing Hash Table

Set item (7)

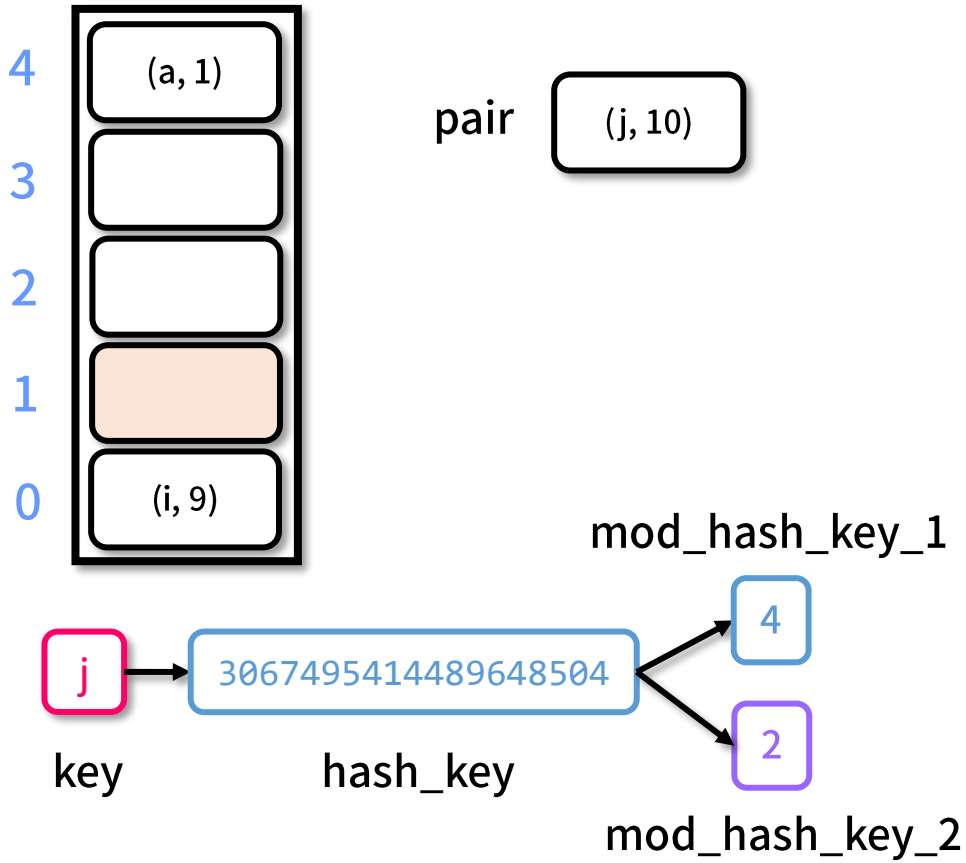
Set Item

```
def __setitem__(self, key: Hashable, value: Any) -> None:
  hash_key = hash(key)
  mod_base_1 = self.size
  mod_base_2 = (self.size // 2 + 1)
  mod_hash_key_1 = hash_key % mod_base_1
  mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
  candidate_index = mod_hash_key_1
  pair = Pair(key, value)

  while True:
    if (self.table[candidate_index] is None or
        isinstance(self.table[candidate_index], Tombstone) or
        hash(self.table[candidate_index].key) == hash_key):
      self.table[candidate_index] = pair
      self.n += 1
      if self.alpha >= 0.5:
        self.resize(self.get_next_prime_number_size(expand=True))
      break
    else:
      candidate_index += mod_hash_key_2
    if candidate_index >= self.size:
      candidate_index -= self.size
```

Command

```
hash_table['j'] = 10
```

Visualization



Linear Probing Hash Table

Set item (8)

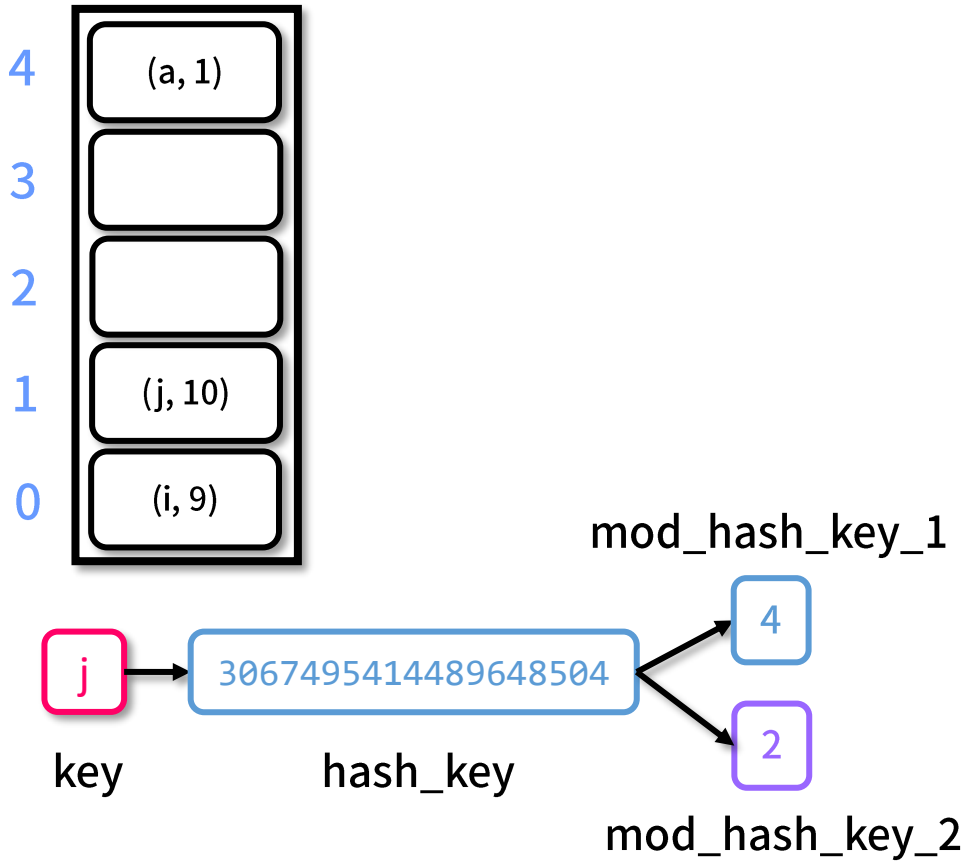
Set Item

```python
def __setitem__(self, key: Hashable, value: Any) -> None:
    hash_key = hash(key)
    mod_base_1 = self.size
    mod_base_2 = (self.size // 2 + 1)
    mod_hash_key_1 = hash_key % mod_base_1
    mod_hash_key_2 = mod_base_2 - (hash_key % mod_base_2)
    candidate_index = mod_hash_key_1
    pair = Pair(key, value)

    while True:
        if (self.table[candidate_index] is None or
            isinstance(self.table[candidate_index], Tombstone) or
            hash(self.table[candidate_index].key) == hash_key):
            self.table[candidate_index] = pair
            self.n += 1
            if self.alpha >= 0.5:
                self.resize(self.get_next_prime_number_size(expand=True))
            break
        else:
            candidate_index += mod_hash_key_2
        if candidate_index >= self.size:
            candidate_index -= self.size
```

Command

```python
hash_table['j'] = 10
```

Visualization



Hash Table

Summary

Separate Chaining

1. Useful when multiple deletion is expected.
2. Performance scale better when the array is near full.
3. Scattered locality of memory reference.
4. Less memory efficient.

Probing

1. More memory efficient without deletion.
2. Operations in continuous memory.
3. Need to deal with the formation of clusters (need careful design of the hash function).
4. Need to deal with tombstone (or rehashing) for multiple deletion.
5. The performance degrade when the array is near full.

Hash Table

Practices

- Repeated DNA Sequences (Leetcode Problem 187)
- Maximum Length of Repeated Subarray (Leetcode Problem 718)