Disjoint Set - Union/Find

Disjoint Set – Union/Find

Disjoint Set – Union/Find

- n distinct elements named 1, 2, …, n

Disjoint Set – Union/Find

- n distinct elements named 1, 2, …, n
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

Disjoint Set – Union/Find

- n distinct elements named 1, 2, …, n
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

- Each set has a representative element

Disjoint Set – Union/Find

- n distinct elements named $1, 2, \ldots, n$
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

- Each set has a representative element
- S_x : Set represented by element x

Disjoint Set – Union/Find

- n distinct elements named 1, 2, …, n
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

- Each set has a representative element
- S_x : Set represented by element x

Operations:

Disjoint Set – Union/Find

- n distinct elements named $1, 2, \ldots, n$
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

- Each set has a representative element
- S_x : Set represented by element x

Operations:

Union(S_x, S_y): Create set $S = S_x \cup S_y$ and return the representative of S

Disjoint Set – Union/Find

- n distinct elements named $1, 2, \ldots, n$
- Initially, each element is in its own set

$$S_1 = \{1\}, \quad S_2 = \{2\}, \ldots, \quad S_n = \{n\}$$

- Each set has a representative element
- S_x : Set represented by element x

Operations:

Union(S_x, S_y): Create set $S = S_x \cup S_y$ and return the representative of S

Find(z): Given (a ptr to) z, find set S that contains z and return the representative of S

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}

S_j : Set represented by j

Example

	S_1	S_2	S_3	S_4	S_5
Initially :	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4) :					

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{$\underline{3}$, 4}	X	{5}

Find(4) =

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{$\underline{3}$, 4}	X	{5}

Find(4) = S_3

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}

Find(4) = S_3

Union(S_1, S_5) :

S_j : Set represented by j

Example

	S_1	S_2	S_3	S_4	S_5
Initially :	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4) :	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3					
Union(S_1, S_5) :	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3)	:					

S_j : Set represented by j

Example

	S_1	S_2	S_3	S_4	S_5
Initially :	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4) :	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3					
Union(S_1, S_5) :	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3) :	{<u>1</u>, 5, 3, 4}	{2}	X	X	X

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3)	:	{<u>1</u>, 5, 3, 4}	{2}	X	X	X
Find(4) =						

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3)	:	{<u>1</u>, 5, 3, 4}	{2}	X	X	X
Find(4) = S_1						

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{$\underline{3}$, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{$\underline{1}$, 5}	{2}	{$\underline{3}$, 4}	X	X
Union(S_1, S_3)	:	{$\underline{1}$, 5, 3, 4}	{2}	X	X	X
Find(4) = S_1						
Find(2) =						

S_j : Set represented by j

Example

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3)	:	{<u>1</u>, 5, 3, 4}	{2}	X	X	X
Find(4) = S_1						
Find(2) = S_2						

Example

S_j : Set represented by j

		S_1	S_2	S_3	S_4	S_5
Initially	:	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4)	:	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3						
Union(S_1, S_5)	:	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3)	:	{<u>1</u>, 5, 3, 4}	{2}	X	X	X
Find(4) = S_1						
Find(2) = S_2						
Union(S_1, S_2)	:					

S_j : Set represented by j

Example

	S_1	S_2	S_3	S_4	S_5
Initially :	{1}	{2}	{3}	{4}	{5}
Union(S_3, S_4) :	{1}	{2}	{<u>3</u>, 4}	X	{5}
Find(4) = S_3					
Union(S_1, S_5) :	{<u>1</u>, 5}	{2}	{<u>3</u>, 4}	X	X
Union(S_1, S_3) :	{<u>1</u>, 5, 3, 4}	{2}	X	X	X
Find(4) = S_1					
Find(2) = S_2					
Union(S_1, S_2) :	{<u>1</u>, 5, 3, 4, 2}	X	X	X	X

Each **Union** reduces # of sets by 1

Each **Union** reduces # of sets by 1 $\Rightarrow$ Can do at most $n - 1$ **Unions**

Each **Union** reduces # of sets by 1 $\Rightarrow$ Can do at most n – 1 **Unions**

σ : Sequence of n – 1 **Unions** mixed with m $\geq$ n **Finds**

Each **Union** reduces # of sets by 1 $\Rightarrow$ Can do at most n – 1 **Unions**

σ : Sequence of n – 1 **Unions** mixed with m $\geq$ n **Finds**

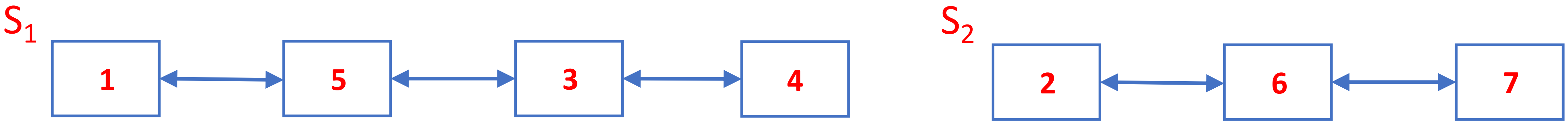
Goal: a data structure that minimizes the total cost of executing such sequences

Data Structures for Disjoint Sets

1. Linked Lists

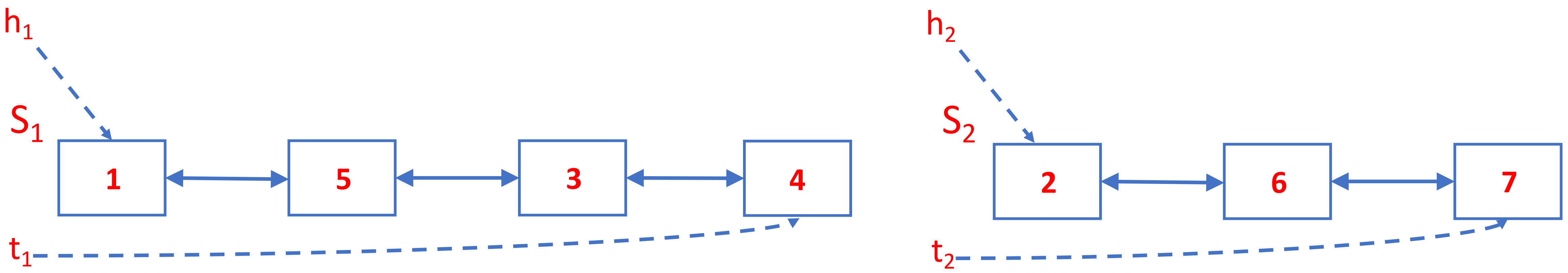
1. Linked Lists

- One list per set



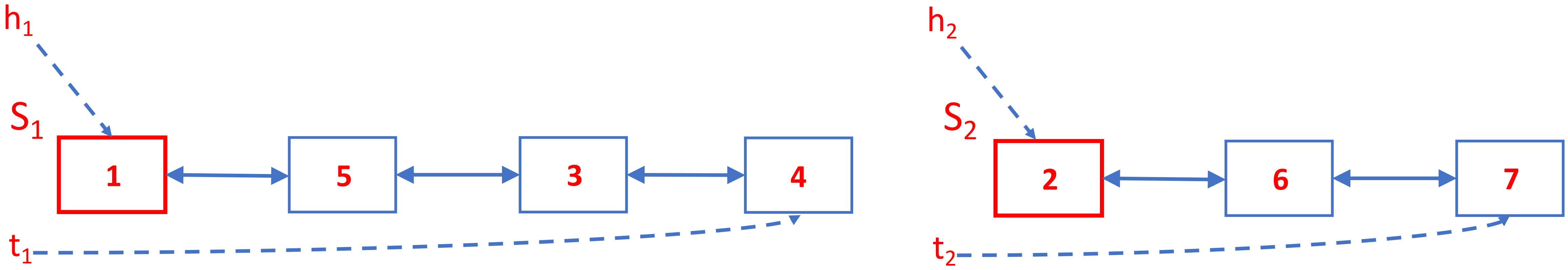
1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set



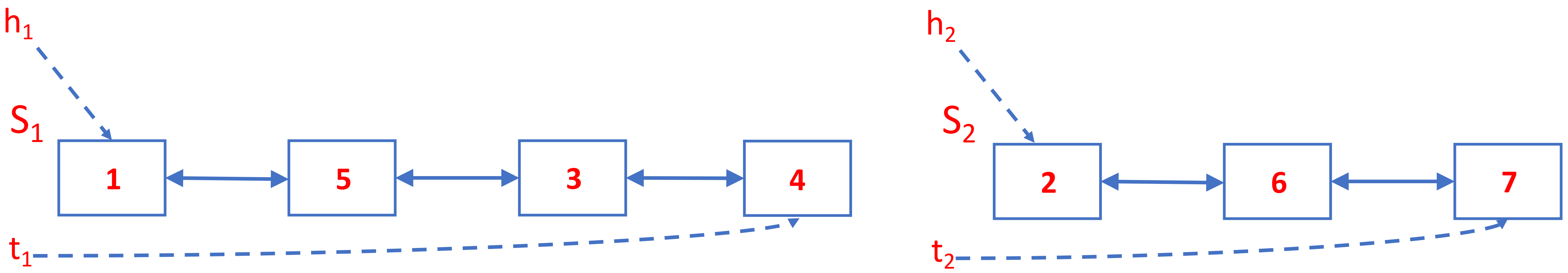
1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



1. Linked Lists

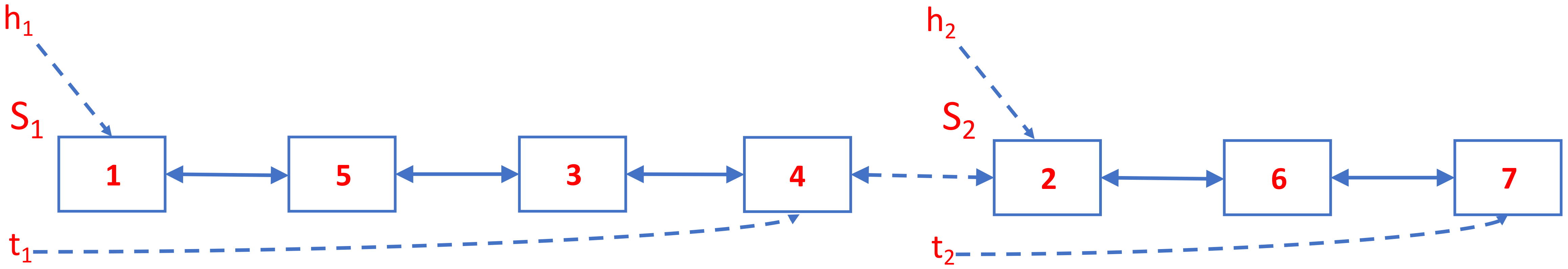
- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



Union

1. Linked Lists

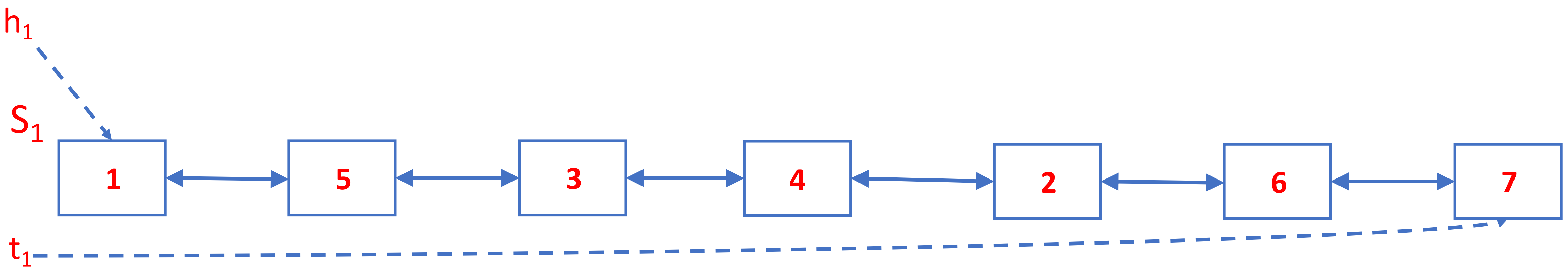
- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



Union

1. Linked Lists

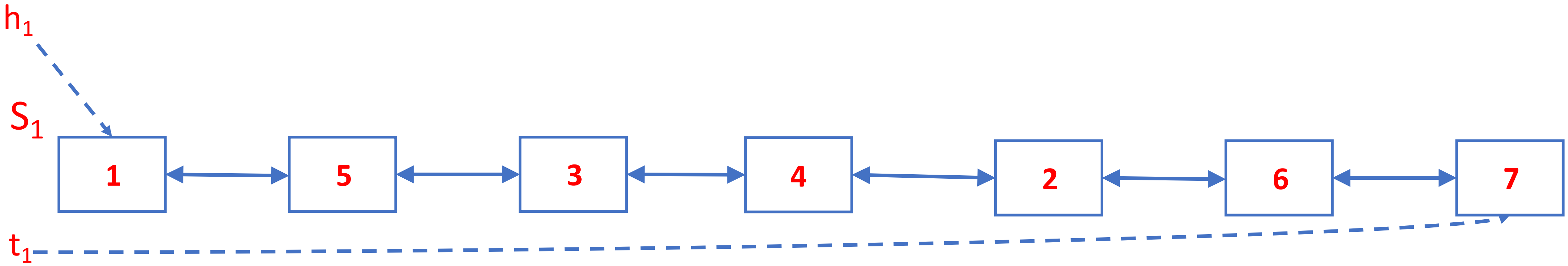
- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



Union

1. Linked Lists

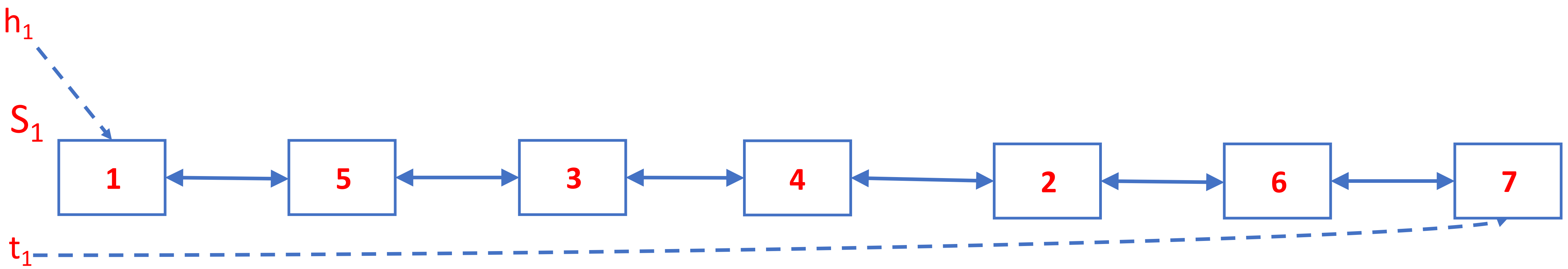
- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



Each **Union** : O(1)

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative

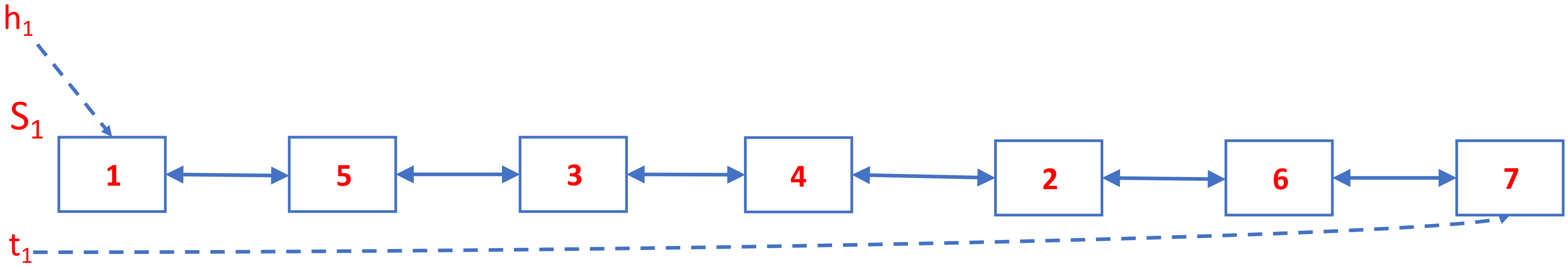


Each **Union** : O(1)

Find

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative

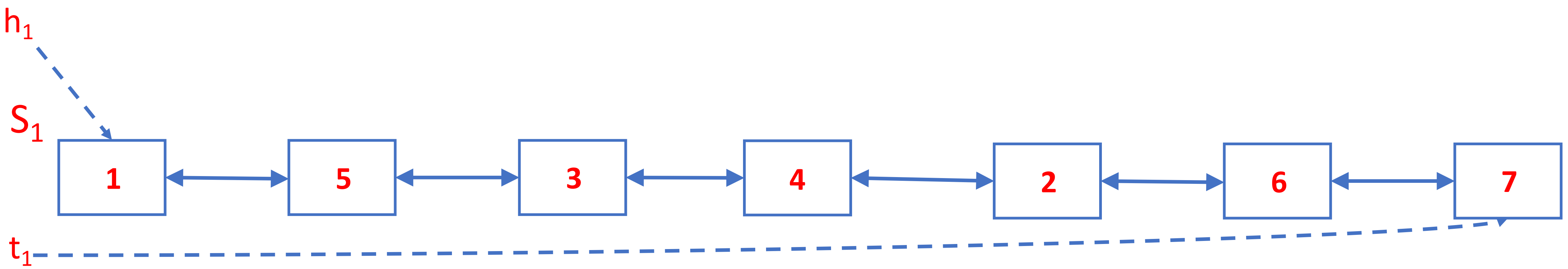


Each **Union** : O(1)

Each **Find** : O(n)

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



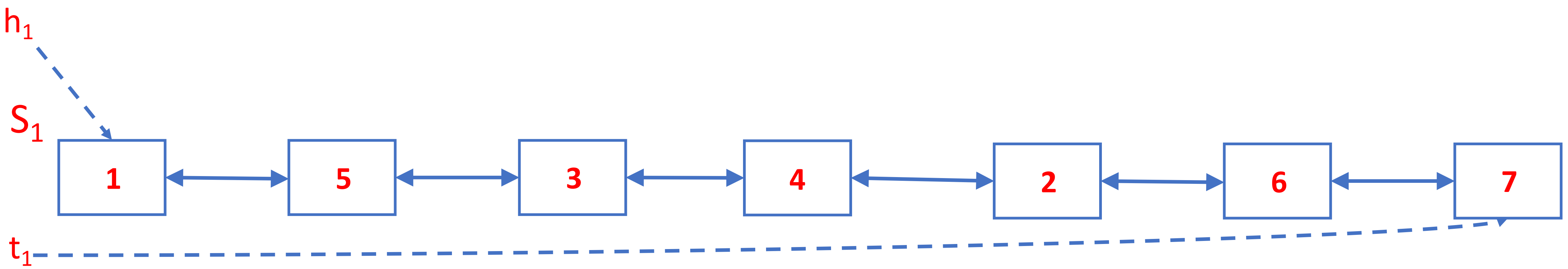
σ : Any Sequence of $n - 1$ **Unions** mixed with $m \geq n$ **Finds**

Each **Union** : $O(1)$

Each **Find** : $O(n)$

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



σ : Any Sequence of $n-1$ **Unions** mixed with $m \geq n$ **Finds**

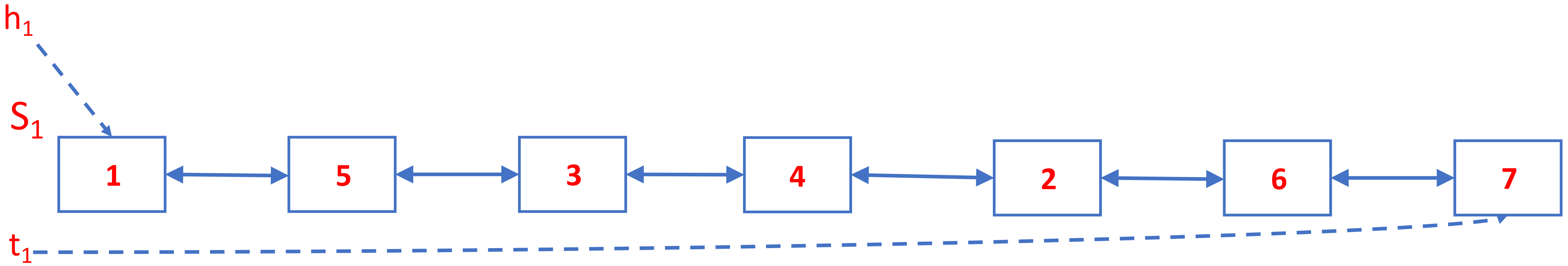
Each **Union** : $O(1)$

Each **Find** : $O(n)$

Worst-case cost of σ :

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



σ : Any Sequence of $n - 1$ **Unions** mixed with $m \geq n$ **Finds**

Each **Union** : $O(1)$

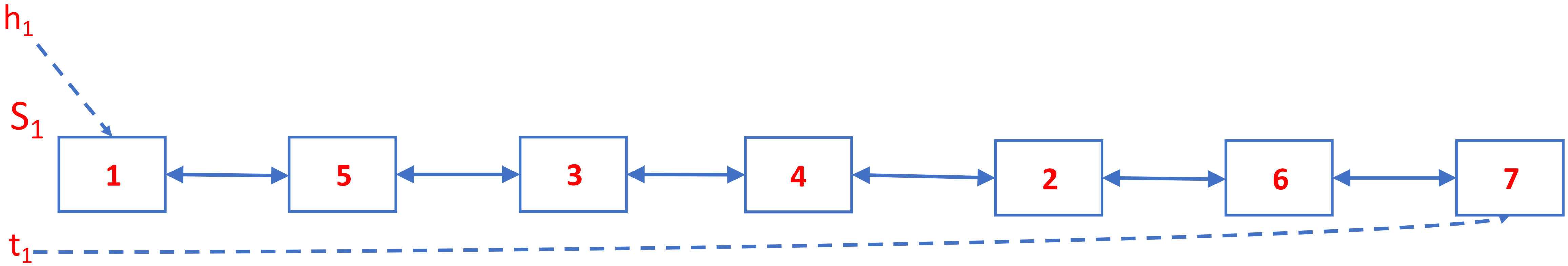
Each **Find** : $O(n)$

Worst-case cost of σ : $O(n.\ 1 + m.\ n)$

Per **Union** Per **Find**

1. Linked Lists

- One list per set
- Store a head pointer and tail pointer for each set
- First element of the set is the representative



σ : Any Sequence of $n - 1$ **Unions** mixed with $m \geq n$ **Finds**

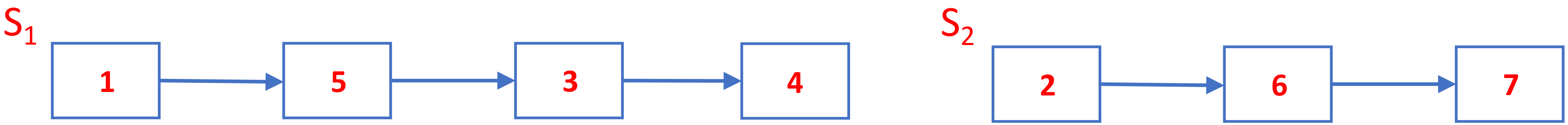
Each **Union** : $O(1)$

Each **Find** : $O(n)$

Worst-case cost of σ : $O(m.\ n)$

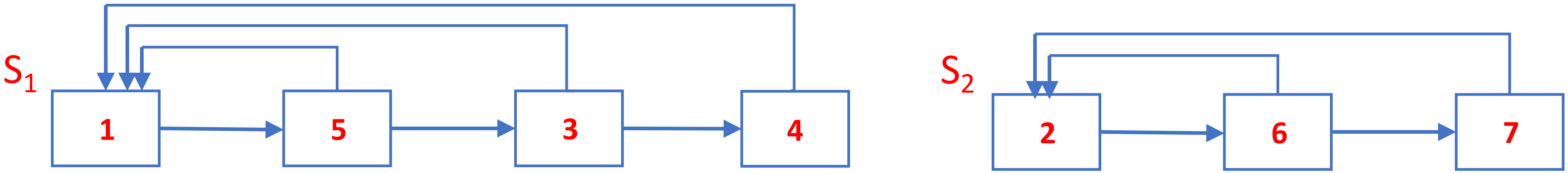
2. Augmented Linked Lists

- Each element also points to its representative



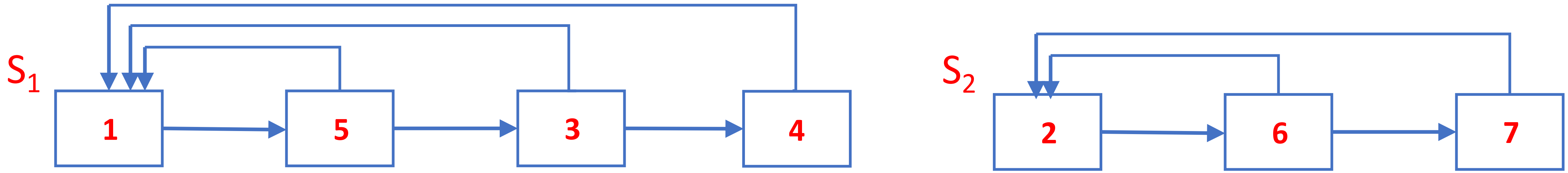
2. Augmented Linked Lists

- Each element also points to its representative



2. Augmented Linked Lists

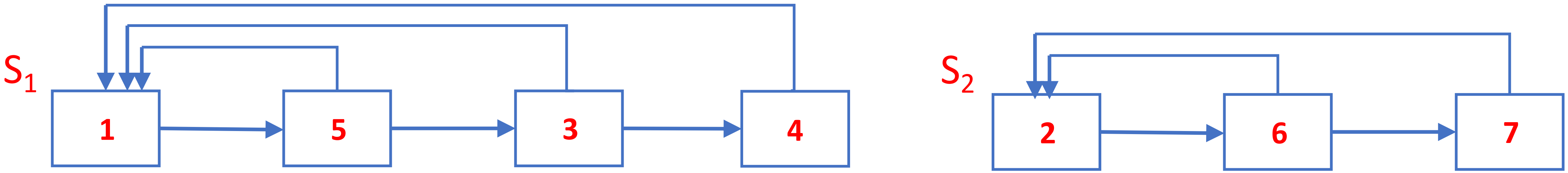
- Each element also points to its representative



Each **Find** : O(1)

2. Augmented Linked Lists

- Each element also points to its representative

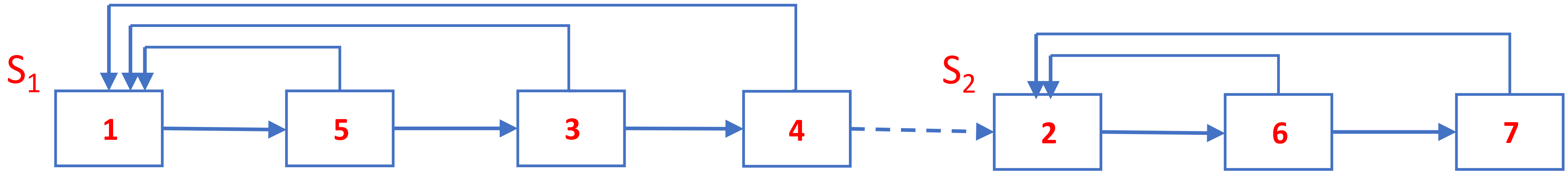


Each **Find** : O(1)

Union

2. Augmented Linked Lists

- Each element also points to its representative

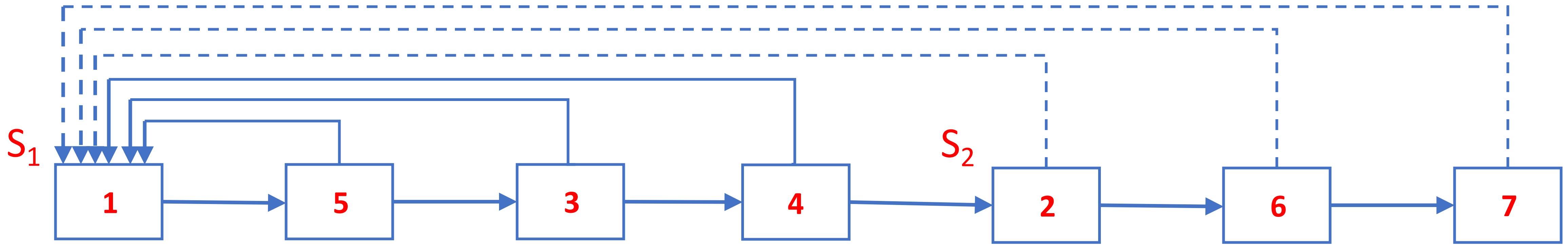


Each **Find** : O(1)

Union

2. Augmented Linked Lists

- Each element also points to its representative

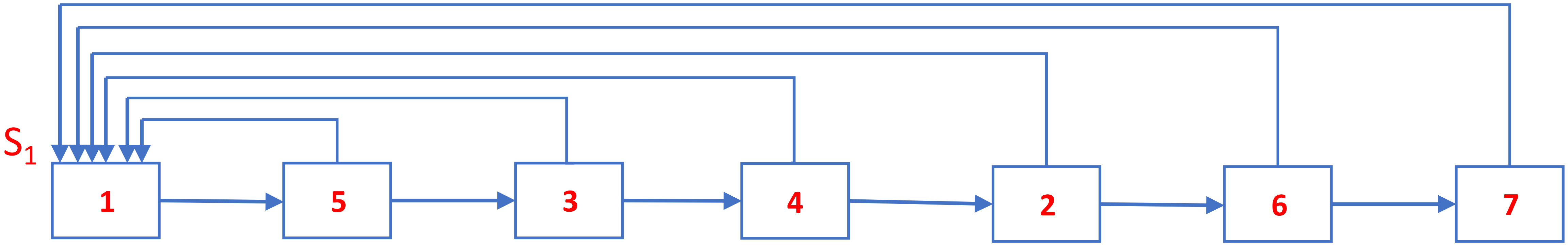


Each **Find** : O(1)

Union : must redirect O(n) pointers to the new representative

2. Augmented Linked Lists

- Each element also points to its representative

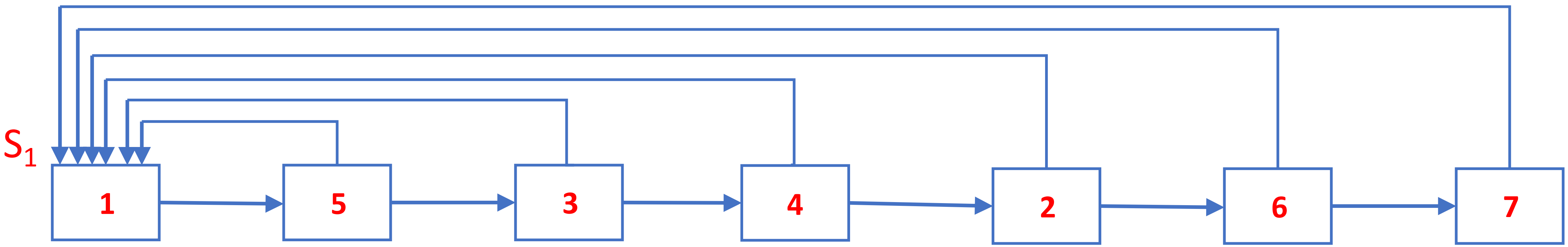


Each **Find** : O(1)

Each **Union** : O(n)

2. Augmented Linked Lists

- Each element also points to its representative



Each **Find** : O(1)

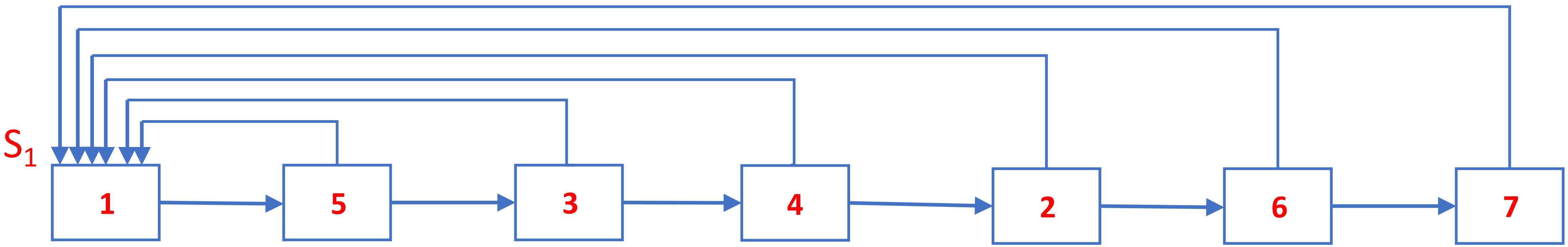
Each **Union** : O(n)

Worst-case cost of σ : O(n . n + m . 1)

Per **Union** Per **Find**

2. Augmented Linked Lists

- Each element also points to its representative



Each **Find** : O(1)

Each **Union** : O(n)

Worst-case cost of σ : $O(n^2 + m)$

2. Augmented Linked Lists with Weighted Union

2. Augmented Linked Lists with Weighted Union

WU rule: Append the smaller list onto the bigger list (keep track of size of each list)

2. Augmented Linked Lists with Weighted Union

WU rule: Append the smaller list onto the bigger list (keep track of size of each list)

- Each **Find** : O(1)
- Each **Union** : O(n)

2. Augmented Linked Lists with Weighted Union

WU rule: Append the smaller list onto the bigger list (keep track of size of each list)

- Each **Find** : O(1)
- Each **Union** : O(n)

Claim: worst-case cost of executing σ : O(m + n log n)

Proof :

2. Augmented Linked Lists with Weighted Union

WU rule: Append the smaller list onto the bigger list (keep track of size of each list)

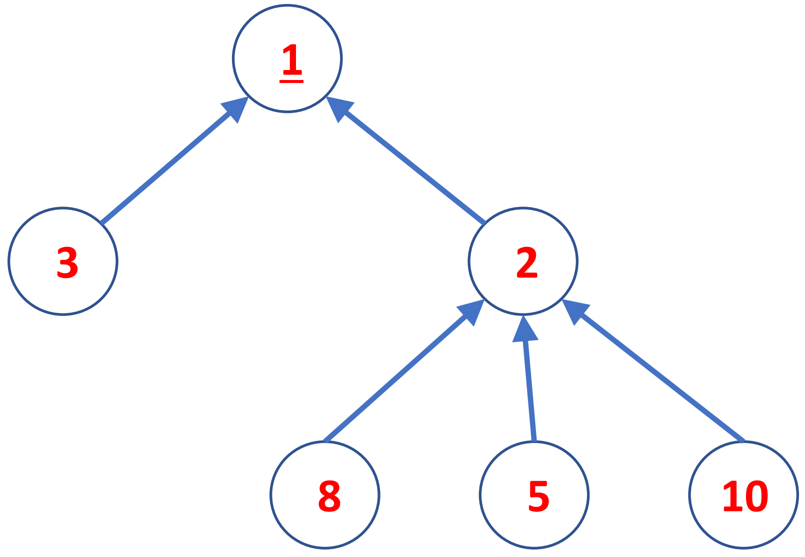
- Each **Find** : O(1)
- Each **Union** : O(n)

Claim: worst-case cost of executing σ : O(m + n log n)

Proof : Go to the tutorial ☺

3. Forest structure for Union-Find

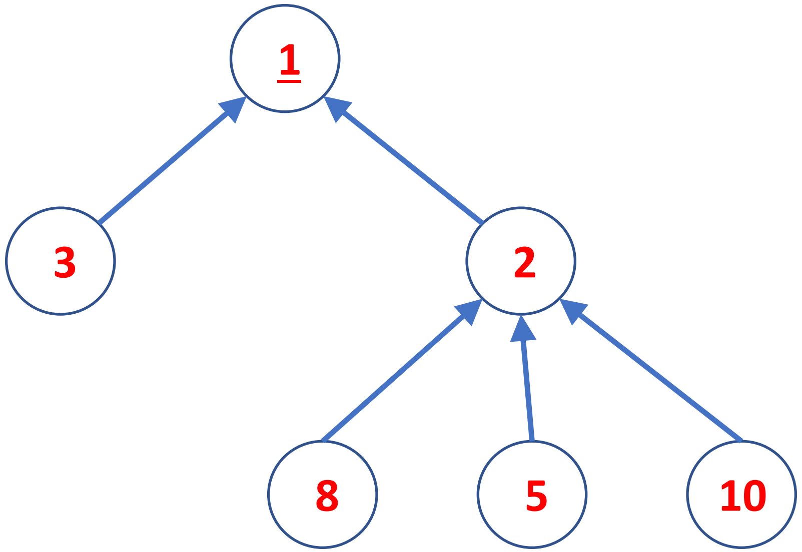
S_1 = {<u>1</u>, 3, 2, 8, 5, 10}



3. Forest structure for Union-Find

- Each set is represented by a tree

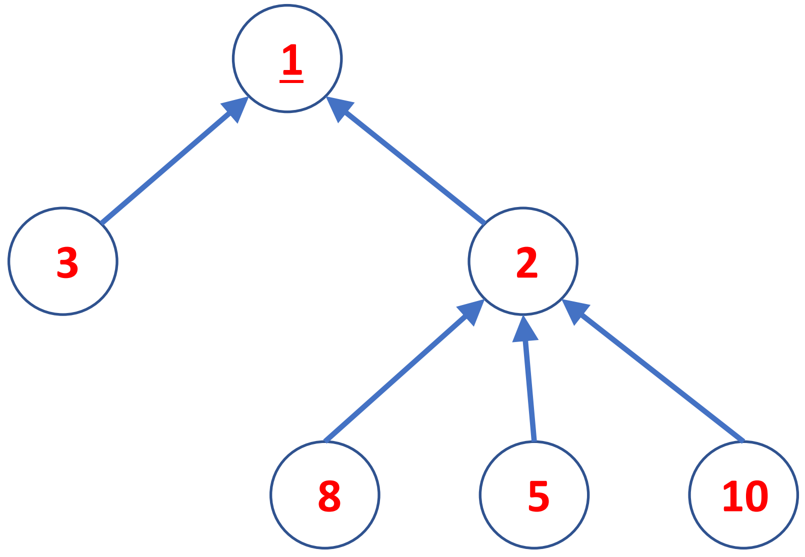
$S_1 = \{\underline{1}, 3, 2, 8, 5, 10\}$



3. Forest structure for Union-Find

- Each set is represented by a tree
- Each node represents an element

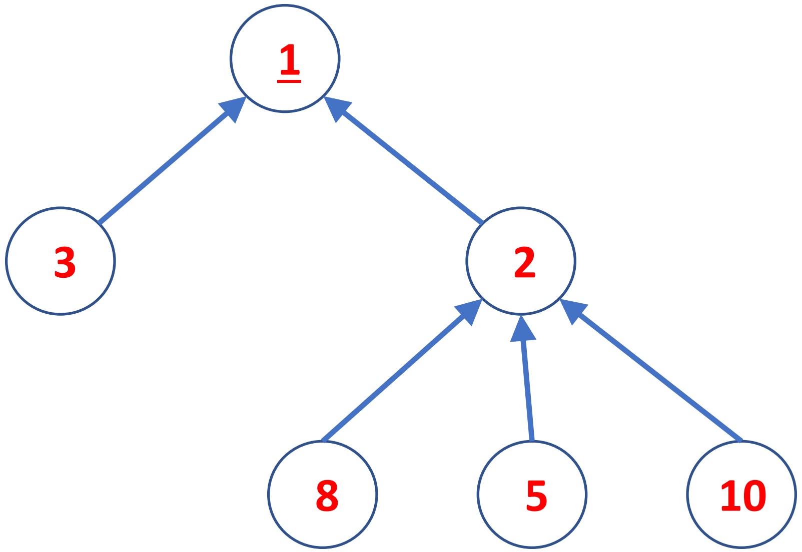
$S_1 = \{\underline{1}, 3, 2, 8, 5, 10\}$



3. Forest structure for Union-Find

- Each set is represented by a tree
- Each node represents an element
- Each non-root node points to its parent

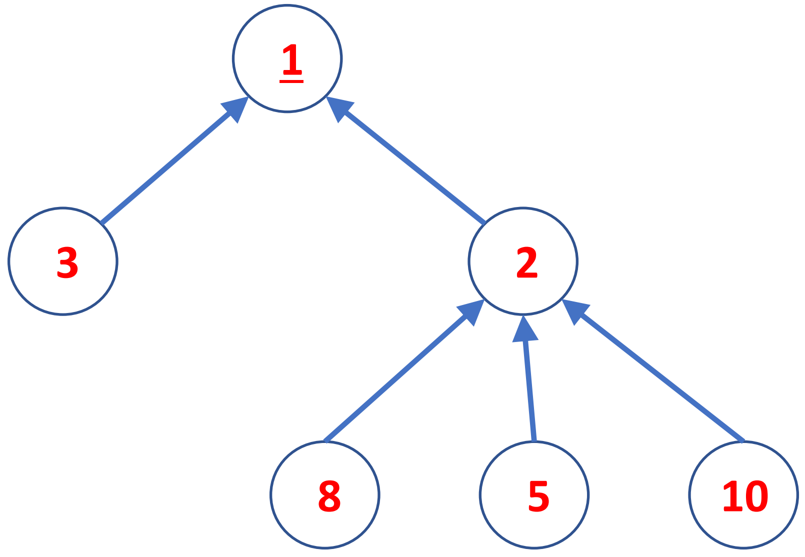
$S_1 = \{\underline{1}, 3, 2, 8, 5, 10\}$



3. Forest structure for Union-Find

- Each set is represented by a tree
- Each node represents an element
- Each non-root node points to its parent
- The root contains the set representative

$S_1 = \{\underline{1}, 3, 2, 8, 5, 10\}$



Example with $n = 10$

$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Example with $n = 10$

$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$S_1 = \{1, 3, 2, 8, 5, 10\}$

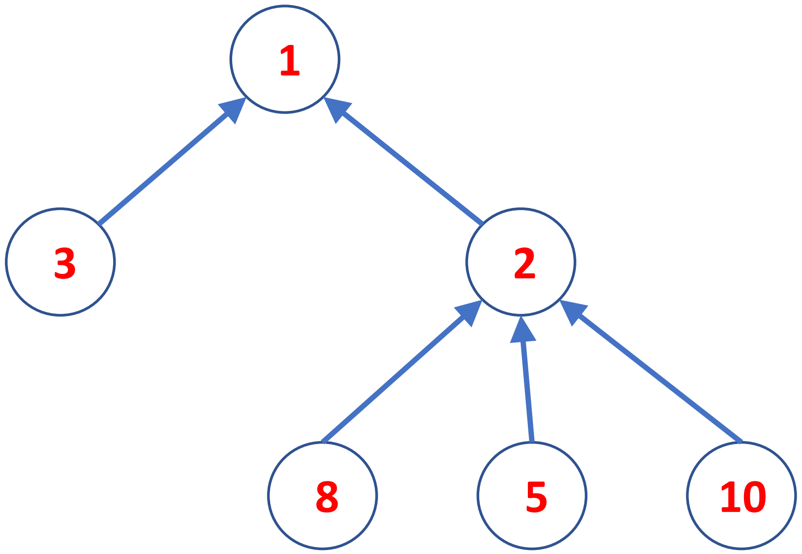
$S_6 = \{6\}$

$S_9 = \{9, 7, 4\}$

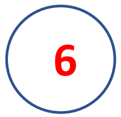
Example with $n = 10$

$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

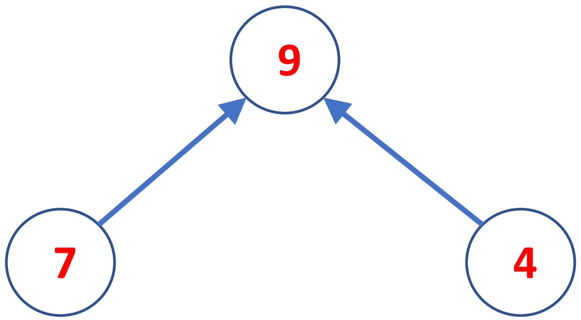
$S_1 = \{1, 3, 2, 8, 5, 10\}$



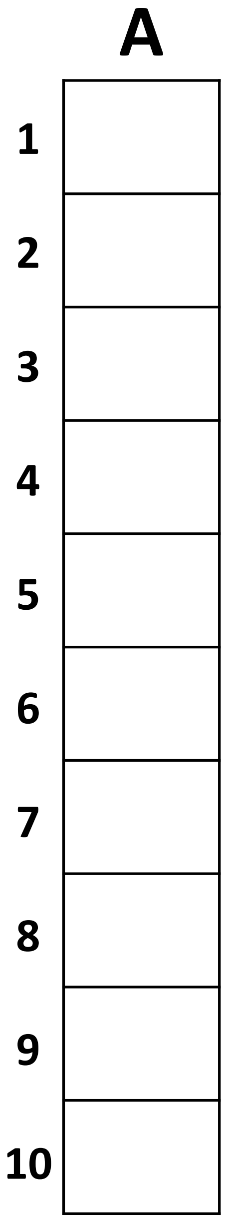
$S_6 = \{6\}$



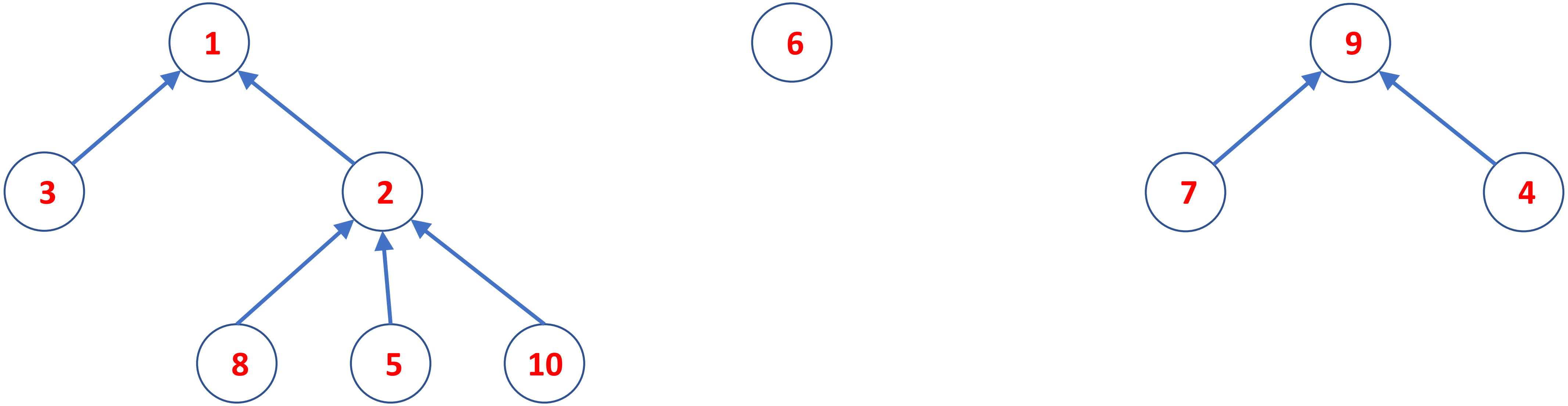
$S_9 = \{9, 7, 4\}$



Example with n = 10

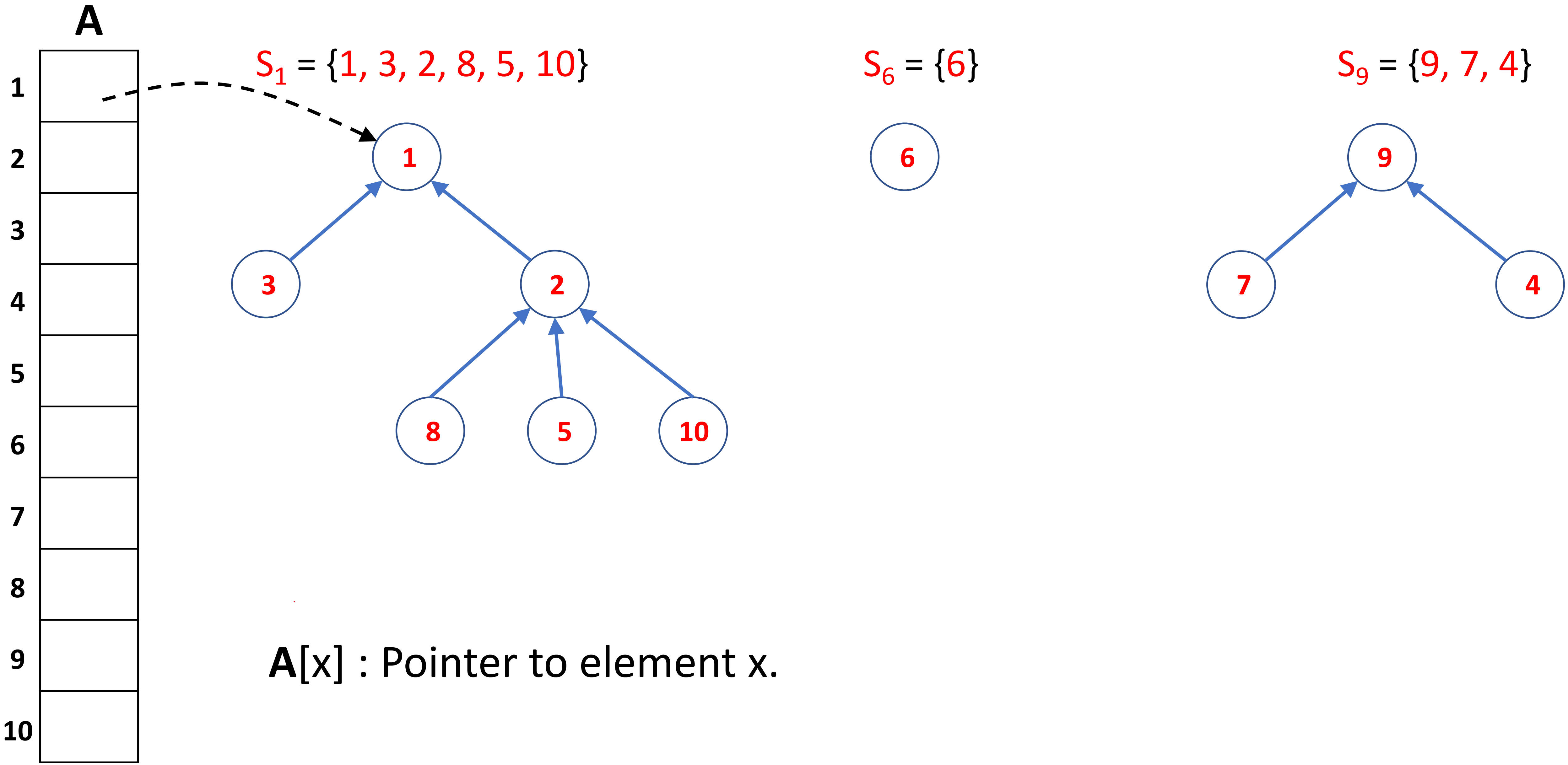


S_1 = {1, 3, 2, 8, 5, 10} S_6 = {6} S_9 = {9, 7, 4}



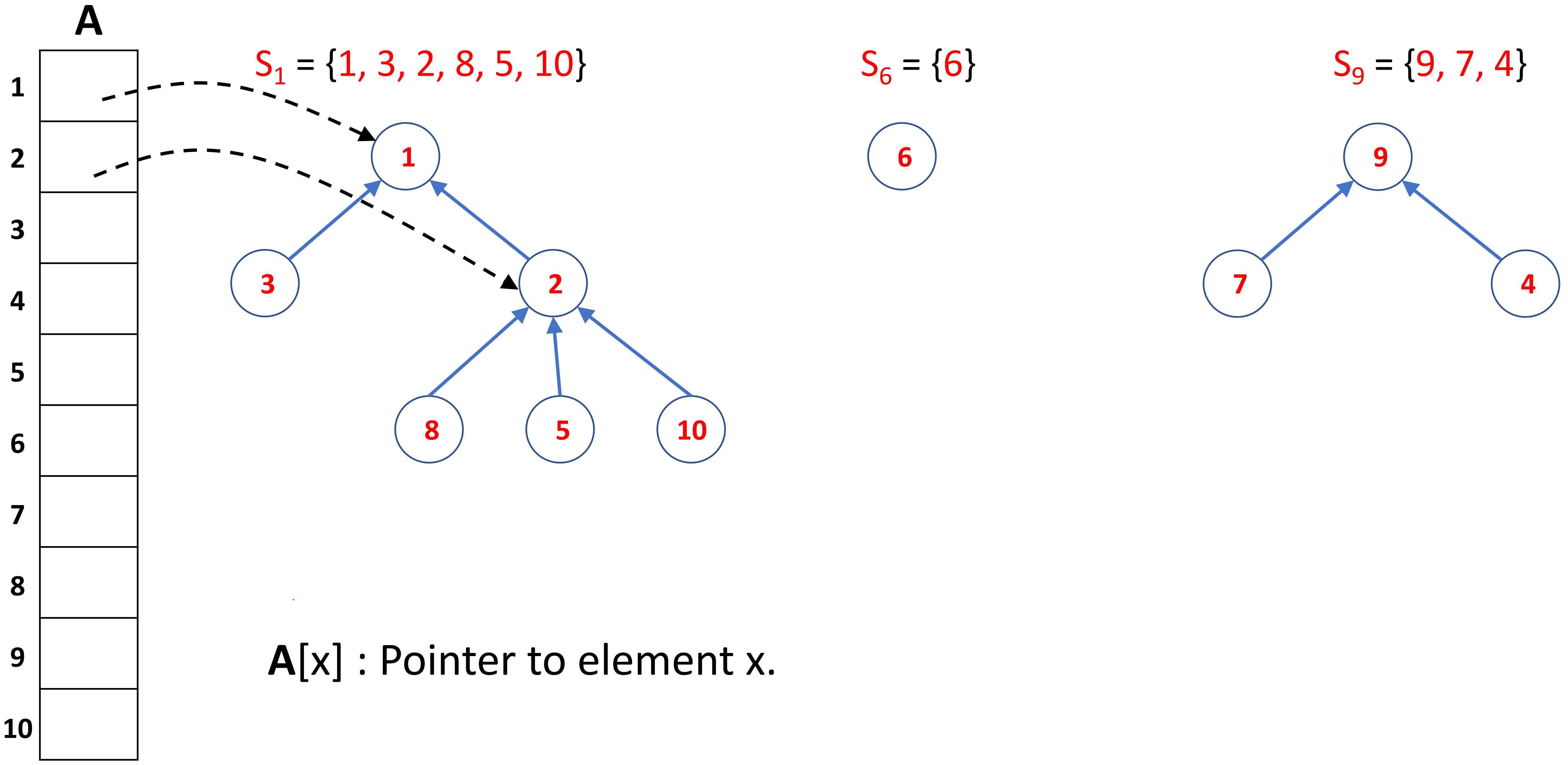
A[x] : Pointer to element x.

Example with $n = 10$



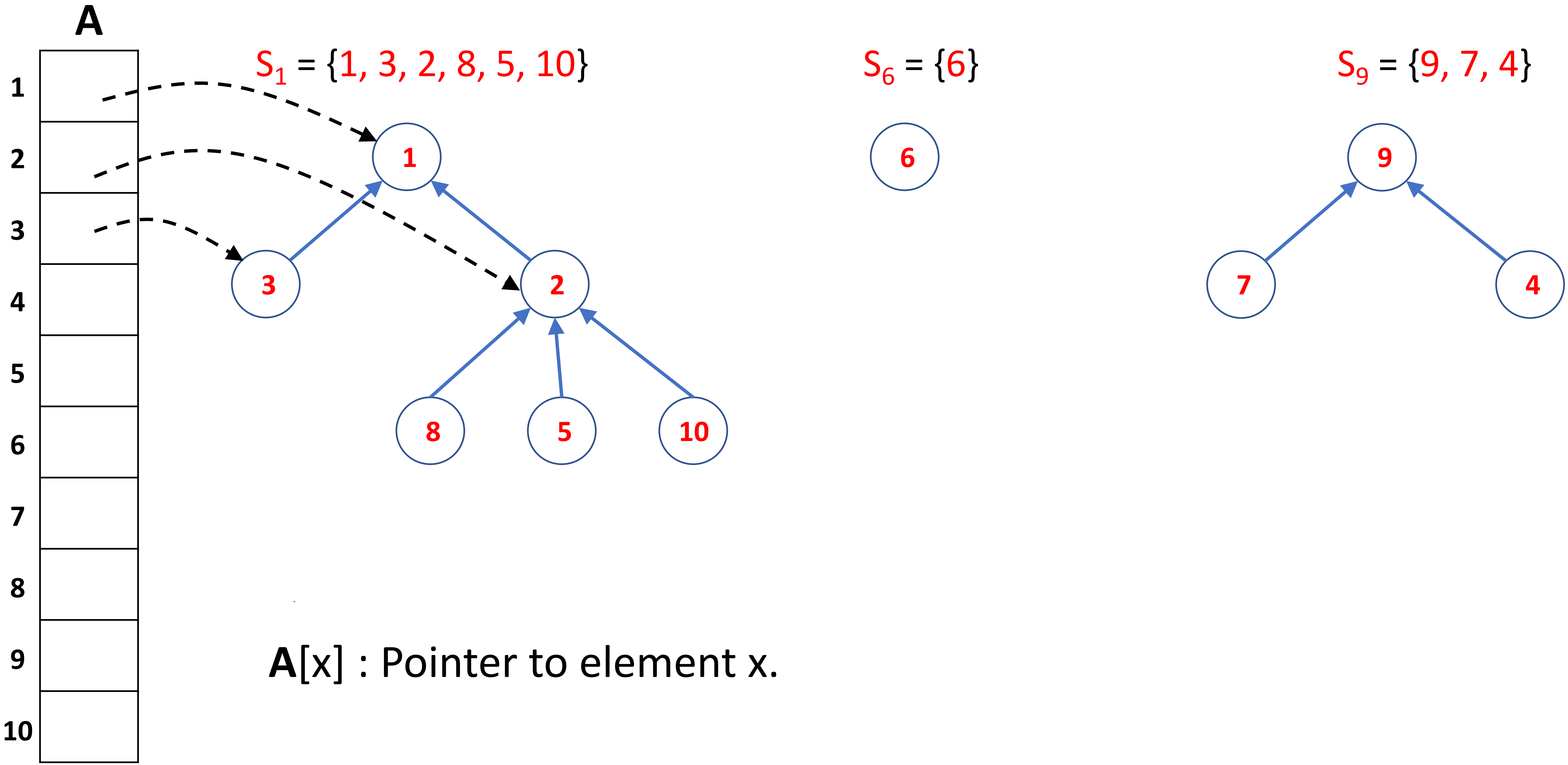
A[x] : Pointer to element x.

Example with n = 10



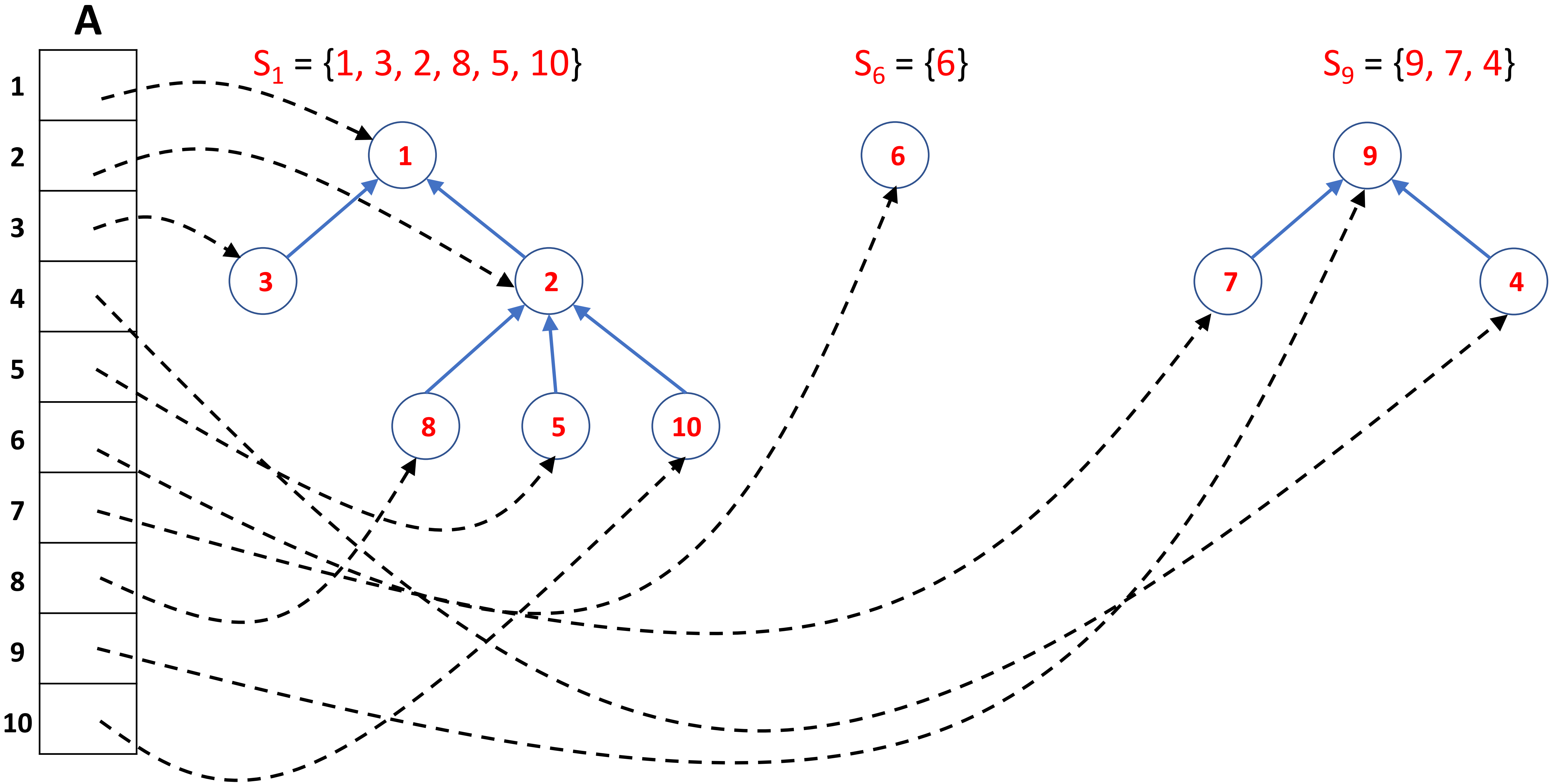
A[x] : Pointer to element x.

Example with n = 10

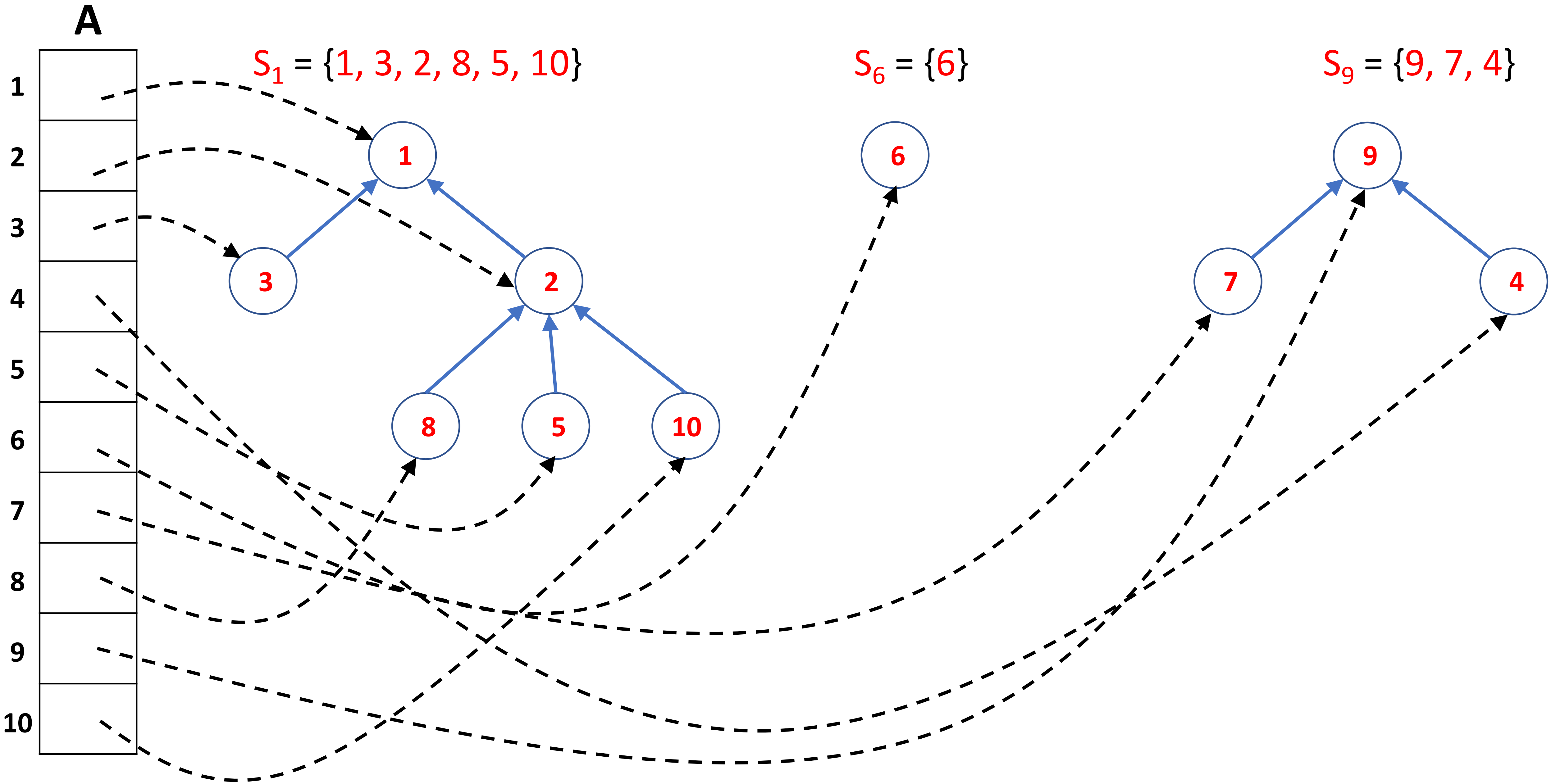


A[x] : Pointer to element x.

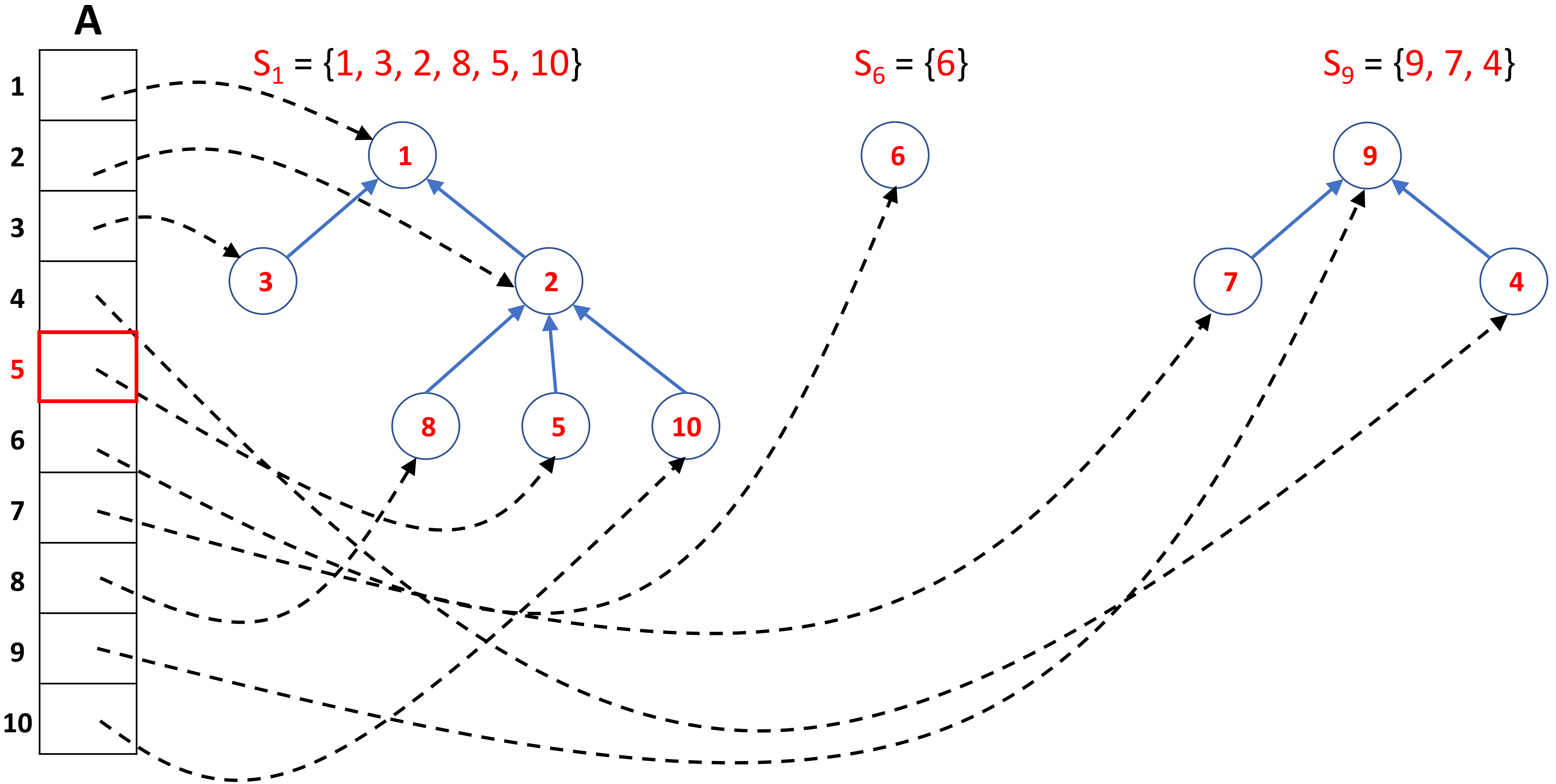
Example with n = 10



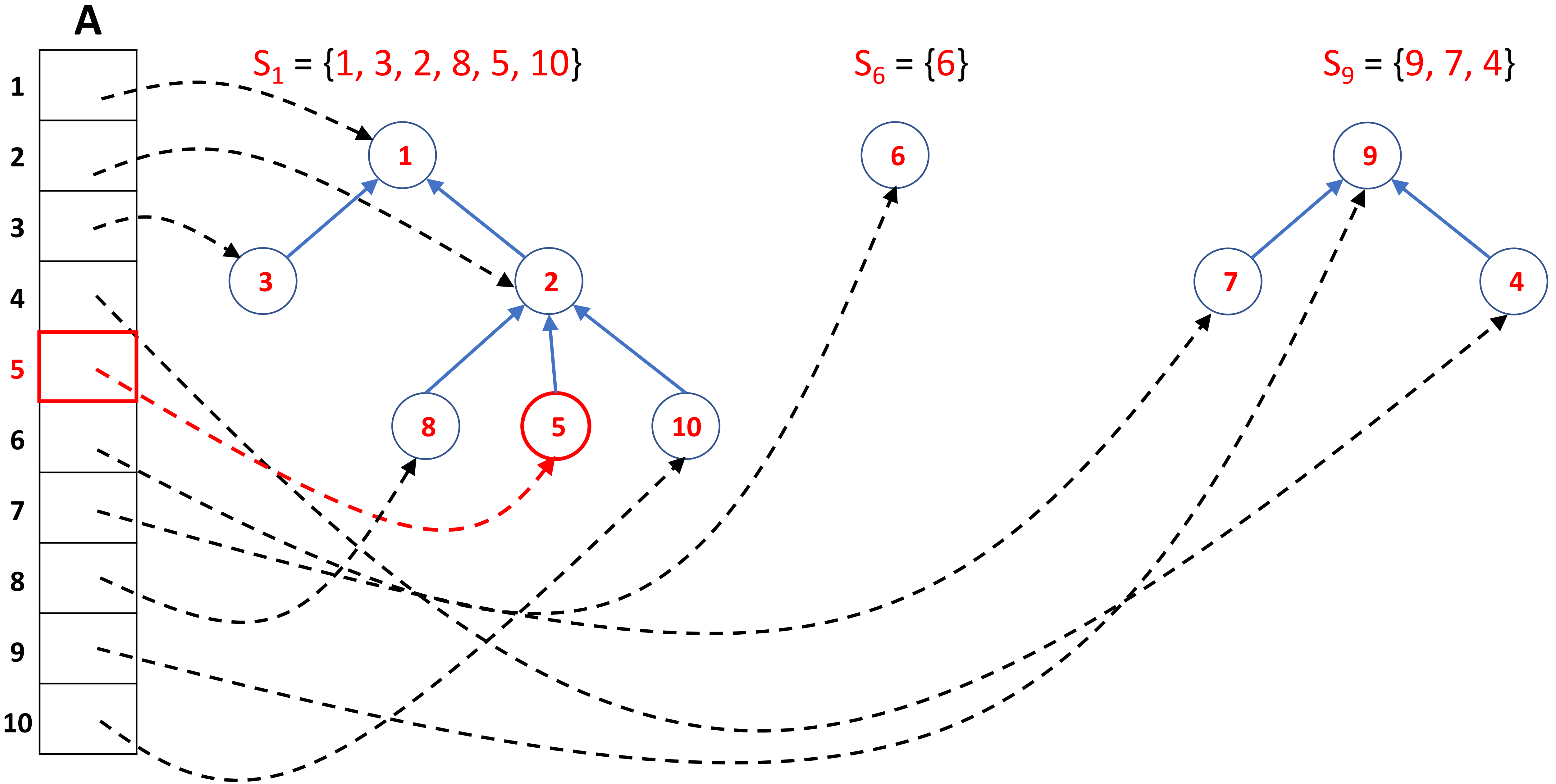
Find(5)



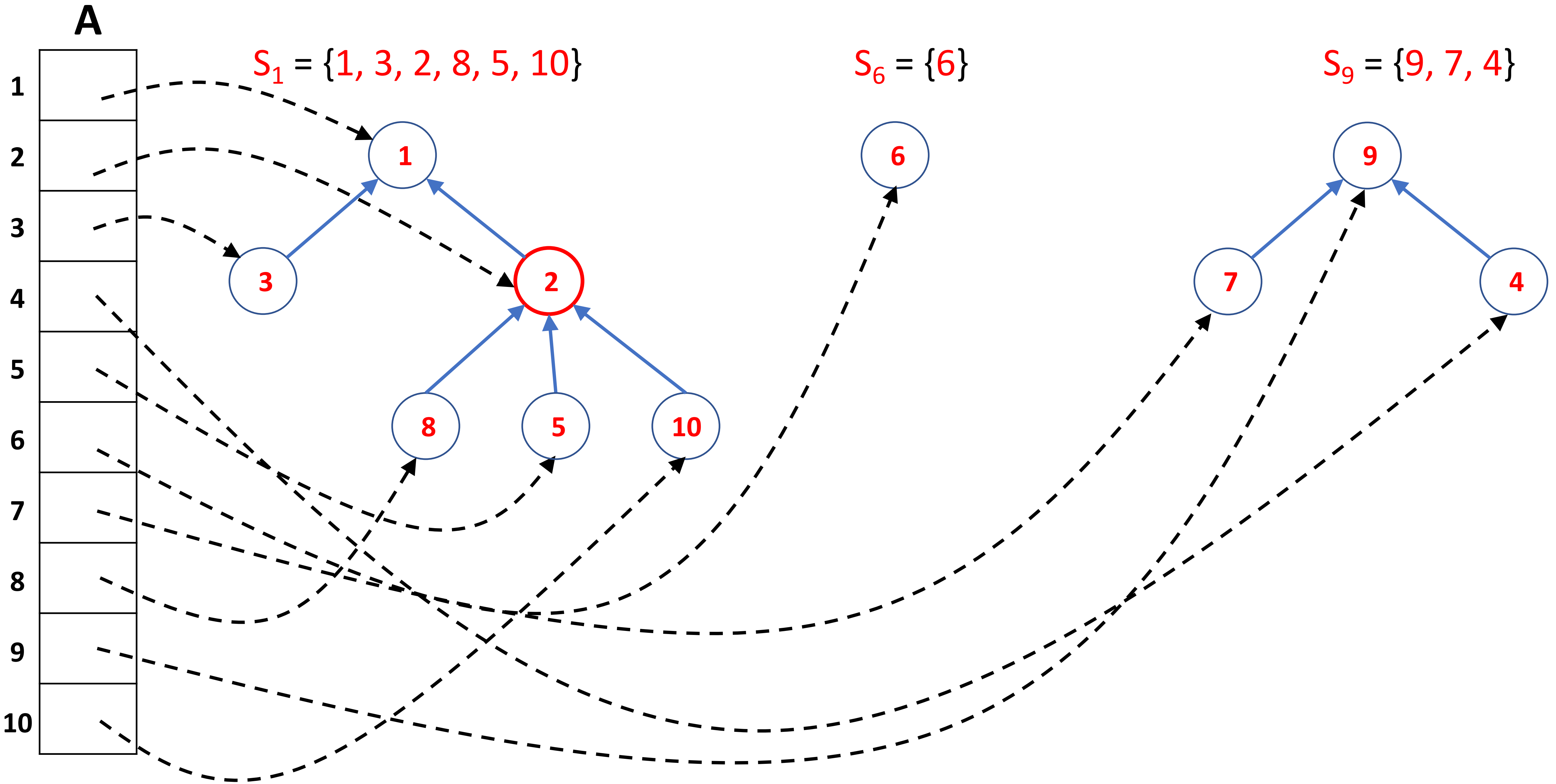
Find(5)



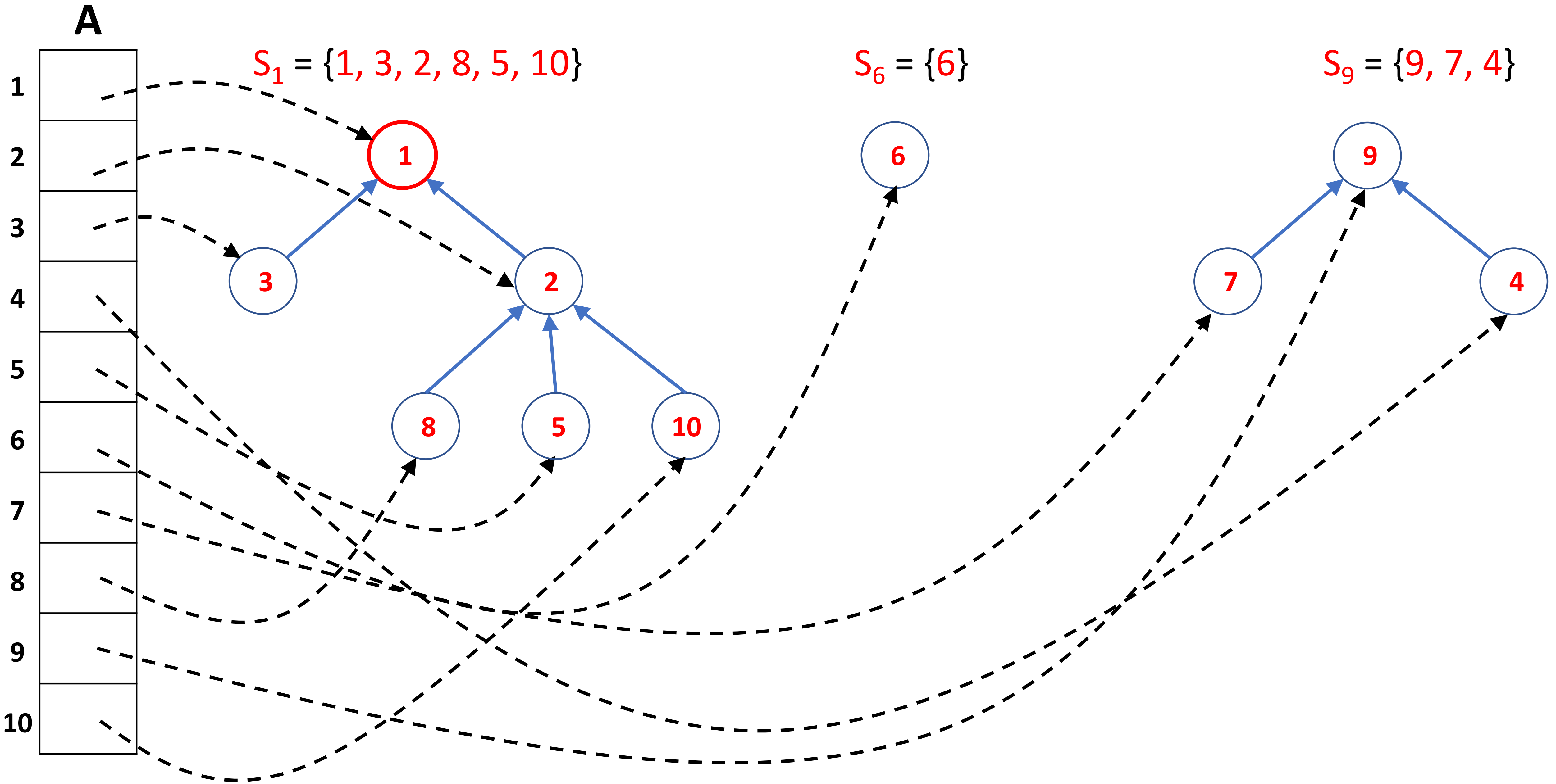
Find(5)



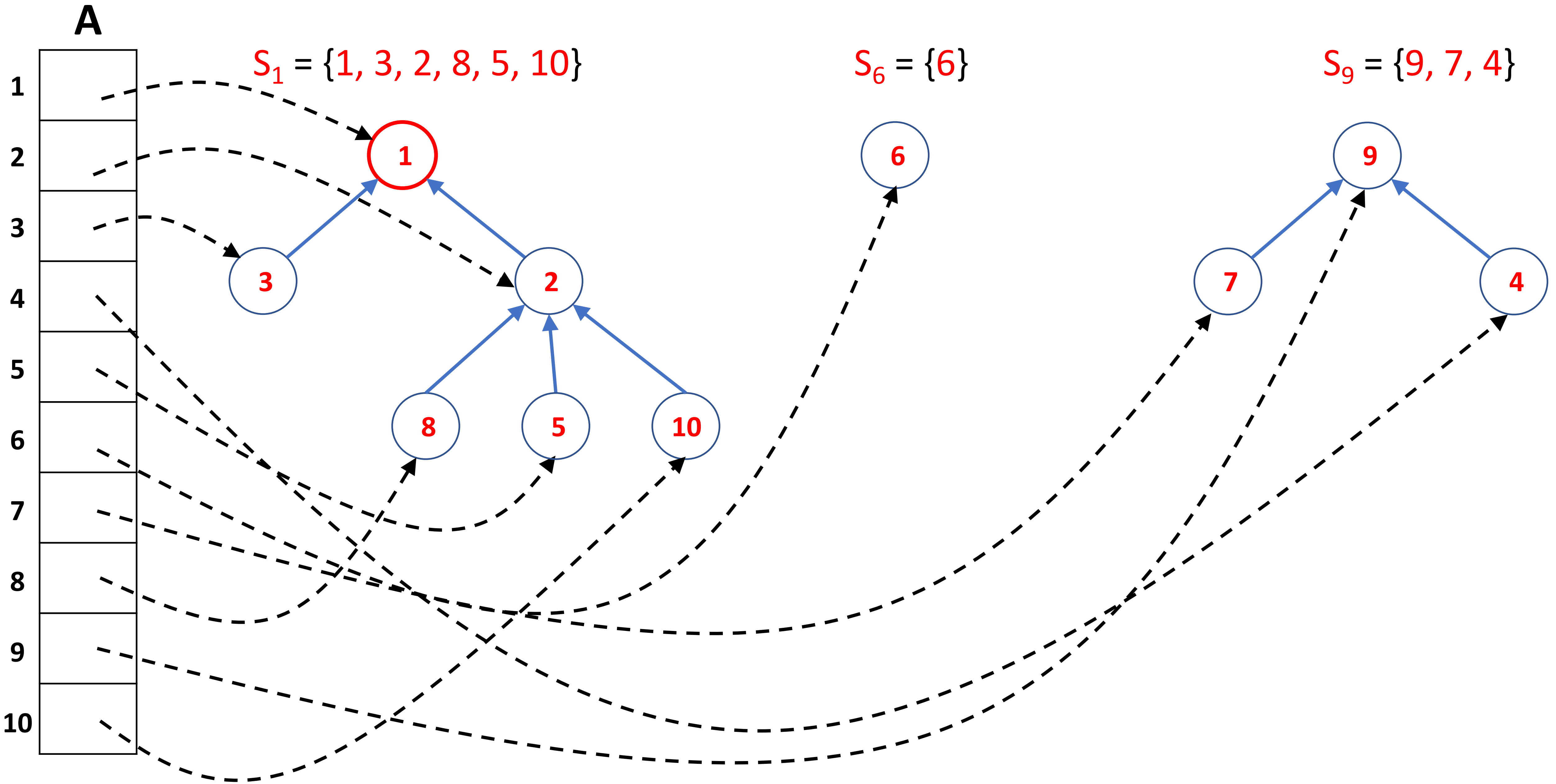
Find(5)



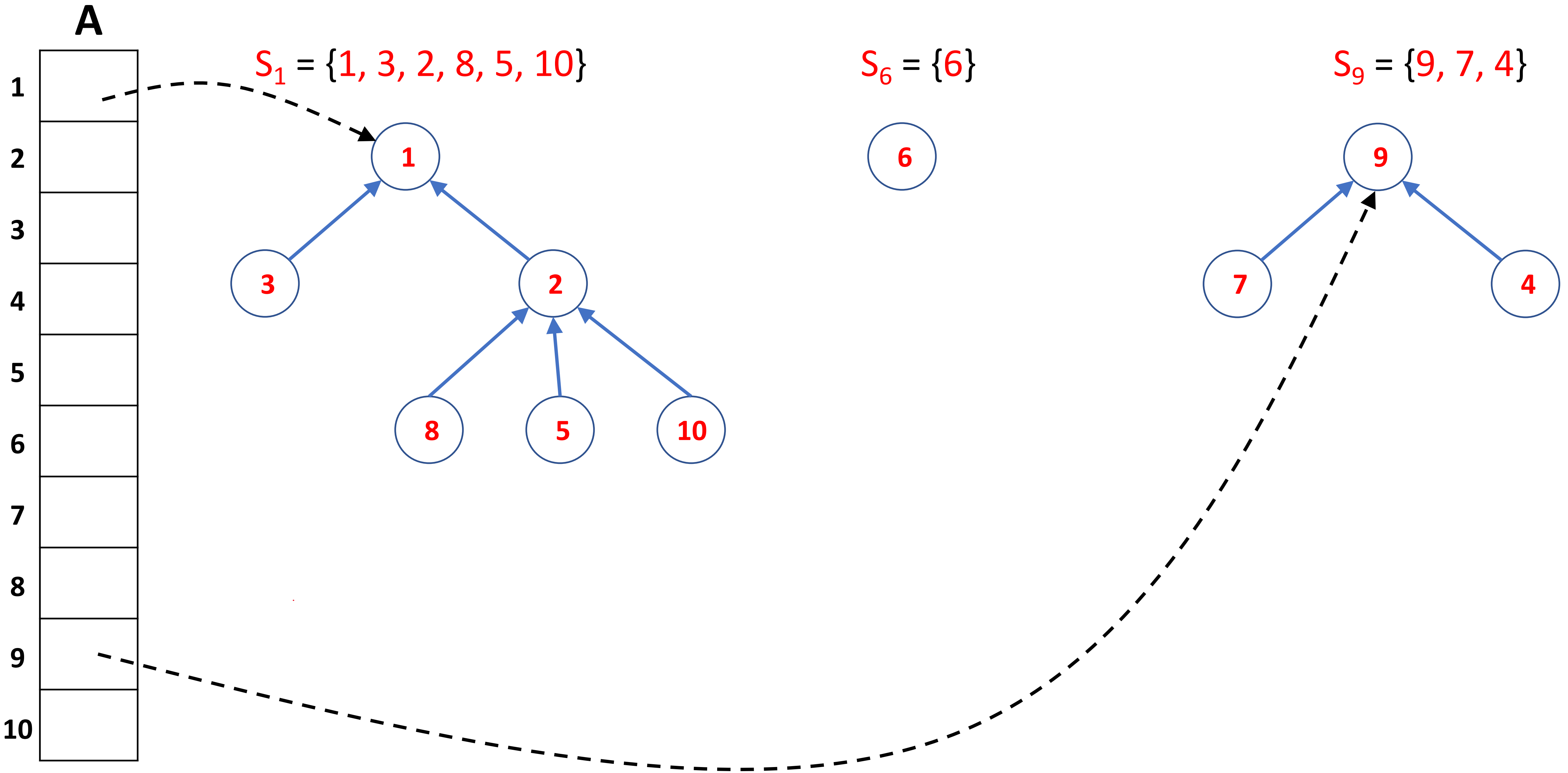
Find(5)



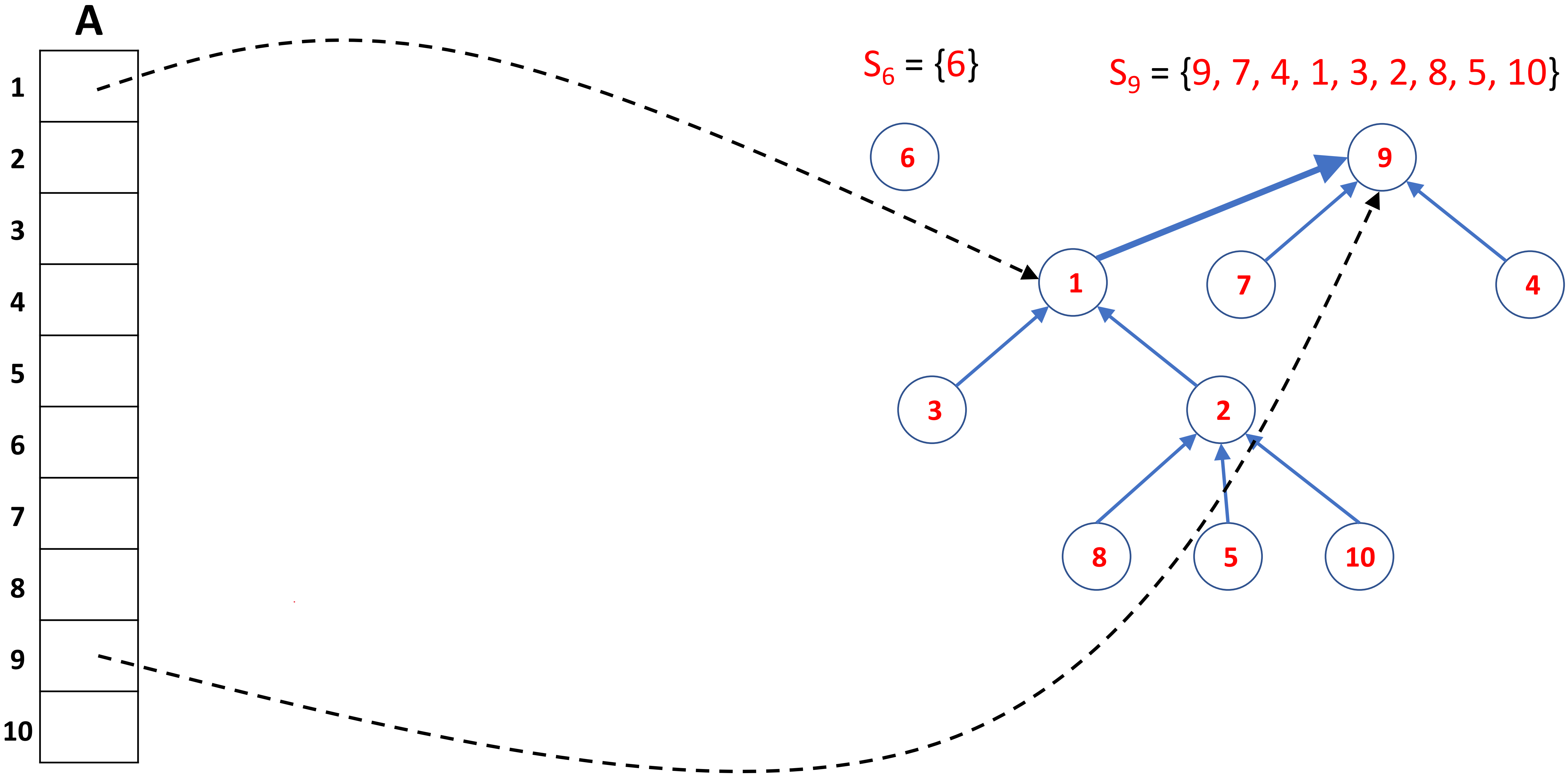
Find(5) : Return (ptr to) 1



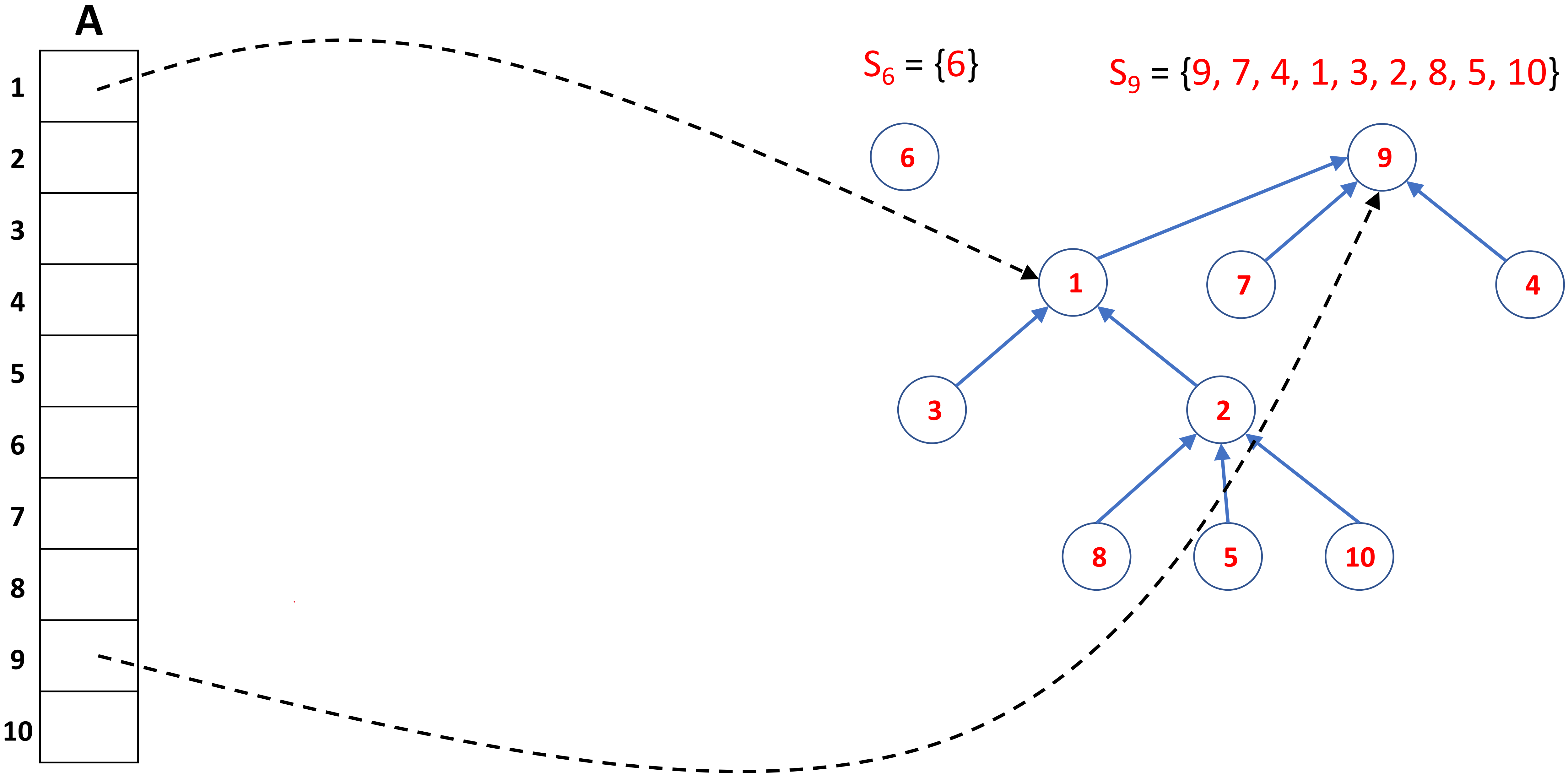
Union(1, 9)



Union(1, 9)



Union(1, 9) : Return (ptr to) 9

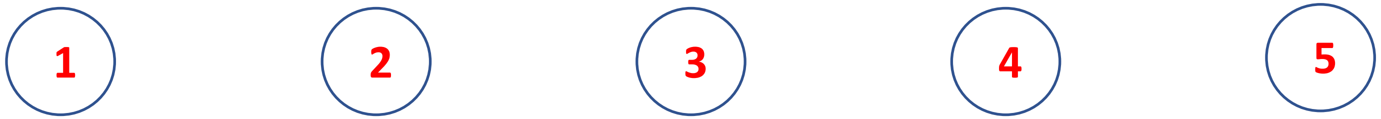


Operations: Example with n = 5

Initially : 1 2 3 4 5

Operations: Example with n = 5

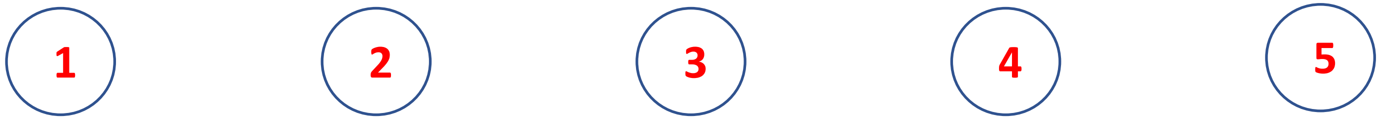
Initially :



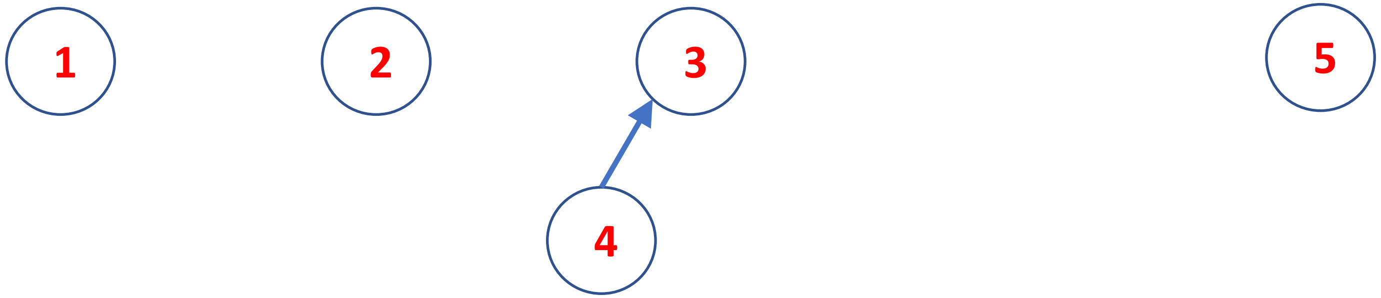
Union(4, 3) :

Operations: Example with n = 5

Initially :

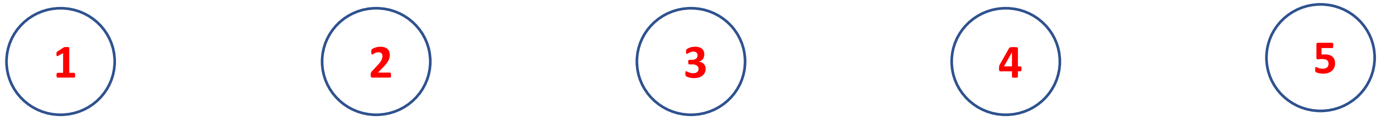


Union(4, 3) :

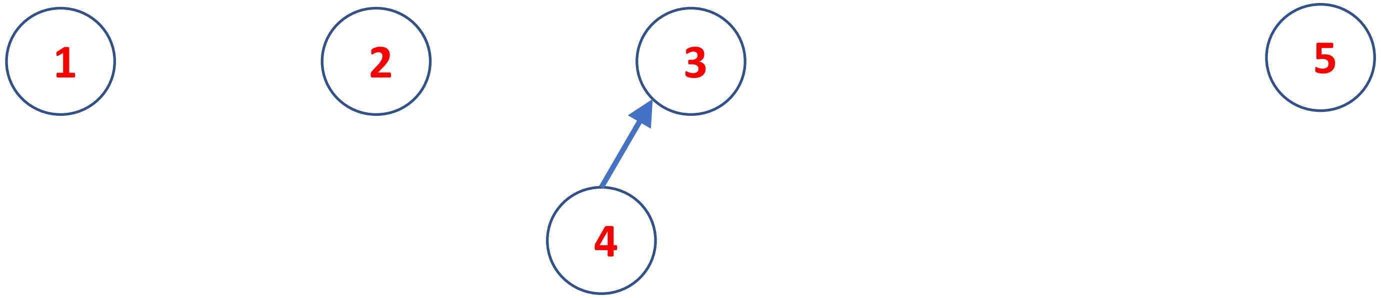


Operations: Example with n = 5

Initially :



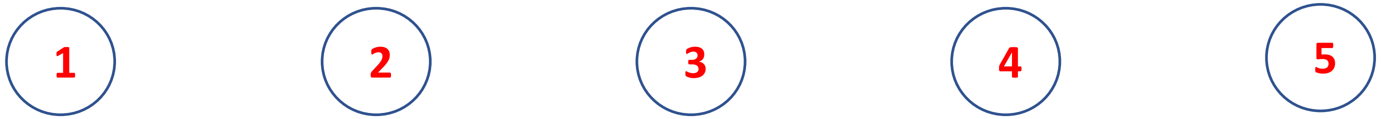
Union(4, 3) :



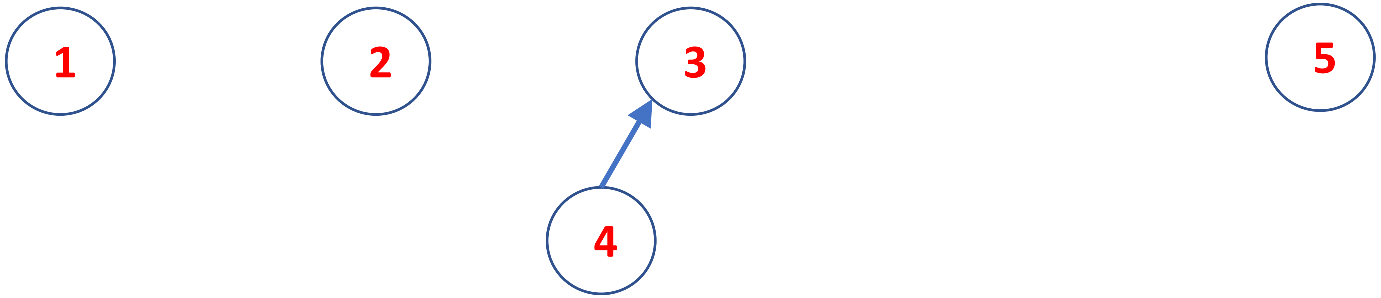
Union(5, 1) :

Operations: Example with n = 5

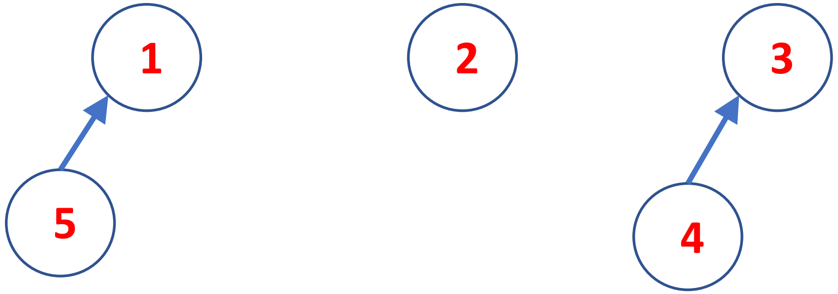
Initially :



Union(4, 3) :

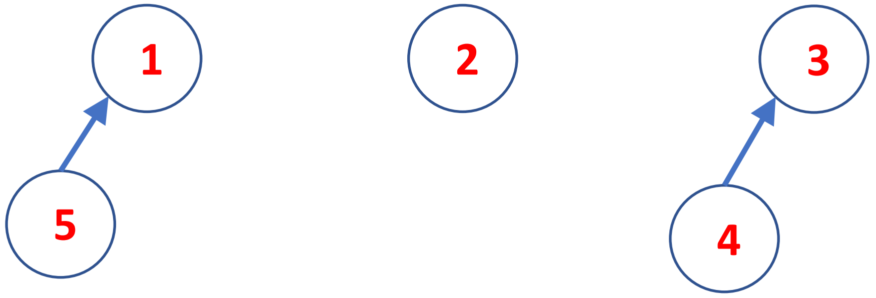


Union(5, 1) :



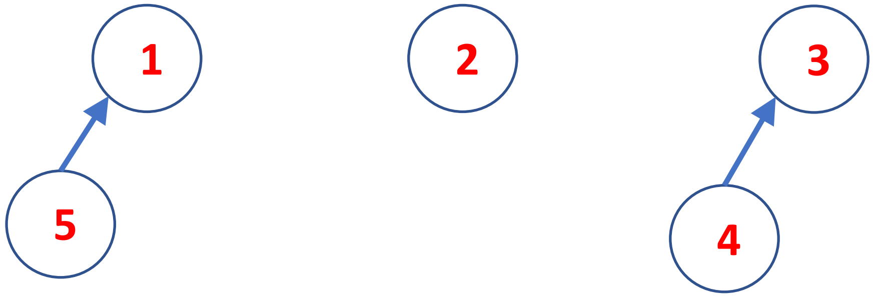
Operations: Example with n = 5

Union(5, 1) :



Operations: Example with n = 5

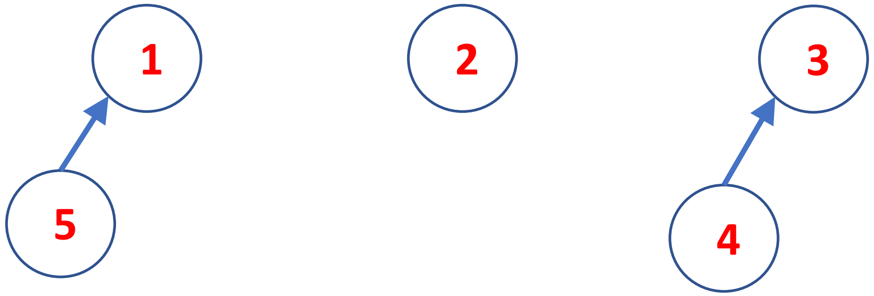
Union(5, 1) :



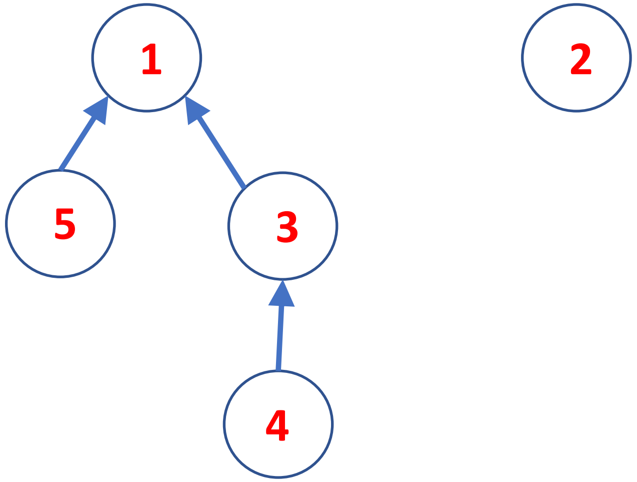
Union(3, 1) :

Operations: Example with n = 5

Union(5, 1) :

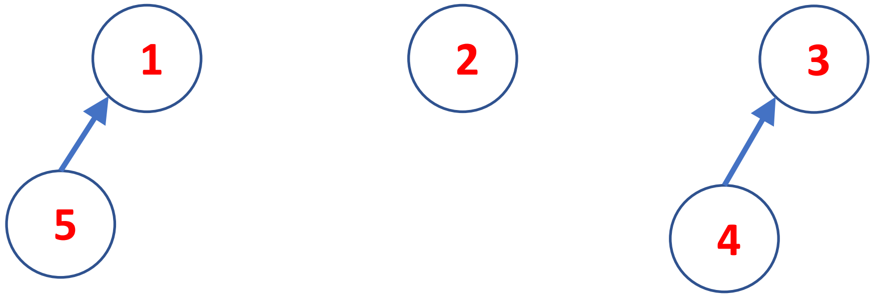


Union(3, 1) :

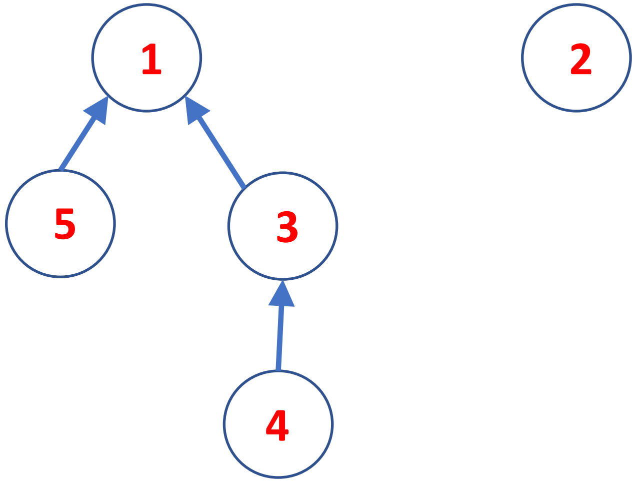


Operations: Example with $n = 5$

Union(5, 1) :



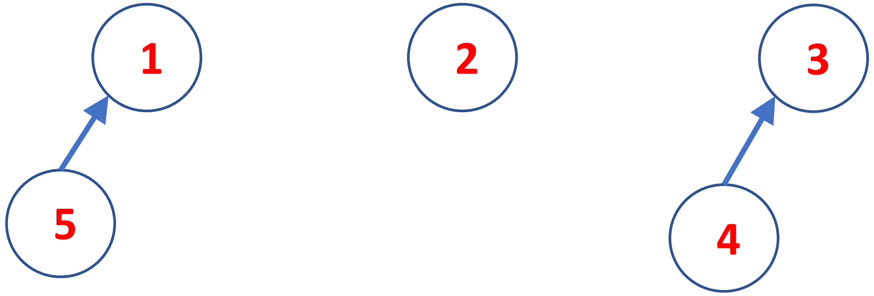
Union(3, 1) :



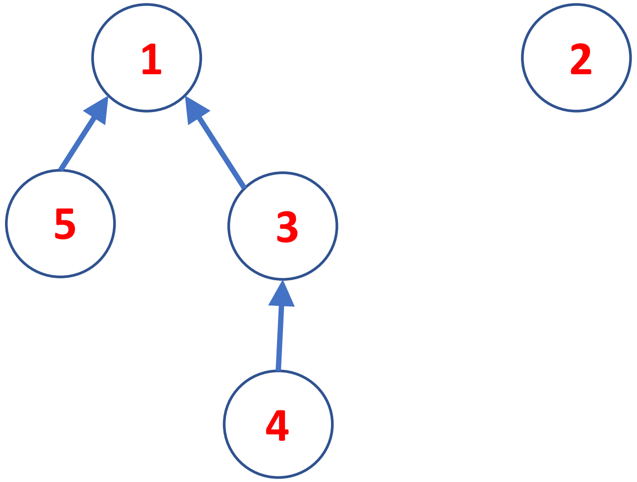
Union(2, 1) :

Operations: Example with n = 5

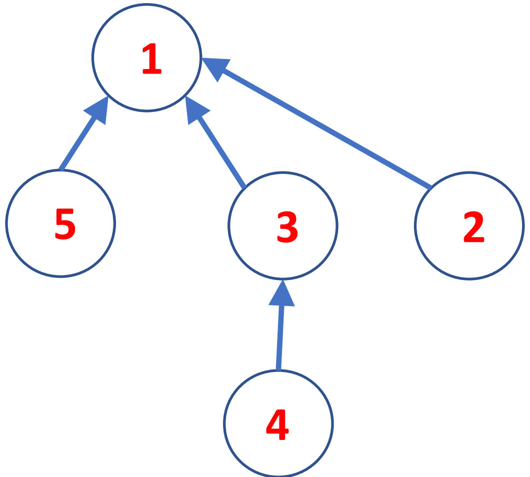
Union(5, 1) :



Union(3, 1) :



Union(2, 1) :



Operations

- **Find**(x): Follow path from x up to root, return ptr to the root

- **Union**(S_x, S_y): Make root of S_x the child of root of S_y

Operations

- **Find**(x): Follow path from x up to root, return ptr to the root

  Cost is O(1 + length of the **Find** path)

- **Union**(S_x, S_y): Make root of S_x the child of root of S_y

Operations

- **Find**(x): Follow path from x up to root, return ptr to the root

  Cost is O(1 + length of the **Find** path)

- **Union**(S_x, S_y): Make root of S_x the child of root of S_y

  Cost is O(1)

Disjoint Forest: Analysis of Time Complexity

- Cost of **Find** : O(1 + length of the **Find** path)
- Cost of **Union** : O(1)

Disjoint Forest: Analysis of Time Complexity

- Cost of **Find** : O(1 + length of the **Find** path)
- Cost of **Union** : O(1)

σ : Sequence of $n - 1$ **Unions** mixed with $m \geq n$ **Finds**

Disjoint Forest: Analysis of Time Complexity

- Cost of **Find** : O(1 + length of the **Find** path)
- Cost of **Union** : O(1)

σ : Sequence of n – 1 **Unions** mixed with m $\geq$ n **Finds**

What is the cost of σ, in the worst-case?

Disjoint Forest: Analysis of Time Complexity

Initially :

Disjoint Forest: Analysis of Time Complexity

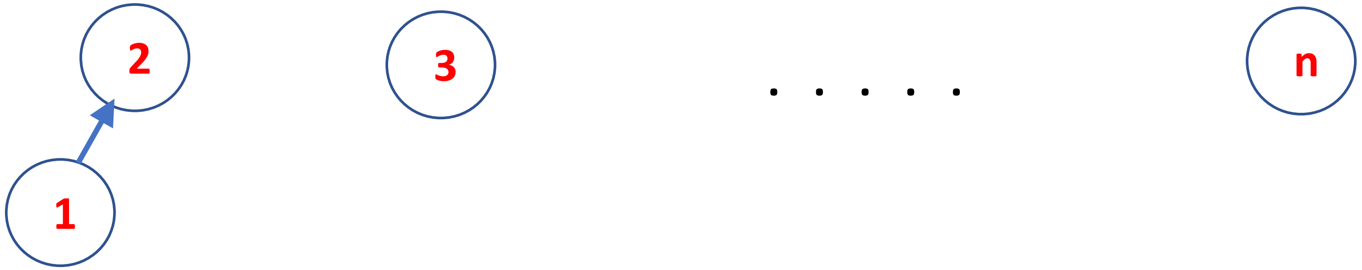
Initially :

Union(1, 2) :

Disjoint Forest: Analysis of Time Complexity

Initially :

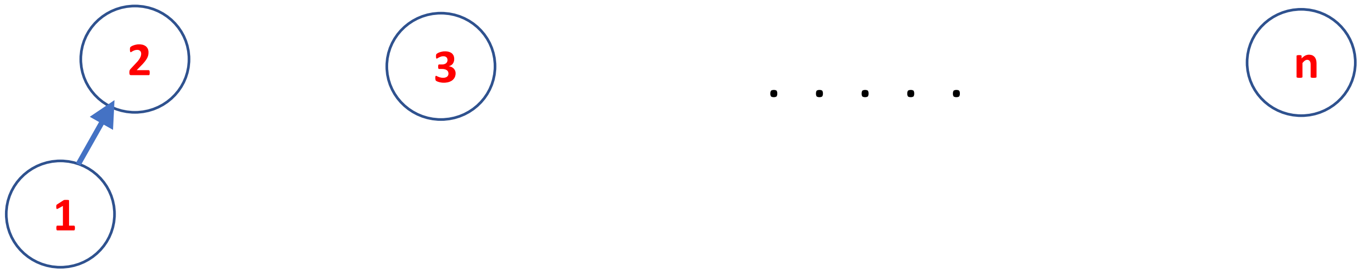
Union(1, 2) :



Disjoint Forest: Analysis of Time Complexity

Initially :

Union(1, 2) :

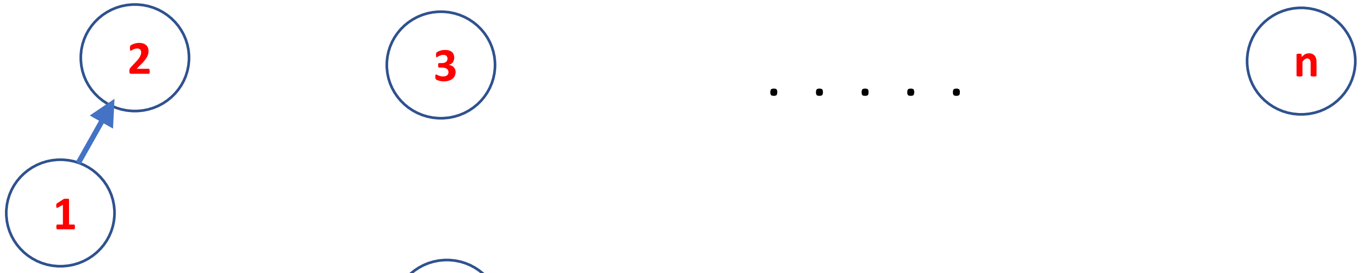


Union(2, 3) :

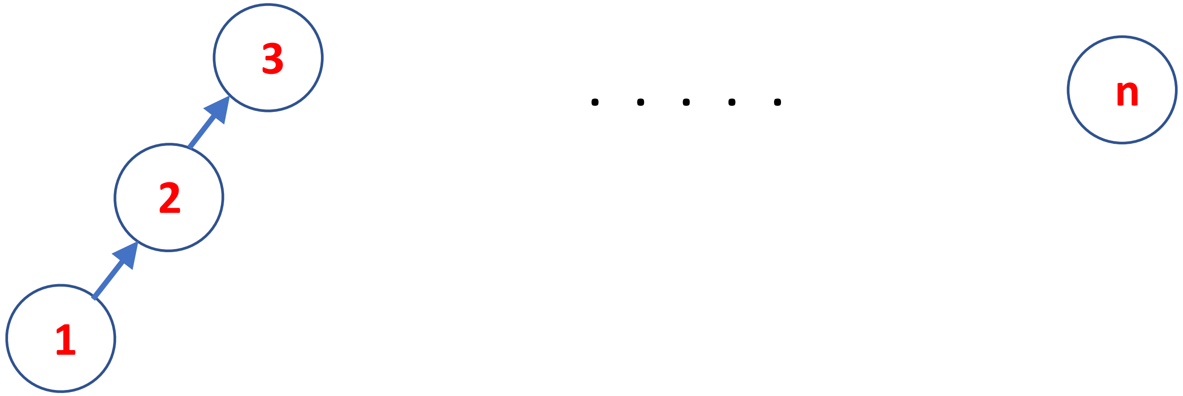
Disjoint Forest: Analysis of Time Complexity

Initially :

Union(1, 2) :



Union(2, 3) :



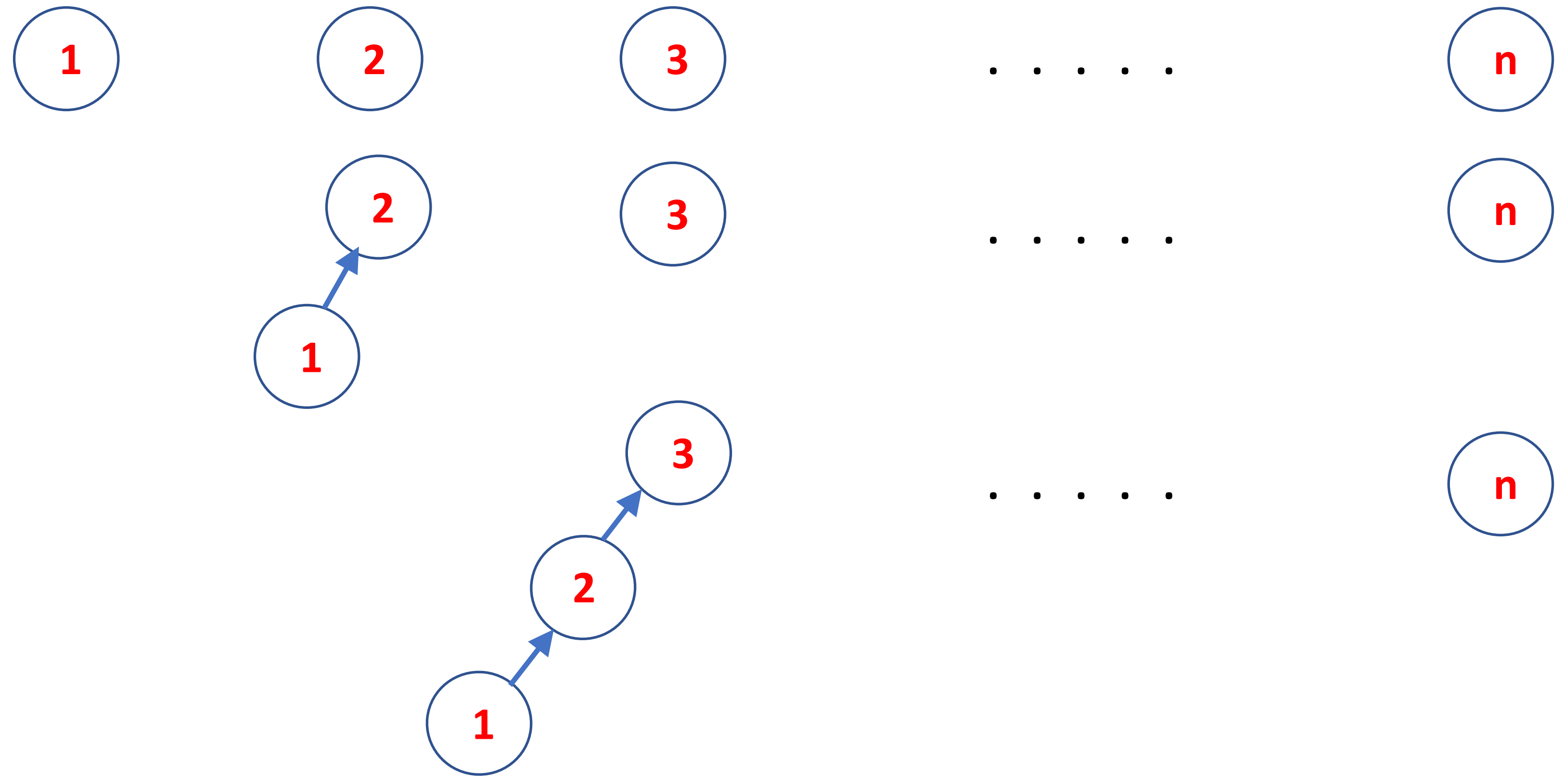
Disjoint Forest: Analysis of Time Complexity

Initially :

Union(1, 2) :

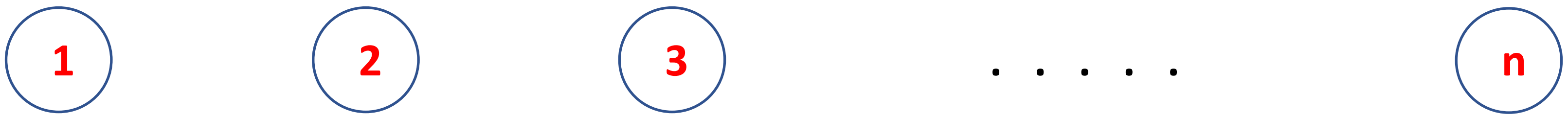
Union(2, 3) :

.
.
.
.
.

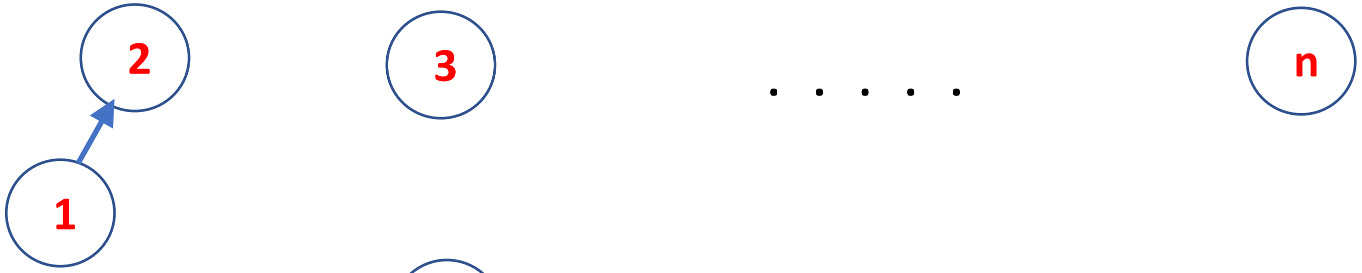


Disjoint Forest: Analysis of Time Complexity

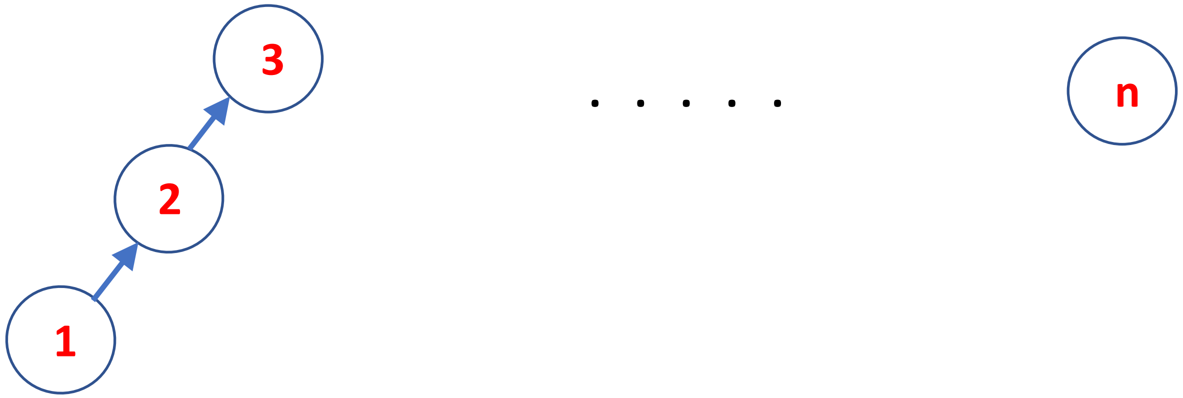
Initially :



Union(1, 2) :



Union(2, 3) :



.
.
.
.
.

Union(n-1, n) :

Disjoint Forest: Analysis of Time Complexity

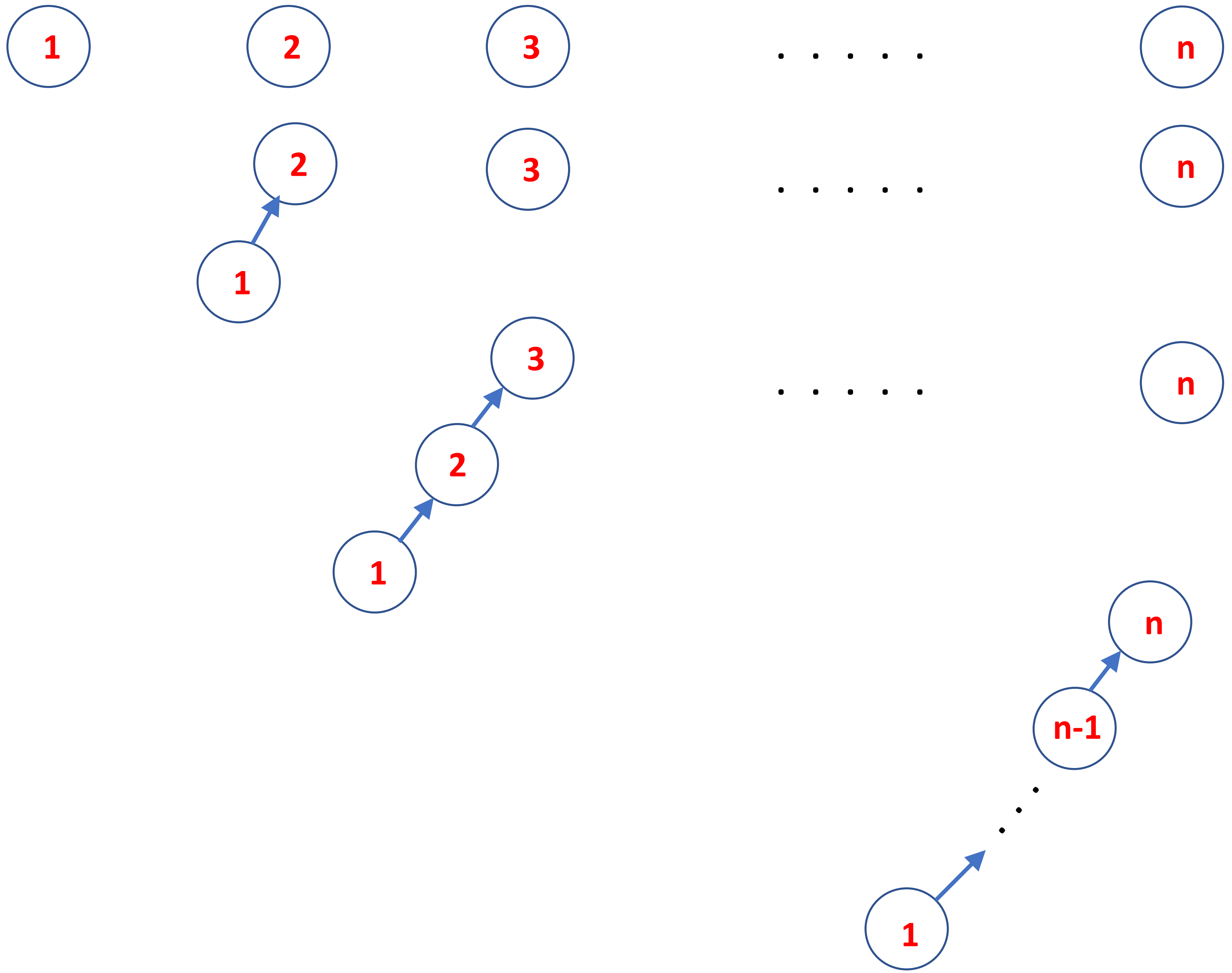
Initially :

Union(1, 2) :

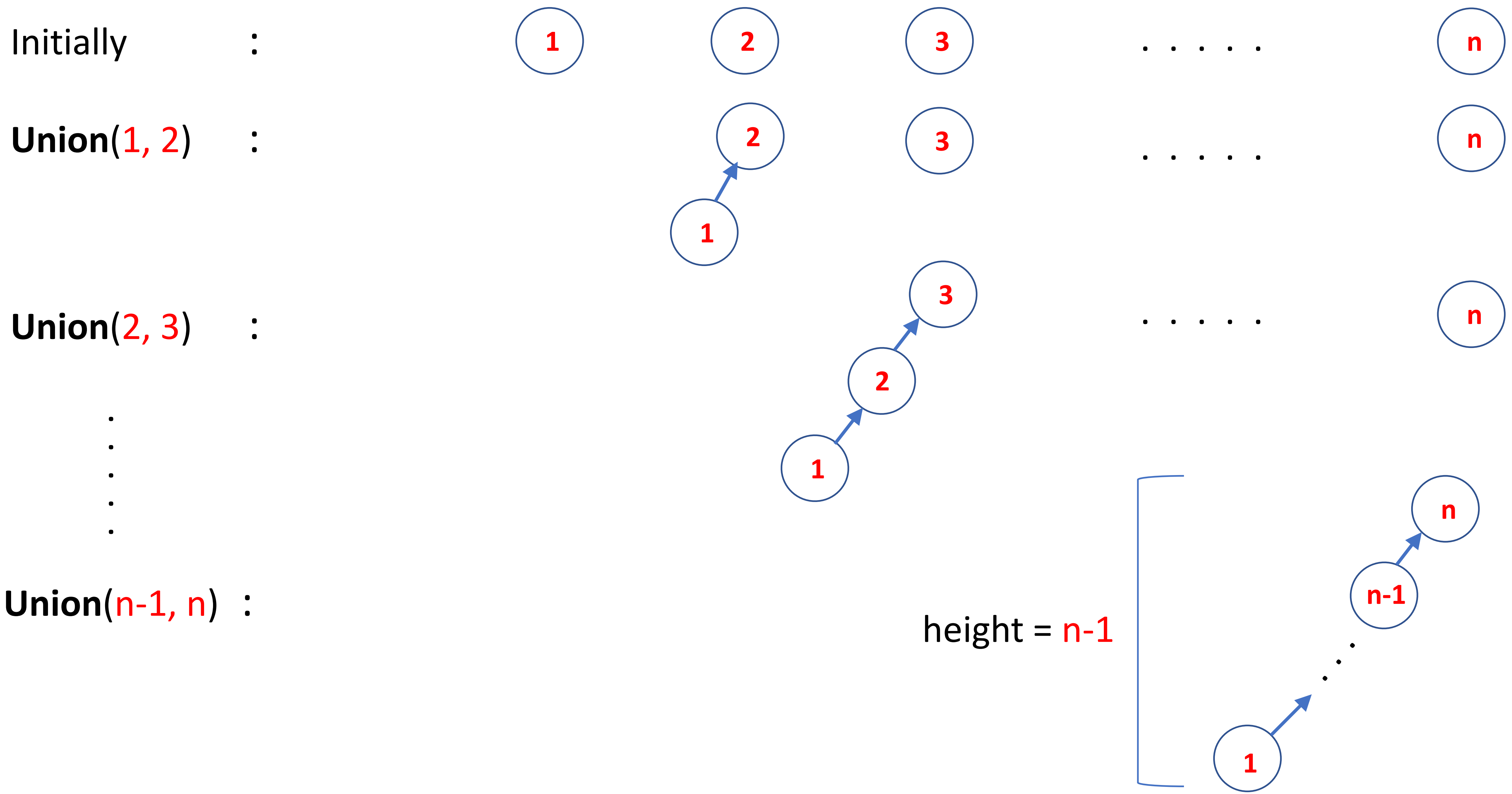
Union(2, 3) :

.
.
.
.
.

Union(n-1, n) :



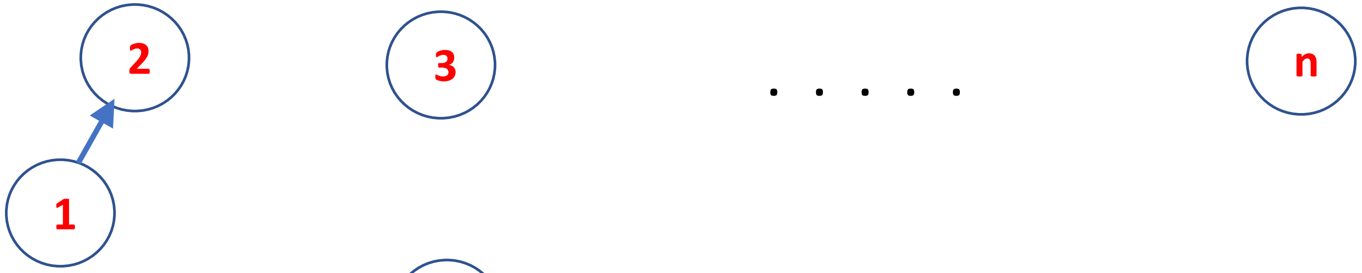
Disjoint Forest: Analysis of Time Complexity



Disjoint Forest: Analysis of Time Complexity

Initially :

Union(1, 2) :



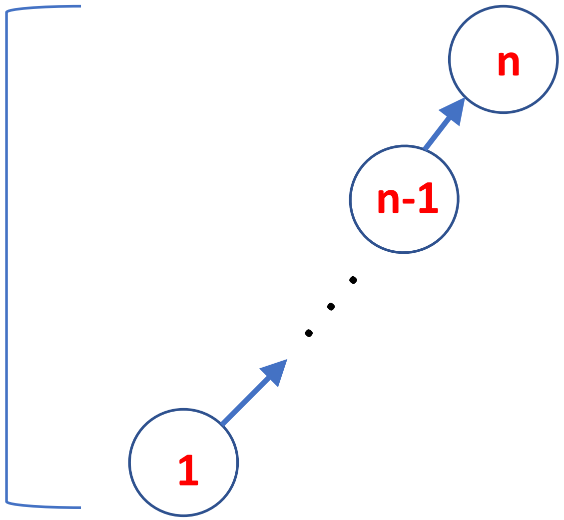
Union(2, 3) :

.
.
.
.
.

Union(n-1, n) :

height = n-1

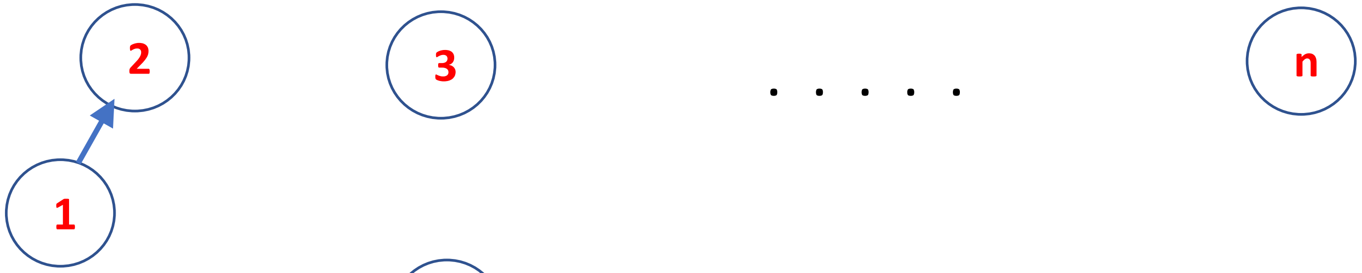
Cost of m **Find**(1) : Ω(m.n) !



Disjoint Forest: Analysis of Time Complexity

Initially :

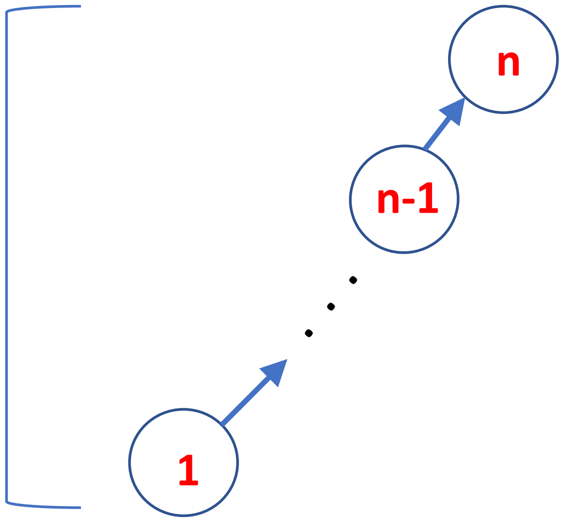
Union(1, 2) :



Union(2, 3) :

.
.
.
.
.

Union(n-1, n) :



height = n-1

Cost of m **Find**(1) : Ω(m.n) !

$\Rightarrow$ Cost of σ, in the worst-case, is Ω(m.n)

Disjoint Forest: Analysis of Time Complexity

Worst-case cost of executing σ is Ω(m.n)

Cost of each **Find** : O(1 + length of the **Find** path)

To reduce cost of σ, reduce length of **Find** paths

Disjoint Forest: Analysis of Time Complexity

Worst-case cost of executing σ is Ω(m.n)

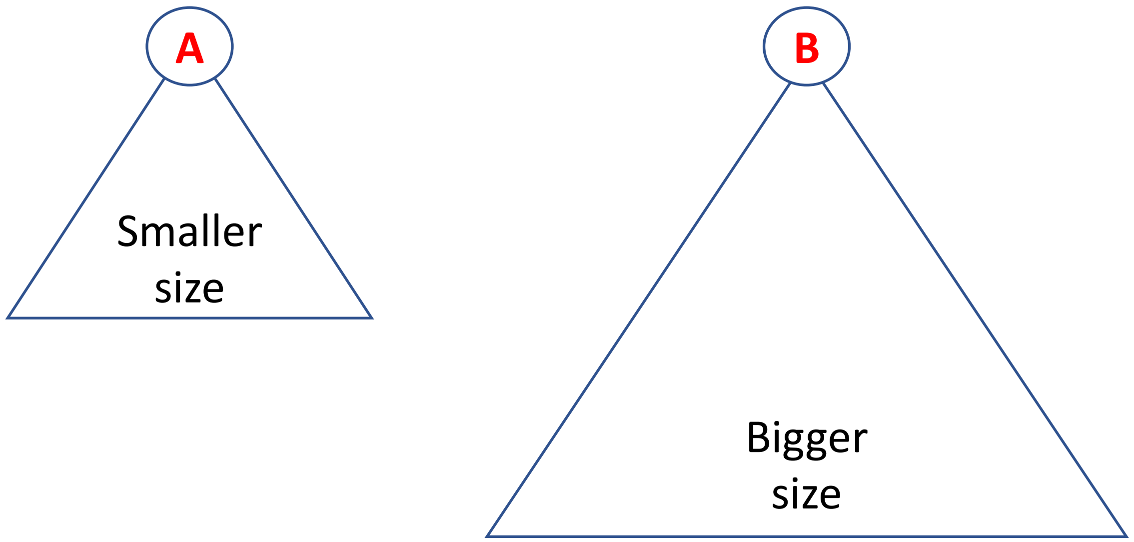
Cost of each **Find** : O(1 + length of the **Find** path)

To reduce cost of σ, reduce length of **Find** paths

$\Rightarrow$ reduce height of the trees formed during the execution of σ

Heuristic 1: **Weighted Union (WU)** by Size

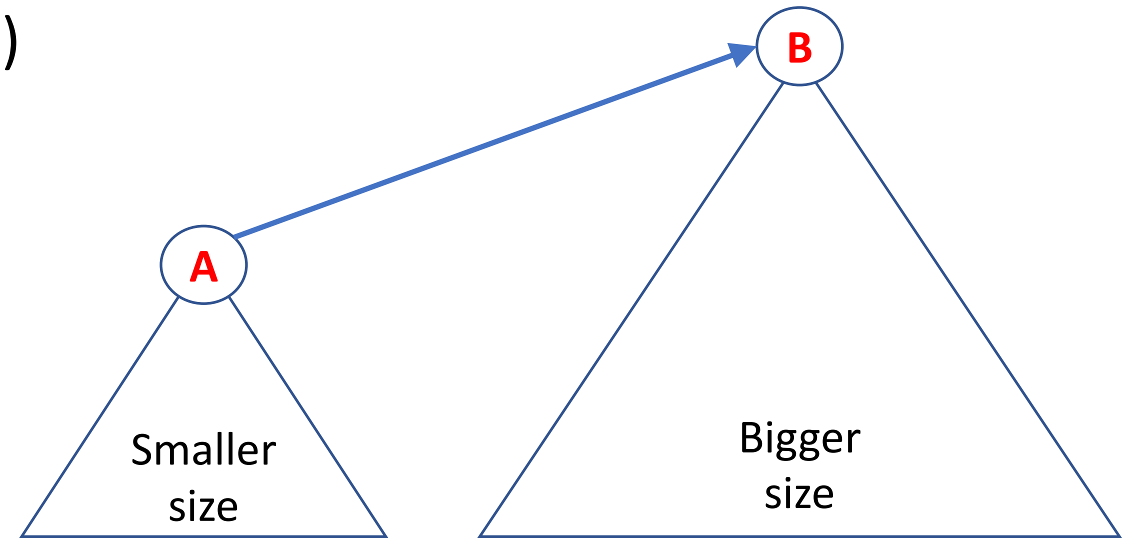
Union(A, B)



size: # of nodes in the tree

Heuristic 1: **Weighted Union (WU)** by Size

Union(A, B)

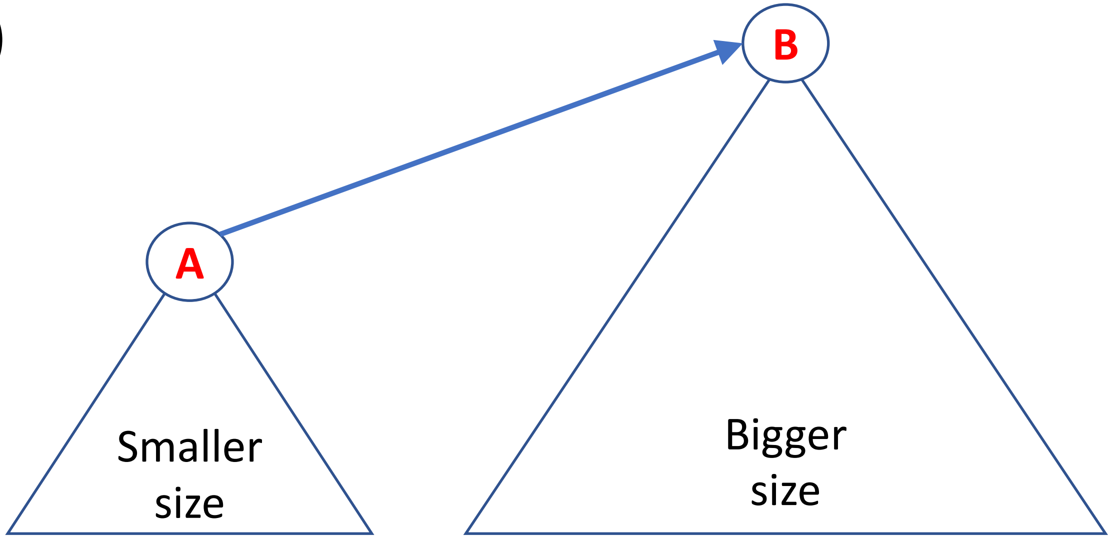


size: # of nodes in the tree

WU rule (by size): Smaller size tree becomes the child of the bigger size tree

Heuristic 1: **Weighted Union (WU)** by Size

Union(A, B)



size: # of nodes in the tree

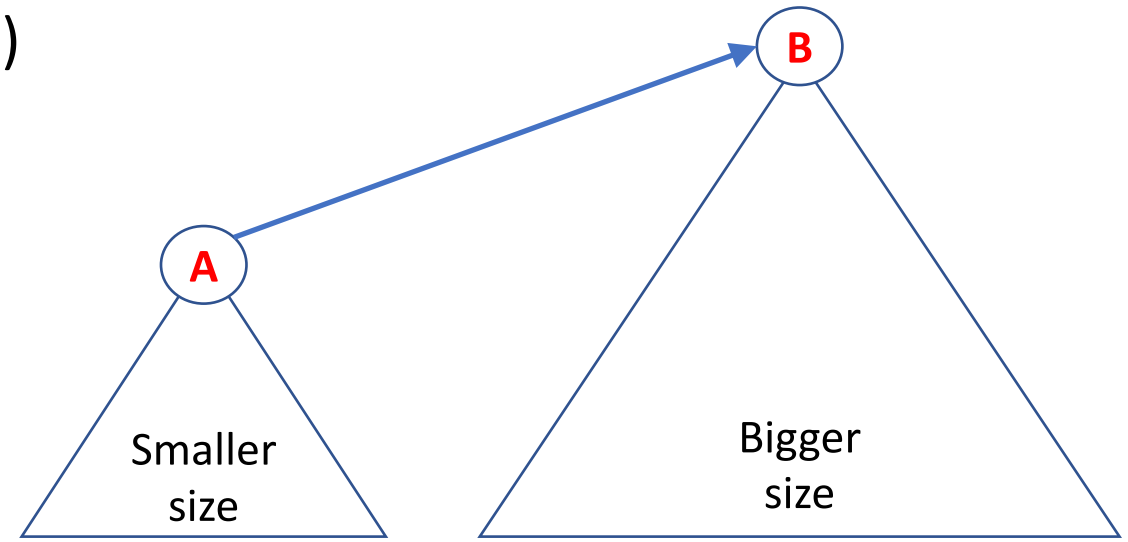
WU rule (by size): Smaller size tree becomes the child of the bigger size tree

With **WU**:

- Any tree T created during the execution of σ has height at most $\log_2 n$

Heuristic 1: **Weighted Union (WU)** by Size

Union(A, B)



size: # of nodes in the tree

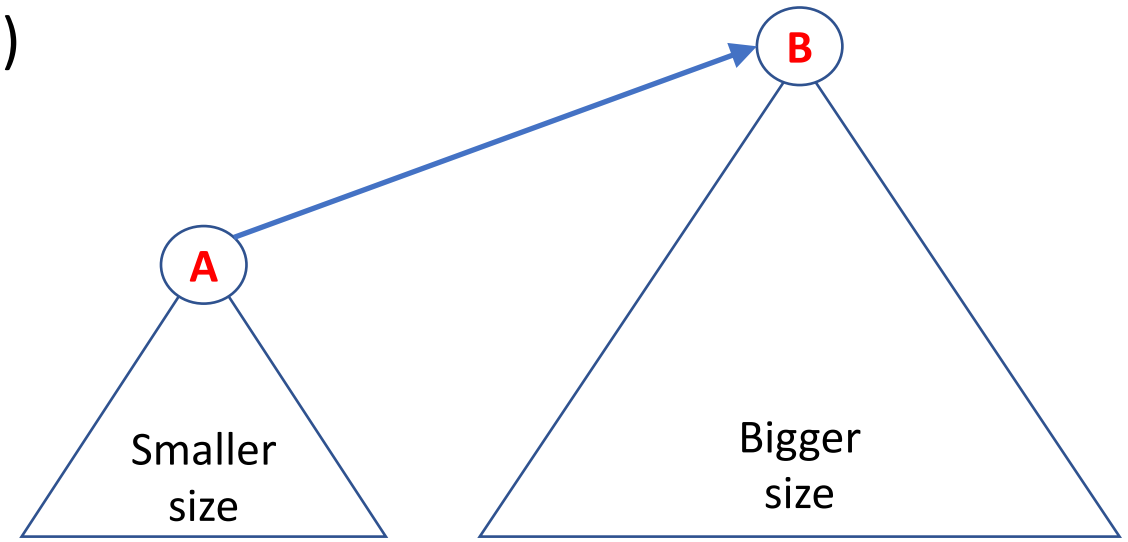
WU rule (by size): Smaller size tree becomes the child of the bigger size tree

With **WU**:

- Any tree T created during the execution of σ has height at most $\log_2 n$
- The worst-case cost of executing σ is $O(n + m \log n)$

Heuristic 1: **Weighted Union** (**WU**) by Size

Union(A, B)



size: # of nodes in the tree

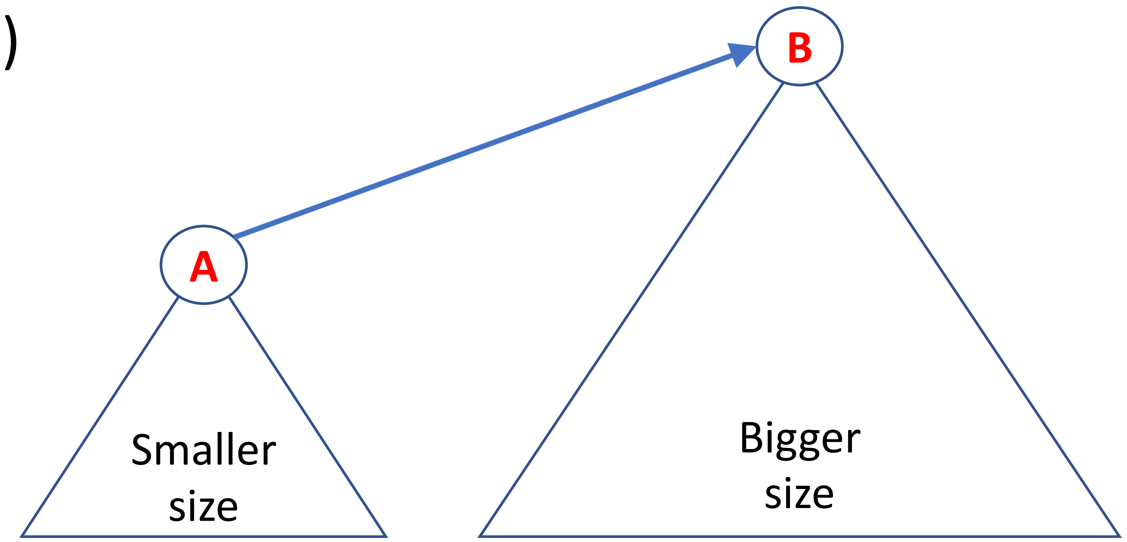
WU rule (by size): Smaller size tree becomes the child of the bigger size tree

With **WU**:

- Any tree T created during the execution of σ has height at most $\log_2 n$
- The worst-case cost of executing σ is $O(m \log n)$

Heuristic 1: **Weighted Union** (**WU**) by Size

Union(A, B)



size: # of nodes in the tree

WU rule (by size): Smaller size tree becomes the child of the bigger size tree

With **WU**:

- Any tree T created during the execution of σ has height at most $\log_2 n$
- The worst-case cost of executing σ is O(m log n)

We now prove this!

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Since $|T| \leq n$, $2^h \leq |T| \leq n \Rightarrow$ $h \leq \log_2 n$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

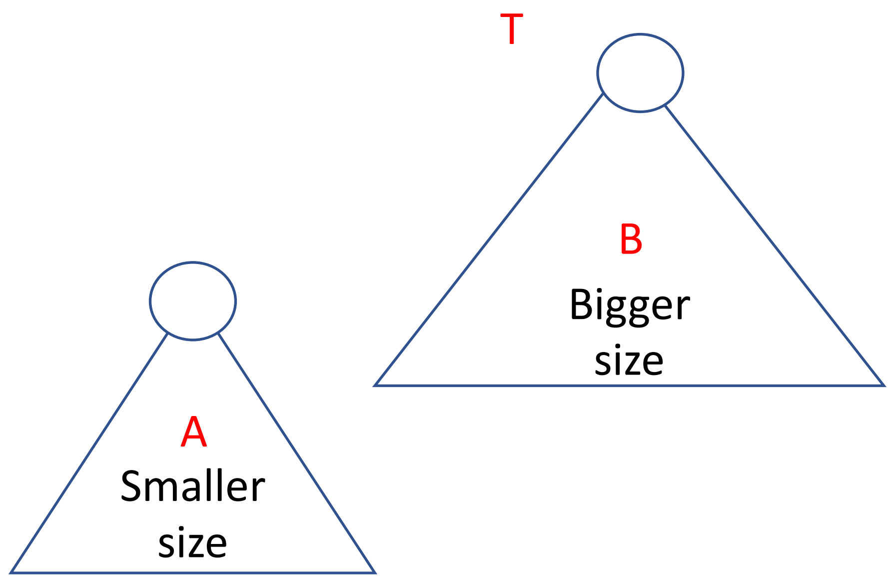
Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

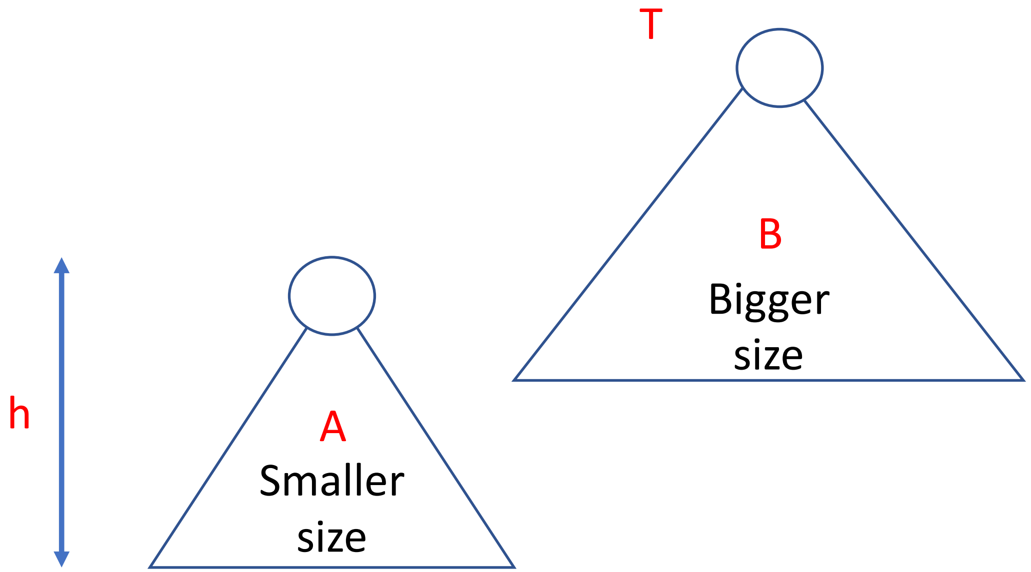
- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

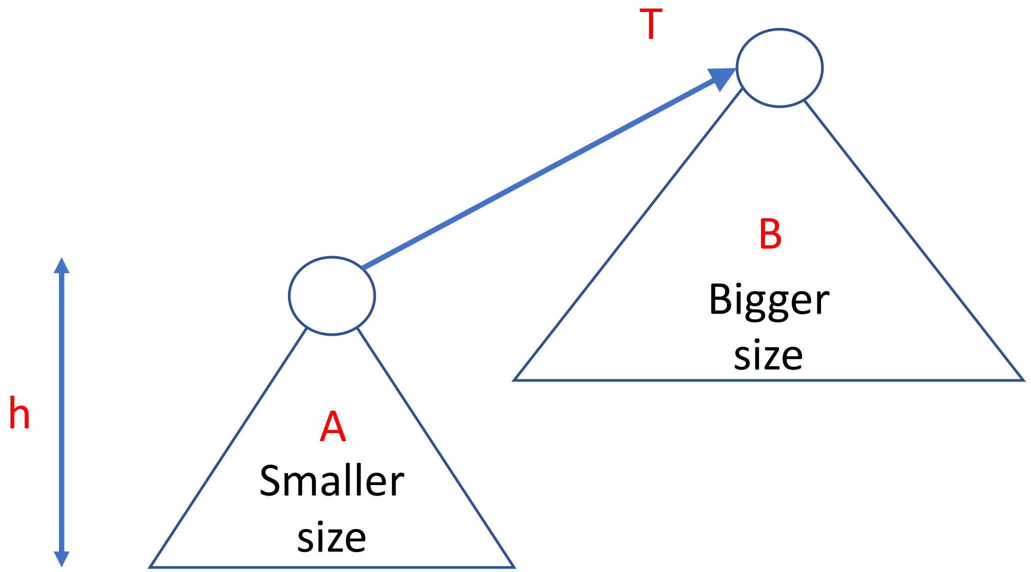
- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

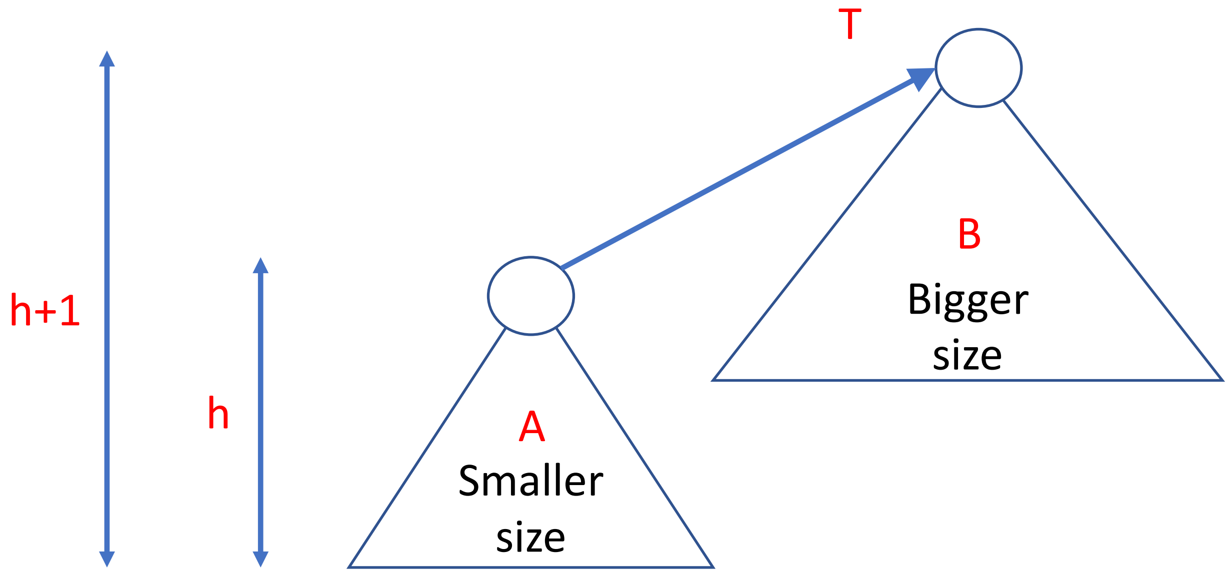
- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

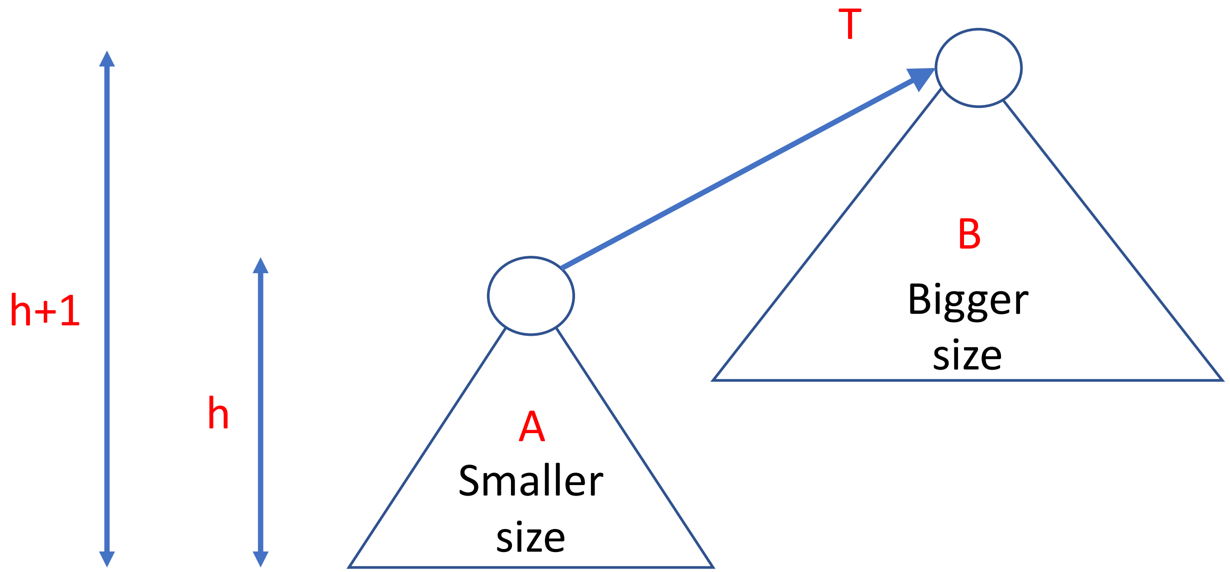
- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$

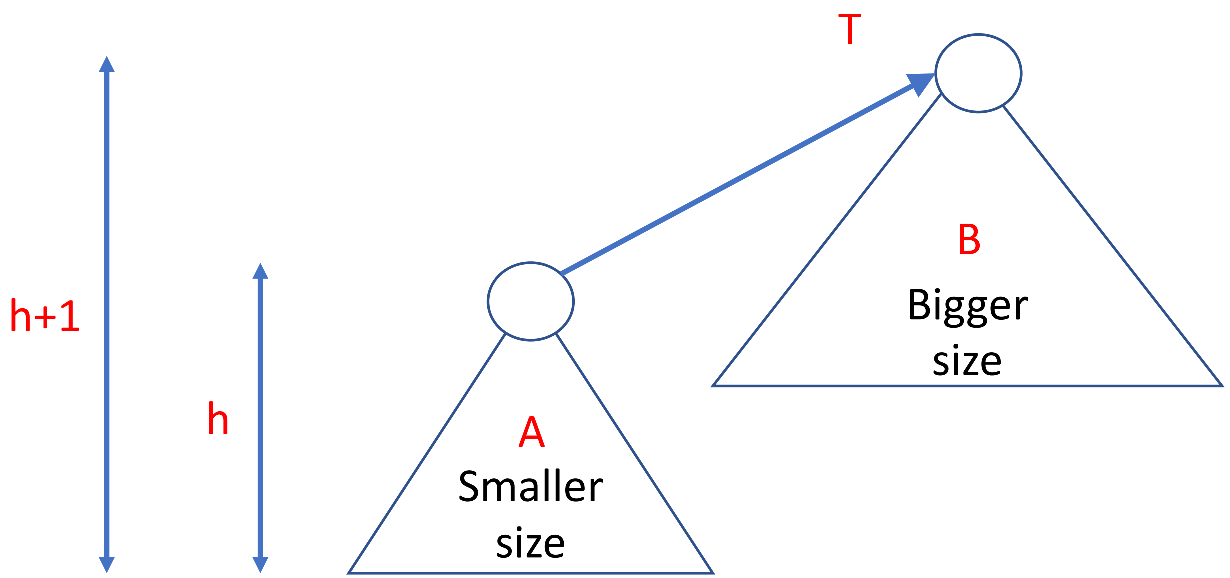


By I.H. : $|A| \geq 2^h$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



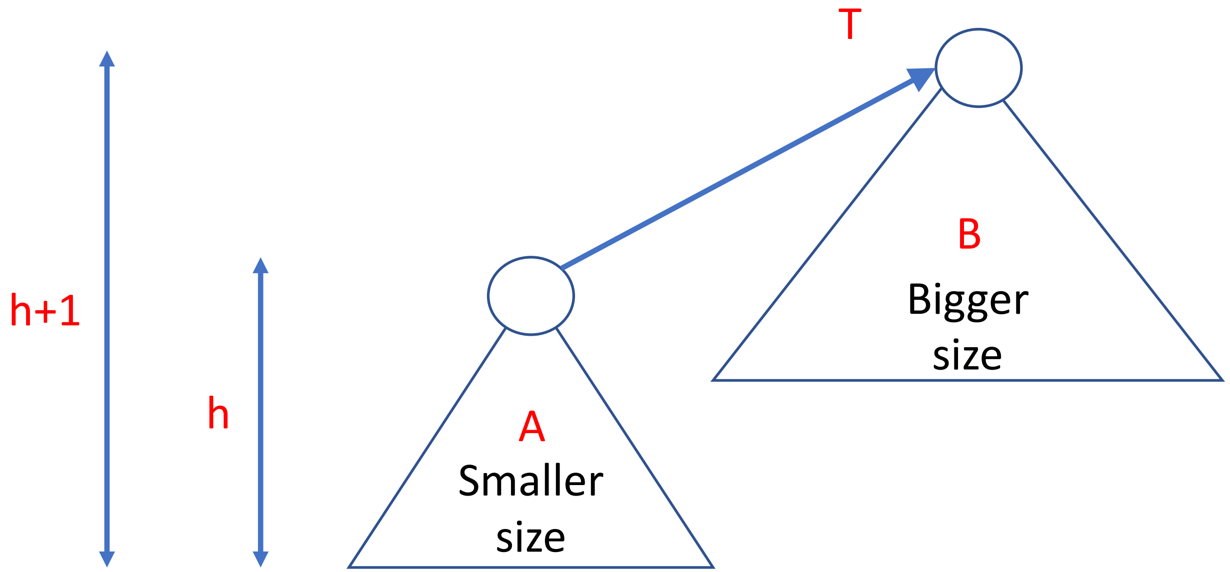
By I.H. : $|A| \geq 2^h$

By **WU** : $|B| \geq |A|$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



By I.H. : $|A| \geq 2^h$

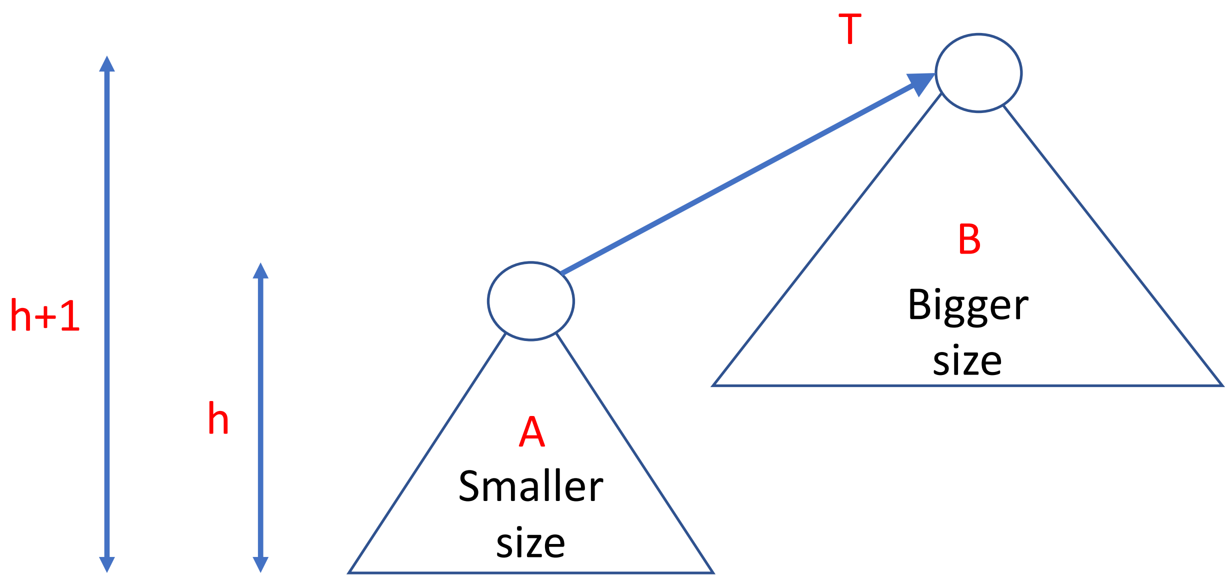
By **WU** : $|B| \geq |A|$

$|B| \geq 2^h$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



By I.H. : $|A| \geq 2^h$

By **WU** : $|B| \geq |A|$

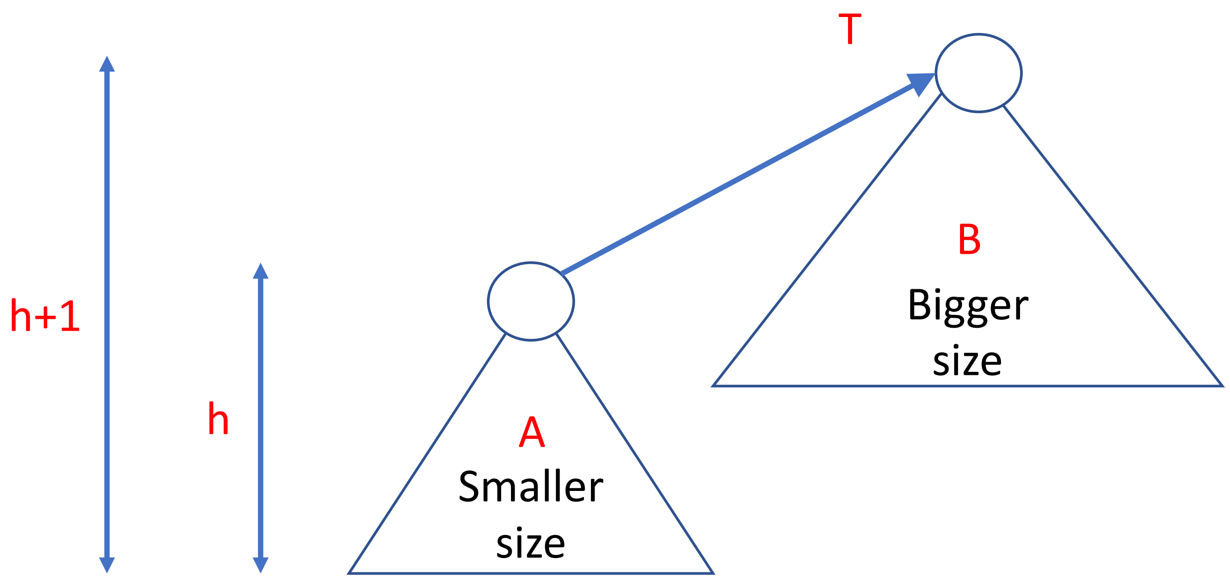
$|B| \geq 2^h$

$|T| = |A| + |B|$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



By I.H. : $|A| \geq 2^h$

By **WU** : $|B| \geq |A|$

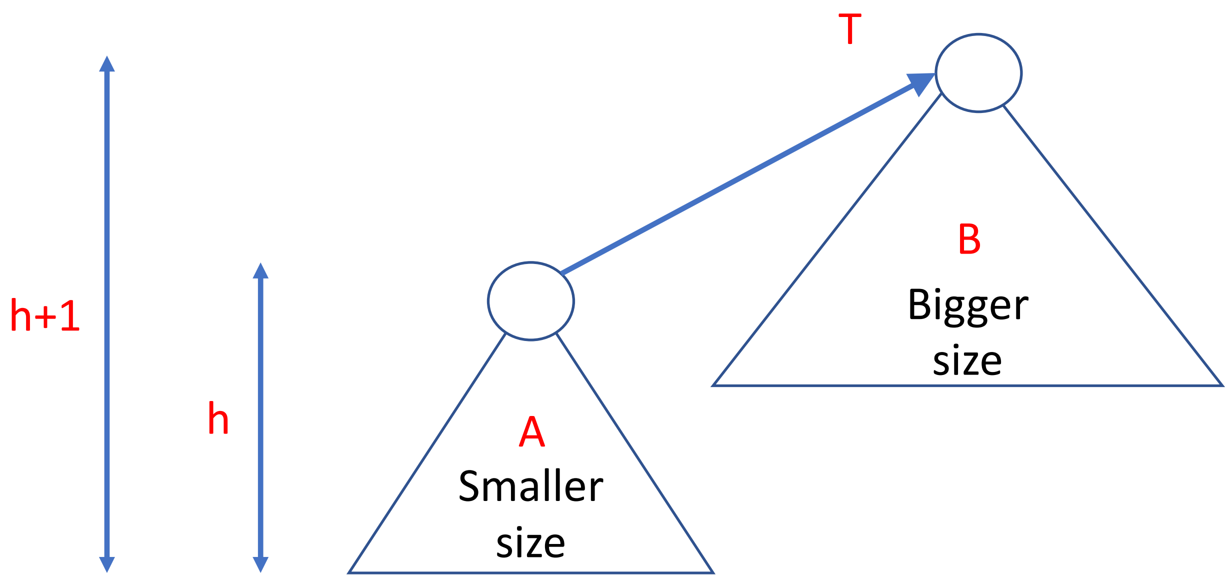
$|B| \geq 2^h$

$|T| = |A| + |B| \geq 2^h + 2^h$

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for $h+1$



By I.H. : $|A| \geq 2^h$

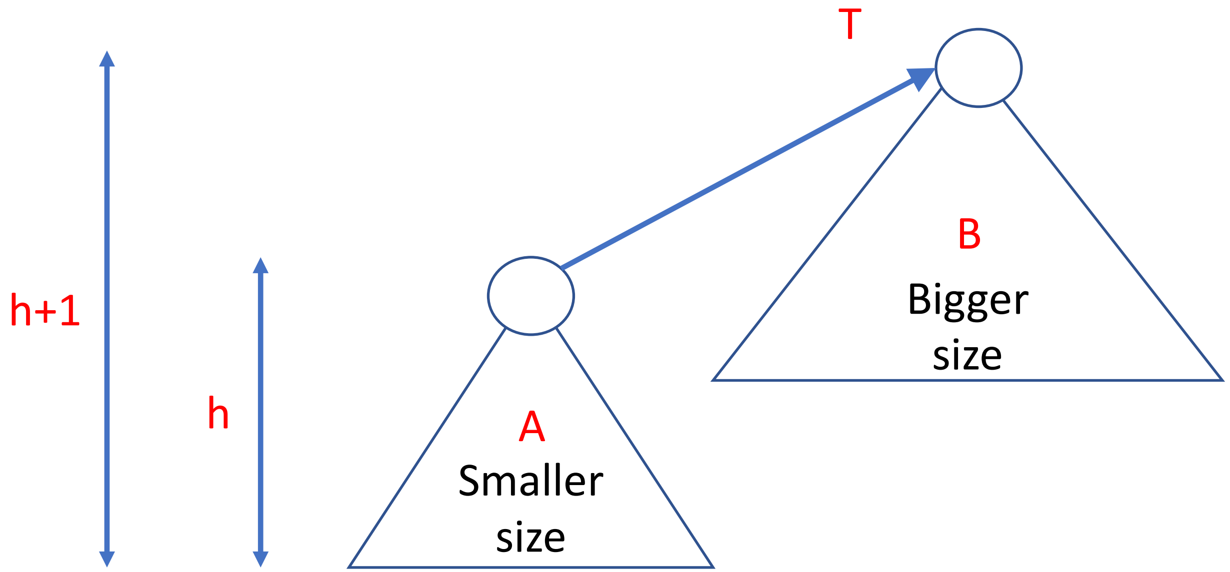
By **WU** : $|B| \geq |A|$
$|B| \geq 2^h$

$|T| = |A| + |B| \geq 2^h + 2^h = 2^{h+1}$ nodes

Lemma: With **WU**, any tree T of height h created during the execution of σ has at least 2^h nodes, i.e., $|T| \geq 2^h$

Proof Sketch: Induction on h

- $h = 0$. Any tree of height zero has at least $2^0 = 1$ nodes
- Suppose the lemma holds for $h \geq 0$ (I.H.). We show it holds for h+1



By I.H. : $|A| \geq 2^h$

By **WU** : $|B| \geq |A|$

$|B| \geq 2^h$

$|T| = |A| + |B| \geq 2^h + 2^h = 2^{h+1}$ nodes $\Rightarrow$ T has at least 2^{h+1} nodes