CSC263H
Data Structures and Analysis

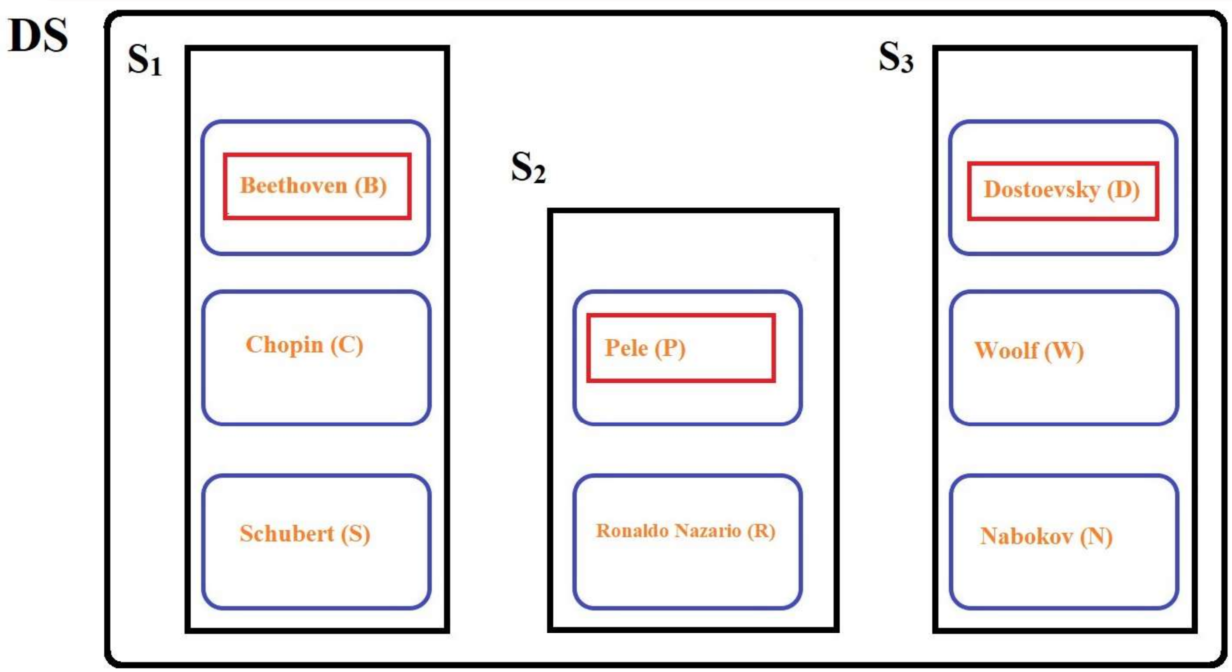
Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 7

Disjoint Sets

Objects:

- A collection of nonempty **disjoint sets** $S = \{S_1, S_2, ..., S_k\}$.
  Two sets S_i, S_j are **disjoint** iff $S_i \cap S_j = \{\}$.
  That is, for each pair $S_i, S_j \in S$ $(1 \leq i, j \leq k)$, we have $S_i \cap S_j = \{\}$.
- Each set is identified by a **unique** element called its **representative**.



$DS = \{S_1, S_2, S_3\}$

Disjoint Set

Objects:

- A collection of nonempty **disjoint sets** $S = \{S_1, S_2, ..., S_k\}$.
- Each set is identified by a **unique** element called its **representative**.

Operations:

- MakeSet(DS, v): Given element v that does not already belong to one of the sets, create a new set $\{v\}$. The new item is the **representative** element of the new set.
- FindSet(DS, x): return the **representative** element of the (unique) set containing x.
  Note: We know the set that contains x, we just need to find its **representative**.
- Union(DS, x, y): Given two items x and y, merge the sets that contain these items. The representative of the new set might be one of the two representatives of the original sets, one of x or y, or something completely different.
  Note: if both x and y belong to the same set already, operation has no effect.

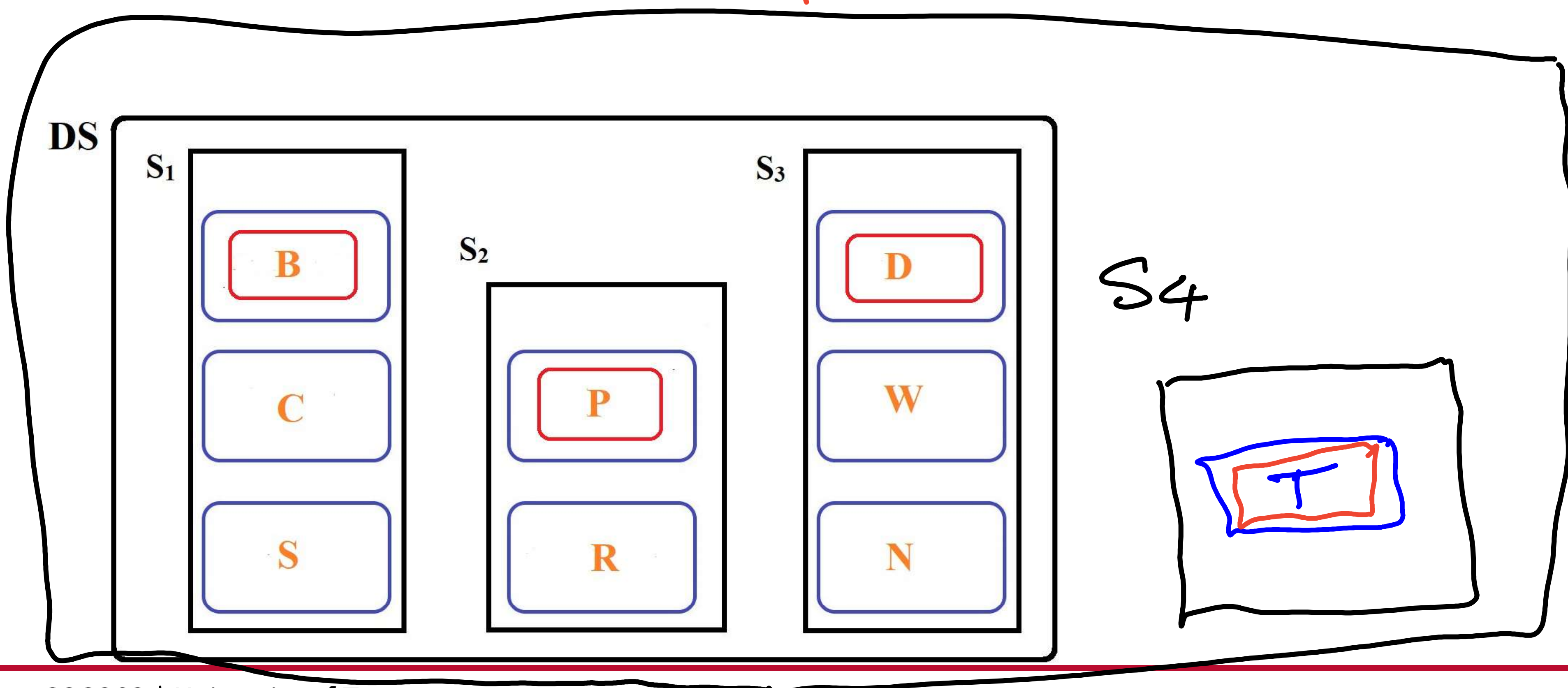
Disjoint Set ADT

$DS = \{S_1, S_2, S_3\}$

FindSet(DS, *S*) returns: B

FindSet(DS, *R*) returns: P

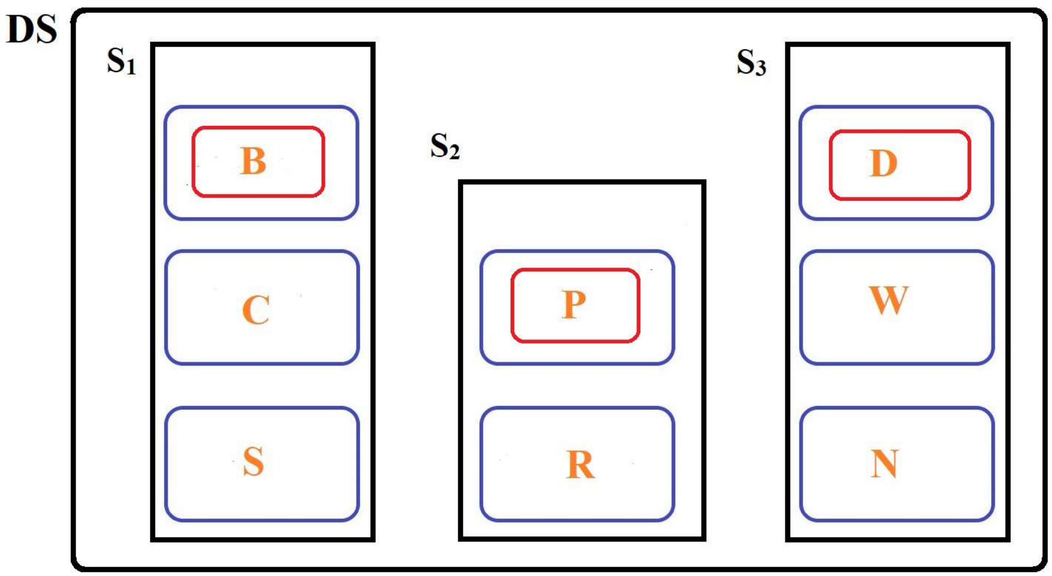
MakeSet(DS, *T*) $\Rightarrow$ $DS = \{S_1, S_2, S_3, S_4\}$



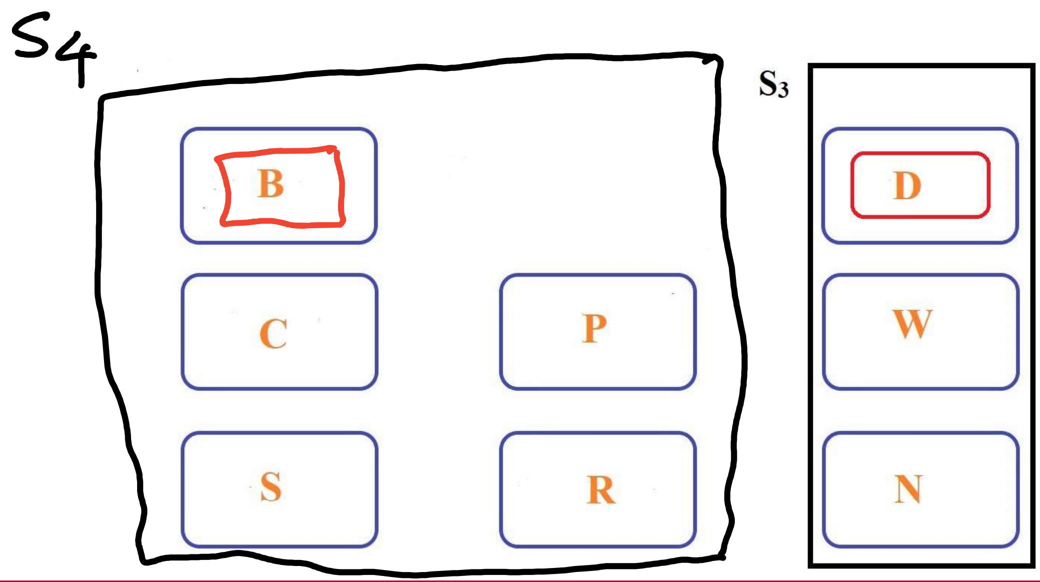
Disjoint Set ADT

Union(DS, C, R)

$DS = \{S_1, S_2, S_3\}$



$DS = \{S_4, S_3\}$

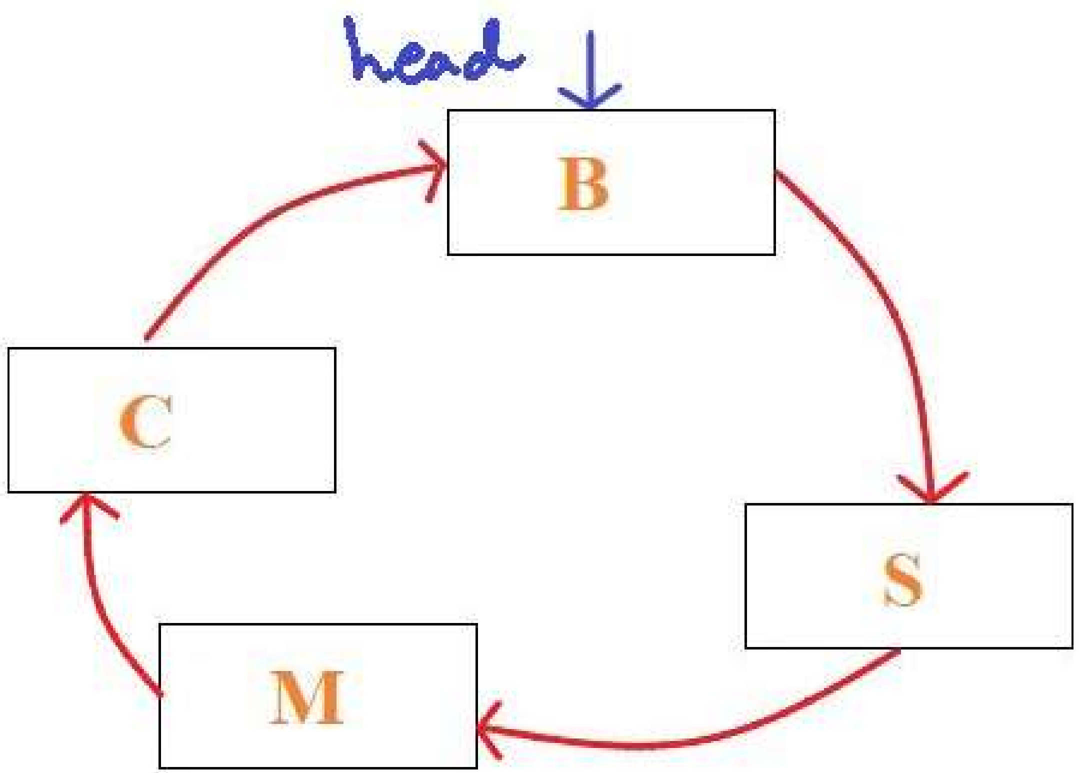


Data Structures for Disjoint Sets

1. Circularly-linked Lists.
2. Linked Lists with Extra Pointer.
3. Linked Lists with Extra Pointer and with Union-by-Weight.
4. Trees.
5. Trees with Union-by-Rank.
6. Trees with Path-Compression.
7. Trees with Union-by-Rank and Path Compression.

Disjoint Sets: Circularly-linked Lists

- One circularly-linked list for each set.
- Head of the linked list is the **representative** of the set.



MakeSet(DS, v):

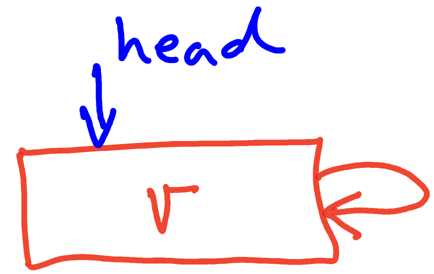
- create a new linked list with a node x storing element v;
- set $x.next = x$.

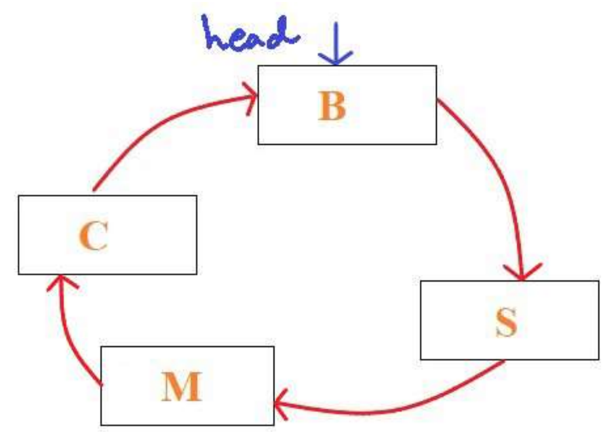
FindSet(DS, x):

- follow the links until reaching the head node r;
- return r.

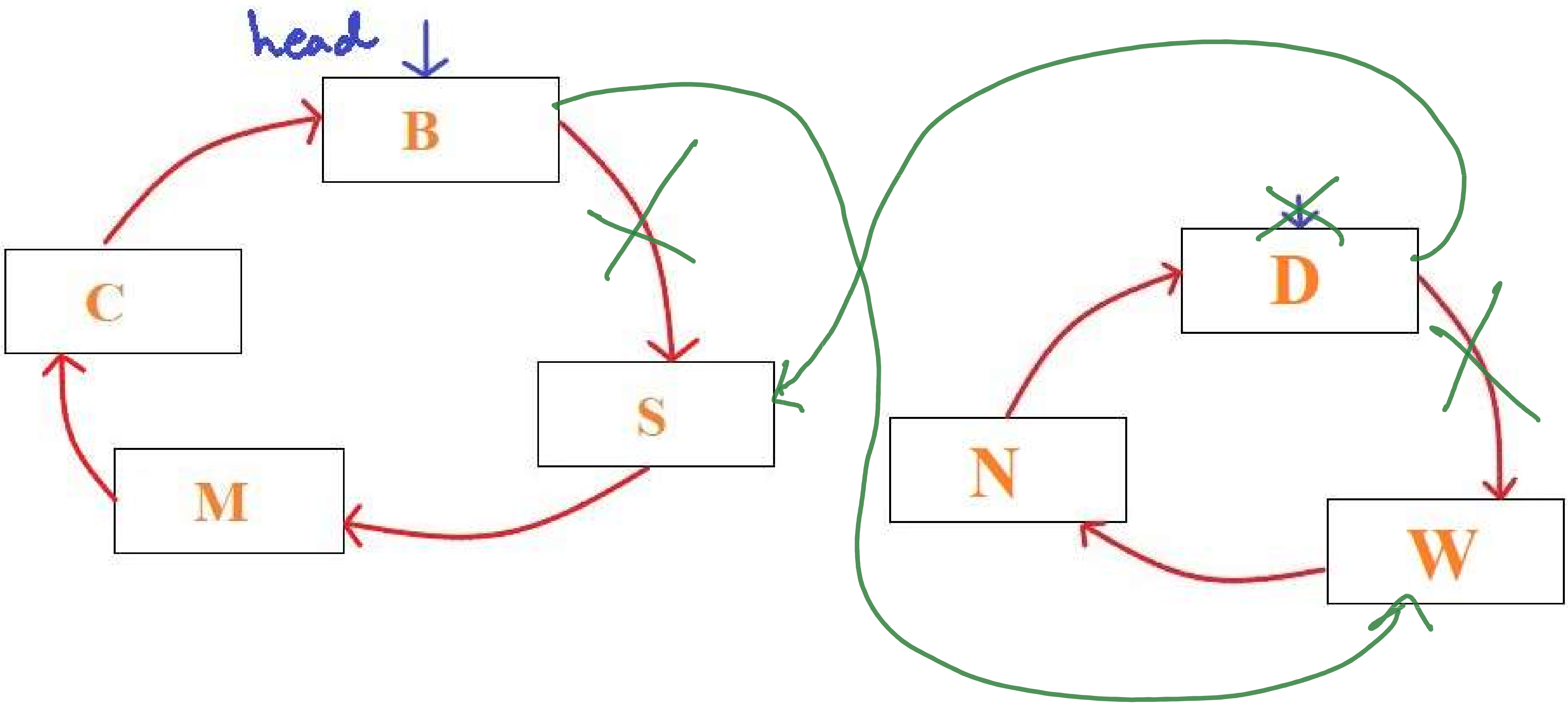
Union(DS, x, y):

- locate the head of each list by calling
  l_1 = FindSet(DS, x)
  l_2 = FindSet(DS, y);
- Exchange $l_1.next$ and $l_2.next$.





Union(DS, N, S):



MakeSet(DS, v): $\Theta(1)$

- create a new linked list with a node x storing element v;
- set $x.next = x$. r

FindSet(DS, x): $\Theta(L)$ where L is the length of the list containing x

- follow the links until reaching the head node r;
- return r.

Union(DS, x, y): $\Theta(L_1 + L_2)$

- locate the head of each list by calling
  $l_1 =$ FindSet(DS, x) $\quad l_2 =$ FindSet(DS, y);
- Exchange $l_1.next$ and $l_2.next$.

Consider a **bad** sequence where you have $m/4$ MakeSet, then $m/4 - 1$ Union, then $m/2 + 1$ FindSet.

$$\frac{m}{4} + \left(\frac{m}{4} - 1\right) + \left(\frac{m}{2} + 1\right) = m$$

After $\frac{m}{4}$ MakeSet and $\left(\frac{m}{4} - 1\right)$ Union, there will be a list with size $\frac{m}{4}$

For each FindSet, run time is $\Theta\left(\frac{m}{4}\right)$ in the worst-case

$\Rightarrow$ For $\left(\frac{m}{2} + 1\right)$ FindSet, total run time is

$$\left(\frac{m}{2} + 1\right) \times \frac{m}{4} \in \Theta(m^2)$$

For $\frac{m}{4}$ MakeSet, total run time is $\Theta(m)$

For $(\frac{m}{4}-1)$ Union, total is

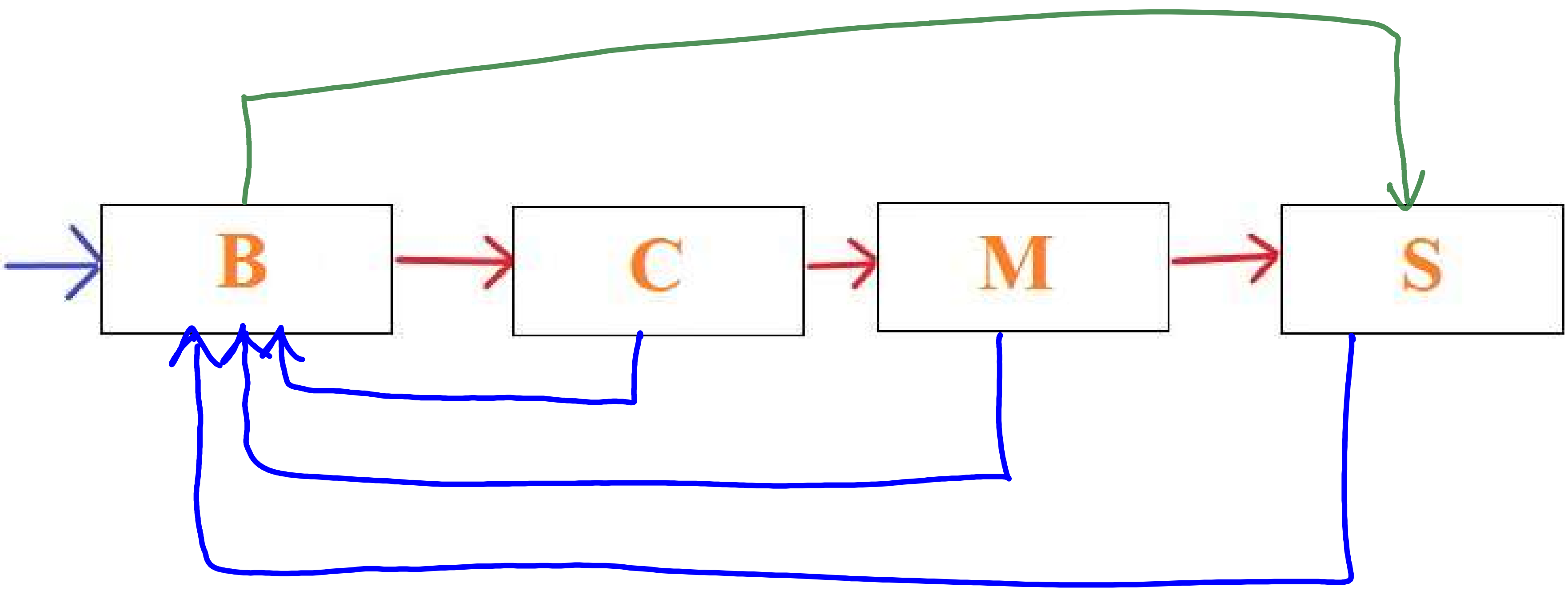
$$1+2+3+\cdots+(\frac{m}{4}-1) \in \Theta(m^2)$$

$$\Rightarrow T_m^{Sq} \in \Theta(m^2)$$

$$\frac{T_m^{Sq}}{m} \in \Theta(m)$$

Disjoint Sets: Linked Lists with Pointer to Head

- One linked list for each set.
- All nodes in a list, except the head node, have a pointer to the head of the list;
- Head of the linked list is the **representative** of the set.
- The head node has a pointer to the tail of the list.



Linked Lists with Pointer to Head: Implementation

MakeSet(DS, v):

- create a new linked list with a node x storing element v;
- $x.rep = x$.
- $x.tail = x$.

FindSet(DS, x):

- return $x.rep$.

Union(DS, x, y):

- append one list to the tail of the other;
- update the tail pointer;
- update the pointers to the head.

Worst-case Run Time:

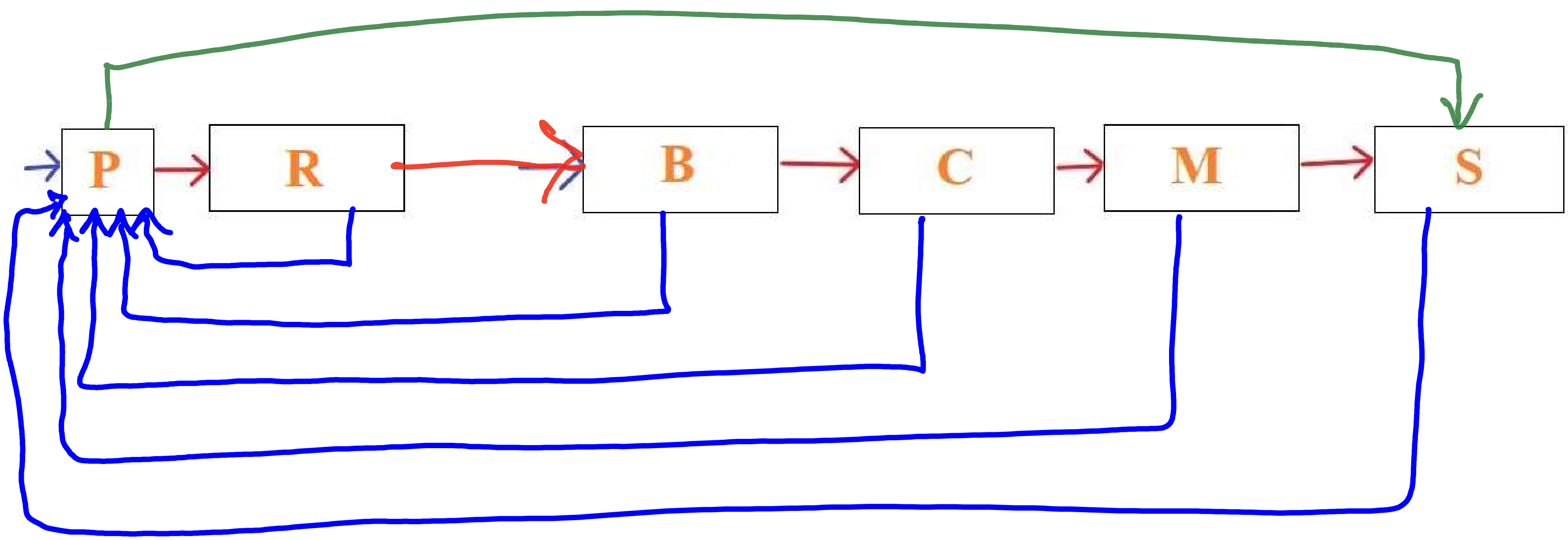
$\Theta(1)$

$\Theta(1)$

$\Theta(L_1 + L_2)$

n

Union(DS, R, S):



- Union is expensive, especially if appending a long list.
  A bad sequence includes many Union.

- Consider a **bad** sequence where you have $(\frac{m}{2}+1)$ MakeSet, then $(\frac{m}{2}-1)$ Union, always appending longer list to the end of single-element list.

$1+2+3+\cdots+(\frac{m}{2}-1)$ for updating head Pointers

$\in \Theta(m^2)$

$(\frac{m}{2}+1)$ MakeSet takes $\Theta(m)$ in total

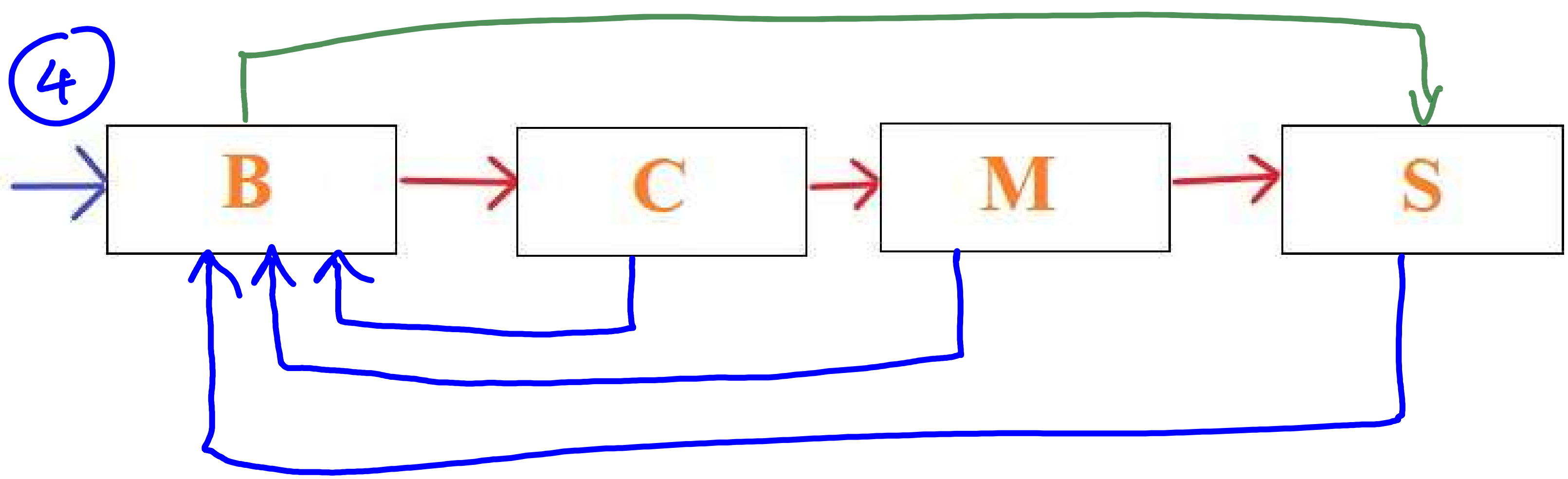
$$\Rightarrow T_m^{Sq} \in \Theta(m^2)$$

$$\frac{T_m^{Sq}}{m} \in \Theta(m)$$

Linked Lists with Pointer to Head and Union-by-Weight

- One Linked list for each set.
- All nodes in a list, except the head node, have a pointer to the head of the list;
- Head of the linked list is the **representative** of the set.
- The head node has a pointer to the tail of the list.
- The head node stores the size of each list.

B.size = 4



Linked Lists with Pointer to Head and Union-by-Weight

MakeSet(DS, v):

- create a new linked list with a node x storing element v;
- $x.rep = x$;
- $x.tail = x$;
- $x.size = 1$.

FindSet(DS, x):

- return $x.rep$.

Union(DS, x, y):

- append the shorter list to the longer list;
- update the tail pointer;
- update the size of the new list;
- update the pointers to the head.

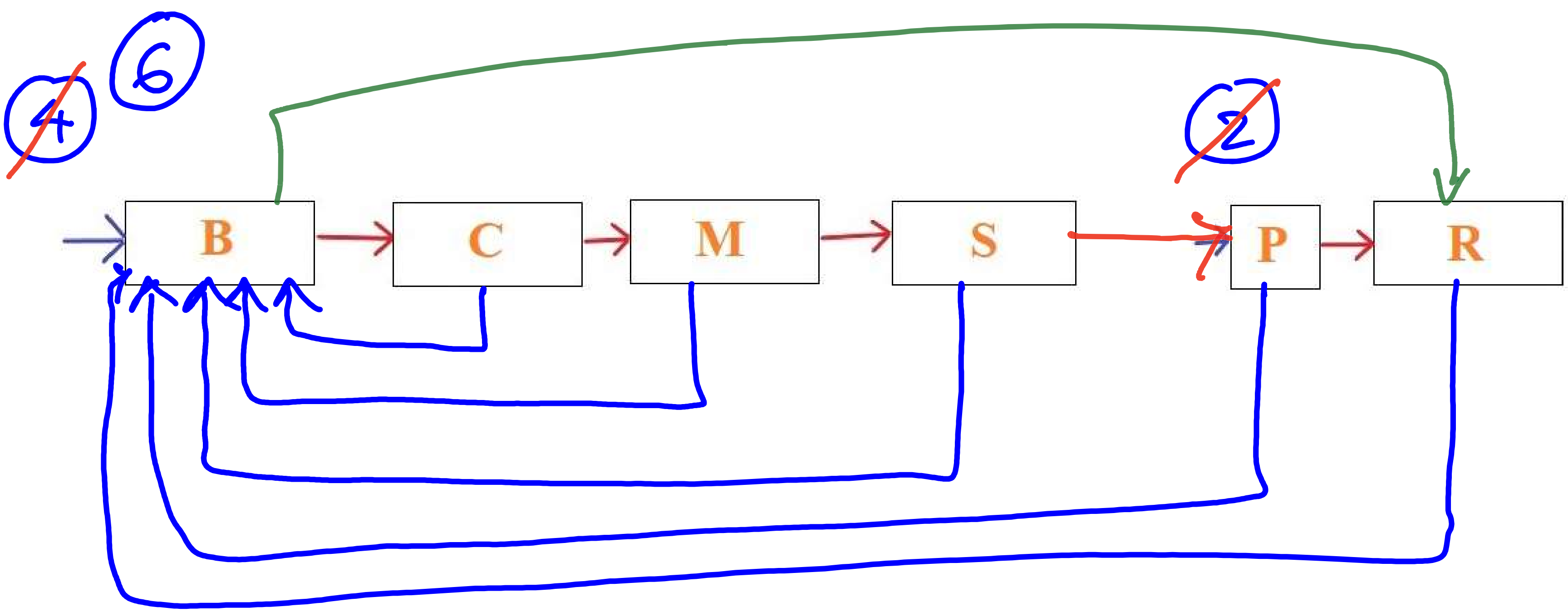
Worst-case Run Time:

$\Theta(1)$

$\Theta(1)$

$\Theta(L_1 + L_2)$

Union(DS, R, S):



Extra Pointer and Union-by-Weight: Amortized Analysis

- Consider a sequence of m operations.
- Let n be the number of MakeSet operations in the sequence. So there are **never more than** n elements in total.
- For some arbitrary element x, we want to prove an **upper bound** on the number of times that $x.rep$ can be updated.

- $x.rep$ gets updated only when the set that contains x is **unioned** with a set that is **not smaller**.
- So each time $x.rep$ is updated, the size of the resulting set must be at least **double** the size of the list containing x.
- That is, every time $x.rep$ is updated, set size **doubles**.
- There are only n elements in **total**, so we can double at most Lg n times
- So $x.rep$ cannot be updated more than Lg n times

Total number of .rep updates in at most n lg n

For the othe operations, total run time is $\Theta(m)$

$T_m^{Sq} \in \Theta(m + n \lg n)$

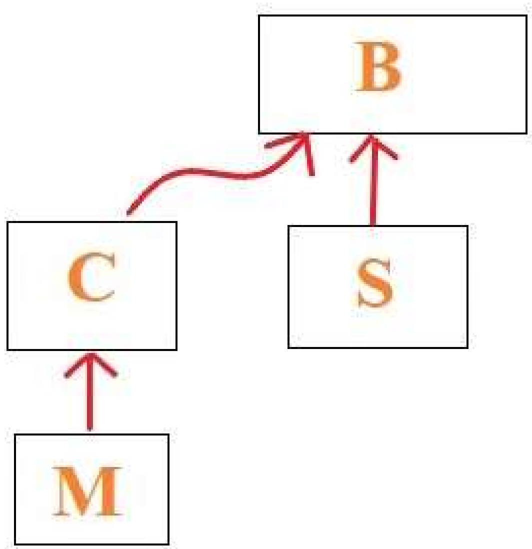
$\frac{T_m^{Sq}}{m} \in \Theta(\lg m)$

For a sequence of $\overbrace{(\frac{m}{2}+1)}^{n}$ MakeSet, and then $(\frac{m}{2}-1)$ Union: $T_m^{Sq} \in \Theta(m \lg m)$

Disjoint Sets: Trees

- One inverted tree for each set.
- Each element points to its **parent** only (or to itself if it's the root).
- The root of the tree is the **representative** of the set and points to itself.

Note: trees are **not** necessarily binary, number of children of a node can be arbitrary.



MakeSet(DS, v):

- create a tree with a node x storing element v;
- $x.p = x$.

FindSet(DS, x):

- trace up the parent pointer until the root r is reached;
- return r.

Union(DS, x, y):

- locate the head of each list by calling
  $l_1 =$ FindSet(DS, x)
  $l_2 =$ FindSet(DS, y);
- Let one tree's root point to the other tree's root: $l_1.p = l_2$.

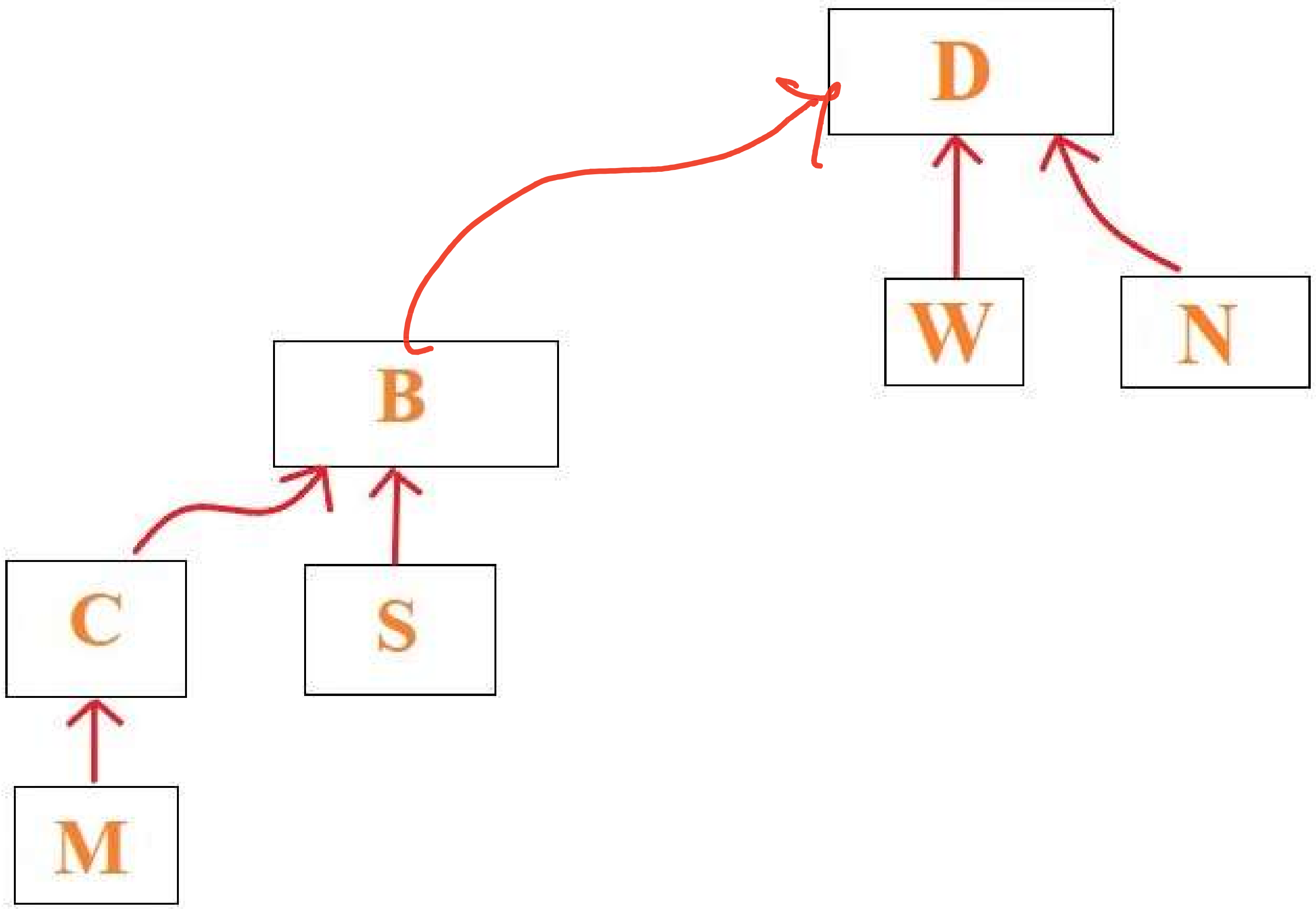
Worst-case Run Time:

$\Theta(1)$

$\Theta(\overset{\curvearrowright n}{h})$, where h is the height of the tree containing x

$\Theta(h_1 + h_2)$

Union(DS, N, S):



- FindSet, and so Union, are expensive.
  A **bad** sequence creates a tree that is just one long chain with $\frac{m}{4}$ elements.

- Consider a sequence where you have $\frac{m}{4}$ MakeSet, then $(\frac{m}{4} - 1)$ Union so that one long chain with $\frac{m}{4}$ elements is created.
  Then $(\frac{m}{2} + 1)$ FindSet.

$$\frac{m}{4}\left(\frac{m}{2}+1\right) \in \Theta(m^2)$$

$$\Rightarrow T_m^{Sq} \in \Theta(m^2)$$

$$\frac{T_m^{Sq}}{m} \in \Theta(m)$$

Intuition:

- FindSet takes $\Theta(h)$, where h is the height of the tree.
- Append **smaller** tree to larger one during Union to keep the unioned tree's height small.

- One inverted tree for each set.
- Each element points to its **parent** only.
- The root of the tree is the **representative** of the set and points to itself.
- The **root** stores the rank of the tree.
- Rank of a tree: the **height** of the tree (for now).

Trees with Union-by-Rank: Implementation

MakeSet(DS, v):

- create a tree with a node x storing element v;
- $x.rank = 0$;
- $x.p = x$.

FindSet(DS, x):

- trace up the parent pointer until the root r is reached;
- return r.

Union(DS, x, y):

- locate the head of each list by calling $l_1 =$ FindSet(DS, x) $l_2 =$ FindSet(DS, y);
- let the root with lower rank point to the root with higher rank;
- if the two roots have the same rank, choose either root as the new root and increment the rank of the new root by one.

Worst-case Run Time:

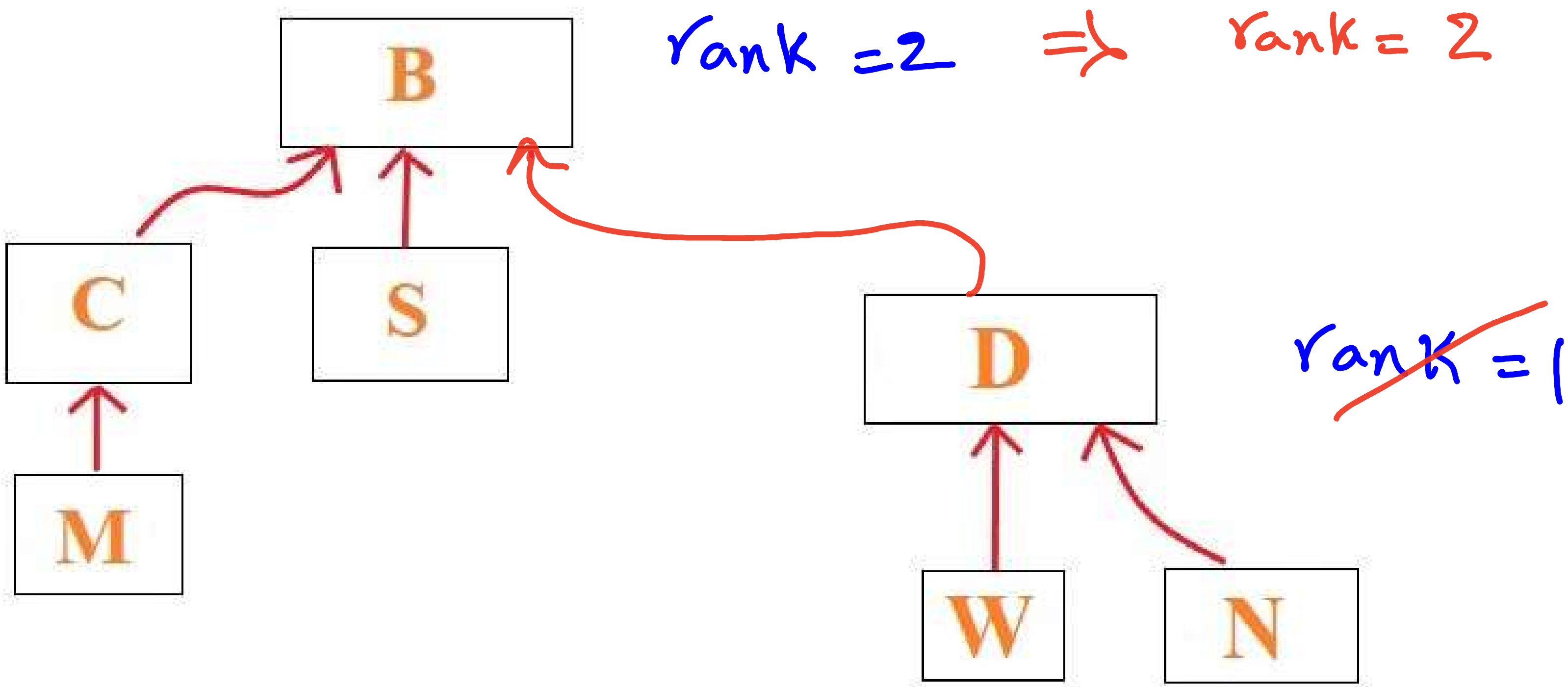
$\Theta(1)$

$\Theta(h)$

$\lg n$

$\Theta(h_1 + h_2)$

Union(DS, N, S):



Let T be a tree generated by a series of MakeSet and Union operations using the union-by-rank heuristic. Let r be the rank of T, and n be the number of nodes in T.
Then $2^r \leq n$ $\Rightarrow$ $r \leq \lg_2 n$

Proof: For MakeSet, the new tree which is created has one node, and its rank is 0.

$r = 0$ and $n = 1$ $\Rightarrow$ $2^r = 2^0 \leq n$

For Union, induction over structure of T.
Suppose we take the union of two trees T_1 and T_2, with sizes n_1 and n_2 and ranks r_1 and r_2 respectively.

$2^{r_1} \leq n_1$ and $2^{r_2} \leq n_2$ [IH]

Since T is the union of T_1 and T_2,

$n = n_1 + n_2$ (i)

Case 1: $r_1 > r_2$

Then $r = r_1$ since heigth of T_1 is larger than T_2 and root of T_1 is chosen as the root of T. So

$$2^r = 2^{r_1}$$

$$\leq n_1 \quad \text{by IH}$$

$$< n \quad \text{since } n = n_1 + n_2 \text{ and } n_2 > 0$$

Case 2: $r_2 > r_1$

Similar to Case 1

Case 3: $r_1 = r_2$

Assume root of T_1 is chosen as the root of T. Then $r = r_1 + 1$ (2)

By IH, $2^{r_1} \leq n_1$

$2^{r_2} \leq n_2$

$$\Rightarrow 2^{r_1} + 2^{r_2} \leq n_1 + n_2$$

$$\Rightarrow 2^{r_1} + 2^{r_1} \leq n$$ by (1) and assumption

$$\Rightarrow 2^{r_1+1} \leq n$$ since $r_1 \geq 1$

$$\Rightarrow 2^r \leq n \quad \text{by } ②$$

$$\Rightarrow r \leq \lg_2 n$$

Consider a sequence of operations. Then each operation in the sequence takes at most $\lg n$

$$\Rightarrow T_m^{Sq} \in O(m \lg n)$$

$$\frac{T_m^{Sq}}{m} \in O(\lg m)$$

For a sequence of $\frac{m}{4}$ MakeSet, then $(\frac{m}{4} - 1)$ Union and $(\frac{m}{2} + 1)$ FindSet: $T_m^{Sq} \in O(m \lg m)$

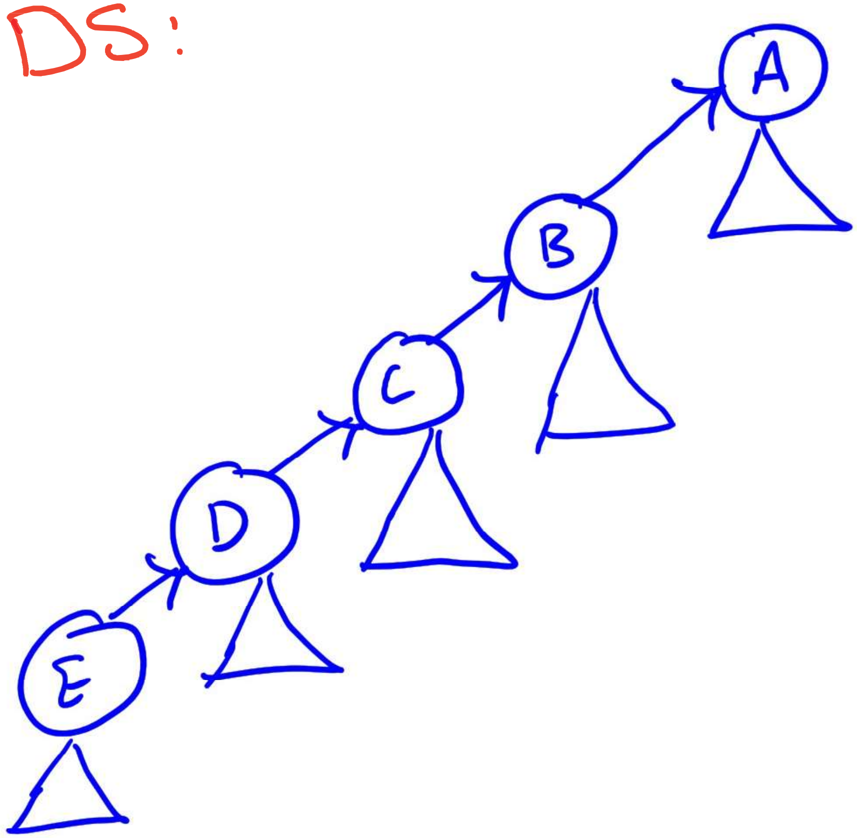
n

Disjoint Sets: Trees with Path-Compression

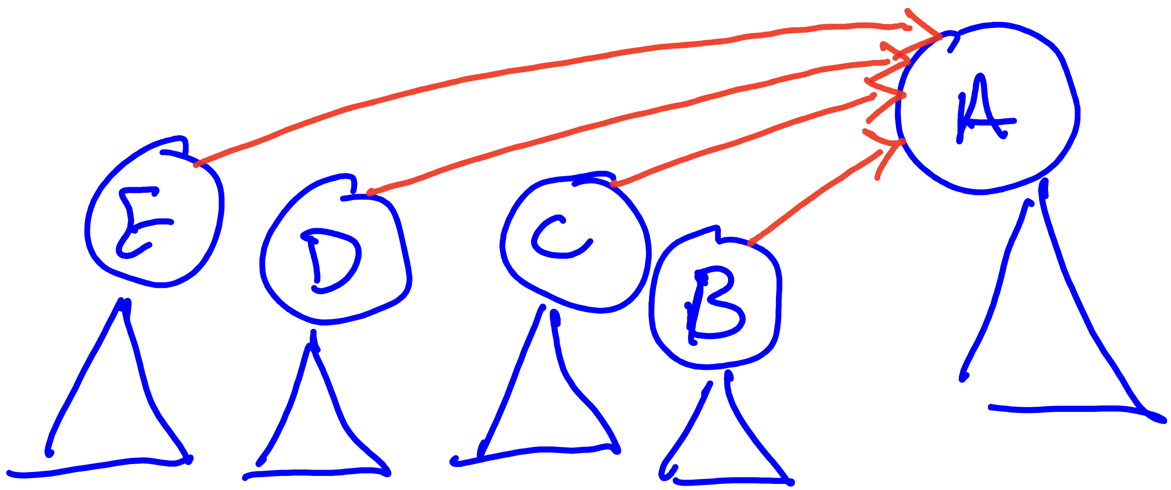
Idea:

- When calling FindSet(DS, x) for some x, keep track of nodes visited on path from x to the root (use a stack or queue, or write FindSet recursively).
- Once the root is found, make each visited node to **point directly** to the root.
- **Benefit:** Can speed up future operations considerably.

DS:



FindSet(DS, E)


E
D
C
B
A

MakeSet(DS, v):

- create a tree with a node x storing element v;
- $x.p = x$.

FindSet(DS, x):

- trace up the parent pointer until the root r is reached;
- once the root is found, make each visited node to point directly to the root.
- return r.

Union(DS, x, y):

- locate the head of each list by calling
  $l_1 =$ FindSet(DS, x)
  $l_2 =$ FindSet(DS, y);
- Let one tree's root point to the other tree's root:
  $l_1.p = l_2$.

Worst-case Run Time:

$\Theta(1)$

$\Theta(h)$

$\Theta(h_1 + h_2)$

For a sequence of n MakeSet (and hence at most $n-1$ Union) and f FindSet:

$$T_m^{sq} \in \Theta\Big(n + f \times (1 + \log_{2+\frac{f}{n}} n)\Big)$$

Disjoint Sets: Trees with Union-by-Rank and Path-Compression

Idea:

- Path compression happens in FindSet.
- Union-by-rank happens in Union (outside FindSet).
- So we can use them **both**!

- **Small Issue:** Path compression does **NOT** maintain height info because updating ranks would be **too expensive**.

- **Solution:** A node's rank is an **upper-bound** on its height.
  It can be proved that comparing ranks (and not the heights) does **not** affect the efficiency of the algorithm considerably.

MakeSet(DS, v):

- create a tree with a node x storing element v;
- $x.rank = 0$;
- $x.p = x$.

FindSet(DS, x):

- trace up the parent pointer until the root r is reached;
- once the root r is found, make each visited node to point directly to r.
- return r.

Union(DS, x, y):

- locate the head of each list by calling
  $l_1 =$ FindSet(DS, x)
  $l_2 =$ FindSet(DS, y);
- let the root with lower rank point to the root with higher rank;
- if the two roots have the same rank, choose either root as the new root and increment the rank of the new root by one.

(Complete implementation is in Section 21.3 of CLRS)

For a sequence of m operations with n MakeSet (so at most $n-1$ Union):

$$T_m^{sq} \in \mathcal{O}(m \times \underline{\alpha(n)})$$

where $\alpha(n)$ is the **inverse Ackerman function**, which grows really, really, really slowly. So we can basically treat it as **constant**. So

$$T_m^{sq} \in \mathcal{O}(m) \quad \Rightarrow \frac{\overline{T}_m^{sq}}{m} \in O(1)$$

$$\alpha(10^8) = 4$$

$$\alpha(2^{65536}) = 5$$

After Lecture

- After-lecture Readings: Chapter 8 of the course notes.
- Optional Readings: CLRS Sections 21.1, 21.2, 21.3.
- Problems in Chapter 8 of the course notes.
- Problems 21.2-2. in CLRS.