NUS
National University of Singapore

School of Computing

Tutorial 6: Union-Find

October 3, 2022

Gu Zhenhao

* *Partly adopted from tutorial slides by [Wang Zhi Jian](#).*

Disjoint Set ADT

Why do we need the Disjoint Set ADT?

Disjoint Set ADT

- **Purpose**: quickly find the category of a key.
- **Operations**:
  1. `findSet(i)`: Find which set a key belongs to.
  2. `isSameSet(i, j)`: Check if two keys are in same set.
  3. `unionSet(i, j)`: merge sets containing `i` and `j`.

Suppose we already know the reference to the keys.

Implementation of Disjoint Set

- Using hash table: store `<key, set>` pair.

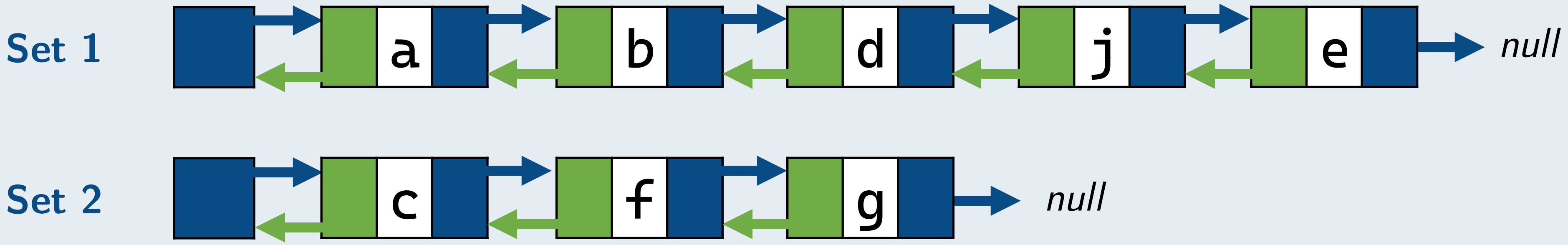
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
<a,1>	<b,1>	<c,2>	<d,1>	<e,1>	<f,2>	<g,2>			<j,1>

- Finding a key is fast, but merging two sets would be slow.

`findSet`	`isSameSet`	`unionSet`
$O(1)$	$O(1)$	$O(n)$

Implementation of Disjoint Set

- Using linked list: for **`findSet`**, trace back to the head.

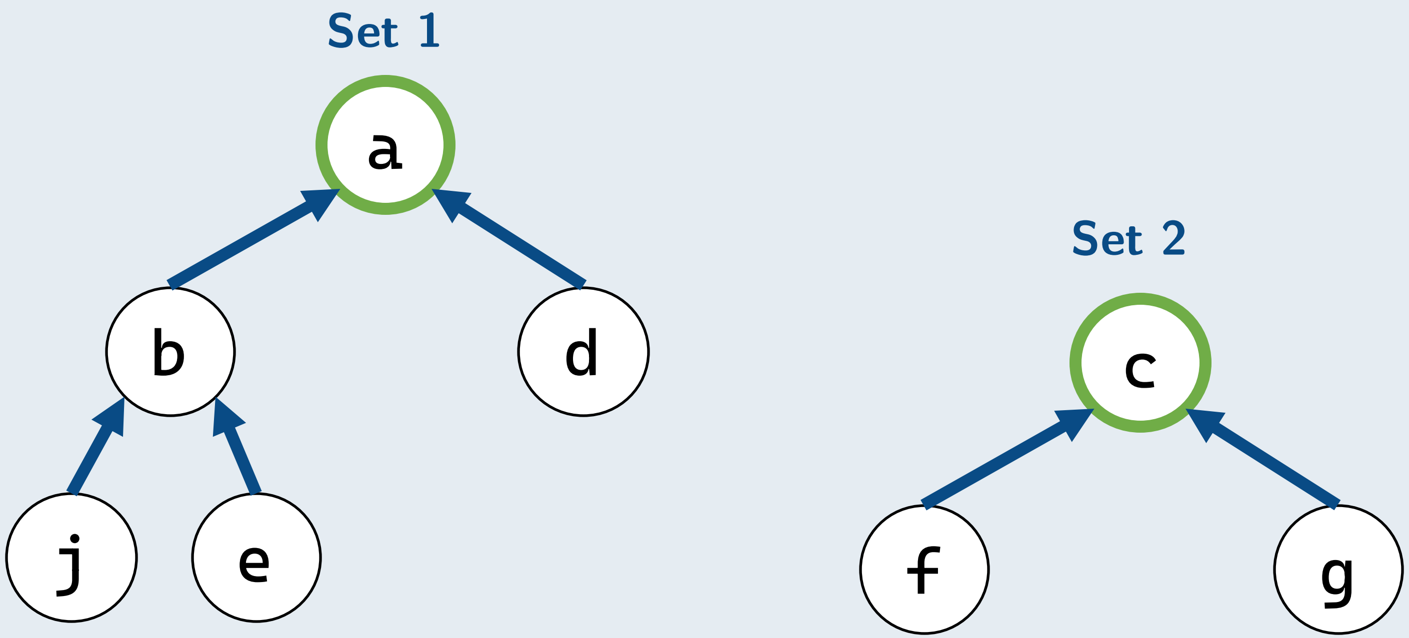


- Merging sets would be faster, but finding set a key belongs to is slow.

`findSet`	`isSameSet`	`unionSet`
$O(n)$	$O(n)$	`findSet` $+ O(1)$

Implementation of Disjoint Set

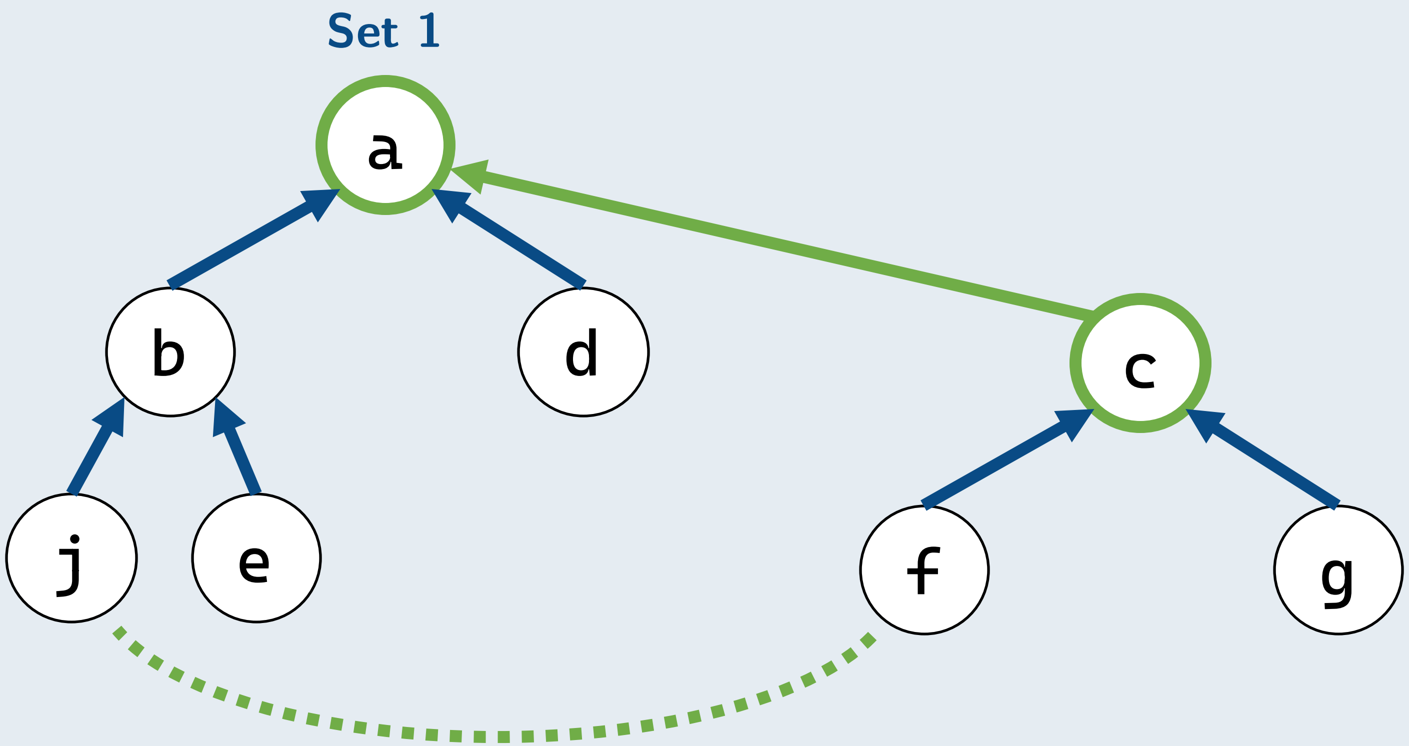
- Using tree:



- for `findSet`, find the root of the tree.

Implementation of Disjoint Set

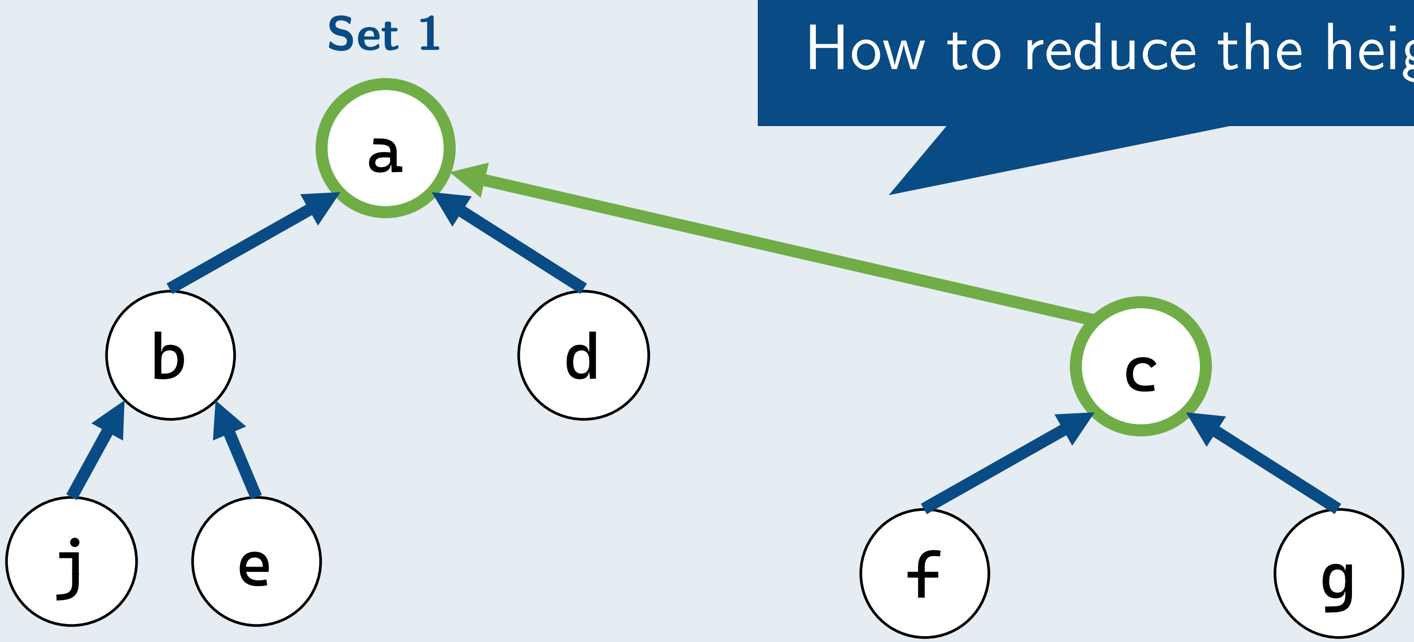
- Using tree:



- for **`unionSet`**, simply let both trees share the same root.

Implementation of Disjoint Set

- Using tree:

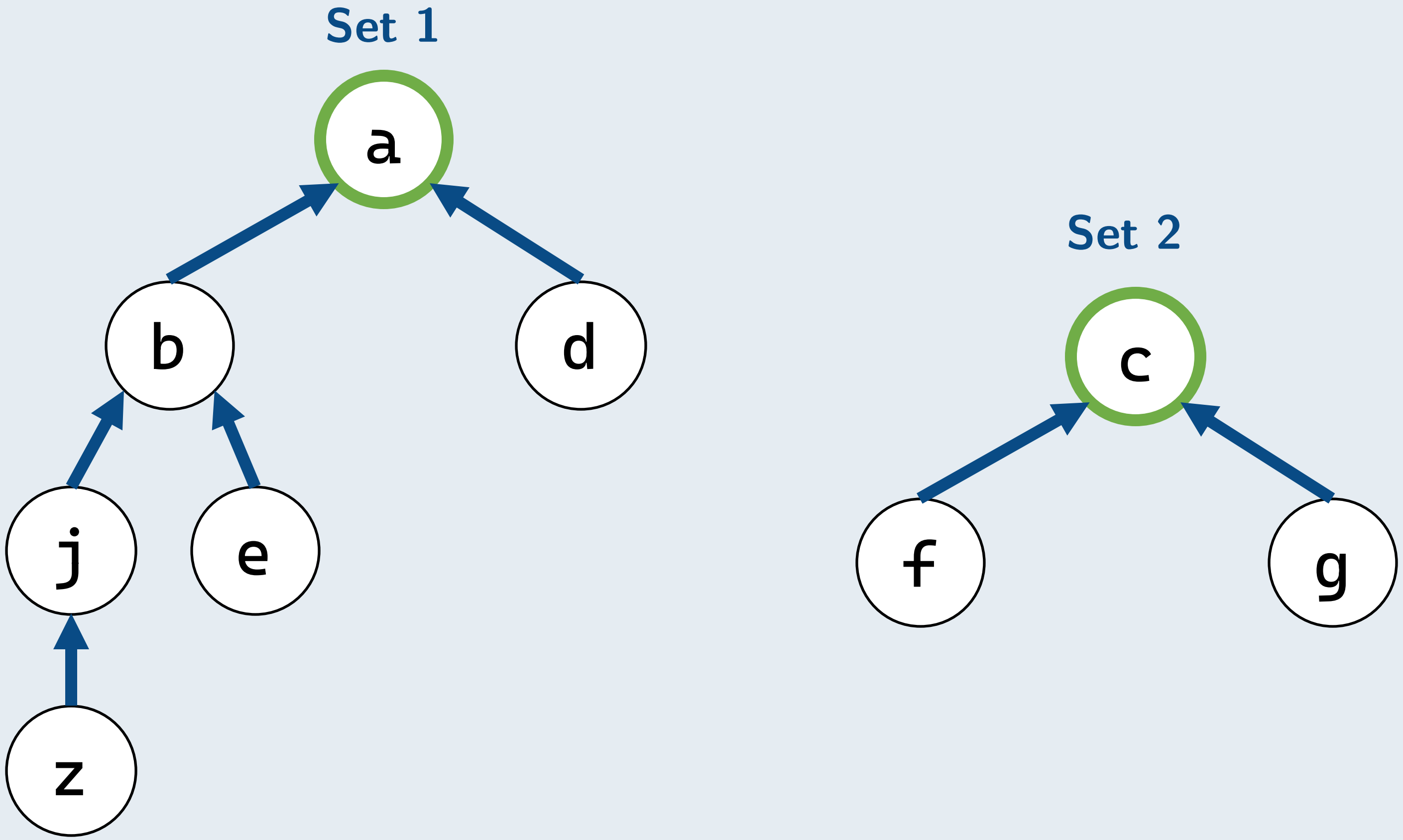


How to reduce the height of the trees?

findSet	isSameSet	unionSet
O(height of tree)	O(height of tree)	findSet $+ O(1)$

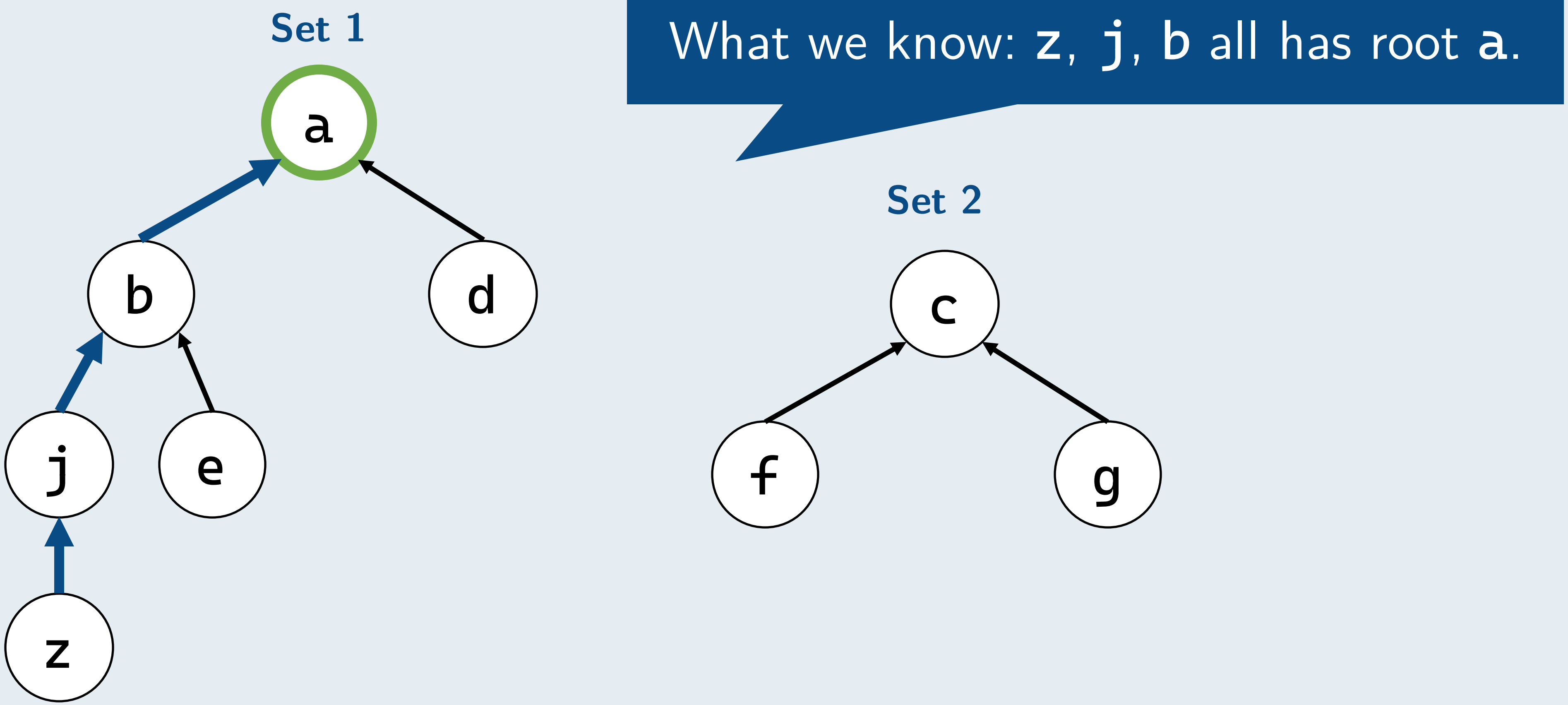
Implementation of Disjoint Set

- Example: `findSet(z)`



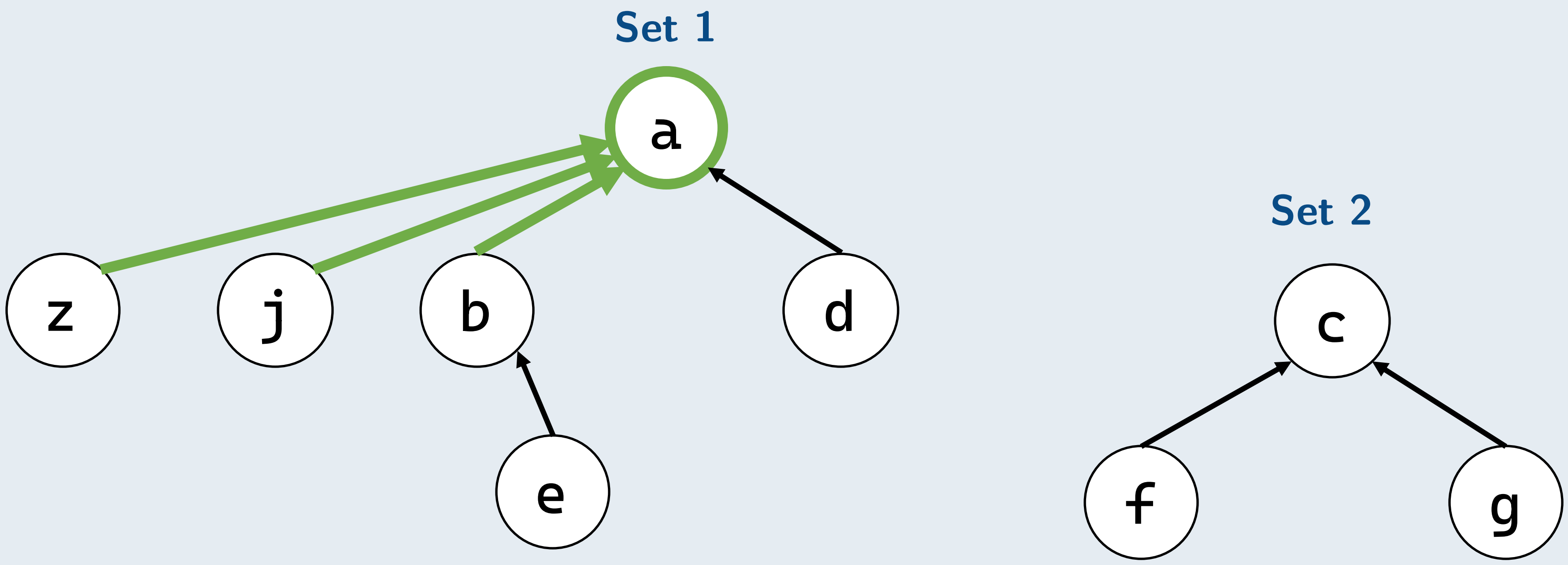
Implementation of Disjoint Set

- Example: `findSet(z)`



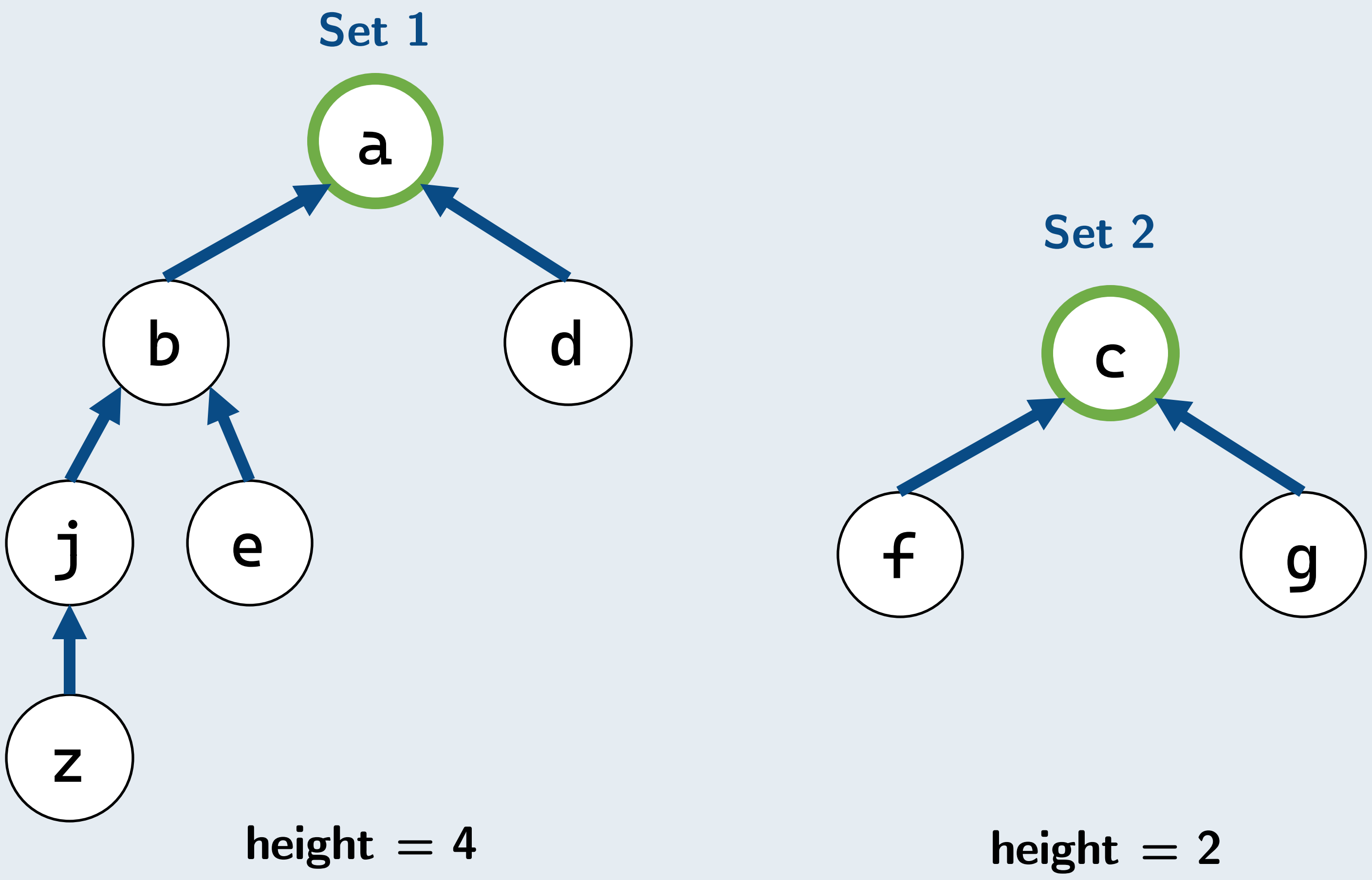
Implementation of Disjoint Set

- **Idea**: let those keys point directly to the root **a**!



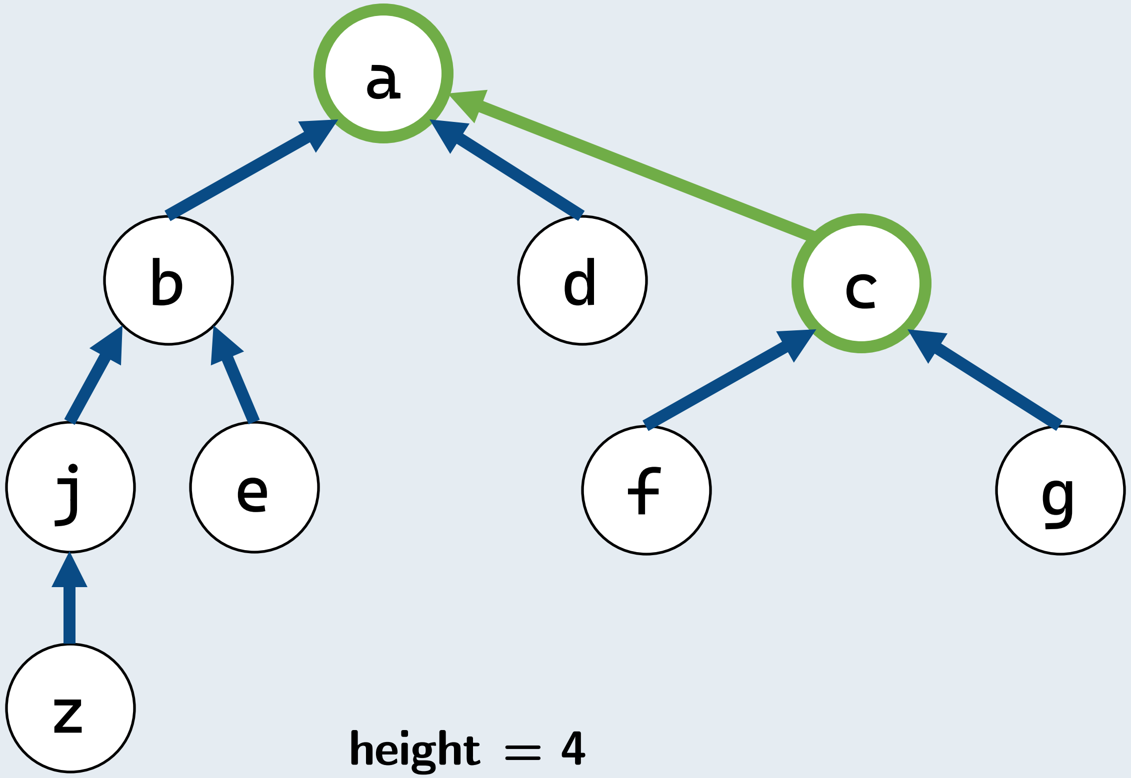
Implementation of Disjoint Set

- Example: `unionSet(b, f)`

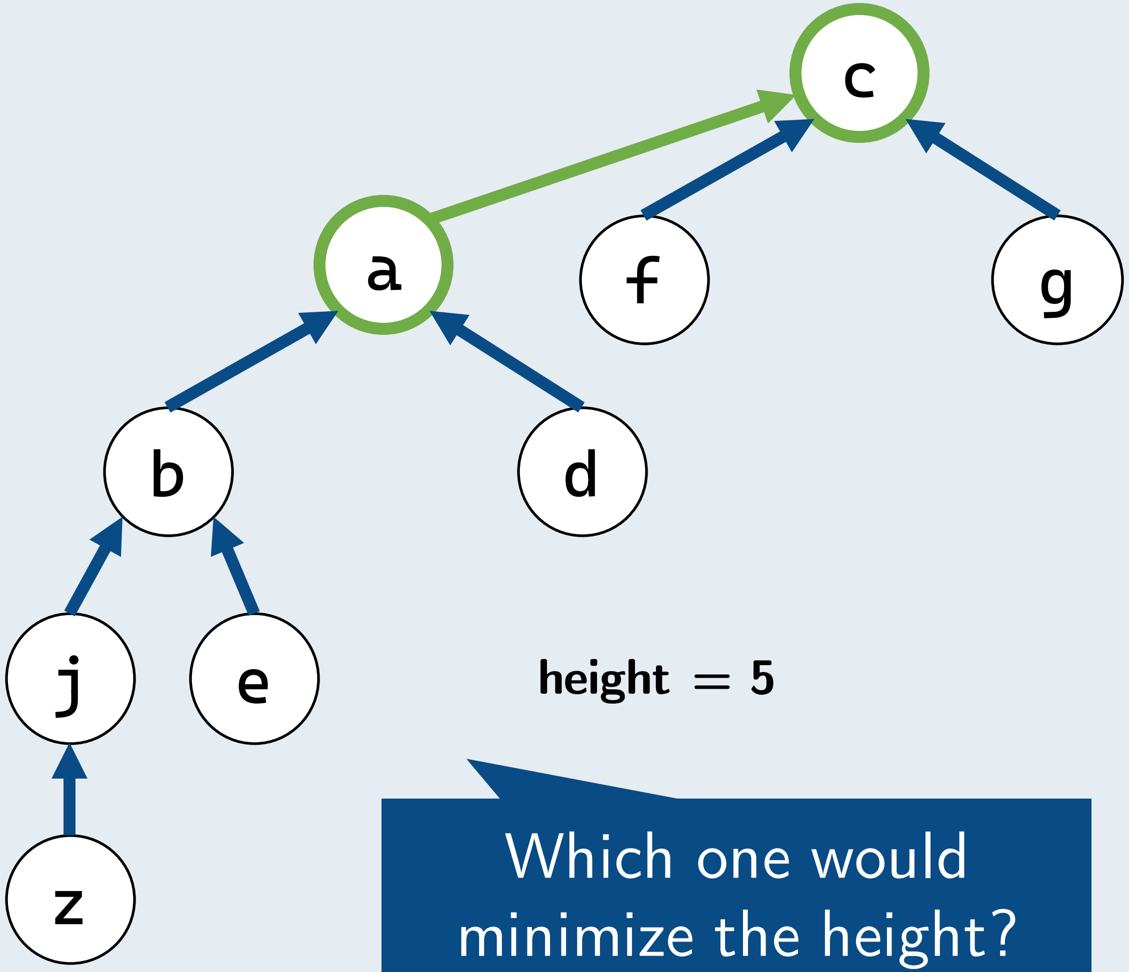


Implementation of Disjoint Set

- Example: `unionSet(b, f)`



height $= 4$

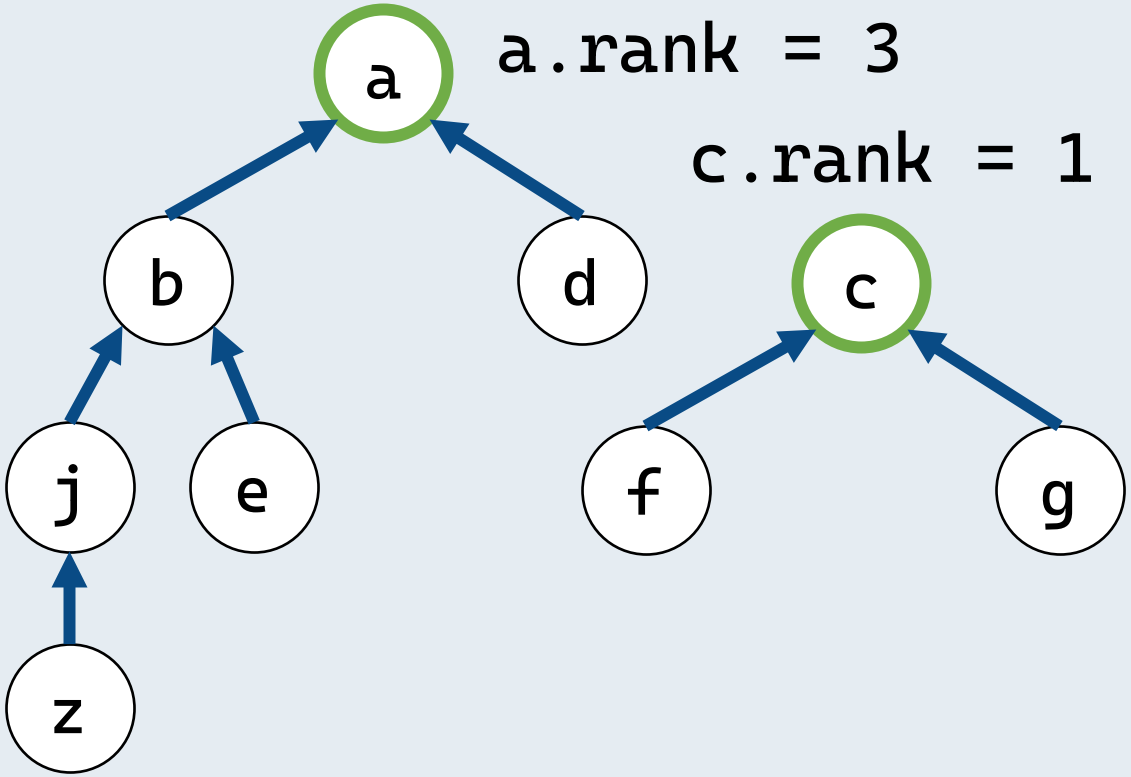


height $= 5$

Which one would minimize the height?

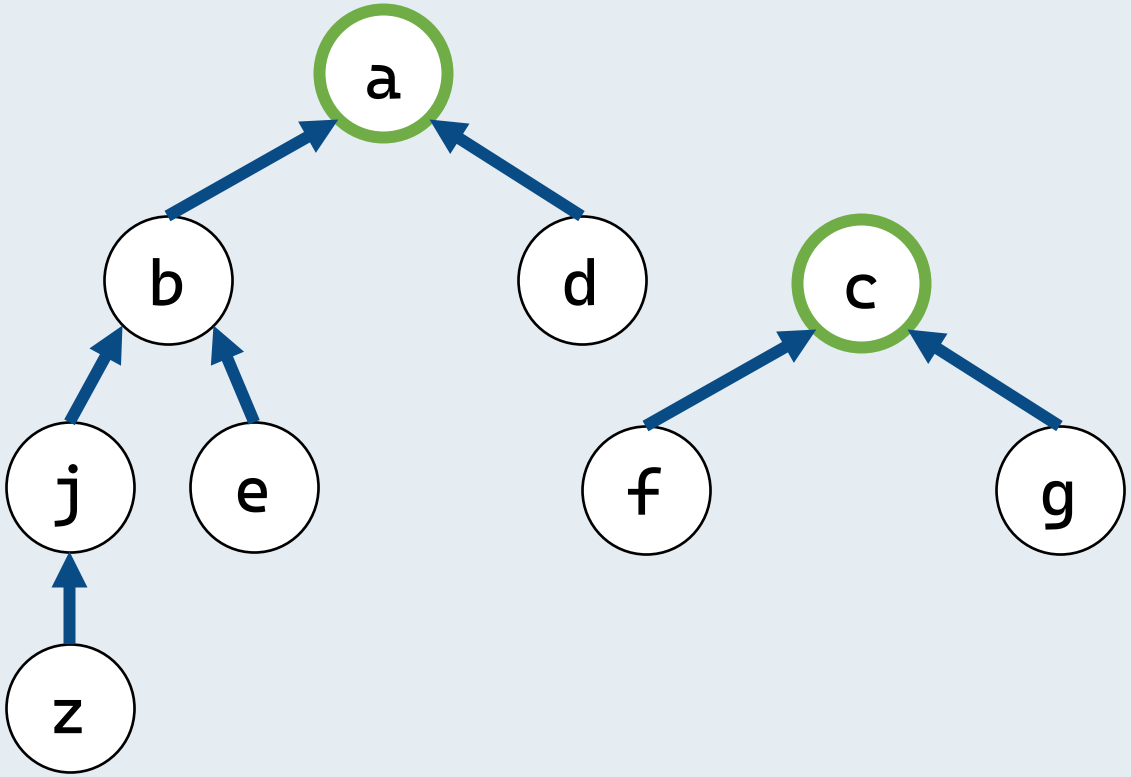
Implementation of Disjoint Set

- Example: `unionSet(b, f)`



- **Observation**: Let the shorter tree point to the taller tree would minimize total height.
- **Idea**:
    1. keep a `rank` parameter for the root, denoting **upper bound of the height of tree** (excluding root).
    2. When merging trees, let the root with smaller `rank` be the child.

Analysis of Union-Find DS *



Amortized analysis:

- n keys in total,
- m `findSet`, `isSameSet`, `unionSet` operations will run in $O\big(m\alpha(n)\big)$ time.
- Each operation runs in average $O\big(\alpha(n)\big)$ time.

** See more detail about inverse Ackerman's function $\alpha(n)$ in the appendix.*

Problem 1

*Given a UFDS initialised with n disjoint sets, what is the maximum possible rank h that can be obtained from calling **any combination** of* `unionSet(i, j)` *and/or* `findSet(i)` *operations? Assume that both the path-compression and union-by-rank heuristics are used.*

Questions:

1. What operation will increase the rank?
2. How to maximize the rank using the operation?

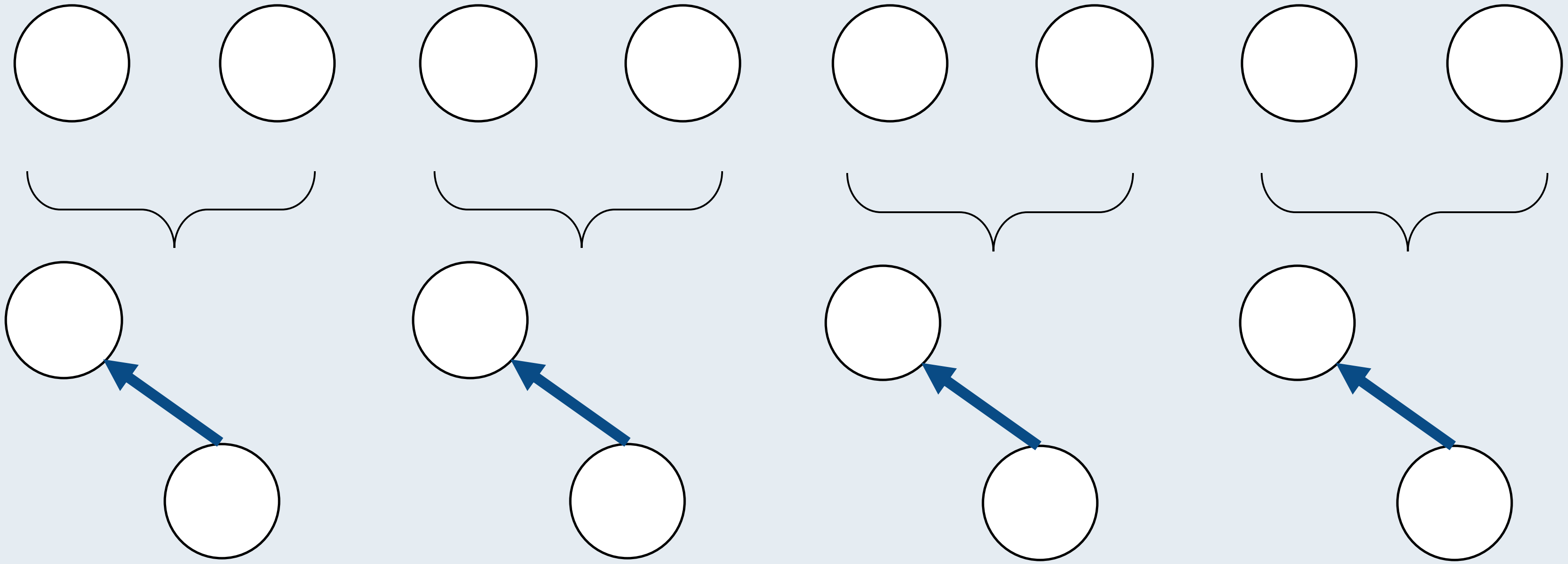
Problem 1

Given a UFDS initialised with n disjoint sets, what is the maximum possible rank h that can be obtained from calling ***any combination*** *of* `unionSet(i, j)` *and/or* `findSet(i)` *operations? Assume that both the path-compression and union-by-rank heuristics are used.*

Observation: only <u>merging two sets with equal rank</u> will increase the rank of a root.

Strategy: merge sets of equal rank as often as possible.

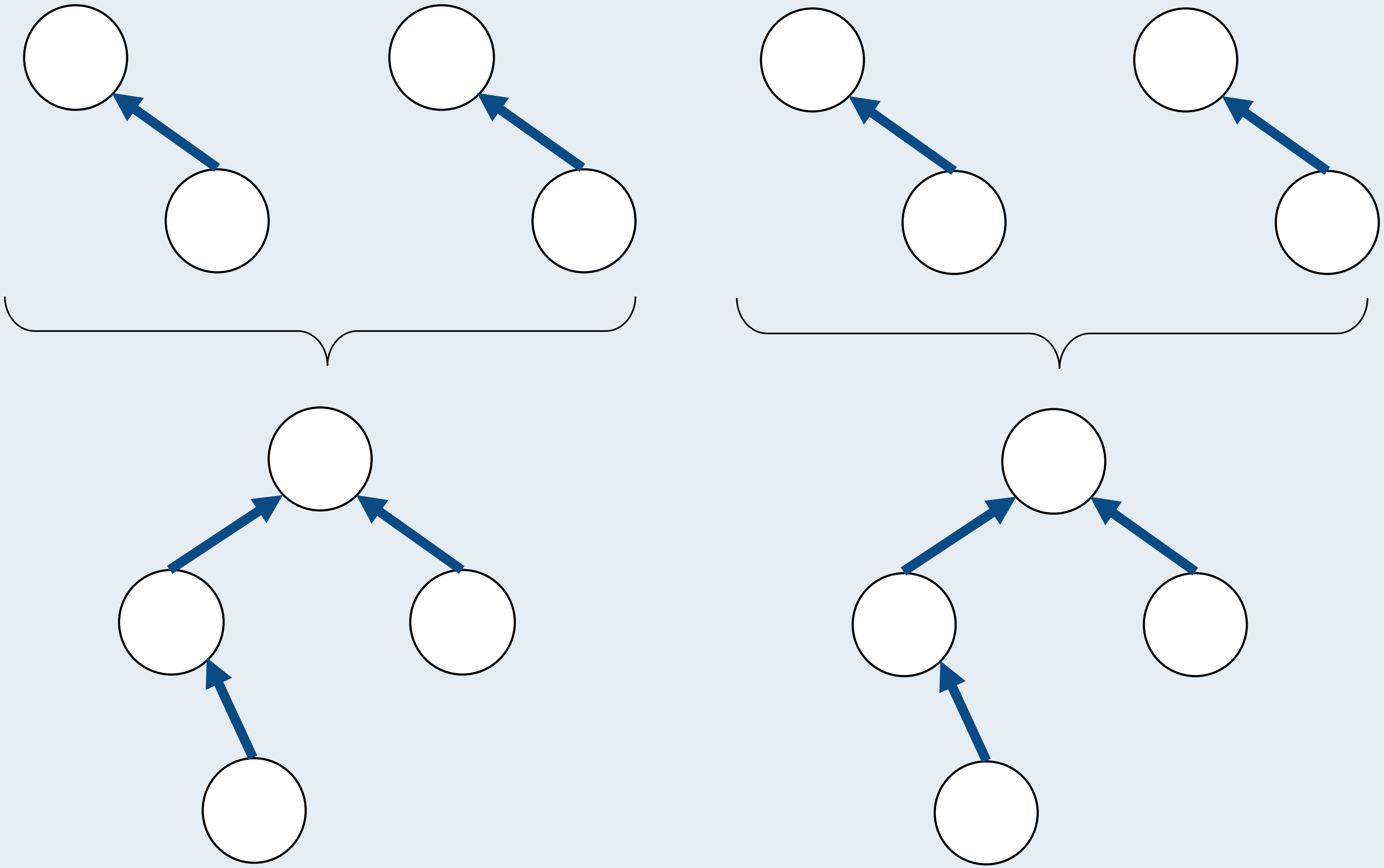
Problem 1



n sets of rank 0

$\left\lfloor \frac{n}{2} \right\rfloor$ sets of rank 1

Problem 1



$\left\lfloor \frac{n}{2} \right\rfloor$ sets of rank 1

$\left\lfloor \frac{n}{4} \right\rfloor$ sets of rank 2

$\vdots$

1 set of rank $\lfloor \log_2 n \rfloor$.

Application of UFDS

How to efficiently use UFDS for grouping?

Problem 2.a

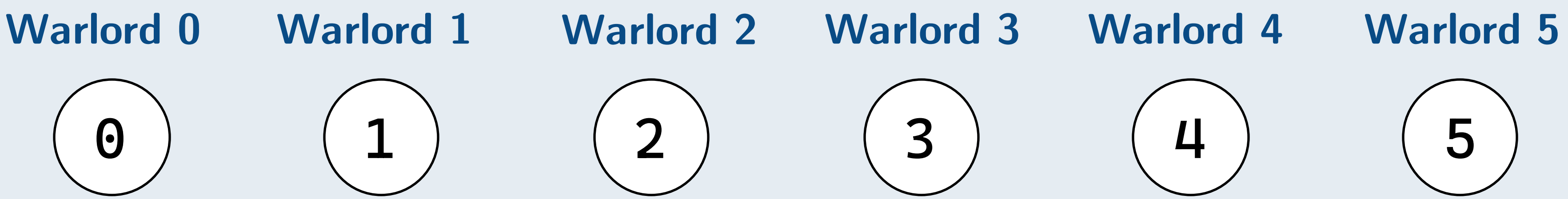
- n warlords, each controlling a world.
- A conquered warlord will have all the worlds under him/her added to the victorious warlord.

These are just `unionSet` and `findSet` operations!

Goals:

- Add worlds of a conquered warlord to the victorious warlord.
- Query whether a world is in a warlord's control.

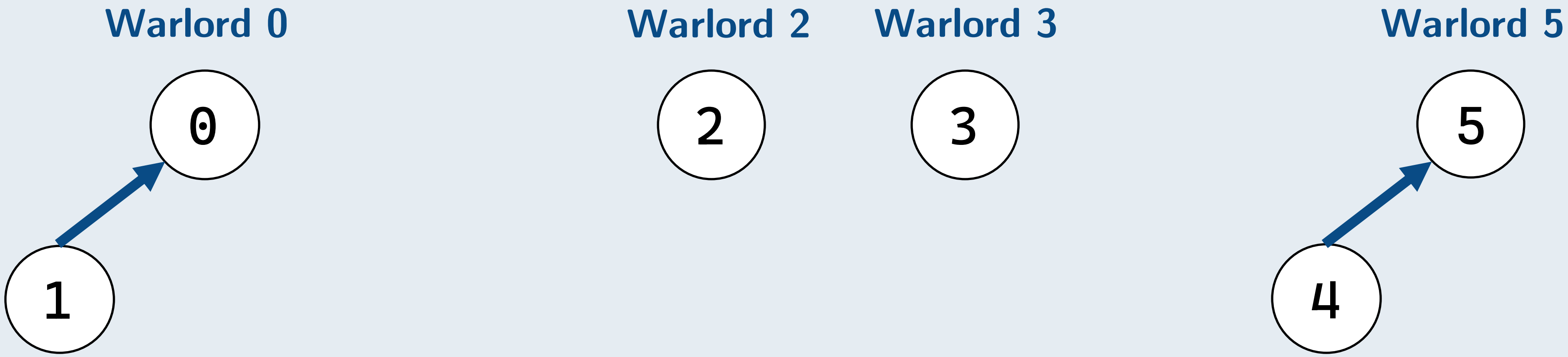
Problem 2.a



Example:

1. Warlord 0 conquers warlord 1.
2. Warlord 5 conquers warlord 4.

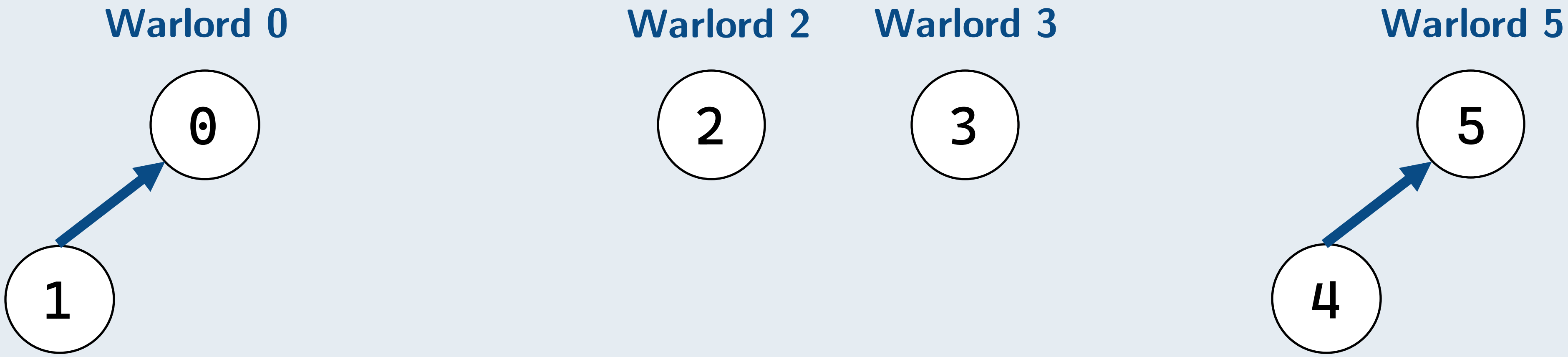
Problem 2.a



Example:

1. Warlord 0 conquers warlord 1.
2. Warlord 5 conquers warlord 4.

Problem 2.a



Another Example: Warlord 2 conquers warlord 0.

Problem 2.a

Another Example: Warlord 2 conquers warlord 0.

Problem: the root doesn't indicate the victorious warlord!

Problem 2.a

Possible Strategies:

- Swap the root with the id of victorious warlord.
- Keep an additional array of size n, storing the root node for each warlord. (-1 if the warlord is conquered).

Problem 2.b

- n warlords, each controlling a world.
- A conquered warlord will have all the worlds under him/her added to the victorious warlord.

Goal:

- Check if a warlord has conquered all n worlds.

Problem 2.b

Trivial solution:

- Use `findSet` on all worlds and check if it belongs to the warlord.
- This takes $O(n)$ time.

Problem 2.b

Warlord	0	1	2	3	4	5
No. of worlds	0	0	3	1	0	2

Better solution:

- Keep an array W of length n, $W[i]$ is the number of worlds warlord i controls.
- Update the value in W whenever we do `unionSet`.
- Checking done in $O(1)$ time.

Problem 3

We have h numbers (coming in one by one) between 1 and n.

- **Goal**: whenever a number arrives, find the largest number in an unbroken sequence starting from 1.
- **Example**: the h numbers are

`1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

The longest unbroken sequence starting from 1 is 1, 2, 3, 4, 5, 6, **7**.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Trivial Solution:

- Keep a sorted arrays of the numbers,
- Starting from 1, check if 2, 3, 4,..., m are in the array.
- Each insert will cost $O(h)$ time and each query $O(m)$ time.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Better Idea:

- Keep a reference to the largest number x in the unbroken sequence.
- When $x + 1$ arrives, update the reference.

Problem 3

```
1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116
```

Better Idea:

- Keep a reference to the largest number x in the unbroken sequence.
- When $x + 1$ arrives, update the reference.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Can we make this updating faster?

Better Idea:

- Keep a reference to the largest number x in the unbroken sequence.
- When $x + 1$ arrives, update the reference.
- Each query will be $O(1)$ but inserting costs $O(m)$.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Even Better Idea:

- Group numbers that are close together!

Problem 3

`1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Even Better Idea:

- Group numbers that are close together!
- When a number arrives, check
  - if one of its adjacent numbers presents, insert it into the same group.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Even Better Idea:

- Group numbers that are close together!
- When a number arrives, check
  - if one of its adjacent numbers presents, insert it into the same group.
  - if both of its adjacent numbers present, merge the two groups!
  - Otherwise, create a new group with it.

Problem 3

`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116`

Even Better Idea:

- Group numbers that are close together!
- When a number arrives, check
  - if one of its adjacent numbers presents, insert it into the same group.
  - if both of its adjacent numbers present, merge the two groups!
  - Otherwise, create a new group with it.

Can be efficiently done using `findSet` and `unionSet` in UFDS!

Problem 4

- We have n seats and m children $(m < n)$. Each child i want to sit at seat s_i.
  - If s_i is empty, put child i there.
  - If not, check $s_i + 1$, $s_i + 2$, ..., $n - 1$, 0, 1 ,..., until we find an empty seat.

Similar to linear probing!

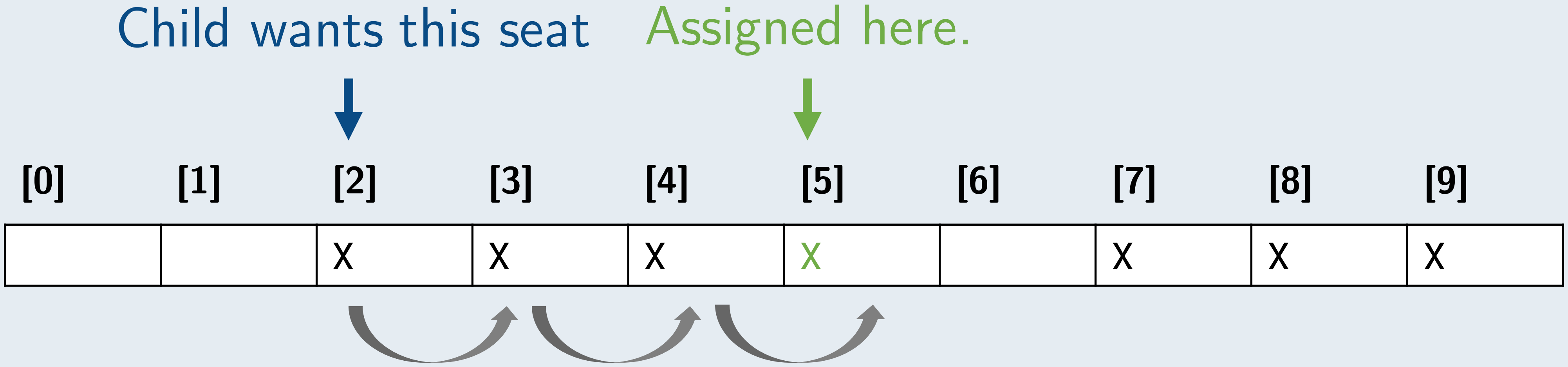
* *For simplicity, suppose seats are numbered 0, 1, ..., $n - 1$.*

Problem 4

Child wants this seat

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		X	X	X			X	X	X

Problem 4



Can we skip the cluster in one go?

Trivial answer:

- Just use linear probing.
- Might take $O(m)$ if there is a large cluster!

Problem 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		X	X	X	X		X	X	X

`max = 5`

`max = 9`

Idea:

- Similar to problem 3.
- Keep the clusters in disjoint sets, and record the maximum number.

Problem 4

Child wants this seat

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		X	X	X	X		X	X	X

`max = 5`

`max = 9`

Idea: When assigning a new child i, check:

- If s_i is already in a group,

Problem 4

Assigned here.

Child wants this seat

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
X		X	X	X	X		X	X	X

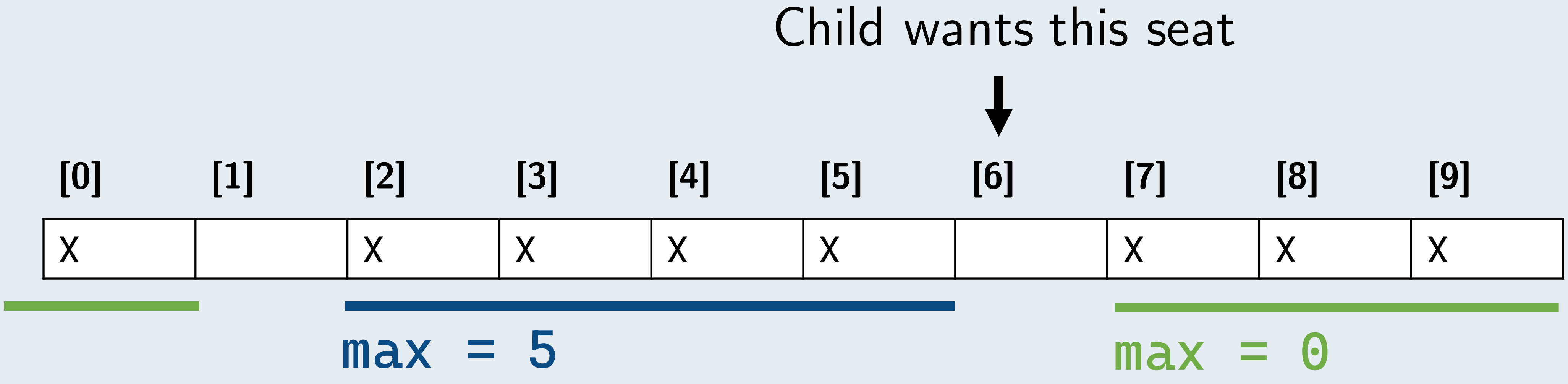
`max = 5`

`max = 0`

Idea: When assigning a new child i, check:

- If s_i is already in a group, insert at `(max + 1) % n` and update `max`.

Problem 4



Idea: When assigning a new child i, check:

- If s_i is already in a group, insert at `(max + 1) % n` and update `max`.
- After inserting, if its adjacent slots are in two different groups,

Problem 4

Child wants this seat

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
X		X	X	X	X	X	X	X	X

`max = 0`

Idea: When assigning a new child i, check:

- If s_i is already in a group, insert at `(max + 1) % n` and update `max`.
- After inserting, if its adjacent slots are in two different groups, merge them!

Appendix

Ackerman's Function *

For any two positive integers m and n, Ackerman's function is recursively defined by

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0 \\ A(m-1,1), & \text{if } m > 0 \text{ and } n = 0 \\ A\big(m-1, A(m,n-1)\big), & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Example.

$A(0,0) = 1,\ A(1,1) = A\big(0, A(1,0)\big) = A(0,2) = 3.$

$A(2,2) = 7,\ A(3,3) = 61,\ A(4,4) = 2^{2^{2^{65536}}} - 3 \ldots$

Increases very rapidly!

* *Adopted from Intro to Algorithm slides by Manuel Charlemagne.*

Inverse Ackerman's Function *

For $m = n$ we can define the inverse of Ackerman's function $\alpha(n)$, which grows very slowly.

$\alpha(3) = 1$, $\alpha(7) = 2$, $\alpha(61) = 3$, $\alpha\left(2^{2^{2^{65536}}} - 3\right) = 4 \ldots$

Can be almost seen as constant!

End of File

Thank you very much for your attention :-)