Rooting for Efficiency:
Mechanised Reasoning about Array-Based Trees in Separation Logic

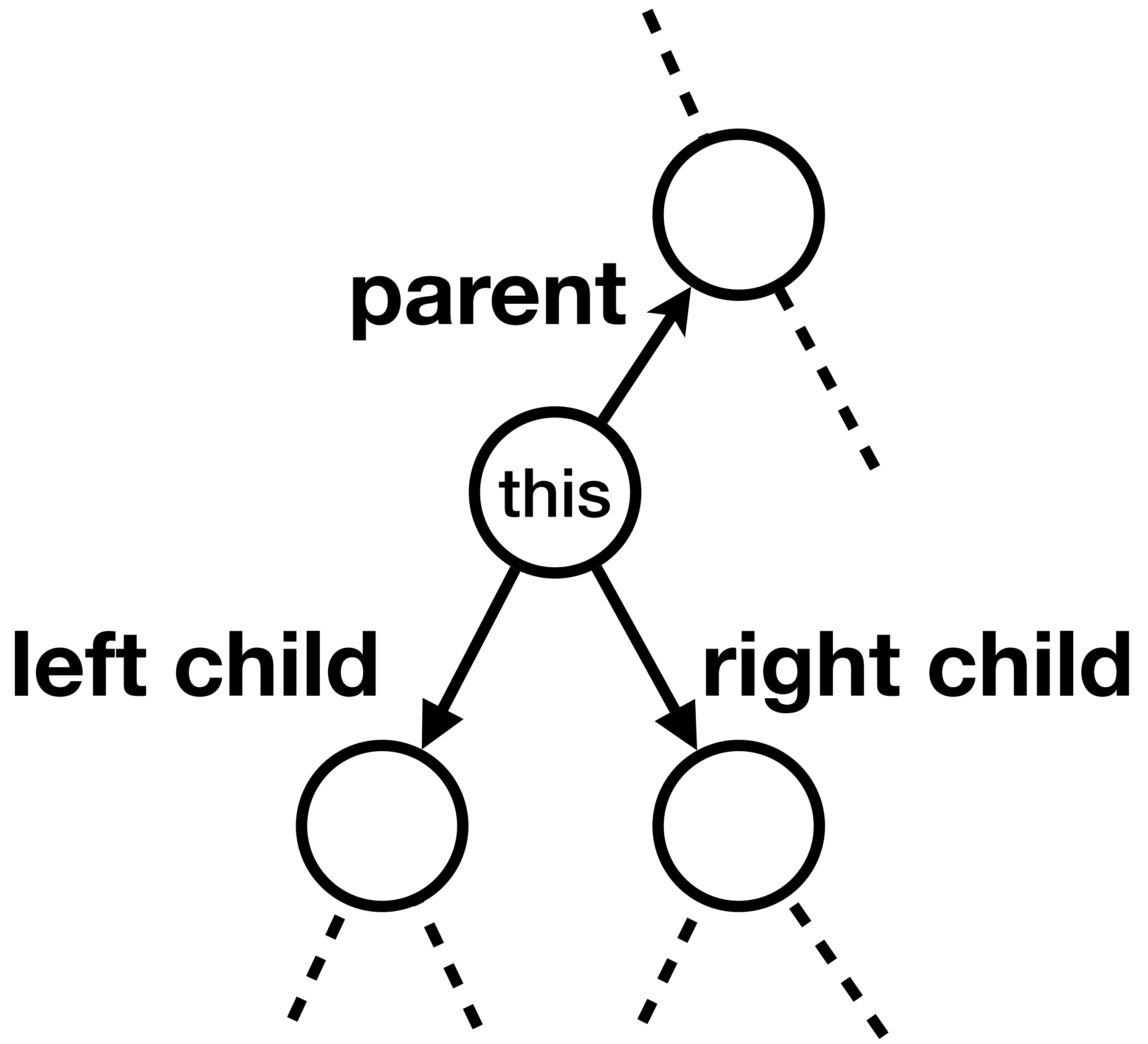
<u>Qiyuan Zhao</u>, George Pîrlea, Zhendong Ang, Umang Mathur, Ilya Sergey

NUS National University of Singapore

For CPP 2024

Pointer-Based Trees

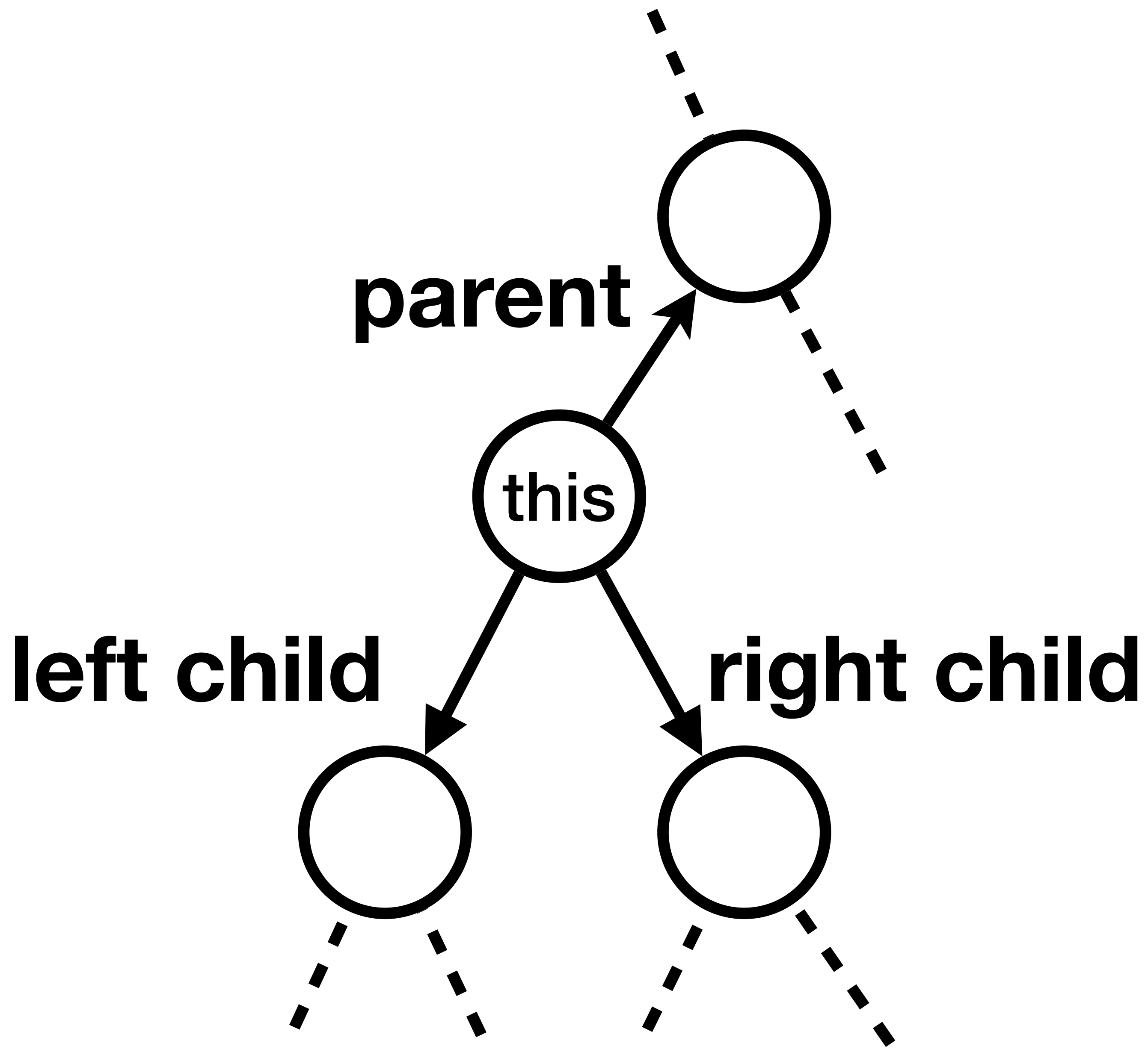
```
// binary tree
struct node {
  struct node
    *parent,
    *left_child,
    *right_child;
};
```

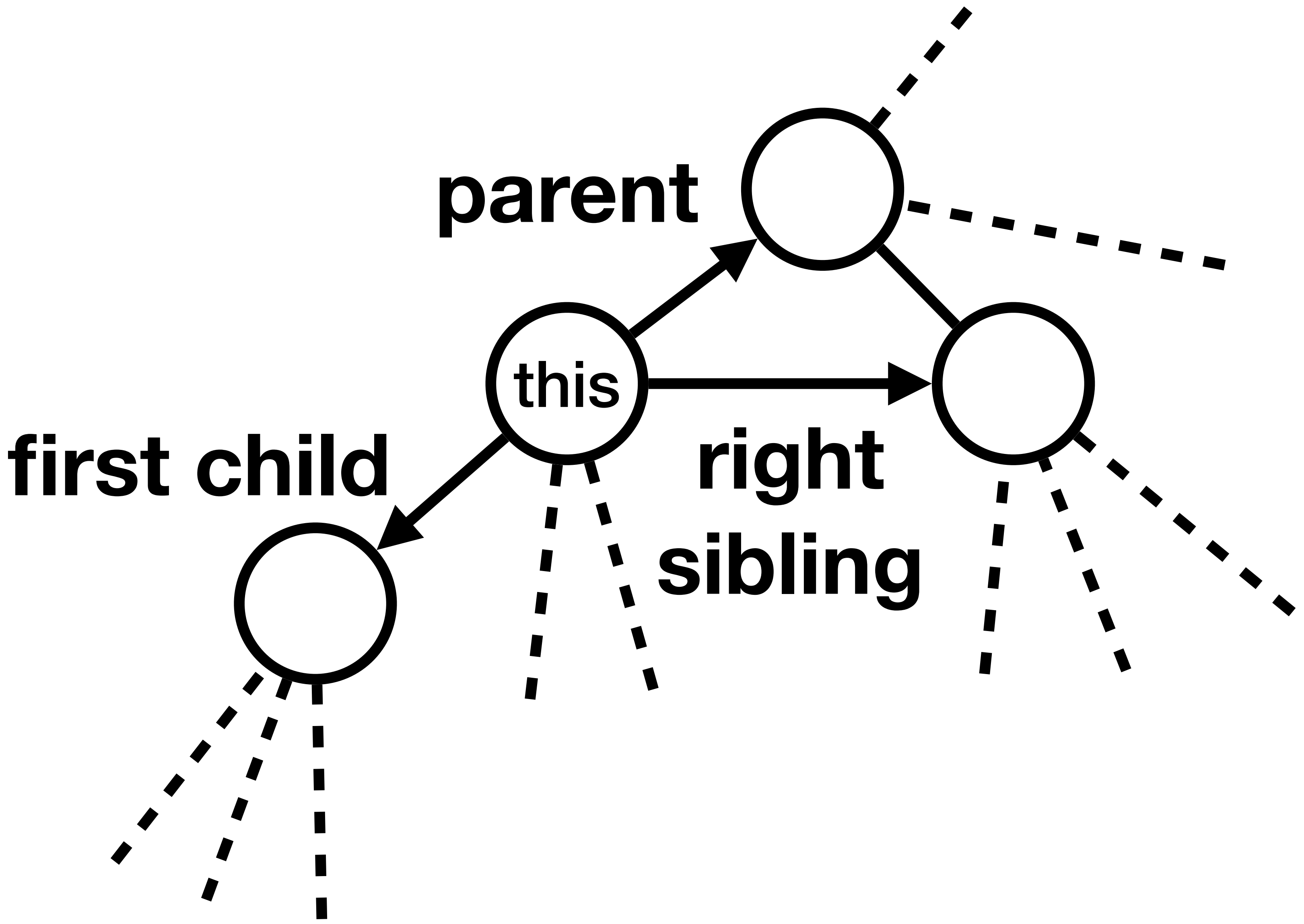


Pointer-Based Trees

```
// binary tree
struct node {
  struct node
    *parent,
    *left_child,
    *right_child;
};
```

```
// generic tree
struct node {
  struct node
    *parent,
    *right_sibling,
    *first_child;
};
```

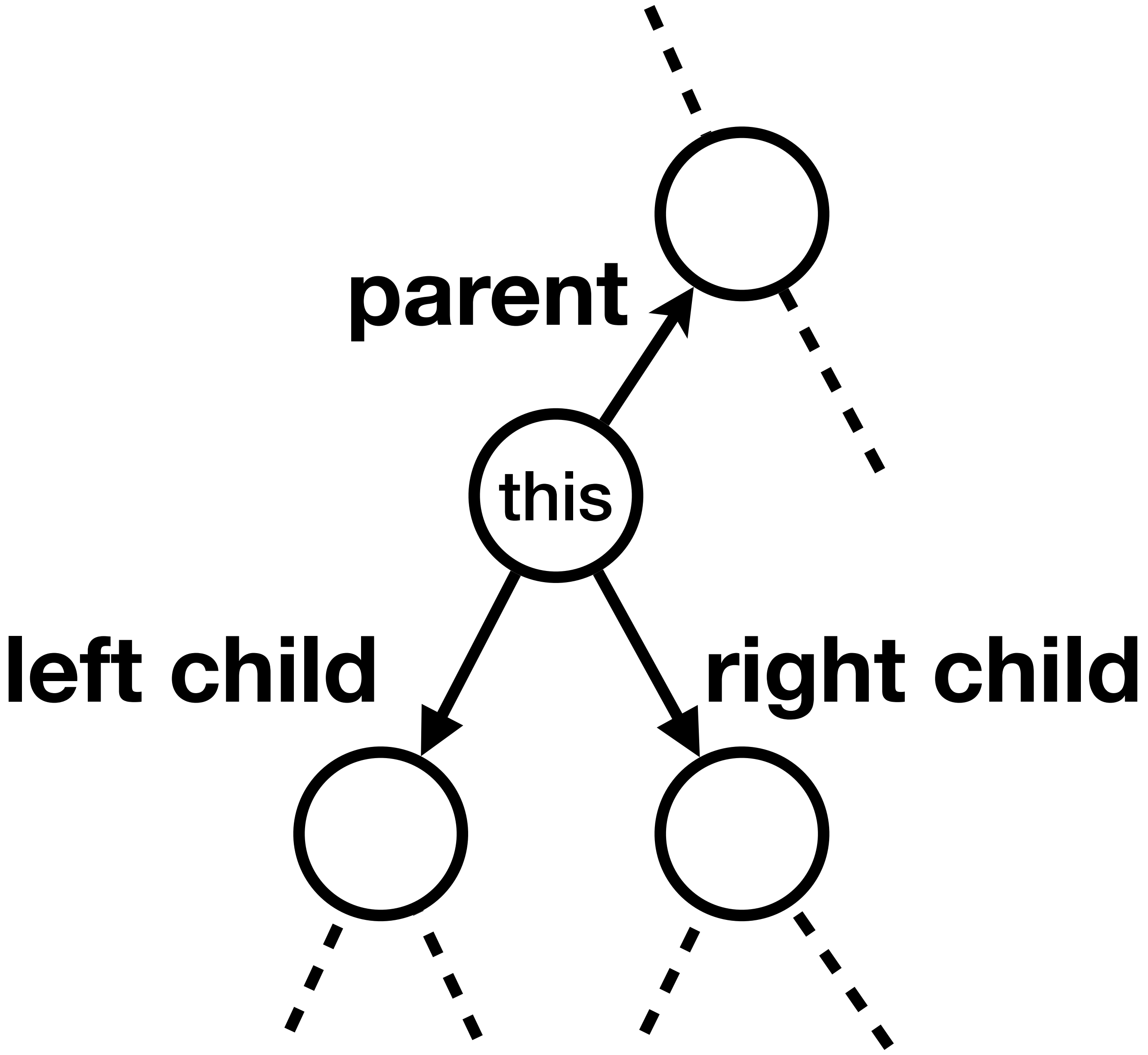


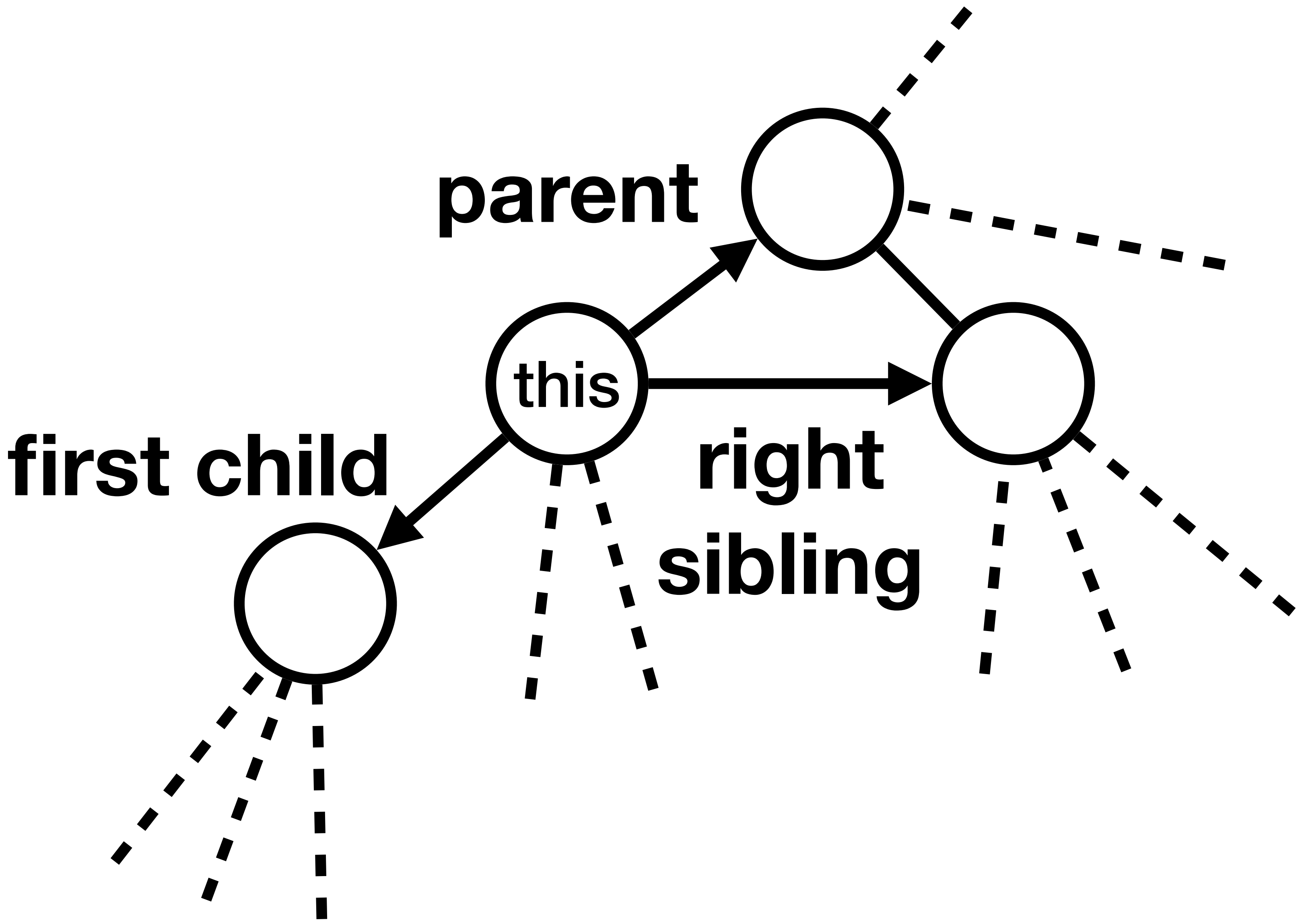


Pointer-Based Trees

```c
// binary tree
struct node {
  struct node
    *parent,
    *left_child,
    *right_child;
};
```

```c
// generic tree
struct node {
  struct node
    *parent,
    *right_sibling,
    *first_child;
};
```

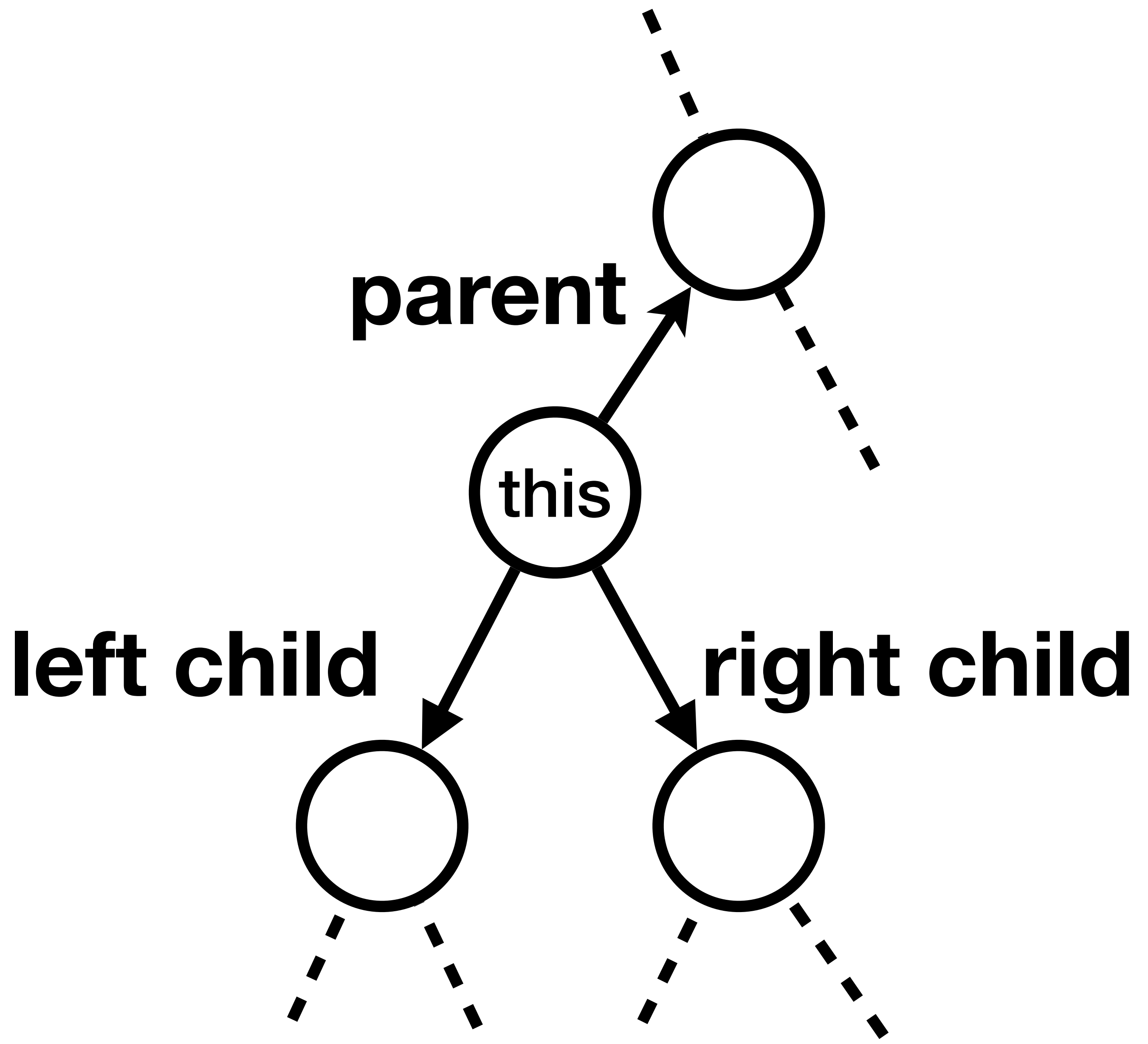


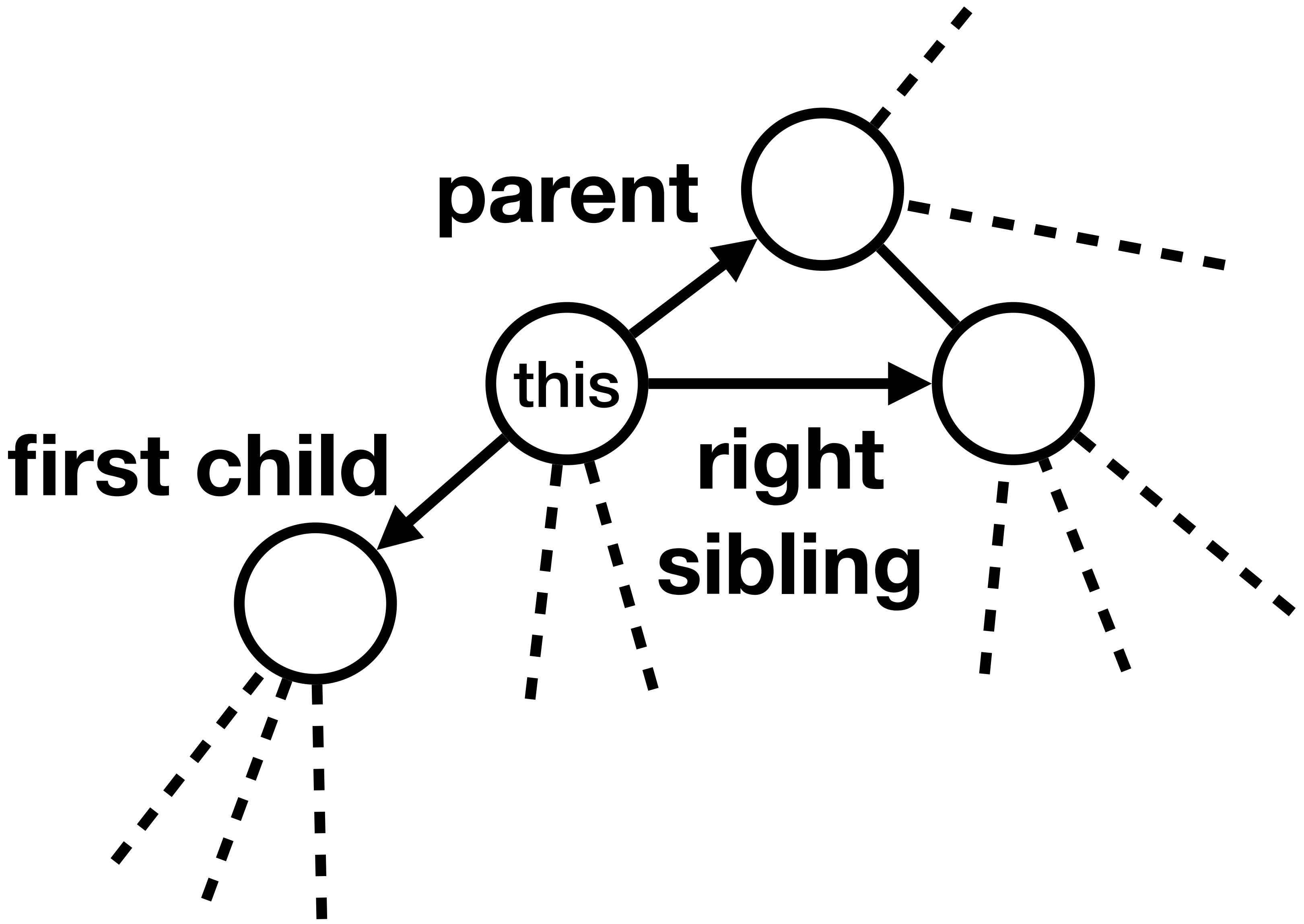


Pointer-Based Trees

```
// binary tree
struct node {
  struct node
    *parent,
    *left_child,
    *right_child;
};
```

```
// generic tree
struct node {
  struct node
    *parent,
    *right_sibling,
    *first_child;
};
```





Array-Based Trees

```
// generic tree
struct node {
  int parent,
      right_sibling,
      first_child;
};
const int NIL = -1;
struct node tree[N];
```

Array-Based Trees

- Use a `struct` array to store a tree

```
// generic tree
struct node {
  int parent,
      right_sibling,
      first_child;
};
const int NIL = -1;
struct node tree[N];
```

Array-Based Trees

- Use a `struct` array to store a tree
- Replace pointers with array indices

```
// generic tree
struct node {
  int parent,
      right_sibling,
      first_child;
};
const int NIL = -1;
struct node tree[N];
```

Array-Based Trees

- Use a `struct` array to store a tree
- Replace pointers with array indices
- Use a dedicated integer (e.g., `-1`) to represent `NULL` pointer

```
// generic tree
struct node {
  int parent,
      right_sibling,
      first_child;
};
const int NIL = -1;
struct node tree[N];
```

Array-Based Trees (Cont’d)

- Sometimes array-based trees are preferable …

Array-Based Trees (Cont'd)

- Sometimes array-based trees are preferable …
  - Potential time/space efficiency benefits

Array-Based Trees (Cont’d)

- Sometimes array-based trees are preferable …
  - Potential time/space efficiency benefits
    - E.g., random access to a node’s information

Array-Based Trees (Cont'd)

- Sometimes array-based trees are preferable …
  - Potential time/space efficiency benefits
    - E.g., random access to a node's information
- **But formally reasoning about array-based trees can be challenging!**

Array-Based Trees in Separation Logic

Array-Based Trees in Separation Logic

$$\text{tree\_rep}_{\text{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \text{tree\_proj}(tr, \ell) \rceil$$
$$* \text{arr}(p, \ell)$$

- A <u>representation predicate</u> for array-based tree

Array-Based Trees in Separation Logic

$$\text{tree\_rep}_{\text{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \text{tree\_proj}(tr, \ell) \rceil$$
$$* \text{arr}(p, \ell)$$

- A <u>representation predicate</u> for array-based tree
  - $\text{arr}(p, \ell)$: heap predicate, "array ℓ is stored at pointer p"

Array-Based Trees in Separation Logic

$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell)$$

- A <u>representation predicate</u> for array-based tree
    - $\mathsf{arr}(p, \ell)$: heap predicate, "array ℓ is stored at pointer p"
    - $\mathsf{tree\_proj}(tr, \ell)$: pure proposition, "array ℓ stores the tree tr"

Array-Based Trees in Separation Logic

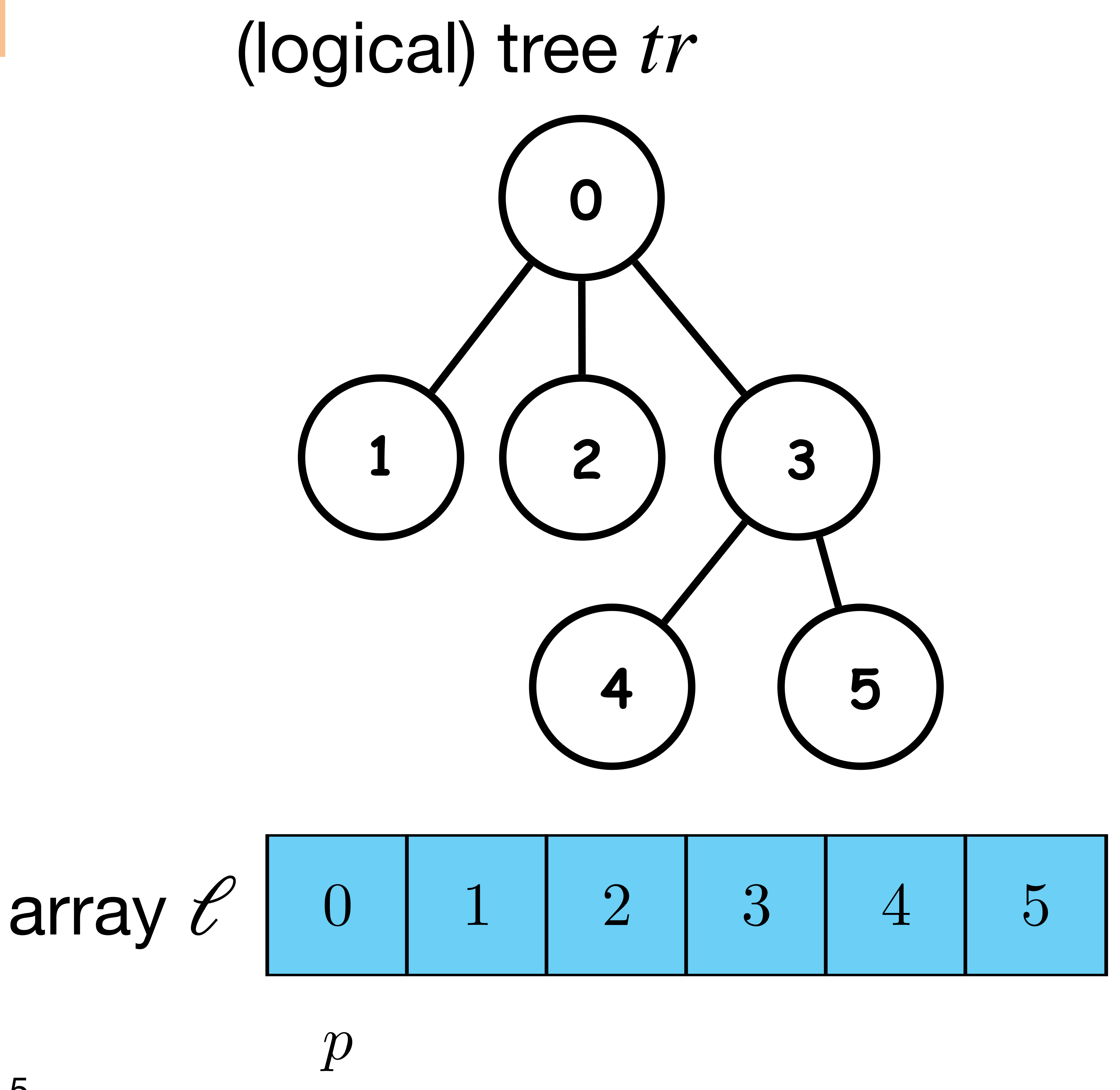
$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell)$$

- A <u>representation predicate</u> for array-based tree
  - $\mathsf{arr}(p, \ell)$: heap predicate, "array ℓ is stored at pointer p"
  - $\mathsf{tree\_proj}(tr, \ell)$: pure proposition, "array ℓ stores the tree tr"
    - defined recursively on tr

Array-Based Trees in Separation Logic

$$\text{tree\_rep}_{\text{arr}}(p, tr) \triangleq \exists \ell,\quad \lceil \text{tree\_proj}(tr, \ell) \rceil * \text{arr}(p, \ell)$$

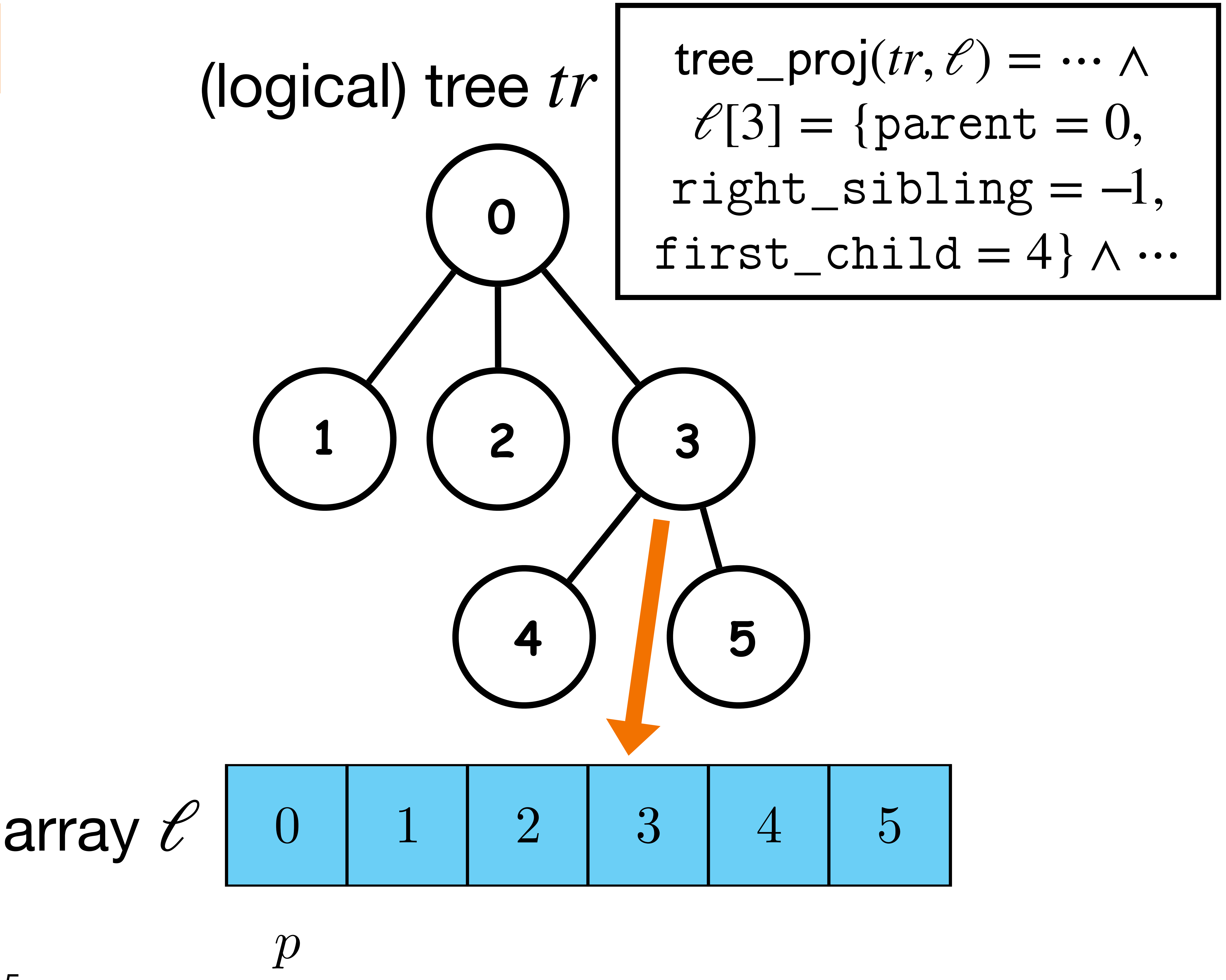
- A <u>representation predicate</u> for array-based tree
  - $\text{arr}(p, \ell)$: heap predicate, "array ℓ is stored at pointer p"
  - $\text{tree\_proj}(tr, \ell)$: pure proposition, "array ℓ stores the tree tr"
    - defined recursively on tr



Array-Based Trees in Separation Logic

$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell)$$

- A <u>representation predicate</u> for array-based tree
  - $\mathsf{arr}(p, \ell)$: heap predicate, "array ℓ is stored at pointer p"
  - $\mathsf{tree\_proj}(tr, \ell)$: pure proposition, "array ℓ stores the tree tr"
    - defined recursively on tr



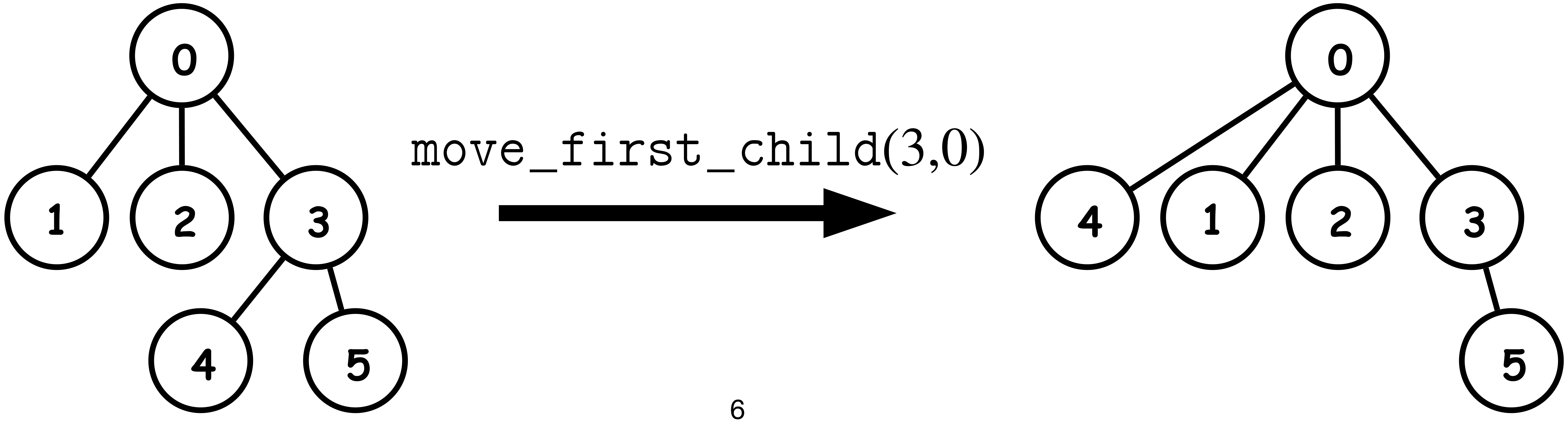
Challenge #1: Structure Changing Operation

Challenge #1: Structure Changing Operation

```
void move_first_child (int src, int dst) {
  // move the first child of node src
  // to be the first child of node dst
}
```

Challenge #1: Structure Changing Operation

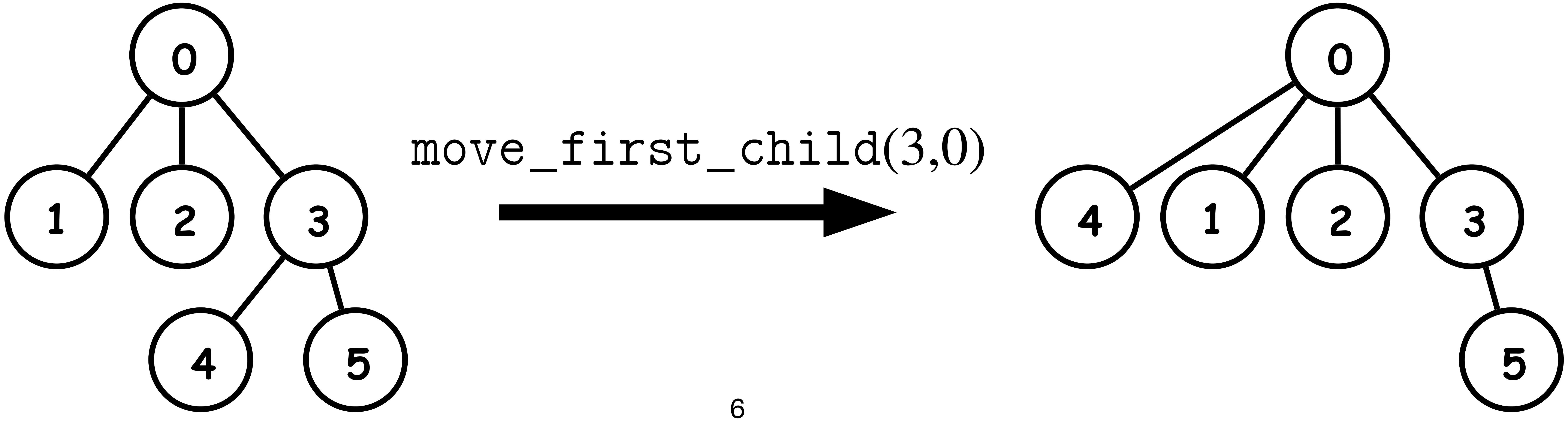
```
void move_first_child (int src, int dst) {
  // move the first child of node src
  // to be the first child of node dst
}
```



Challenge #1: Structure Changing Operation

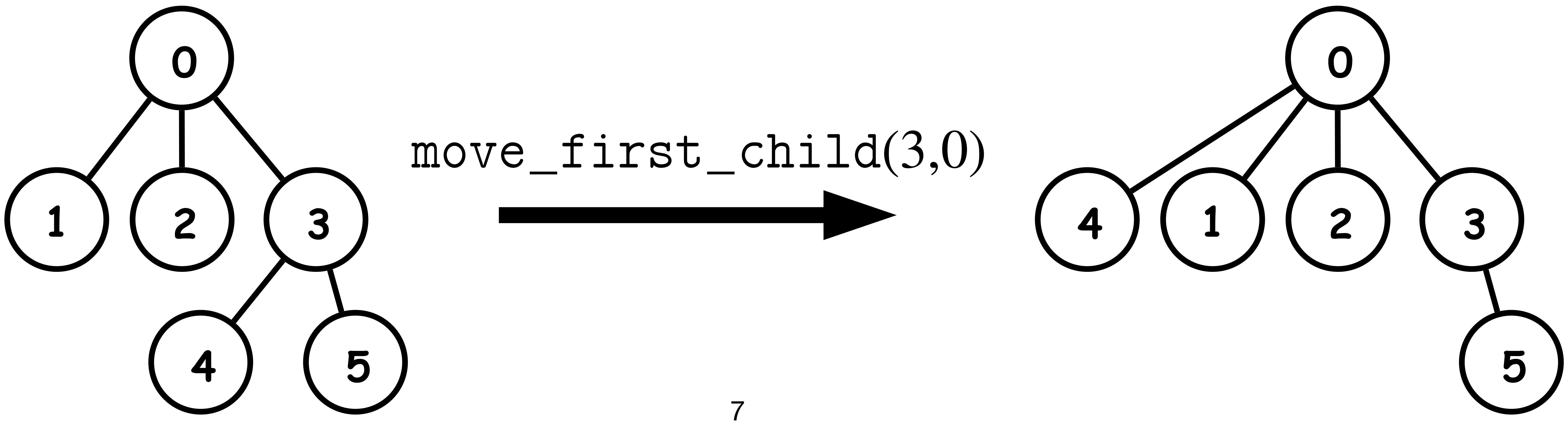
```c
void move_first_child (int src, int dst) {
  // move the first child of node src
  // to be the first child of node dst
}
```

? How to specify and verify this program?



Challenge #1: Structure Changing Operation

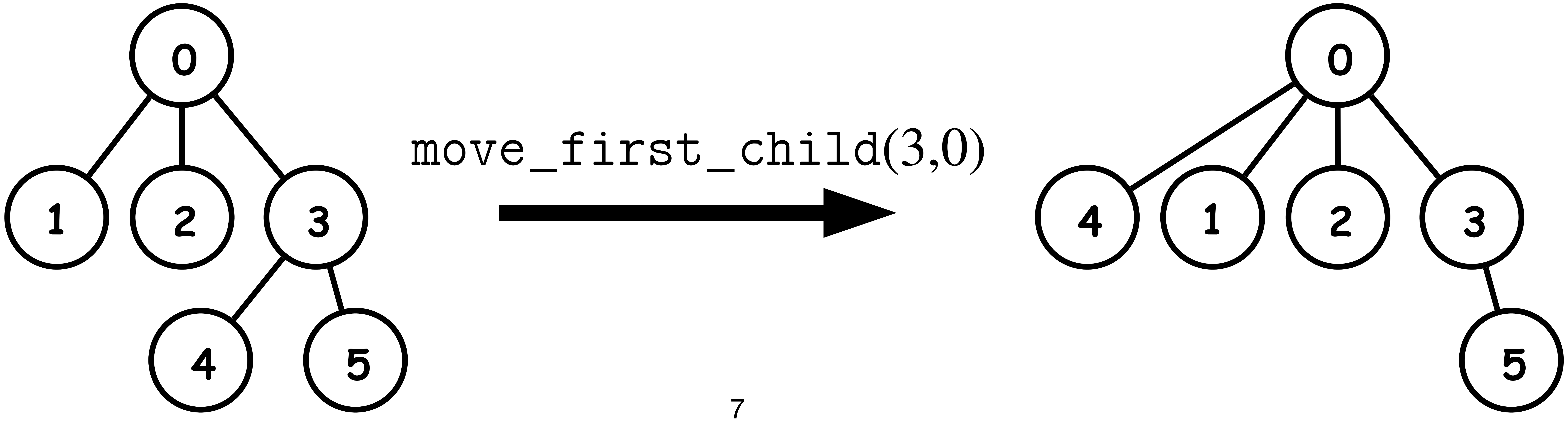
How to specify and verify this program?



Challenge #1: Structure Changing Operation

- Need to specify what parts of the array have changed, and how they change

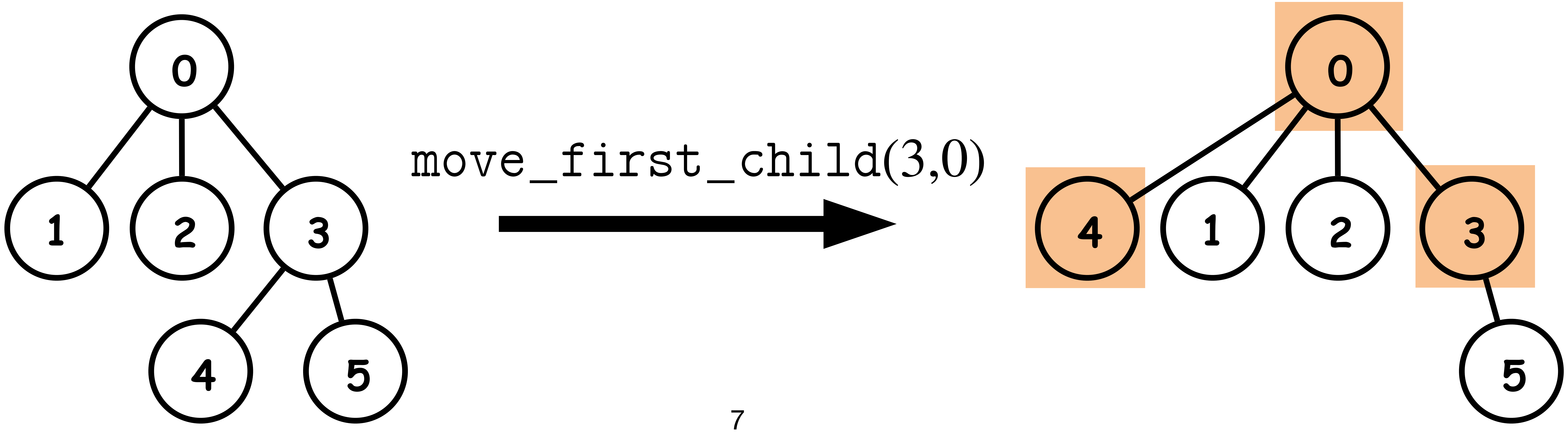
How to specify and verify this program?



Challenge #1: Structure Changing Operation

- Need to specify what parts of the array have changed, and how they change

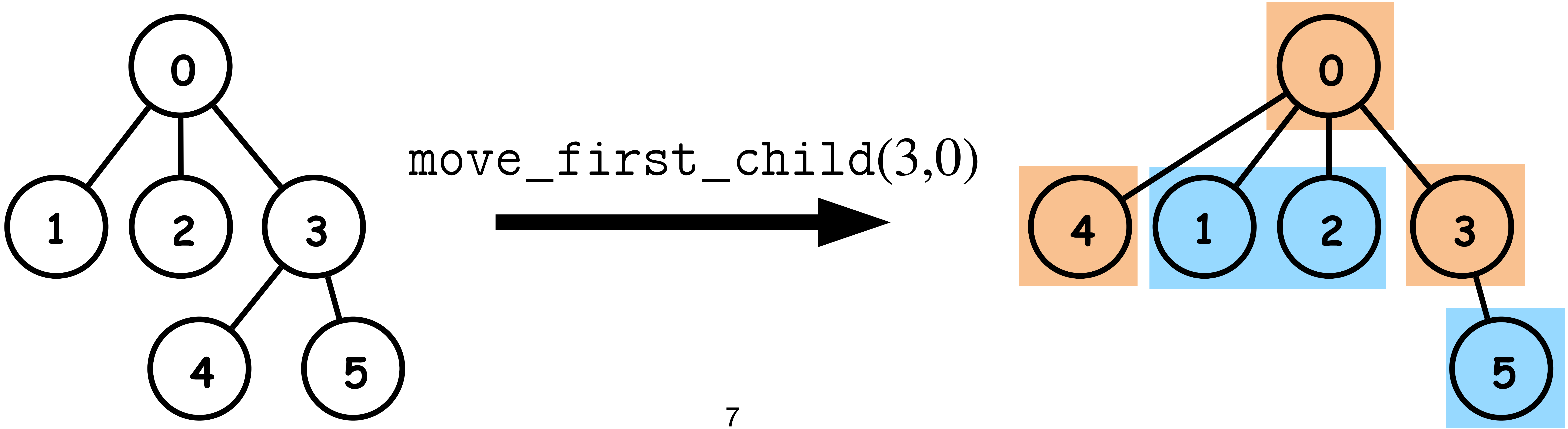
? How to specify and verify this program?



Challenge #1: Structure Changing Operation

- Need to specify what parts of the array have changed, and how they change
- Need to prove that the other parts are kept intact

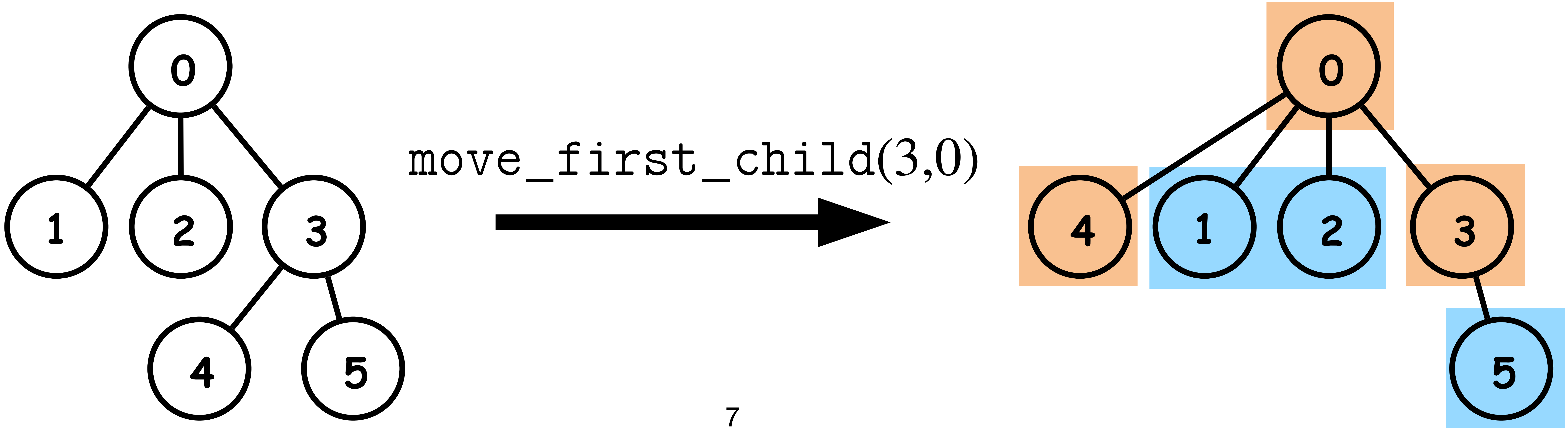
❓ How to specify and verify this program?



Challenge #1: Structure Changing Operation

- Need to specify what parts of the array have changed, and how they change
- Need to prove that the other parts are kept intact
- $\Longrightarrow$ Want a “frame rule” to do **localised** reasoning on changed parts only

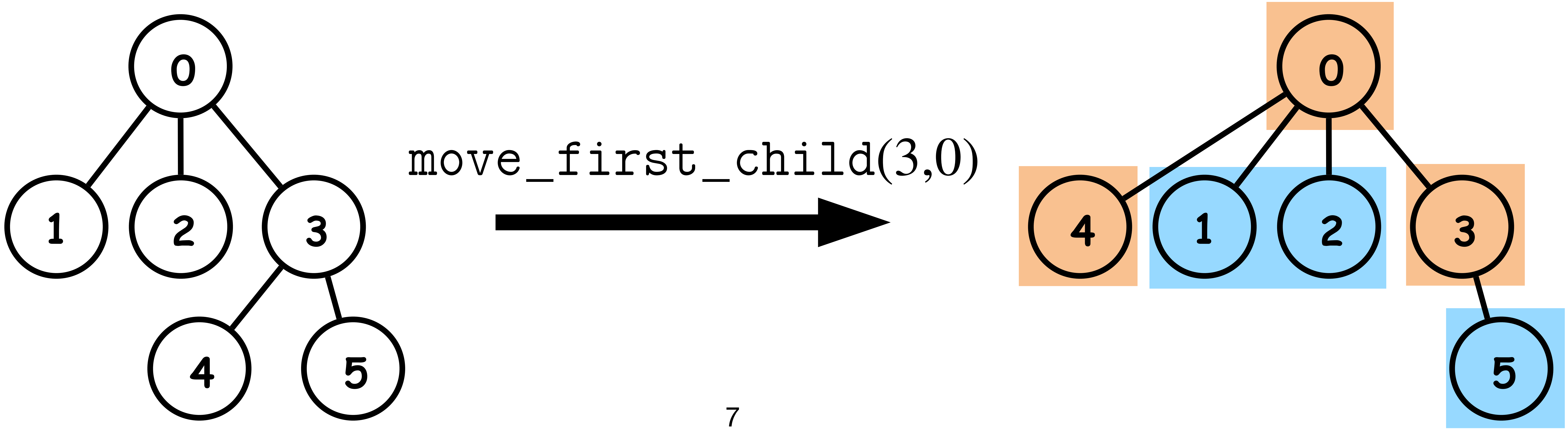
? How to specify and verify this program?



Challenge #1: Structure Changing Operation

- Need to specify what parts of the array have changed, and how they change
- Need to prove that the other parts are kept intact
- $\Longrightarrow$ Want a “frame rule” to do **localised** reasoning on changed parts only
- $\Longrightarrow$ Need to “separate” the array, but a separated part cannot be represented by $\mathsf{tree\_rep}_{\mathsf{arr}}$

❓ How to specify and verify this program?



Challenge #2: Non-Recursive Traversal

Challenge #2: Non-Recursive Traversal

```
void rec_traversal (int root) {
  // ...
  // call rec_traversal
  // for each child of root
}
```

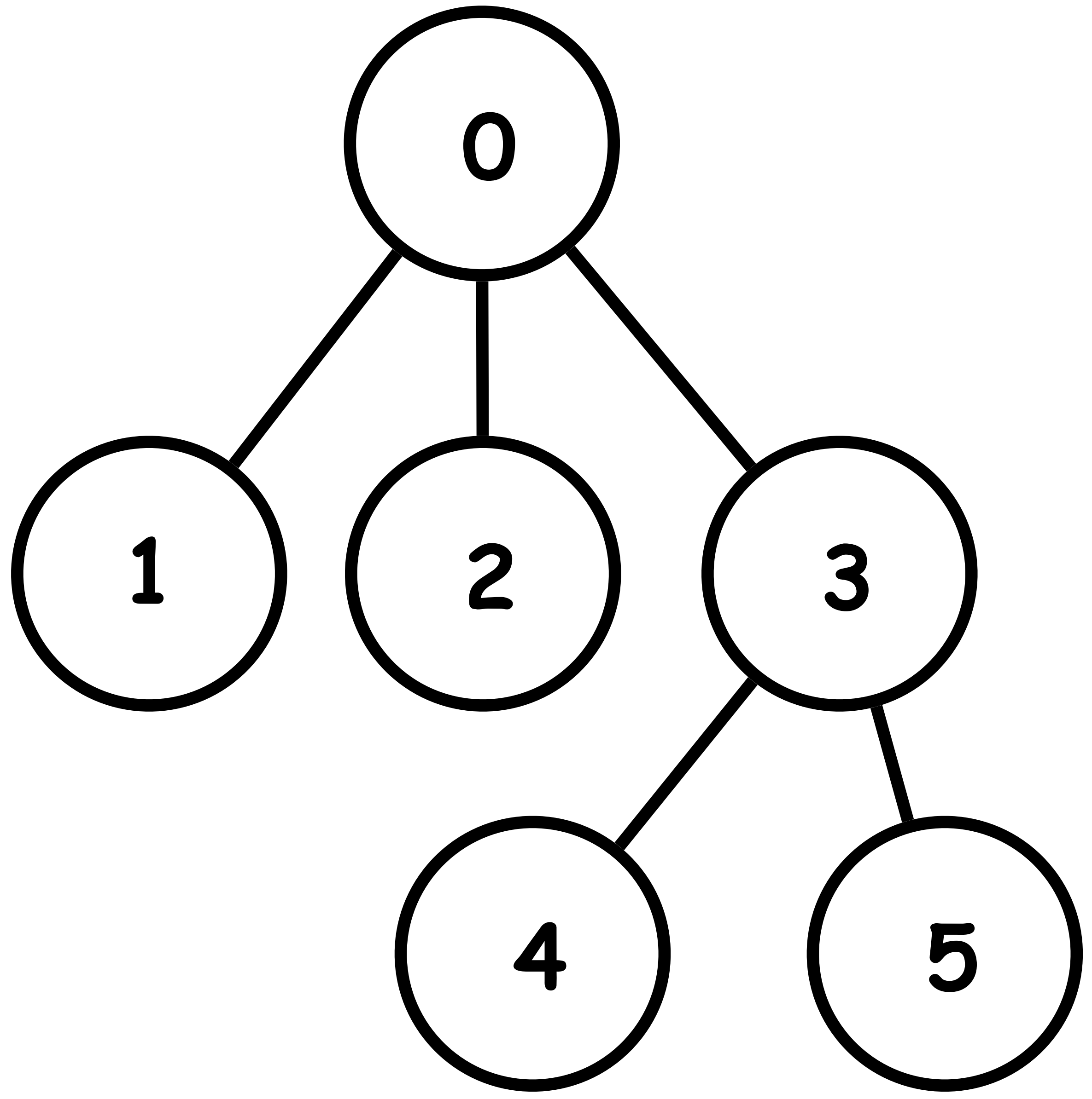
Challenge #2: Non-Recursive Traversal

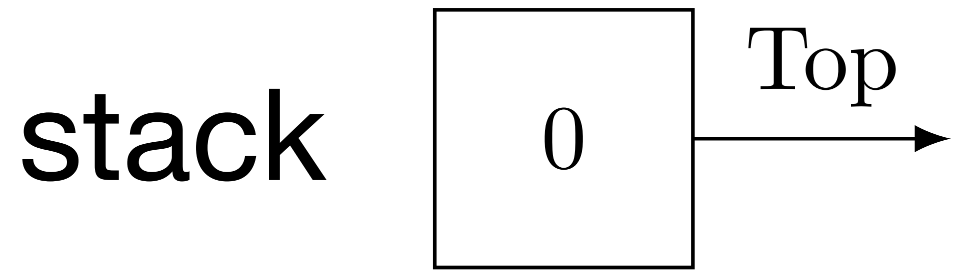
```
void rec_traversal (int root) {
  // ...
  // call rec_traversal
  // for each child of root
}
```

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

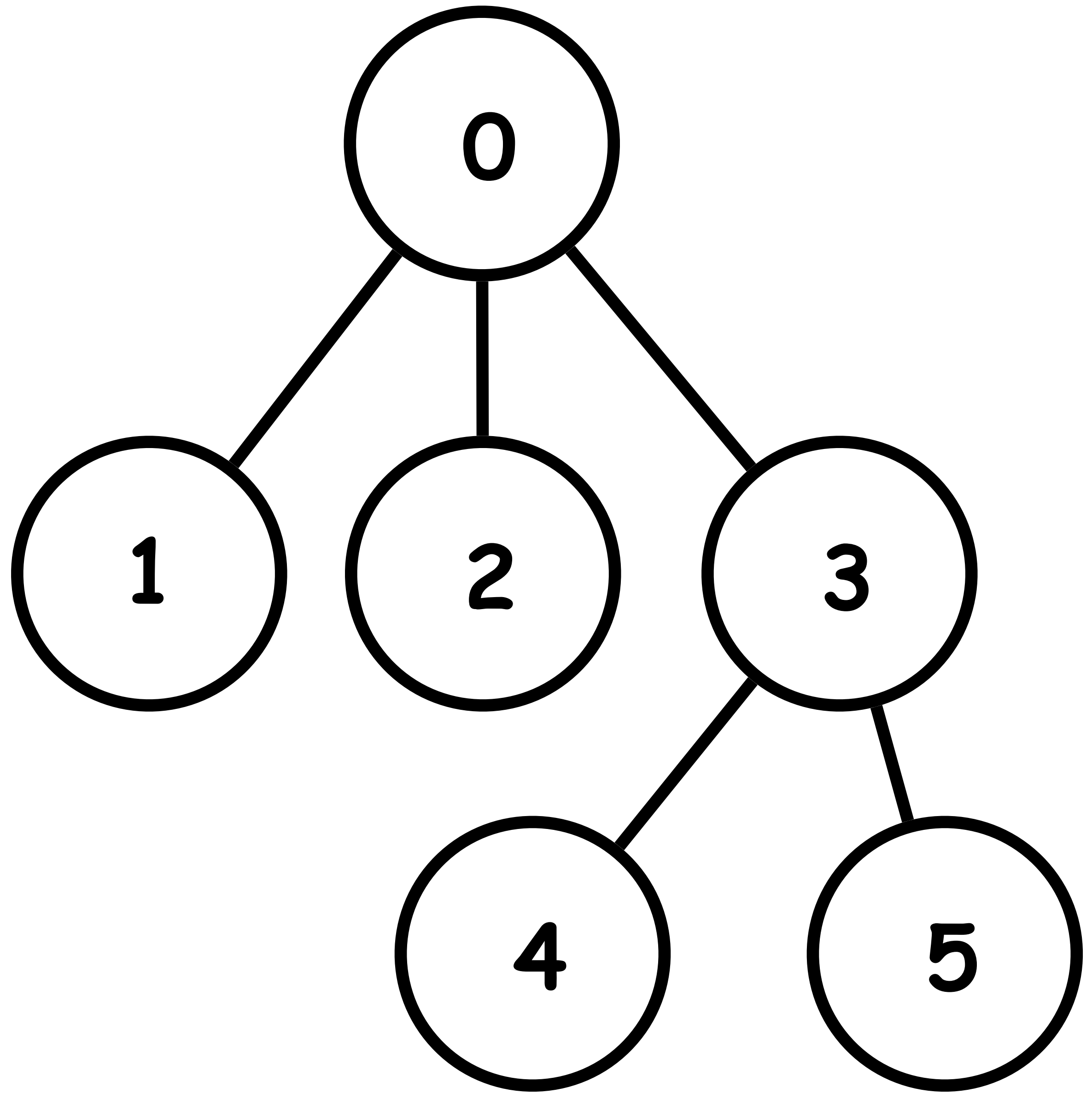




node visiting order

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

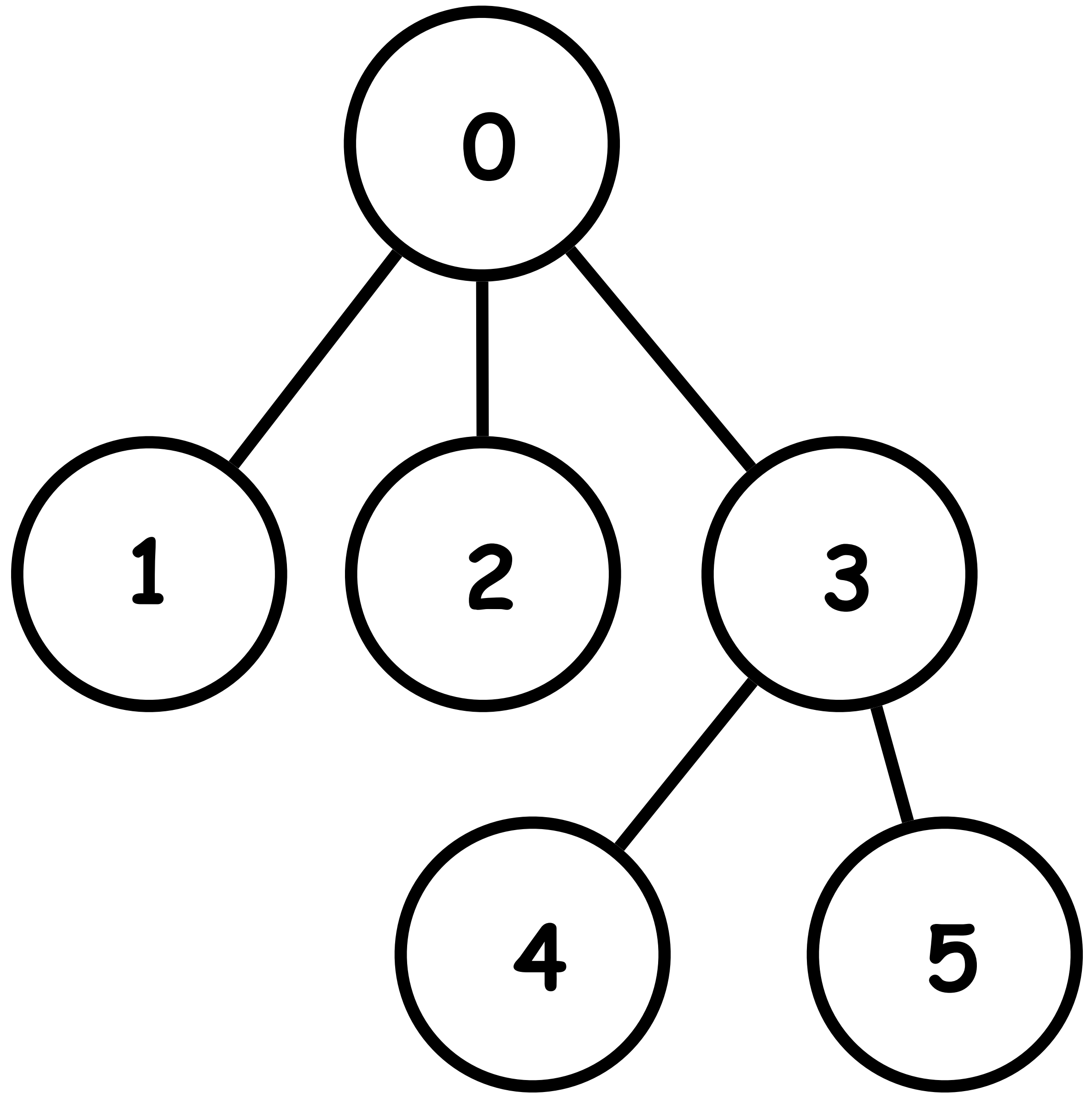


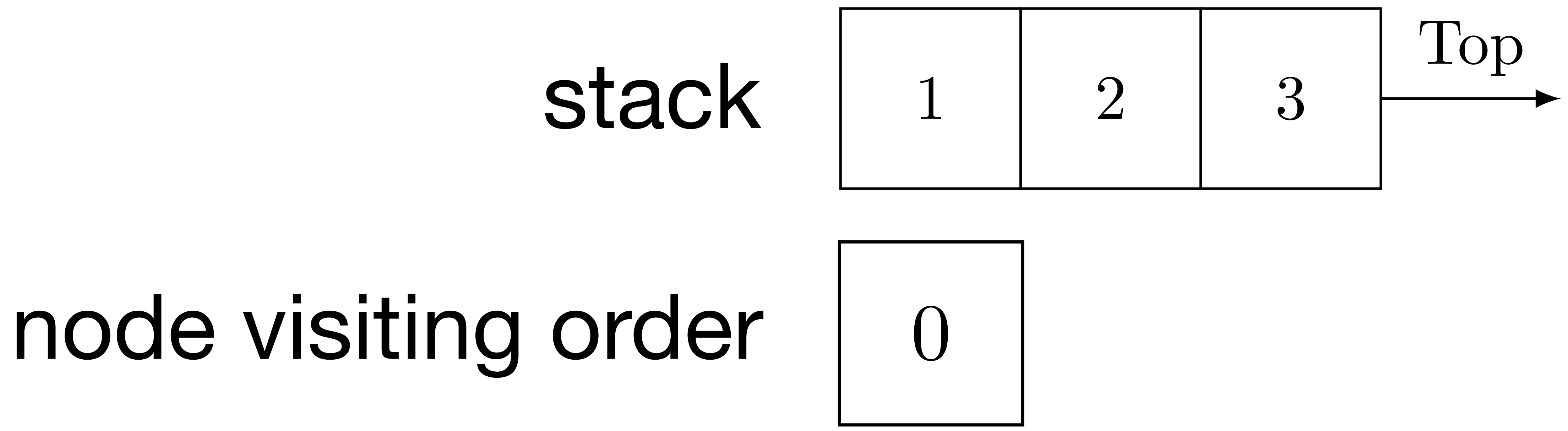
stack

node visiting order | 0

Challenge #2: Non-Recursive Traversal

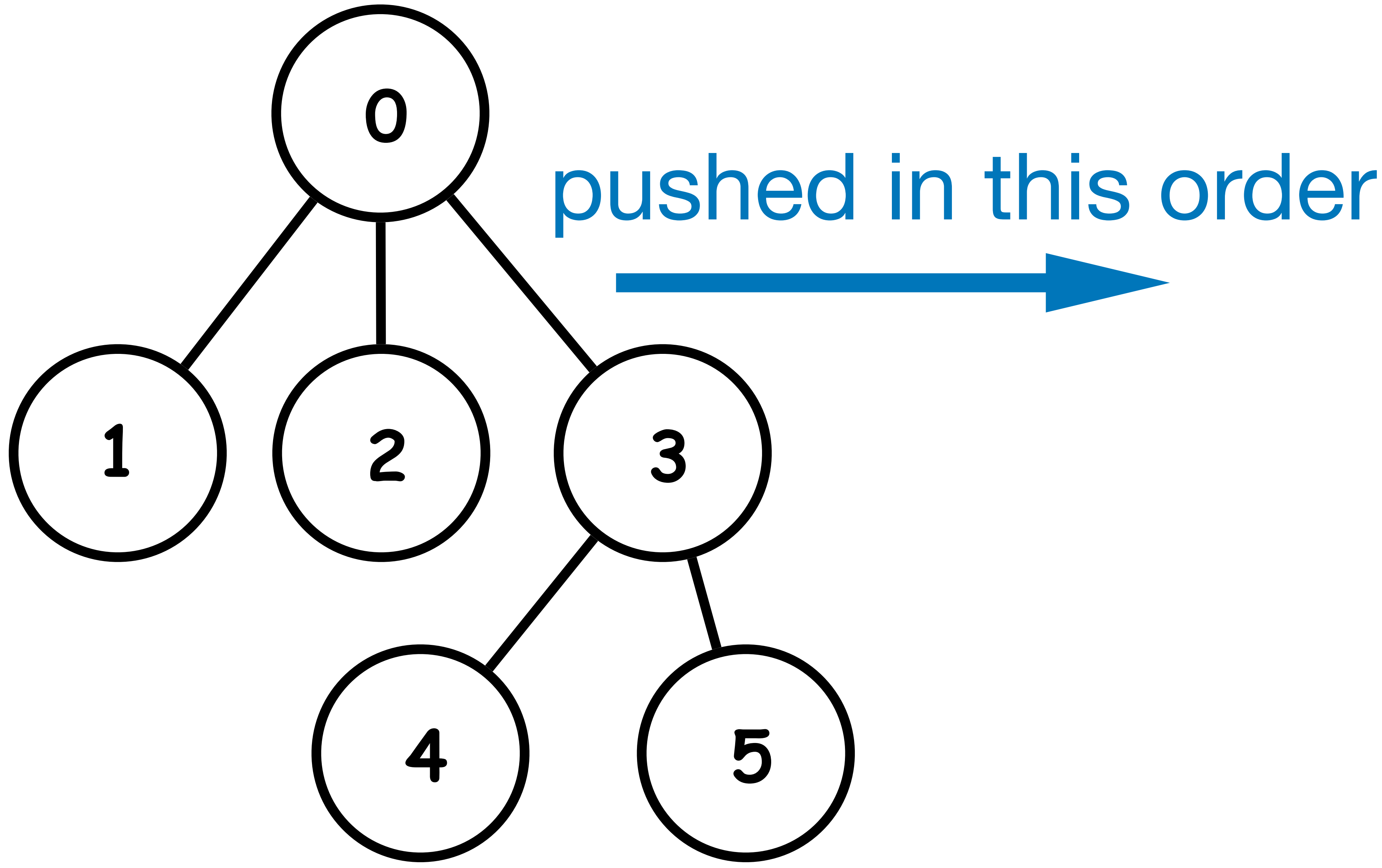
```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

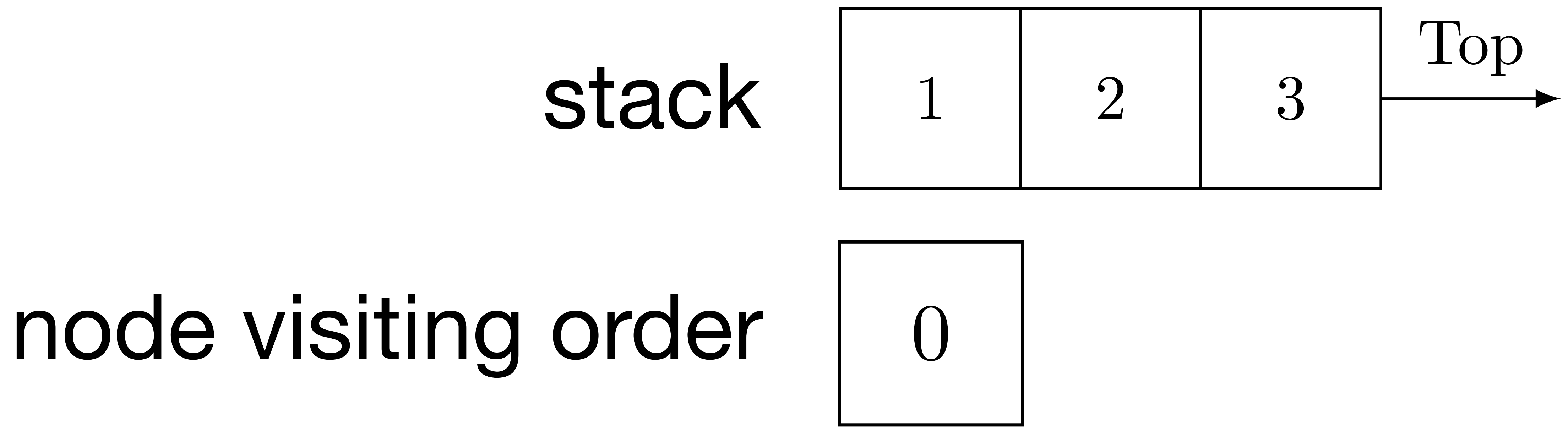




Challenge #2: Non-Recursive Traversal

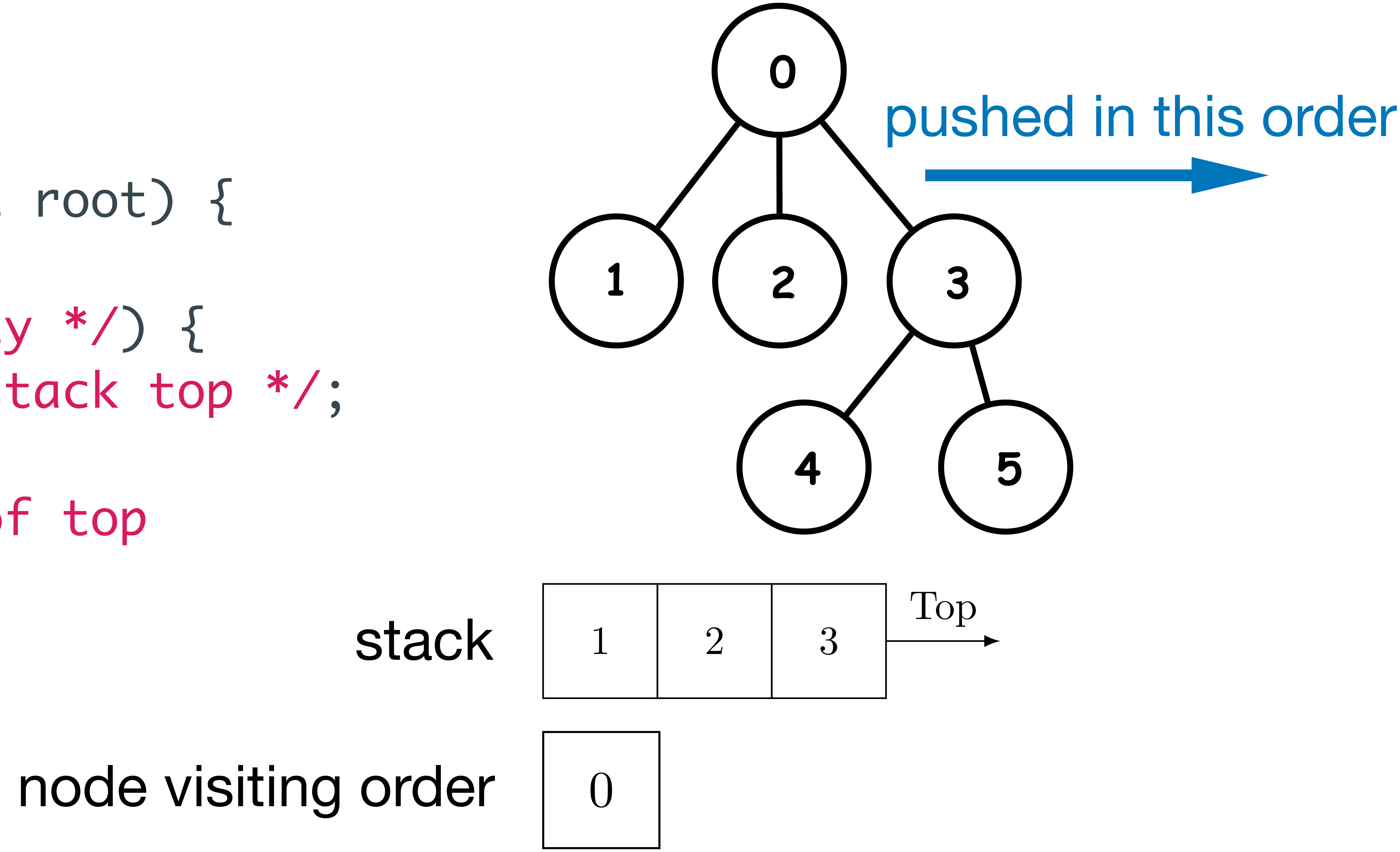
```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```





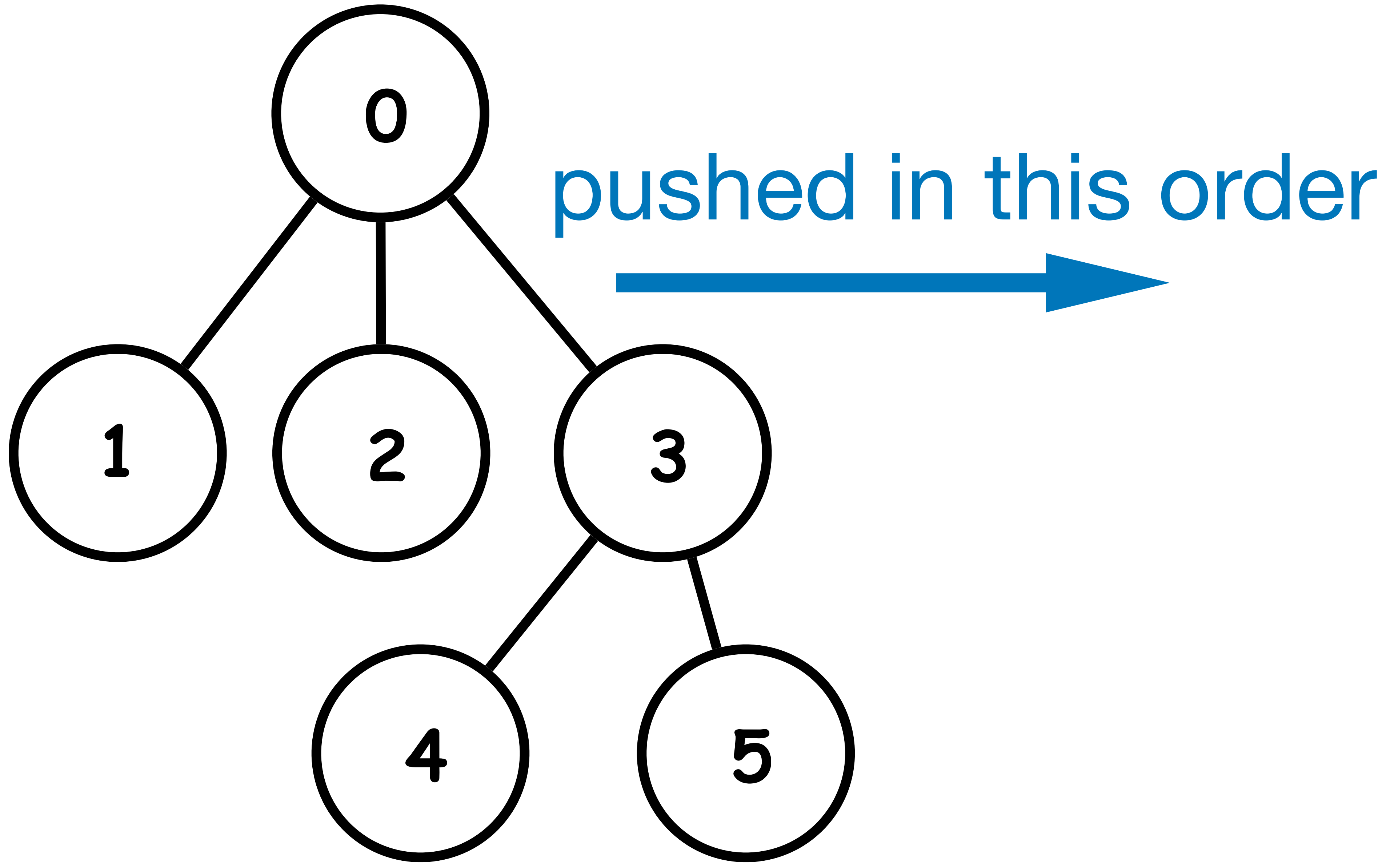
Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```



Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

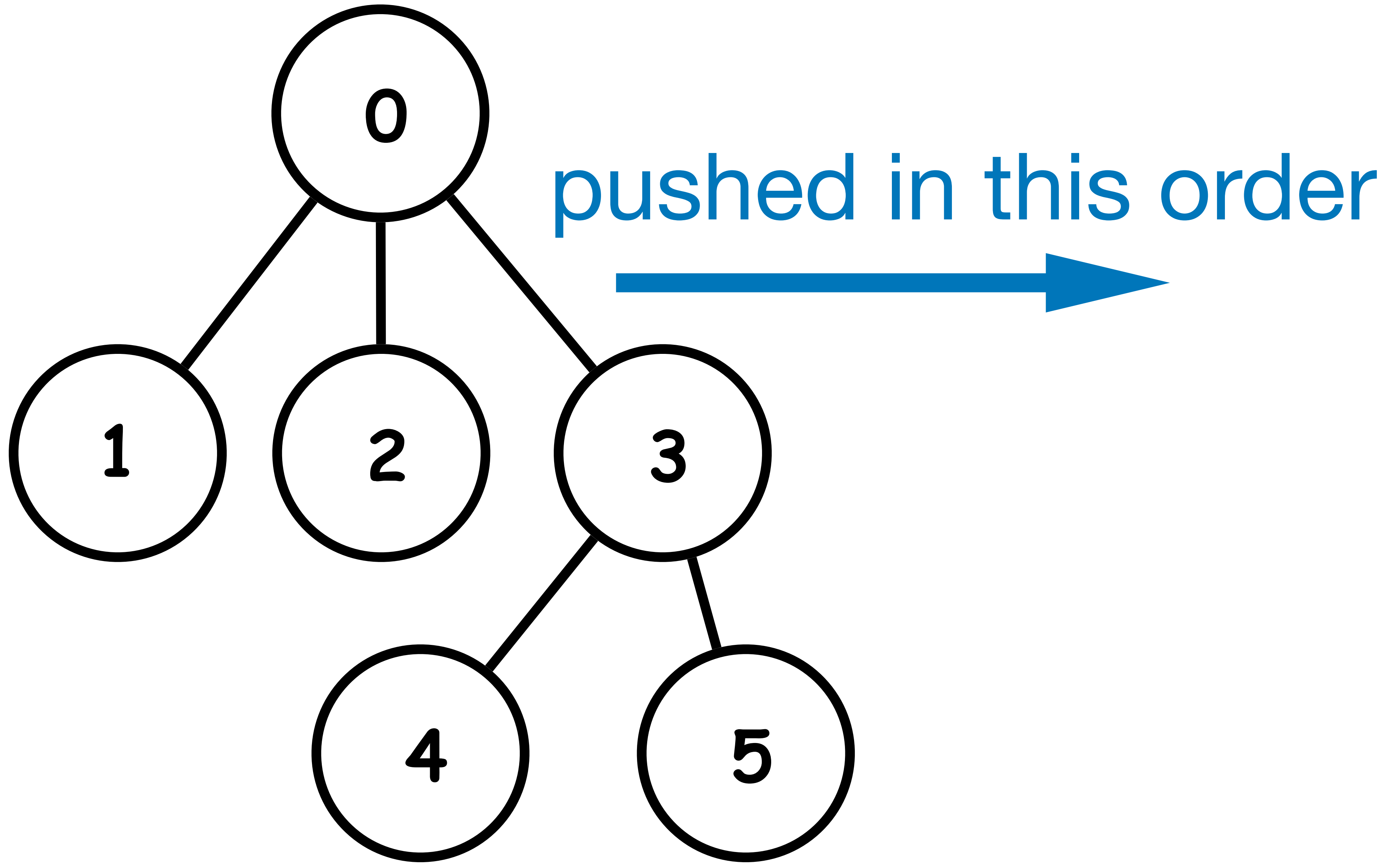


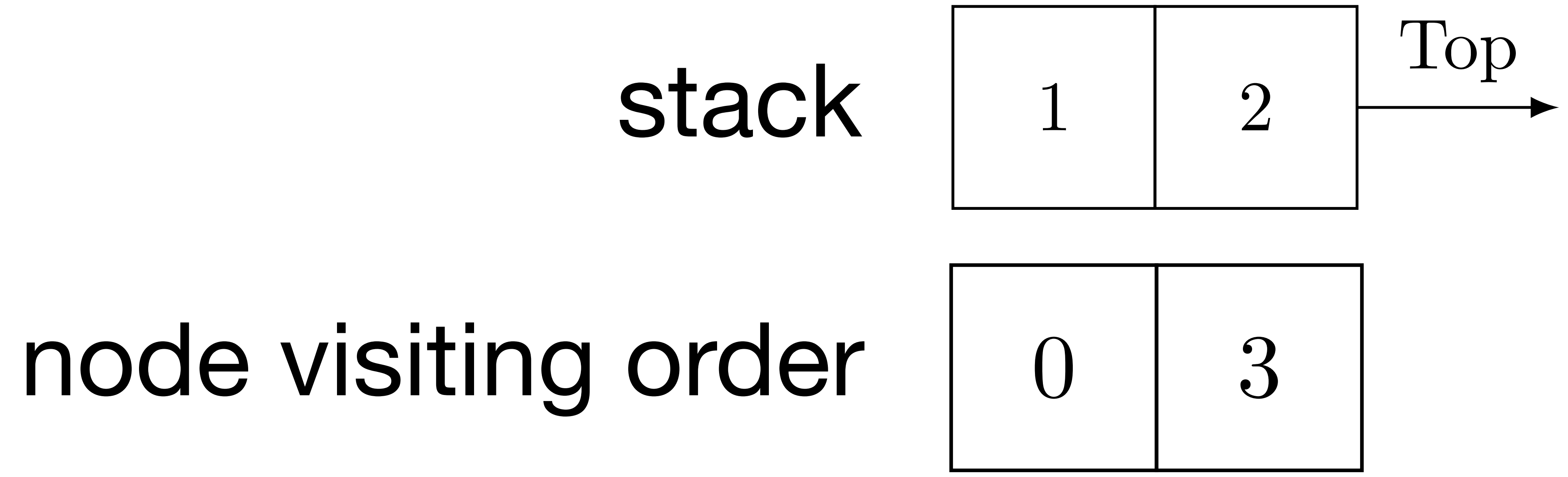
stack: 1 | 2 → Top

node visiting order: 0

Challenge #2: Non-Recursive Traversal

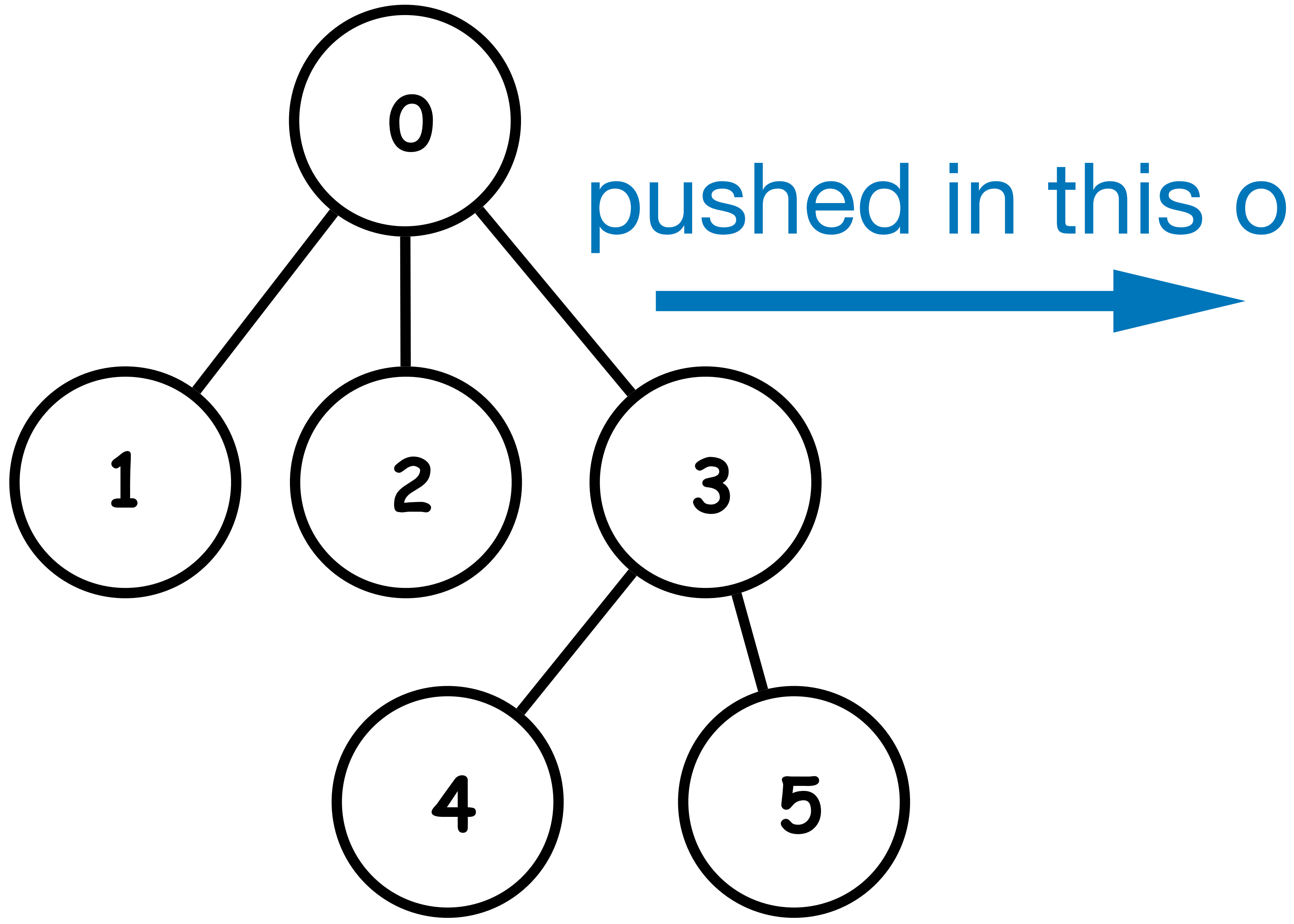
```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```





Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```



stack

1	2	4	5

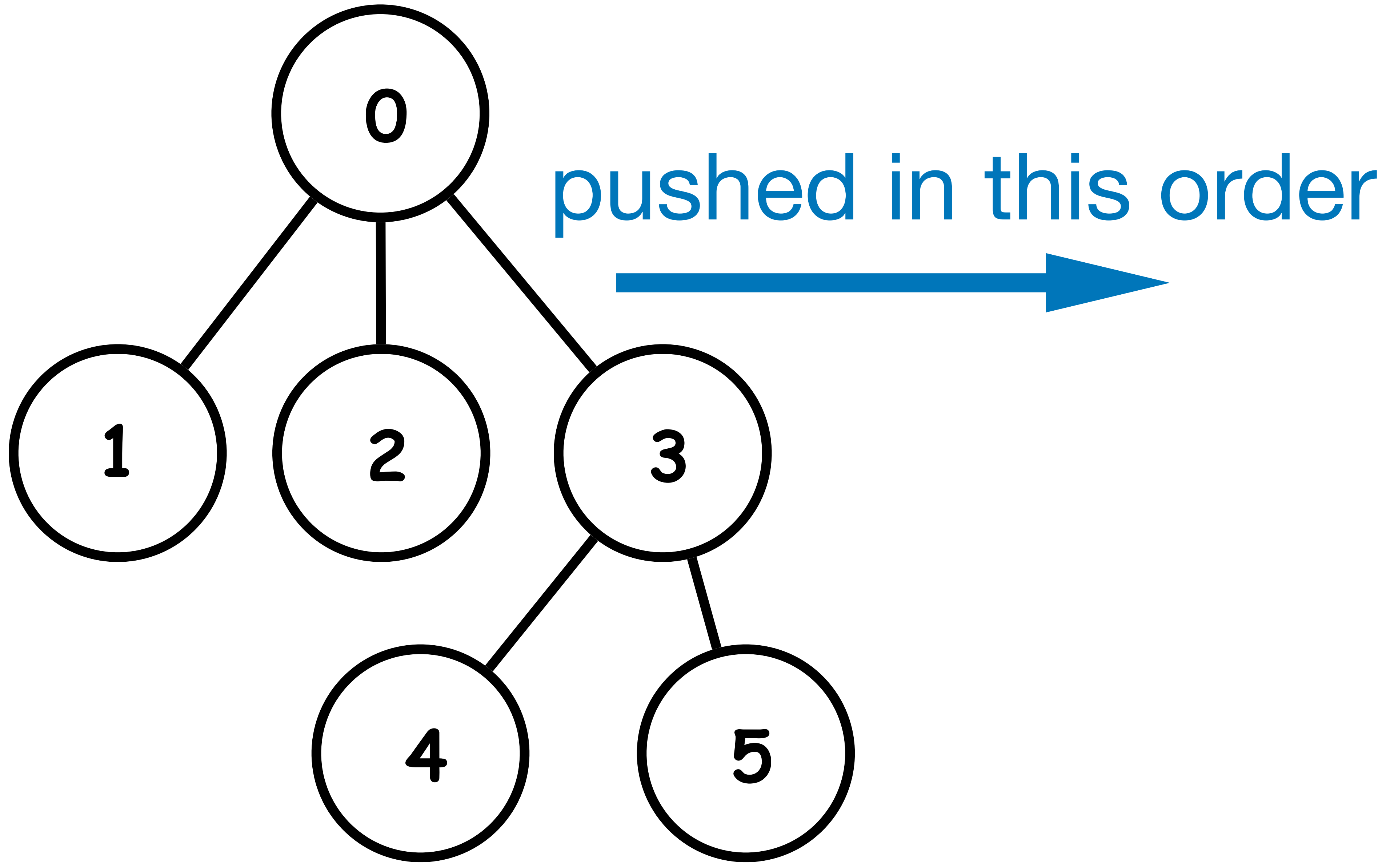
Top

node visiting order

0	3

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```



stack

node visiting order

0	3	5	4	2	1

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

Challenge #2: Non-Recursive Traversal

❓ How to specify the loop invariant?

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

❓ How to specify the loop invariant?

stack content

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

? How to specify the loop invariant?

- stack content
- **exact** visited part

Challenge #2: Non-Recursive Traversal

```
int stack[N];
void nonrec_traversal (int root) {
  // push root
  while (/* stack not empty */) {
    int top = /* pop out stack top */;
    // ...
    // push the children of top
  }
}
```

❓ How to specify the loop invariant?

stack content

exact visited part

Should be consistent & maintained in sync

Roadmap

- Challenges
- Strategies
- Case study

Roadmap

- Challenges
- Strategies
- Case study

Strategy to C1: Dual Views

Array view
(defined before)

$$\text{tree\_rep}_{\text{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \text{tree\_proj}(tr, \ell) \rceil * \text{arr}(p, \ell)$$

Strategy to C1: Dual Views

Array view
(defined before)

$$\mathrm{tree\_rep}_{\mathrm{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \mathrm{tree\_proj}(tr, \ell) \rceil \\ \qquad\qquad\qquad\qquad * \mathrm{arr}(p, \ell)$$

Tree view

$$\mathrm{tree\_rep}_{\mathrm{tree}}(p, tr) \triangleq \cdots$$

Strategy to C1: Dual Views

Array view
(defined before)

$$\text{tree\_rep}_{\text{arr}}(p, tr) \triangleq \exists \ell, \quad \lceil \text{tree\_proj}(tr, \ell) \rceil \\ \qquad * \text{arr}(p, \ell)$$

- specialised in verifying random access operations

Tree view

$$\text{tree\_rep}_{\text{tree}}(p, tr) \triangleq \cdots$$

- specialised in verifying structure changing operations

Strategy to C1: Dual Views

Array view
(defined before)

$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \triangleq \exists \ell,\ \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell)$$

- specialised in verifying random access operations

mutually derivable

switch as needed

Tree view

$$\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) \triangleq \cdots$$

- specialised in verifying structure changing operations

Tree View Predicate

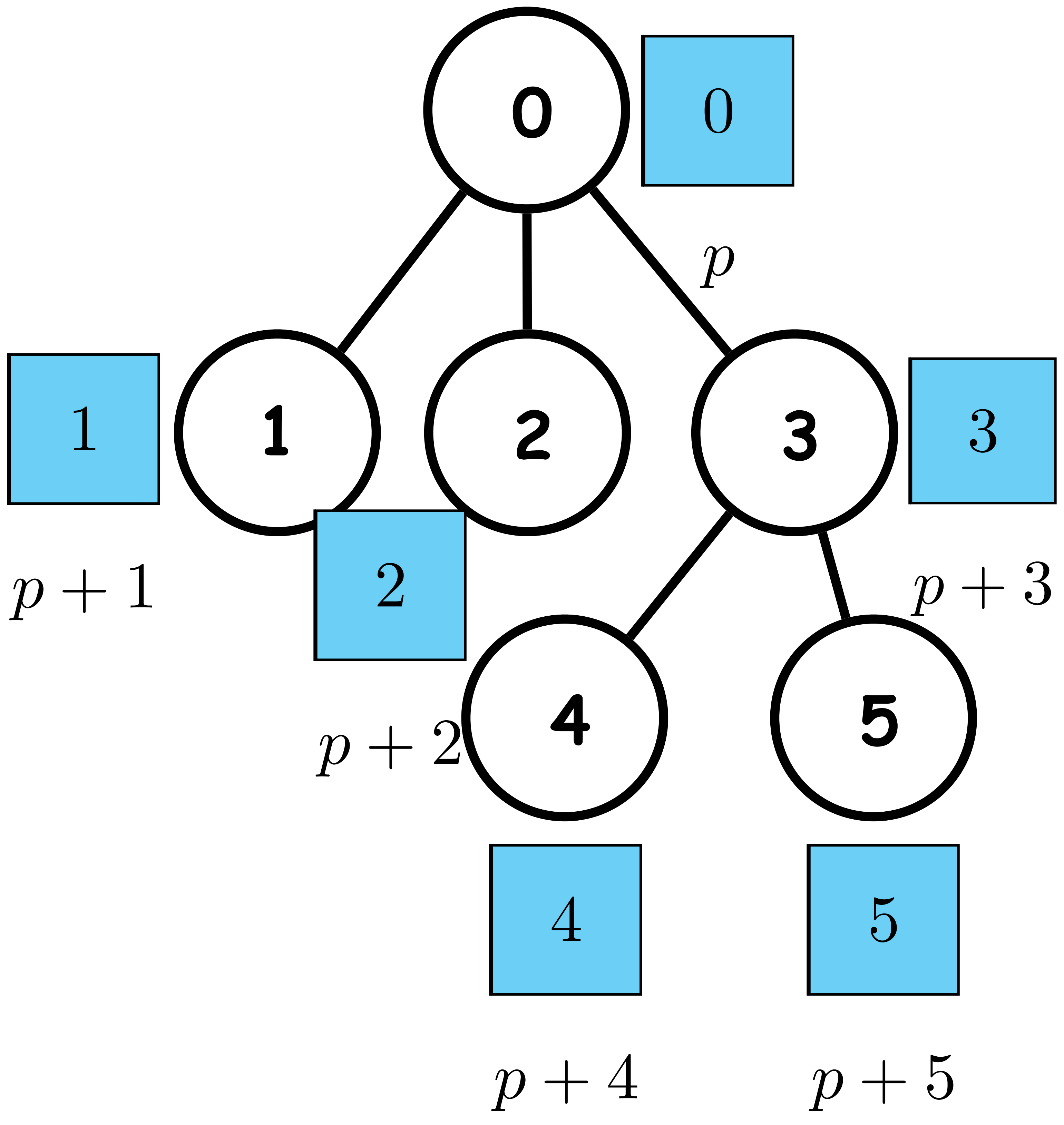
- tree_rep$_{\text{tree}}(p, tr)$: recursively defined over tr

Tree View Predicate

- $\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)$: recursively defined over tr
  - Describes each element of the array with the singleton $(\cdot \mapsto \cdot)$ heap predicate

Tree View Predicate

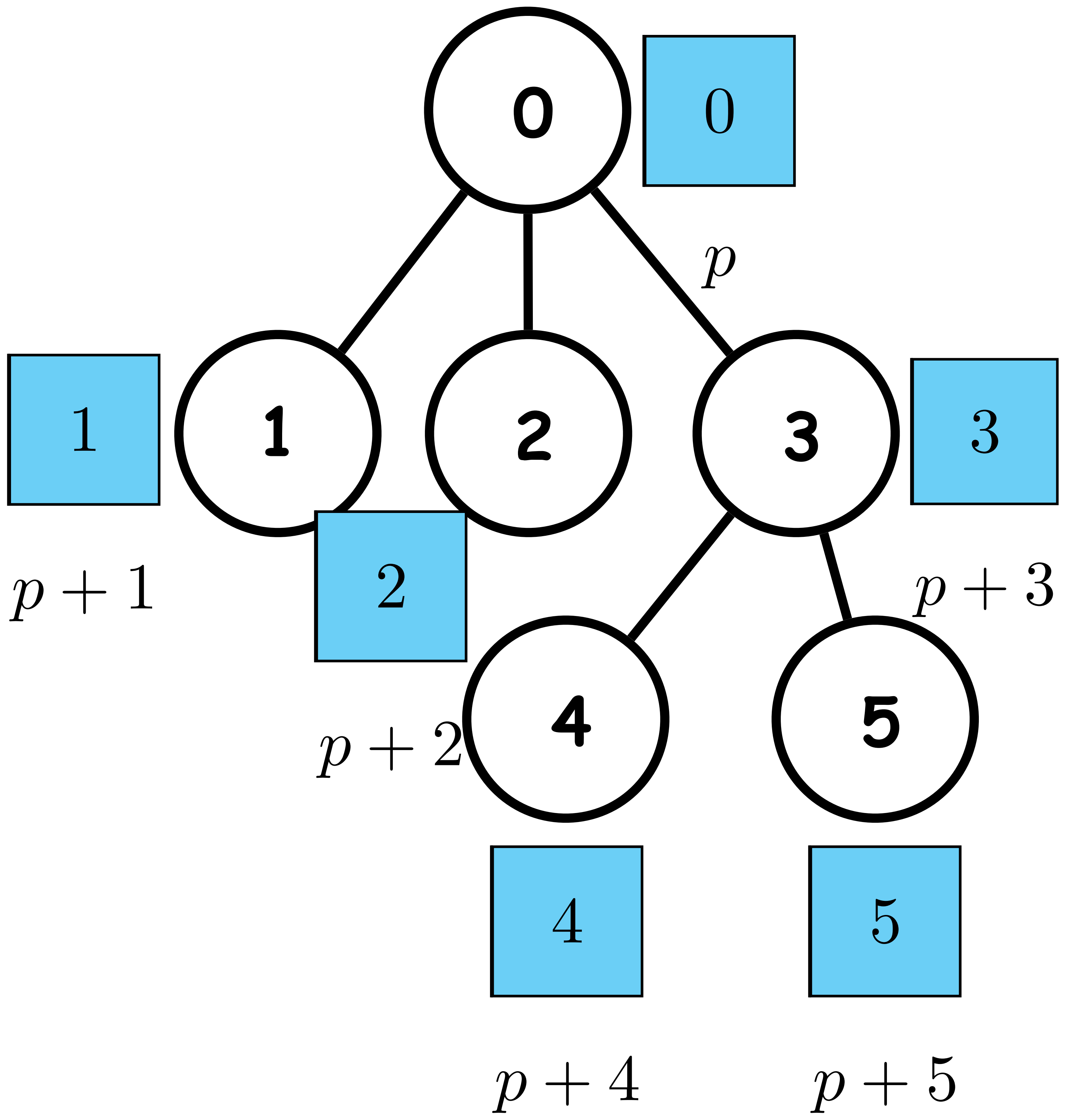
- tree_rep$_{\text{tree}}(p, tr)$: recursively defined over tr
  - Describes each element of the array with the singleton $(\cdot \mapsto \cdot)$ heap predicate



Tree View Predicate

- tree_rep$_{\text{tree}}(p, tr)$: recursively defined over tr
  - Describes each element of the array with the singleton $(\cdot \mapsto \cdot)$ heap predicate

$$\text{tree\_rep}_{\text{tree}}(p, tr) = \cdots * \\ p + 3 \mapsto \{\texttt{parent} = 0, \texttt{right\_sibling} = -1, \\ \texttt{first\_child} = 4\} * \cdots$$



Tree View Predicate (Cont'd)

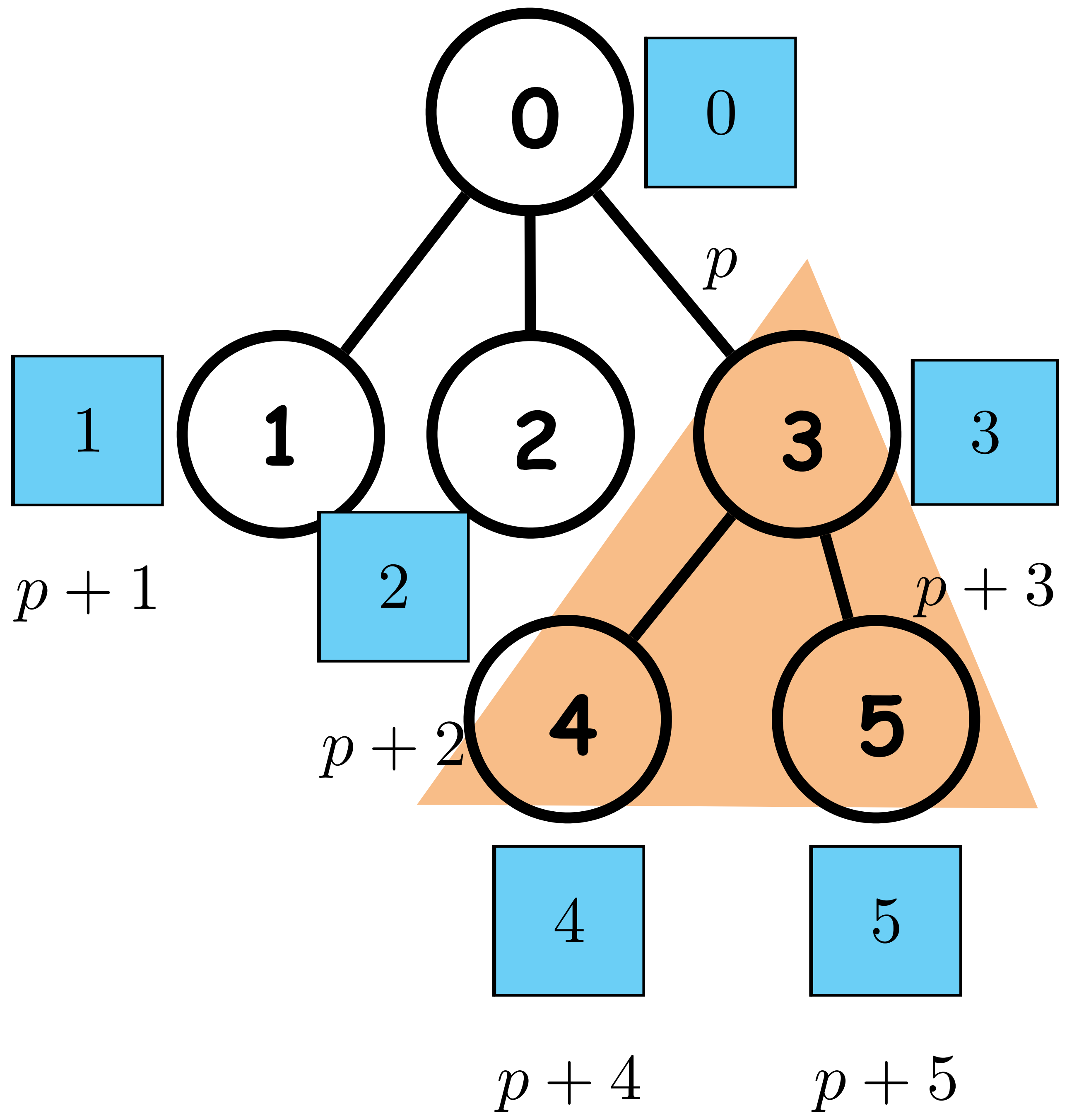
- Essentially a large separating conjunction

Tree View Predicate (Cont'd)

- Essentially a large separating conjunction
  - $\implies$ Easy to perform localised reasoning!

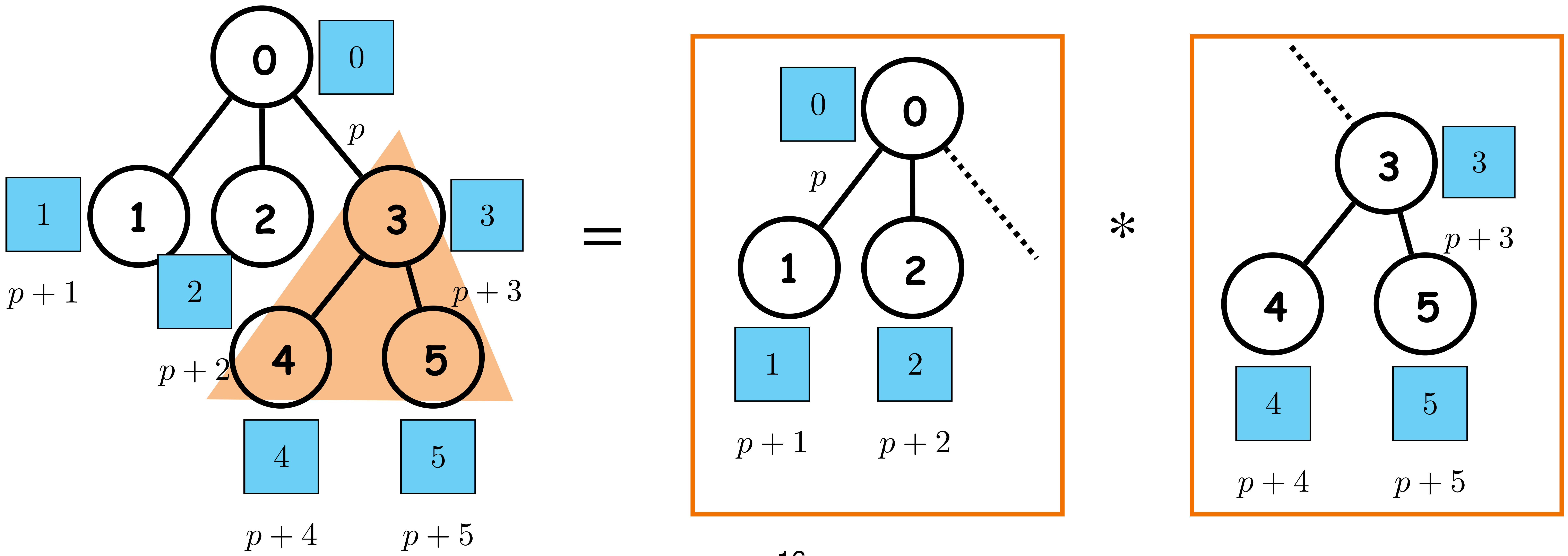
Tree View Predicate (Cont'd)

- Essentially a large separating conjunction
  - $\implies$ Easy to perform localised reasoning!
- E.g., "focusing" on a subtree



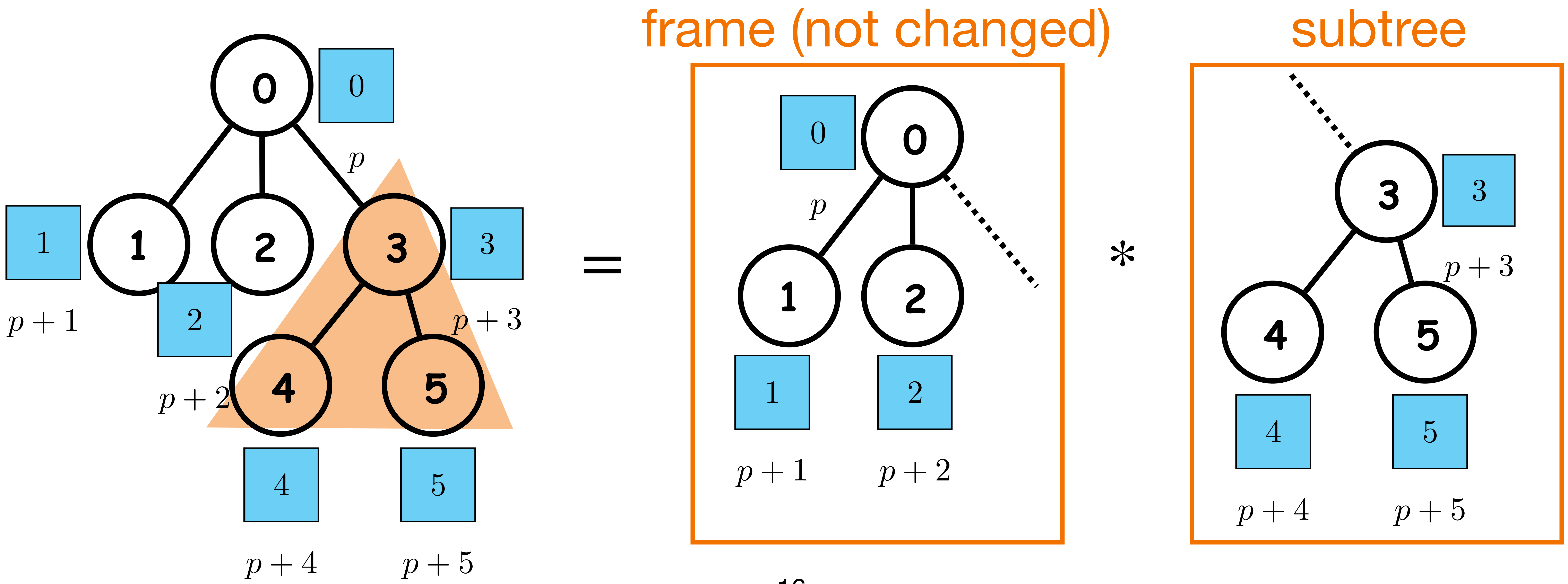
Tree View Predicate (Cont'd)

- Essentially a large separating conjunction
  - $\implies$ Easy to perform localised reasoning!
- E.g., "focusing" on a subtree



Tree View Predicate (Cont'd)

- Essentially a large separating conjunction
  - $\Longrightarrow$ Easy to perform localised reasoning!
- E.g., "focusing" on a subtree



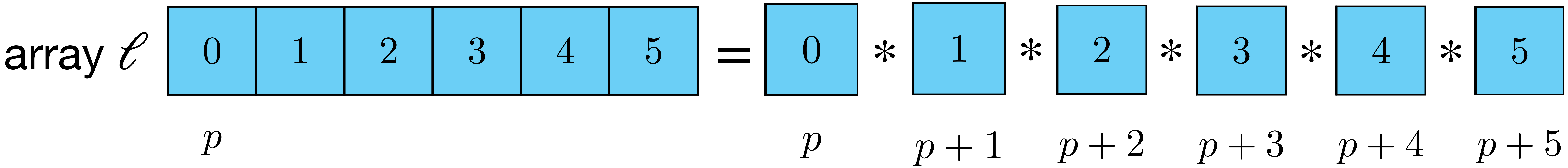
Mutual Derivability of Dual Views

Mutual Derivability of Dual Views

$$\mathrm{arr}(p, \ell) \triangleq \mathop{\Huge *}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$

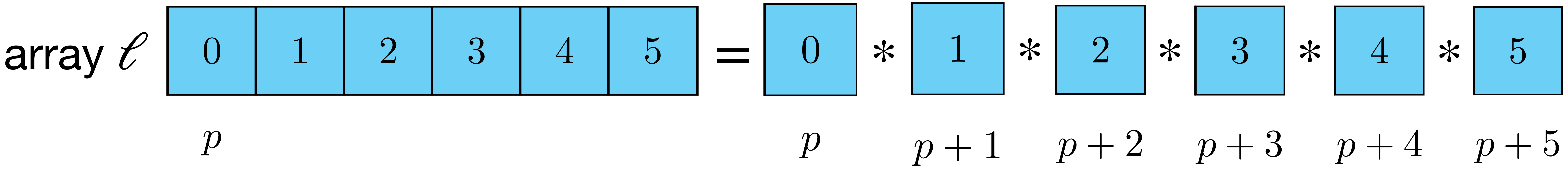
Mutual Derivability of Dual Views

$$\mathrm{arr}(p, \ell) \triangleq \mathop{\Huge\ast}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$



Mutual Derivability of Dual Views

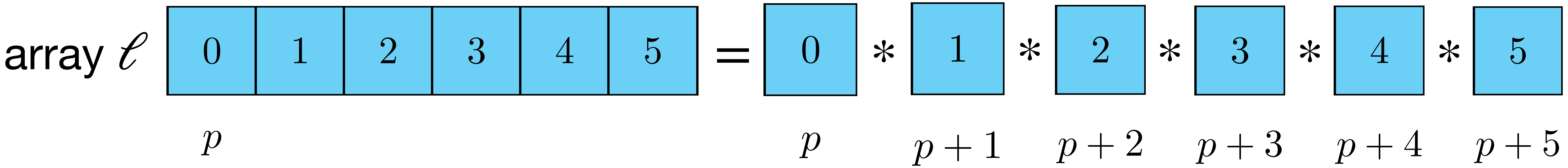
$$\mathsf{arr}(p, \ell) \triangleq \mathop{\Huge *}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$



$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) = \exists \ell, \lceil \mathsf{tree\_proj}(tr, \ell) \rceil$$
$$* \, \mathsf{arr}(p, \ell)$$

Mutual Derivability of Dual Views

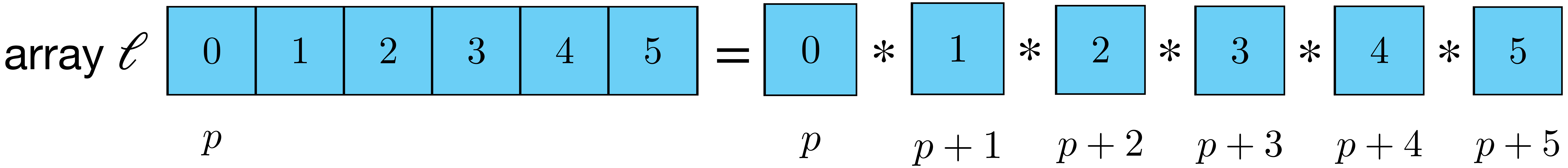
$$\mathsf{arr}(p, \ell) \triangleq \mathop{\Large *}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$



$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) = \exists \ell, \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell) \iff \exists \ell, \lceil \cdots \wedge \ell[3] = \ldots \wedge \cdots \rceil * (\cdots * \ p + 3 \mapsto \ell[3] * \cdots)$$

Mutual Derivability of Dual Views

$$\mathsf{arr}(p, \ell) \triangleq \mathop{\Huge\ast}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$

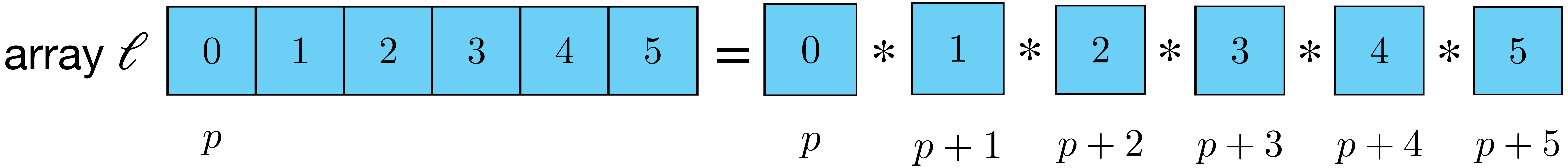


$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) = \exists \ell, \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell) \iff \exists \ell, \lceil \cdots \land \ell[3] = \ldots \land \cdots \rceil * (\cdots * p + 3 \mapsto \ell[3] * \cdots)$$

$$\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) = \cdots * p + 3 \mapsto \ldots * \cdots$$

Mutual Derivability of Dual Views

$$\mathsf{arr}(p, \ell) \triangleq \mathop{\Large *}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$



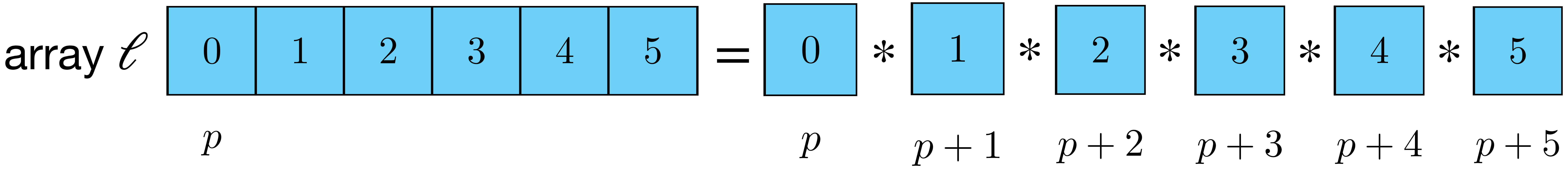
$$\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) = \exists \ell, \lceil \mathsf{tree\_proj}(tr, \ell) \rceil * \mathsf{arr}(p, \ell) \iff \exists \ell, \lceil \cdots \wedge \ell[3] = \ldots \wedge \cdots \rceil * (\cdots * p + 3 \mapsto \ell[3] * \cdots)$$

$$\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) = \cdots * p + 3 \mapsto \ldots * \cdots$$

mutually derivable!

Mutual Derivability of Dual Views

$$\mathsf{arr}(p, \ell) \triangleq \mathop{\Large\ast}_{i \in [0, |\ell|)} p + i \mapsto \ell[i]$$



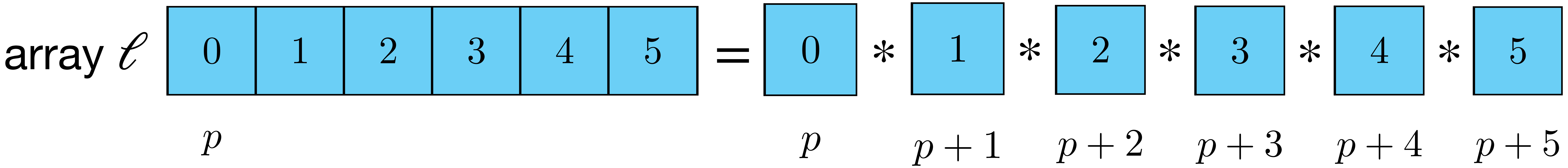
$$\text{tree\_rep}_{\text{arr}}(p, tr) = \exists \ell, \lceil \text{tree\_proj}(tr, \ell) \rceil * \text{arr}(p, \ell) \iff \exists \ell, \lceil \cdots \wedge \ell[3] = \ldots \wedge \cdots \rceil * (\cdots * p + 3 \mapsto \ell[3] * \cdots)$$

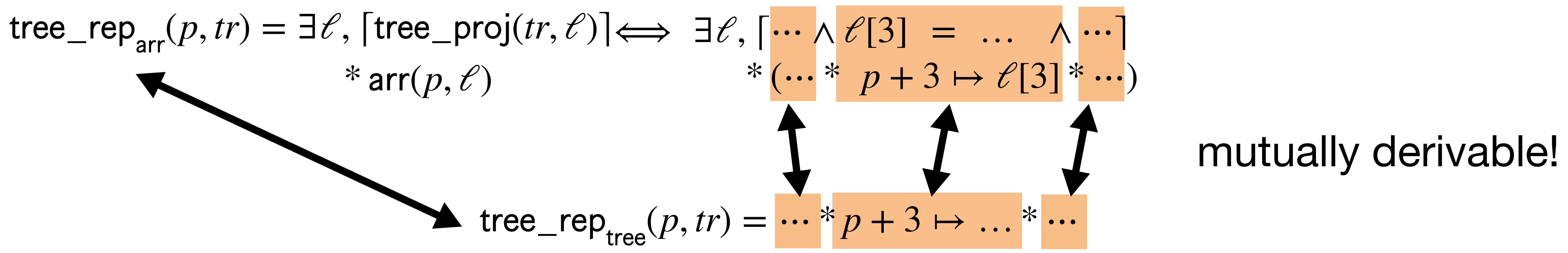
$$\text{tree\_rep}_{\text{tree}}(p, tr) = \cdots * p + 3 \mapsto \ldots * \cdots$$

mutually derivable!

Mutual Derivability of Dual Views

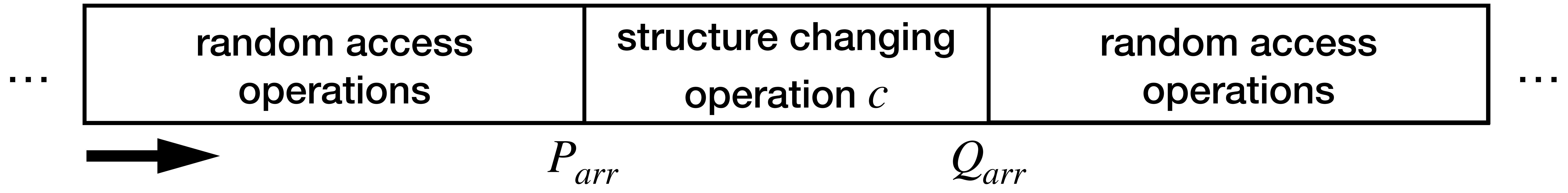
$$\mathrm{arr}(p,\ell) \triangleq \mathop{\bigstar}_{i \in [0,|\ell|)} p + i \mapsto \ell[i]$$



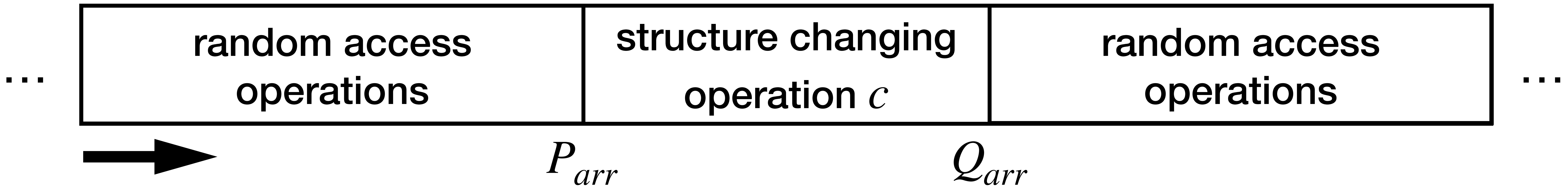


Dual Views in Action

Dual Views in Action

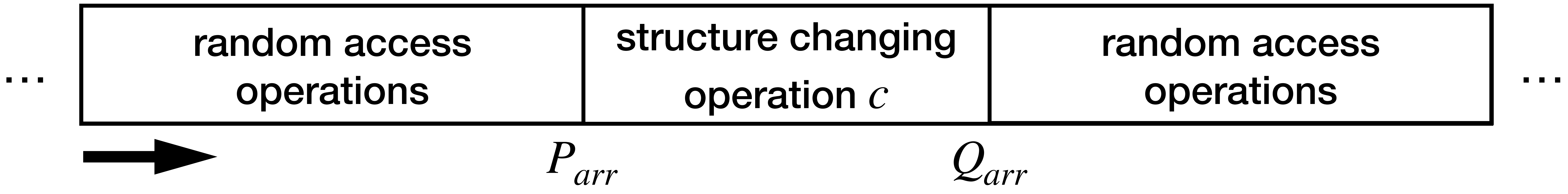


Dual Views in Action



$$\frac{P_{arr} \vdash P_{tree} \qquad \{P_{tree}\}\ c\ \{Q_{tree}\} \qquad Q_{tree} \vdash Q_{arr}}{\{P_{arr}\}\ c\ \{Q_{arr}\}}$$

Dual Views in Action



tree_rep$_{\text{arr}}(p, tr) \vdash$ tree_rep$_{\text{tree}}(p, tr)$

tree_rep$_{\text{tree}}(p, tr') \vdash$ tree_rep$_{\text{arr}}(p, tr')$

$$\frac{P_{arr} \vdash P_{tree} \qquad \{P_{tree}\}\ c\ \{Q_{tree}\} \qquad Q_{tree} \vdash Q_{arr}}{\{P_{arr}\}\ c\ \{Q_{arr}\}}$$

Dual Views in Action

… | random access operations | structure changing operation c | random access operations | …

P_{arr} $\quad$ Q_{arr}

$\text{tree\_rep}_{\text{arr}}(p, tr) \vdash \text{tree\_rep}_{\text{tree}}(p, tr)$

$\{\text{tree\_rep}_{\text{tree}}(p, tr)\}$
c
$\{\text{tree\_rep}_{\text{tree}}(p, tr')\}$

$\text{tree\_rep}_{\text{tree}}(p, tr') \vdash \text{tree\_rep}_{\text{arr}}(p, tr')$

$$\frac{P_{arr} \vdash P_{tree} \qquad \{P_{tree}\}\ c\ \{Q_{tree}\} \qquad Q_{tree} \vdash Q_{arr}}{\{P_{arr}\}\ c\ \{Q_{arr}\}}$$

$\{\text{tree\_rep}_{\text{arr}}(p, tr)\}$
c
$\{\text{tree\_rep}_{\text{arr}}(p, tr')\}$

Strategy to C2: Tree Splitting

- Key ideas:

Strategy to C2: Tree Splitting

- Key ideas:
  - Exploit the correspondence between a node and the path from the root to it

Strategy to C2: Tree Splitting

- Key ideas:
  - Exploit the correspondence between a node and the path from the root to it
  - The stack content and the visited part: functions of the stack top node

Strategy to C2: Tree Splitting

- Key ideas:
  - Exploit the correspondence between a node and the path from the root to it
  - The stack content and the visited part: functions of the stack top node
    - The visited part: expressed as the right half of tree splitting

Vertical Split

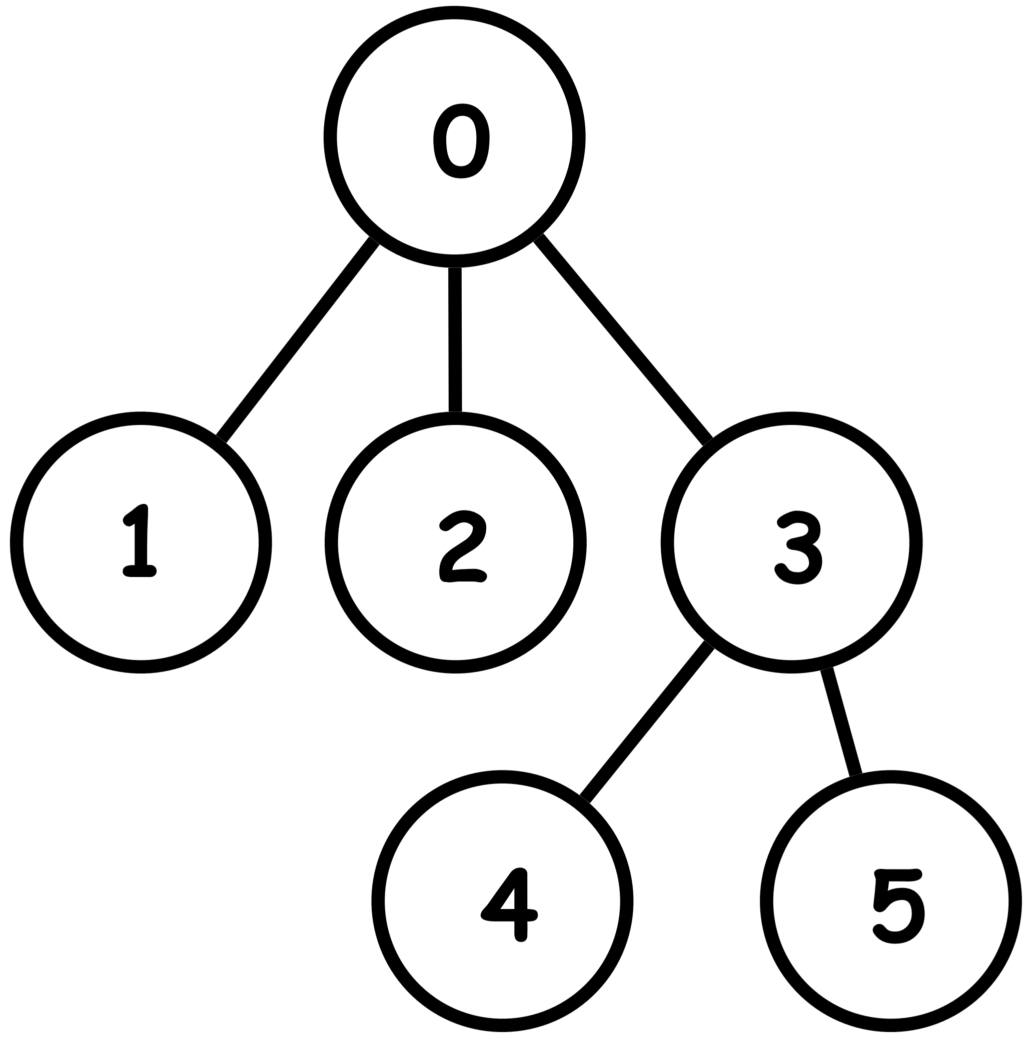
Reminder: children are pushed from left to right

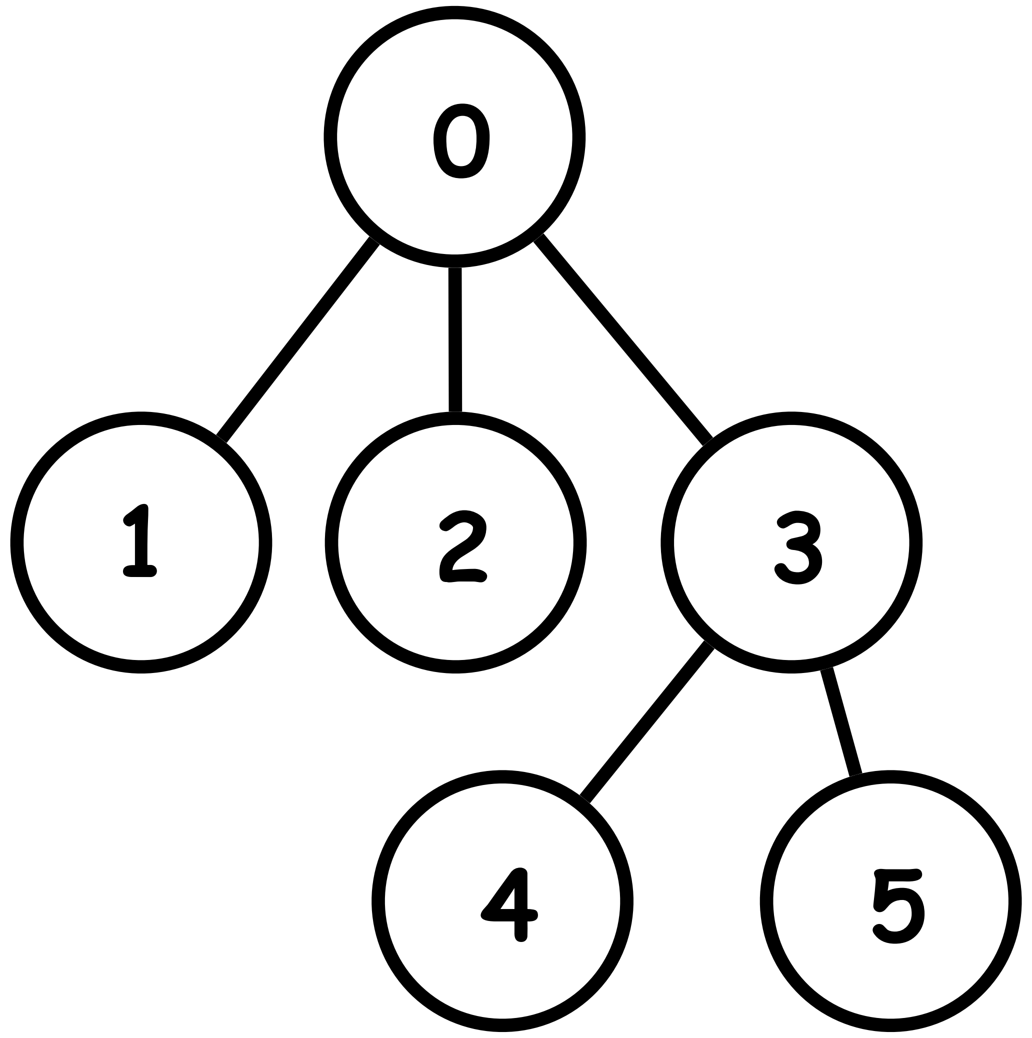
- Split the tree vertically along the path from the root to a node

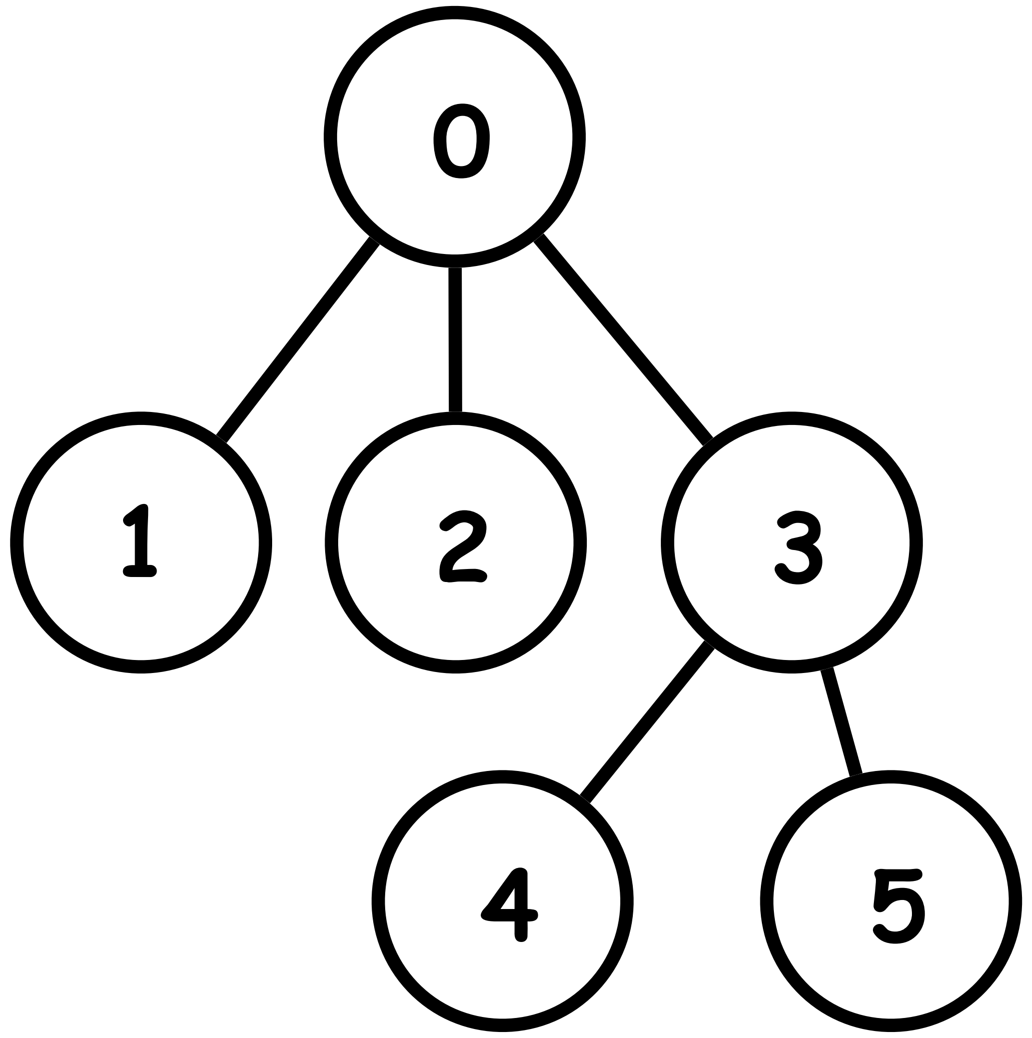
Vertical Split

Reminder: children are pushed from left to right

- Split the tree vertically along the path from the root to a node



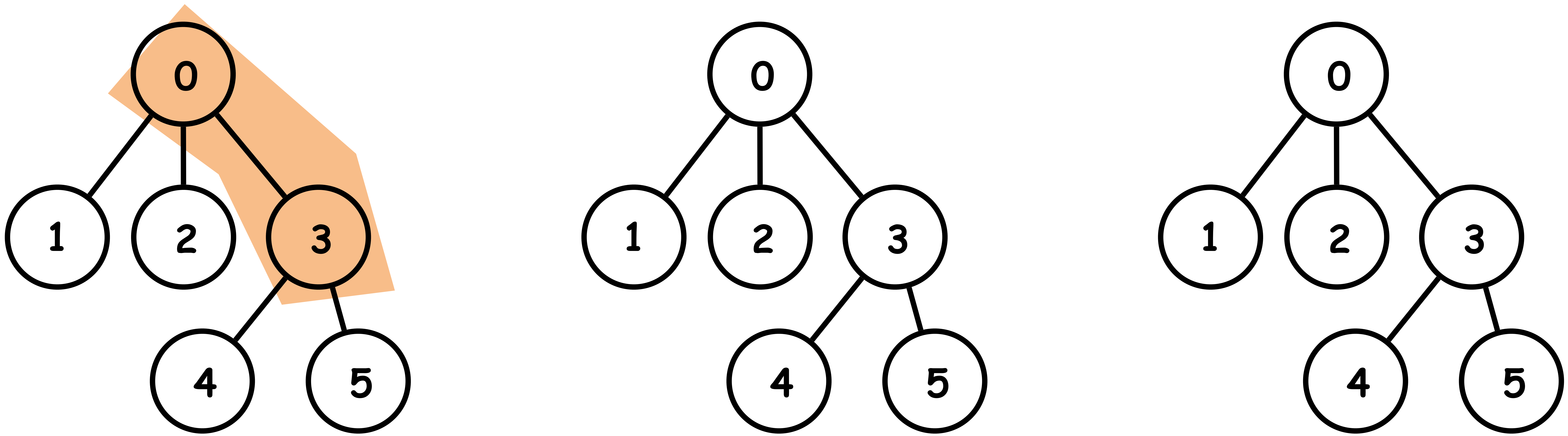




Vertical Split

Reminder: children are pushed from left to right

- Split the tree vertically along the path from the root to a node

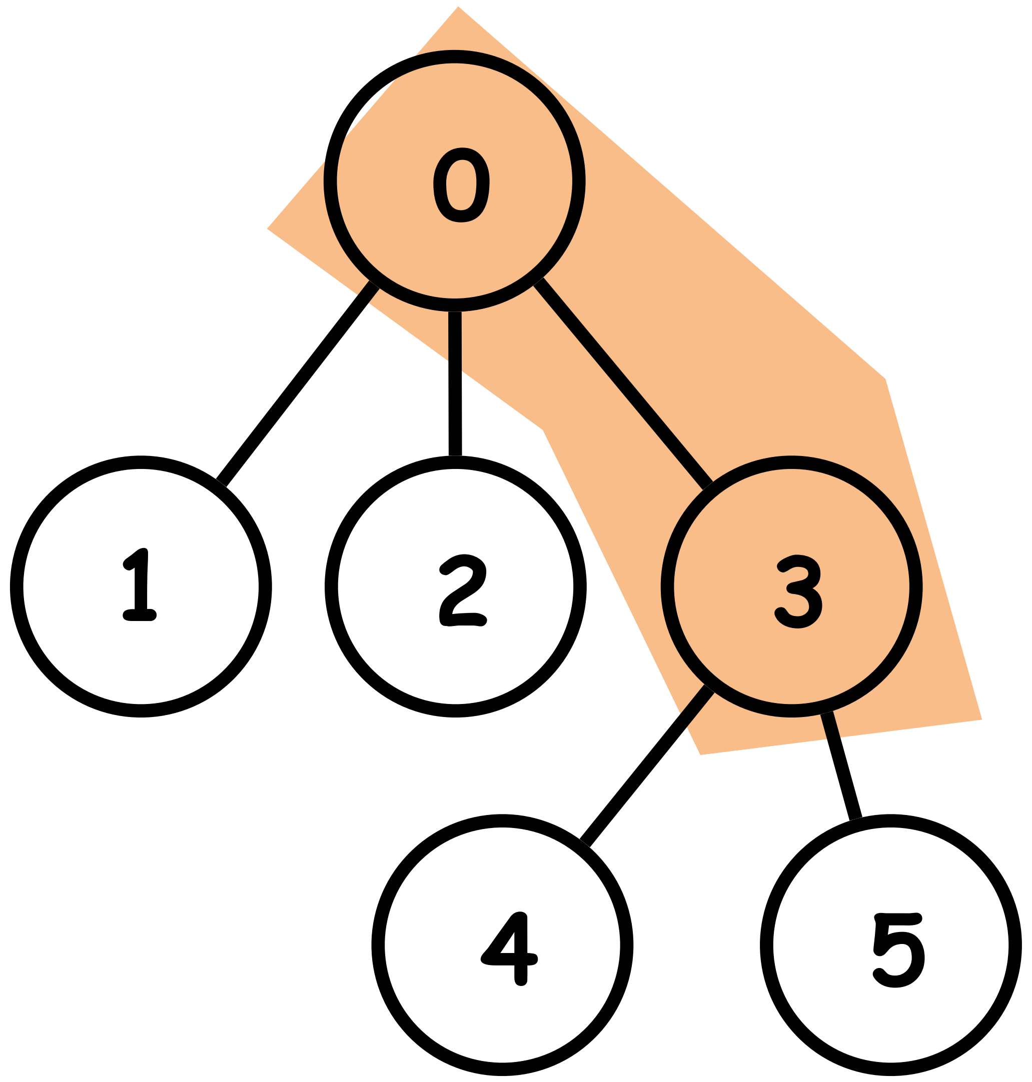


split by node 3

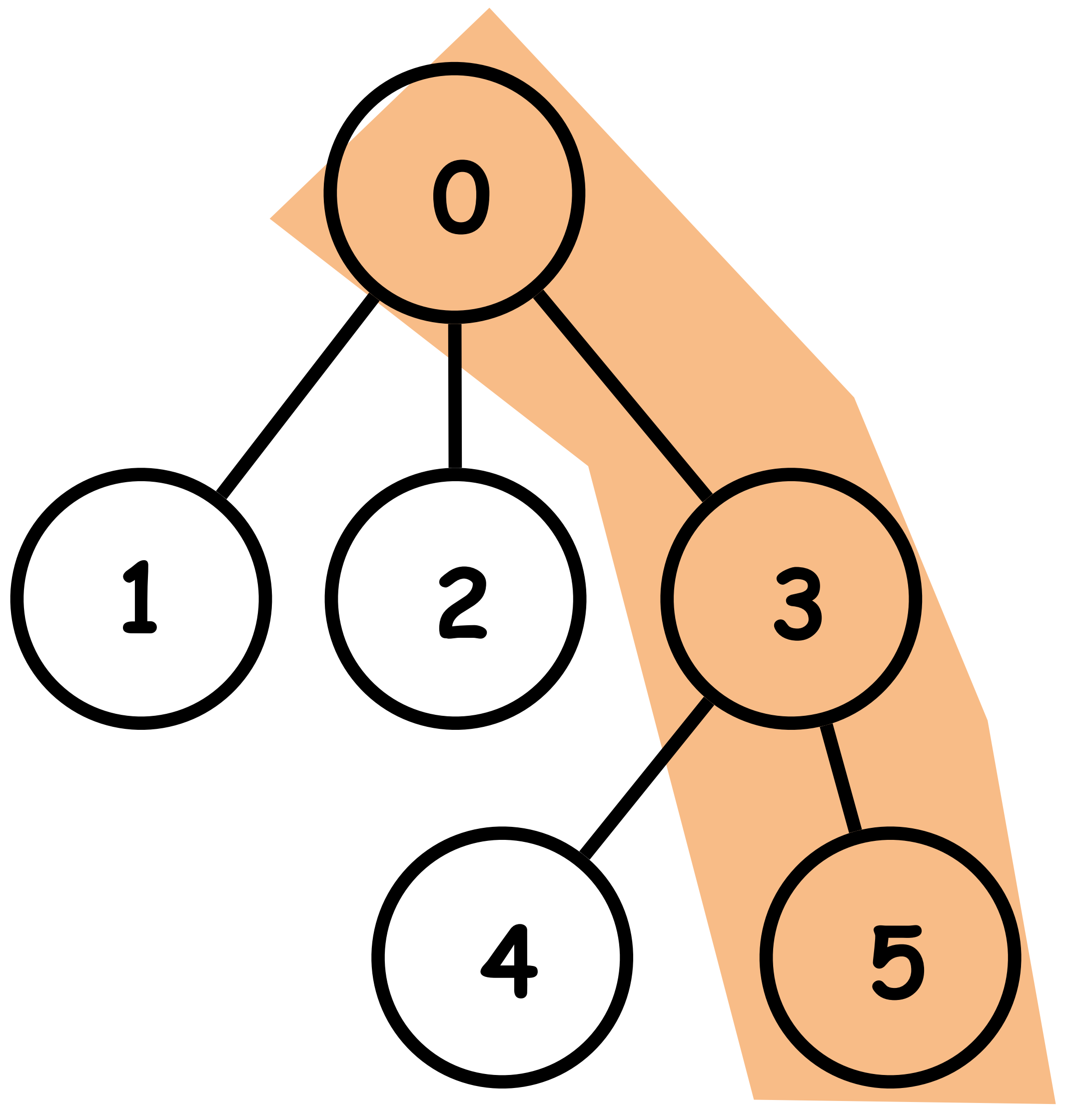
Vertical Split

Reminder: children are pushed from left to right

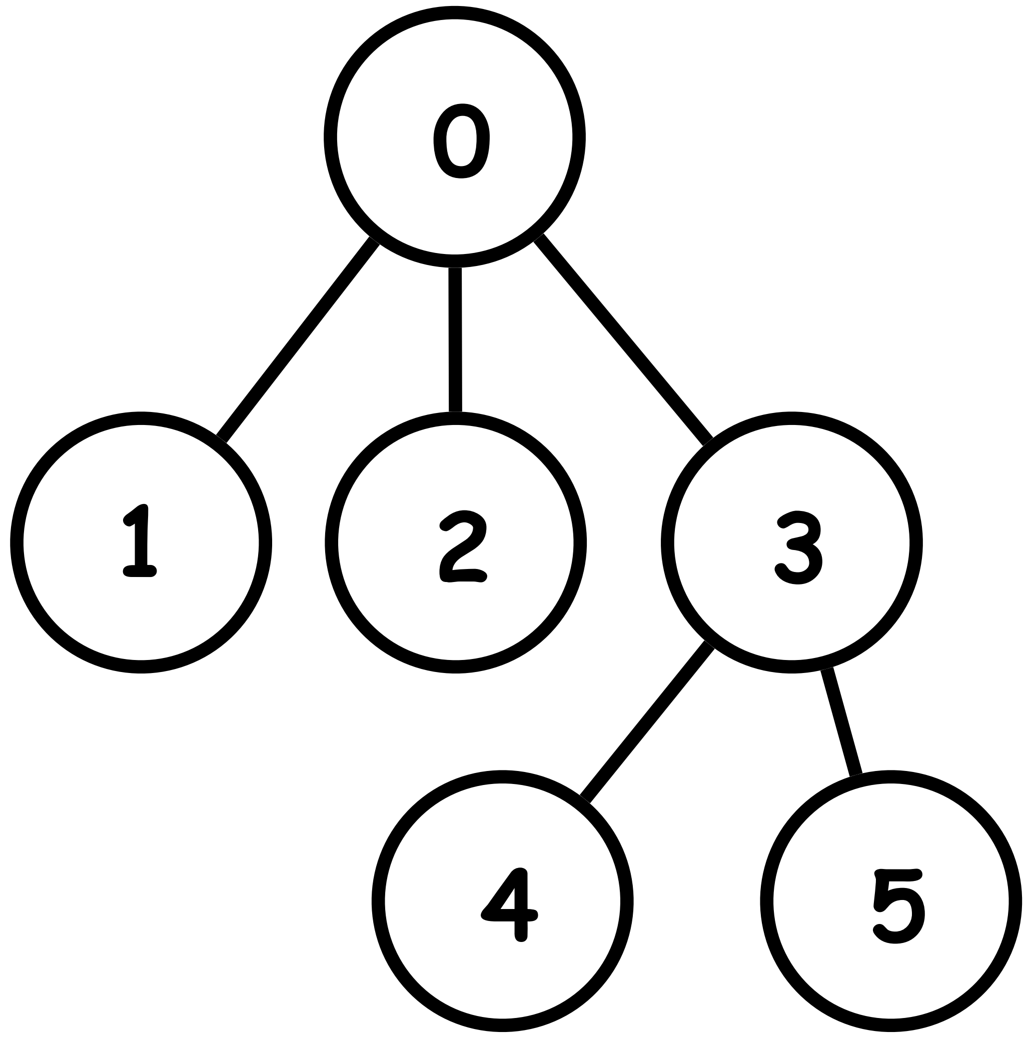
- Split the tree vertically along the path from the root to a node



split by node 3



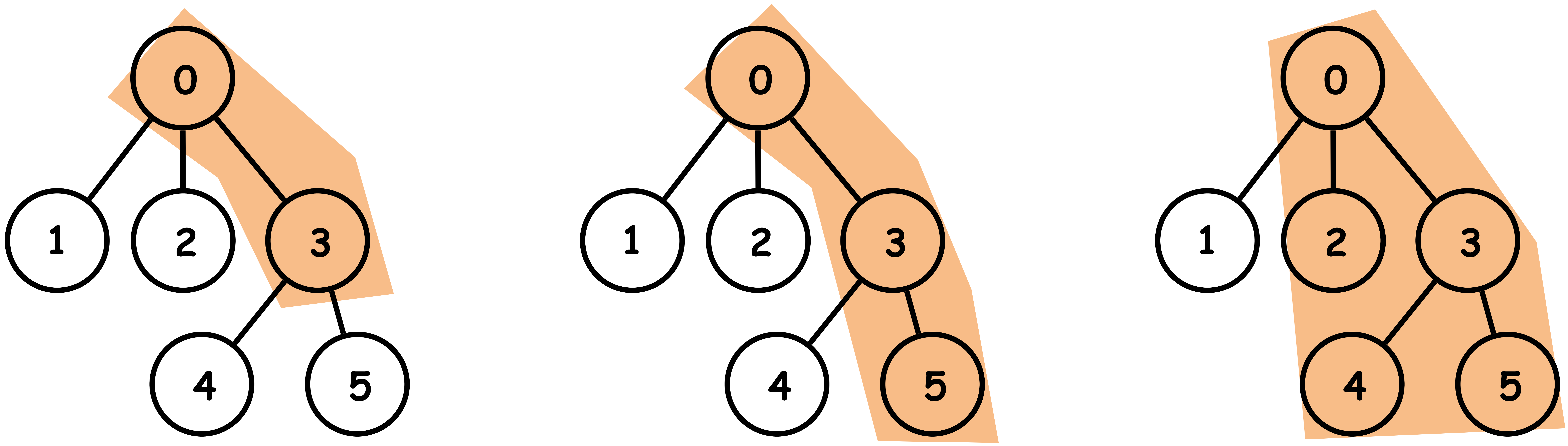
split by node 5



Vertical Split

Reminder: children are pushed from left to right

- Split the tree vertically along the path from the root to a node

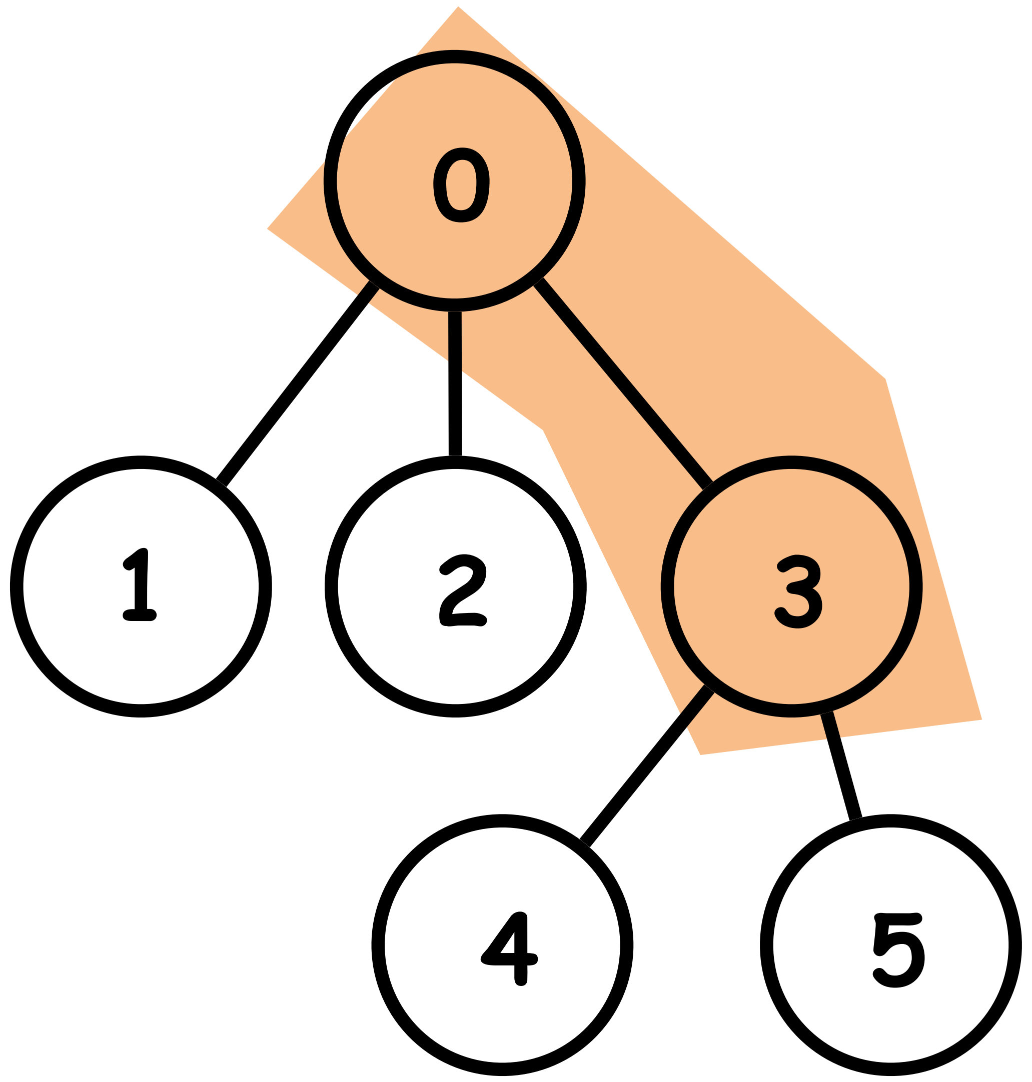


split by node 3 split by node 5 split by node 2

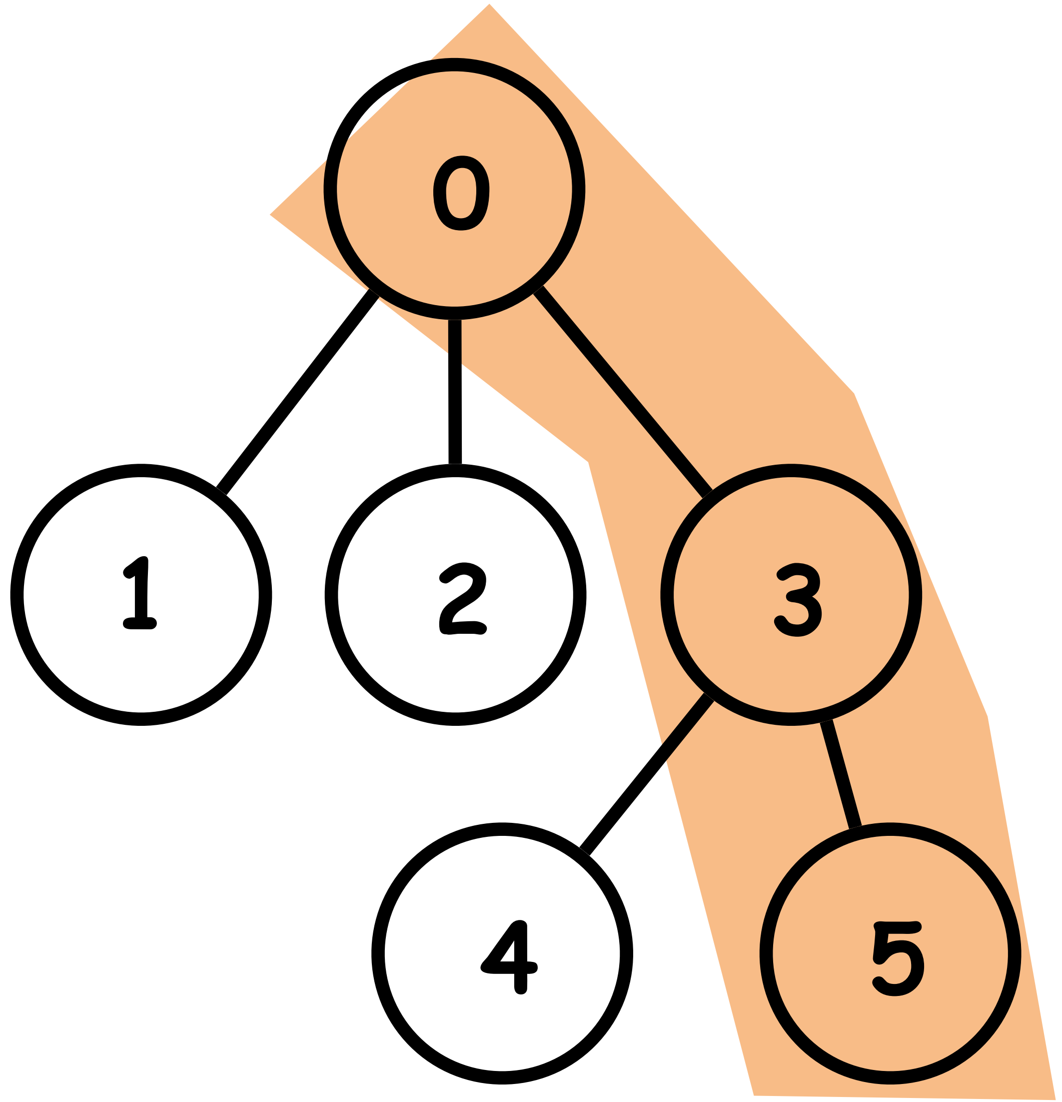
Vertical Split

Reminder: children are pushed from left to right

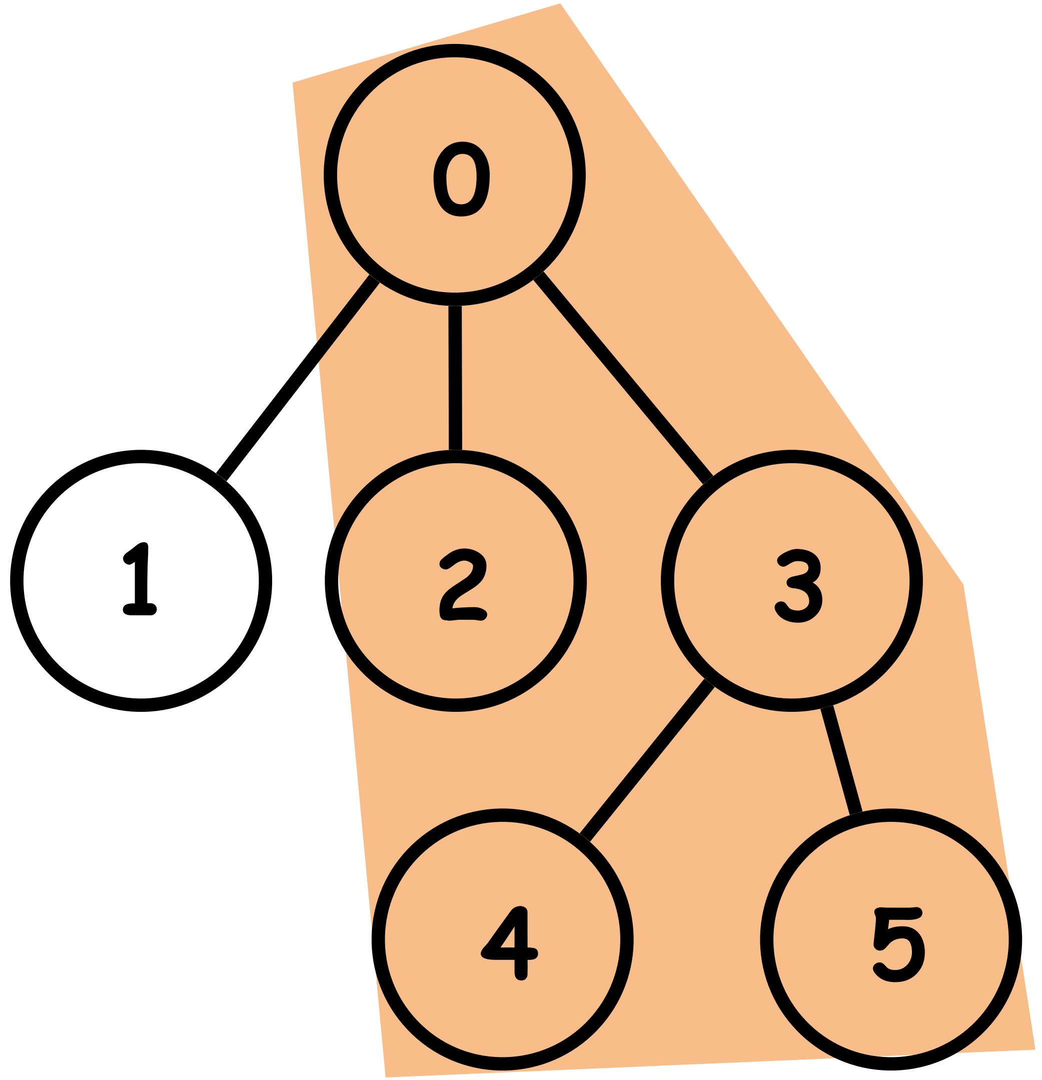
- Split the tree vertically along the path from the root to a node
  - The right half: including that node, its ancestors and the subtrees on their right



split by node 3



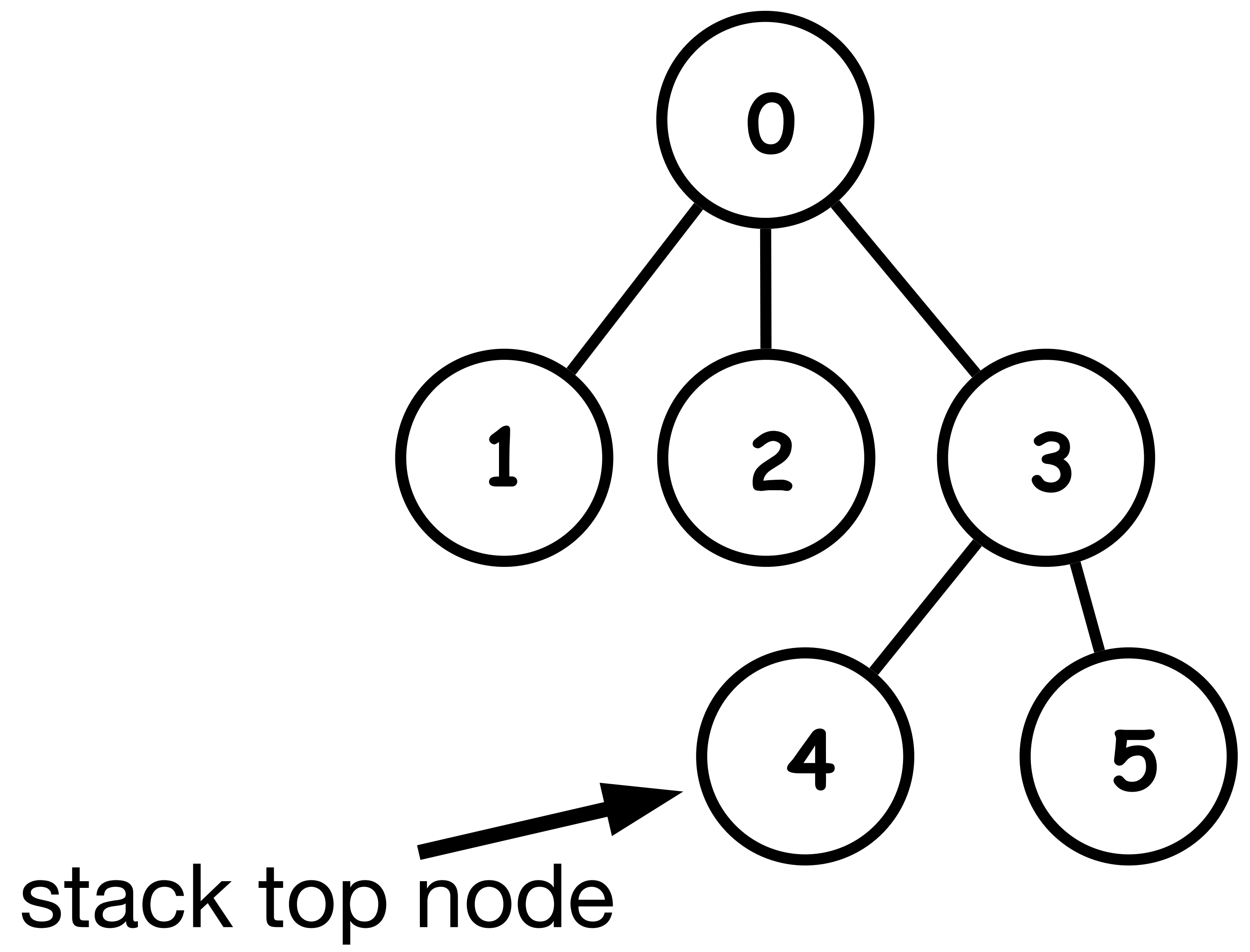
split by node 5



split by node 2

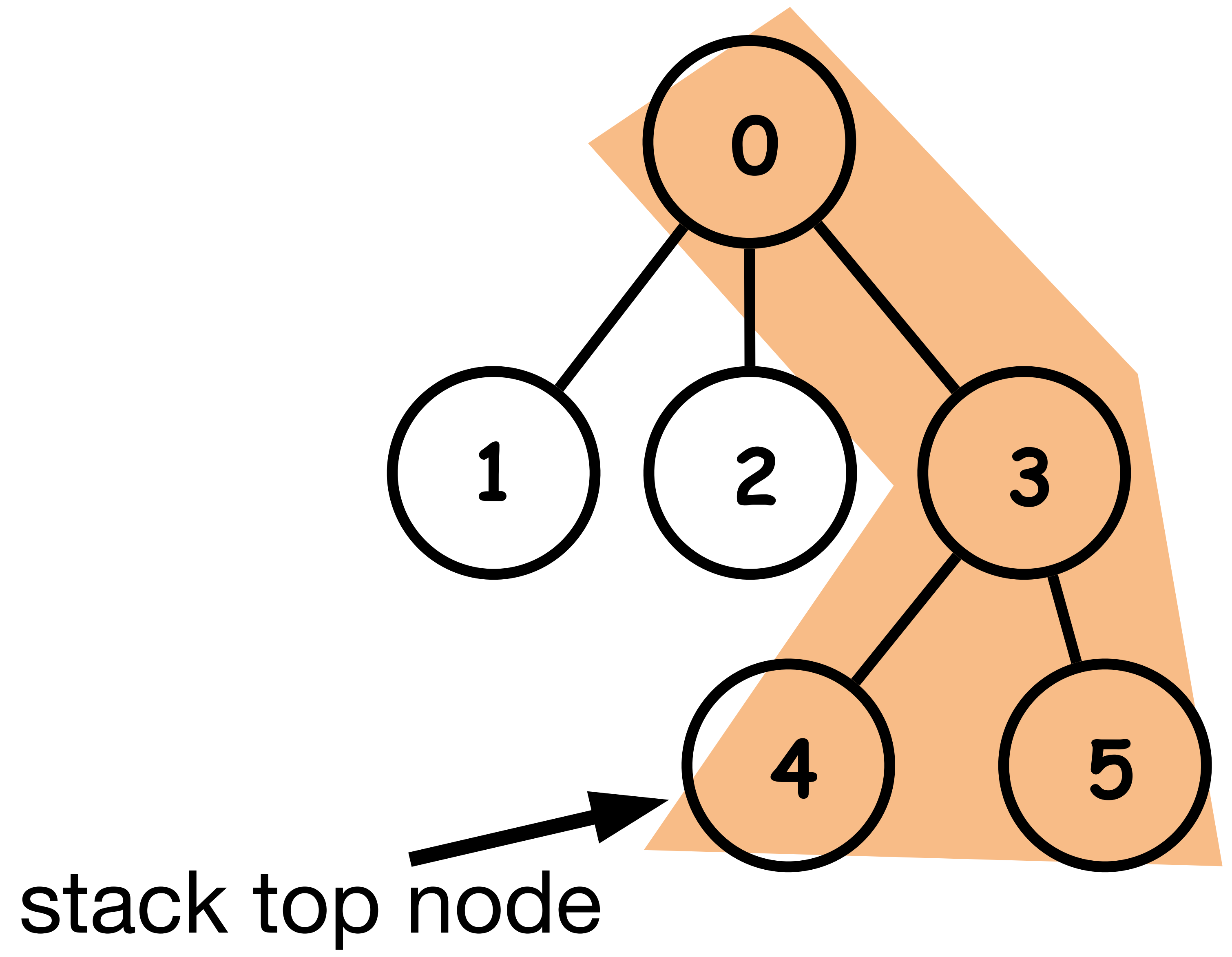
Vertical Split and Visited Part

```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



Vertical Split and Visited Part

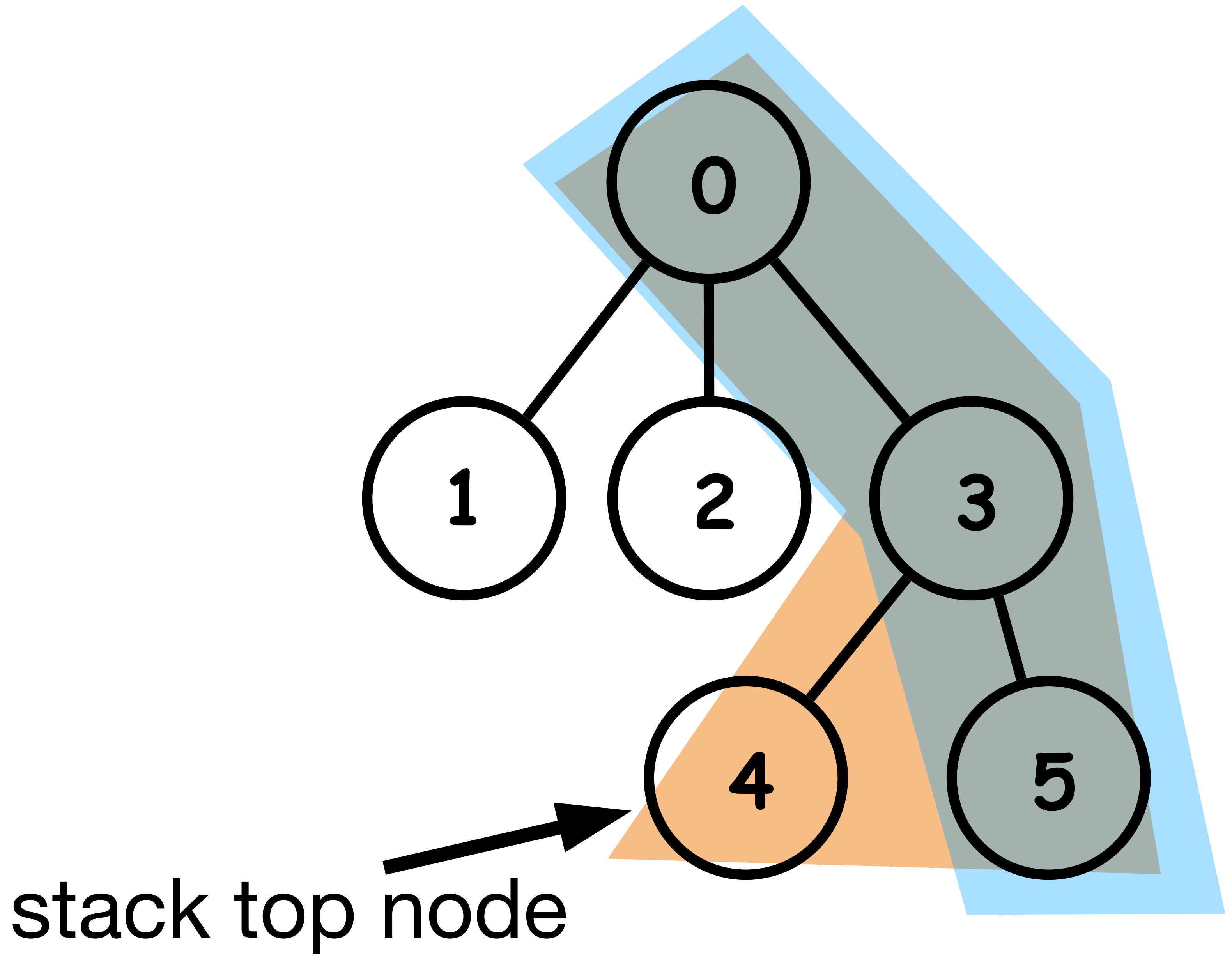
```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



- Post-iteration visited part = the right half of vertical split by stack top

Vertical Split and Visited Part

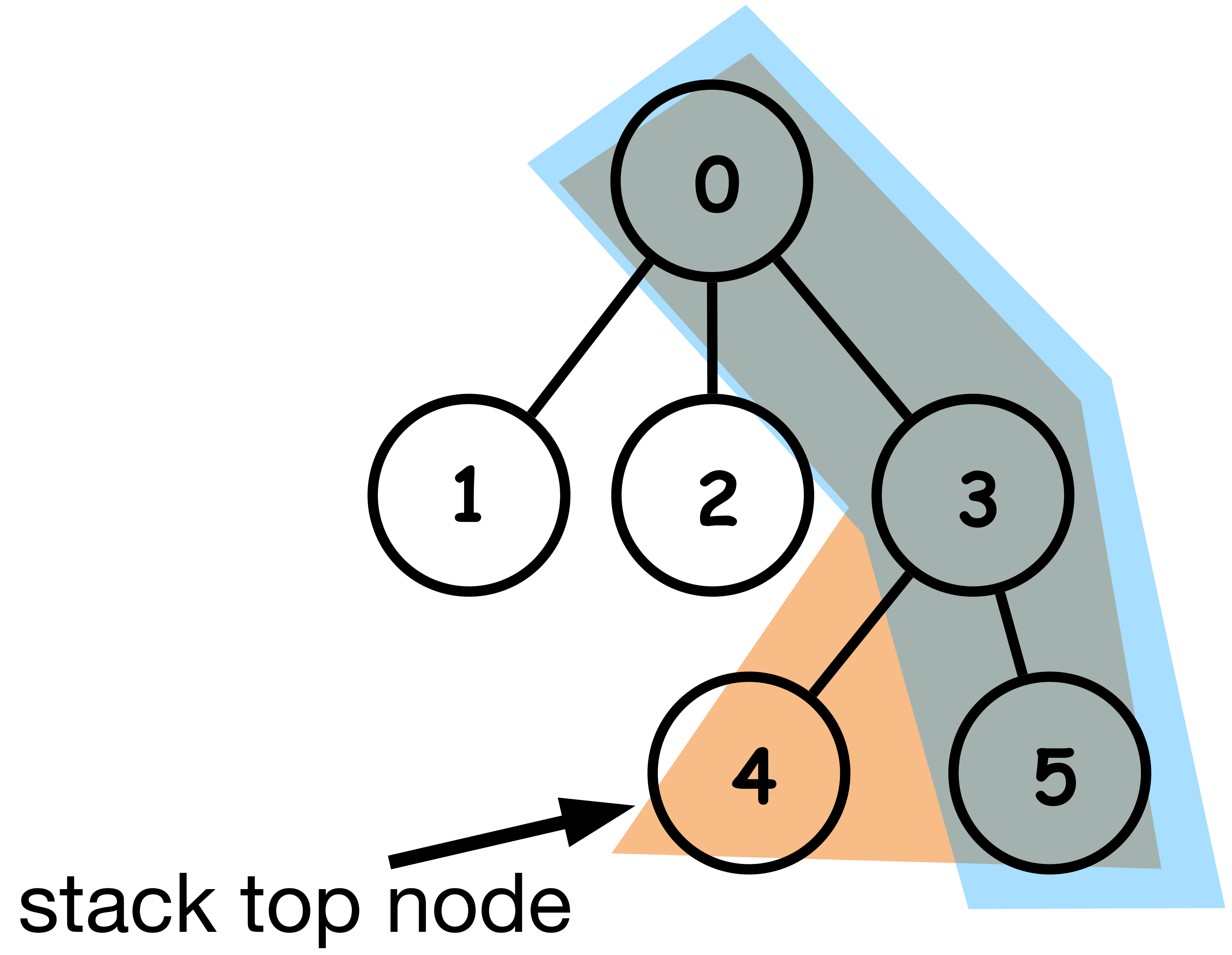
```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



- Post-iteration visited part = the right half of vertical split by stack top
- Pre-iteration visited part = post-iteration visited part **minus** stack top

Vertical Split and Visited Part

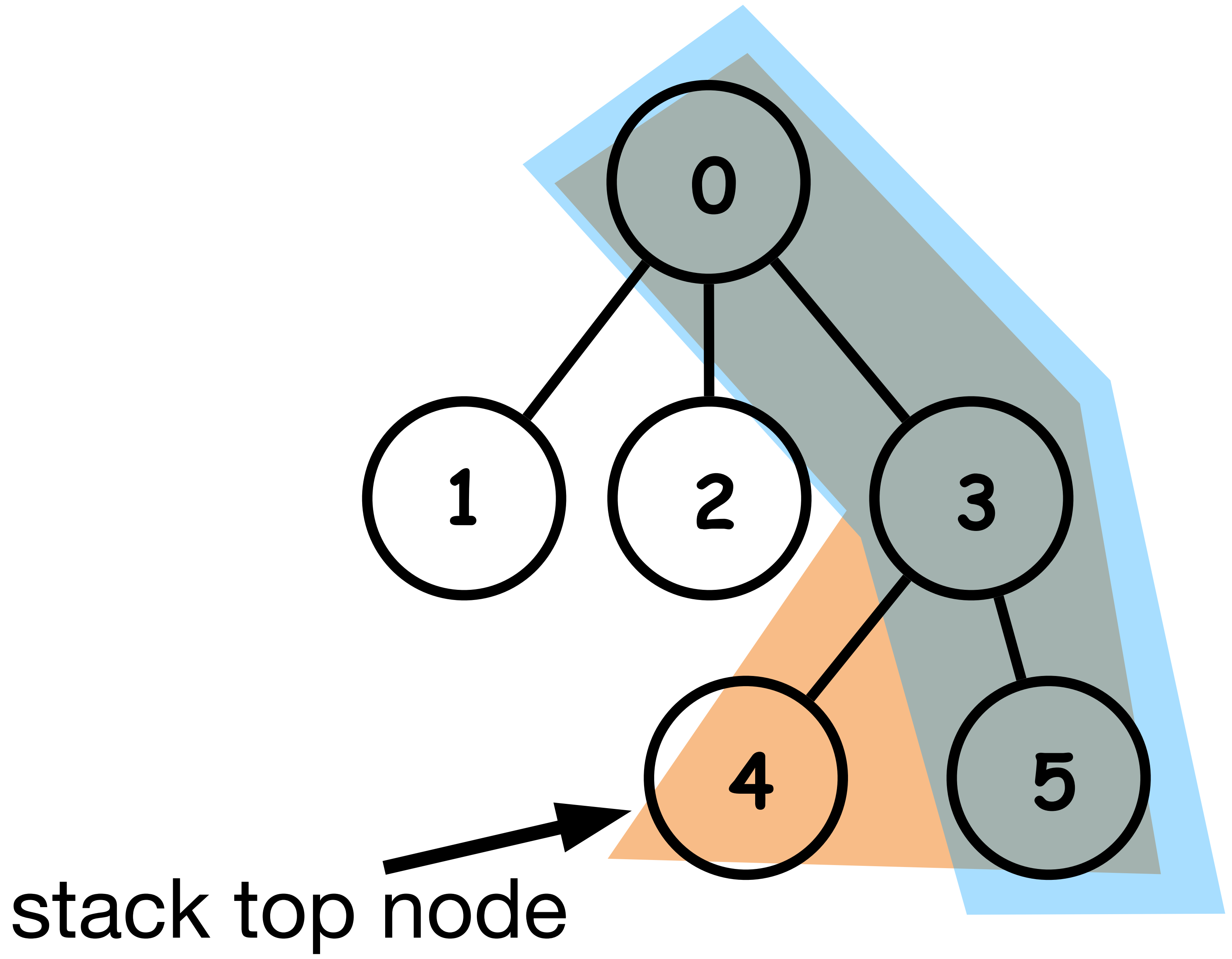
```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



- Post-iteration visited part = the right half of vertical split by stack top
- Pre-iteration visited part = post-iteration visited part **minus** stack top
  - Intuitively, = the right half of vertical split by the right sibling of stack top

Vertical Split and Visited Part

```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```

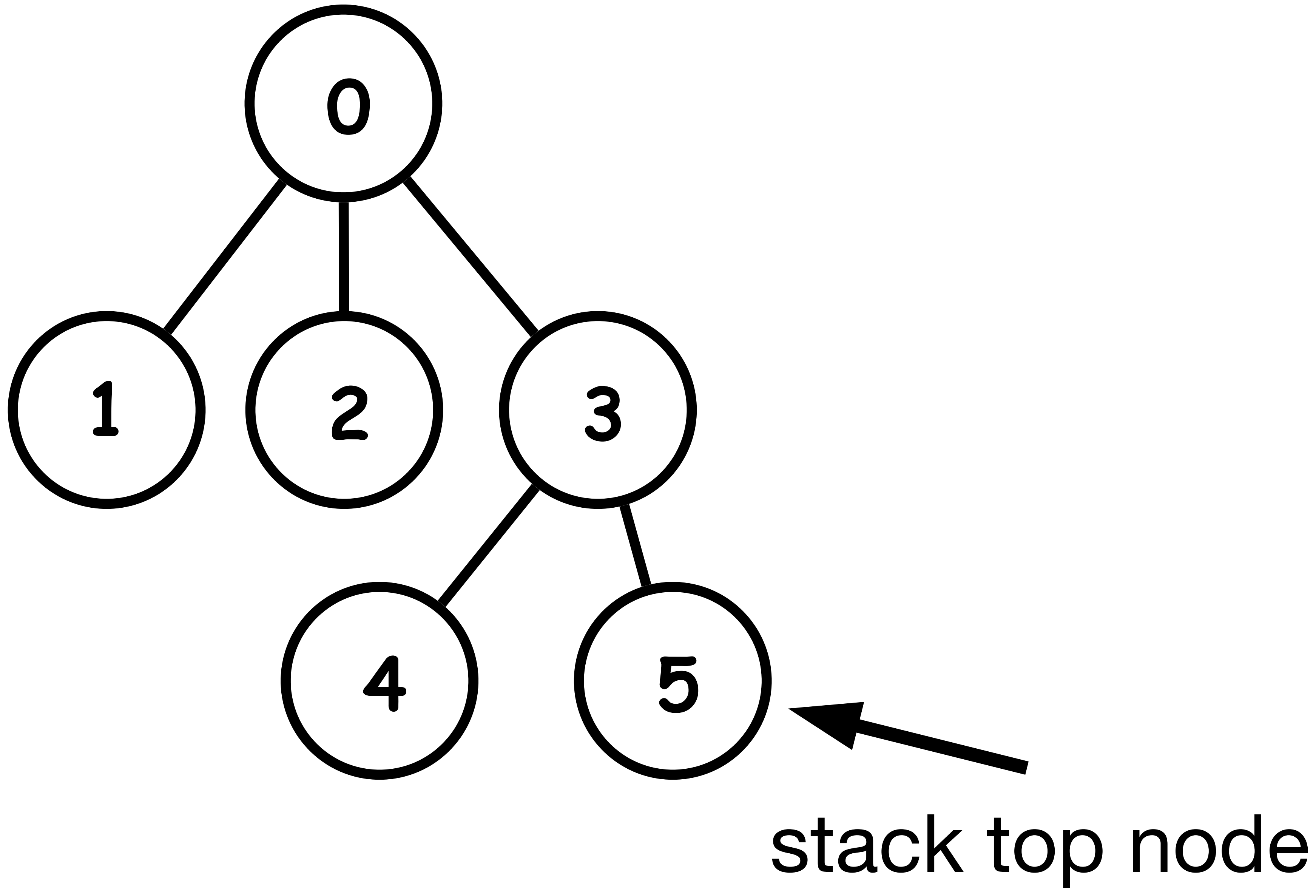


- Post-iteration visited part = the right half of vertical split by stack top
- Pre-iteration visited part = post-iteration visited part **minus** stack top
  - Intuitively, = the right half of vertical split by the right sibling of stack top

Check our paper for exact definition!

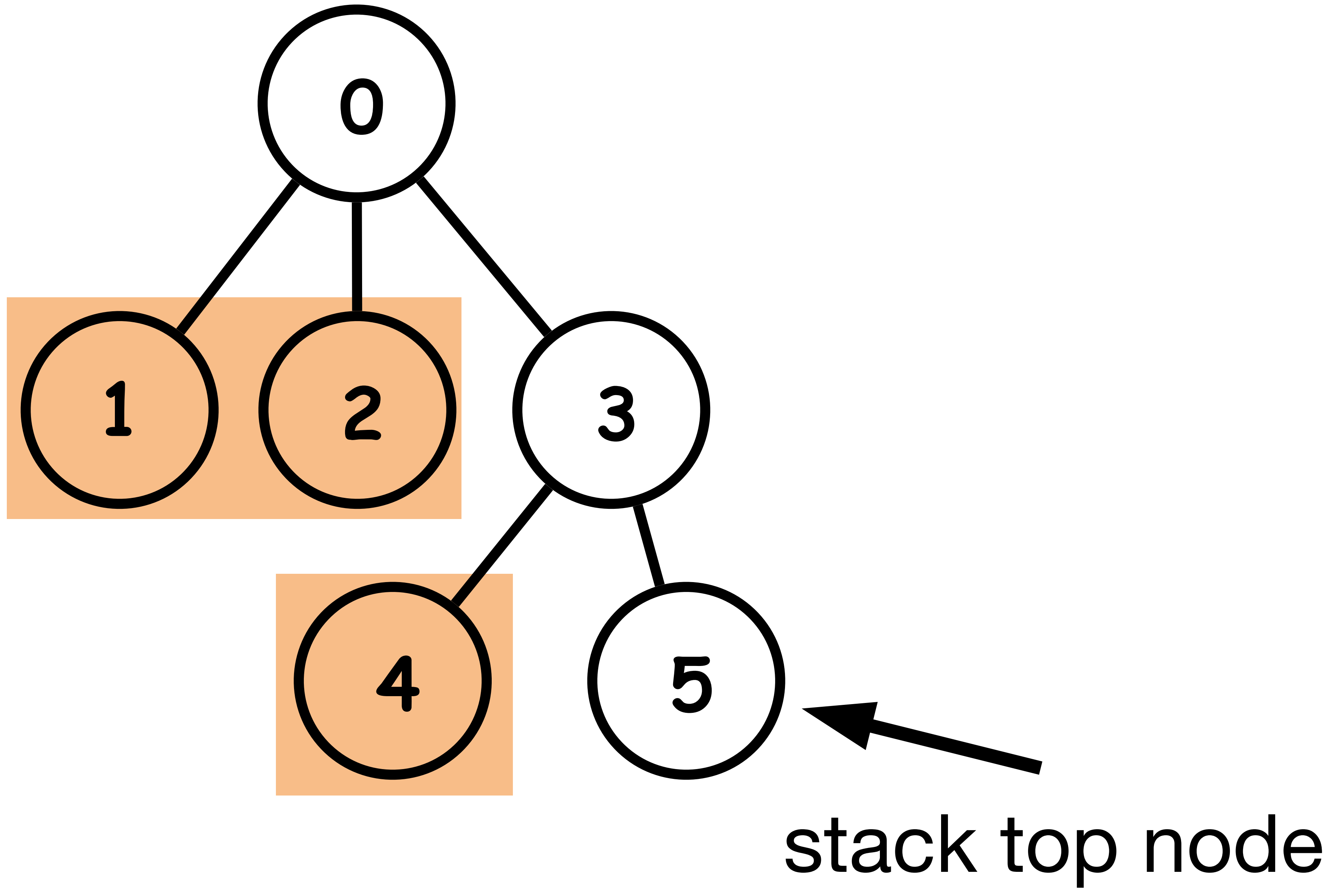
Retrieve the Stack Content

```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



Retrieve the Stack Content

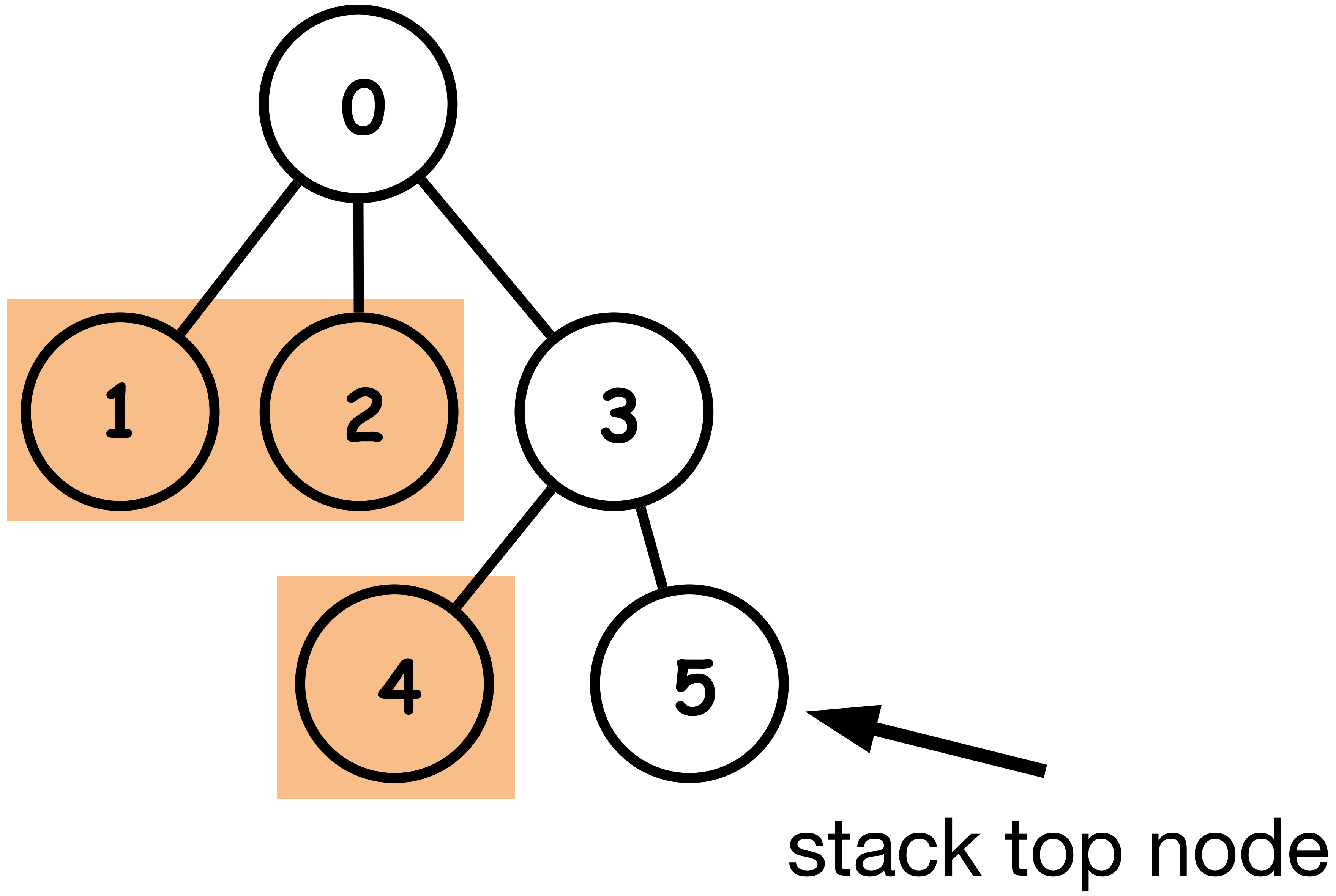
```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



- The stack before an iteration = nodes on the left of the ancestors of stack top

Retrieve the Stack Content

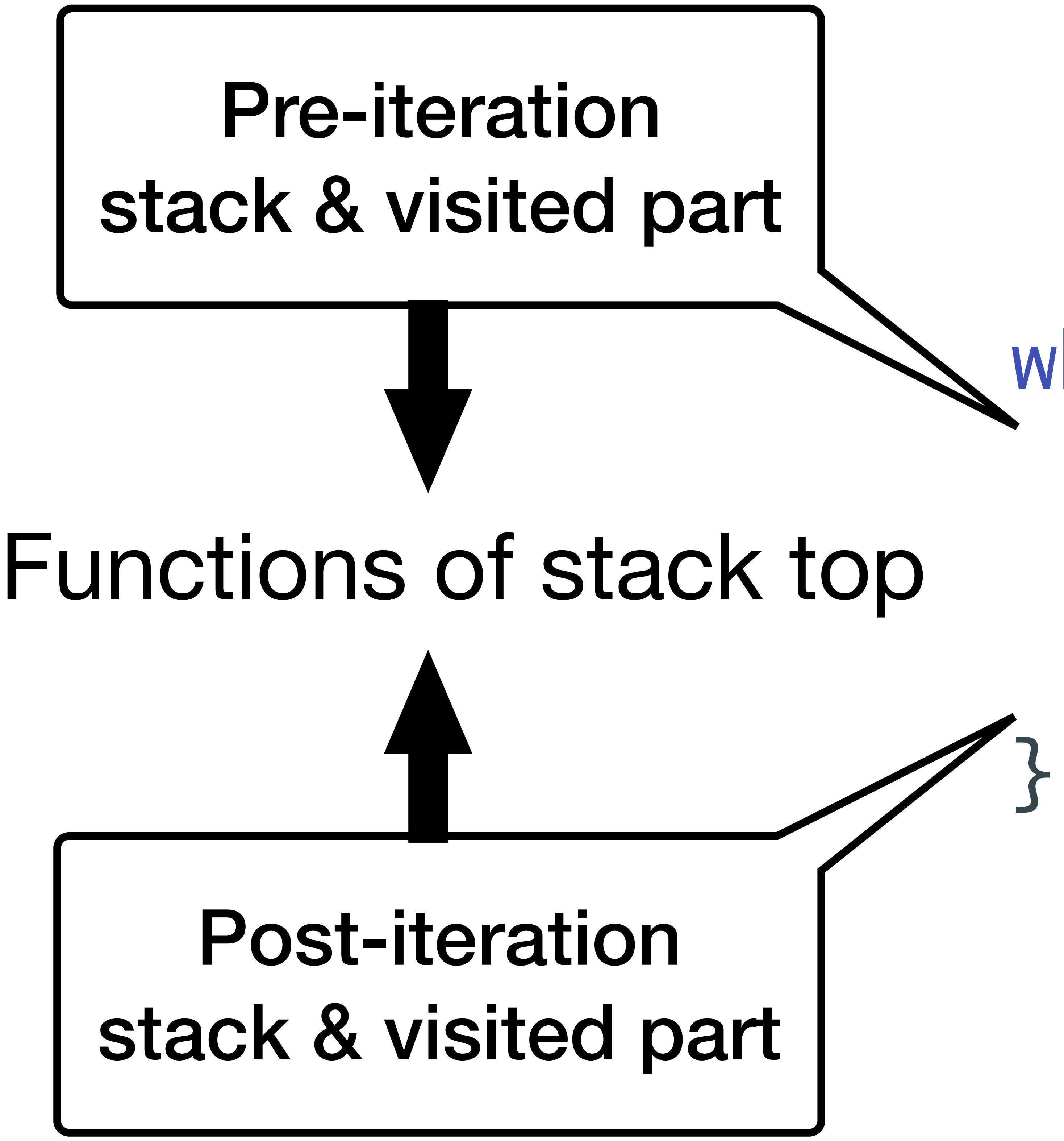
```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```



- The stack before an iteration = nodes on the left of the ancestors of stack top
- The stack after an iteration = the stack before an iteration **minus** stack top **plus** the children of stack top

Loop Invariant of Non-Recursive Traversal

- Sufficient to define by keeping track of the stack top node




```
while (/* stack not empty */) {
  int top = /* pop out stack top */;
  // ...
  // push the children of top
}
```

Roadmap

- Challenges
- Strategies
- Case study

(published in ASPLOS 2022)

A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Denmark
pavlogiannis@cs.au.dk

Hünkar Can Tunç
Aarhus University
Denmark
tunc@cs.au.dk

Mahesh Viswanathan
University of Illinois at Urbana-Champaign
USA
vmahesh@illinois.edu

ABSTRACT

Dynamic techniques are a scalable and effective way to analyze concurrent programs. Instead of analyzing all behaviors of a program, these techniques detect errors by focusing on a single program execution. Often a crucial step in these techniques is to define a causal ordering between events in the execution, which is then computed using *vector clocks*, a simple data structure that stores logical times of threads. The two basic operations of vector clocks, namely join and copy, require $\Theta(k)$ time, where k is the number of threads. Thus they are a computational bottleneck when k is large.

In this work, we introduce *tree clocks*, a new data structure that re-

KEYWORDS

concurrency, happens-before, vector clocks, dynamic analyses

ACM Reference Format:
Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507734

(published in ASPLOS 2022)

A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Denmark
pavlogiannis@cs.au.dk

Hünkar Can Tunç
Aarhus University
Denmark
tunc@cs.au.dk

Mahesh Viswanathan
University of Illinois at Urbana-Champaign
USA
vmahesh@illinois.edu

ABSTRACT

Dynamic techniques are a scalable and effective way to analyze concurrent programs. Instead of analyzing all behaviors of a program, these techniques detect errors by focusing on a single program execution. Often a crucial step in these techniques is to define a causal ordering between events in the execution, which is then computed using *vector clocks*, a simple data structure that stores logical times of threads. The two basic operations of vector clocks, namely join and copy, require $\Theta(k)$ time, where k is the number of threads. Thus they are a computational bottleneck when k is large.

In this work, we introduce *tree clocks*, a new data structure that re-

KEYWORDS

concurrency, happens-before, vector clocks, dynamic analyses

ACM Reference Format:
Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507734

Tree Clock

- Implementing logical clocks using generic trees

Tree Clock

- Implementing logical clocks using generic trees

$$(t_1 : 16,\ t_2 : 20,\ t_3 : 17,$$
$$t_4 : 23,\ t_5 : 4,\ t_6 : 15,\ t_7 : 11)$$

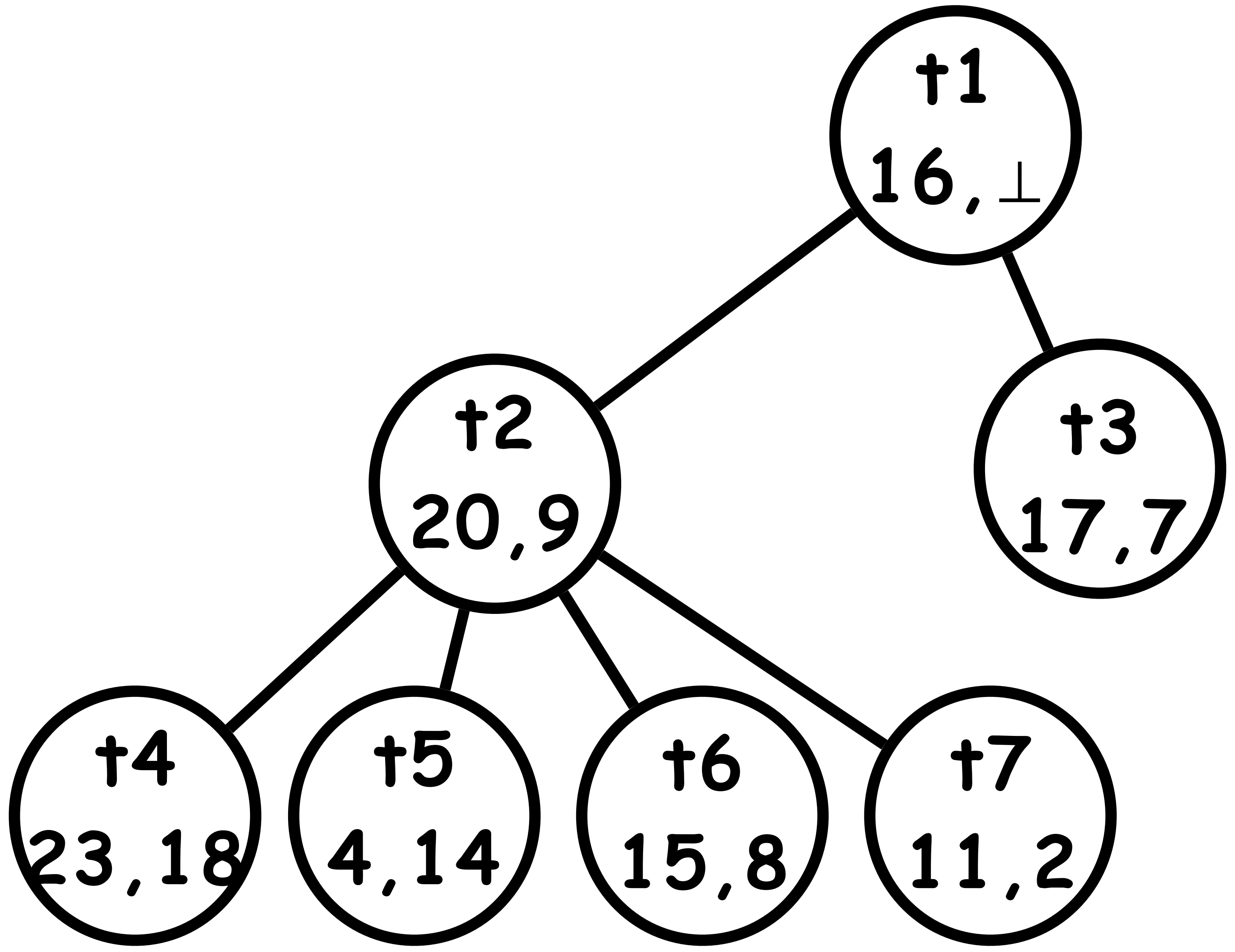
vector clock

Tree Clock

- Implementing <u>logical clocks</u> using generic trees

$$(t_1 : 16,\ t_2 : 20,\ t_3 : 17,$$
$$t_4 : 23,\ t_5 : 4,\ t_6 : 15,\ t_7 : 11)$$

vector clock



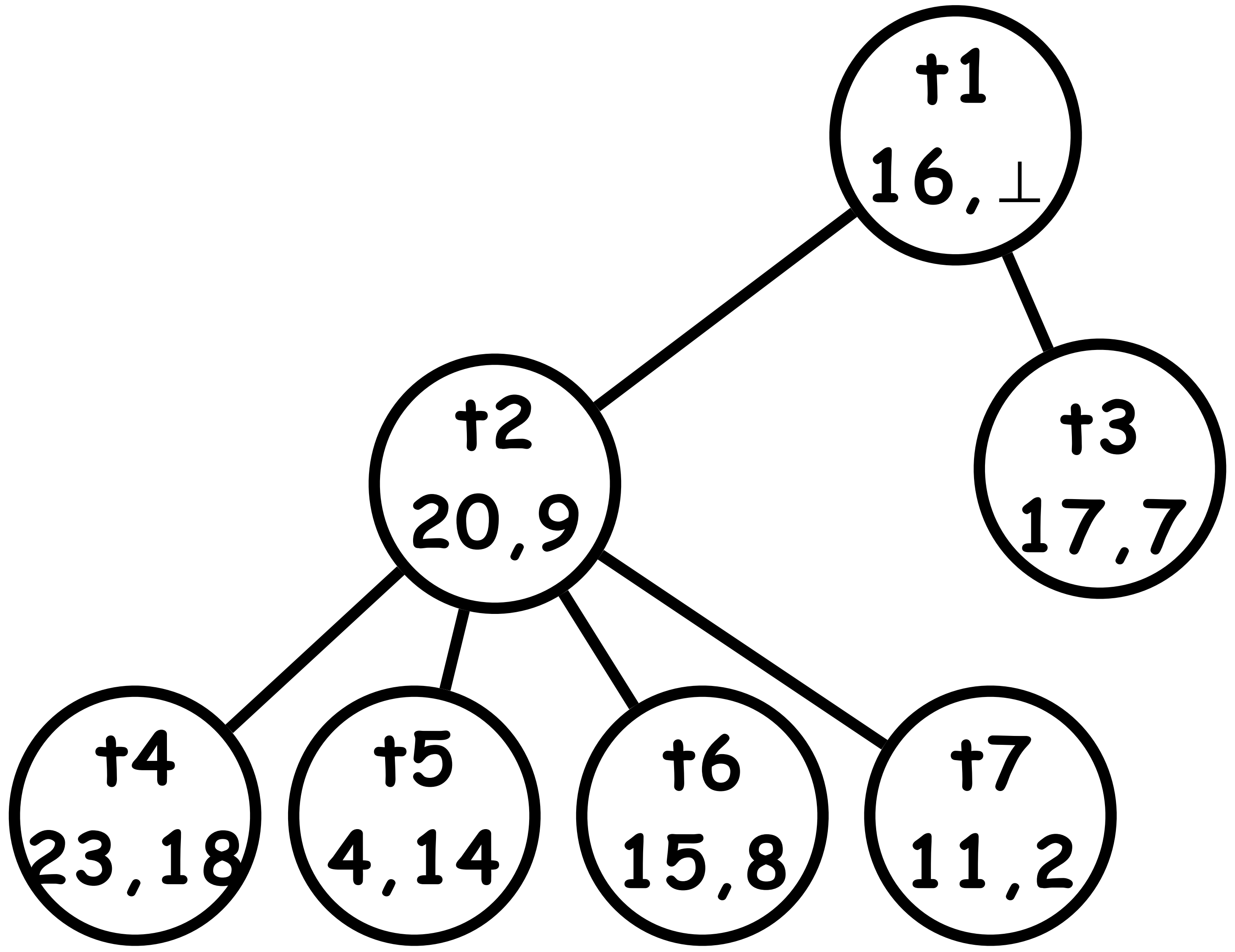
tree clock

Tree Clock

- Implementing logical clocks using generic trees
- **Optimal** asymptotic time complexity in performing logical clock operations

$$(t_1 : 16,\ t_2 : 20,\ t_3 : 17,\ t_4 : 23,\ t_5 : 4,\ t_6 : 15,\ t_7 : 11)$$

vector clock



tree clock

Tree Clock (Cont’d)

- The number of threads are bounded $\implies$ suitable as array-based trees

Tree Clock (Cont'd)

- The number of threads are bounded $\Longrightarrow$ suitable as array-based trees
- Its join operation: happens between two tree clocks, TC_1 and TC_2

Tree Clock (Cont'd)

- The number of threads are bounded $\Longrightarrow$ suitable as array-based trees
- Its join operation: happens between two tree clocks, TC_1 and TC_2
  - Performing non-recursive traversal over TC_2

Tree Clock (Cont'd)

- The number of threads are bounded $\implies$ suitable as array-based trees
- Its join operation: happens between two tree clocks, TC_1 and TC_2
  - Performing non-recursive traversal over TC_2
  - Performing structure changing operations on TC_1 according to the stack top node of TC_2

Tree Clock (Cont'd)

- The number of threads are bounded $\Longrightarrow$ suitable as array-based trees
- Its join operation: happens between two tree clocks, TC_1 and TC_2
  - Performing non-recursive traversal over TC_2
  - Performing structure changing operations on TC_1 according to the stack top node of TC_2
  - $\Longrightarrow$ Manifesting both challenges

Verifying Tree Clock

- Tree clock is originally implemented in Java
  - Faithfully translated into C
- Its functional model: verified in Coq
- Its imperative join operation: verified using Verified Software Toolchain (VST)

Rooting For Efficiency

Table 2. Evaluation: array-based v. pointer-based tree clocks

1	2	3	4	5	6
Trace len.	num.	Avg. len.	Ptr. TC (s)	Arr. TC (s)	Speedup
(0M, 60M]	35	0.14M	0.22	0.16	1.25×
(60M, 112M]	24	102M	162.27	115.32	1.41×
(112M, 136M]	29	125M	206.57	147.22	1.40×
(136M, 215M]	29	169M	222.36	190.72	1.17×
(215M, 1B]	29	391M	657.23	463.32	1.42×
Total	146	31.41	48.90	36.10	1.35×

- Array-based trees do bring efficiency!

Summary

- Array-based trees: performance-oriented implementation of tree structures
- Challenges: structure changing operations and non-recursive tree traversals
- Strategies: dual views and tree splitting
- Case study: verification of tree clock

Mechanised development

Our paper (this talk)

The tree clock paper

Summary

- Array-based trees: performance-oriented implementation of tree structures
- Challenges: structure changing operations and non-recursive tree traversals
- Strategies: dual views and tree splitting
- Case study: verification of tree clock

Mechanised development

Our paper (this talk)

The tree clock paper

Thank you! 😃