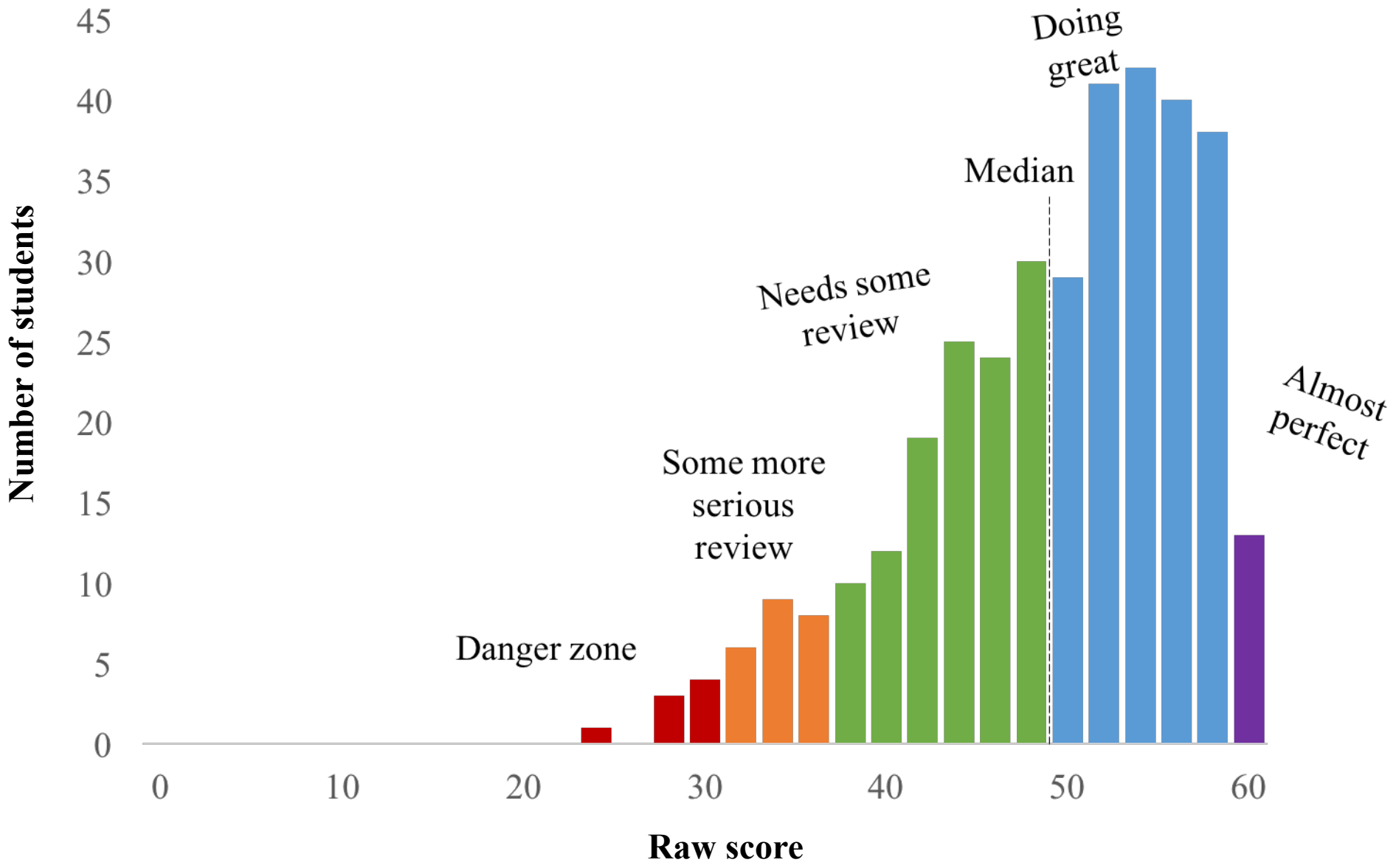
Trees

CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Room: **CS106BFALL**

Midterm Results

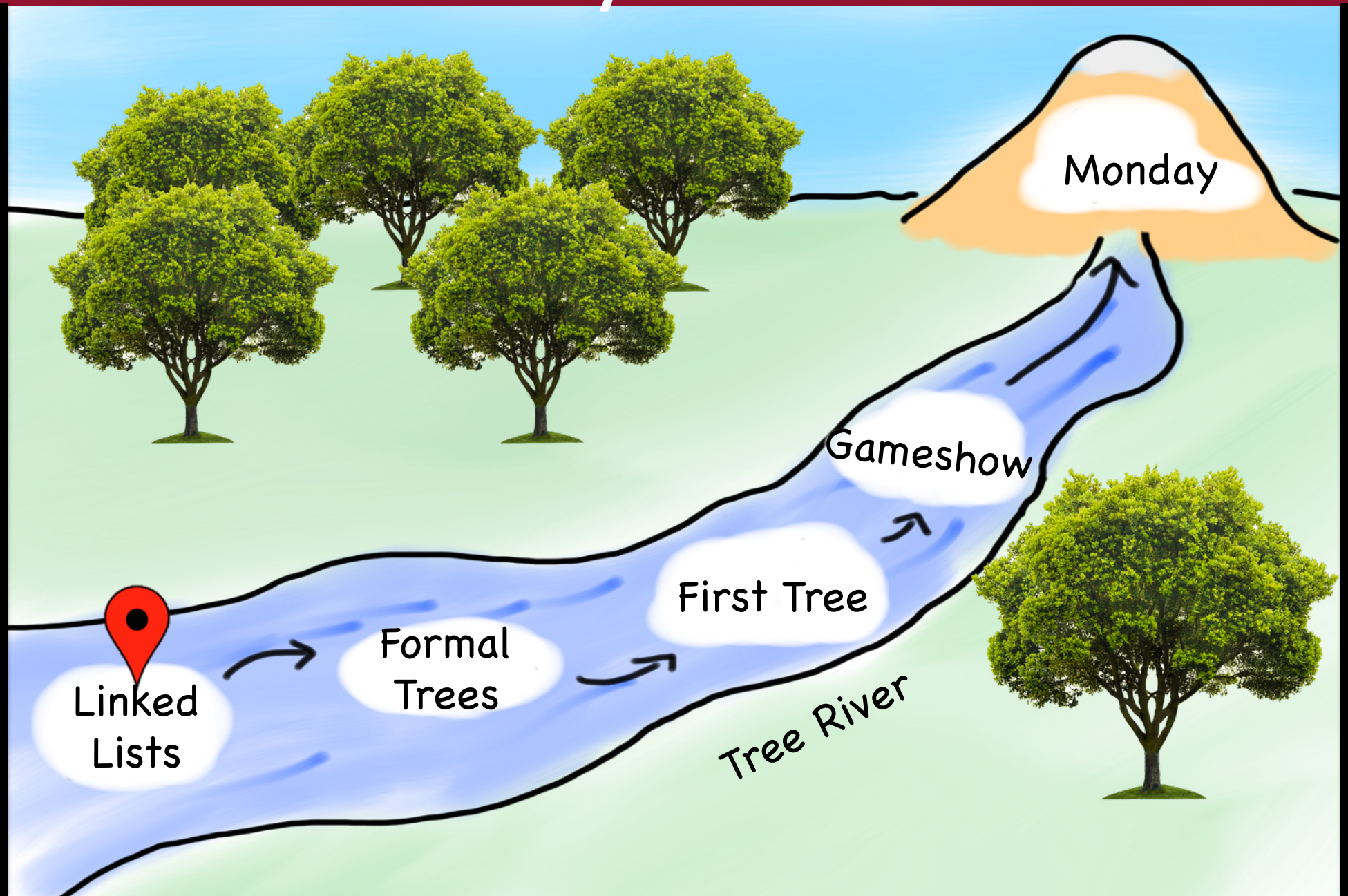


Challenge: Design a Queue with O(1) enqueue and dequeue

Today's Goals

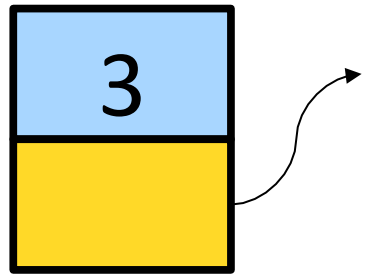
1. Practice Linked Lists
2. Learn Trees

Today's Route



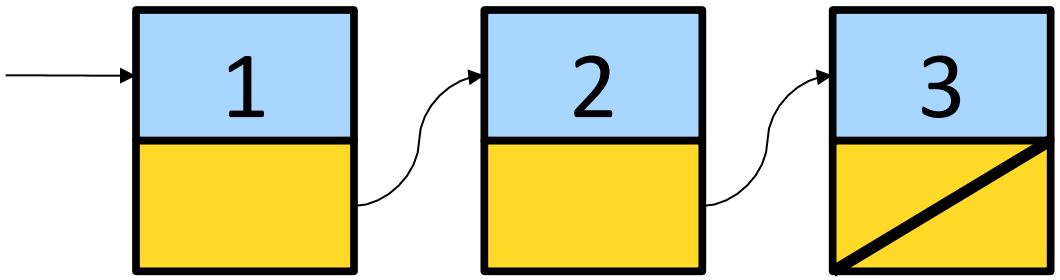
Linked List

- A linked list is a chain of **nodes**.
- Each node contains two pieces of information:
  - Some piece of data that is stored in the sequence
  - A **link** to the next node in the list.
- We can traverse the list by starting at the first cell and repeatedly following its link.



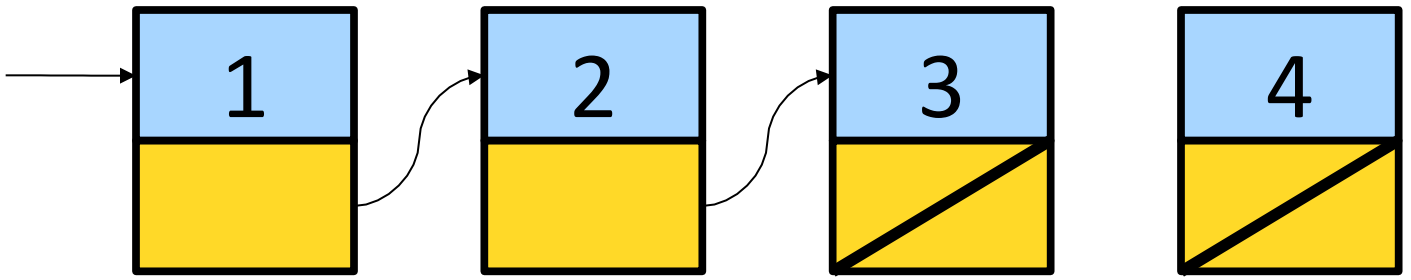
Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



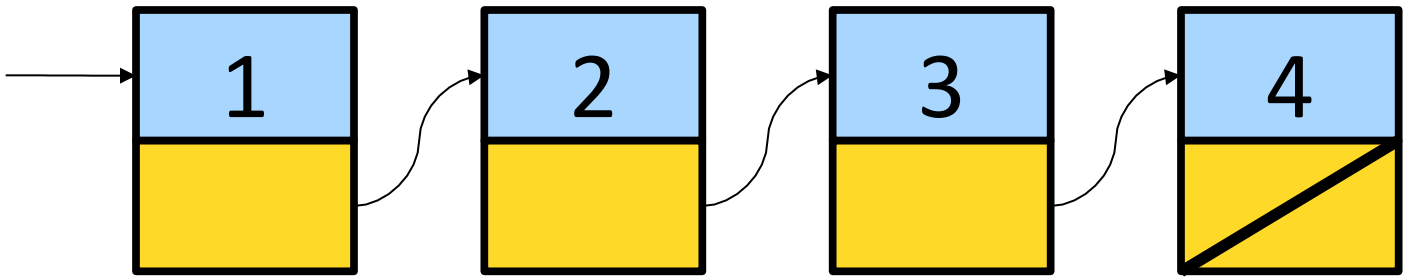
Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



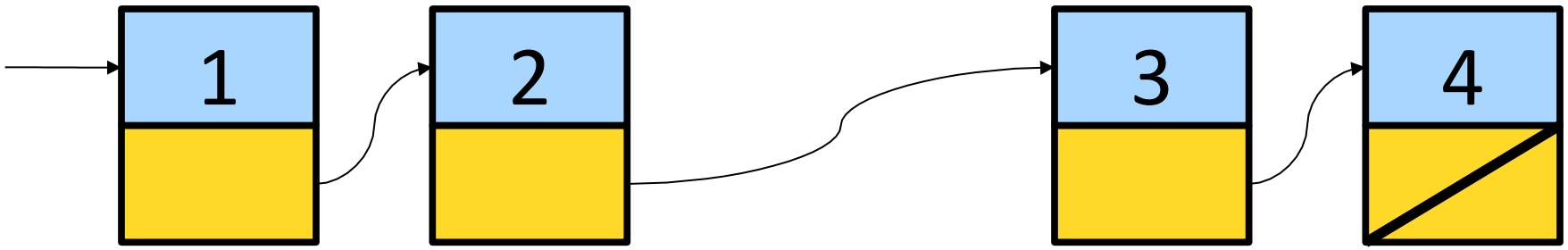
Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



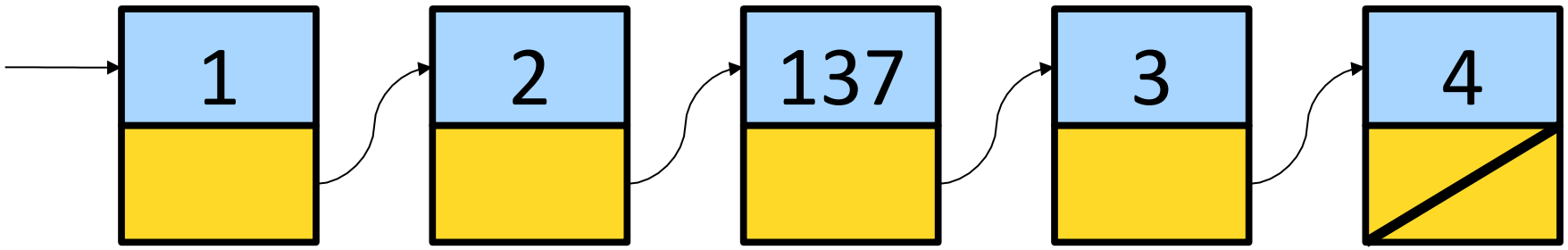
Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



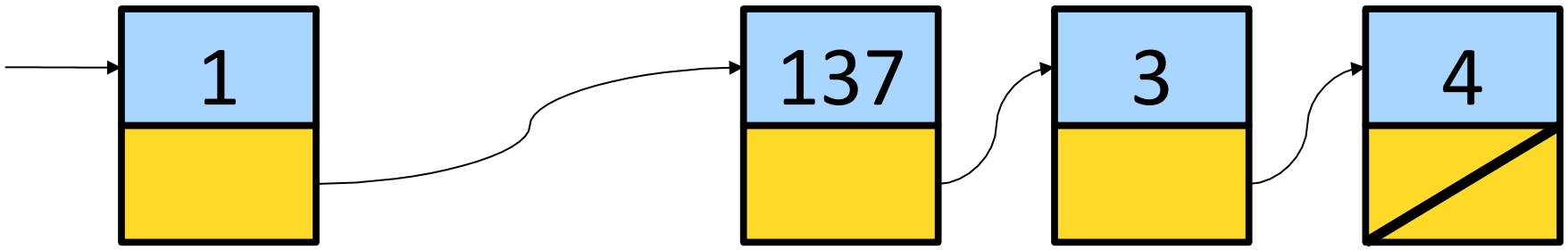
Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Linked Lists

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.



Practice with pointers

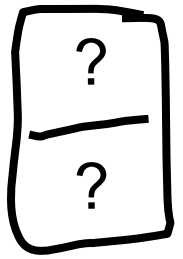
```
struct Node{
   int value;     /* The value of this elem    */
   Node *link;    /* Pointer to the next node */
};
```

```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```

Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

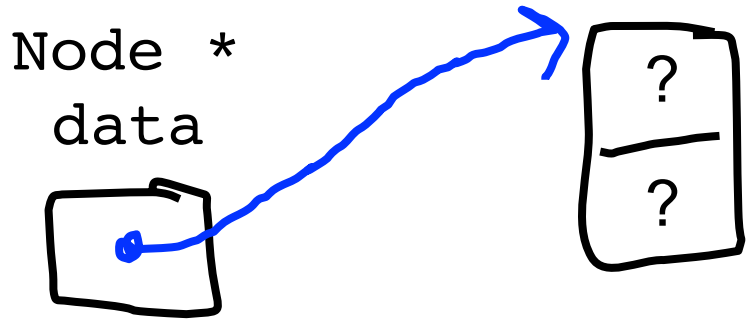
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

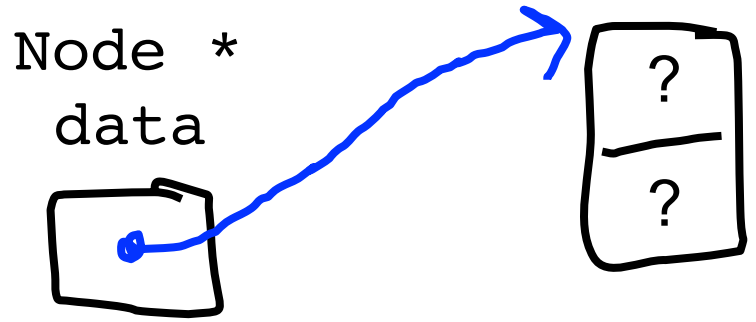
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

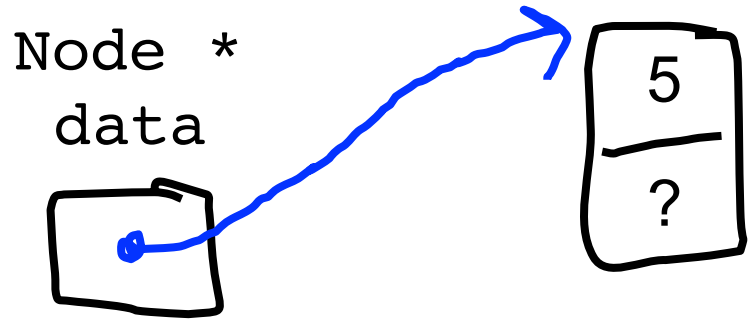
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

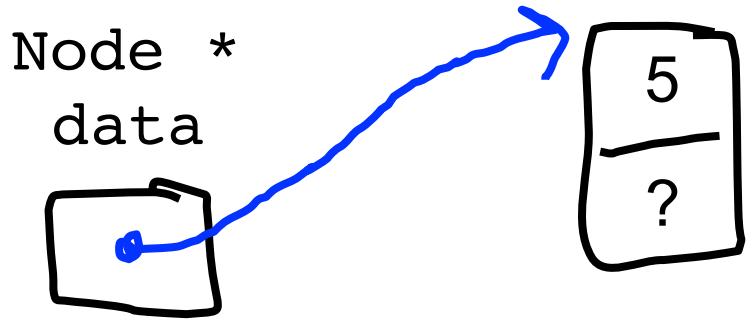
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

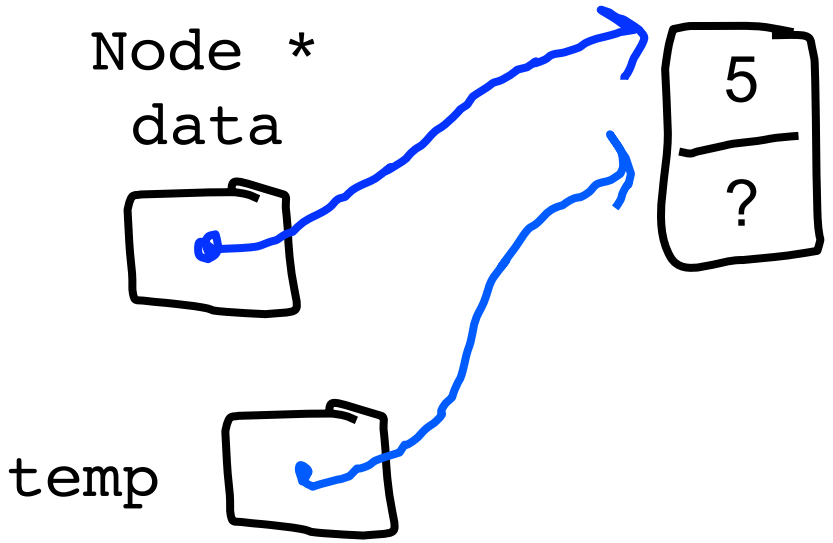
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

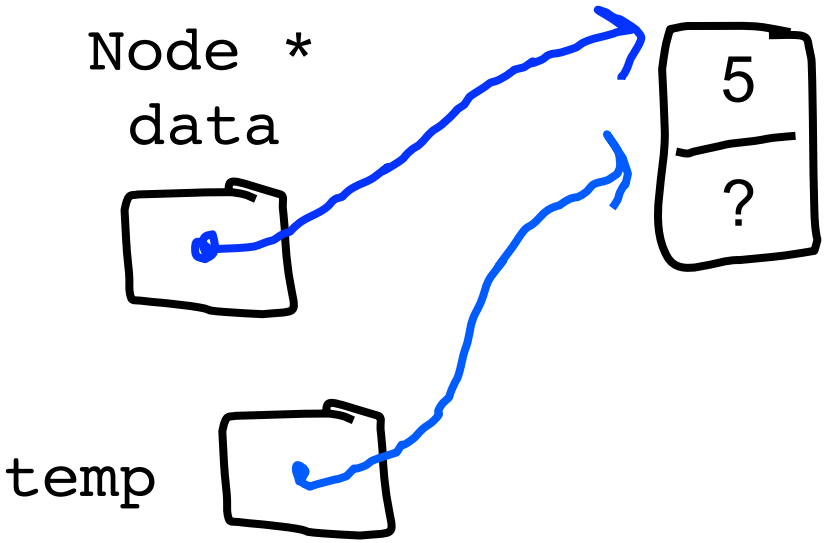
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

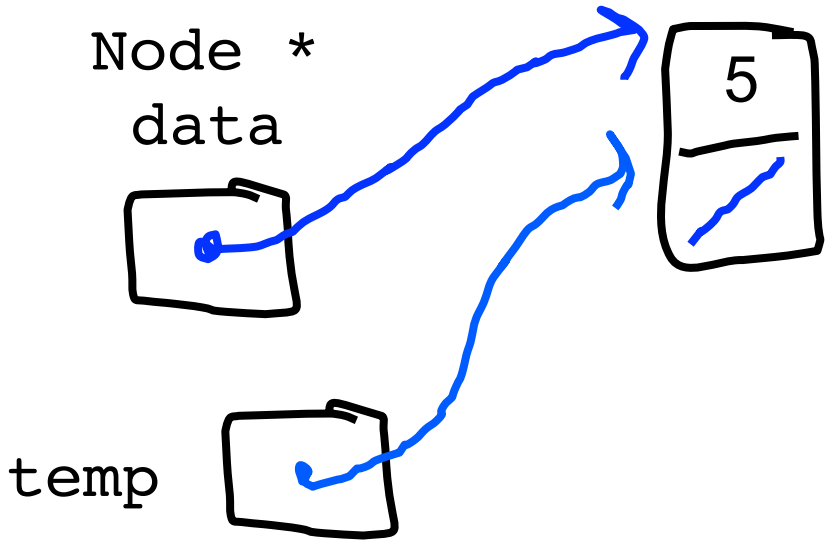
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

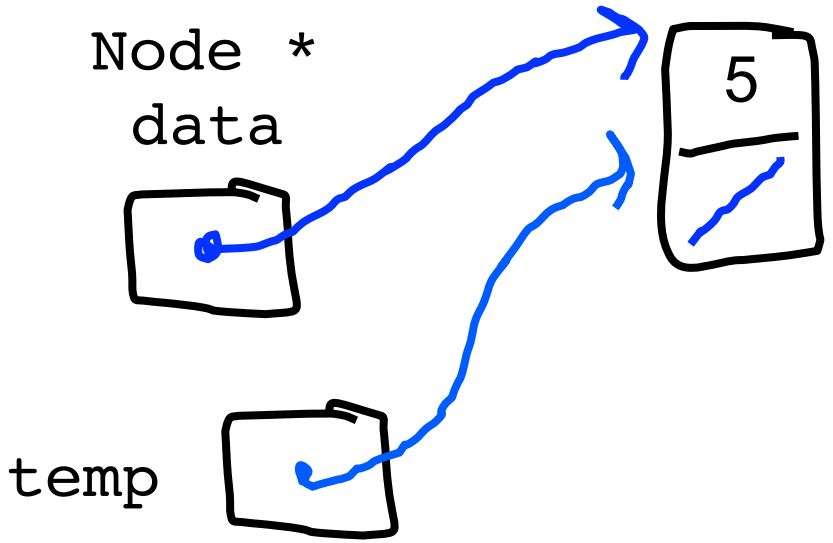
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

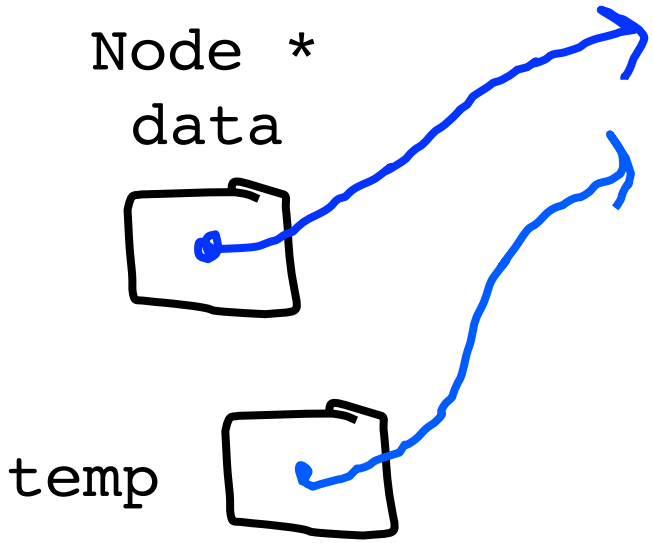
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



Practice with pointers

```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

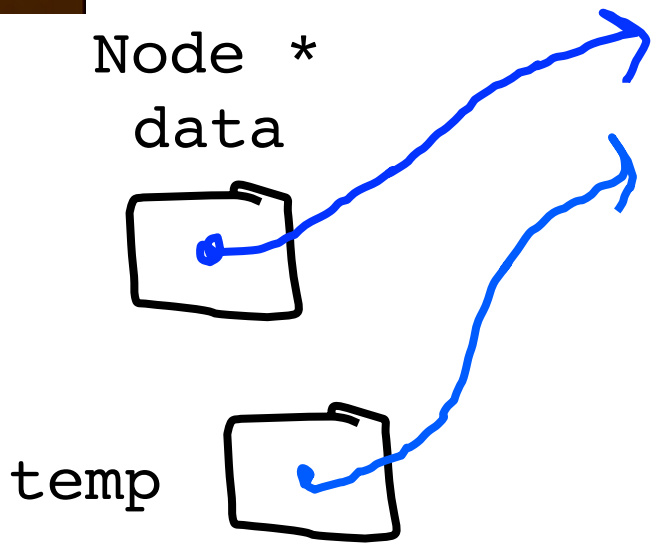
```
Node * data = new Node;
data->value = 5;
Node * temp = data;
data->next = NULL;
delete data;
```



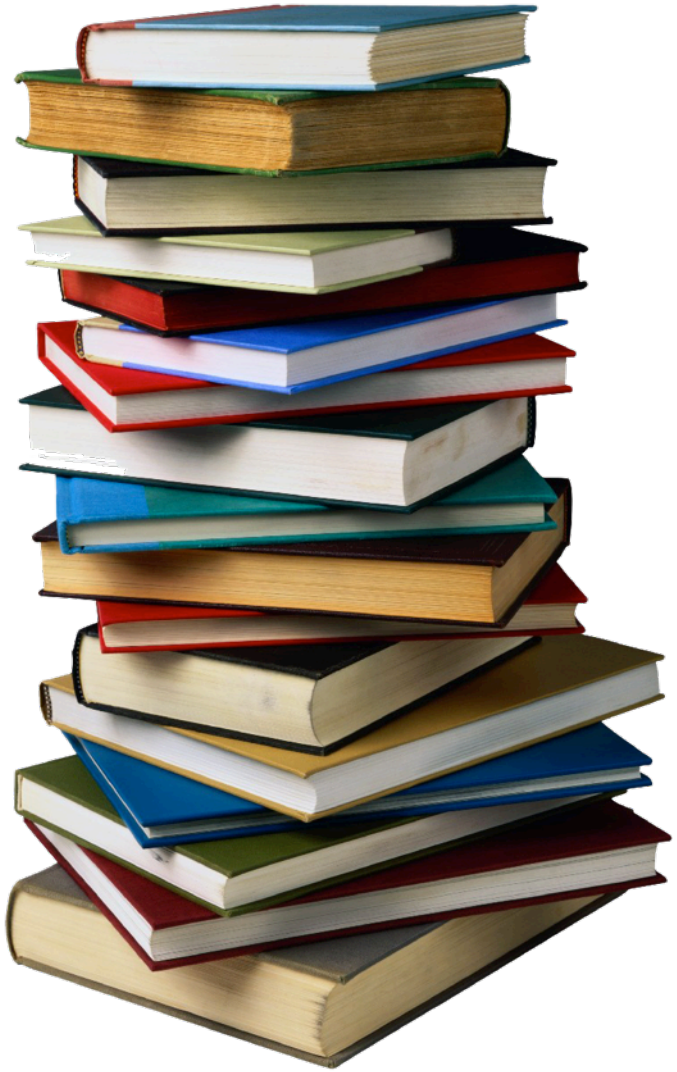
Practice with pointers

```
str
                                  this elem    */
                                 he next node */
};
```


```
cout << temp->value;
```



How is the Stack Implemented?



Stack

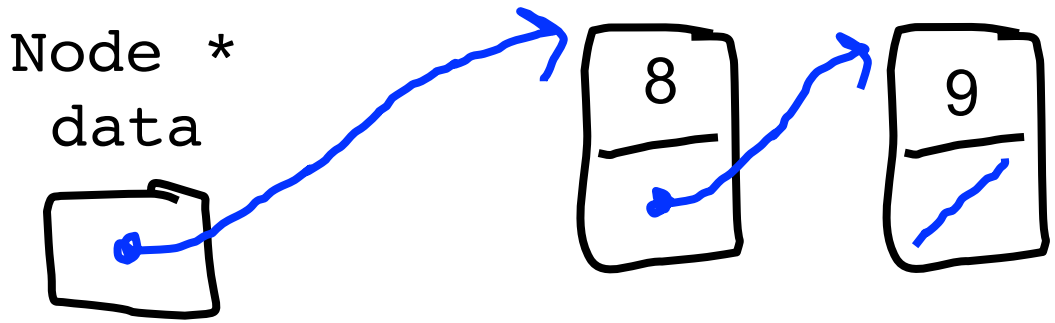
```
struct Node{
    int value;     /* The value of this elem    */
    Node *link;    /* Pointer to the next node */
};
```

```
Node * data;
```

This variable stores an entire linked list!

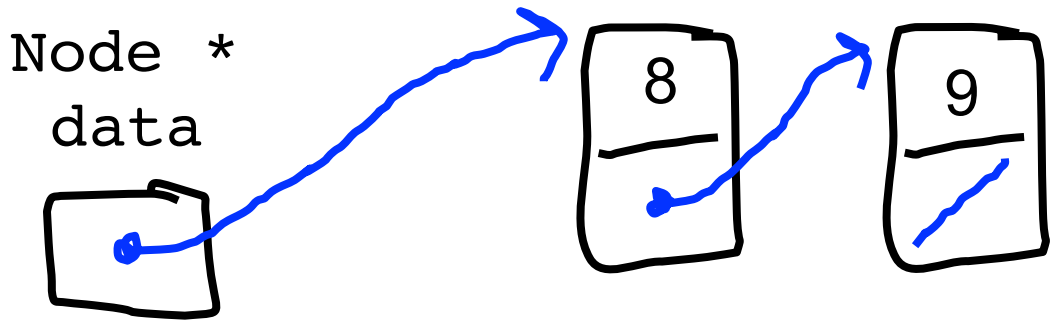
Stack

```
class StackInt {                    // in StackInt.h
public:
    StackInt ();                    // constructor

    void push(value);                // append a value
    int pop();                      // return the first-in value

private:
    struct Node {
      int value;
      Node * link;
    };
    Node * data;          // member variables
};
```

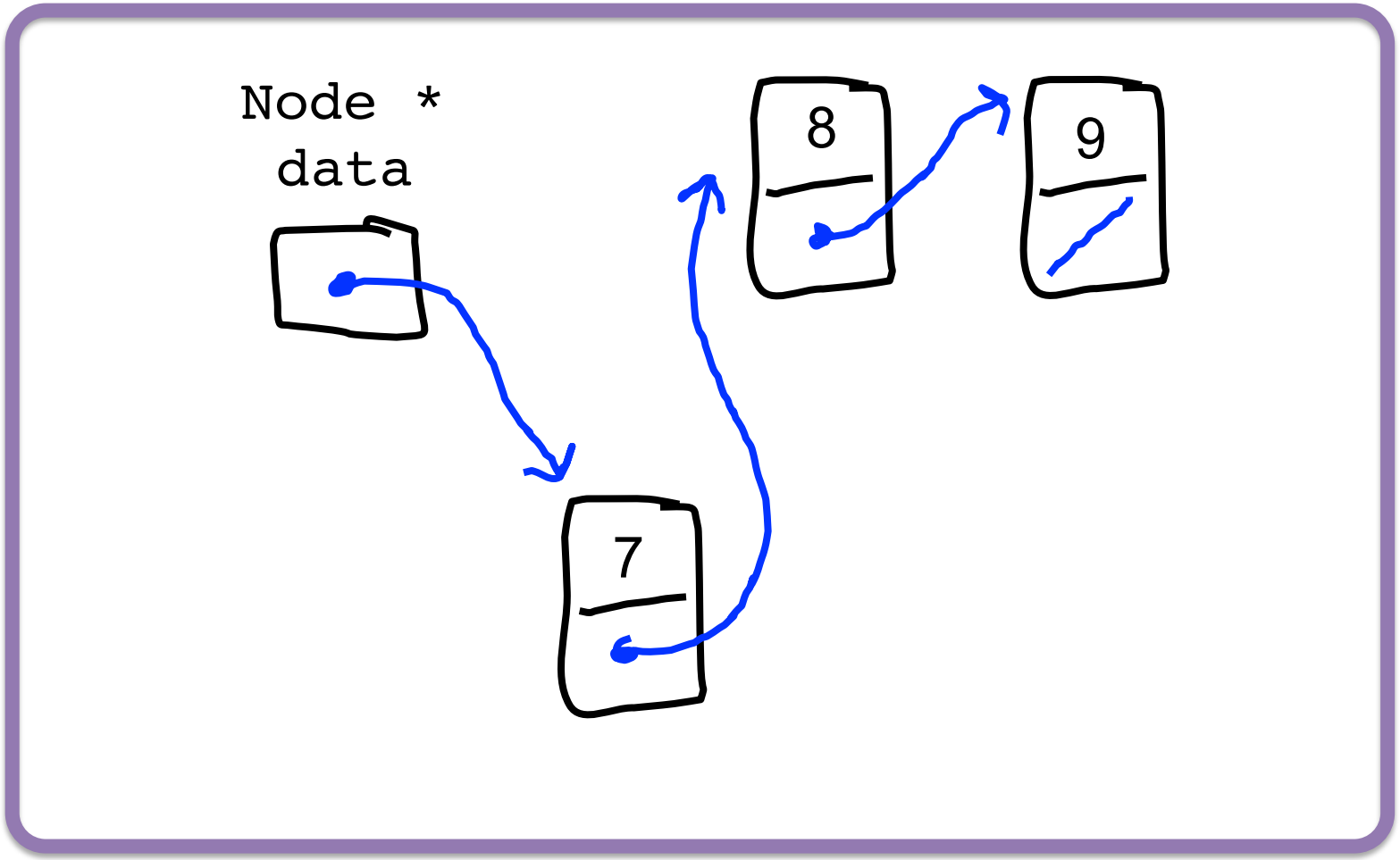

Node *
data
8
9

Push(7)

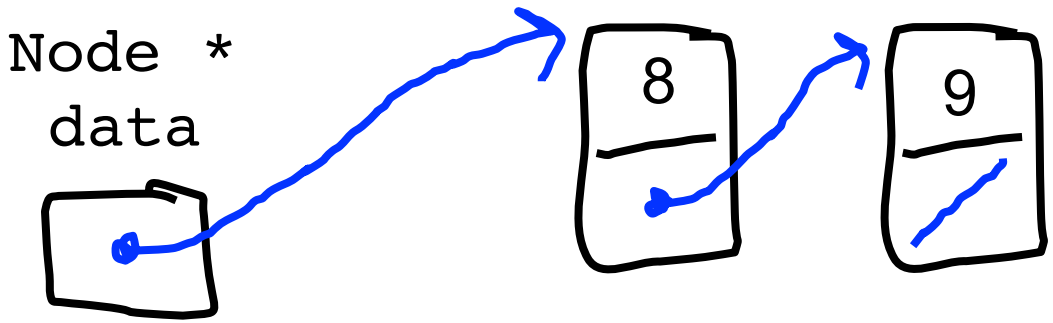
```
push(7);
```



Goal of Push



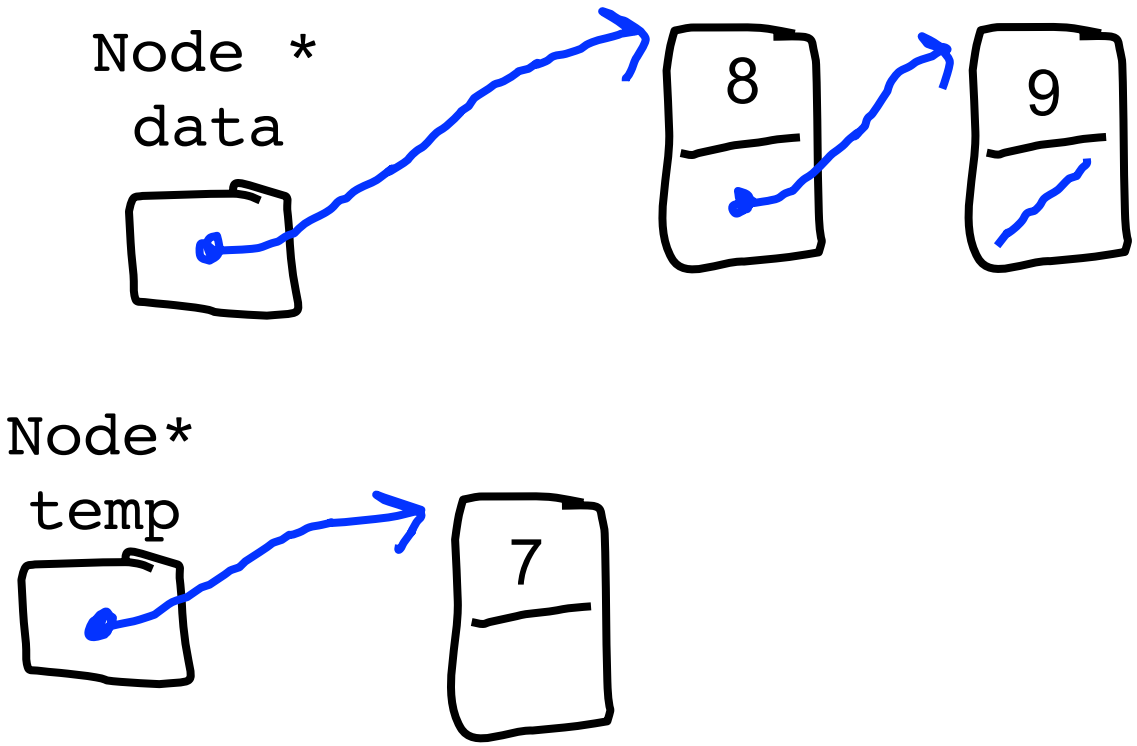
```
push(7);
```



Game plan:

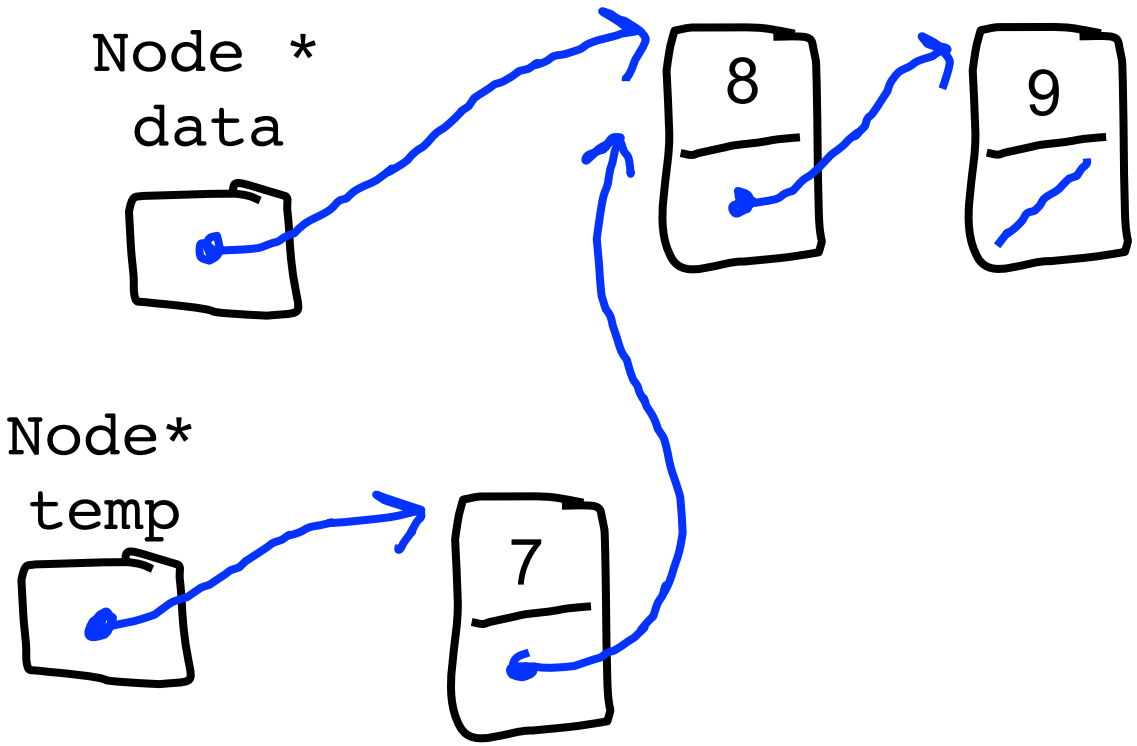
1. Make a new node with the value 7
2. Have the new node point to the old first node
3. Have “data” point to the new node

```
push(7);
```



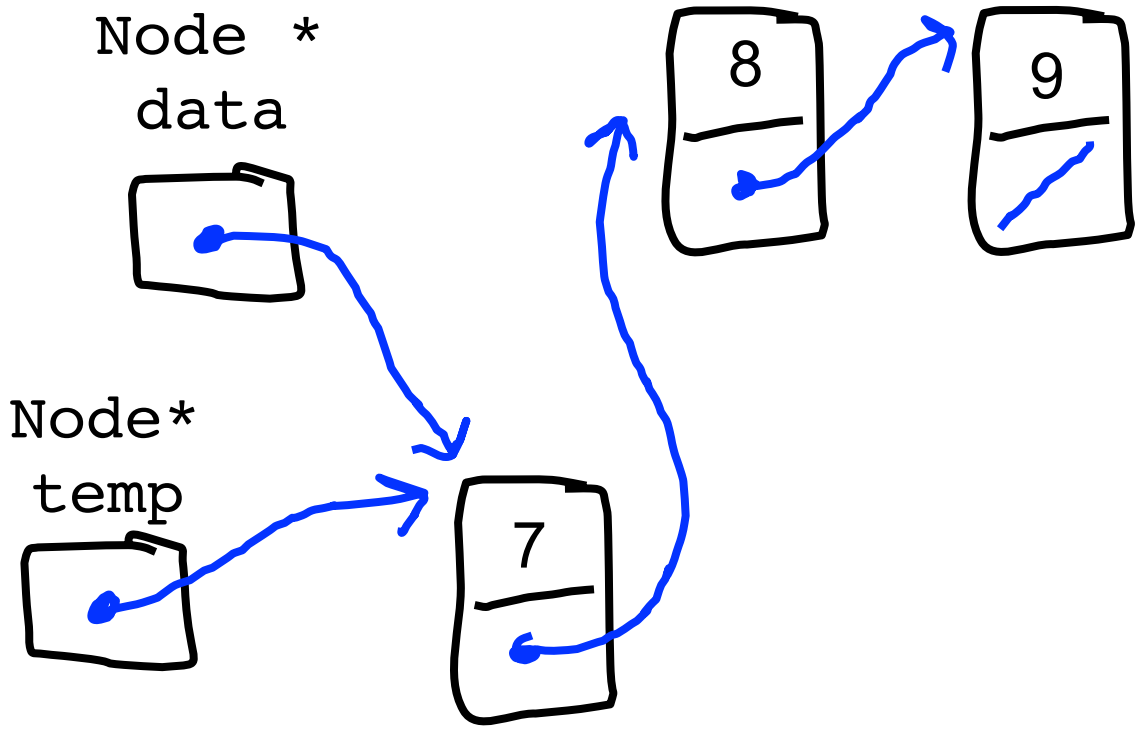

```
Node * temp =  new Node;
temp -> value = 7;
```

```
push(7);
```



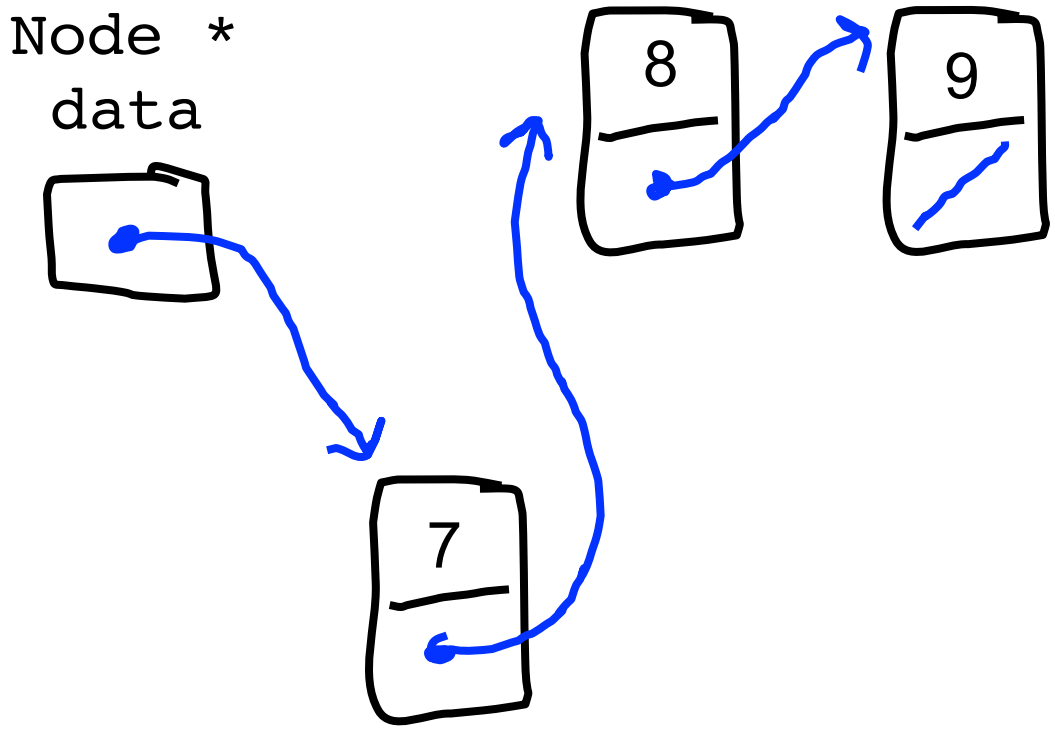

```
temp -> link = data;
```

```
push(7);
```




```
data = temp;
```

```
push(7);
```



exit function

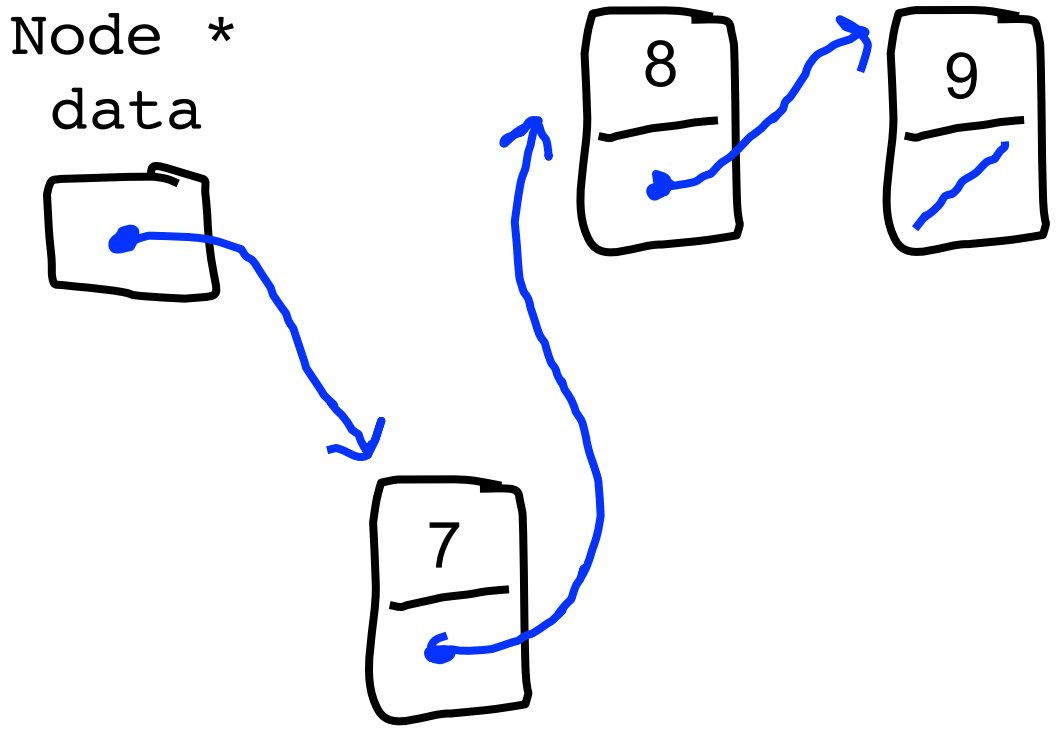
Stack Implementation

```
void StackInt::push(int v) {
    // step 1: make a new node
    Node * temp = new Node;
    temp->value = v;

    // step 2: have the new node point to
    // the old first node
    temp->link = data;

    // step 3: have "data" point to the
    // new node
    data = temp;
}
```

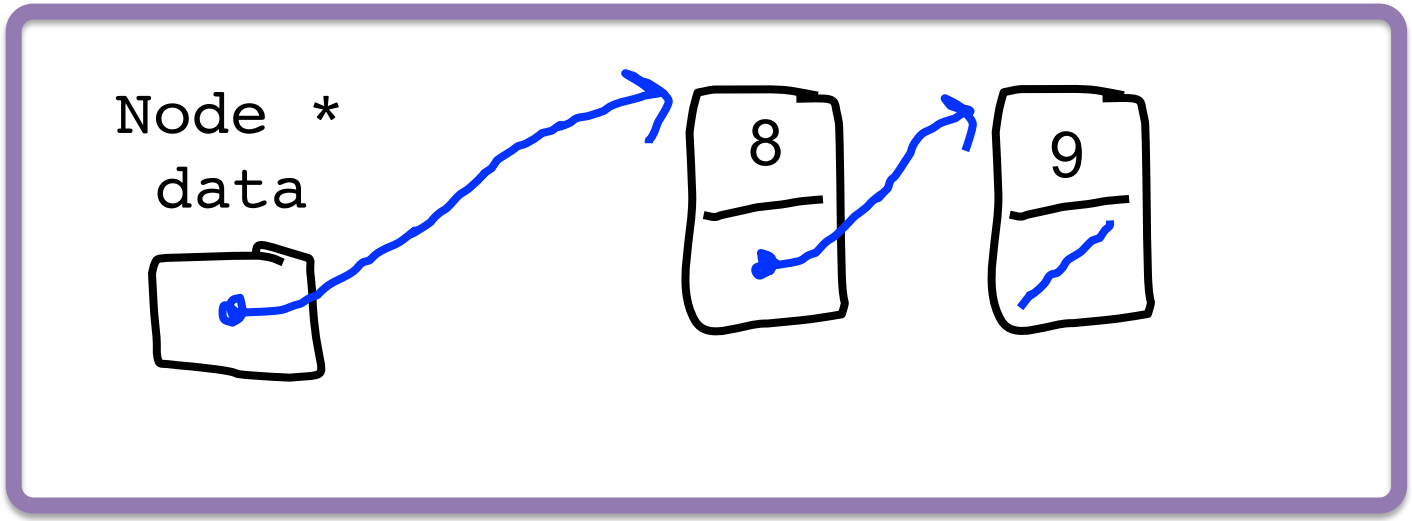
Stack: Big O of Push?

Pop

```
pop();
```



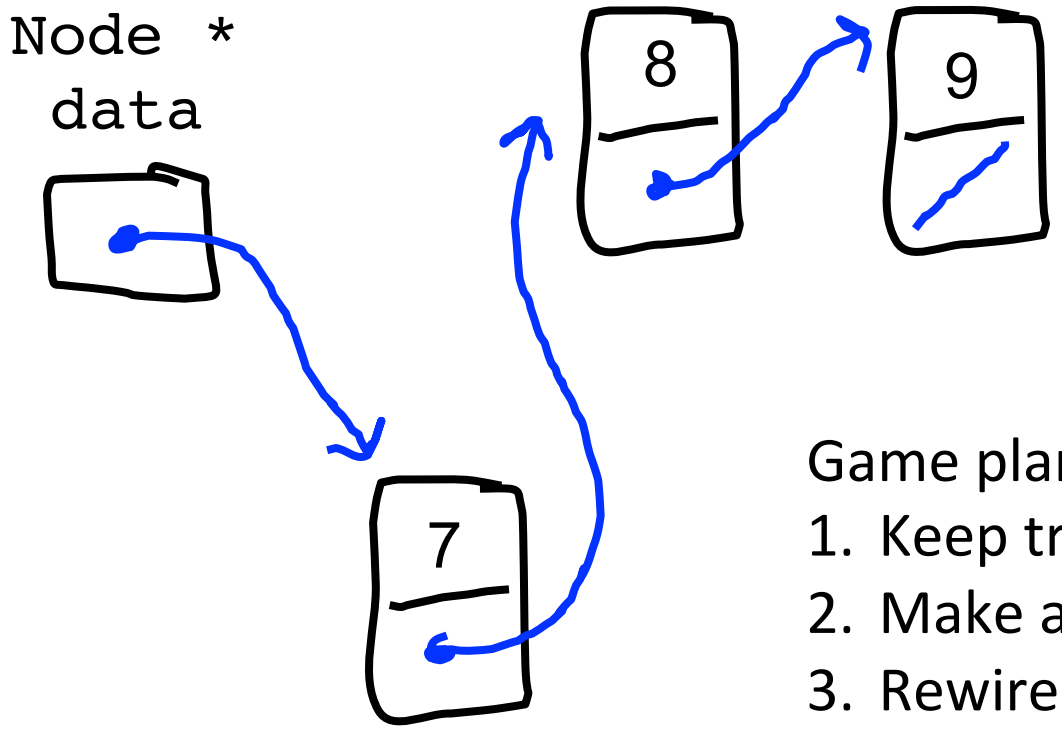
Goal of Pop




```
return 7;
```

Pop

```
pop();
```

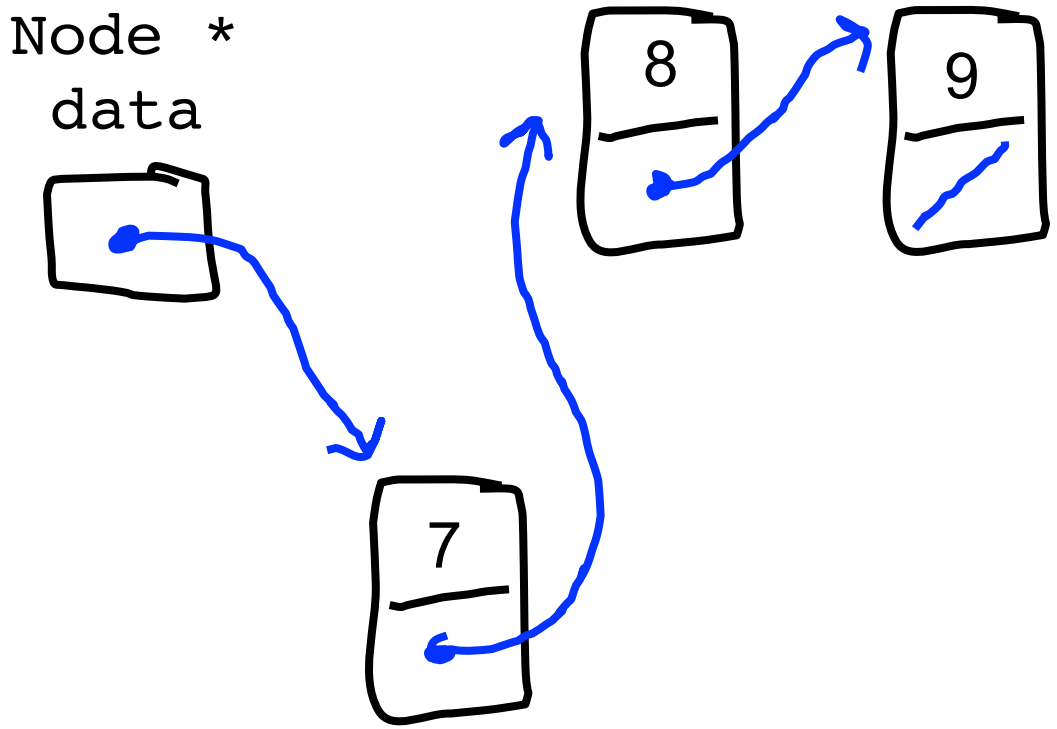


Game plan:

1. Keep track of 7 as the number to give back
2. Make a temporary pointer to first node
3. Rewire the “data” pointer to point to second node
4. Call delete on first node

Pop

```
pop();
```



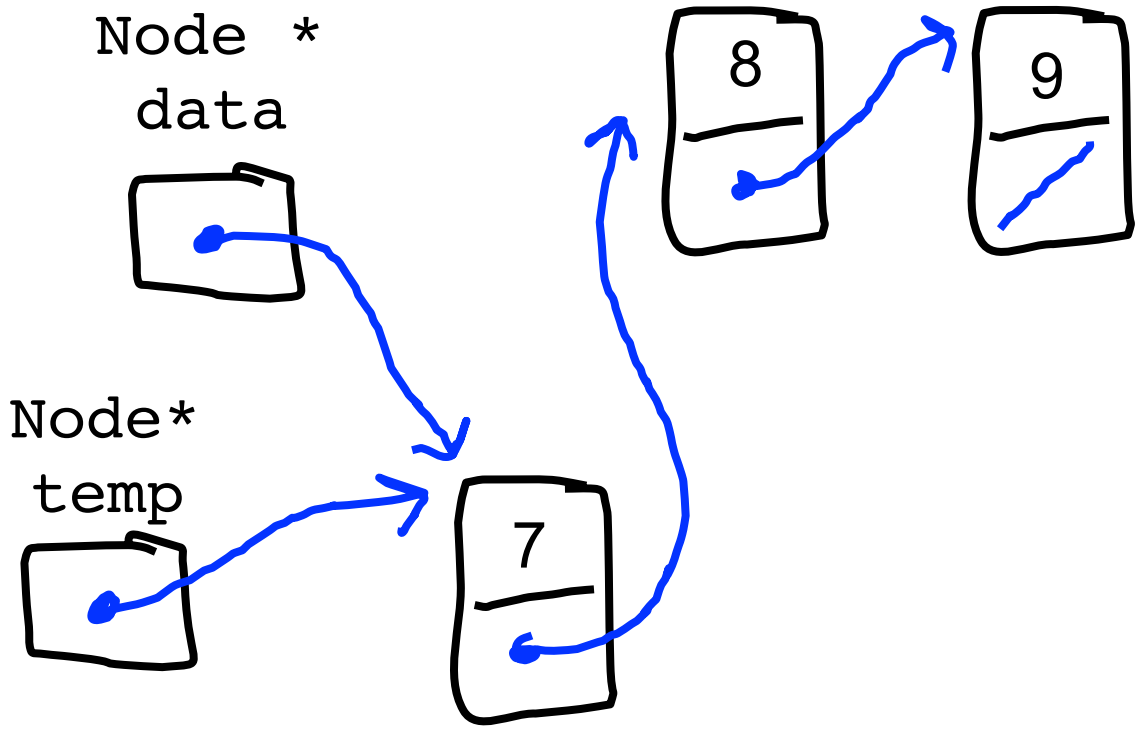
int toReturn

7

```
int toReturn = data->value;
```

Pop

```
pop();
```

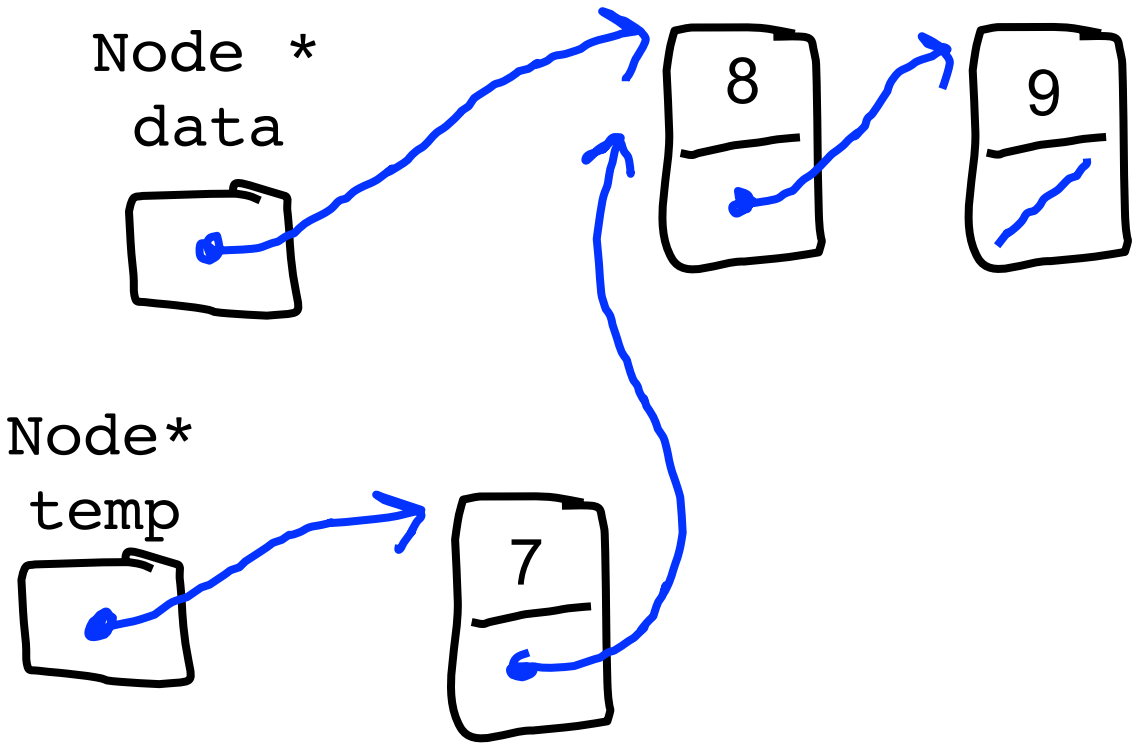



```
int toReturn
```

```
7
```

```
Node * temp = data;
```

Pop




```
pop();
```

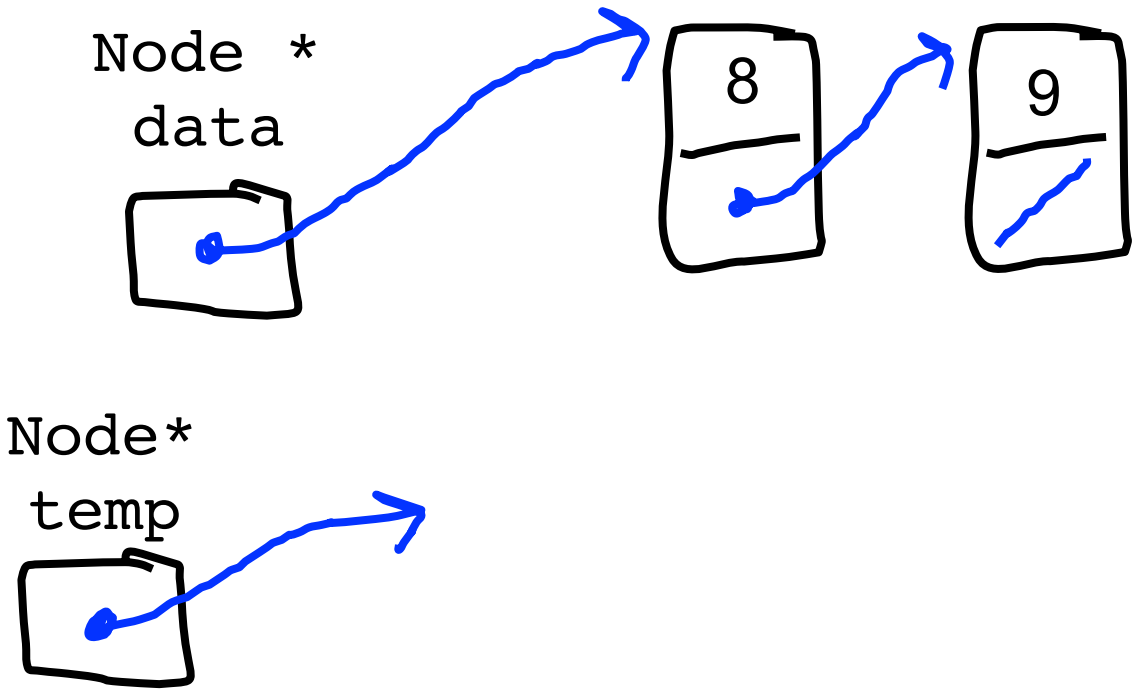
int toReturn

7

```
data = temp->link;
```

Pop

```
pop();
```

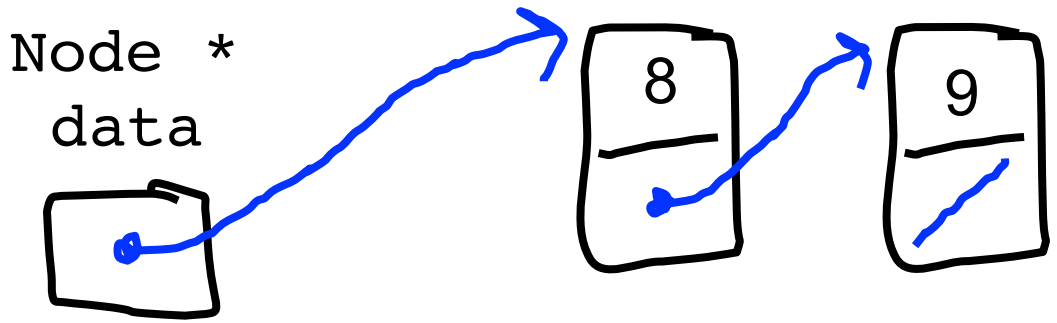


int toReturn

7

```
delete temp;
```

Pop




```
pop();
```

```
int toReturn
```

7

```
return toReturn;
```

Stack Implementation

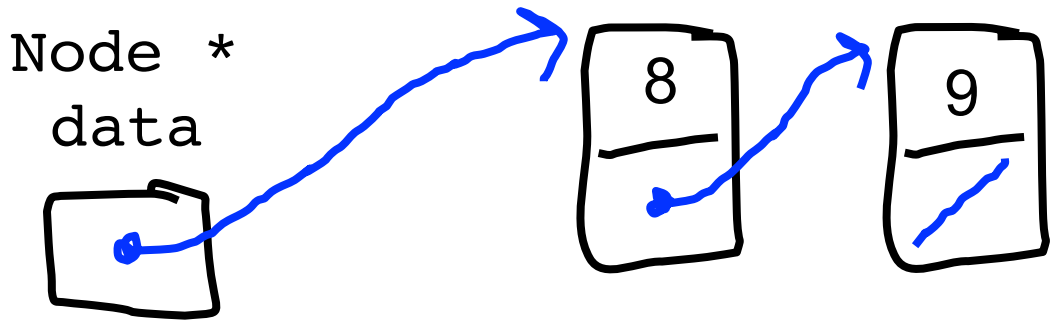
```
int StackInt::pop() {
    // 1. keep track of the number of the
    // top node
    int toReturn = data->value;

    // 2. save a temp pointer to the old
    // first value.
    Node * temp = data;

    // 3. rewire the "data" pointer to
    // point to the second node
    data = temp->link;

    // 4. delete the node that was popped
    delete temp;
    return toReturn;
}
```

Stack: Big O of Pop?

Are Queues The Same?

Not so obvious…

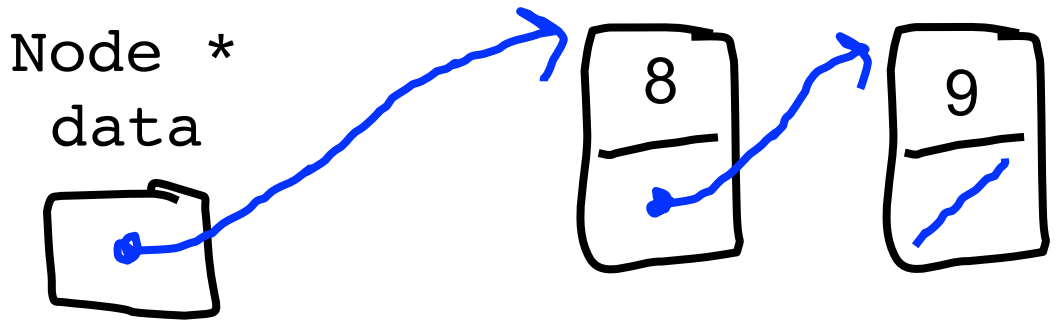
Queue?



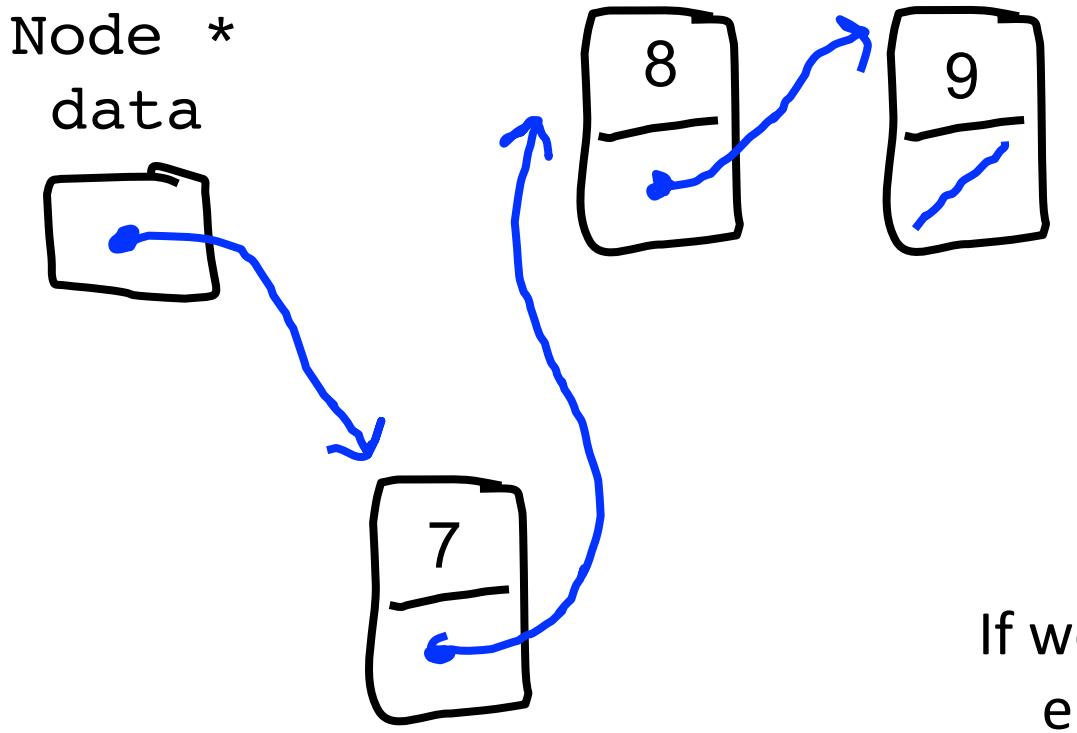
We could implement a Queue just like a Stack… using a regular linked list

The Queue's only instance variable would be a Node *

Queue Enqueue?

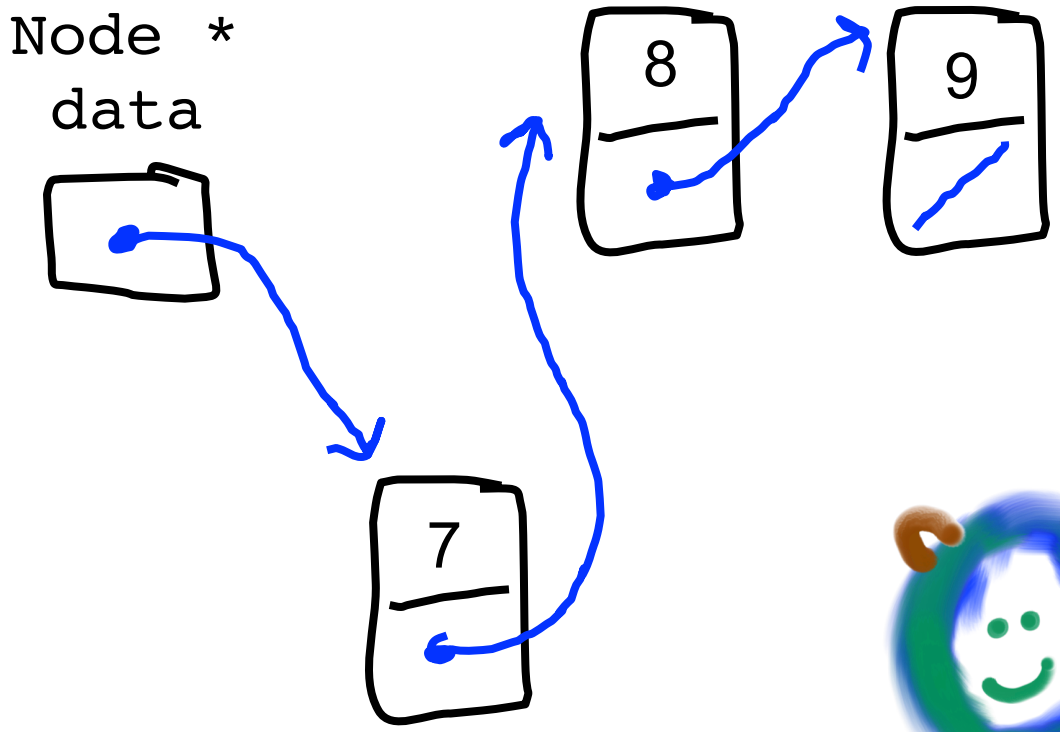


Queue Enqueue?



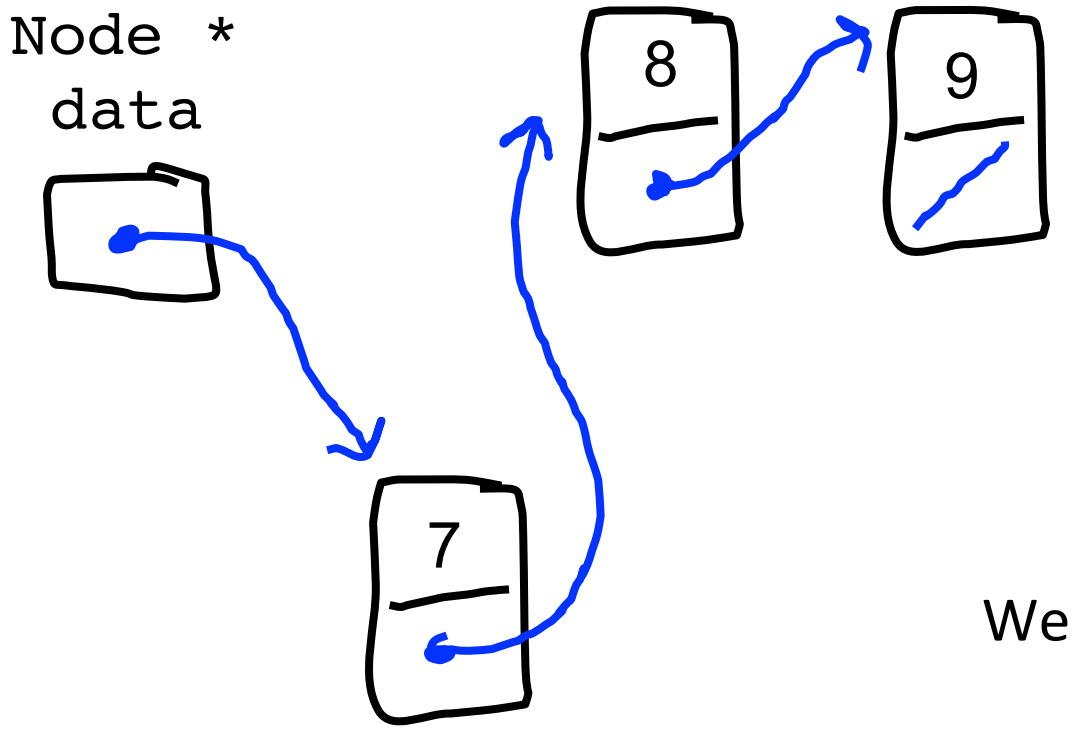
If we enqueue in the front, then enqueue is super fast O(1)!

Queue Enqueue?



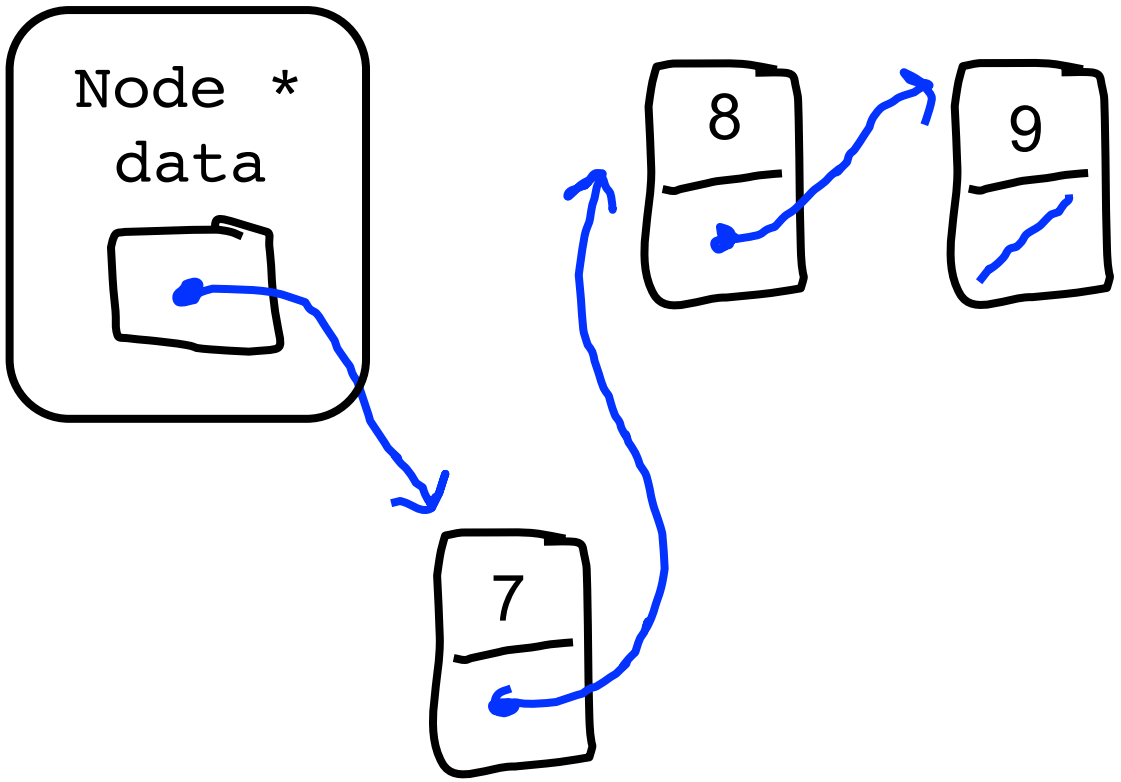
Great! And dequeue?

Queue Dequeue?

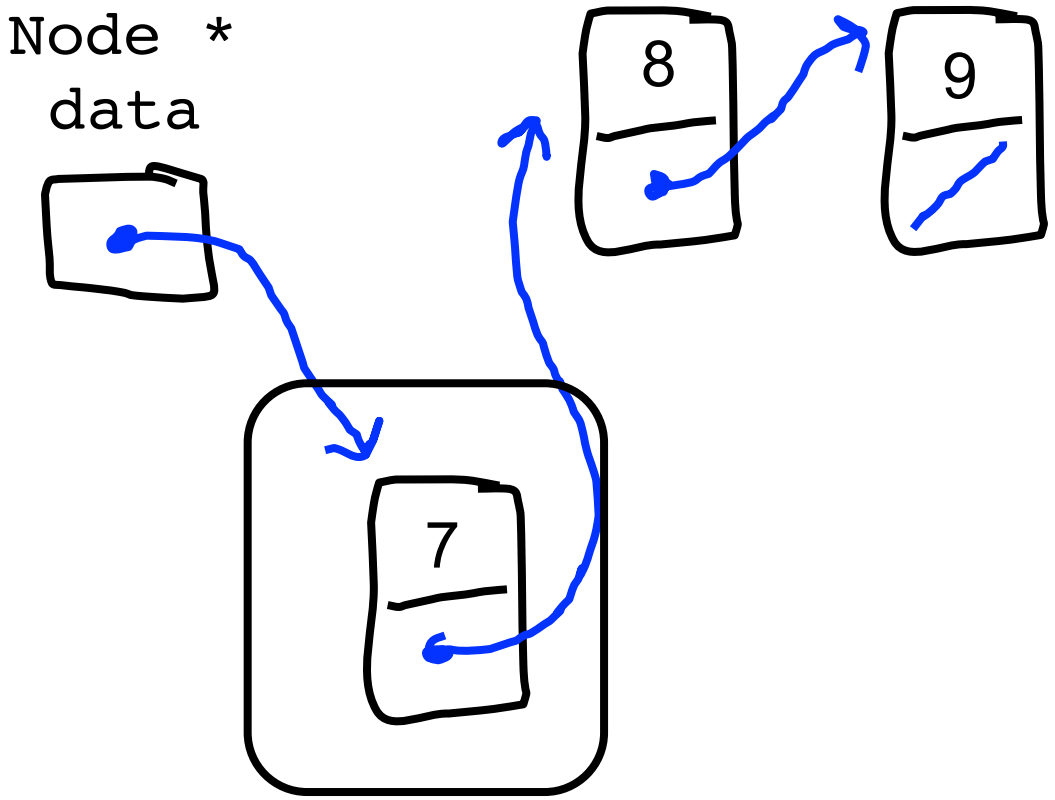


We need to find the back of the queue and remove it.

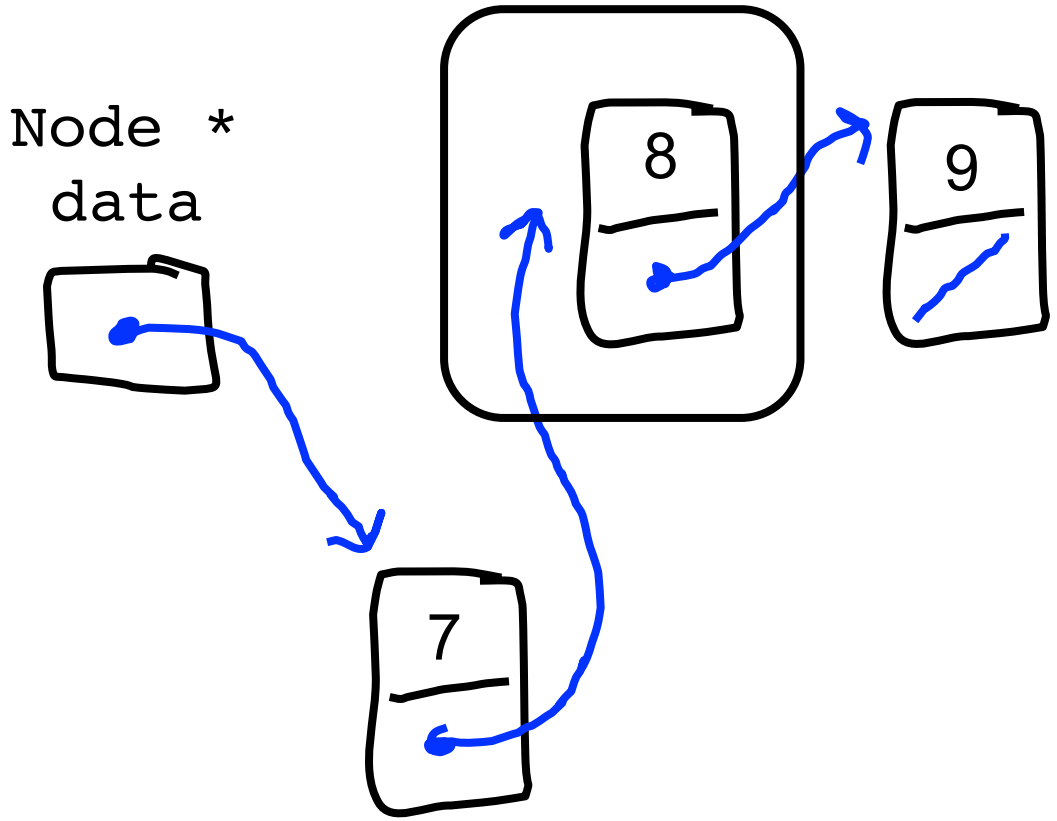
Queue Dequeue?



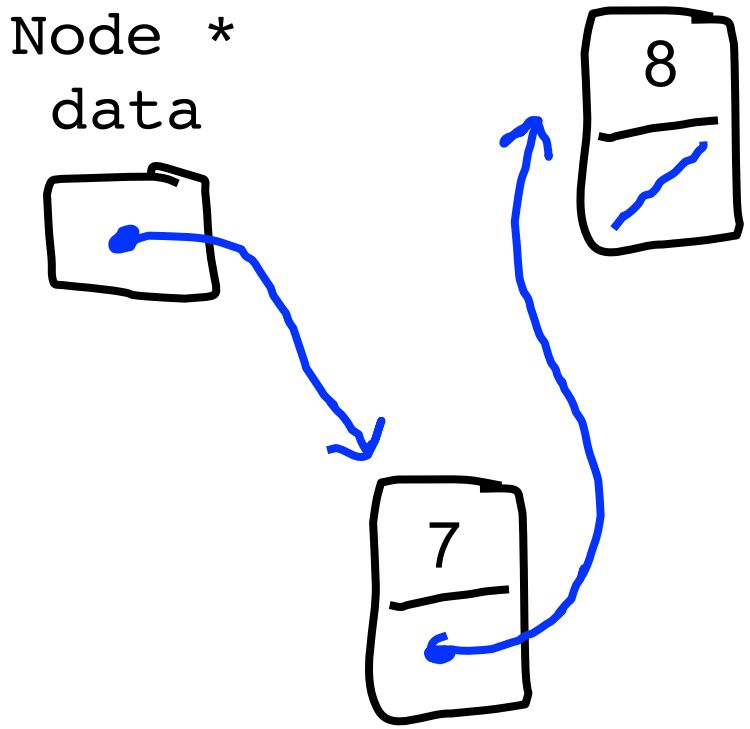
Queue Dequeue?



Queue Dequeue?

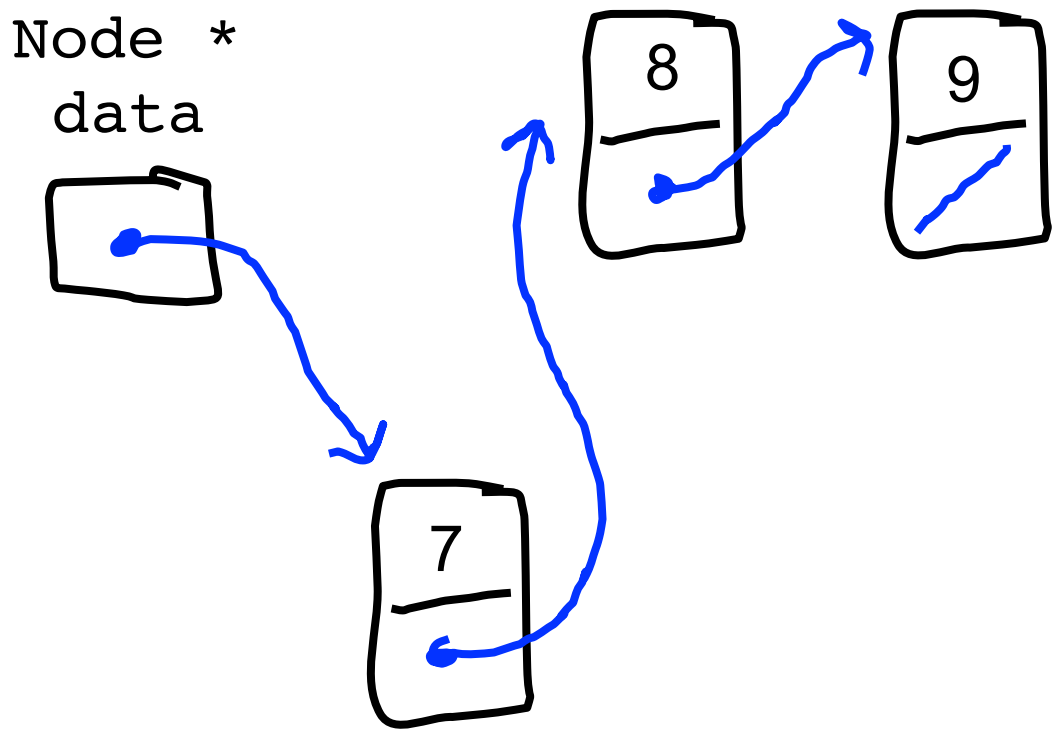


Queue Dequeue?



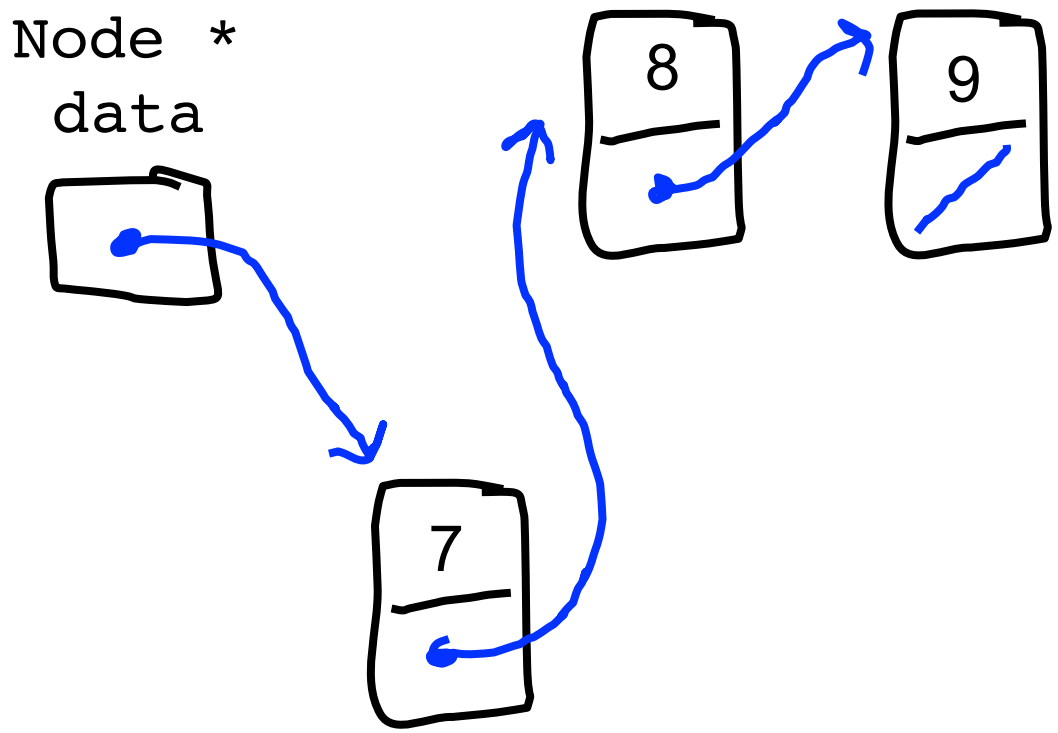
Lets code that dequeue!

Queue Dequeue?



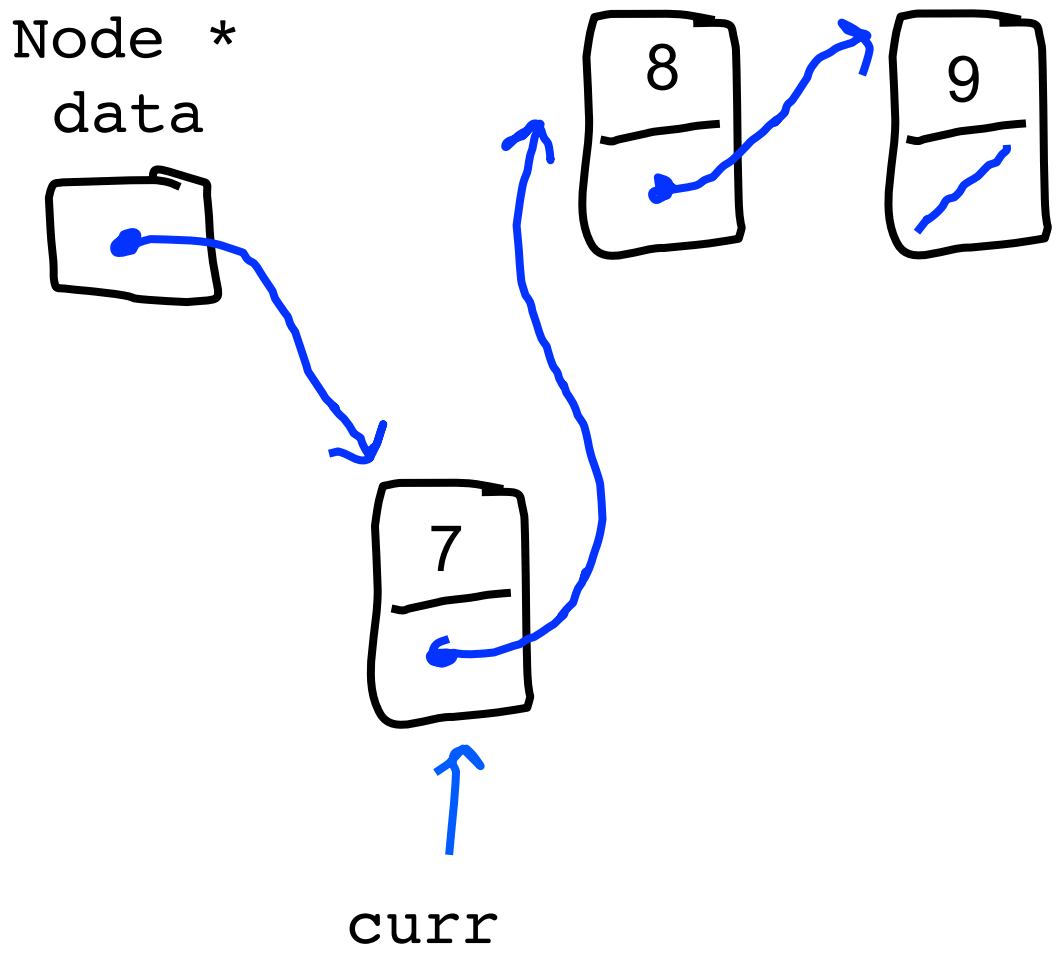

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
       curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



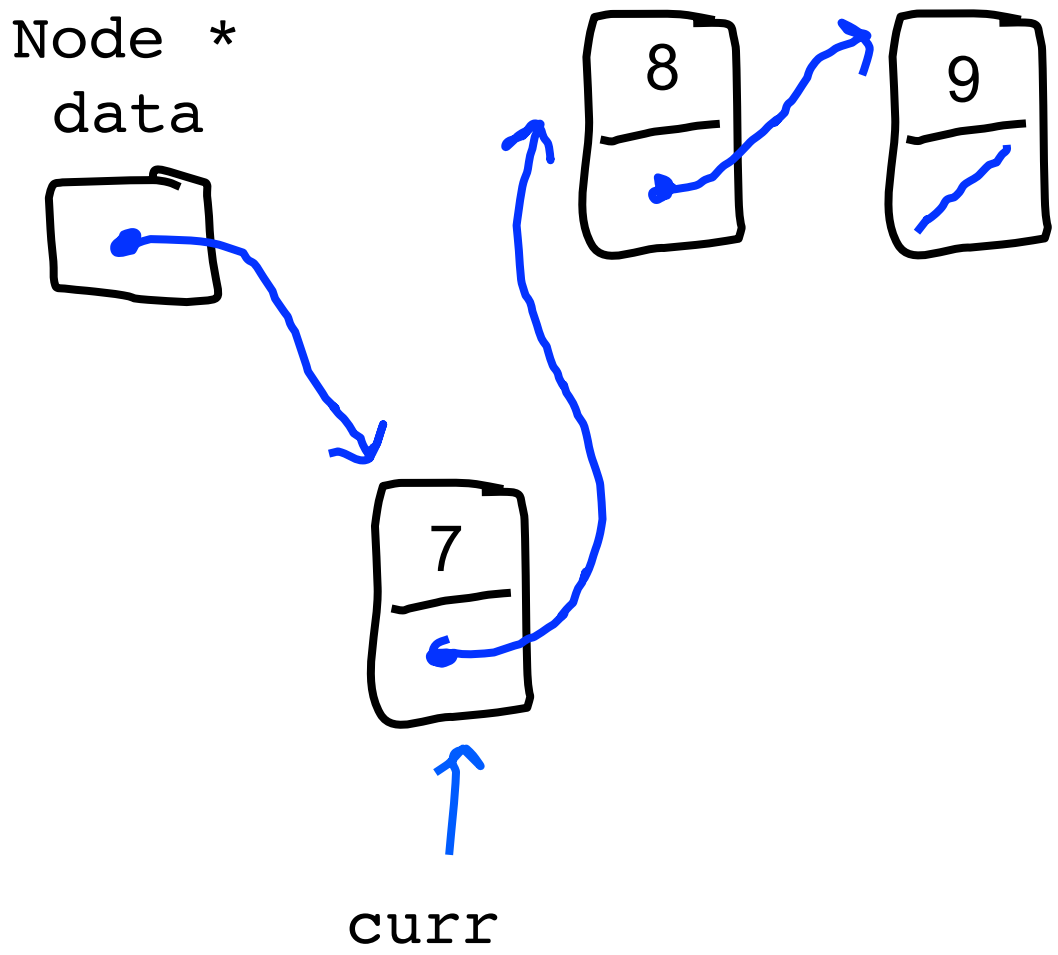

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



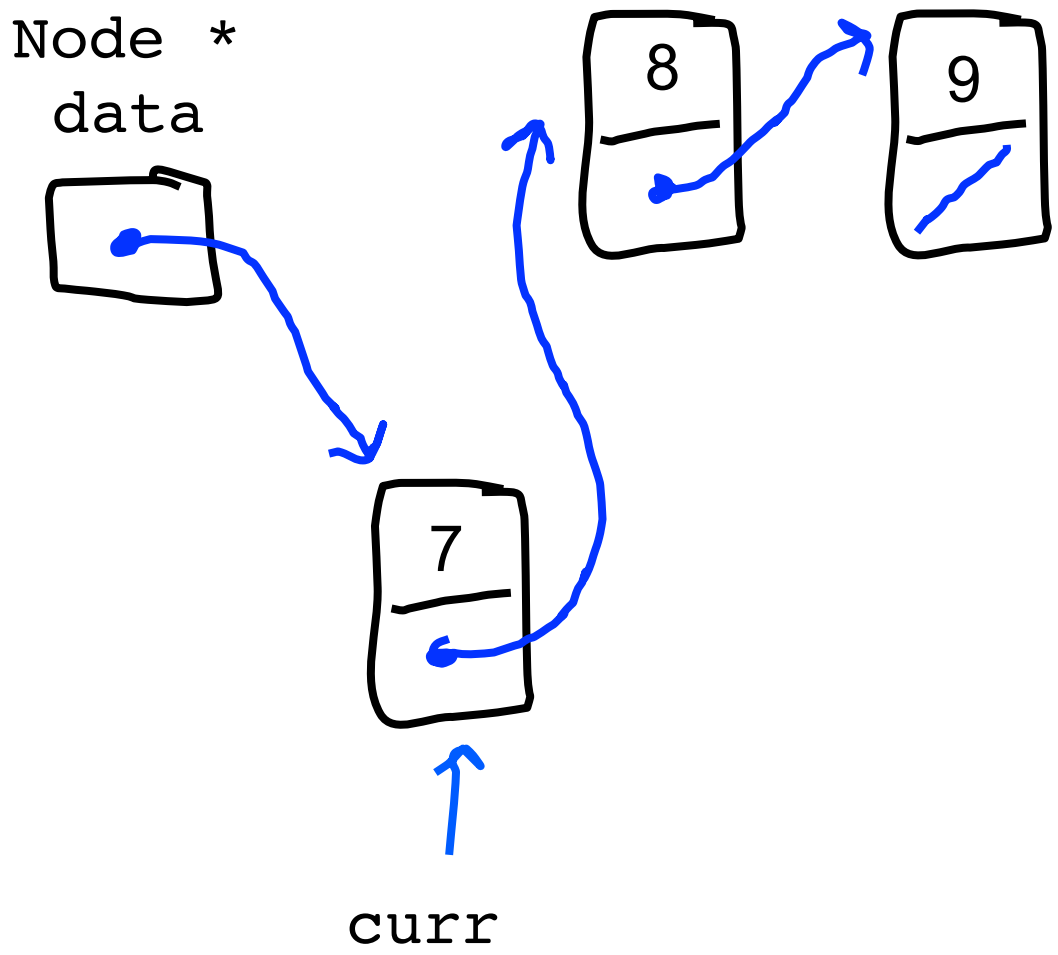

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



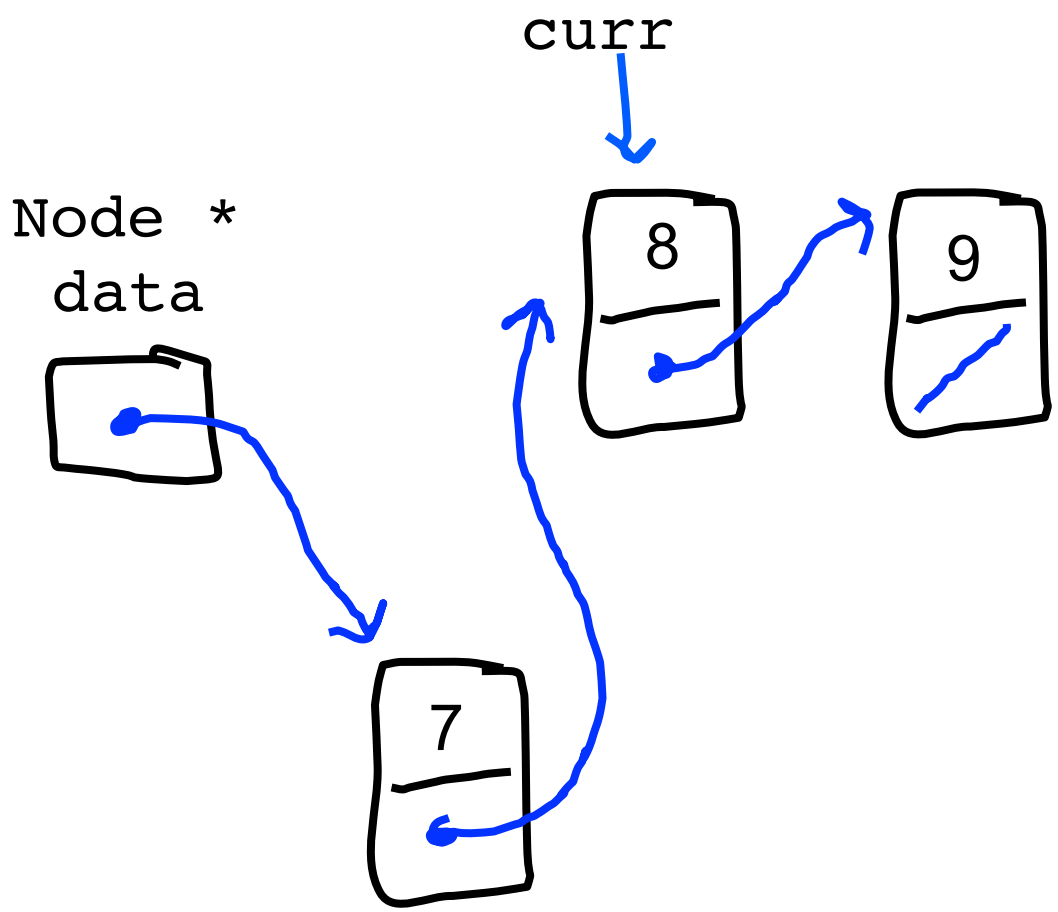

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



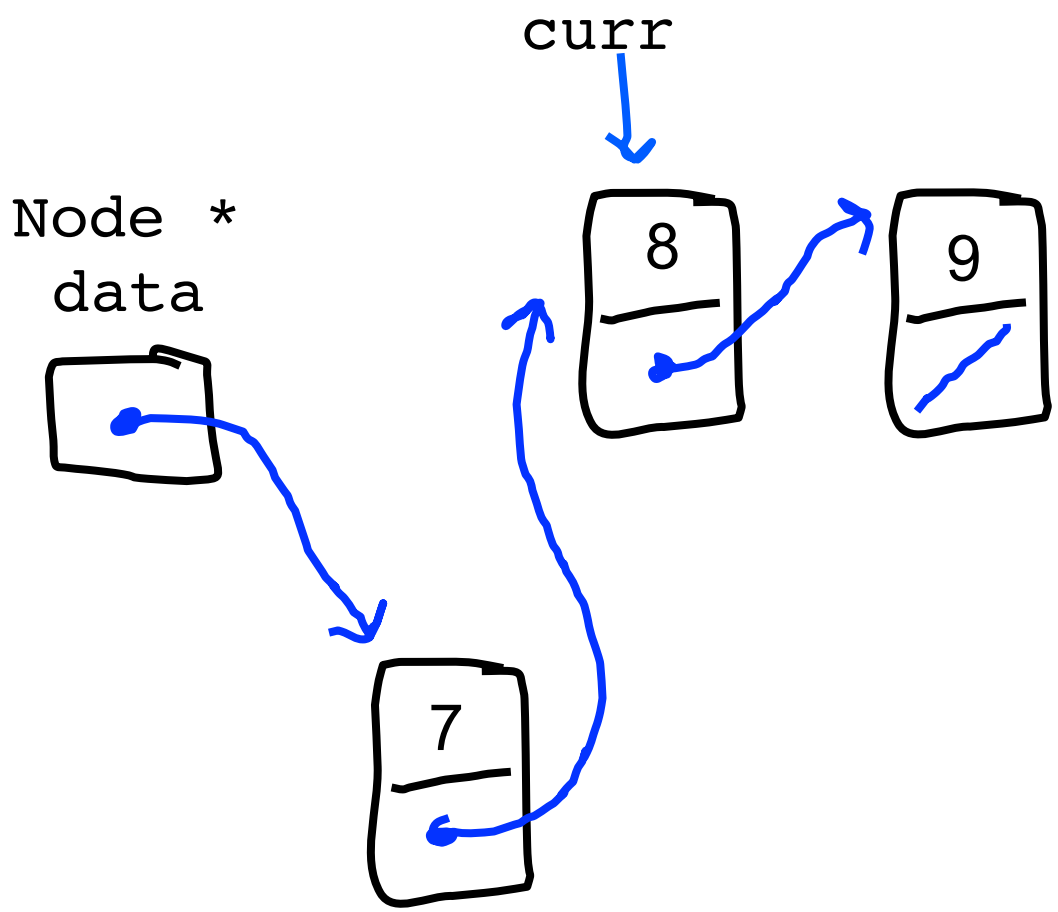

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



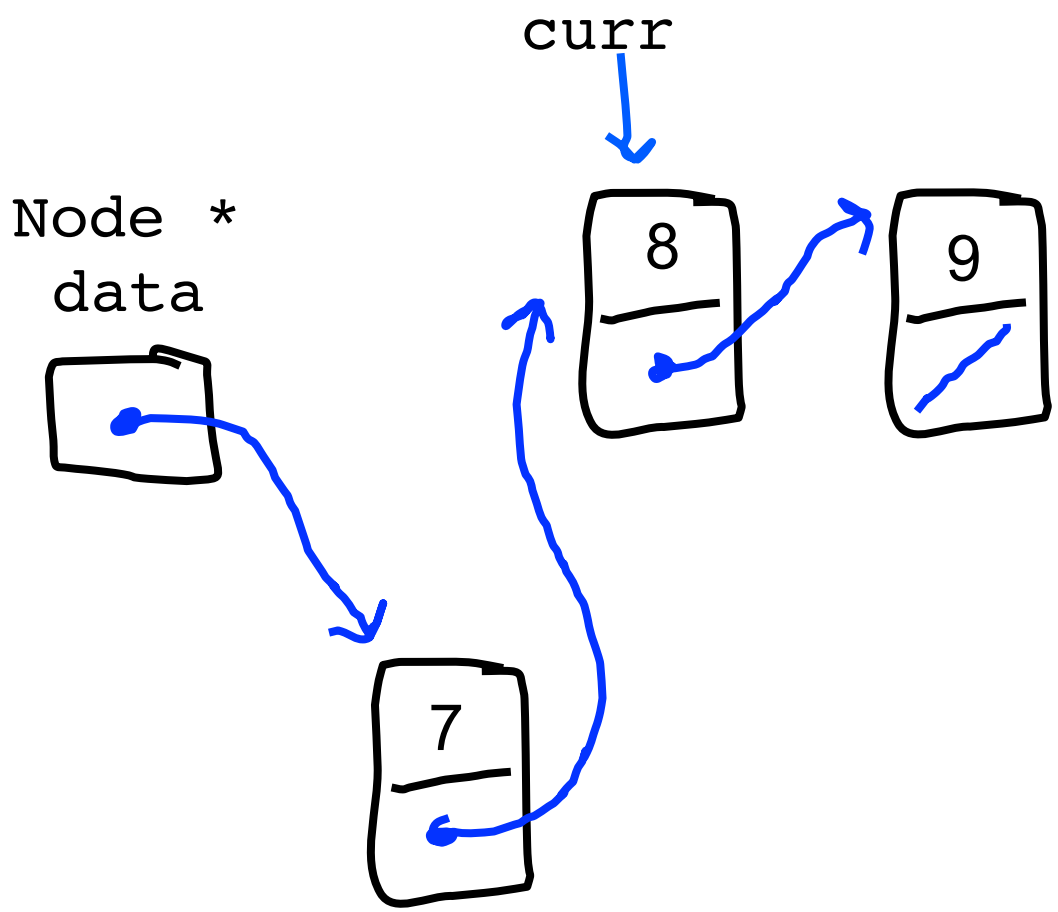

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



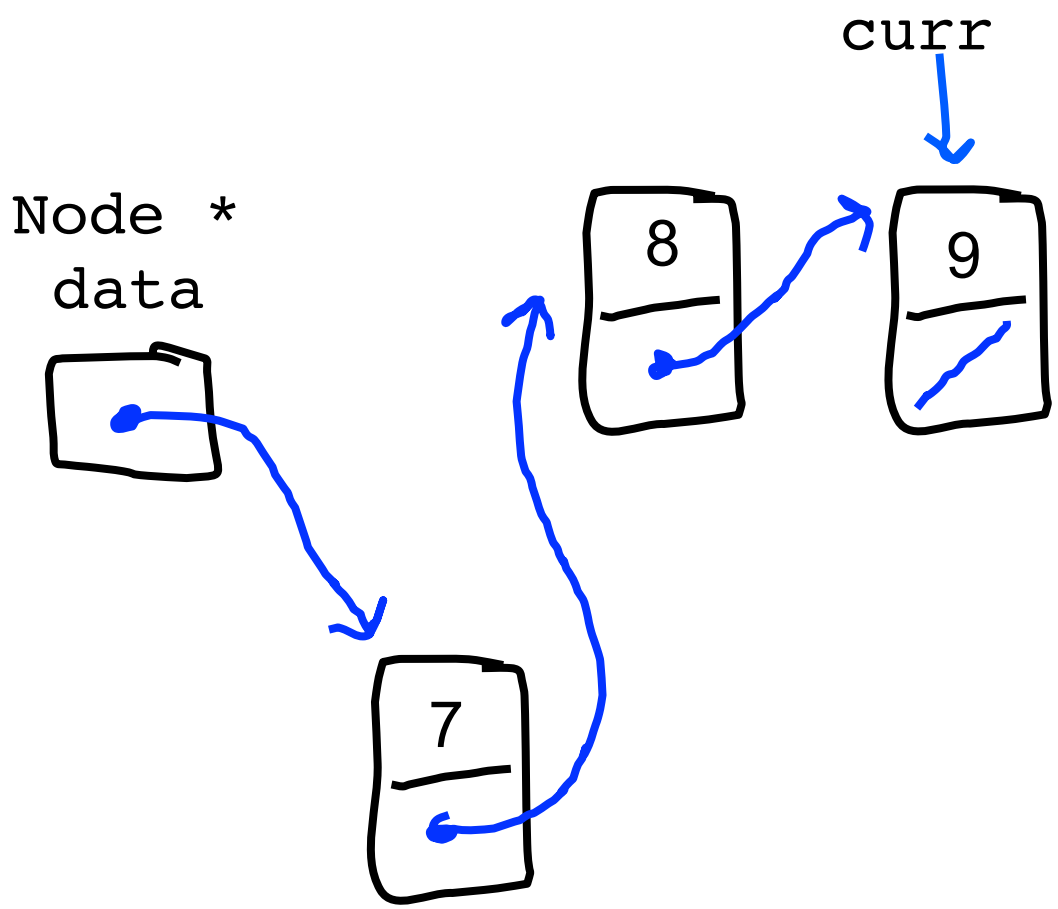

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



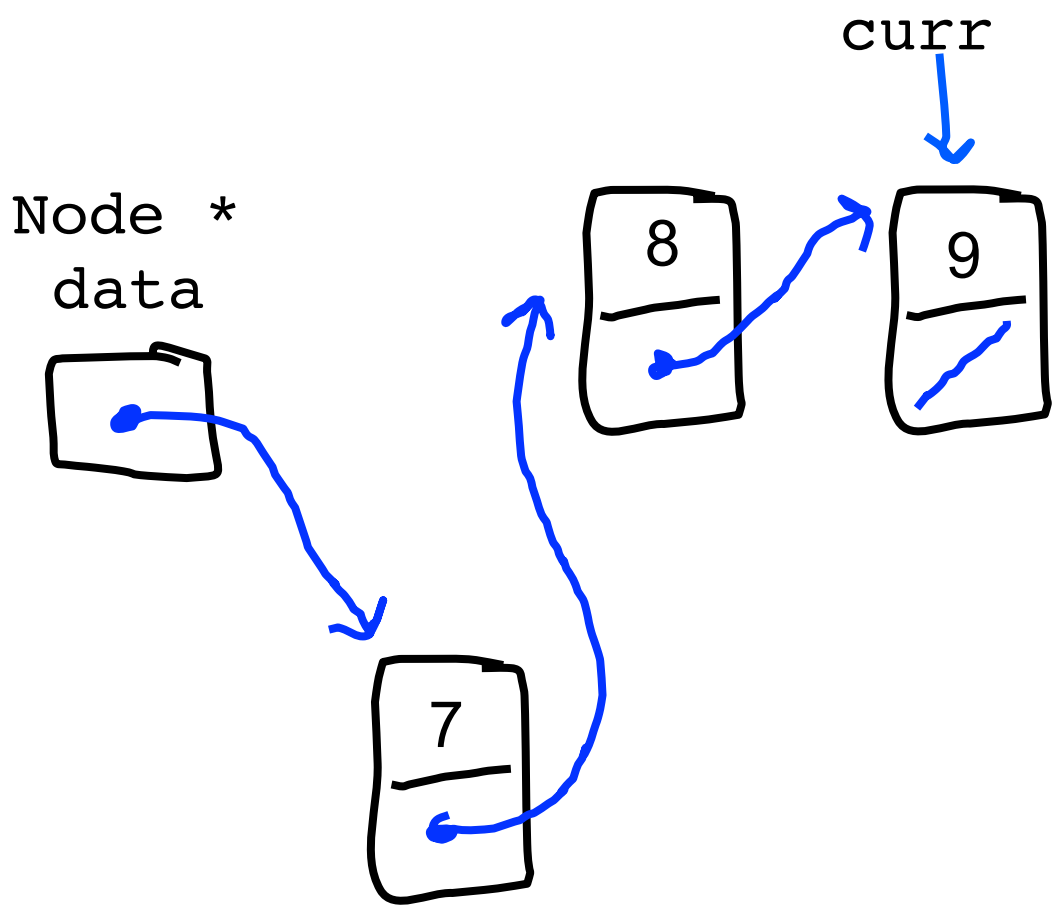

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



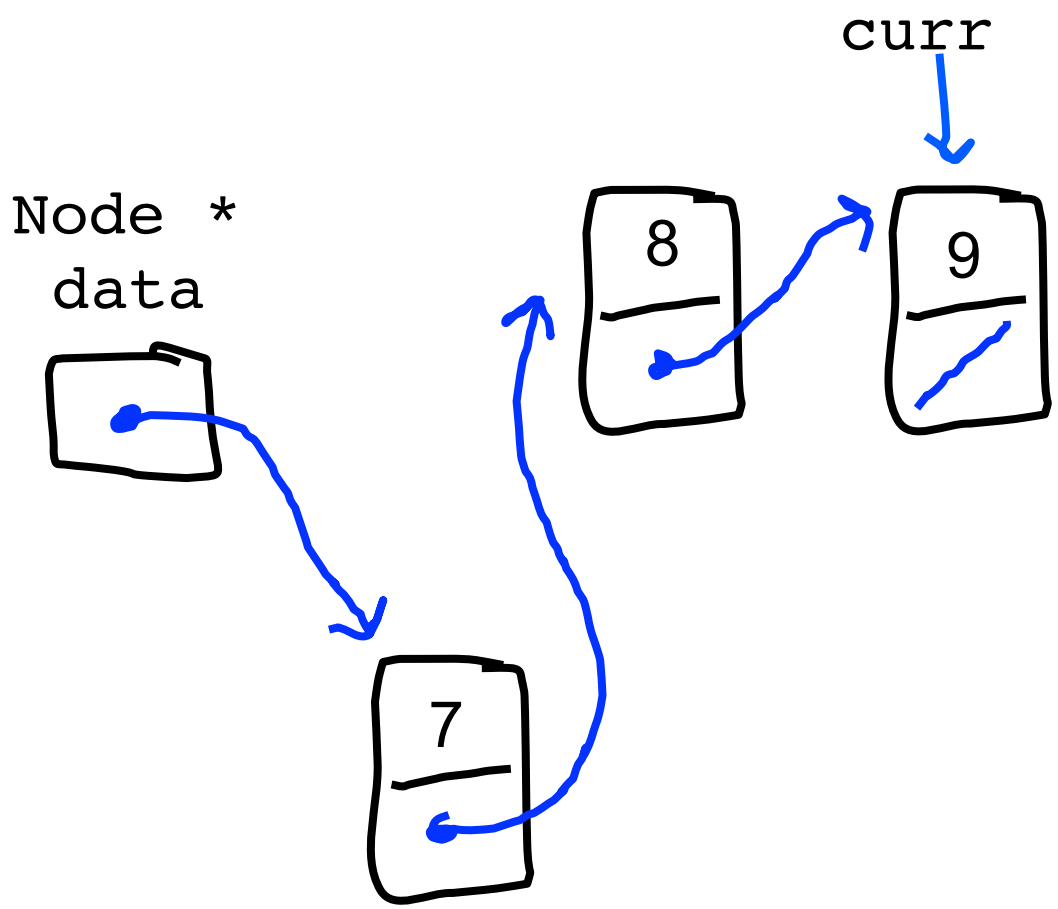

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?




```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

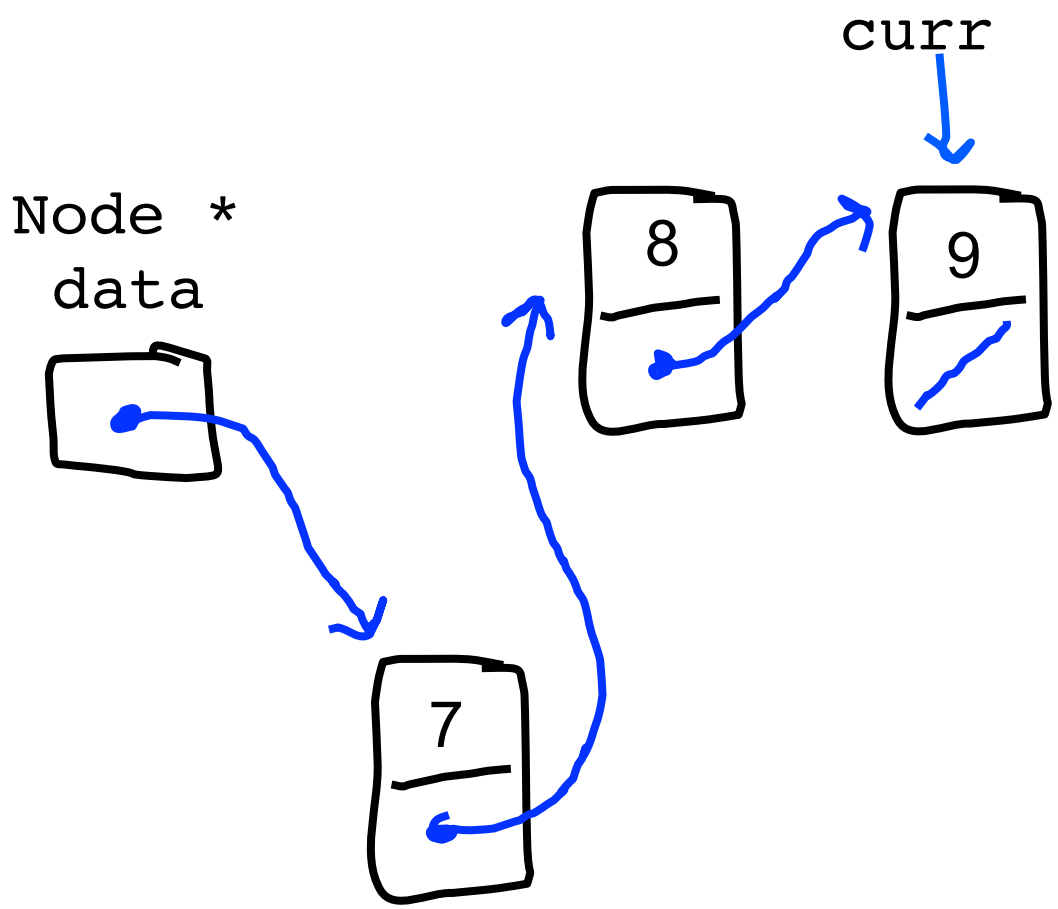
Queue Dequeue?




```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
       curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

toReturn: 9

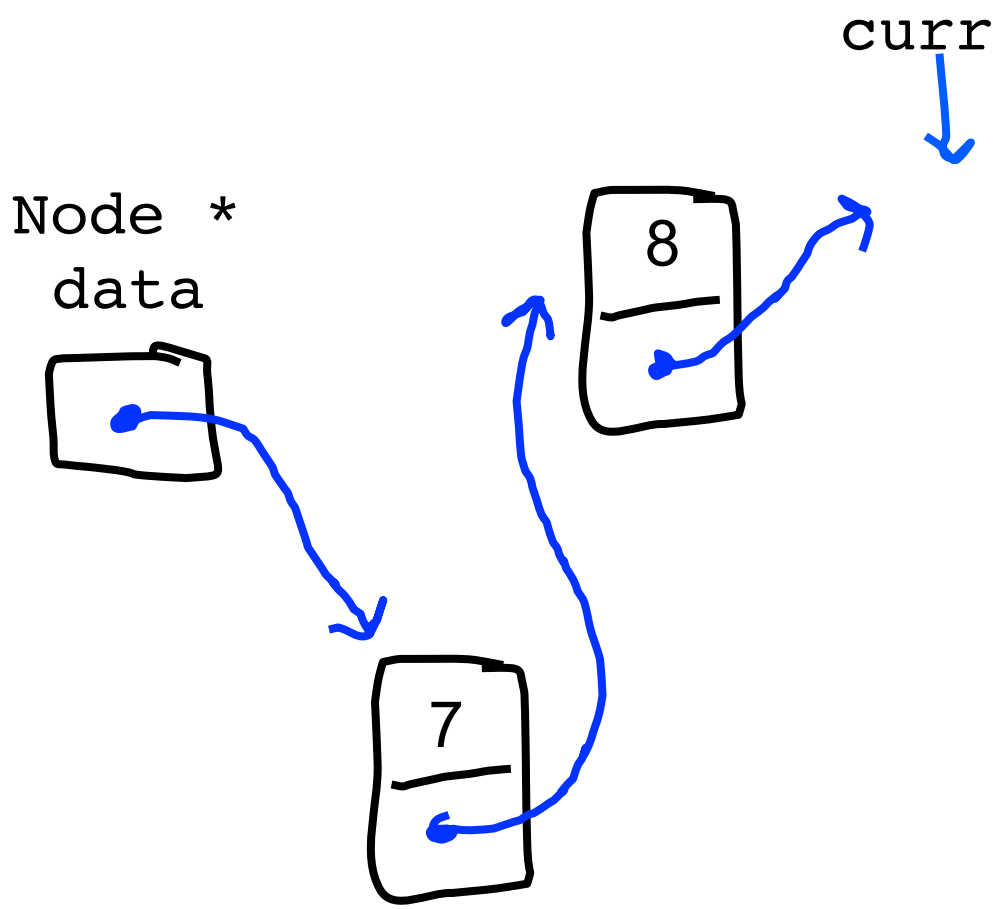
Queue Dequeue?




```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
       curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

toReturn: 9

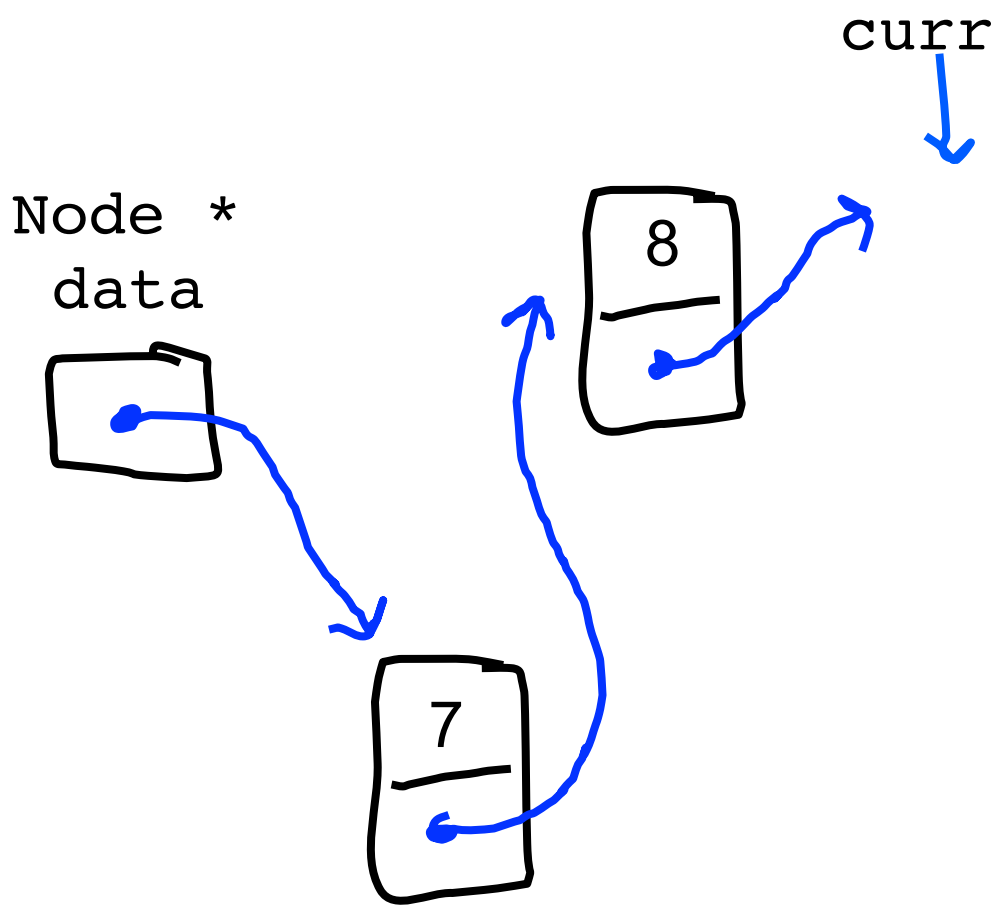
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

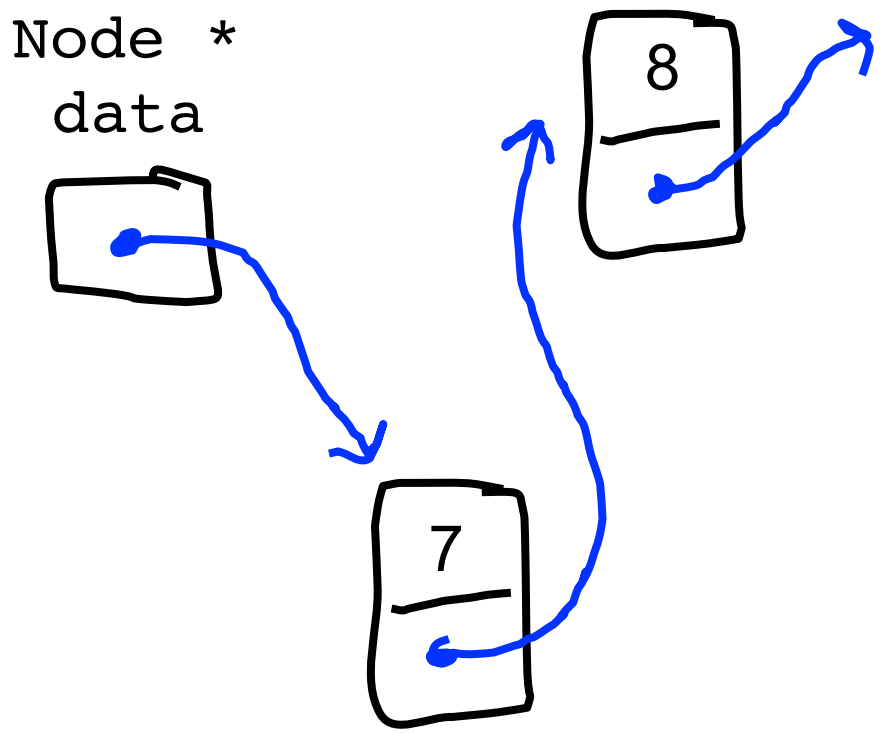
Queue Dequeue?



toReturn: 9

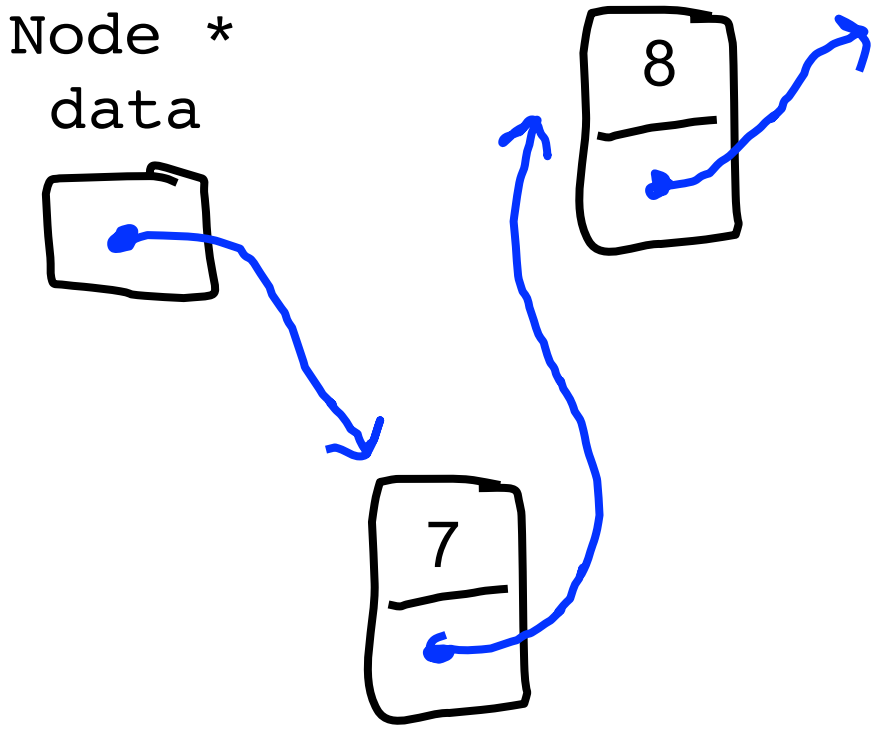
```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
       curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?




```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
       curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

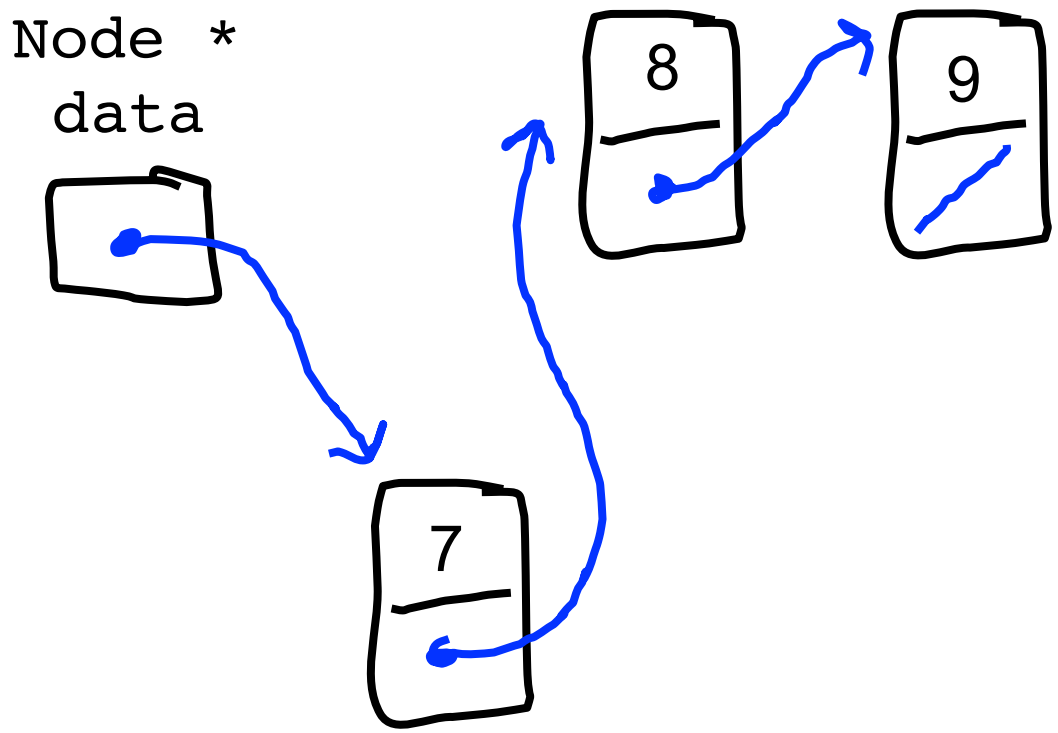
Queue Dequeue?



```
int SlowQueue::dequeue() {
    // find and delete the last node
    Node * curr = data;
    while(curr->link != NULL) {
        curr = curr->link;
    }
    int toReturn = curr->value;
    delete curr;
    return toReturn;
}
```

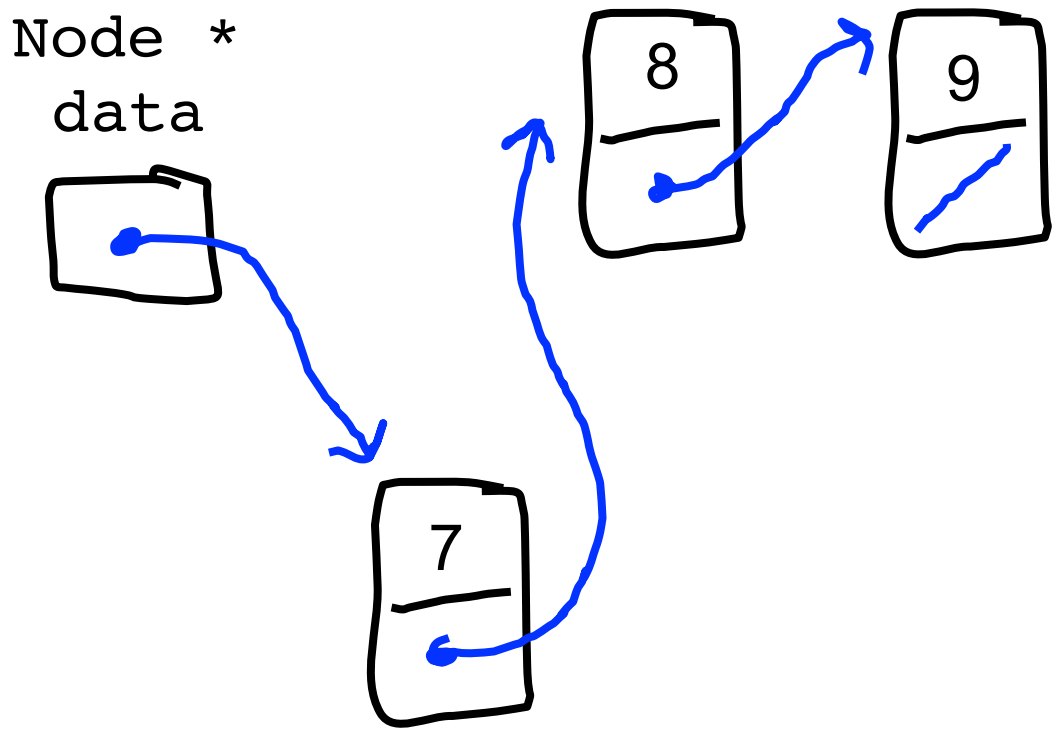
Lets try that again…

Queue Dequeue?



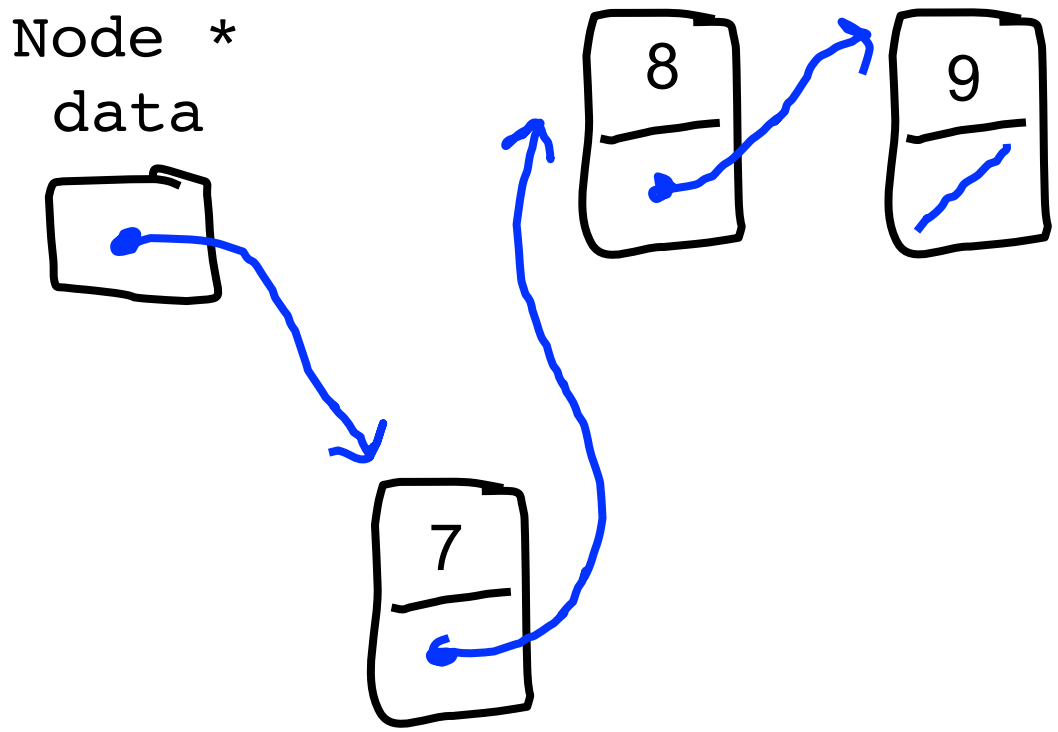

```
int SlowQueue::dequeue() {
    // special case removing first node
    if(data->link == NULL) {
        int toReturn = data->value;
        delete data;
        data = NULL;
        return toReturn;
    }

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



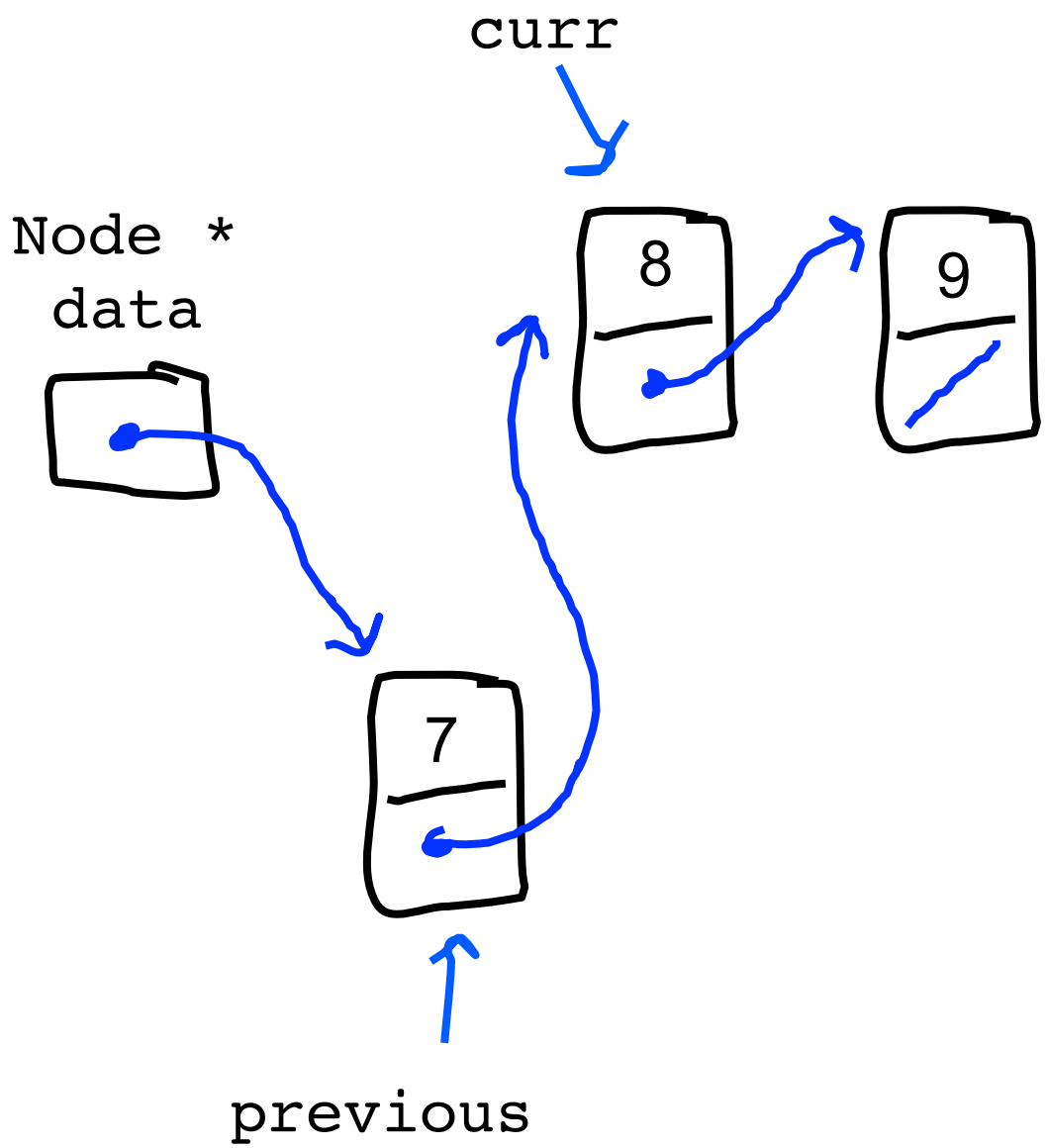

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



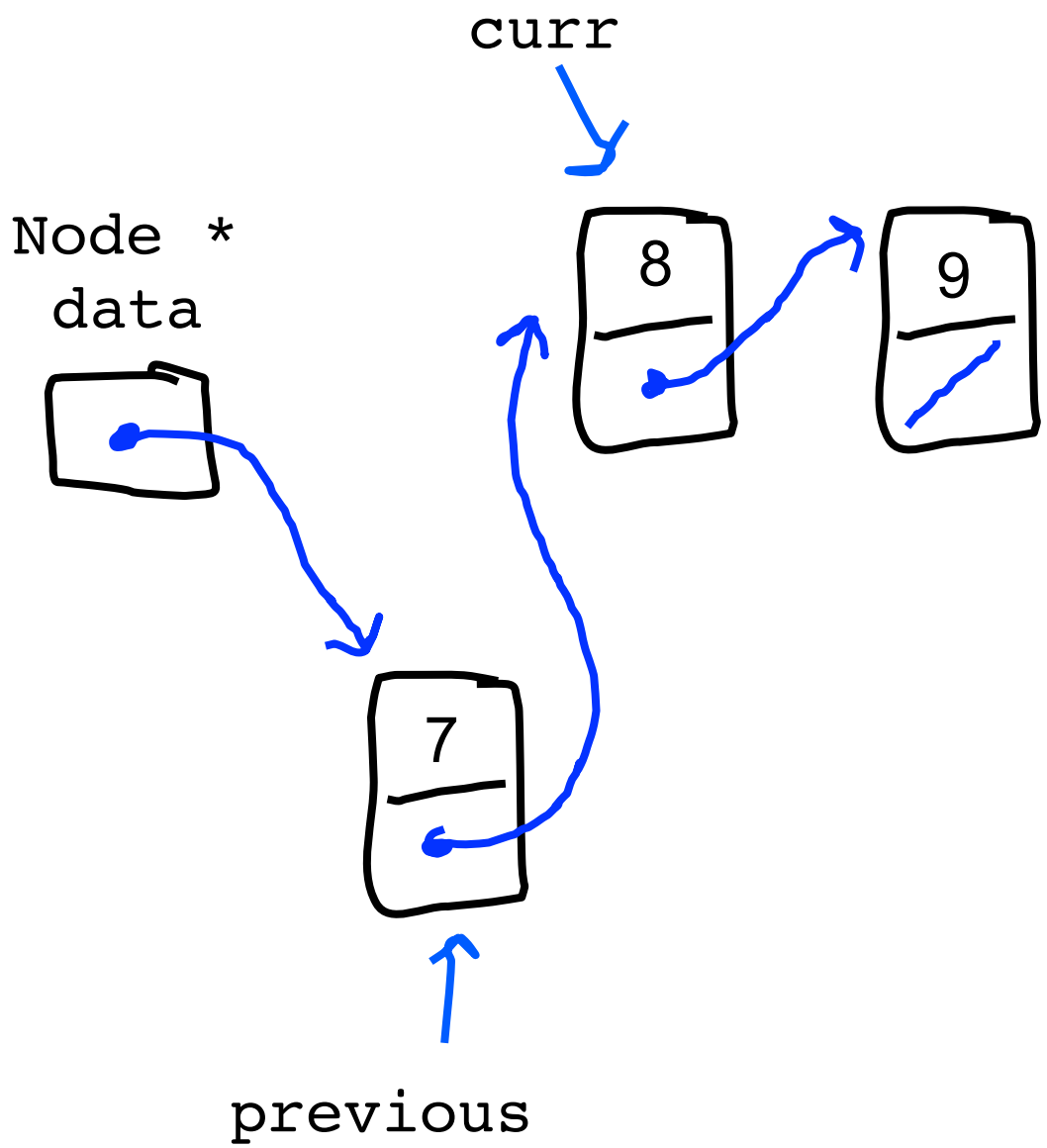

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



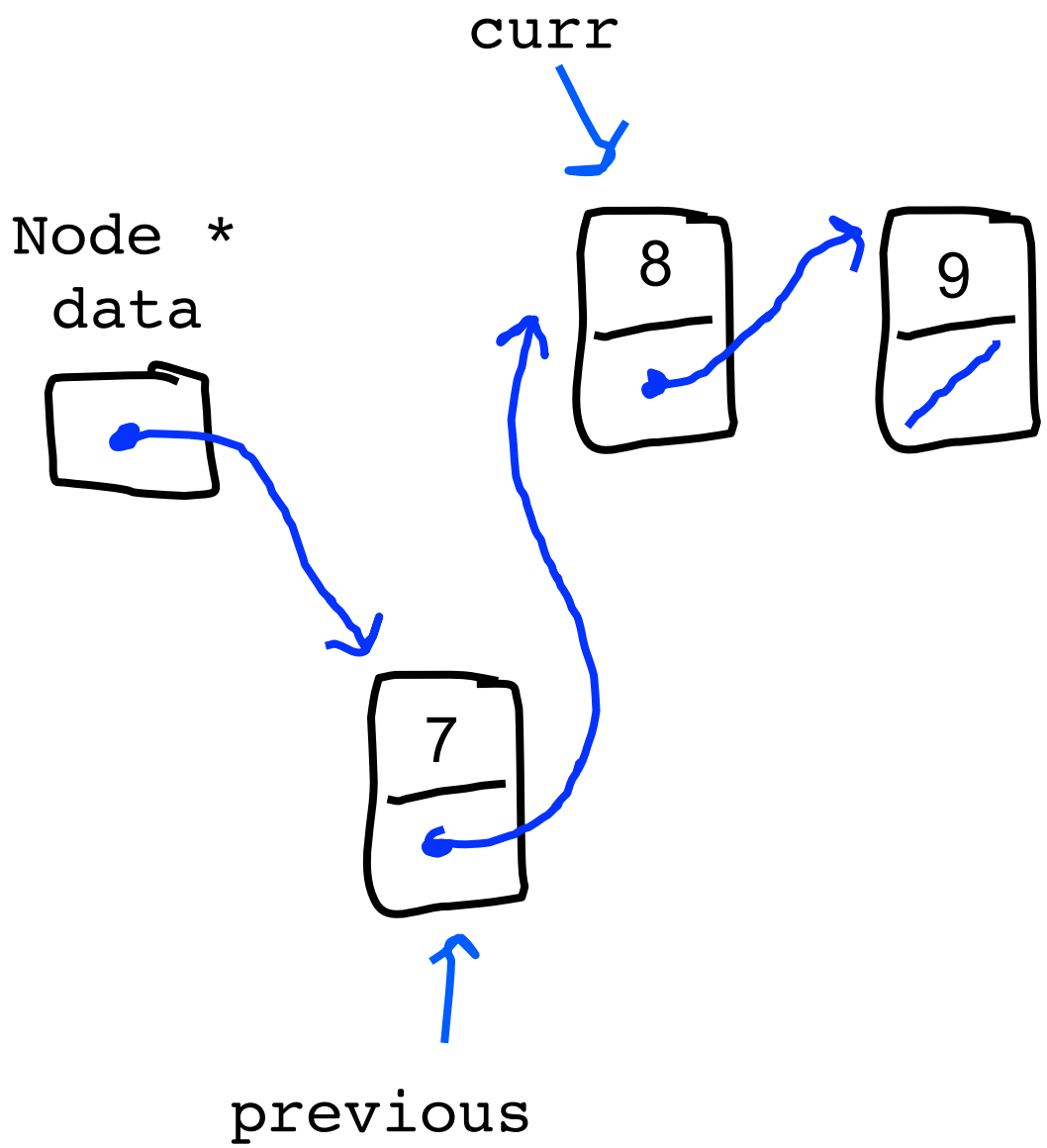

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



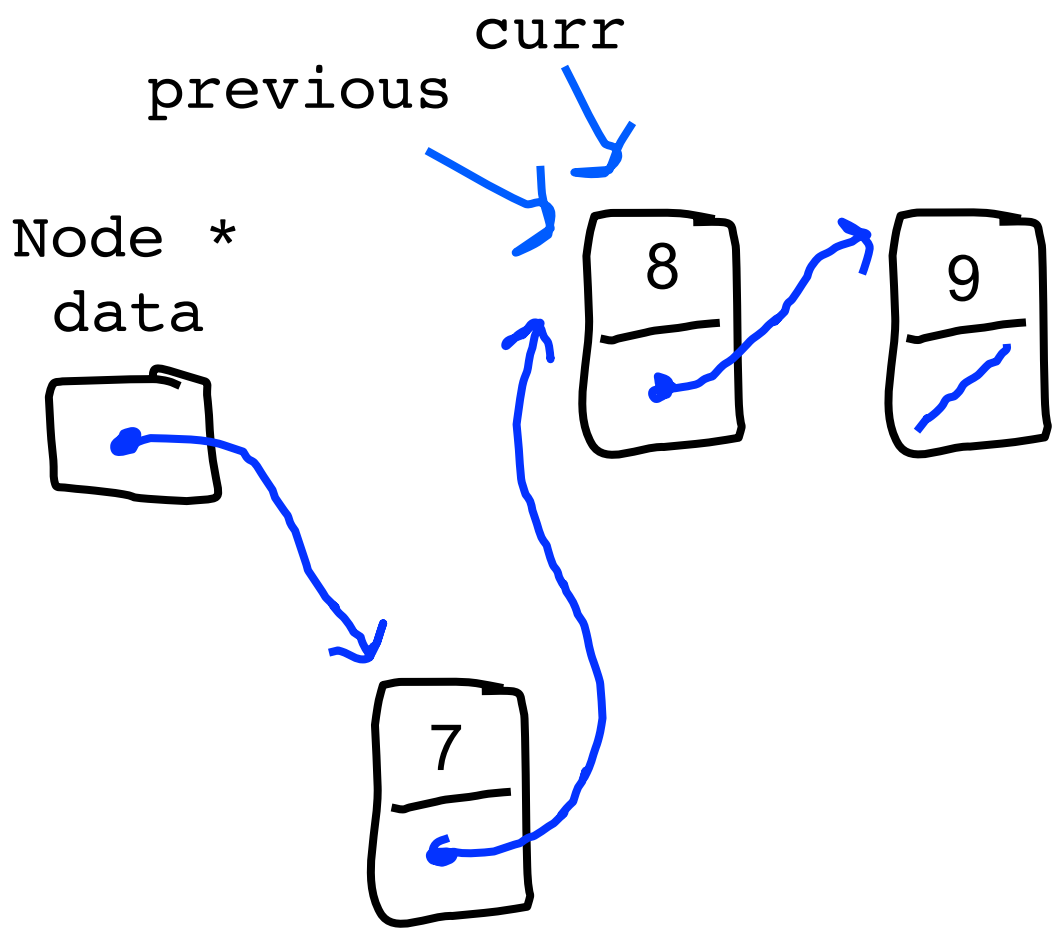

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



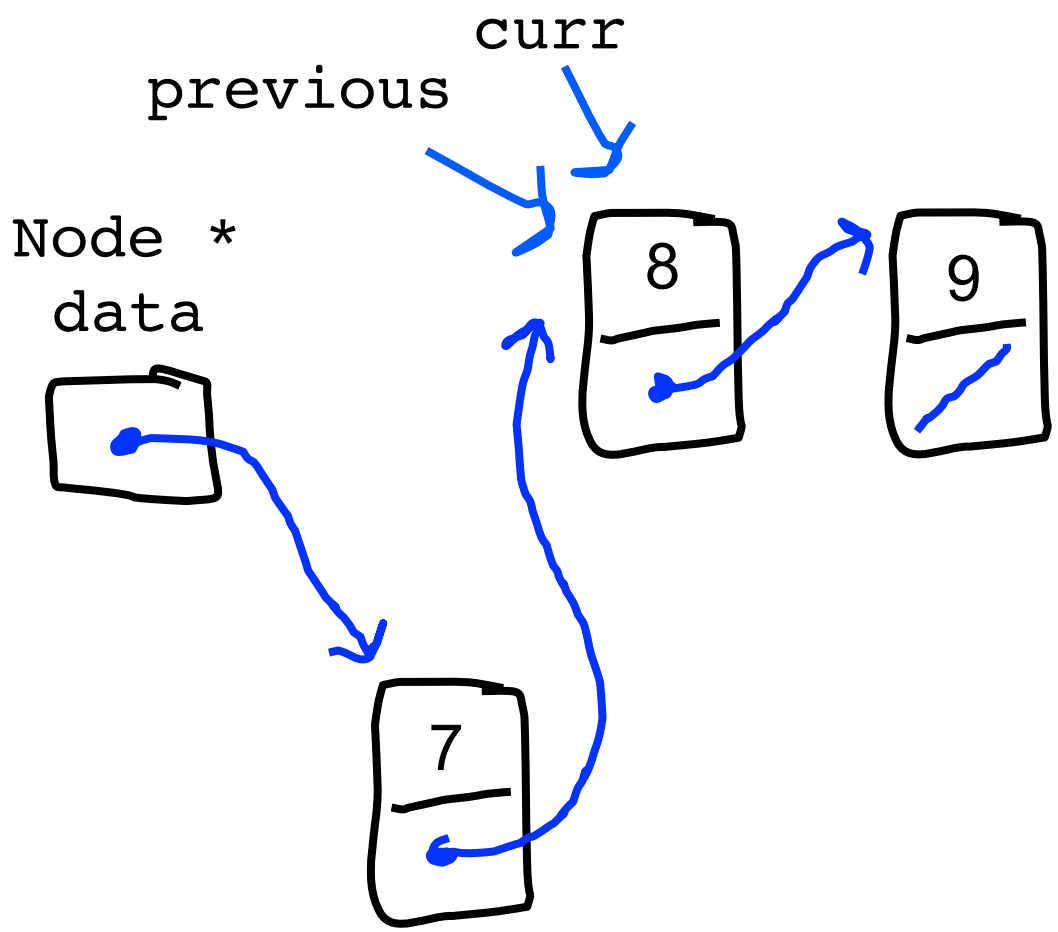

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



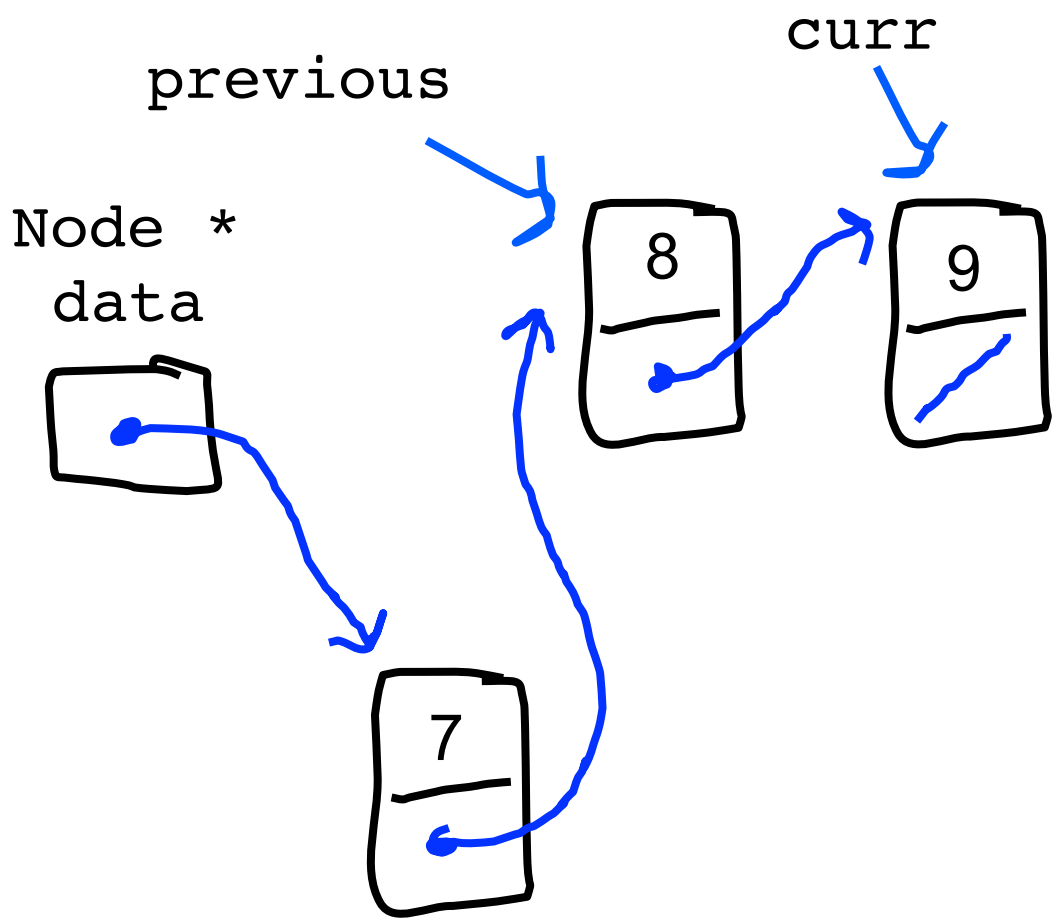

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



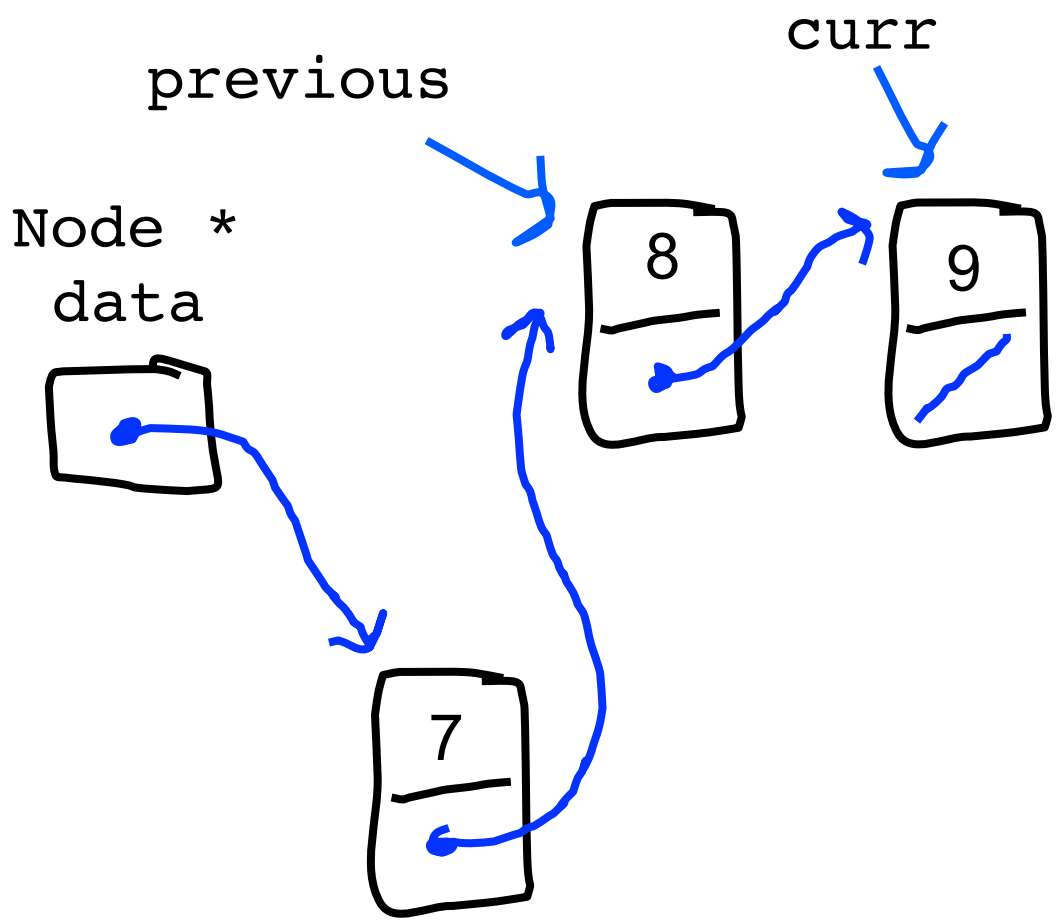

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



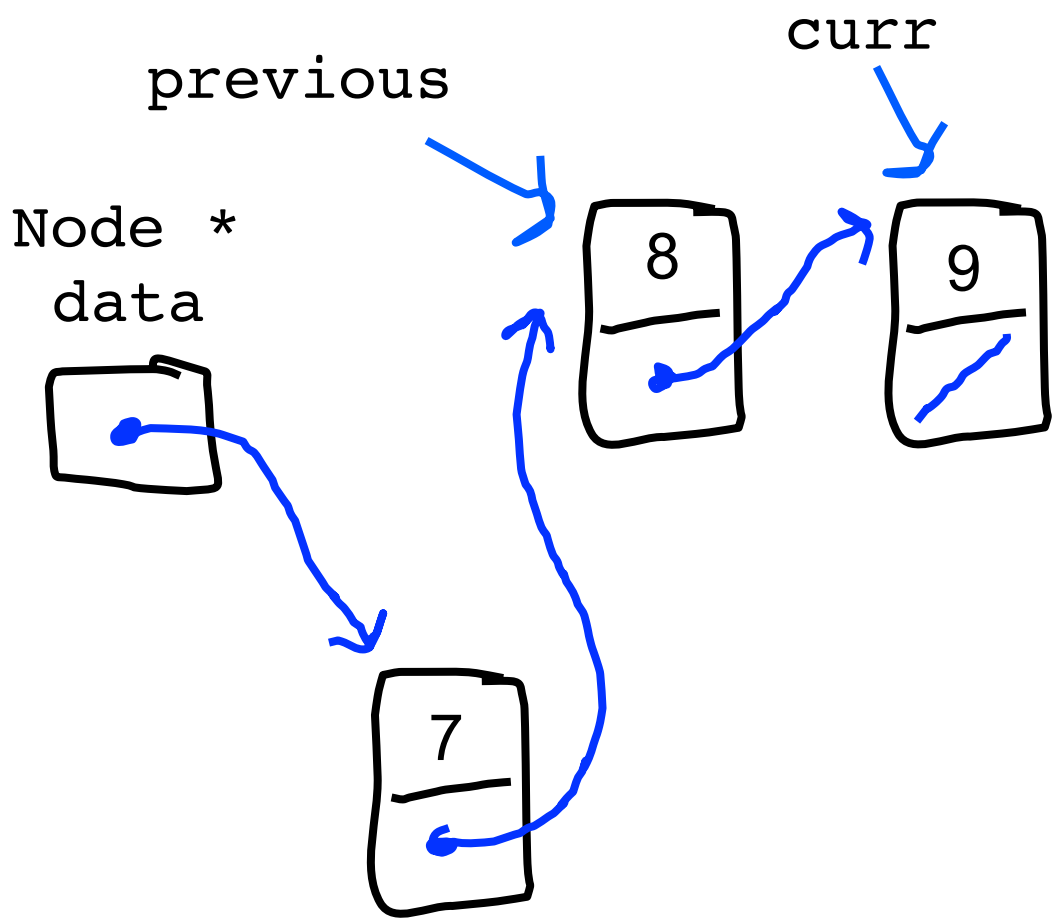

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?




```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

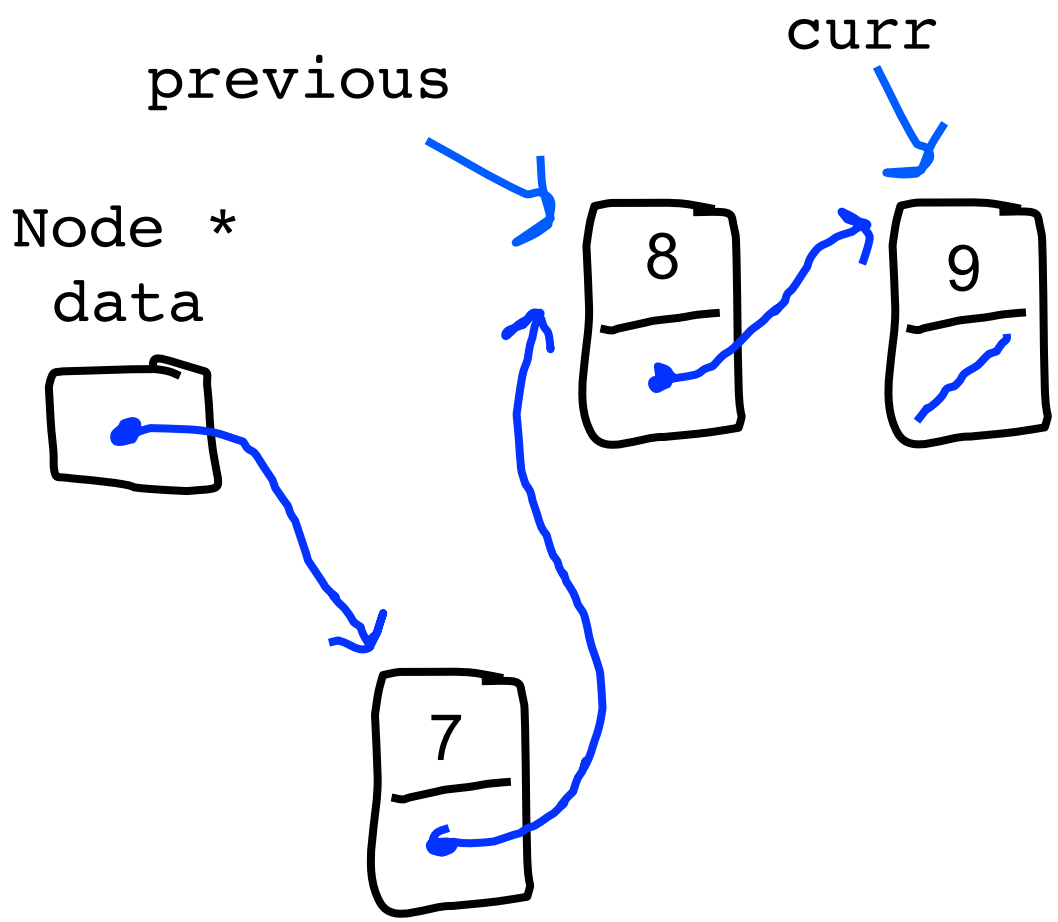
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

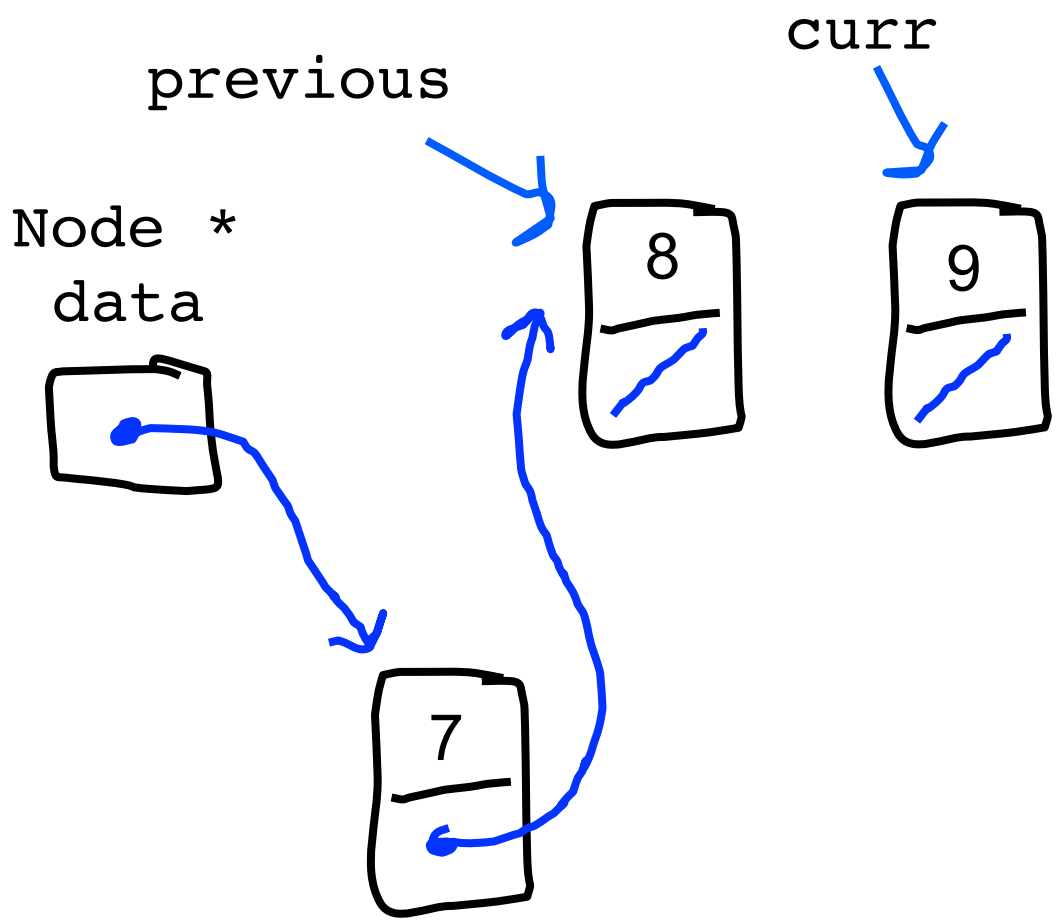
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

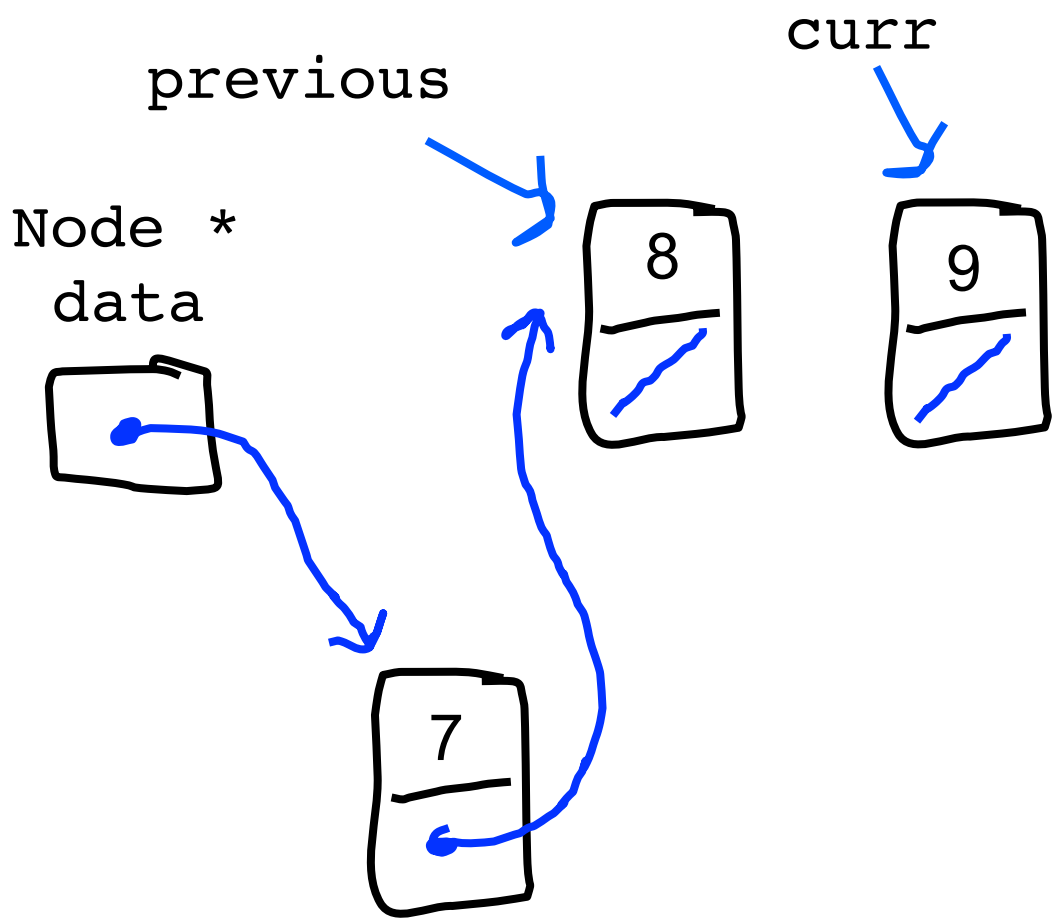
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

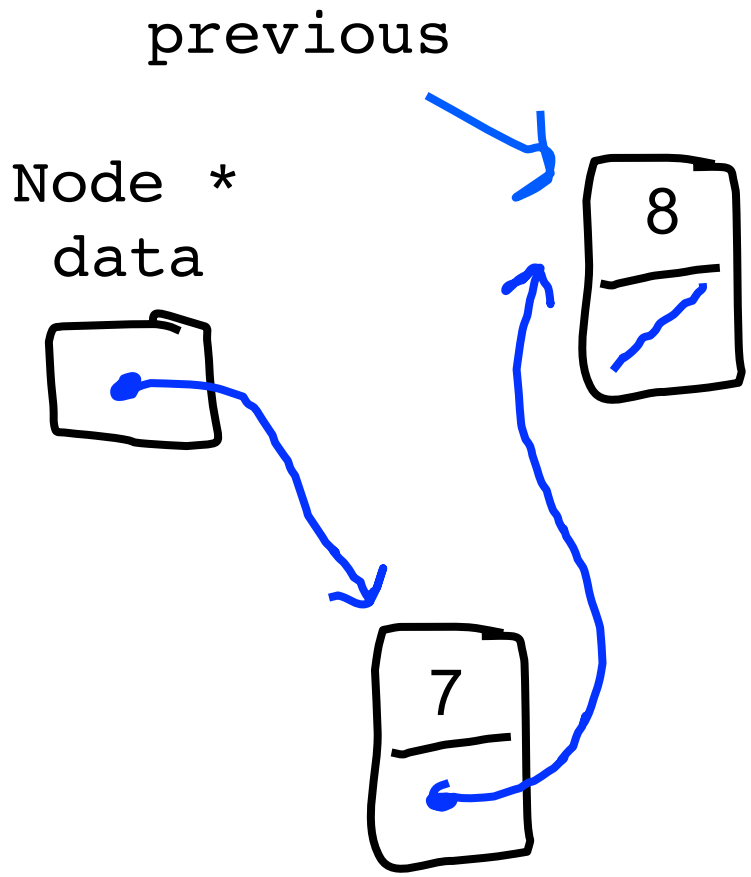
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

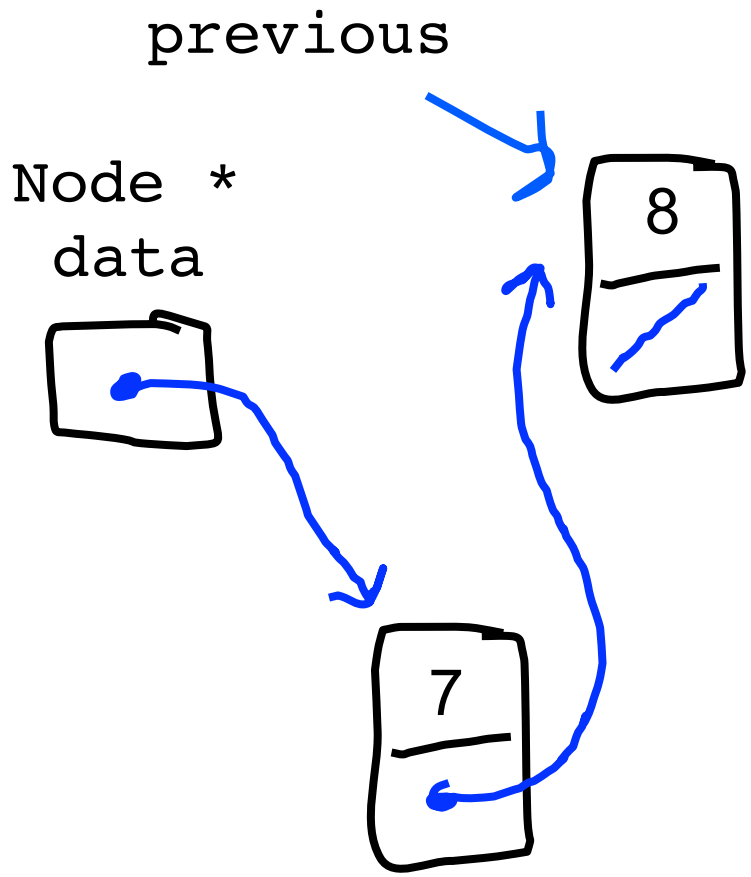
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

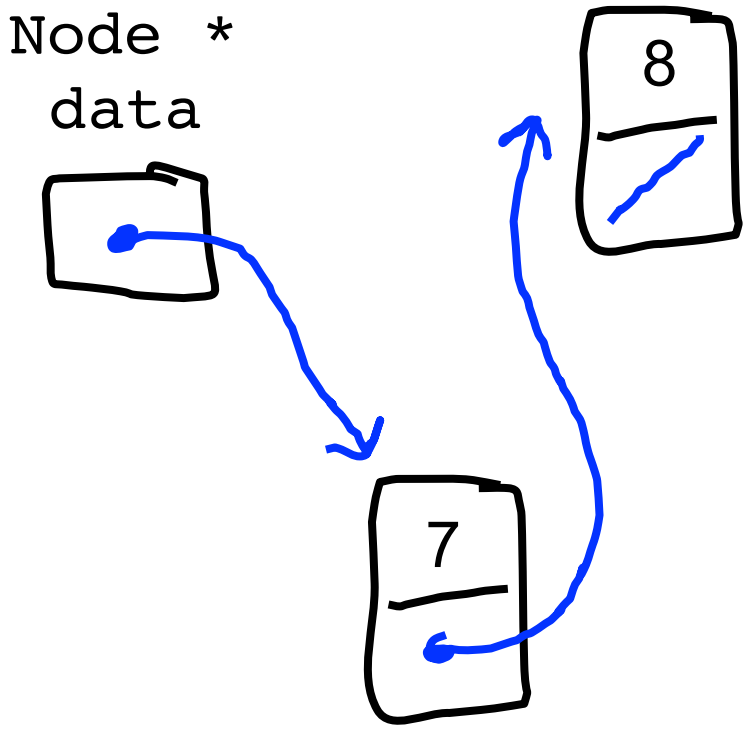
Queue Dequeue?



toReturn: 9

```
int SlowQueue::dequeue() {
    // special case removing first node
    ...

    // if there is more than one node
    Node * curr = data->link;
    Node * previous = data;
    while(curr->link != NULL) {
        previous = curr;
        curr = curr->link;
    }
    int toReturn = curr->value;
    previous->link = NULL;
    delete curr;
    return toReturn;
}
```

Queue Dequeue?



$$\mathcal{O}(n)$$

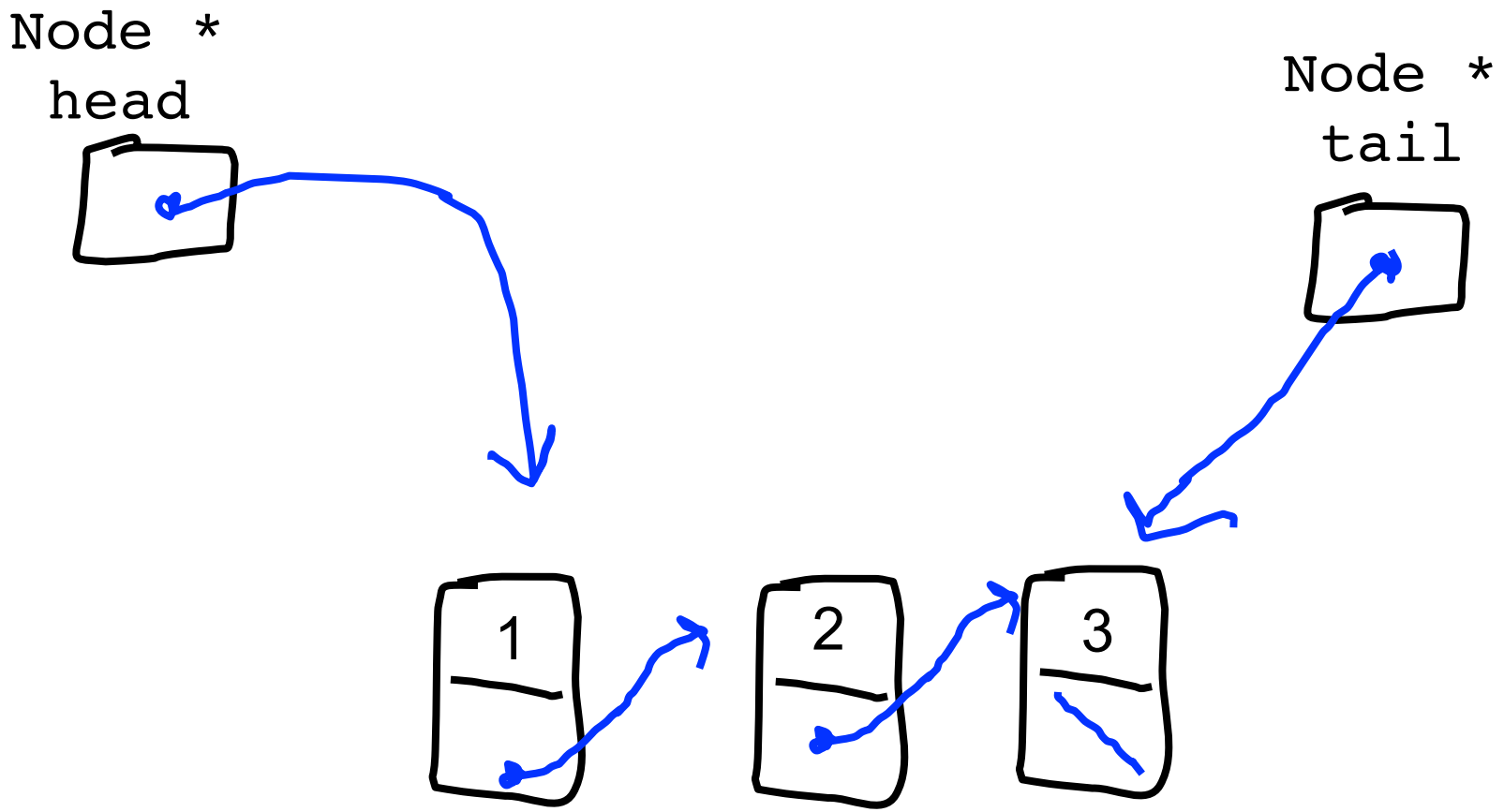
Yay our code is better!

Problem: Dequeue is O(n)

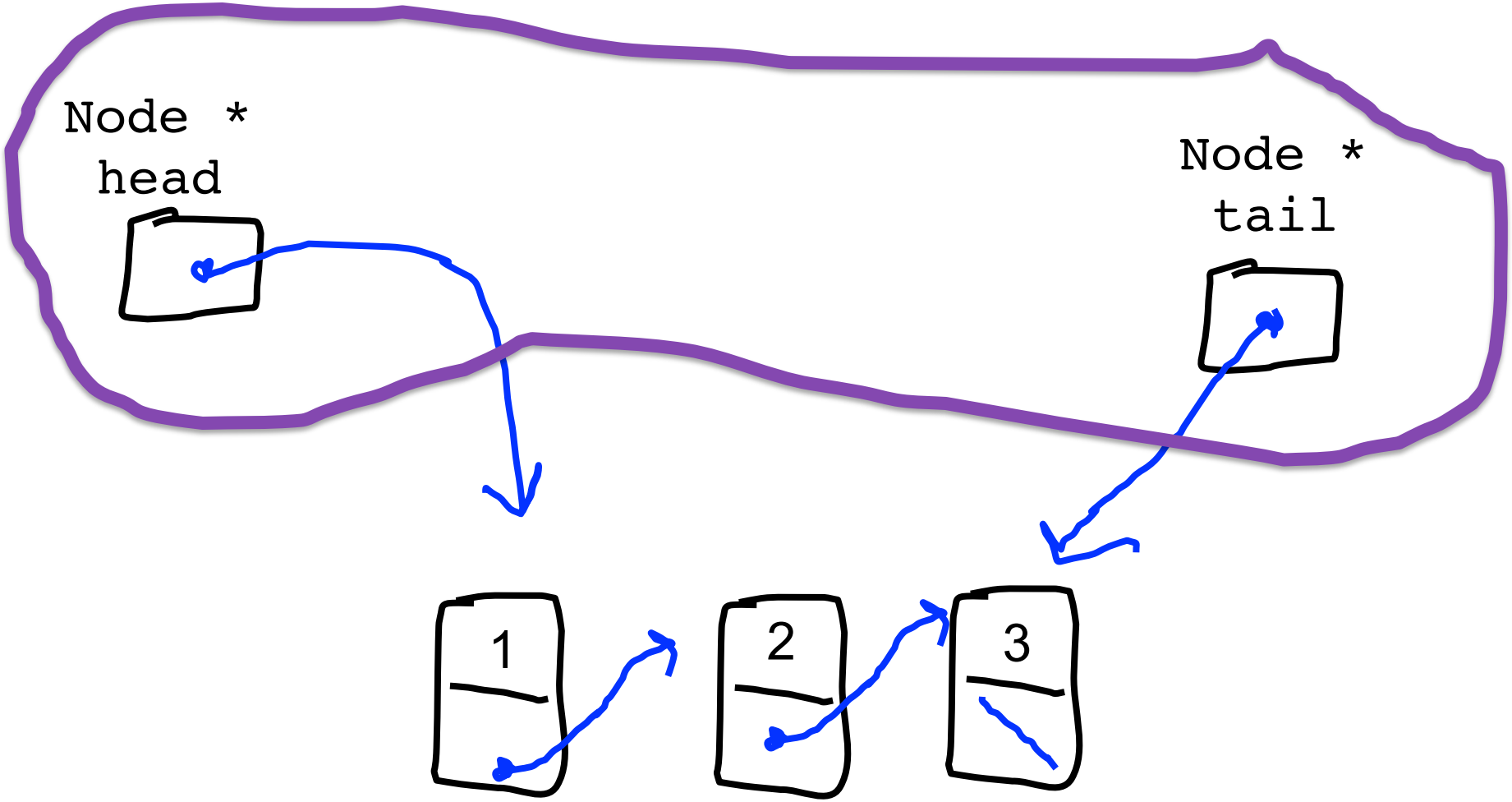
Always a Better Way

Challenge: Design a Queue with O(1) enqueue and dequeue

Actual Queue

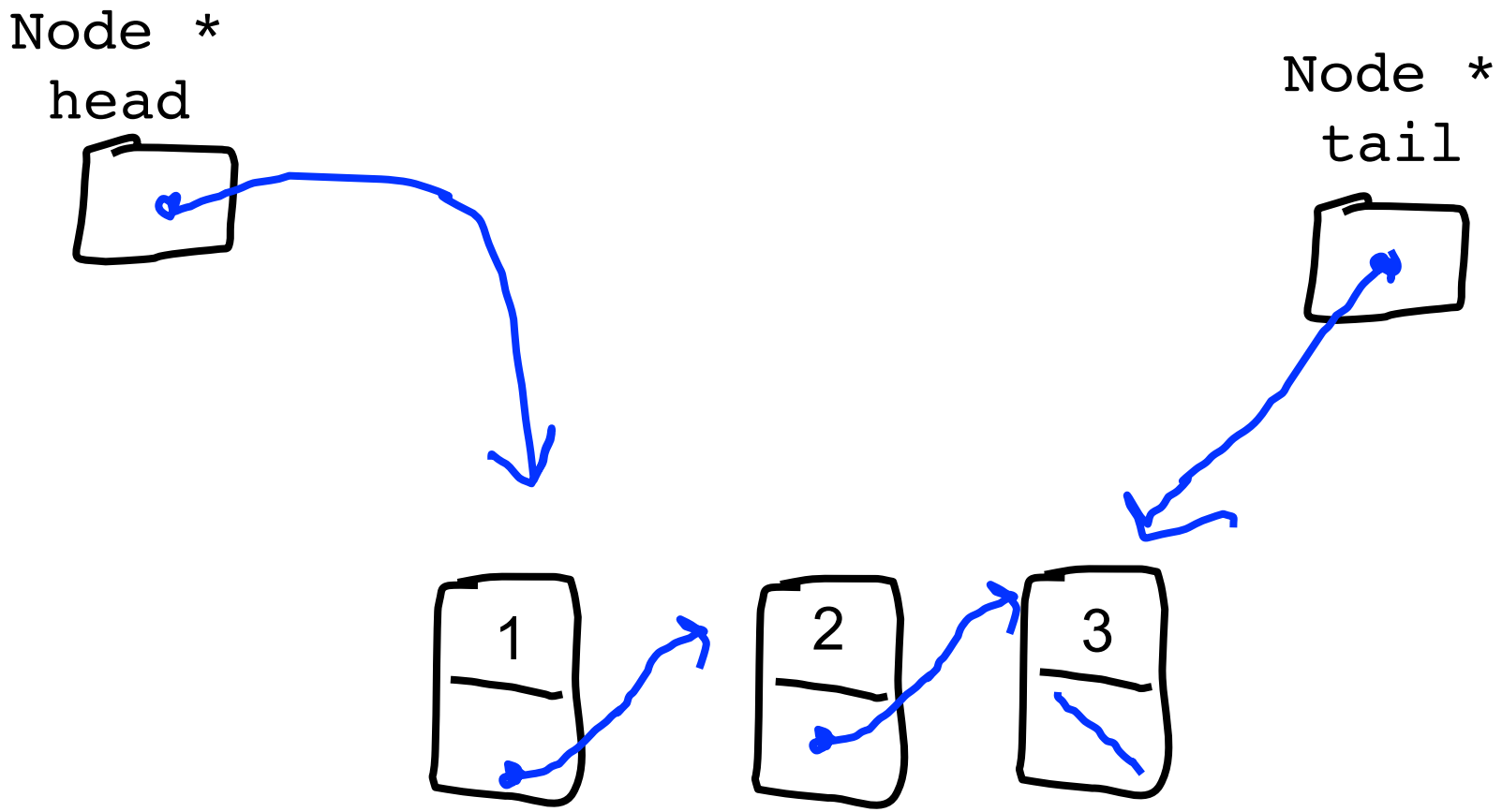


Actual Queue

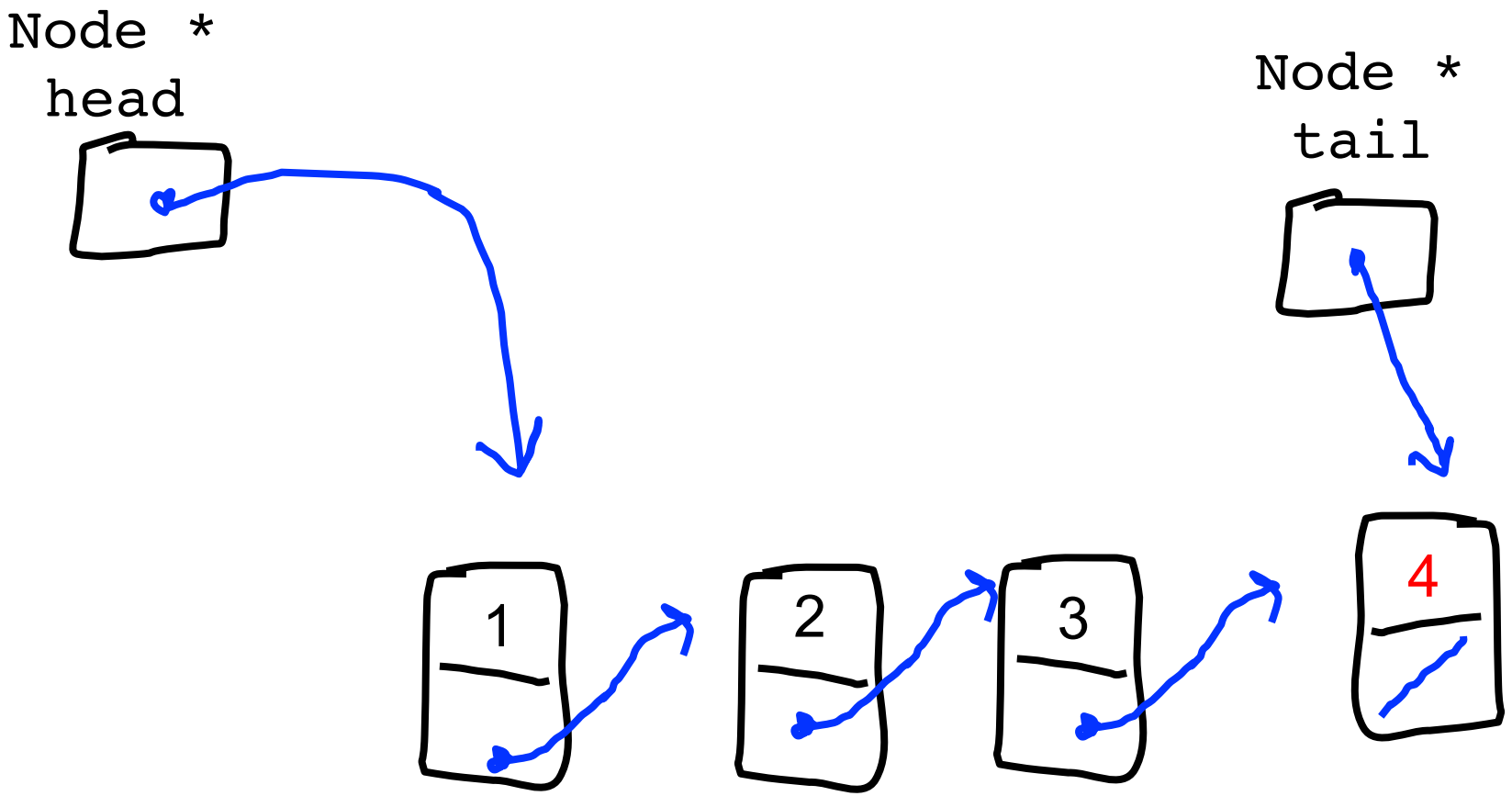


The instance variables of the Queue class consist of two pointers

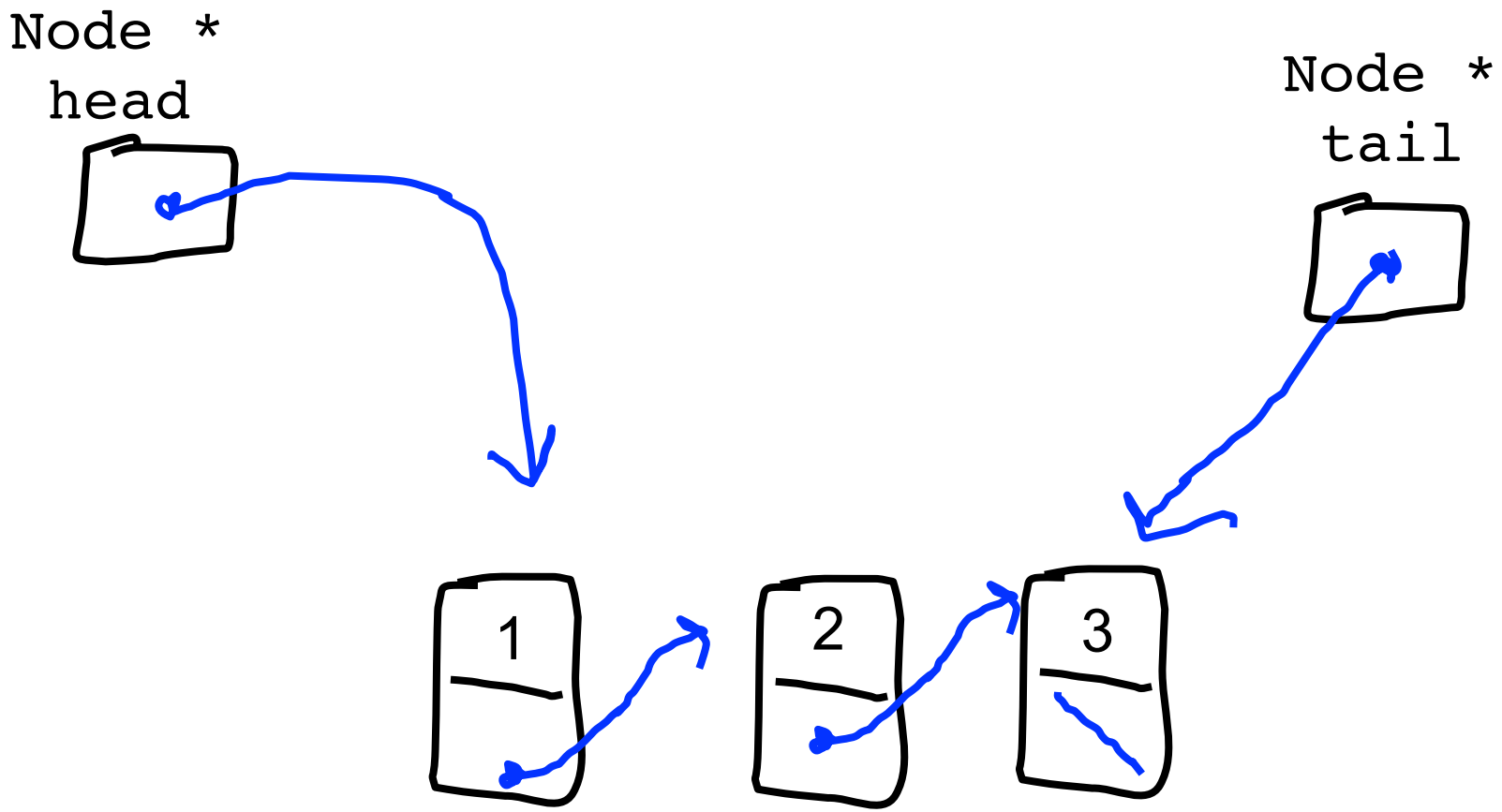
Enqueue



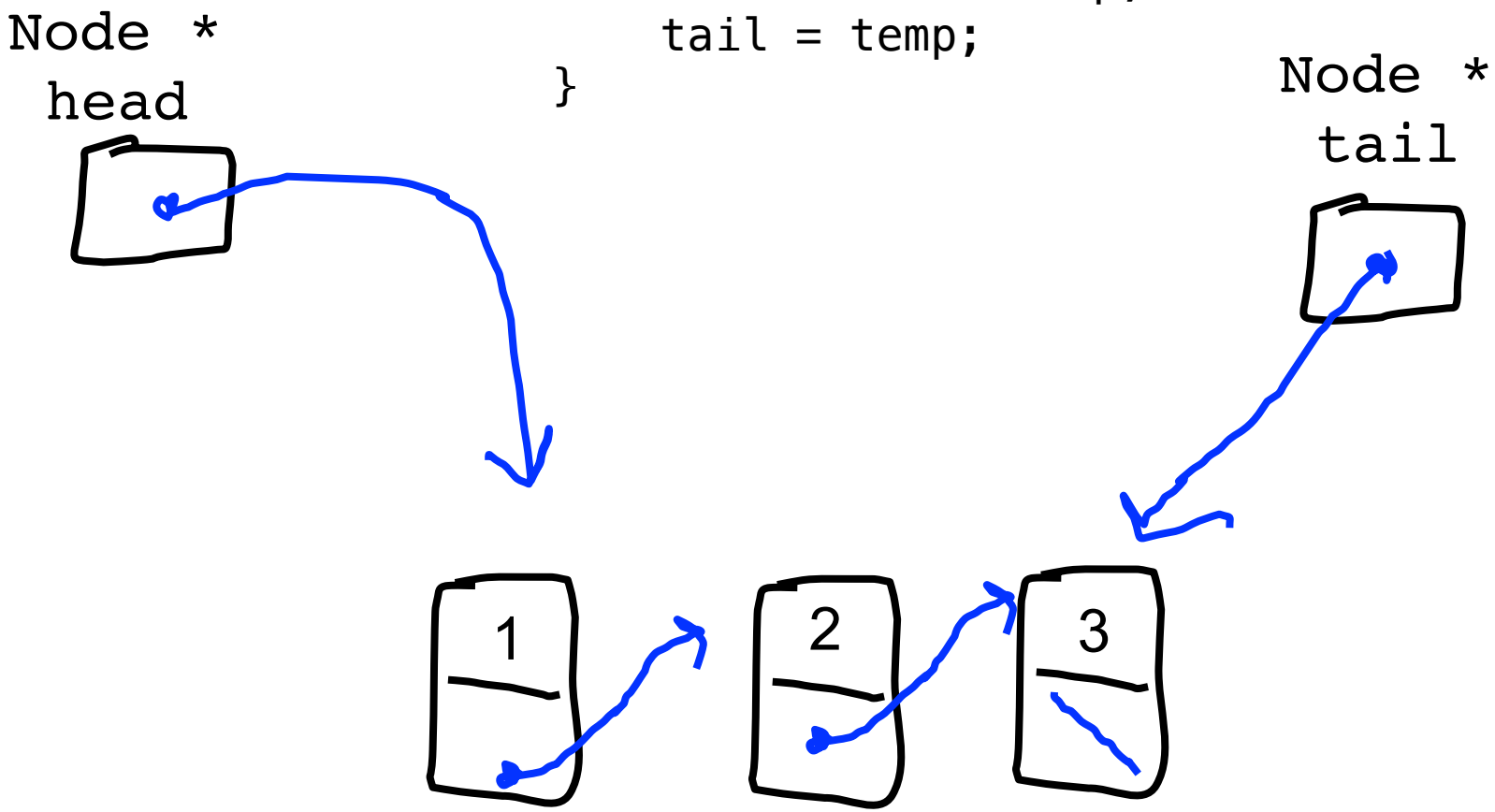
Enqueue: Goal



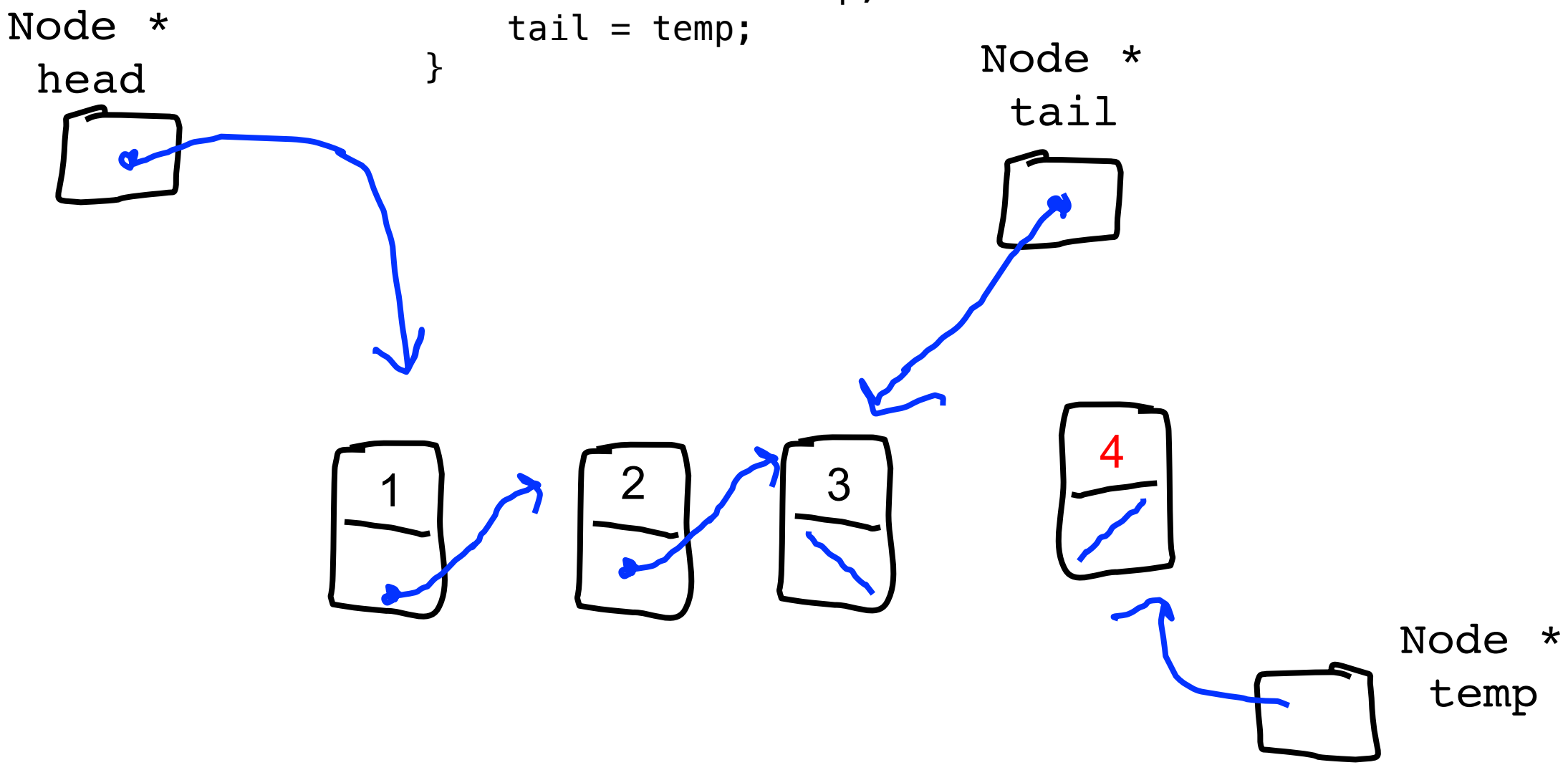
Enqueue



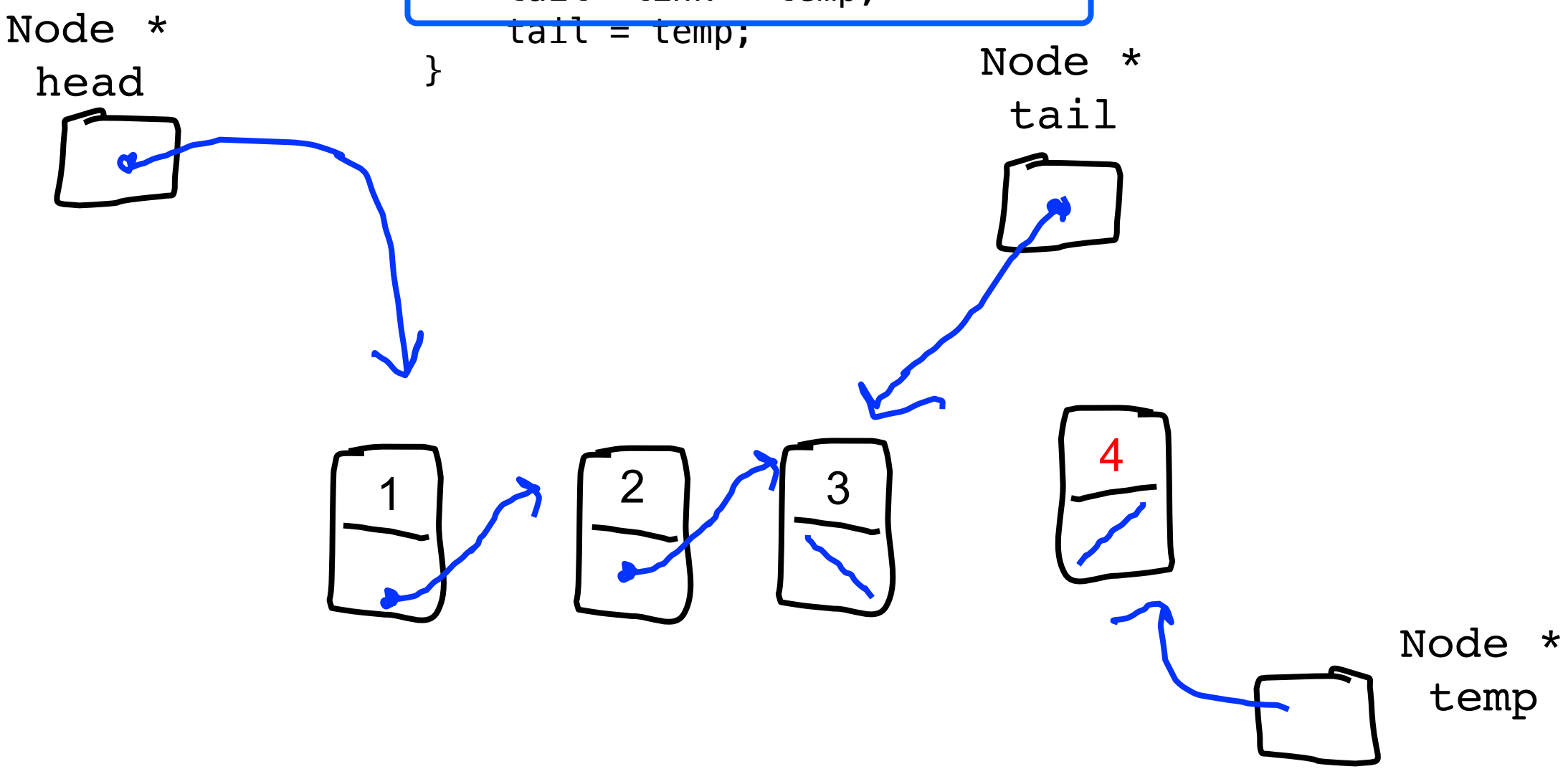
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



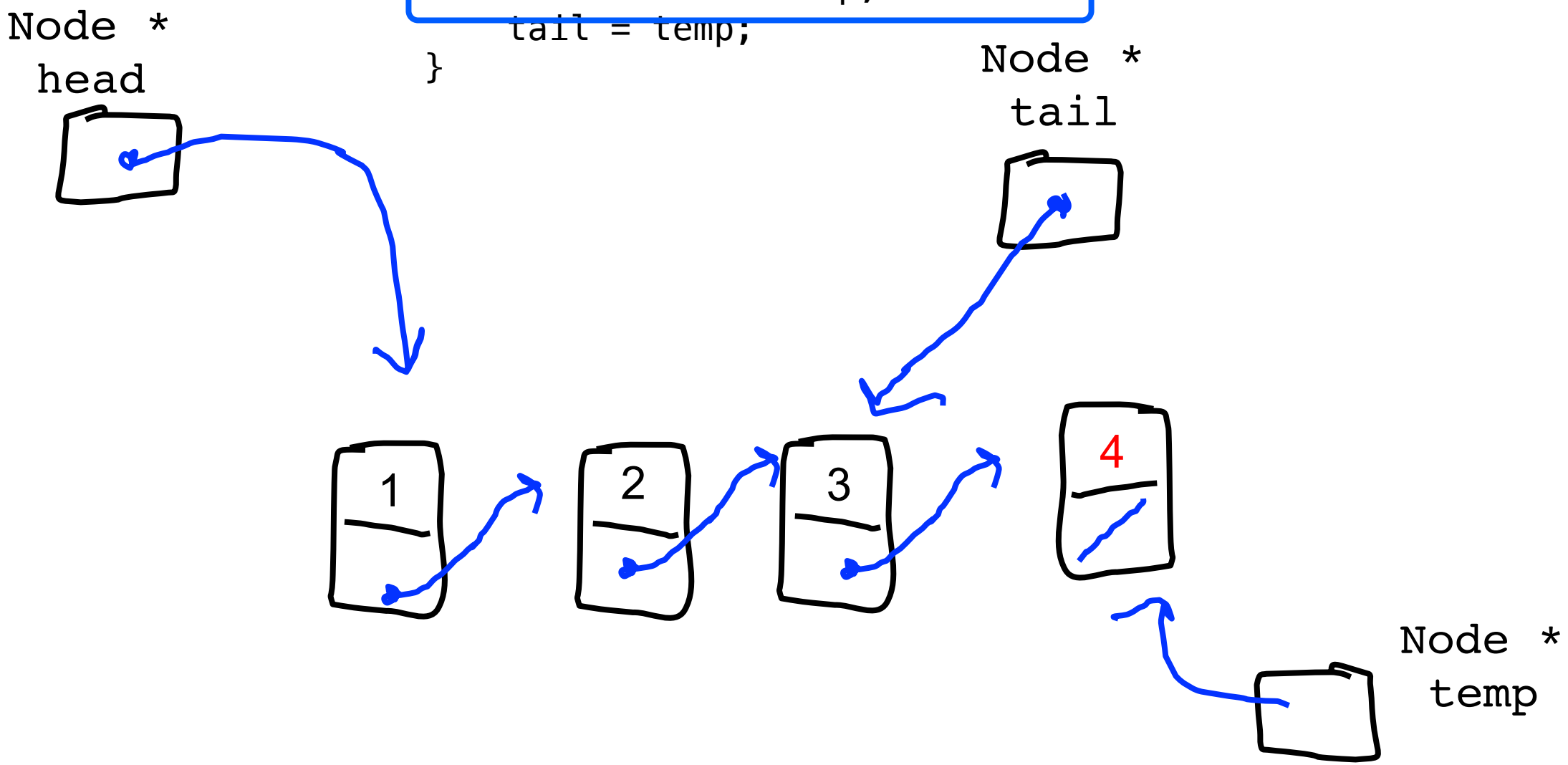
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



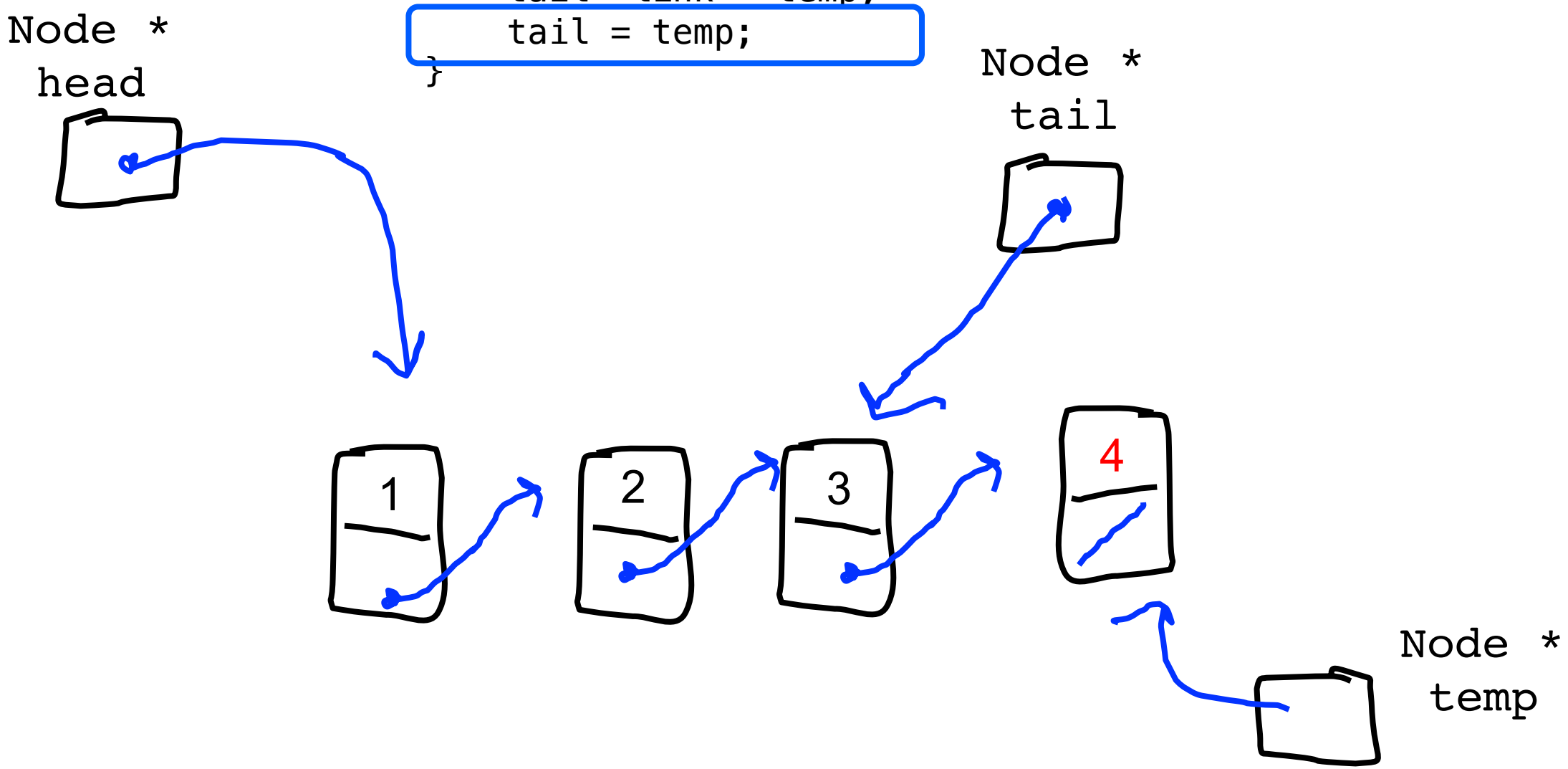
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



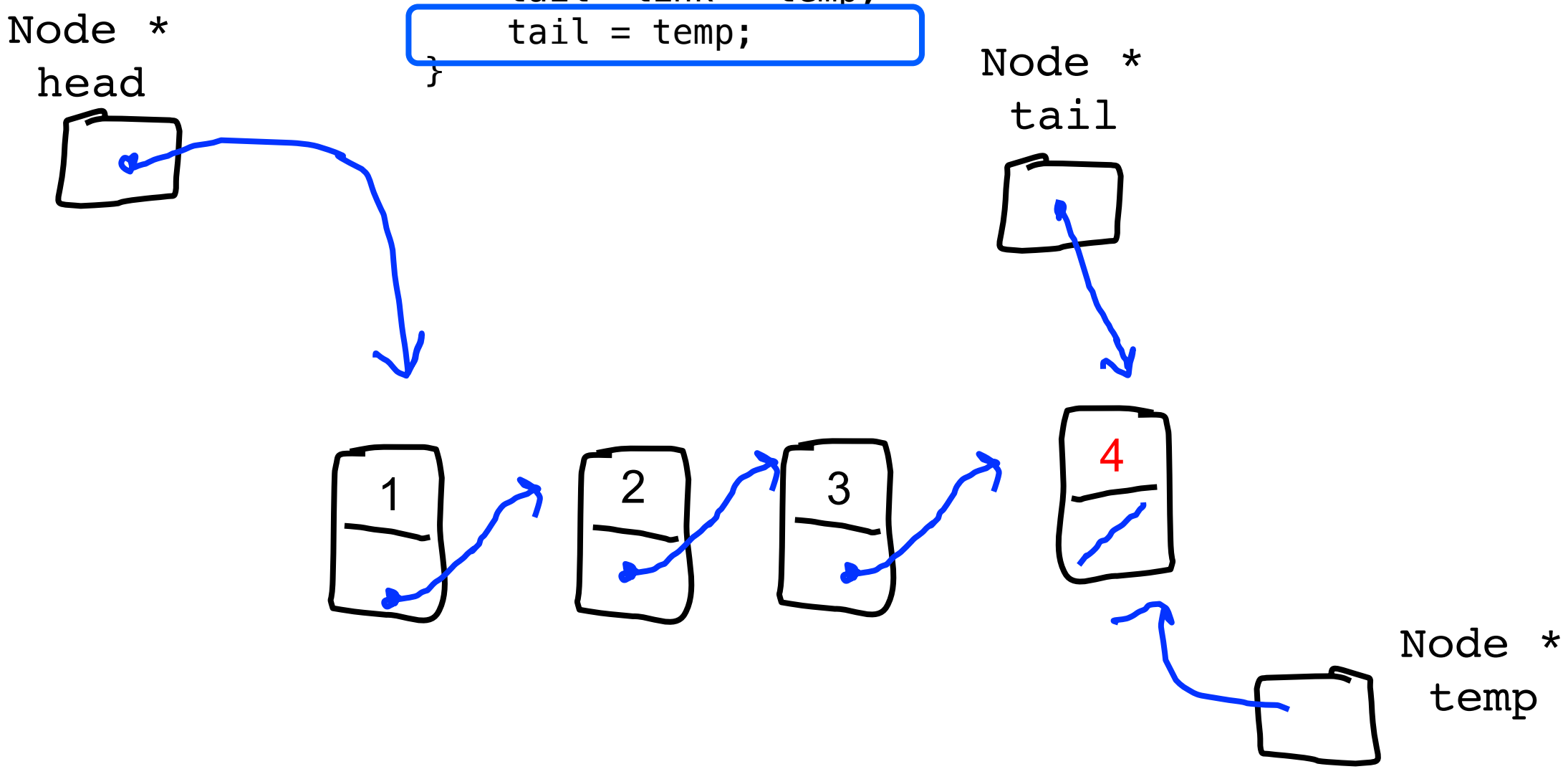
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



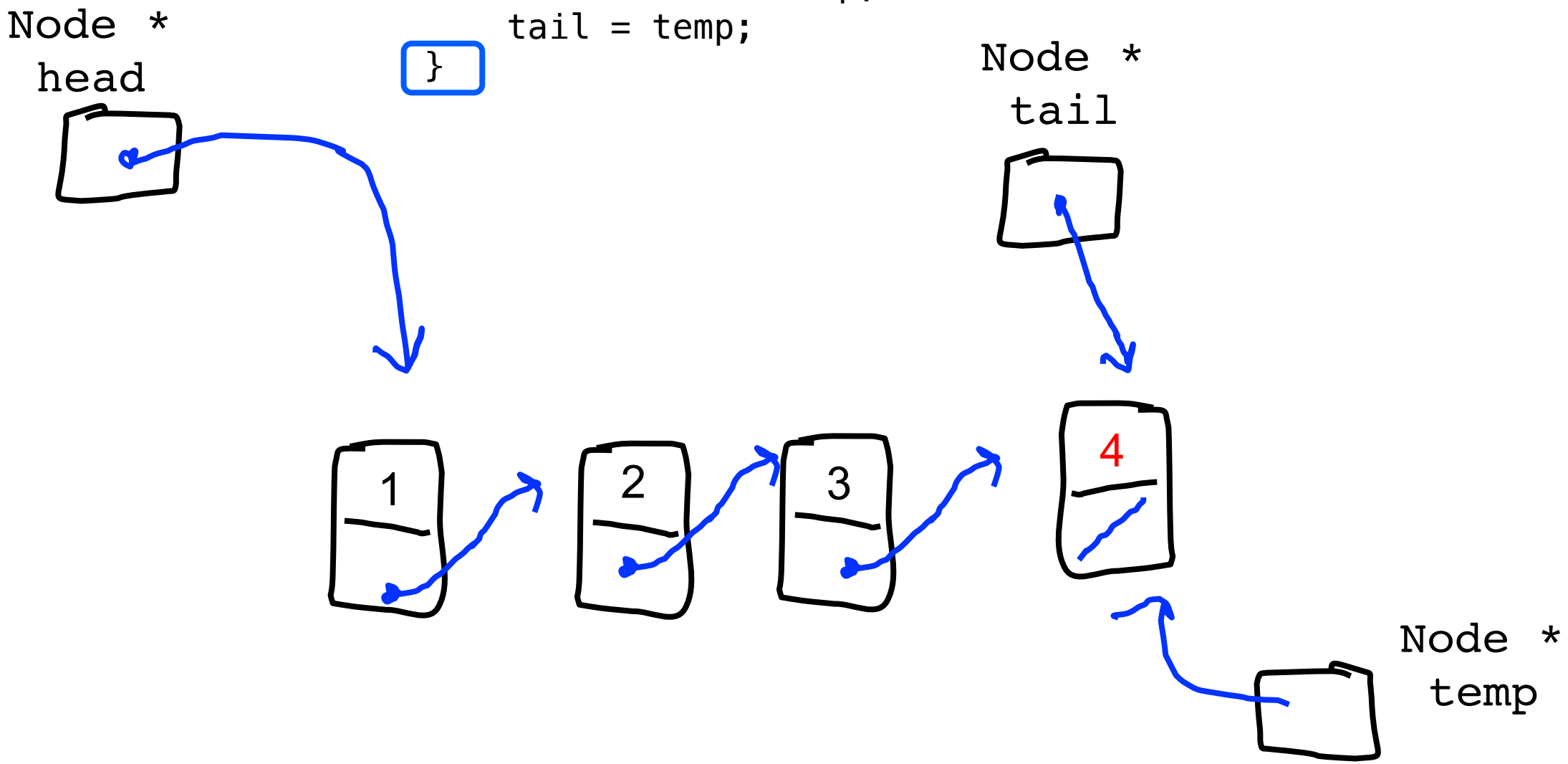
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



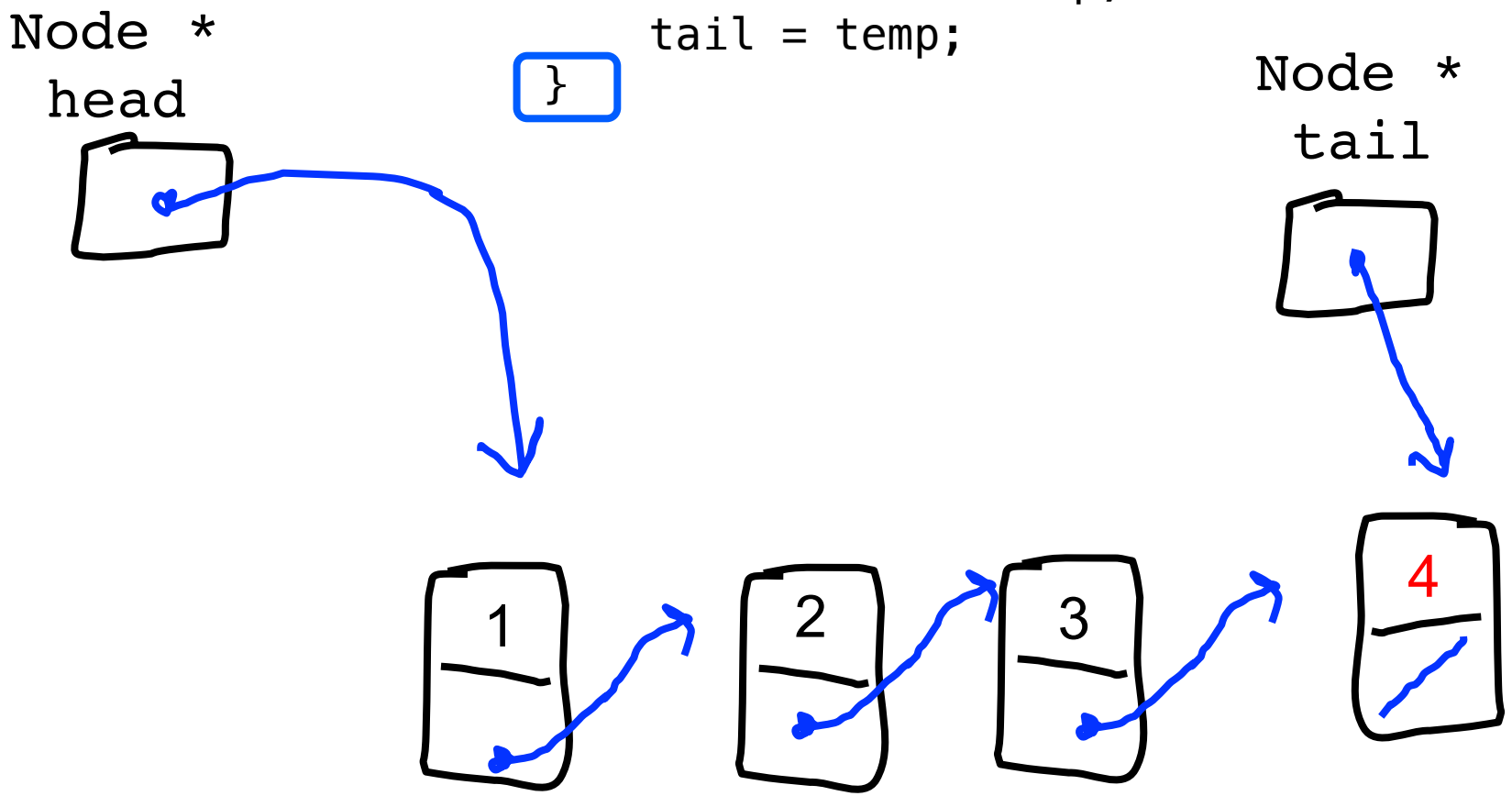
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



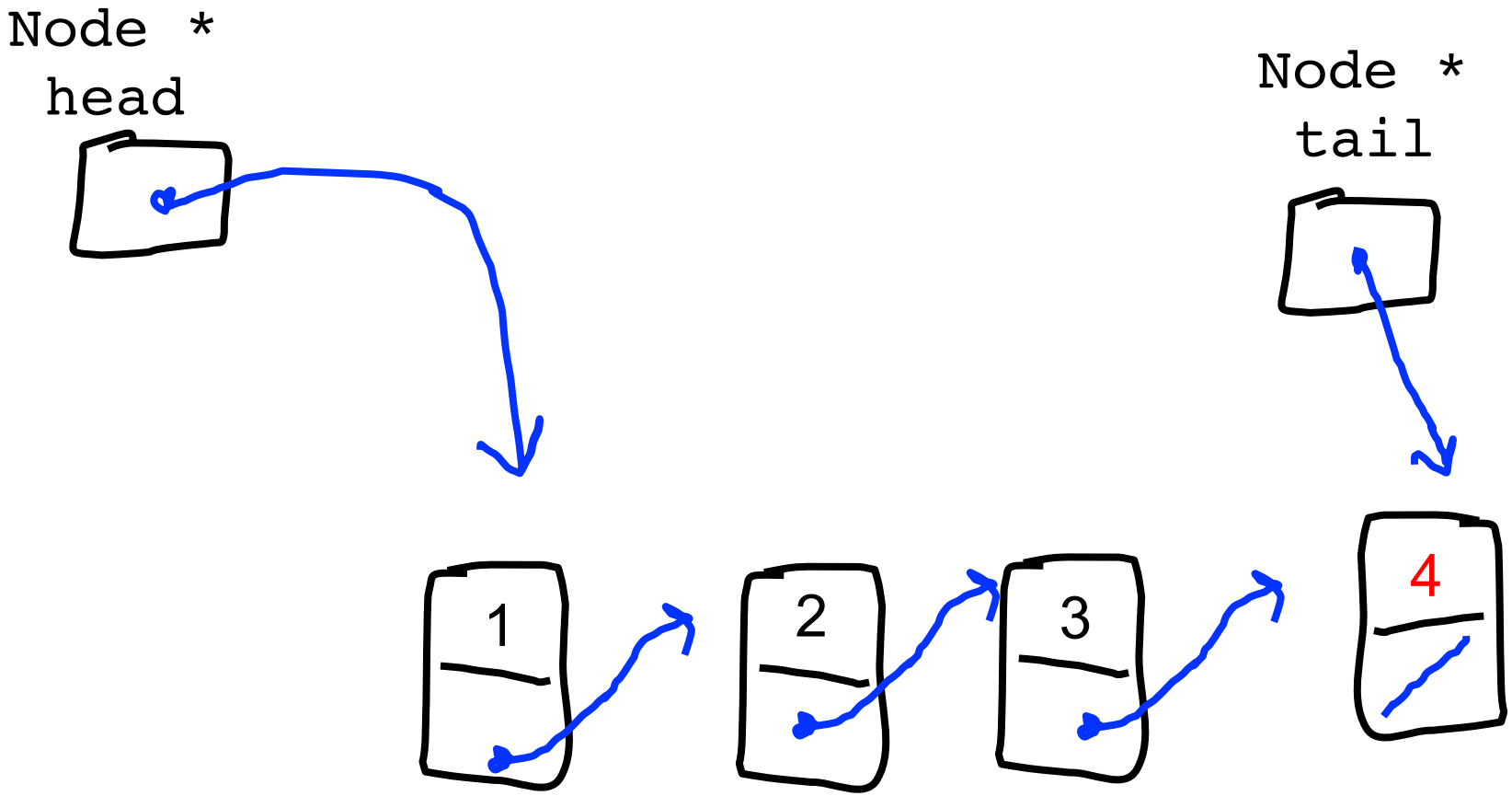
```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```



```
void QueueInt::enqueue(int v) {
    Node * temp = new Node;
    temp->value = v;
    tail->link = temp;
    tail = temp;
}
```

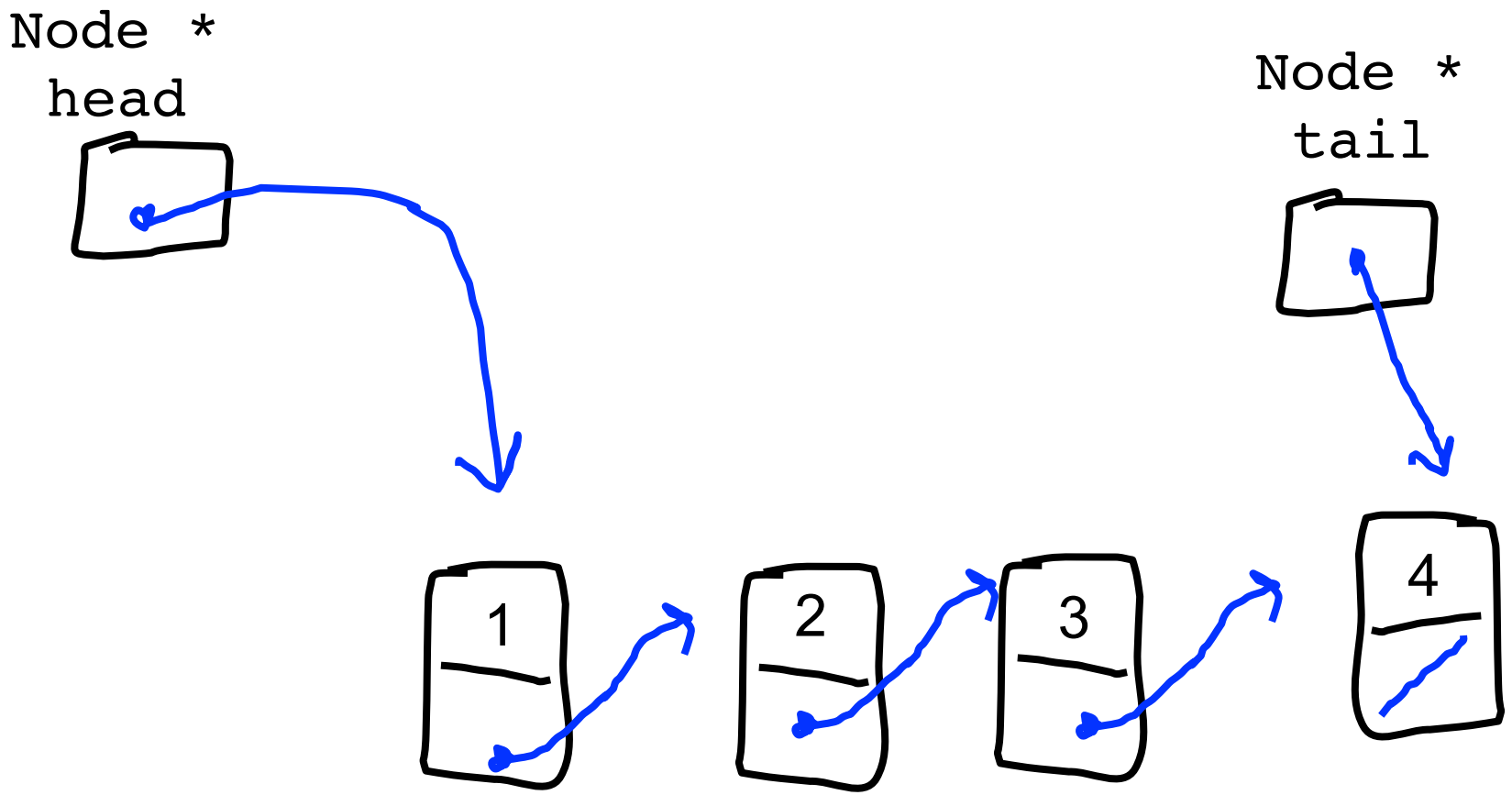


Actual Queue: Enqueue

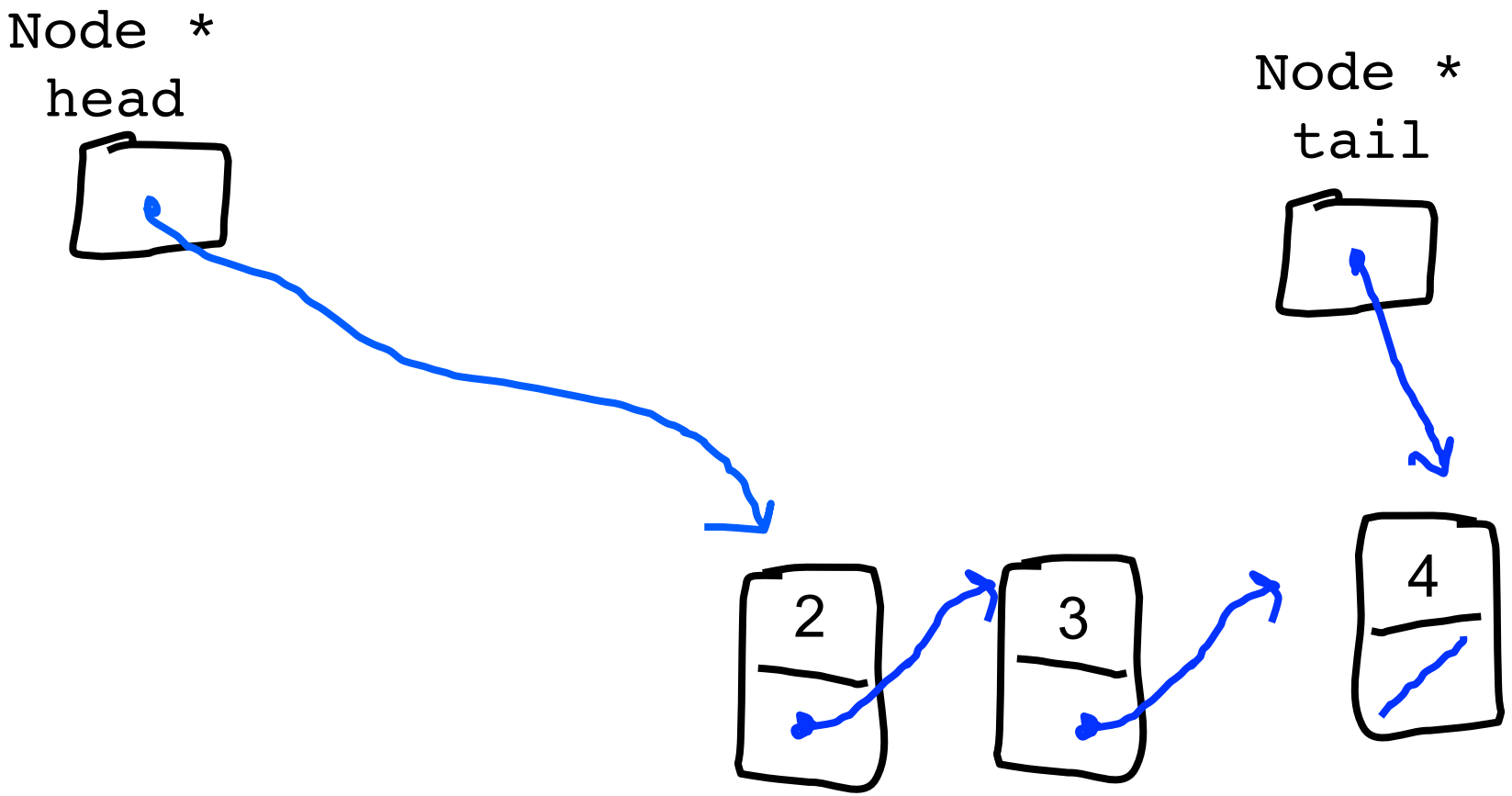


Dequeue

Actual Queue: Dequeue

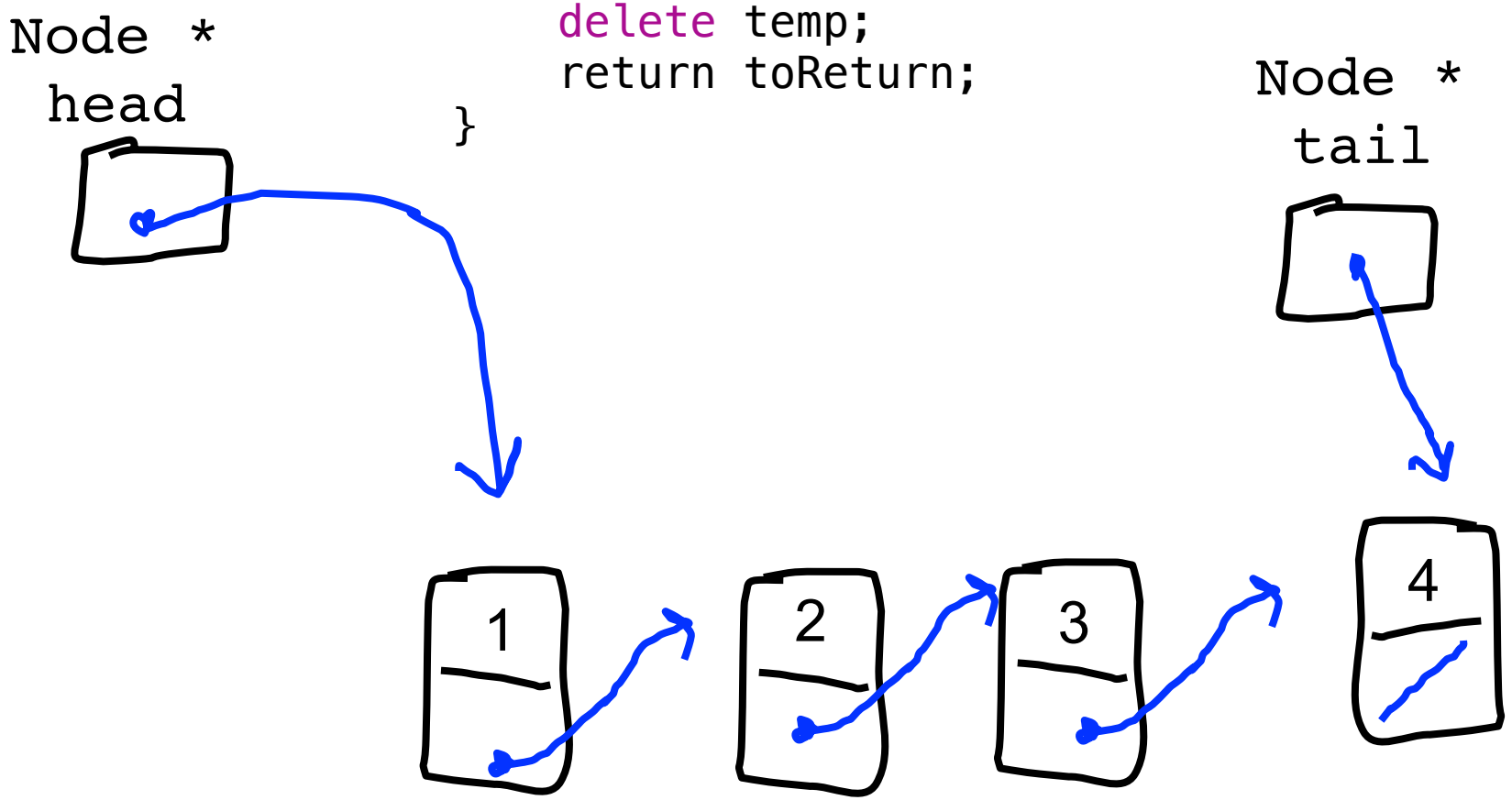


Dequeue: Goal

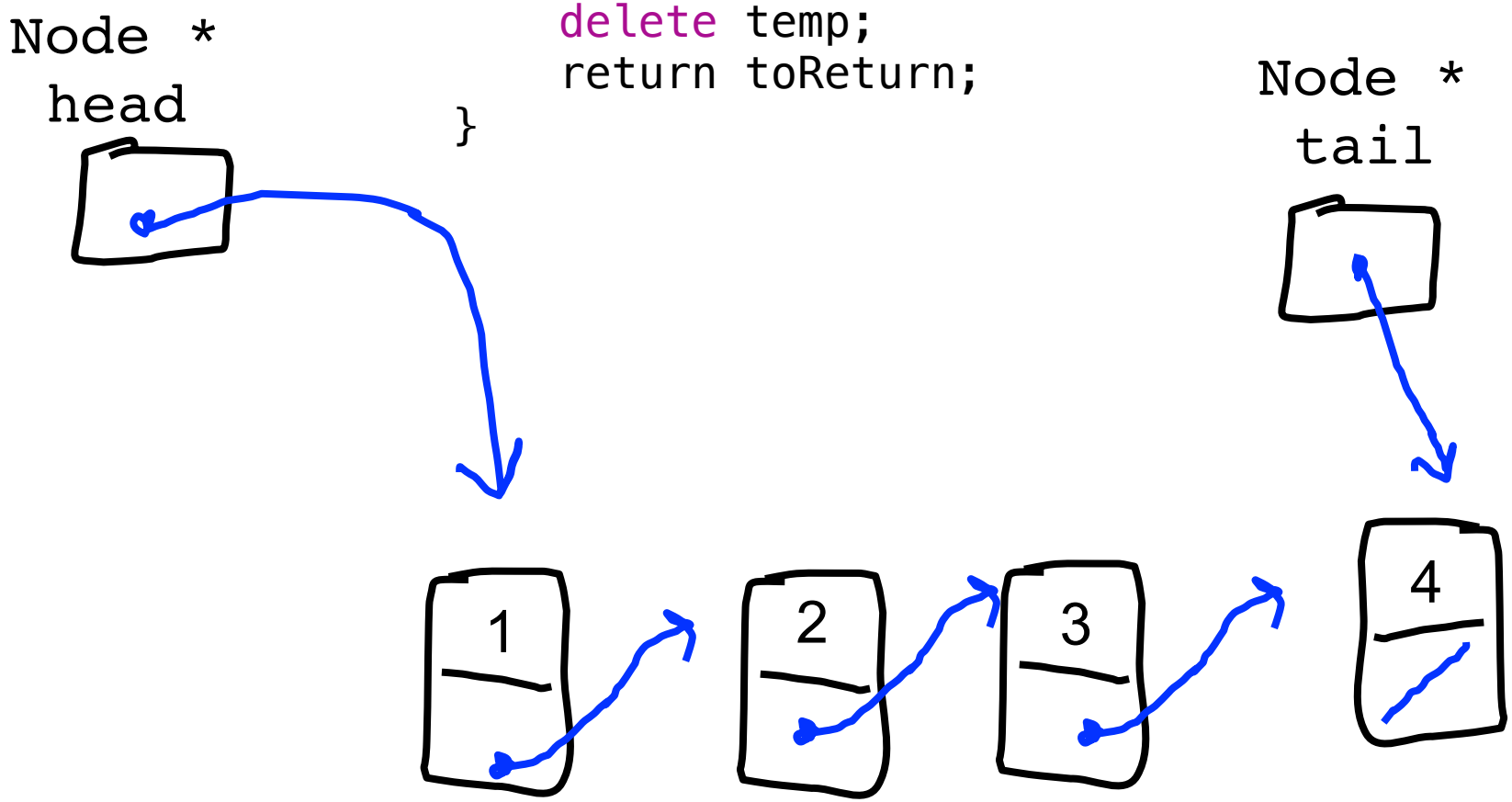


Return the value 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```

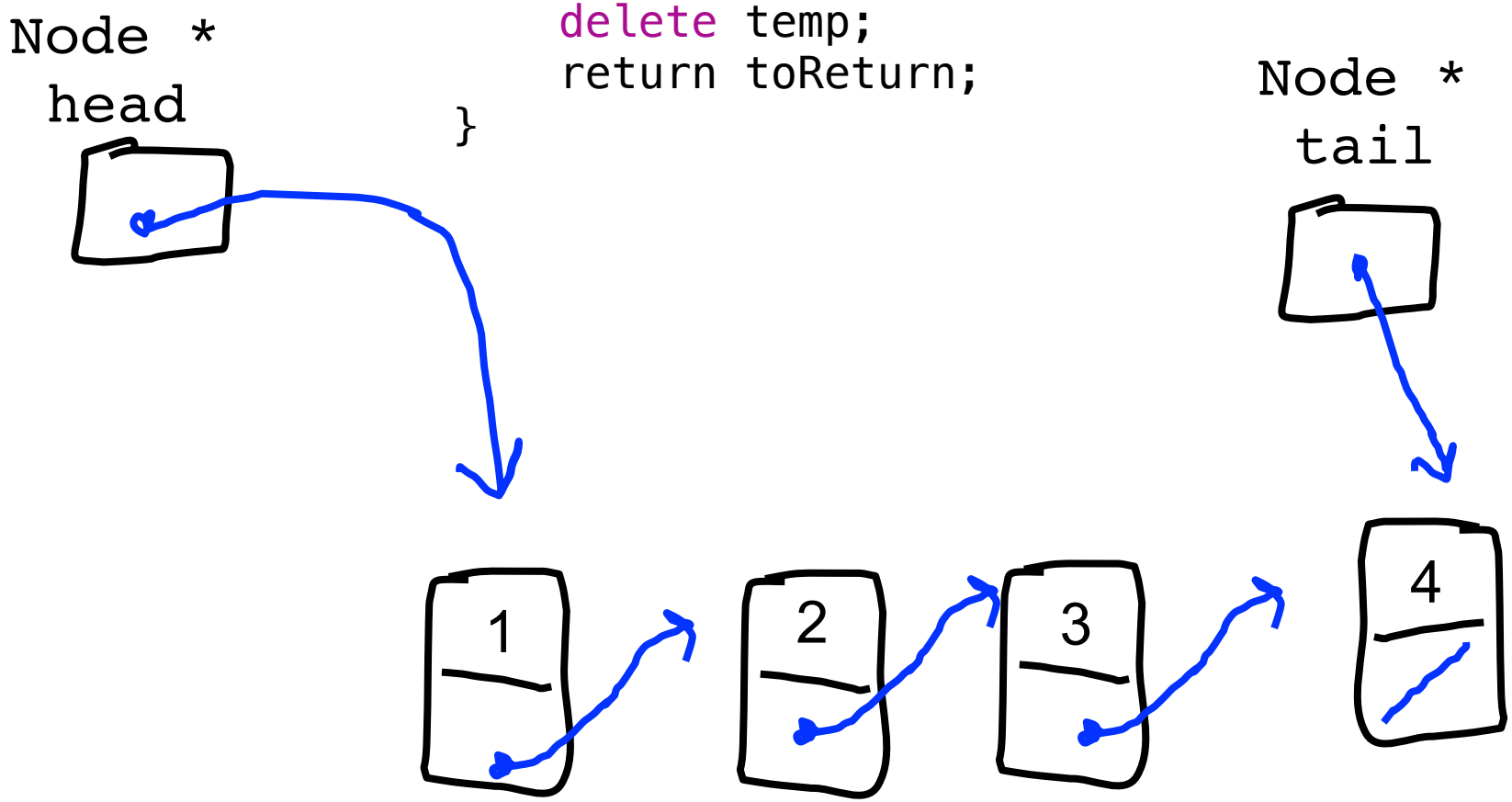


```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



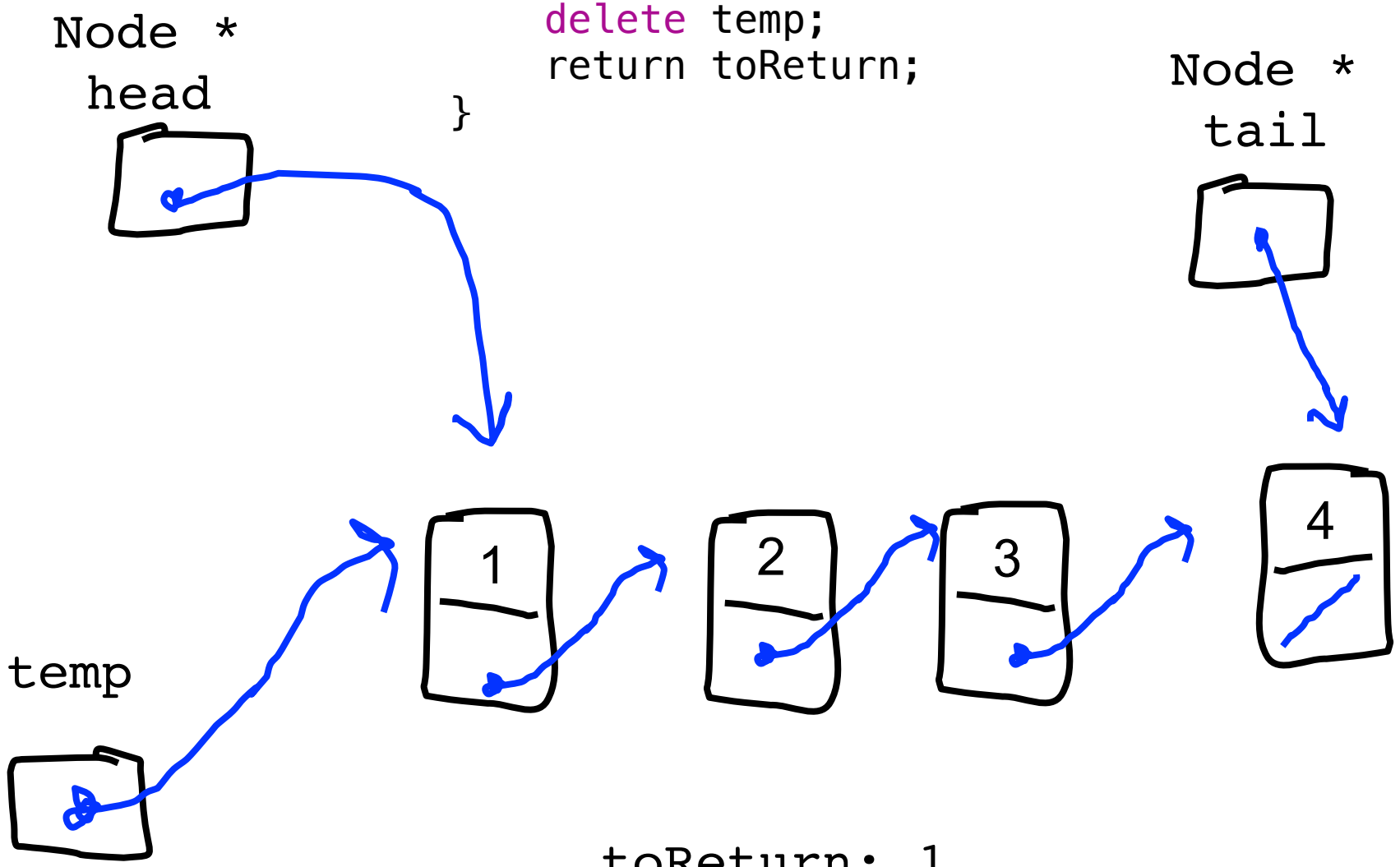
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



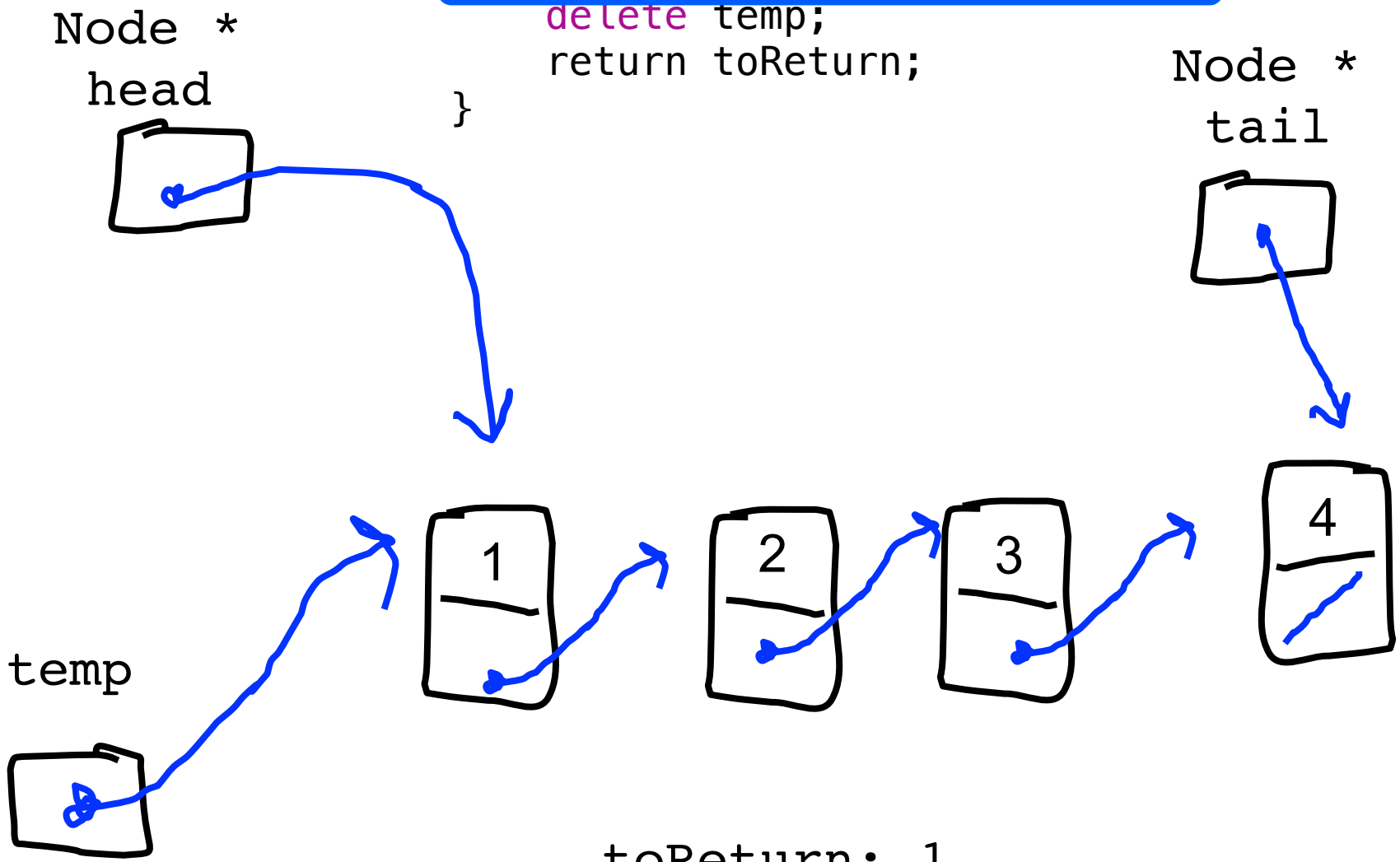
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



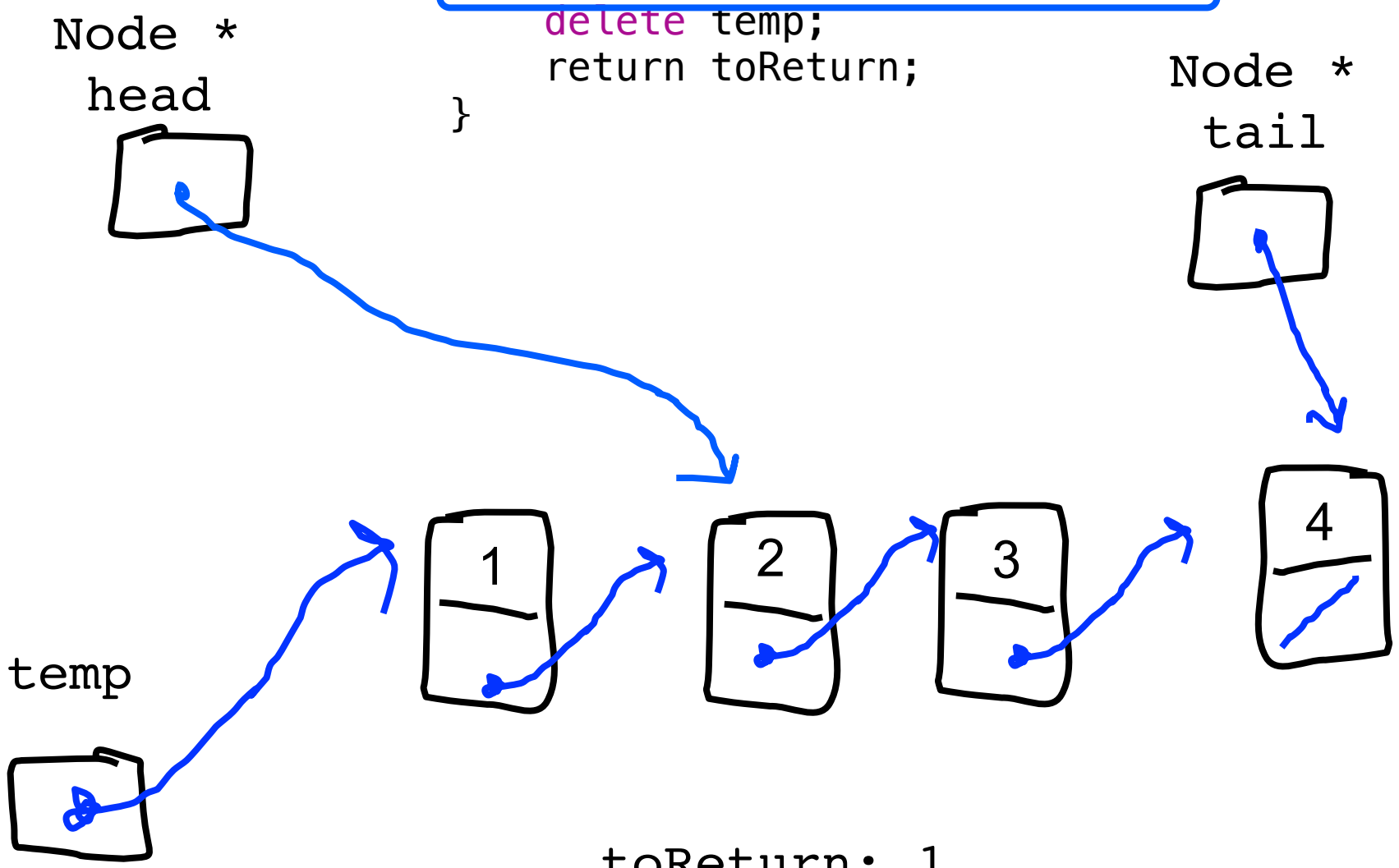
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



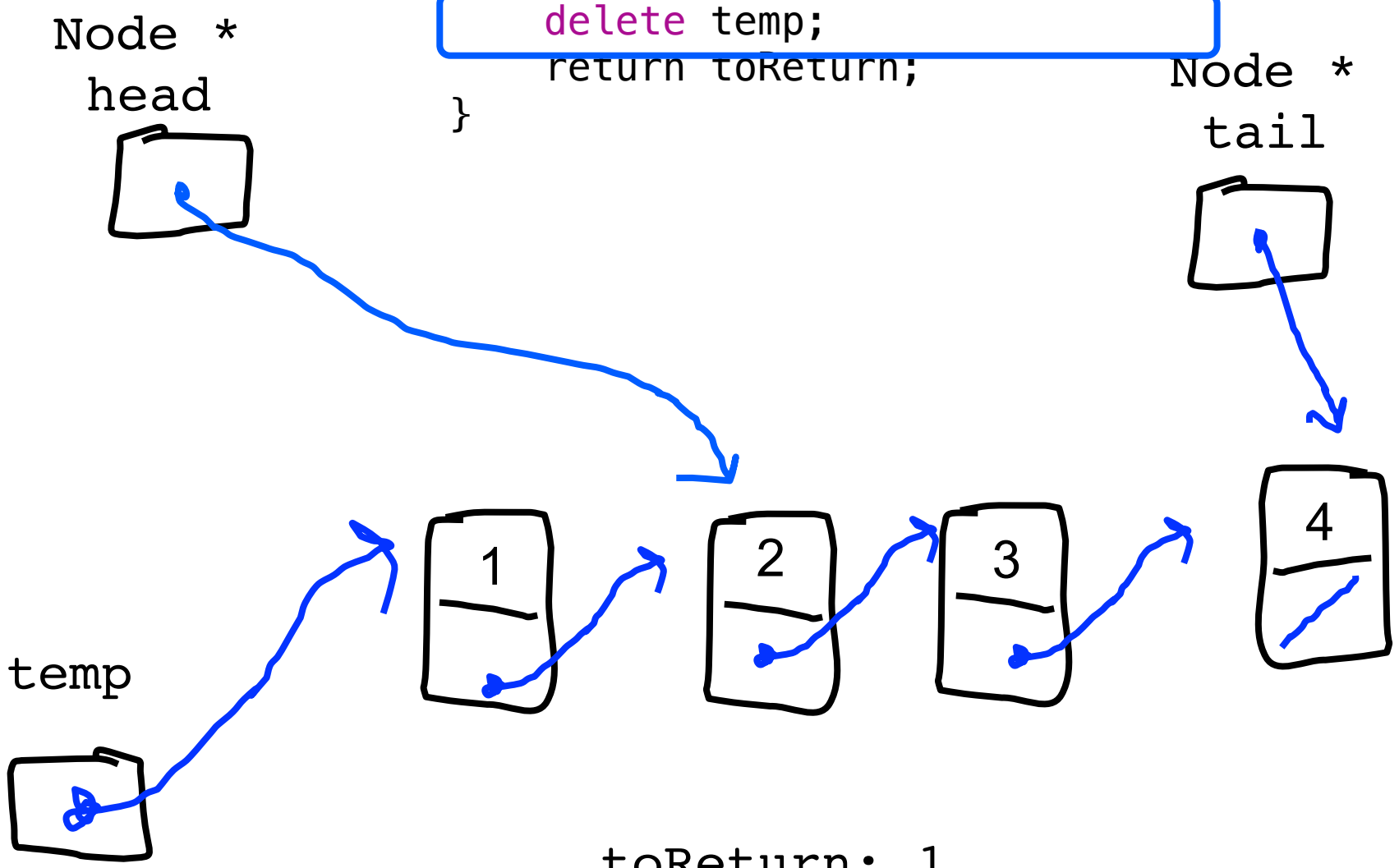
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



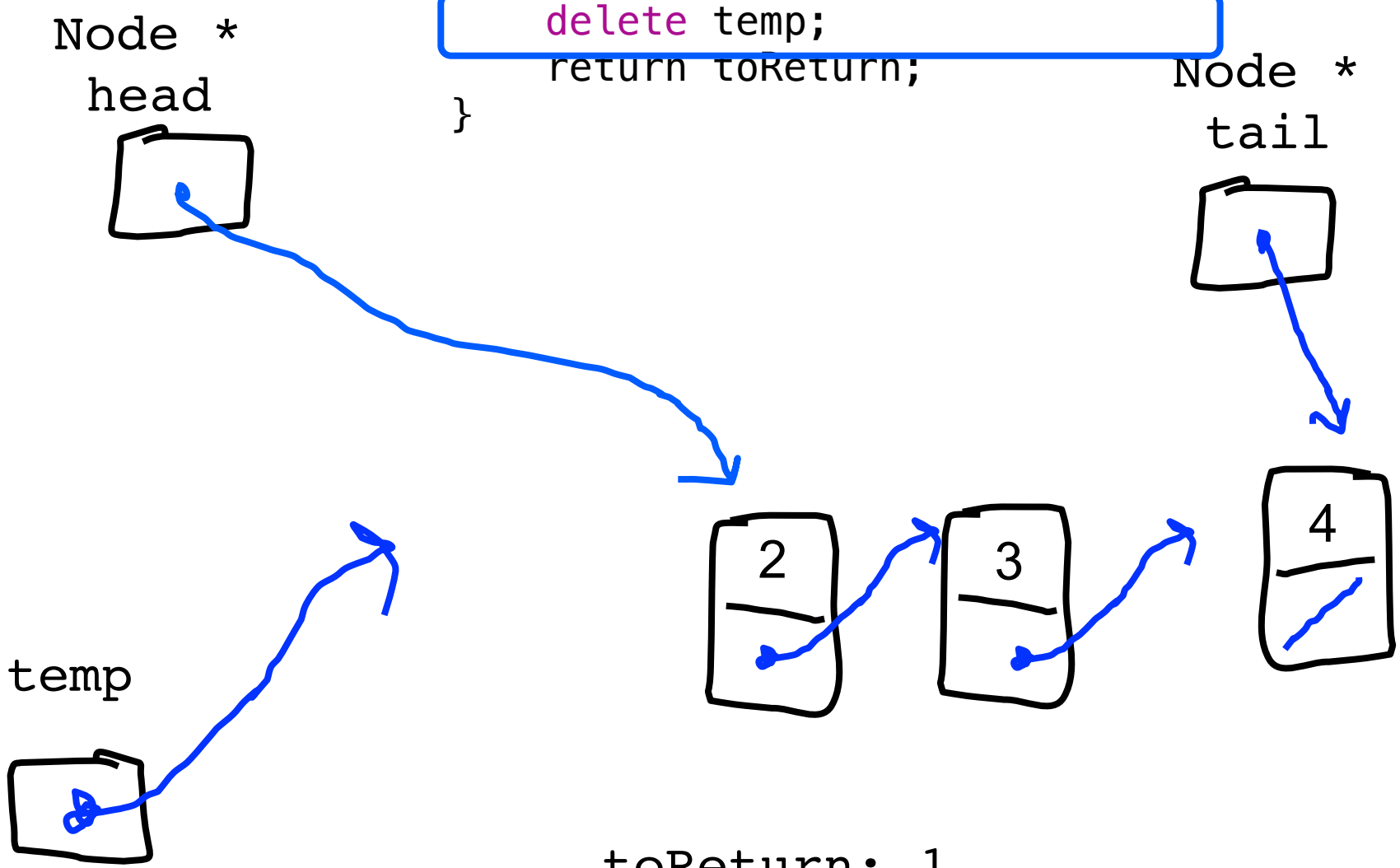
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



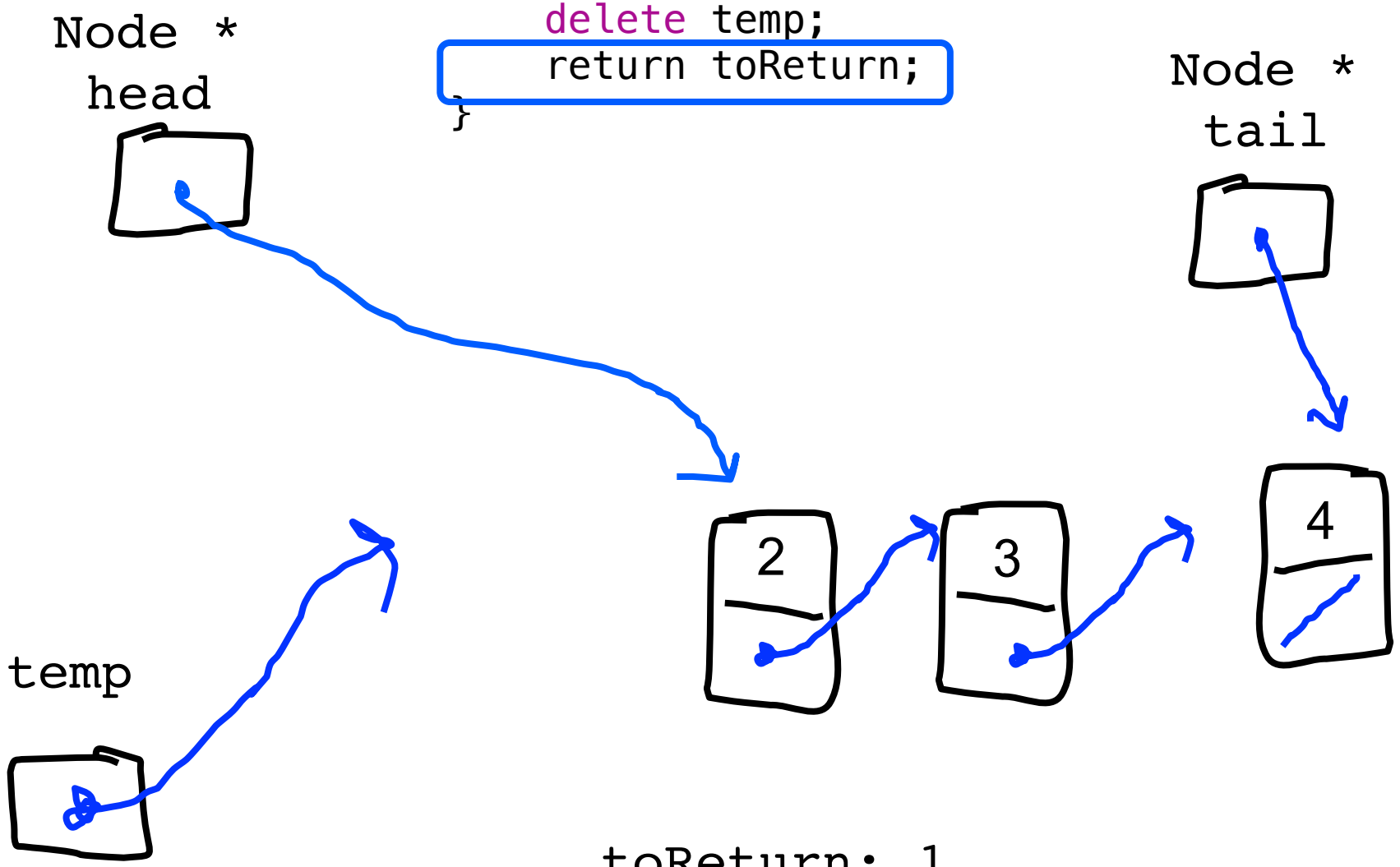
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



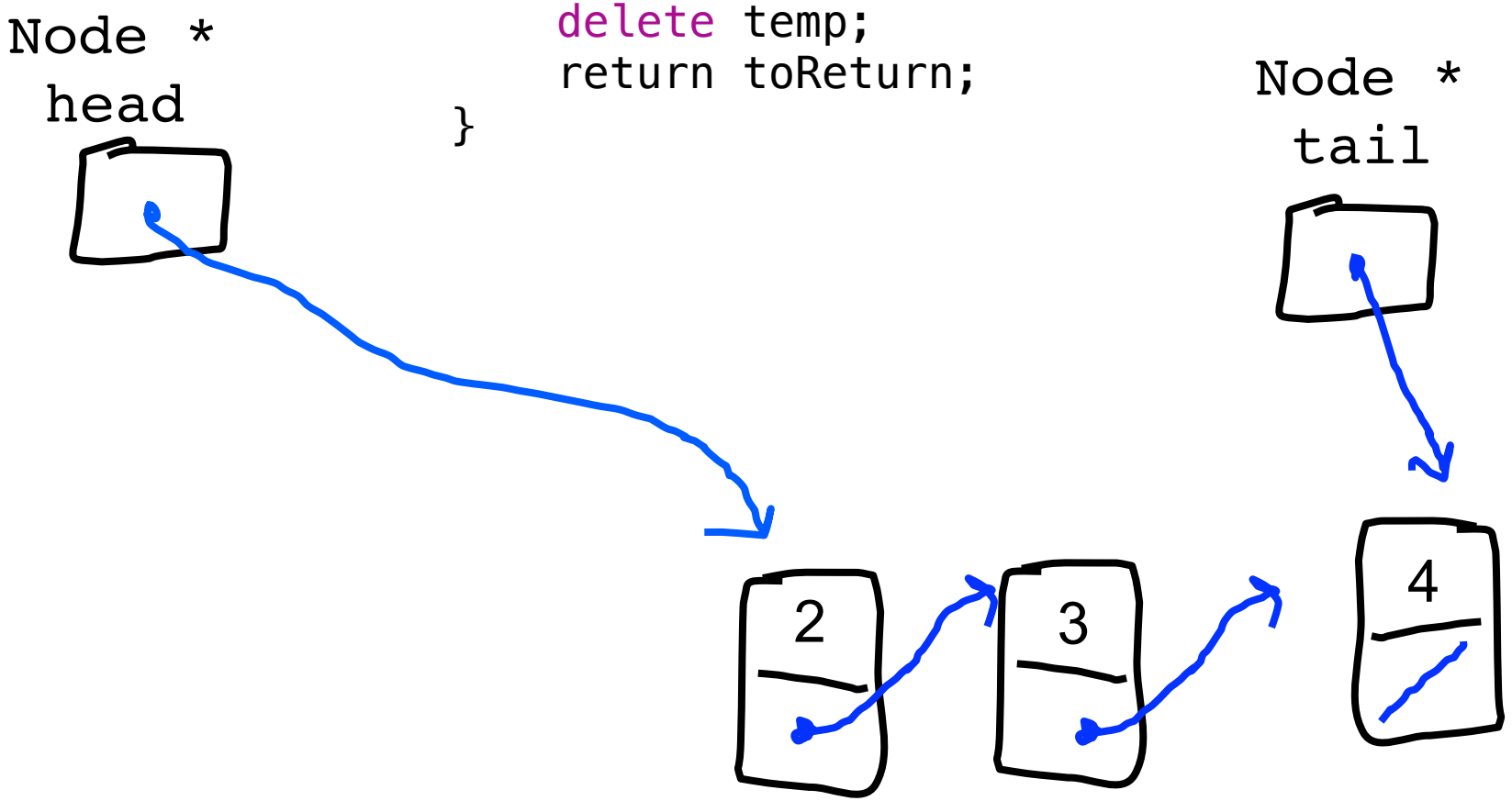
toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```

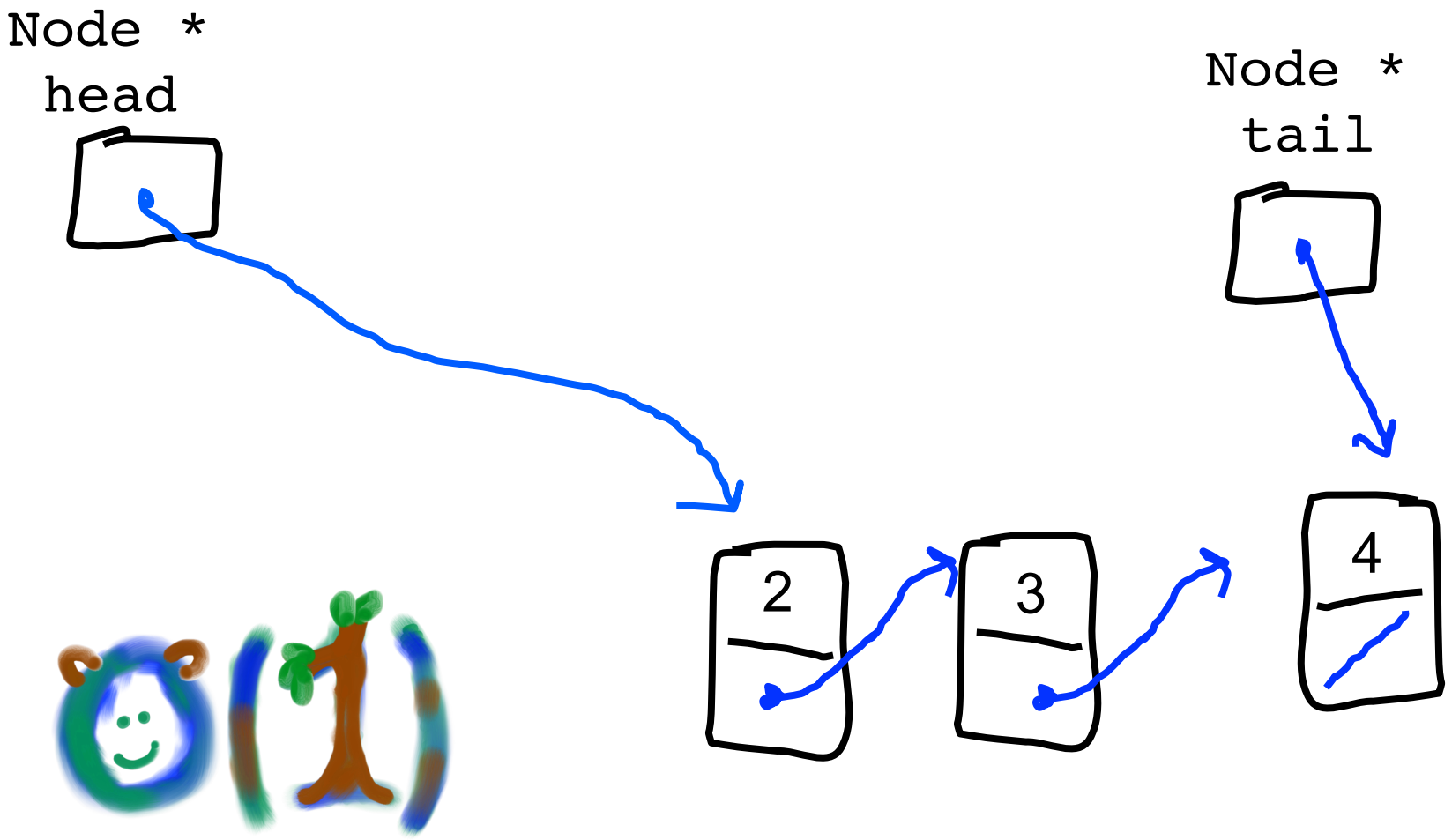


toReturn: 1

```
int QueueInt::dequeue() {
    int toReturn = head->value;
    Node * temp = head;
    head = temp->link;
    delete temp;
    return toReturn;
}
```



Actual Queue: Dequeue



Queue

```
class QueueInt {                // in QueueInt.h
public:
    QueueInt ();                // constructor

    void enqueue(int value); // append a value
    int dequeue();                 // return the first-in value

private:
    struct Node {
      int value;
      Node * link;
    };
    Node * head;      // has a pointer to the first node
    Node * tail;      // and a pointer to the last node
};
```

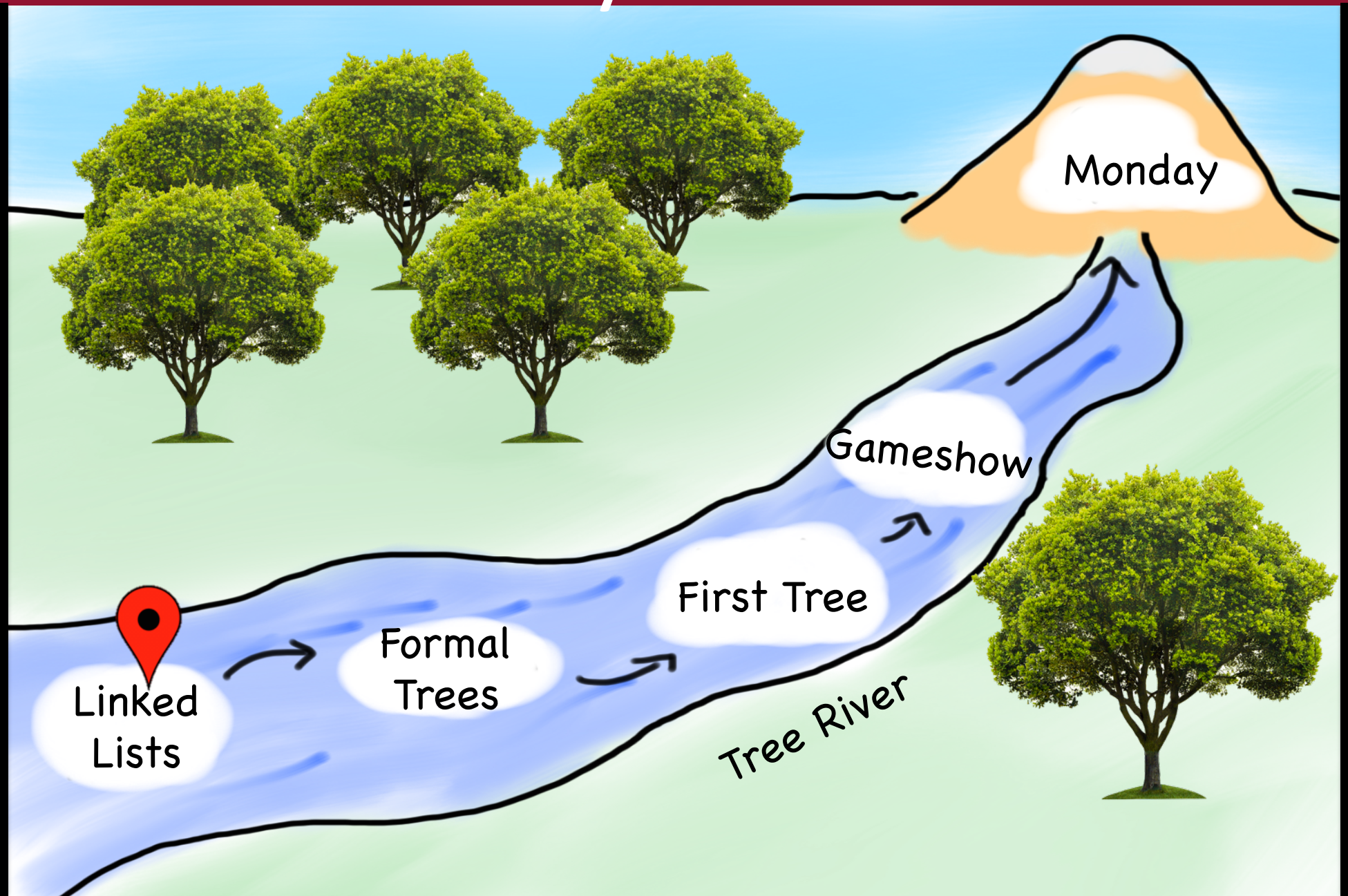
Linked Lists are Excellent

	Worst
Stack Push	$\mathcal{O}(1)$
Stack Pop	$\mathcal{O}(1)$
Queue Enqueue	$\mathcal{O}(1)$
Queue Dequeue	$\mathcal{O}(1)$

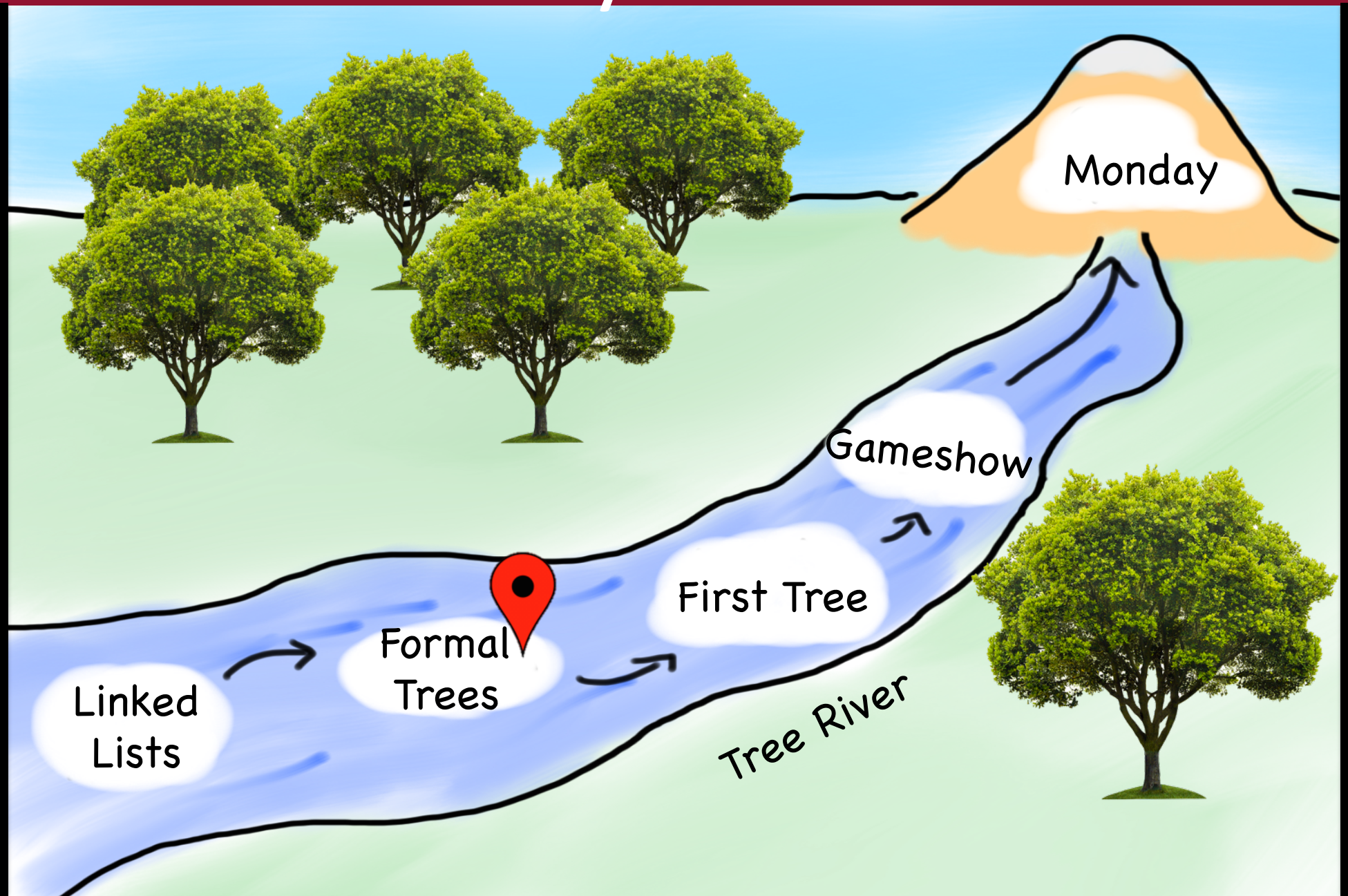
Welcome to the world of linked structures…

It gets more exciting…

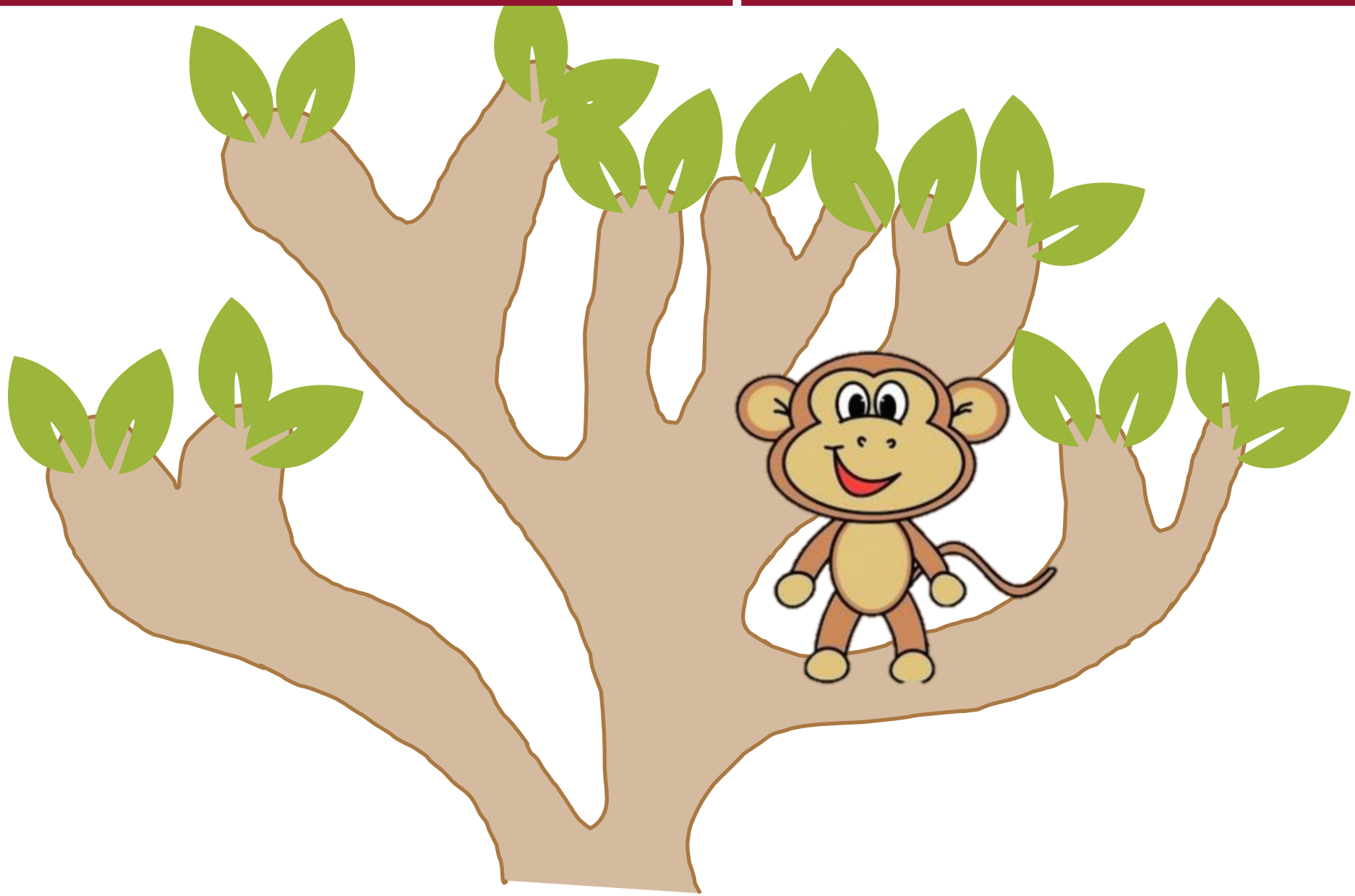
Today's Route



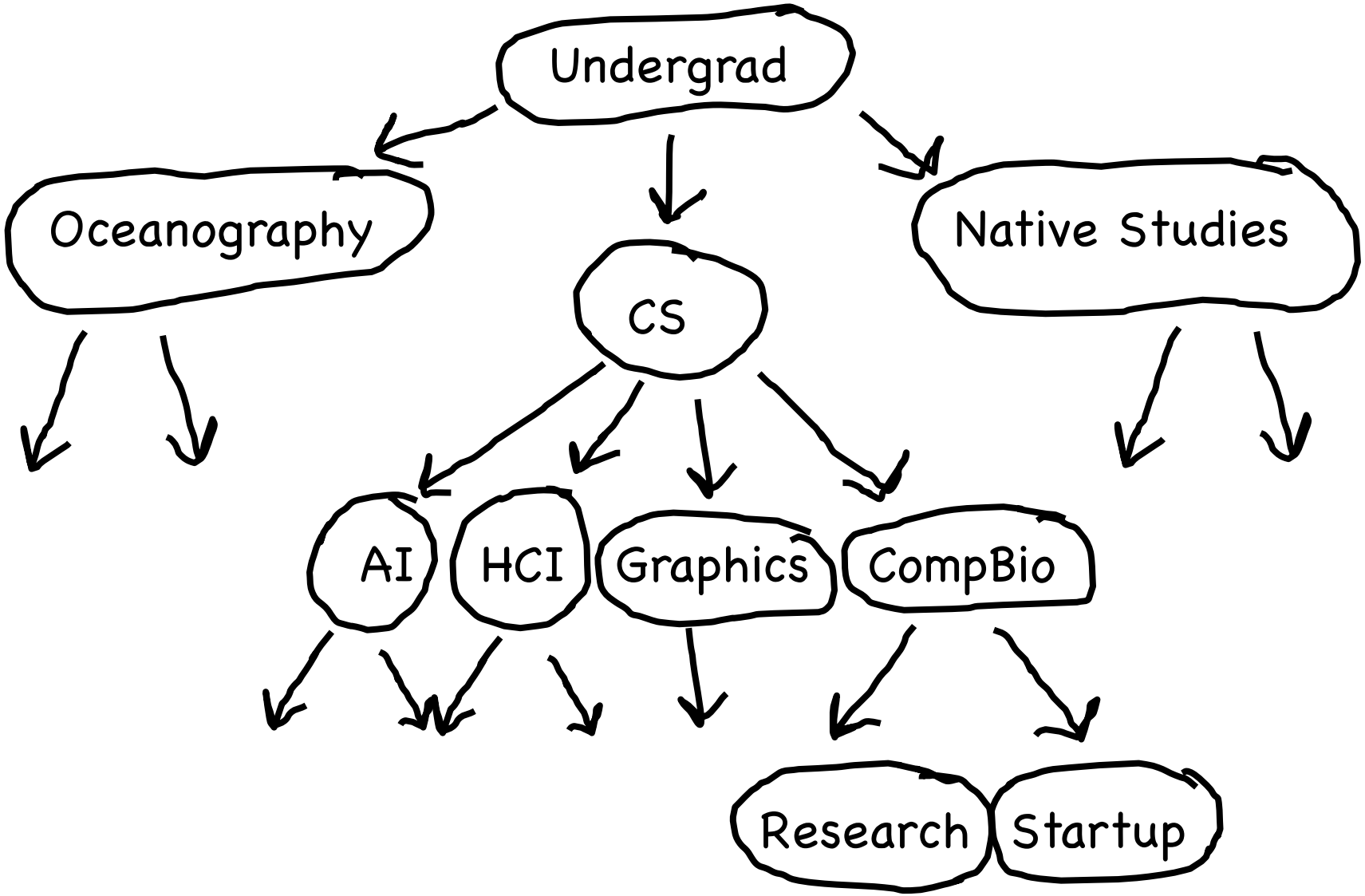
Today's Route



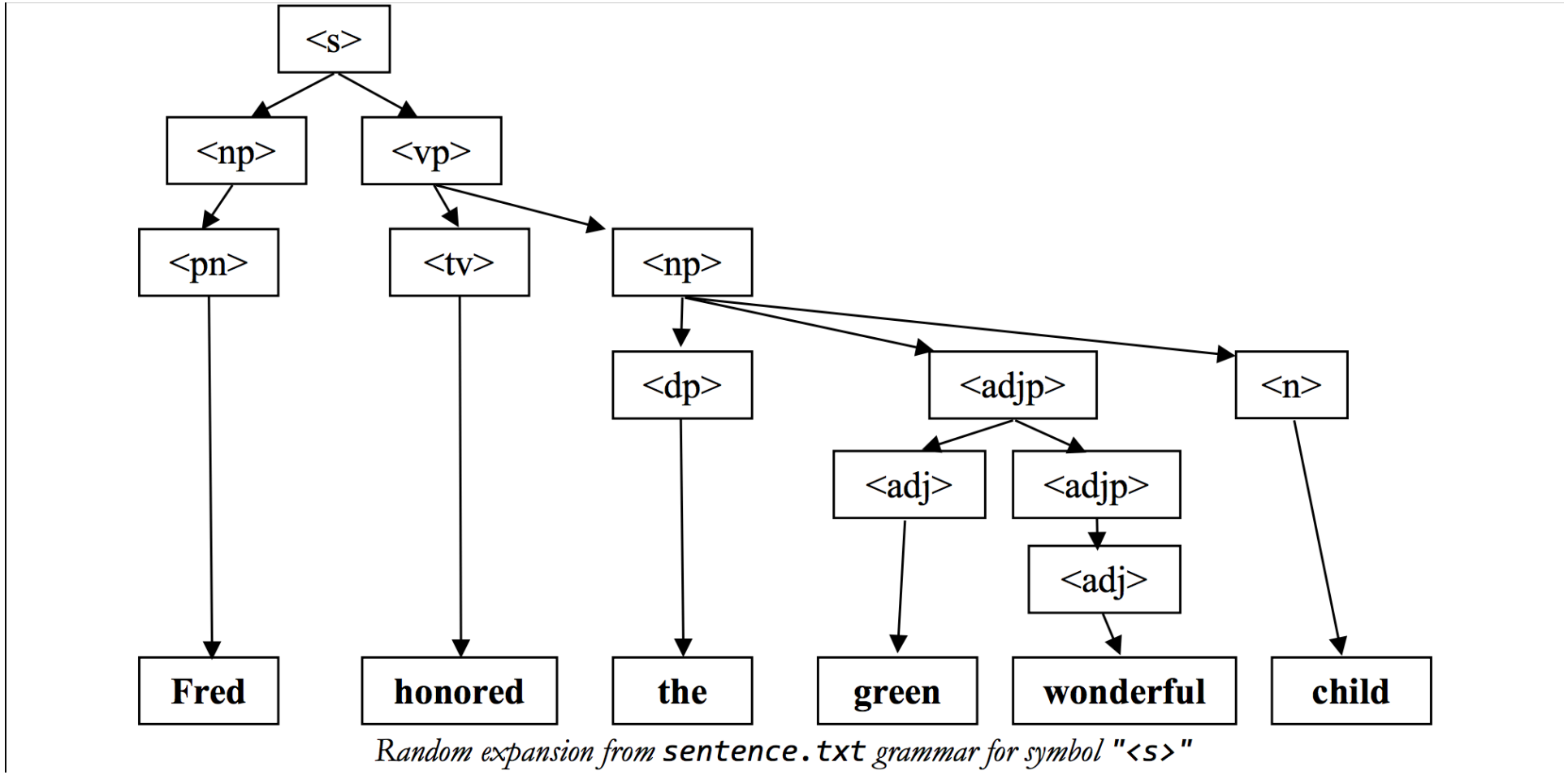
Recursive Exploration



Decision Tree

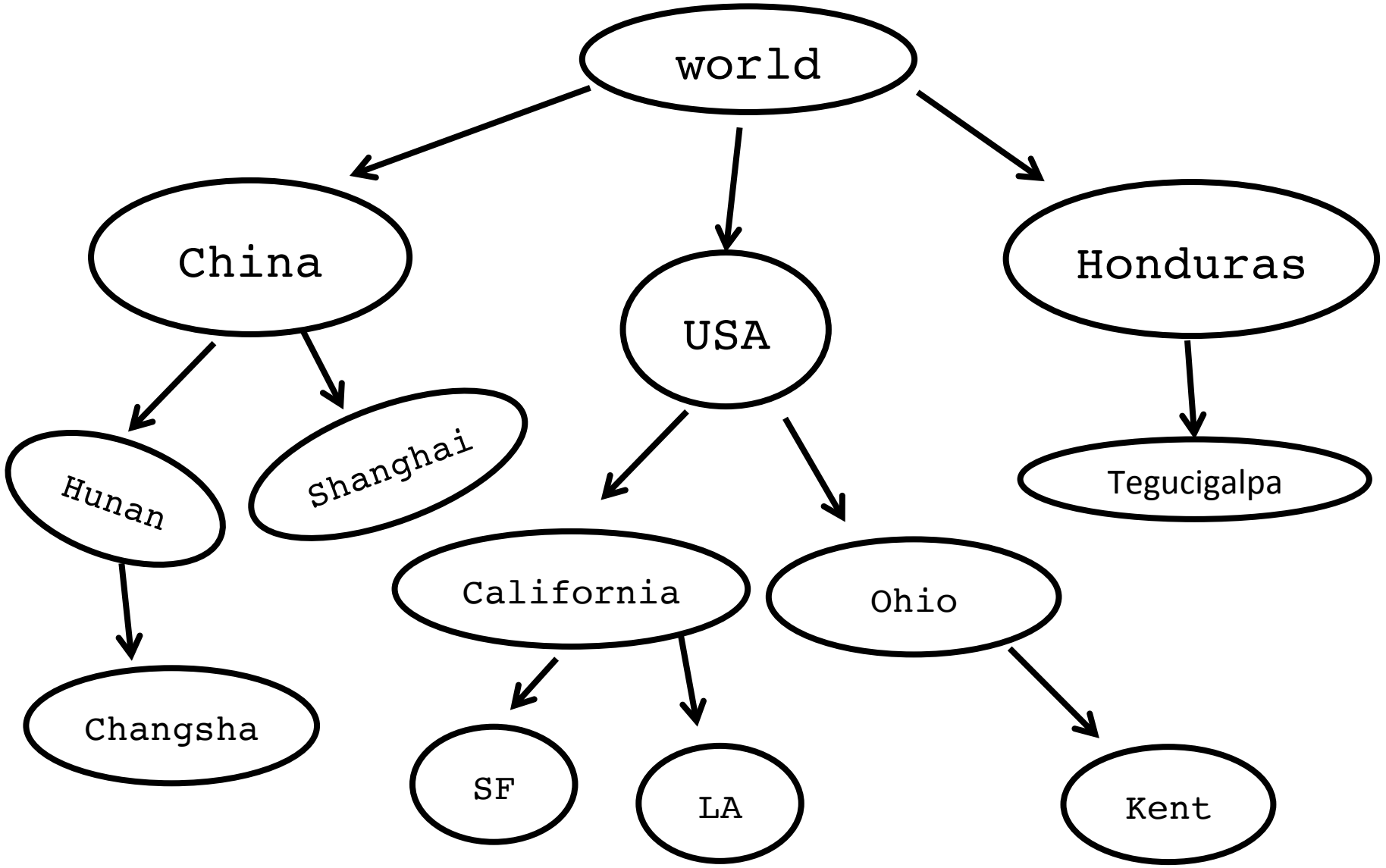


Syntax Tree

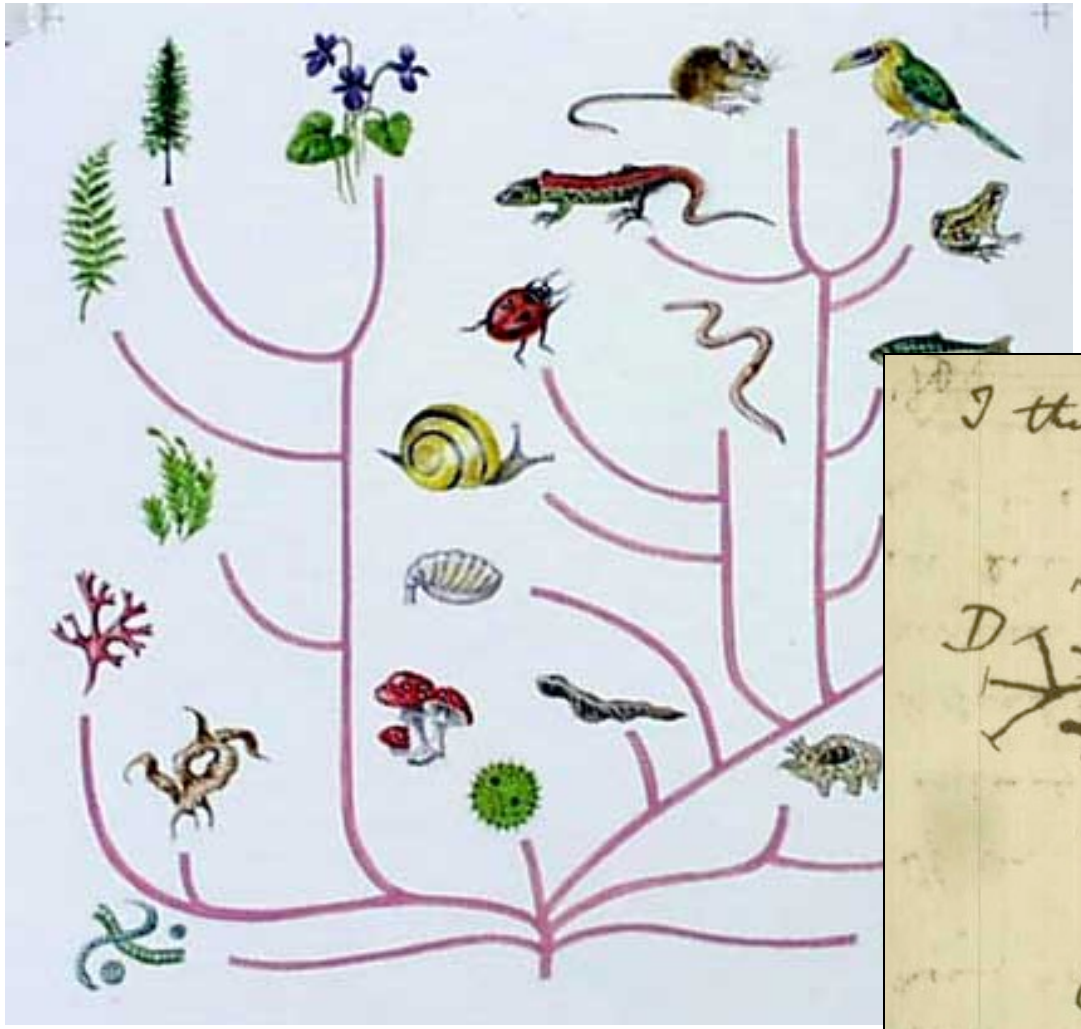


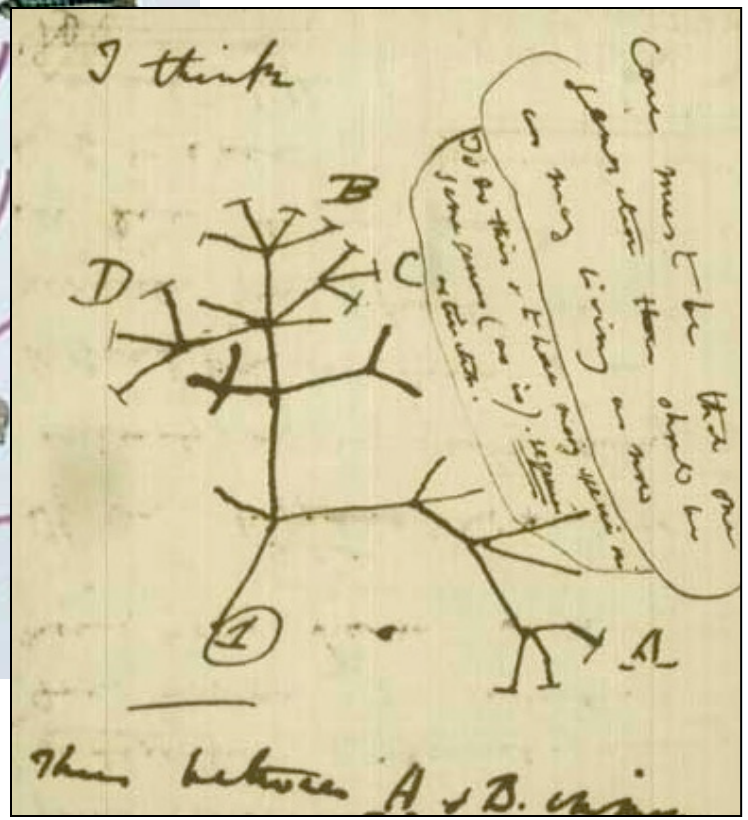
Random expansion from `sentence.txt` *grammar for symbol* `"<s>"`

Hierarchies

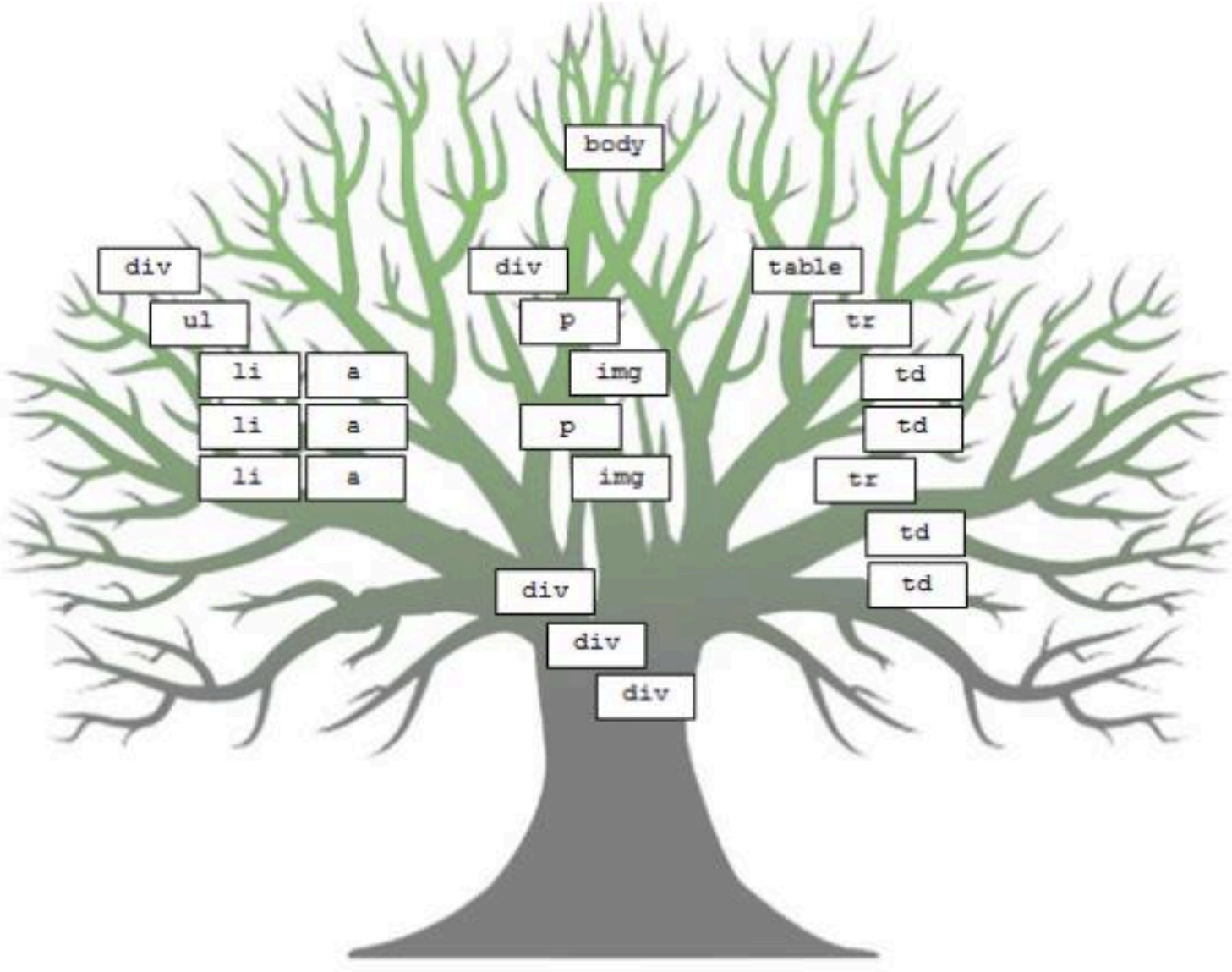


More Hierarchies



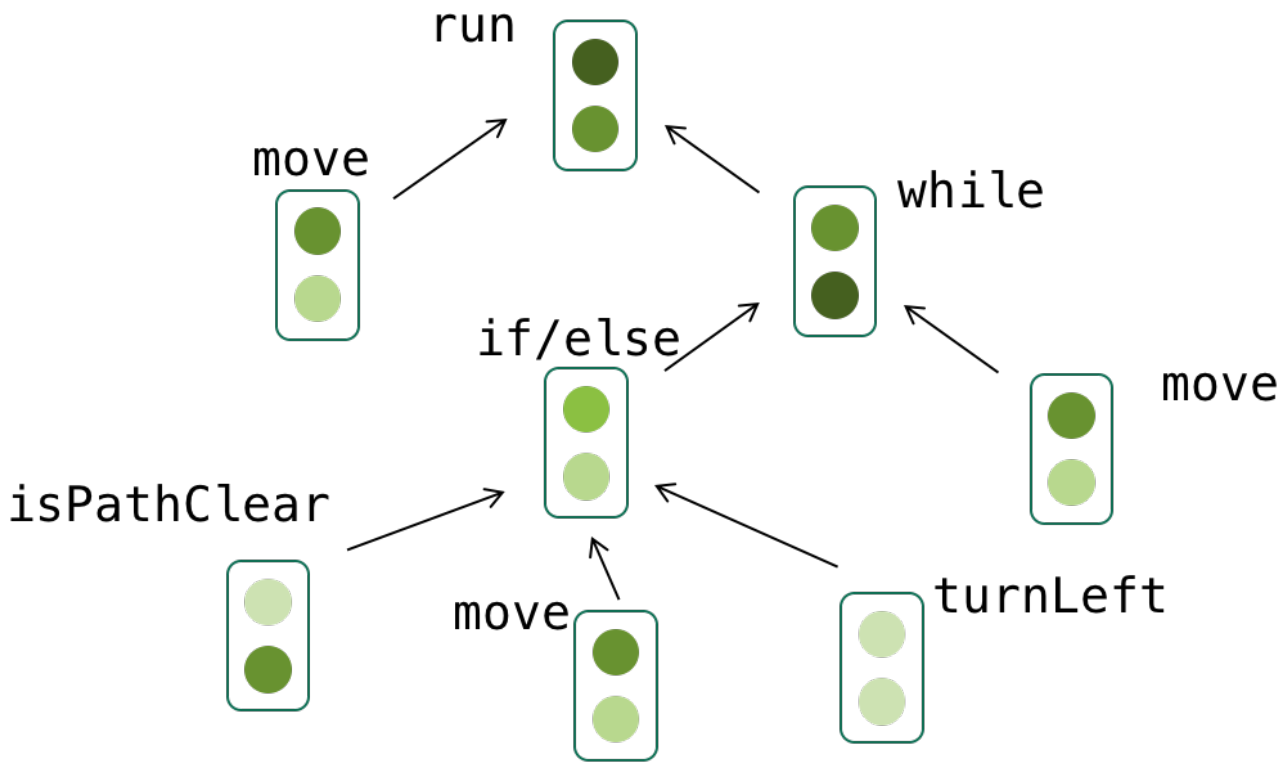


Websites


body
div
ul
li
a
li
a
li
a
div
p
img
p
img
table
tr
td
td
tr
td
td
div
div
div

Programs

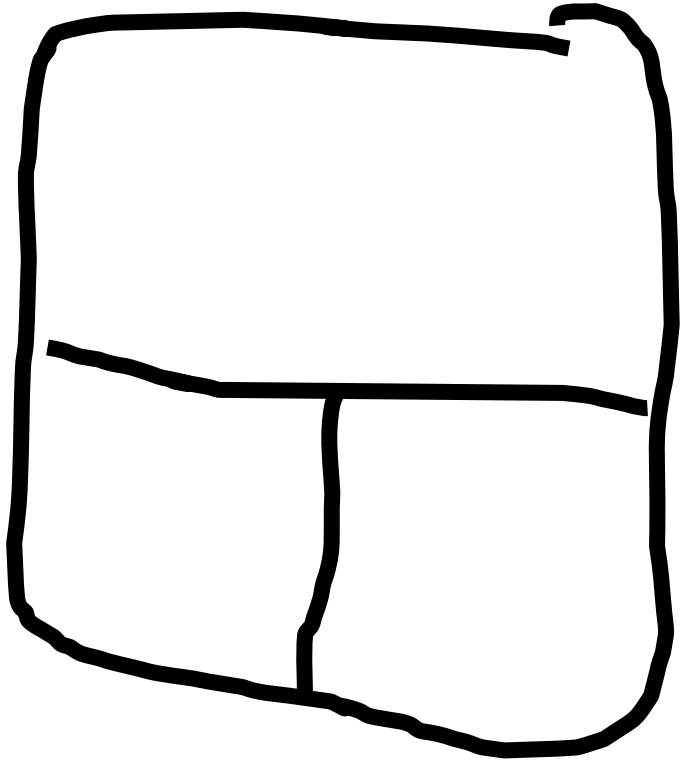
```
// Example student solution
function run() {
    // move then loop
    move();
    // the condition is fixed
    while (notFinished()) {
        if (isPathClear()) {
            move();
        } else {
            turnLeft();
        }
        // redundant
        move();
    }
}
```

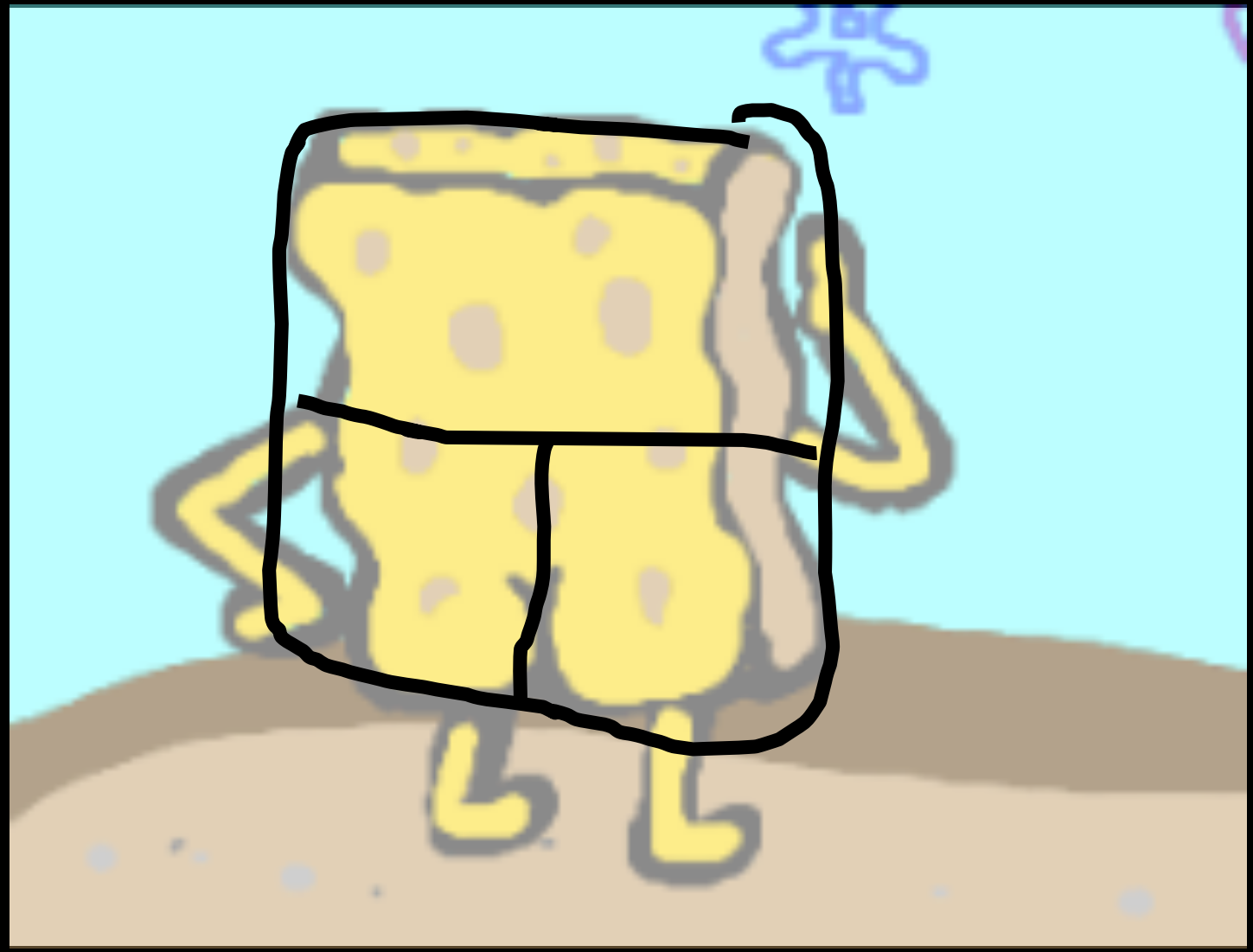


* This is a figure in an academic paper written by a recent CS106B student!

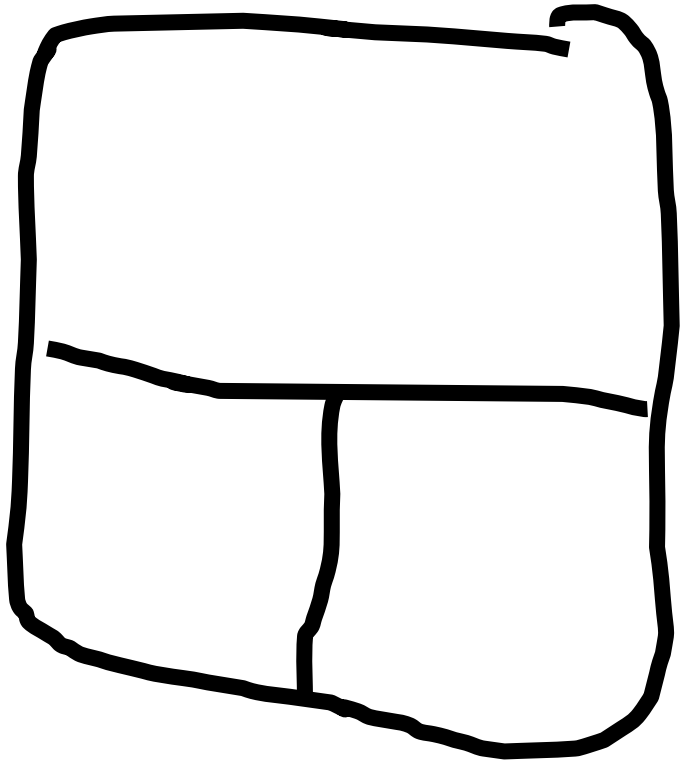
We have seen: Recursion on Trees

But How About Making a Tree Variable?

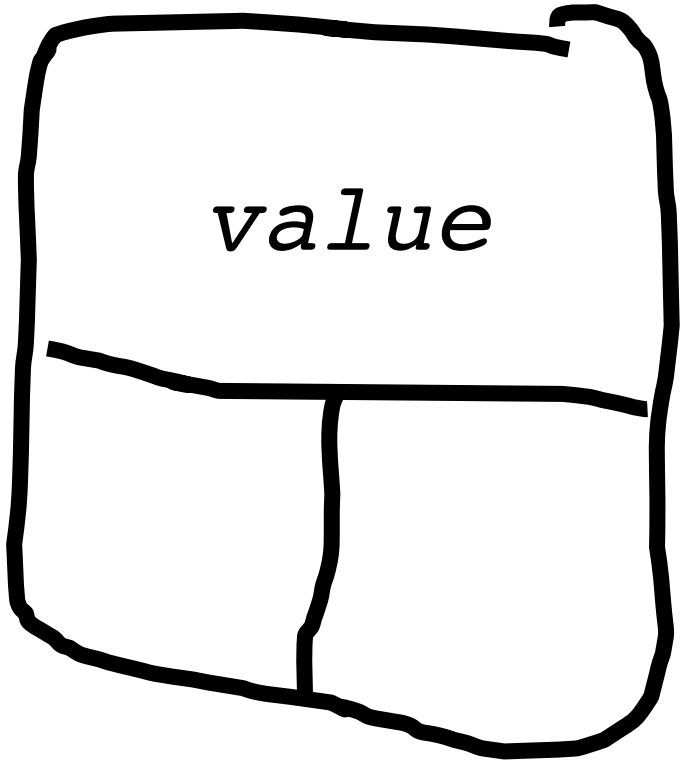




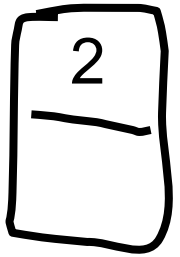
Binary Tree:



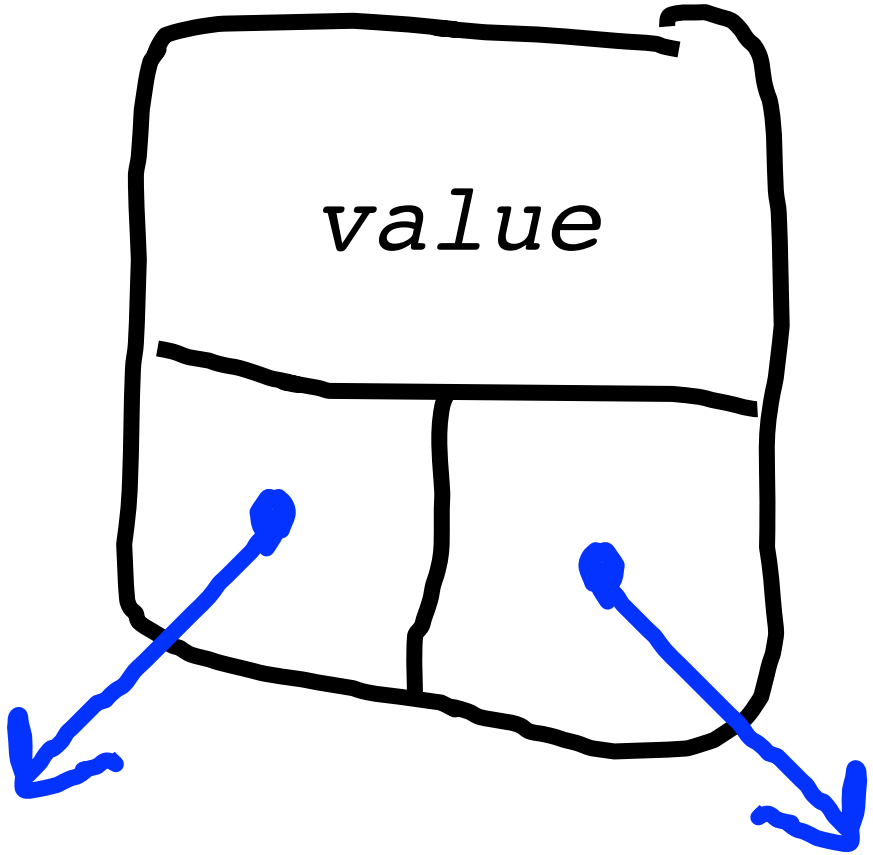
Binary Tree:



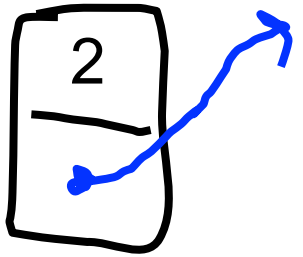
Linked list

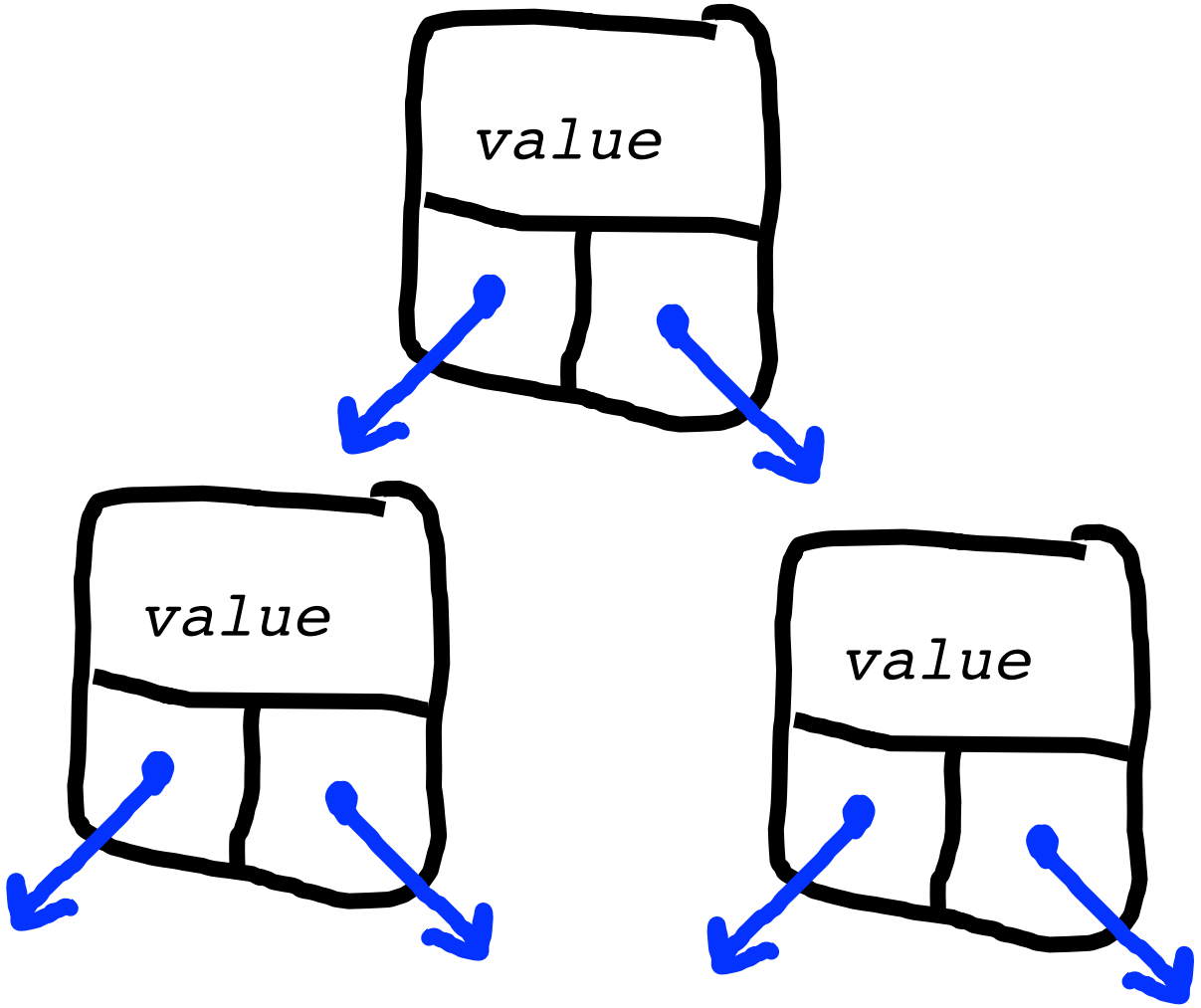


Binary Tree:

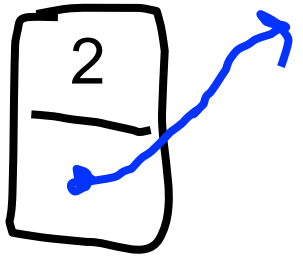


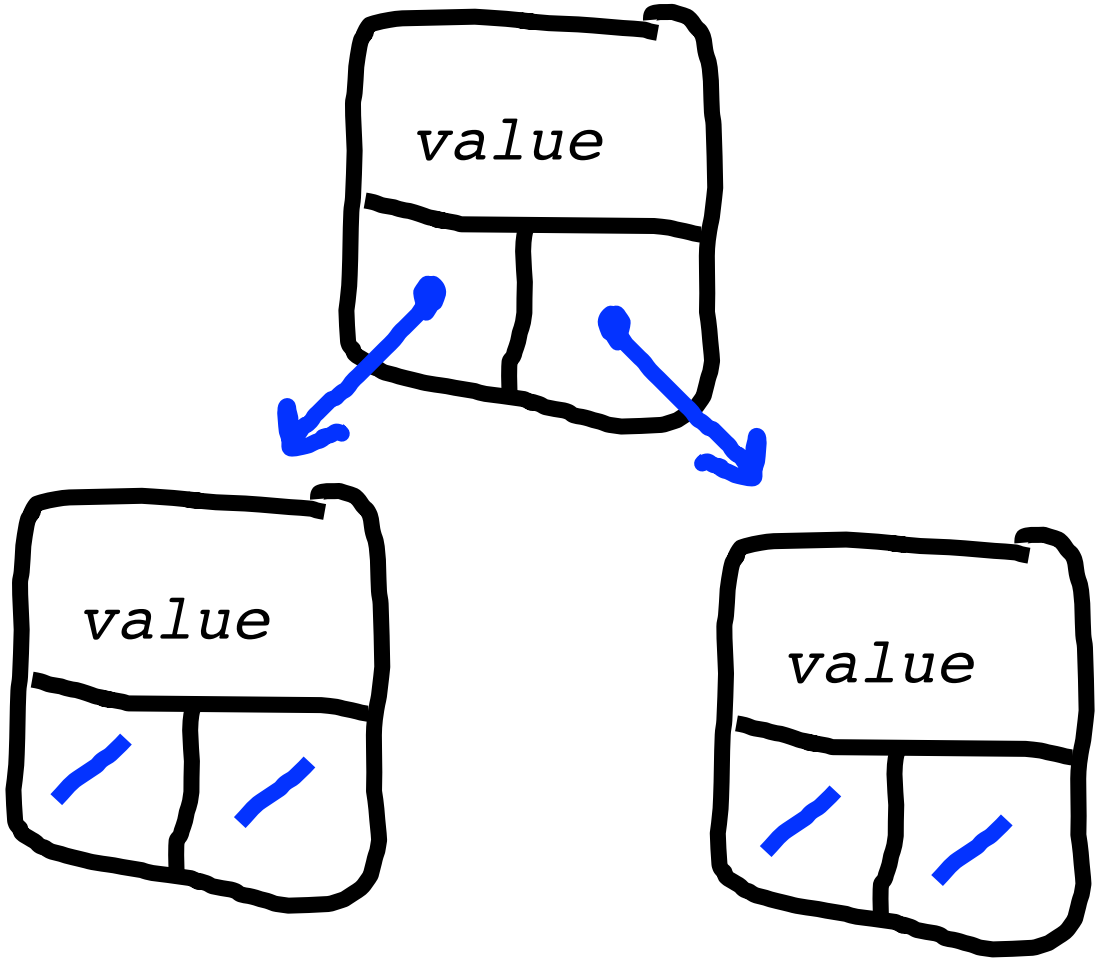
Linked list



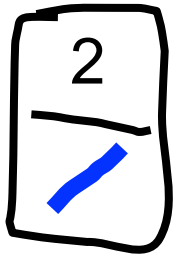


Linked list





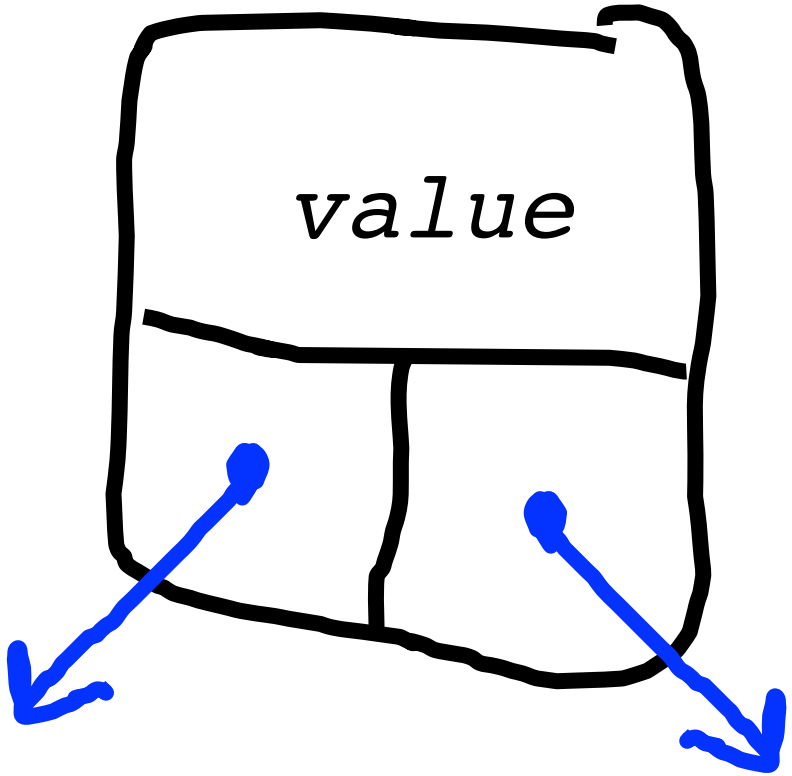
Linked list



Most Important Slide

Binary Tree

```
struct Tree {
  string value;
  Tree * left;
  Tree * right;
};
```

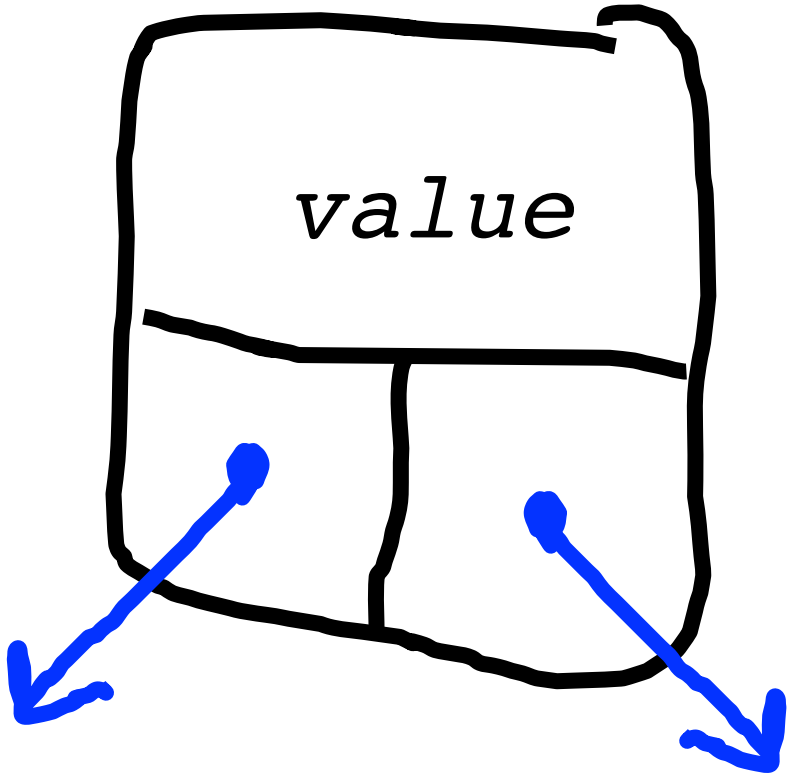


Does value have to be a string?

No

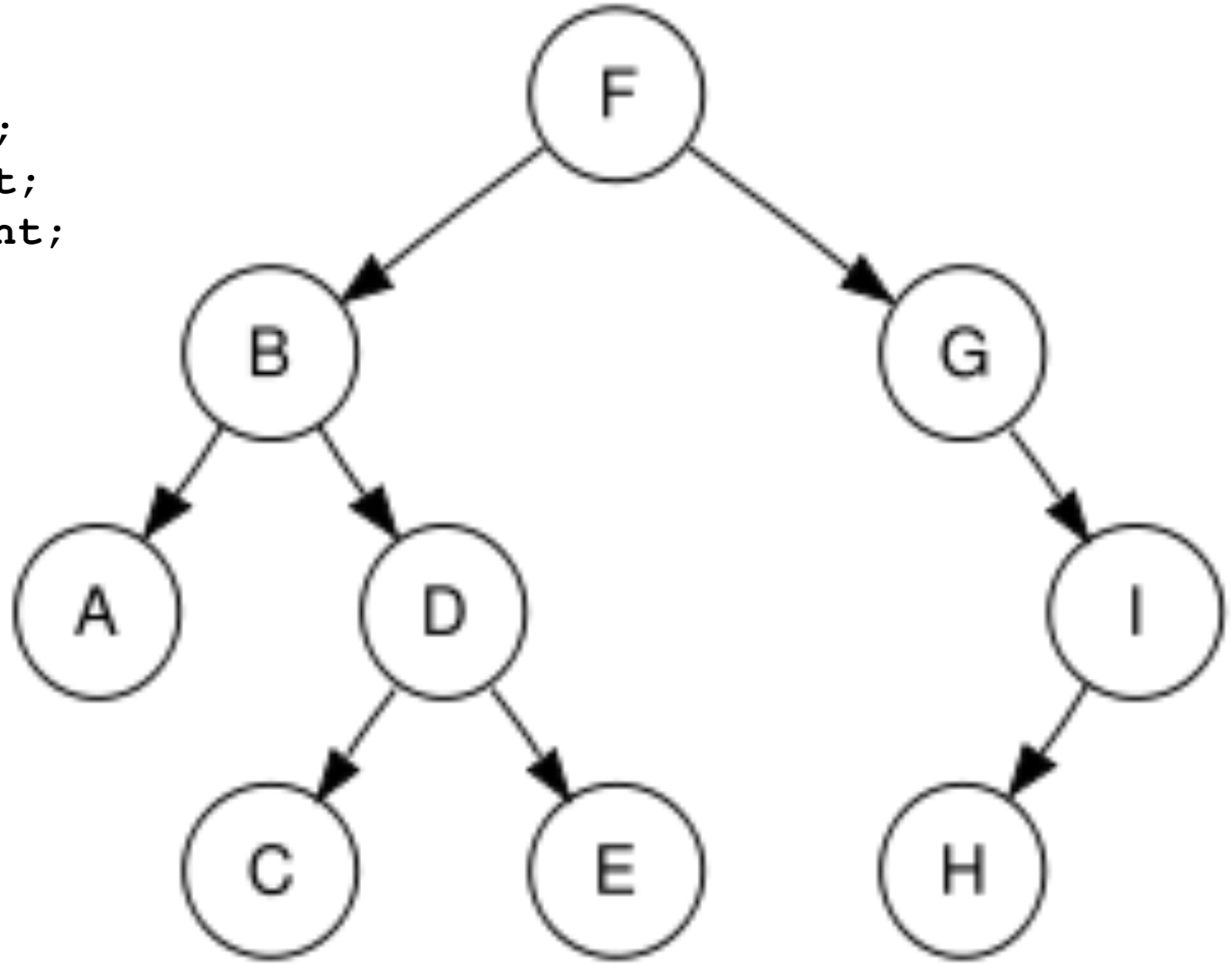
Binary Tree

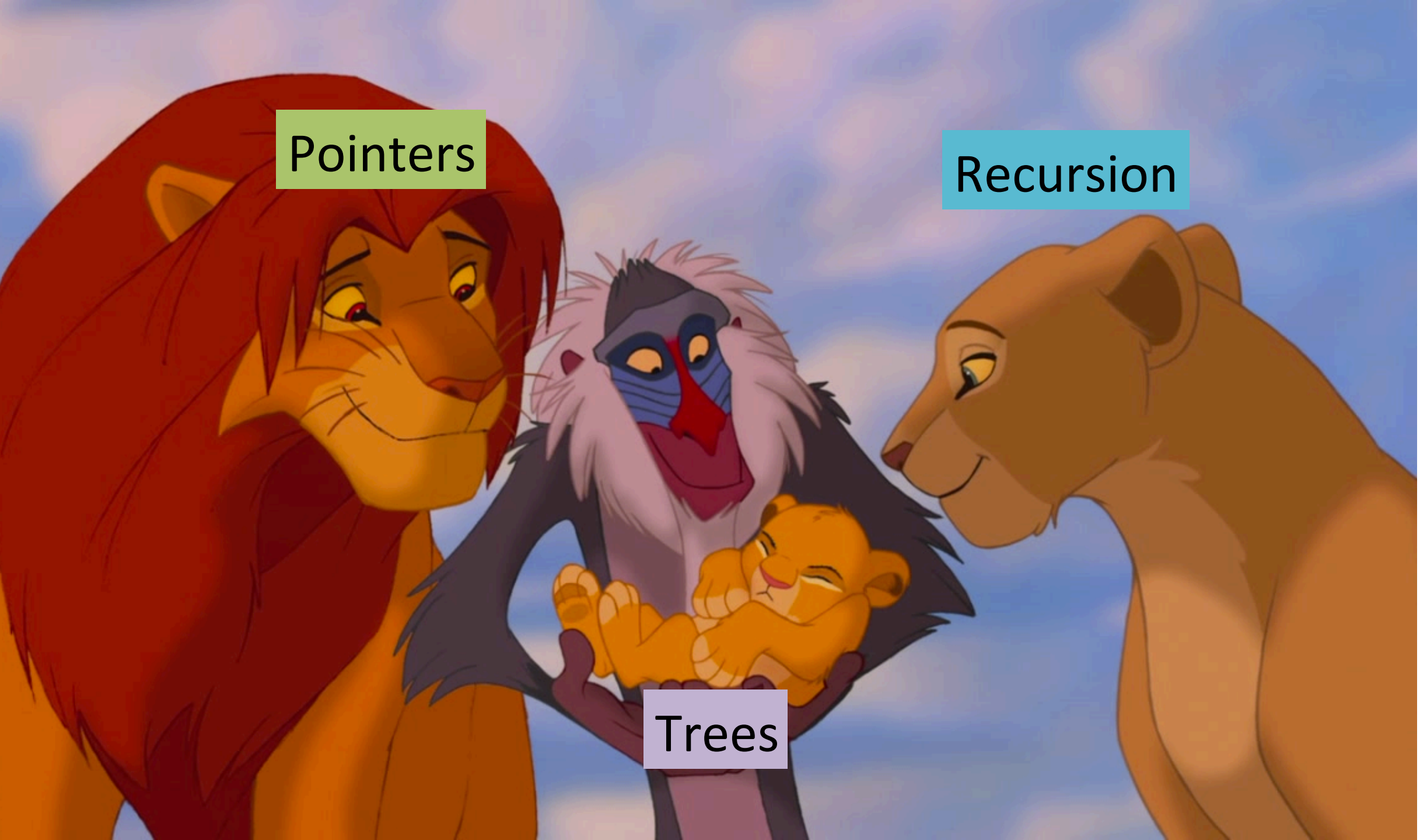
```
struct Tree {
  string value;
  Tree * left;
  Tree * right;
};
```



Example Binary Tree

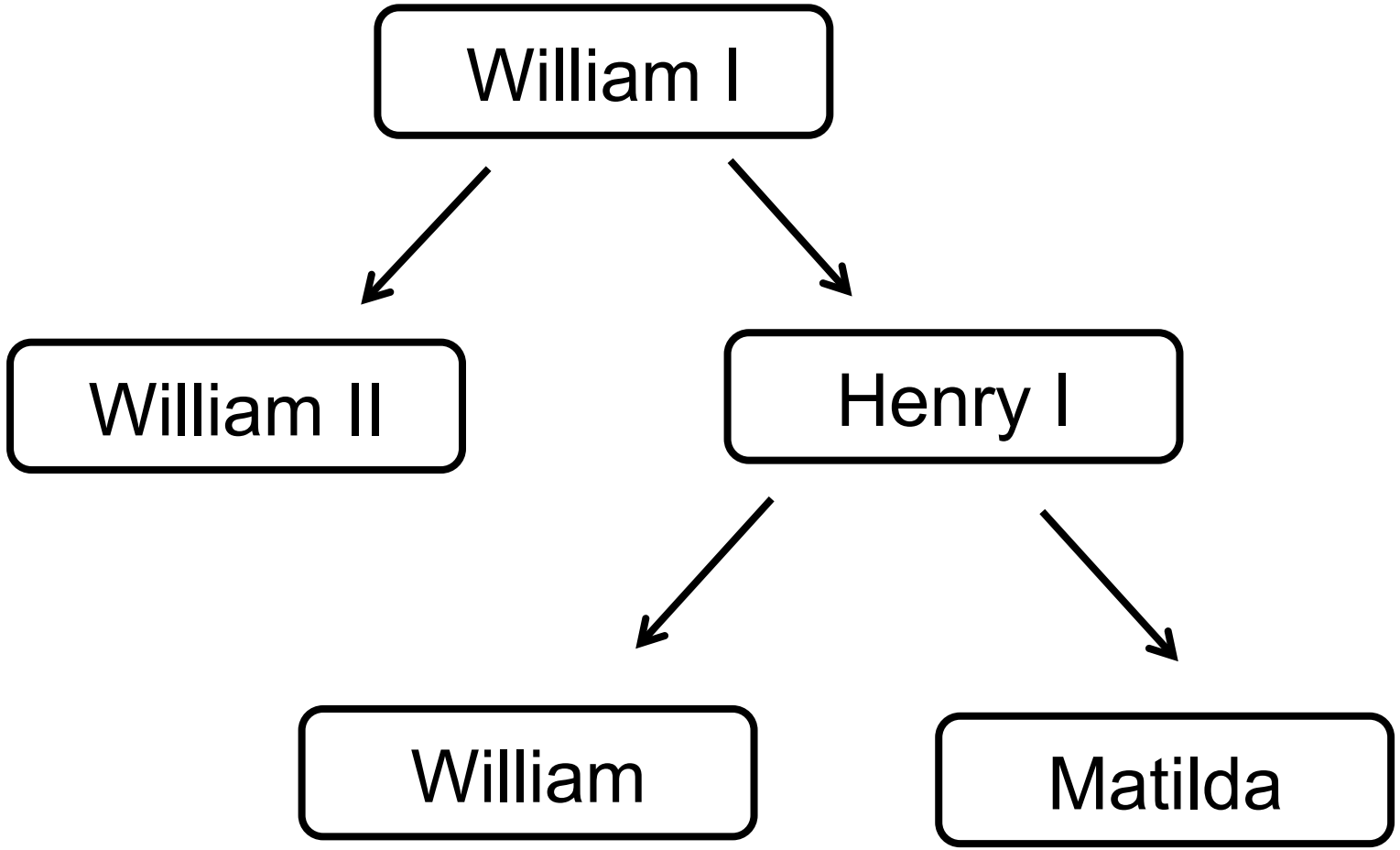
```
struct Tree {
    char value;
    Tree * left;
    Tree * right;
};
```



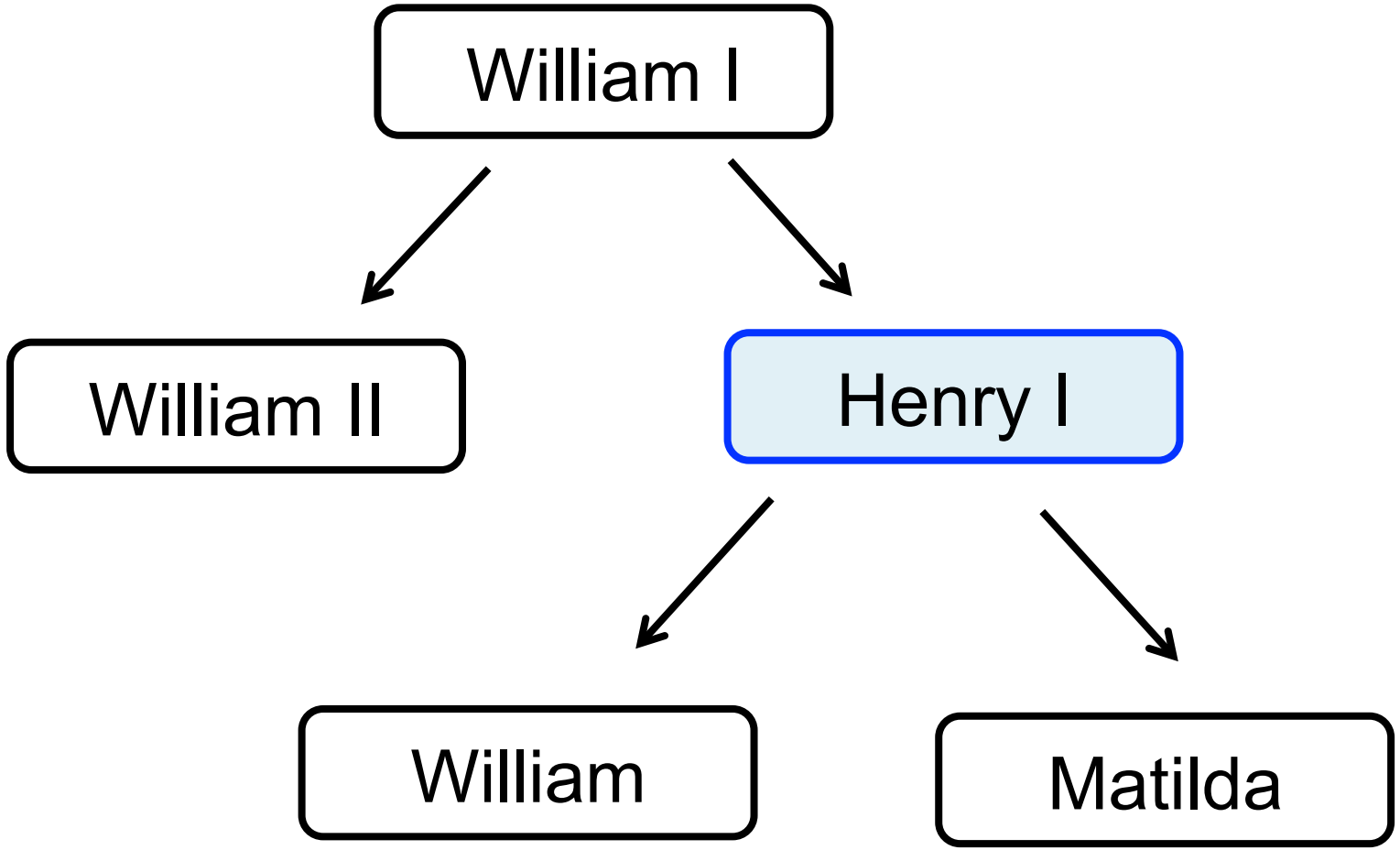

Pointers
Recursion
Trees

Terminology

Terminology

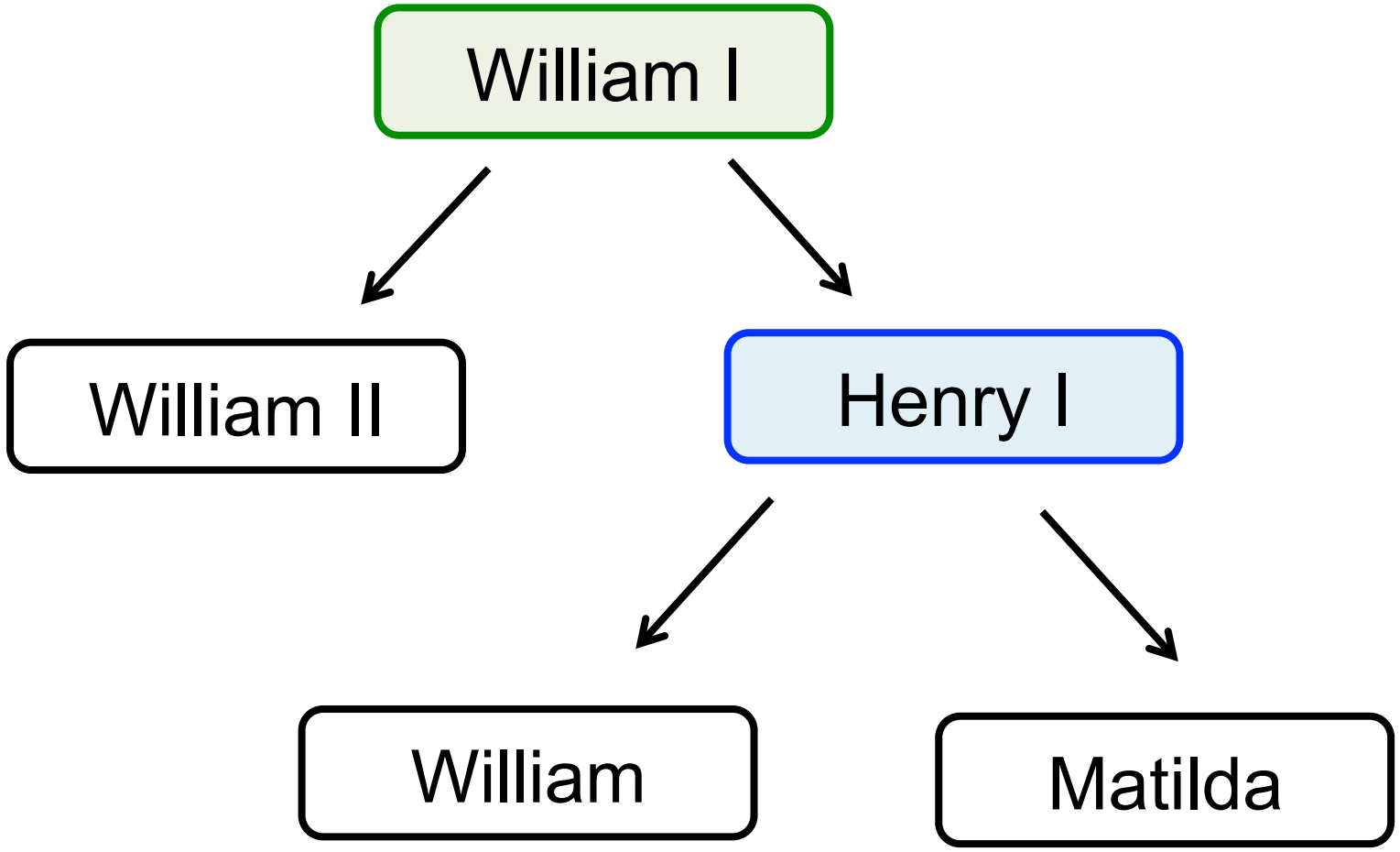


Terminology



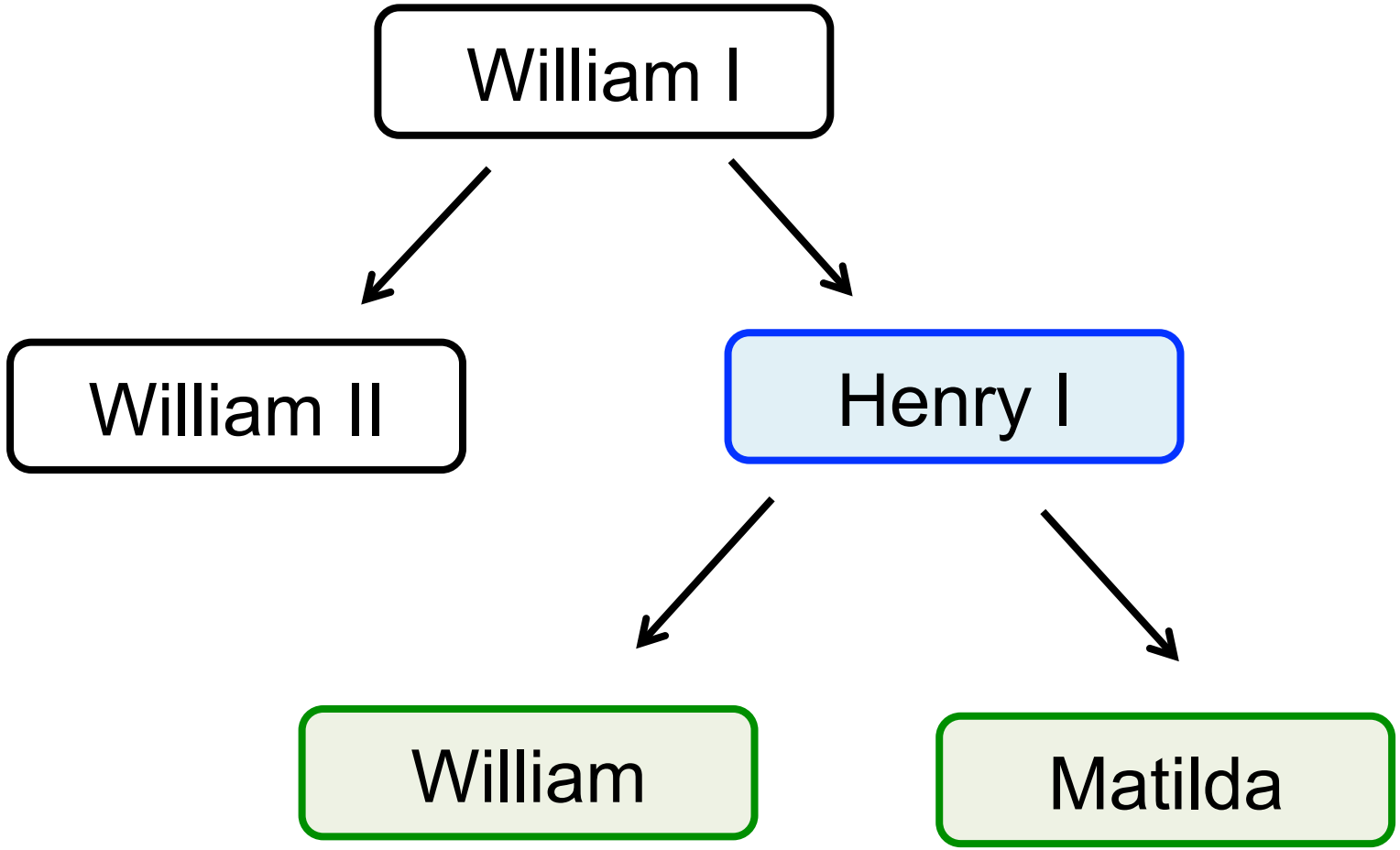
We call a single element a **“Node”** or **“Tree”**

Terminology: Parent



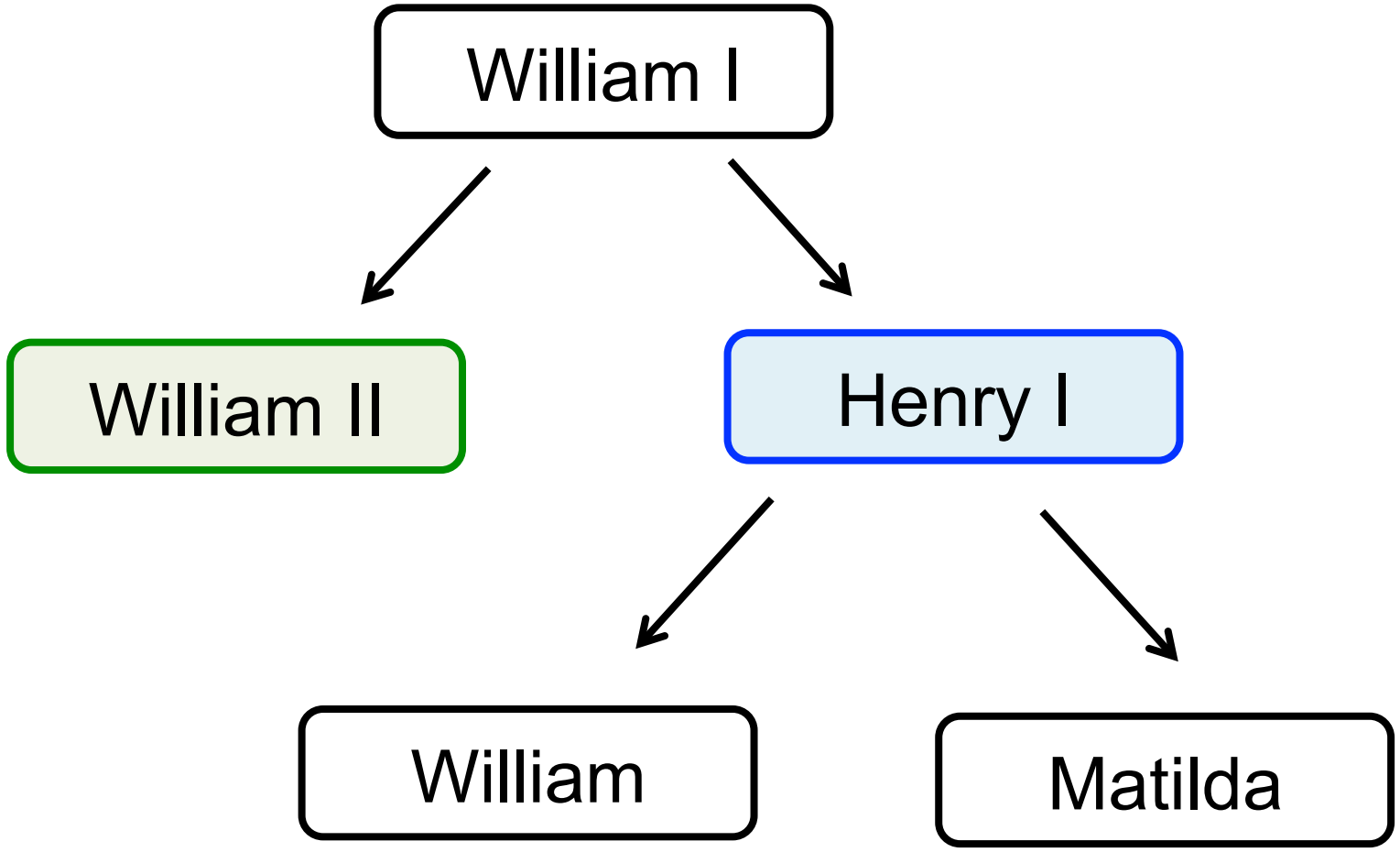
We call the Tree that points to us our **“Parent”**

Terminology: Children



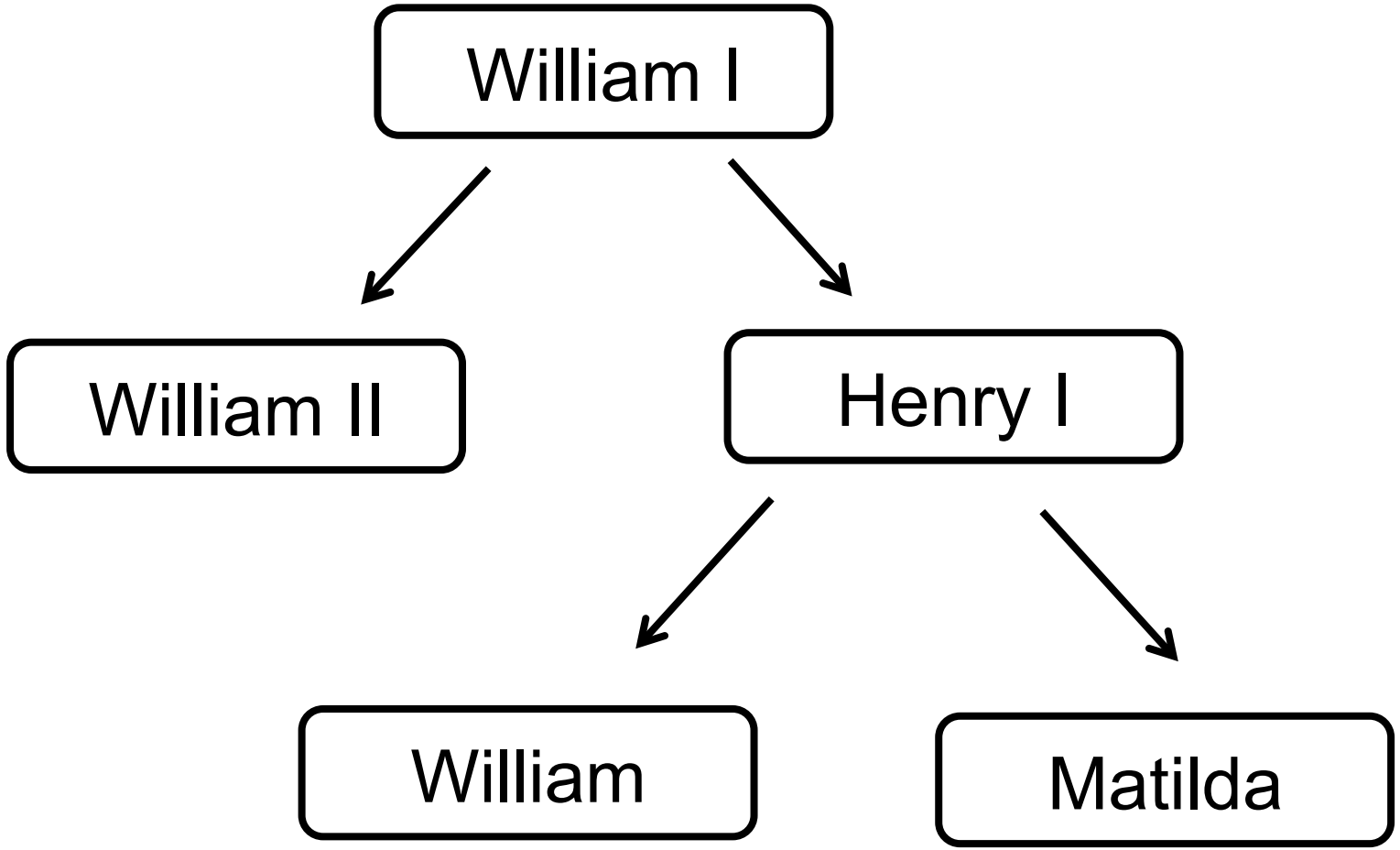
We call the nodes that we point to our **“Children”**

Terminology: Siblings

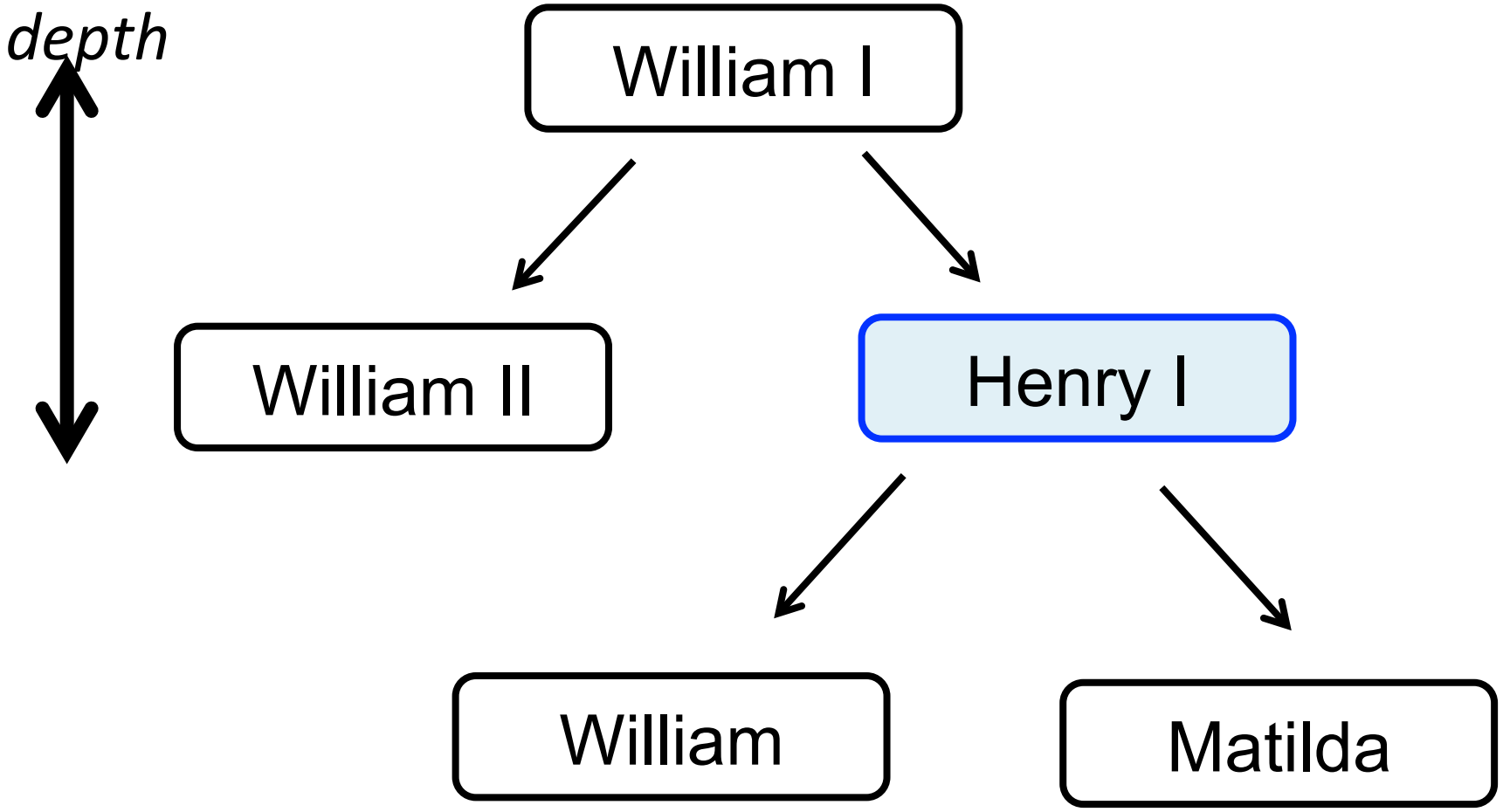


We call nodes that share the same parent **“Siblings”**

Terminology

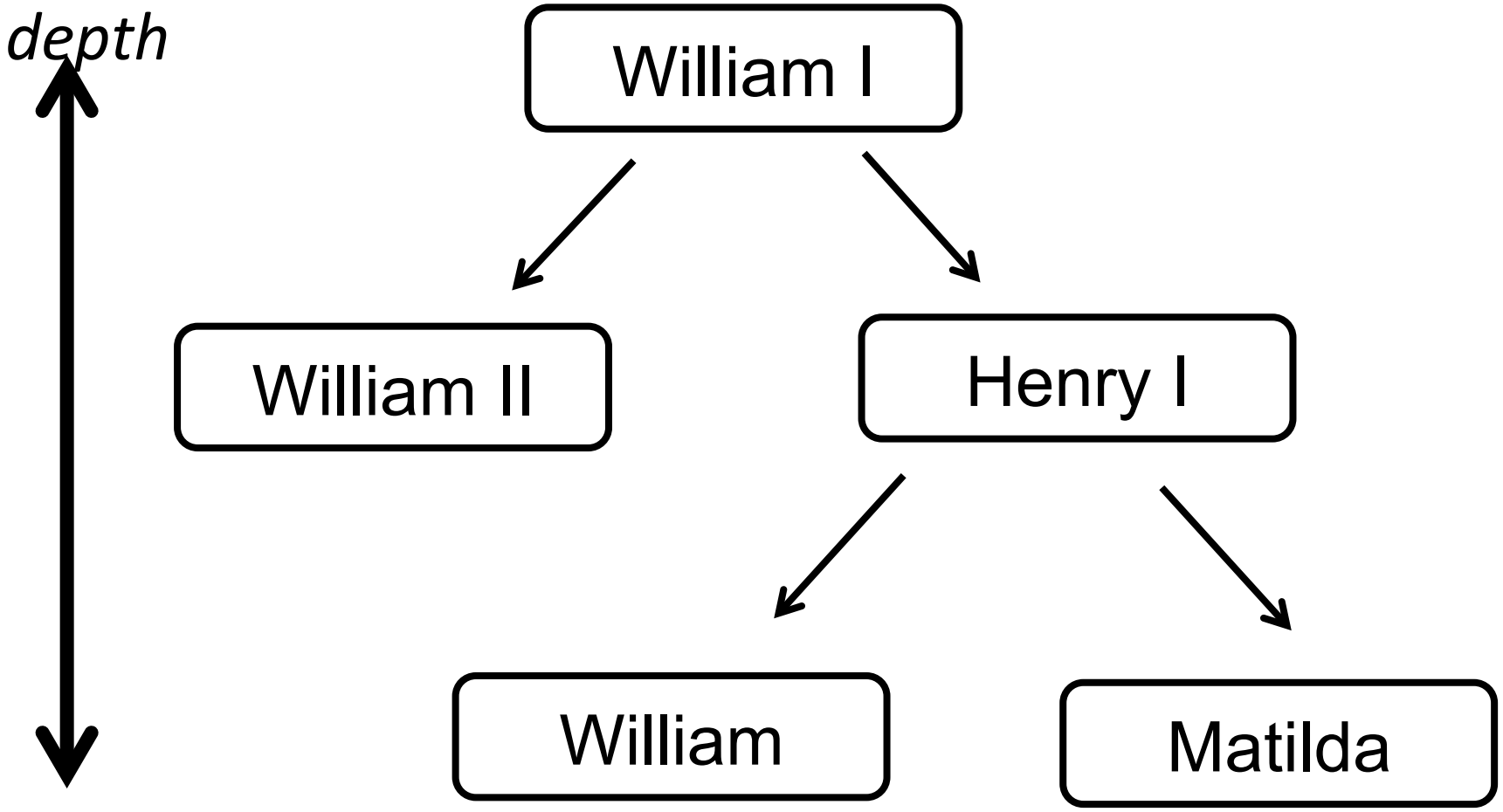


Terminology



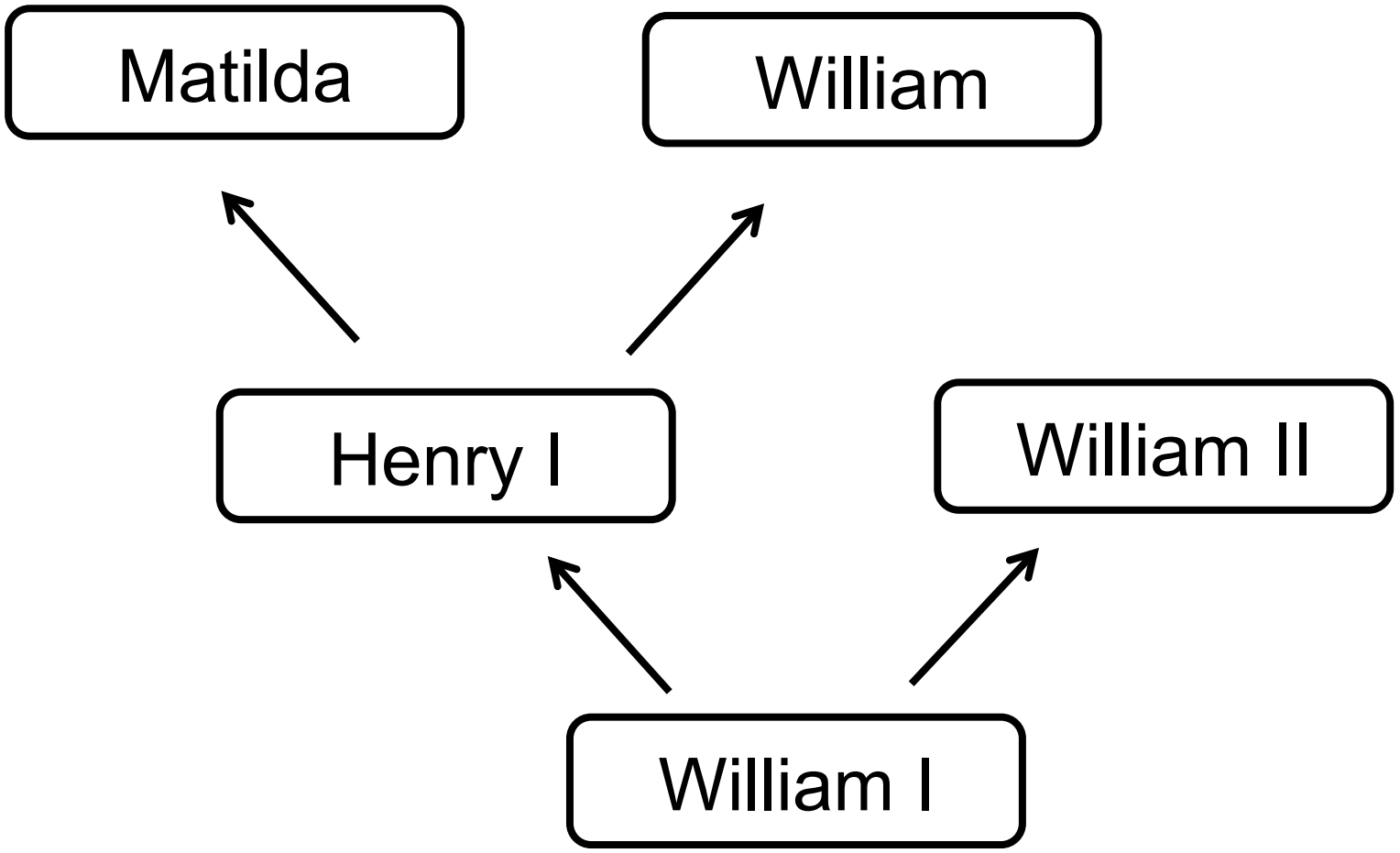
The depth of a node is the number of links from the root.

Terminology

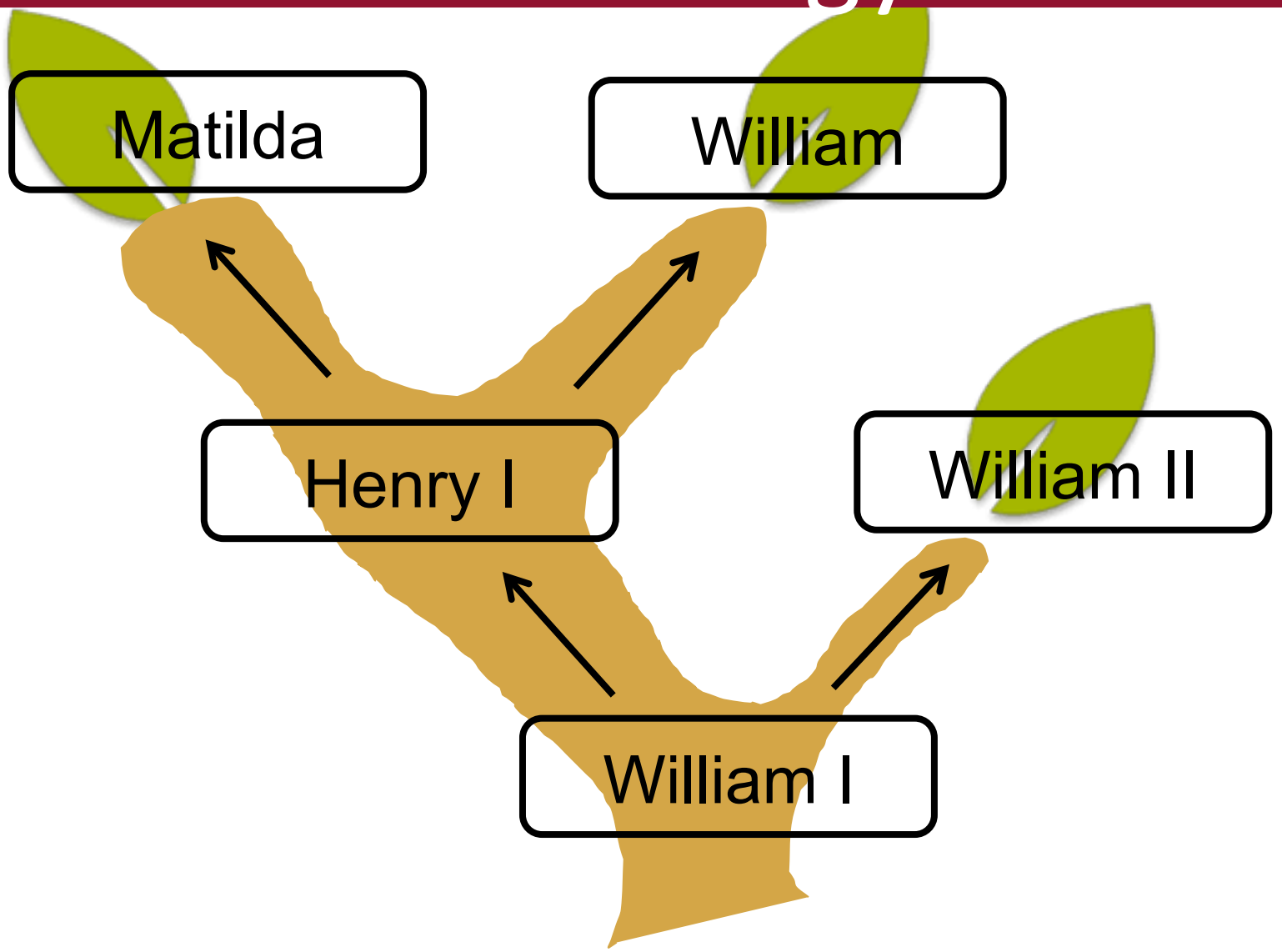


The depth of a tree is the maximum number of links from the root.

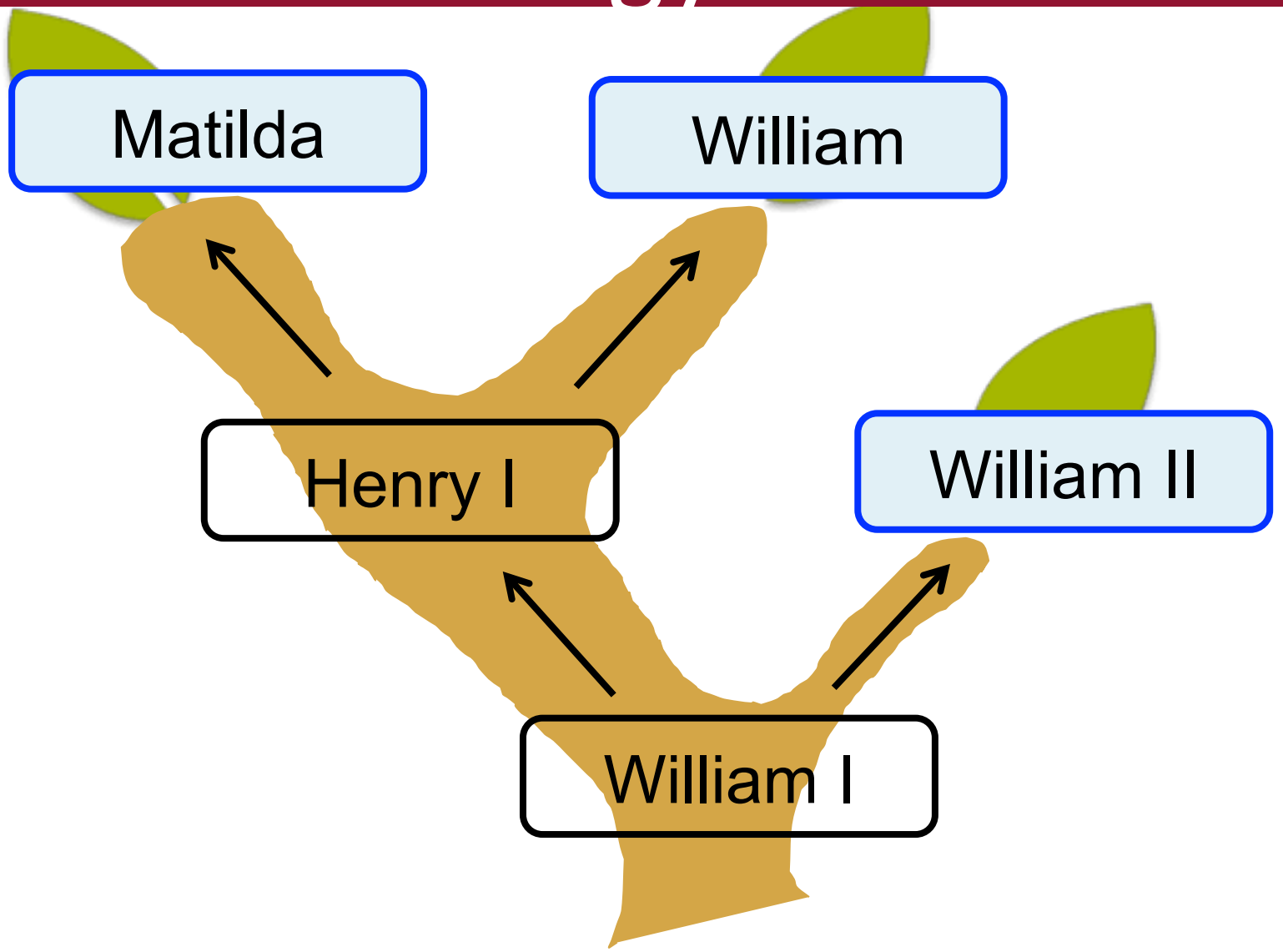
Terminology



Terminology

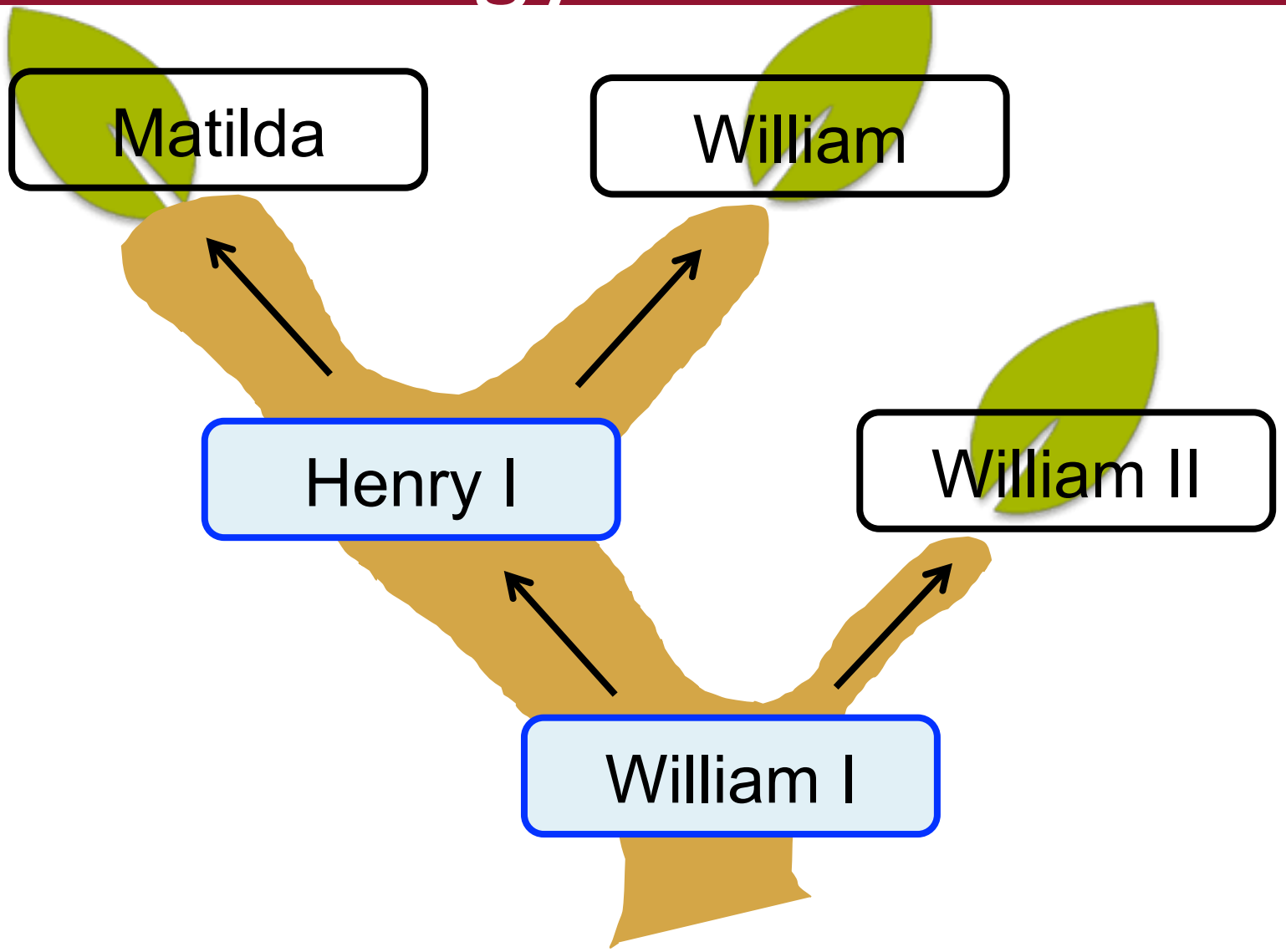


Terminology: Leaves



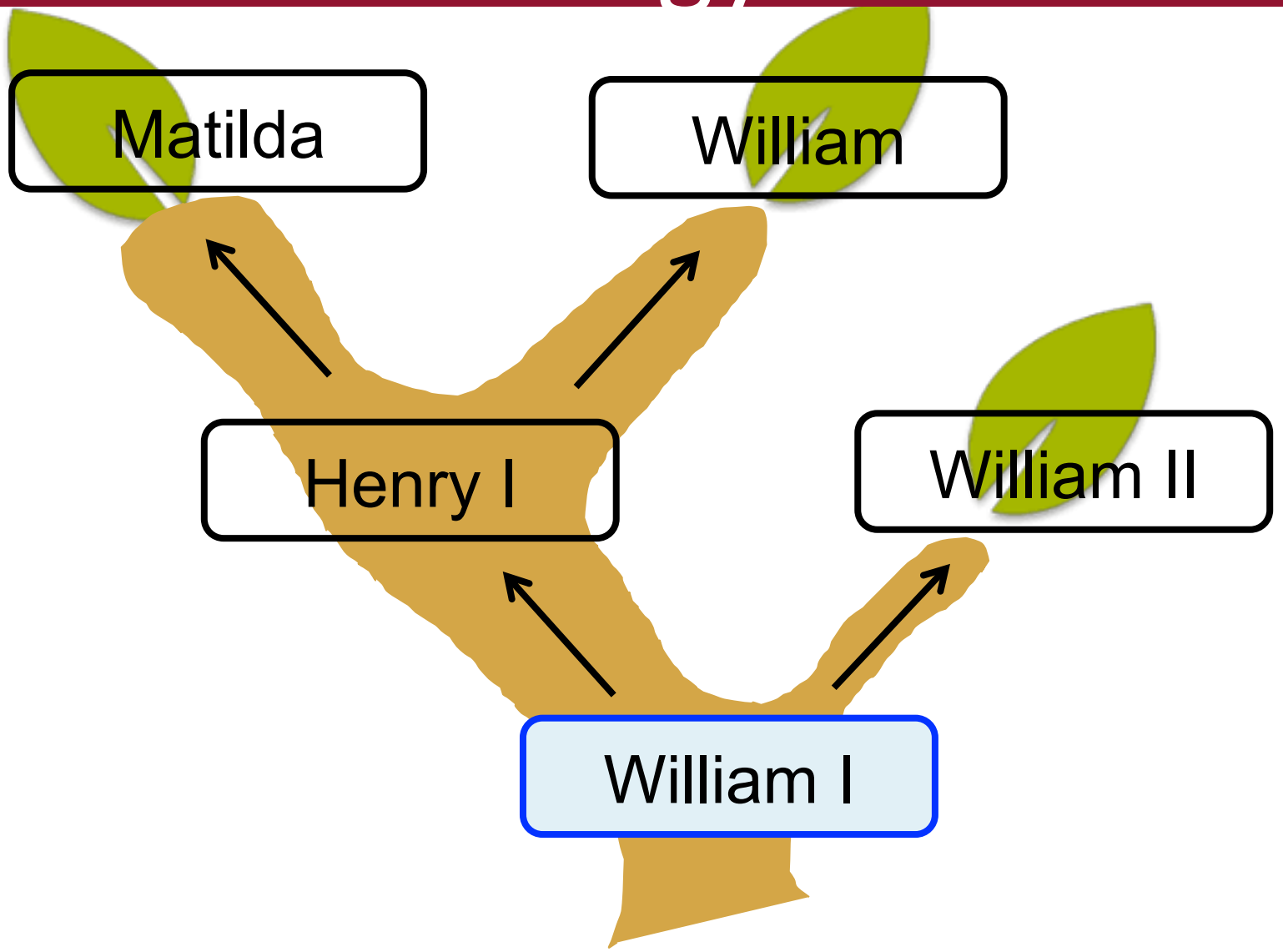
We call nodes that have no children **“Leaves”**

Terminology: Inner Nodes



We call nodes that have children **“Inner Nodes”**

Terminology: Root



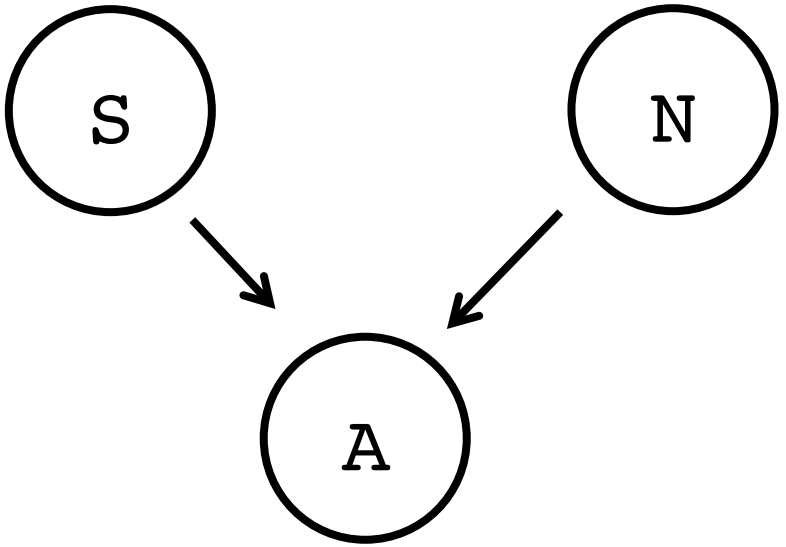
We call the node without parents the **“Root”**

Tree Definition

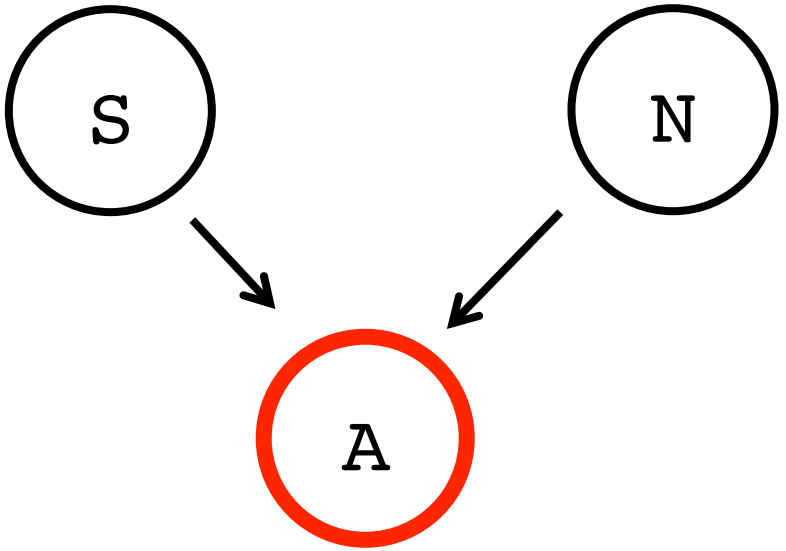
Only One Parent

No Cycles

Only One Parent

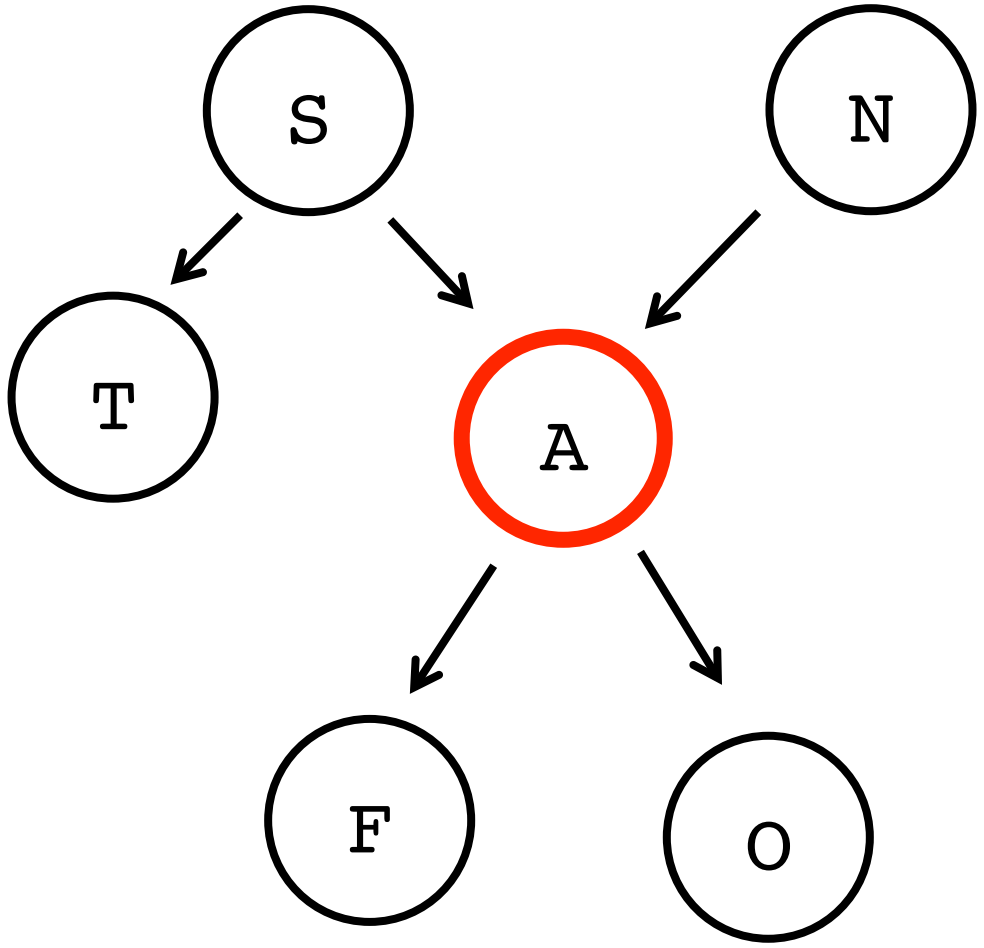


Only One Parent



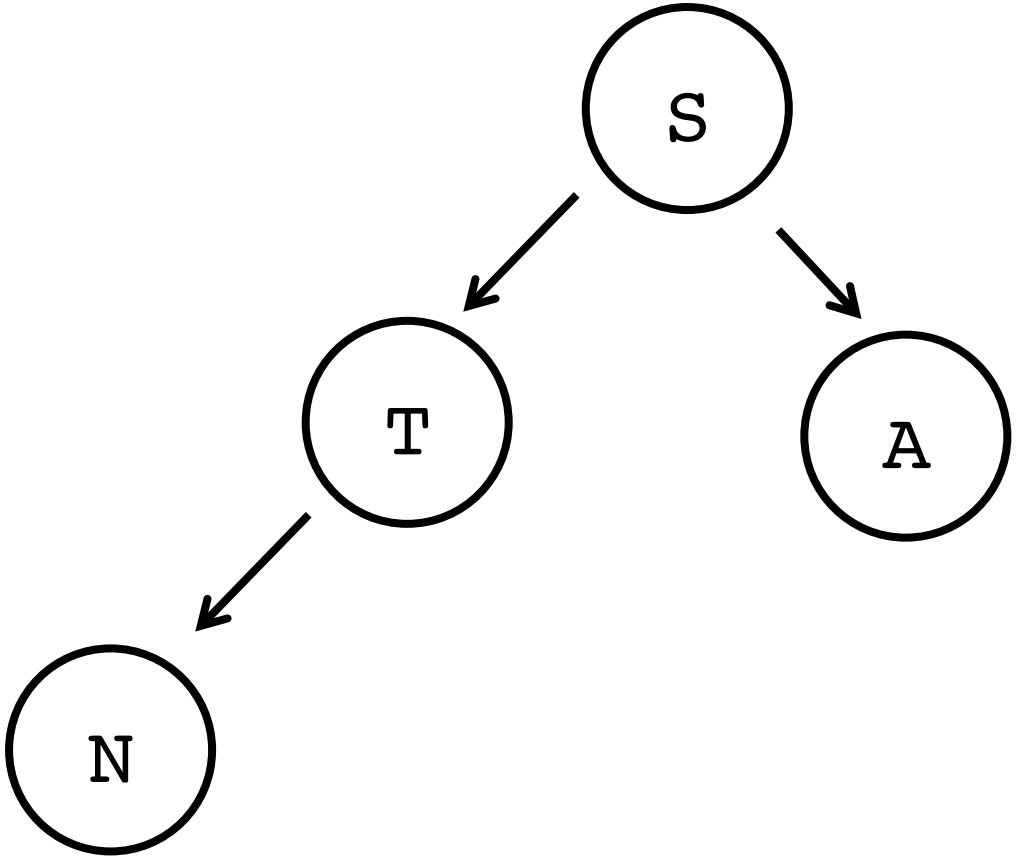
This is not a tree because the red node has two parents

Only One Parent

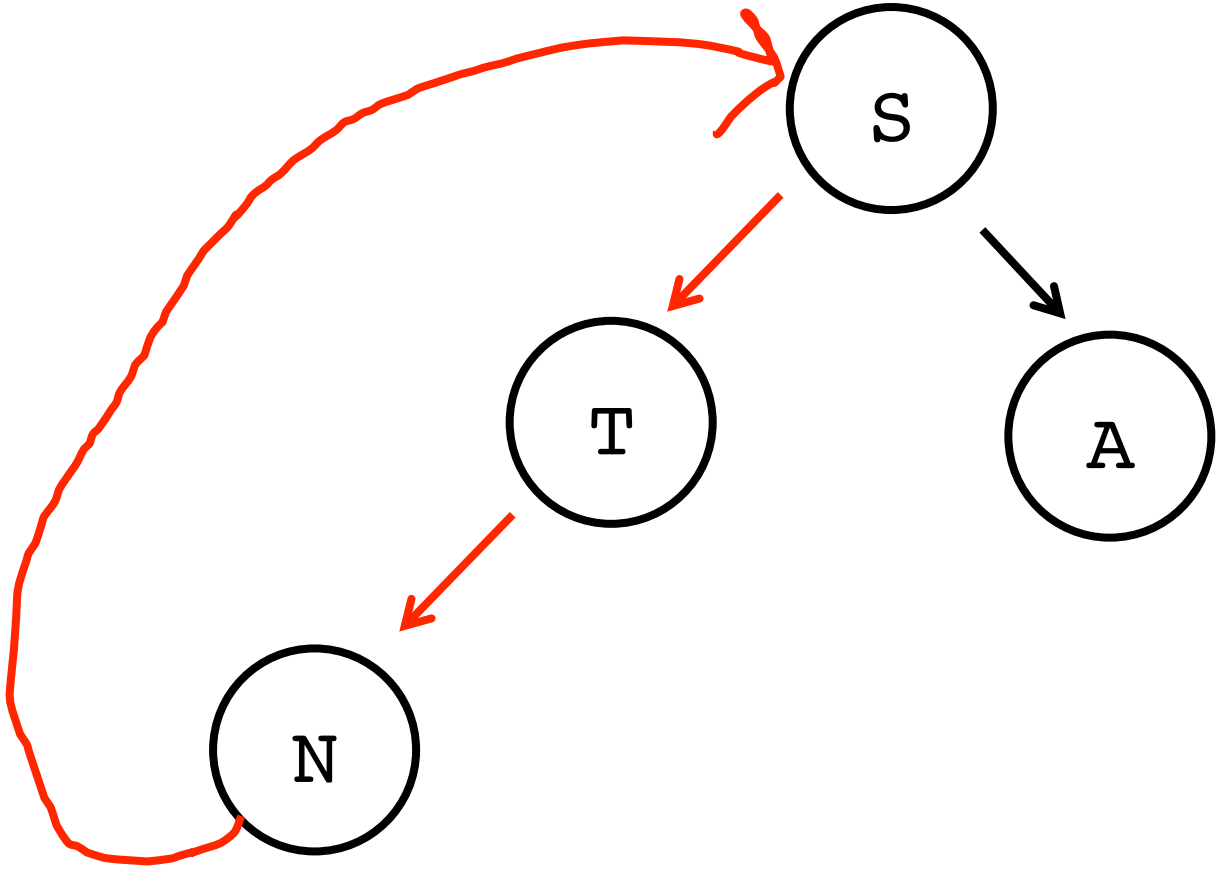


This is not a tree because the red node has two parents

No Cycles

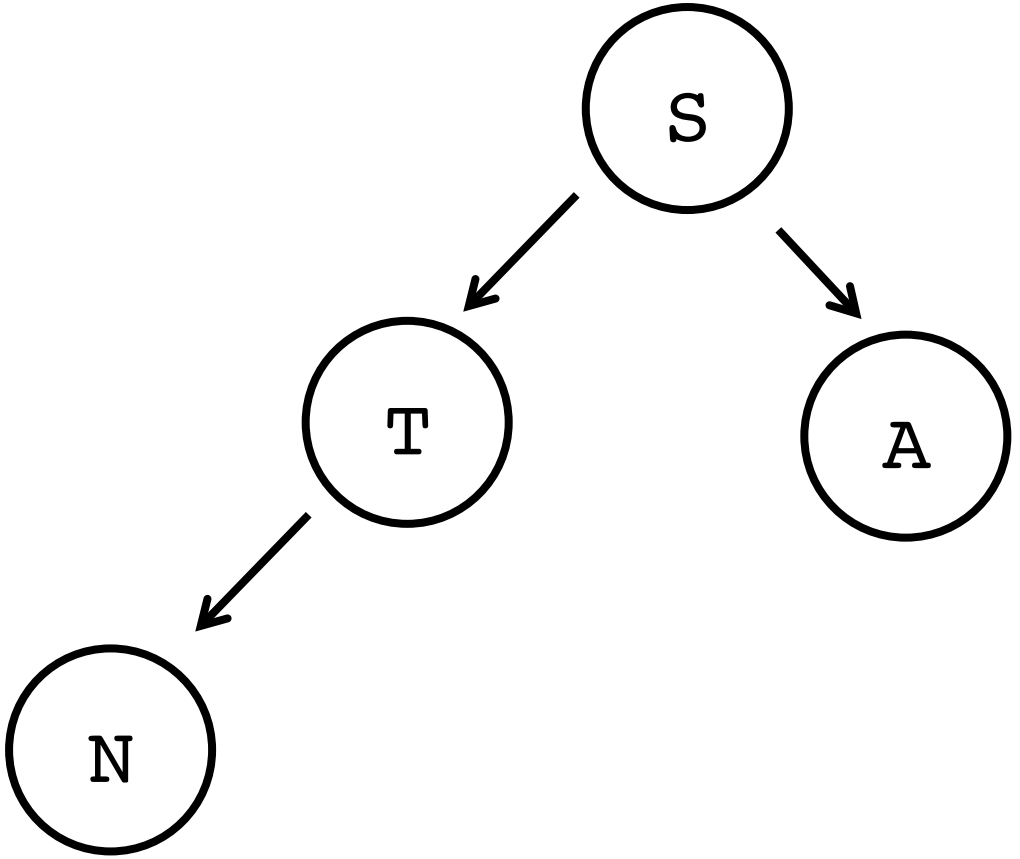


No Cycles

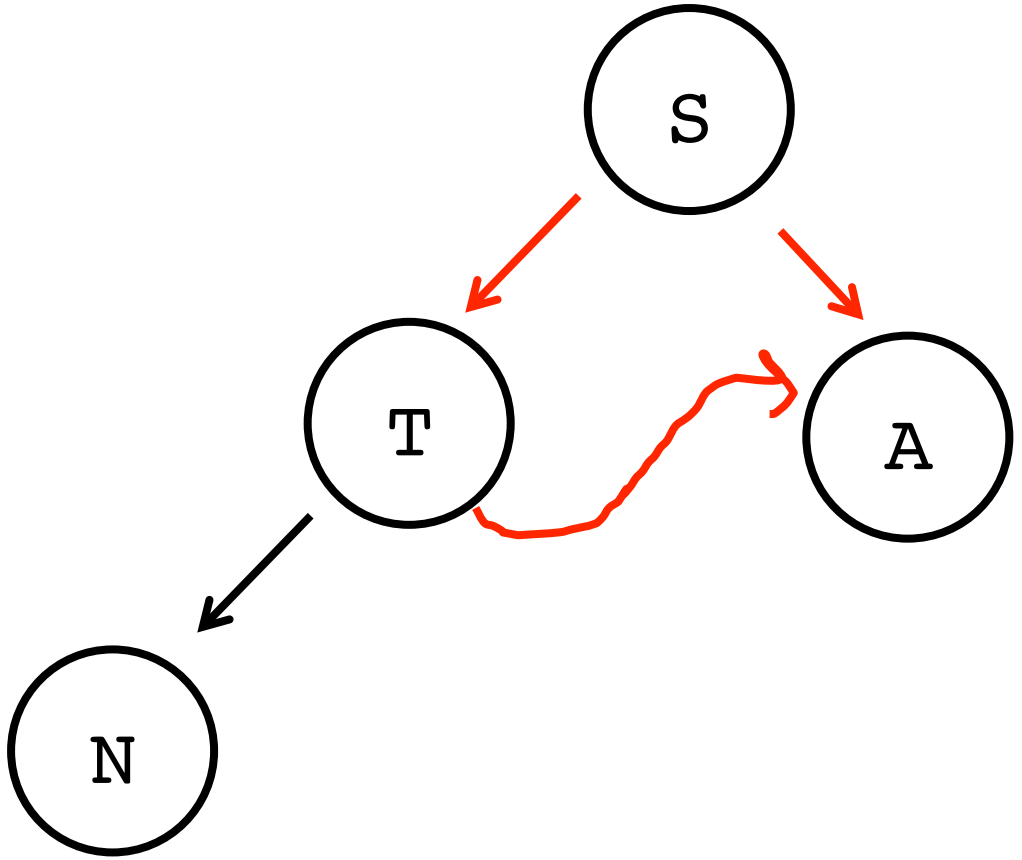


This is not a tree because the red edges make a cycle

No Cycles



No Cycles



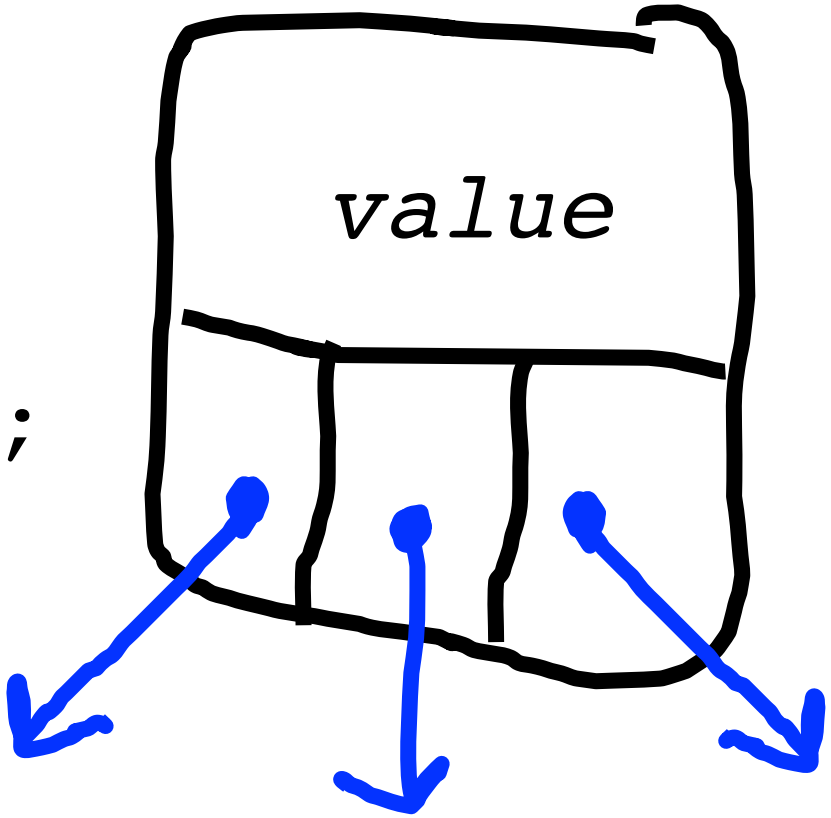
This is not a tree because the red edges make a cycle

Binary tree:
At most two children
and both children are also binary trees

Other Types of Trees

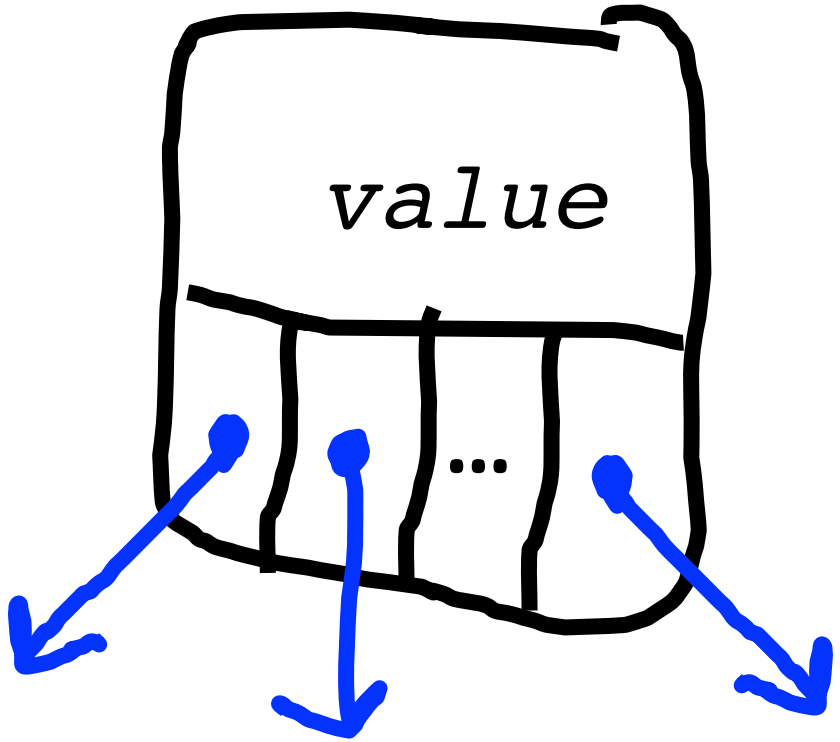
Trinary Tree

```
struct Tree {
  string value;
  Tree * left;
  Tree * middle;
  Tree * right;
};
```



N-ary Tree

```
struct Tree {
   string value;
   Vector<Tree *> children;
};
```



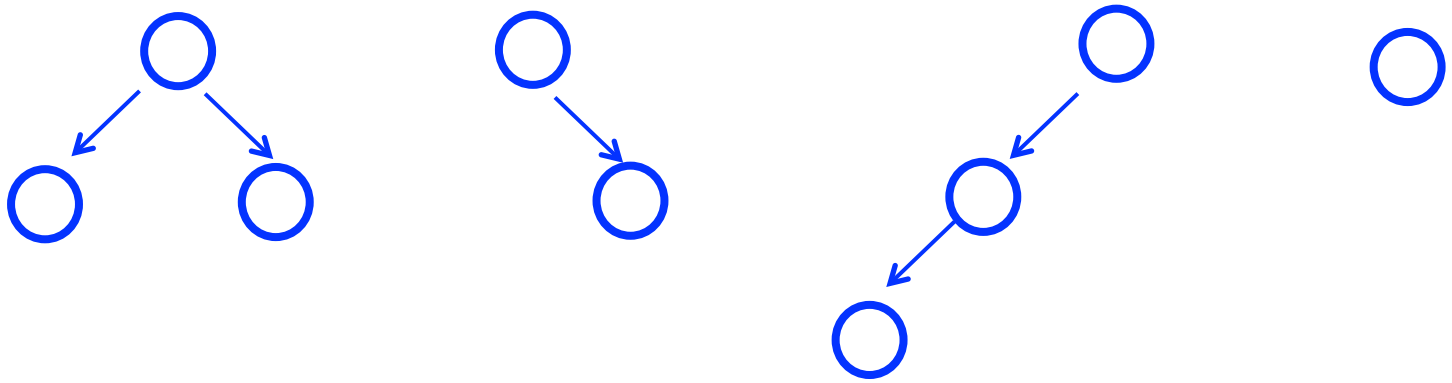
Either Struct or Class

```
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```

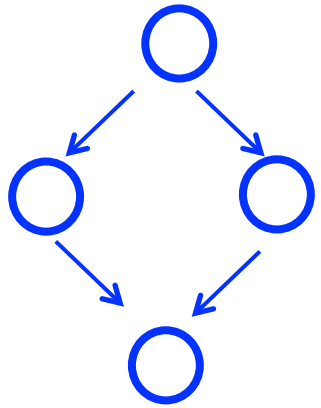
```
class Tree {
private:
    string value;
    Vector<Tree *> children;
};
```

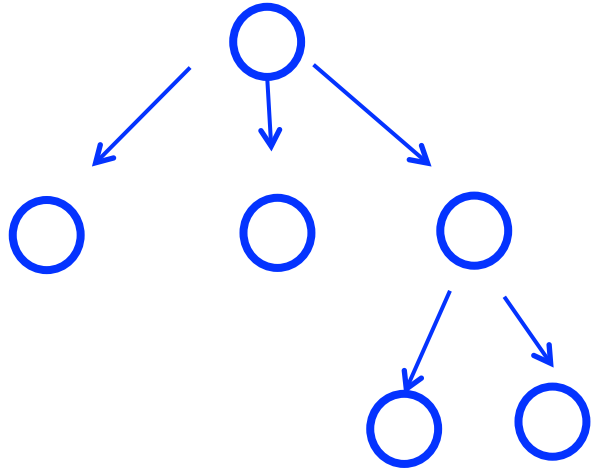
How many are valid binary trees?

A) 3 B) 4 C) 5 D) 6 E) 7



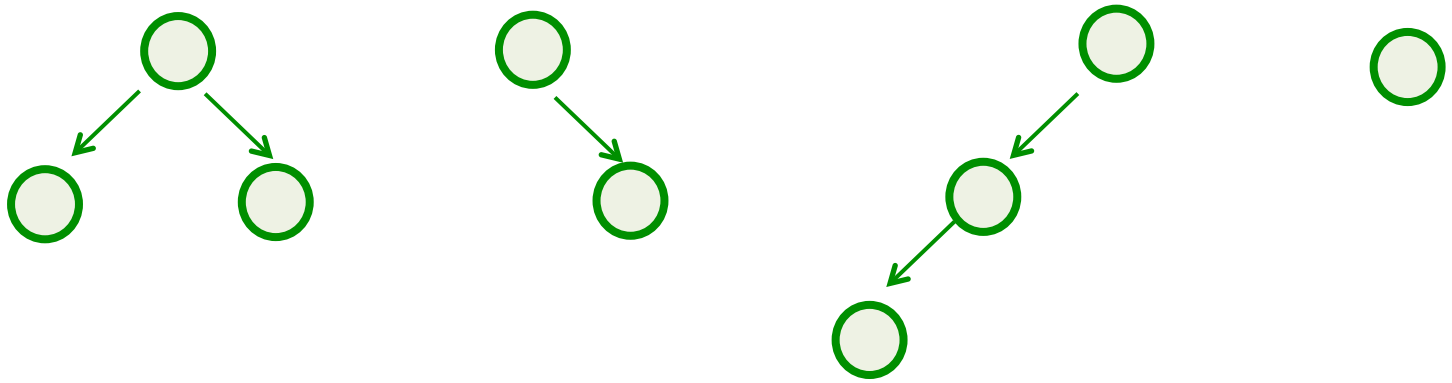
NULL



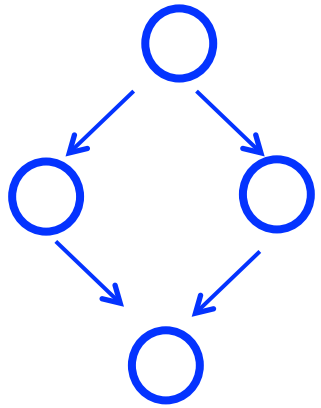


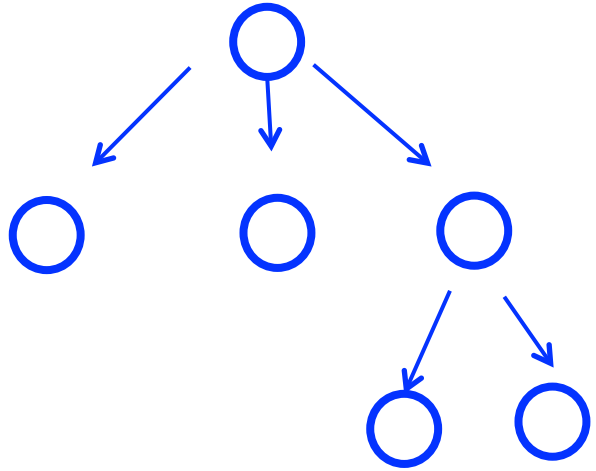
How many are valid binary trees?

A) 3 B) 4 C) 5 D) 6 E) 7



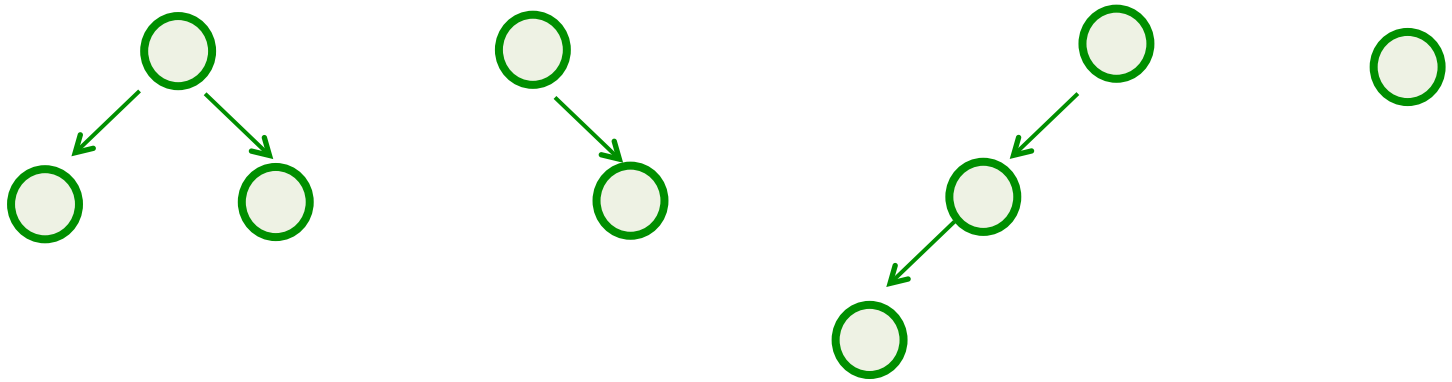
NULL



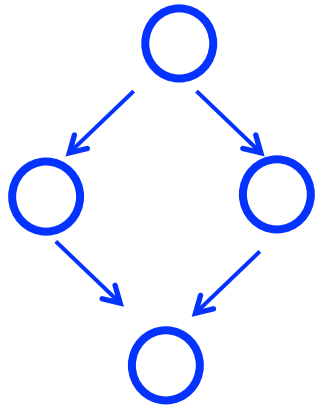


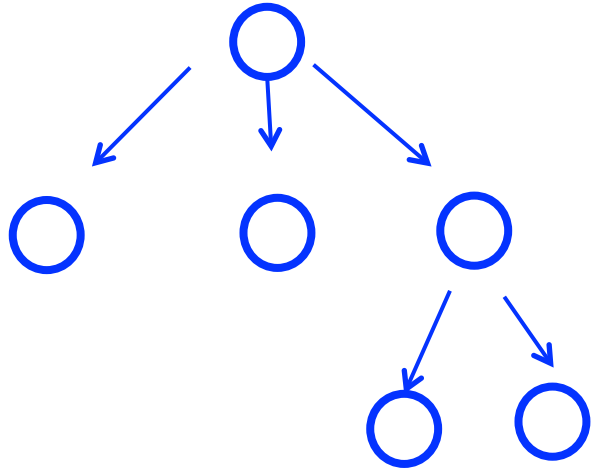
How many are valid binary trees?

A) 3 B) 4 C) 5 D) 6 E) 7



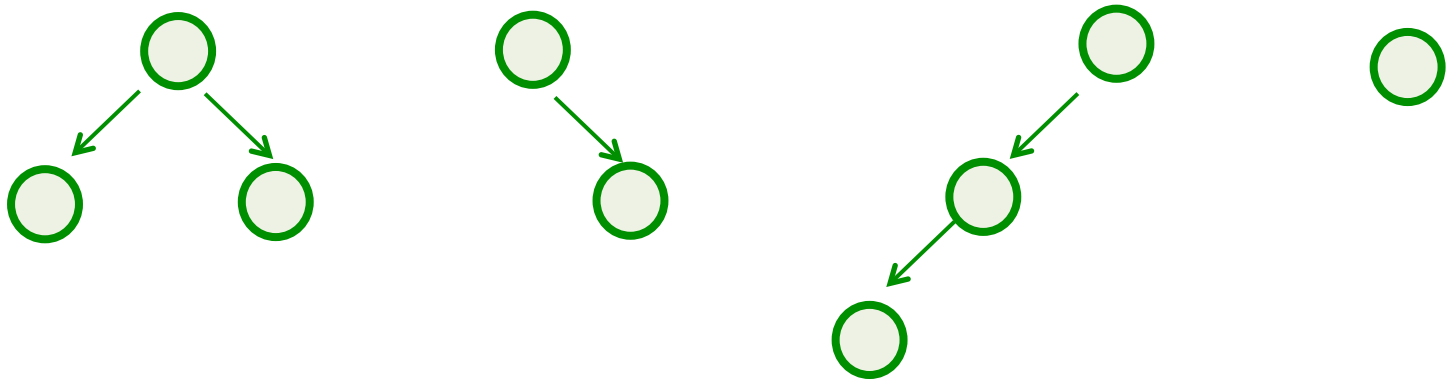
NULL



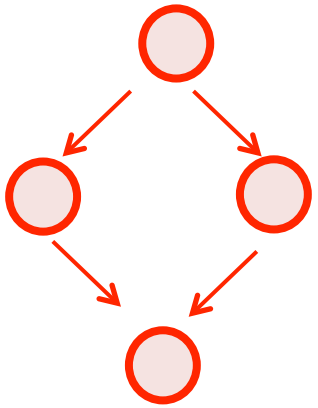


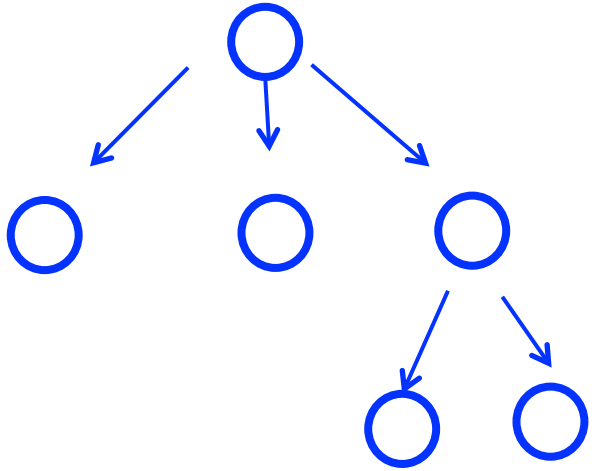
How many are valid binary trees?

A) 3 B) 4 C) 5 D) 6 E) 7



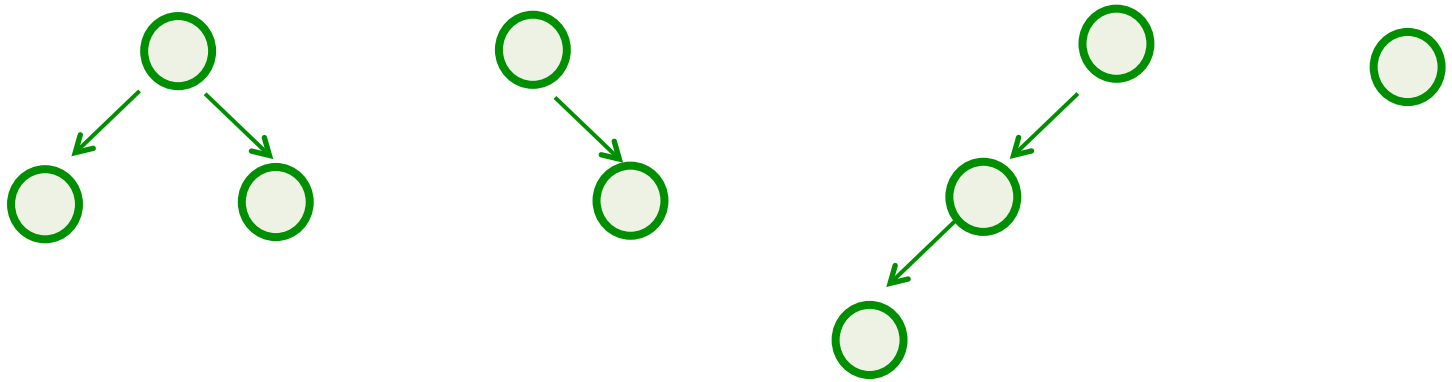
NULL



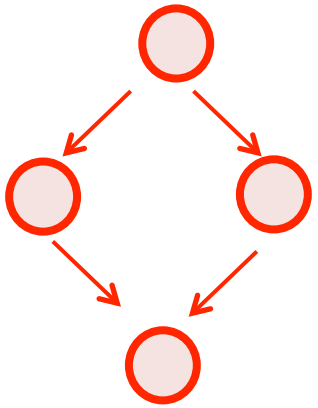


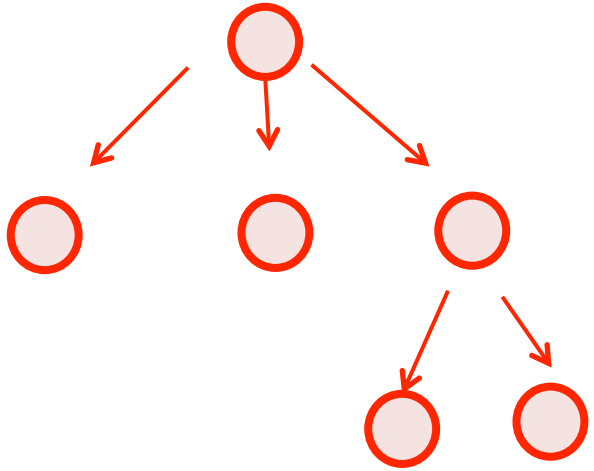
How many are valid binary trees?

A) 3 B) 4 C) 5 D) 6 E) 7



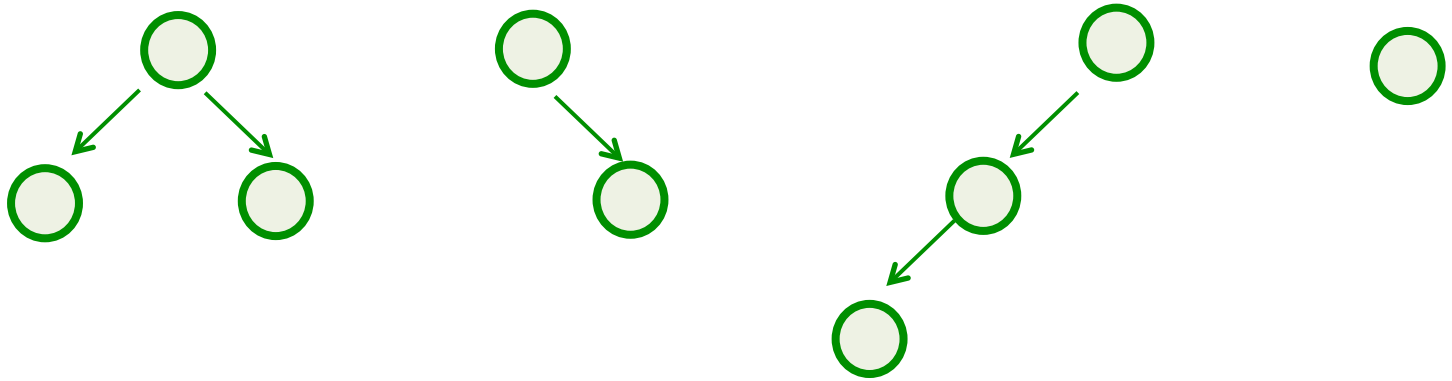
NULL



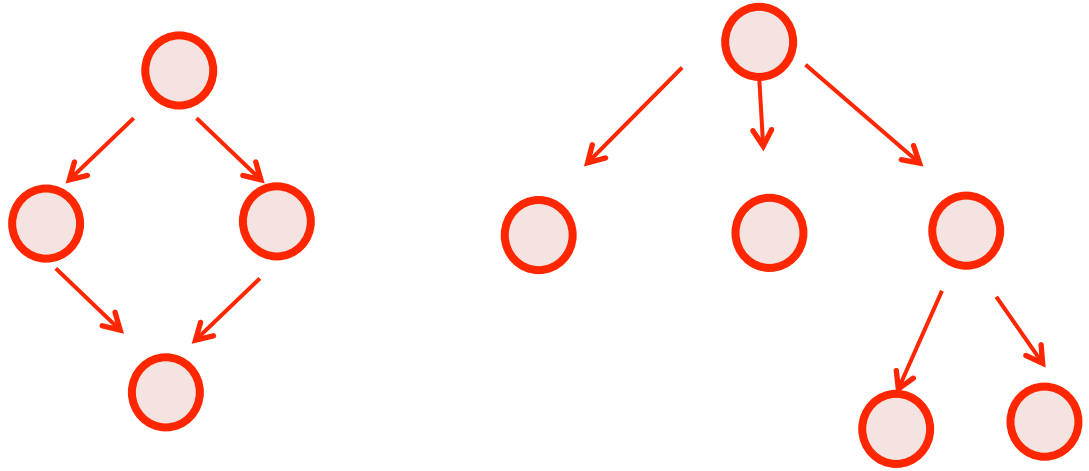


How many are valid binary trees?

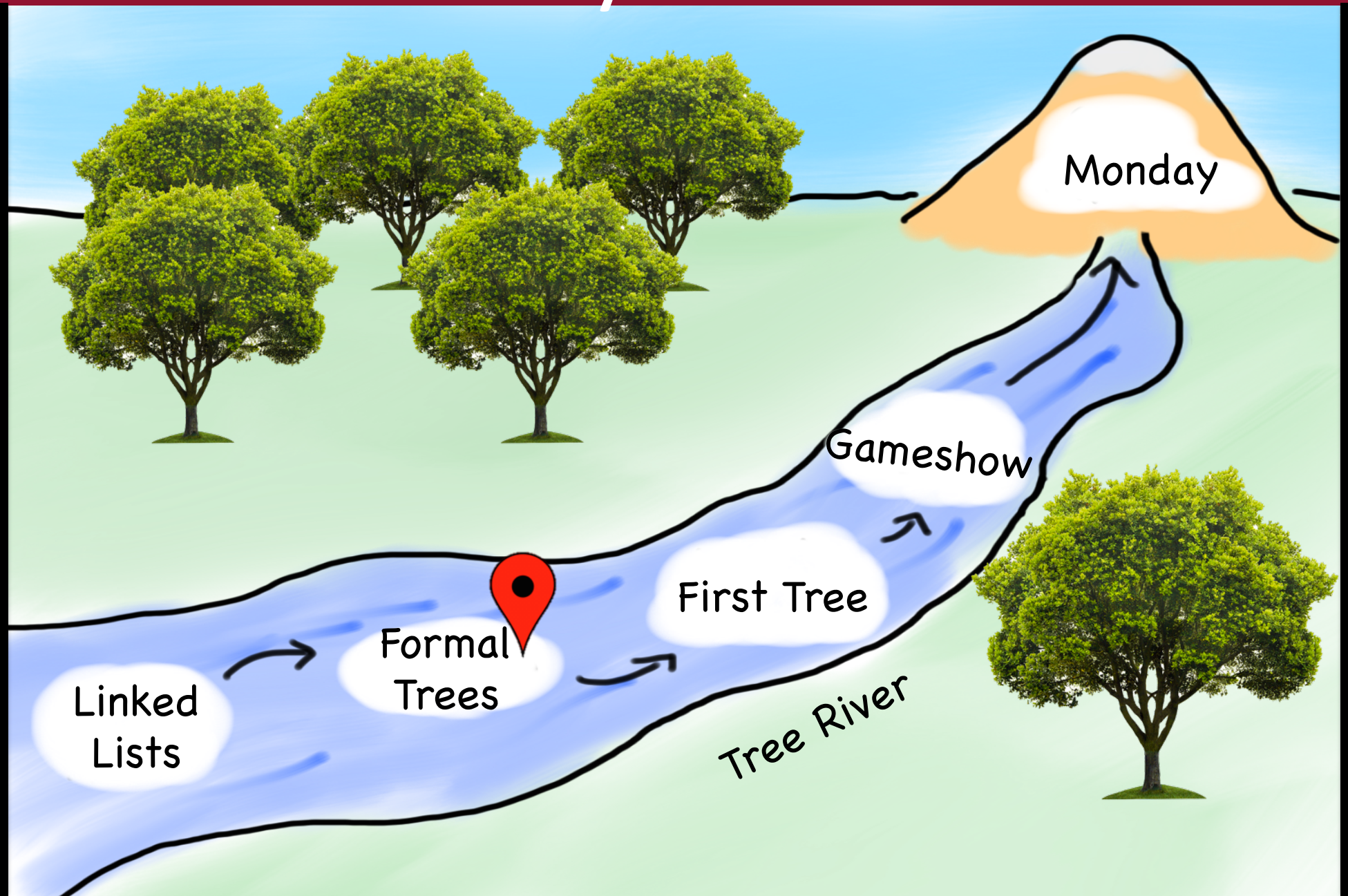
A) 3 B) 4 C) 5 D) 6 E) 7



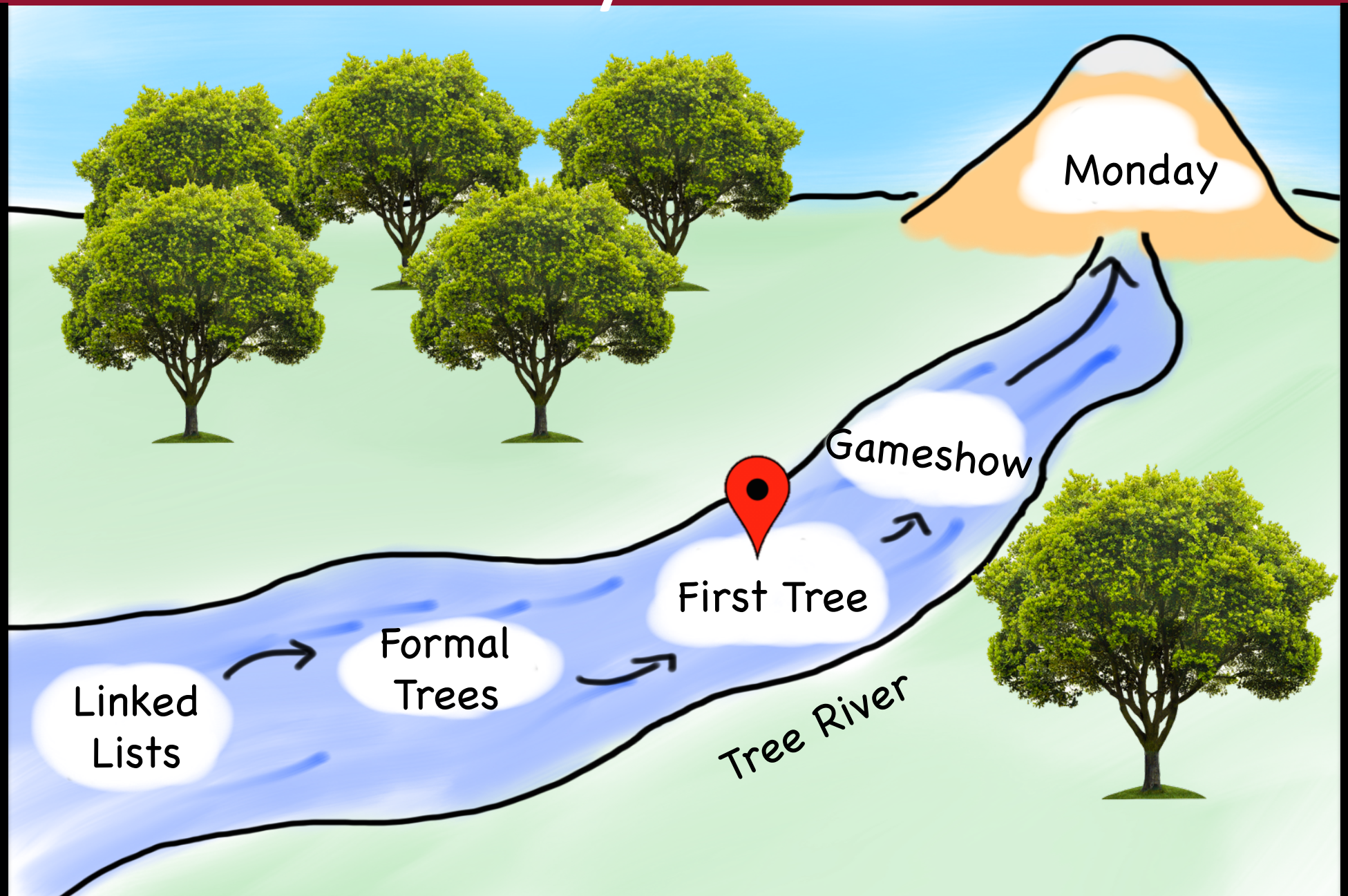
NULL



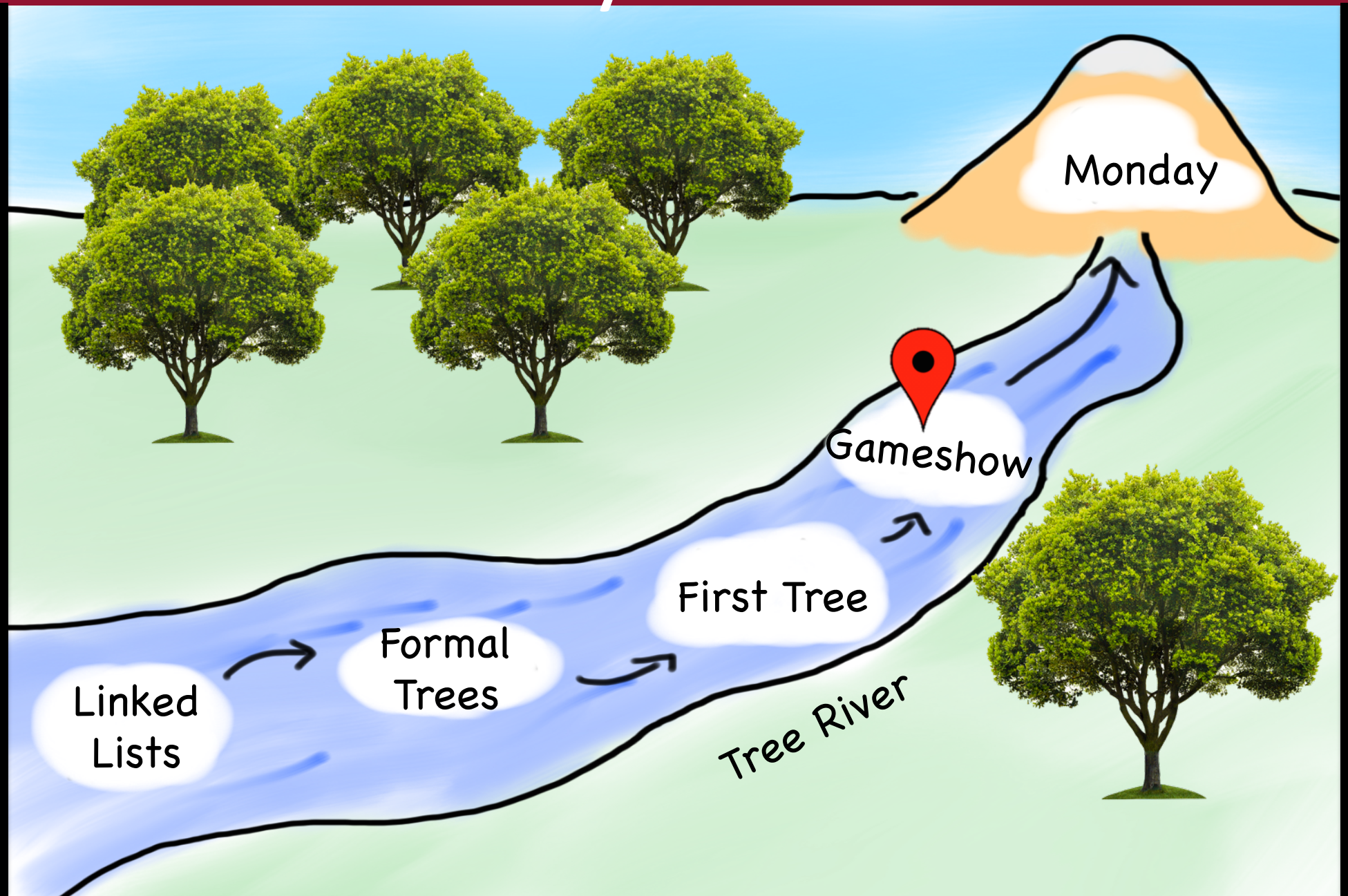
Today's Route



Today's Route



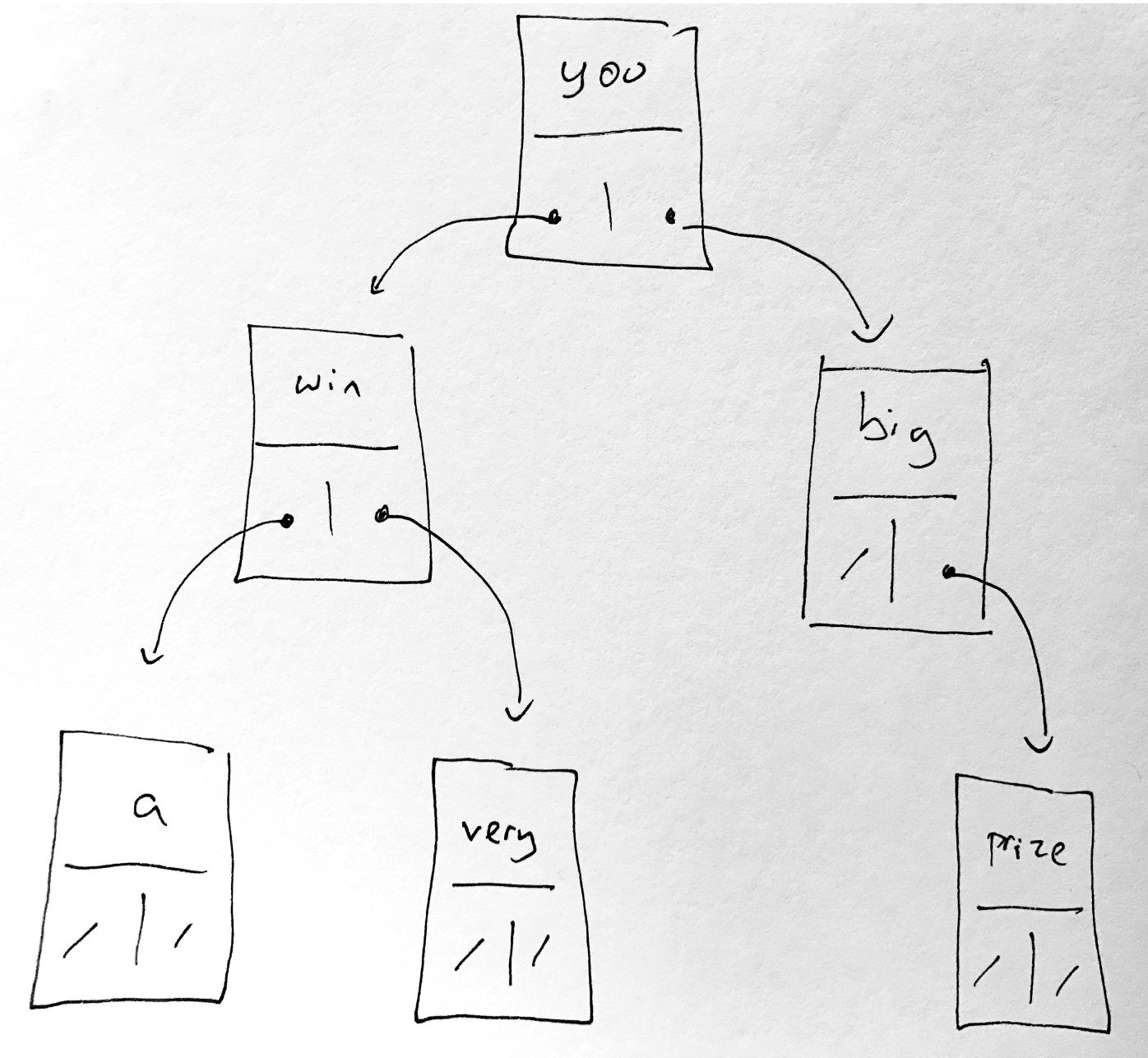
Today's Route




I VOLUNTEER
AS TRIBUTE
memegenerator.net

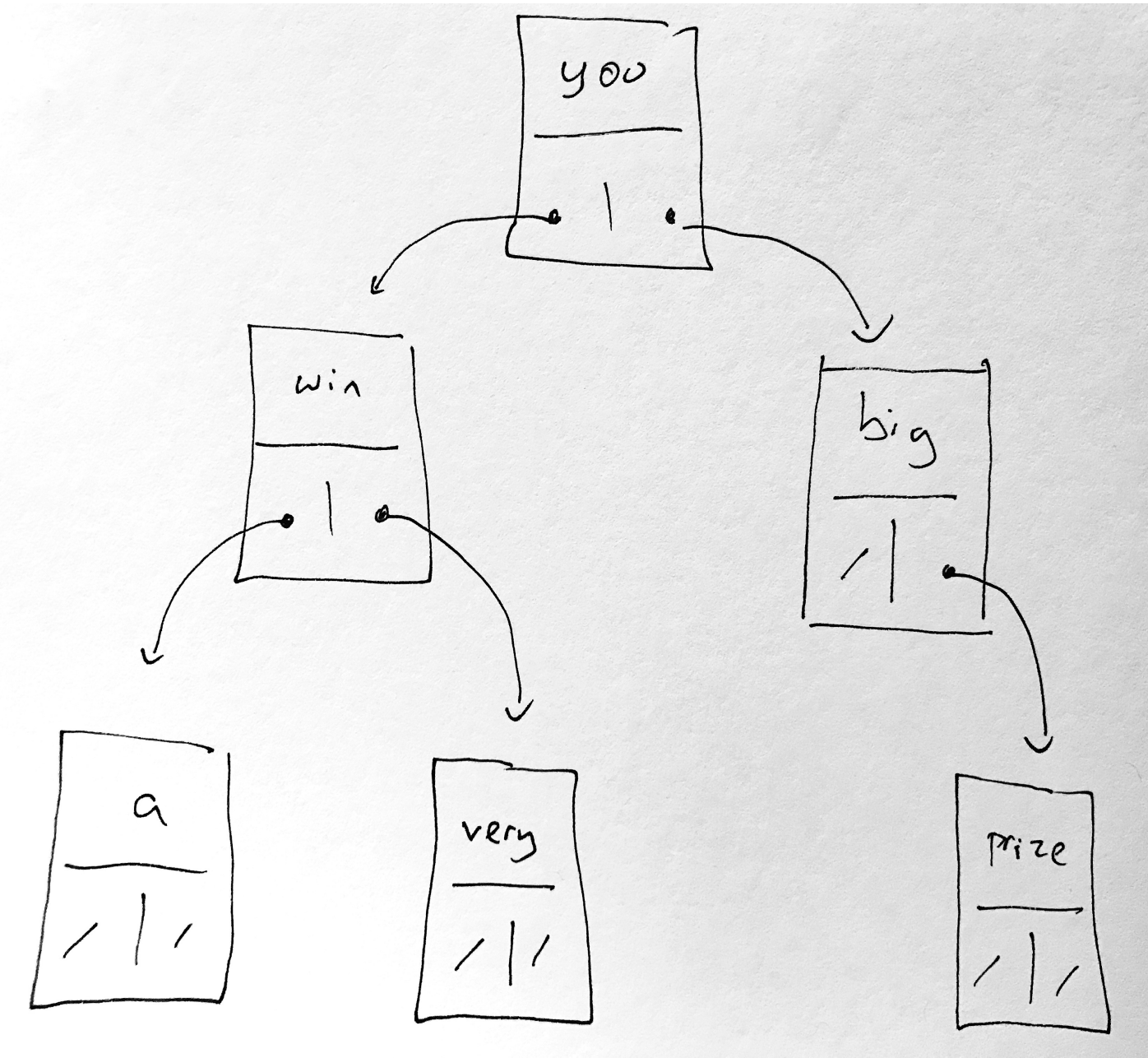
Game Show Tree

```
struct Tree {
    string value;
    Tree * left;
    Tree * right;
};
```



Game Show Tree

```
int main() {
    introduction();
    Tree * tree = initTree();
    int choice = getUserChoice();
    suspense();
    switch (choice) {
       case 1: doorOne(tree); break;
       case 2: doorTwo(tree); break;
      case 3: doorThree(tree); break;
    }
    return 0;
}
```



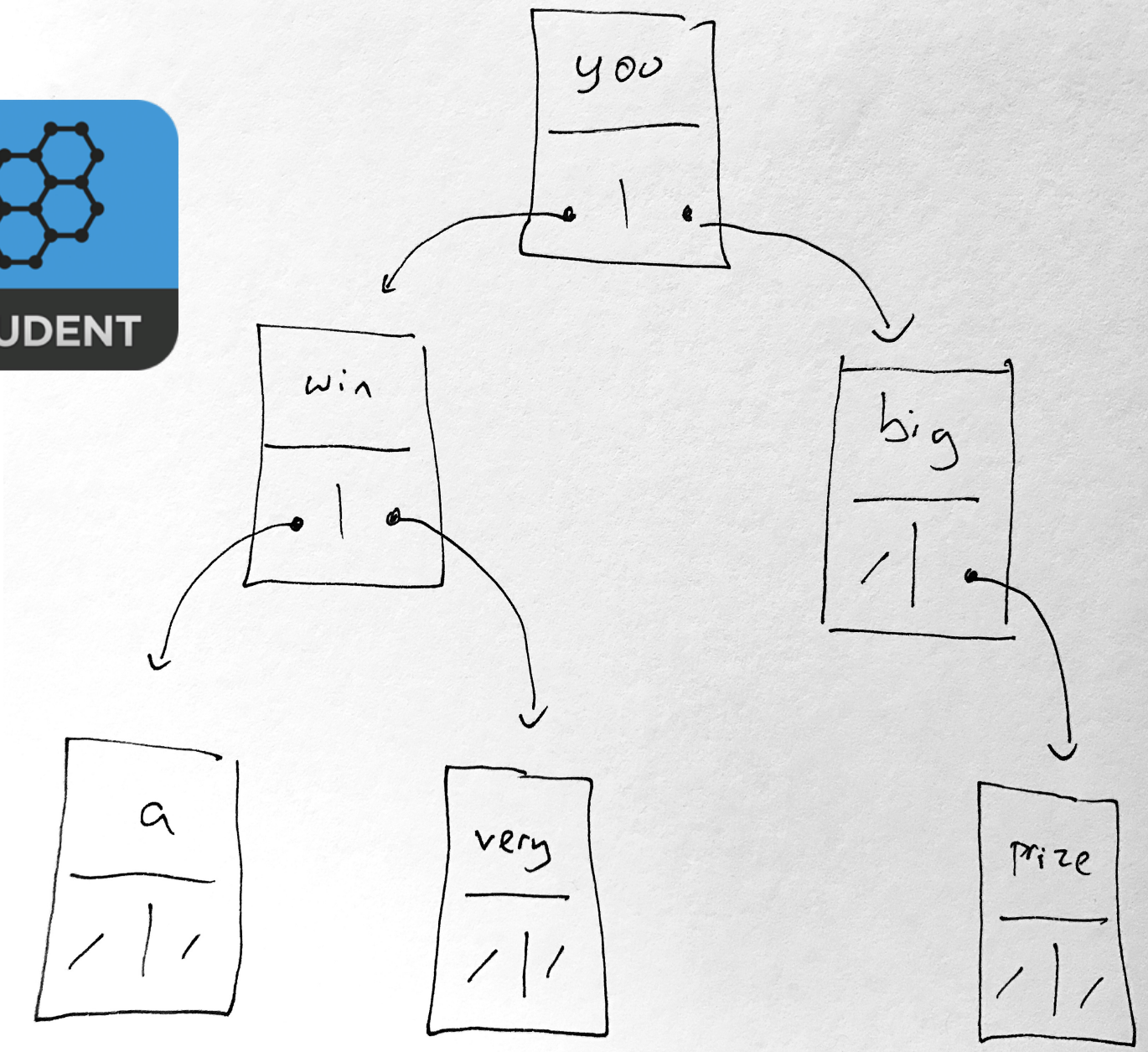
Game Show Tree

A
```
void doorOne(Tree * tree) {
    if(tree == NULL) return;
    cout<<tree->value<<" ";
    doorOne(tree->left);
    doorOne(tree->right);
}
```

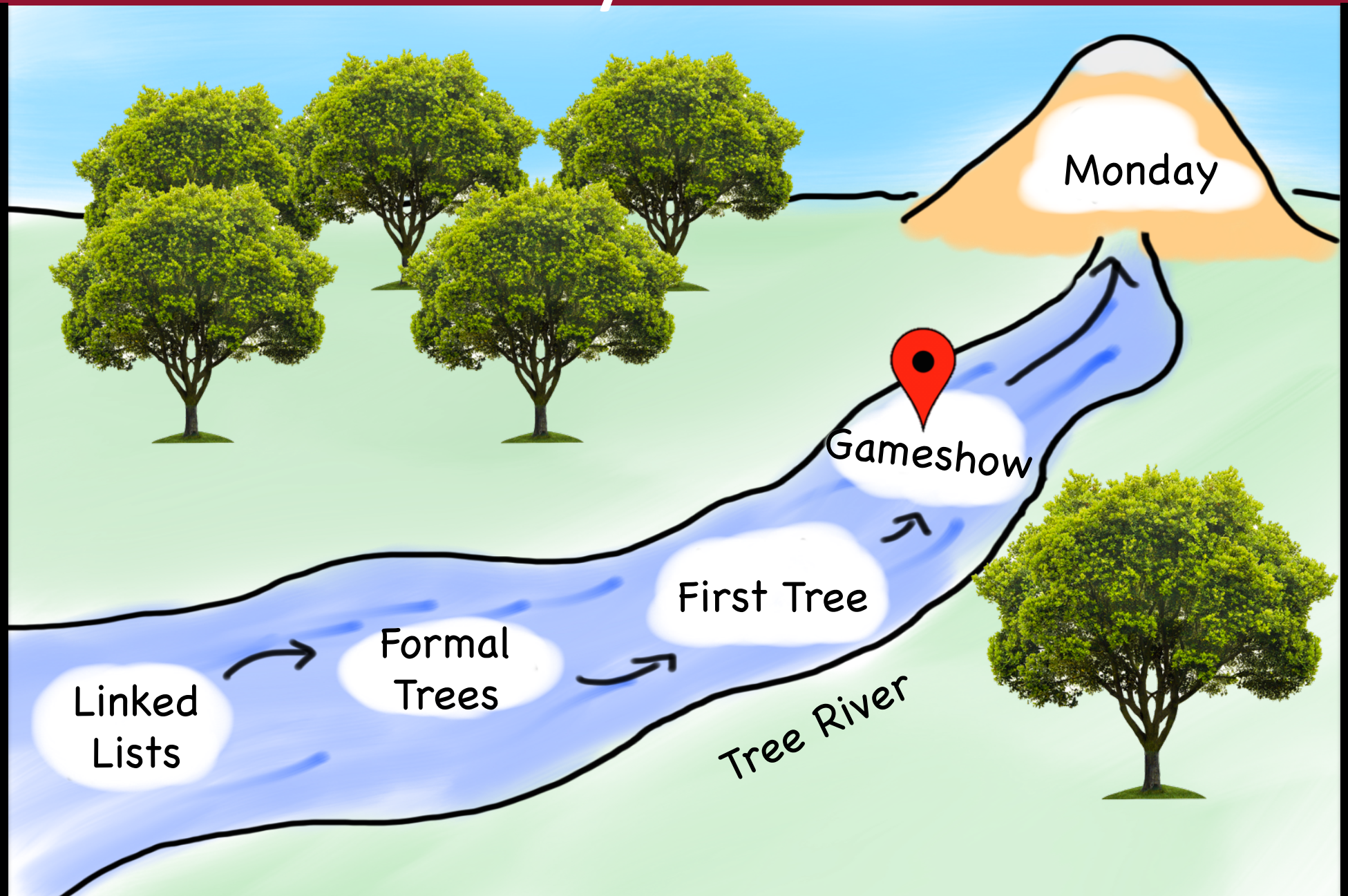
B
```
void doorTwo(Tree * tree) {
    if(tree == NULL) return;
    doorTwo(tree->left);
    cout<<tree->value<<" ";
    doorTwo(tree->right);
}
```

C
```
Void doorThree(Tree * tree) {
    if(tree == NULL) return;
    doorThree(tree->left);
    doorThree(tree->right);
    cout<<tree->value<<" ";
}
```

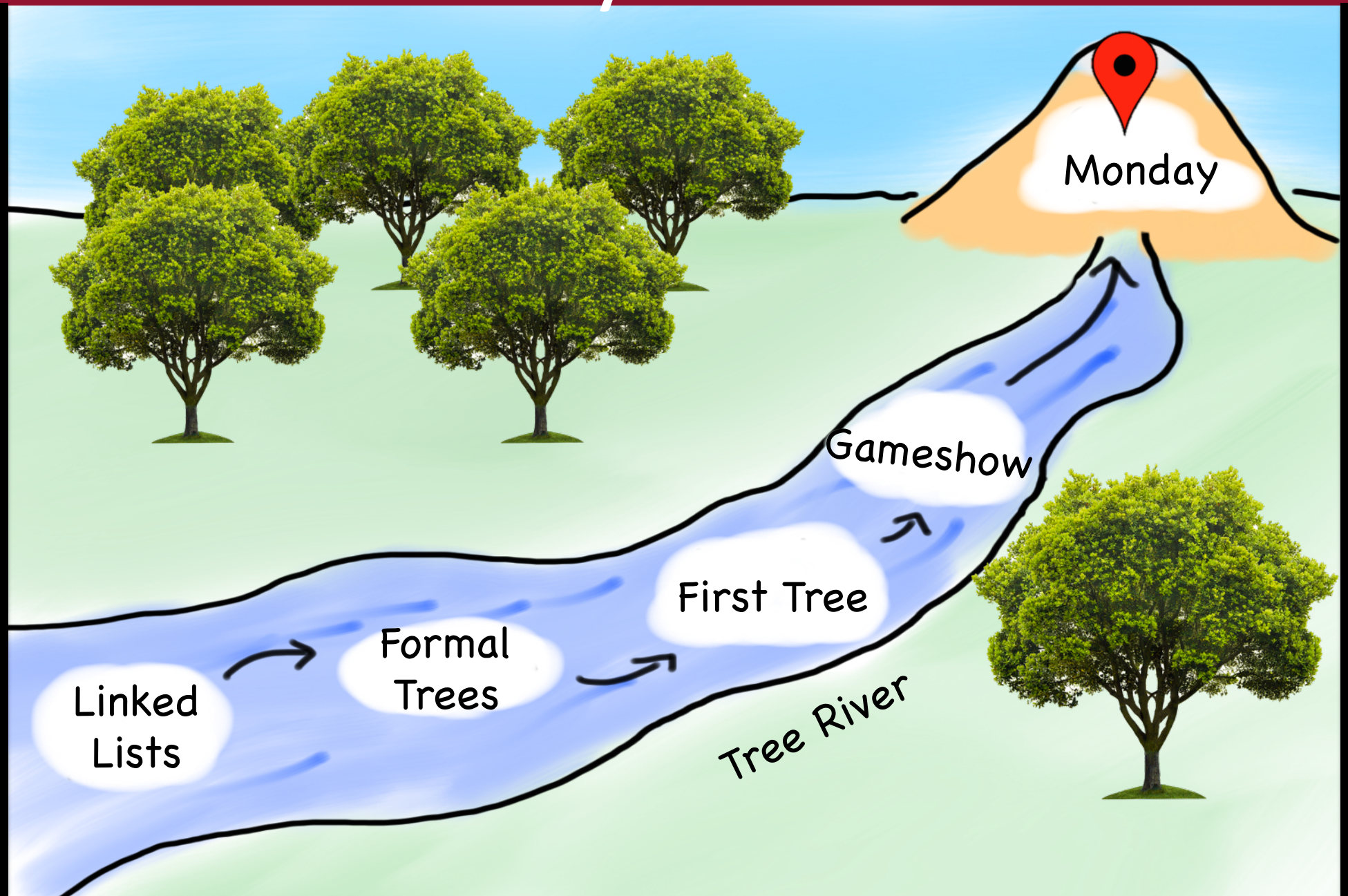
STUDENT



Today's Route



Today's Route



Today's Goals

1. Practice Linked Lists
2. Learn Trees