Traversing binary trees with exactly zero stacks

CMSC 420

The notion of a dictionary

- Recall that Part 1 is “Dictionaries”.
- Efficient search for a value V associated with a key K, when we have K.
- Arrays and lists are clearly not good enough.
  - Algorithms: $\mathcal{O}(n)$ search…
  - Hardware: arrays might provide constant access, but inflexible…

Binary trees as dictionaries

- Binary Trees are also not good enough ☹
  - $\mathcal{O}(\ell o g_2 n)$ search / insert on average, but **worst-cases are bad.**
  - Linked representation **wastes space** in the form of null pointers.
  - Recursion revisits many nodes and the system stack blows up easily.
  - **Balanced binary trees** will be the first kinds of dictionaries we talk about, since they're actually used somewhere.
- But before we balance them, let's push binary trees to their limits!

Let’s write some code… on paper!

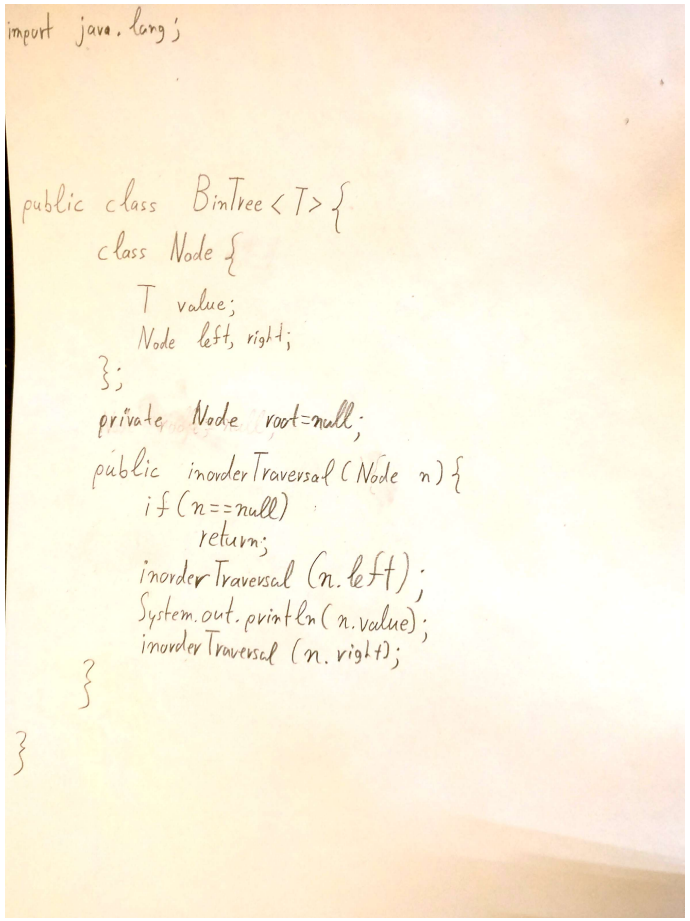
- Write down a recursive implementation of inorder traversal in Java. You have a max of 5 minutes.
- Be as detailed as you can. Do many iterations if necessary!
- Ask me for paper if you need it.
  - Berate me if I you need it and I forgot it.

Grading Key

- You will grade your partner's submission using the following key:

Criterion	Example questions to ask	Grading Percentage
Algorithmic Correctness	• Is the recursion applied correctly? • Does the code print out values when it should?	45%
Runtime error-free	• Is the code prone to a **`NullPointerException`**? • What about a **`StackOverflowException`**?	25%
Elegance	• Is the code readable? • Any arguments that we could avoid? • Generally; could this have been attained with 2/3 rds or ½ of the code?	20%
Compile error-free	• If I typed this in as – is and ran **`javac`** on it, would it compile?	10%

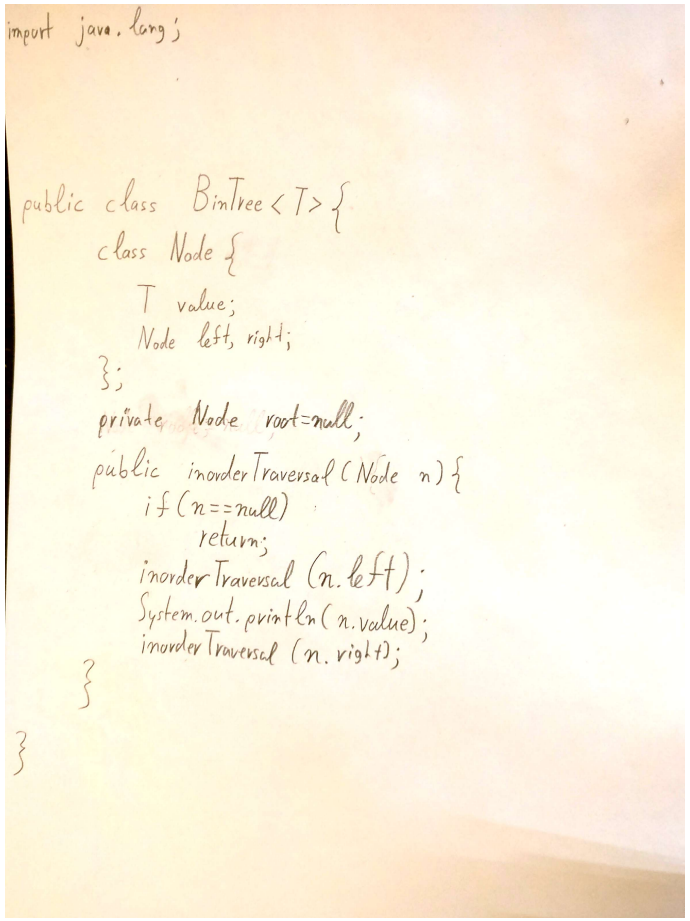
Jason's Solution



```
import java.lang;


public class BinTree<T> {
    class Node {
        T value;
        Node left, right;
    };
    private Node root=null;
    public inorderTraversal(Node n) {
        if(n==null)
            return;
        inorderTraversal(n.left);
        System.out.println(n.value);
        inorderTraversal(n.right);
    }
}
```

Jason's Solution

Grade? Be honest!

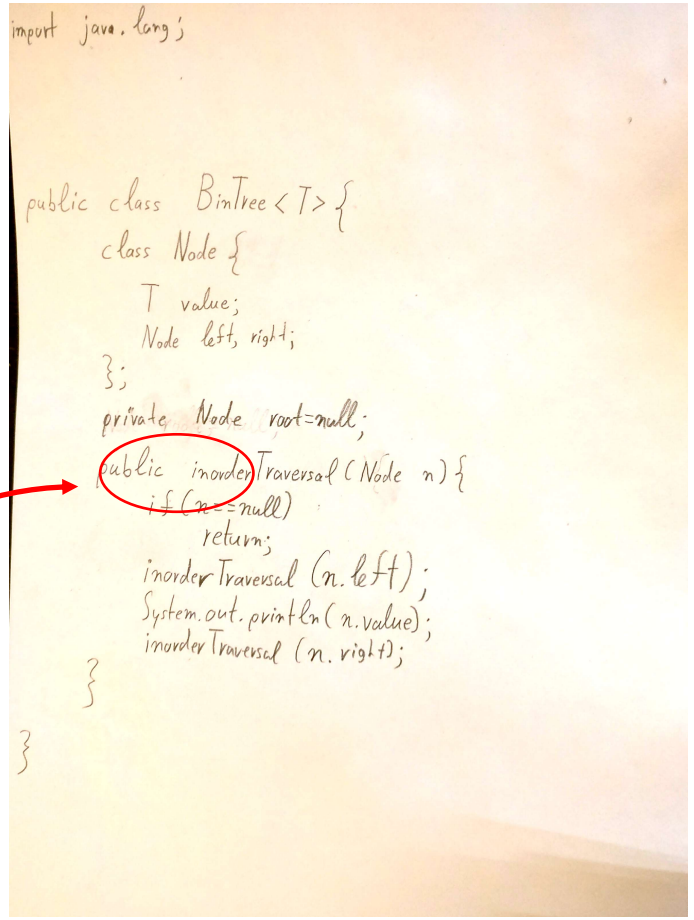


```
import java.lang;


public class BinTree<T> {
    class Node {
        T value;
        Node left, right;
    };
    private Node root=null;
    public inorderTraversal(Node n) {
        if(n==null)
            return;
        inorderTraversal(n.left);
        System.out.println(n.value);
        inorderTraversal(n.right);
    }
}
```

Jason's Solution

Grade? Be honest!

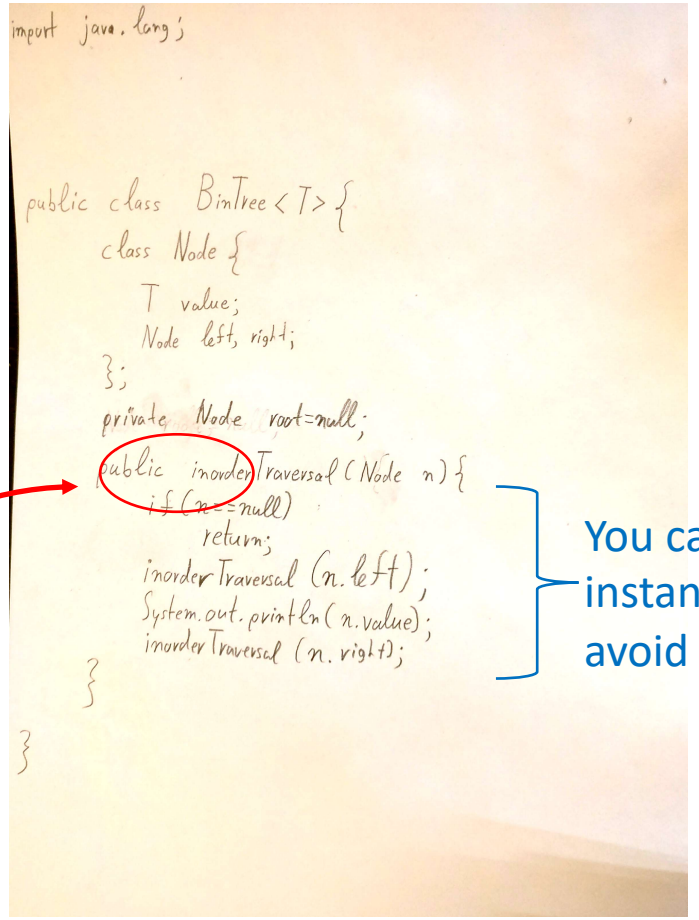
Where's the return type?
This won't compile…
(at least -5%)



```
import java.lang;

public class BinTree<T>{
    class Node{
        T value;
        Node left, right;
    };
    private Node root=null;
    public inorderTraversal(Node n){
        if(n==null)
            return;
        inorderTraversal(n.left);
        System.out.println(n.value);
        inorderTraversal(n.right);
    }
}
```

Jason's Solution

Grade? Be honest!



```
import java.lang;


public class BinTree<T>{
    class Node{
        T value;
        Node left, right;
    };
    private Node root=null;
    public inorderTraversal(Node n){
        if(n==null)
            return;
        inorderTraversal(n.left);
        System.out.println(n.value);
        inorderTraversal(n.right);
    }
}
```

Where's the return type?
This won't compile…

You can also make `inorderTraversal` an instance method of `BinTree.Node`, in order to avoid the `Node` argument

A harder question

- Now, instead of using the system's stack, let's use our own!
- In no more than 5 minutes, write Java code that uses your own Stack to do inorder traversal in a binary tree!
- Then, grade it as before!

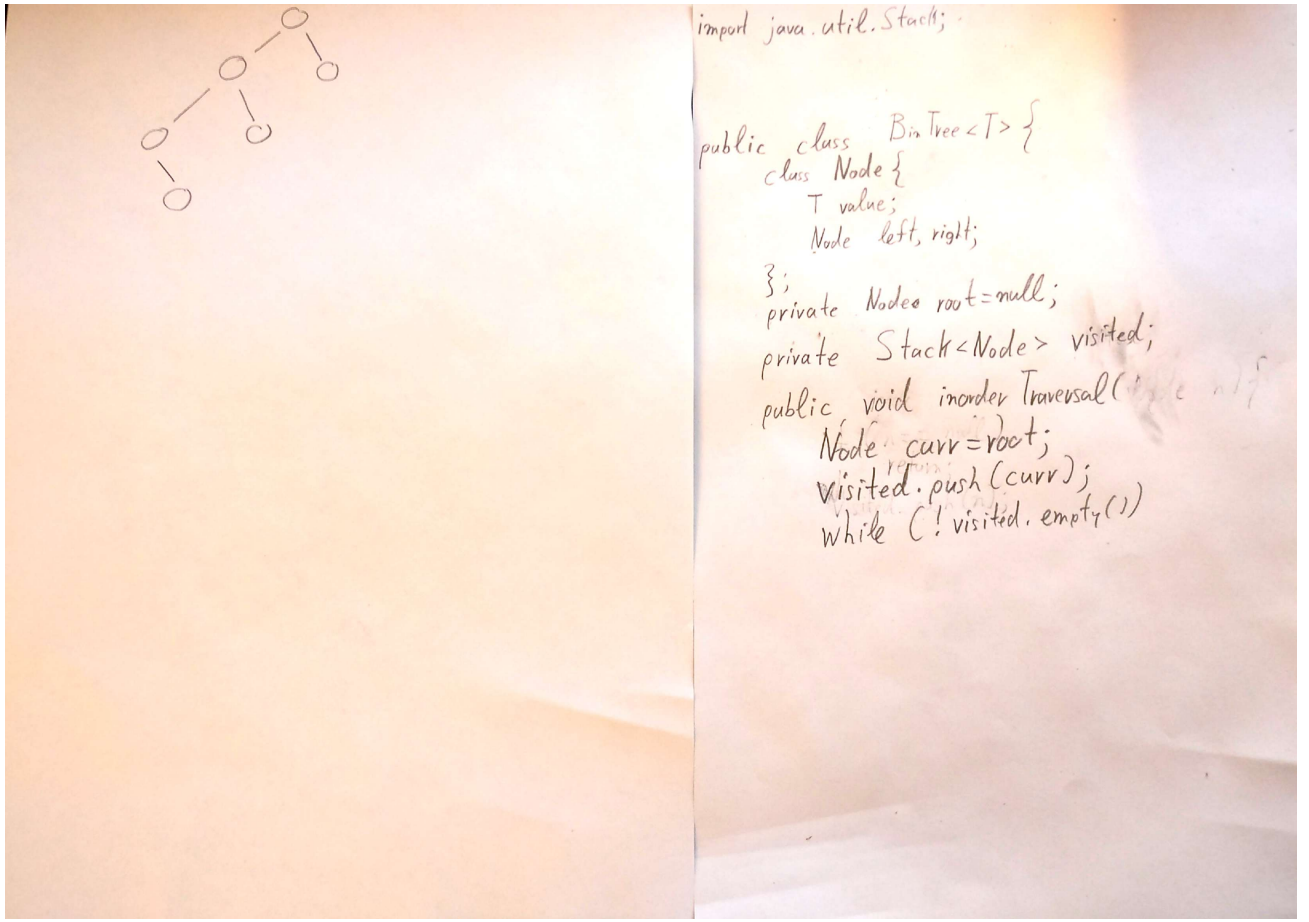
A harder question

- Now, instead of using the system's stack, let's use our own!
- In no more than 5 minutes, write Java code that uses your own Stack to do inorder traversal in a binary tree!
- Then, grade it as before!

Criterion	Example questions to ask	Grading Percentage
Algorithmic Correctness	• Is the recursion applied correctly? • Does the code print out values when it should?	45%
Runtime error-free	• Is the code prone to a NullPointerException? • What about a StackOverflowException?	25%
Elegance	• Is the code readable? • Any arguments that we could avoid? • Generally; could this have been attained with 2/3 rds or ½ of the code?	20%
Compile error-free	• If I typed this in as – is and ran javac on it, would it compile?	10%

Jason's Solution

HAHAHAHA
HAHAHA
HA
HAHAHA
HAHAHAHA



```
import java.util.Stack;

public class BinTree<T> {
    class Node {
        T value;
        Node left, right;
    };
    private Node root=null;
    private Stack<Node> visited;
    public void inorderTraversal(
        Node curr=root;
        visited.push(curr);
        while (!visited.empty())
```

A solution

```
public inorderTraversalWithStack(){
        Stack<Node> s = new Stack<Node>(); // Important that we make a stack of Nodes, note of values
        Node p = root;
        while(!s.empty() || p != null){    // What's the utility of the 2nd constraint?
                if(p == null){             // This means that we were reached by some ancestral node.
                        p = s.pop();       // Visit said ancestral node,
                        print(p.value);
                        p = p.right;       // and see if there's anything on its right subtree.
                } else {                   // So we are actually a living, breathing child of our parent.
                        s.push(p);         // We should record our presence
                        p = p.left;        // And see if there's anything on the left to investigate
                }
        }
}
```

Vote!

- We just compared two different ways to do inorder traversal in a binary tree.
- Which one is the **fastest, and why?**

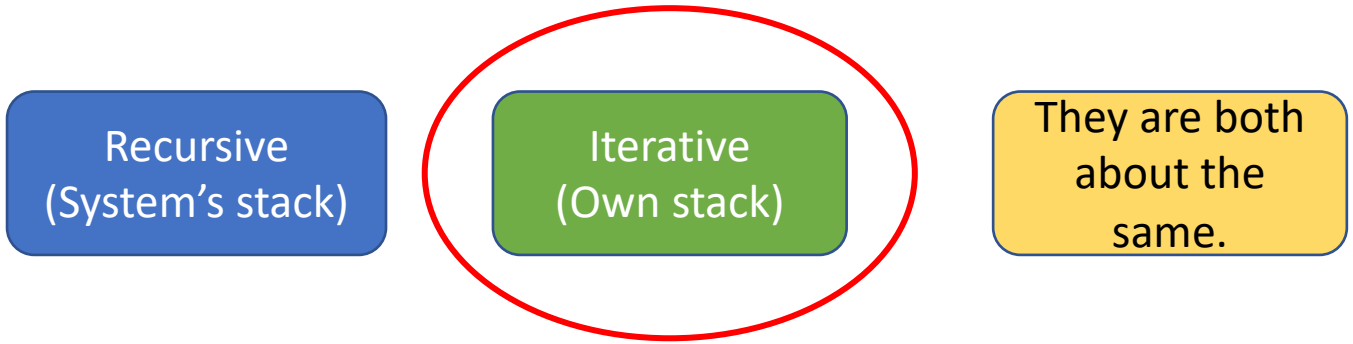
Recursive (System's stack)

Iterative (Own stack)

They are both about the same.

Vote!

- We just compared two different ways to do inorder traversal in a binary tree.
- Which one is the **fastest, and why?**



- Wayyyyy more than just a reference is pushed onto the **system's** stack…
  - Program Counter (Instruction Pointer)
  - Local variables
  - …

But how bad is it really?

- It’s actually pretty bad. ☹
  - Especially in the JVM.
- Let’s have a look at a Java demo where we generate the count for a linked list in 3 different ways.
  1. In a classic iterative fashion.
  2. Using our own stack.
  3. Using recursion.

Another exercise!

- Take a max of 5 minutes to write a non-recursive Binary ***Search*** Tree insertion!

Another exercise!

- Take a max of 5 minutes to write a non-recursive Binary ***Search*** Tree insertion!
- Afterwards, swap papers and grade using our familiar rubric:

Criterion	Example questions to ask	Grading Percentage
Algorithmic Correctness	• Is the recursion applied correctly? • Does the code print out values when it should?	45%
Runtime error-free	• Is the code prone to a NullPointerException? • What about a StackOverflowException?	25%
Elegance	• Is the code readable? • Any arguments that we could avoid? • Generally; could this have been attained with 2/3 rds or ½ of the code?	20%
Compile error-free	• If I typed this in as – is and ran javac on it, would it compile?	10%

Demo time!

Take-home message #1

- When possible, we need to avoid recursion.
  - The system's stack is very limited by the OS.
- Replace with:
  1. Using your own stack (inelegant, but much faster!)
  2. Refactoring the code such that it does tail recursion, which is optimized into iteration by a modern compiler.
     - The keyword `@tailrec` in Scala ***attempts*** to do that even when you don't take care of it yourself.
  3. An iterative piece of code that achieves the same thing (sometimes inelegant, but faster **and** much better in terms of memory).
     - Recall: bottom-up mergesort avoids system stack pushes, yet iteratively solves the same problem.

Vote!

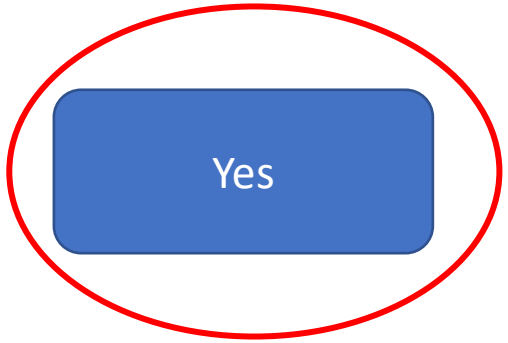
- Is it possible to traverse a tree without using a stack (ours, or the system's) ***<u>at all?</u>***

Yes

I have clearly not been paying any attention to the lecture since I came into class today, with particular (non-)emphasis to the title page of the slide deck that we are currently scanning.

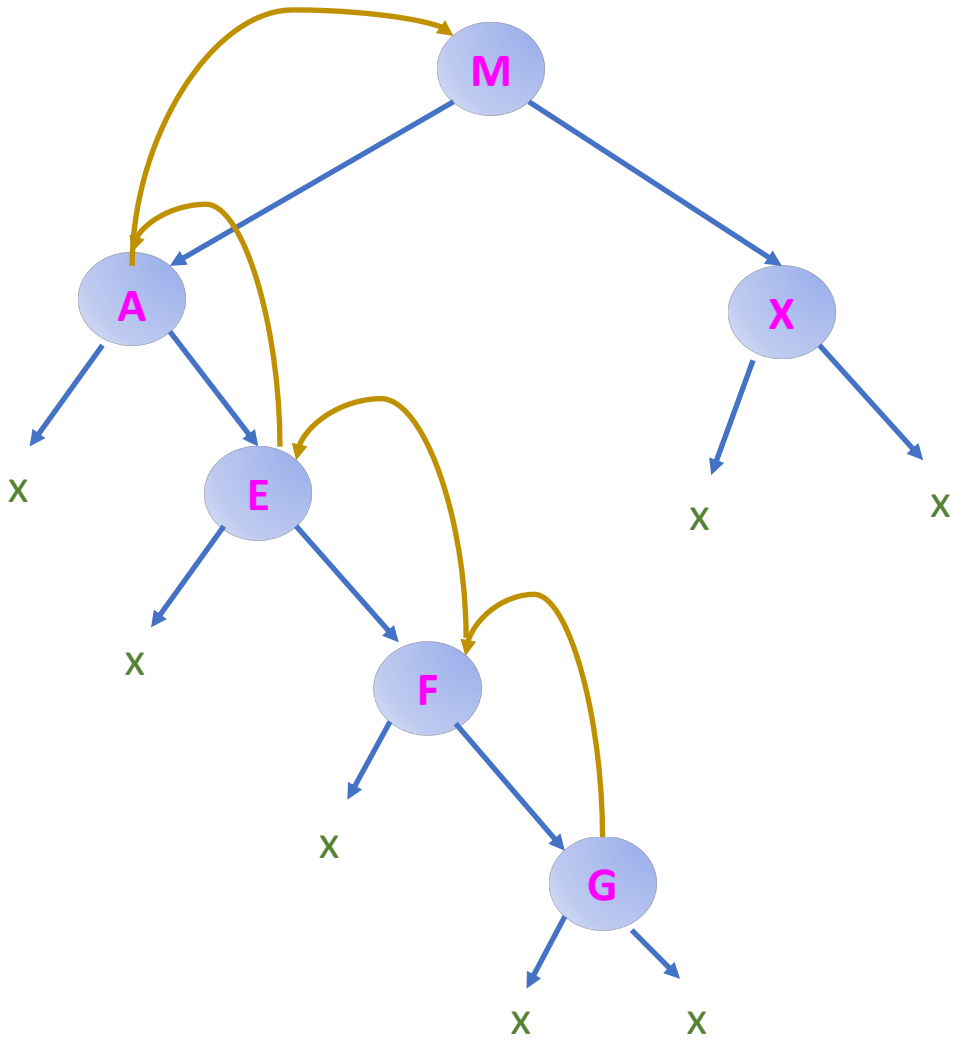
Vote!

- Is it possible to traverse a tree without using a stack (ours, or the system's) ***at all?***



I have clearly not been paying any attention to the lecture since I came into class today, with particular (non-)emphasis to the title page of the slide deck that we are currently scanning.

The hated tree



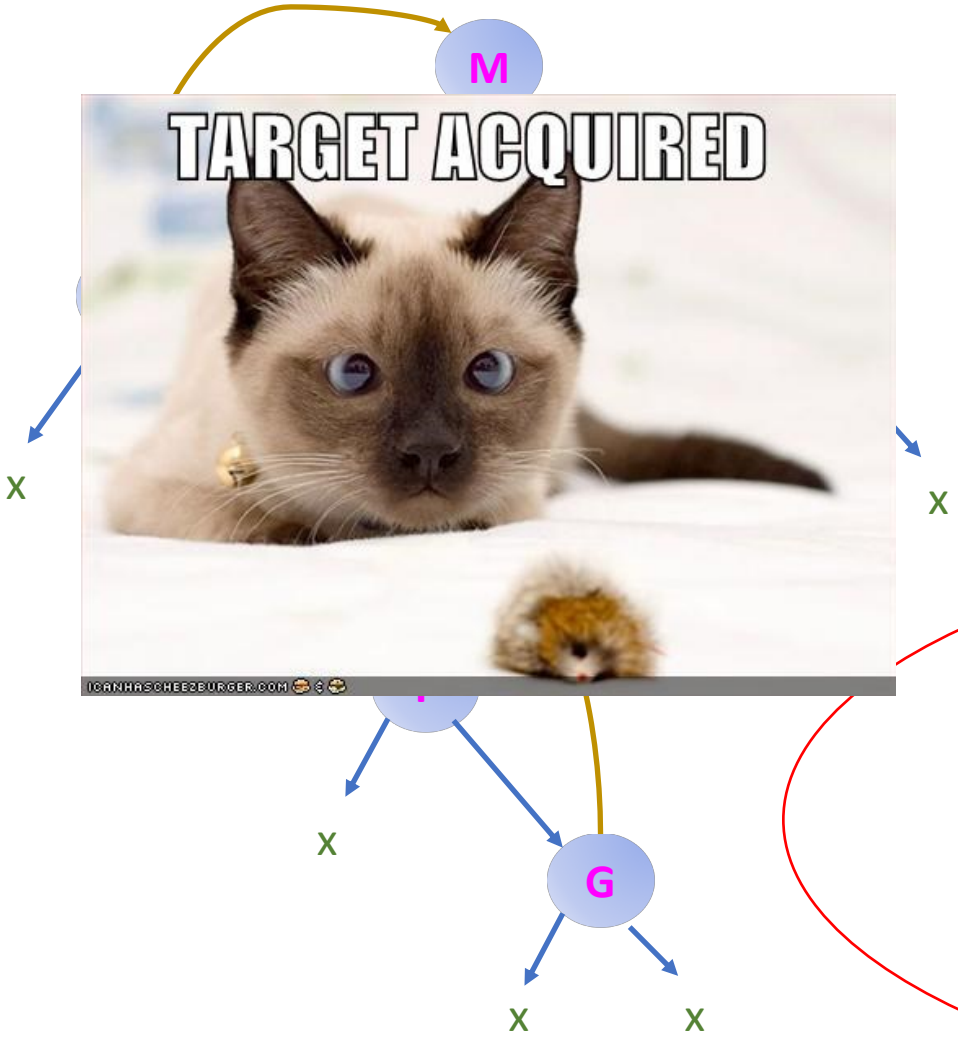
Remember the things Jason didn't like about this tree:

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search.... ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

The hated tree



Remember the things Jason didn't like about this tree:

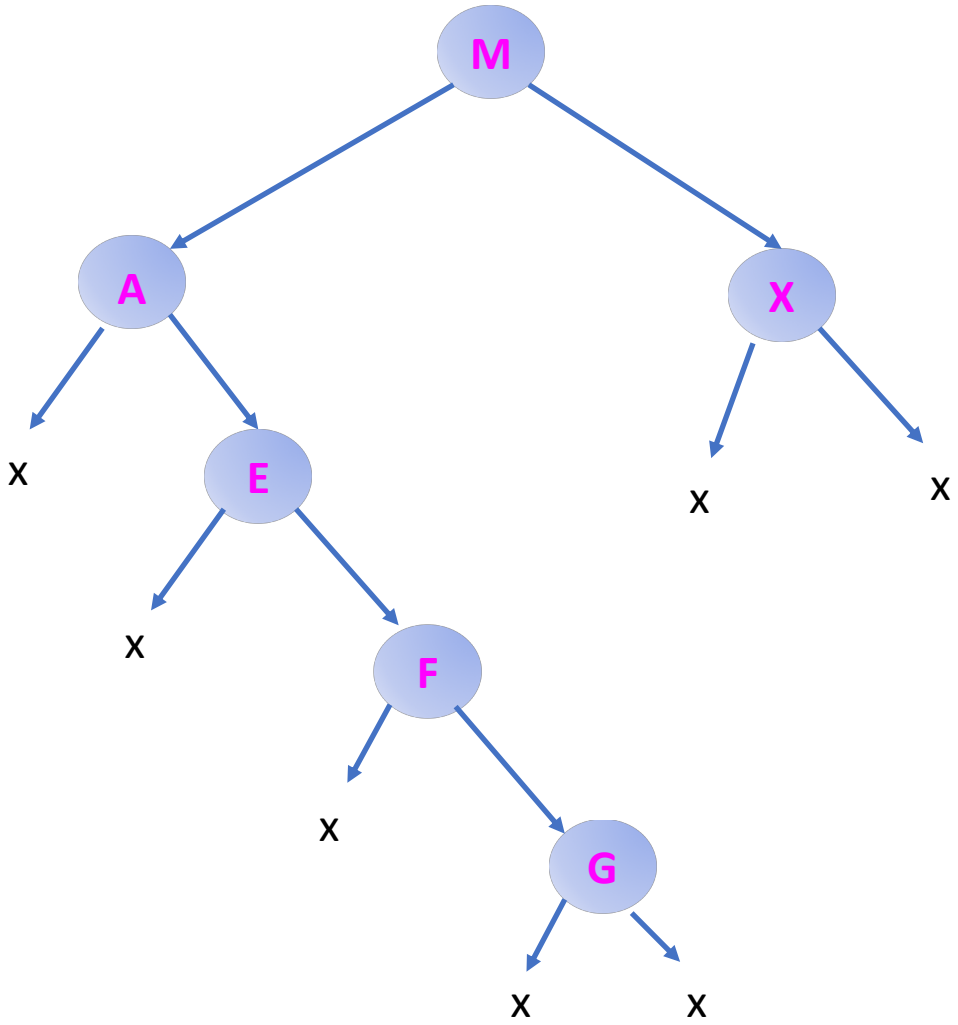
Let's look at one idea that solves both of these problems in one fell swoop!

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

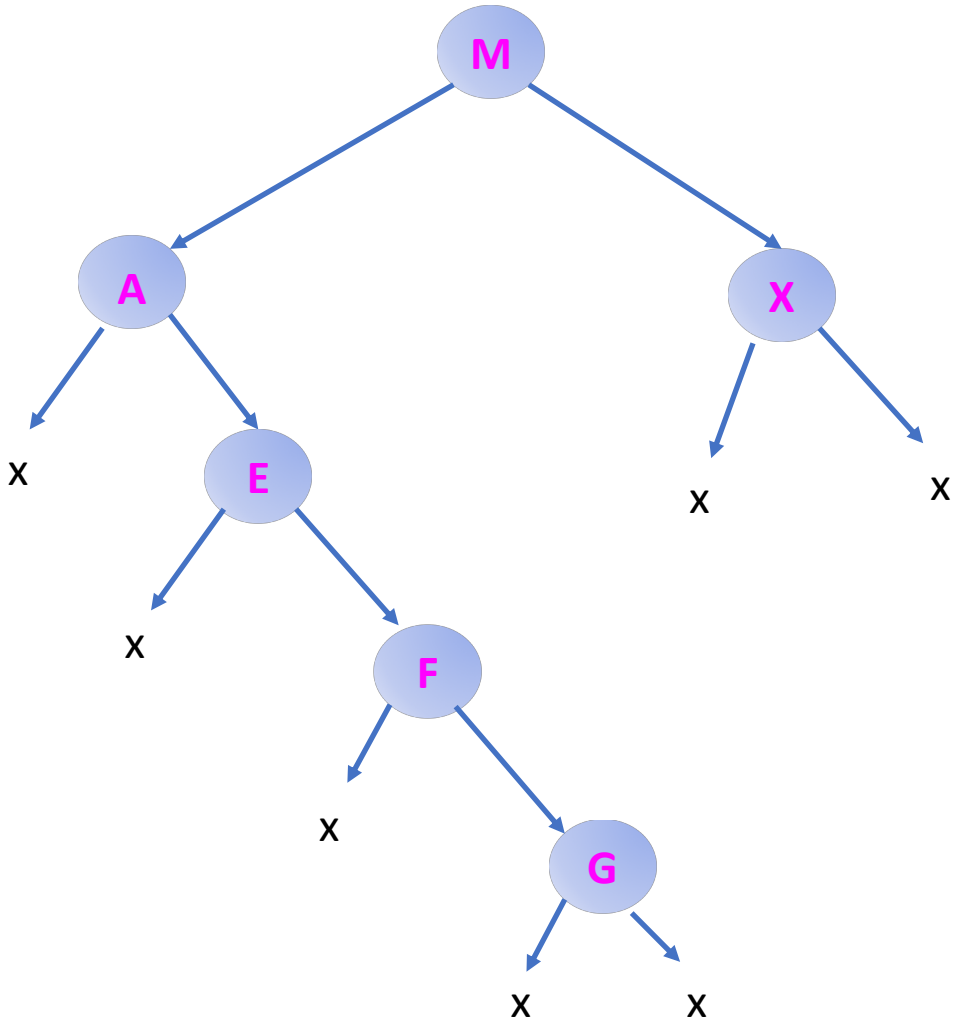
(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

Our 1st solution: Threaded Trees

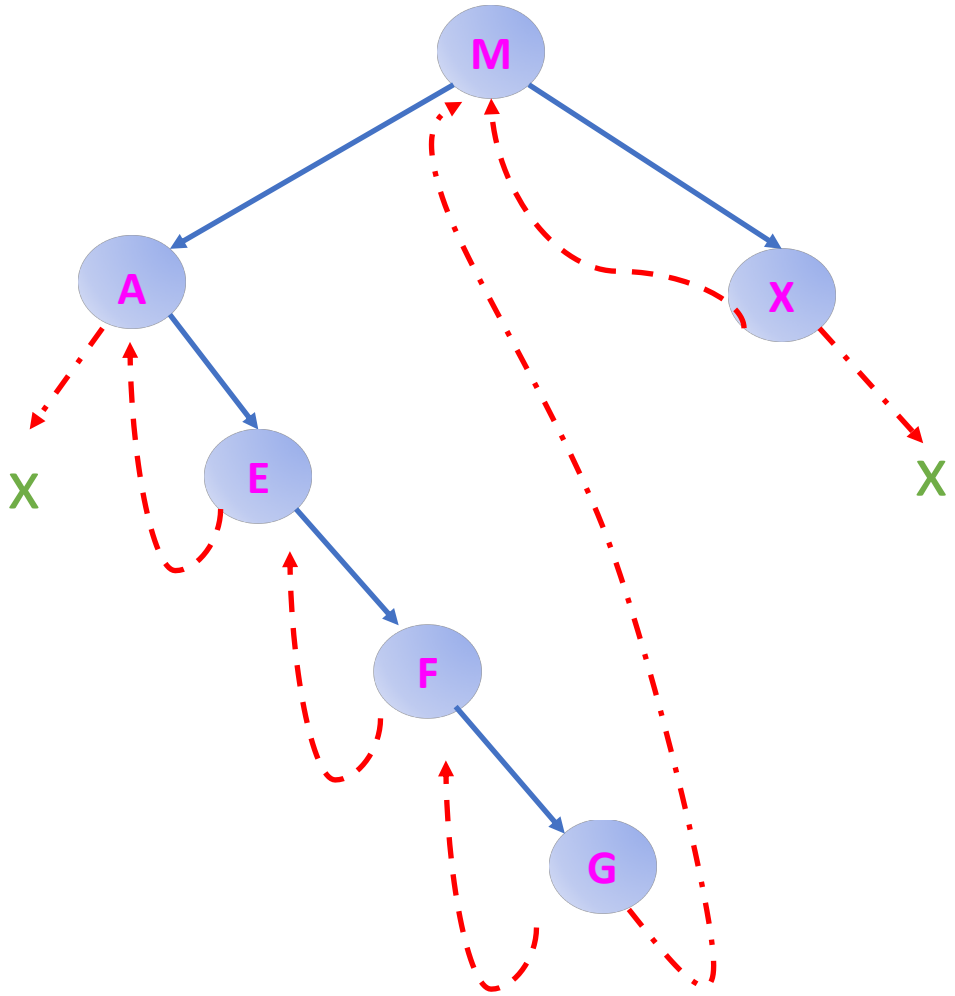


Exactly **nothing** to do with threads of an OS.

Our 1st solution: Threaded Trees

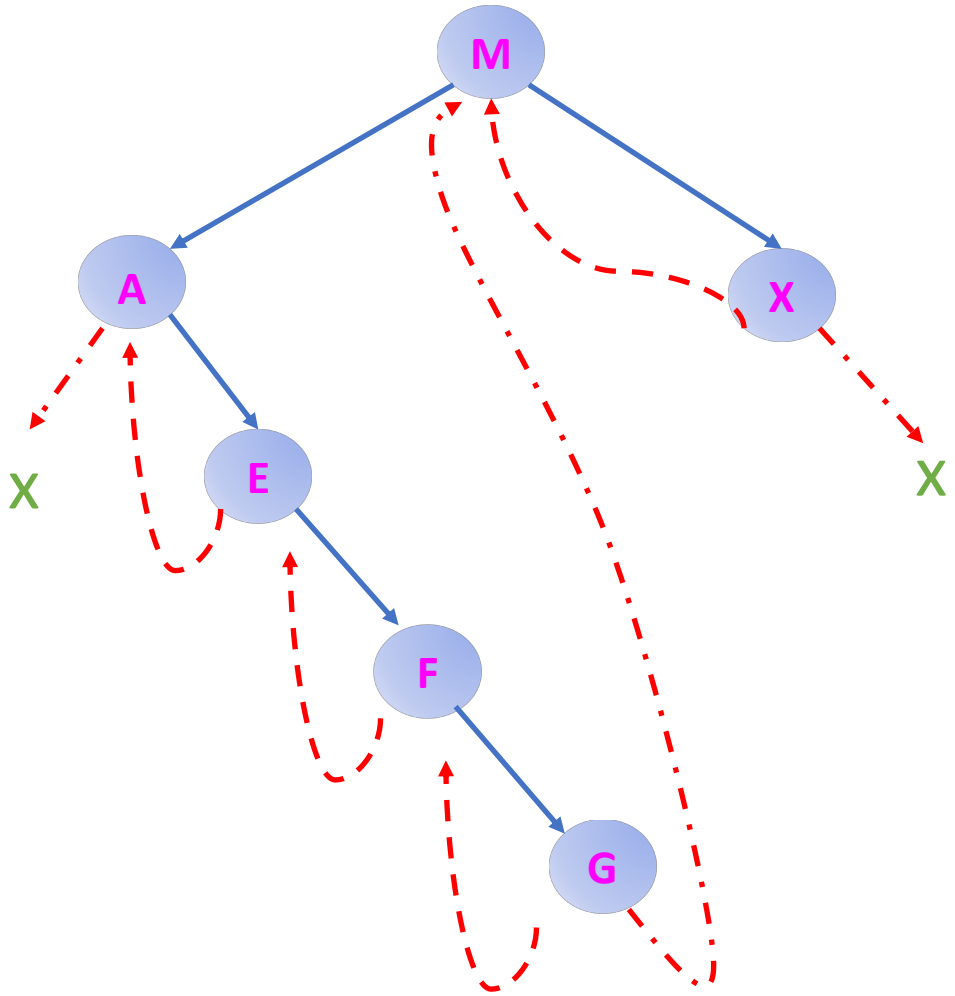


Threaded Trees



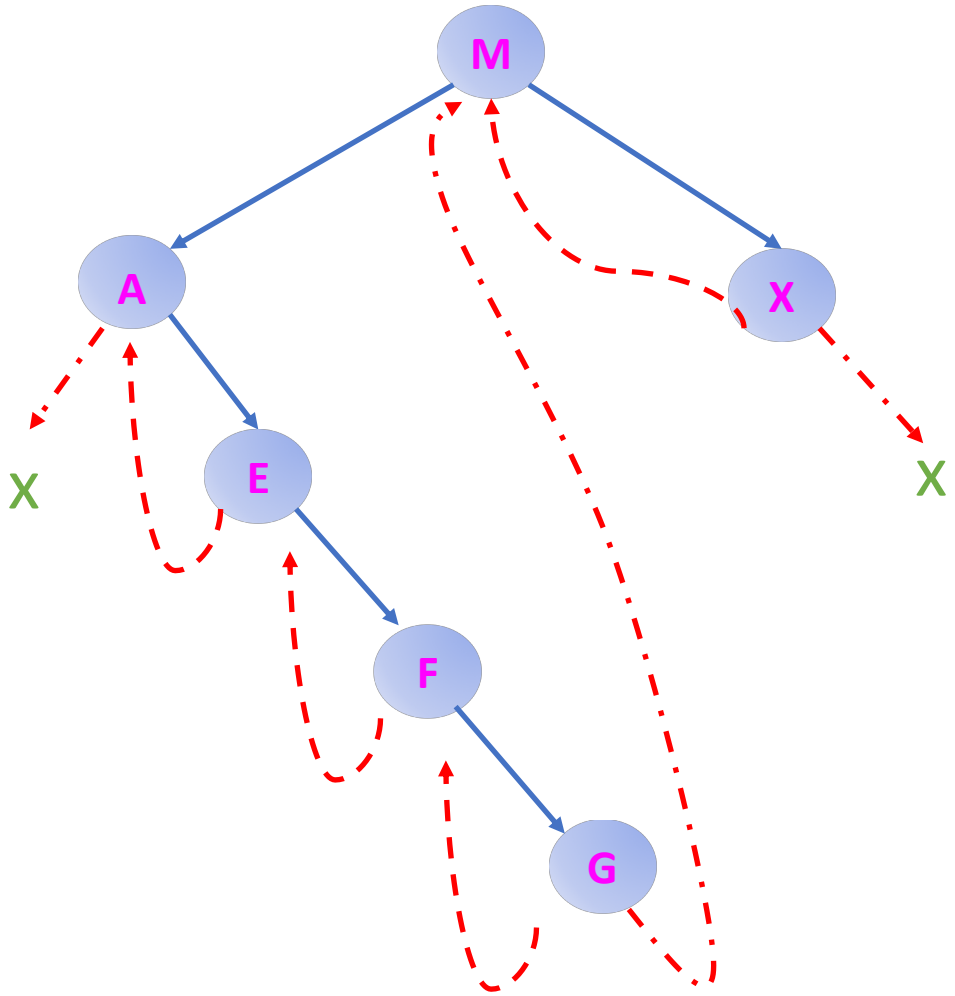
- Key: Replace **all** null pointers with **special pointers**, called **threads.**
- The right thread will point to the node's **inorder successor.**
- The left thread will point to the node's **inorder predecessor.**
  - By convention, the **side pointers** that still point to NULL are also threads.
  - A predecessor or successor being null means that we are at the beginning or end of the sequence respectively.

Threaded Trees



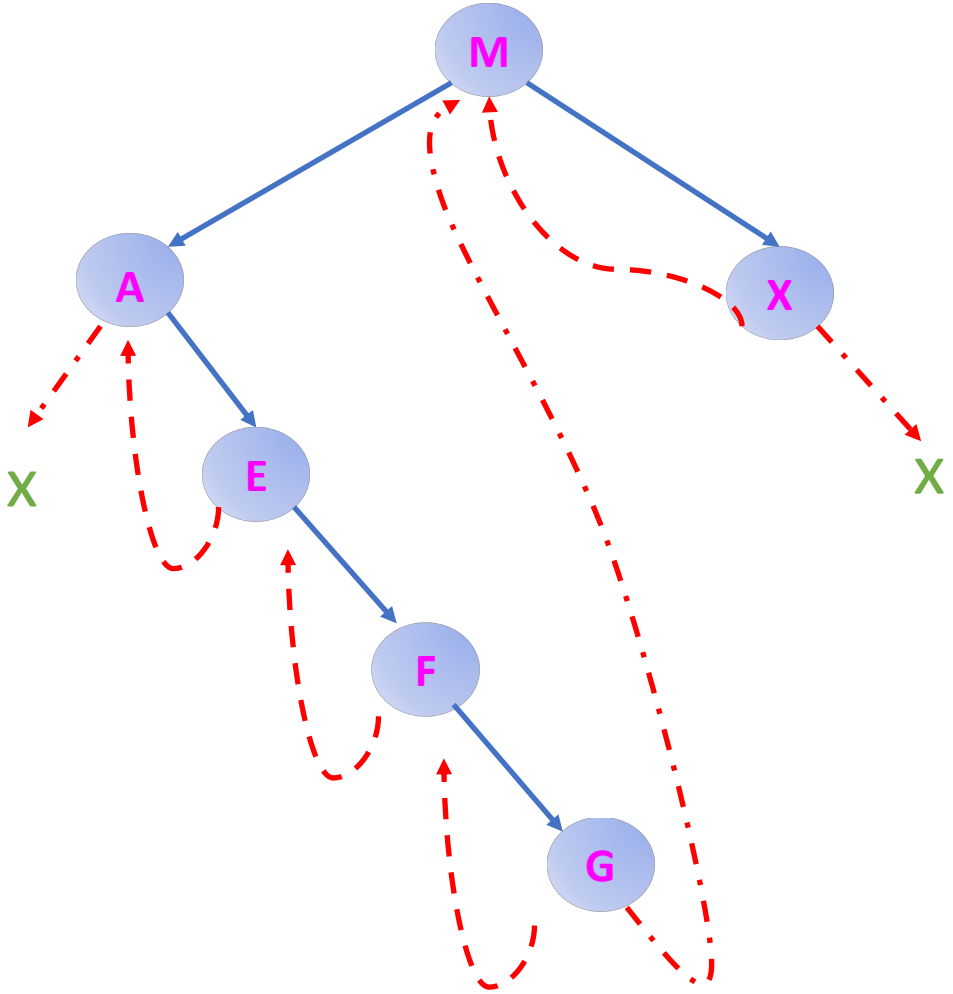
- Key: Replace **most** null pointers with **special pointers**, called **threads.**
  - Except for the "side" ones
- By traversing a thread, you reach an inorder successor in **constant time**!

Threaded Trees



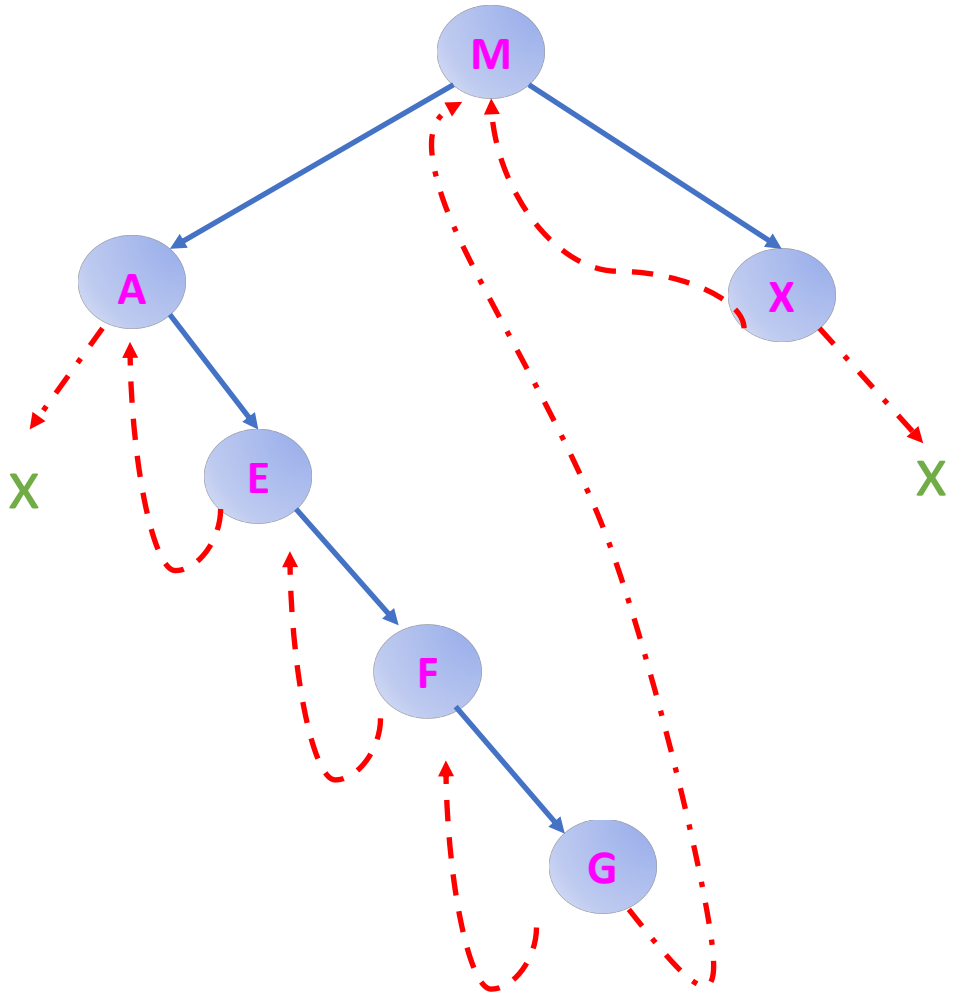
- Key: Replace **most** null pointers with **special pointers**, called **threads.**
  - Except for the "side" ones
- By traversing a thread, you reach an inorder successor in **constant time**!
- But if you are ***not*** traversing a thread, but a classic pointer, you pay linear cost.

So, how do we do inorder traversal?



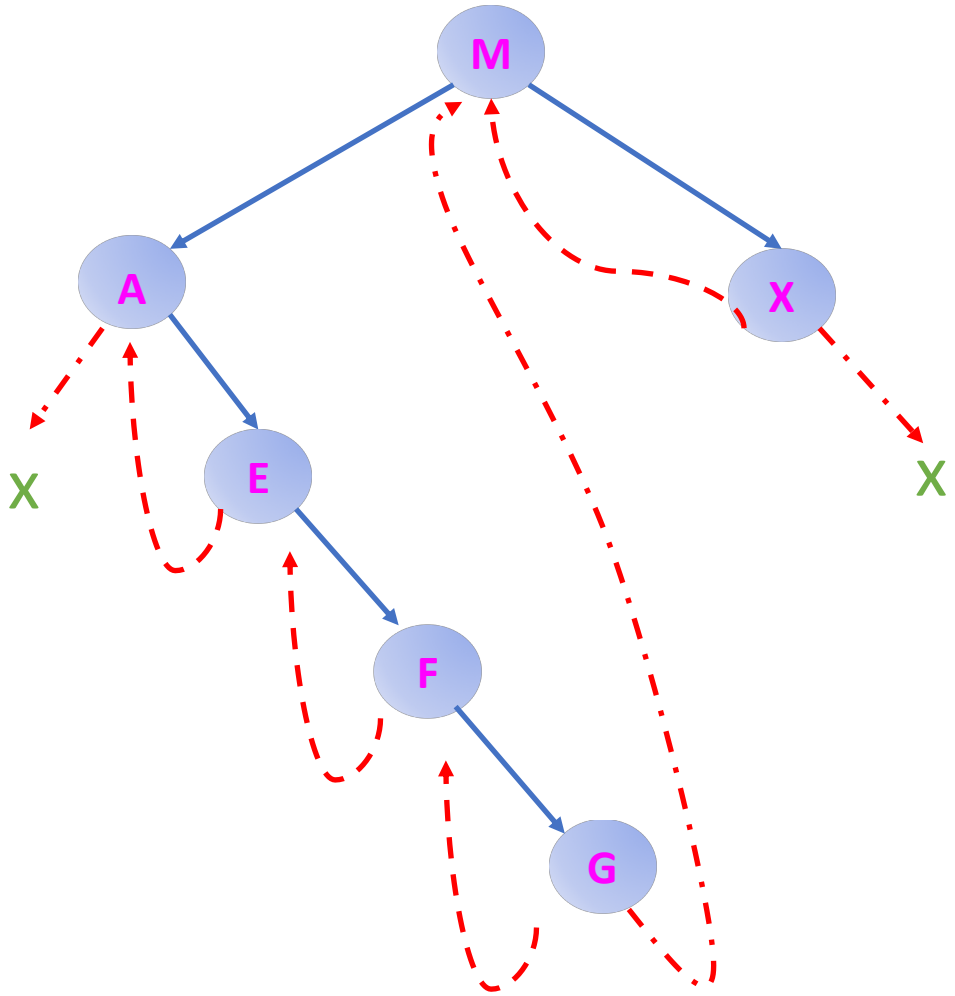
- To do **inorder traversal**, I need to **first** be able to find the inorder successor of any given node!

So, how do we do inorder traversal?



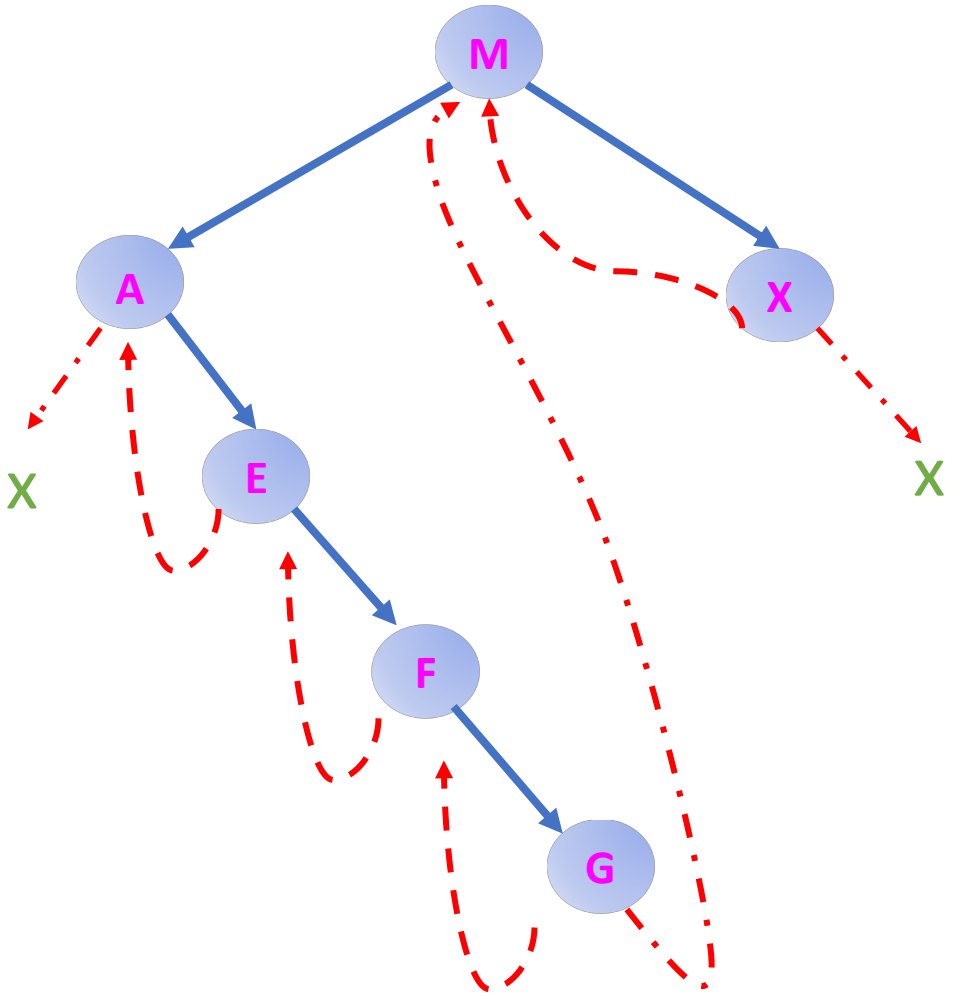
- To **do inorder traversal**, I need to **first** be able to find the inorder successor of any given node!
- Exercise: In no more than 5 minutes, write Java code that, given a Node n, finds $inSucc(n)$.

So, how do we do inorder traversal?



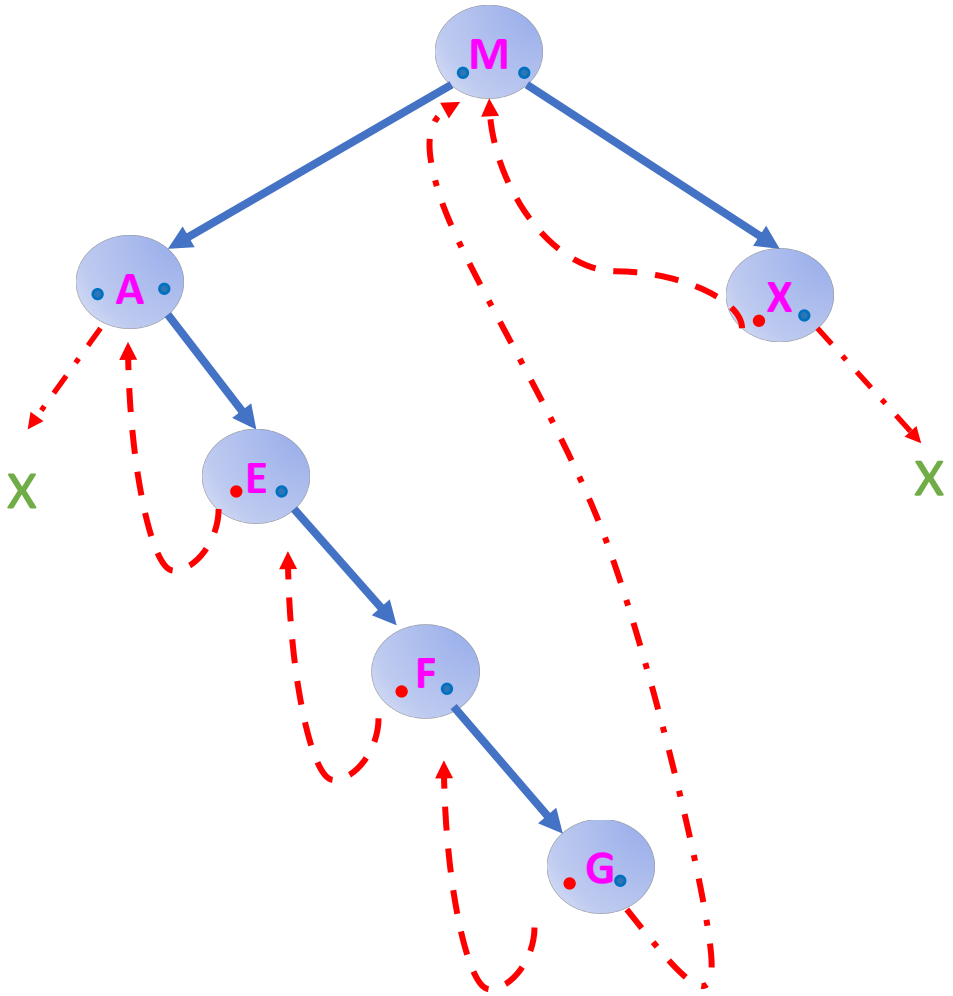
- To do **inorder traversal**, I need to **first** be able to find the inorder successor of any given node!
- Exercise: In no more than 5 minutes, write Java code that, given a Node n, finds $inSucc(n)$.
  - You might need to **add something** to the original linked binary tree model.

Finding the inorder successor



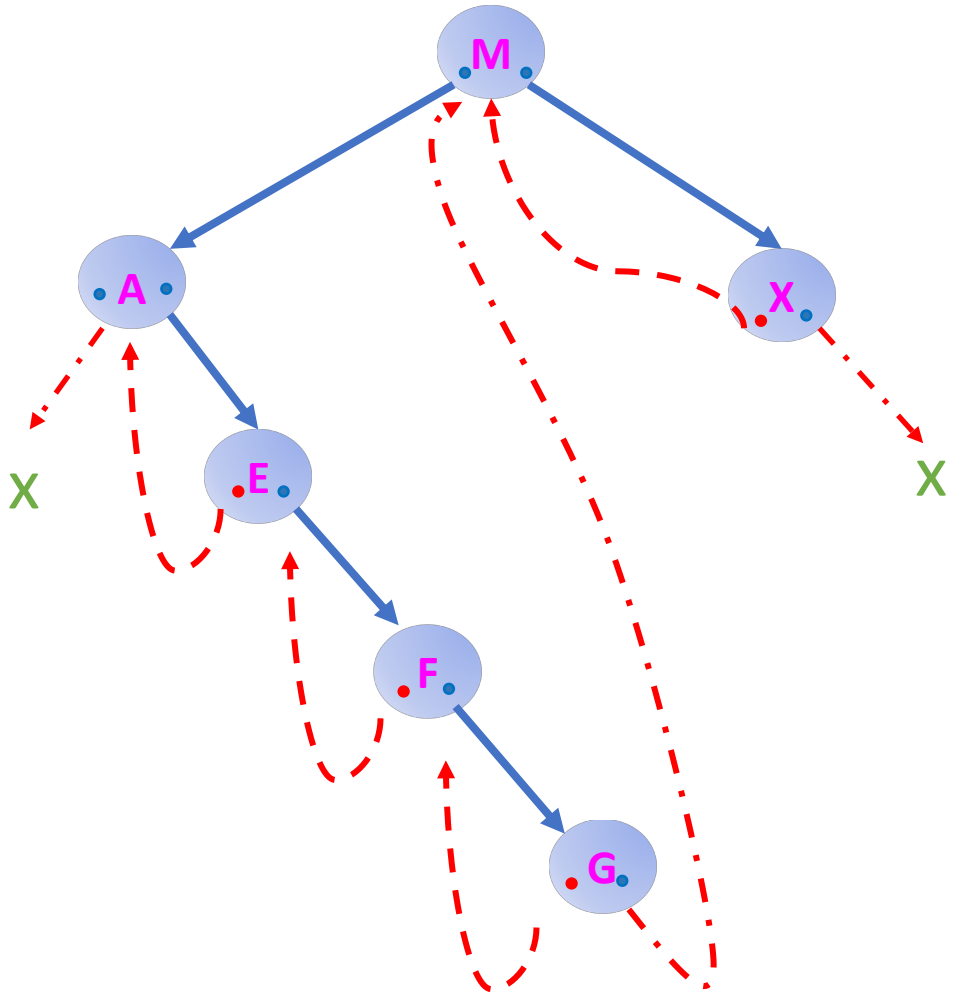
- We need information about whether a pointer is a **classic pointer** or a thread!

Finding the inorder successor



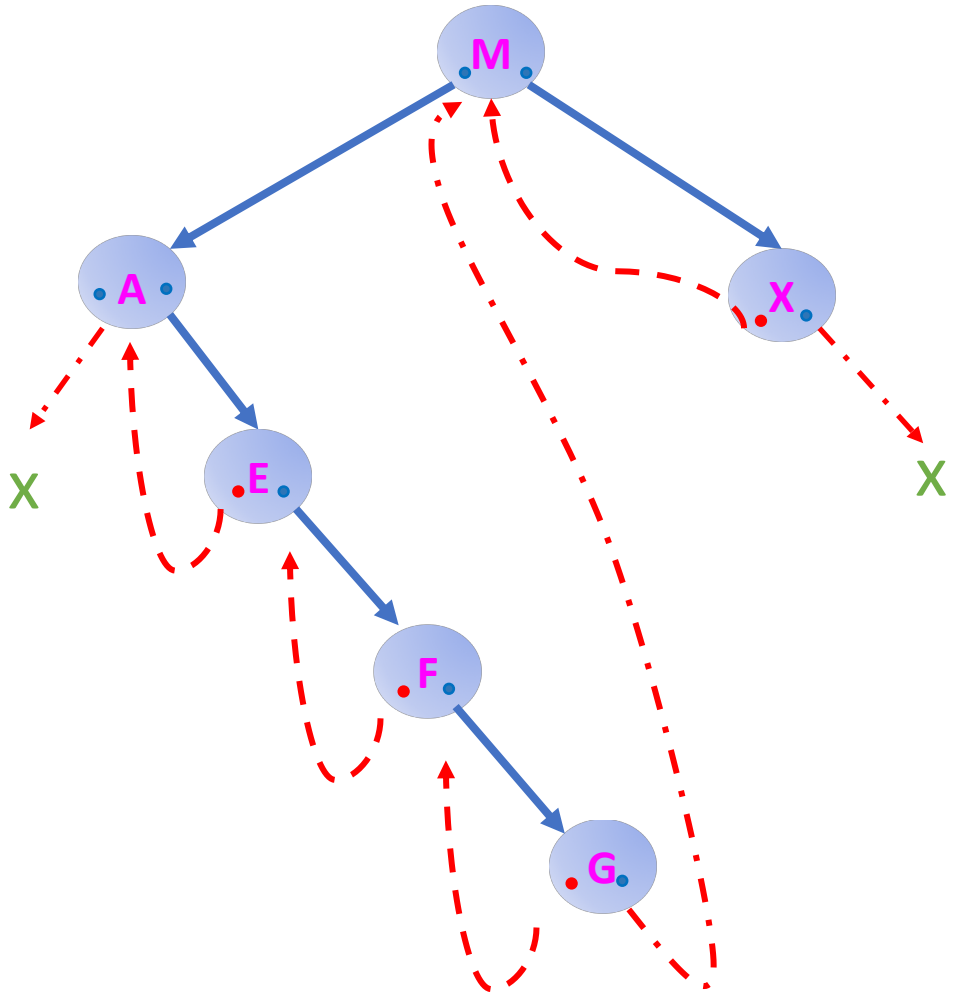
- We need information about whether a pointer is a **classic pointer** or a thread!
- Easiest solution: **one bit per link**.
  - State: 1 for thread, 0 for link (or vice versa)
  - Call those bits lb, rb.

Finding the inorder successor




```
Node inSucc(Node n){
        if(n.rb)
                return n.right; // null or otherwise
        else{ // Go right and then as far left
              // as you can
                n = n.right;
                while(!n.lb) // What does n.lb = 0 mean?
                        n=n.left;
                return n;
        }
}
```

Finding the inorder successor

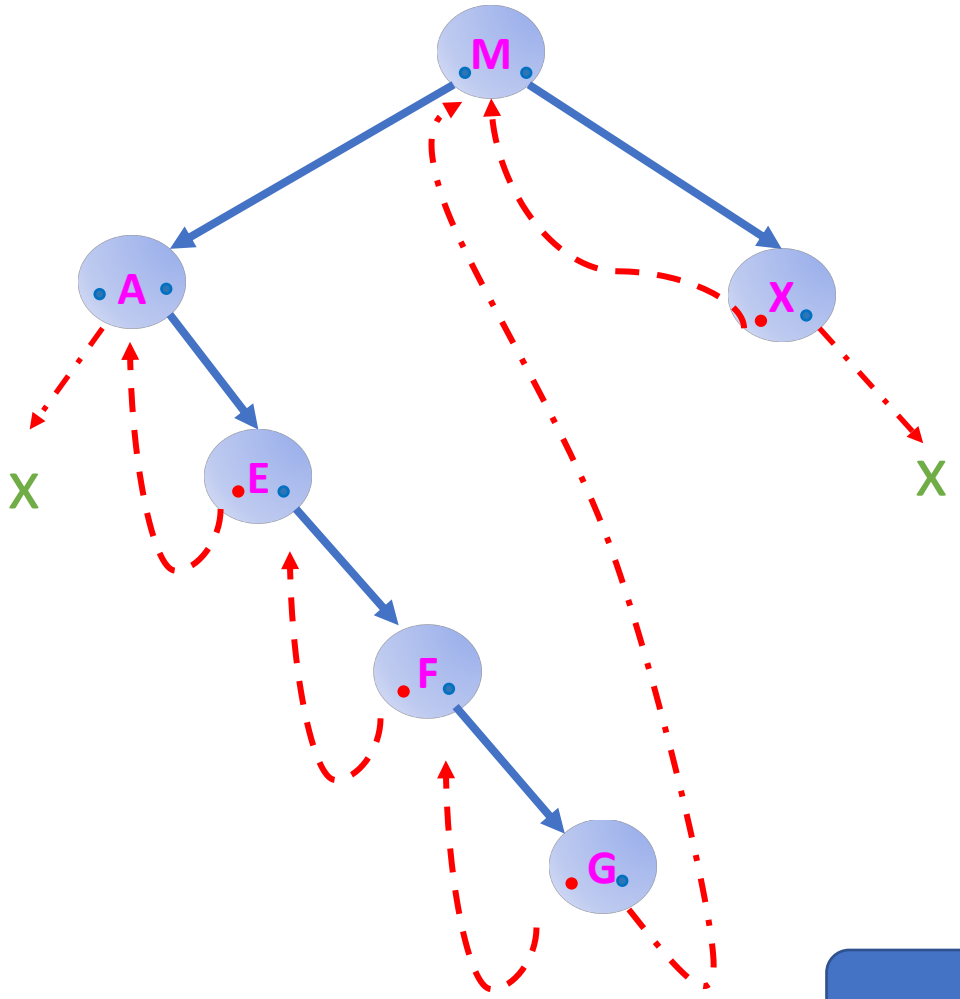



```
Node inSucc(Node n){
        if(n.rb)
                return n.right; // thread
        else{ // Go right and then as far left
              // as you can
                n = n.right;
                while(!n.lb) // What does n.lb = 0 mean?
                        n=n.left;
                return n;
        }
}
```

Constant

Linear

Finding the inorder successor




```
Node inSucc(Node n){
    if(n.rb)
        return n.right; // thread
    else{ // Go right and then as far left
          // as you can
        n = n.right;
        while(!n.lb) // What does n.lb = 0 mean?
            n=n.left;
        return n;
    }
}
```

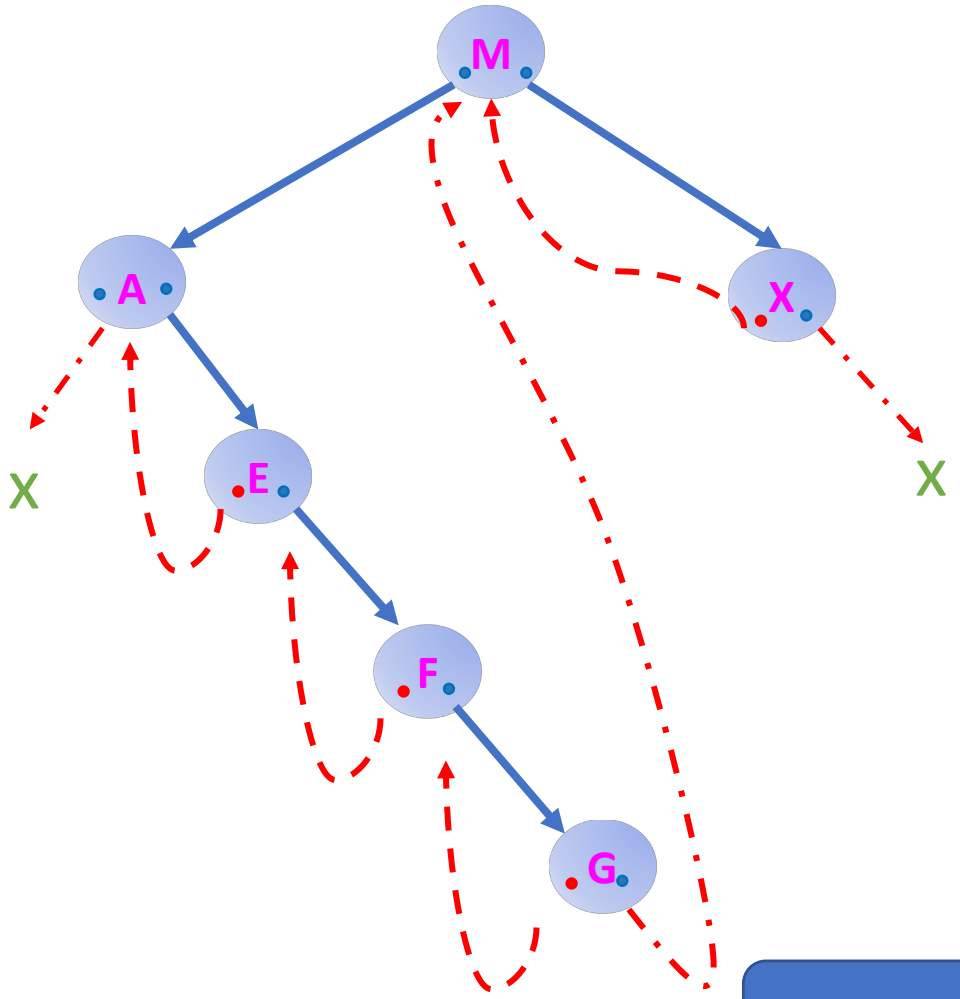
Constant

Linear

So, let's answer this question: Finding the inorder successor of a node in a threaded tree runs in…

Constant time | Linear time | Logarithmic time | Something else (what?)

Finding the inorder successor




```
Node inSucc(Node n){
    if(n.rb)
        return n.right; // thread
    else{ // Go right and then as far left
          // as you can
        n = n.right;
        while(!n.lb) // What does n.lb = 0 mean?
            n=n.left;
        return n;
    }
}
```

Constant

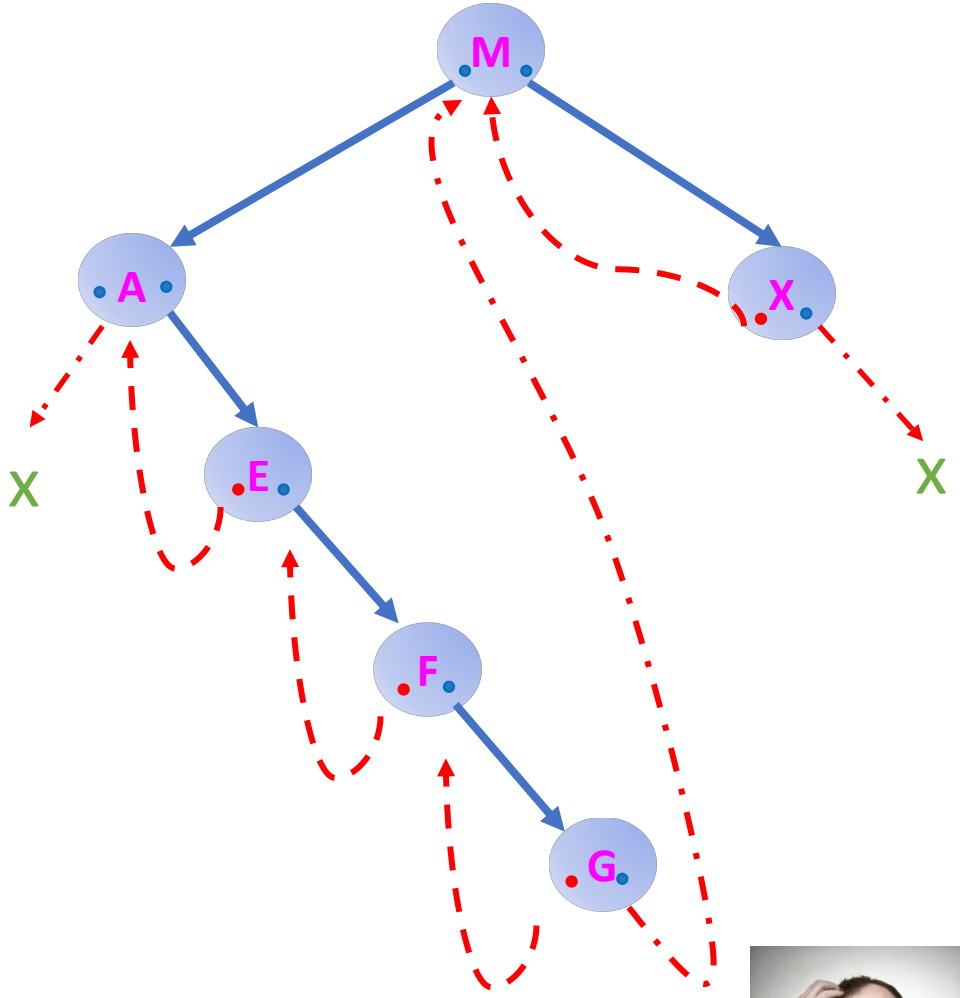
Linear

So, let's answer this question: Finding the inorder successor of a node in a threaded tree runs in…

Amortized **constant time!**

Constant time | Linear time | Logarithmic time | Something else (what?)

Finding the inorder successor



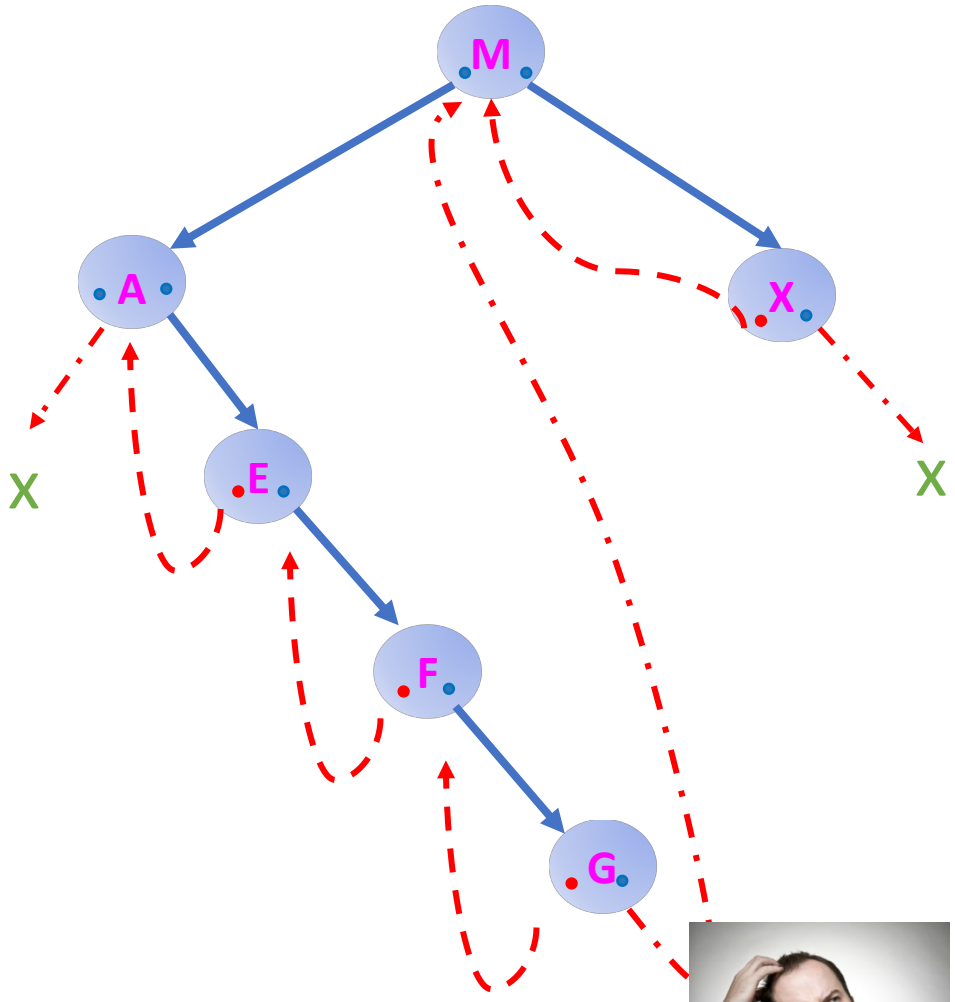

```
Node inSucc(Node n){
    if(n.rb)
        return n.right; // thread
    else{ // Go right and then as far left
          // as you can
        n = n.right;
        while(!n.lb) // What does n.lb = 0 mean?
            n=n.left;
        return n;
    }
}
```

Constant

Linear

- But hold on… **best case constant**, **worst-case linear**, how does that mean ***amortized constant?***

Finding the inorder successor



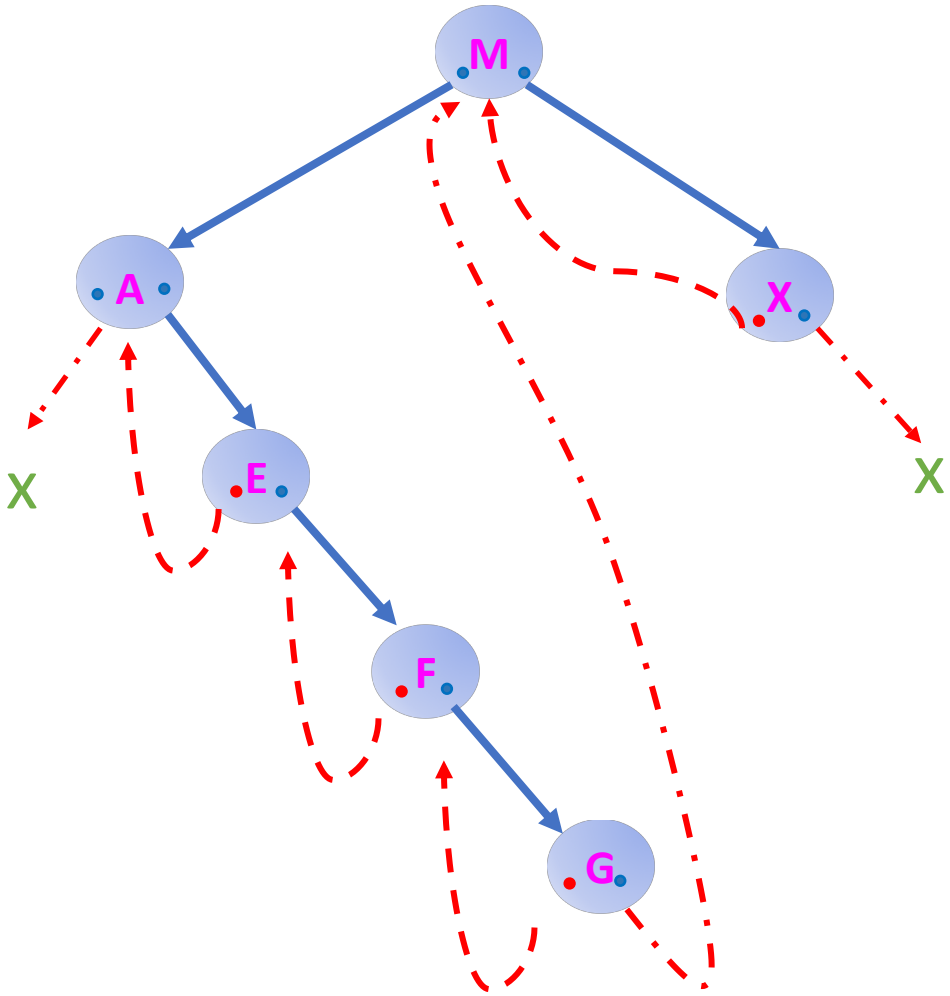

```
Node inSucc(Node n){
    if(n.right == null) return;
    if(n.rb)
        n = n.right; // thread
    else{ // Go right and then as far left
          // as you can
        n = n.right;
        while(!n.lb) // What does n.lb = 0 mean?
            n=n.left;
        return n;
    }
}
```

Constant

Linear

- But hold on… **best case constant**, **worst-case linear**, how does that mean ***amortized constant?***
- Let's present the full **inorder traversal** algo first and then we'll talk about this in detail…

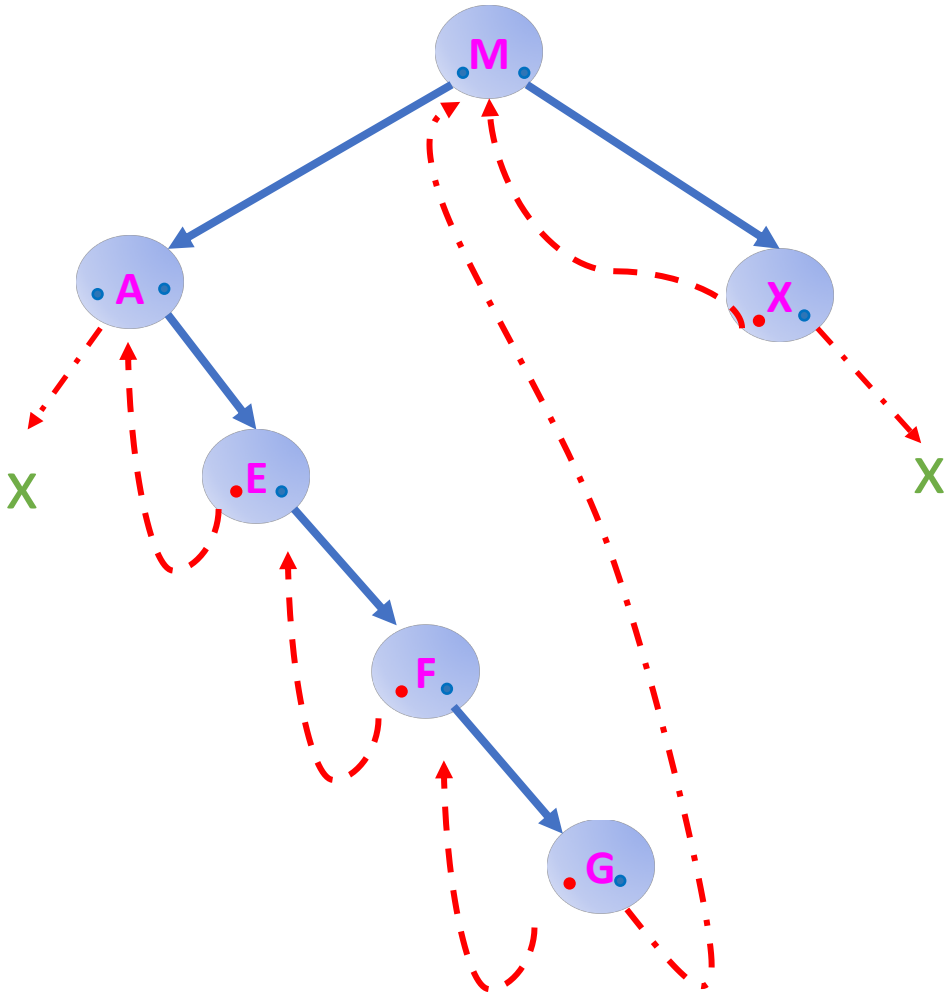
Question for you



How many pointers are there in a binary tree with n nodes?
(threaded, balanced, unbalanced, search, random, whatever)

n	$2n$	$\log_2 n$	Some other $f(n)$ (what?)

Question for you

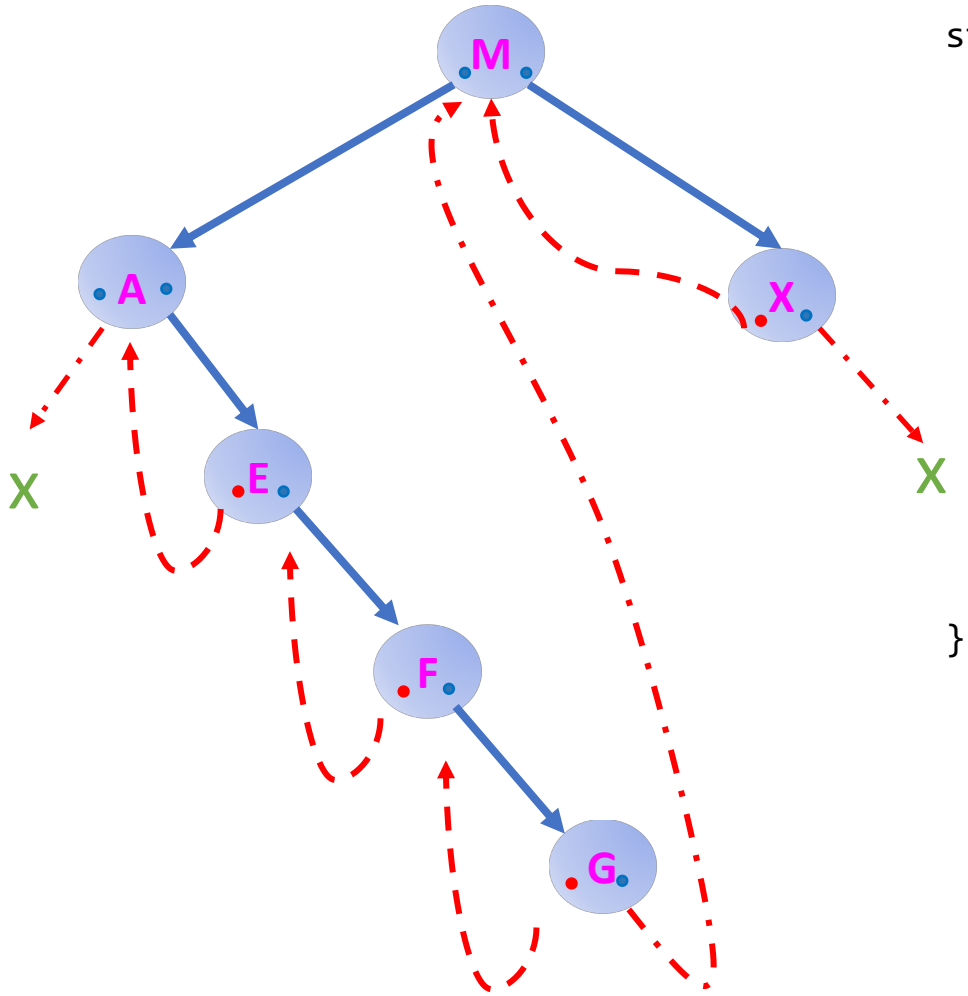


How many pointers are there in a binary tree with n nodes?
(threaded, balanced, unbalanced, search, random, whatever)

n | $2n$ | $\log_2 n$ | Some other $f(n)$ (what?)

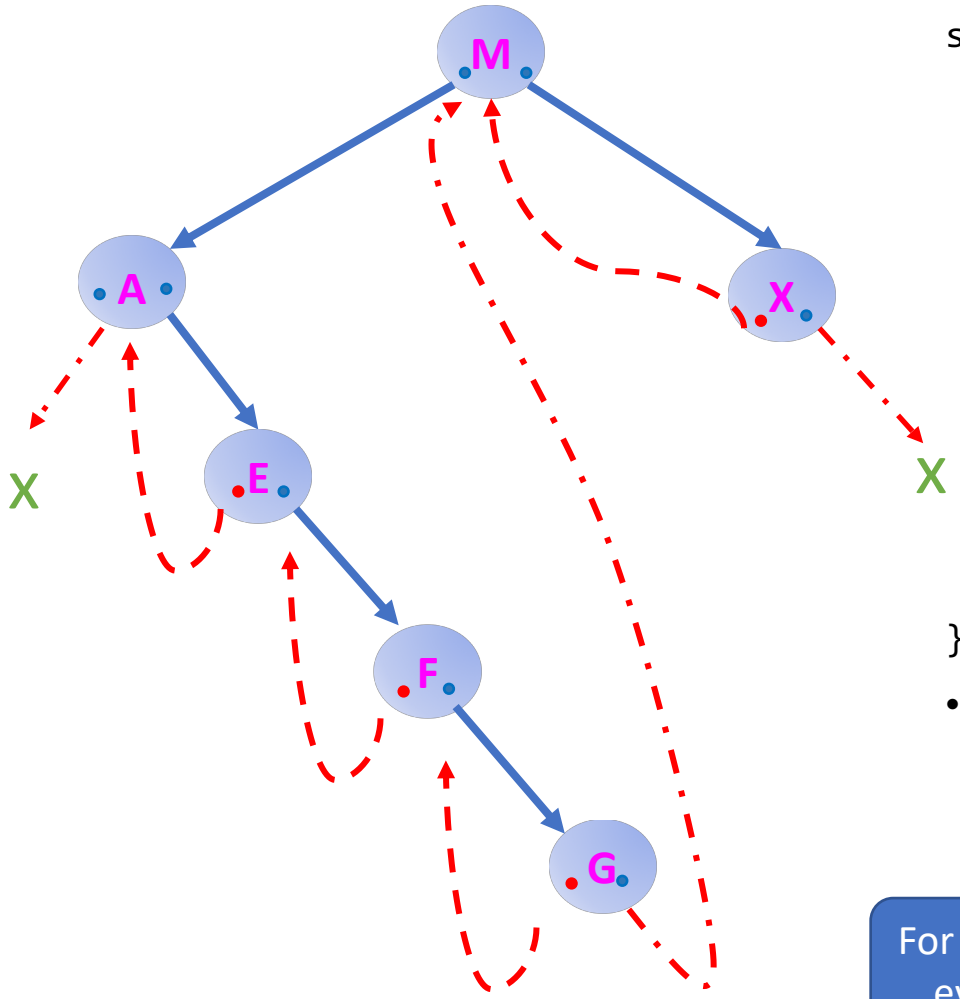
- Two outgoing pointers per node, bruh
- **Let's keep this in mind….**

Inorder traversal in a threaded tree




```
static void inorderTraversal(ThreadedTree t){
        Node n = t.getRoot();
        while(n.getLeft() != null)
                n = n.getLeft();   // Go as left as you can.
        print(n.getValue());
        while(n != null){
                n = inSucc(n); // Call to our previous function
                print(n.getValue());
        }

}
```

Inorder traversal in a threaded tree




```
static void inorderTraversal(ThreadedTree t){
        Node n = t.getRoot();
        while(n.getLeft() != null)
                n = n.getLeft();   // Go as left as you can.
        print(n.getValue());
        while(n != null){
                n = inSucc(n); // Call to our previous function
                print(n.getValue());
        }
}
```

- Let a random edge in a threaded tree be e. How many times will `inorderTraversal` **traverse** e? *(By traversing, we quite literally mean: "Go from the starting node to the ending node.")*

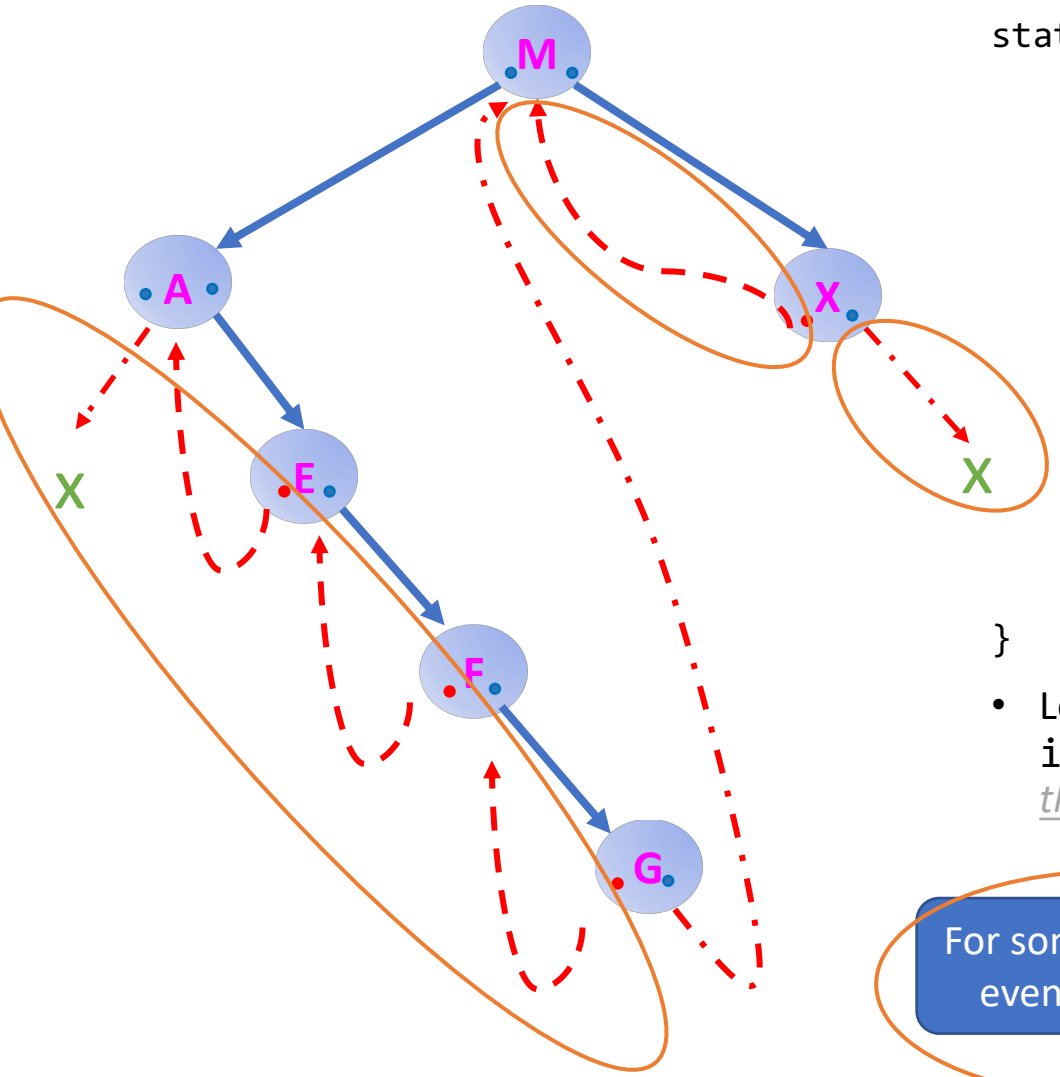
For some e, it won't even visit them.

All of them exactly once

For **some** e, it will scan them twice

For **all** e, it will scan them twice

Inorder traversal in a threaded tree




```
static void inorderTraversal(ThreadedTree t){
        Node n = t.getRoot();
        while(n.getLeft() != null)
                n = n.getLeft();   // Go as left as you can.
        print(n.getValue());
        while(n != null){
                n = inSucc(n); // Call to our previous function
                print(n.getValue());
        }
}
```

- Let a random edge in a threaded tree be e. How many times will `inorderTraversal` **traverse** e? *(By traversing, we quite literally mean: "Go from the starting node to the ending node.")*

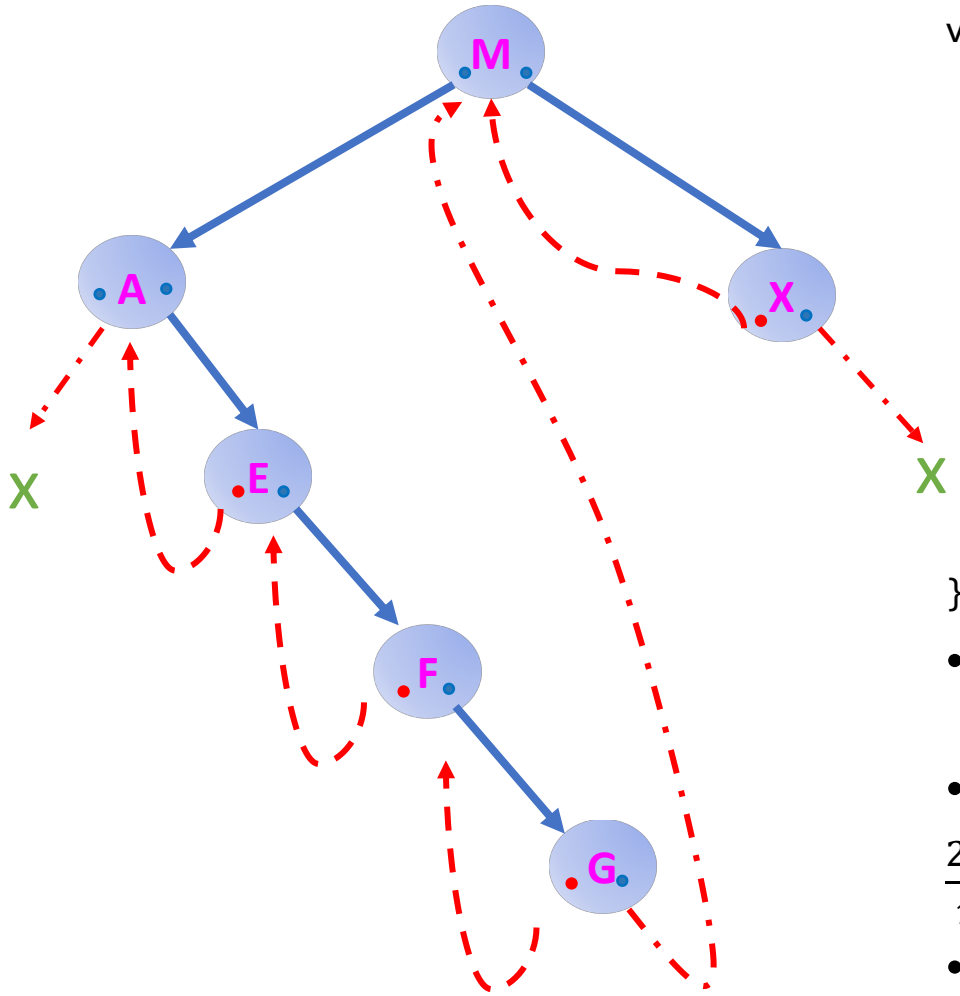
For some e, it won't even visit them.

All of them exactly once

For **some** e, it will scan them twice

For **all** e, it will scan them twice

Inorder traversal in a threaded tree

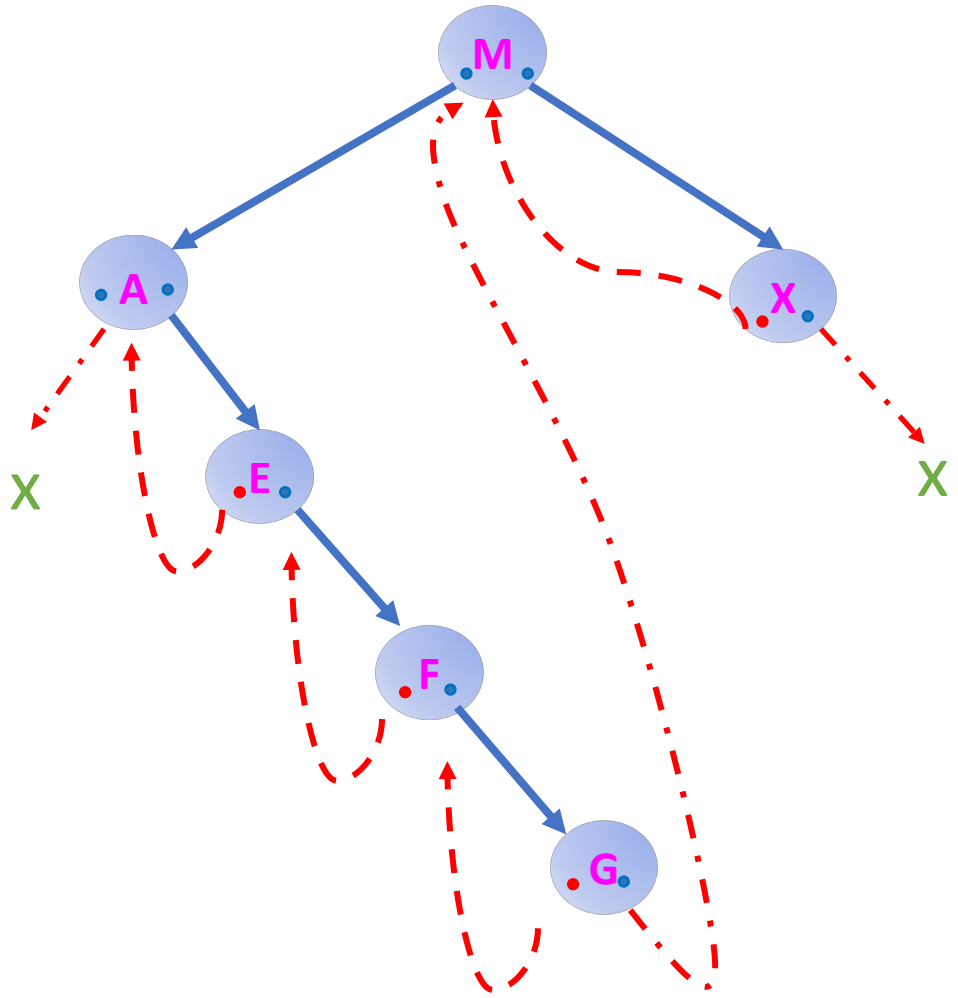



```
void inorderTraversal(ThreadedTree t){
        Node n = t.root;
        while(n.left != null)
                n = n.left;          // Go as left as you can.
        print(n.value);
        while(n != null){
                n = inSucc(n); // Call to our previous function
                print( (n == NULL) ? NULL : n.value);
        }
}
```

- Like we do in Algorithms, let's **pretend that we actually visited all $2n$ links.**
- Then, **on average, when at any given node,** the algorithm visits $\frac{2n}{n} = 2 = \mathcal{O}(1)$ links to find its **inorder successor**!
- Hence **amortized constant** time ☺

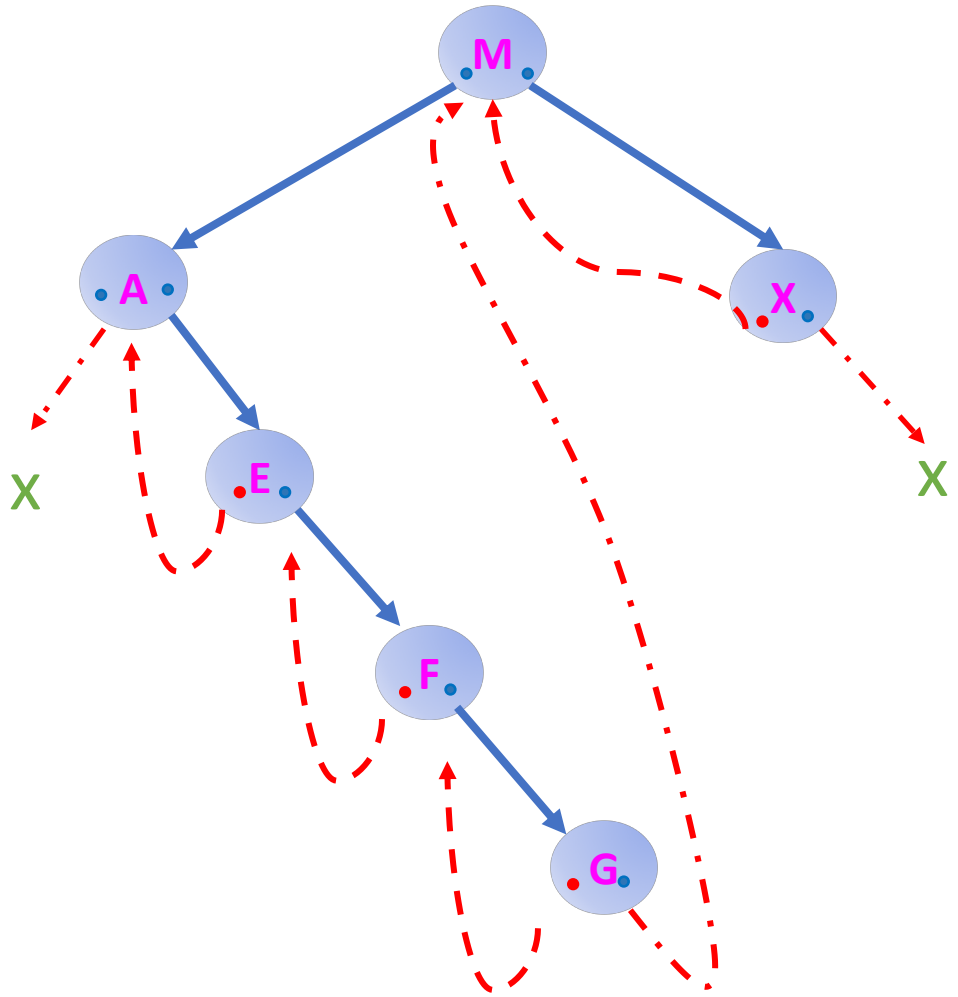
Exercise for you

- The goal: write code that traverses a threaded BST in descending order!
- But! You will do this in pairs!
  1. Student #1 writes a routine that finds the inorder predecessor of a threaded tree
  2. Student #2 uses that routine ***(must agree on name, signature…)*** to write the full method!



(One) Solution

```
Node inPrec(Node n){
        if(n.lb)  // Inorder predecessor or NULL
                return n.left;
        n = n.left;
        while(!n.rb)
                n = n.right; // Go as far right as you can
        return n;
}
void descOrderTraversal(ThreadedTree t){
        Node n = t.root;
        while(n.right != null)
                n = n.right; // Go rightmost
        print(n.value);
        while(n.left != null){
                n = inPrec(n);
      print( (n == NULL) ? NULL : n.value);
}
```



Short break – Memory alignment

- What will the output of ***the last line*** of this C program be (64-bit OS)?

13	17	24	Something else (what?)

```c
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char byte;

typedef struct myStruct {
    int a;
    double d;
    char c;
    byte b;
} myStruct;

int main(int argc, char** argv){
    printf("On this machine, size of int =%lu bytes.\n", sizeof(int));
    printf("On this machine, size of a double =%lu bytes.\n", sizeof(double));
    printf("On this machine, size of a char =%lu bytes. \n", sizeof(char));
    printf("On this machine, size of a byte =%lu bytes. \n", sizeof(byte));
    printf("On this machine, size of a myStruct =%lu bytes.\n", sizeof(myStruct));
    return EXIT_SUCCESS;
}
```

Short break – Memory alignment

- What will the output of ***the last line*** of this C program be (64-bit OS)?

13	17	24	Something else (what?)

```c
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char byte;

typedef struct myStruct {
    int a;
    double d;
    char c;
    byte b;
} myStruct;

int main(int argc, char** argv){
    printf("On this machine, size of int =%lu bytes.\n", sizeof(int));
    printf("On this machine, size of a double =%lu bytes.\n", sizeof(double));
    printf("On this machine, size of a char =%lu bytes. \n", sizeof(char));
    printf("On this machine, size of a byte =%lu bytes. \n", sizeof(byte));
    printf("On this machine, size of a myStruct =%lu bytes.\n", sizeof(myStruct));
    return EXIT_SUCCESS;
}
```

Memory alignment at word length (8 bytes for a 64-bit machine!)

Memory footprint (in theory!)

- Threaded Tree node has:
  - An element of some `Comparable` type `T` (8 bytes at most)
  - Two Node pointers, both 4 bytes (in Java)
  - Two bits per node.
- So in theory, a threaded tree costs $8 * 8 + 4 * 8 + 2 = 98n$ bits

Memory footprint (in practice!)

- Threaded Tree node has:
  - An element of some `Comparable` type T (8 bytes at most)
  - Two Node pointers, both 4 bytes (in Java)
  - Two bits per node.
- So in theory, a threaded tree costs $8 * 8 + 4 * 8 + 2 = 98n$ bits
- But in practice, because of memory alignment, every node will be aligned to 24 bytes ☹

```
class ThreadedNode{
        T element; // 8 bytes, padded or allocated to variable
        ThreadedNode left, right // exactly 8 bytes for both;
        byte leftFlag, rightFlag; // 2 bytes + 6 for the padding = 8 total
}
```

Memory footprint (in practice!)

- Threaded Tree node has:
  - An element of some `Comparable` type T (8 bytes at most)
  - Two Node pointers, both 4 bytes (in Java)
  - Two bits per node.
- So in theory, a threaded tree costs $8 * 8 + 4 * 8 + 2 = 98n$ bits
- But in practice, because of memory alignment, every node will be aligned to 24 bytes ☹

```
class ThreadedNode{
        T element; // 8 bytes, padded or allocated to variable
        ThreadedNode left, right // exactly 8 bytes for both;
        byte leftFlag, rightFlag; // 2 bytes + 6 for the padding = 8 total
}
```

Total size in practice: $192n$ bits (barely under double!)

Issues with threaded trees

1. We have some spatial overhead (2 bits per node).
2. If we thread a tree for inorder traversal and then I want to do post-order or pre-order, gotta re-insert all the elements in a tree that is threaded differently for every traversal ☹
   - $\mathcal{O}(n \cdot \log_2 n)$ operation in the average case, $\mathcal{O}(n^2)$ in the worst case (very unbalanced tree)

Link Inversion

- Turns out we can perform a stack-less traversal with a single bit per node, instead of two!
- Improvement in theoretical storage from $98n$ to $97n$ bits.
  - Optimizing at the bit level is important in Networks.
- We call this method **link inversion.**
- Another benefit of link inversion is that It is **very easy** to modularize to pre-order and post-order traversal, without even re-writing code!
- One drawback: a full traversal tends to be **slower** than the threaded tree, because the sub-routines that are called when we move from ***any*** node to another one have 4 (four) elementary operations, where the threaded tree had 1 (one) (just dereference a pointer).

Question

- Remember the “recursive list count” demo?

Recursive List Demo

- We all agree that iterative is clearly the most natural and the best method to get the count.
  - The other two ones seem kind of stupid, honestly
- We will now talk about another stupid method to traverse this list, with some interesting properties when generalized to trees.

Link inversion for lists

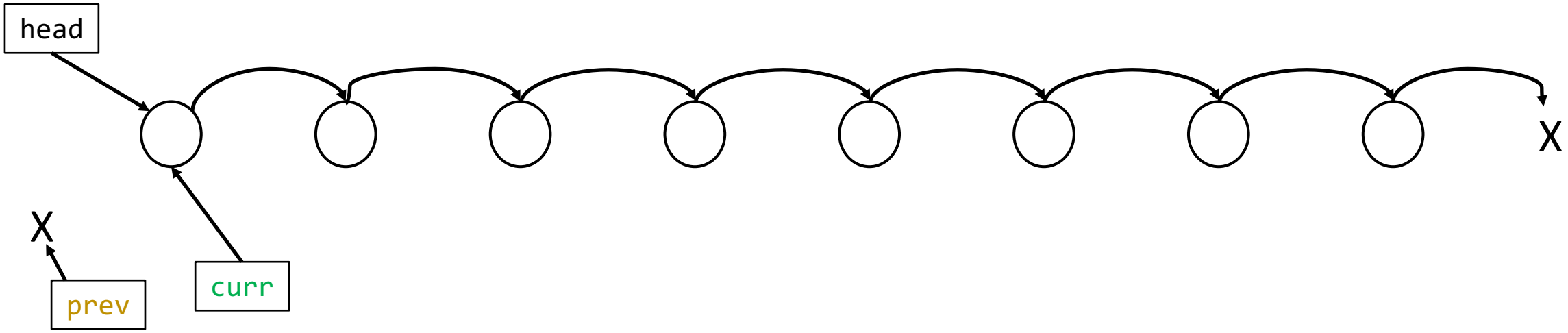
- This method does not use a stack.

Link inversion for lists

- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.

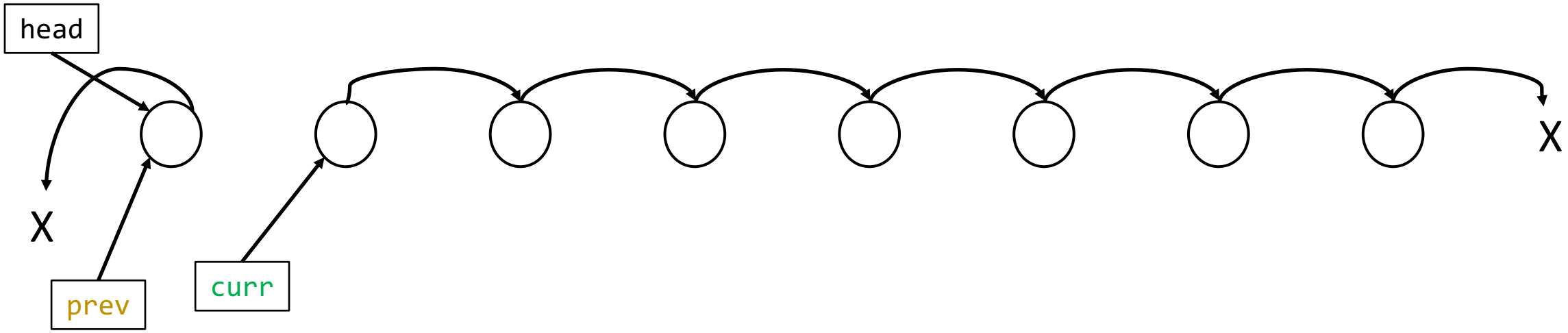
Link inversion for lists

- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.



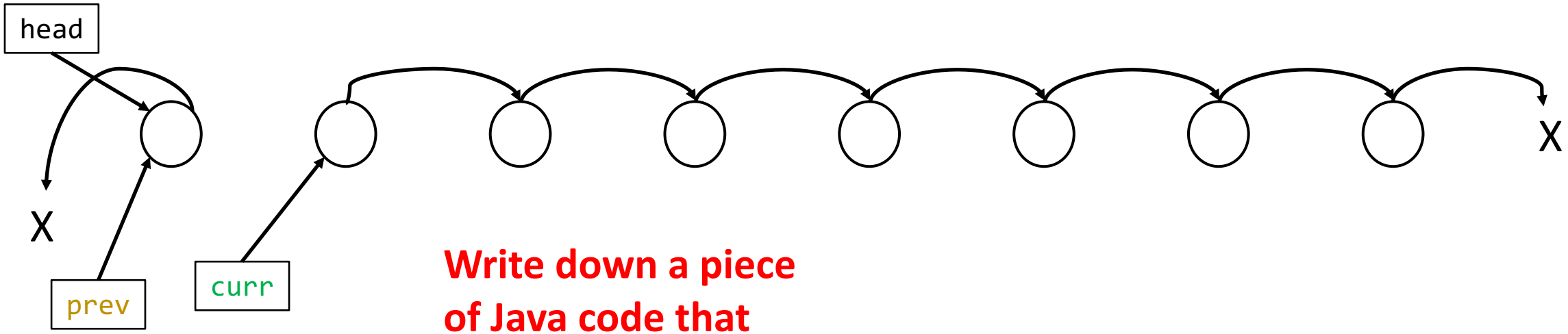
Link inversion for lists

- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.



Link inversion for lists

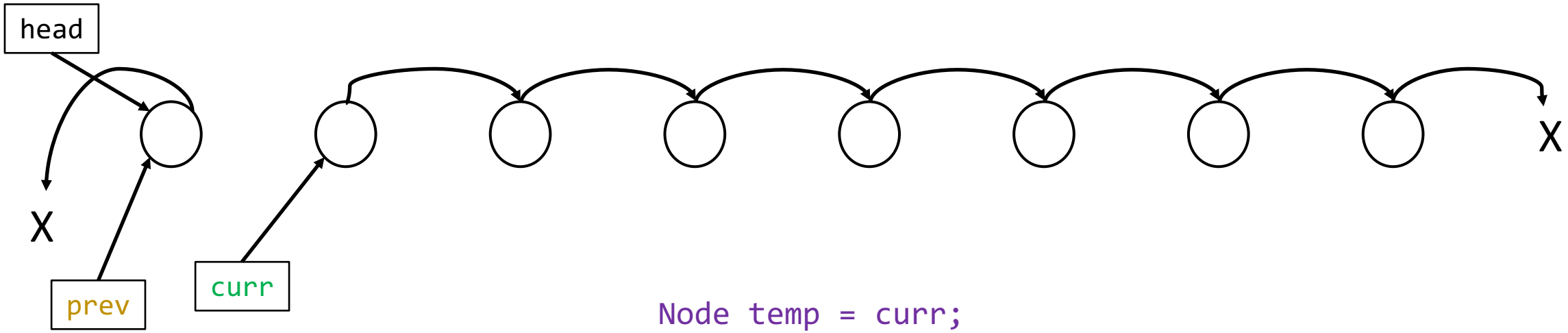
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.



Write down a piece of Java code that achieves this!

Link inversion for lists

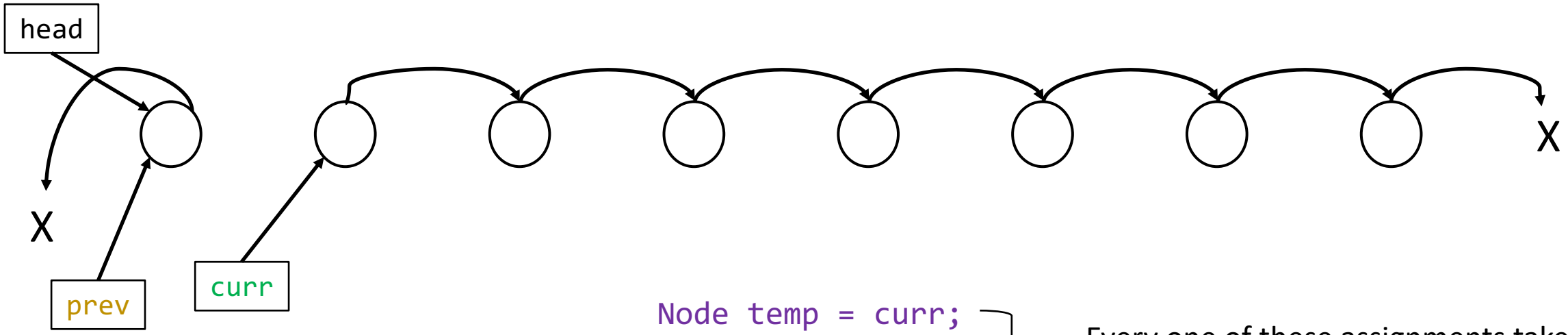
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
Node temp = curr;
curr = curr.next;
temp.next = prev;
prev = temp;
```

Link inversion for lists

- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.

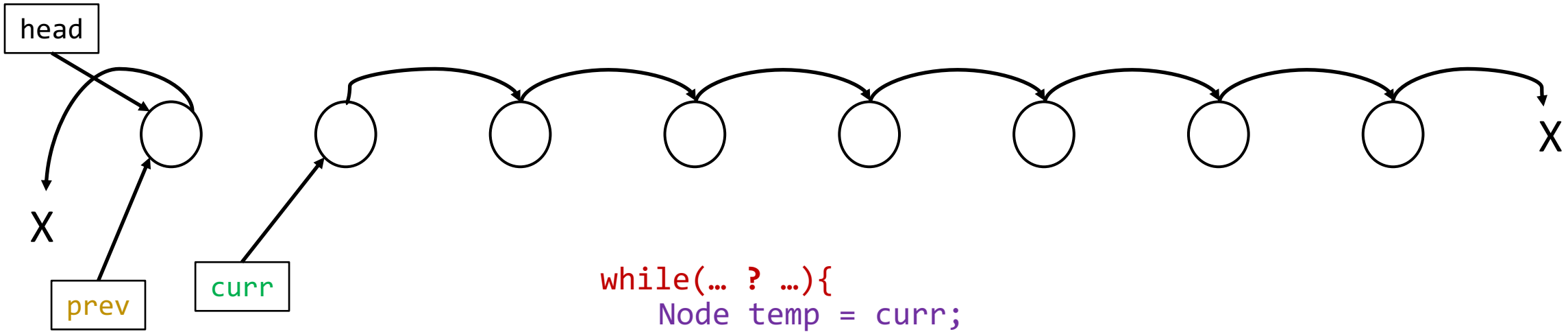



```
Node temp = curr;
curr = curr.next;
temp.next = prev;
prev = temp;
```

Every one of these assignments takes about **one nanosecond** to finish nowadays.

Link inversion for lists

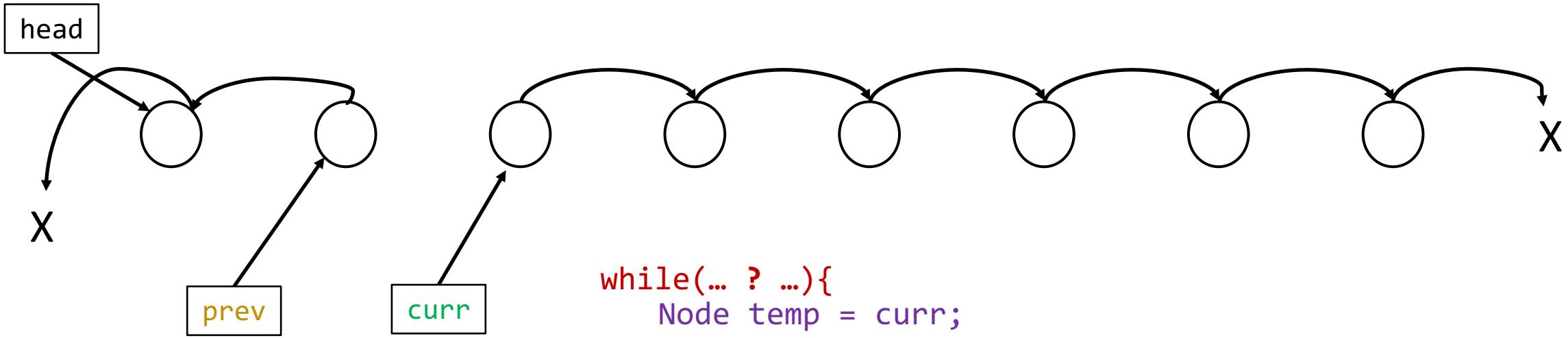
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

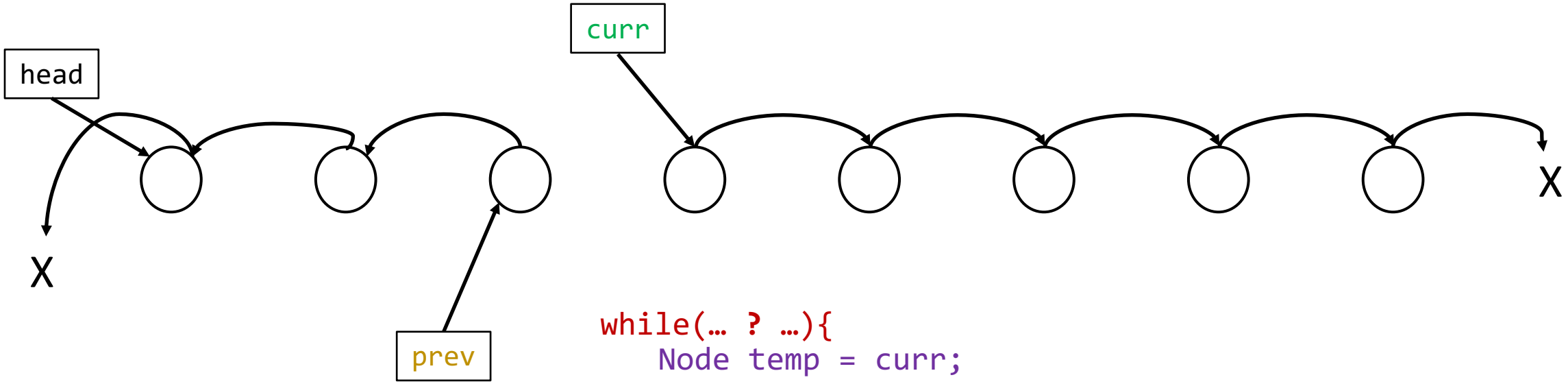
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

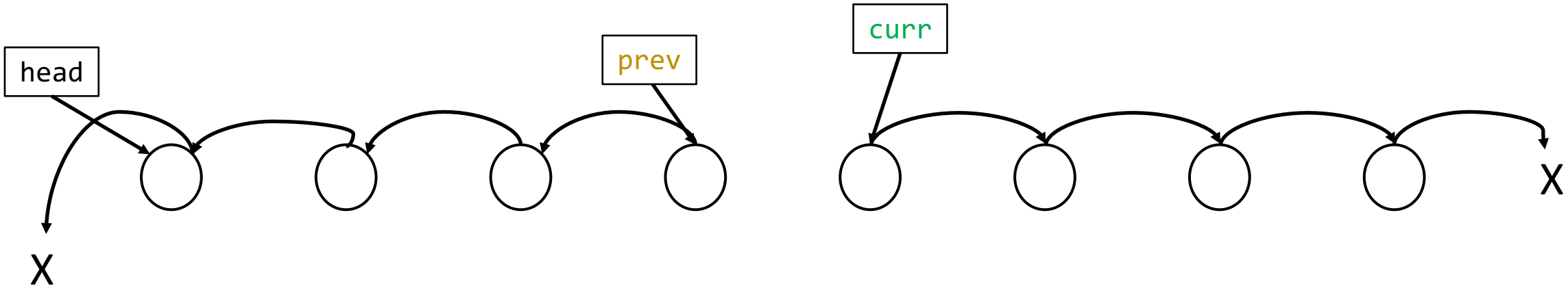
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

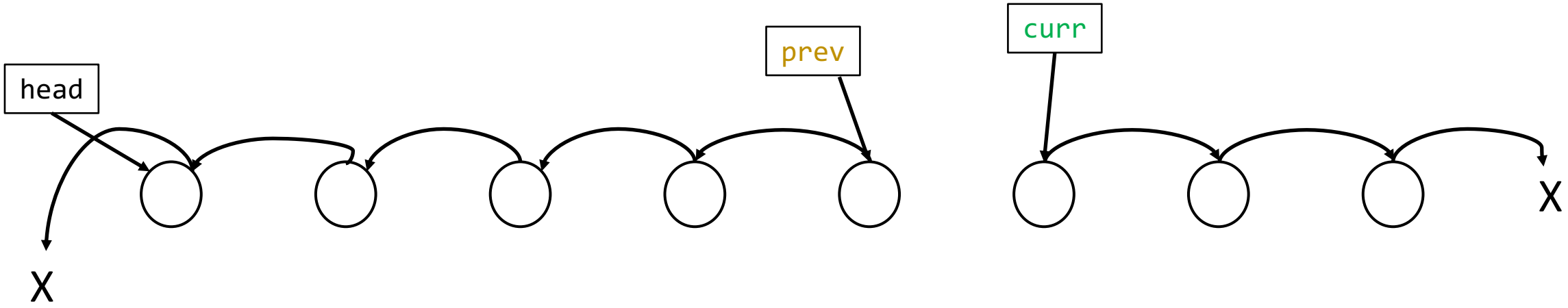
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

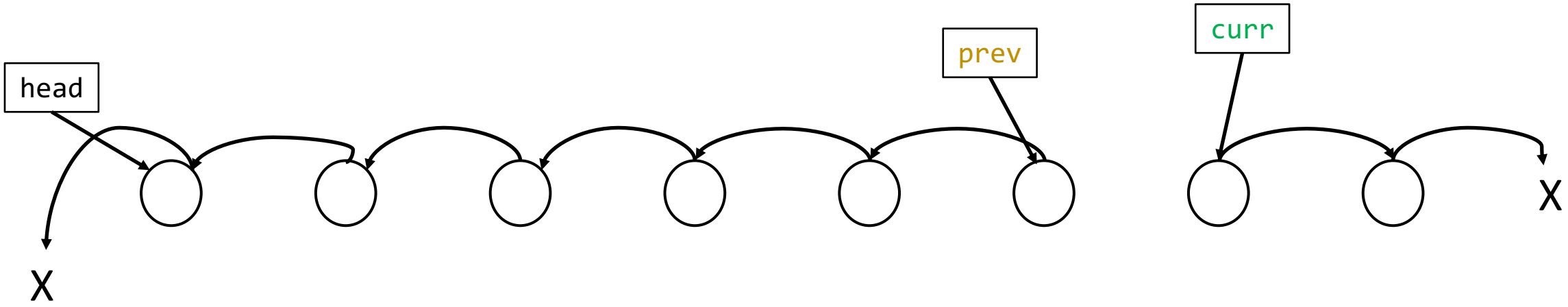
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

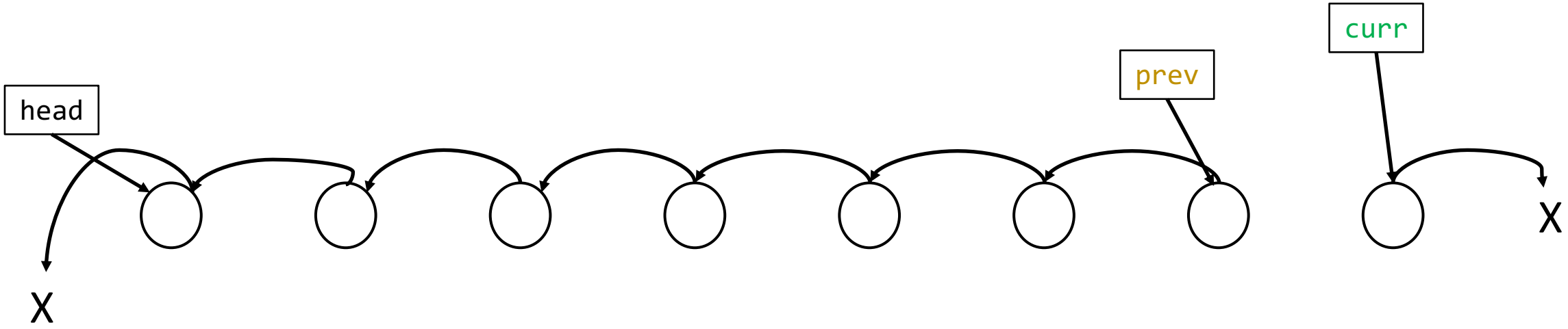
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

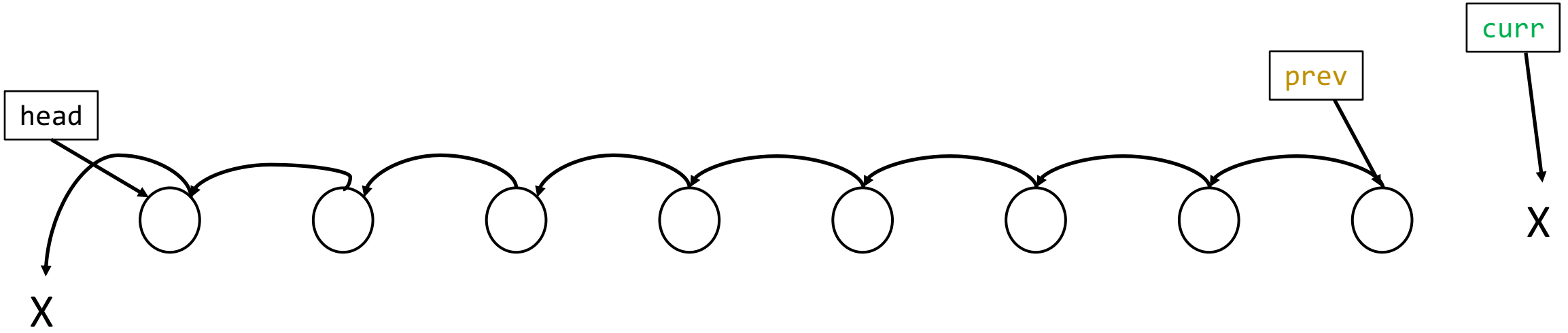
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

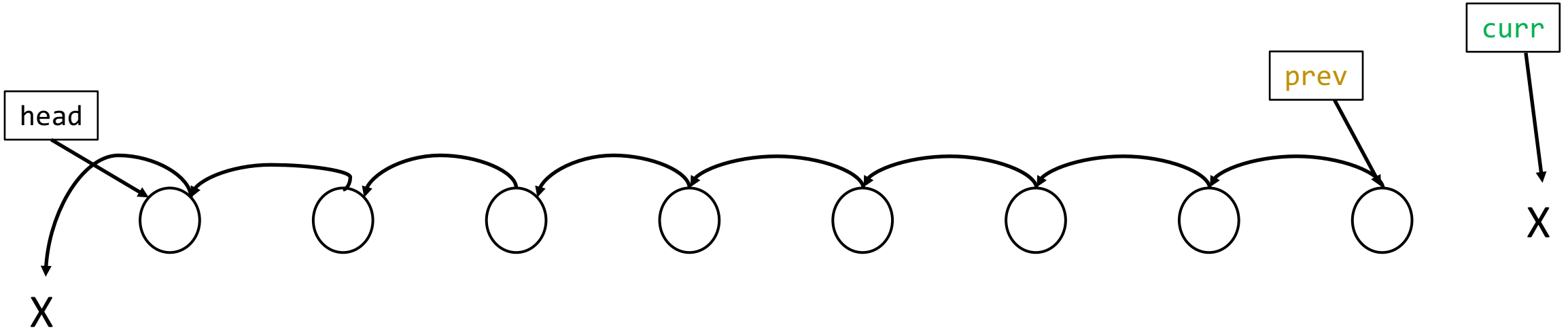
- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.




```
while(… ? …){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

Link inversion for lists

- This method does not use a stack.
- It uses two pointers, `prev` and `curr`.

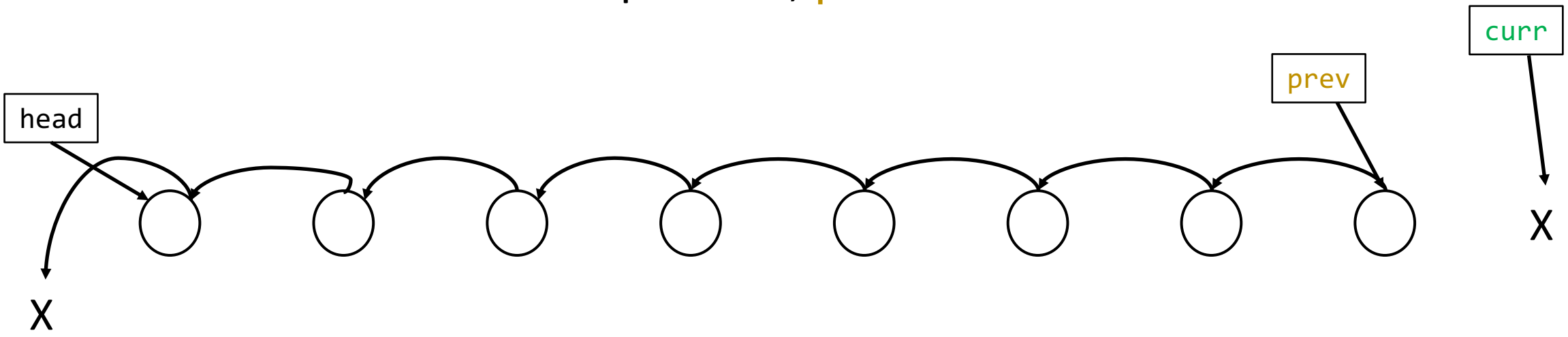



```
while(curr!=null){
    Node temp = curr;
    curr = curr.next;
    temp.next = prev;
    prev = temp;
}
```

To avoid a NullPointerException…

Link inversion for lists

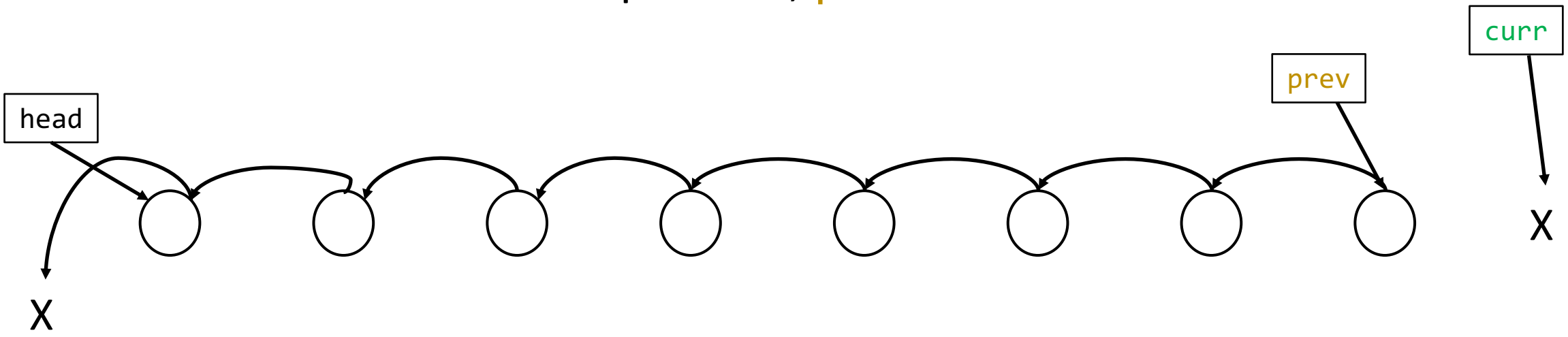
- The method uses two pointers, `prev` and `curr`.



Clearly we can't leave the list like this….

Link inversion for lists

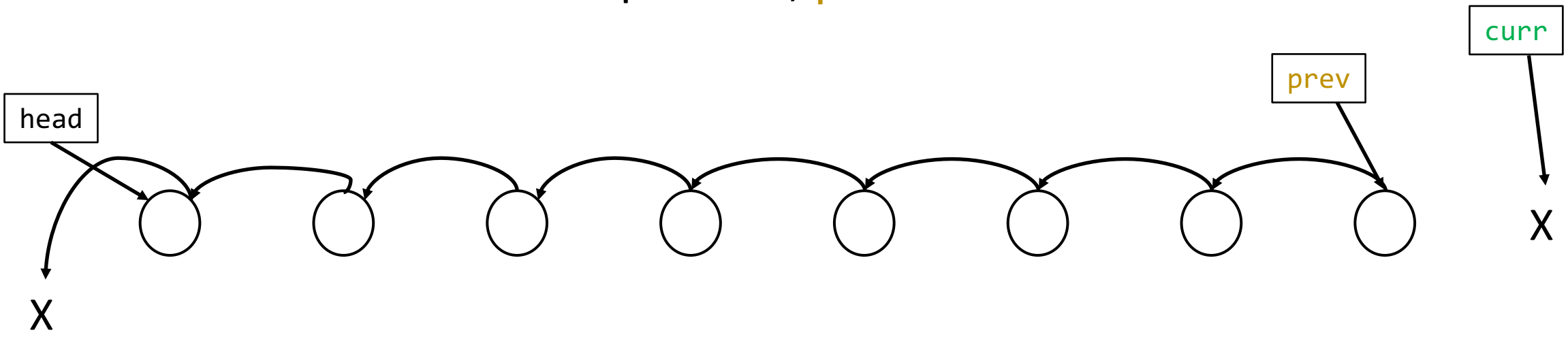
- The method uses two pointers, `prev` and `curr`.



Write down a piece of Java code that helps us backtrack one node!

Link inversion for lists

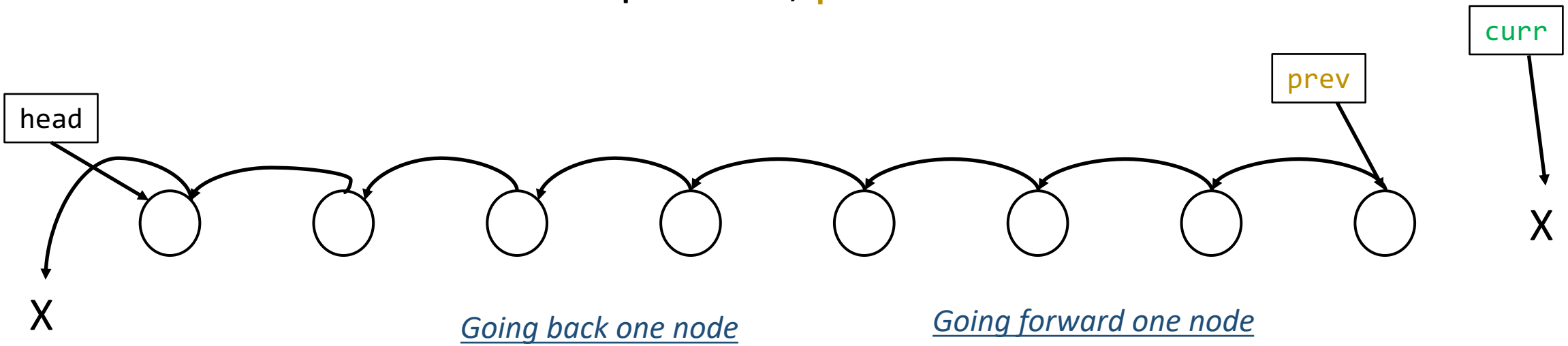
- The method uses two pointers, `prev` and `curr`.




```
Node temp = prev;
prev = prev.next;
temp.next = curr;
curr = temp;
```

Link inversion for lists

- The method uses two pointers, `prev` and `curr`.



Going back one node

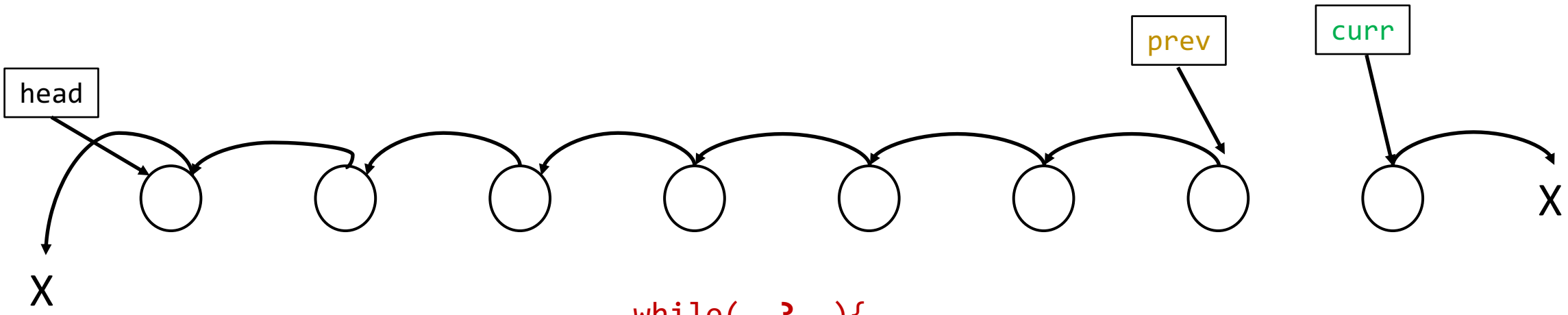
```
Node temp = prev;
prev = prev.next;
temp.next = curr;
curr = temp;
```

Going forward one node

```
Node temp = curr;
curr = curr.next;
temp.next = prev;
prev = temp;
```

Link inversion for lists

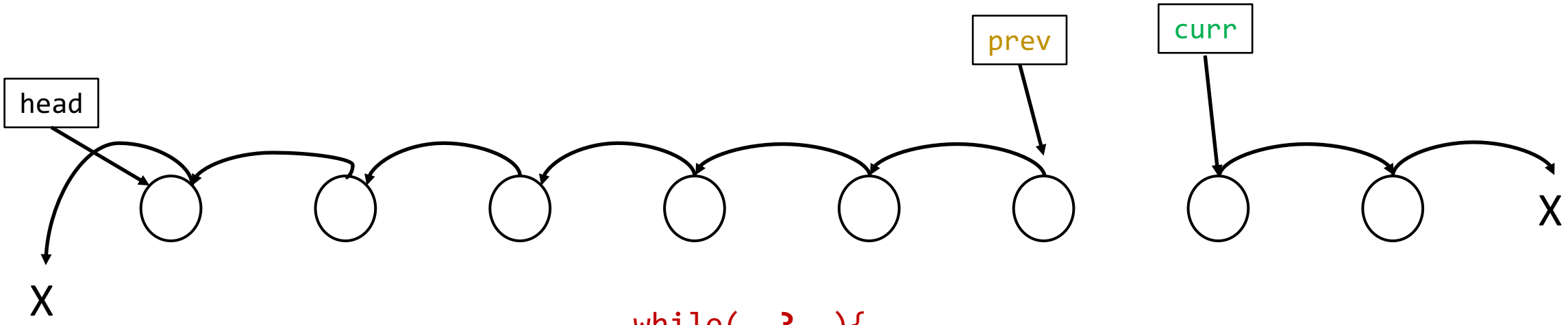
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

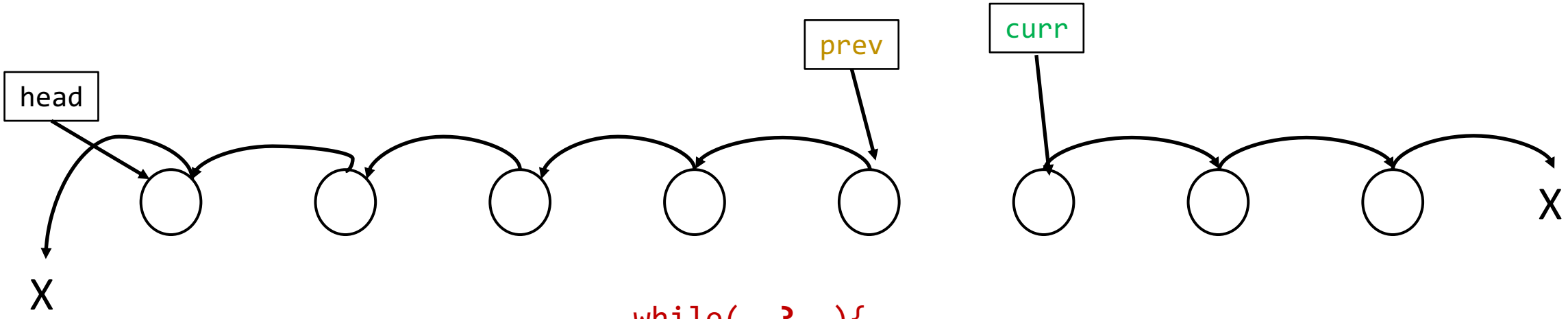
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

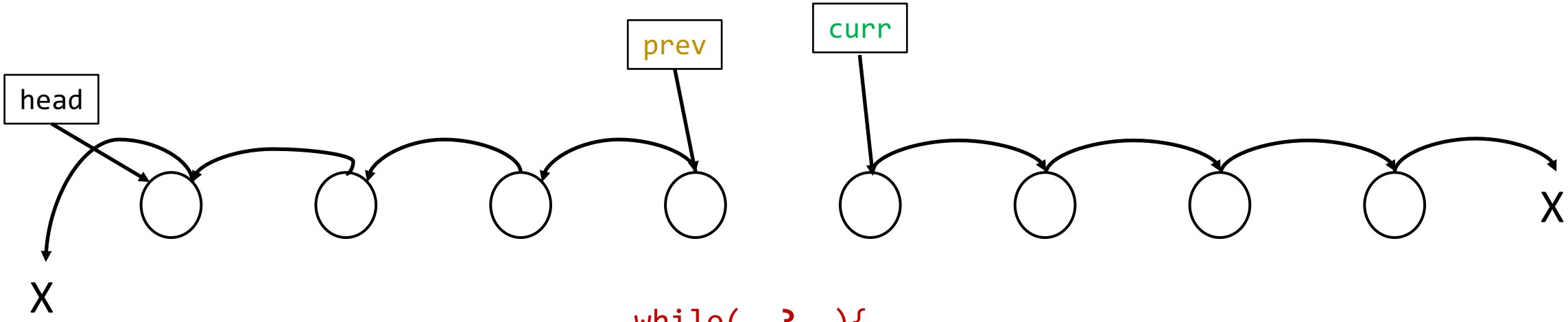
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

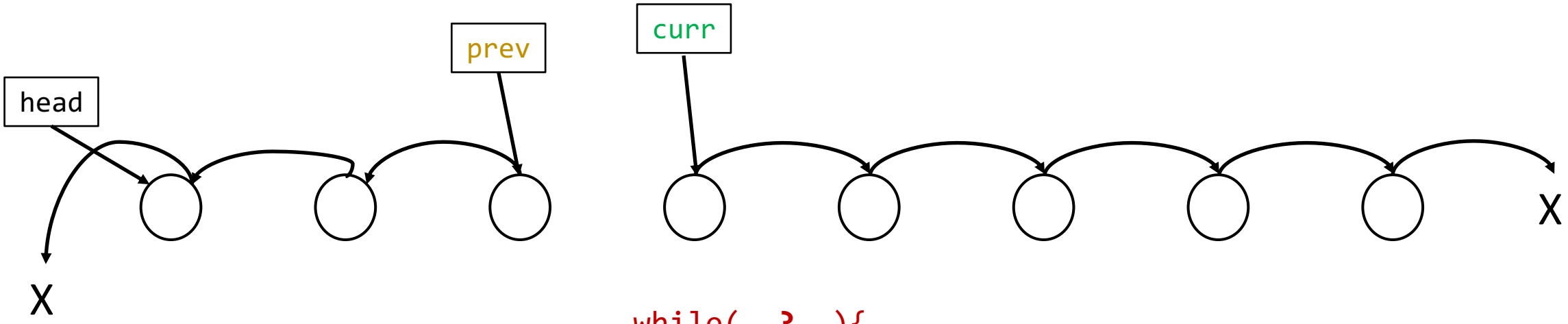
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

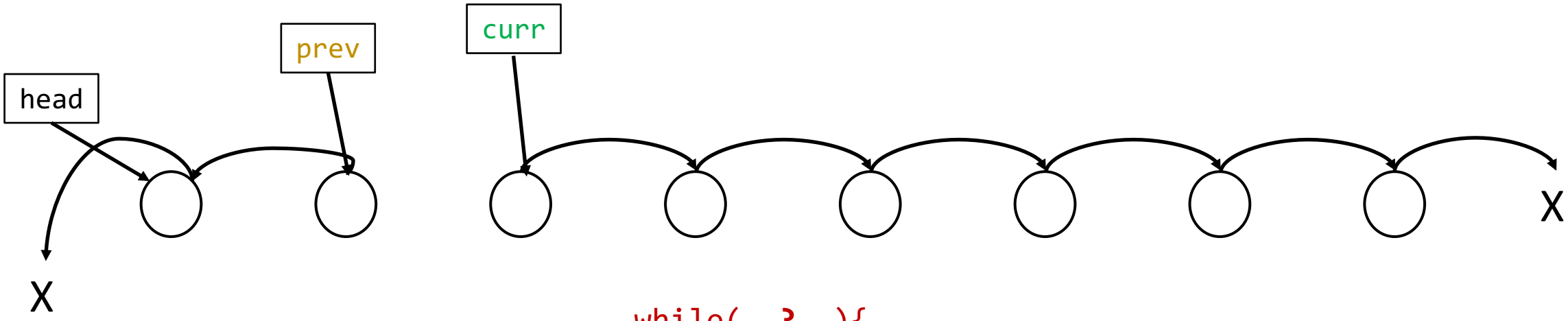
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

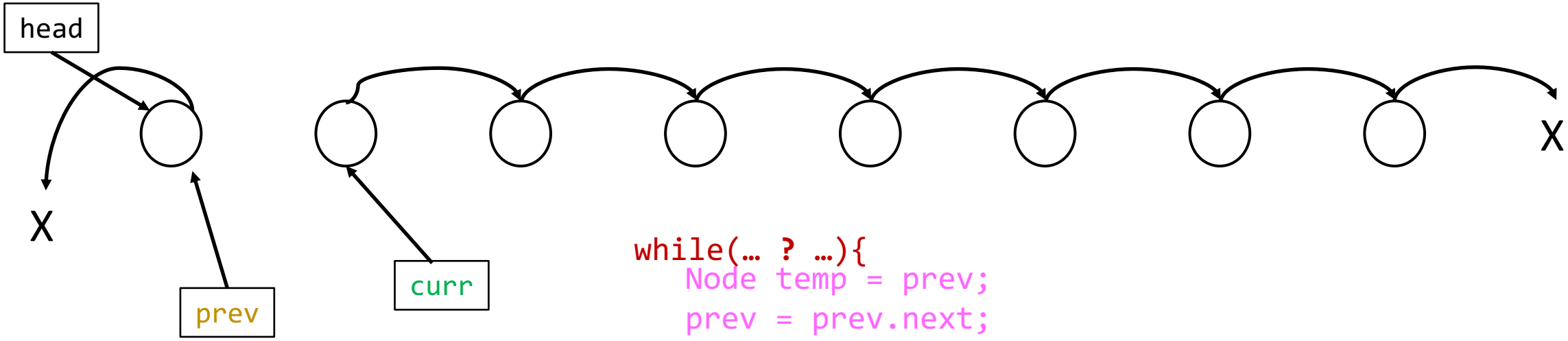
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

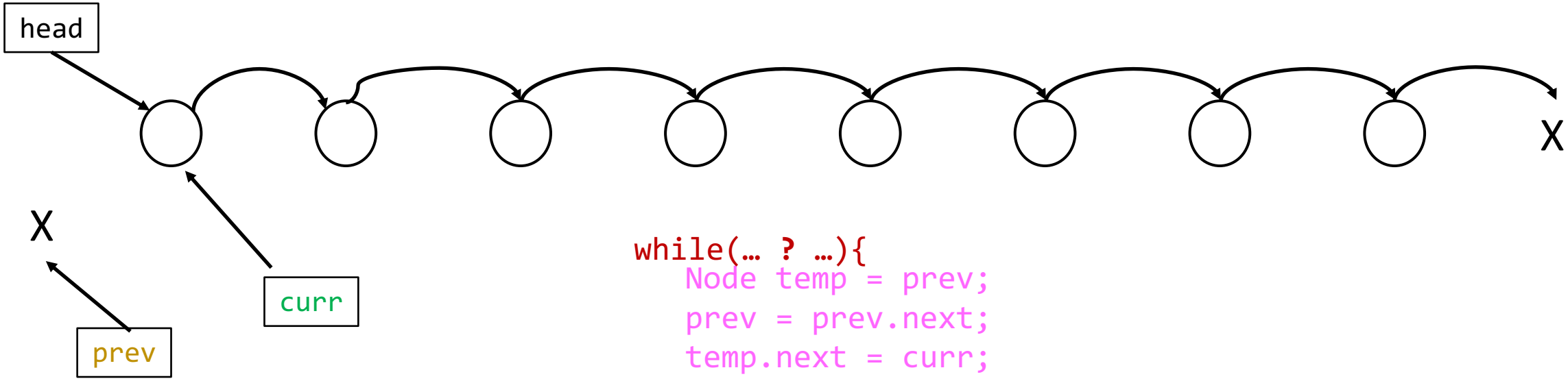
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

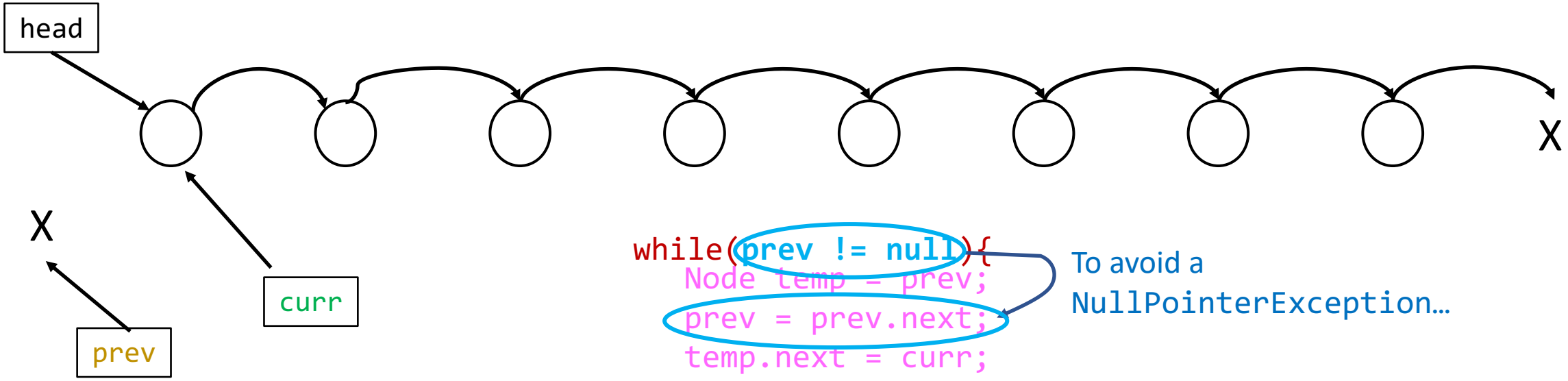
- The method uses two pointers, `prev` and `curr`.




```
while(… ? …){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

Link inversion for lists

- The method uses two pointers, `prev` and `curr`.



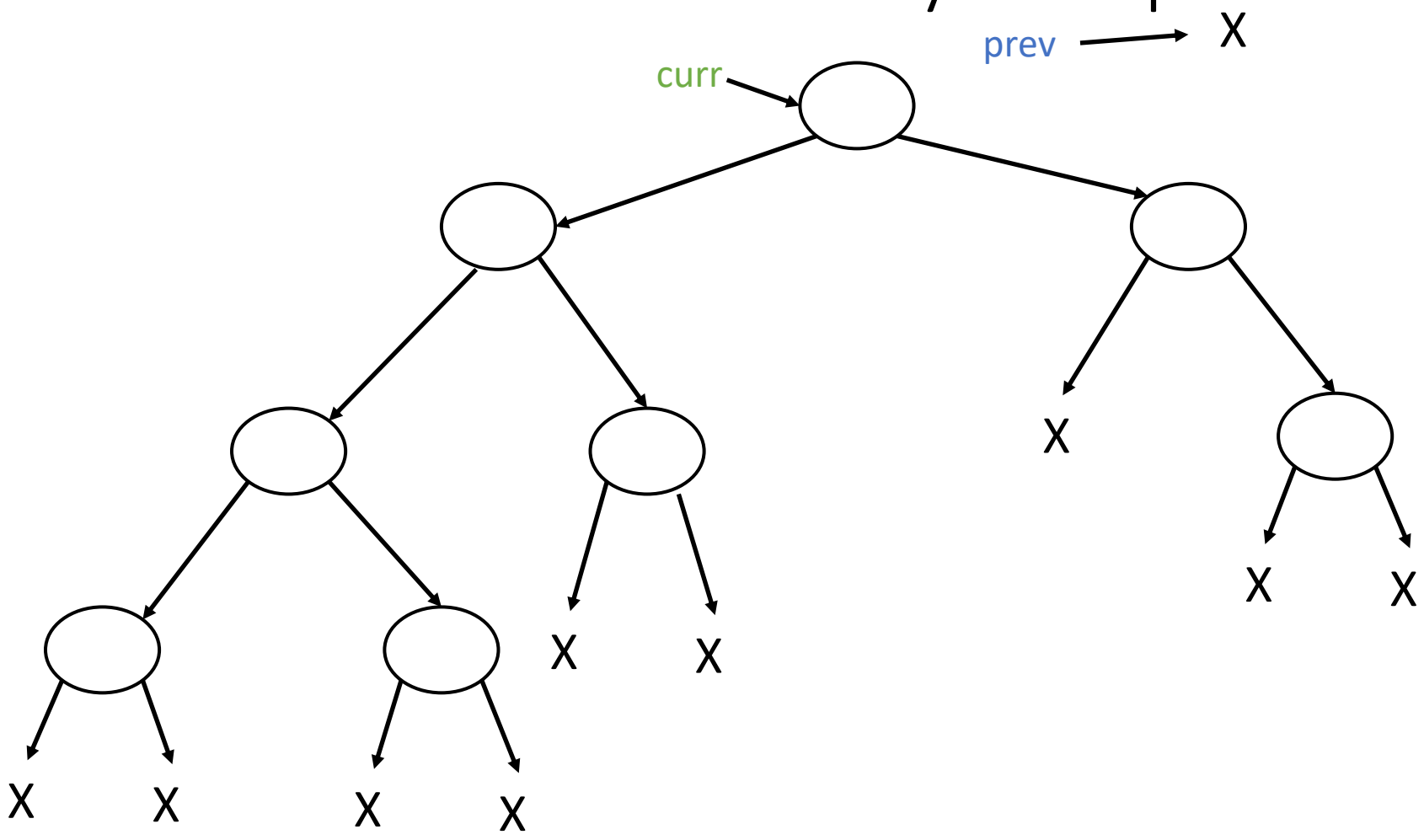

```
while(prev != null){
   Node temp = prev;
   prev = prev.next;
   temp.next = curr;
   curr = temp;
}
```

To avoid a NullPointerException…

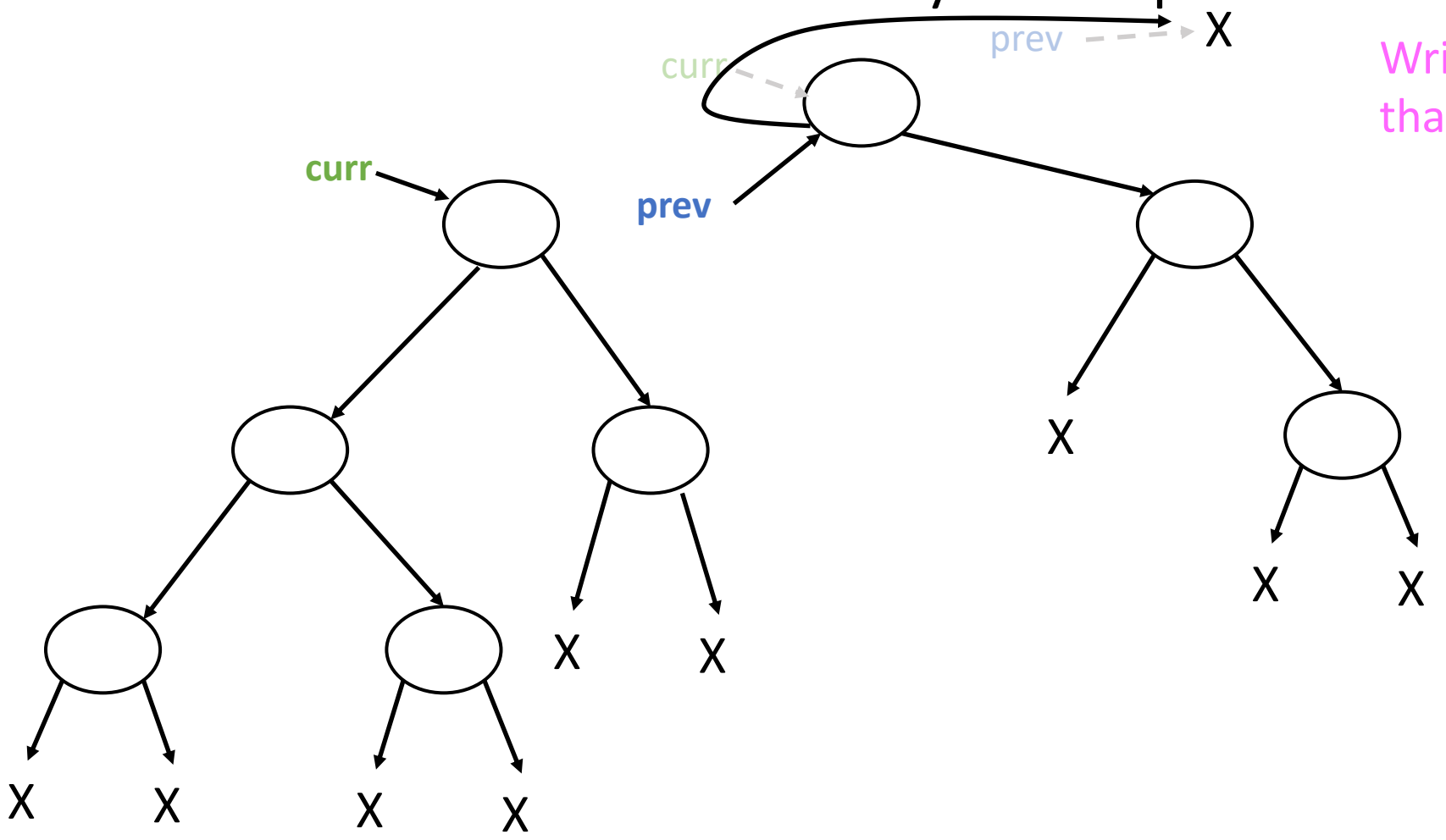
Link inversion for binary trees

- Now suppose I want to do the same thing in a binary tree.
- In effect, let's suppose that I want to do preorder traversal in a binary tree using the method of link inversion.
- Clearly, what I want to do first is go as far to the left as I can, while I print the nodes' values.
- The method will once again use two pointers: prev and curr.

Link inversion for binary tree preorder traversal

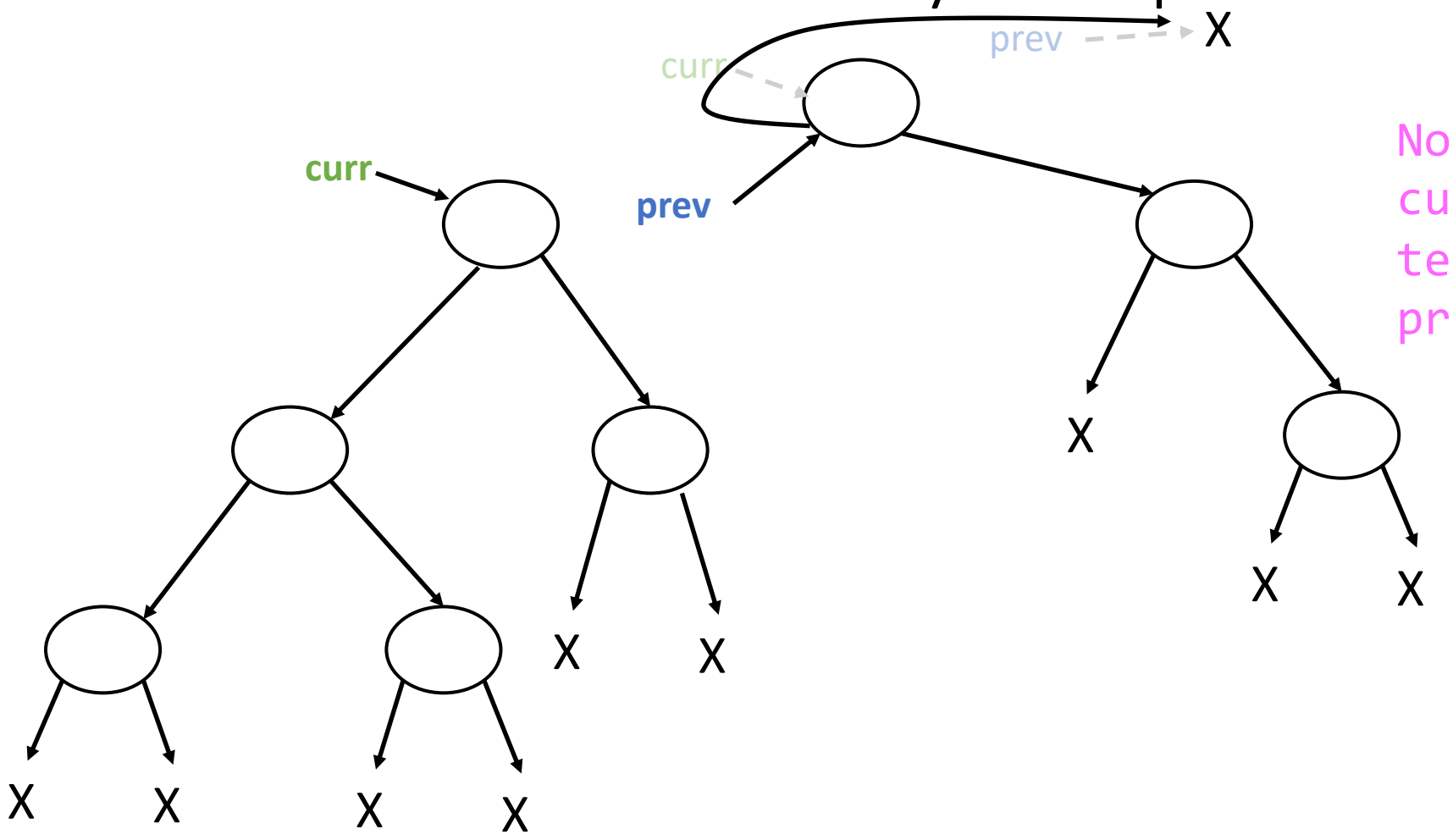


Link inversion for binary tree preorder traversal



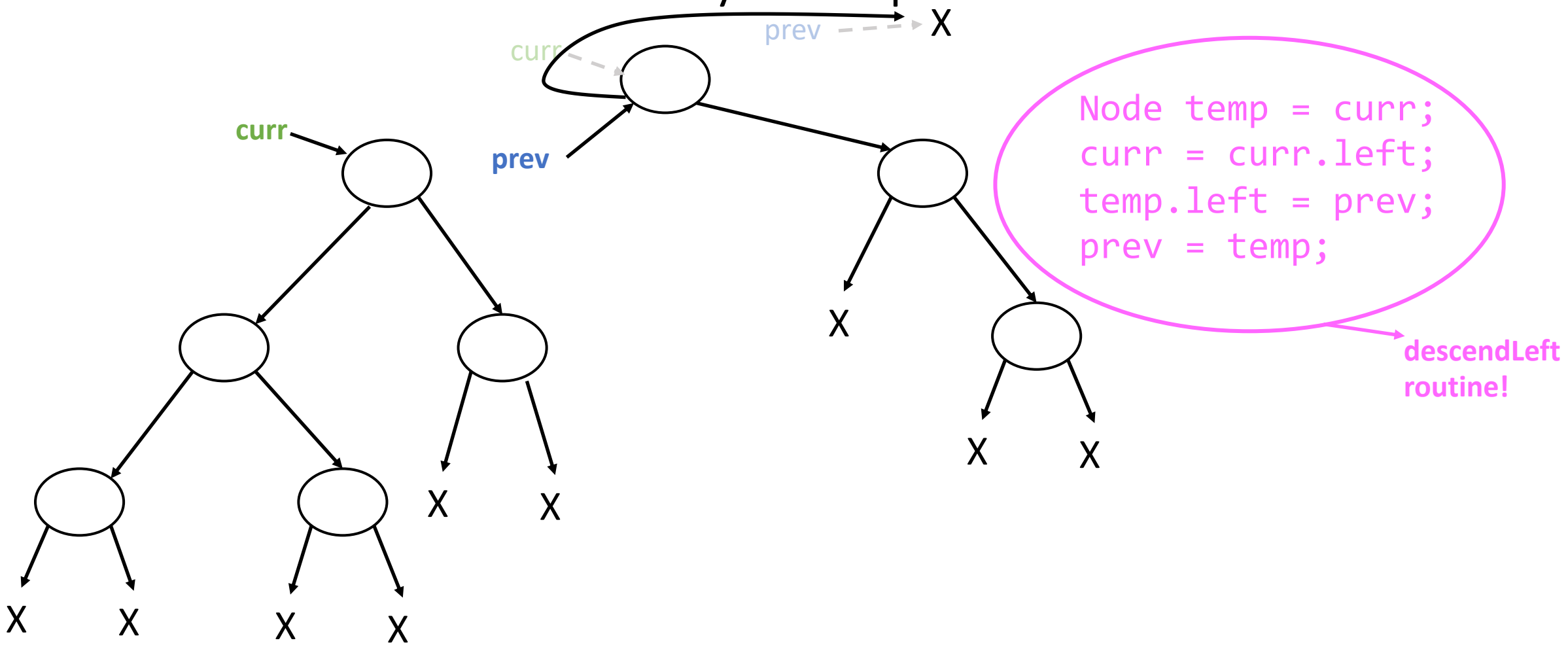
Write a Java code snippet that attains this move!

Link inversion for binary tree preorder traversal

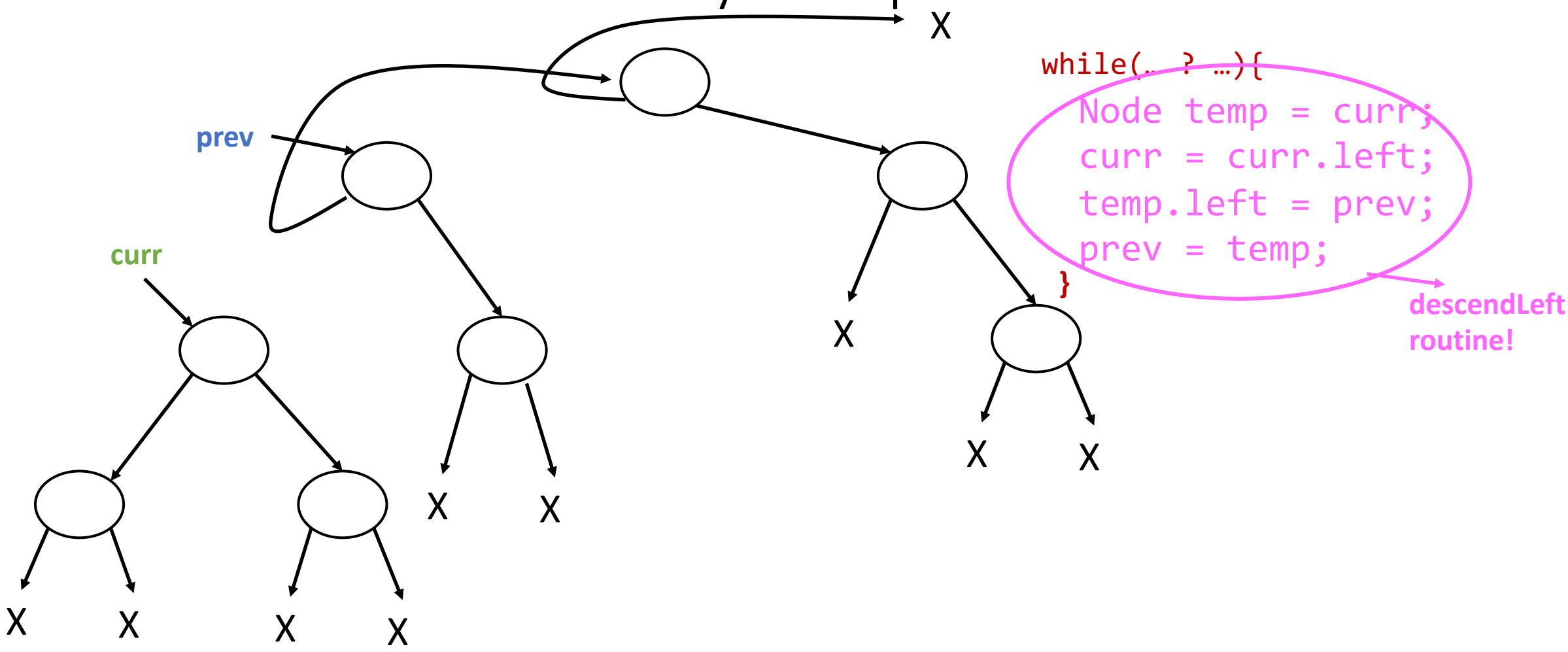



```
Node temp = curr;
curr = curr.left;
temp.left = prev;
prev = temp;
```

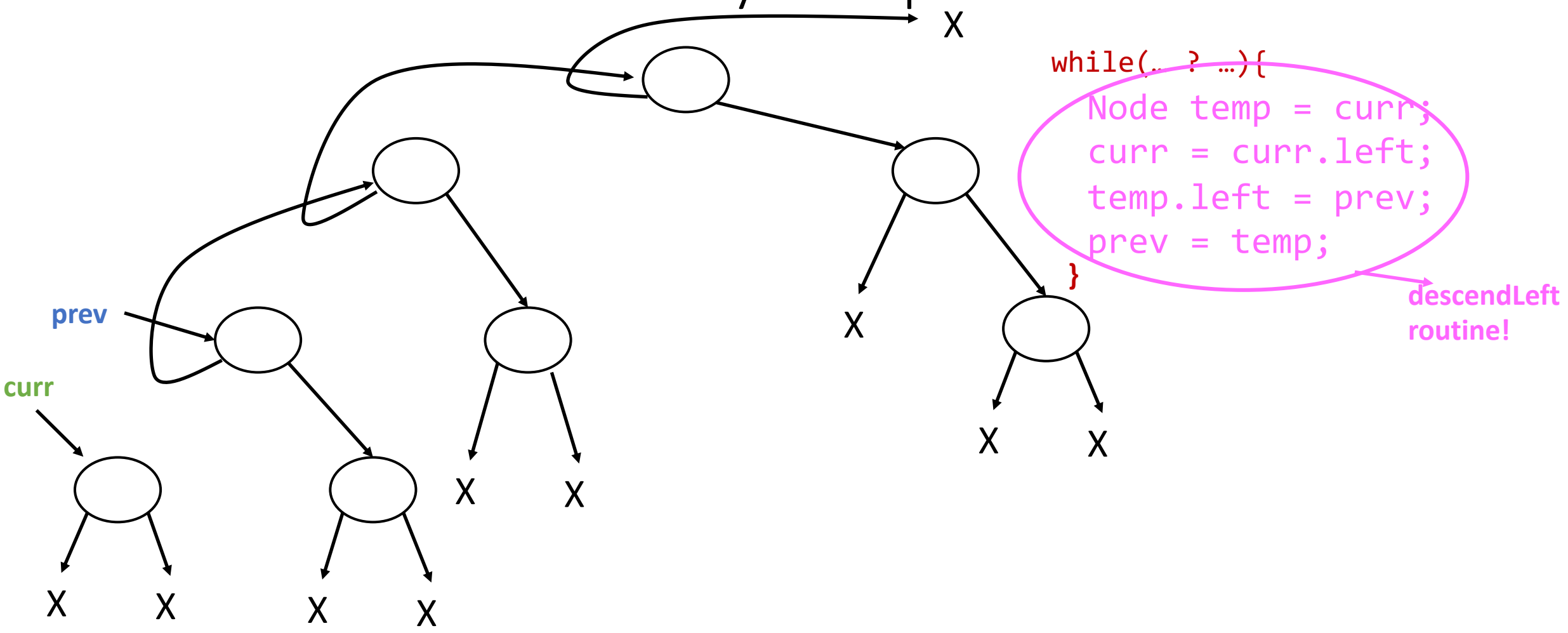
Link inversion for binary tree preorder traversal



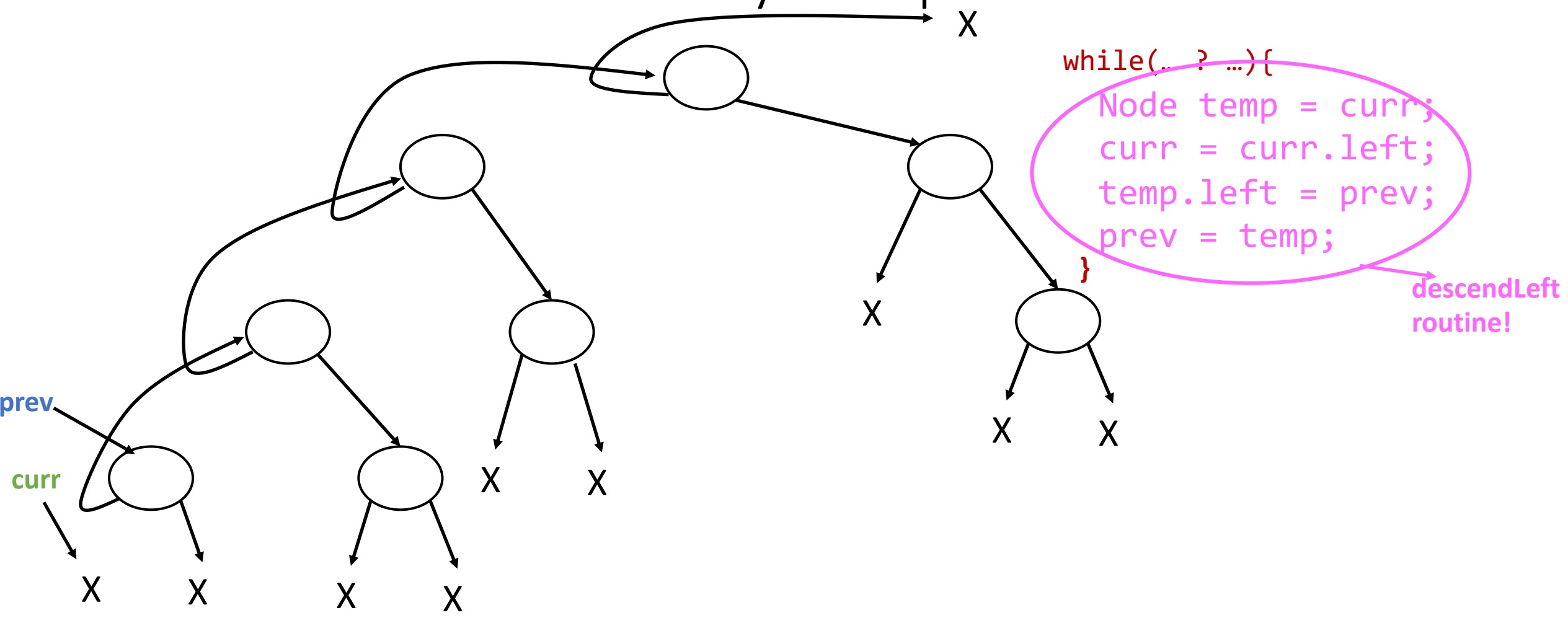
Link inversion for binary tree preorder traversal



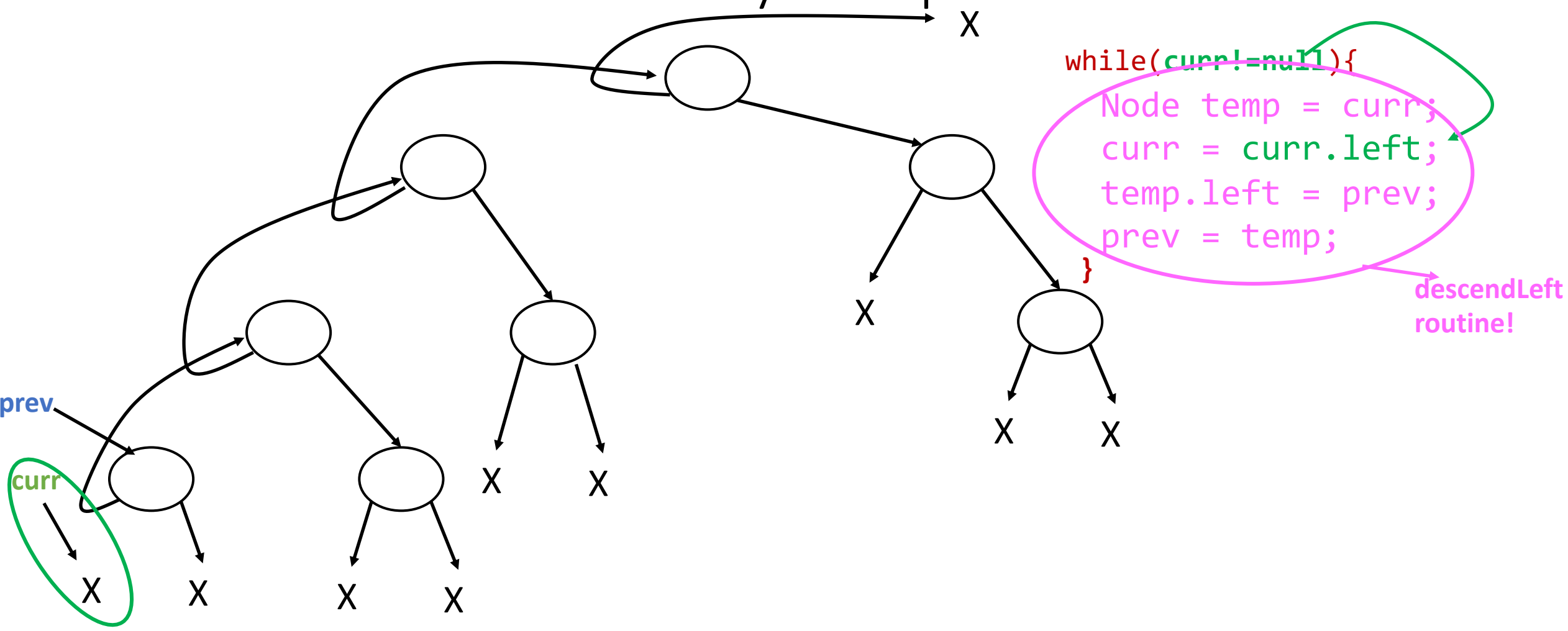
Link inversion for binary tree preorder traversal



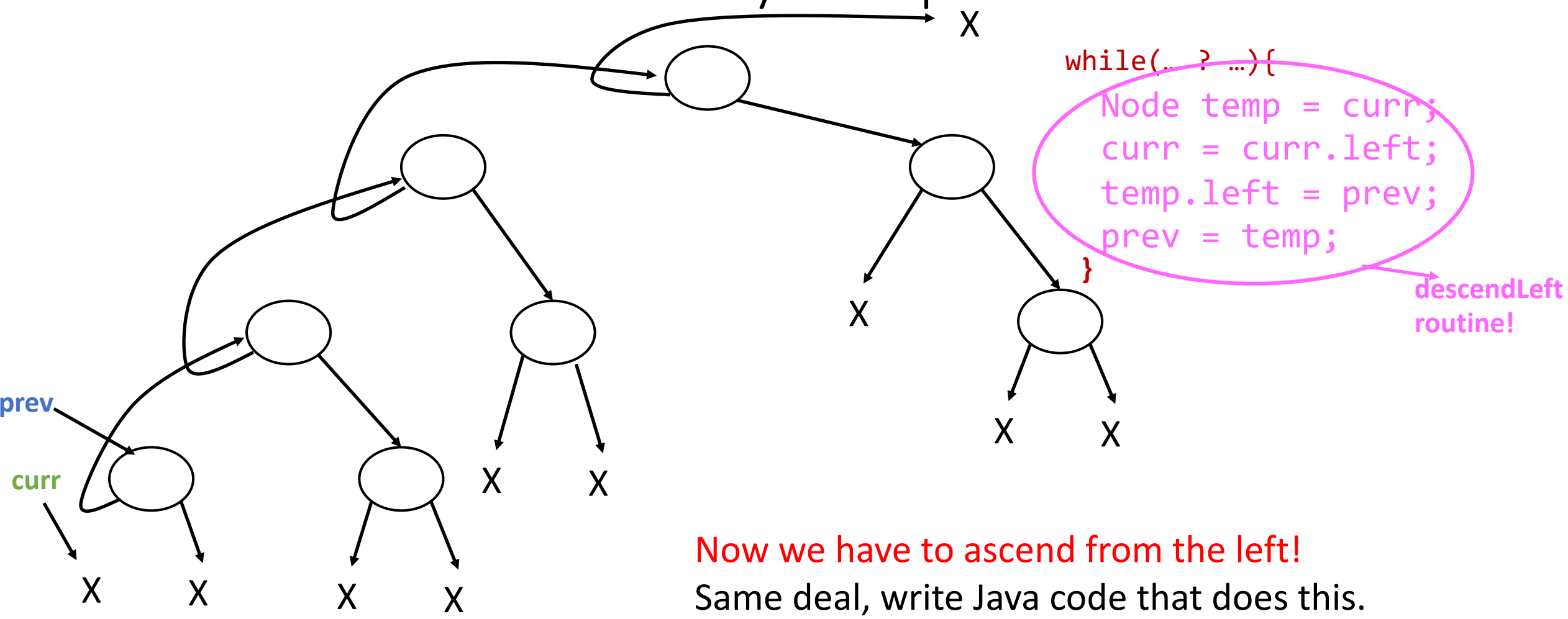
Link inversion for binary tree preorder traversal



Link inversion for binary tree preorder traversal



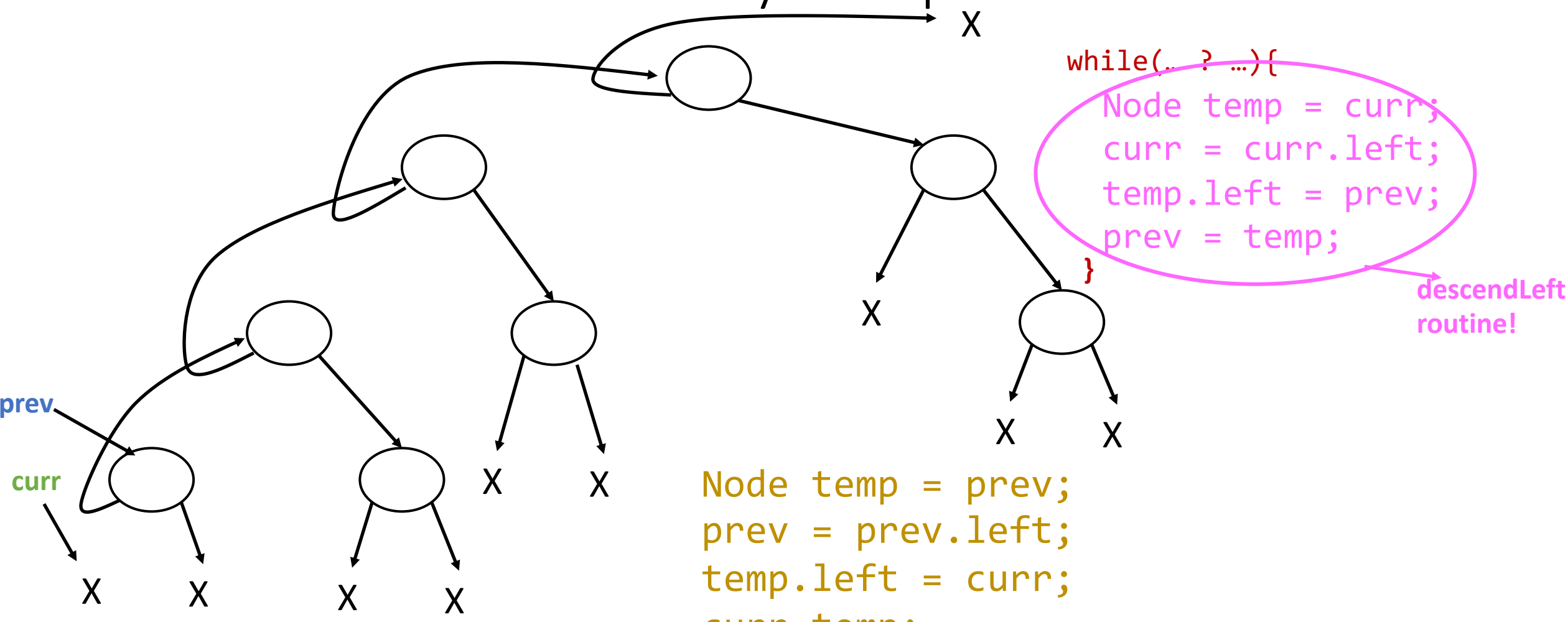
Link inversion for binary tree preorder traversal



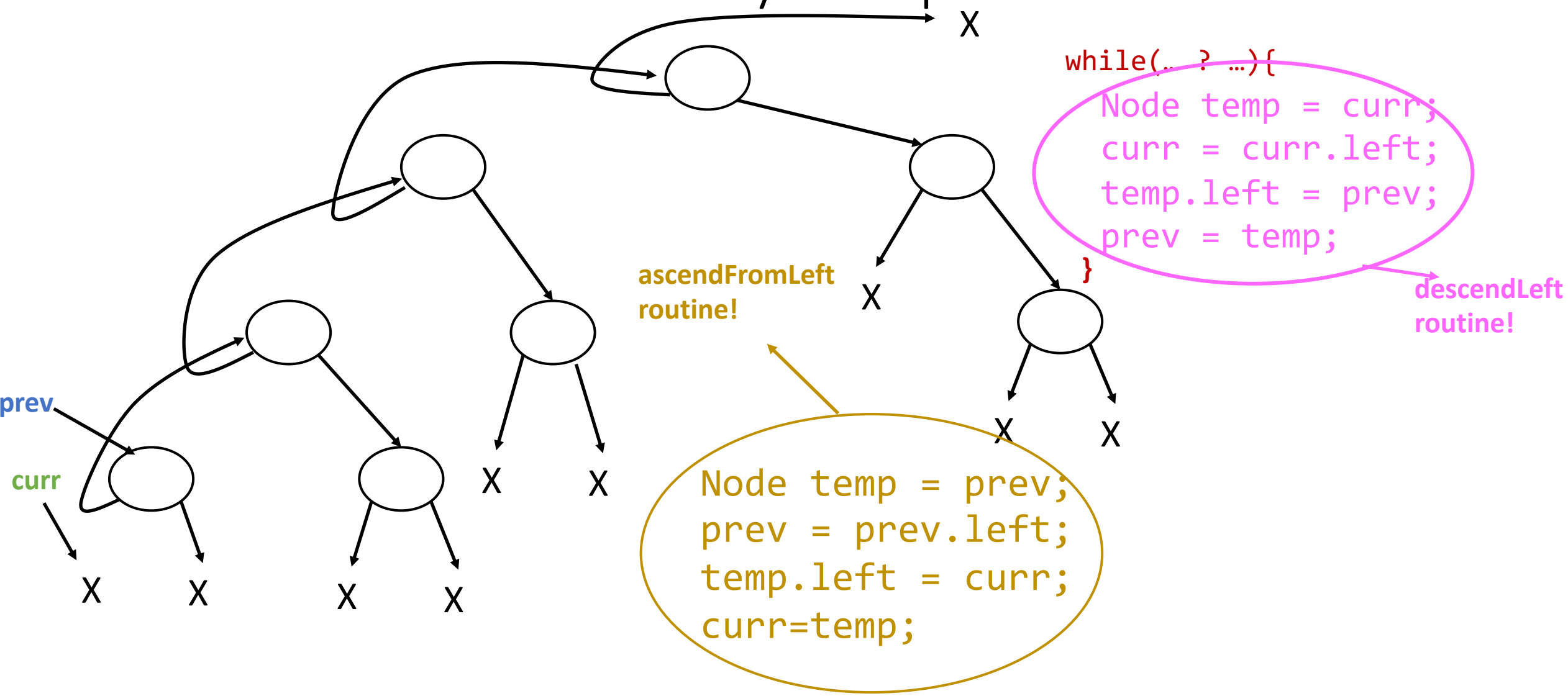
Now we have to ascend from the left!

Same deal, write Java code that does this.

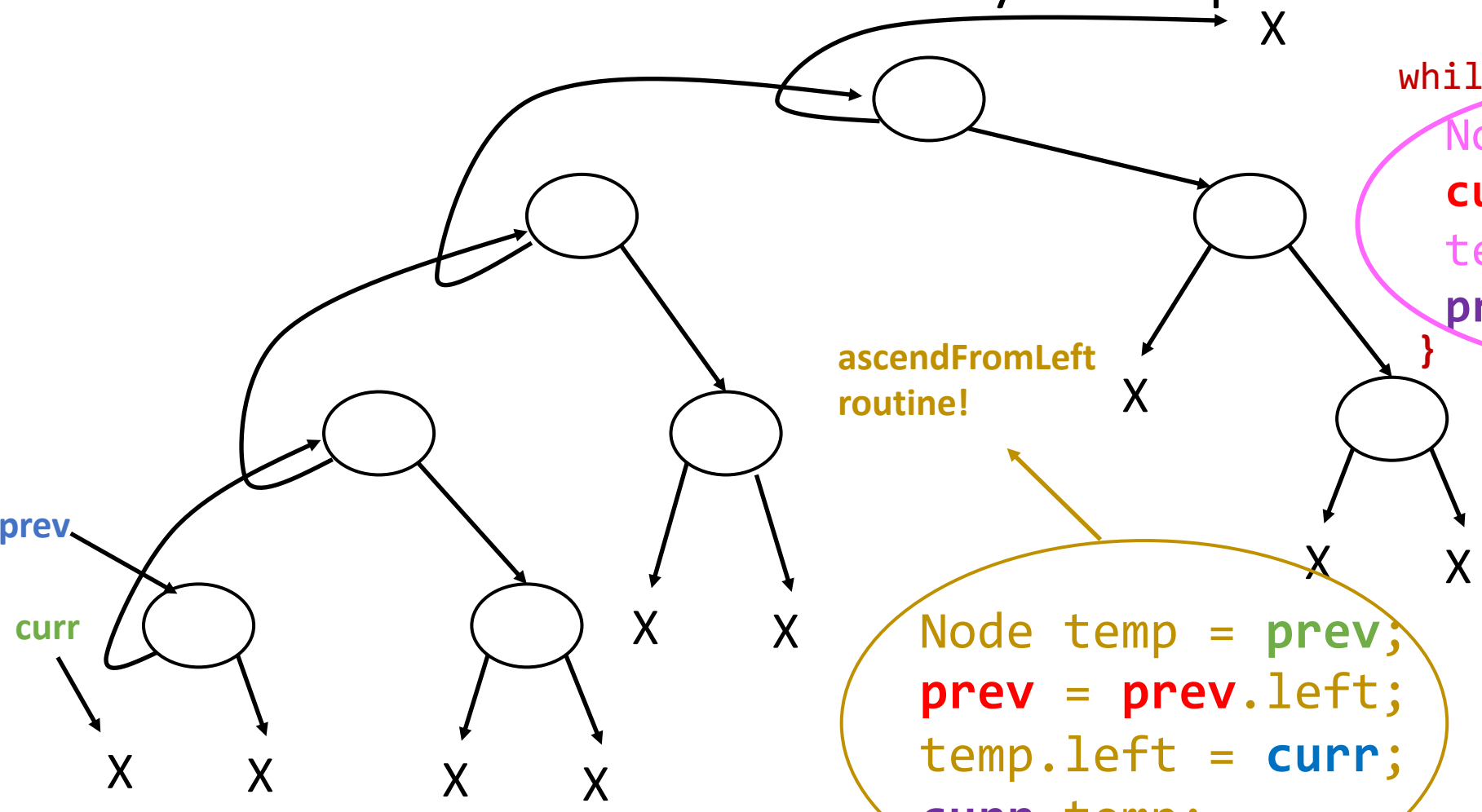
Link inversion for binary tree preorder traversal



Link inversion for binary tree preorder traversal



Link inversion for binary tree preorder traversal



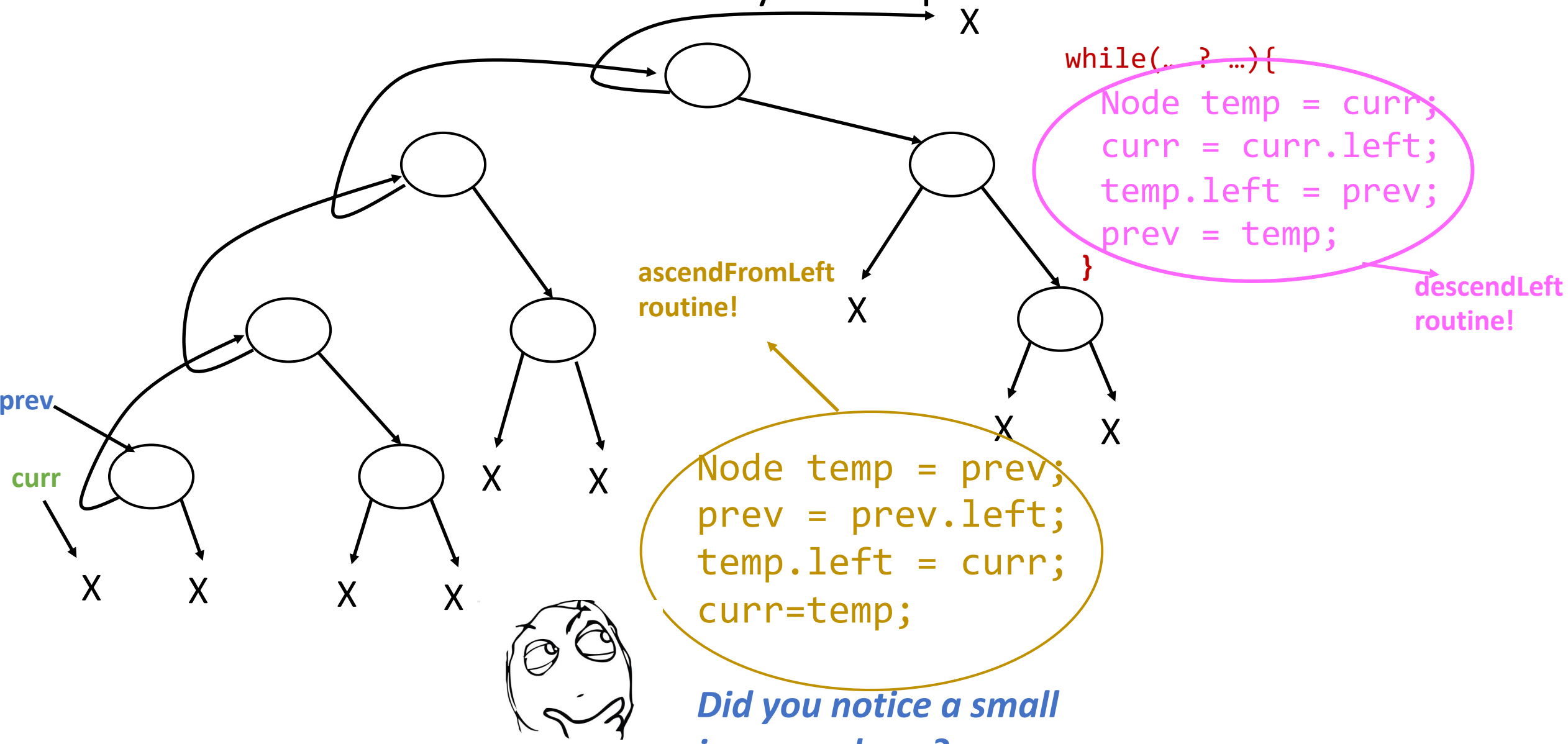

```
while(… ? …){
  Node temp = curr;
  curr = curr.left;
  temp.left = prev;
  prev = temp;
}
```

descendLeft routine!

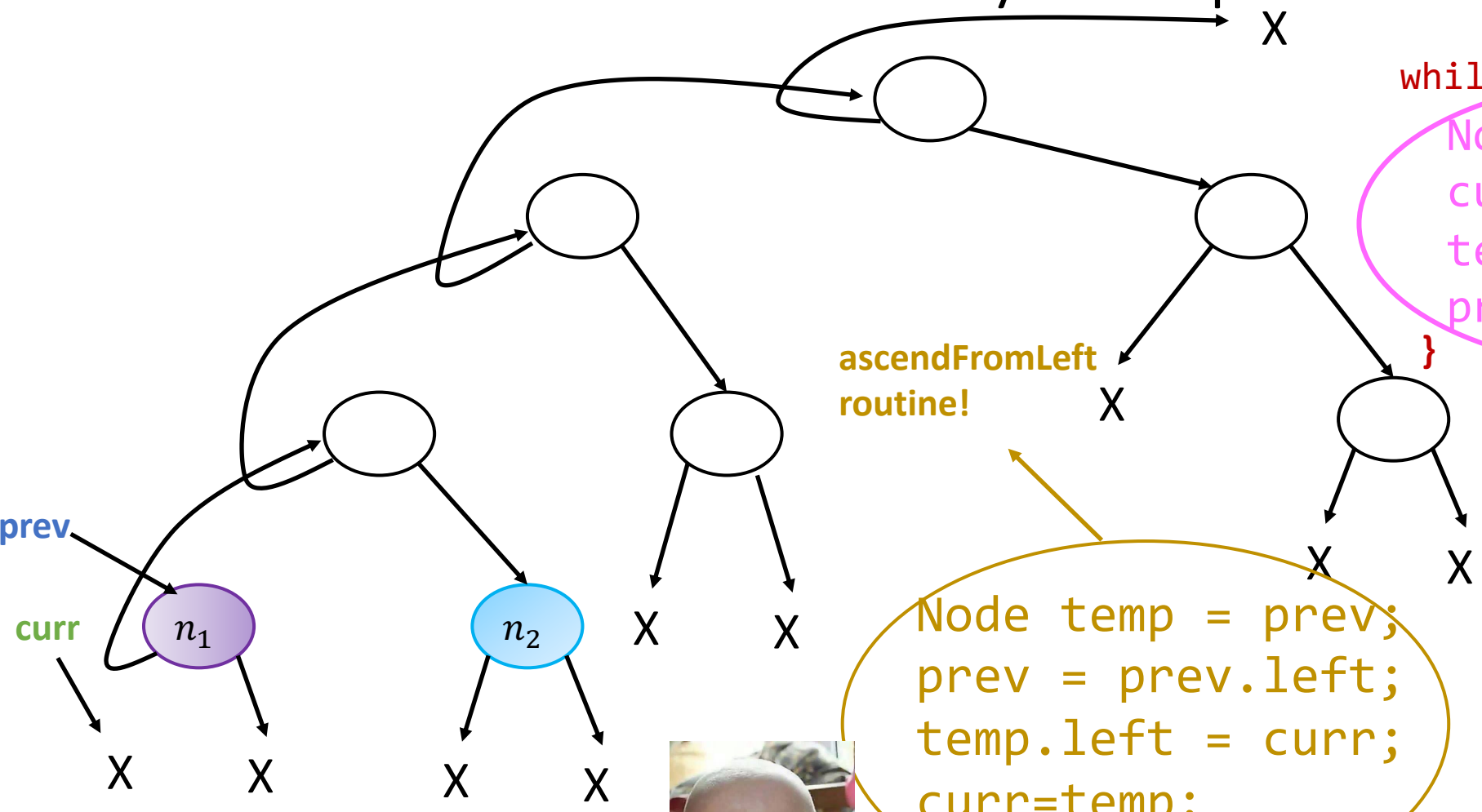
ascendFromLeft routine!

```
Node temp = prev;
prev = prev.left;
temp.left = curr;
curr=temp;
```

Link inversion for binary tree preorder traversal



Link inversion for binary tree preorder traversal




```
while(… ? …){
  Node temp = curr;
  curr = curr.left;
  temp.left = prev;
  prev = temp;
}
```

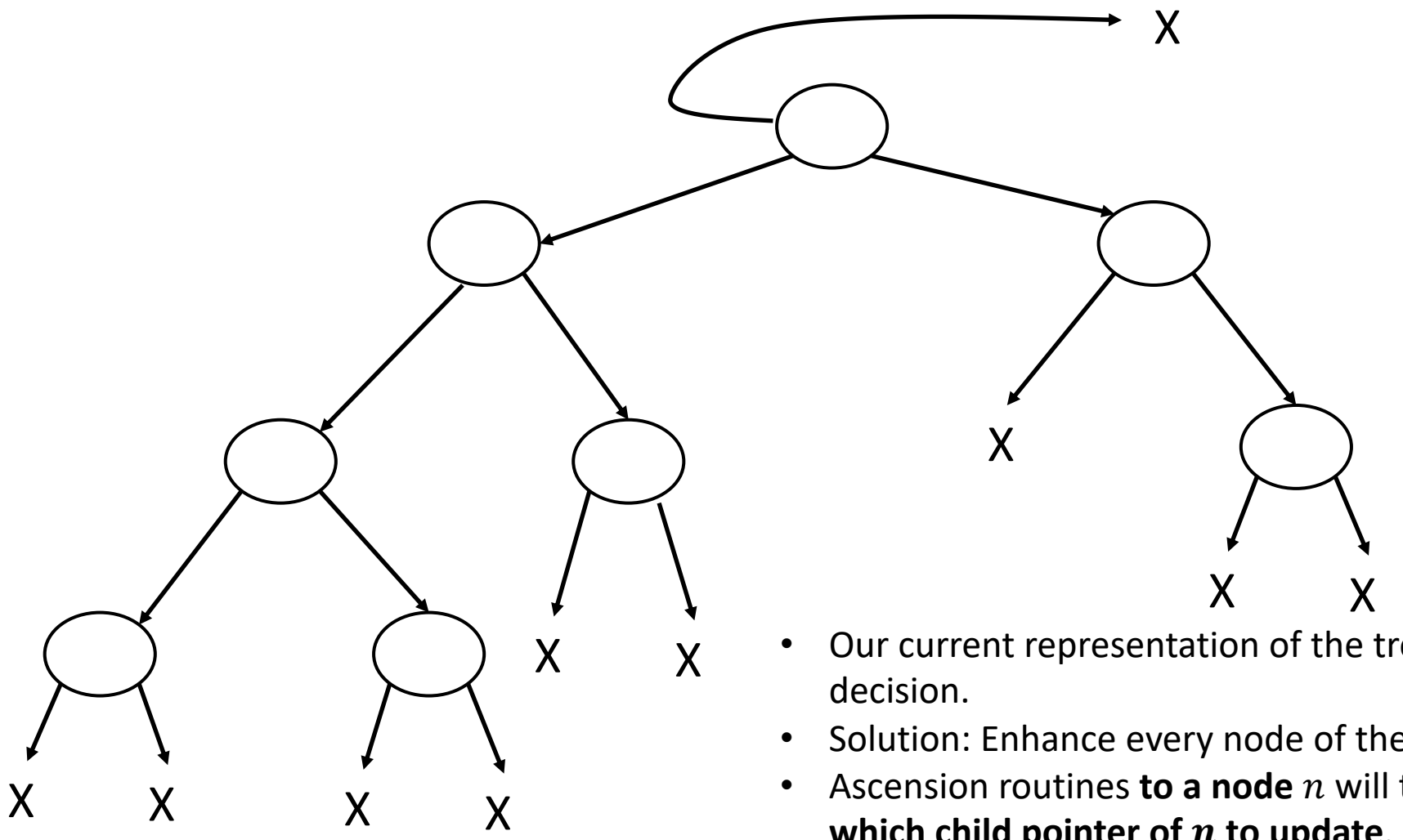
descendLeft routine!

ascendFromLeft routine!

```
Node temp = prev;
prev = prev.left;
temp.left = curr;
curr=temp;
```

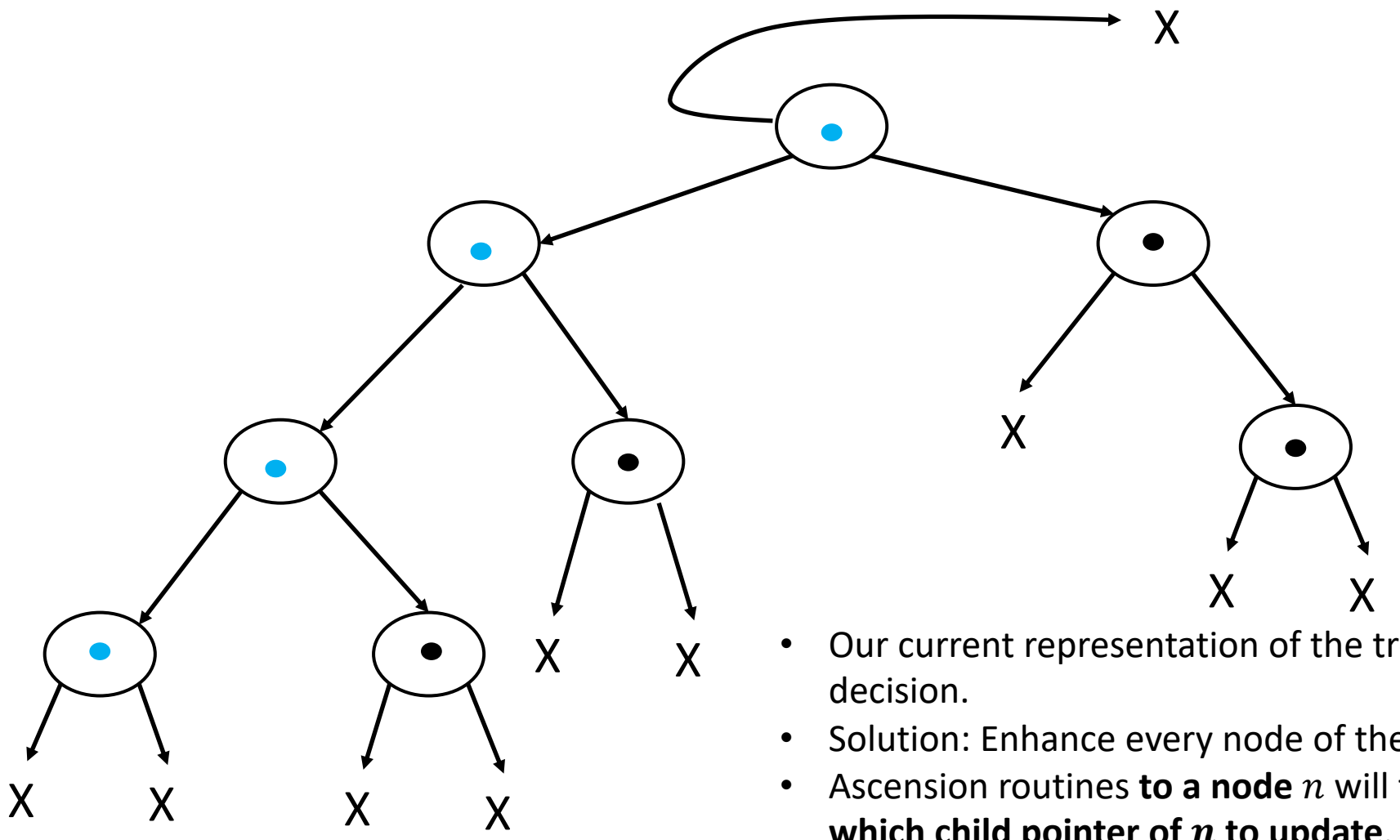
How did we know that we needed to ascend from the ***left*** (node n_1), and ***not*** the ***right***? (node n_2)

We didn't.



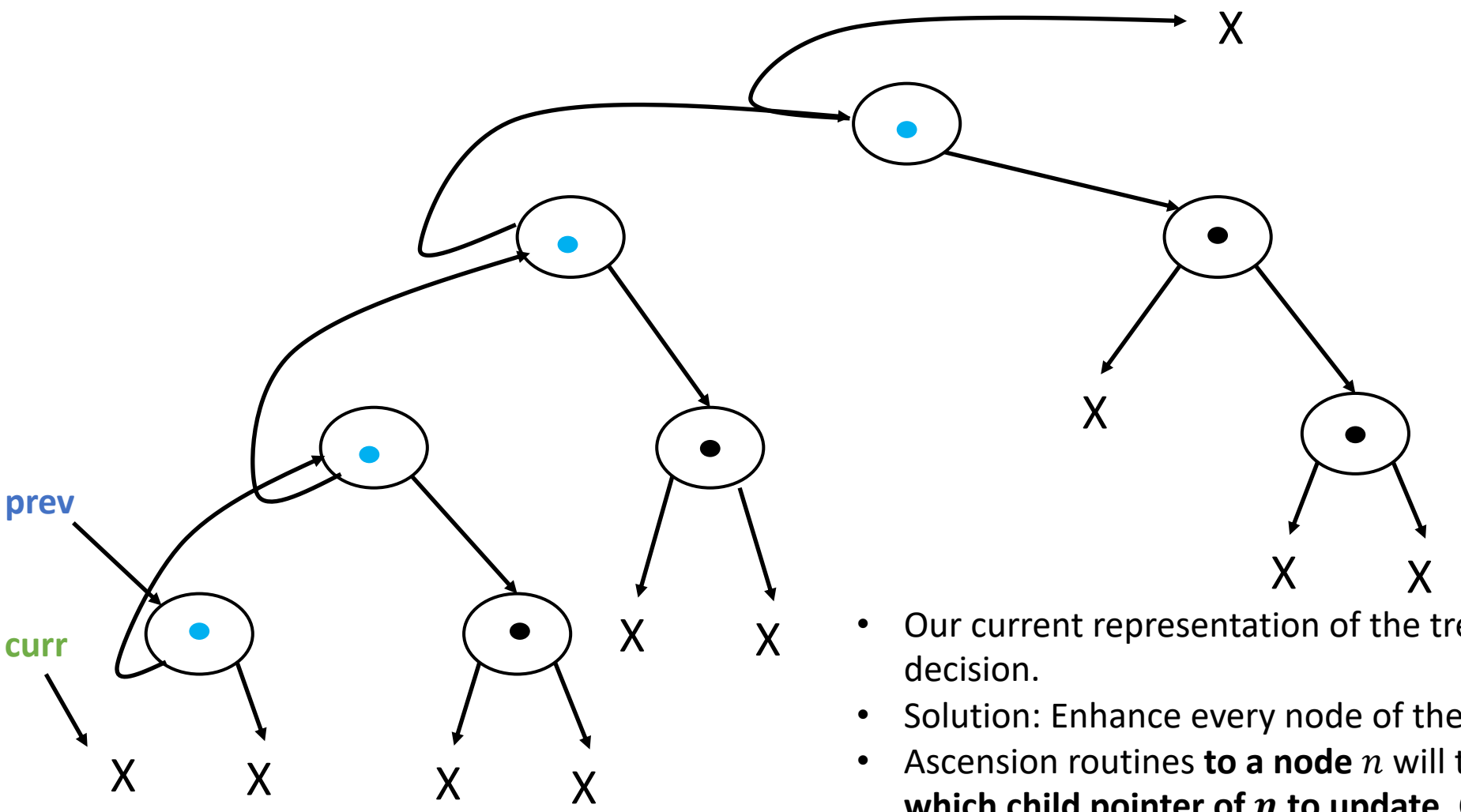
- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

We didn't.



- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

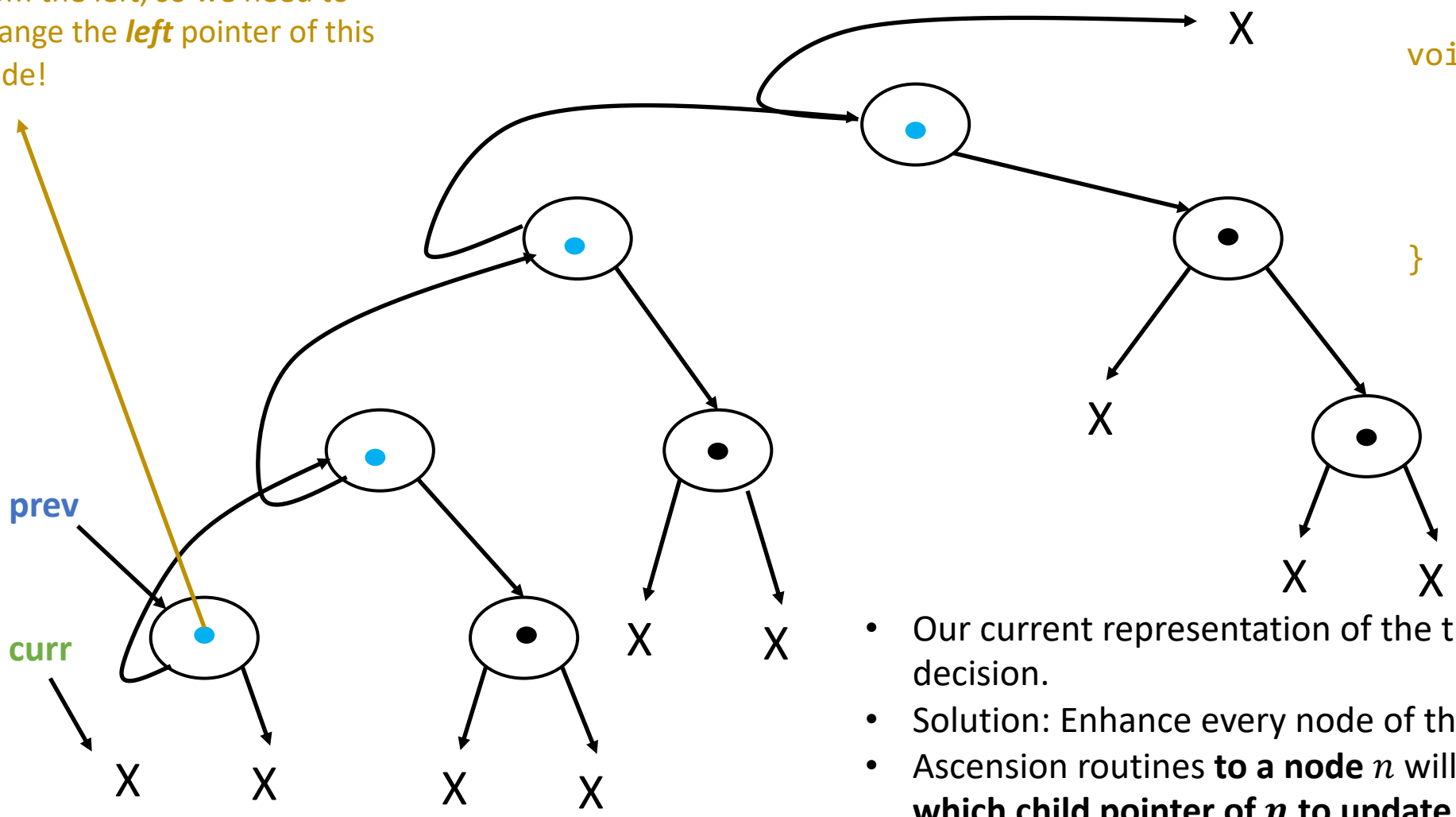
We didn't.



- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

But now we do!

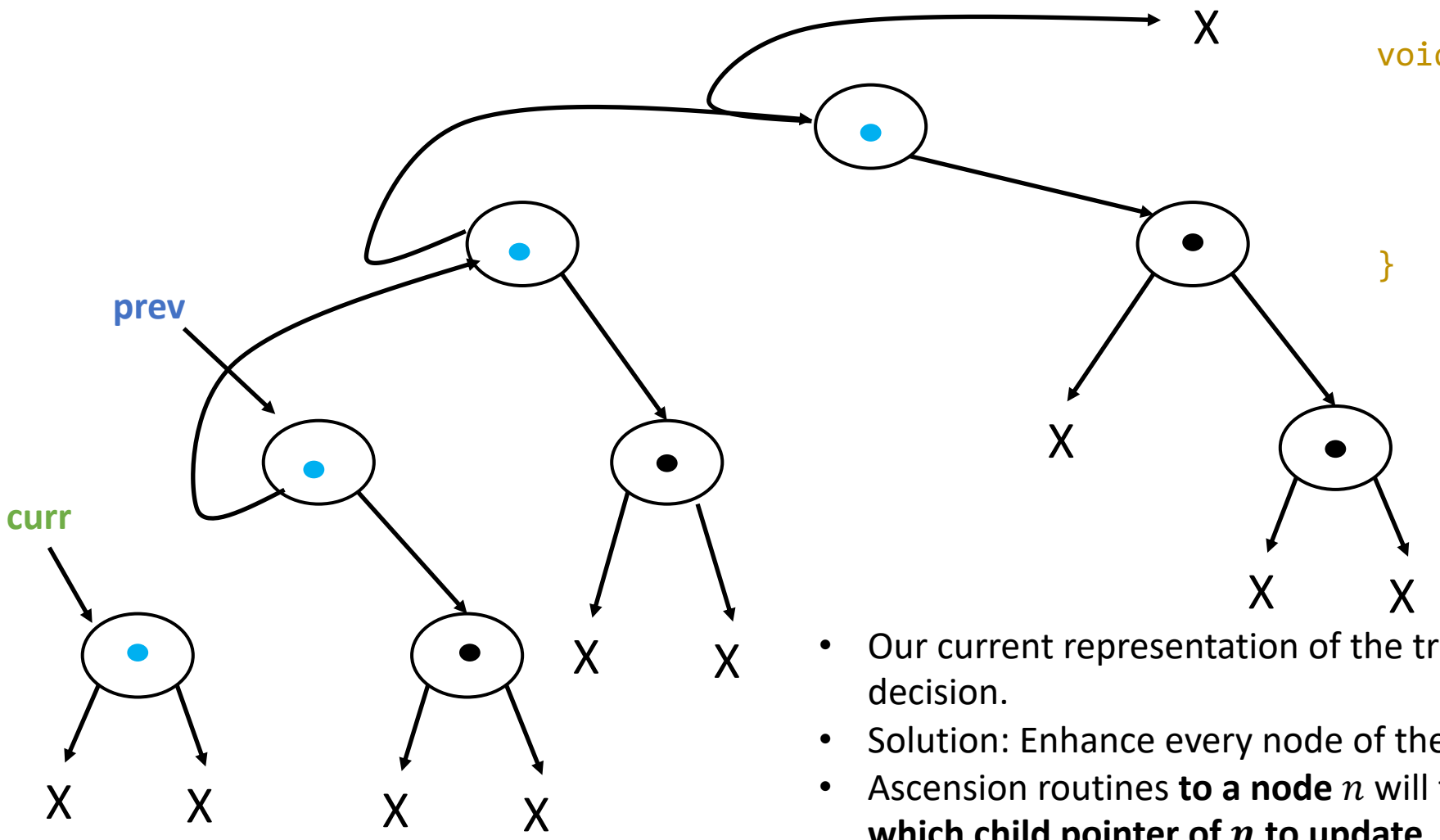
Now we know that we ascend from the left, so we need to change the ***left*** pointer of this node!




```
void ascendFromLeft(){
    Node temp = prev;
    prev = prev.left;
    temp.left = curr;
    curr=temp;
}
```

- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

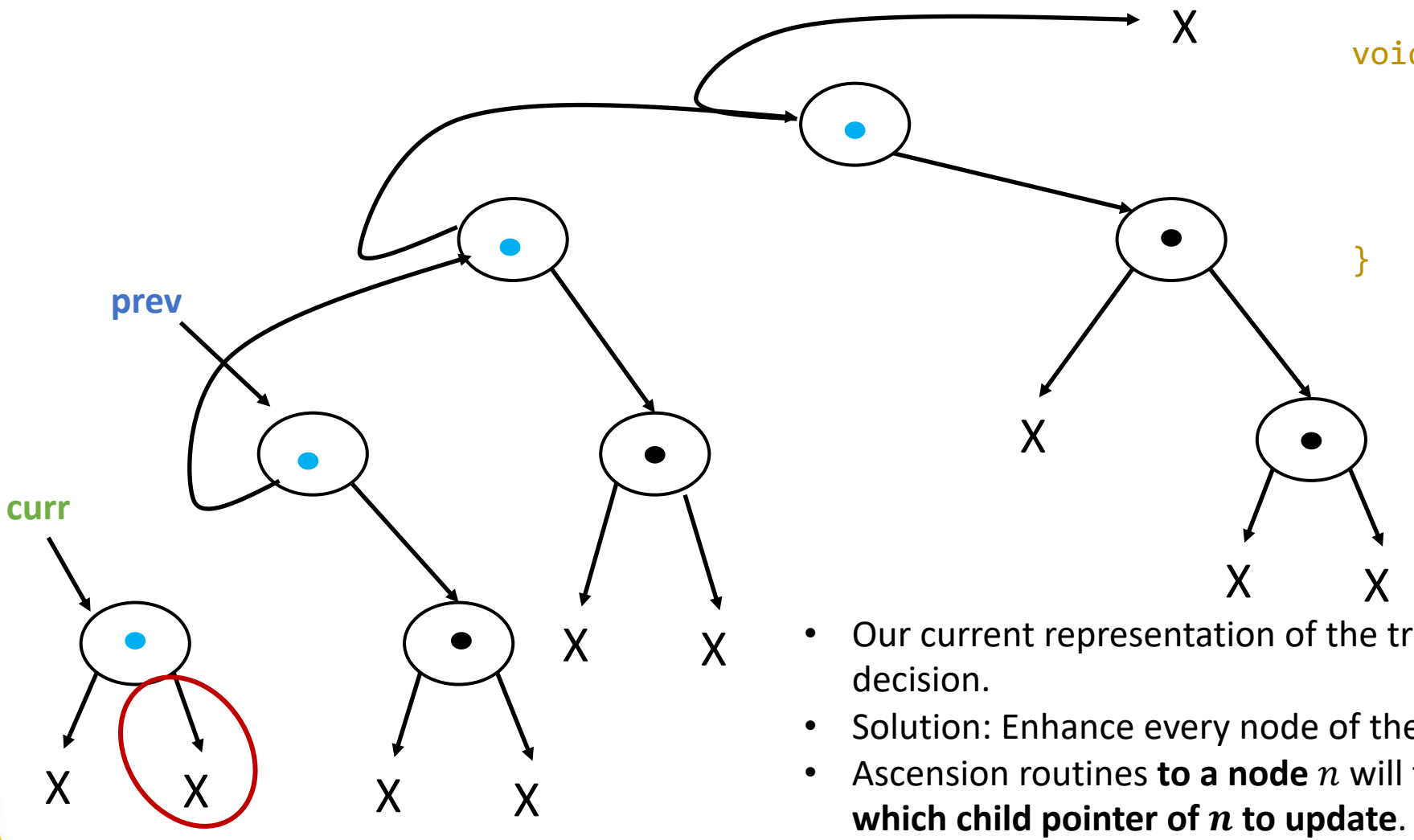
But now we do!




```
void ascendFromLeft(){
    Node temp = prev;
    prev = prev.left;
    temp.left = curr;
    curr=temp;
}
```

- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

But now we do!

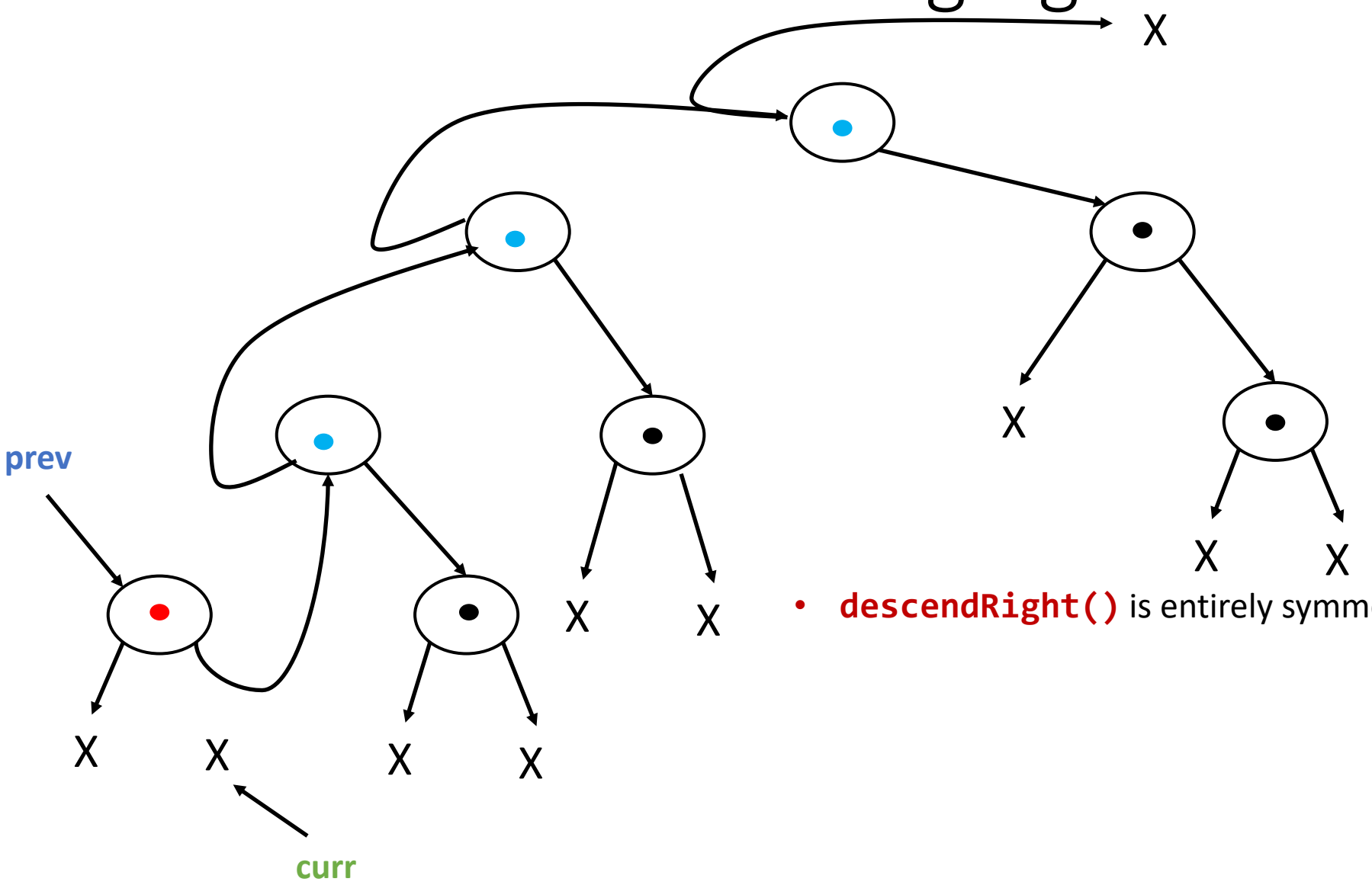



```
void ascendFromLeft(){
    Node temp = prev;
    prev = prev.left;
    temp.left = curr;
    curr=temp;
}
```

- Our current representation of the tree is **insufficient** to make this decision.
- Solution: Enhance every node of the tree **with a single bit** .
- Ascension routines **to a node** n will then check $n.bit$ to find out **which child pointer of n to update**. Conventions:
  - Bit unset (0): Ascend from Left.
  - Bit set (1): Ascend from right.

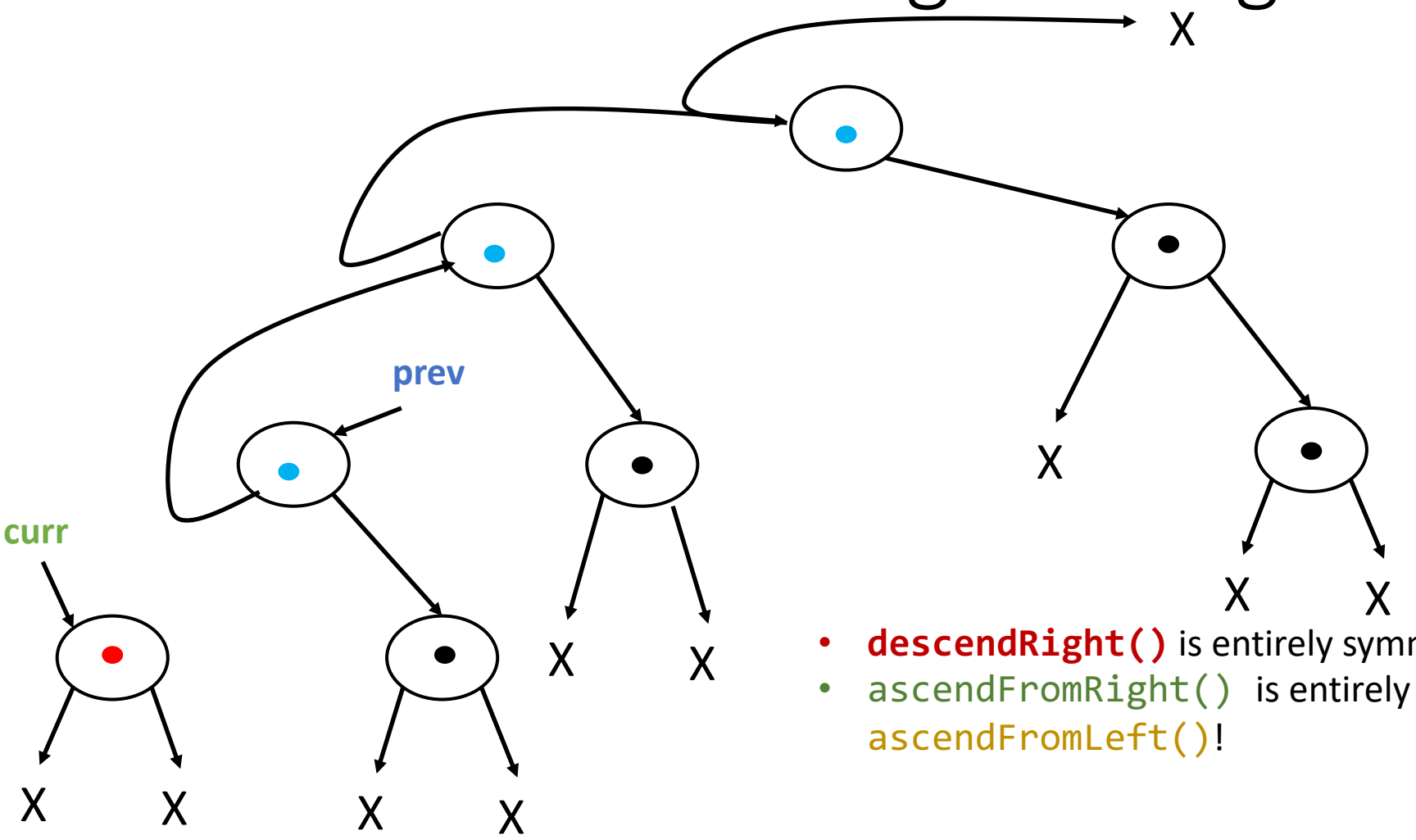
- **But don't forget this is preorder traversal! So we need to descend right!**

Descending right…



- **`descendRight()`** is entirely symmetrical to `descendLeft()`.

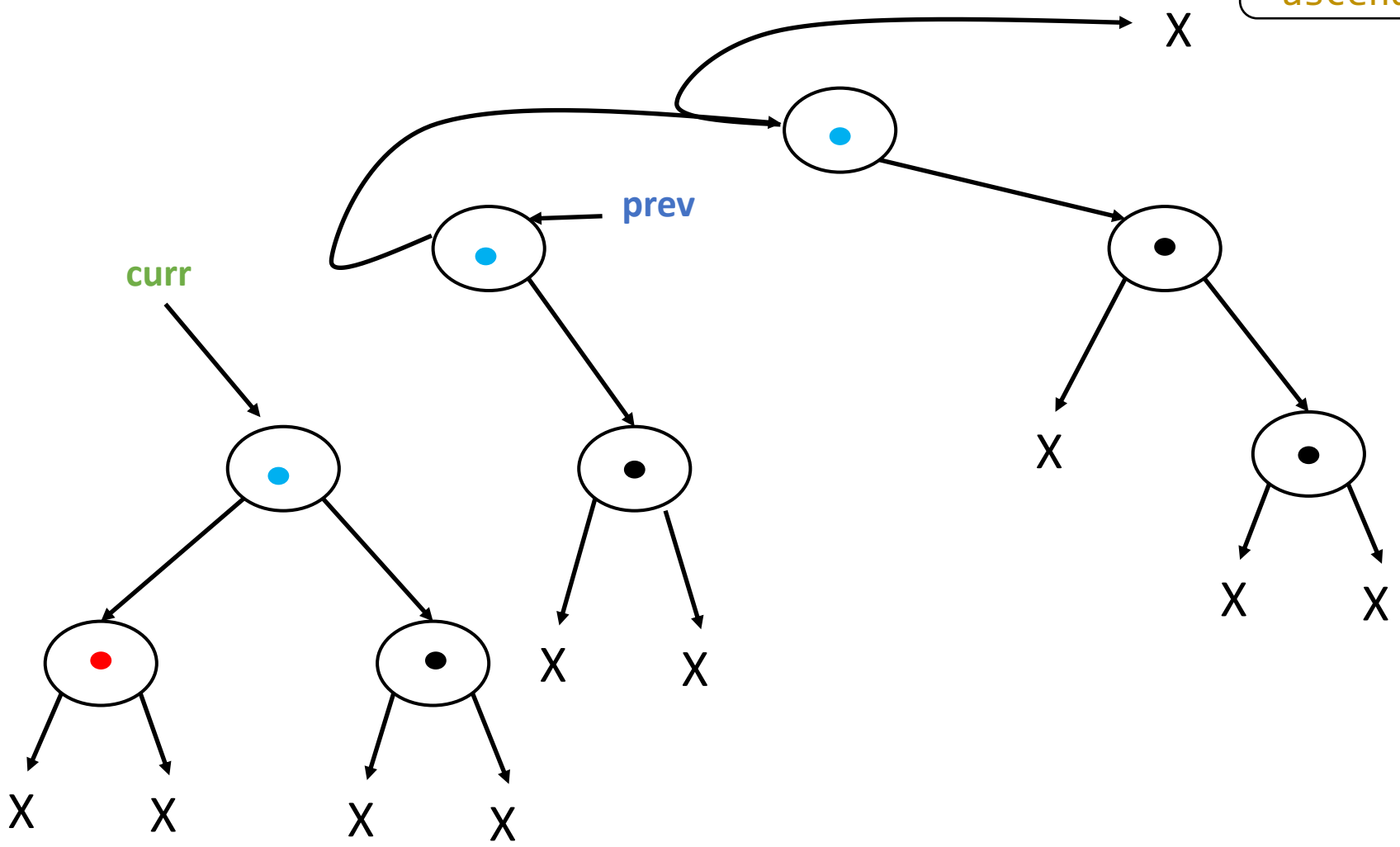
And ascending from right!



- **`descendRight()`** is entirely symmetrical to `descendLeft()`.
- `ascendFromRight()` is entirely symmetrical to `ascendFromLeft()`!

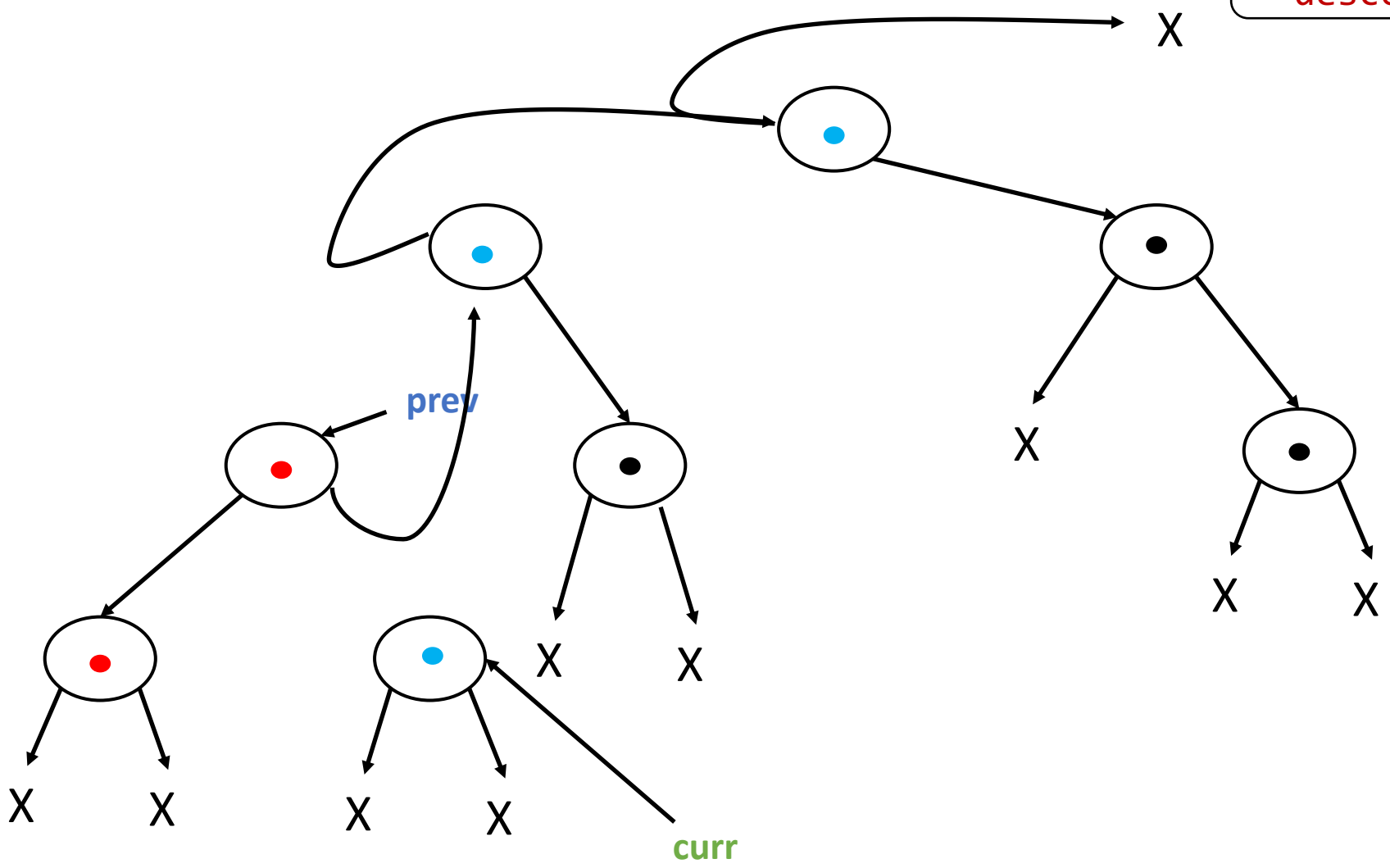
Traversal...

Routine used:
`ascendFromleft()`



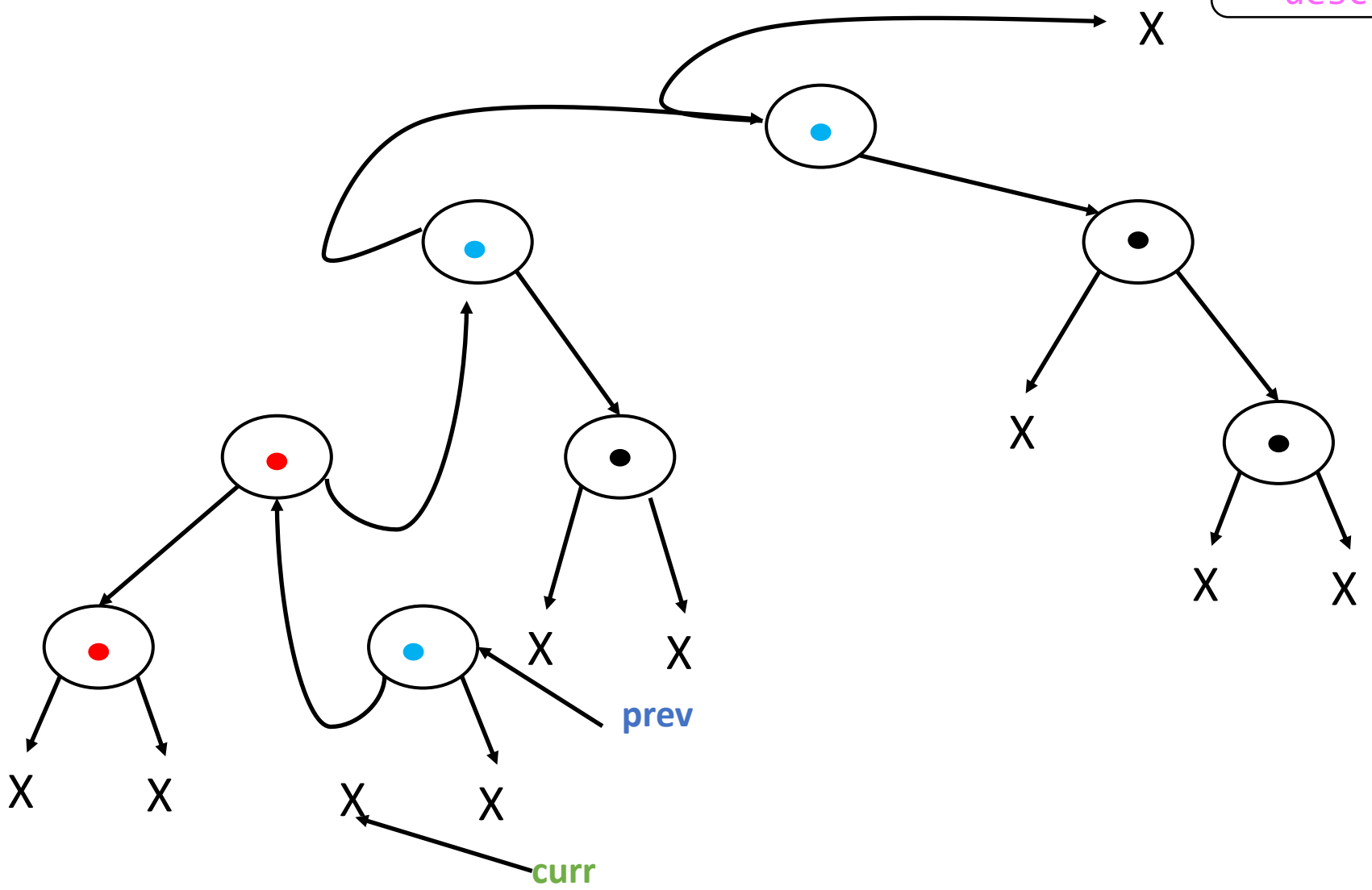
Traversal…

Routine used:
`descendRight()`



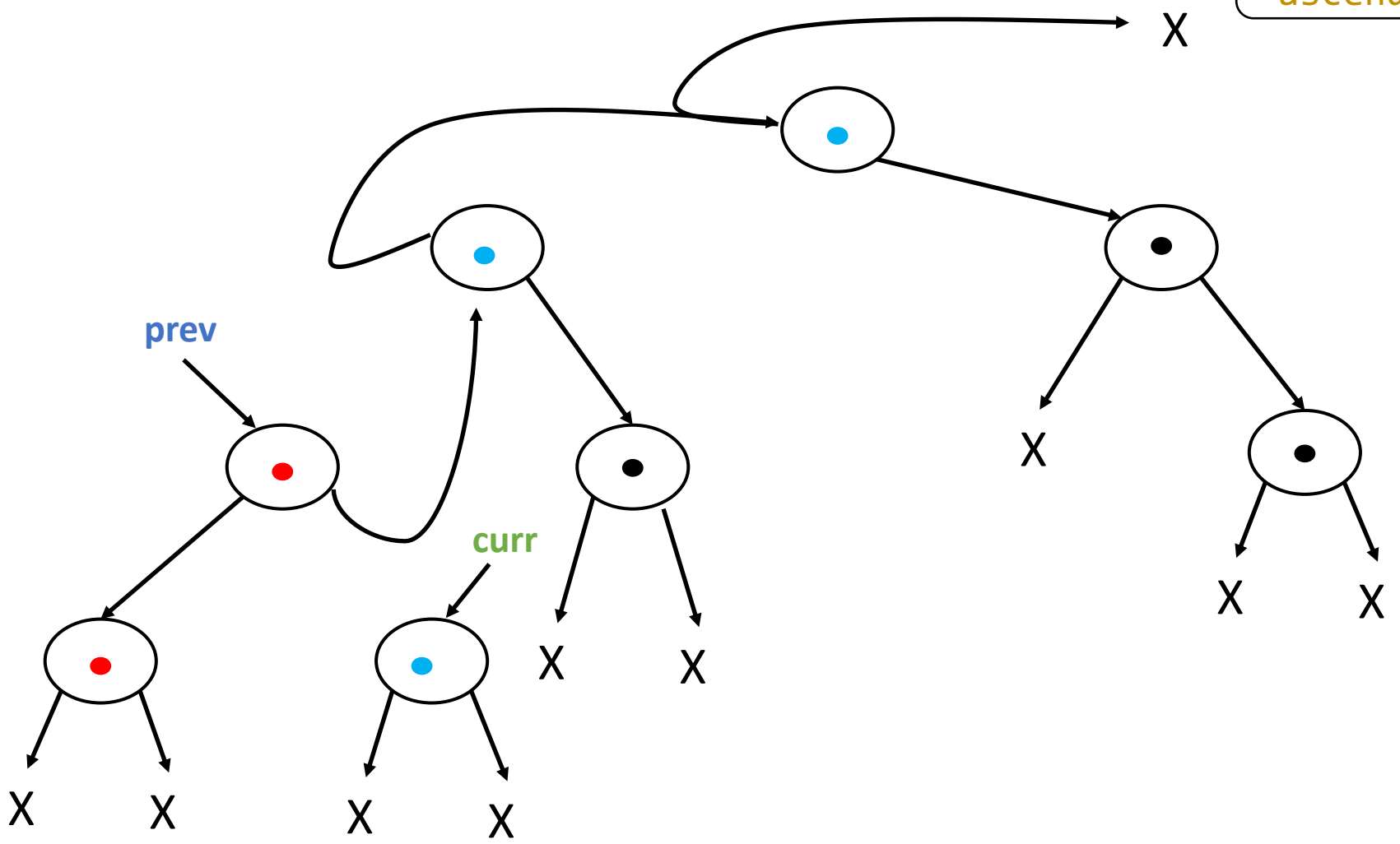
Traversal…

Routine used:
`descendLeft()`



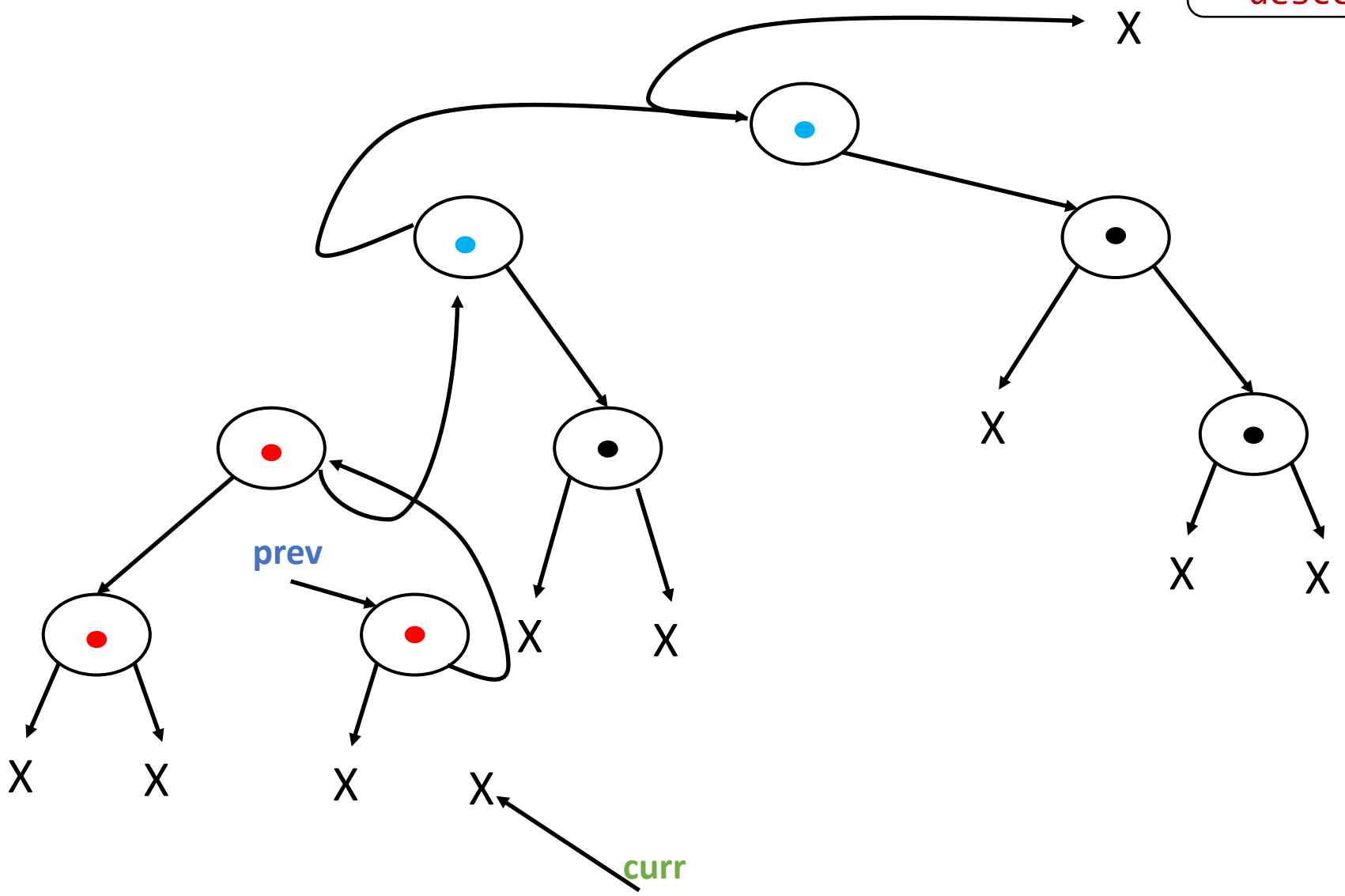
Traversal...

Routine used:
`ascendFromleft()`



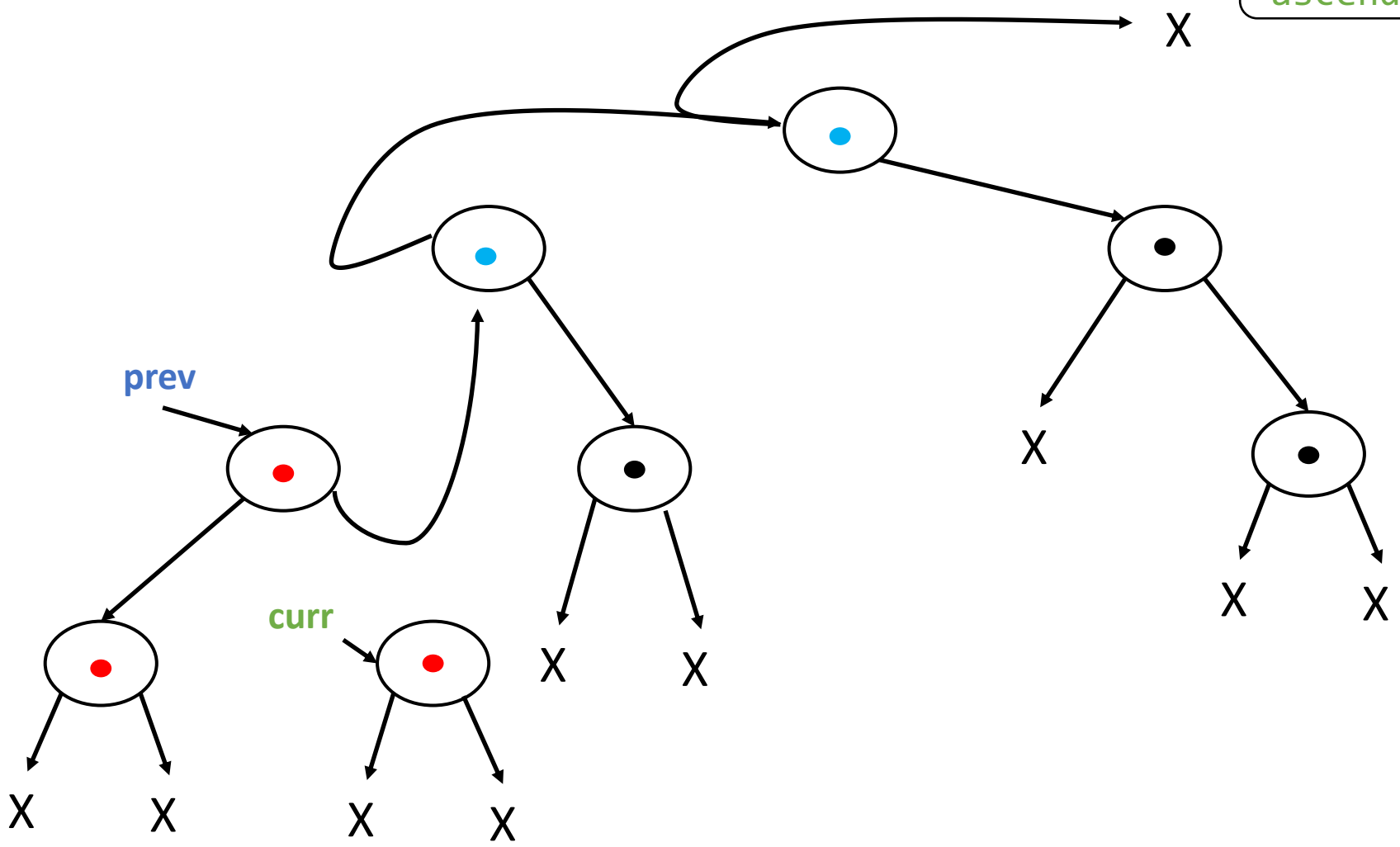
Traversal…

Routine used:
`descendRight()`



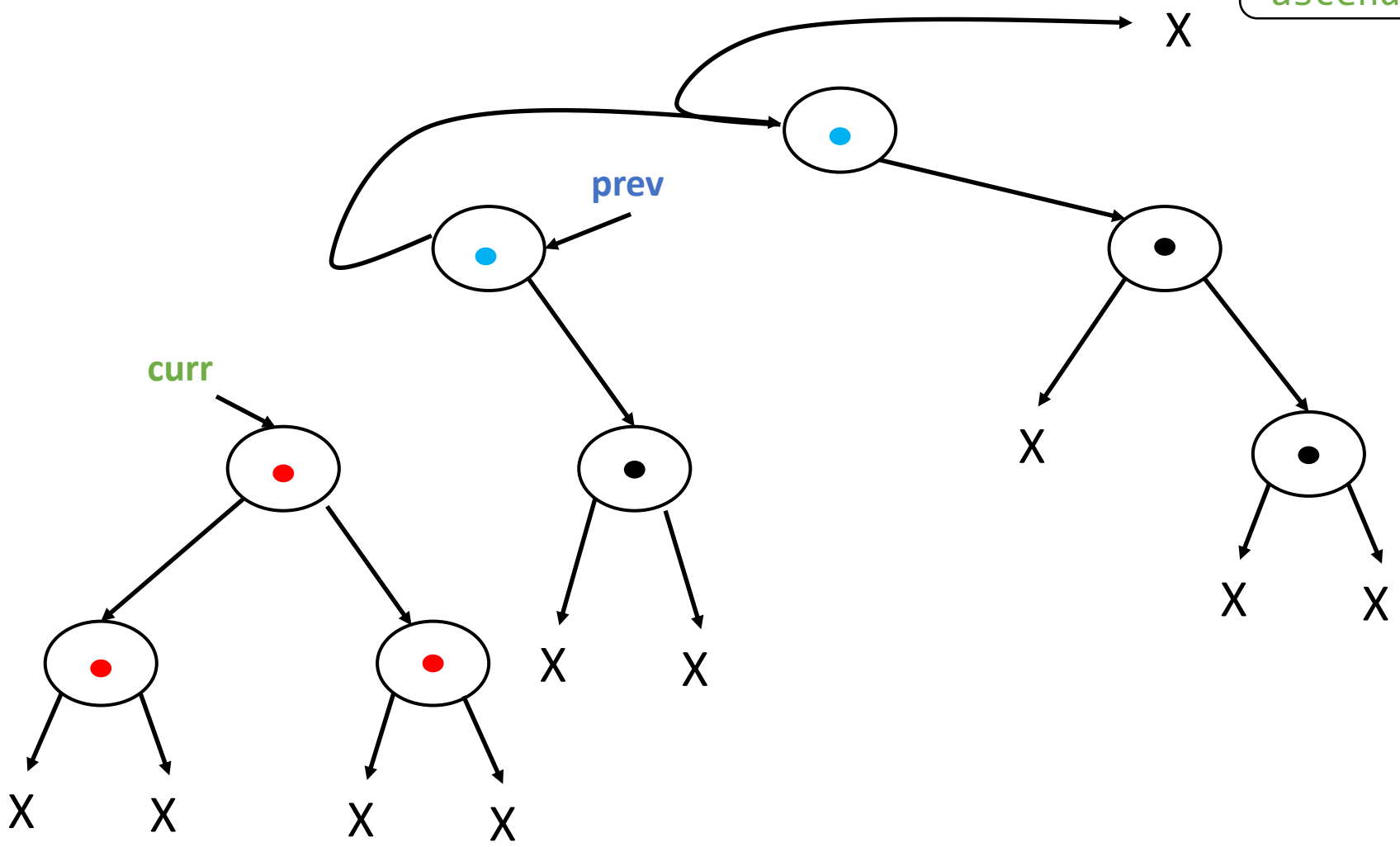
Traversal…

Routine used:
`ascendFromRight()`



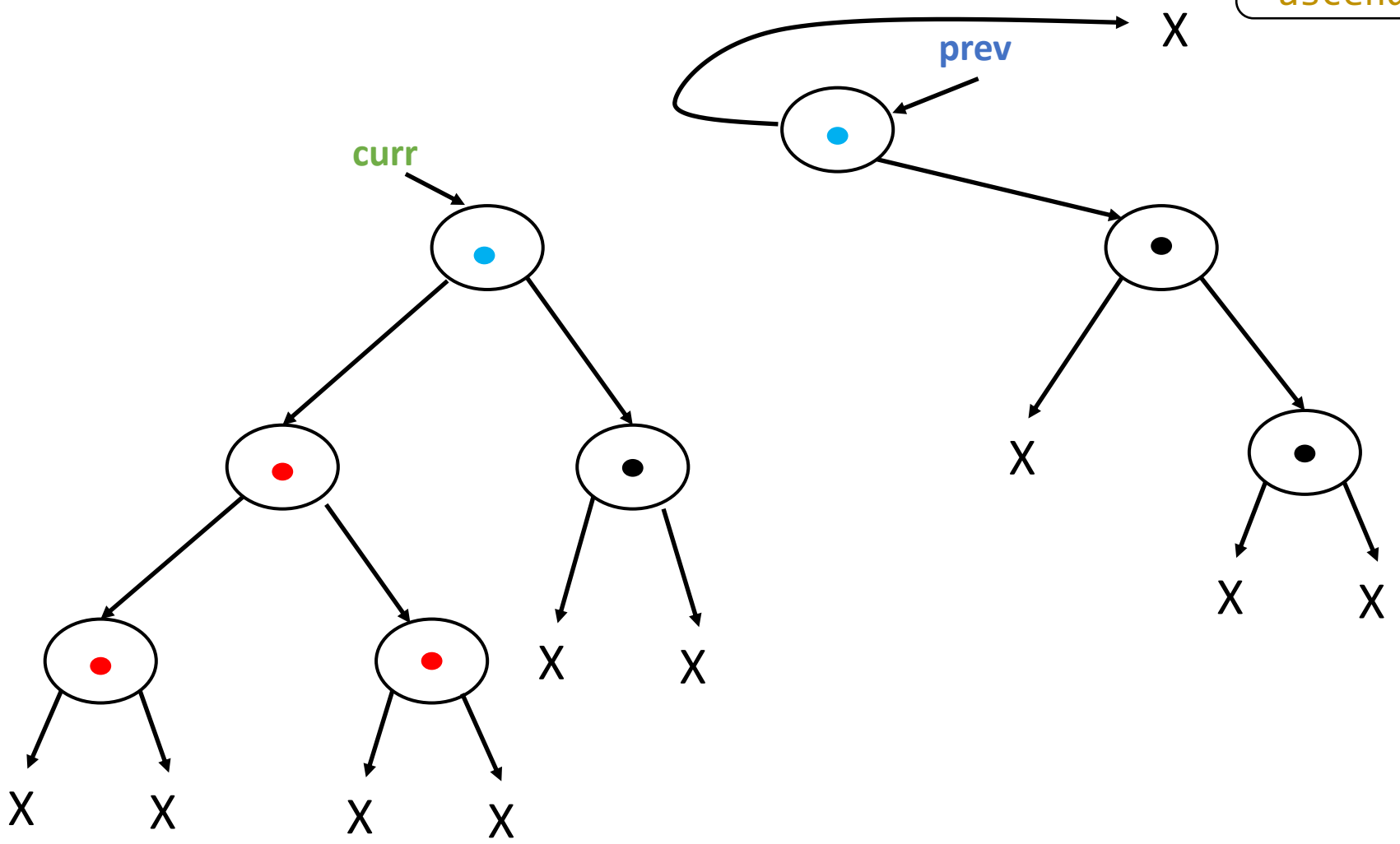
Traversal…

Routine used:
`ascendFromRight()`



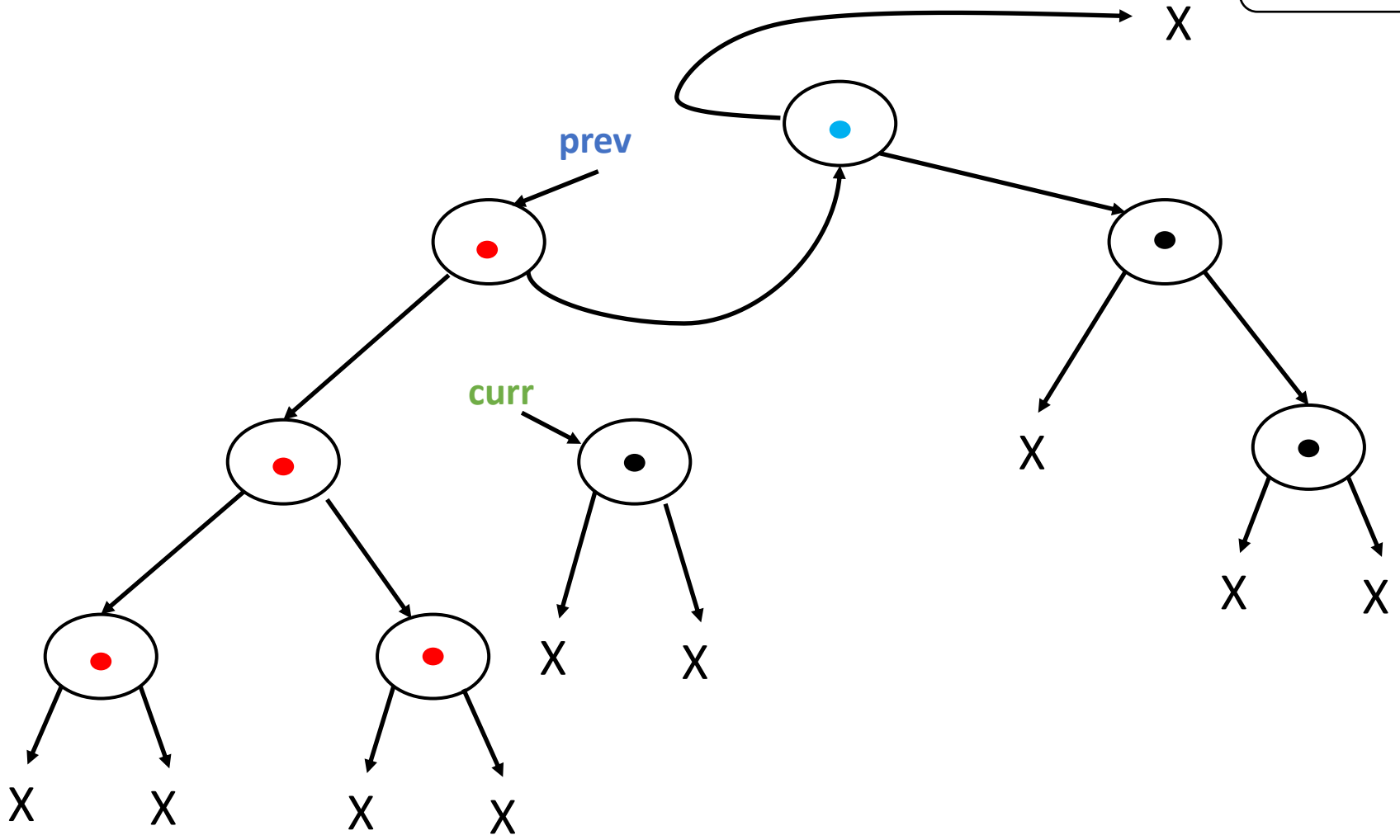
Traversal...

Routine used:
`ascendFromLeft()`



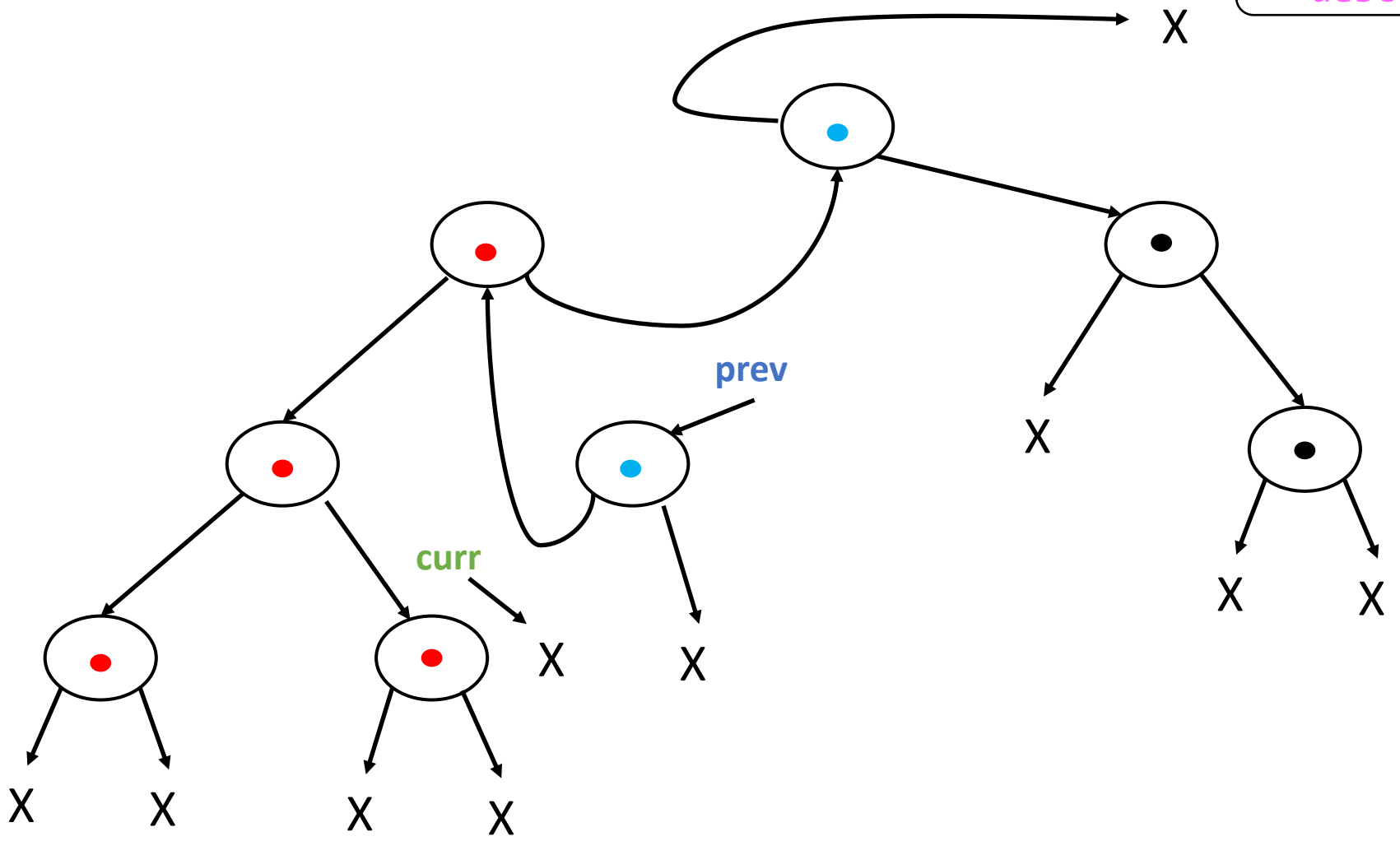
Traversal…

Routine used:
`descendRight()`



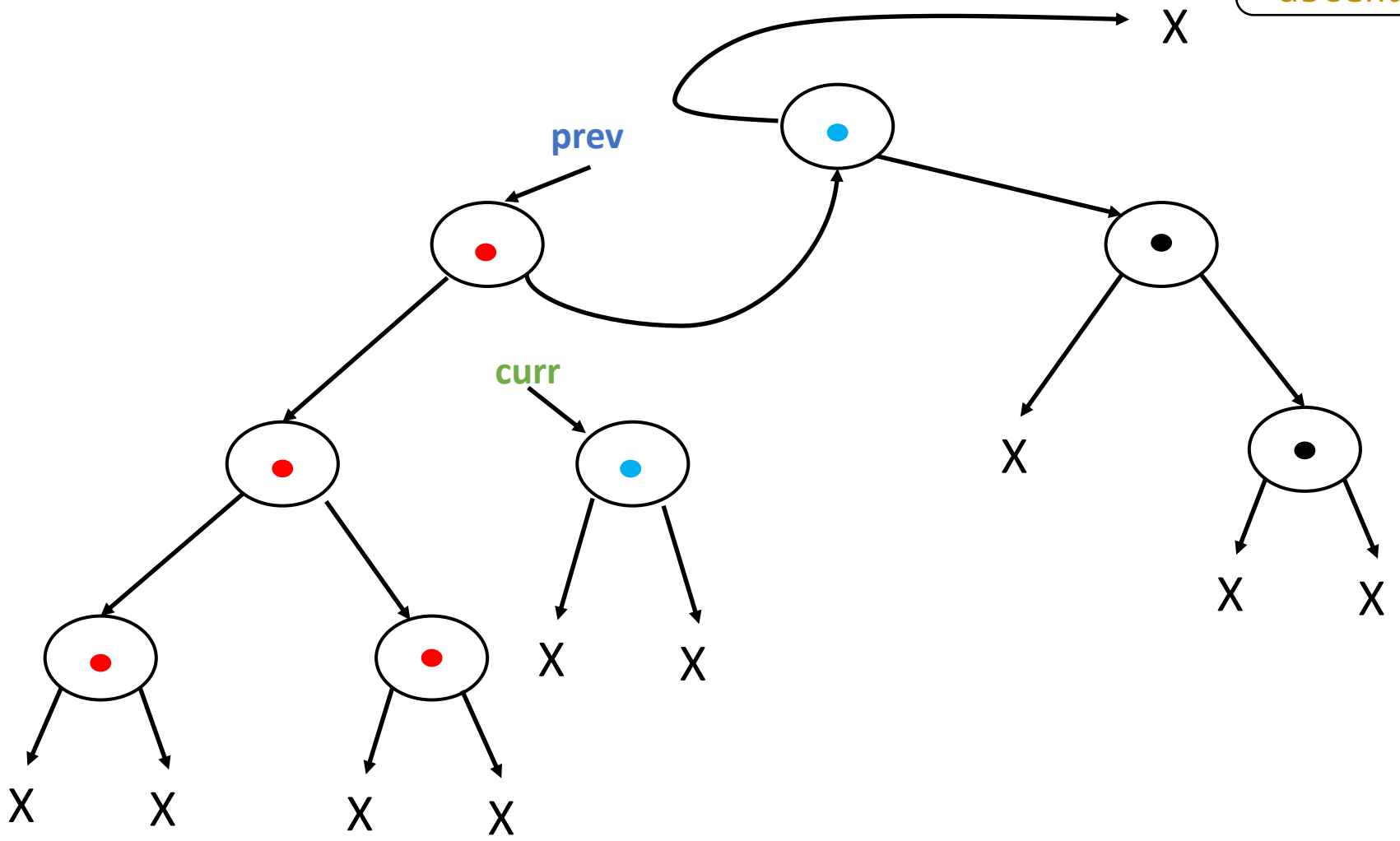
Traversal…

Routine used:
`descendLeft()`



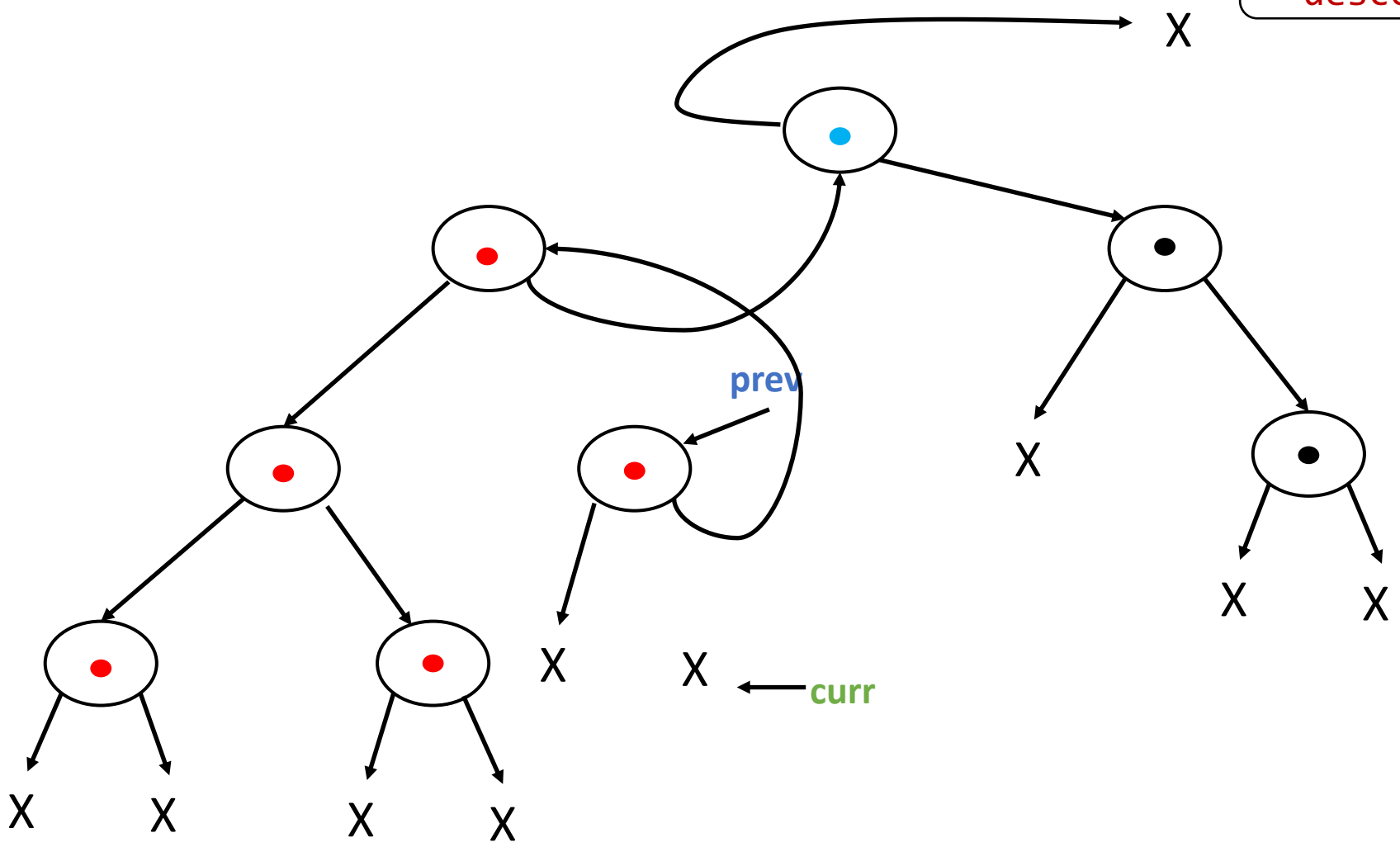
Traversal...

Routine used:
`ascendFromLeft()`



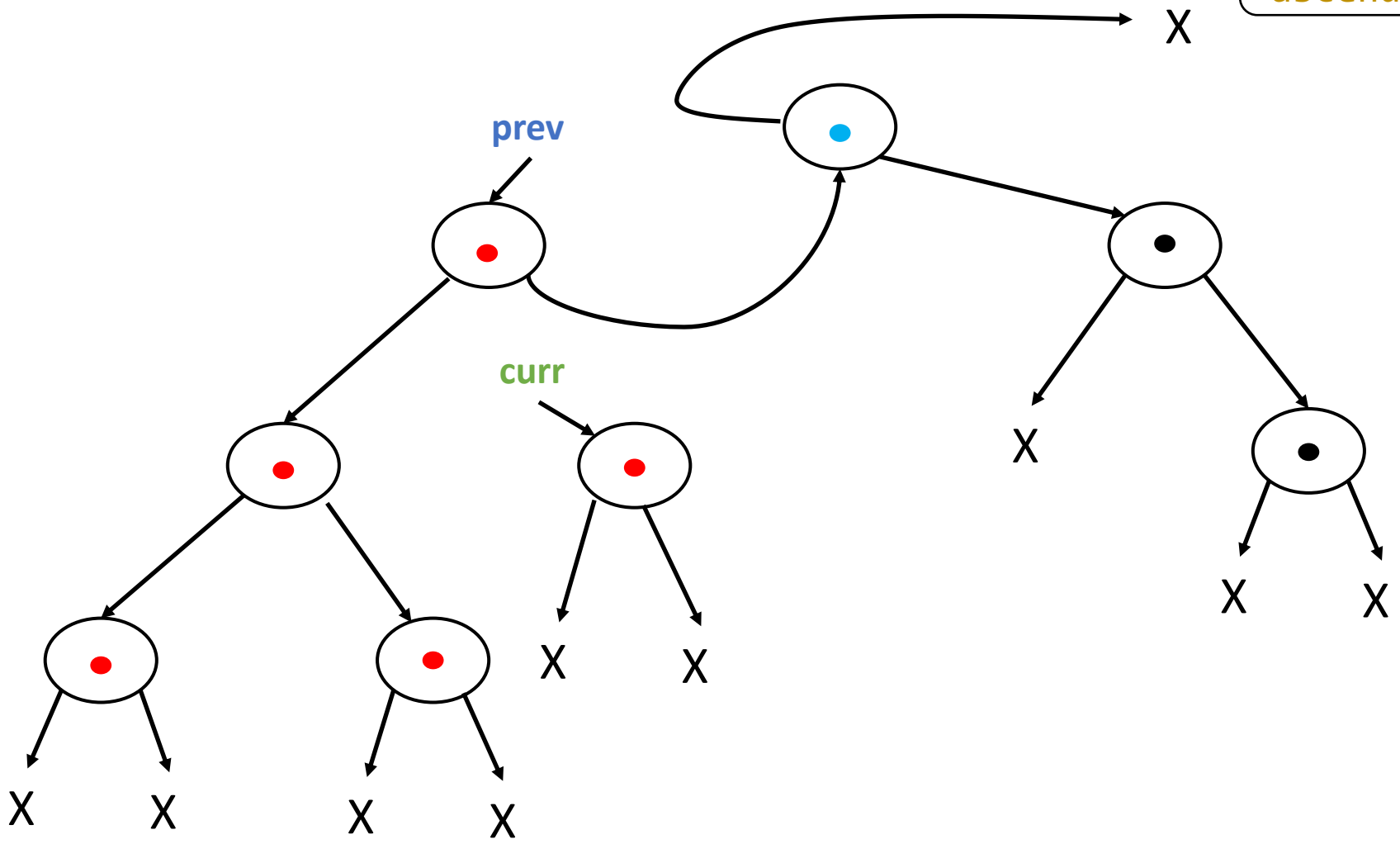
Traversal…

Routine used:
`descendRight()`



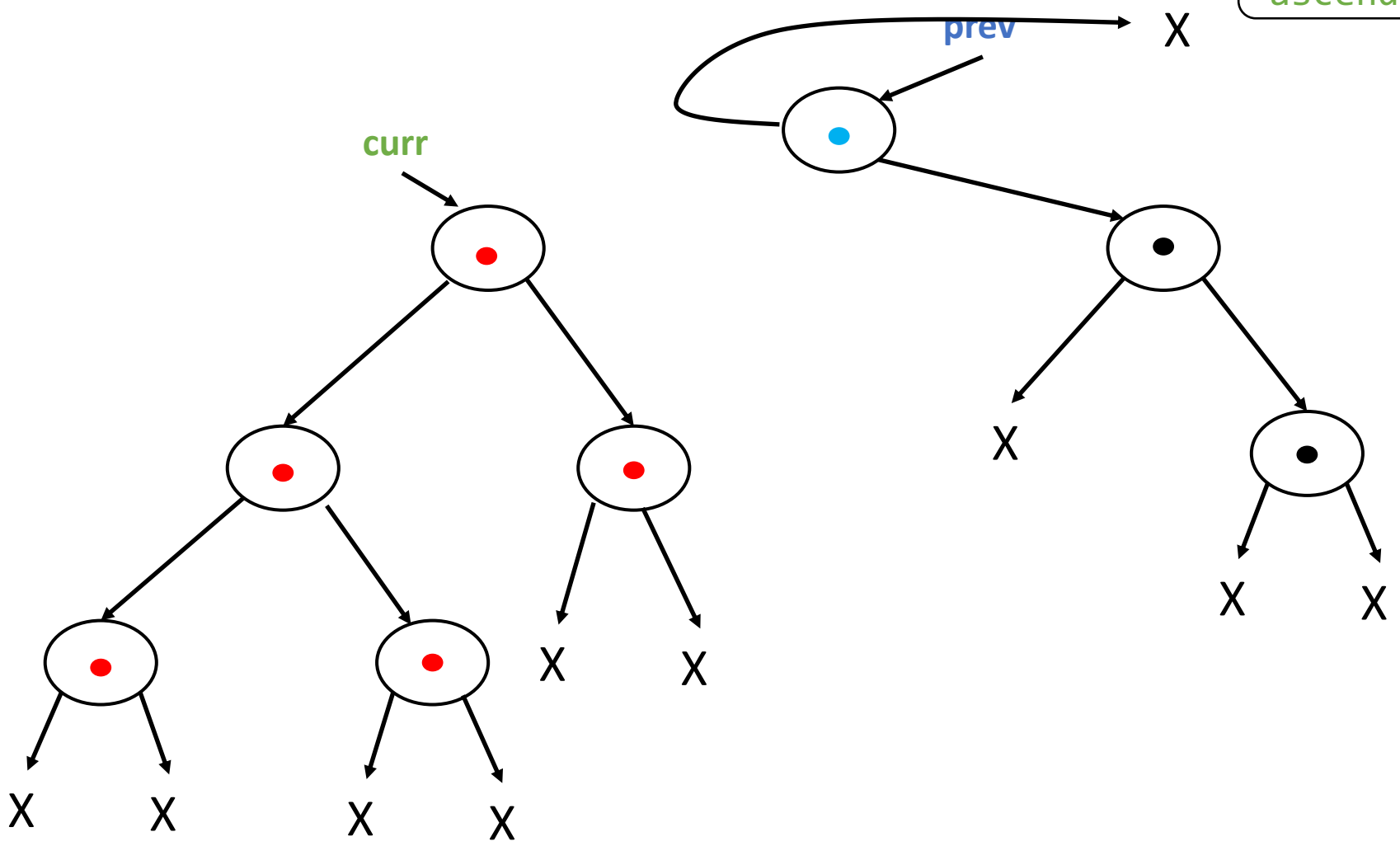
Traversal…

Routine used:
`ascendFromRight()`



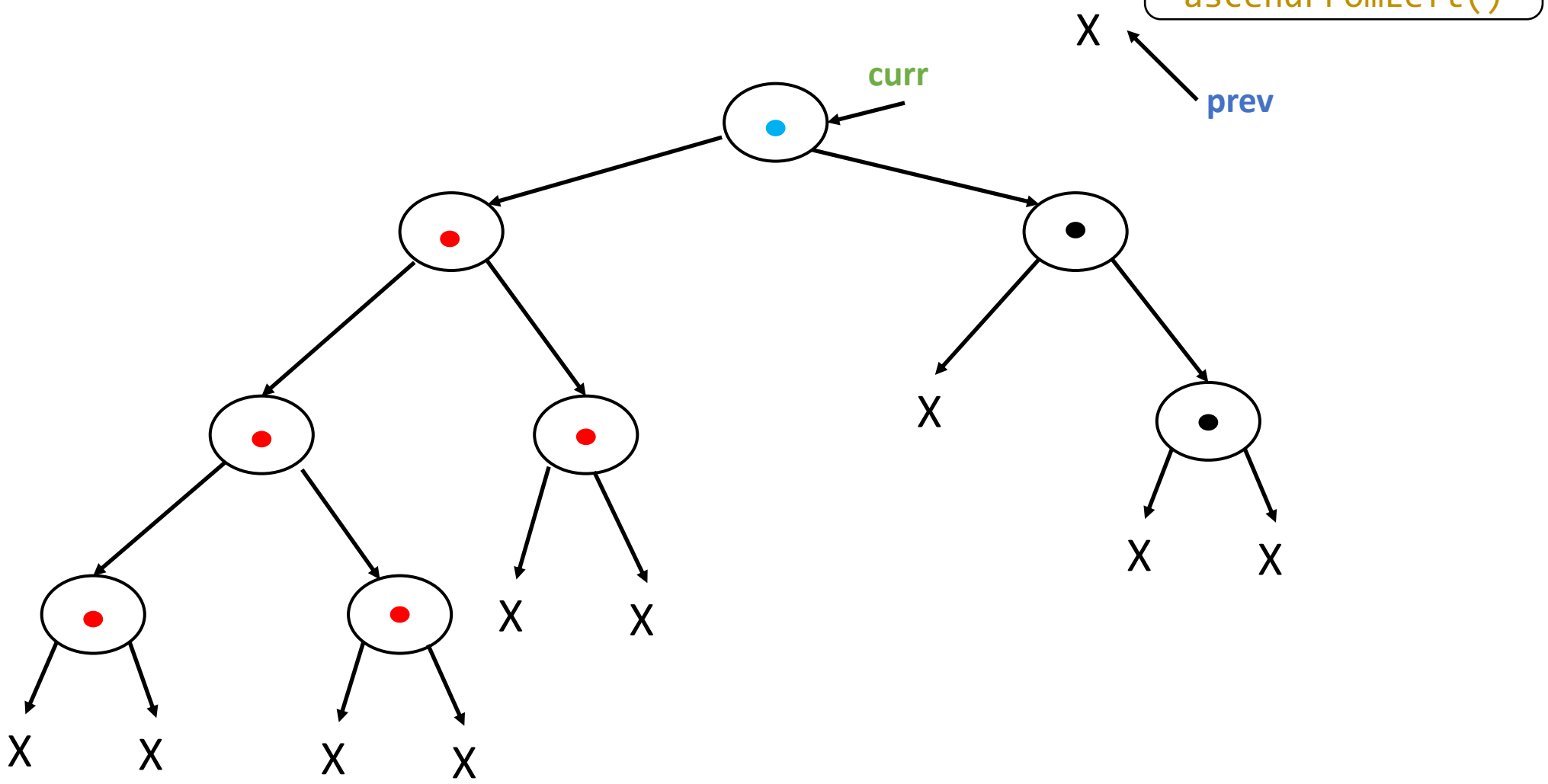
Traversal…

Routine used:
`ascendFromRight()`



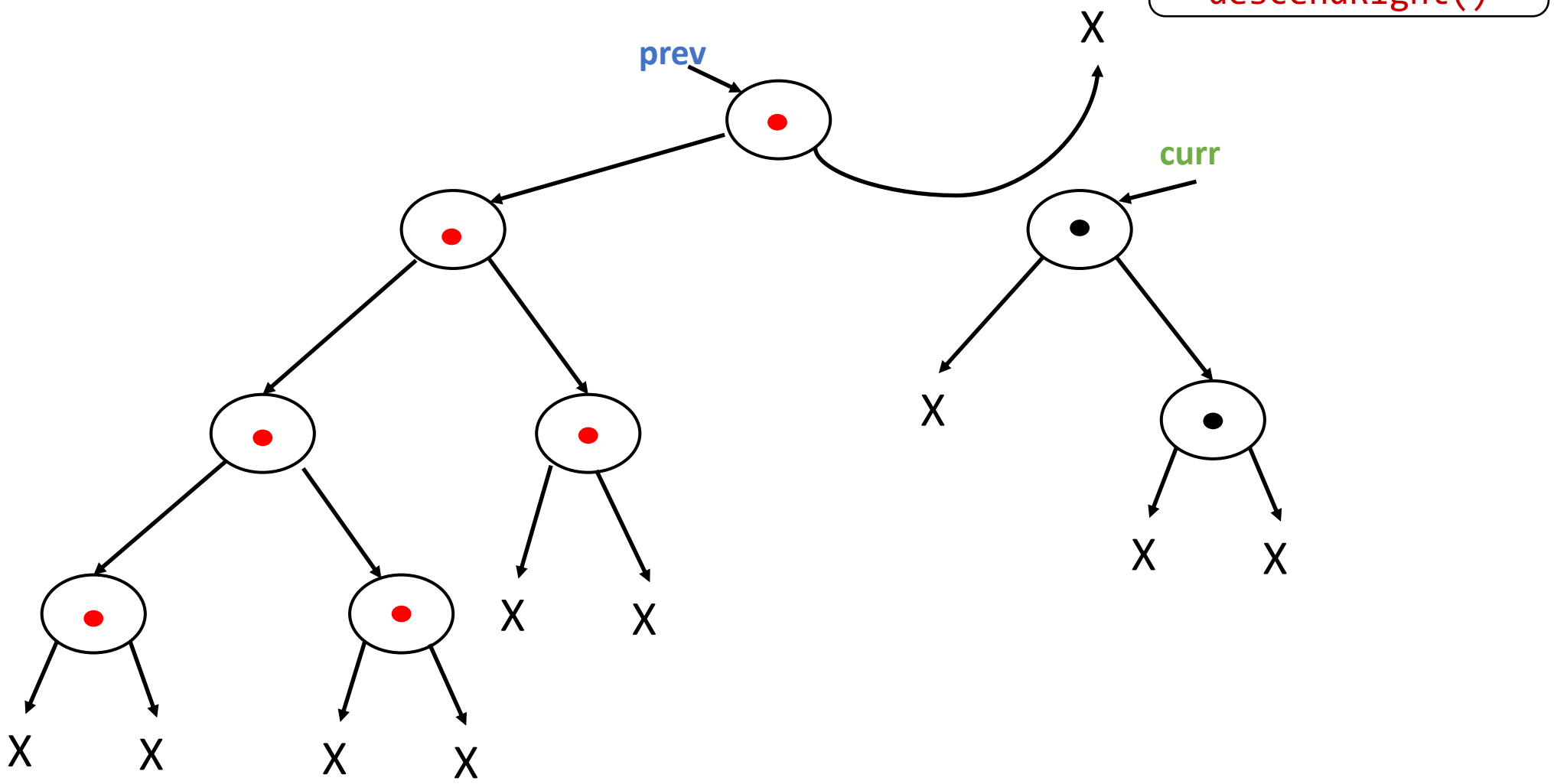
Traversal…

Routine used:
`ascendFromLeft()`



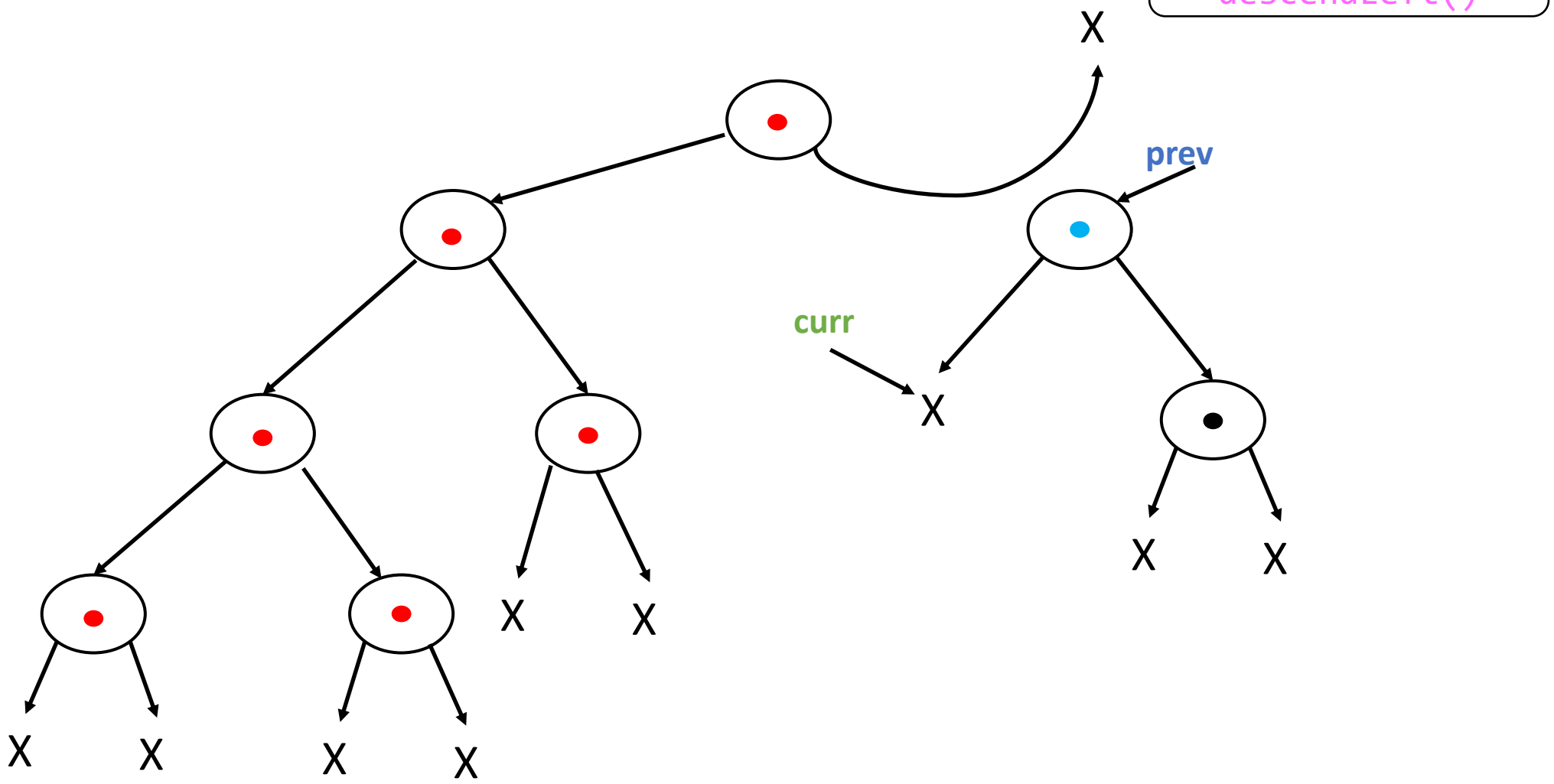
Traversal...

Routine used:
`descendRight()`



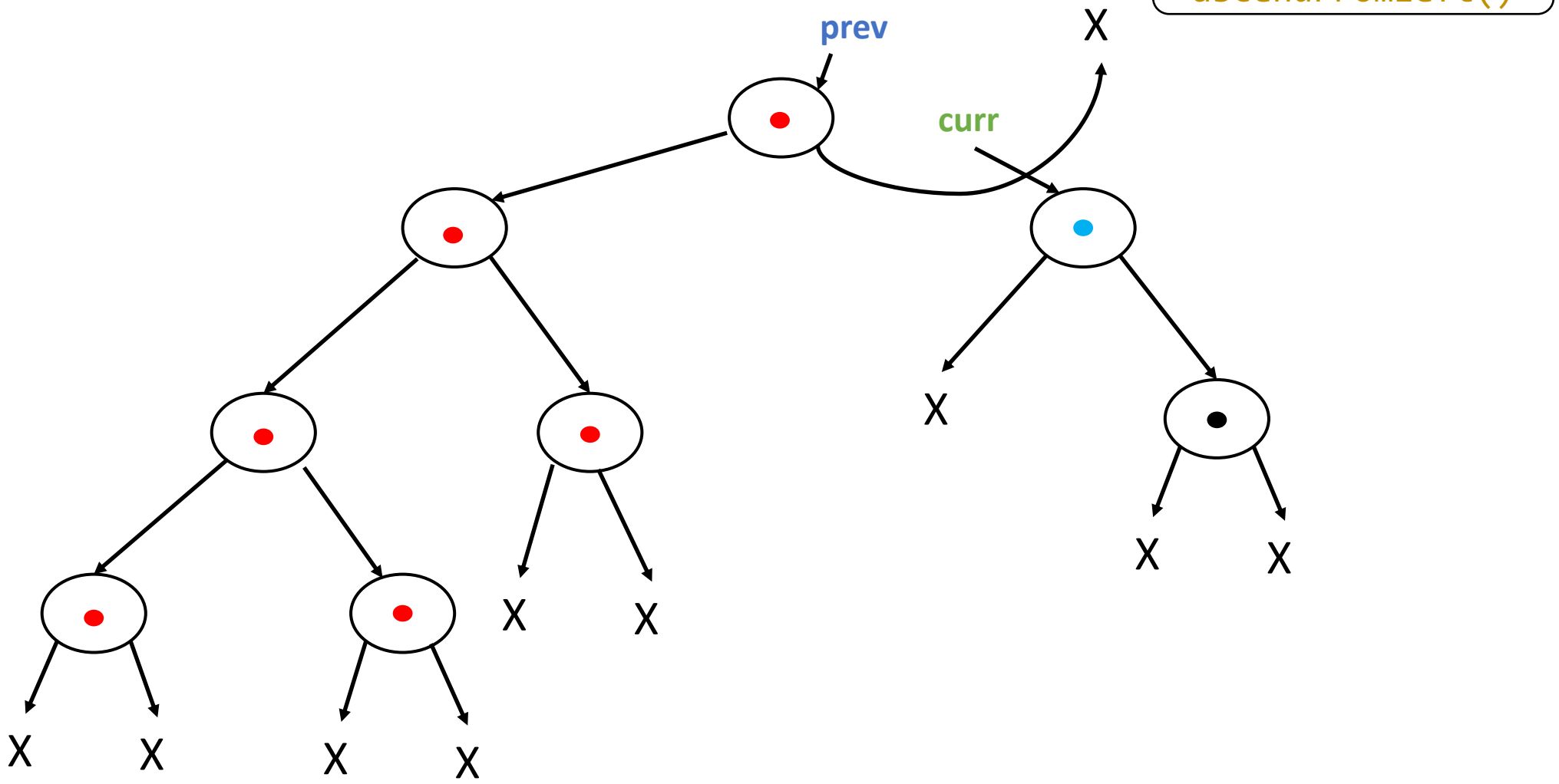
Traversal...

Routine used:
`descendLeft()`



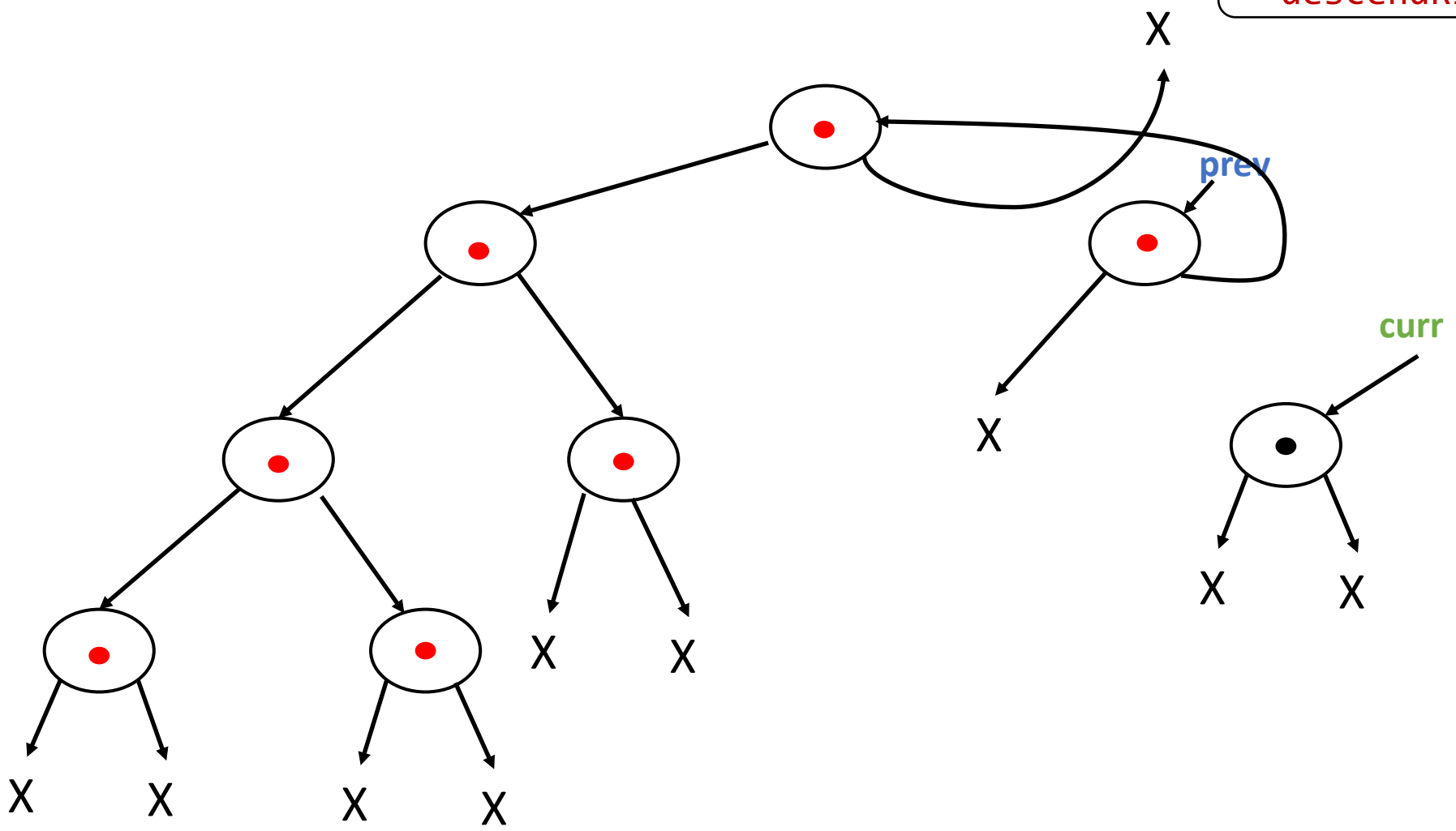
Traversal...

Routine used:
`ascendFromleft()`



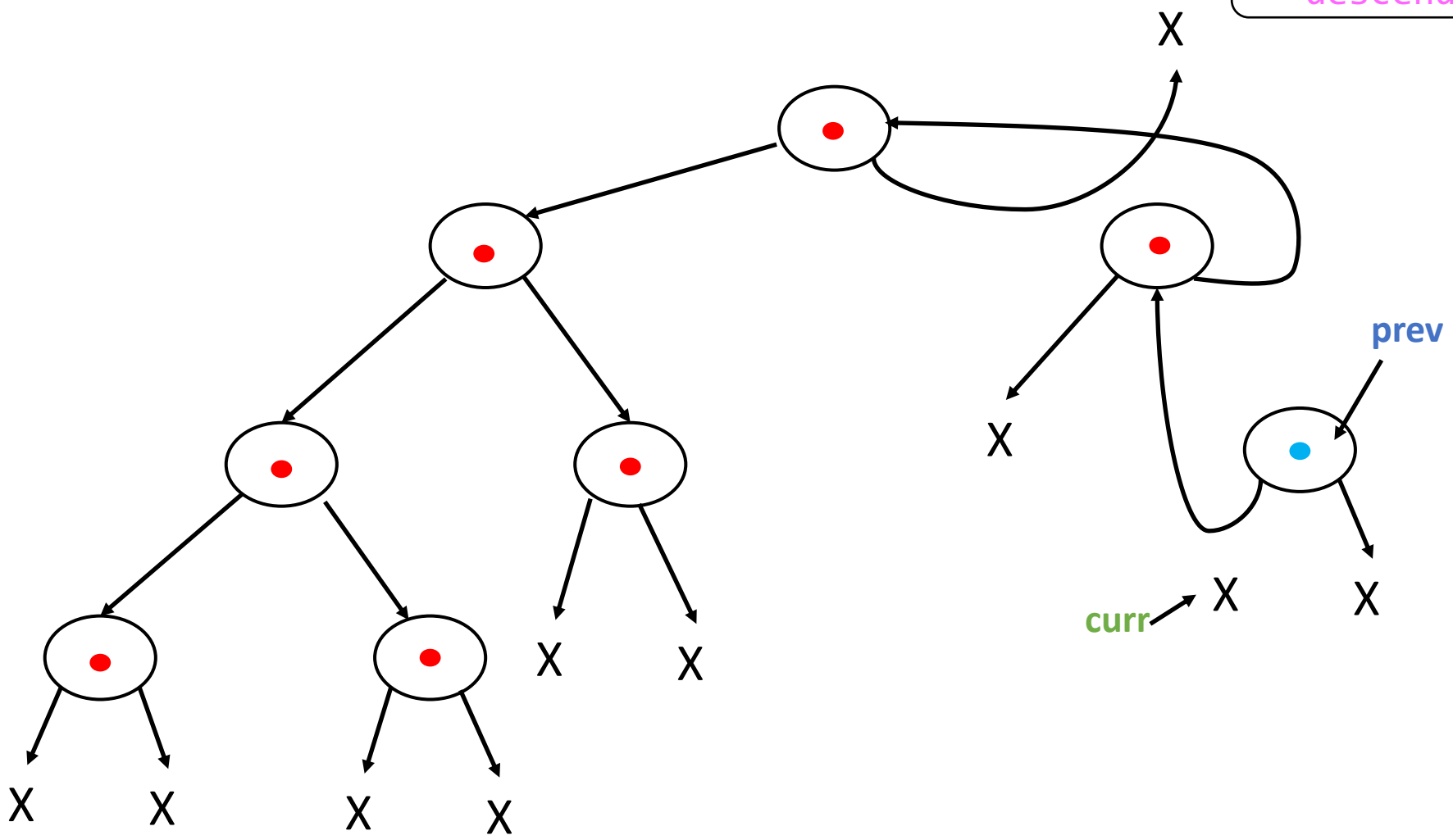
Traversal...

Routine used:
`descendRight()`



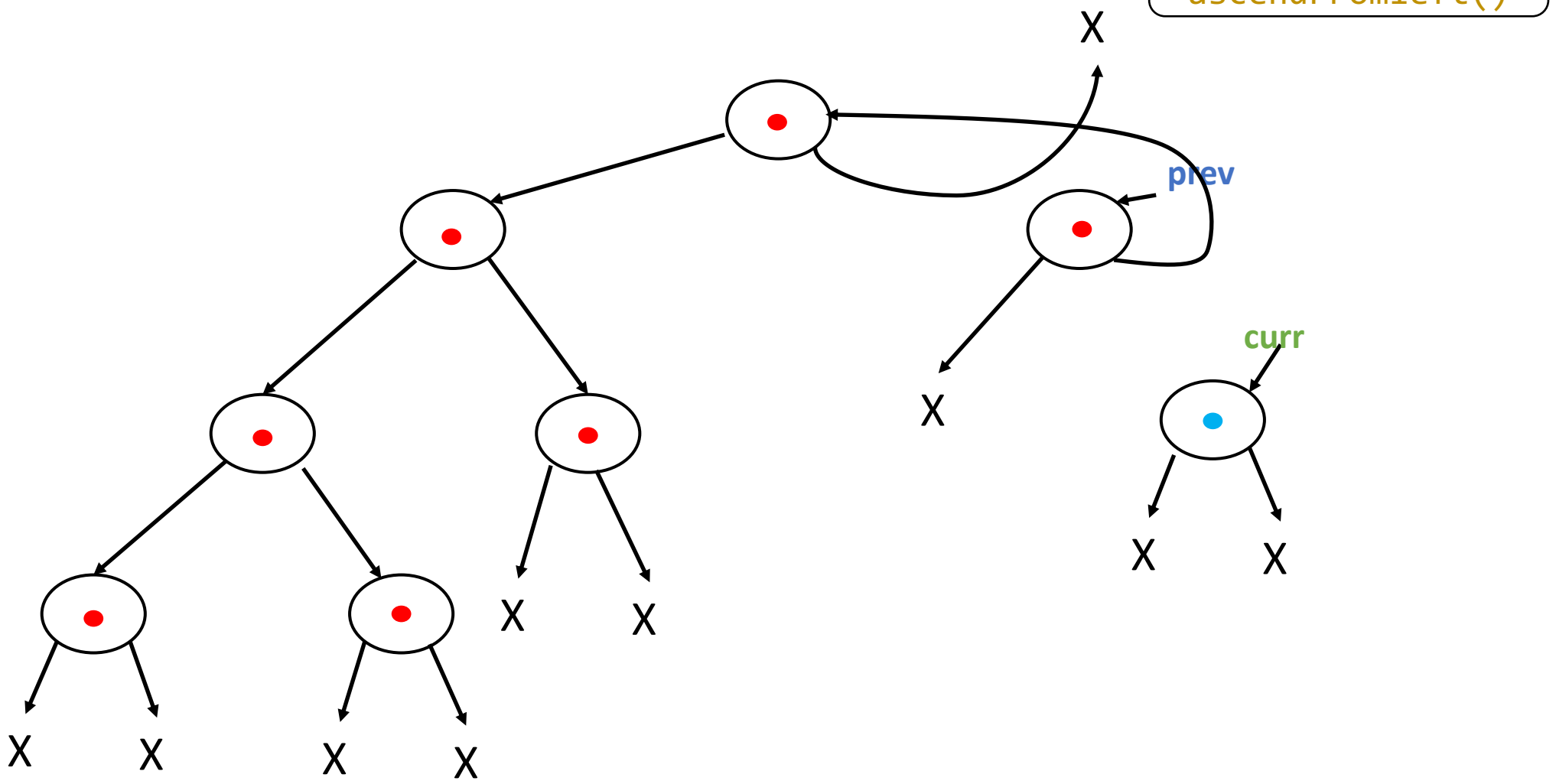
Traversal…

Routine used:
`descendLeft()`



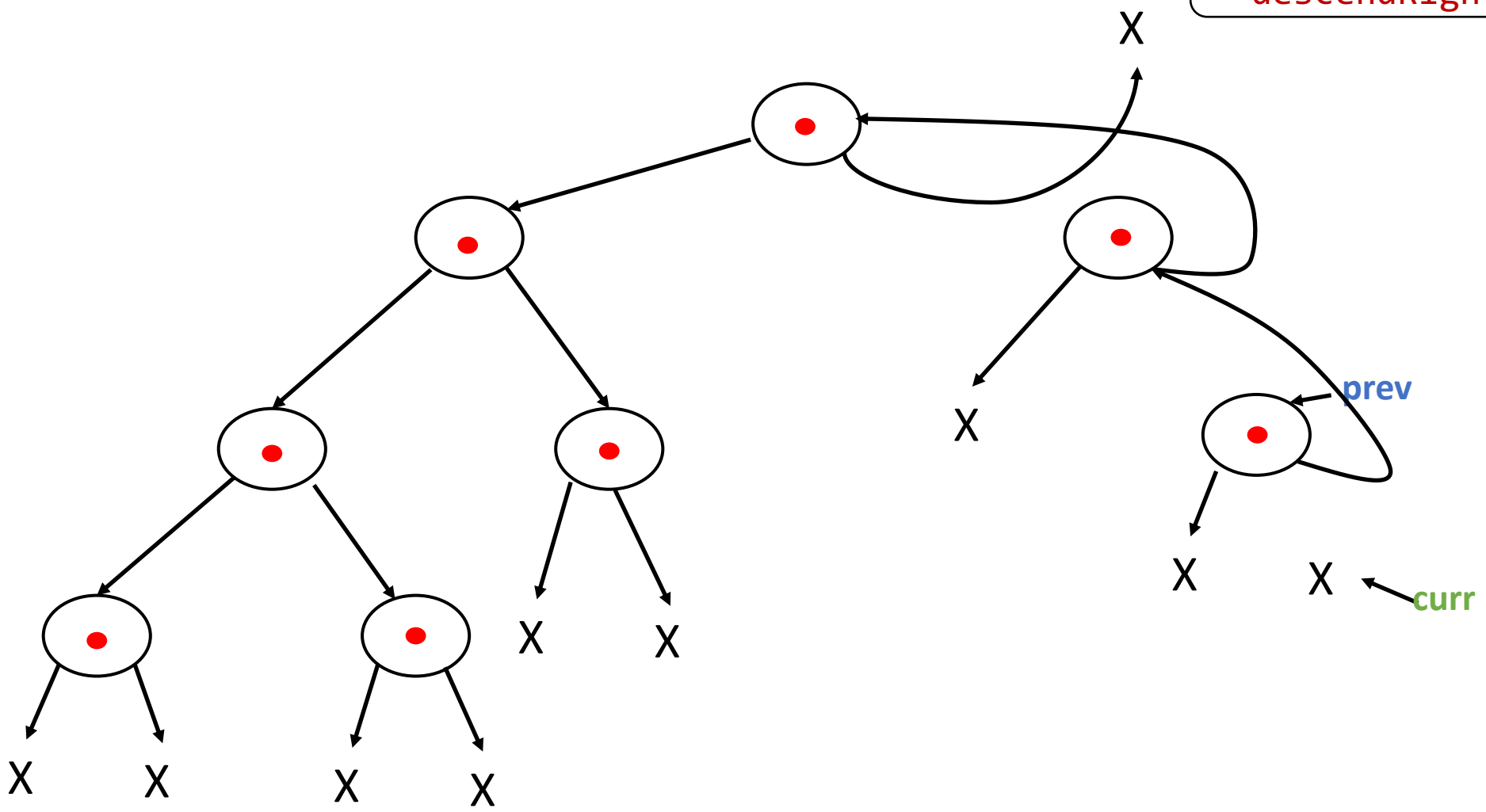
Traversal...

Routine used:
`ascendFromleft()`



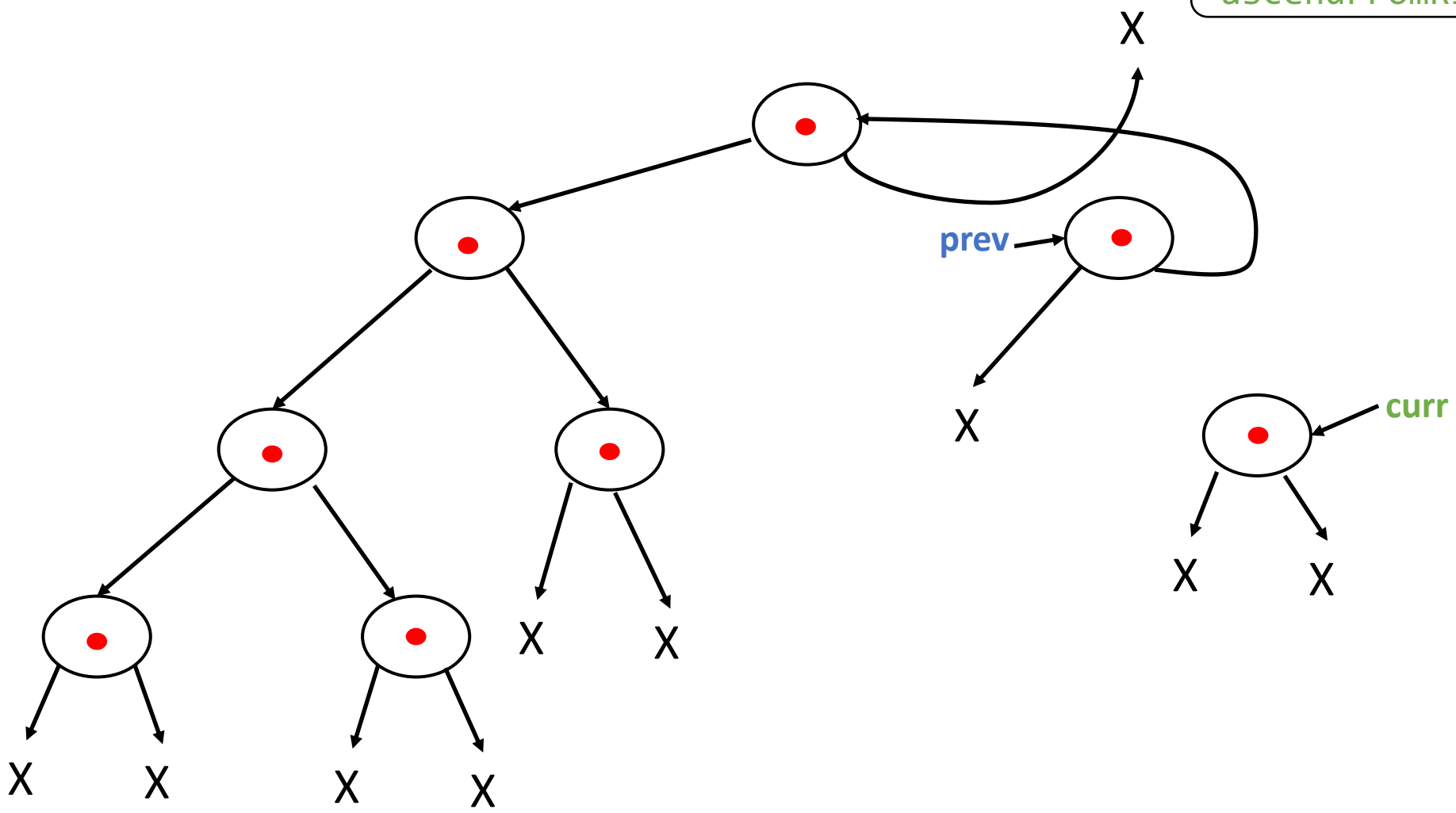
Traversal…

Routine used:
`descendRight()`



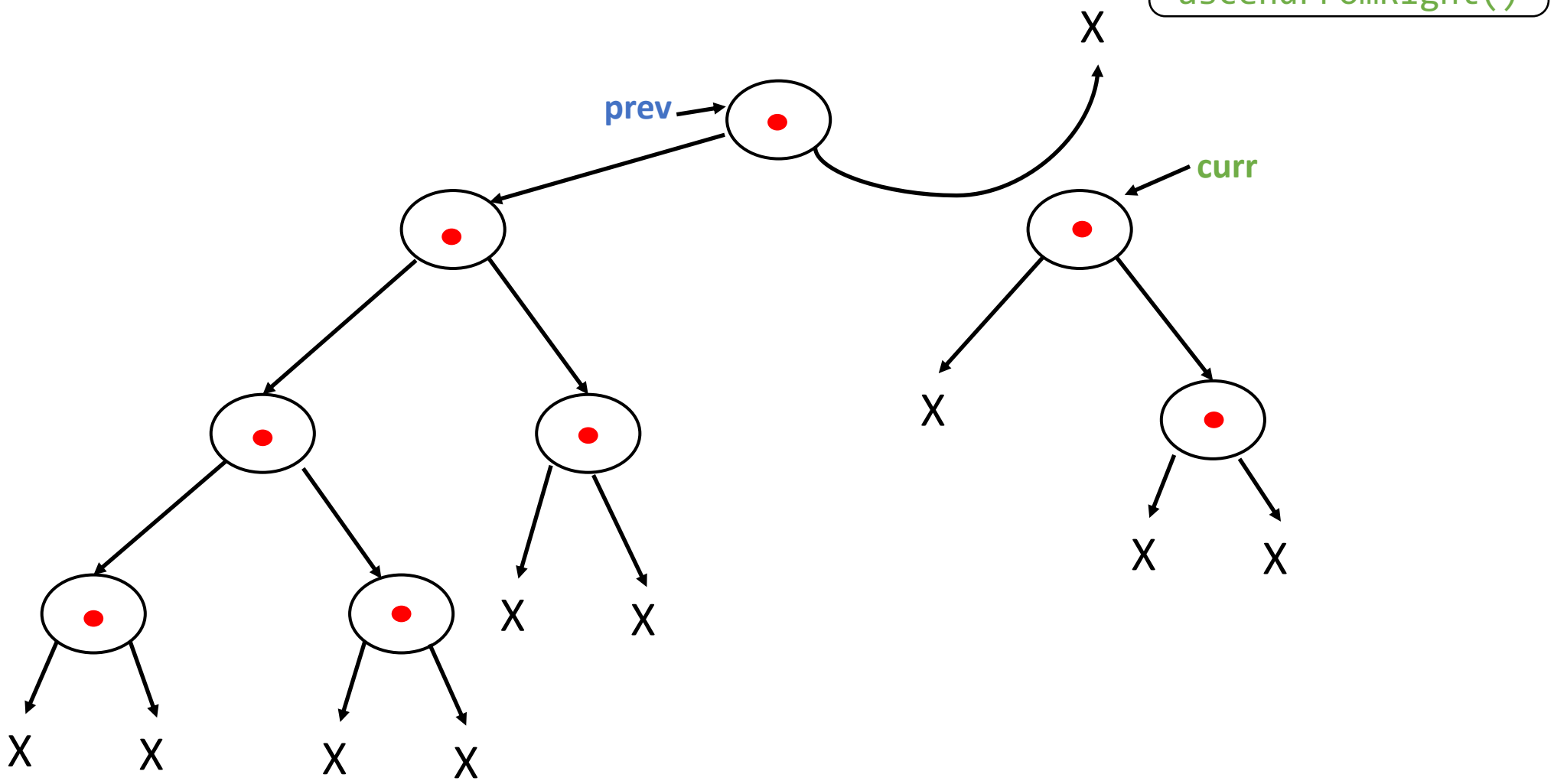
Traversal…

Routine used:
`ascendFromRight()`



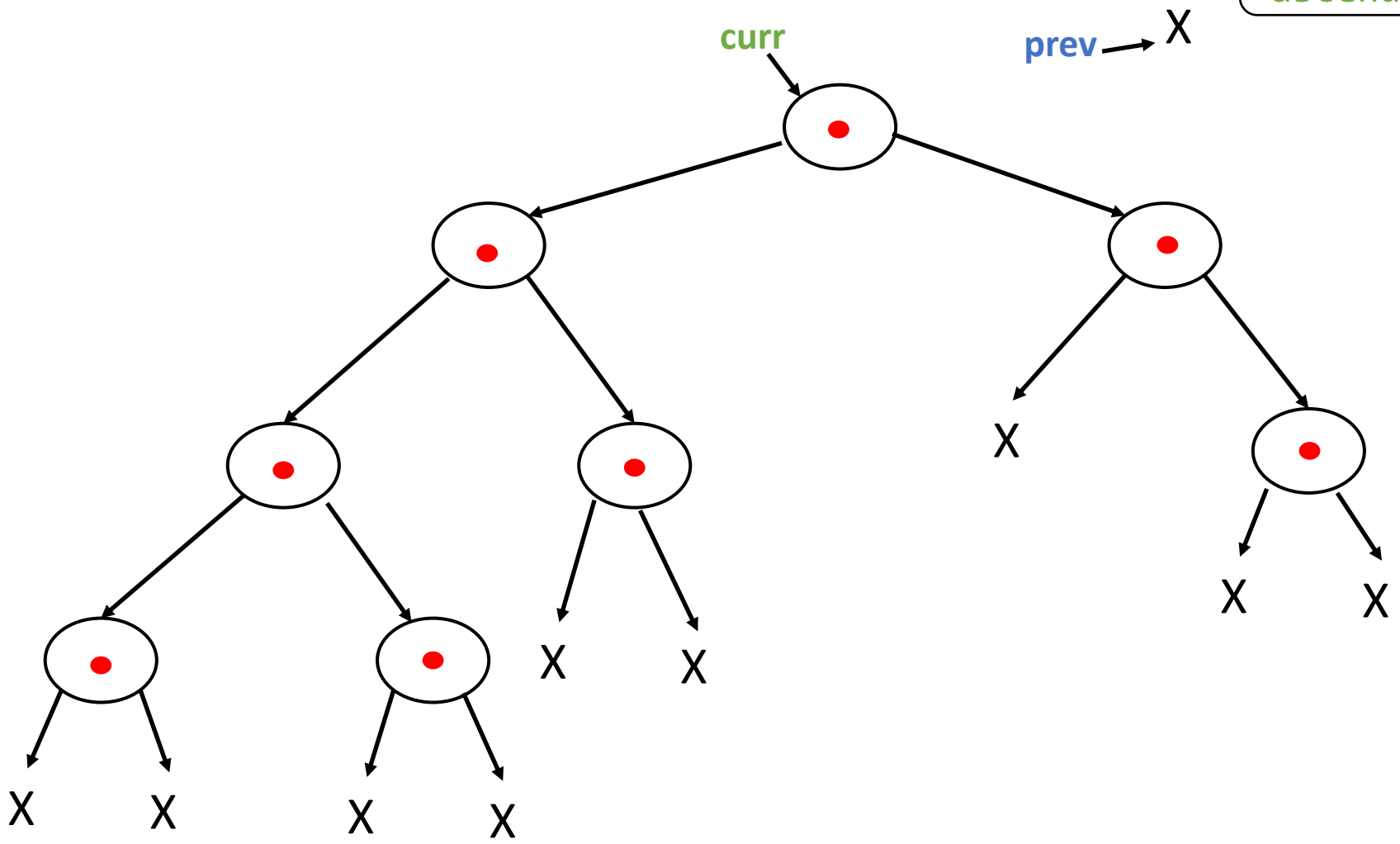
Traversal…

Routine used:
`ascendFromRight()`

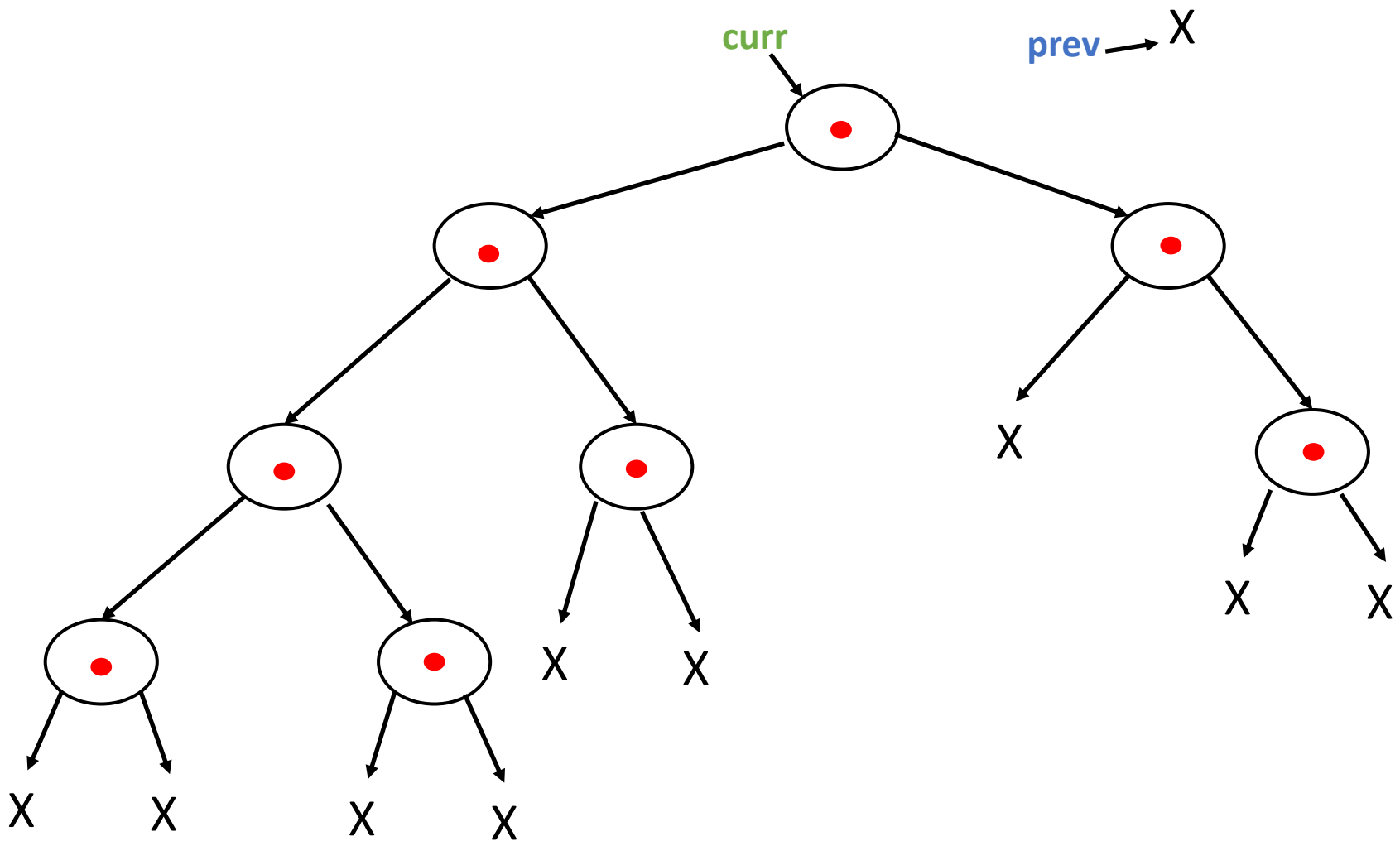


Traversal…

Routine used:
`ascendFromRight()`



Traversal…

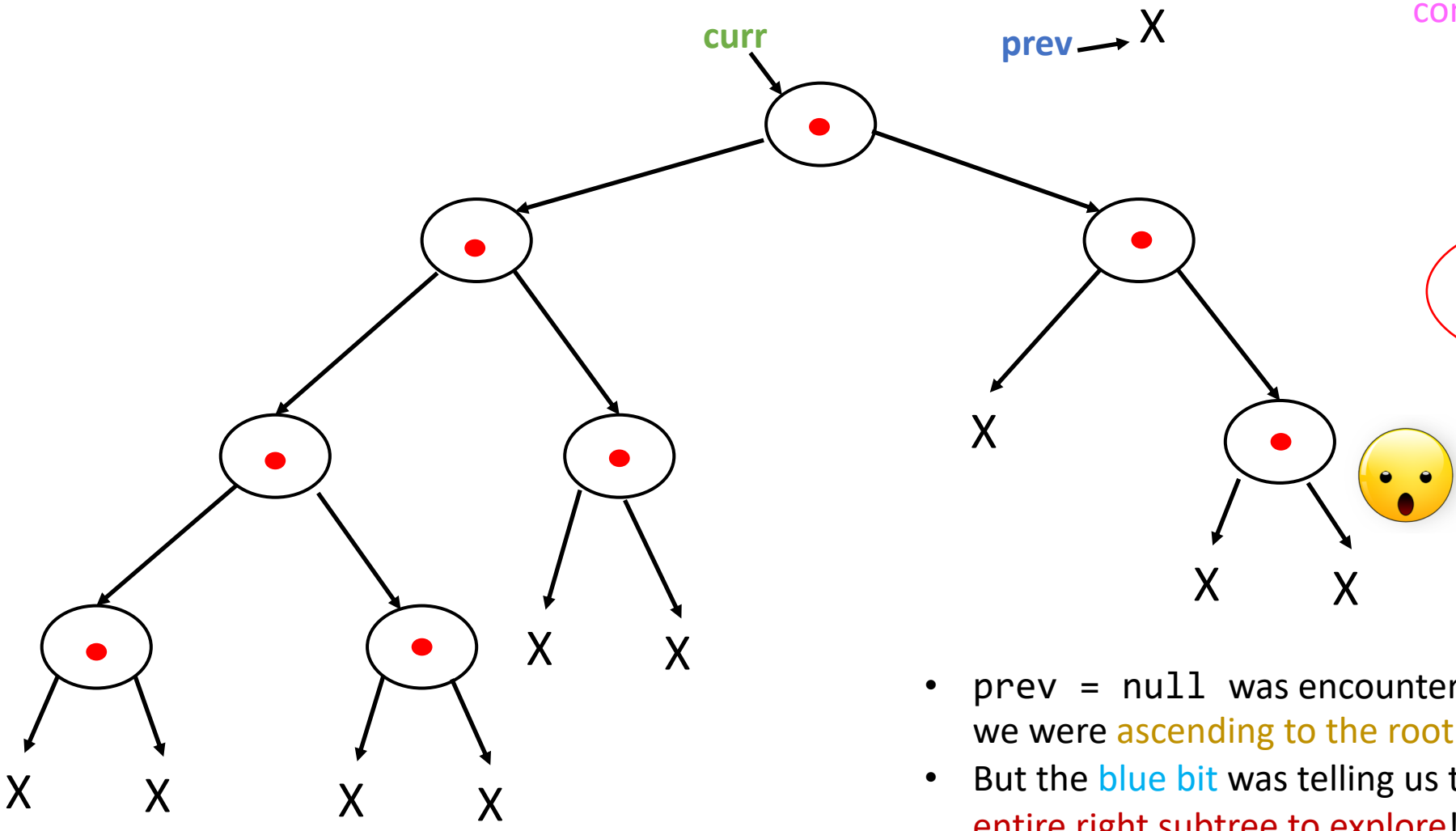


What's the termination condition of Link Inversion?

`prev = null`

Something Else

Traversal…



What's the termination condition of Link Inversion?

`prev = null`

Something Else

`prev = null`
AND
`curr.bit = 1`

- `prev = null` was encountered before as well, when we were ascending to the root from the left!
- But the blue bit was telling us that there might be an entire right subtree to explore!
- And, of course, there was.

Fill-in the full algorithm!

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null))){

        while(curr != null){
            print(curr.value);
            curr.bit = 0;
(1)         [redacted]
        }

        while(prev != null && prev.bit == 1)
(2)        [redacted]

        if(prev == null)
           return;

        else {

(3)        [redacted]
           curr.bit=1;
(4)        [redacted]

         }

    }

}
```

descendToLeft()

ascendFromLeft()

descendToRight()

ascendFromRight()

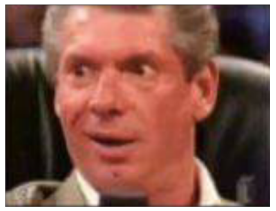
Fill-in the full algorithm!

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null))){

        while(curr != null){
            print(curr.value);
            curr.bit = 0;
(1)         descendToLeft()
        }

        while(prev != null && prev.bit == 1)
(2)        [redacted]

        if(prev == null)
           return;

        else {
(3)        [redacted]
           curr.bit=1;
(4)        [redacted]
         }

    }

}
```

ascendFromLeft()

descendToRight()

ascendFromRight()



Fill-in the full algorithm!

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null))){

        while(curr != null){
            print(curr.value);
            curr.bit = 0;
(1)         descendToLeft()
        }

        while(prev != null && prev.bit == 1)
(2)       ascendFromRight()
        if(prev == null)
           return;

        else {
(3)
            [redacted]
            curr.bit=1;
(4)         [redacted]

         }

    }

}
```

ascendFromLeft()

descendToRight()

Fill-in the full algorithm!

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null))){

        while(curr != null){
            print(curr.value);
            curr.bit = 0;
(1)         descendToLeft()
        }

        while(prev != null && prev.bit == 1)
(2)       ascendFromRight()
        if(prev == null)
           return;

        else {
(3)       ascendFromLeft()
          curr.bit=1;
(4)       descendToRight()
         }

    }

}
```

The full algorithm

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null)){

        while(curr != null){                    // Descend left as far as you can.
            print(curr.value);                  // Remember: Preorder Traversal.
            curr.bit = 0;
            descendToLeft();
        }

        while(prev != null && prev.bit == 1)    // Ascend from right as far as you can.
           ascendFromRight();                   // There might be no ascension; that's fine

        if(prev == null)                        //Termination condition.
           return;

        else {
           ascendFromLeft();                    // This time, however, we are guaranteed an ascension.
           curr.bit=1;                          // The bit is only set when you ascend to a node from the left…
           descendToRight();                    // … and this signals an immediate descent to the right!
         }
    }
}
```

The full algorithm

Now modify it to do:

a) Inorder Traversal

b) Postorder Traversal

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null)){

        while(curr != null){                    // Descend left as far as you can.
            print(curr.value);                  // Remember: Preorder Traversal.
            curr.bit = 0;
            descendToLeft();
        }

        while(prev != null && prev.bit == 1)    // Ascend from right as far as you can.
           ascendFromRight();                   // There might be no ascension; that's fine

        if(prev == null)                        //Termination condition.
           return;

        else {
           ascendFromLeft();                    // This time, however, we are guaranteed an ascension.
           curr.bit=1;                          // The bit is only set when you ascend to a node from the left…
           descendToRight();                    // … and this signals an immediate descent to the right!
         }
    }
}
```

Inorder:

Now modify it to do:

a) Inorder Traversal

b) Postorder Traversal

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null)){

        while(curr != null){                    // Descend left as far as you can.
            print(curr.value);                  // Remember: Preorder Traversal.
            curr.bit = 0;
            descendToLeft();
        }

        while(prev != null && prev.bit == 1)    // Ascend from right as far as you can.
           ascendFromRight();                   // There might be no ascension; that's fine

        if(prev == null)                        //Termination condition.
           return;

        else {
           ascendFromLeft();                    // This time, however, we are guaranteed an ascension.
           print(curr.value);                   // Remember: Inorder Traversal.
           curr.bit=1;                          // The bit is only set when you ascend to a node from the left…
           descendToRight();                    // … and this signals an immediate descent to the right!
         }
    }
}
```

Postorder:

Now modify it to do:

a) Inorder Traversal

b) Postorder Traversal

```
void preorderLI(){

    if(root == null) return;
    Node curr = root, prev = null;

    while(!((curr.bit == 1) && (prev ==null)){

        while(curr != null){                    // Descend left as far as you can.
            print(curr.value);                  // Remember: Preorder Traversal.
            curr.bit = 0;
            descendToLeft();
        }

        while(prev != null && prev.bit == 1){   // Ascend from right as far as you can.
           ascendFromRight();                   // There might be no ascension; that's fine
           print(curr.value);                   // Remember: Postorder Traversal.
        }

        if(prev == null)                        //Termination condition.
           return;

        else {
           ascendFromLeft();                    // This time, however, we are guaranteed an ascension.
           curr.bit=1;                          // The bit is only set when you ascend to a node from the left…
           descendToRight();                    // … and this signals an immediate descent to the right!
         }
    }
}
```

Analysis

- We cut down the **spatial** cost of threaded trees by one bit per node.
  - From $98n$ to $97n$.

Analysis

- We cut down the **spatial** cost of threaded trees by one bit per node.
  - From $98n$ to $97n$.
- What about time?

Analysis

- We cut down the **spatial** cost of threaded trees by one bit per node.
  - From $98n$ to $97n$.
- What about time?
- In Threaded Trees, we performed a standard algorithmic pessimistic analysis and deduced that **inorder traversal** will scan all $2n$ links.
  - In practice, it will scan $c \cdot n$ links, $1 < c < 2$.
  - Some unit cost u per link leads to $T_{threaded} \leq 2 \cdot u \cdot n$
    - Since our unit cost u here is simple dereferencing, it is usually in the nanoseconds (4-5 CPU cycles).

Analysis

- Unfortunately, in link inversion, ***every one*** of the $2n$ links total will be mutated (changed) twice by the pair `(curr,prev)`!
  - For example, if it's a link pointing right, we will scan it "downward" with `descendToRight()` and "upwards" in `ascendFromRight()`
  - With assumed unit cost u, this leads to to $T_{\ell ink\_inversion} = 2 * 2 \cdot u \cdot n = 4 \cdot u \cdot n = 2 * T_{threaded\_tree}$

Analysis

- Algo also mutates the tree, so it needs to restore it if the application gets interrupted.
- Big benefit over threaded trees: **Modularity** for other traversals
  - **Inorder**: Sorted order
  - **Pre-order**: best for **search** and for other applications, e.g create a carbon copy of a tree.
  - **Post-order**: Used by every garbage collector out there (freeing leaves before parents).

Robson traversal

- Runs in constant storage!
- Space requirements: 5 pointers (so 40 bytes).
- We will not cover it
  - Past honor's project; can share student solution / presentation.
- Implemented in almost all garbage collectors out there.