Verification of Heap Manipulating with Ordered Data Extended Forest Automata

BRNO UNIVERSITY OF TECHNOLOGY
FIT FACULTY OF INFORMATION TECHNOLOGY

Lukas Holik

Ondrej Lengal

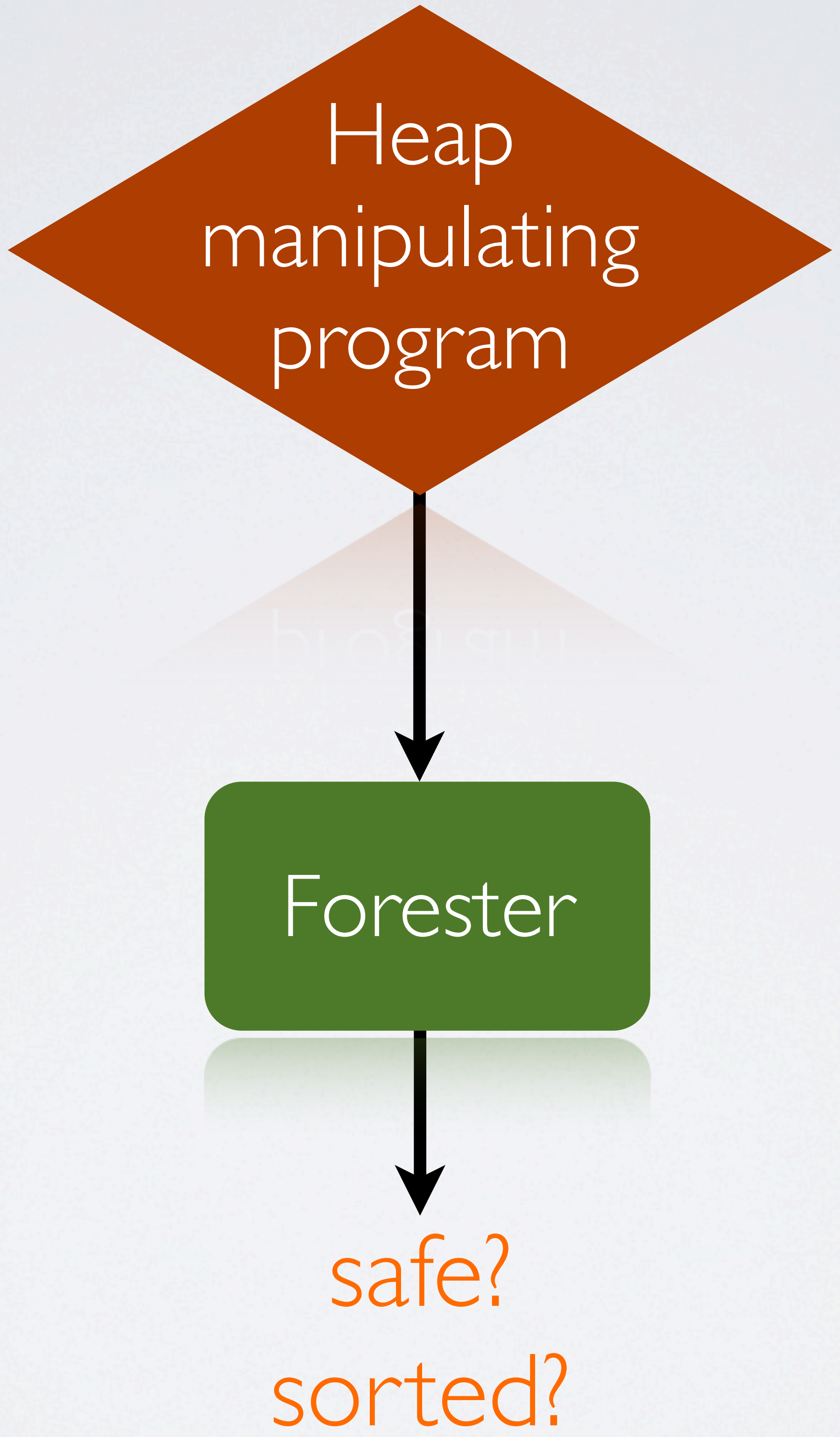
Tomas Vojnar

UPPSALA UNIVERSITET

Parosh Aziz Abdulla

Bengt Jonsson

Cong Quy Trinh


Heap manipulating program
Forester
safe?
sorted?

Heap manipulating program

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

Unbounded heaps   Data dependence   Multiply selectors

Heap manipulating program

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

Unbounded heaps   Data dependence   Multiply selectors

Heap manipulating program

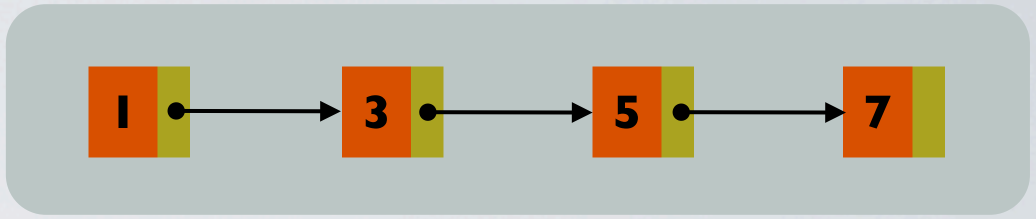
```
Node *insert(Node *root, Data d){
      Node* newNode = calloc(sizeof(Node));
      if (!newNode) return NULL;
      newNode→data = d;
      if (!root) return newNode;
      Node *x = root;
      while (x→data != newNode→data)
            if (x→data < newNode→data)
                  if (x→right) x = x→right;
                  else x→right = newNode;
            else
                  if (x→left) x = x→left;
                  else x→left = newNode;
      if (x != newNode)
            free(newNode);
      return root;
}
```

Unbounded heaps

Data dependence

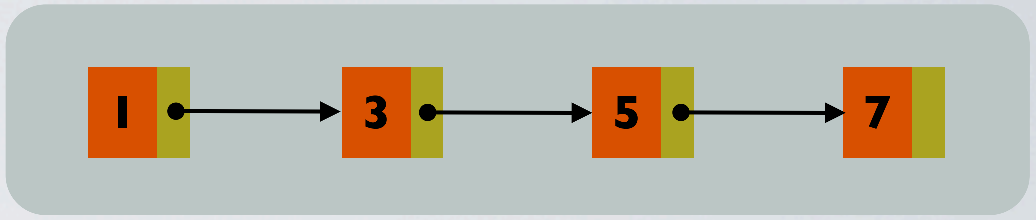
Multiply selectors

Heap manipulating program

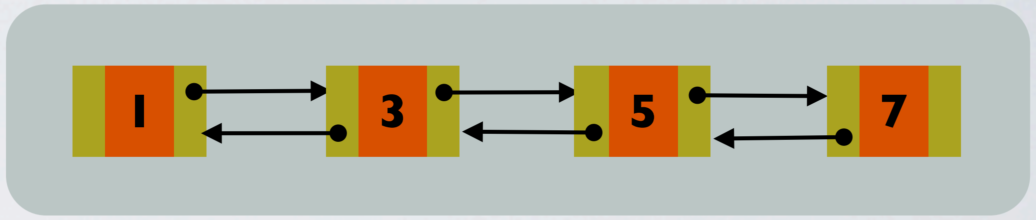


Singly linked lists

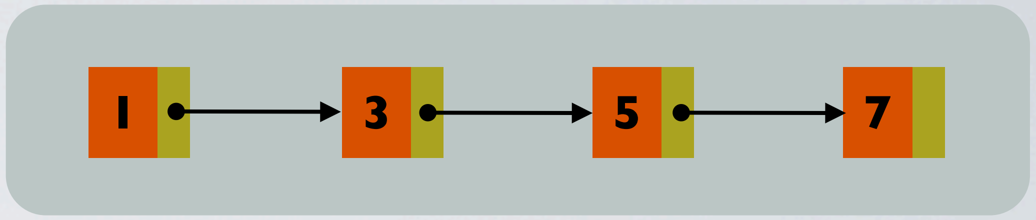
Heap manipulating program



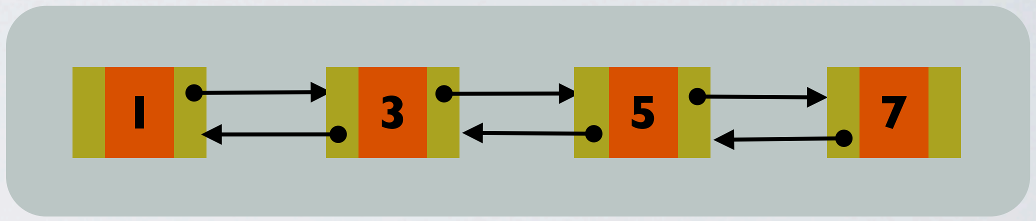
Singly linked lists



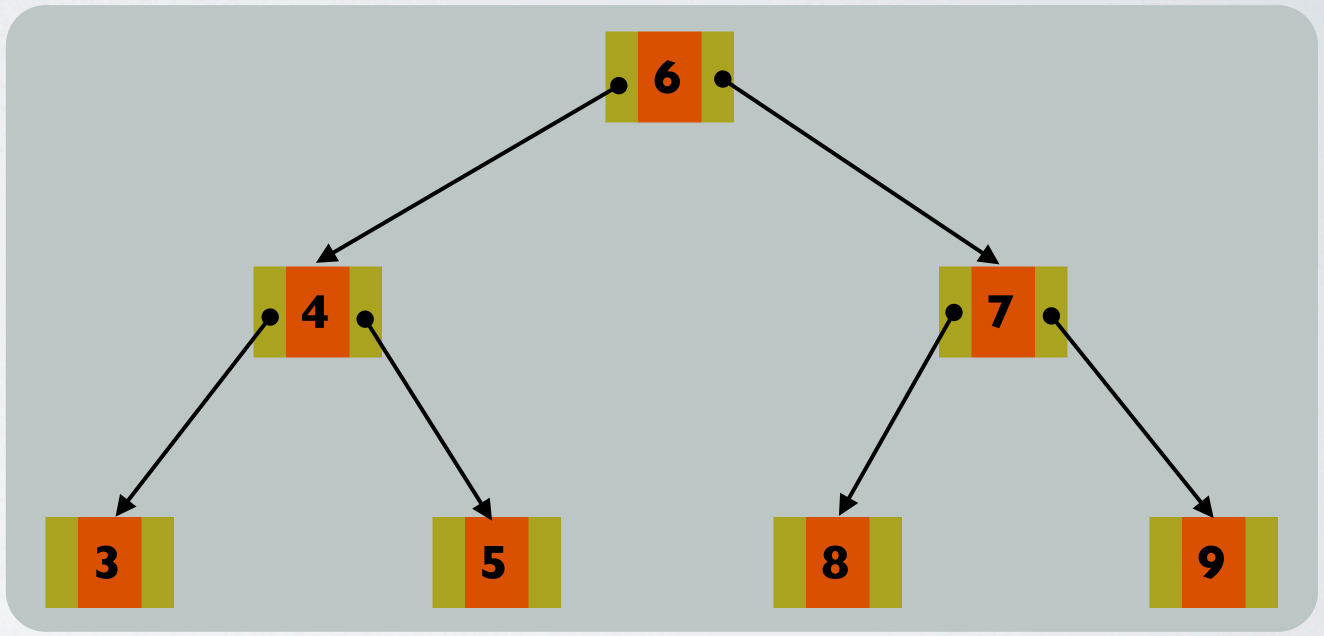
Doubly linked lists



Singly linked lists

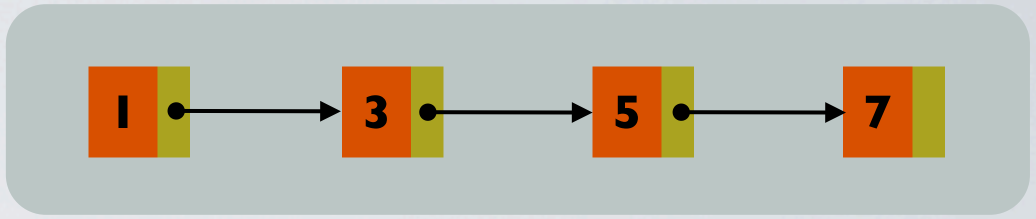


Doubly linked lists

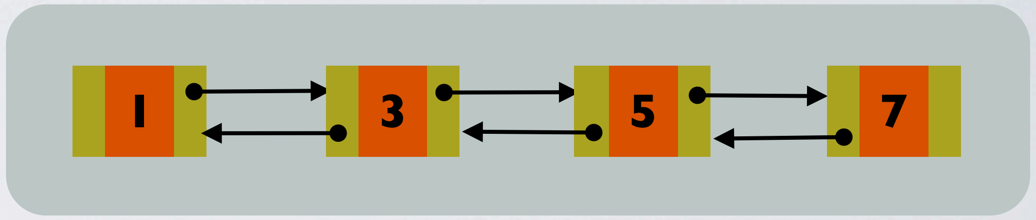


Binary Search Trees

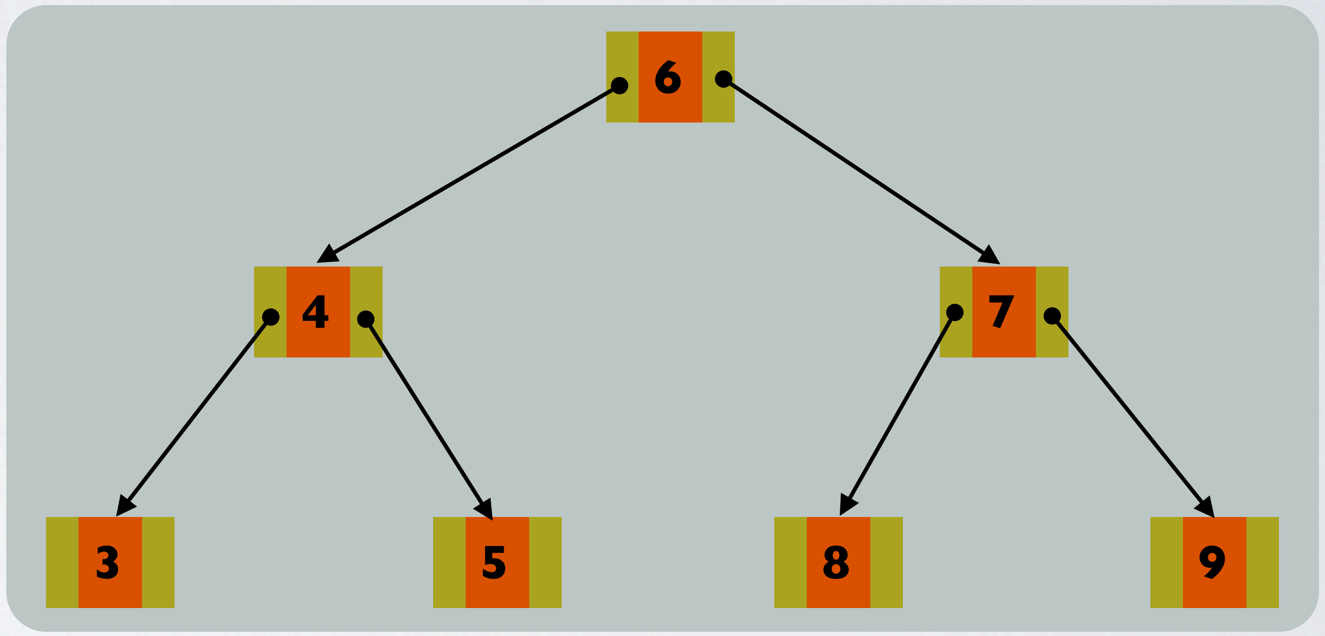
Heap manipulating program



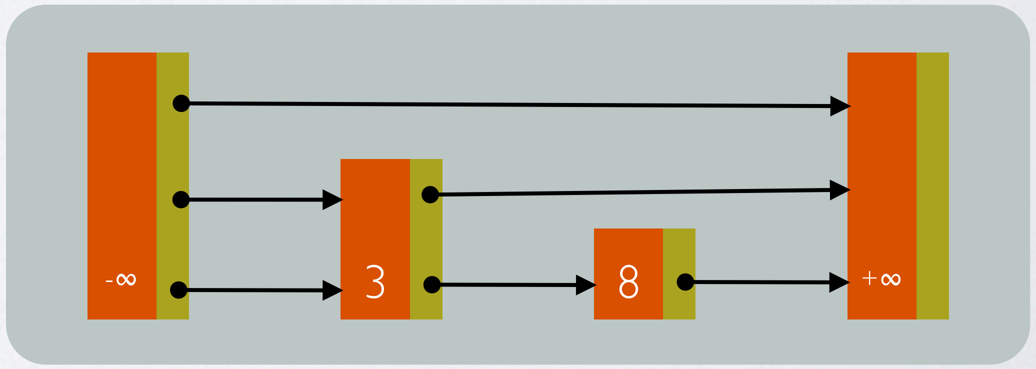
Singly linked lists



Doubly linked lists



Binary Search Trees



Skip-lists

Forester

How does it work?

Program analysis

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

Program analysis

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left; ●
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

program point

Program analysis

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata

program point

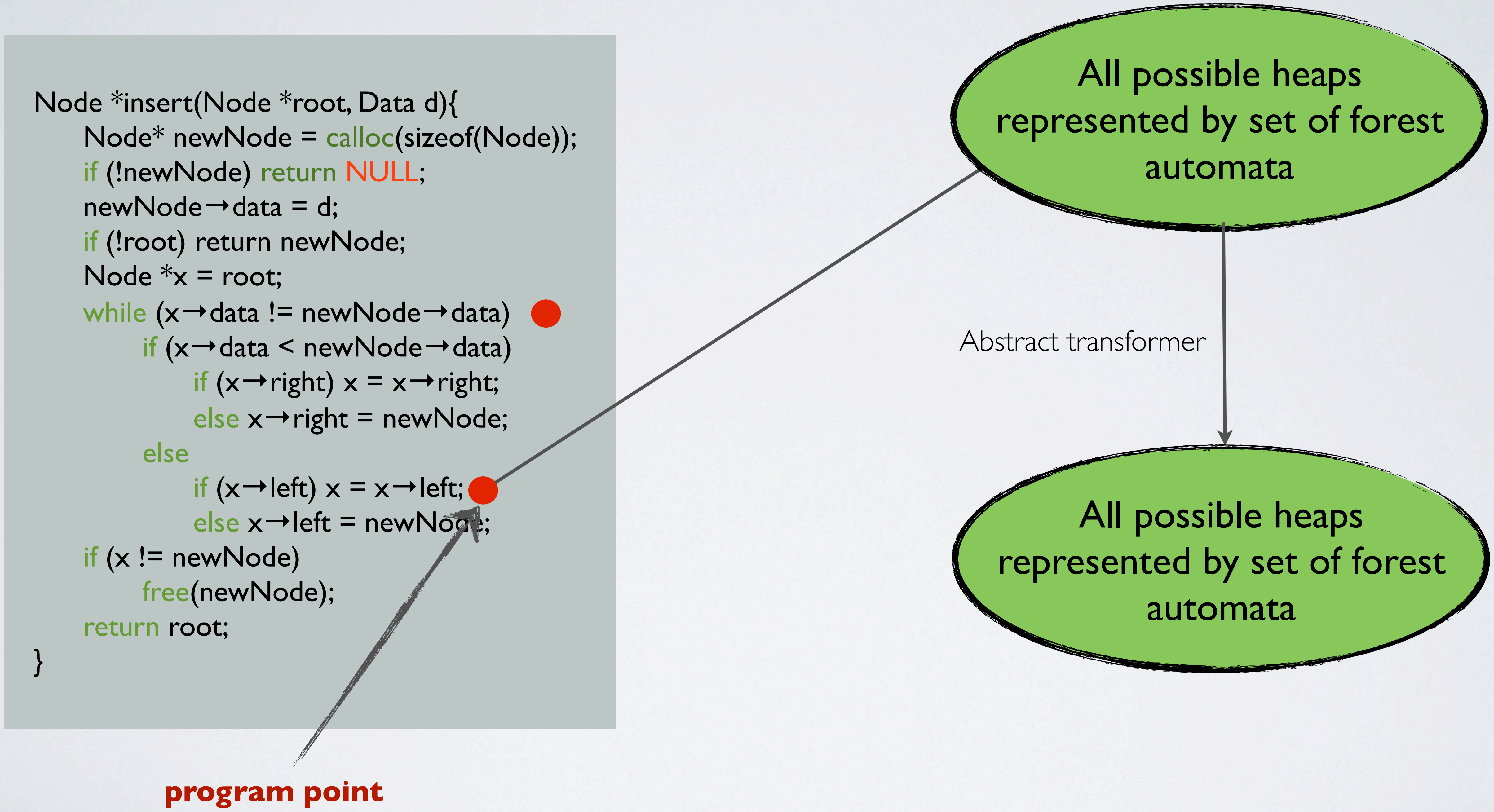
Program analysis

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)  ●
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left; ●
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata

program point

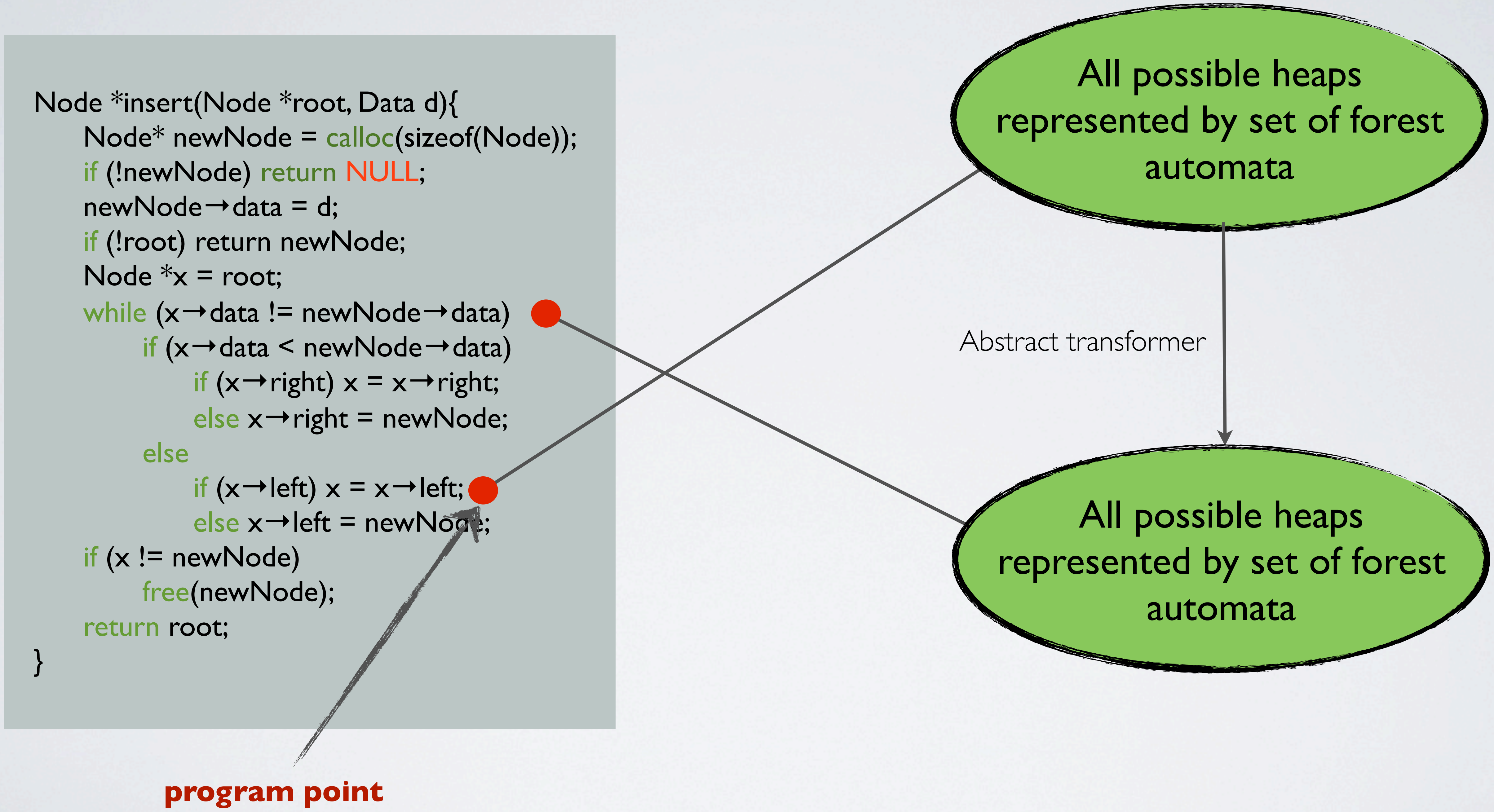
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
program point

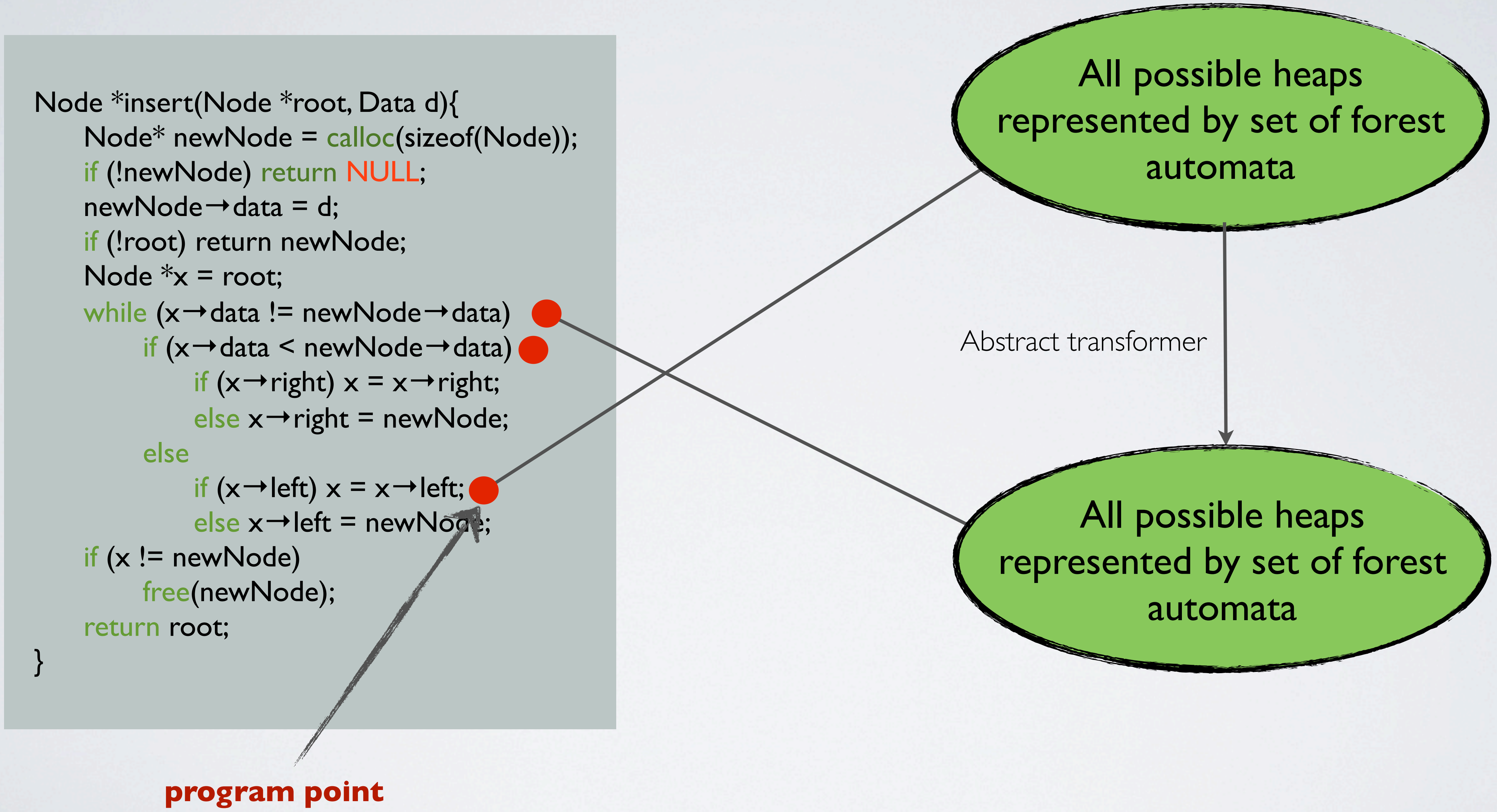
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
program point

Program analysis



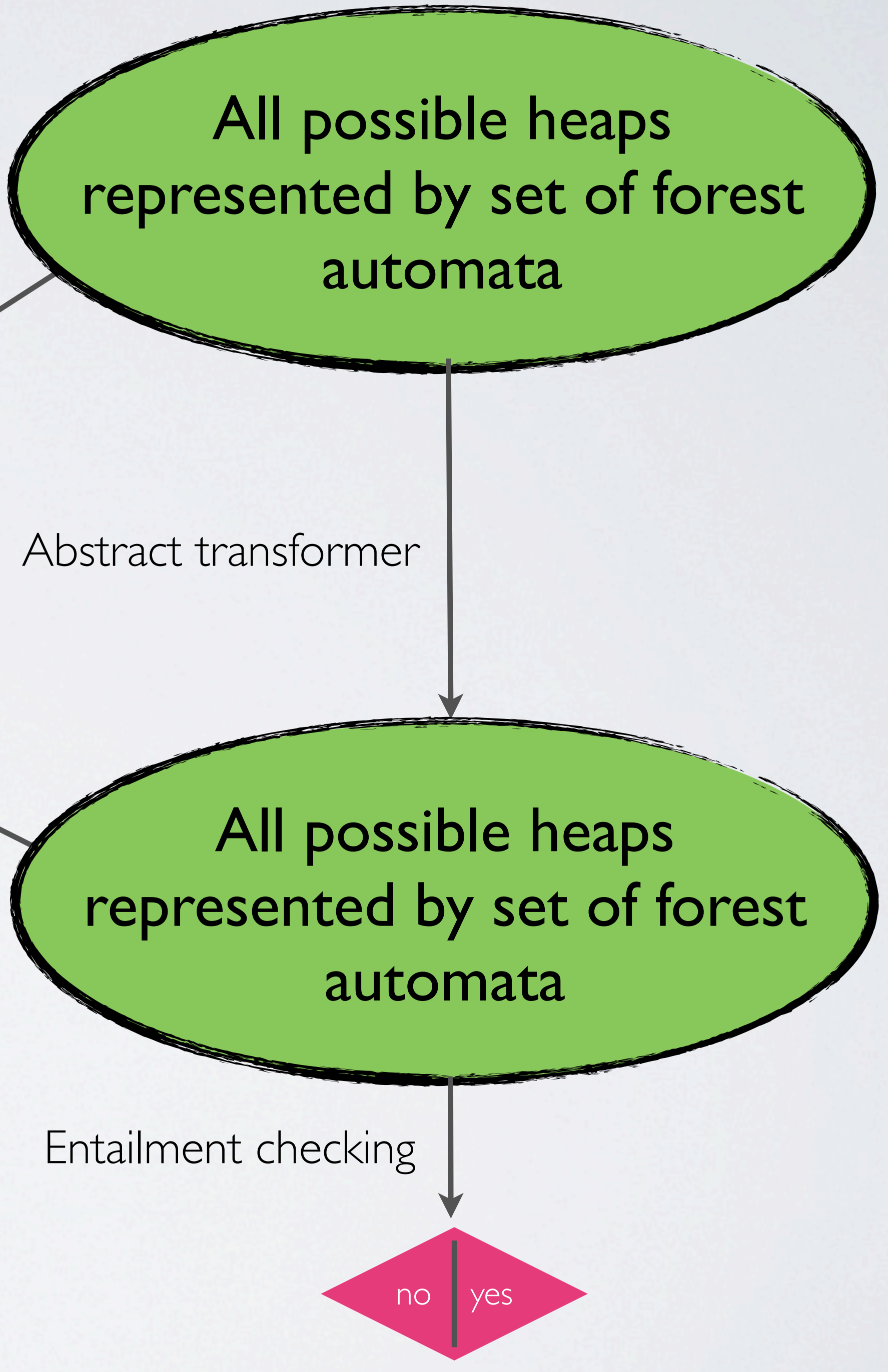
```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
program point

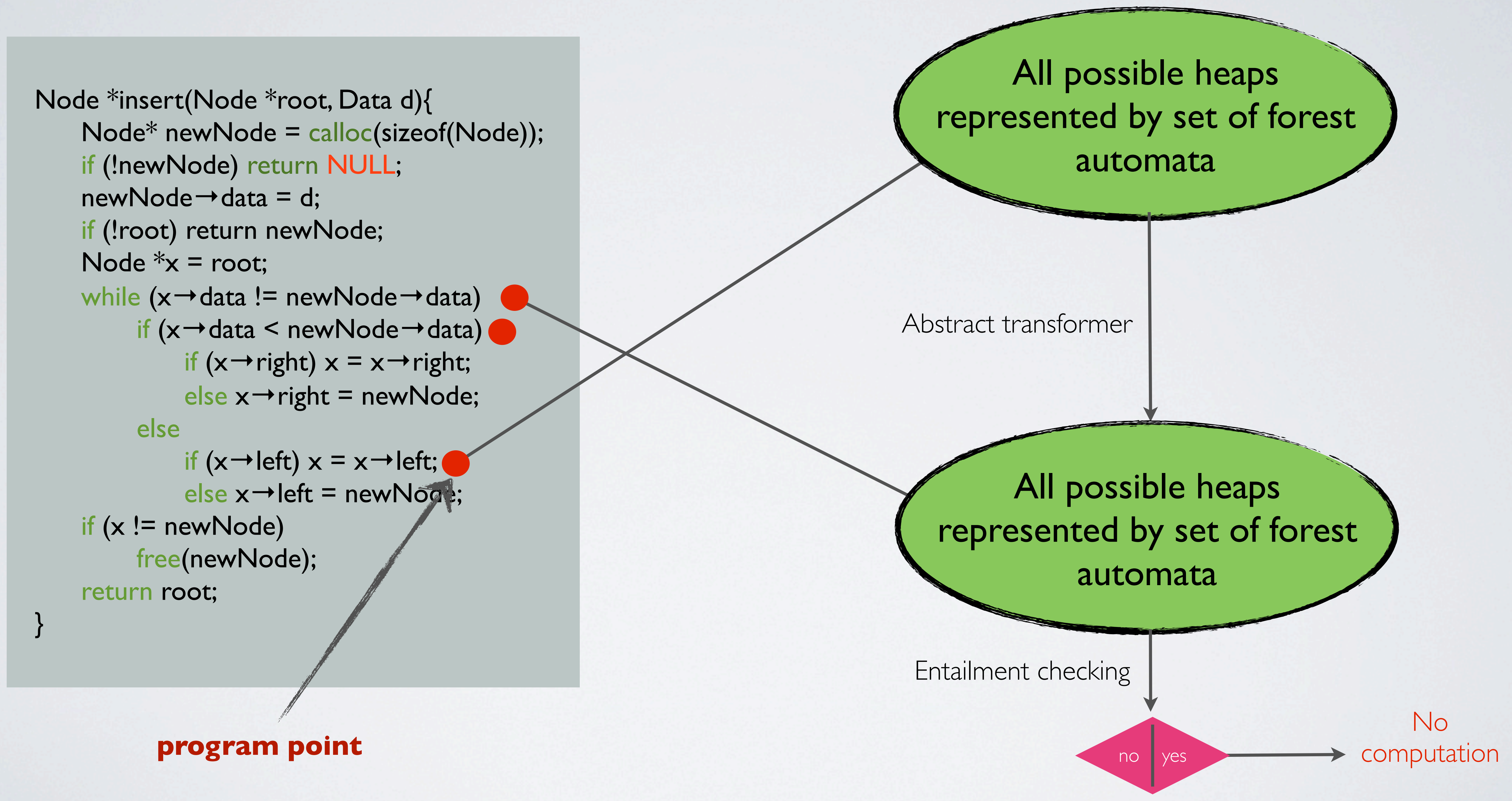
Program analysis

```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data) ●
        if (x→data < newNode→data) ●
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left; ●
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

program point


All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes

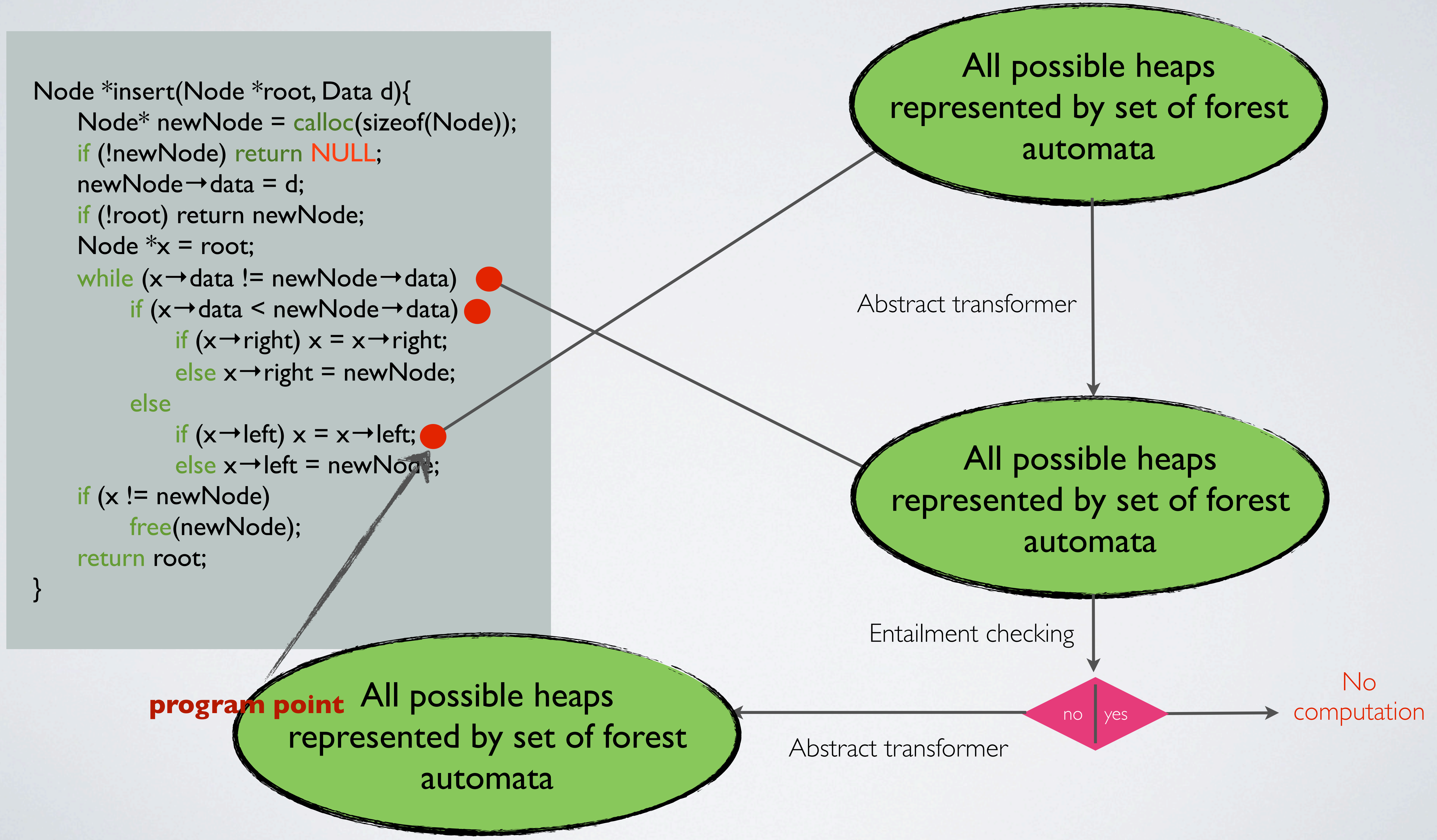
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

program point
All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation

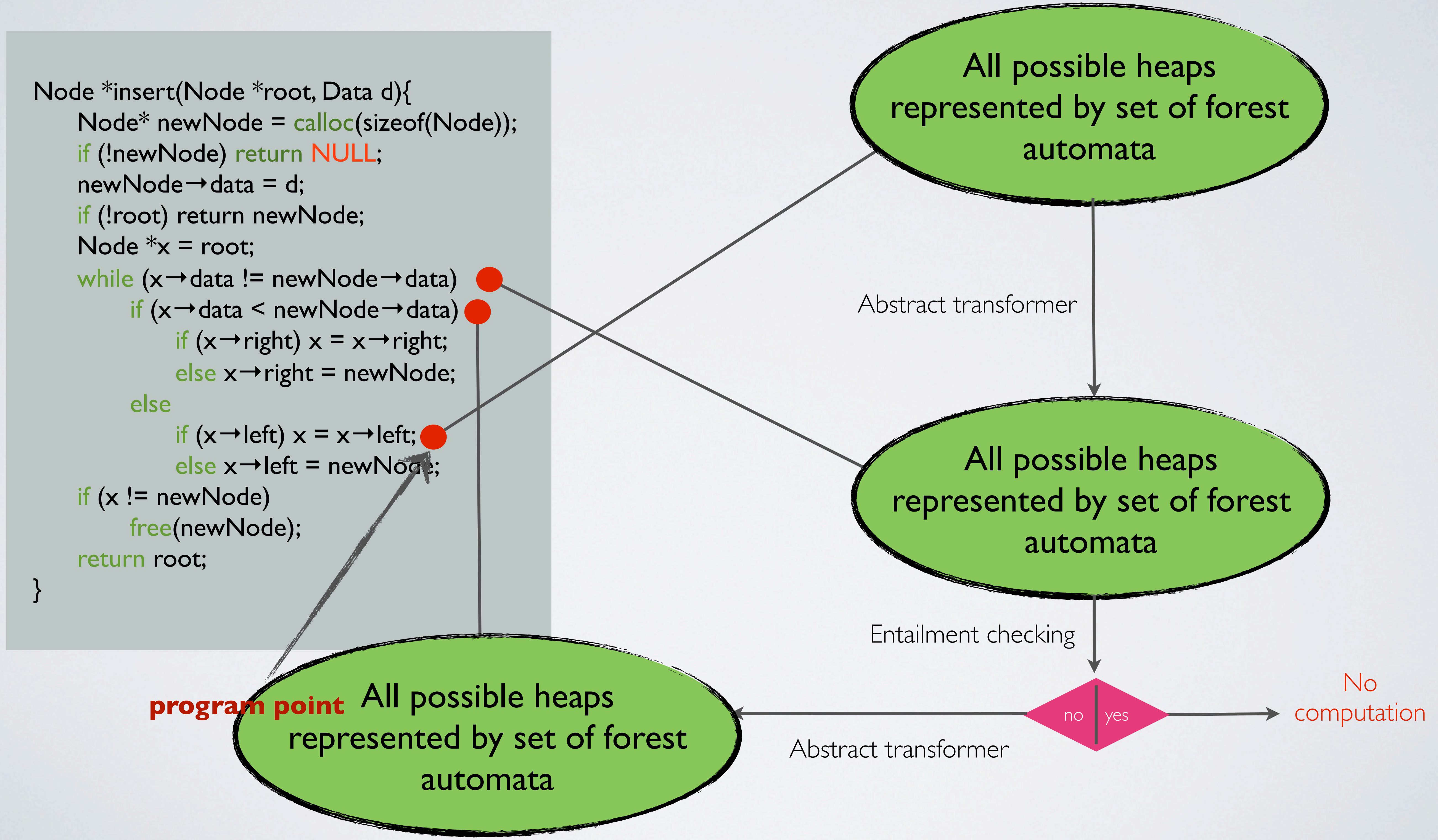
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

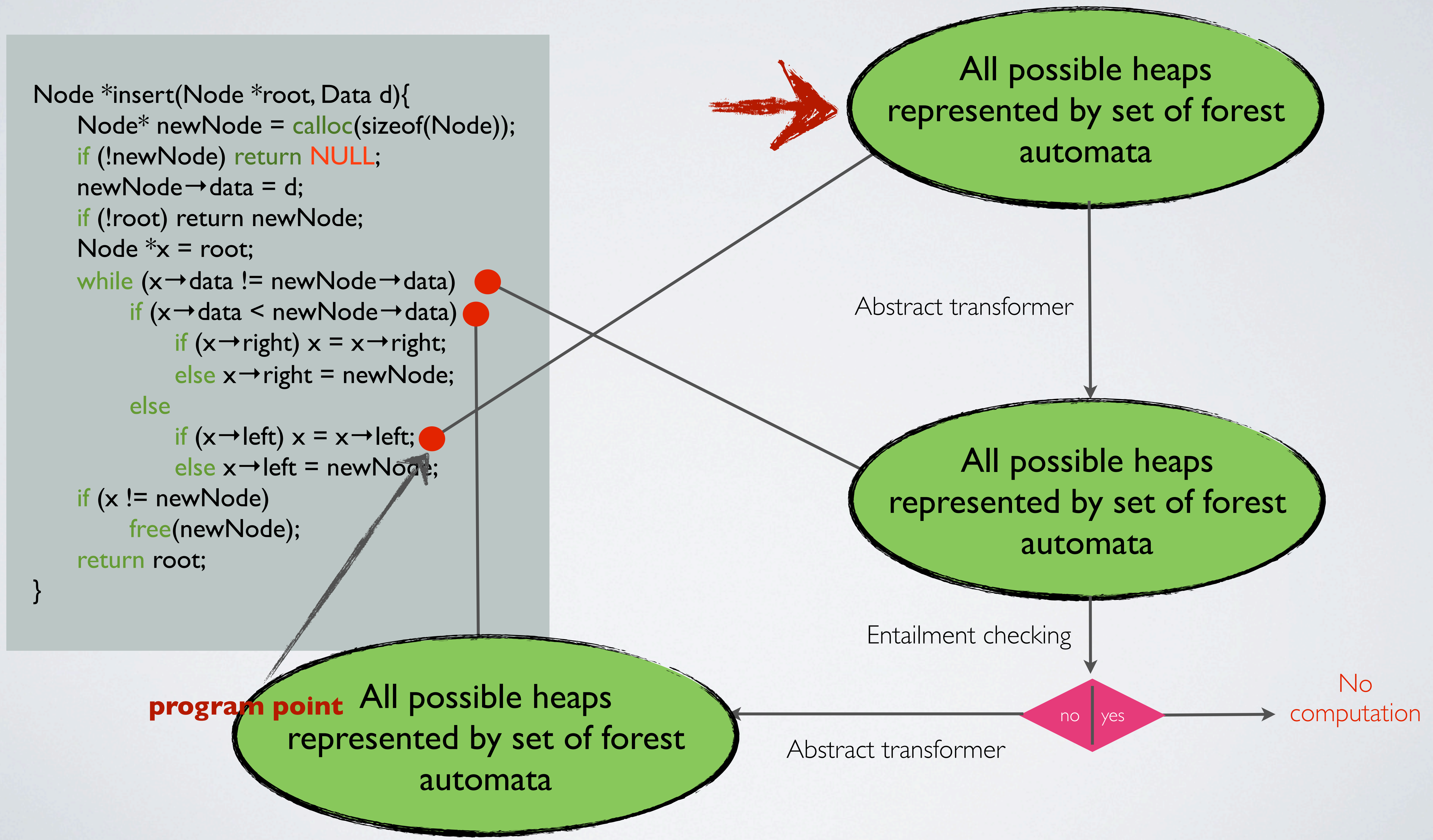
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

Program analysis



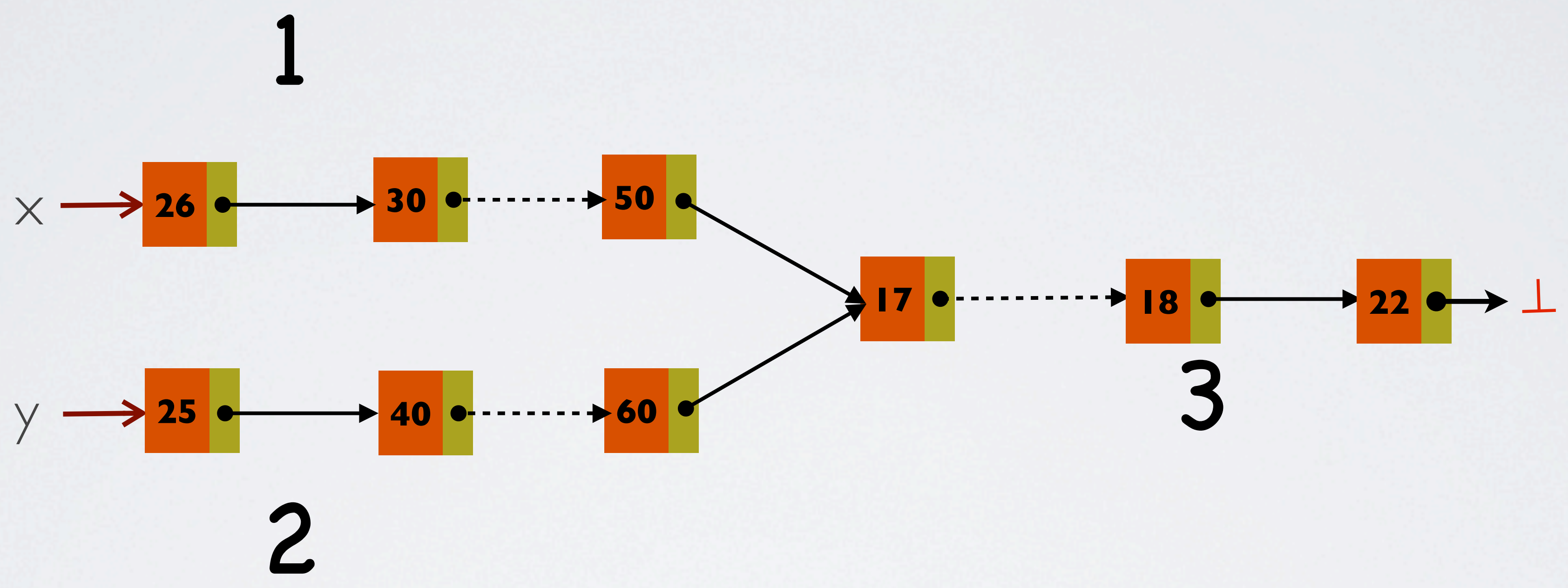
```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

Forest automata representation

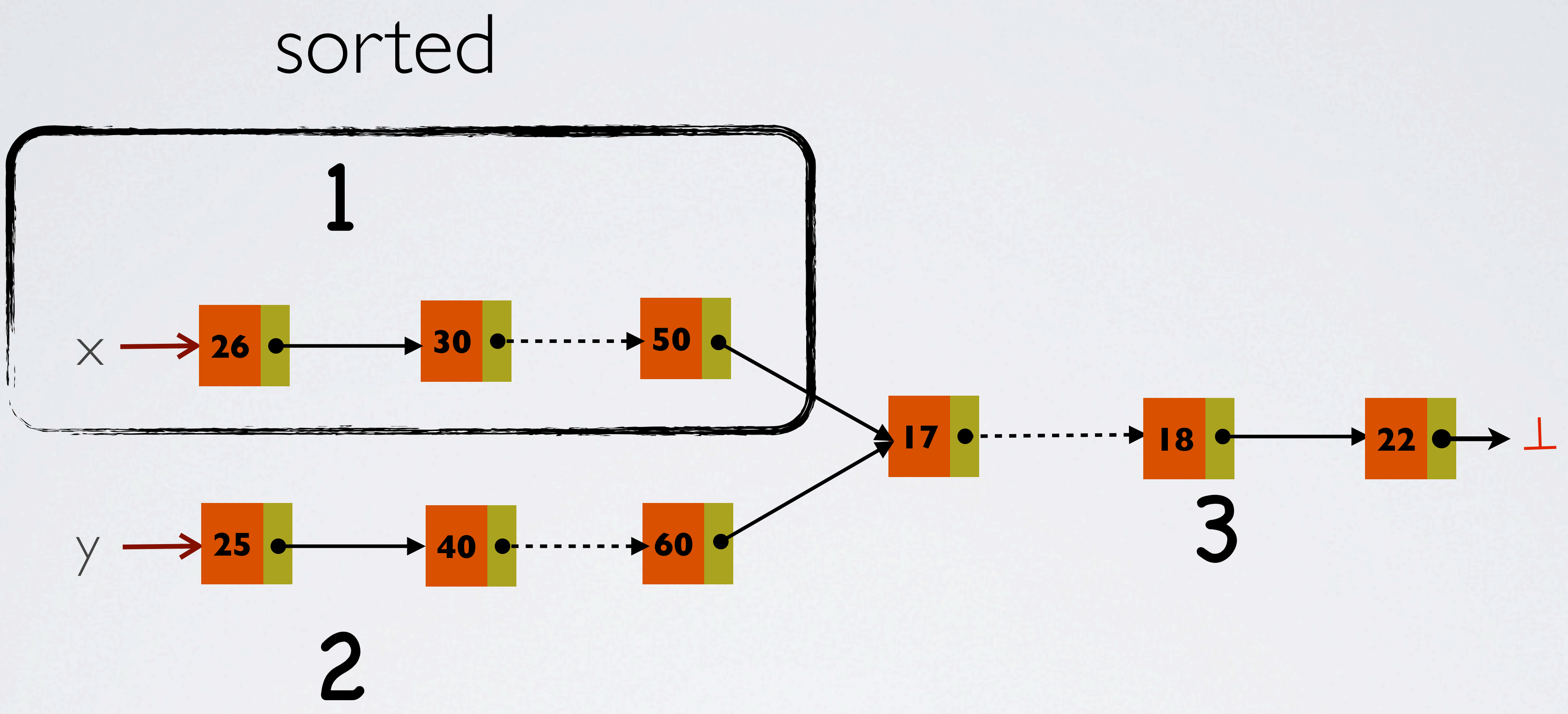
1. Heap representation
2. Decompose heaps into forests
3. Represent forests by forest automata

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata



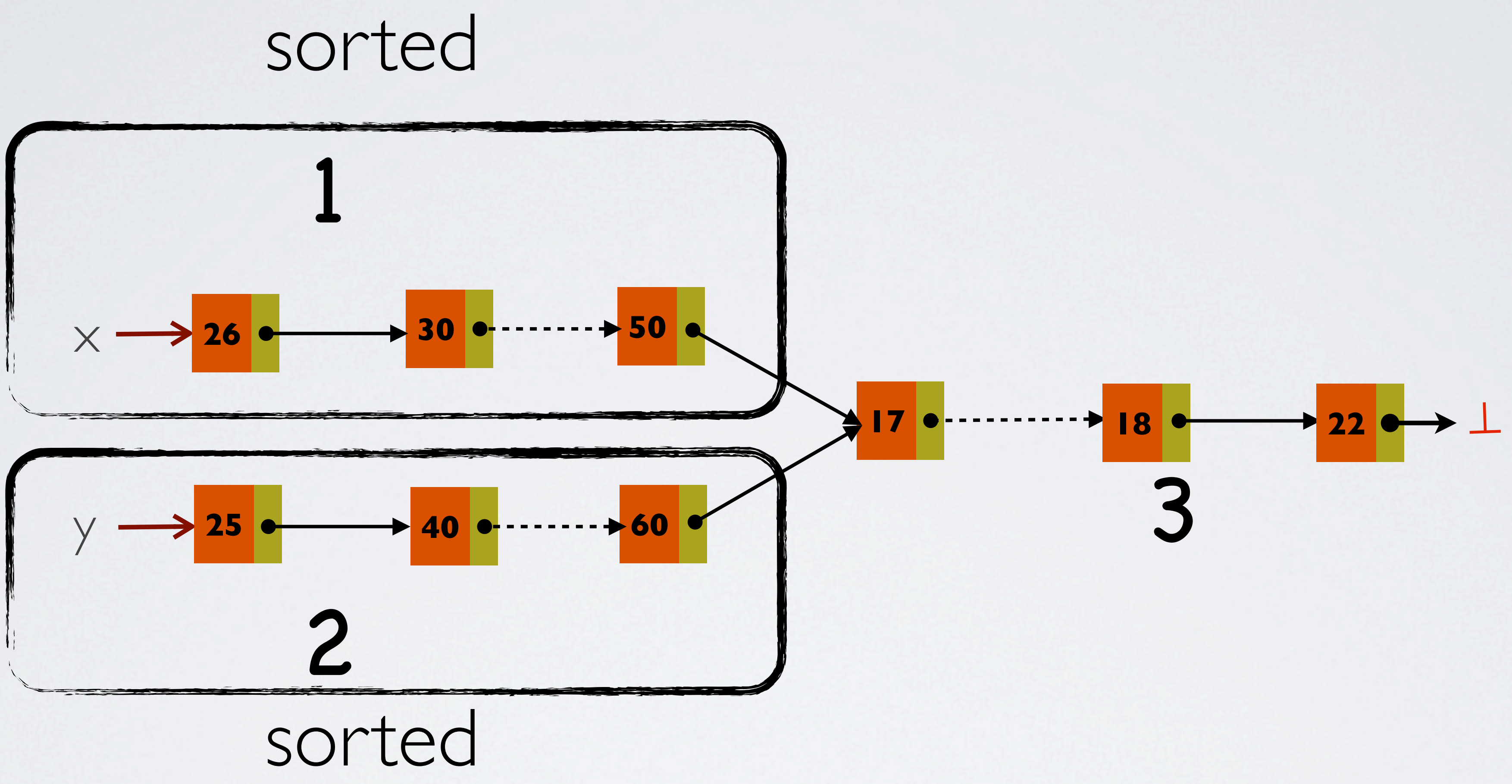
- Set of heaps
- List 1,2,3 are sorted and data of all cells in 3 are bigger than data of all cells in 1 and 2

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata



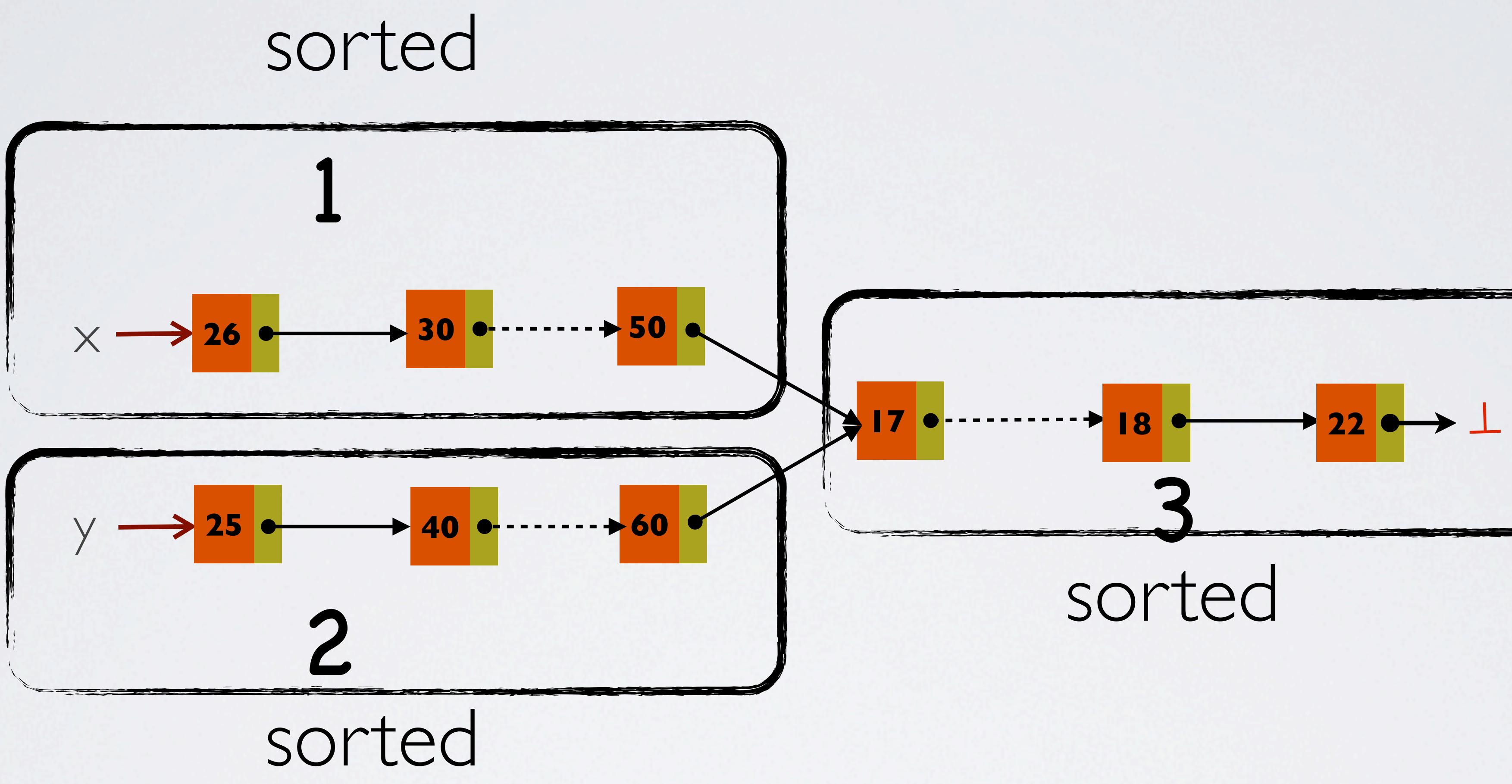
- Set of heaps
- List 1,2,3 are sorted and data of all cells in 3 are bigger than data of all cells in 1 and 2

1. Heap representation   2. Decompose heaps into forests   3. Represent forests by forest automata



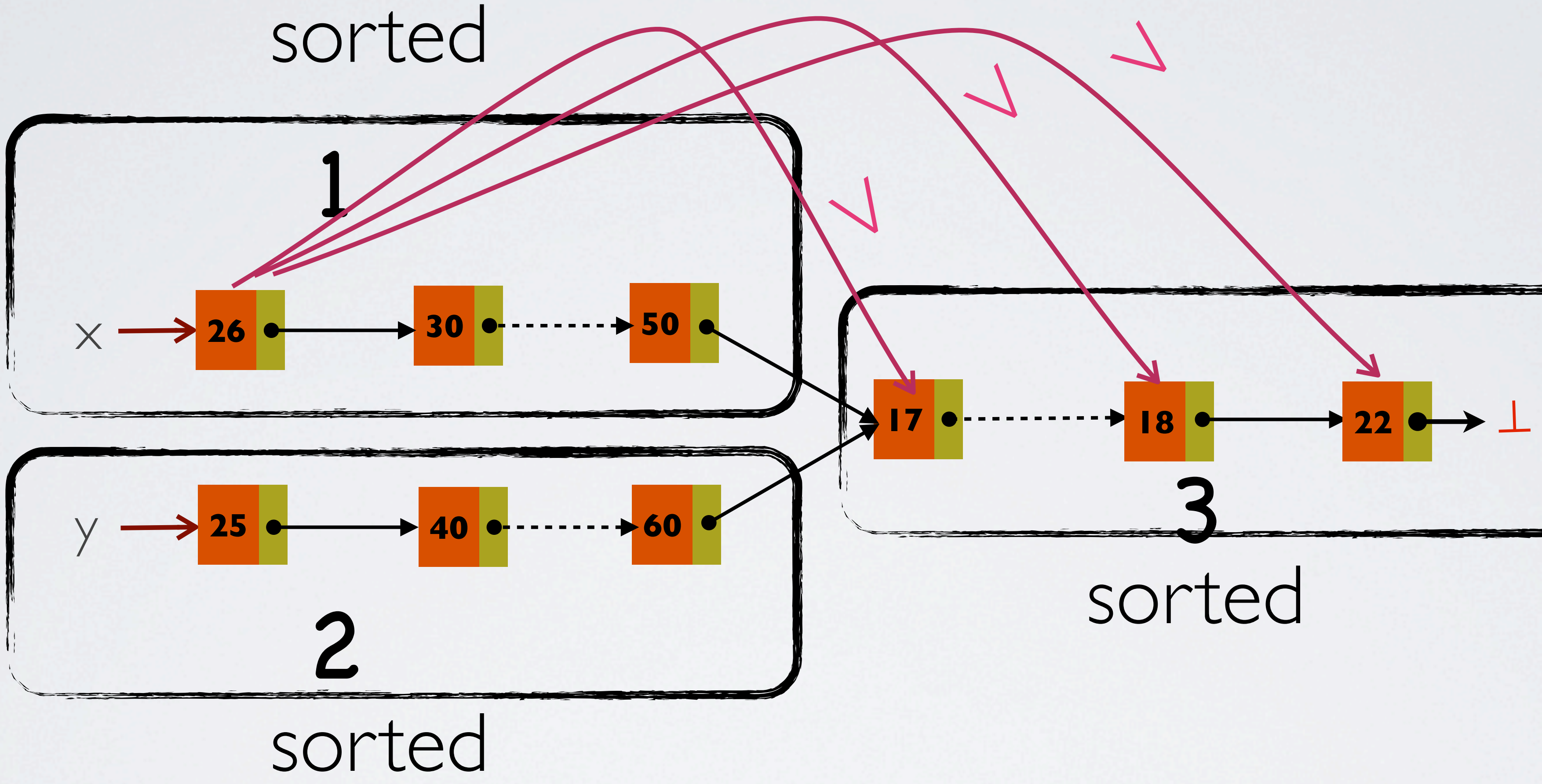
- Set of heaps
- List 1,2,3 are sorted and data of all cells in 3 are bigger than data of all cells in 1 and 2

1. Heap representation   2. Decompose heaps into forests   3. Represent forests by forest automata



- Set of heaps
- List 1,2,3 are sorted and data of all cells in 3 are bigger than data of all cells in 1 and 2

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

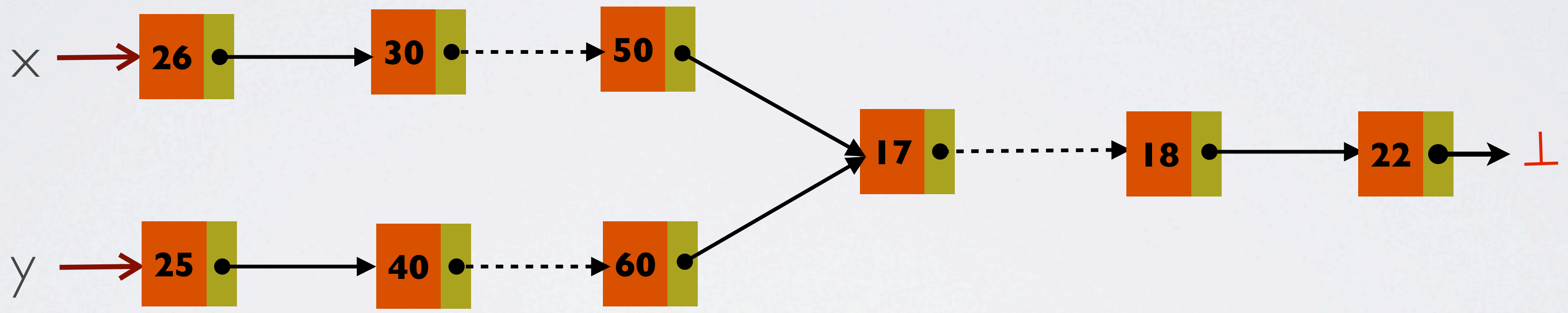


- Set of heaps
- List 1,2,3 are sorted and data of all cells in 3 are bigger than data of all cells in 1 and 2

1. Heap representation

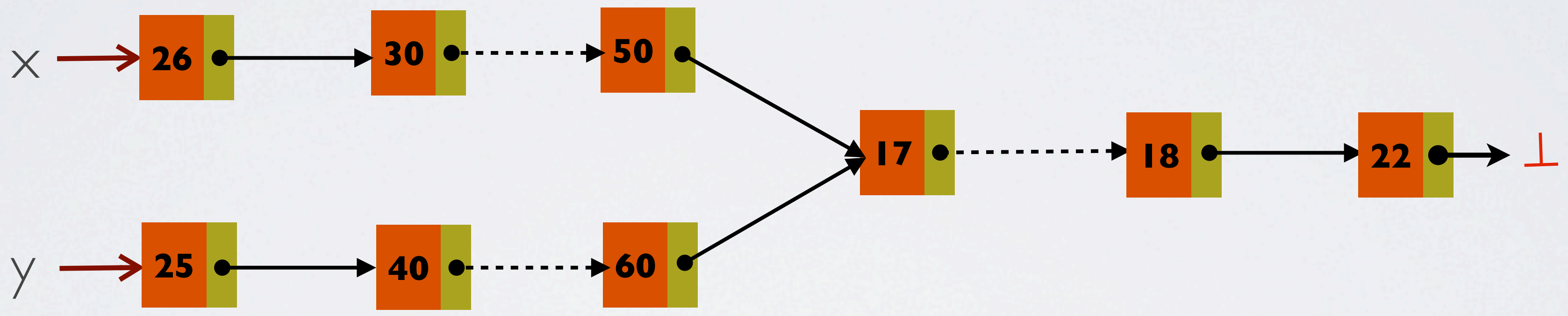
2. Decompose heaps into forests

3. Represent forests by forest automata



1. Heap representation | **2. Decompose heaps into forests** | 3. Represent forests by forest automata

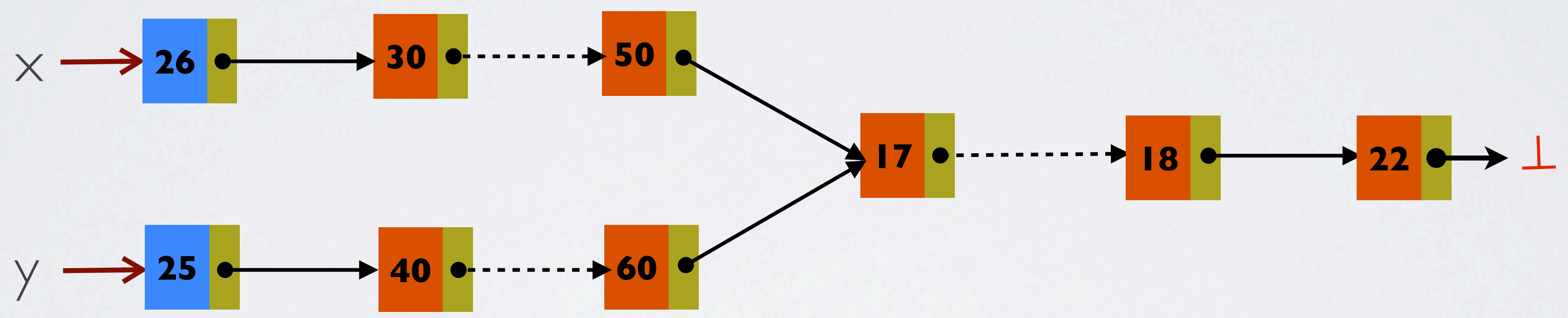
Determine **cut-points:**
- Nodes pointed by variables
- Nodes with more incoming pointers



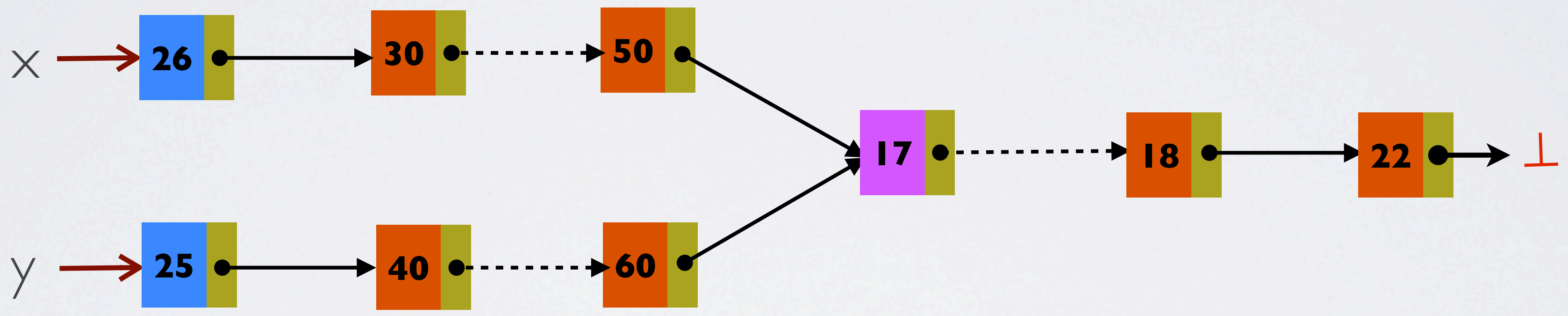
1. Heap representation | **2. Decompose heaps into forests** | 3. Represent forests by forest automata

Determine **cut-points:**

- Nodes pointed by variables
- Nodes with more incoming pointers



Determine **cut-points:**
- Nodes pointed by variables
- Nodes with more incoming pointers



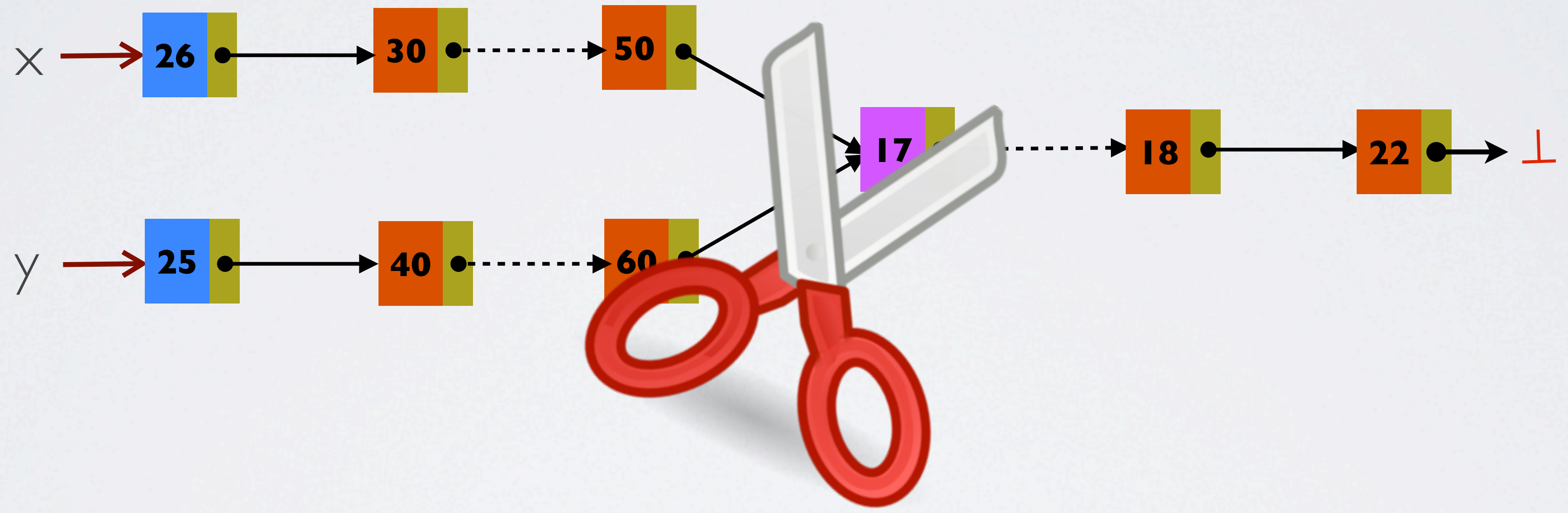
1. Heap representation

2. Decompose heaps into forests

3. Represent forests by forest automata

Determine **cut-points:**

- Nodes pointed by variables
- Nodes with more incoming pointers



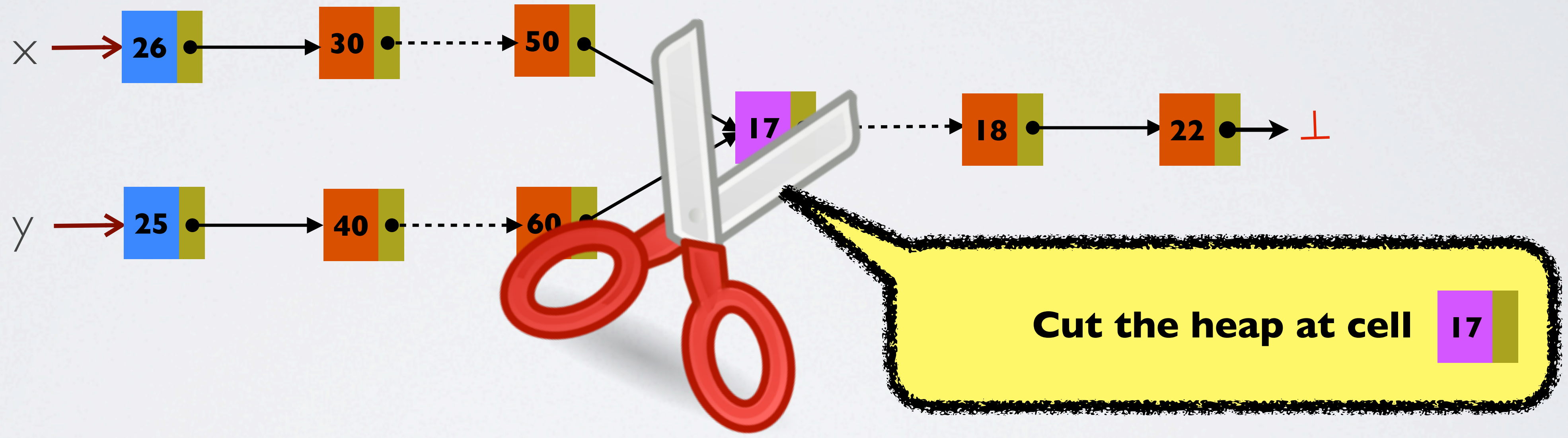
1. Heap representation

2. Decompose heaps into forests

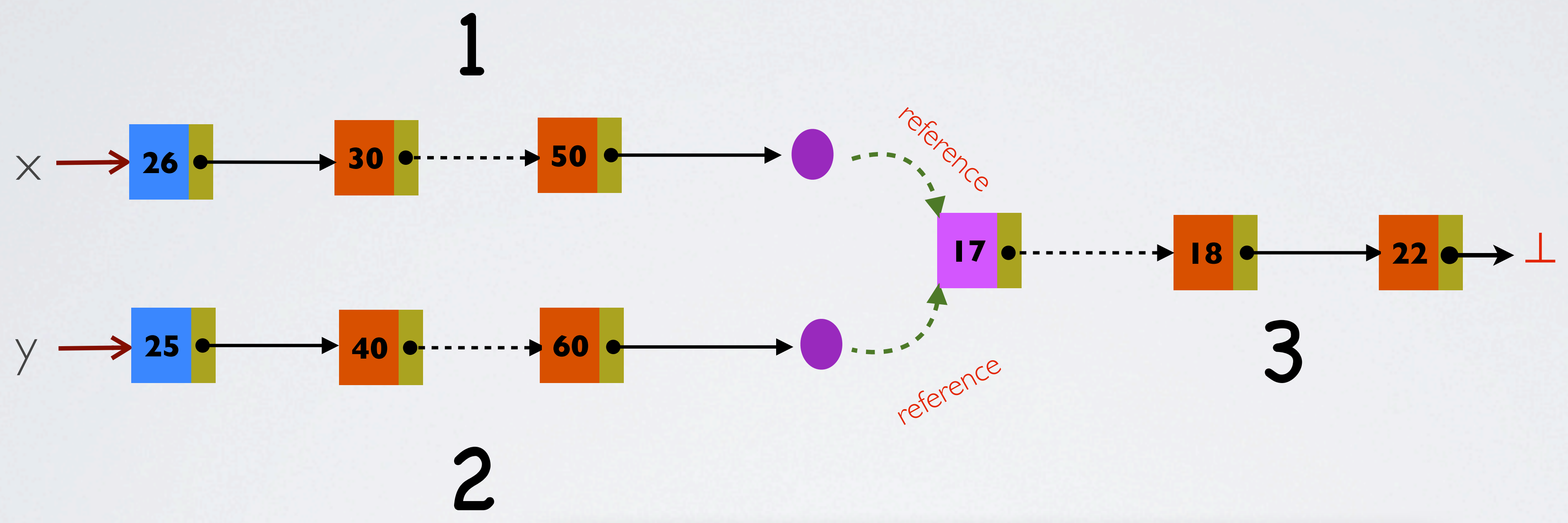
3. Represent forests by forest automata

Determine **cut-points:**

- Nodes pointed by variables
- Nodes with more incoming pointers



1. Heap representation | **2. Decompose heaps into forests** | 3. Represent forests by forest automata

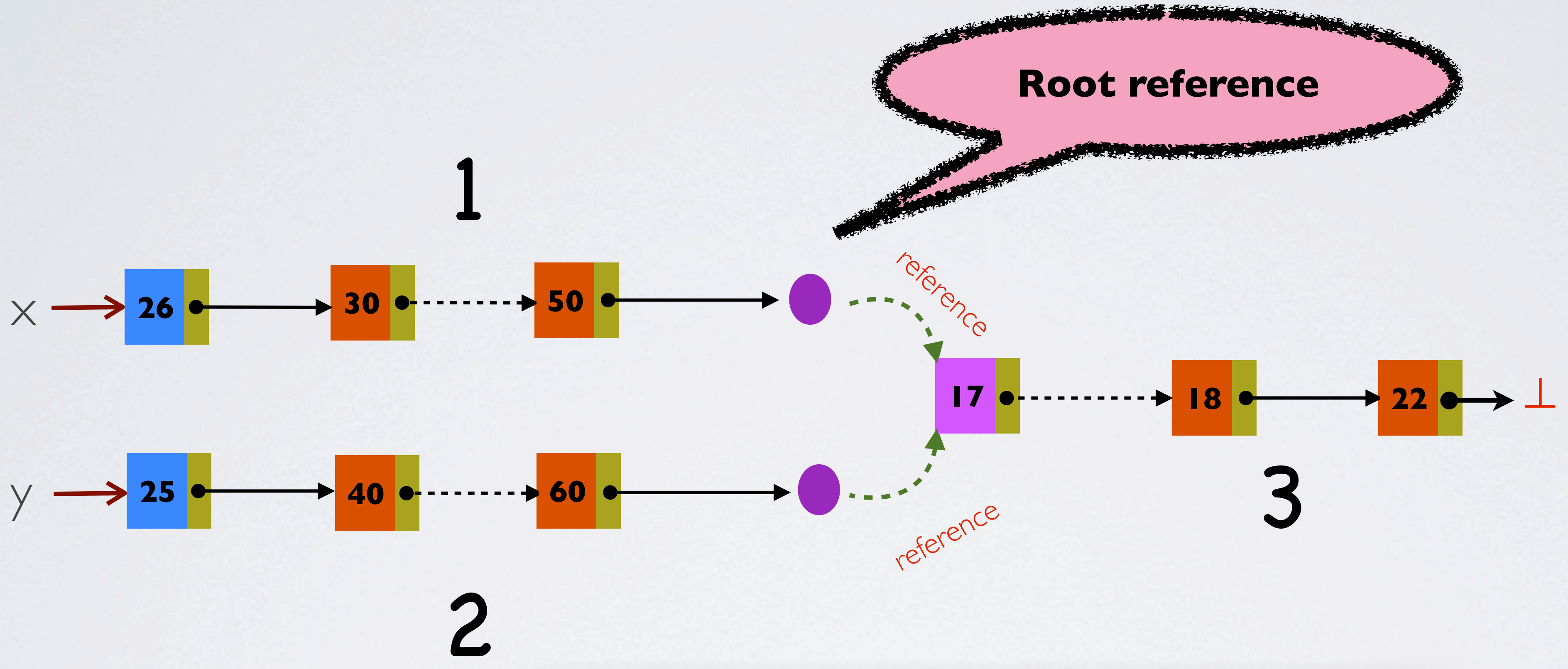


Forest presentation of the set of heaps

1. Heap representation

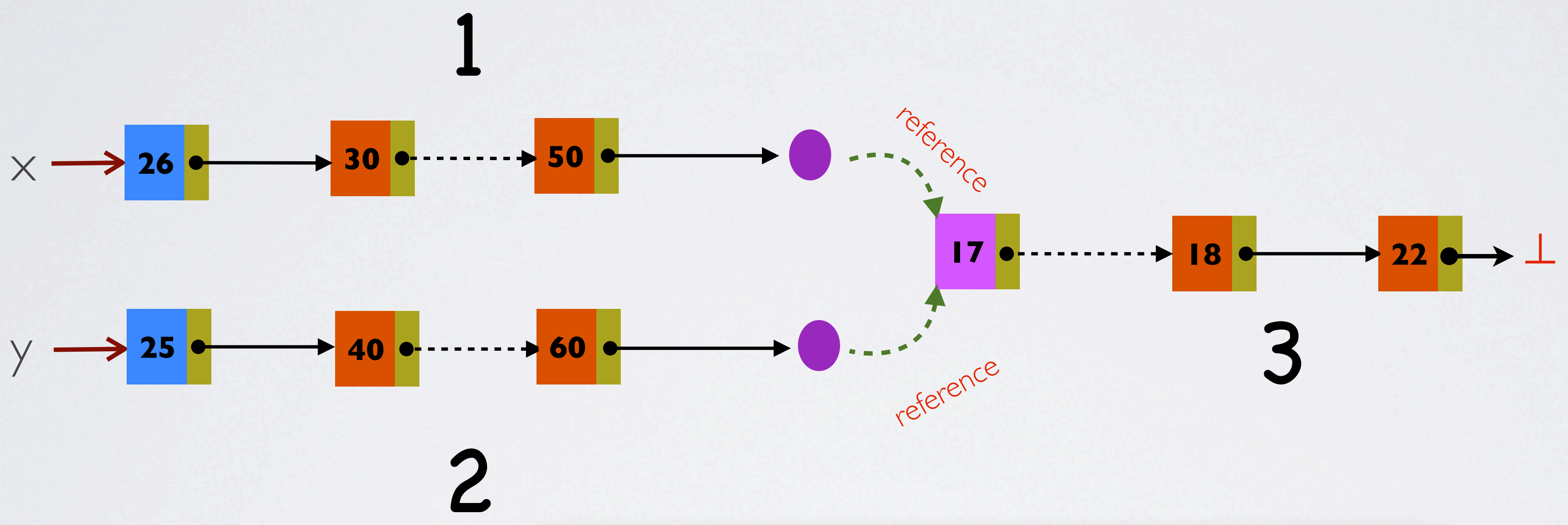
2. Decompose heaps into forests

3. Represent forests by forest automata



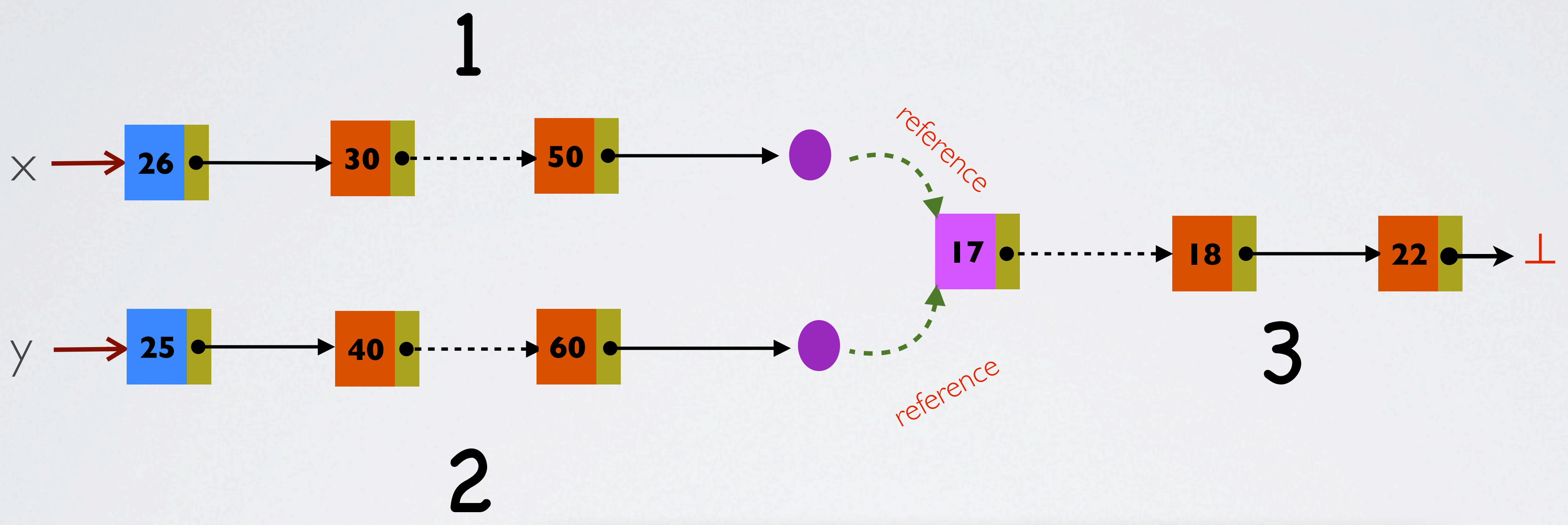
Forest presentation of the set of heaps

1. Heap representation 2. Decompose heaps into forests **3. Represent forests by forest automata**



Forest presentation of the set of heaps

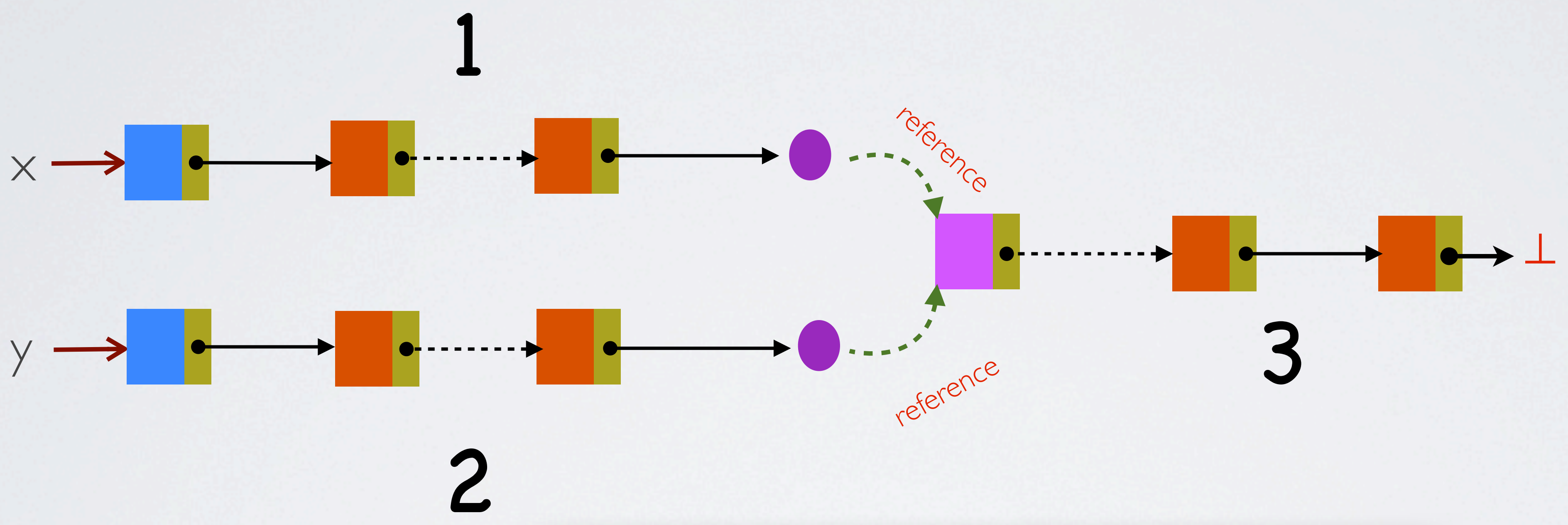
Without data



Forest presentation of the set of heaps

1. Heap representation   2. Decompose heaps into forests   3. Represent forests by forest automata

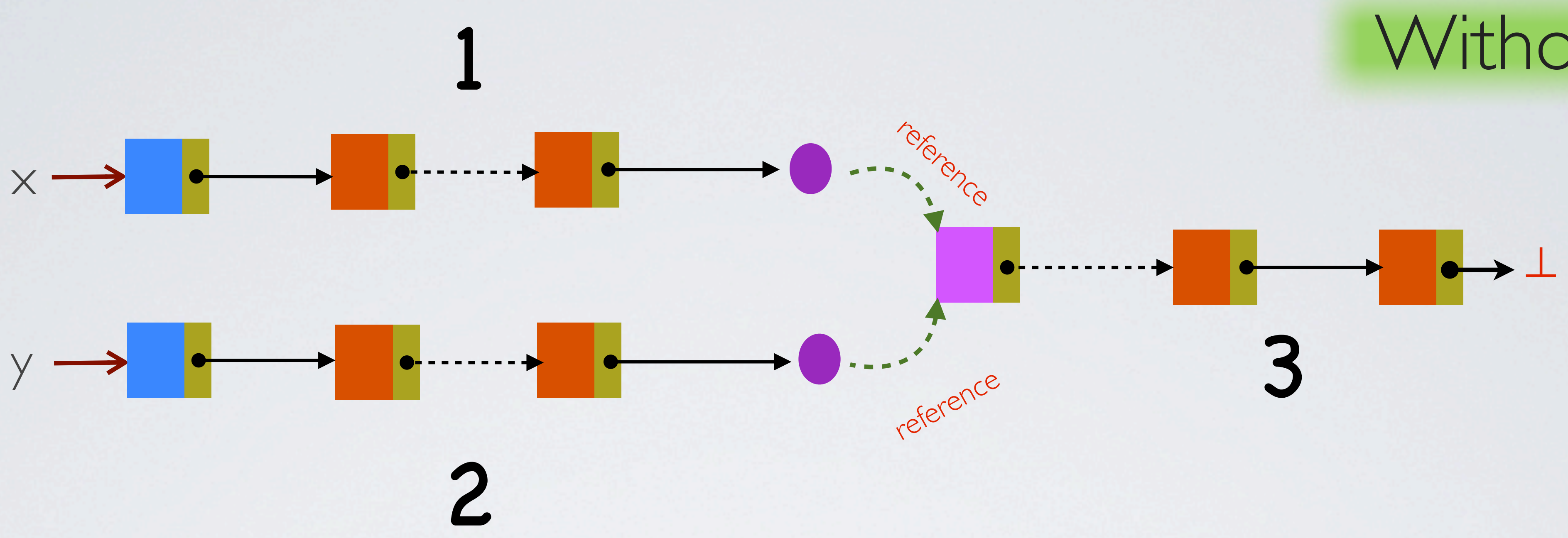
Without data



Forest presentation of the set of heaps

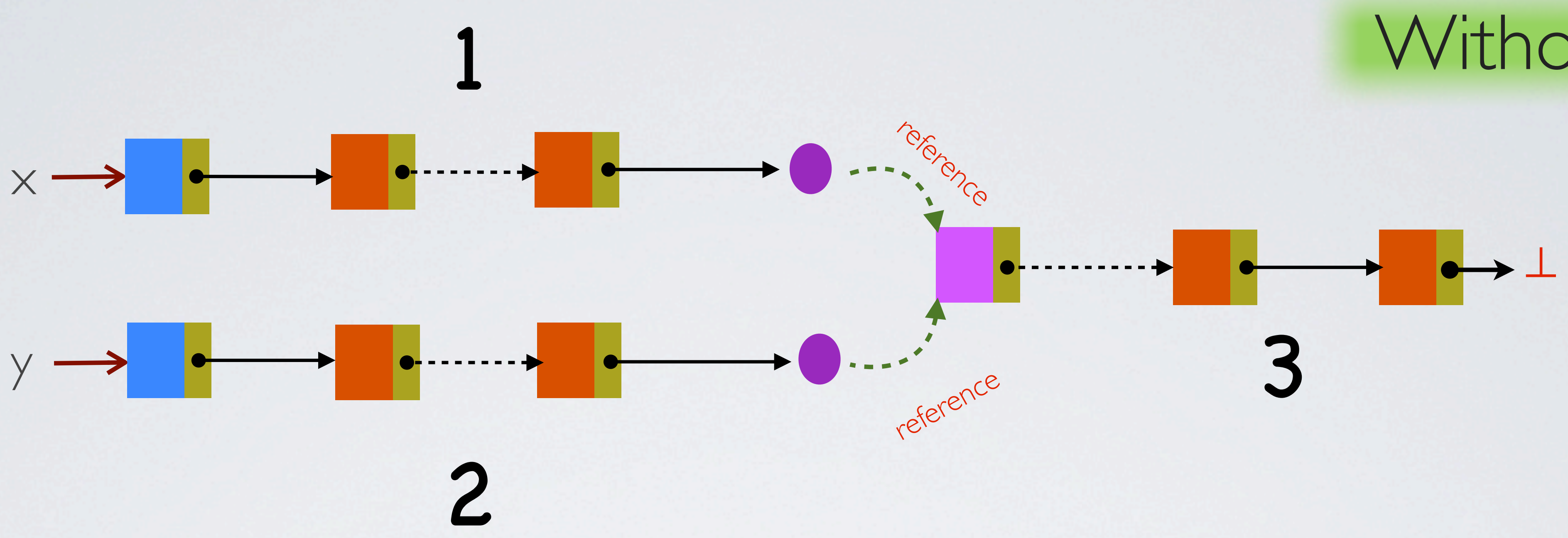
1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

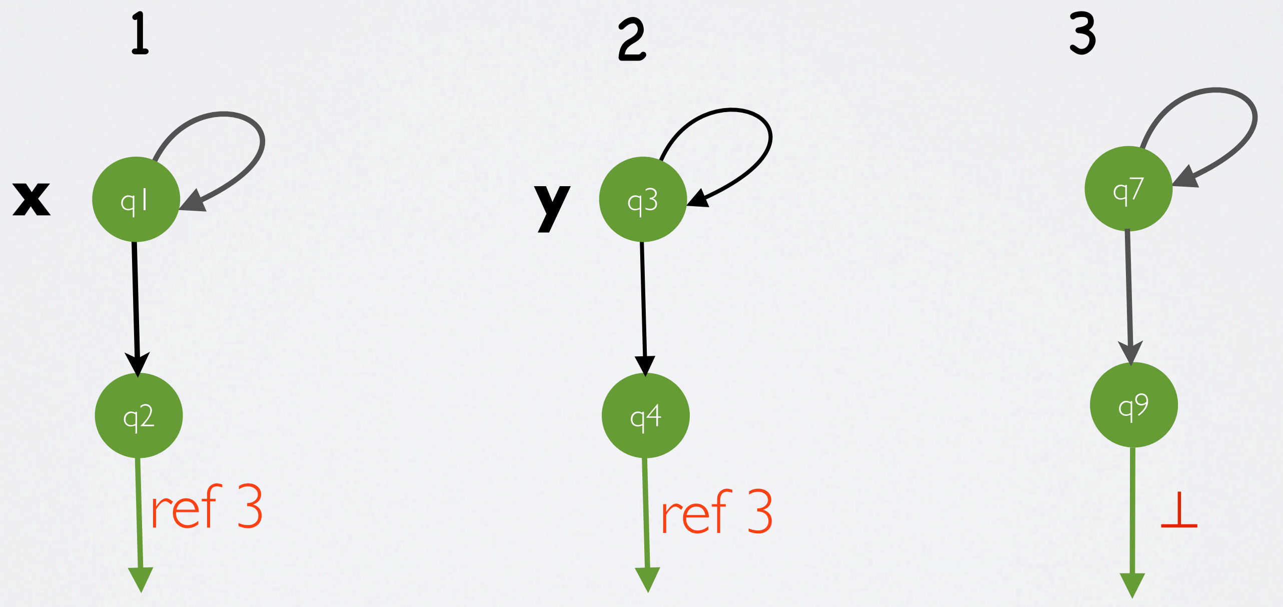
Without data



1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

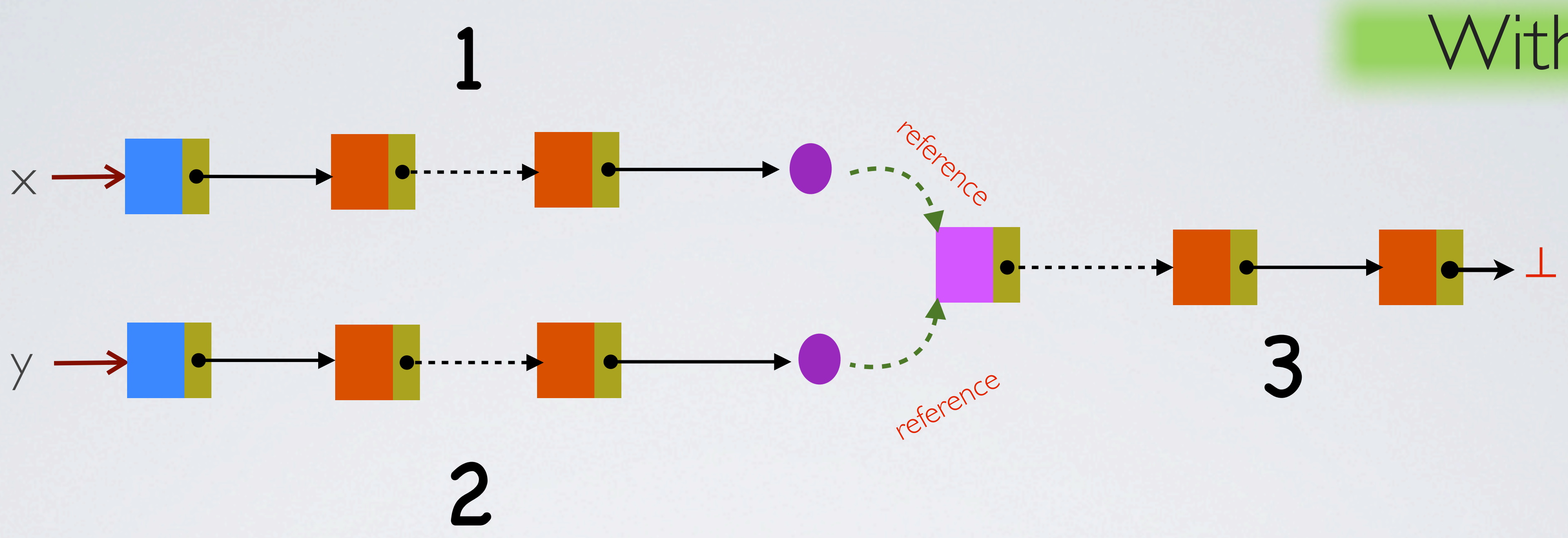
Without data

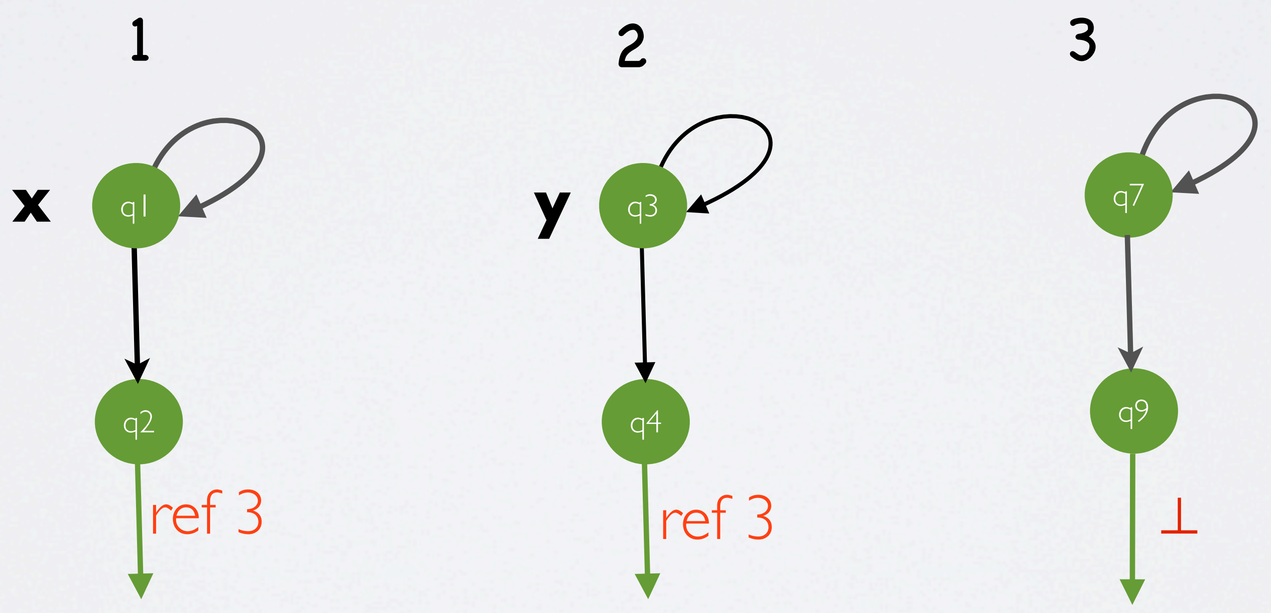




1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

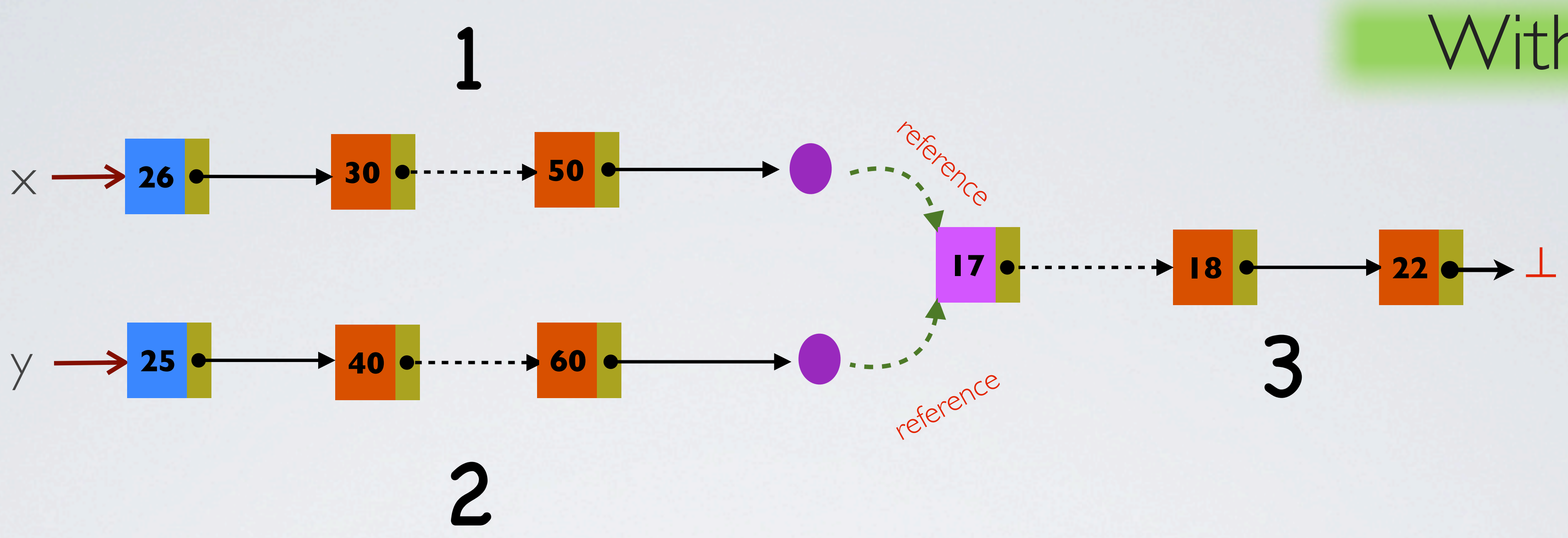
With data

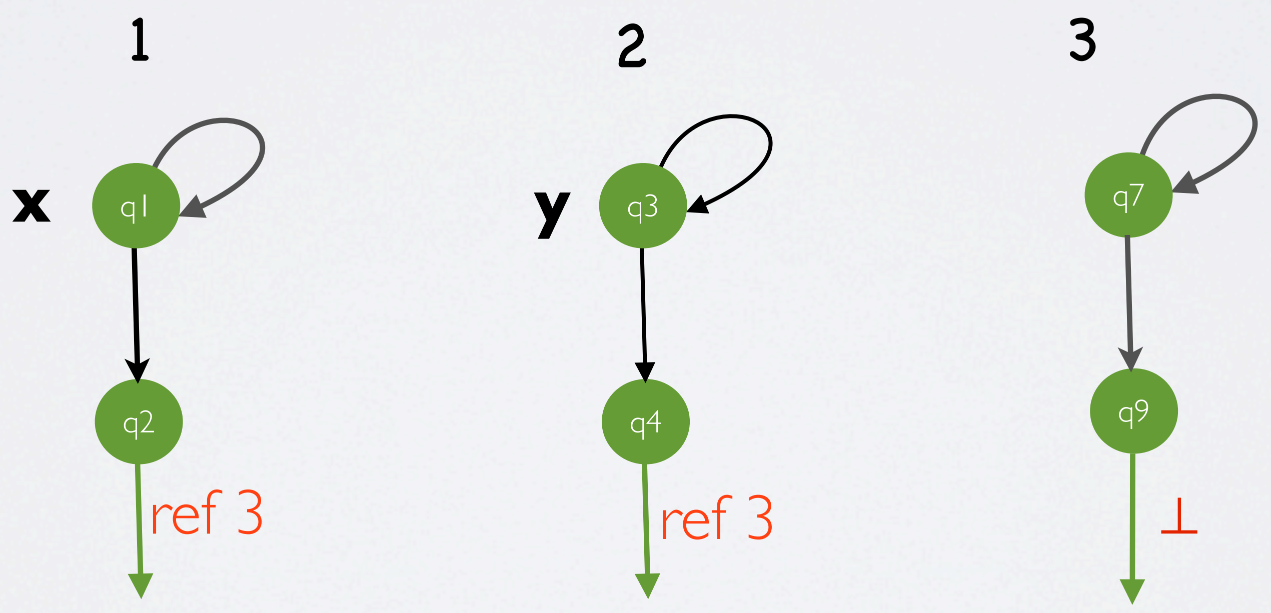




1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

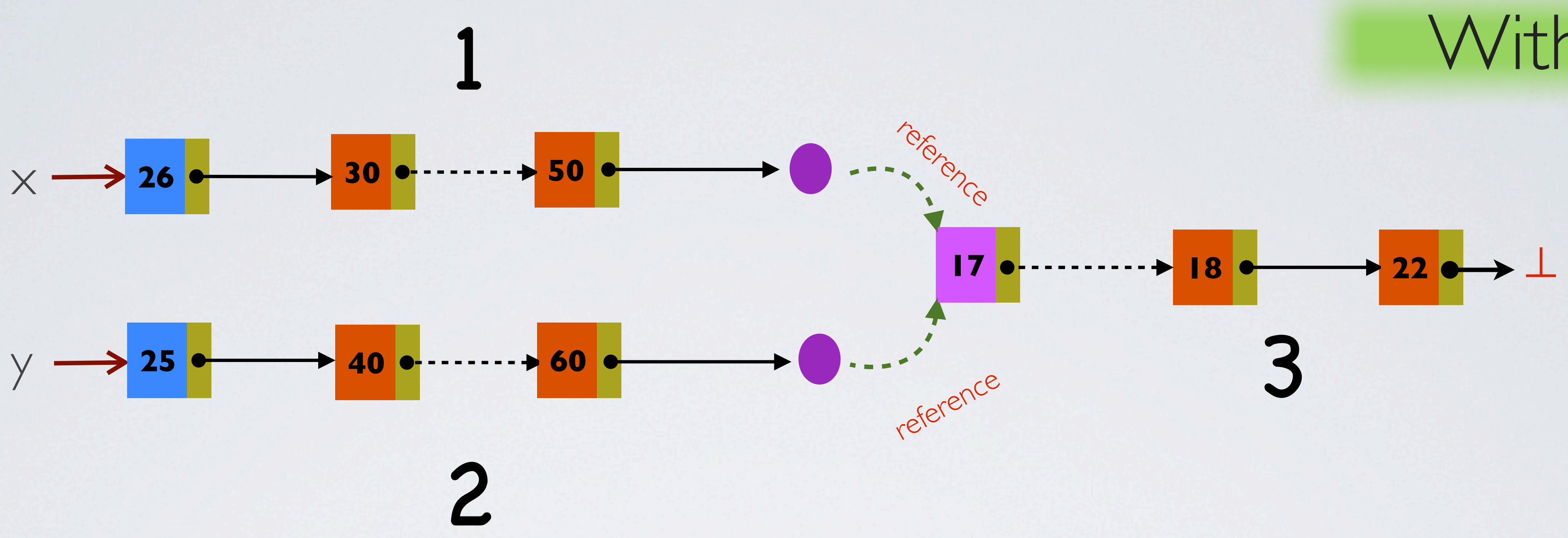
With data

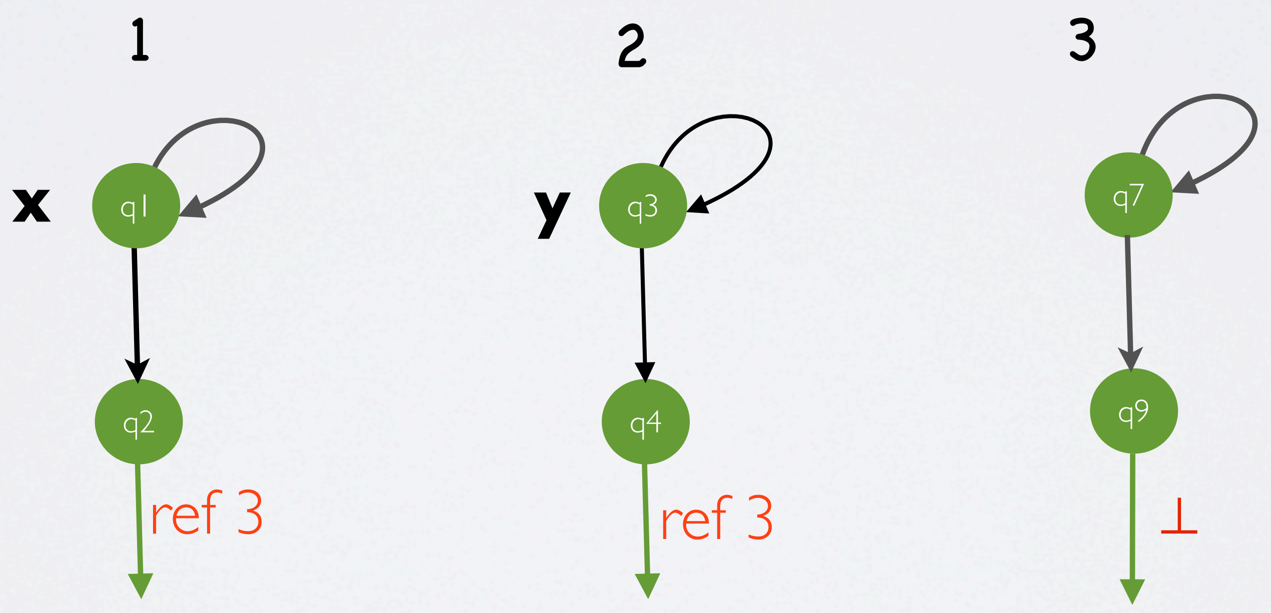




1. Heap representation 2. Decompose heaps into forests **3. Represent forests by forest automata**

With data

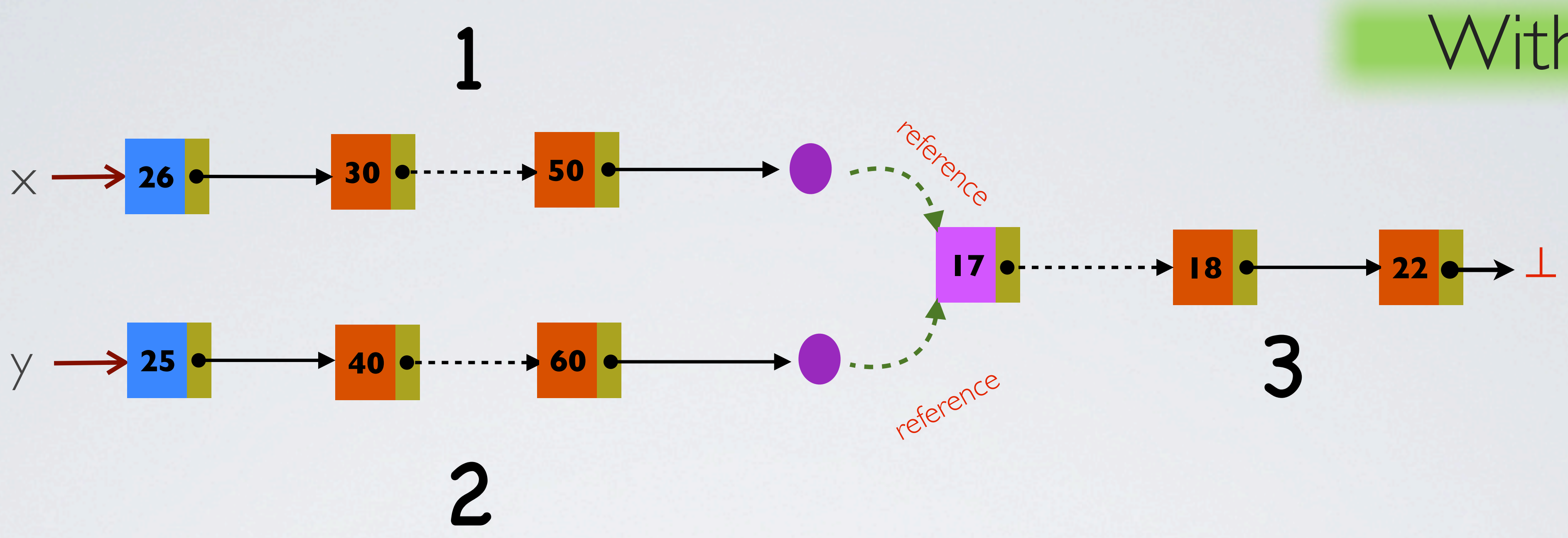


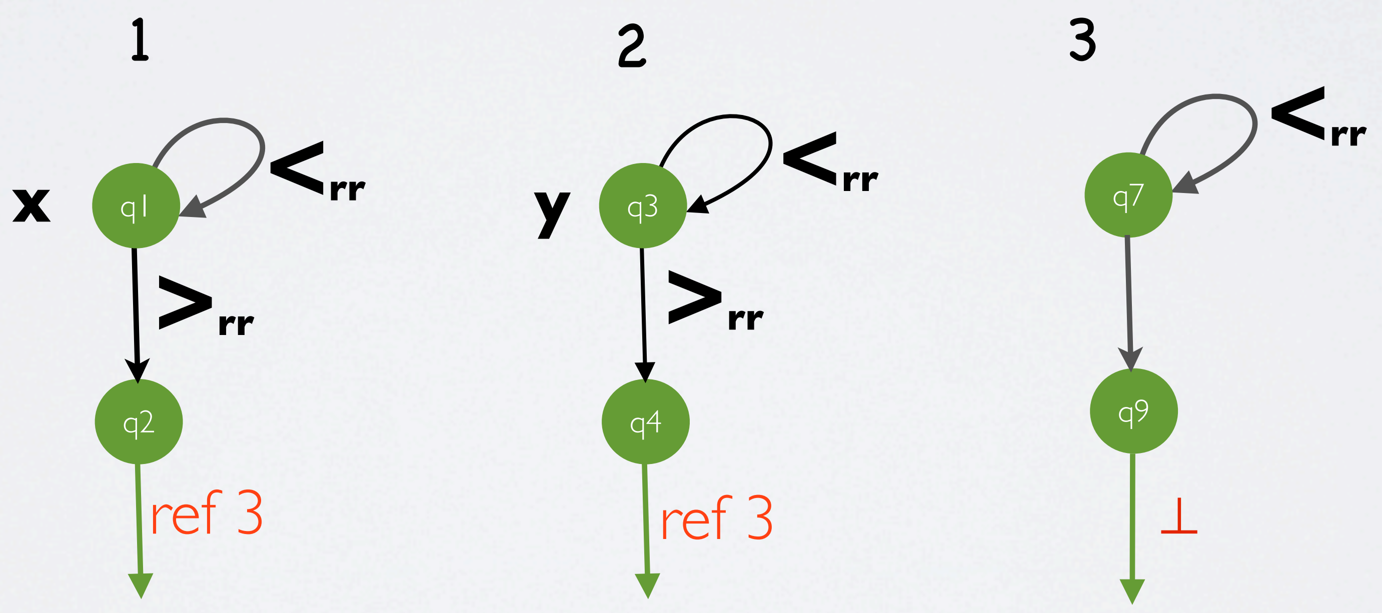


Add data constraints

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

With data

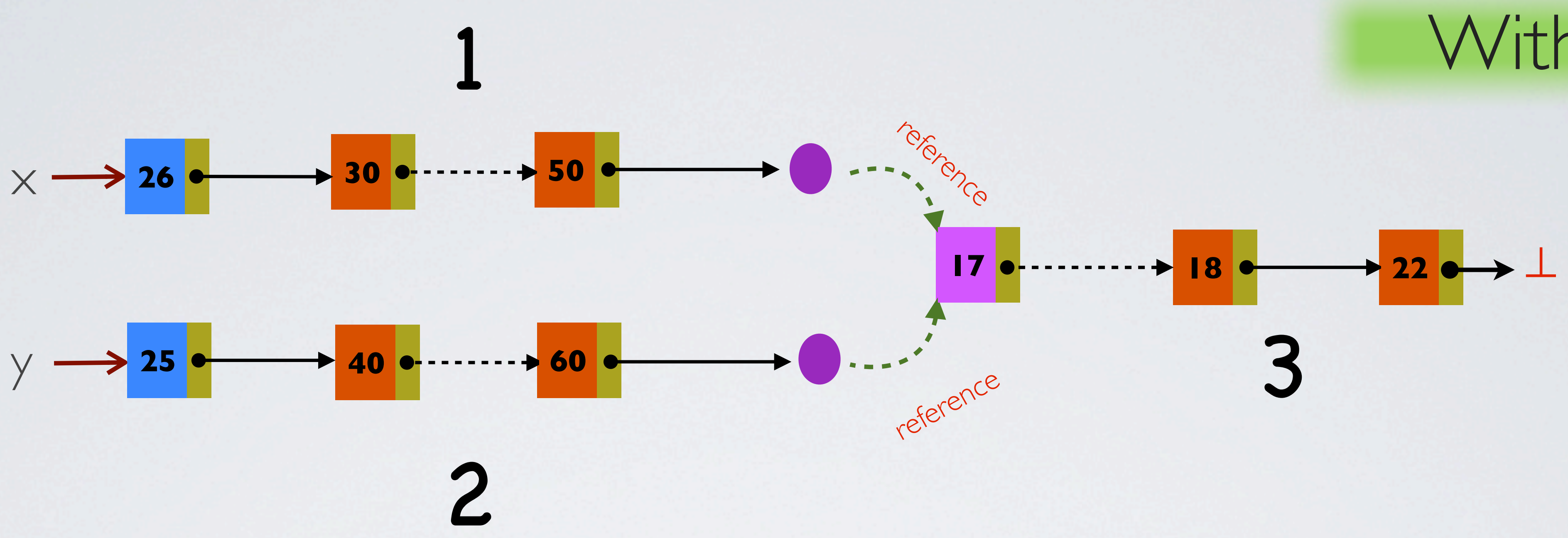


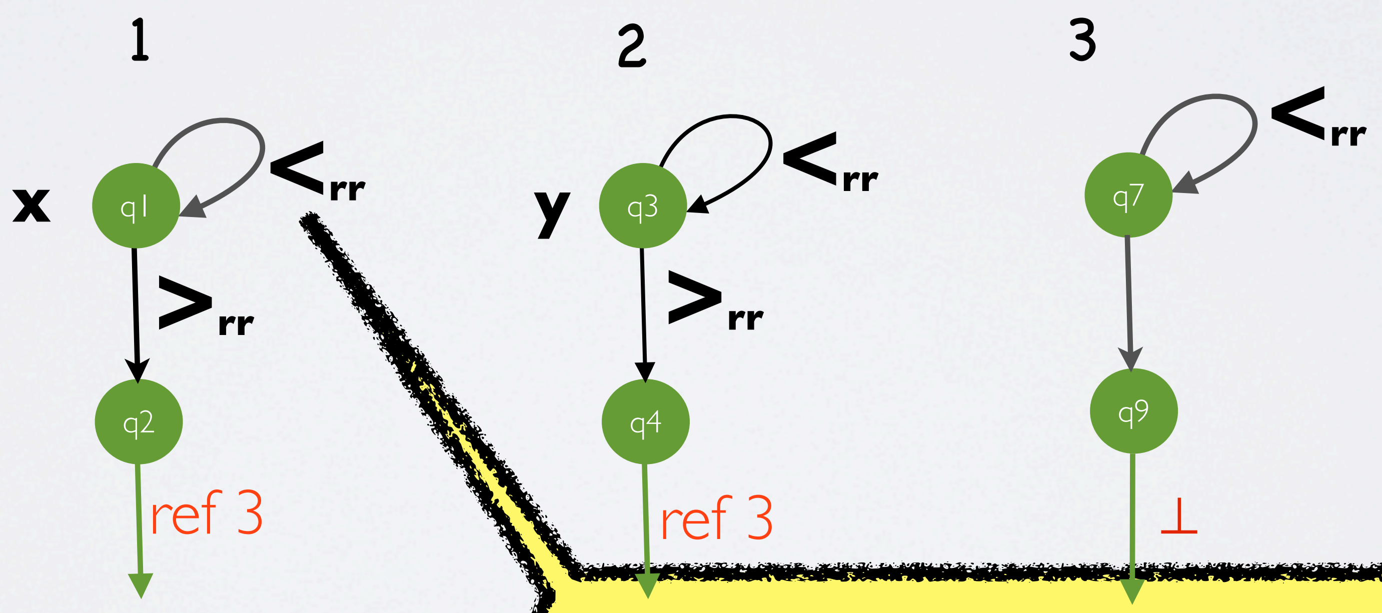


Add data constraints

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

With data

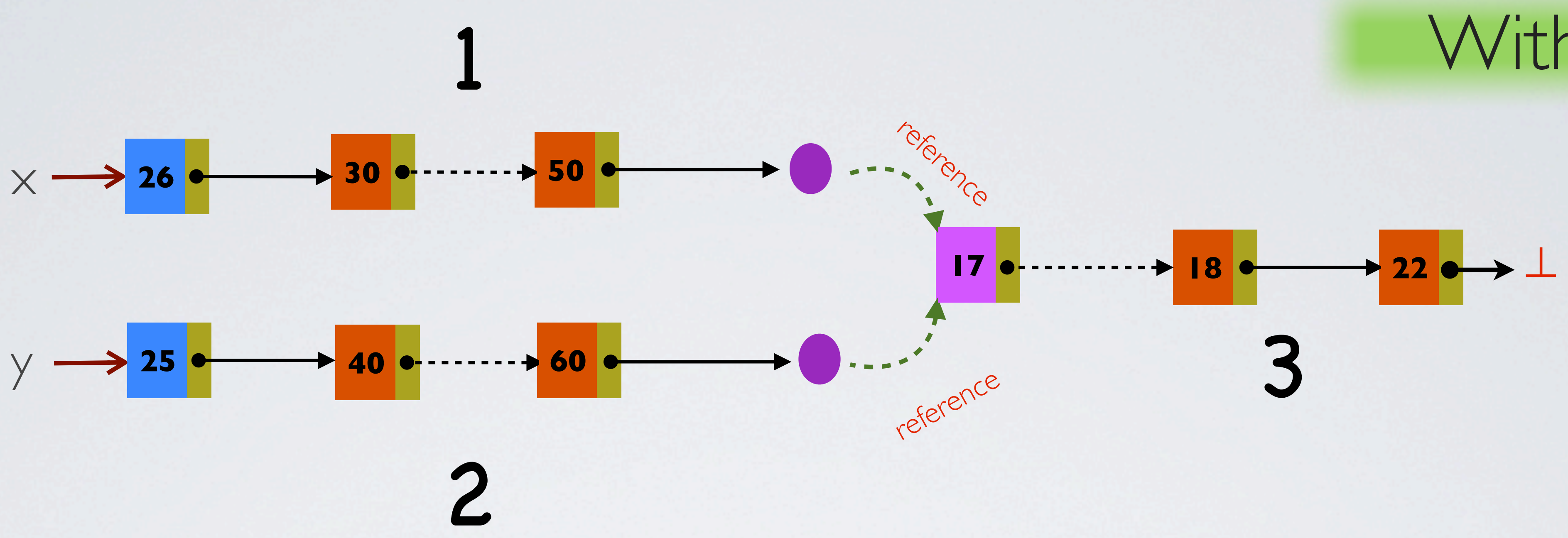


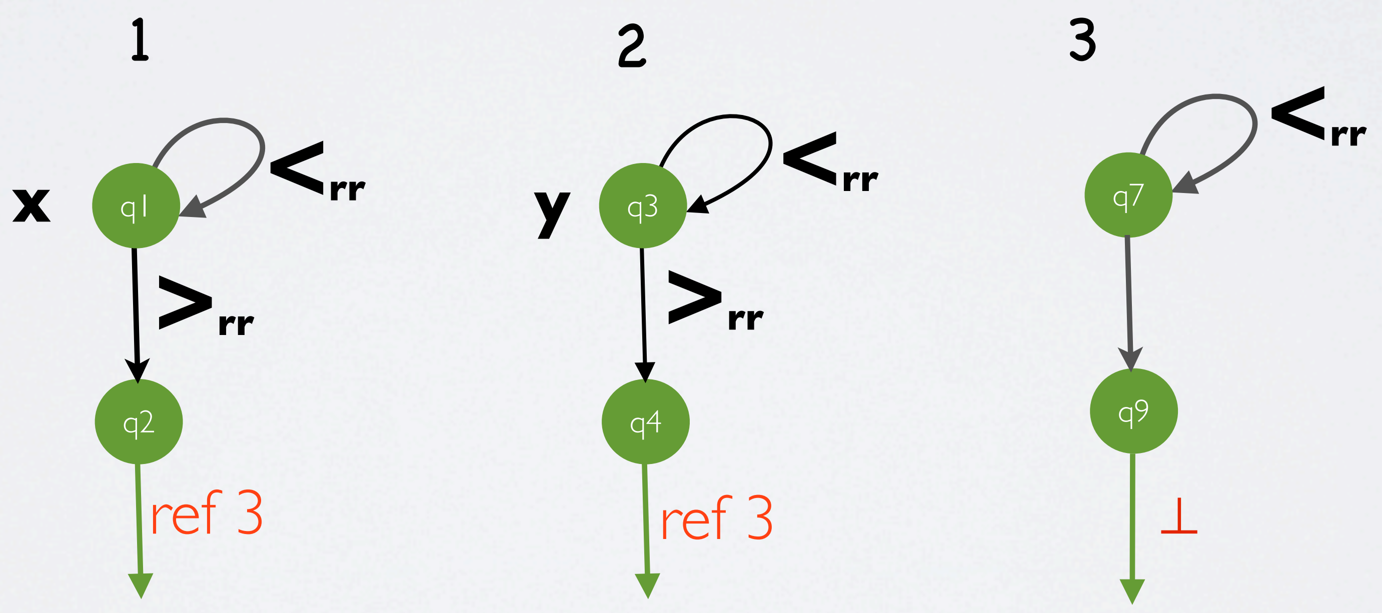


Data value of root node of any tree accepted at q1 is smaller than data value of root node of tree accepted at the next q1 of the cycle

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

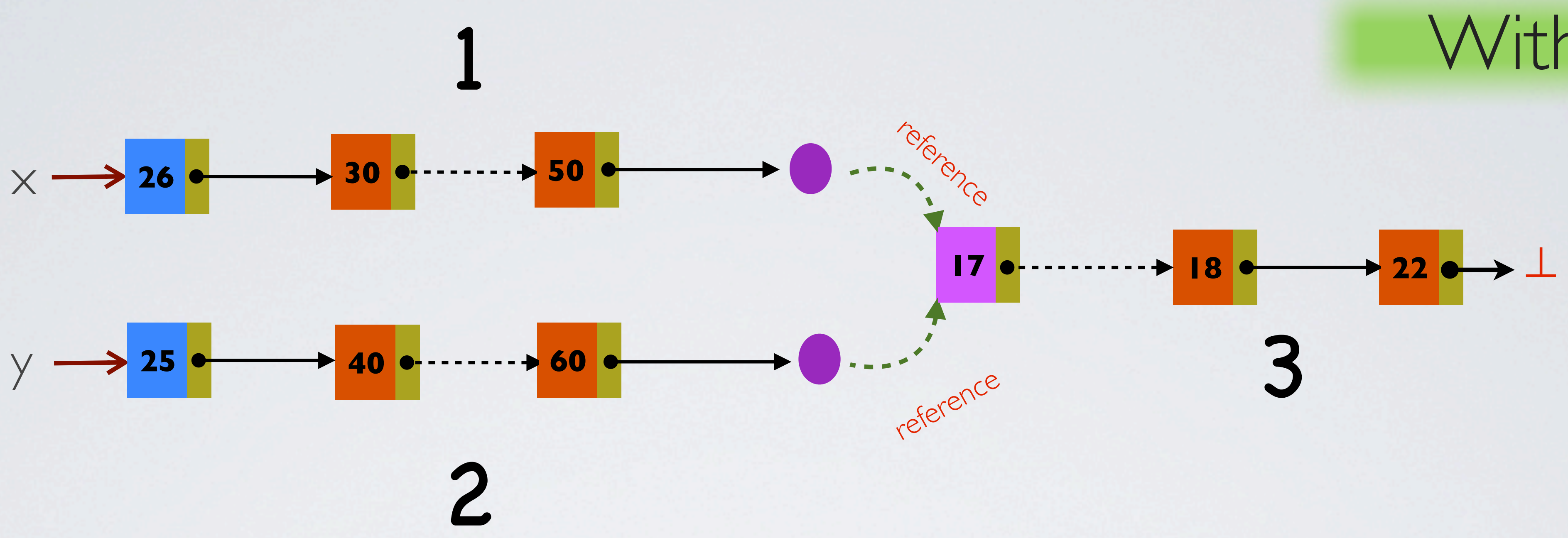
With data

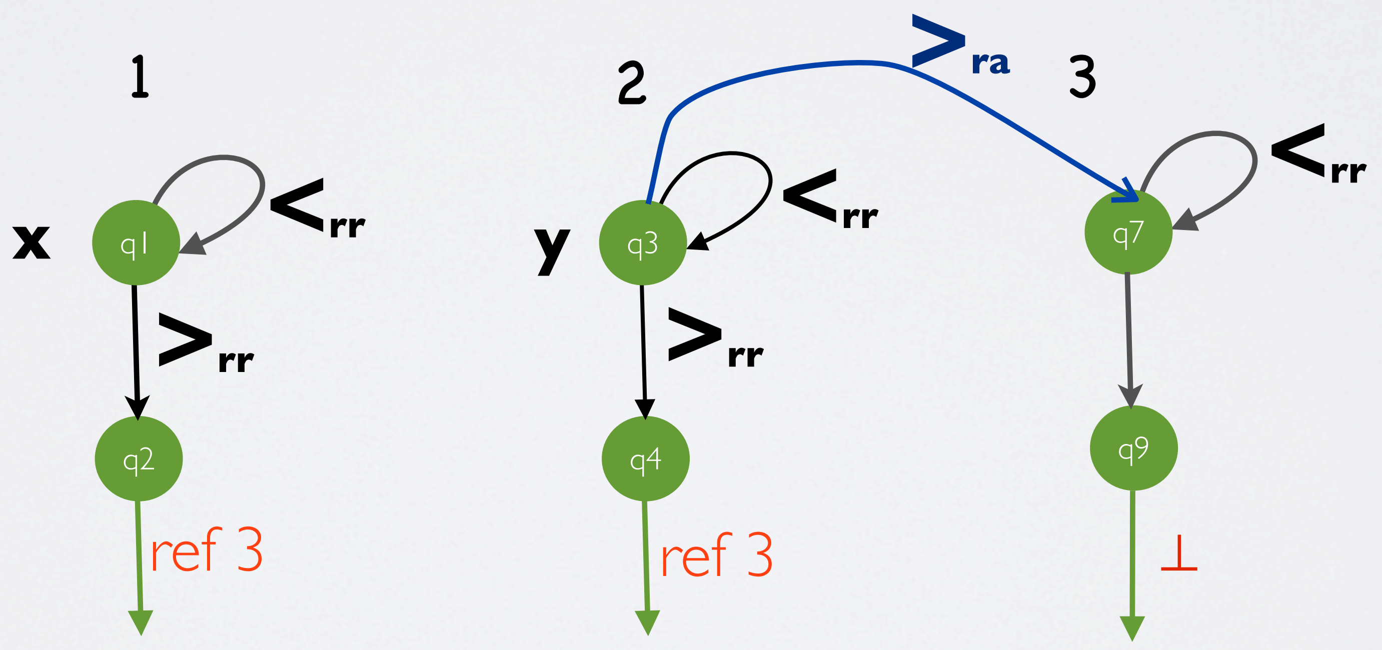




1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

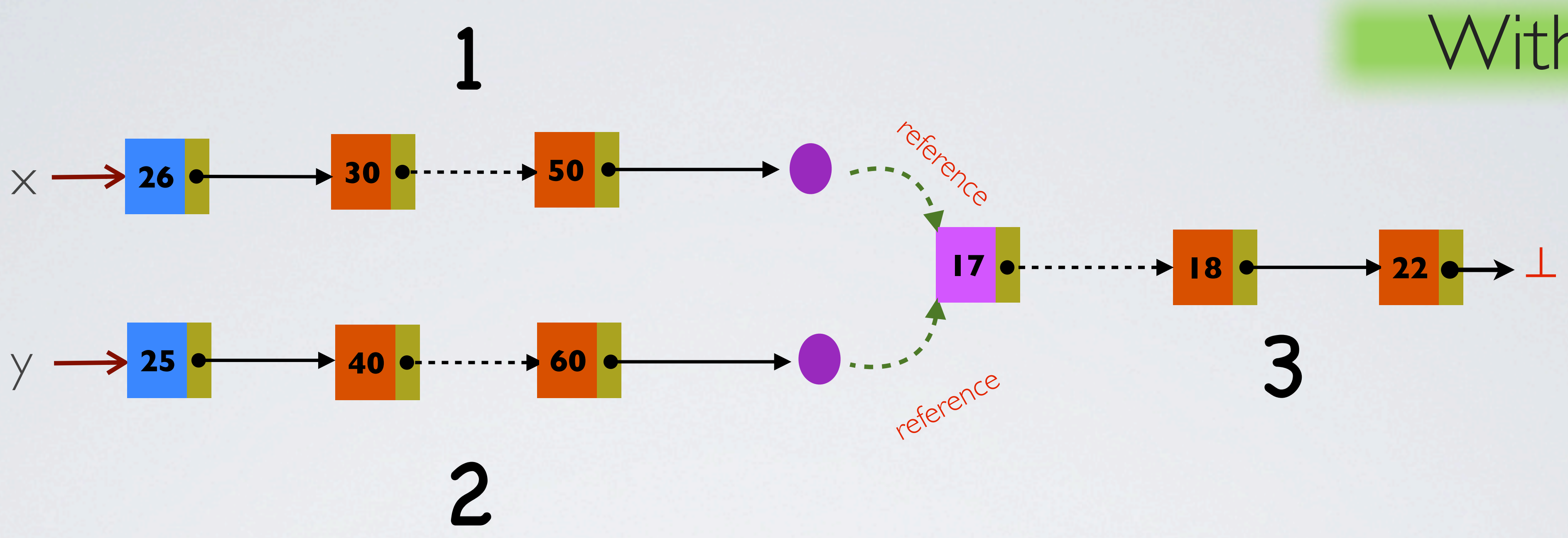
With data

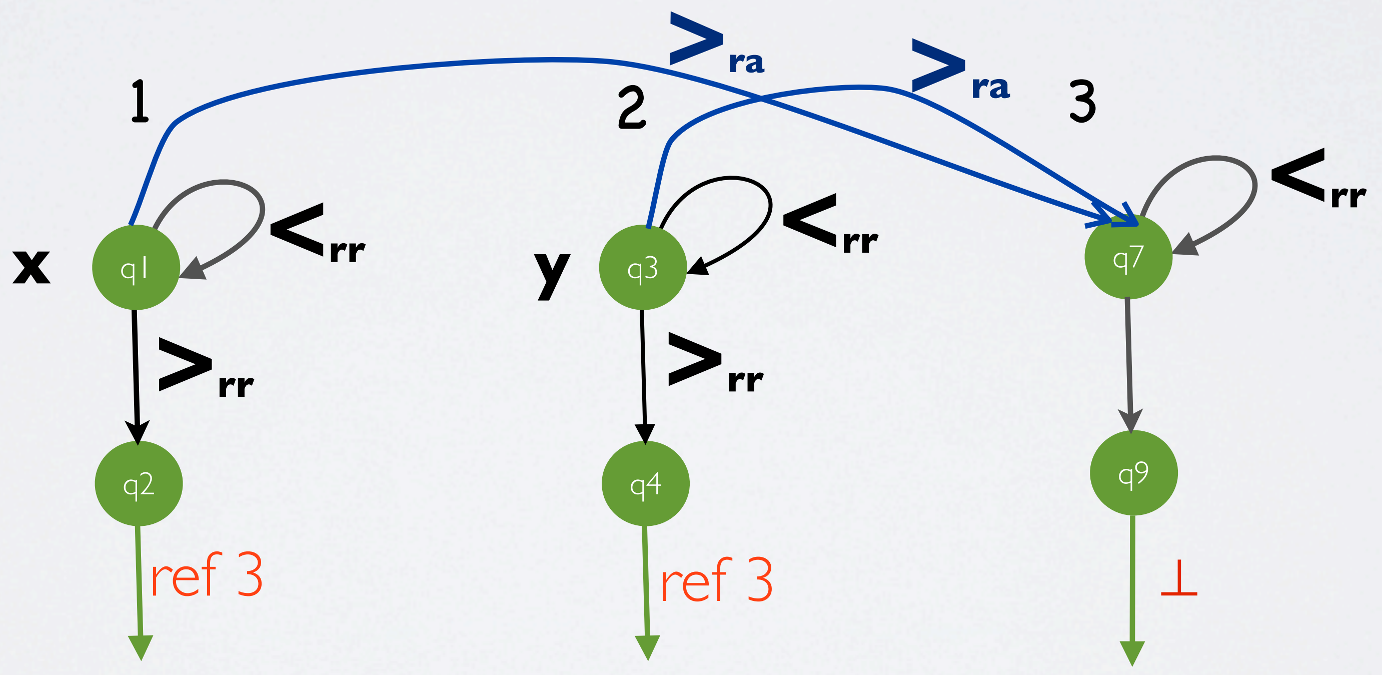




1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

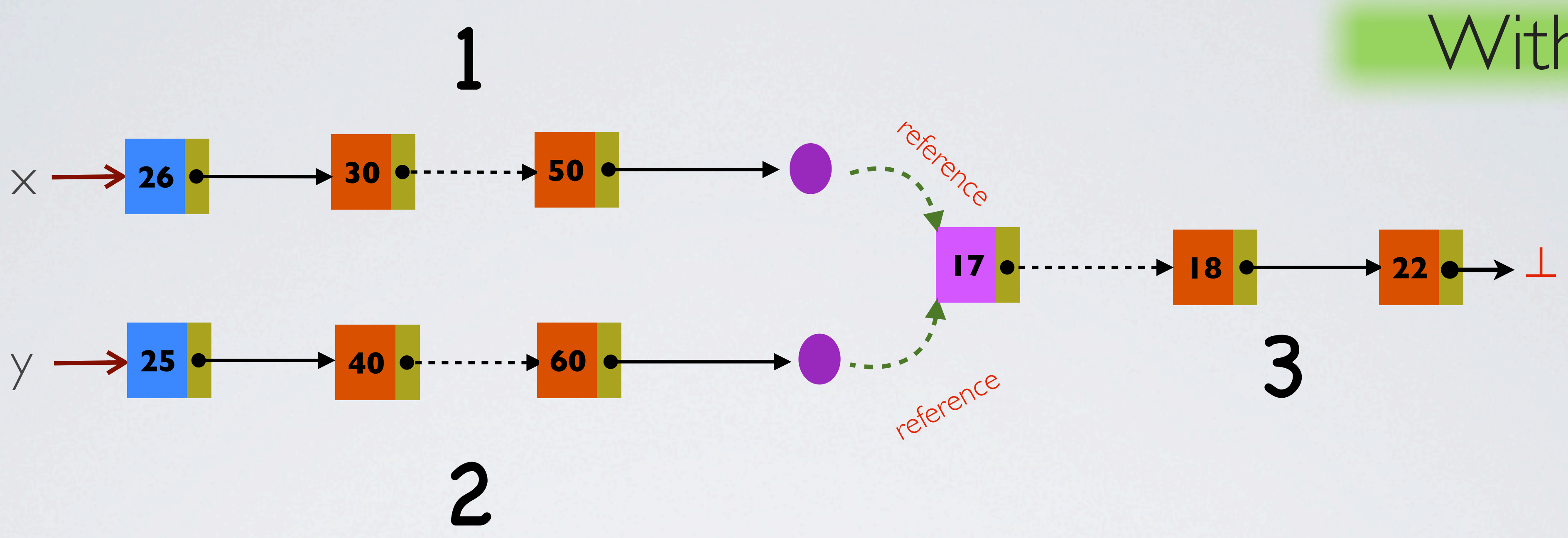
With data

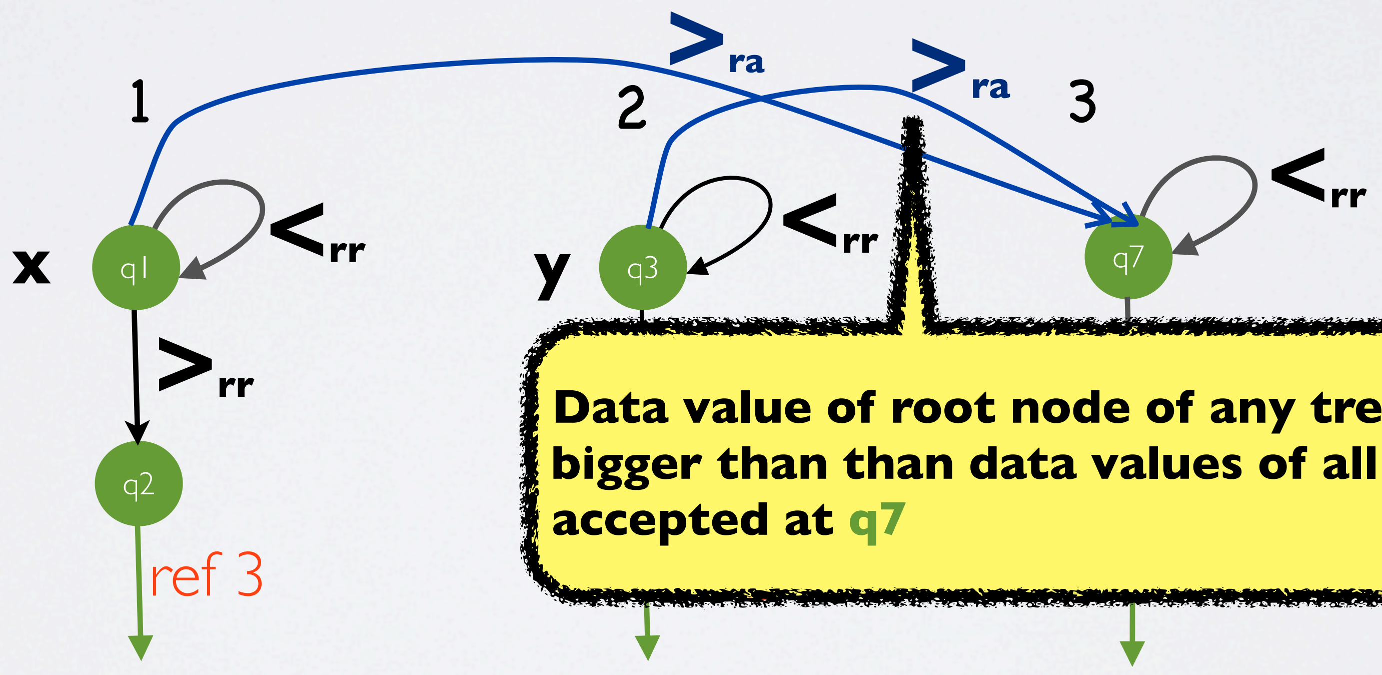




1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

With data

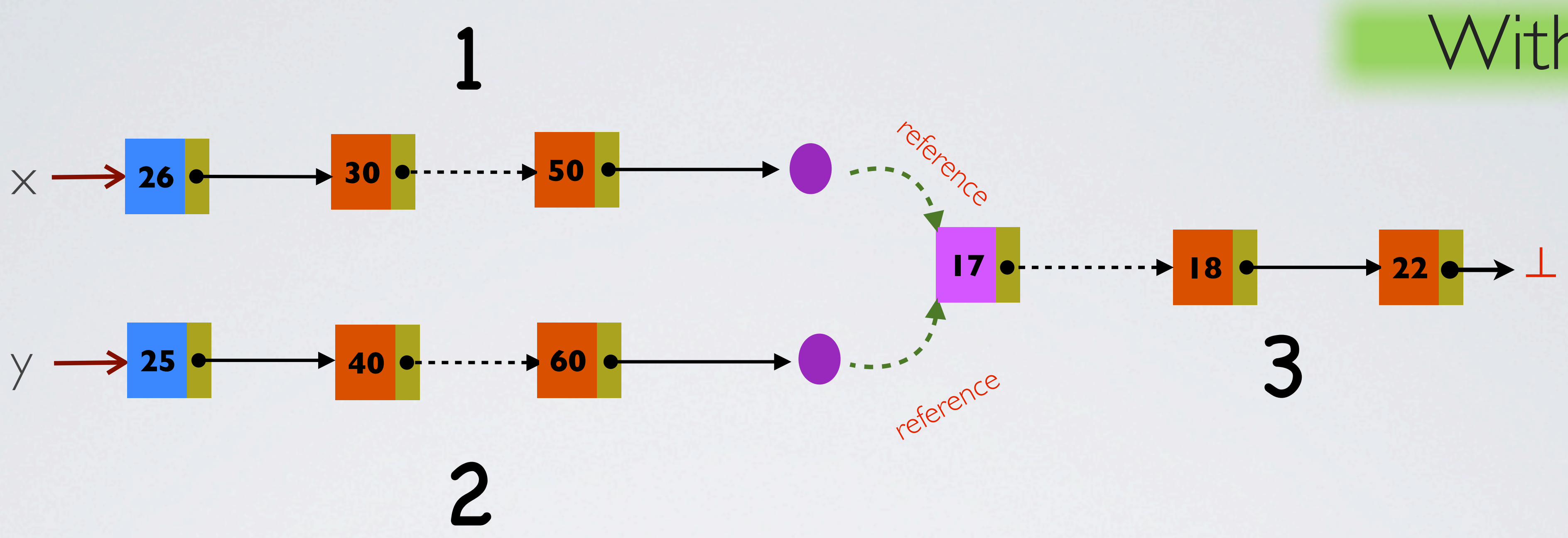


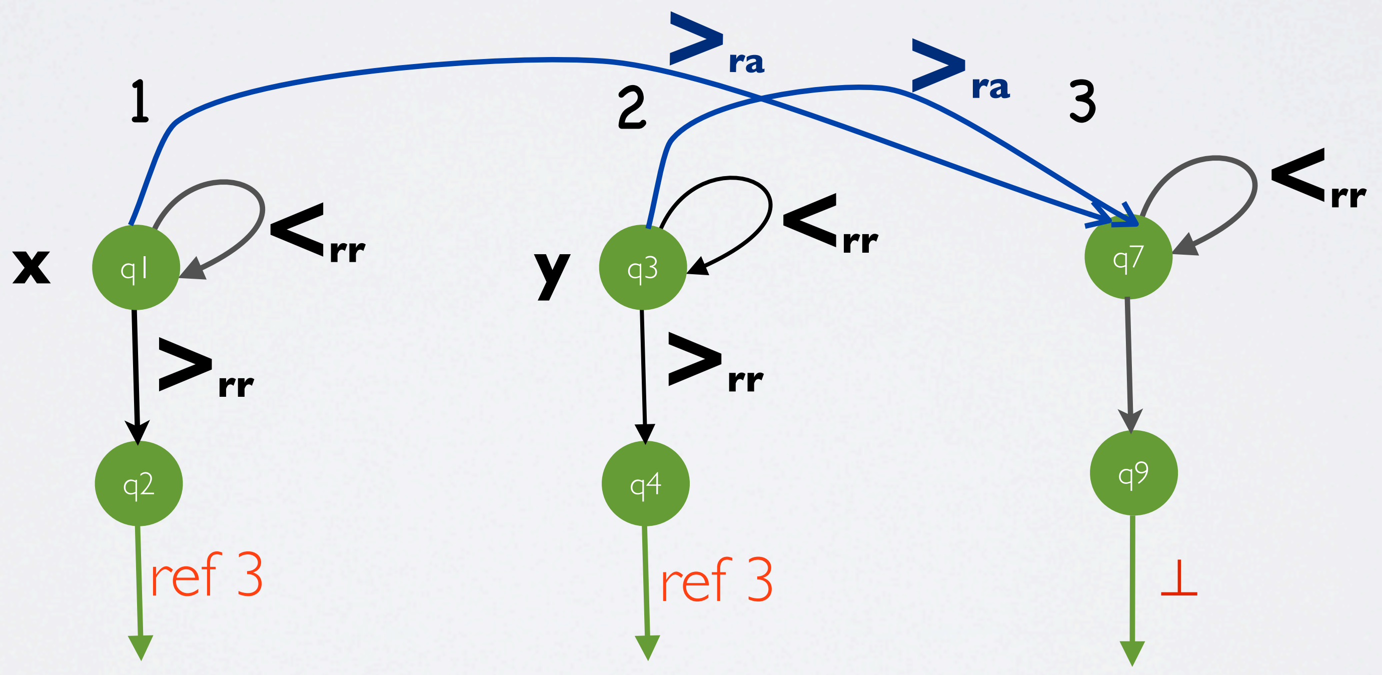


Data value of root node of any tree accepted at q1 is bigger than than data values of all nodes of any tree accepted at q7

1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

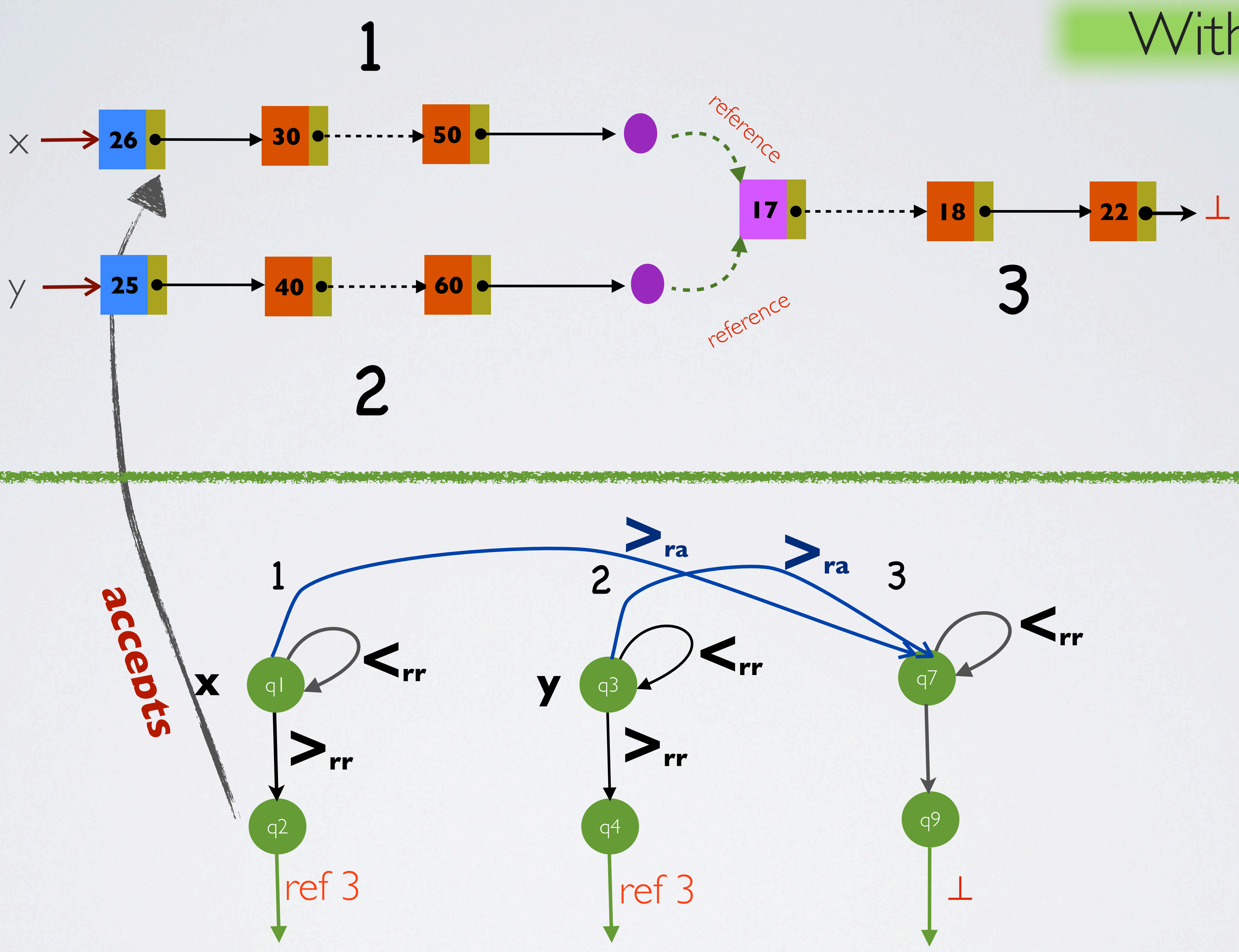
With data





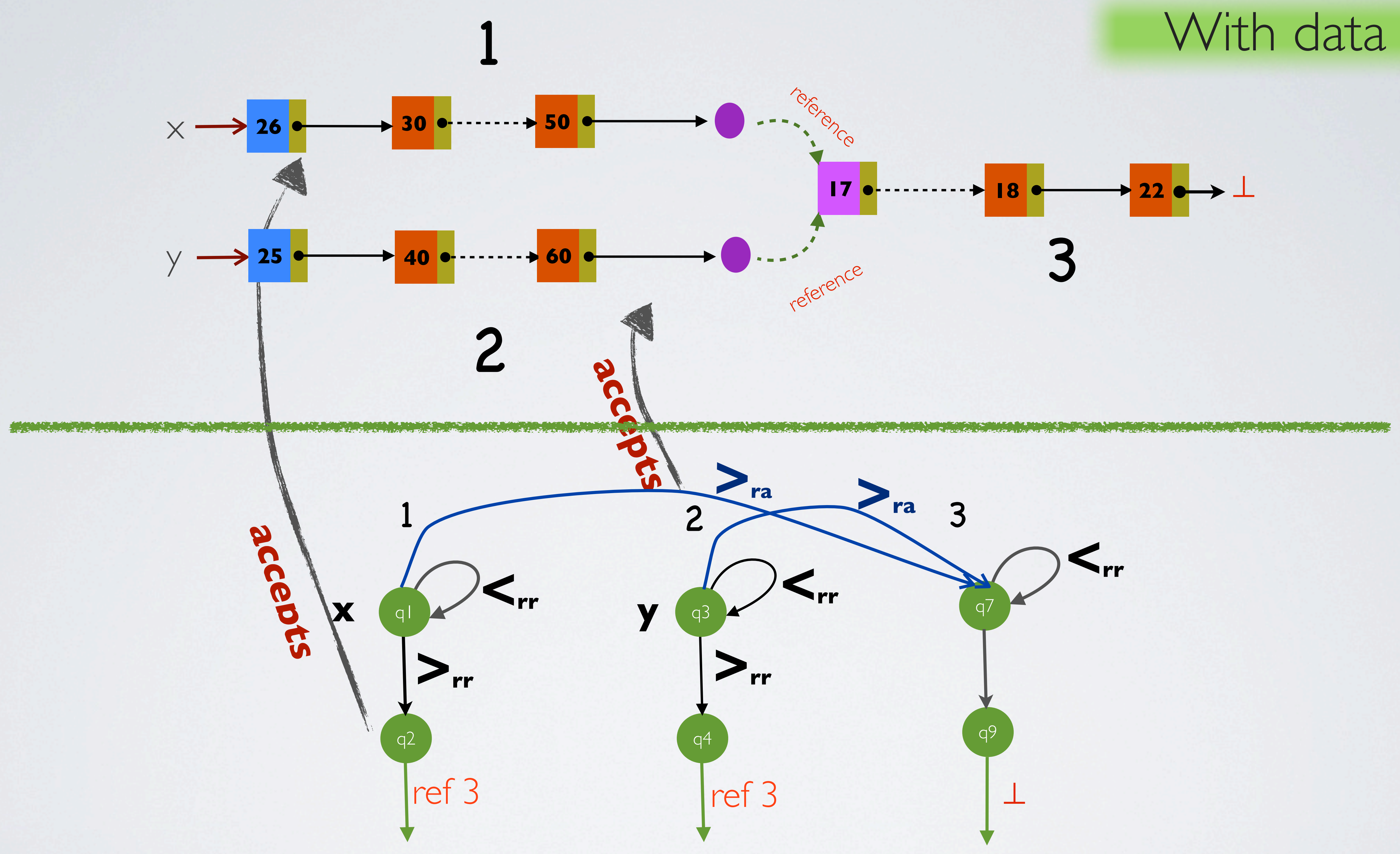
1. Heap representation 2. Decompose heaps into forests 3. Represent forests by forest automata

With data



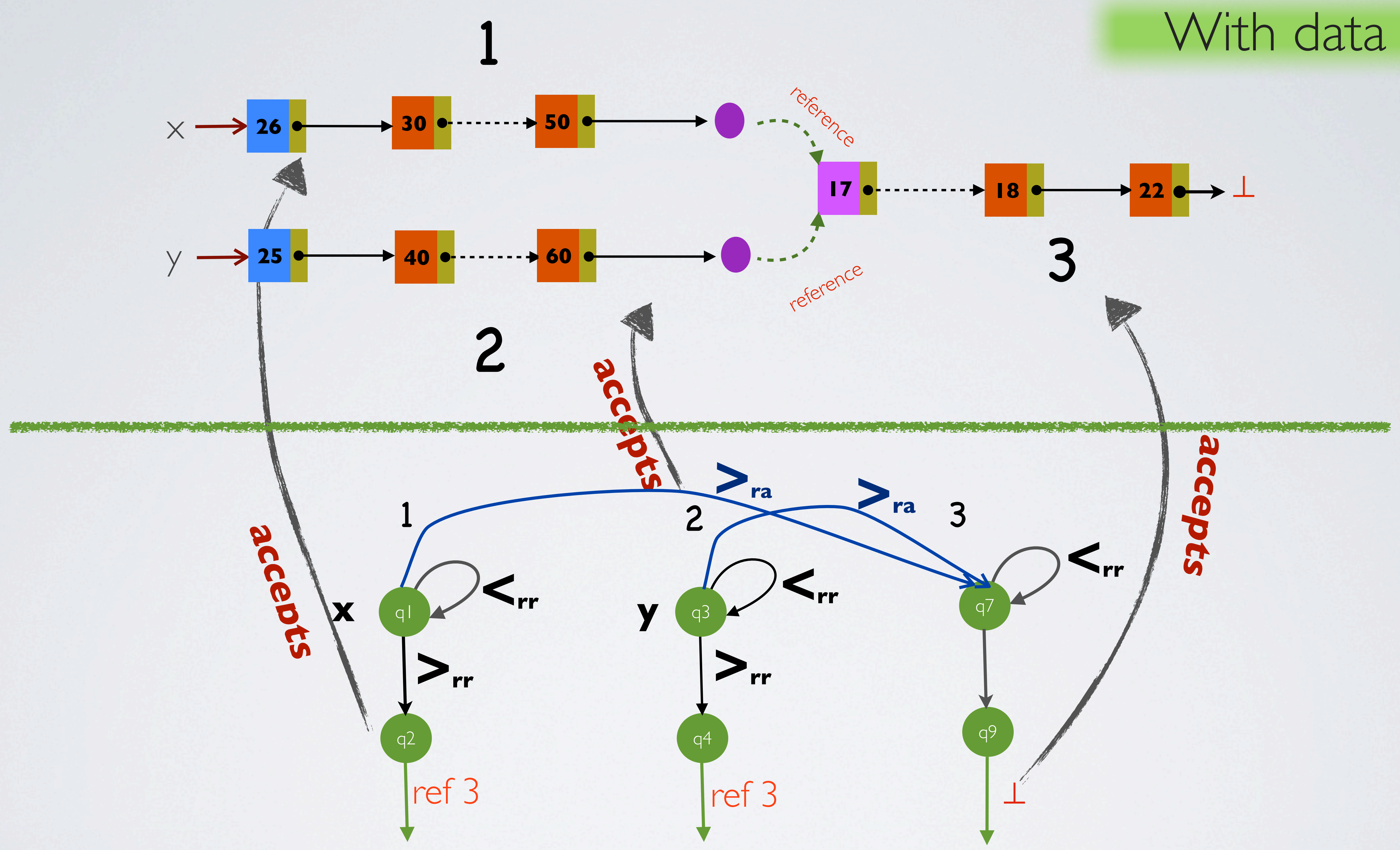
1. Heap representation   2. Decompose heaps into forests   3. Represent forests by forest automata

With data

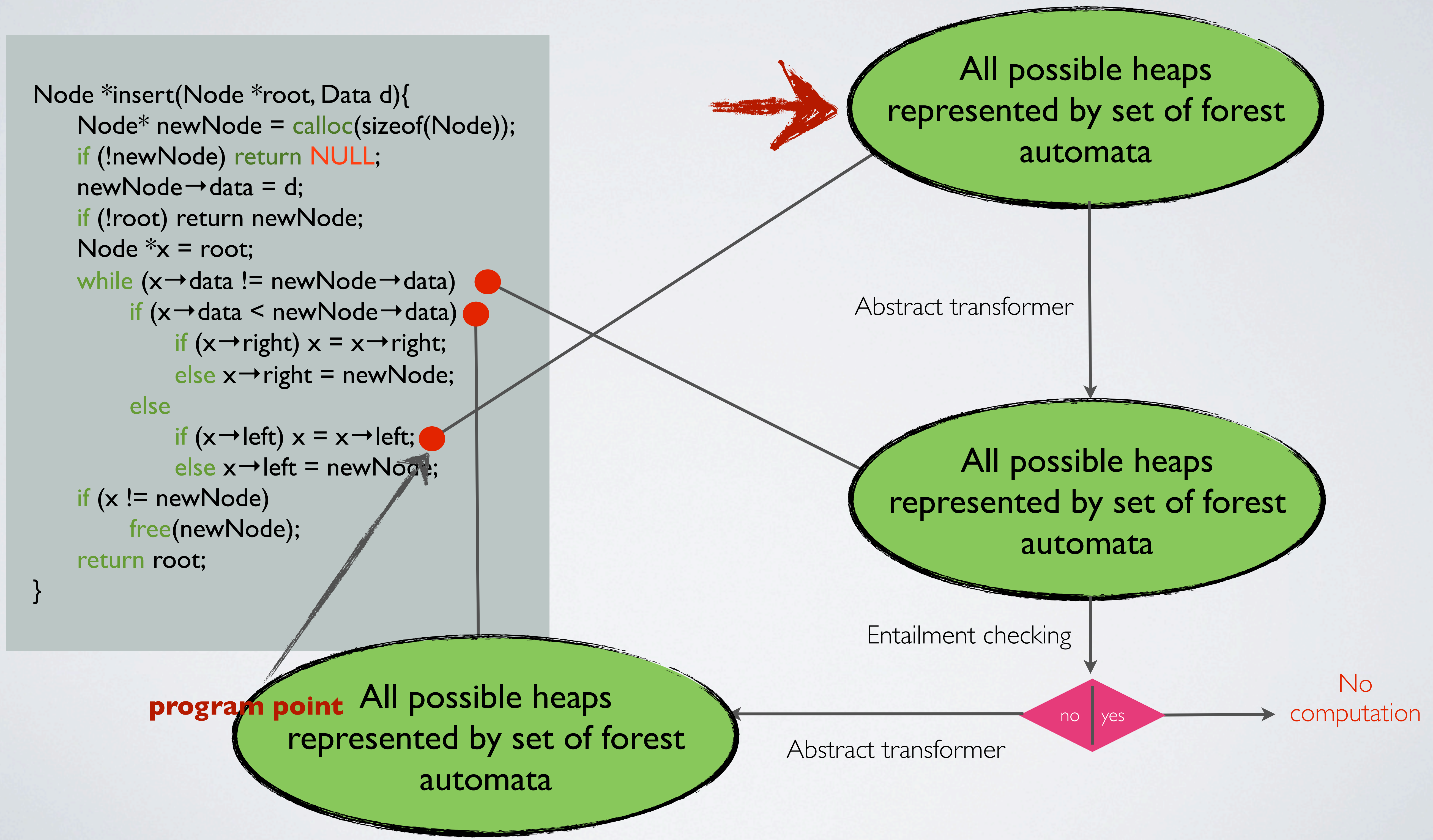


1. Heap representation   2. Decompose heaps into forests   3. Represent forests by forest automata

With data



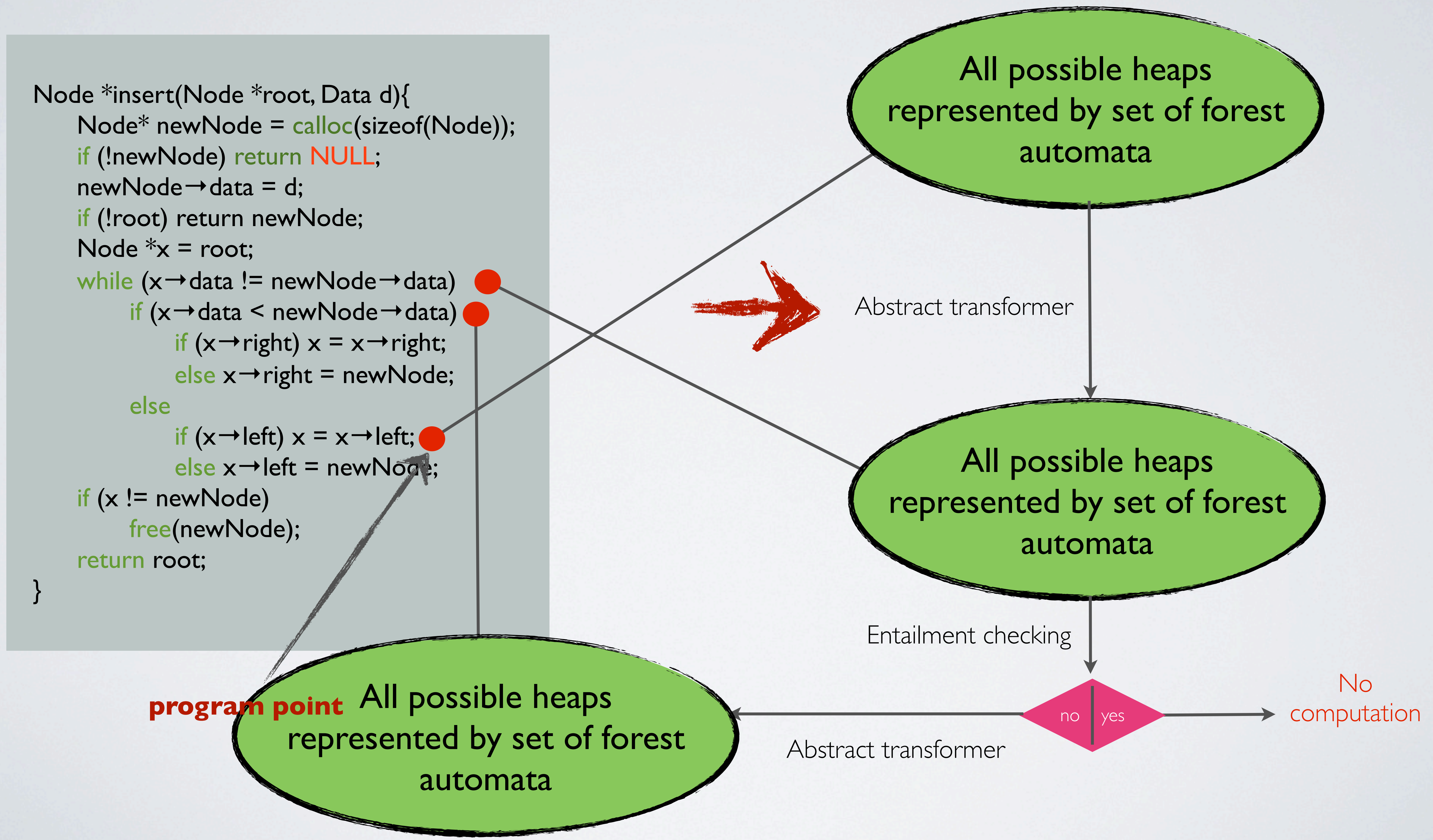
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

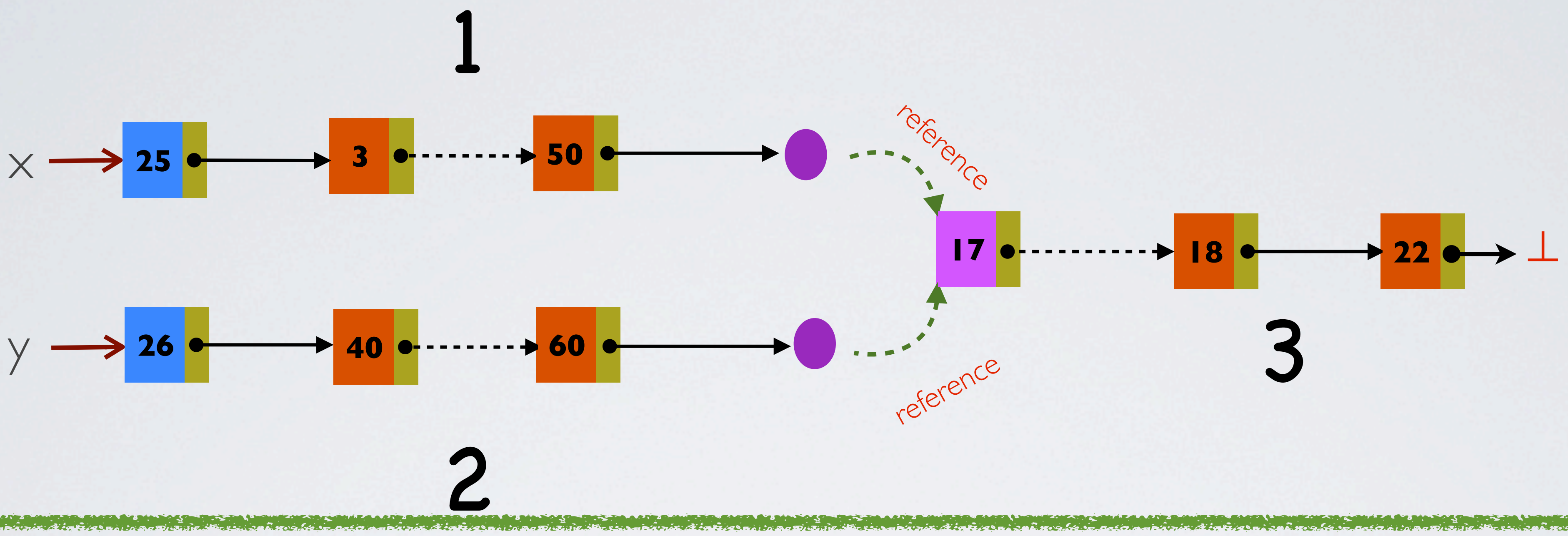
Program analysis

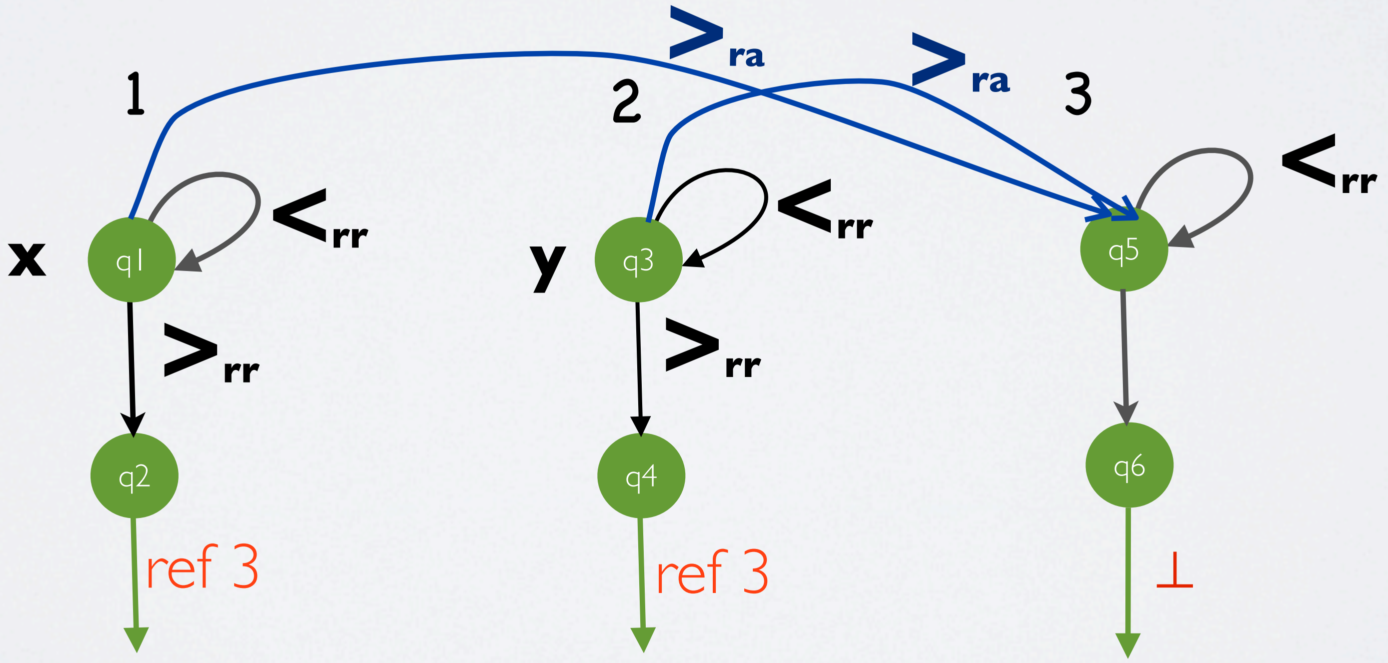


```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

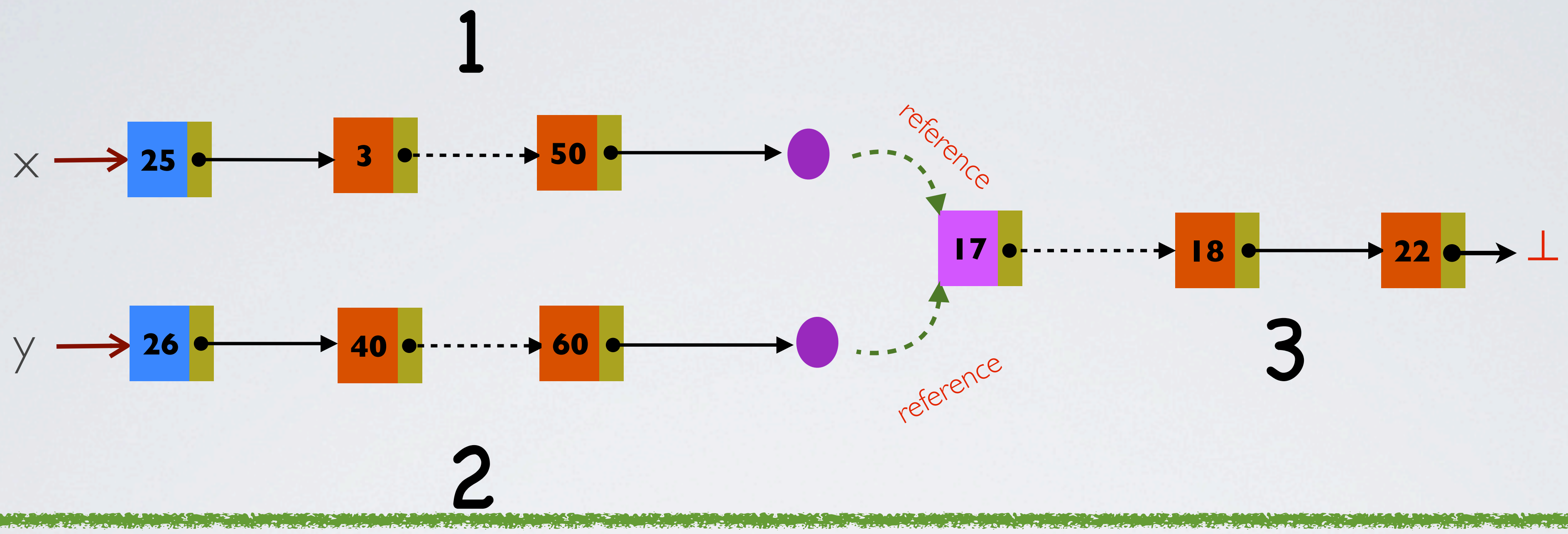
All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

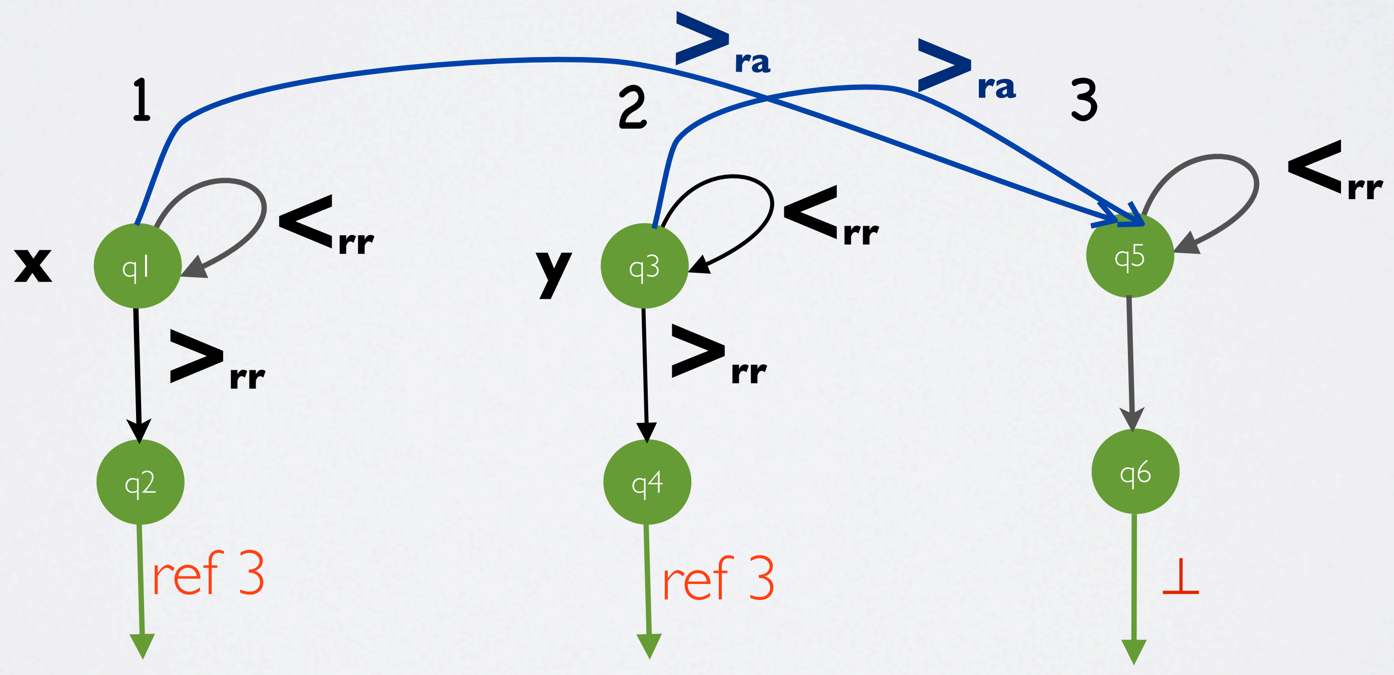
Abstract transformers


1
x
25
3
50
reference
17
18
22
⊥
3
y
26
40
60
reference
2

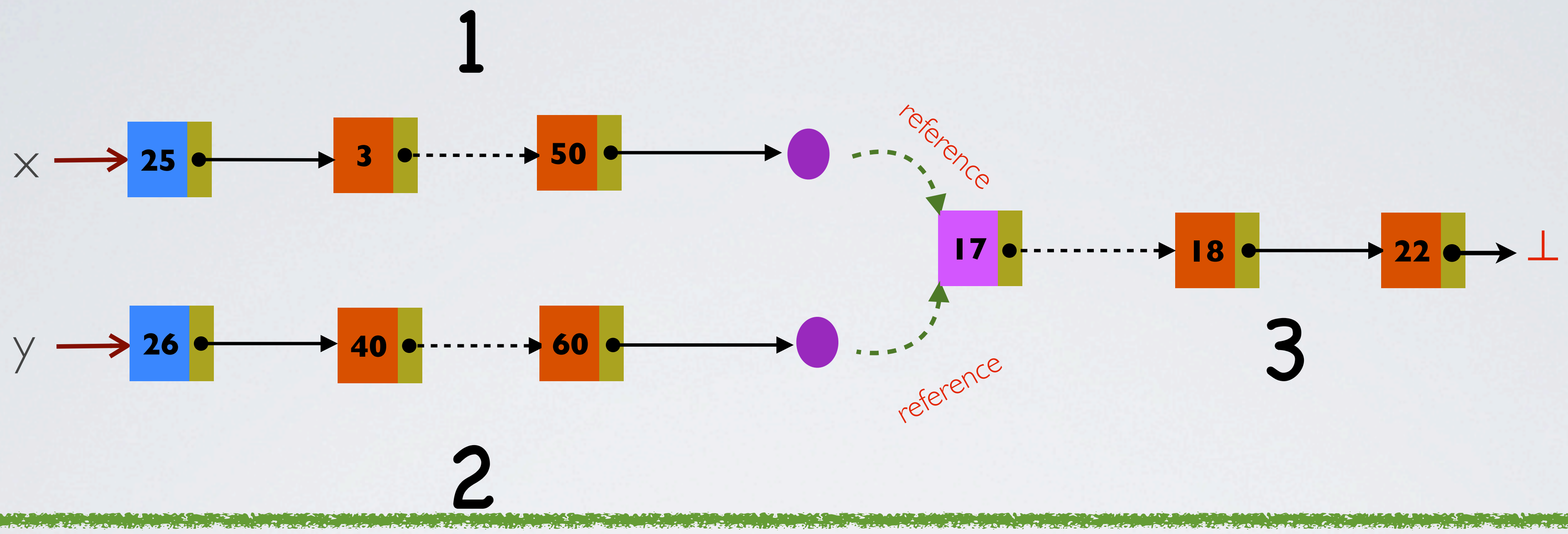

>ra
>ra
1
2
3
<rr
<rr
<rr
x
q1
y
q3
q5
>rr
>rr
q2
q4
q6
ref 3
ref 3
⊥

Effect of **z = x.next** on a concrete forest

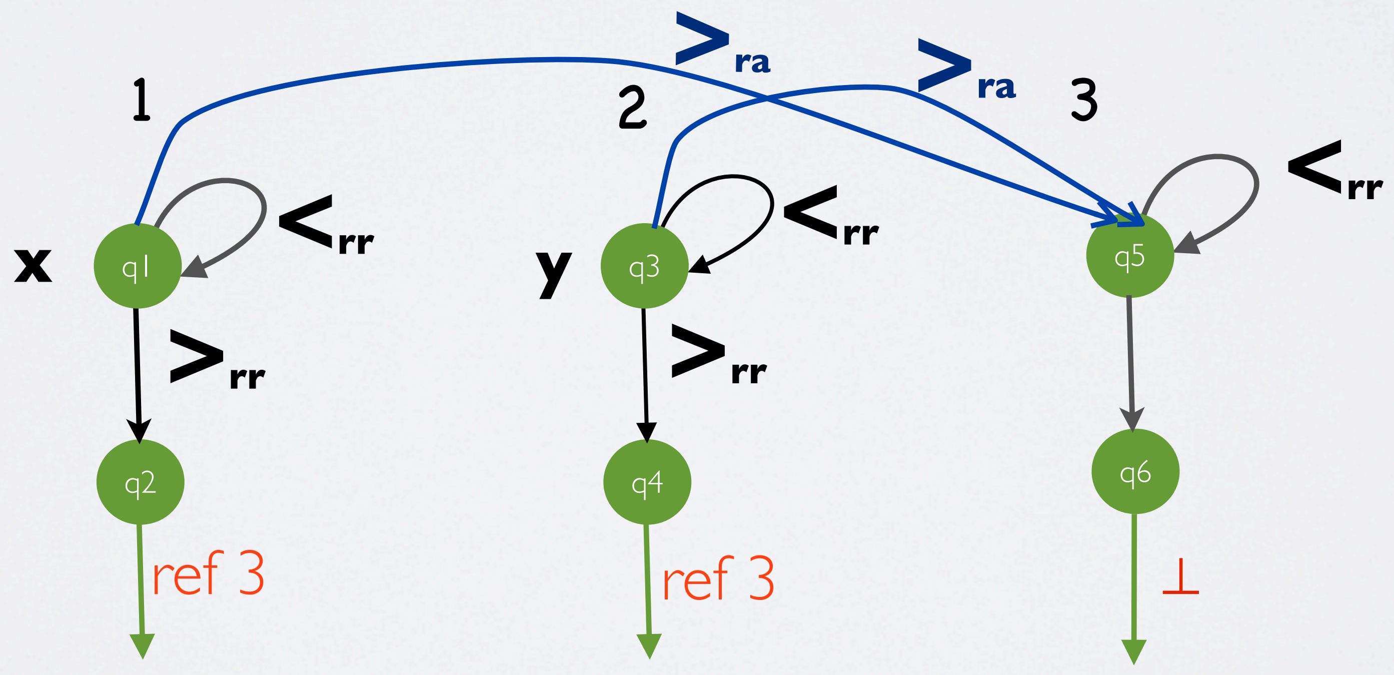




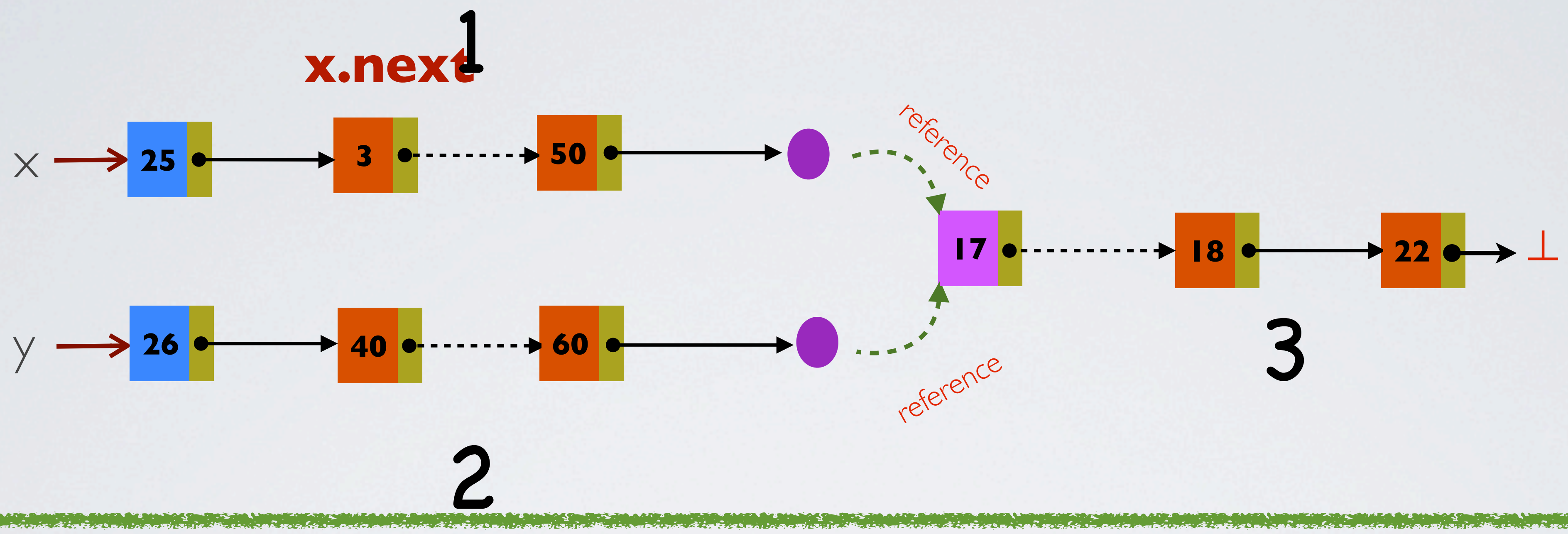
Effect of **z = x.next** on a concrete forest



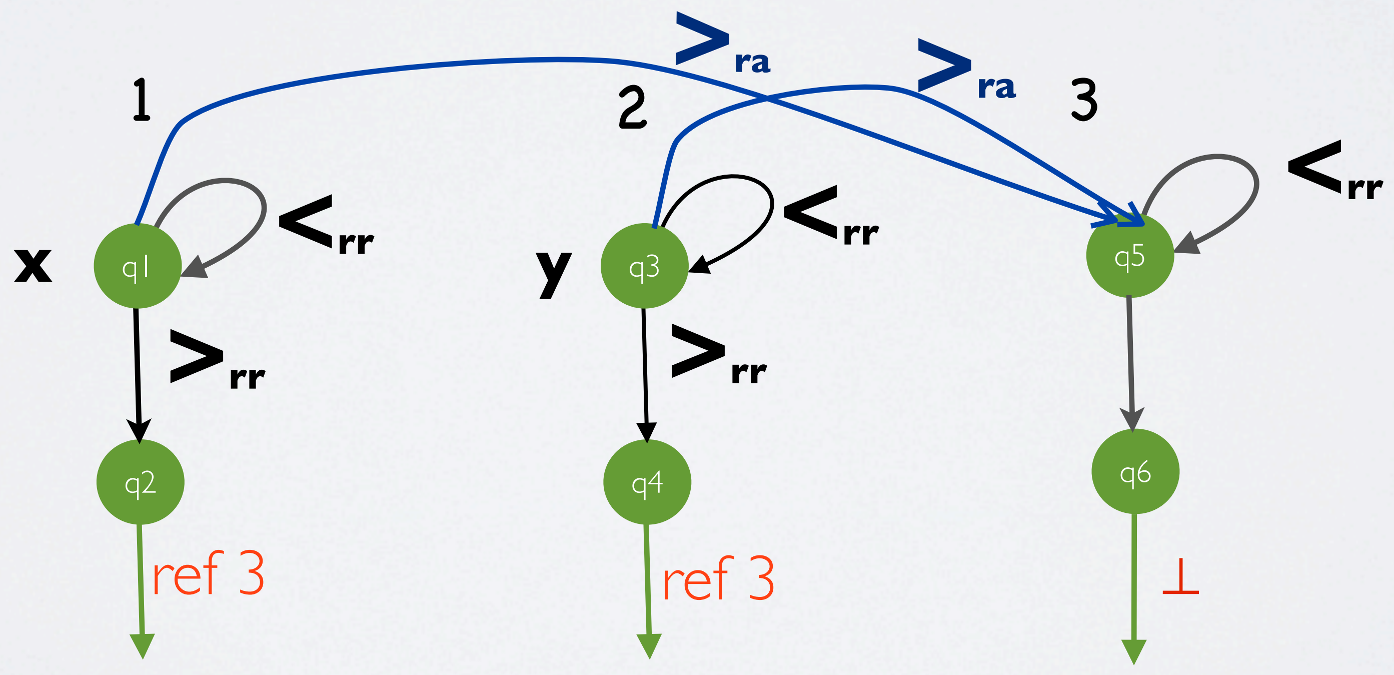
✓ Access to **x.next**
✓ Split the tree 1 at this node



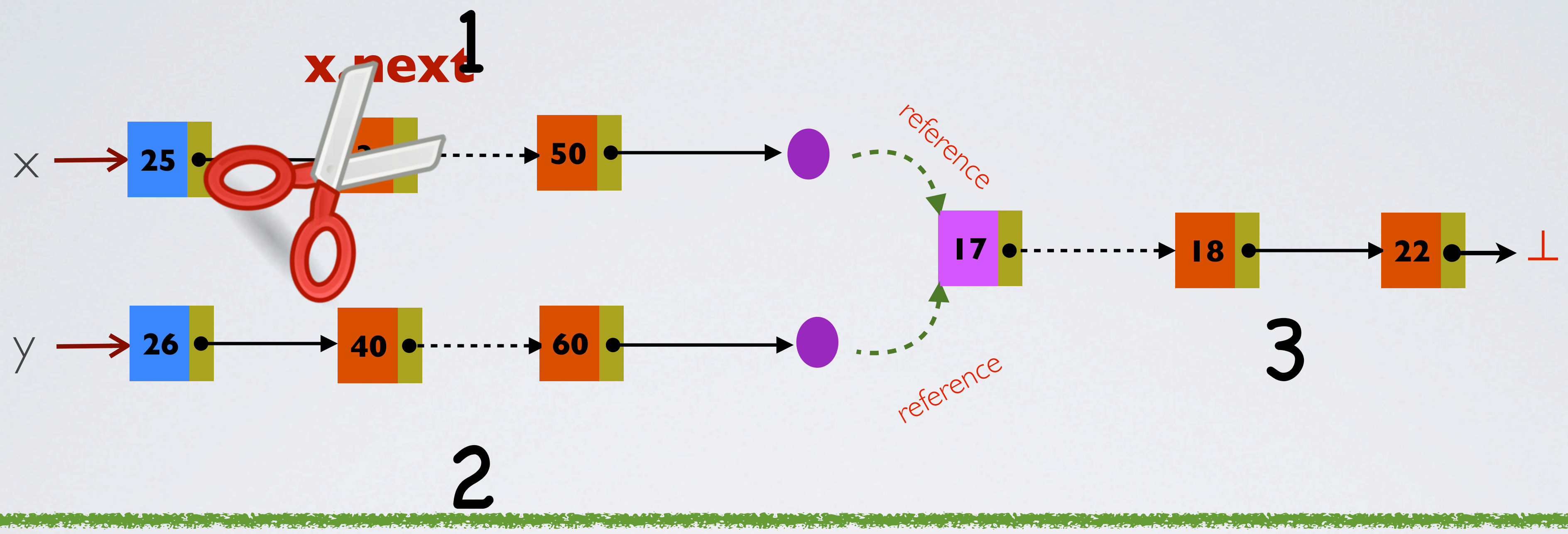
Effect of **z = x.next** on a concrete forest



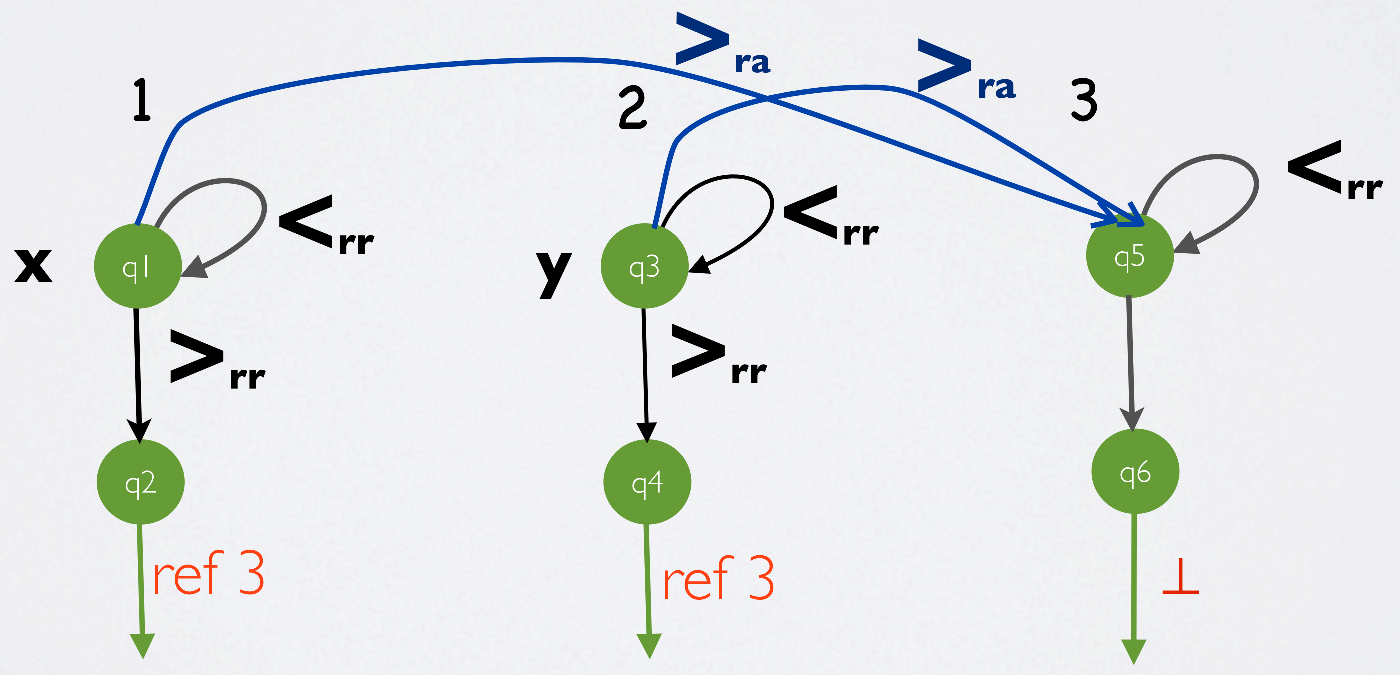
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node



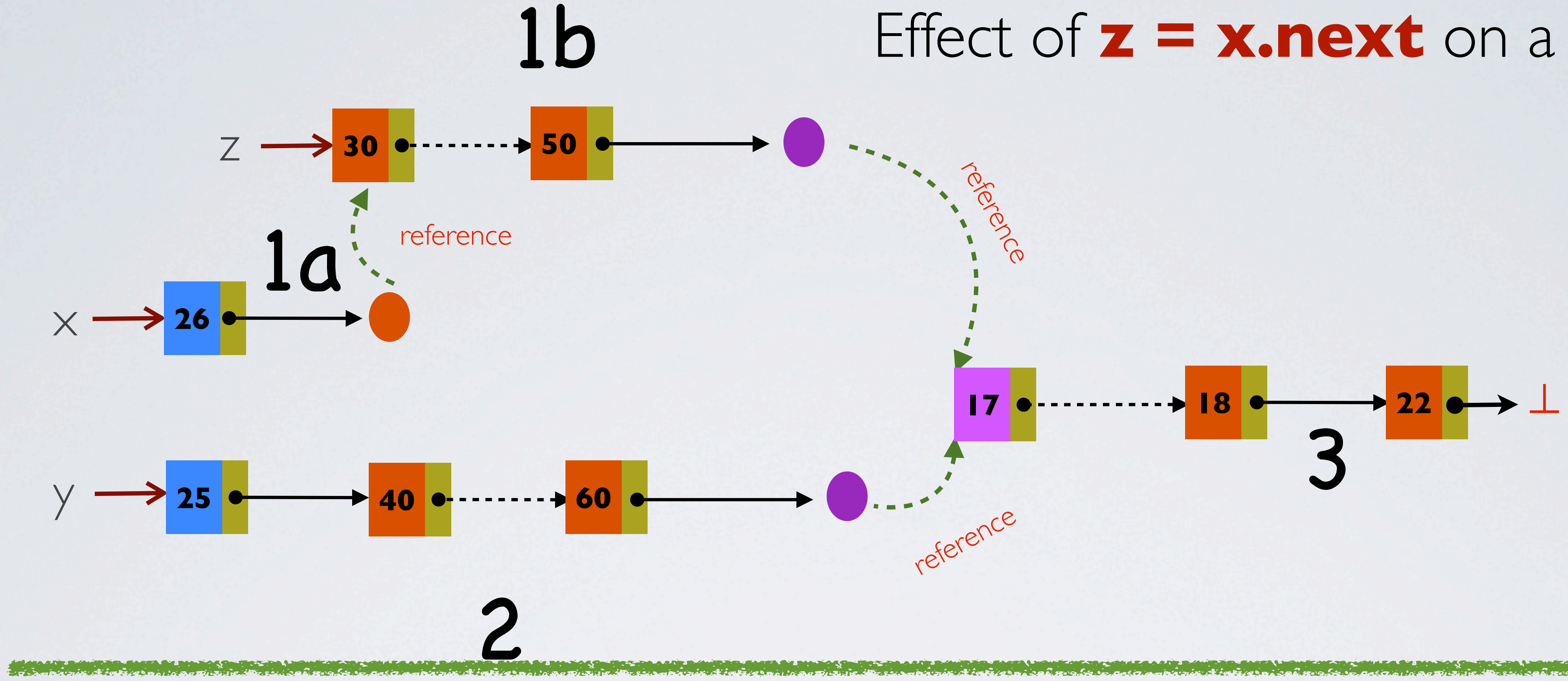
Effect of **z = x.next** on a concrete forest



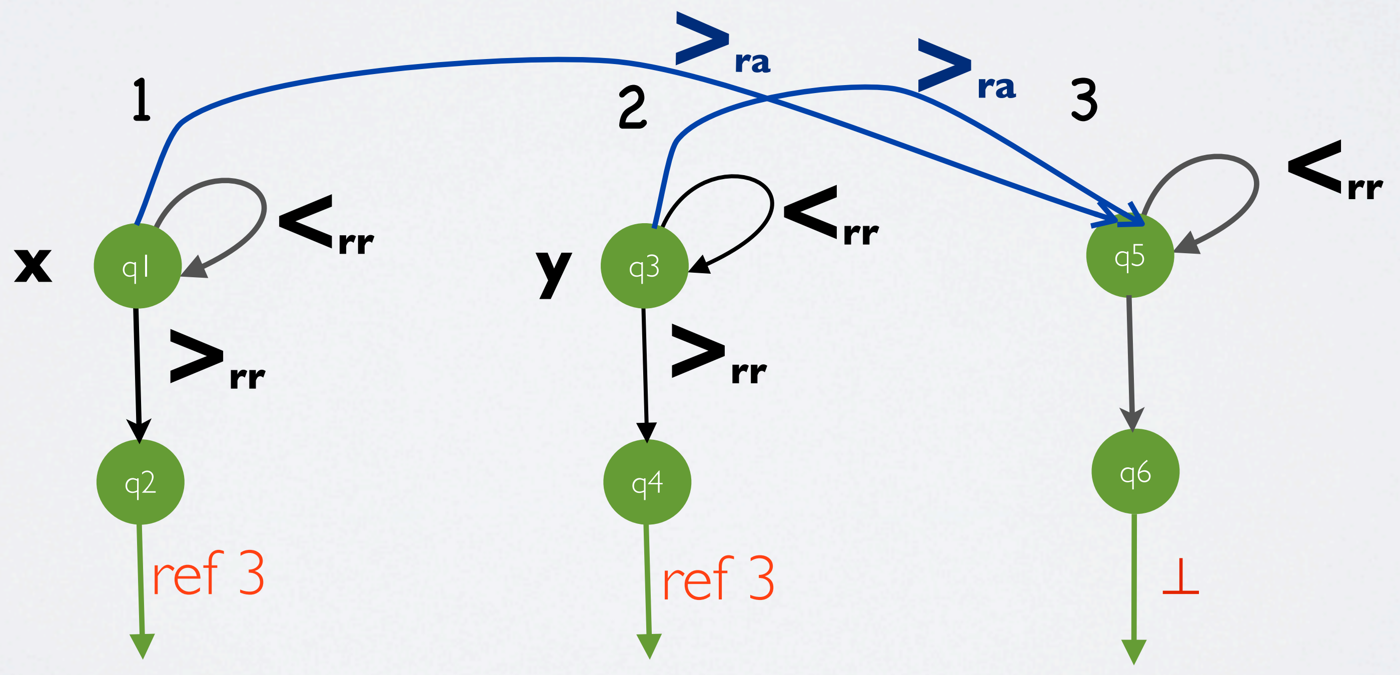
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node



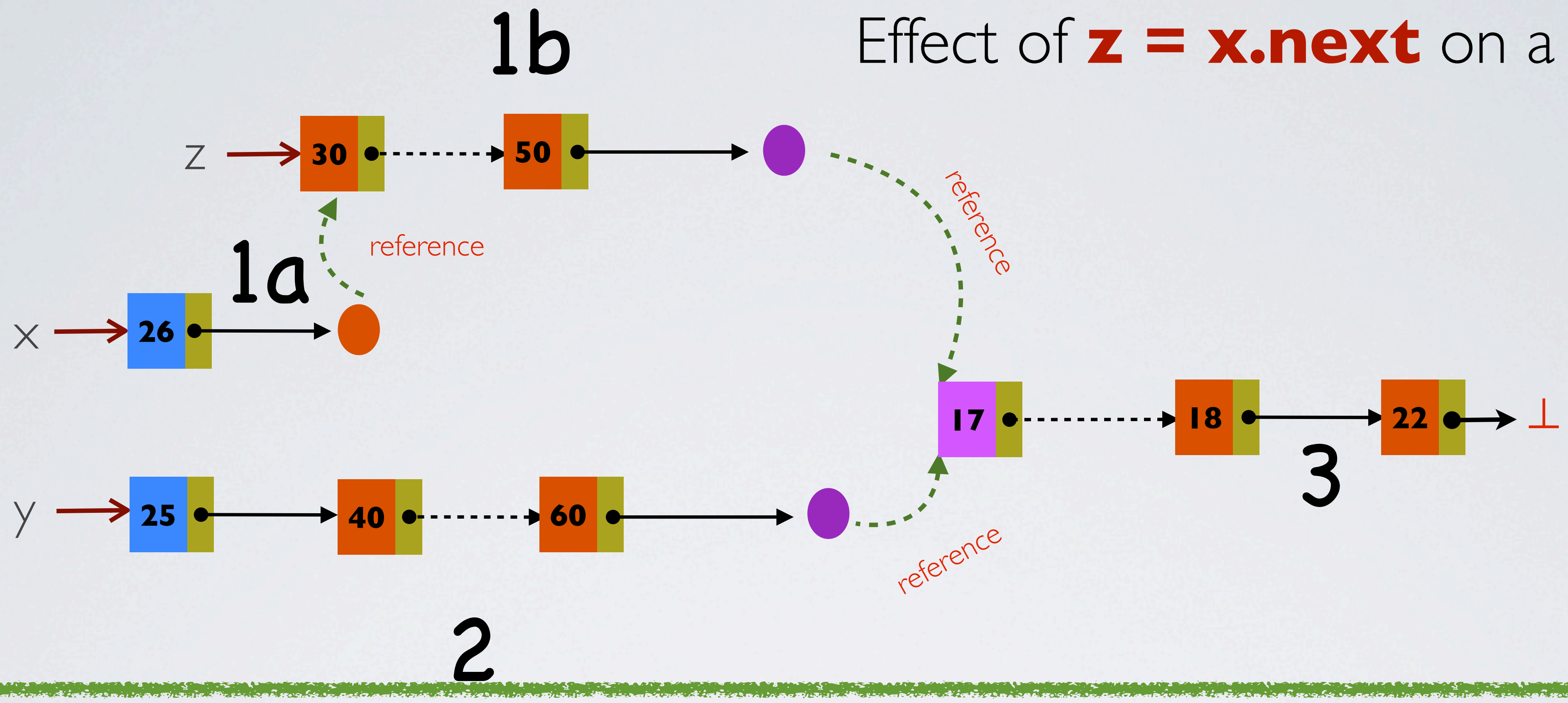
Effect of **z = x.next** on a concrete forest



- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

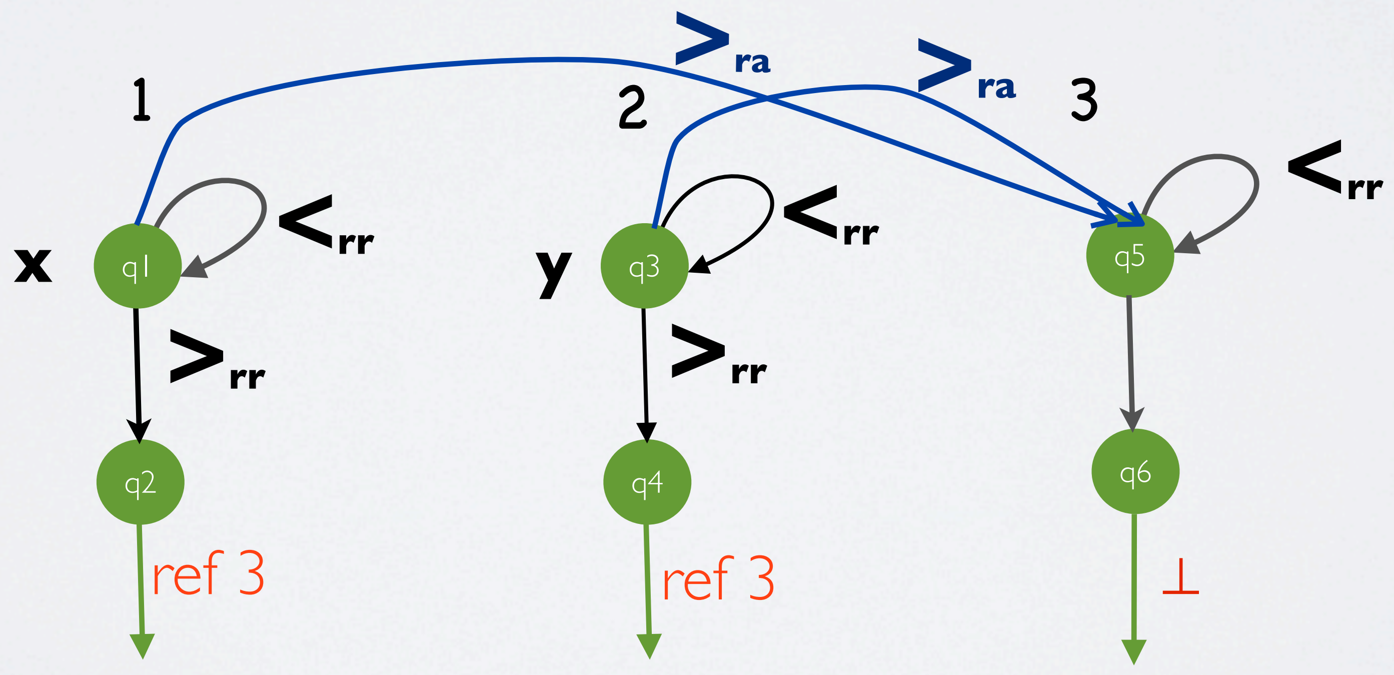


Effect of **z = x.next** on a concrete forest

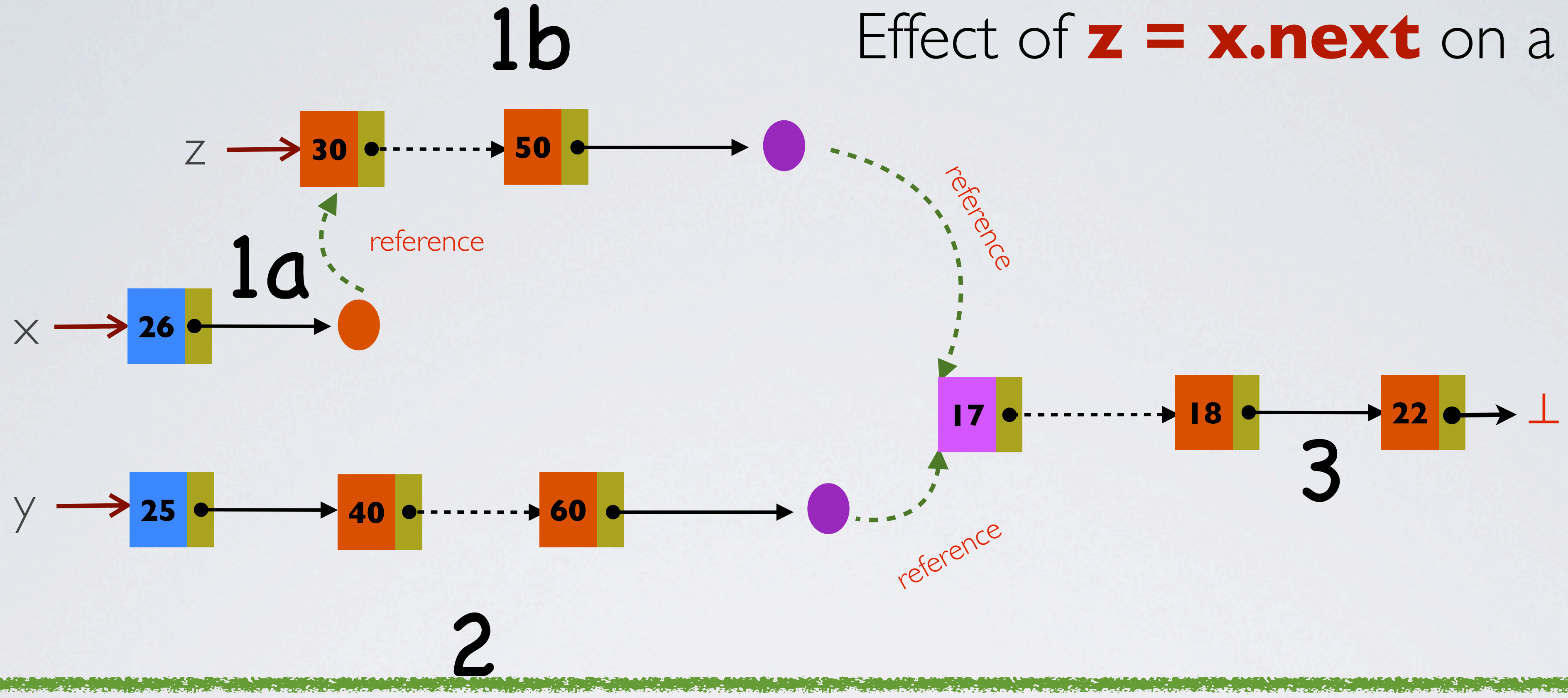


✓ Access to **x.next**
✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata

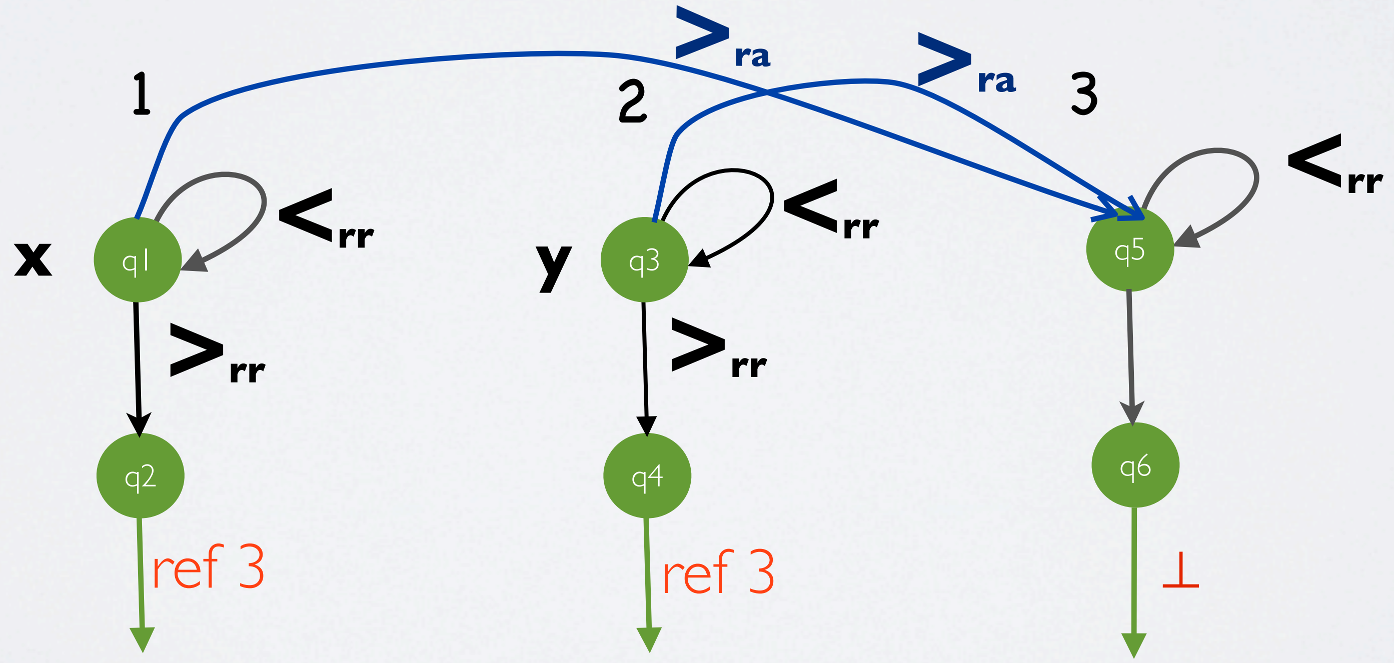


Effect of **z = x.next** on a concrete forest



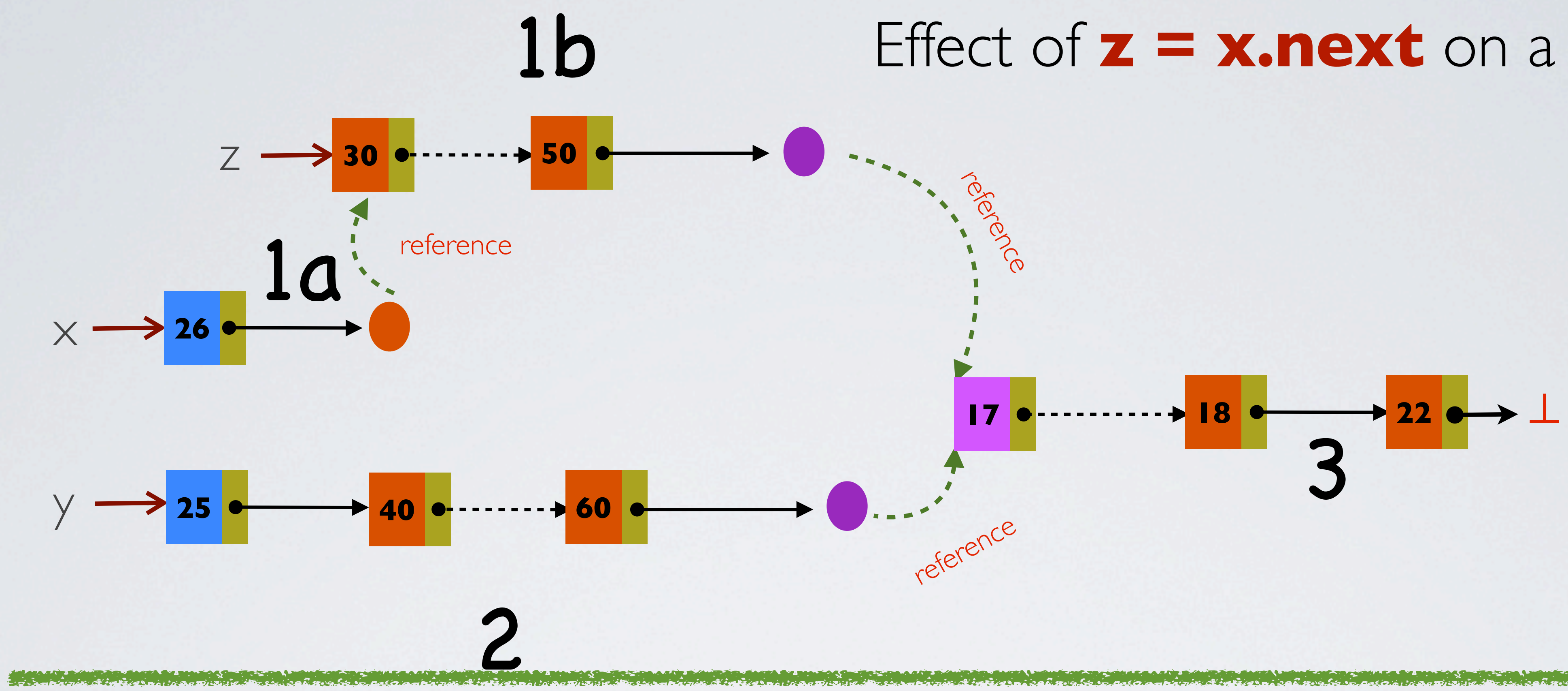
✓ Access to **x.next**
✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



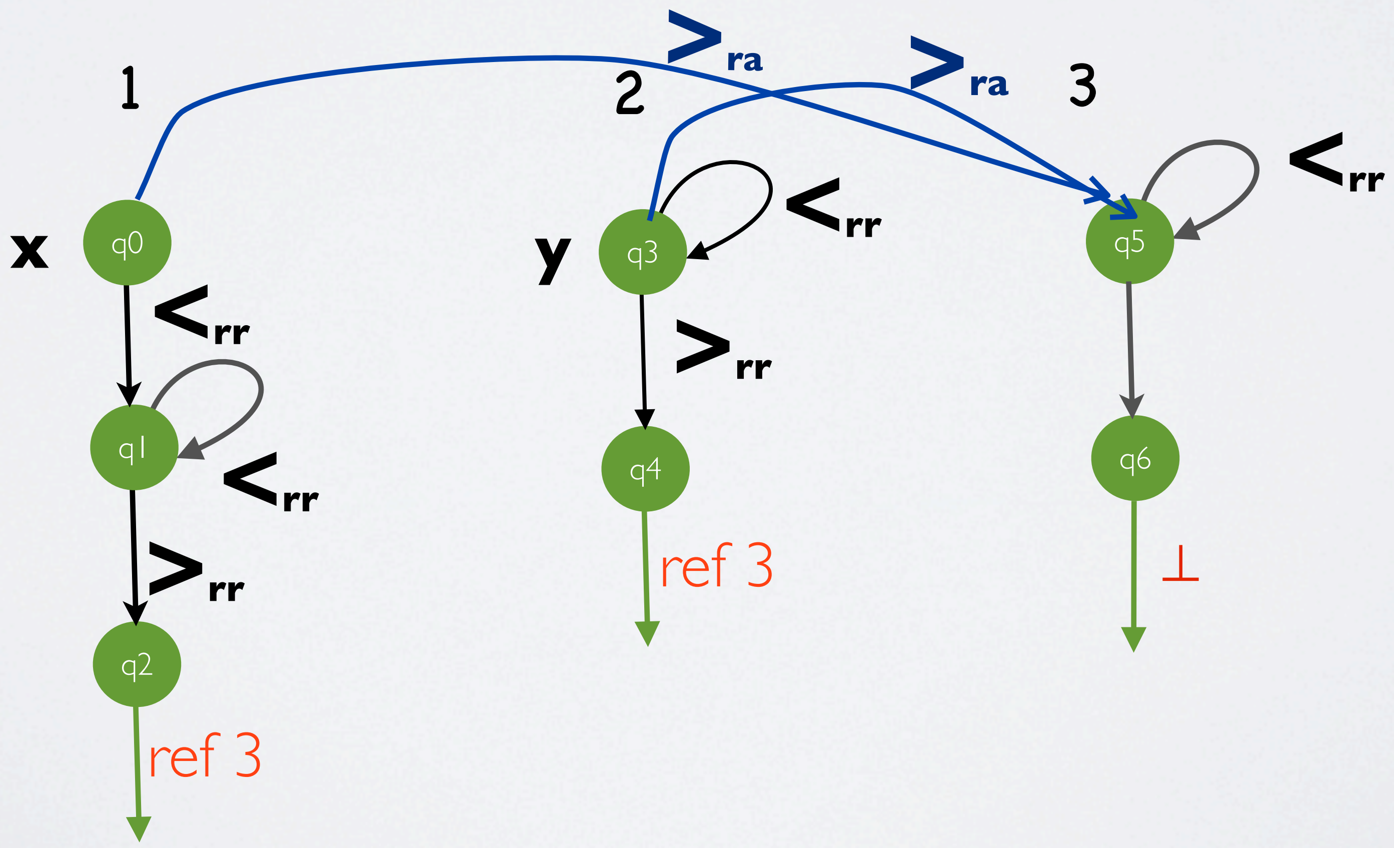
✓ Access to **x.next** by expanding cycle at **q1**
✓ Split the TA1 at the accessed state

Effect of **z = x.next** on a concrete forest



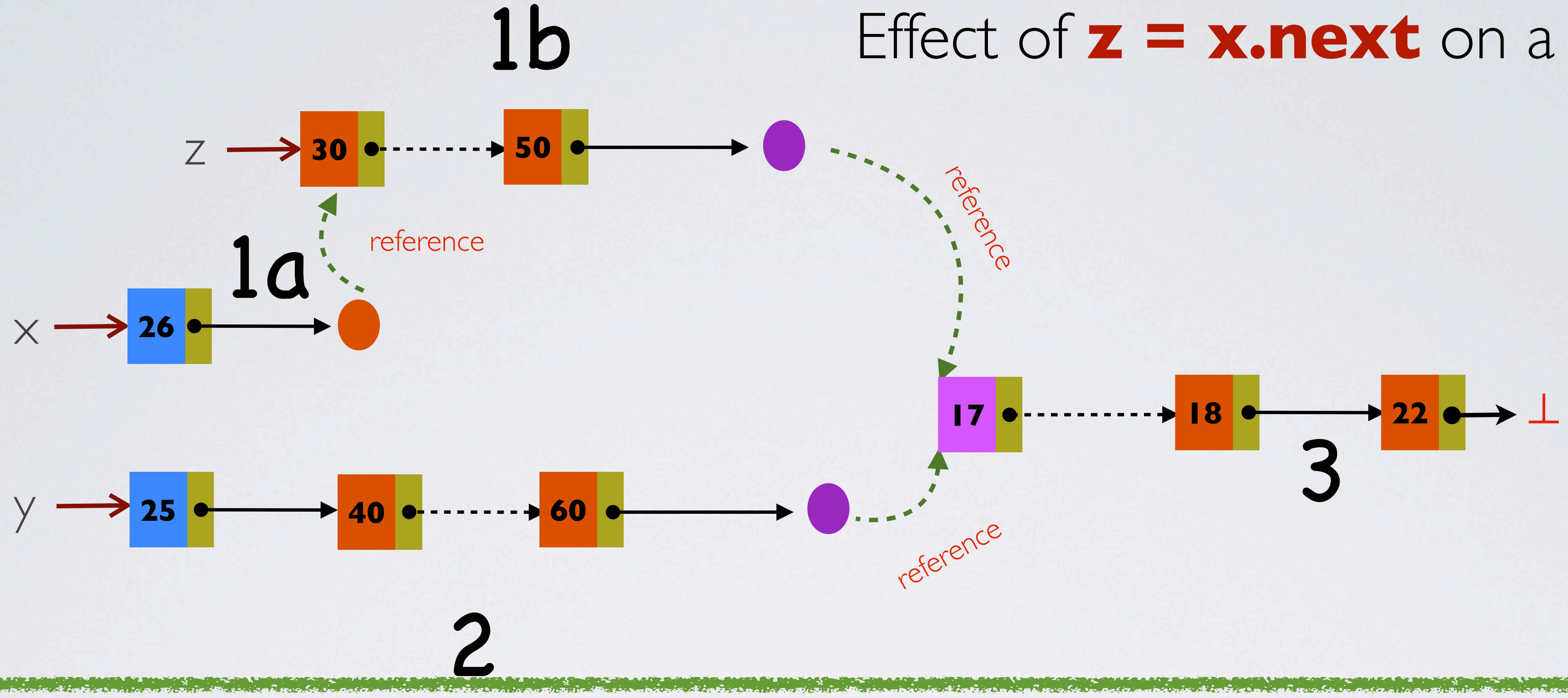
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



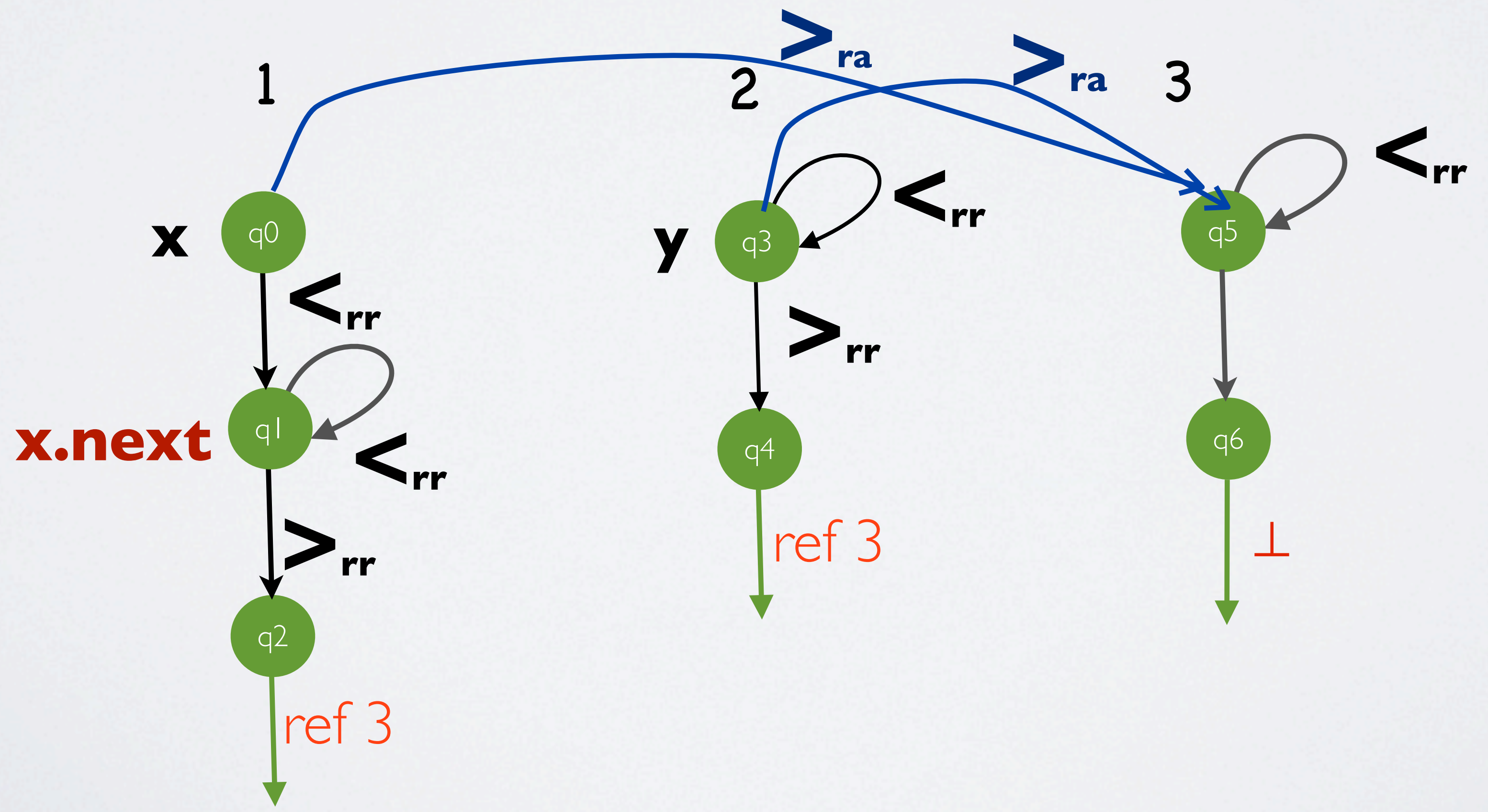
- ✓ Access to **x.next** by expanding cycle at **q1**
- ✓ Split the TA1 at the accessed state

Effect of **z = x.next** on a concrete forest



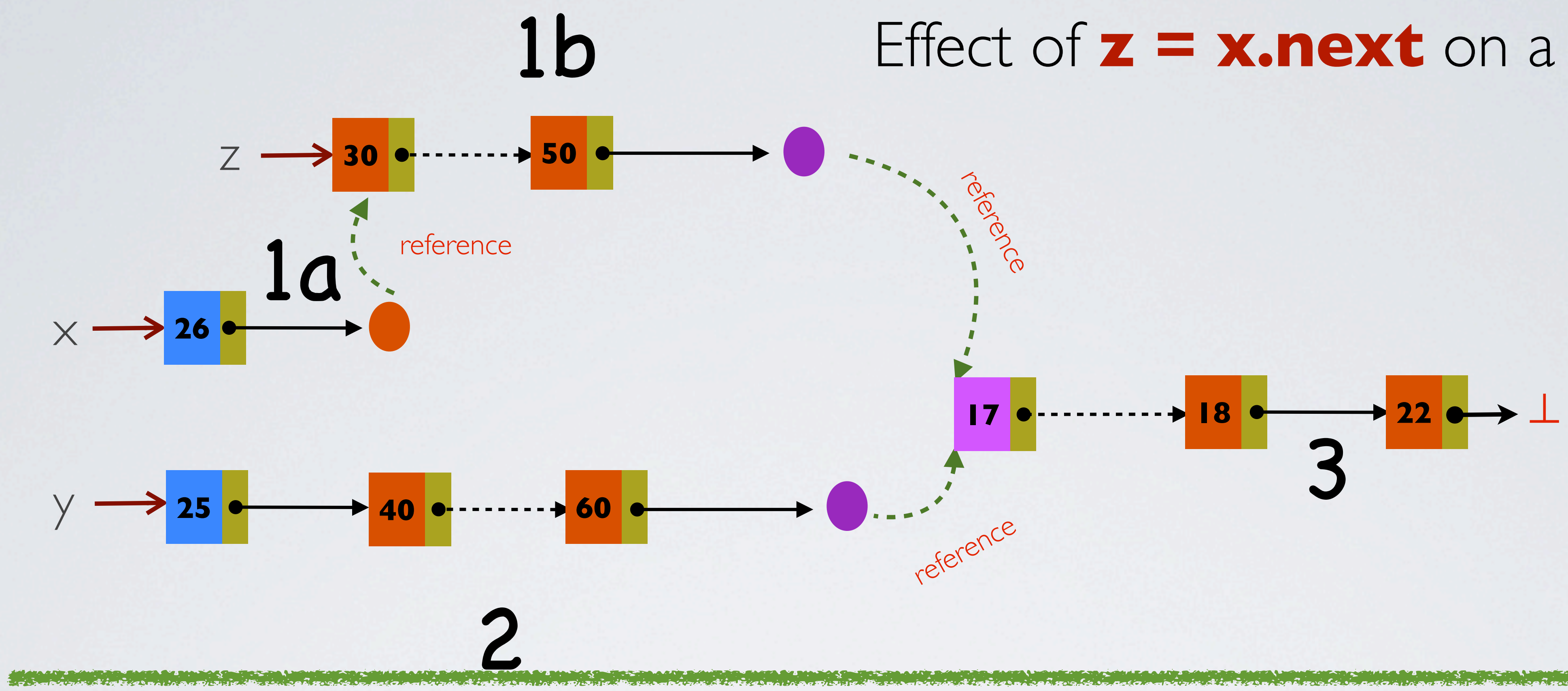
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



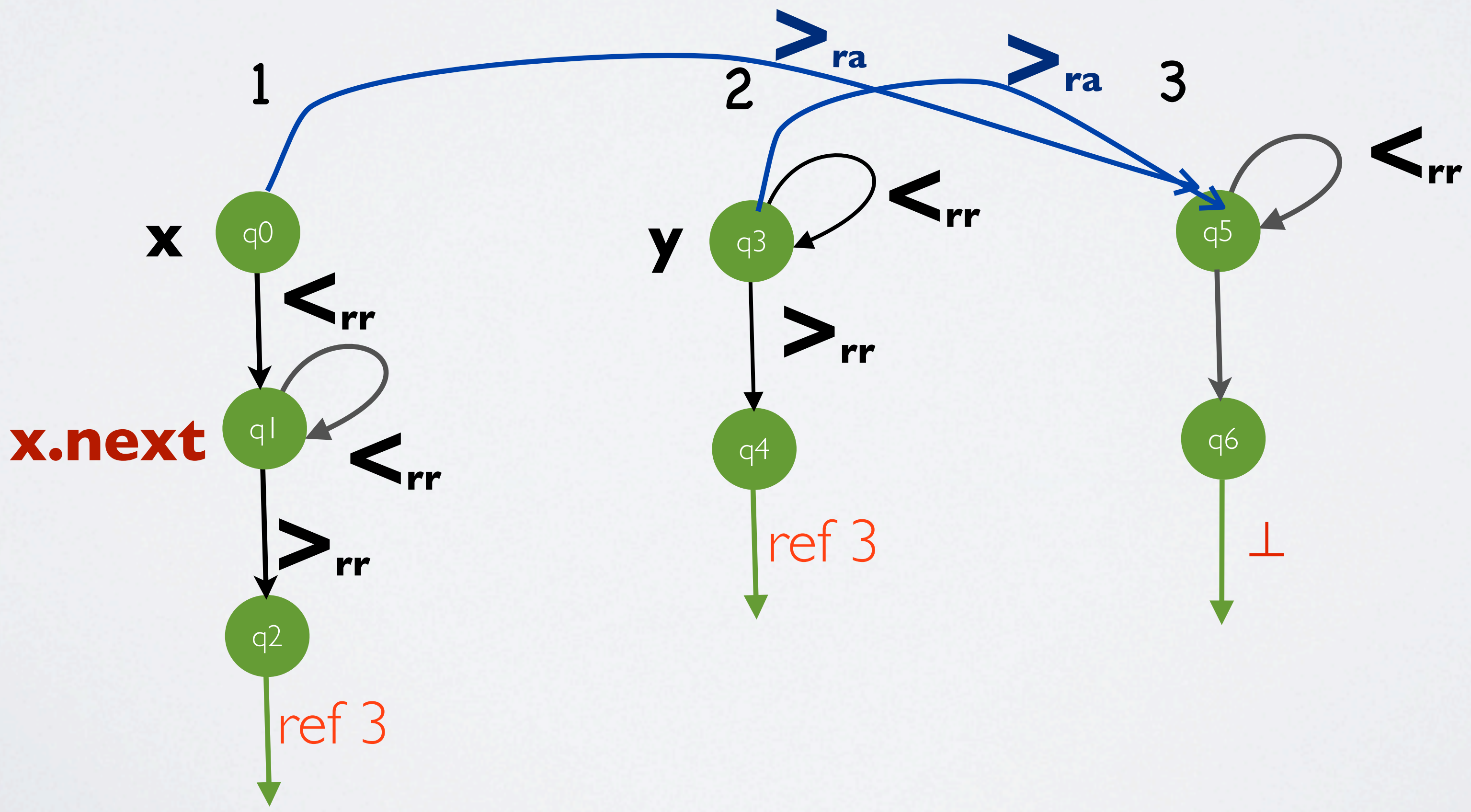
- ✓ Access to **x.next** by expanding cycle at **q1**
- ✓ Split the TA1 at the accessed state

Effect of **z = x.next** on a concrete forest



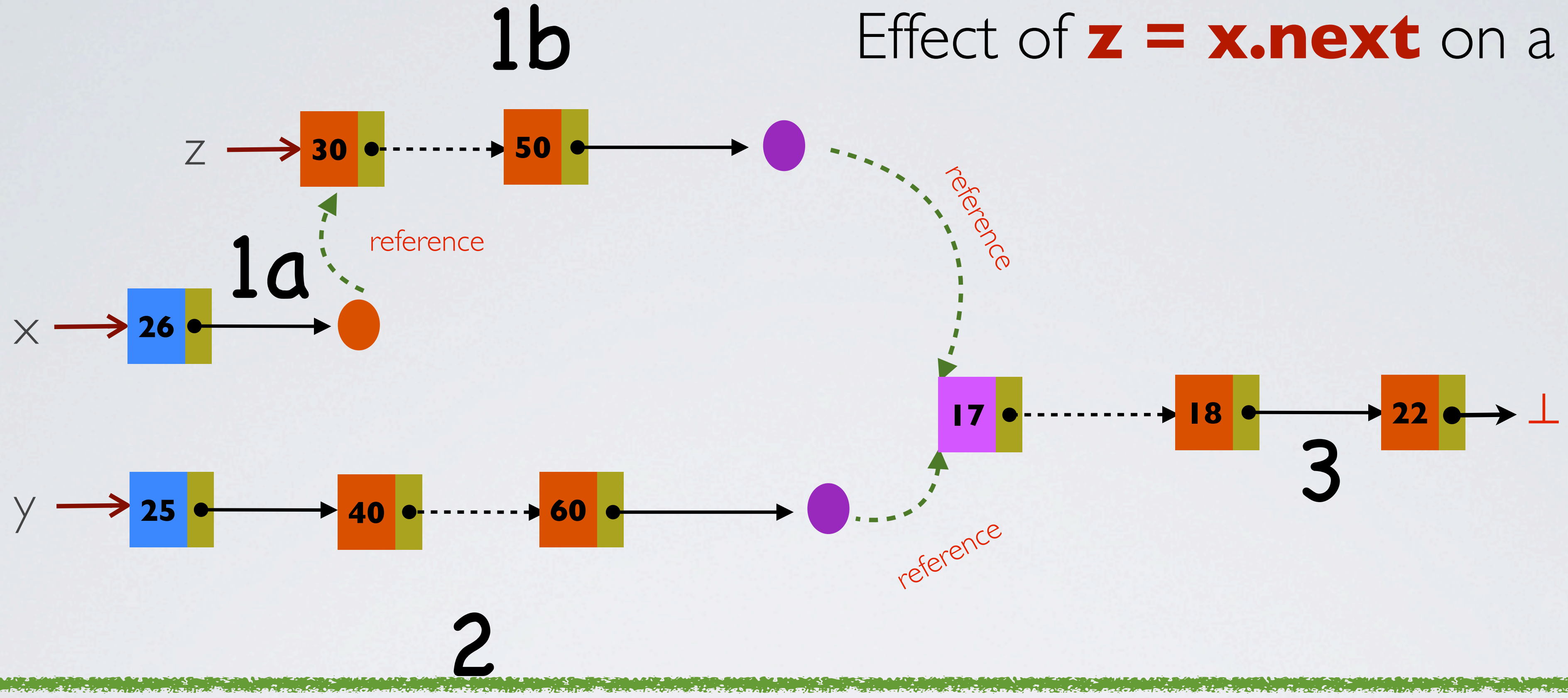
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



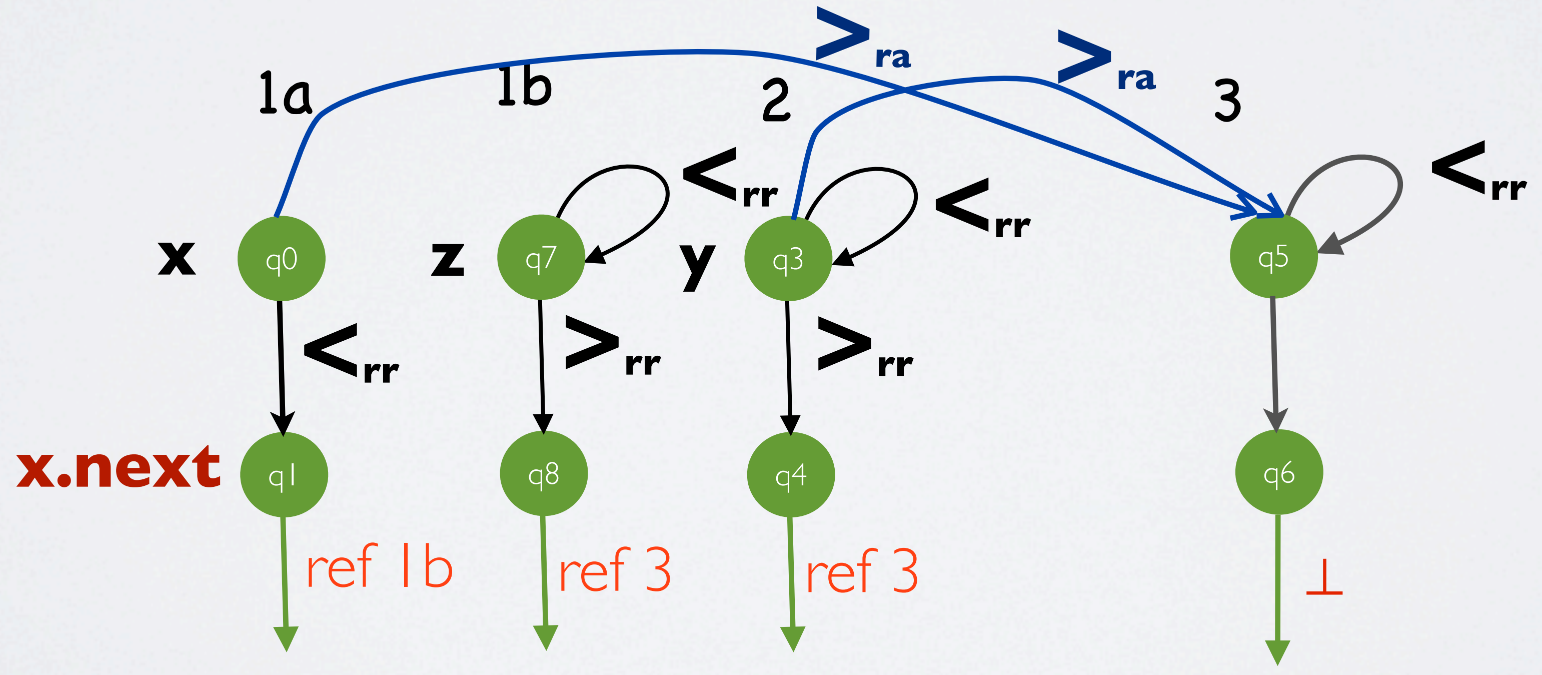
- ✓ Access to **x.next** by expanding cycle at **q1**
- ✓ Split the TA1 at the accessed state

Effect of **z = x.next** on a concrete forest



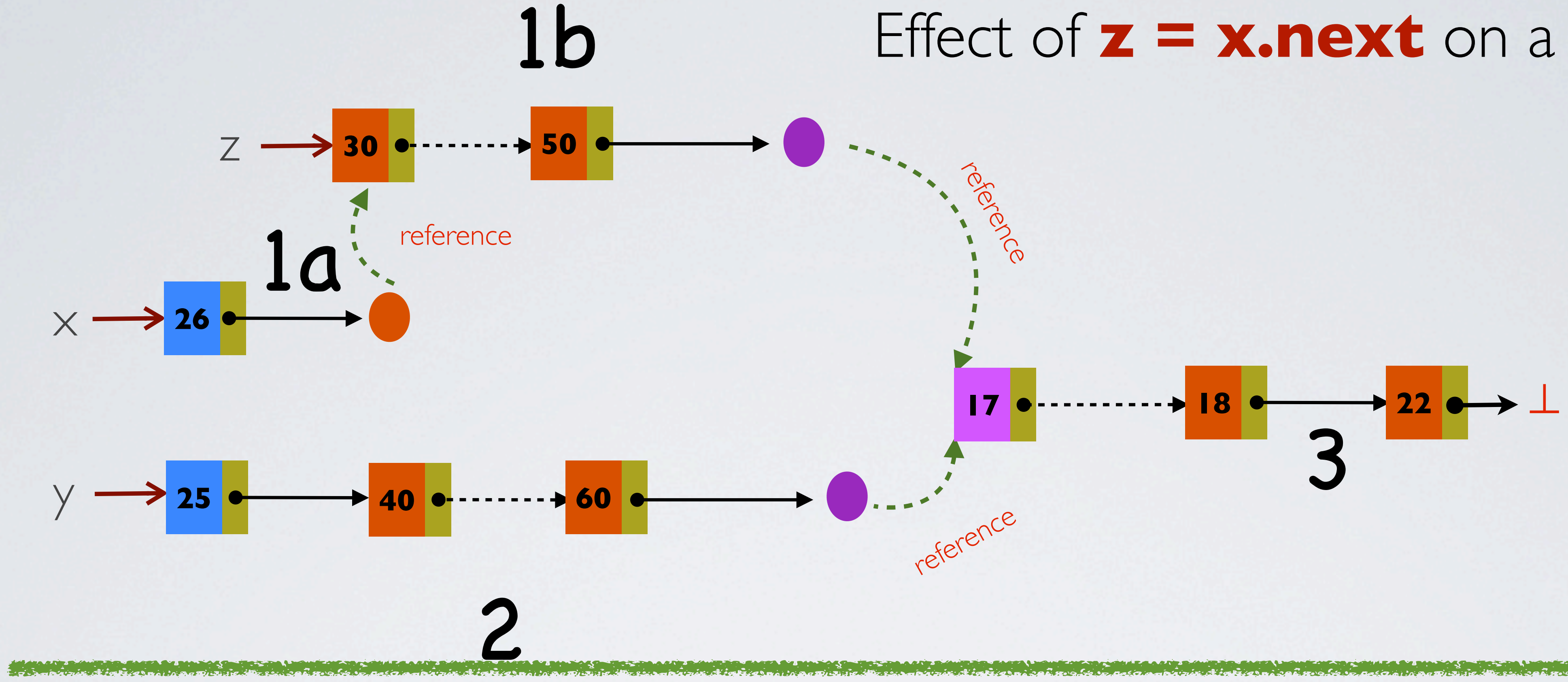
✓ Access to **x.next**
✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata

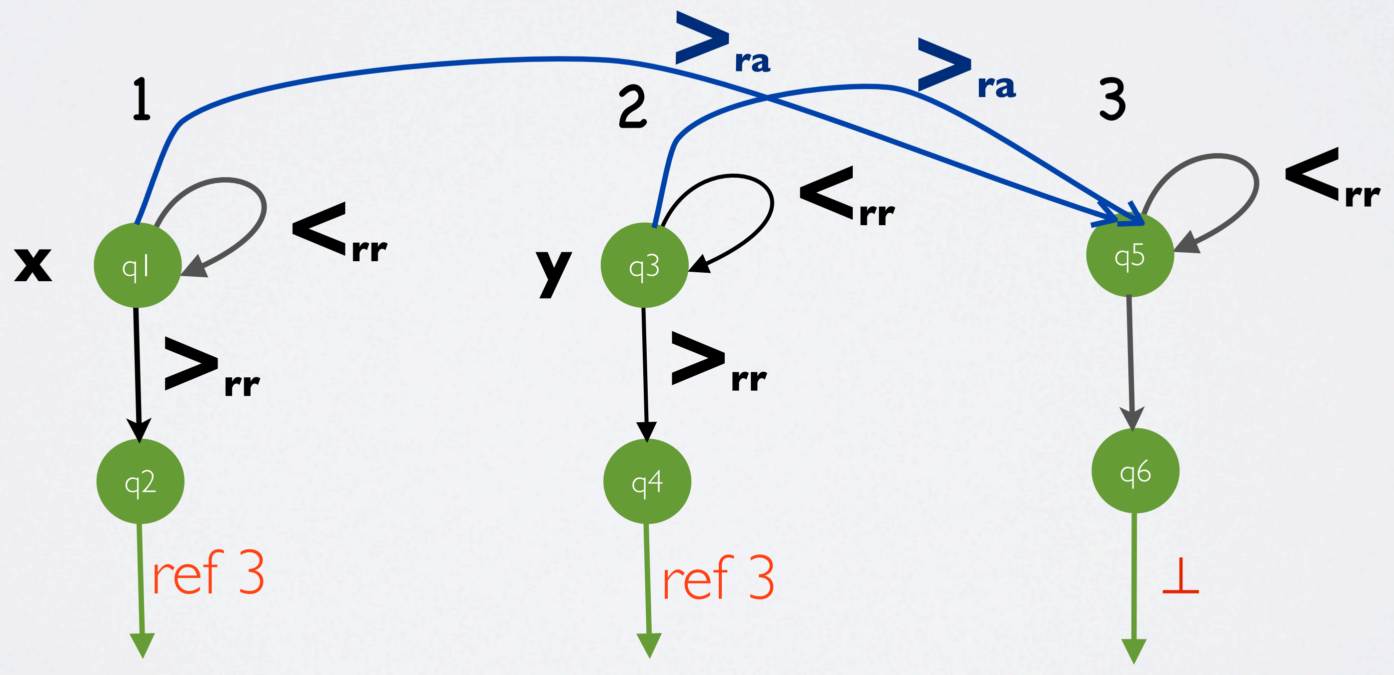


✓ Access to **x.next** by expanding cycle at **q1**
✓ Split the TA1 at the accessed state

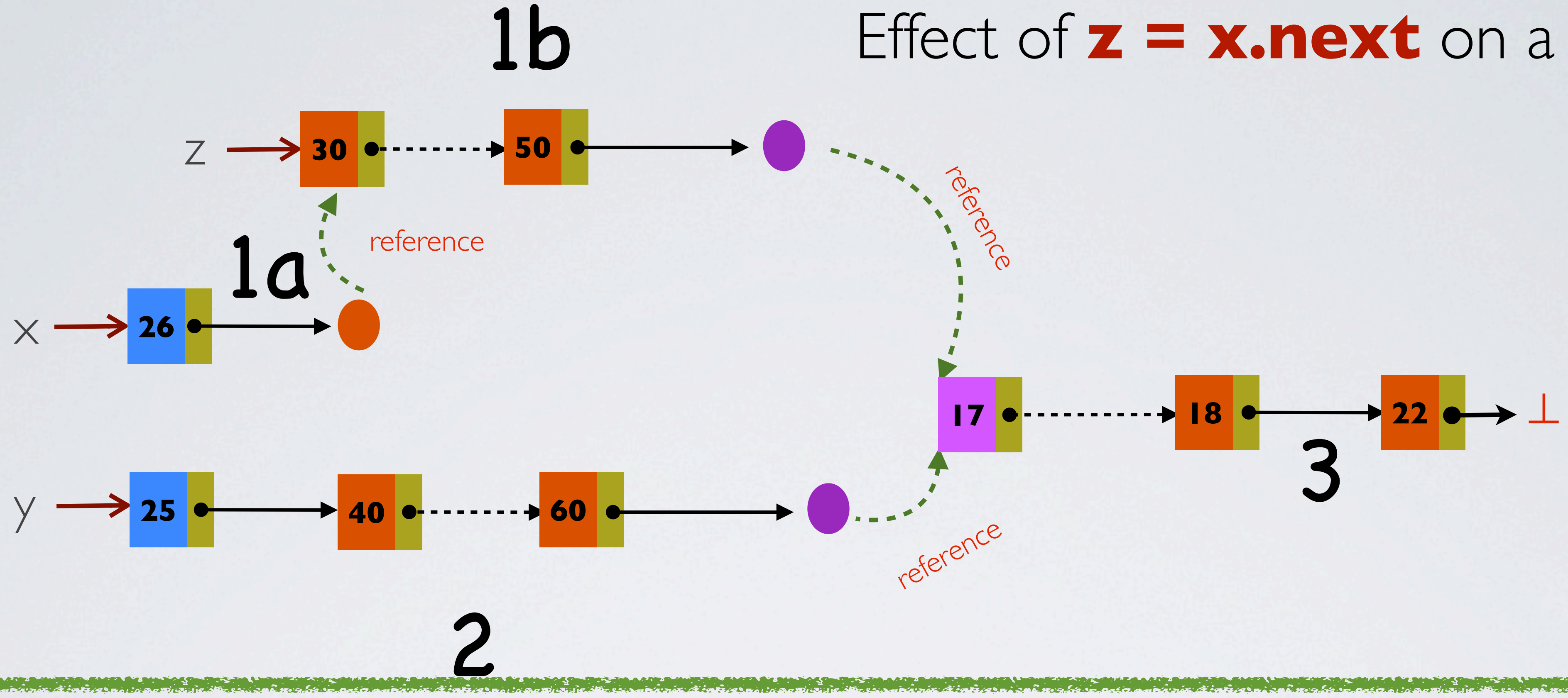
Effect of **z = x.next** on a concrete forest



- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

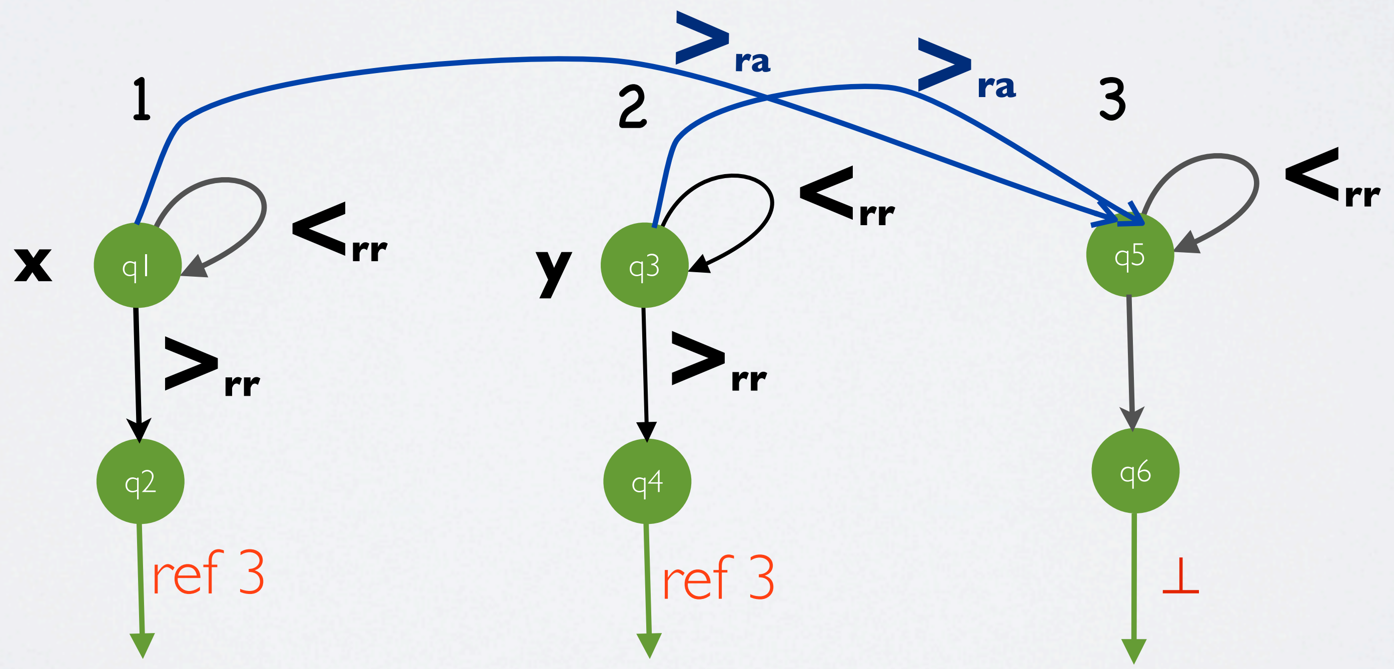


Effect of **z = x.next** on a concrete forest

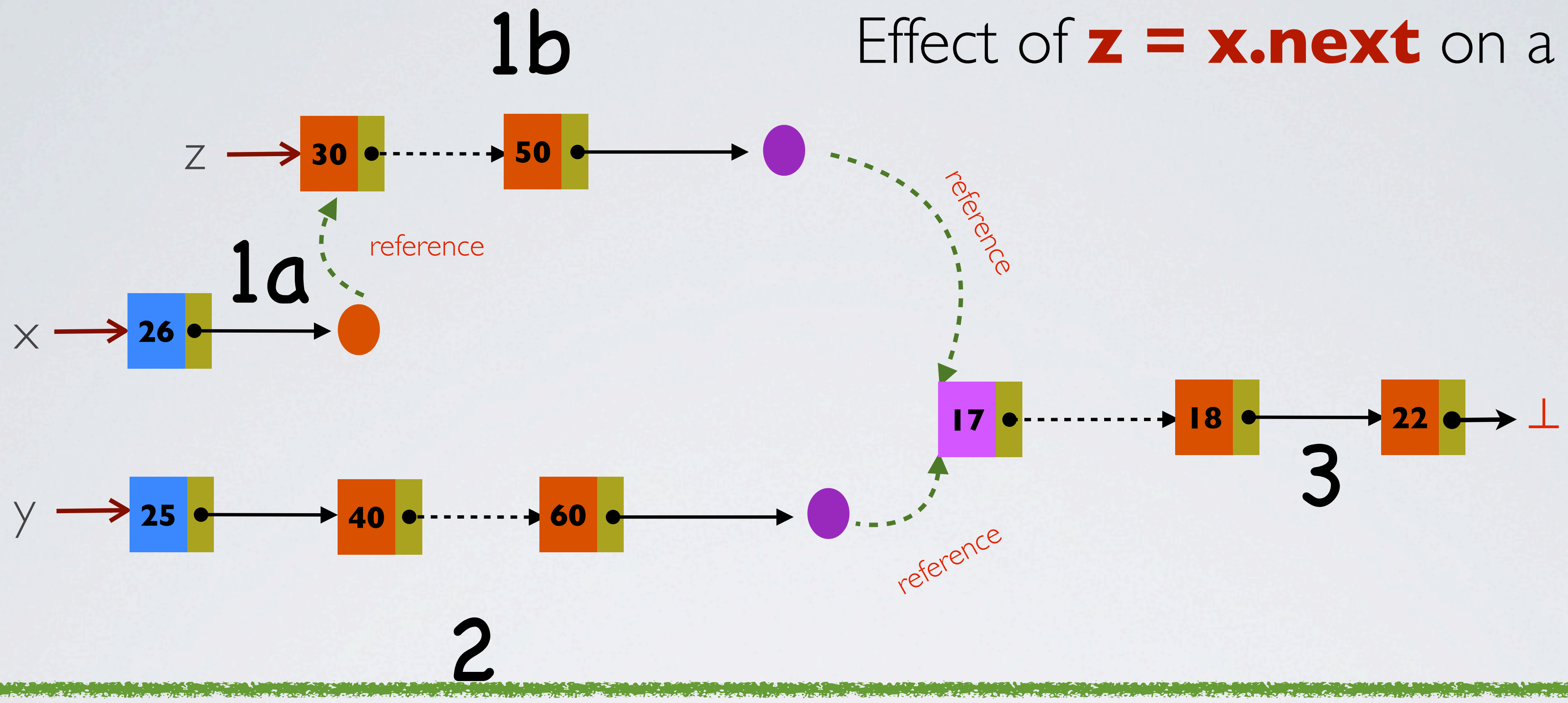


- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata

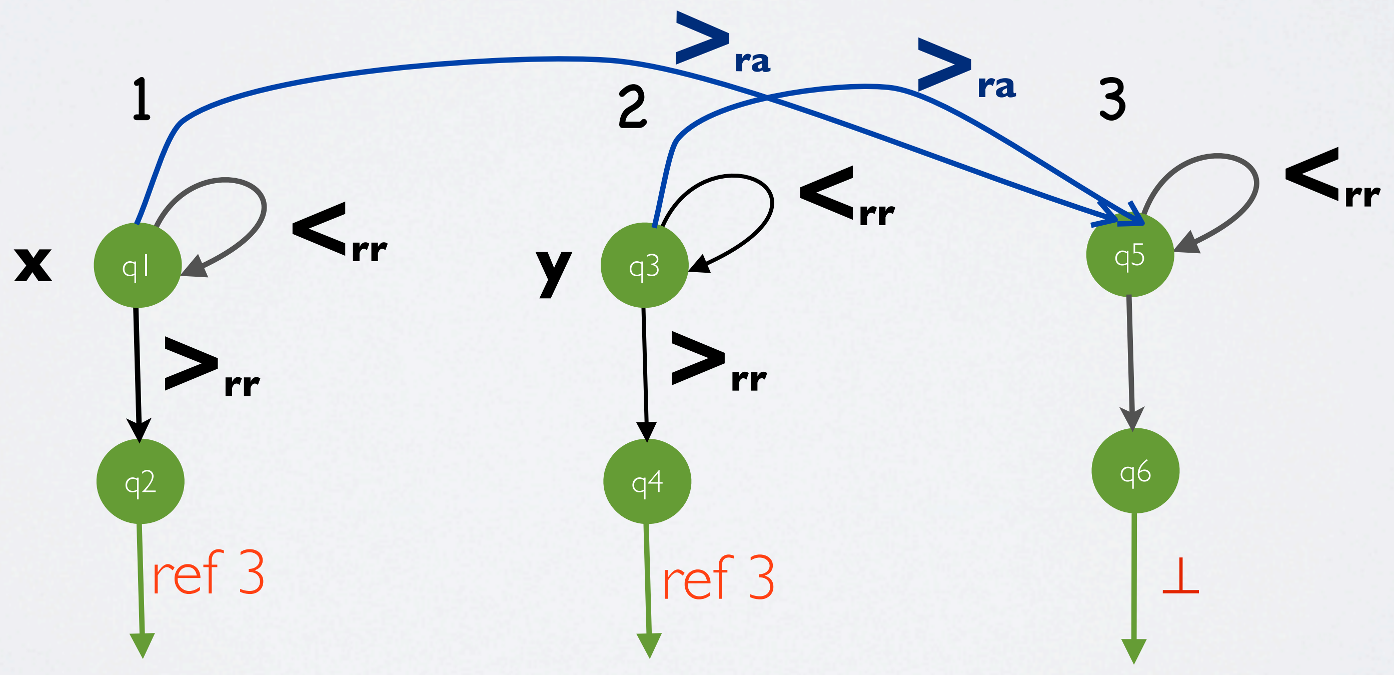


Effect of **z = x.next** on a concrete forest



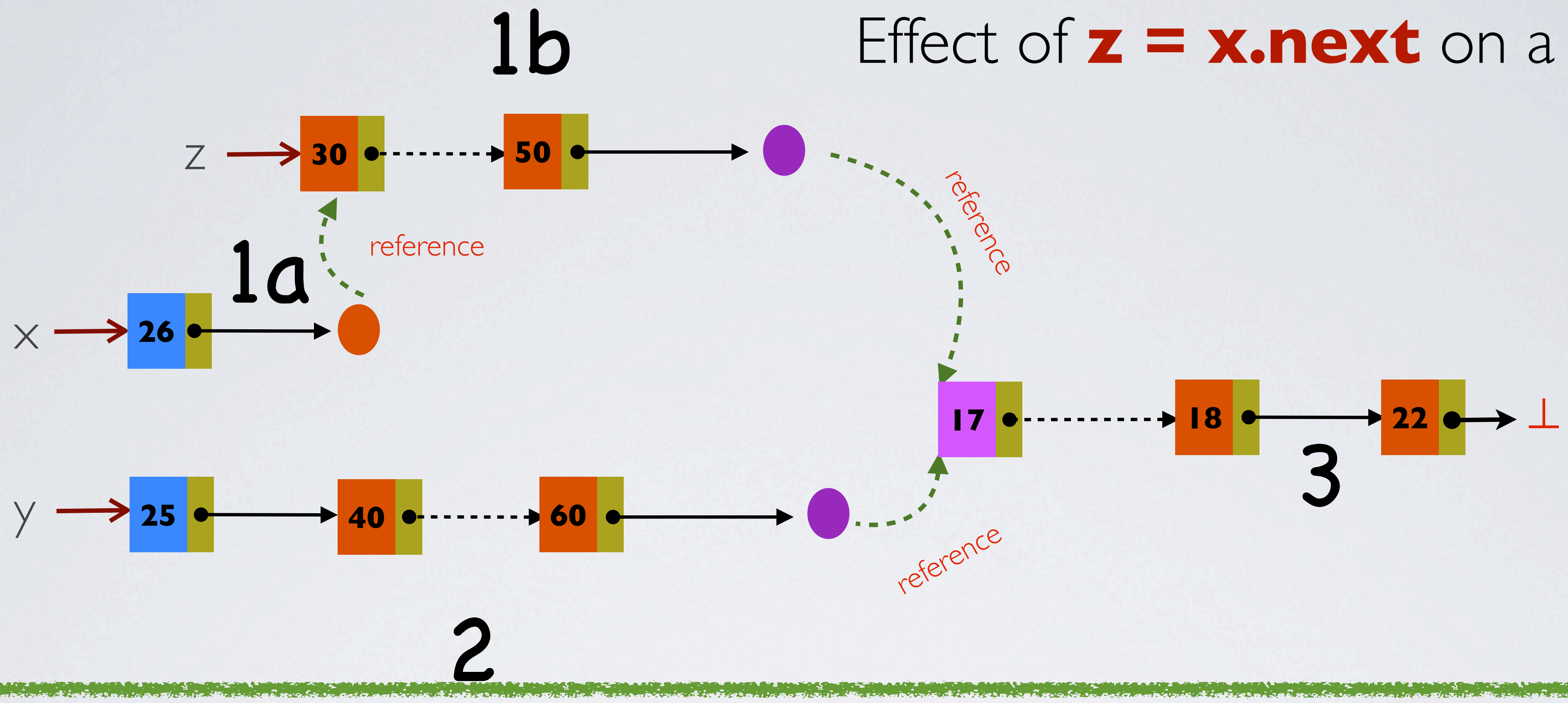
- ✓ Access to **x.next**
- ✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



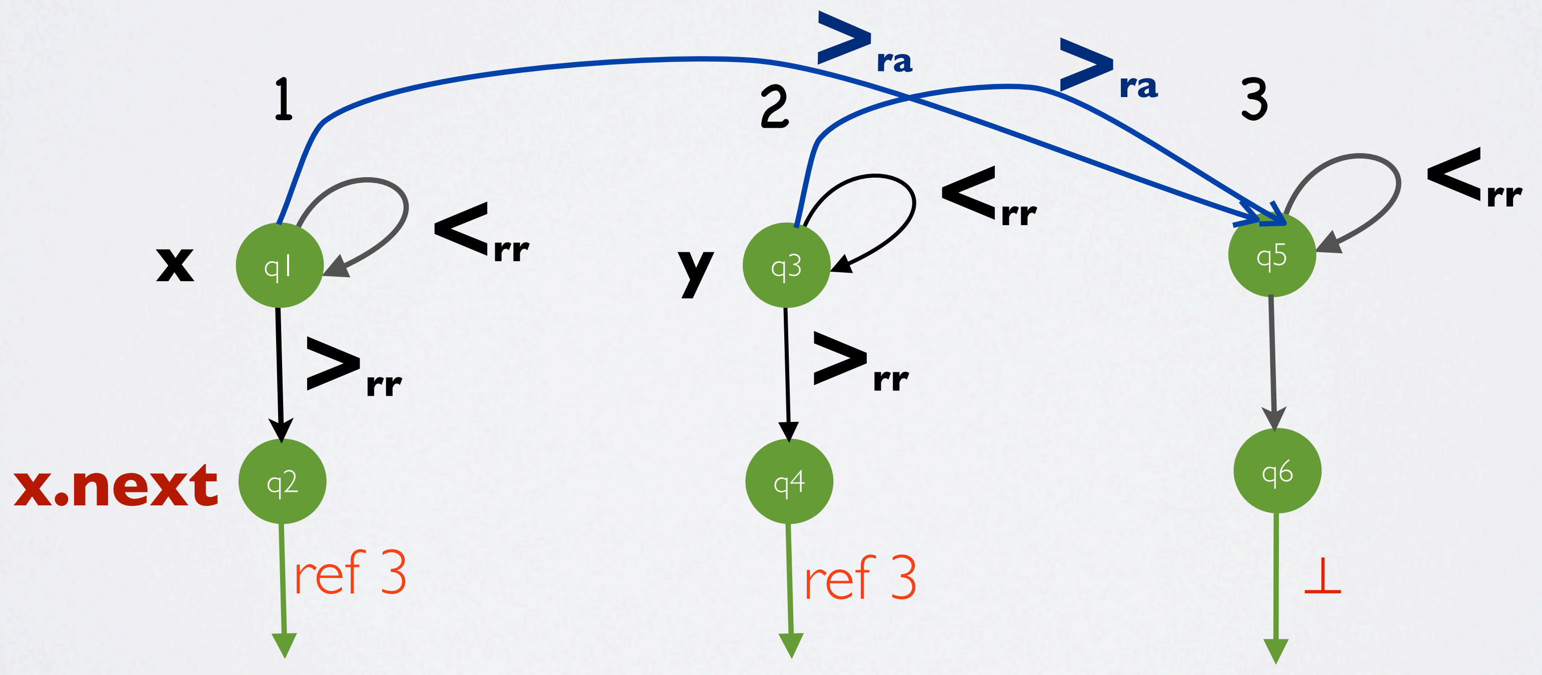
- ✓ Access to **x.next** directly at **q2**
- ✓ Assign variable **z** to the root of the TA 3

Effect of **z = x.next** on a concrete forest



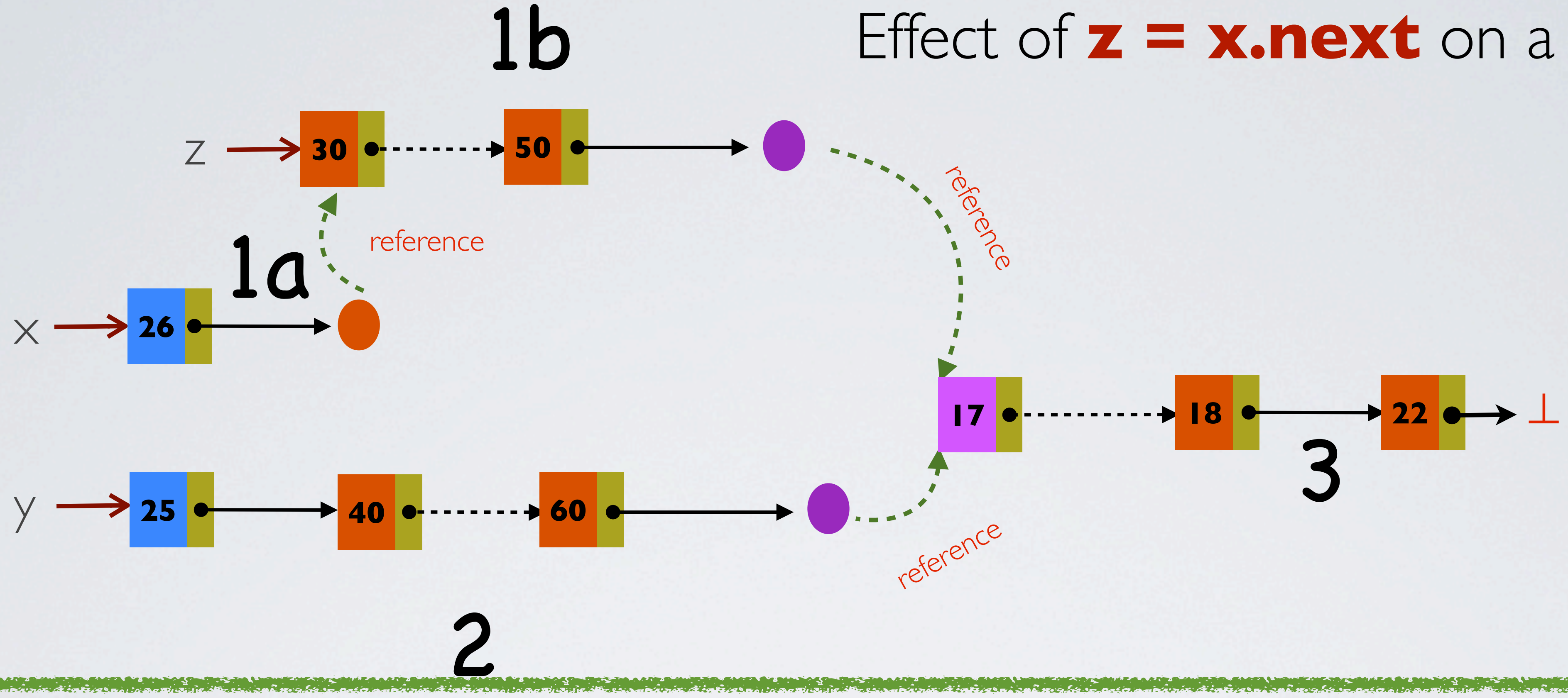
✓ Access to **x.next**
✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



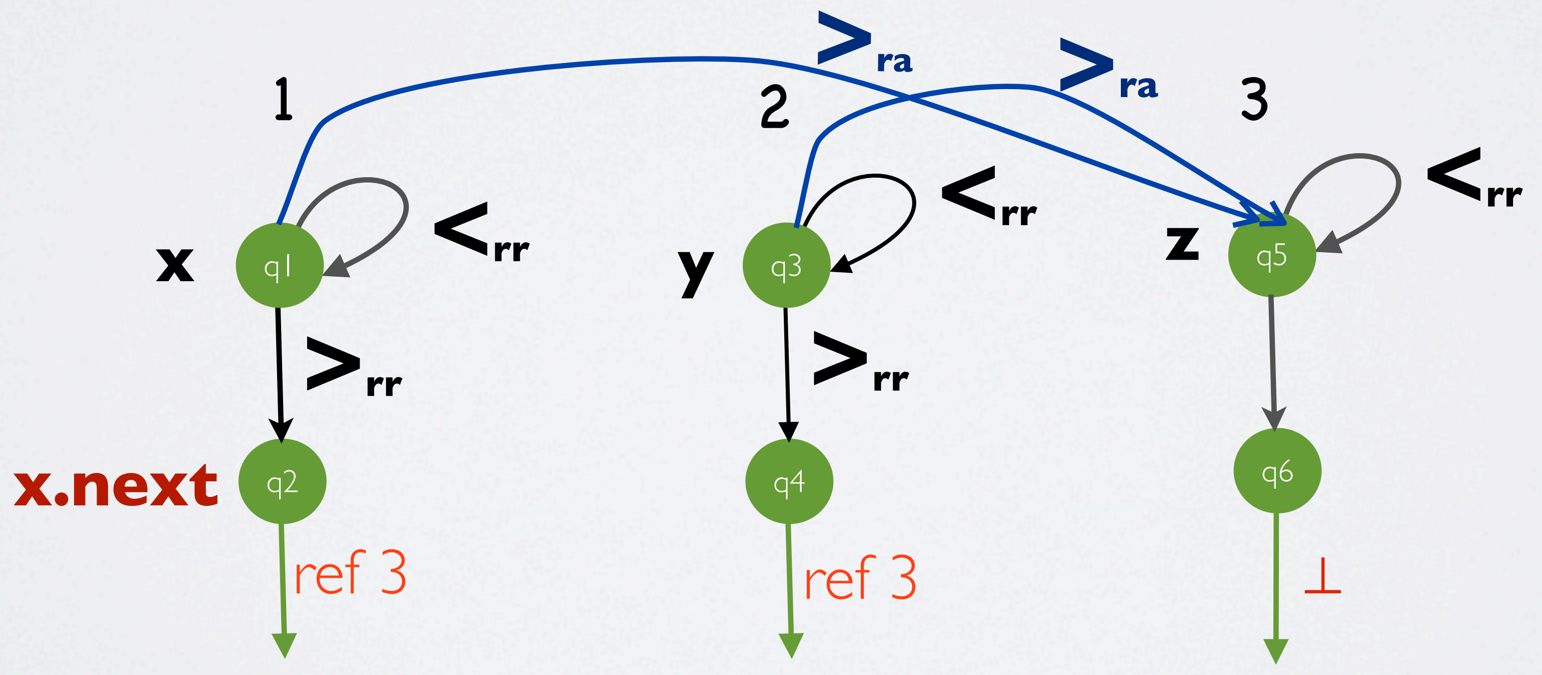
✓ Access to **x.next** directly at **q2**
✓ Assign variable **z** to the root of the TA 3

Effect of **z = x.next** on a concrete forest



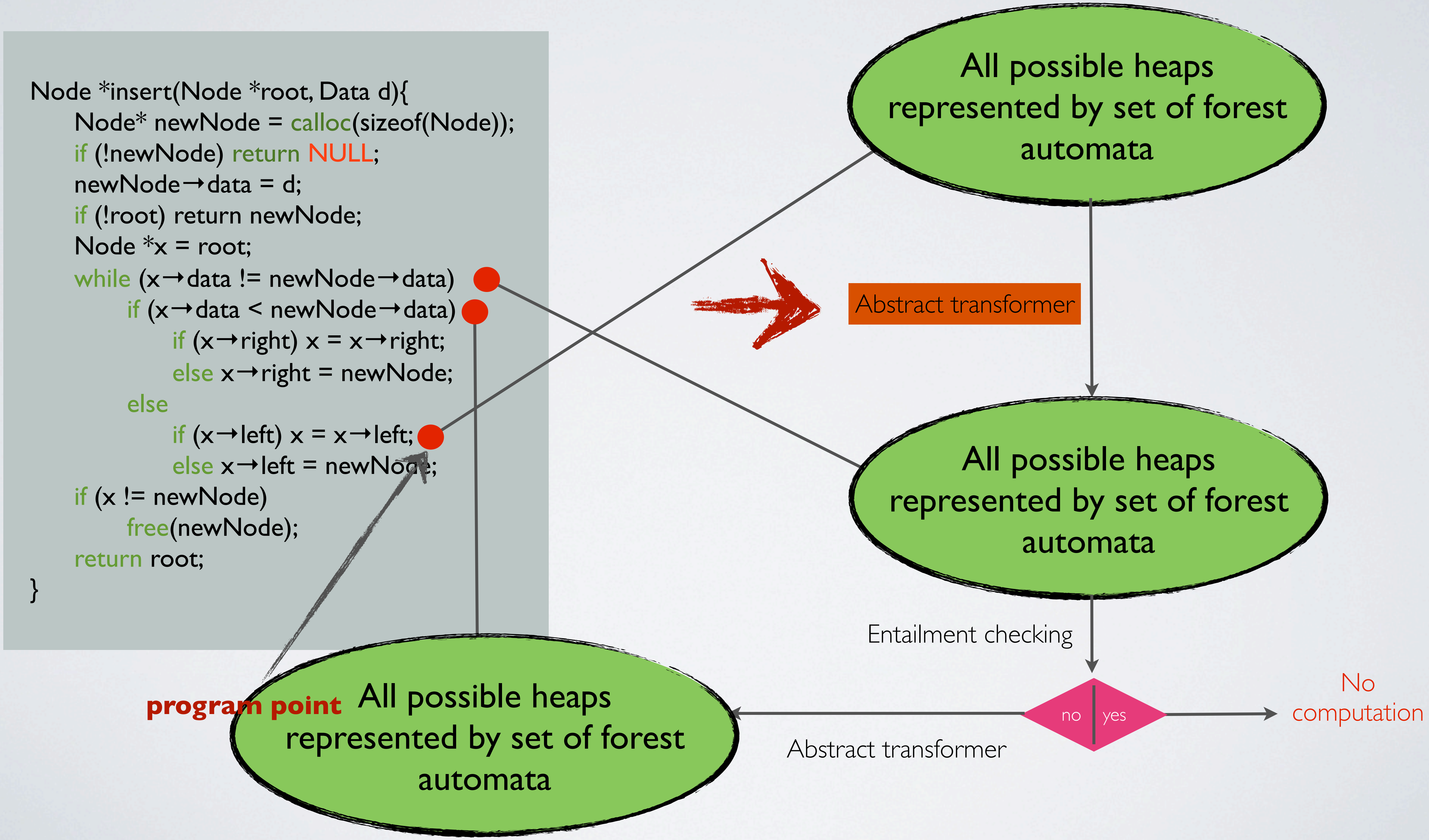
✓ Access to **x.next**
✓ Split the tree 1 at this node

Effect of **z = x.next** on forest automata



✓ Access to **x.next** directly at **q2**
✓ Assign variable **z** to the root of the TA 3

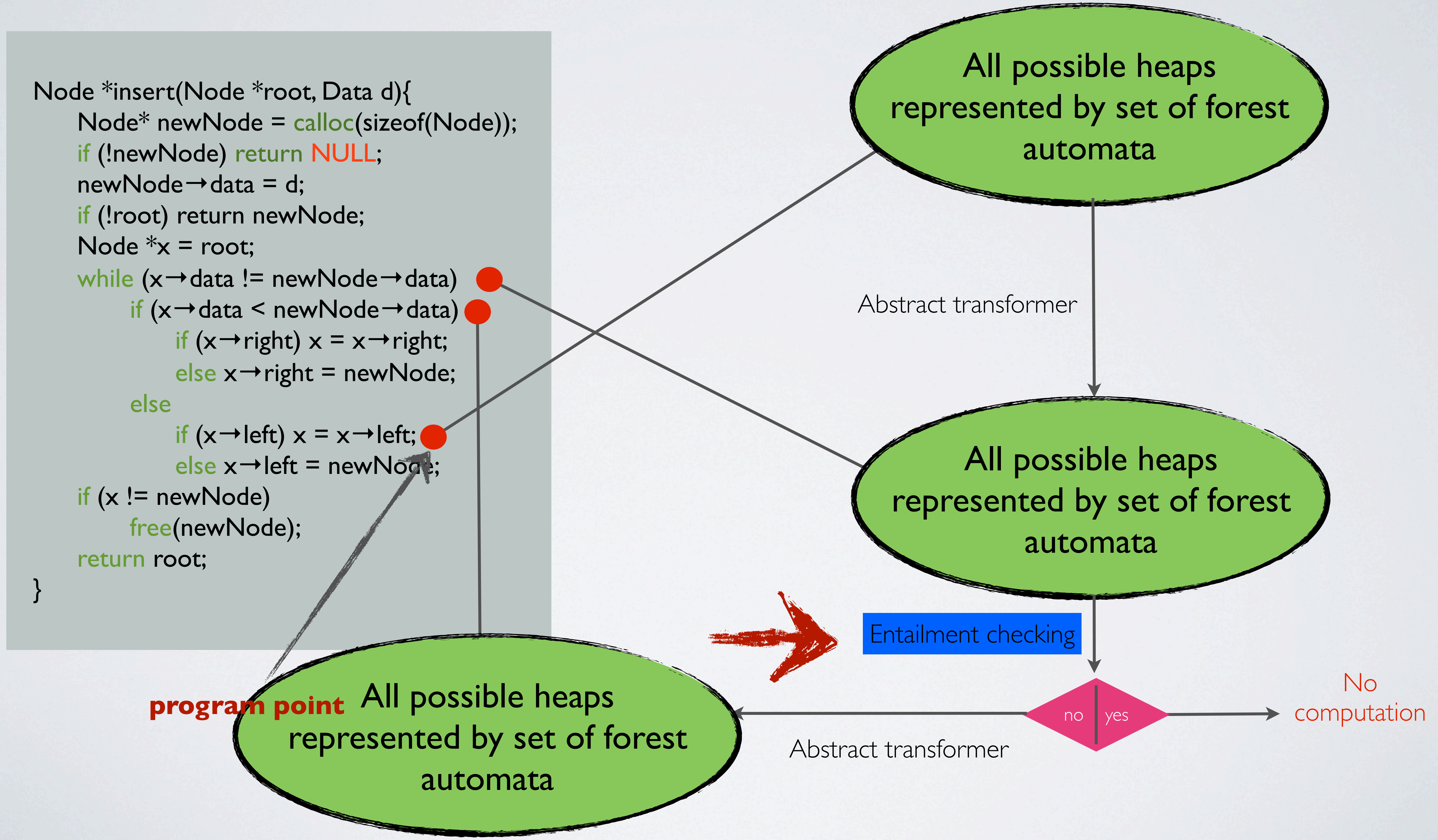
Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

Program analysis



```
Node *insert(Node *root, Data d){
    Node* newNode = calloc(sizeof(Node));
    if (!newNode) return NULL;
    newNode→data = d;
    if (!root) return newNode;
    Node *x = root;
    while (x→data != newNode→data)
        if (x→data < newNode→data)
            if (x→right) x = x→right;
            else x→right = newNode;
        else
            if (x→left) x = x→left;
            else x→left = newNode;
    if (x != newNode)
        free(newNode);
    return root;
}
```

All possible heaps represented by set of forest automata
Abstract transformer
All possible heaps represented by set of forest automata
Entailment checking
no
yes
No computation
Abstract transformer
program point
All possible heaps represented by set of forest automata

Entailment checking

Widening technique

Language Inclusion

EXPERIMENTS

Examples	Time (in s)
SLL insert	0.06
SLL delete	0.08
SLL reverse	0.07
SLL bubblesort	0.13
SLL insert-sort	0.10

Singly linked list(SLL)

Examples	Time (in s)
BST insert	6.87
BST delete	15.8
BST left rotate	7.35
BST right rotate	6.25

Binary search tree(BST)

Examples	Time (in s)
DLL insert	0.14
DLL delete	0.38
DLL reverse	0.16
DLL bubblesort	0.39
DLL insert-sort	0.43

Double linked list(DLL)

Examples	Time (in s)
SL2 insert	9.65
SL2 delete	10.14
SL3 insert	56.99
SL3 delete	57.35

Skiplist with 2 & 3 levels(SL2 & SL3)

EXPERIMENTS

Examples	Time (in s)
SLL insert	0.06
SLL delete	0.08
SLL reverse	0.07
SLL bubblesort	0.13
SLL insert-sort	0.10

Singly linked list(SLL)

Safety

Examples	Time (in s)
BST insert	6.87
BST delete	15.8
BST left rotate	7.35
BST right rotate	6.25

Binary search tree(BST)

Examples	Time (in s)
DLL insert	0.14
DLL delete	0.38
DLL reverse	0.16
DLL bubblesort	0.39
DLL insert-sort	0.43

Double linked list(DLL)

Examples	Time (in s)
SL2 insert	9.65
SL2 delete	10.14
SL3 insert	56.99
SL3 delete	57.35

Skiplist with 2 & 3 levels(SL2 & SL3)

EXPERIMENTS

Examples	Time (in s)
SLL insert	0.06
SLL delete	0.08
SLL reverse	0.07
SLL bubblesort	0.13
SLL insert-sort	0.10

Singly linked list(SLL)

Examples	Time (in s)
BST insert	6.87
BST delete	15.8
BST left rotate	7.35
BST right rotate	6.25

Binary search tree(BST)

Safety

Examples	Time (in s)
DLL insert	0.14
DLL delete	0.38
DLL reverse	0.16
DLL bubblesort	0.39
DLL insert-sort	0.43

Double linked list(DLL)

Sortedness

Examples	Time (in s)
SL2 insert	9.65
SL2 delete	10.14
SL3 insert	56.99
SL3 delete	57.35

Skiplist with 2 & 3 levels(SL2 & SL3)

EXPERIMENTS

Examples	Time (in s)
SLL insert	0.06
SLL delete	0.08
SLL reverse	0.07
SLL bubblesort	0.13
SLL insert-sort	0.10

Singly linked list(SLL)

Safety

Examples	Time (in s)
BST insert	6.87
BST delete	15.8
BST left rotate	7.35
BST right rotate	6.25

Binary search tree(BST)

Examples	Time (in s)
DLL insert	0.14
DLL delete	0.38
DLL reverse	0.16
DLL bubblesort	0.39
DLL insert-sort	0.43

Double linked list(DLL)

Sortedness

Examples	Time (in s)
SL2 insert	9.65
SL2 delete	10.14
SL3 insert	56.99
SL3 delete	57.35

Skiplist with 2 & 3 levels(SL2 & SL3)

SUMMARY

- Verify heap manipulating programs with
- Data dependence
- Unbounded heaps
- Multiple selectors
- We can verify both memory safety and data-dependent properties

FUTURE WORKS

- Fine-grained locking programs
- Concurrent heap manipulating programs
- Recursive heap manipulating programs

Thank you for attention!