Abstract Local Reasoning for Concurrent Libraries: Mind the Gap

Philippa Gardner

Azalea Raad

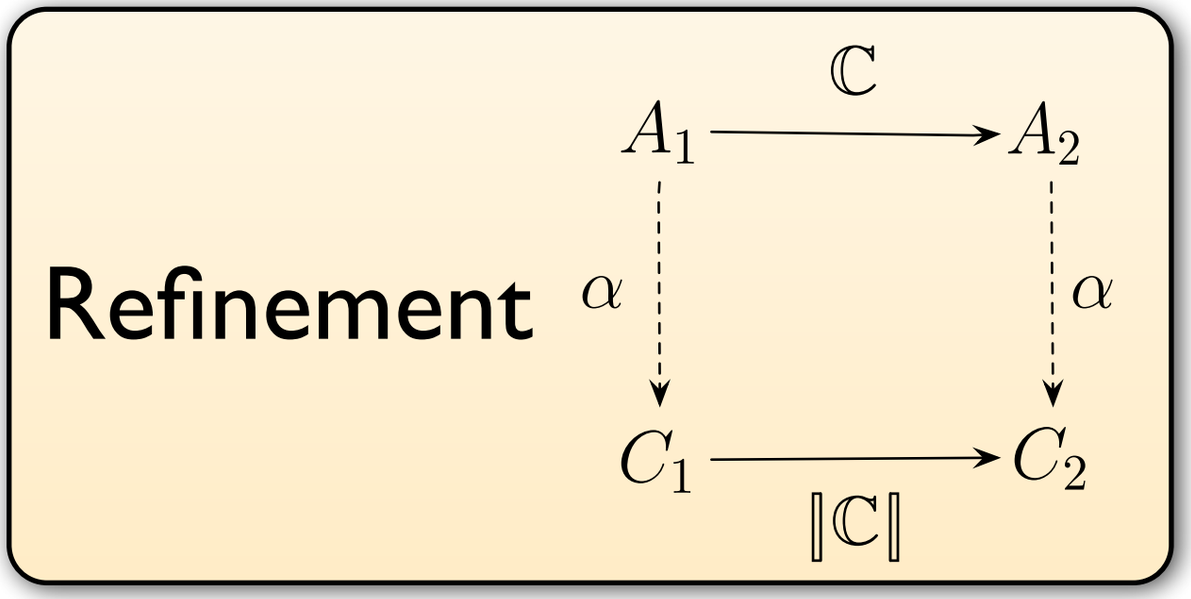
Mark Wheelhouse

Adam Wright

Imperial College London

Abstract Tree Module
(High level)

- Abstract data
  - e.g. DOM Trees
- High level commands

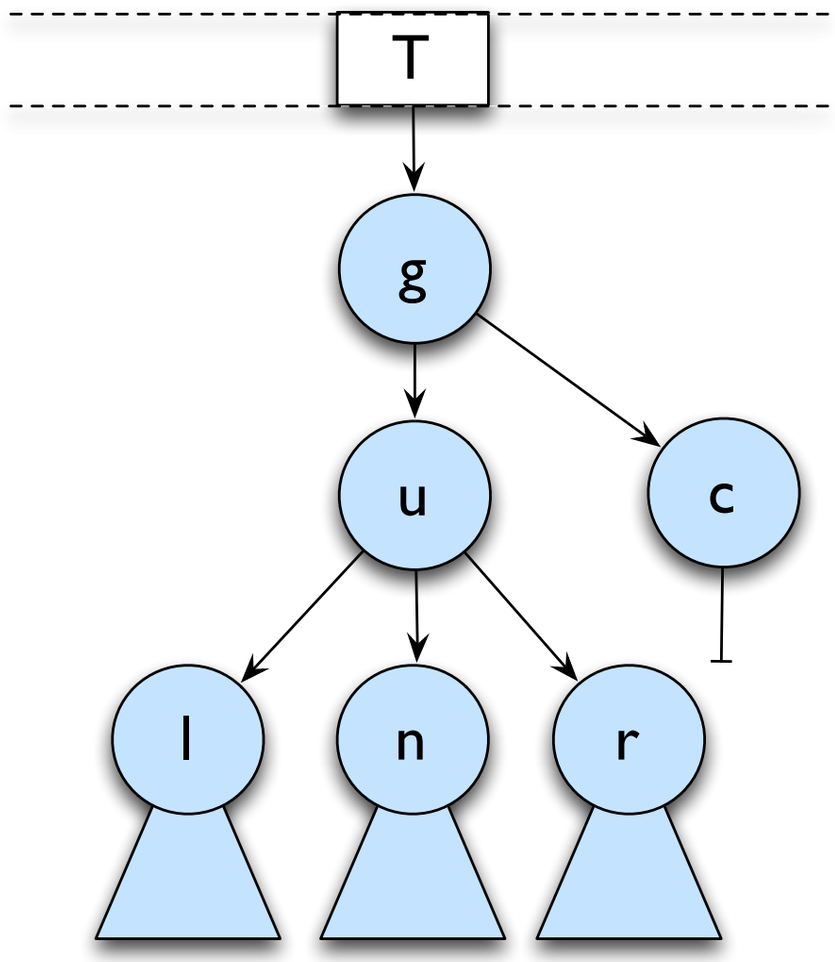


Concrete Tree Module
(Low level)

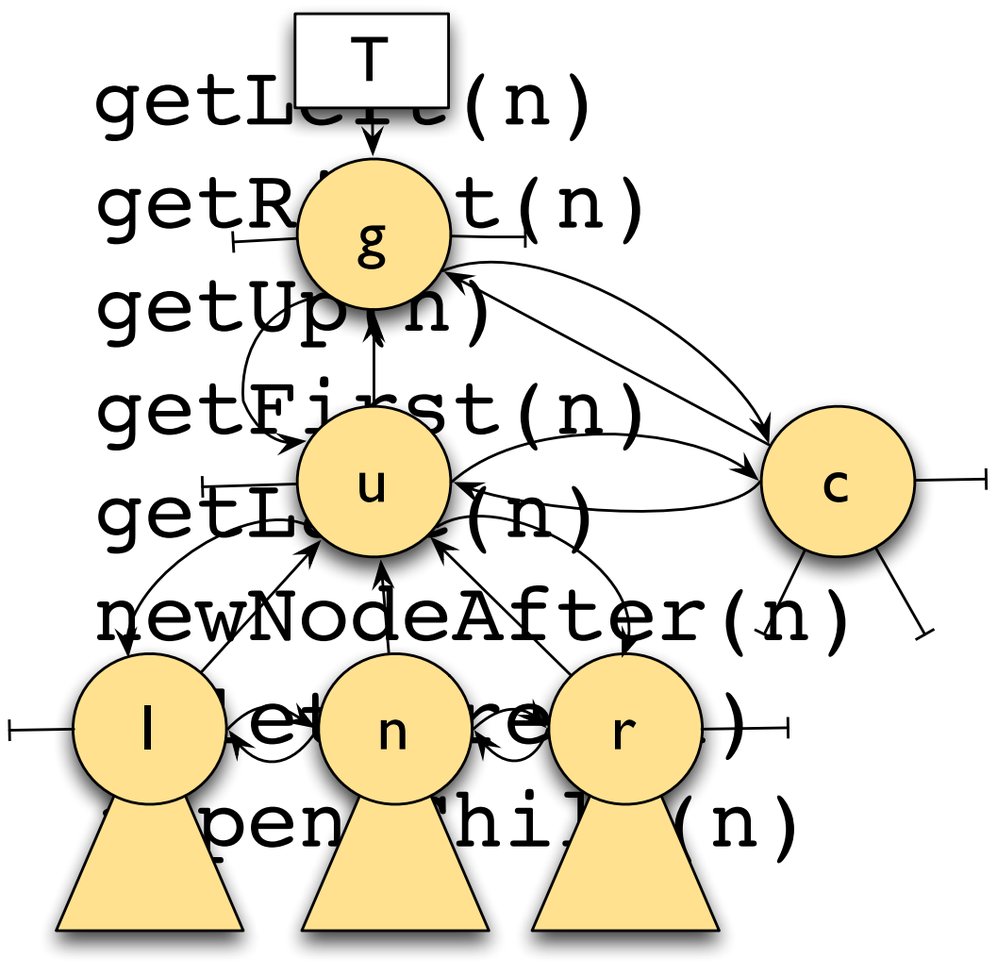
- Concrete data
  - e.g WebKit's DOM
- Implementation

High Level Trees

Abstract Tree Representation

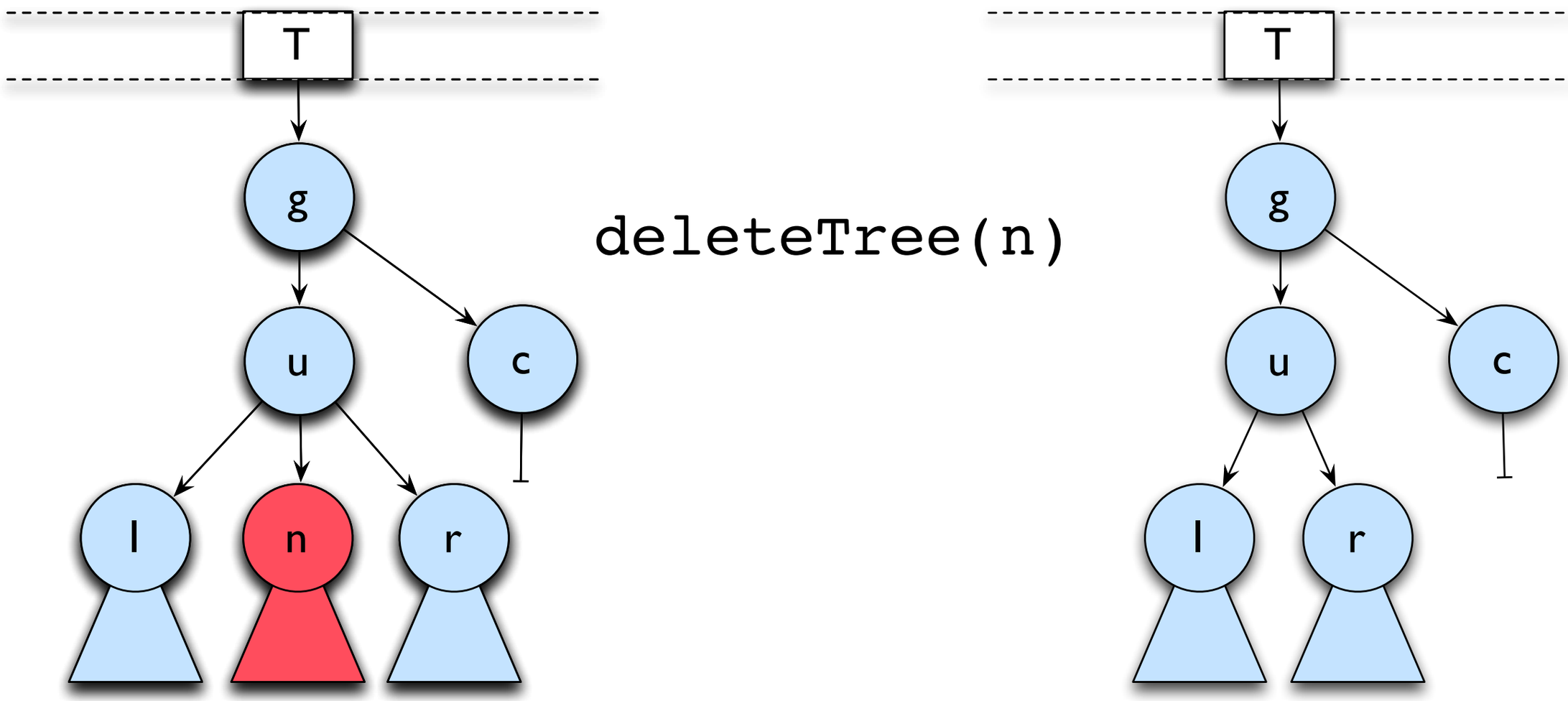


Tree Manipulation Commands



Unique identifiers to locate data

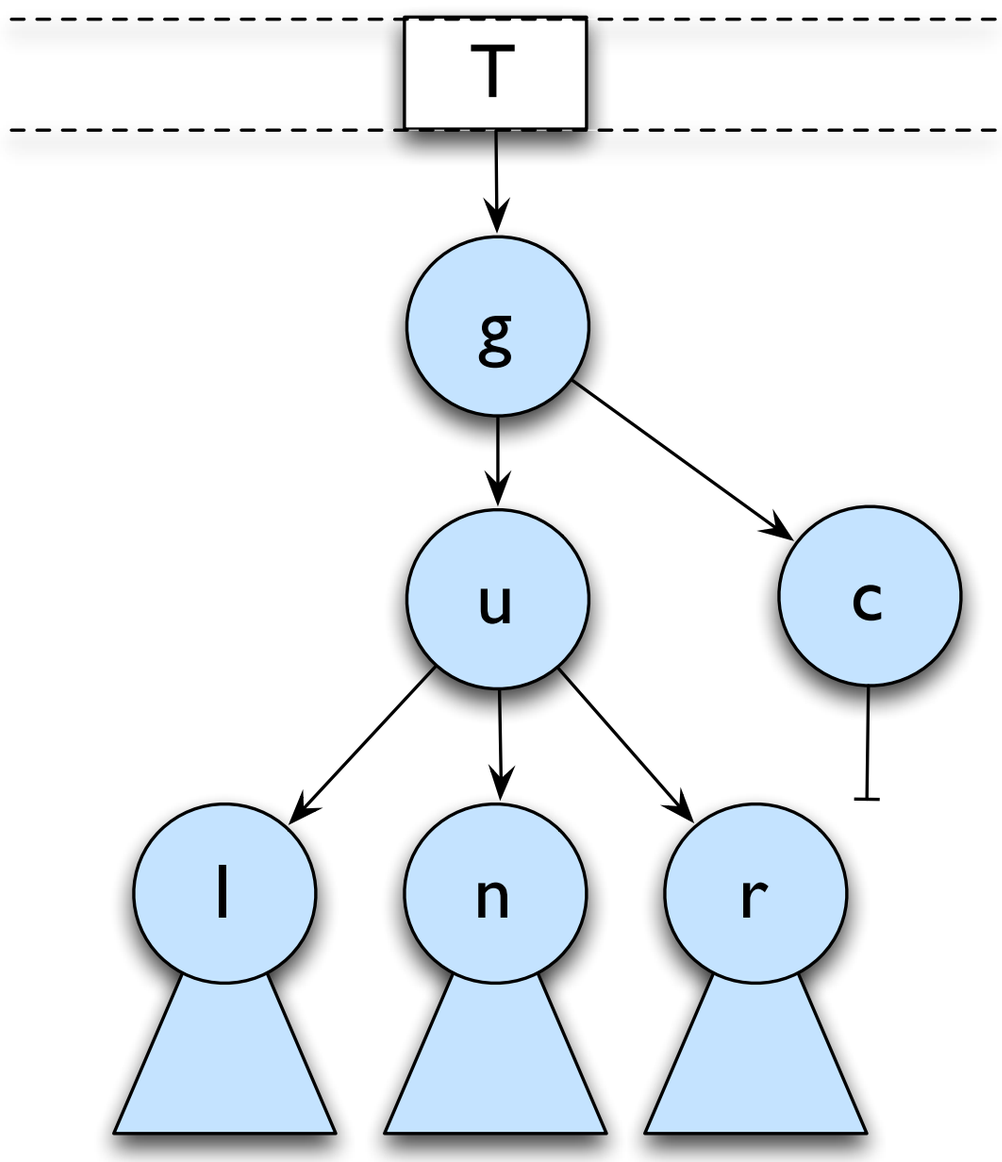
High Level Trees



Unnecessarily large footprint!

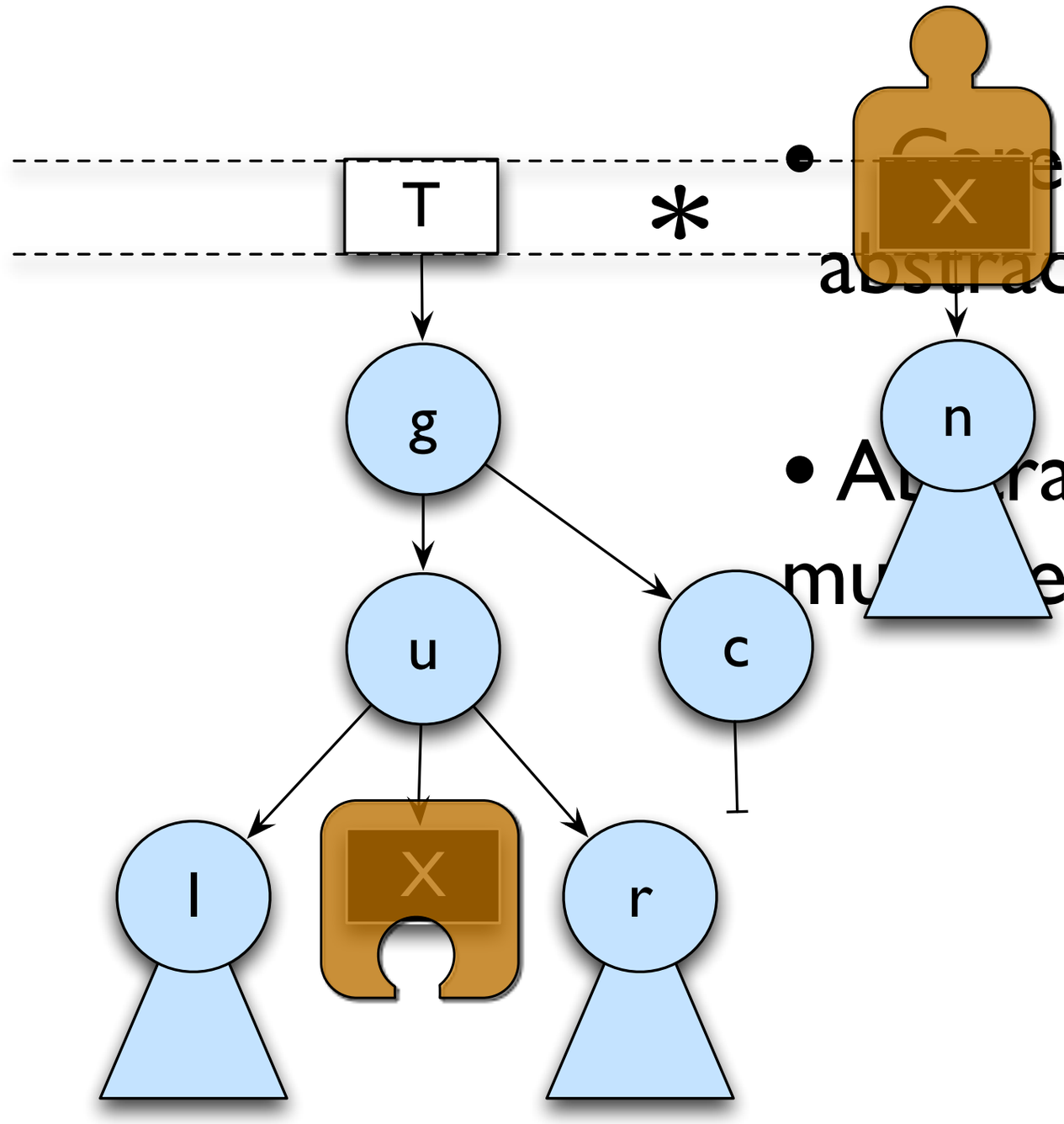
High Level Trees

Structural Separation Logic to the rescue!



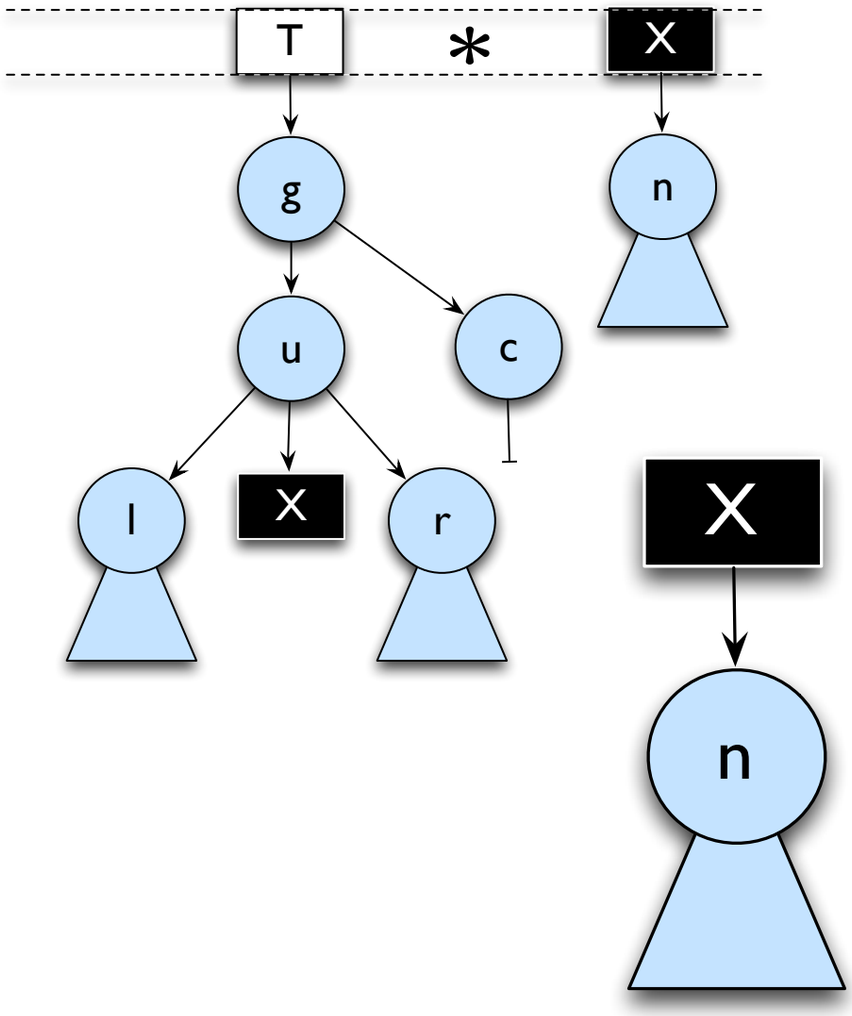
High Level Trees

Structural Separation Logic to the rescue!

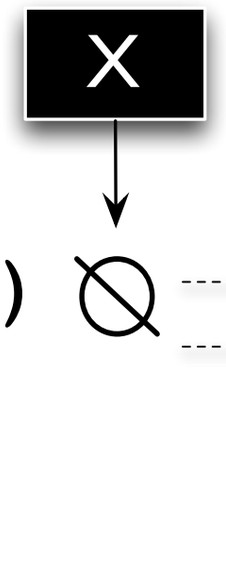


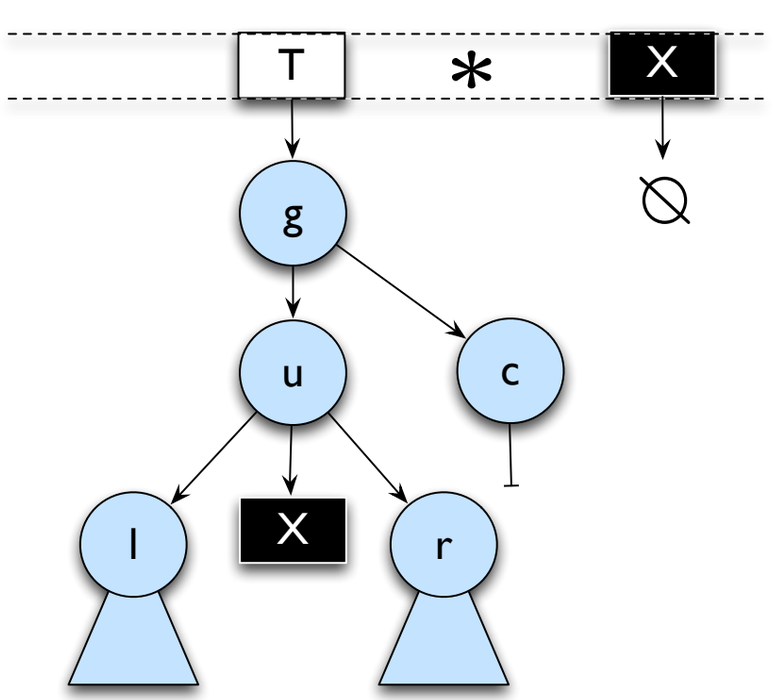
- Careful choice of abstract address set
- Abstract addresses must be preserved

High Level Trees

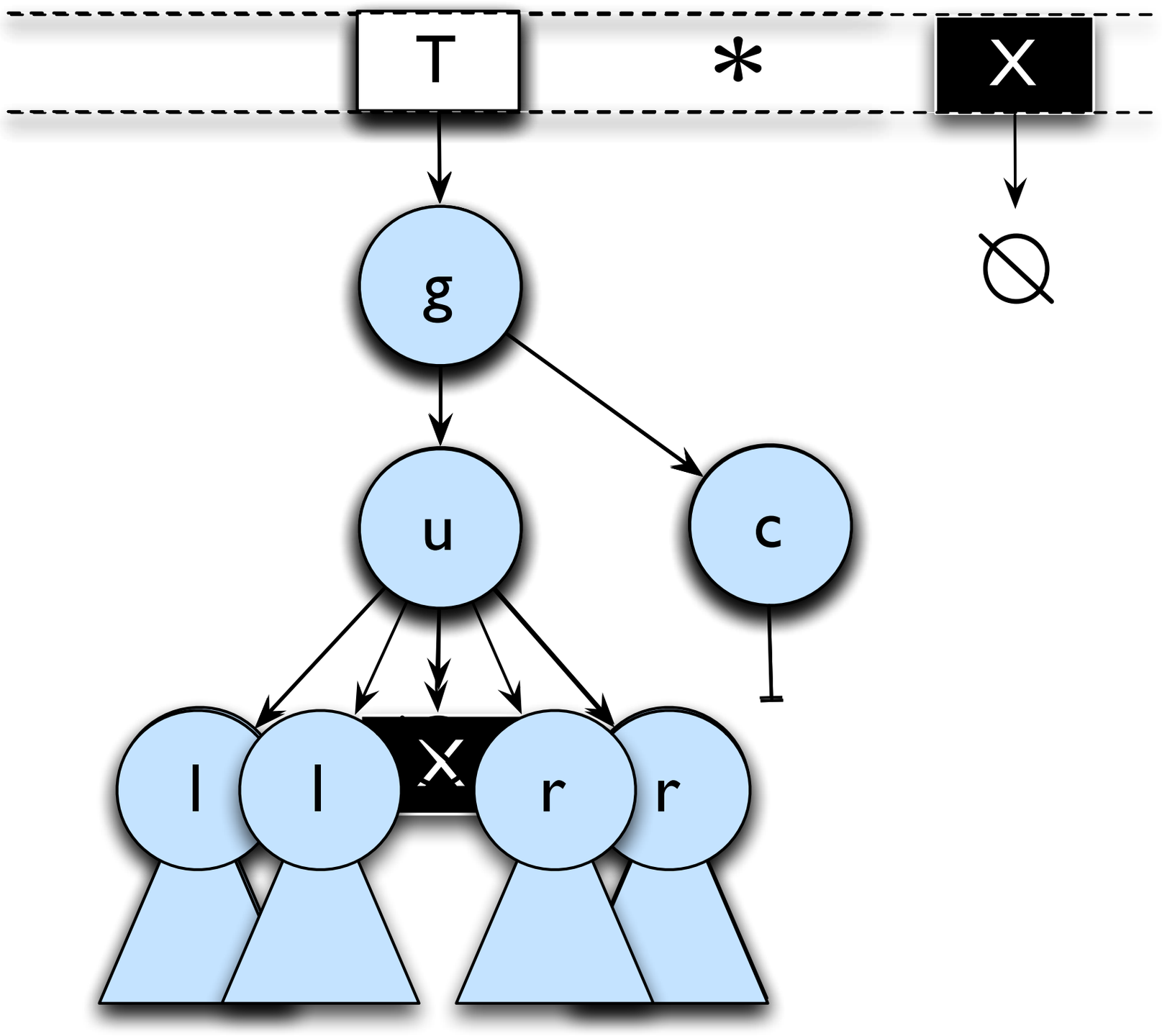



```
deleteTree(n)
```





High Level Trees

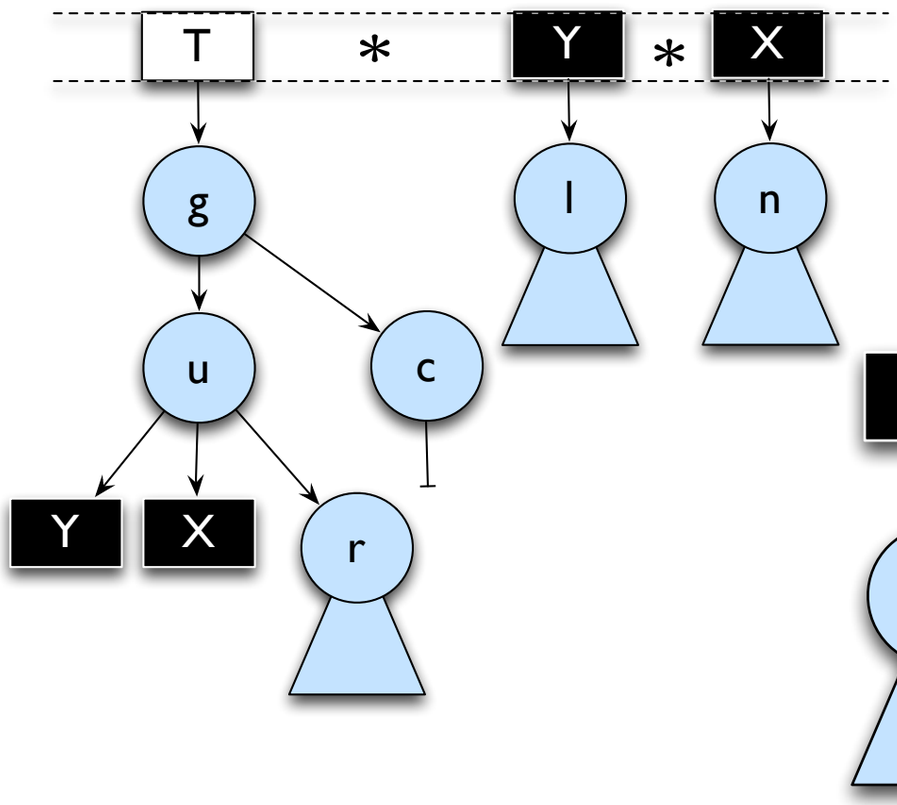


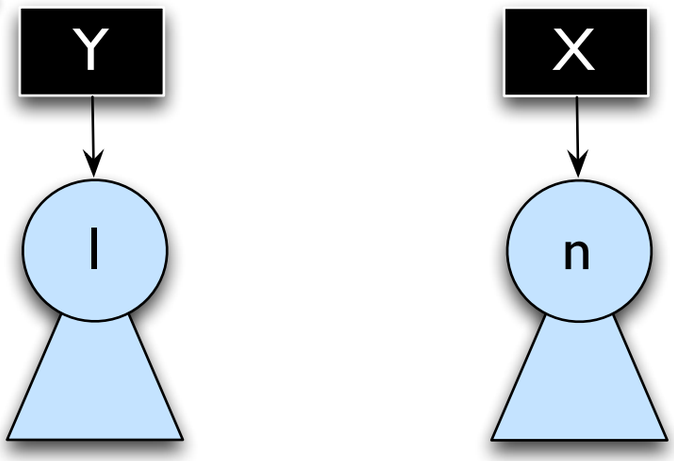
High Level Trees

Separation gives us disjoint concurrency!

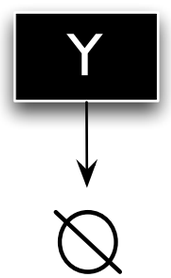
$$\frac{\{\, P_1 \,\}\ C_1\ \{\, Q_1 \,\} \quad \{\, P_2 \,\}\ C_2\ \{\, Q_2 \,\}}{\{\, P_1 * P_2 \,\}\ C_1 || C_2\ \{\, Q_1 * Q_2 \,\}}$$

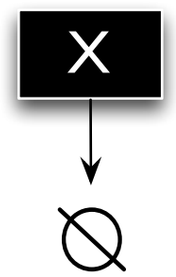
High Level Trees

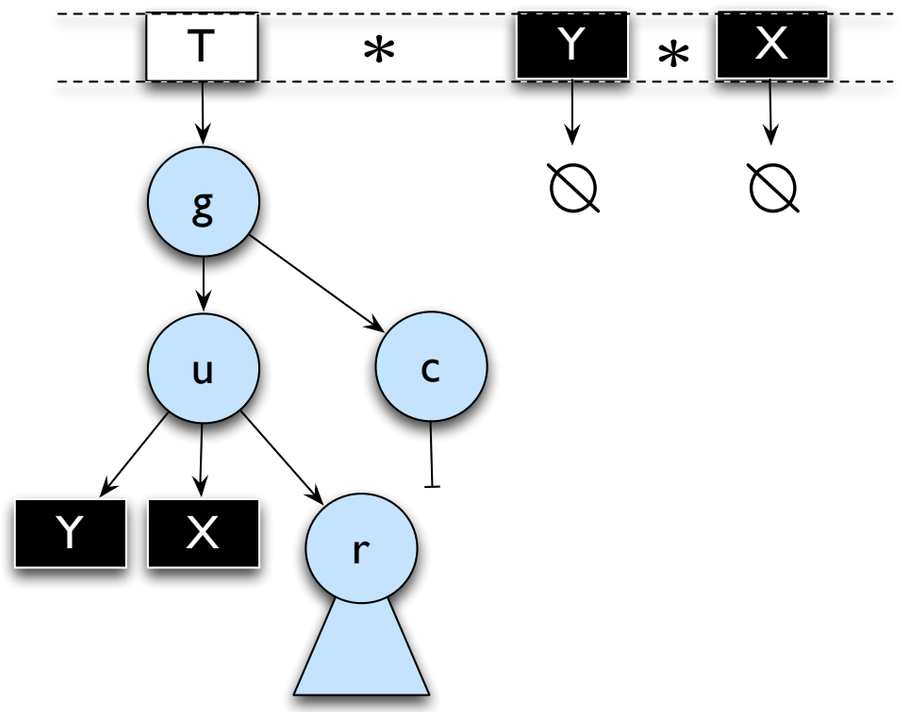




`deleteTree(l)` || `deleteTree(n)`

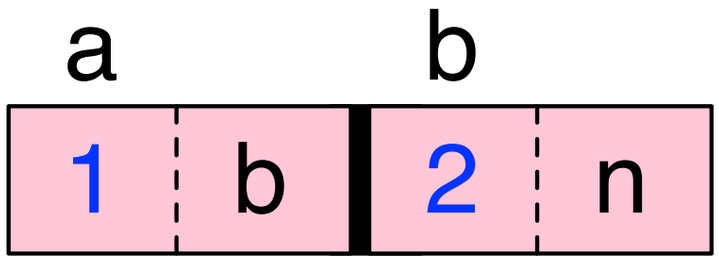






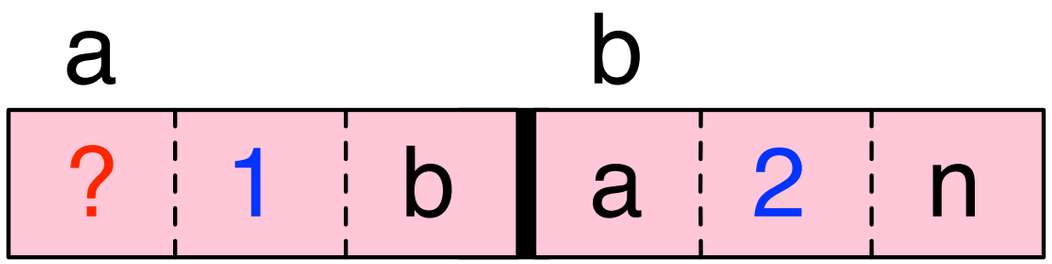
Abstract Predicates

listSeg(1⊗2, a, n)



(Singly-Linked List)

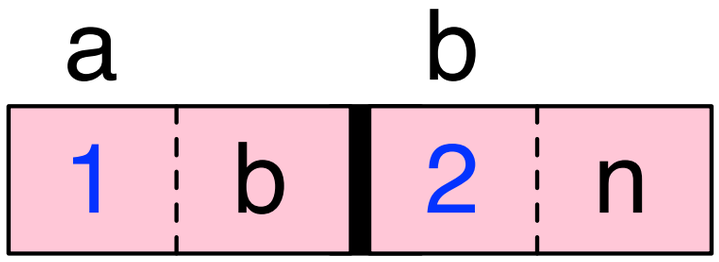
listSeg(1⊗2, a, n, p)



(Doubly-Linked List)

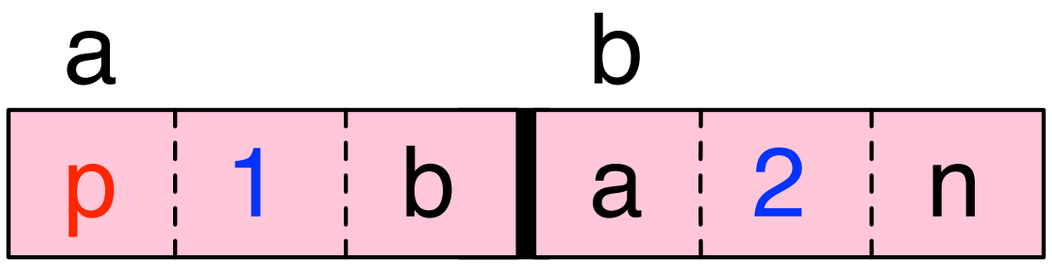
Abstract Predicates

listSeg(1⊗2, a, n)



(Singly-Linked List)

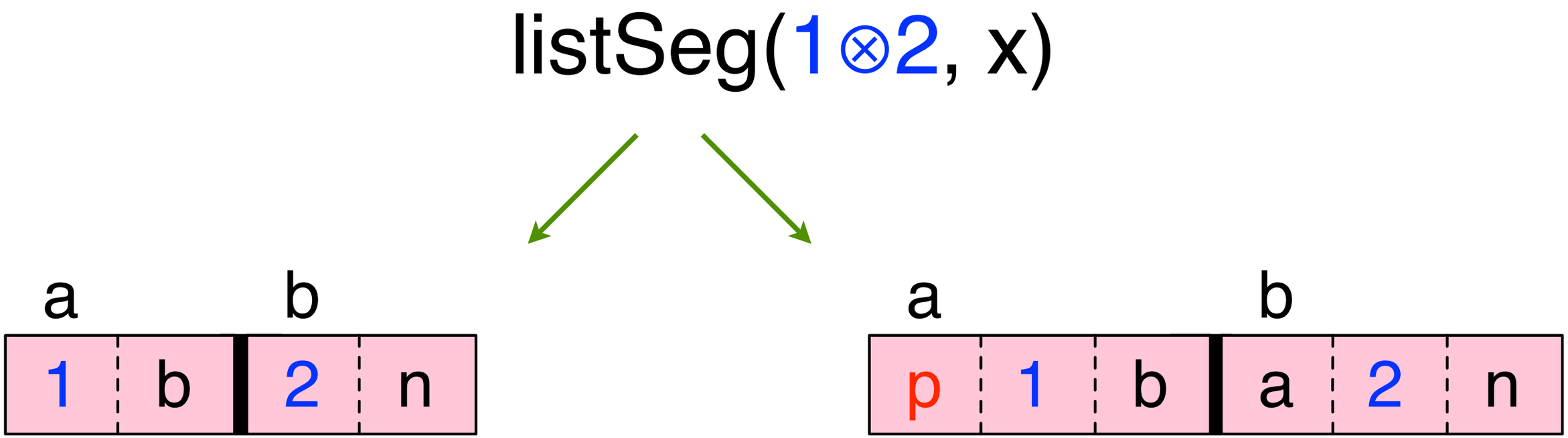
listSeg(1⊗2, a, n, p)



(Doubly-Linked List)

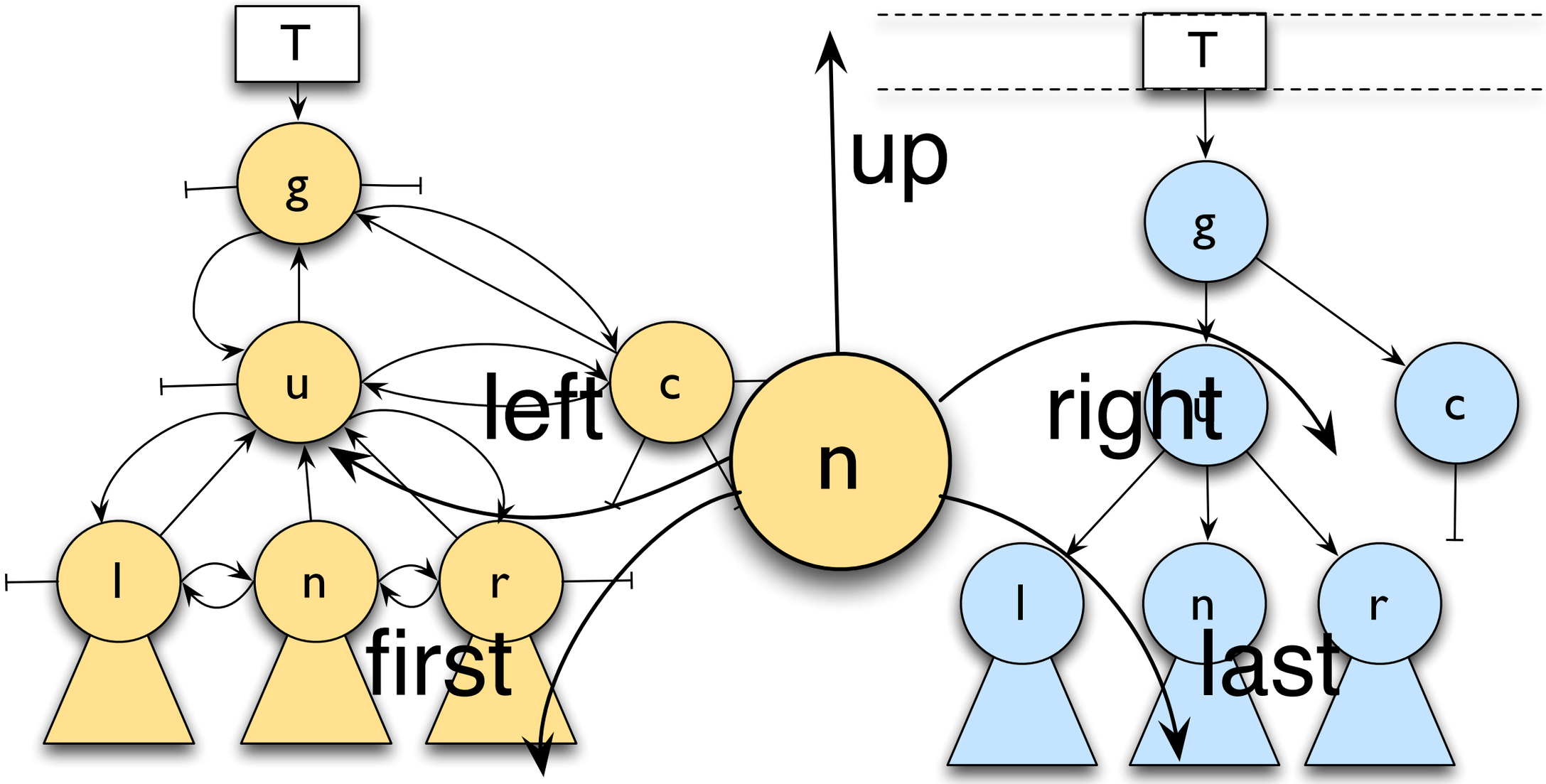
Leaks Implementation details into abstraction!

SSL: Abstract Connectivity



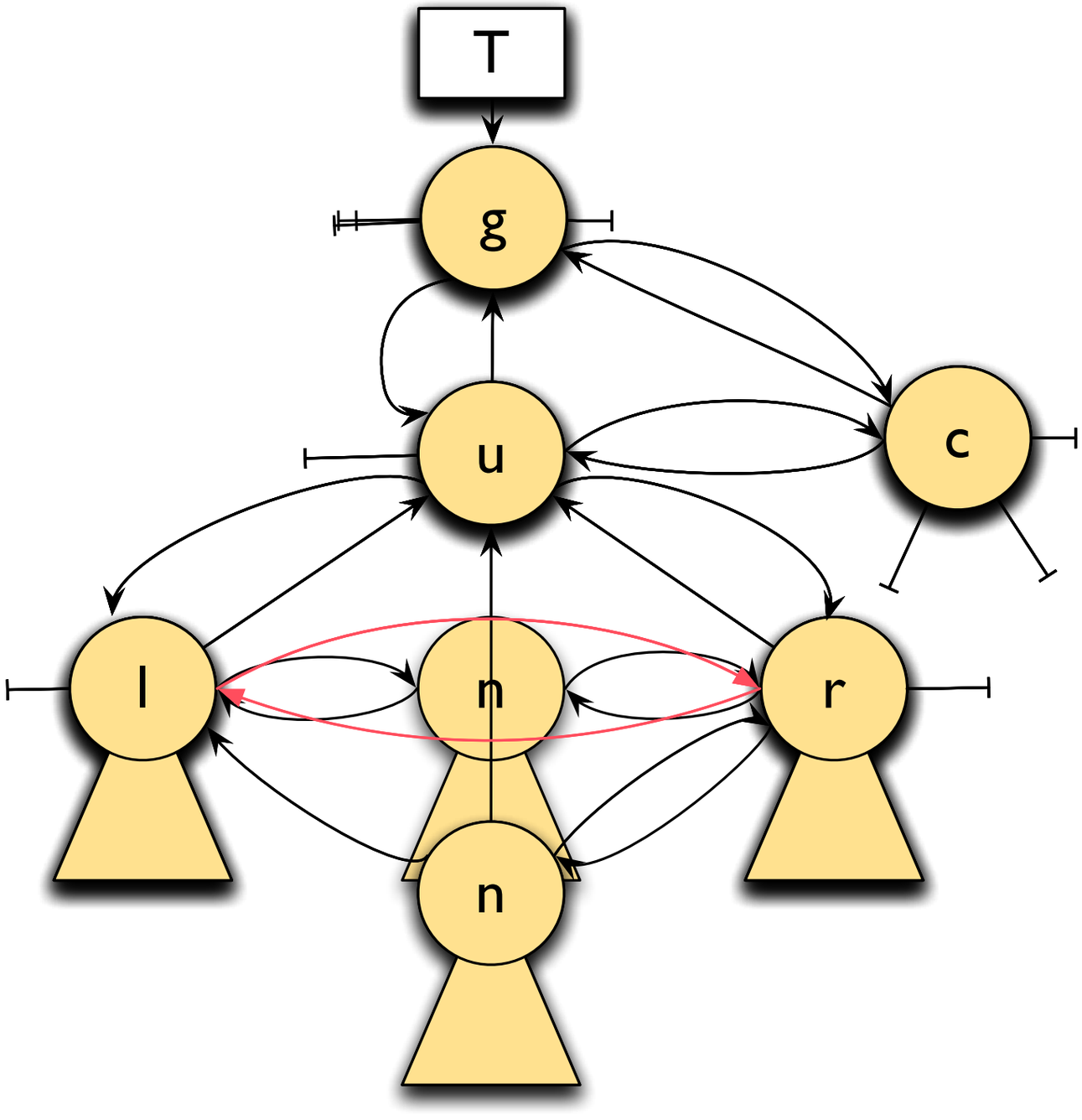
Low Level Trees

Concrete Tree Representation



Low Level Trees

`deleteTree` Procedure

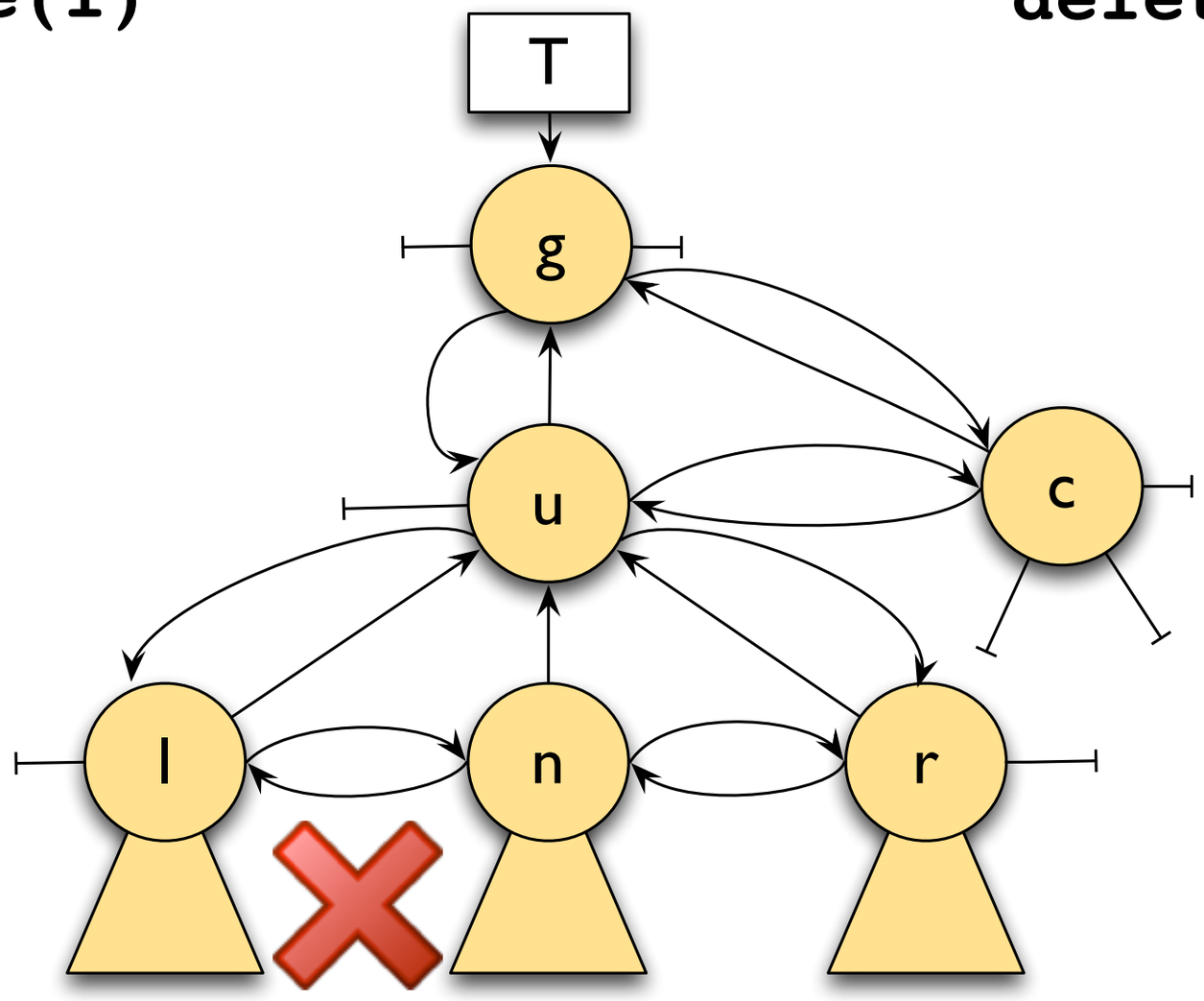


Low Level Trees

Concurrent `deleteTree` Procedure

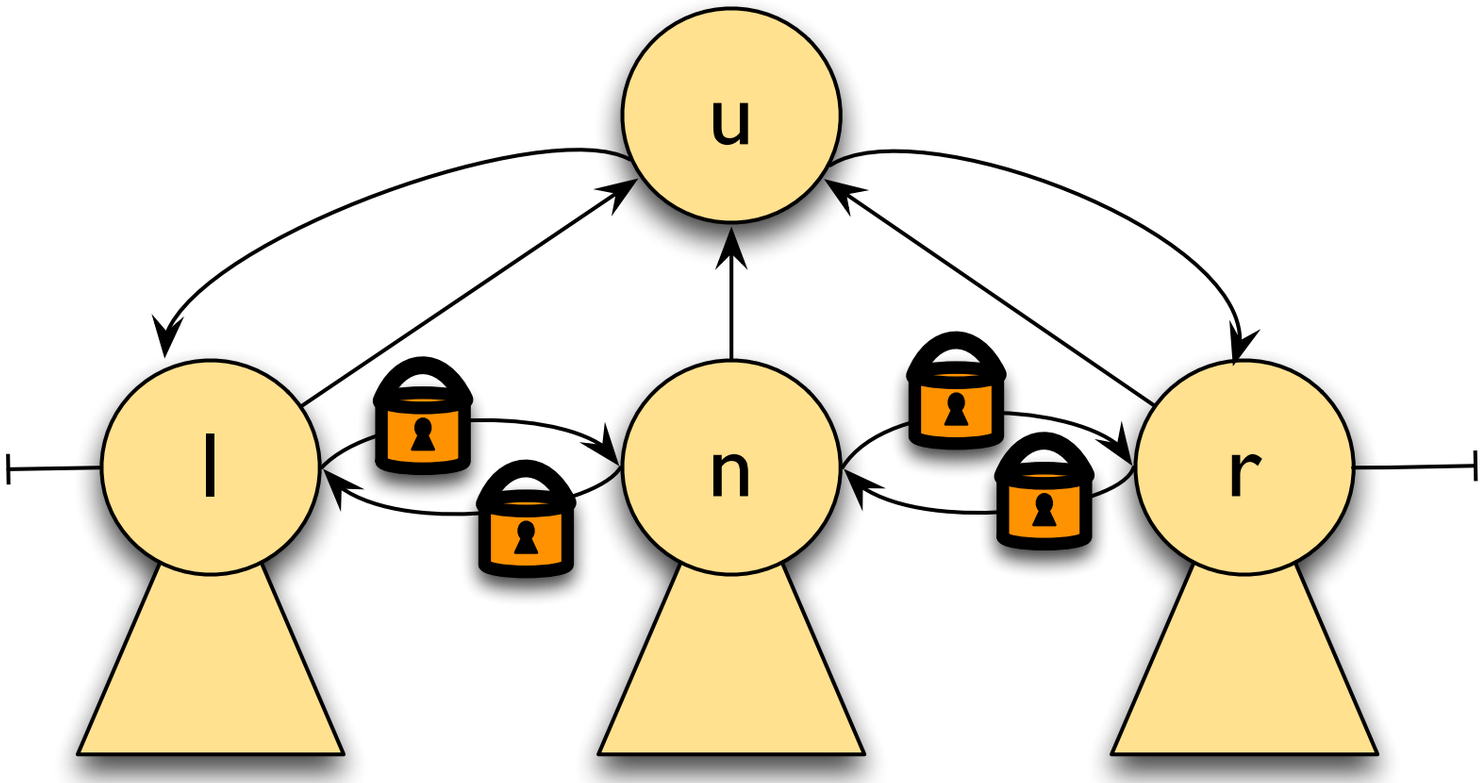
`deleteTree(l)`

`deleteTree(n)`



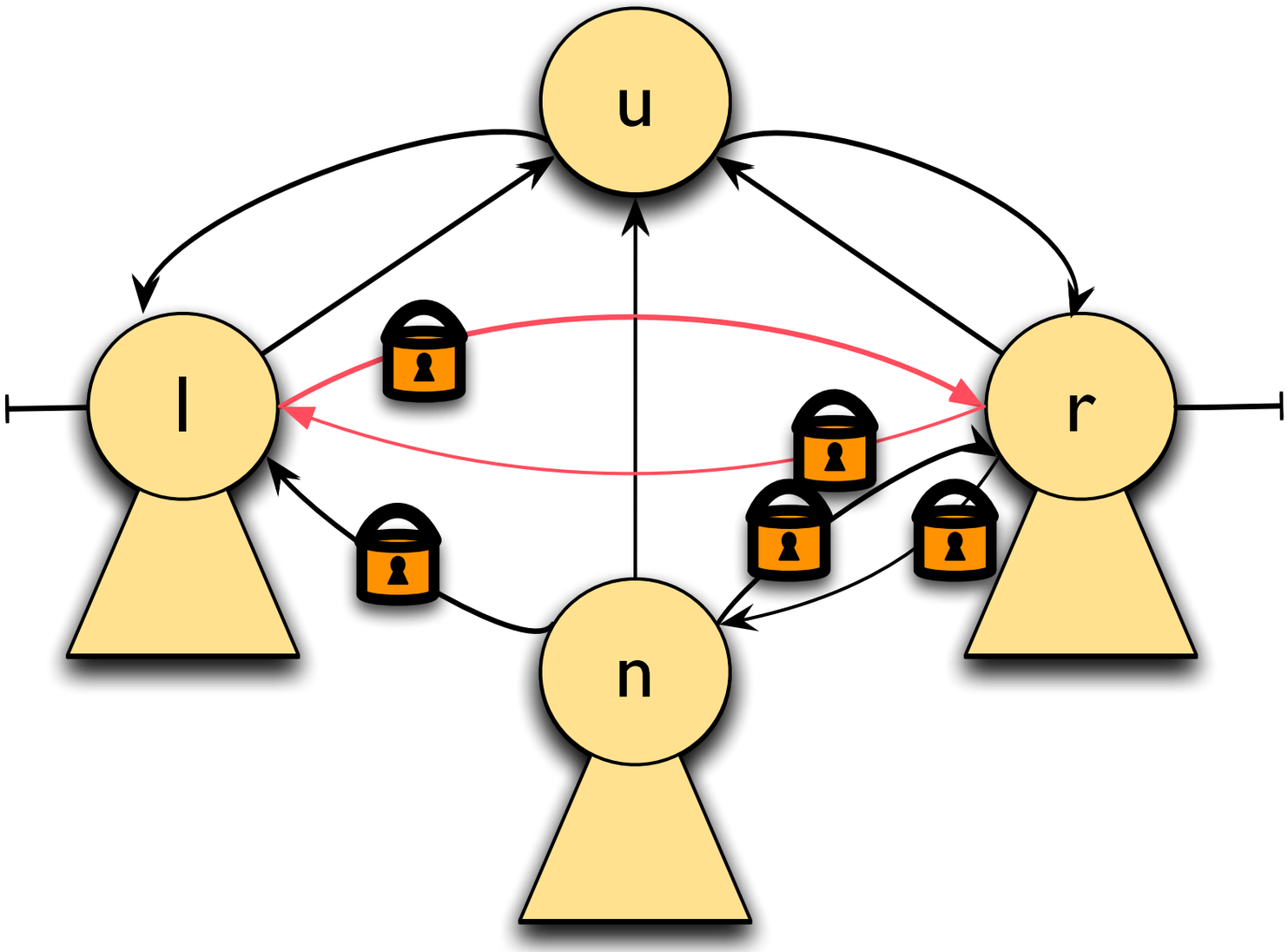
Low Level Trees

Concurrent `deleteTree` Procedure



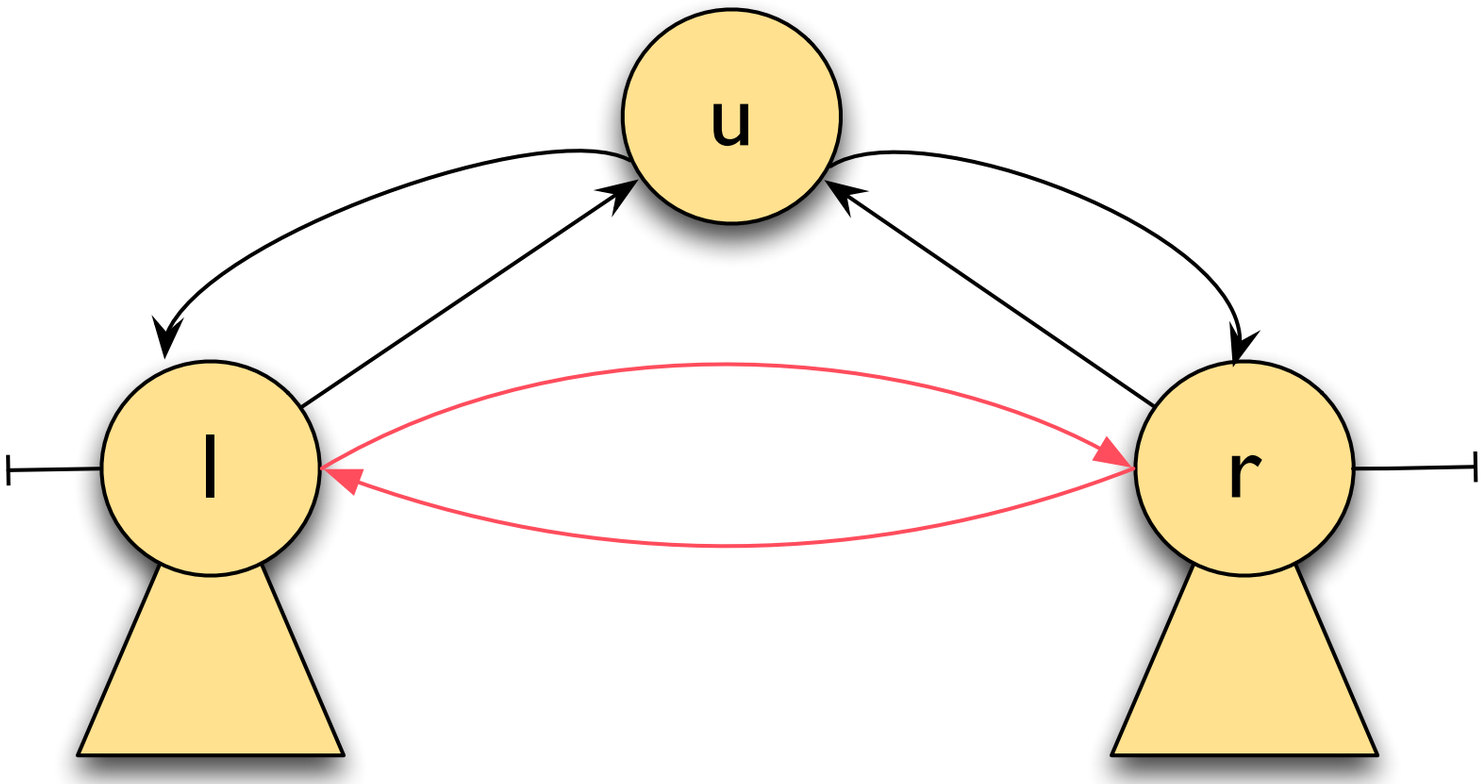
Low Level Trees

Concurrent `deleteTree` Procedure



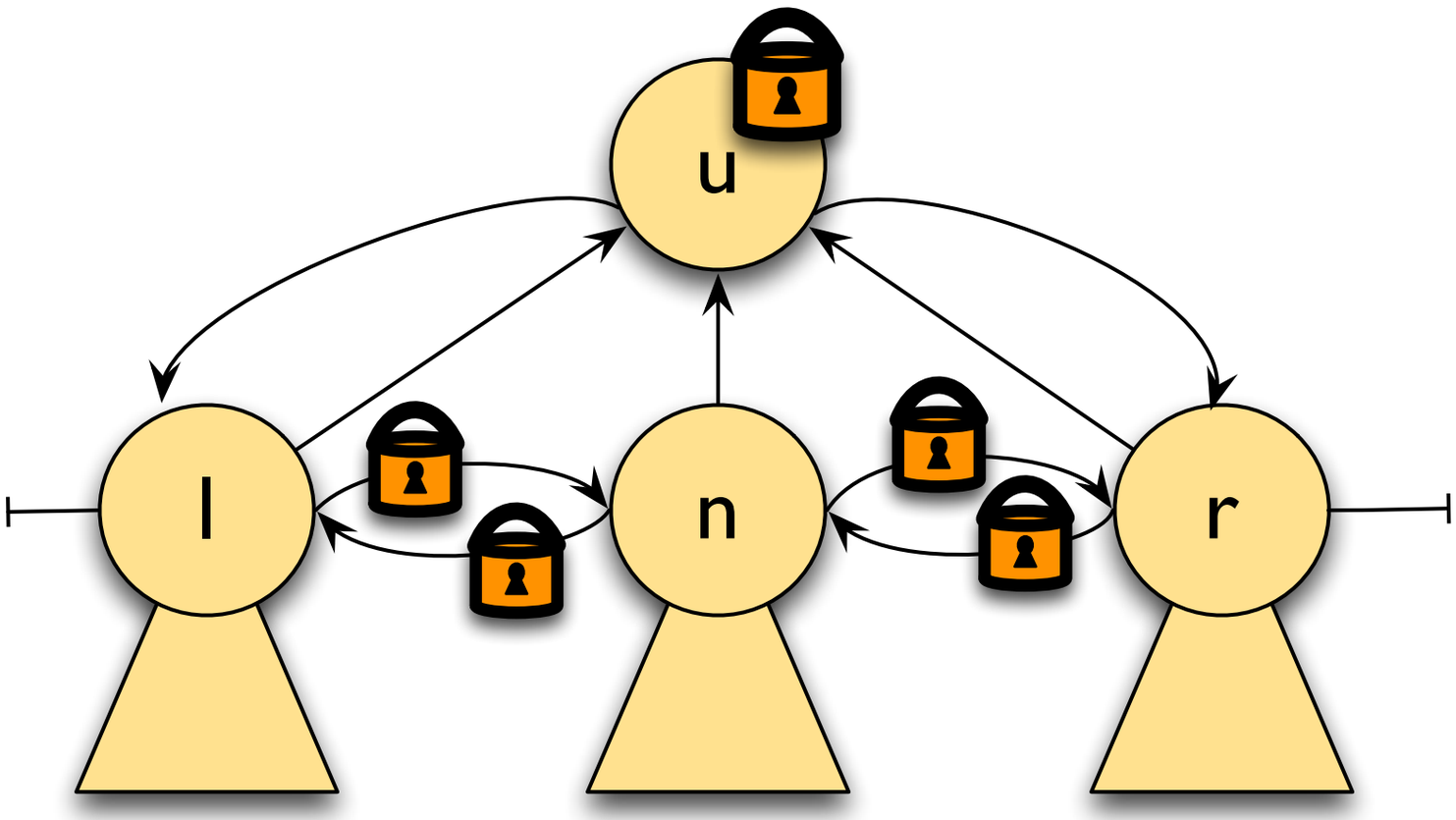
Low Level Trees

Concurrent `deleteTree` Procedure



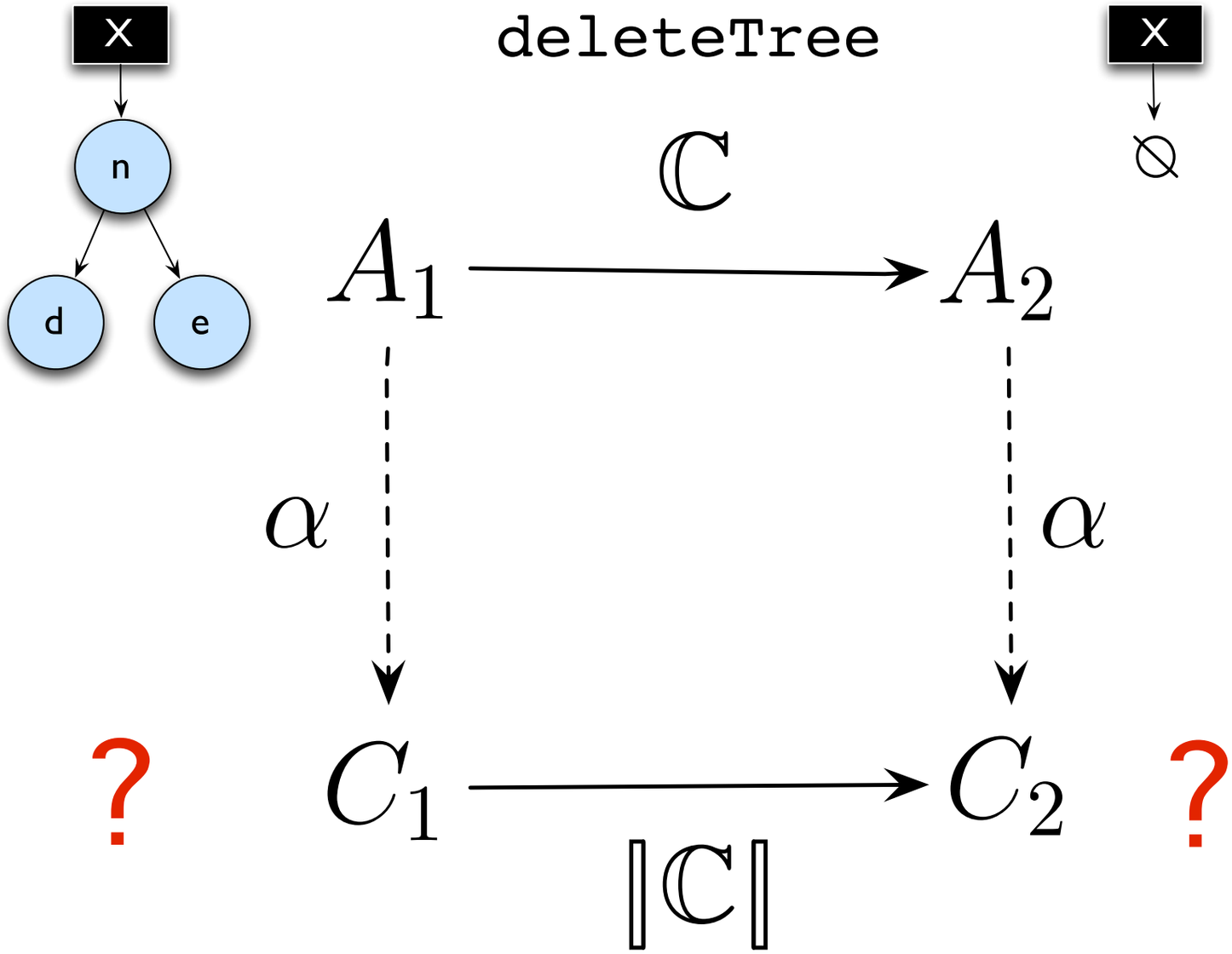
Low Level Trees

Concurrent `deleteTree` Procedure



Low Level Trees

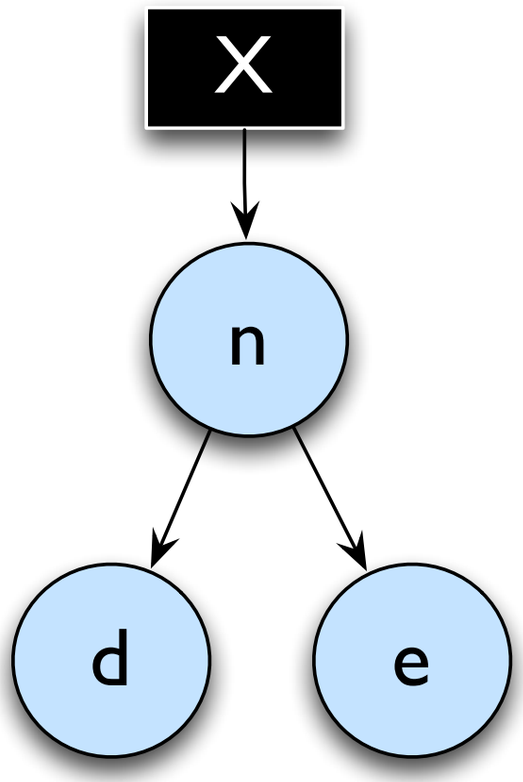
Concrete tree representation for partial trees

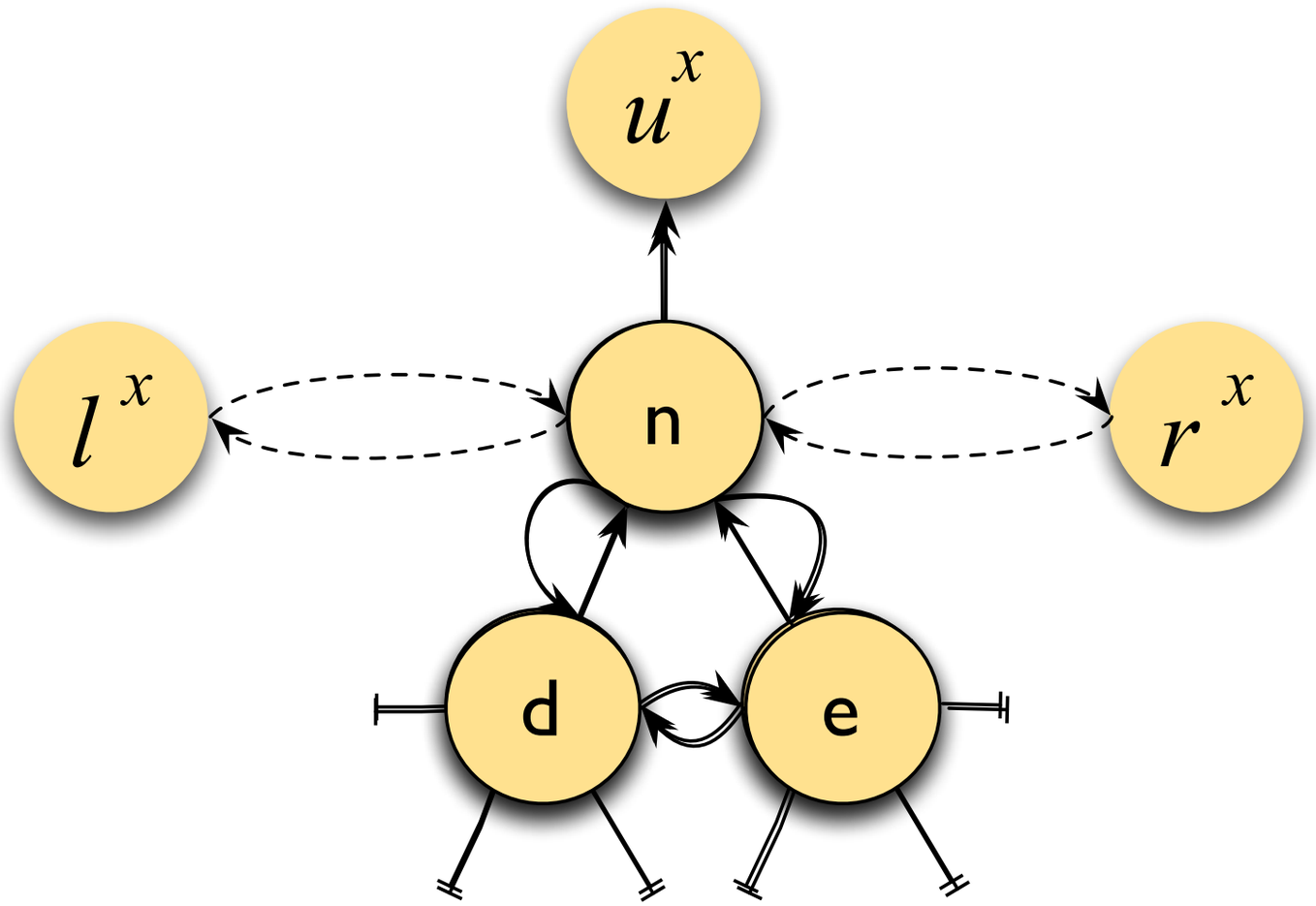


(Our locking implementation)

Low Level Trees

Concrete tree representation for partial trees



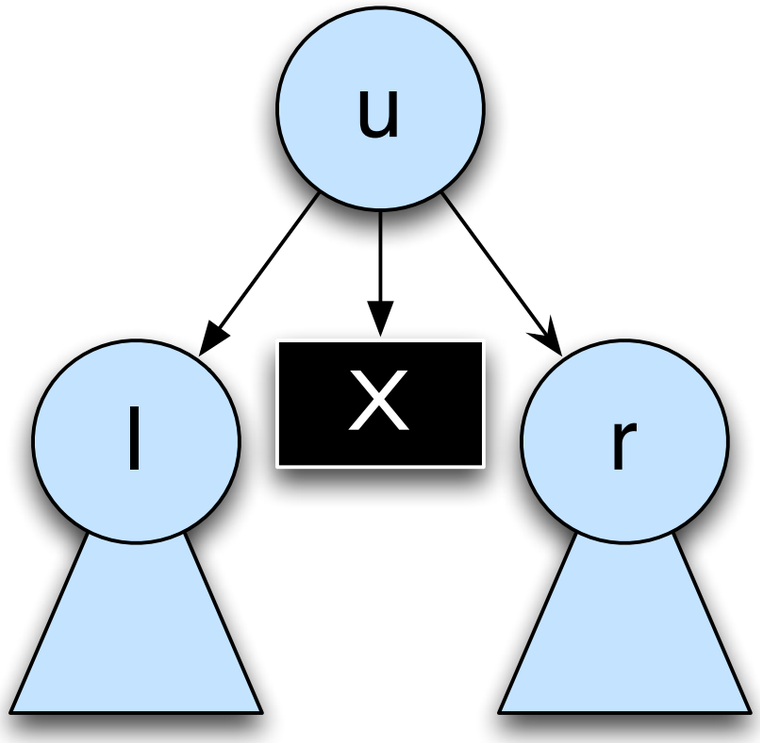


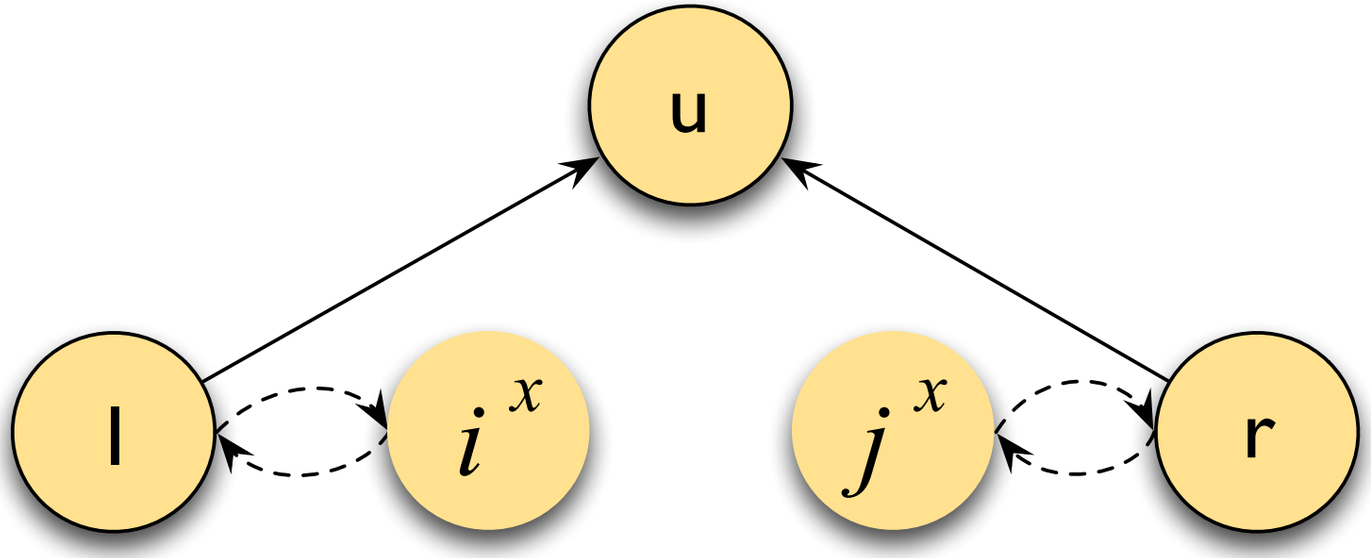
Abstract address: x

$$I(\mathsf{X}) = (n, n)(l^x, u^x, r^x)$$

Low Level Trees

Concrete tree representation for partial trees



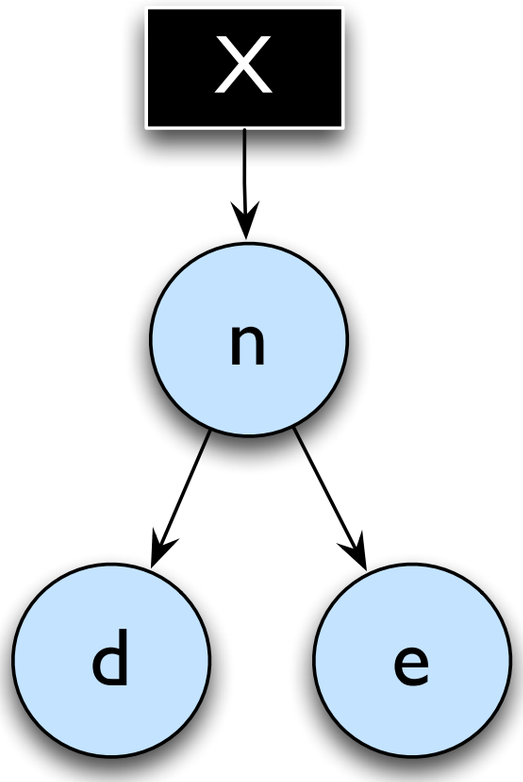


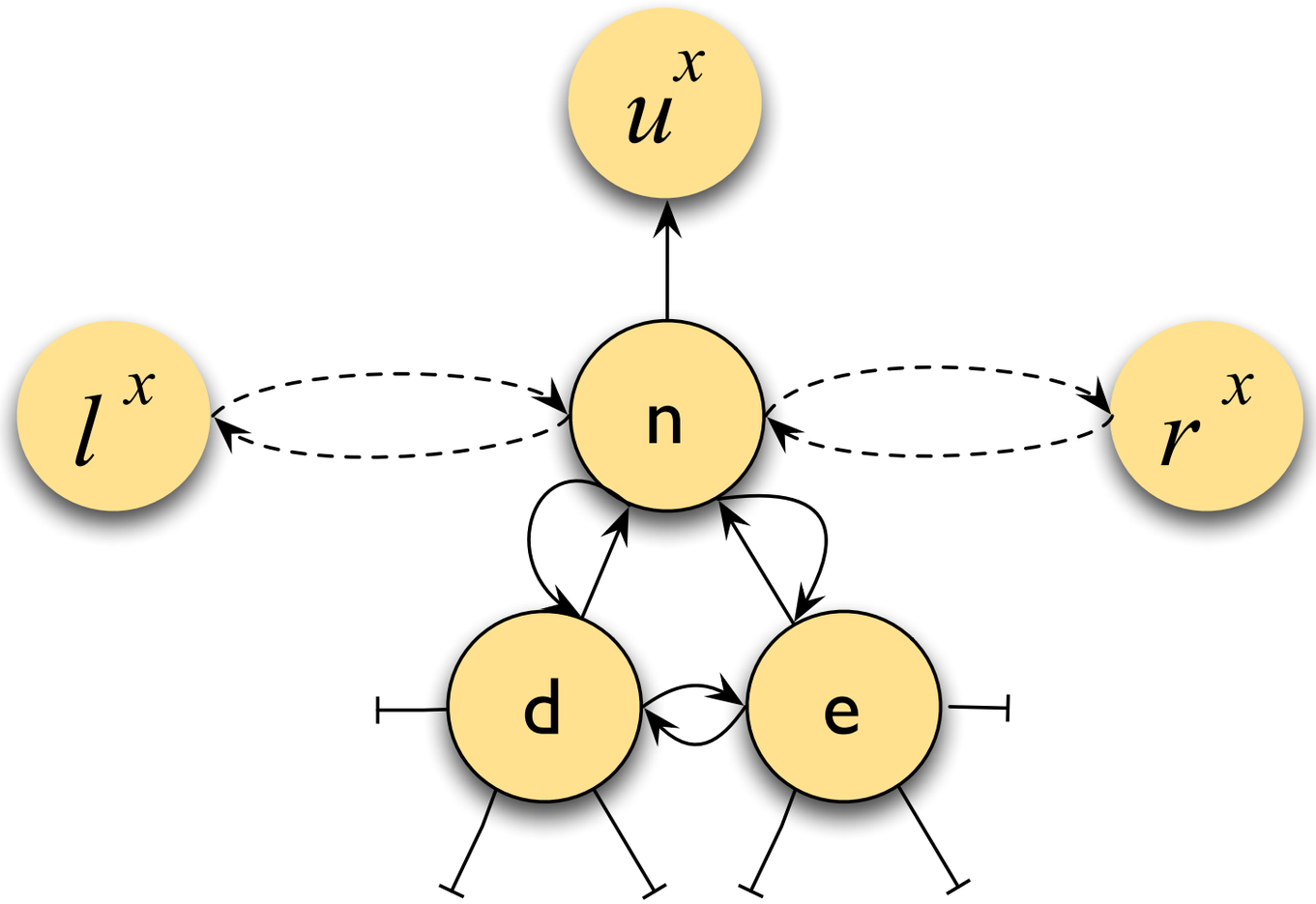
Abstract address: x

$$I(\mathrm{X}) = (i^{x}, j^{x})\ (l, u, r)$$

Low Level Trees

Concrete tree representation for partial trees





Abstract address: x

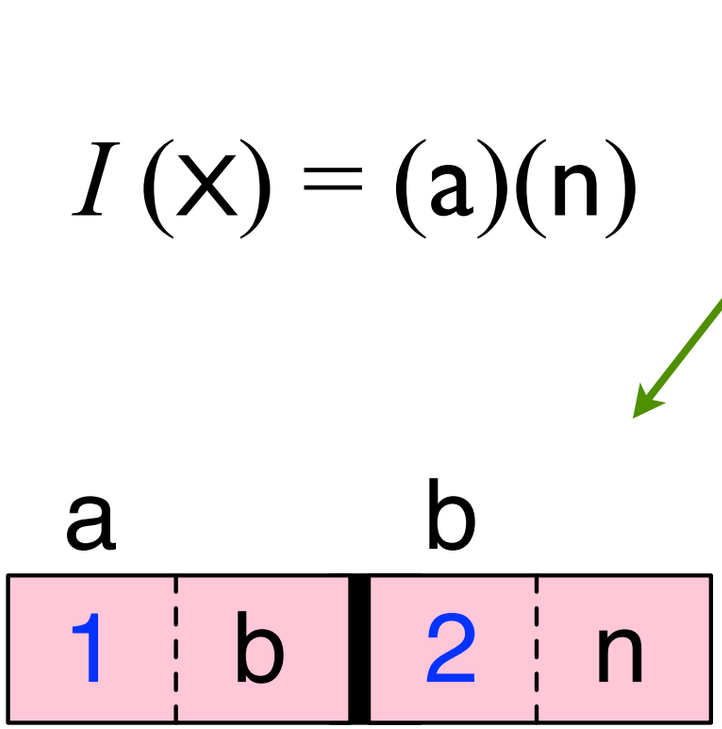
$$I(\mathsf{X}) = (n, n)(l^x, u^x, r^x)$$

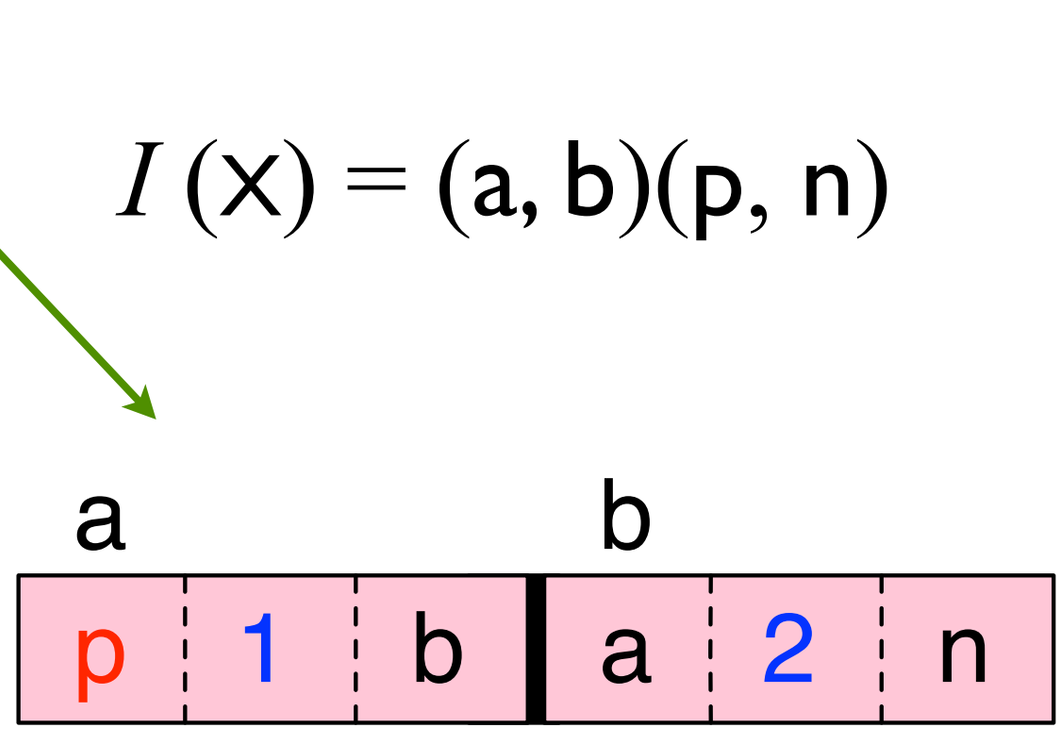
SSL: Abstract Connectivity

listSeg(1⊗2, x)

$I(x) = (a)(n)$

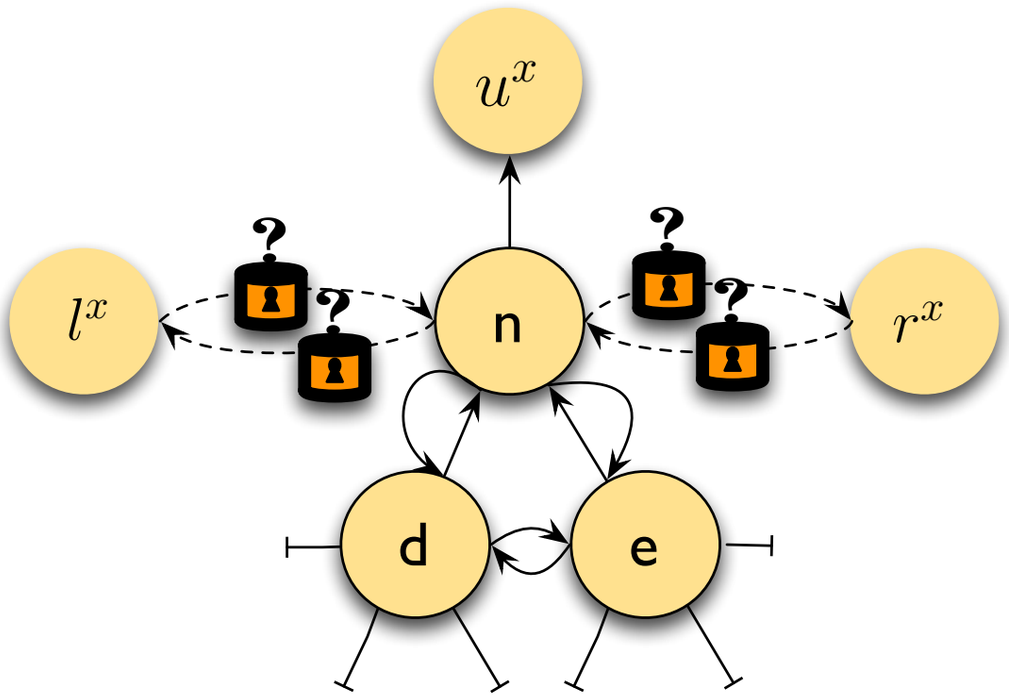
$I(x) = (a, b)(p, n)$

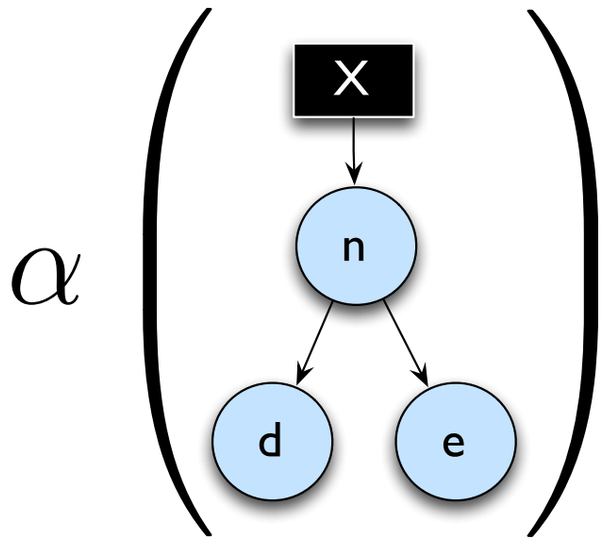




Refinement (Axiomatic Correctness)

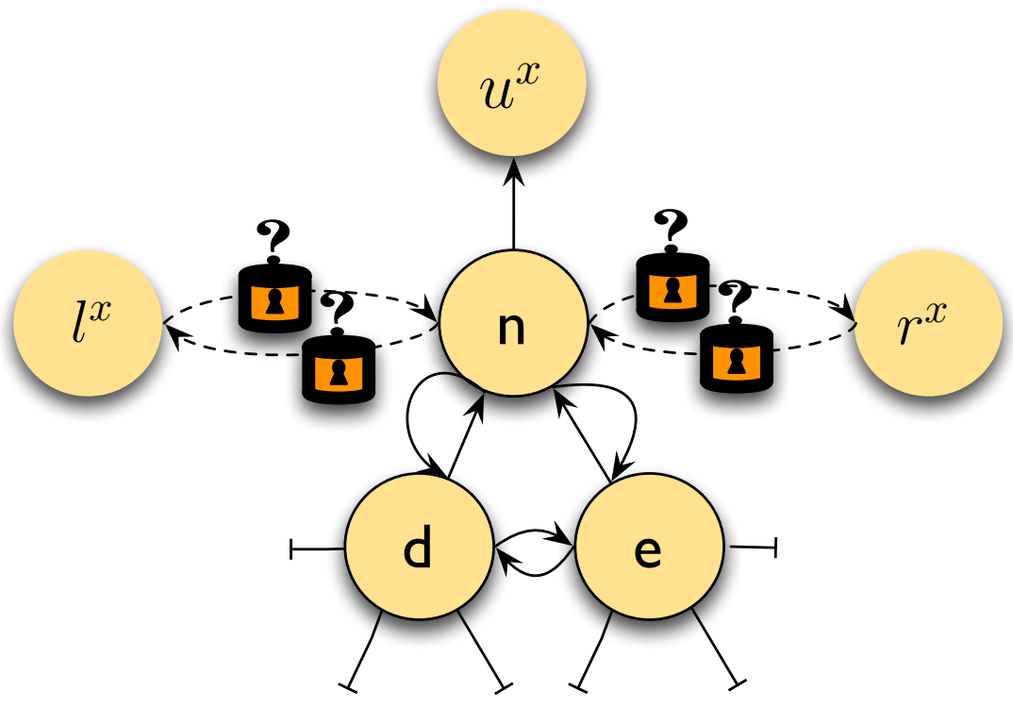
```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```



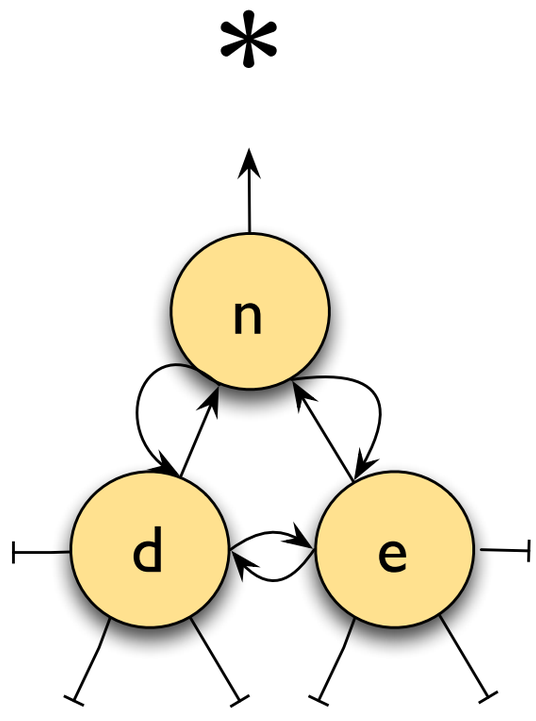


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

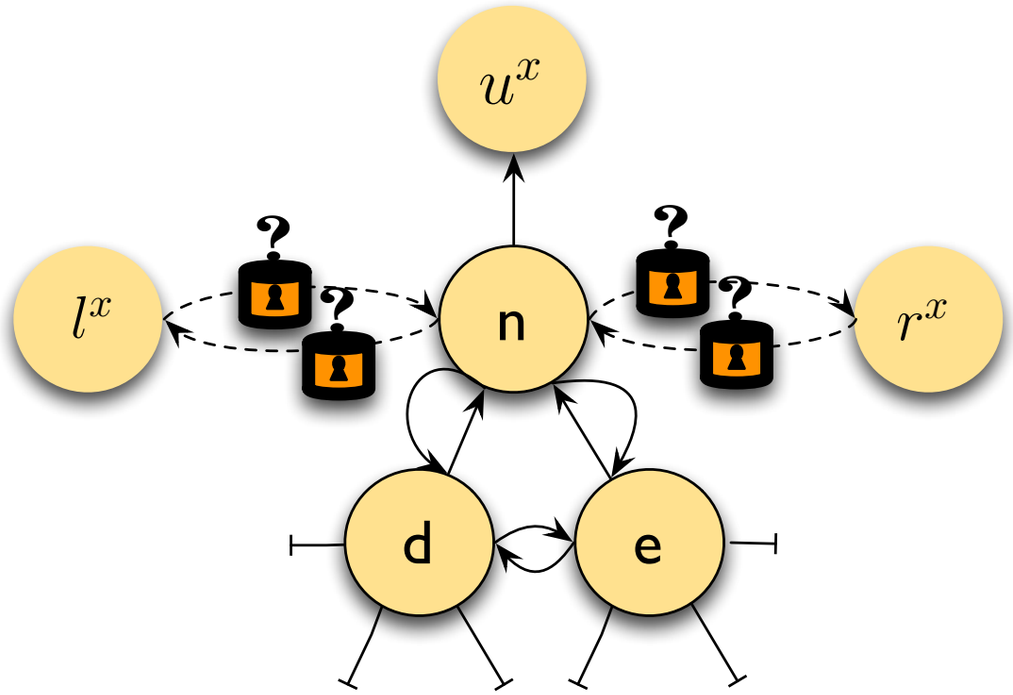


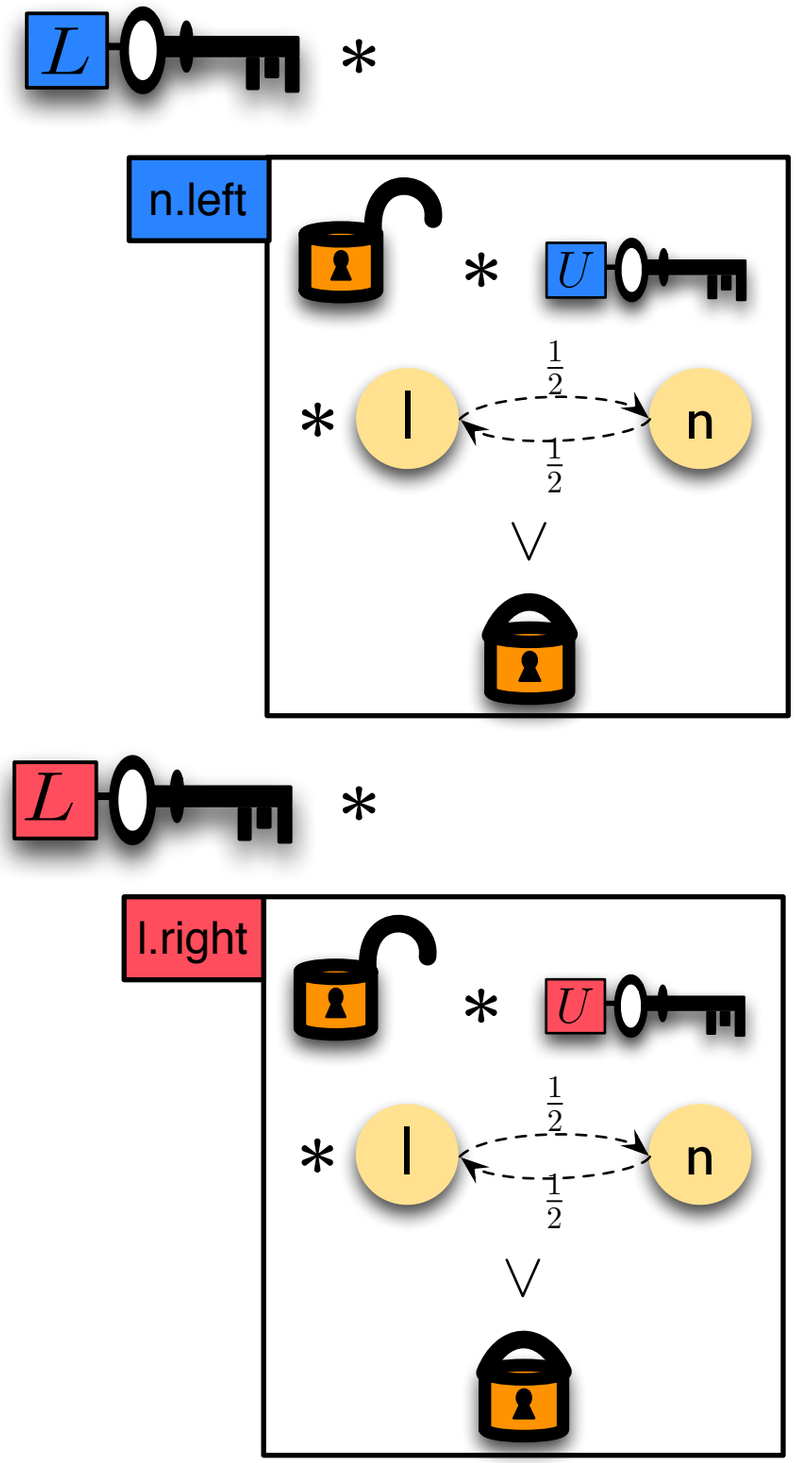
$$\mathsf{Crust}(n, n)(l^x, u^x, r^x)$$



Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

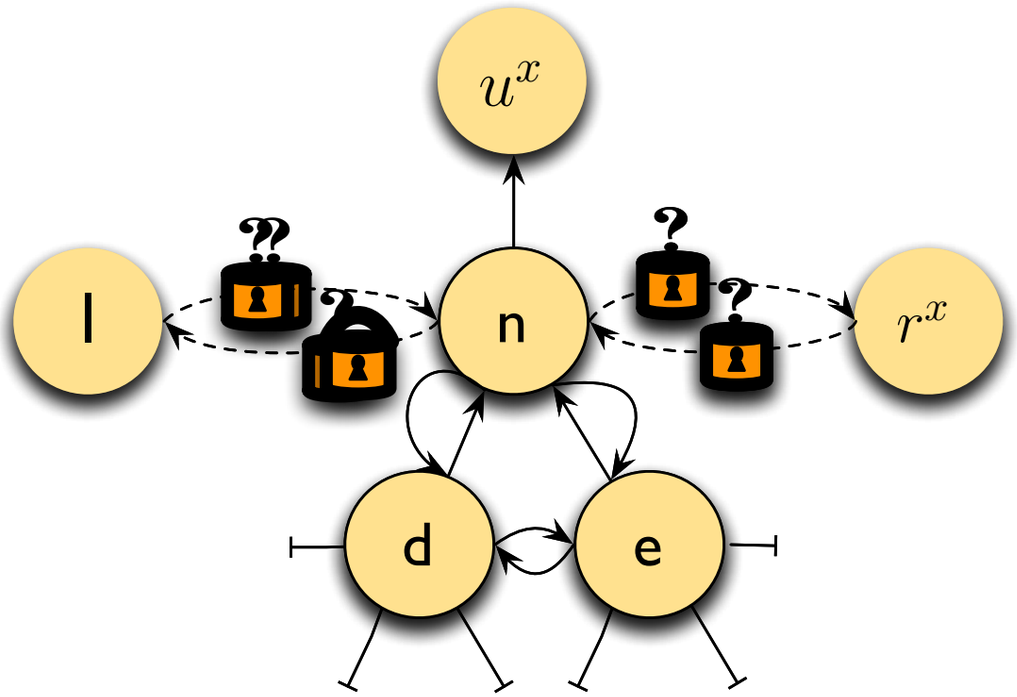


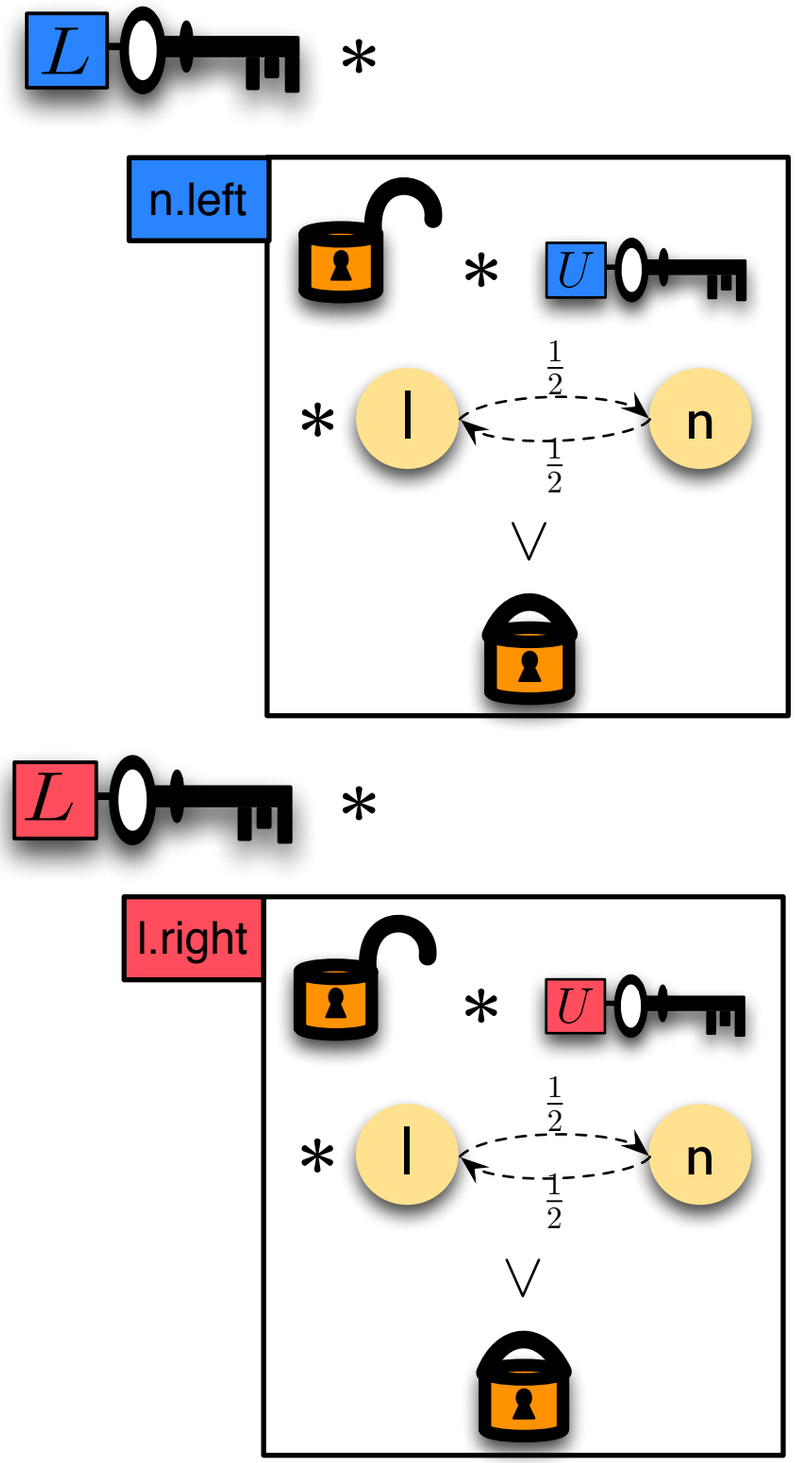


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

lock(n.left);

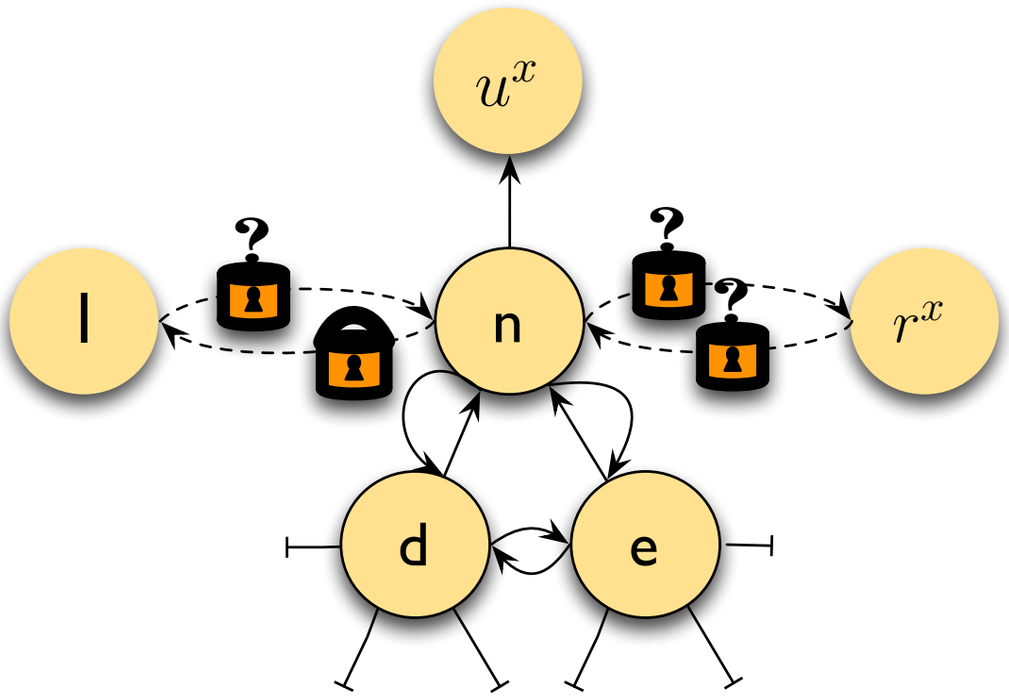


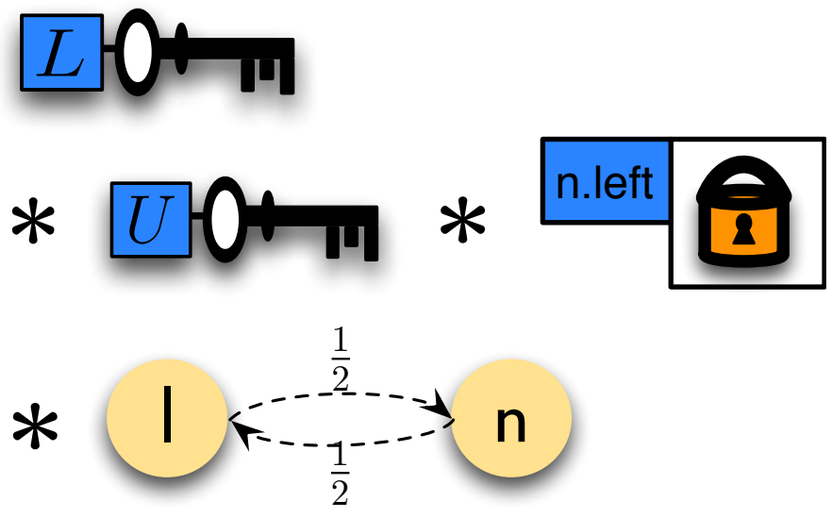


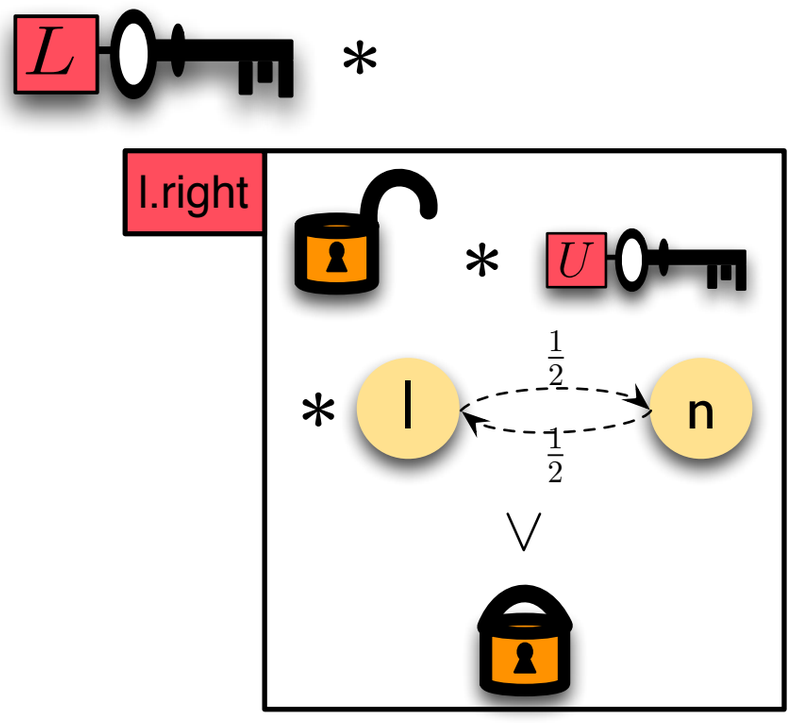
Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

lock(n.left);



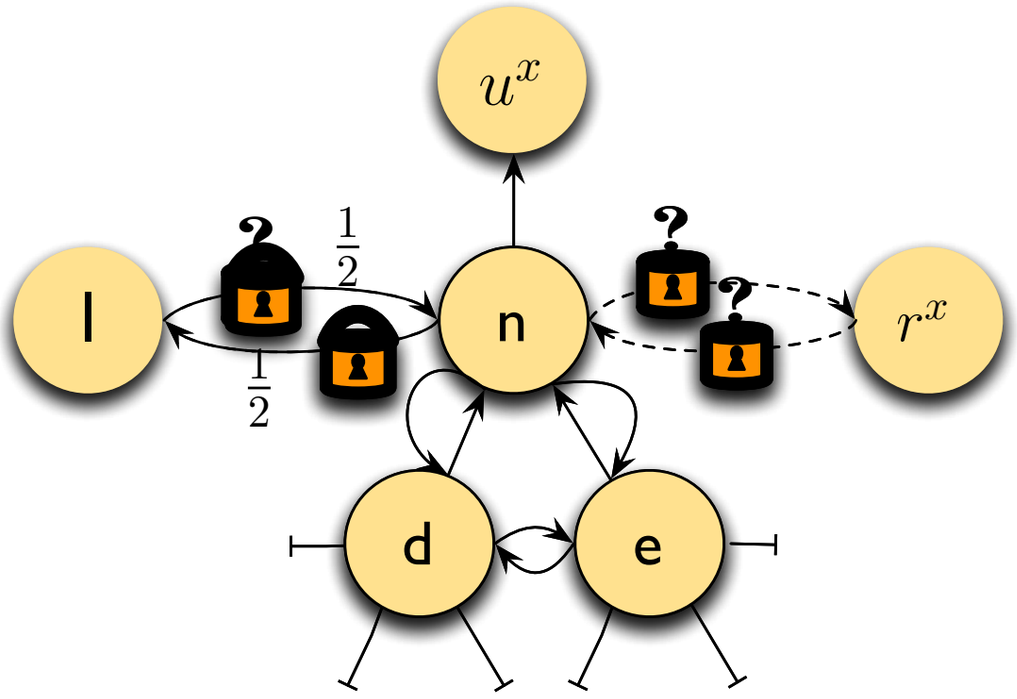


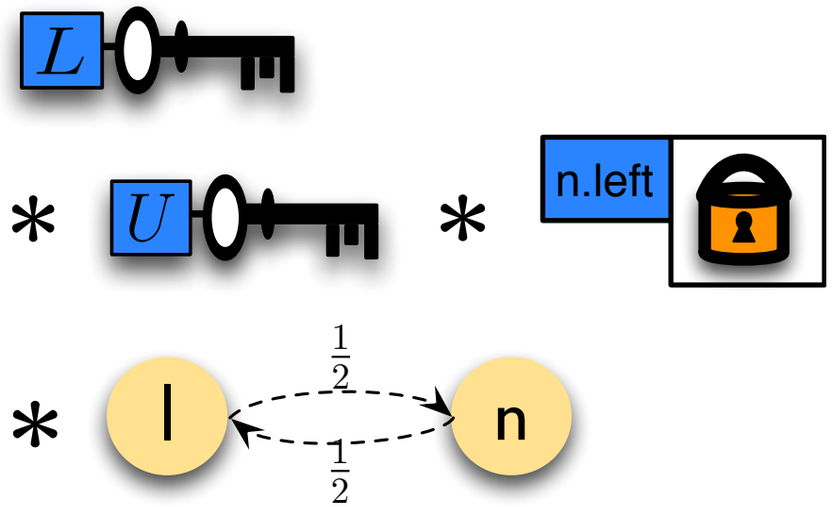


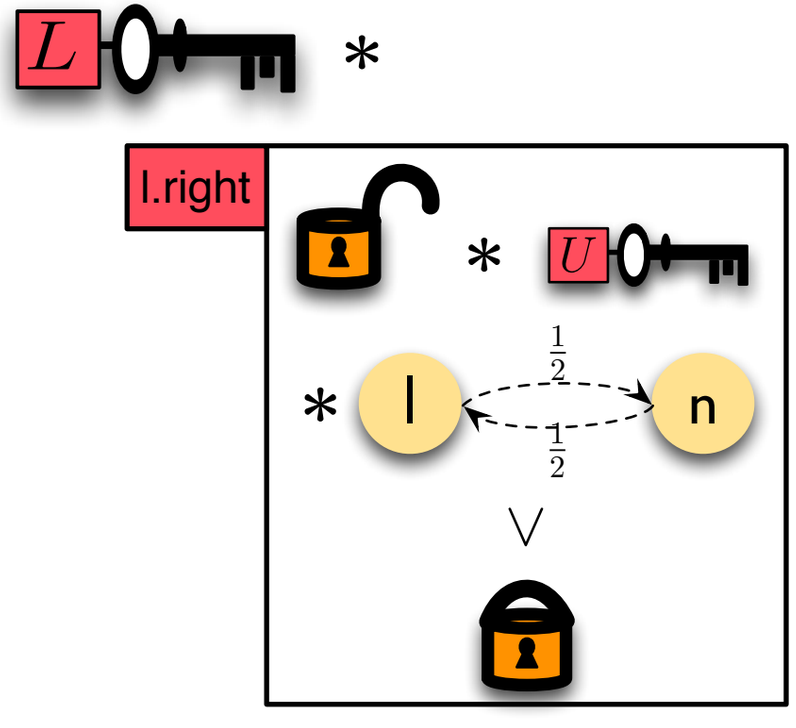
Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
l:= n.left;
lock(l.right);
```



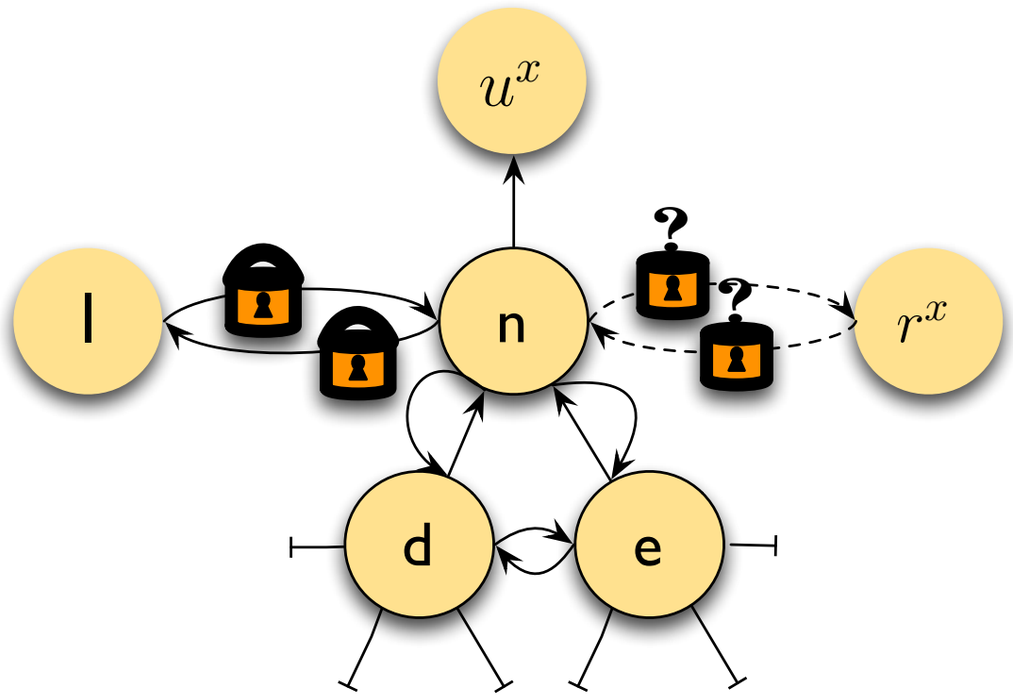


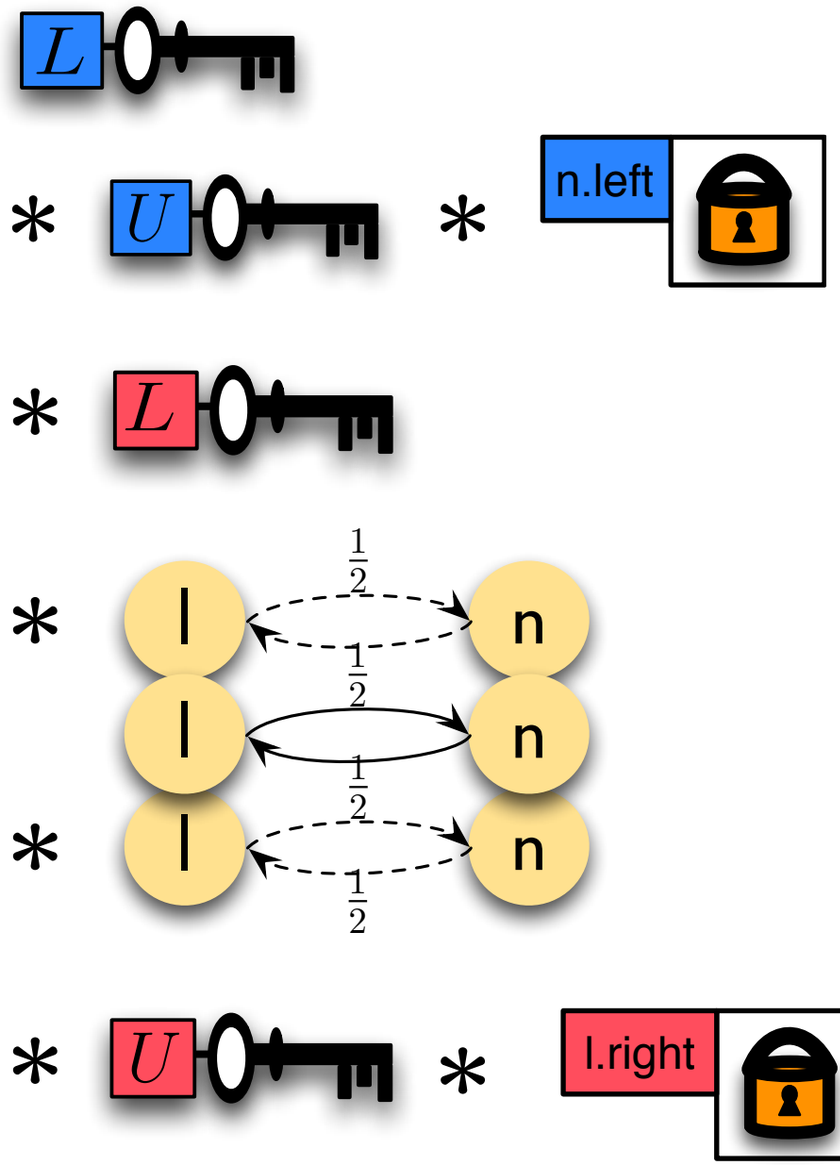


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
l:= n.left;
lock(l.right);
```

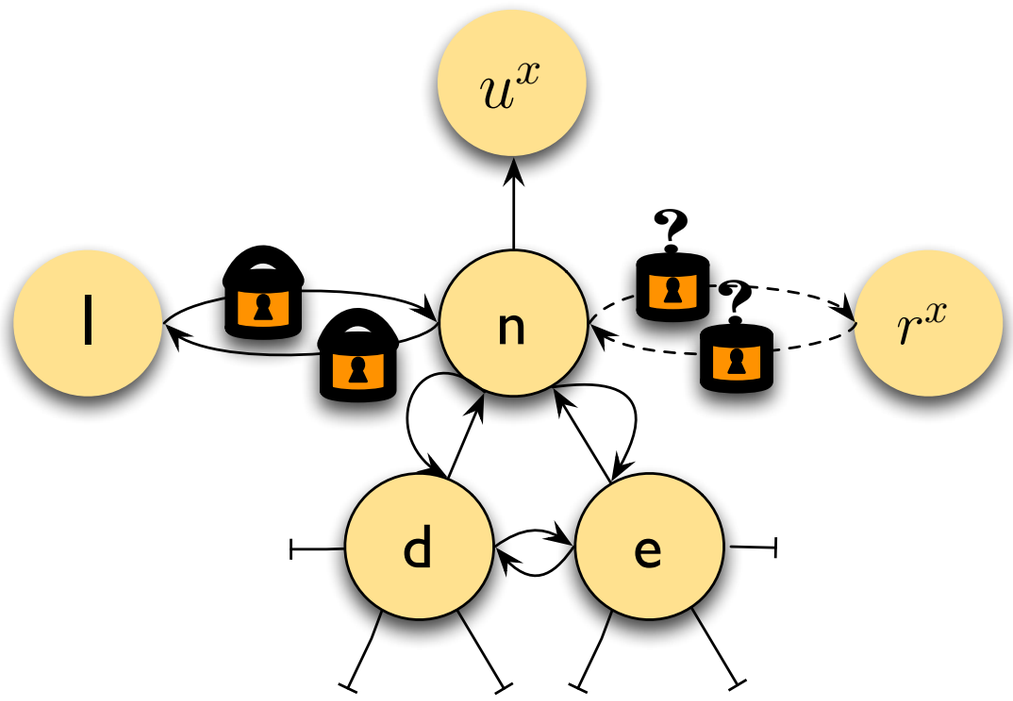


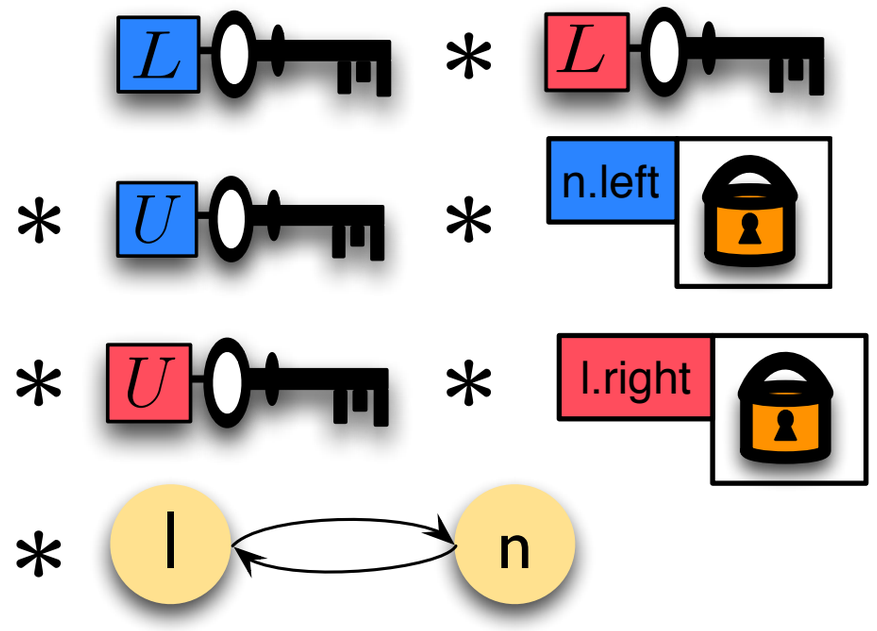


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
l:= n.left;
lock(l.right);
```





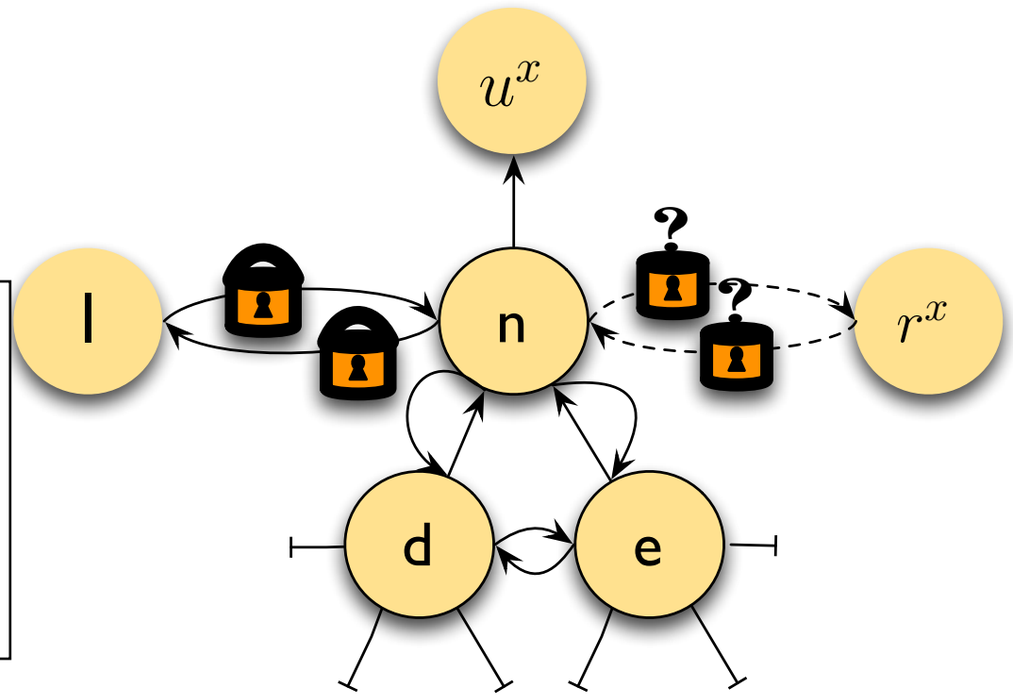
Refinement (Axiomatic Correctness)

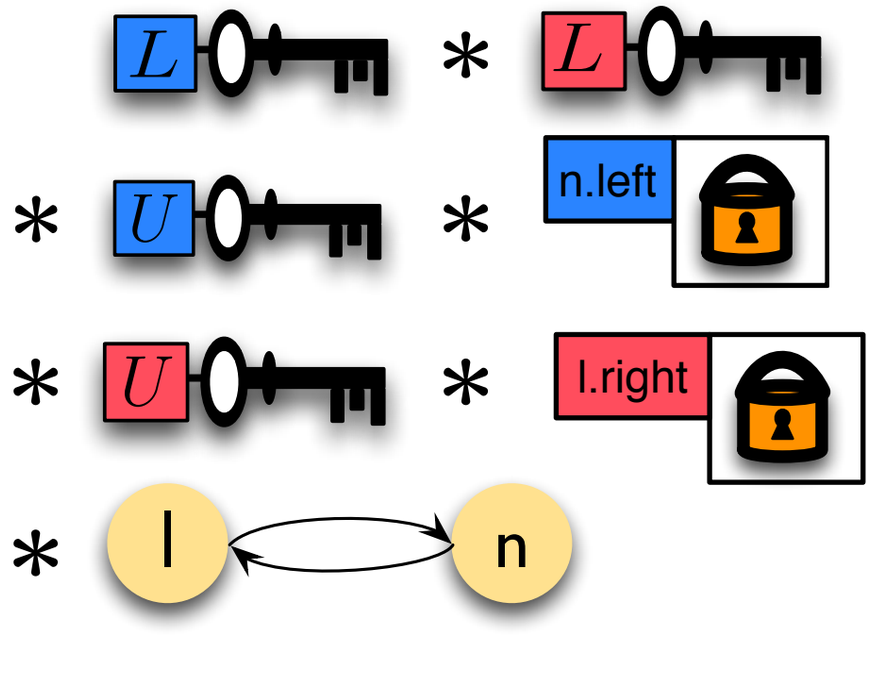
```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
```

```
//Do the same for RHS

lock(n.right);
r:= n.right;
lock(r.left);
```

```
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```





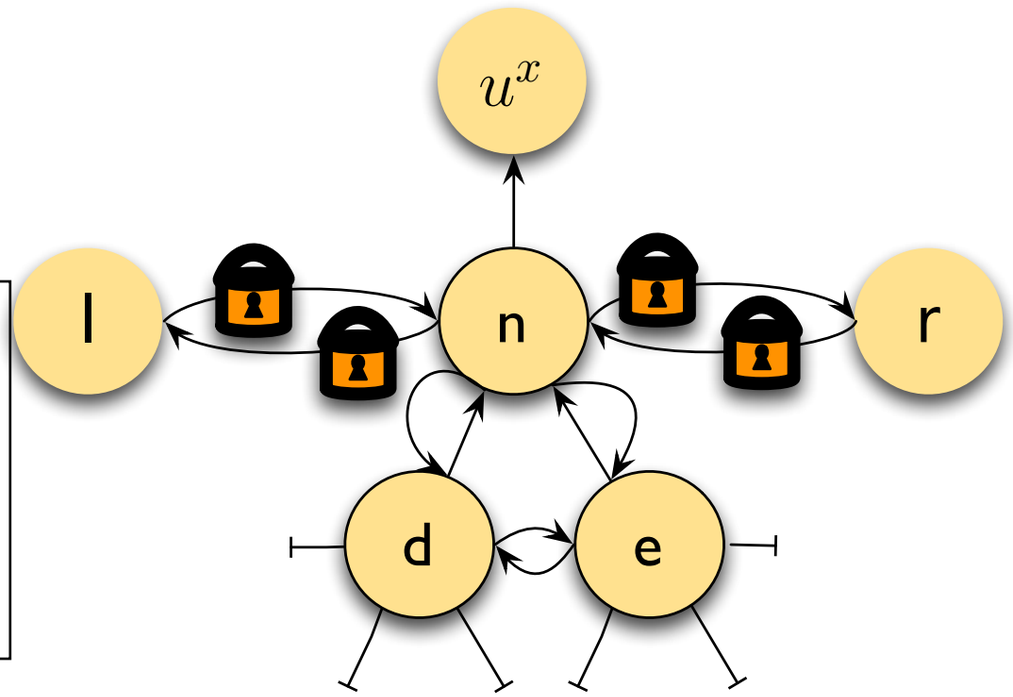
Refinement (Axiomatic Correctness)

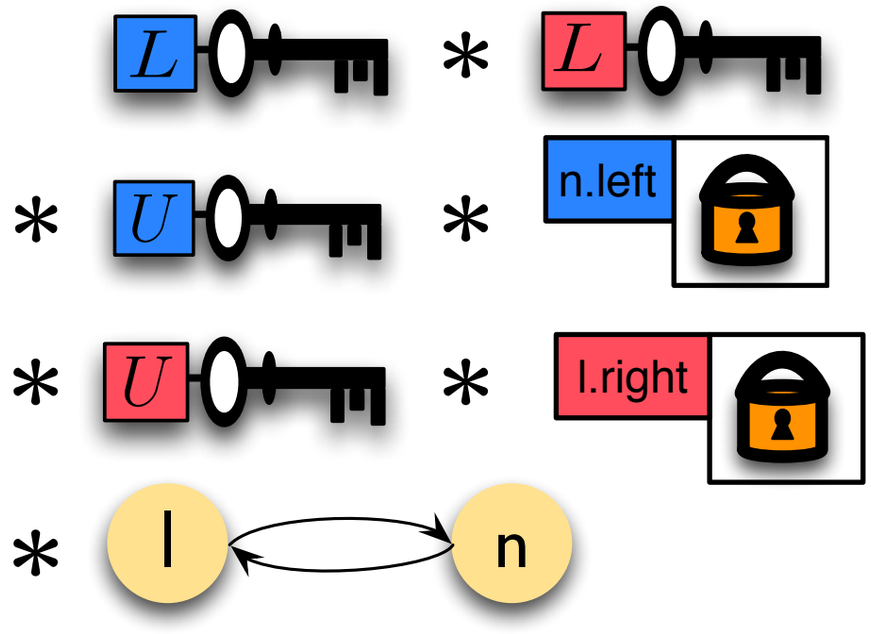
```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
```

//Do the same for RHS

```
lock(n.right);
r:= n.right;
lock(r.left);
```

```
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```



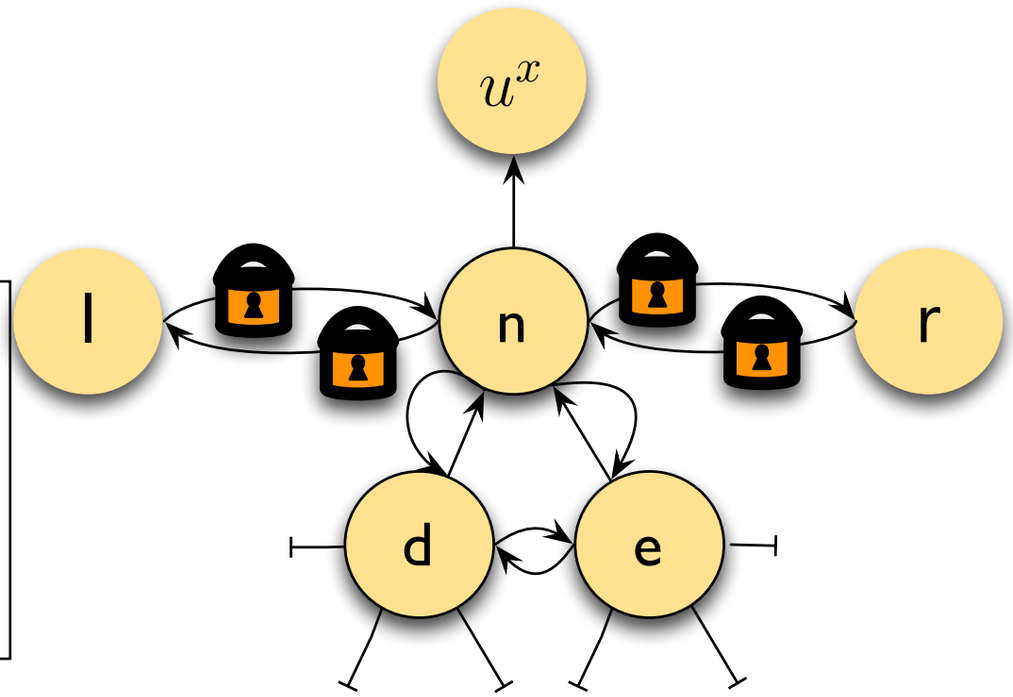


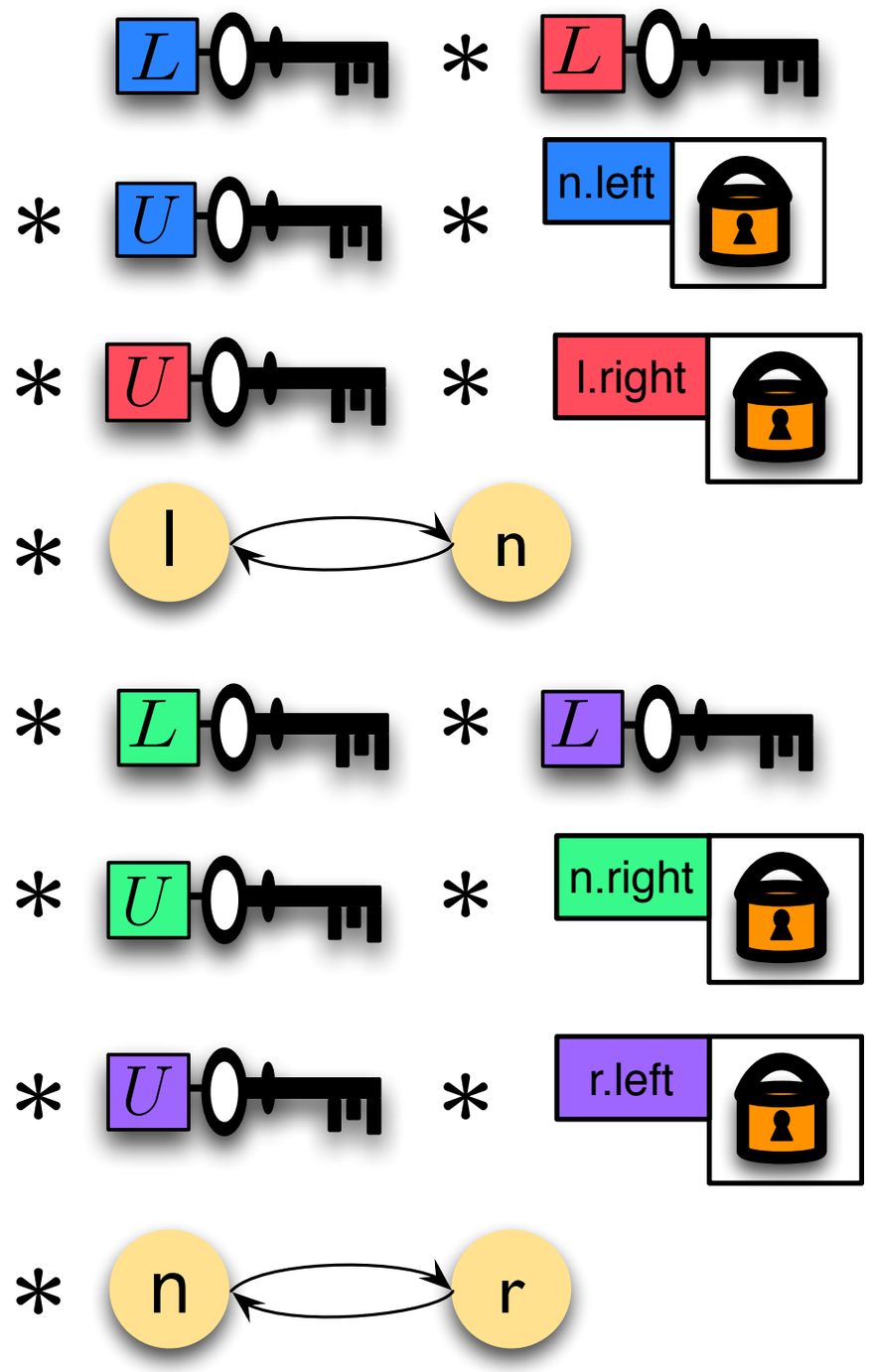
Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
//Do the same for RHS

lock(n.right);
r:= n.right;
lock(r.left);
```

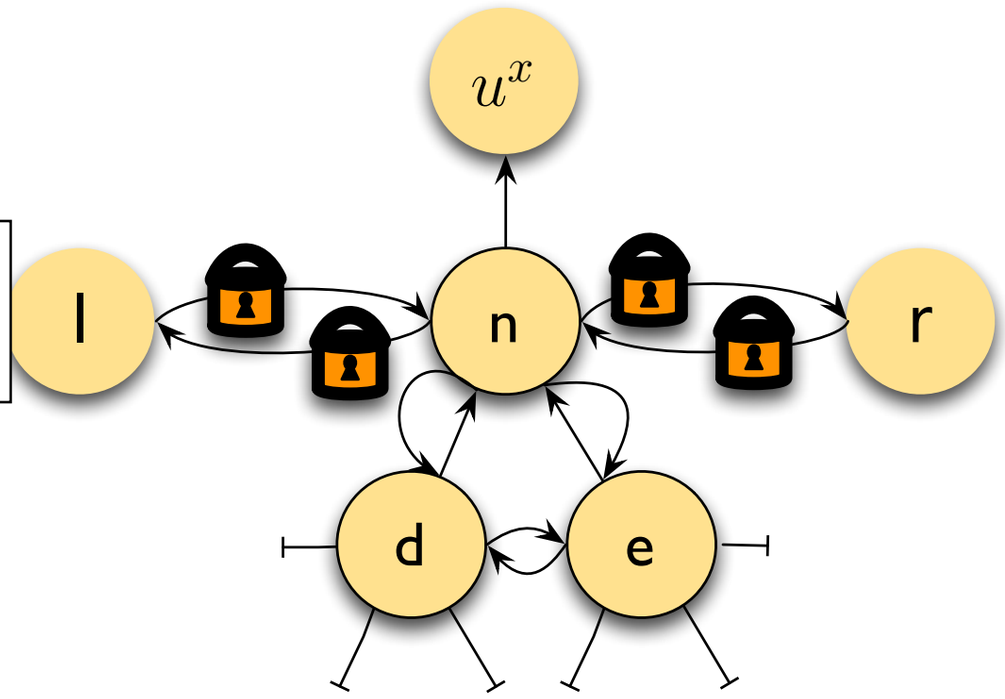


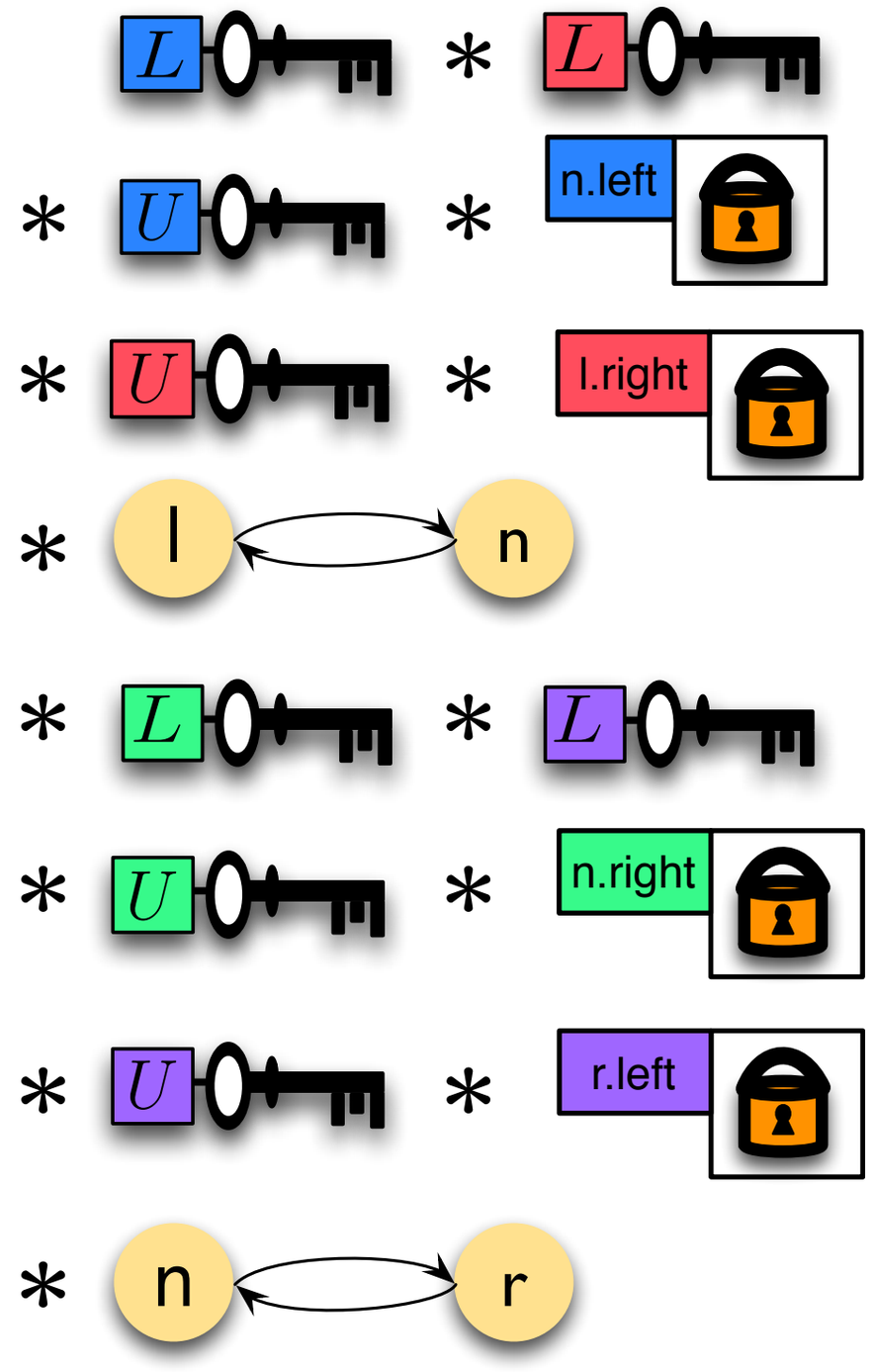


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
[l.right]:=r;
[r.left]:=l;
```

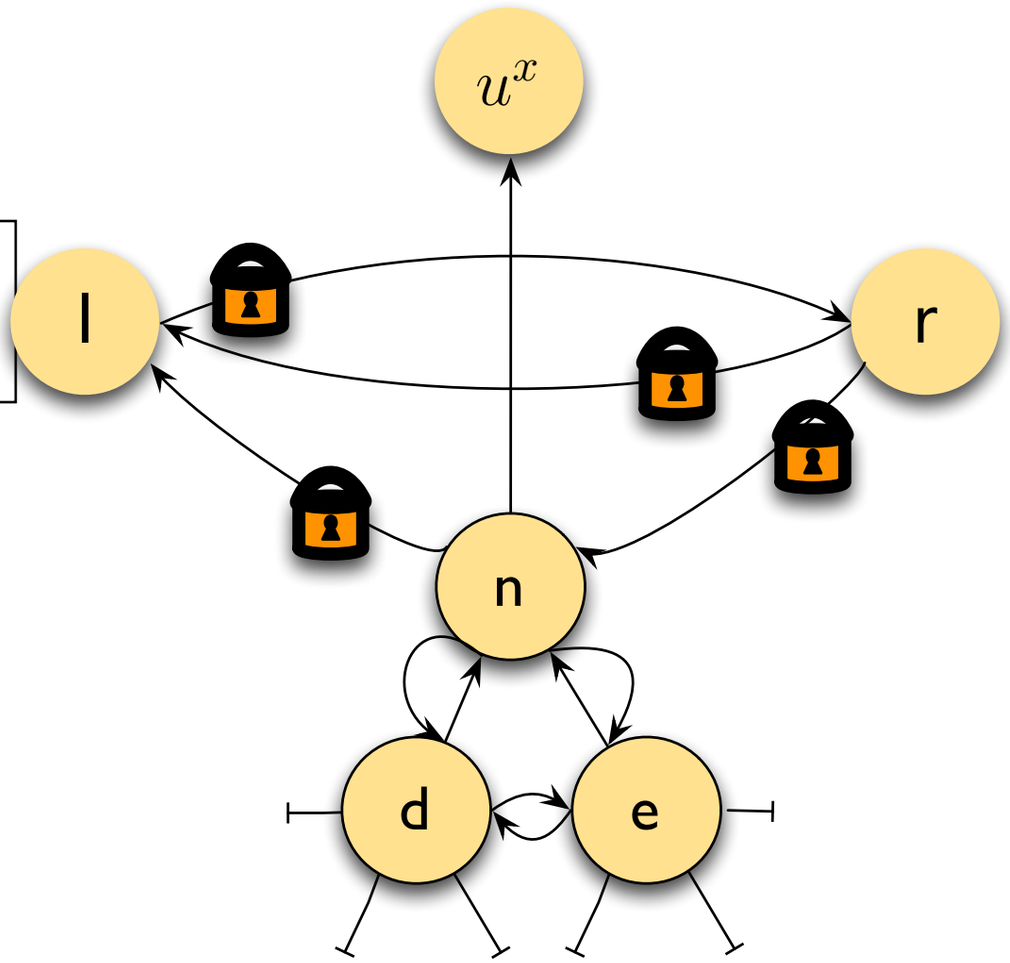


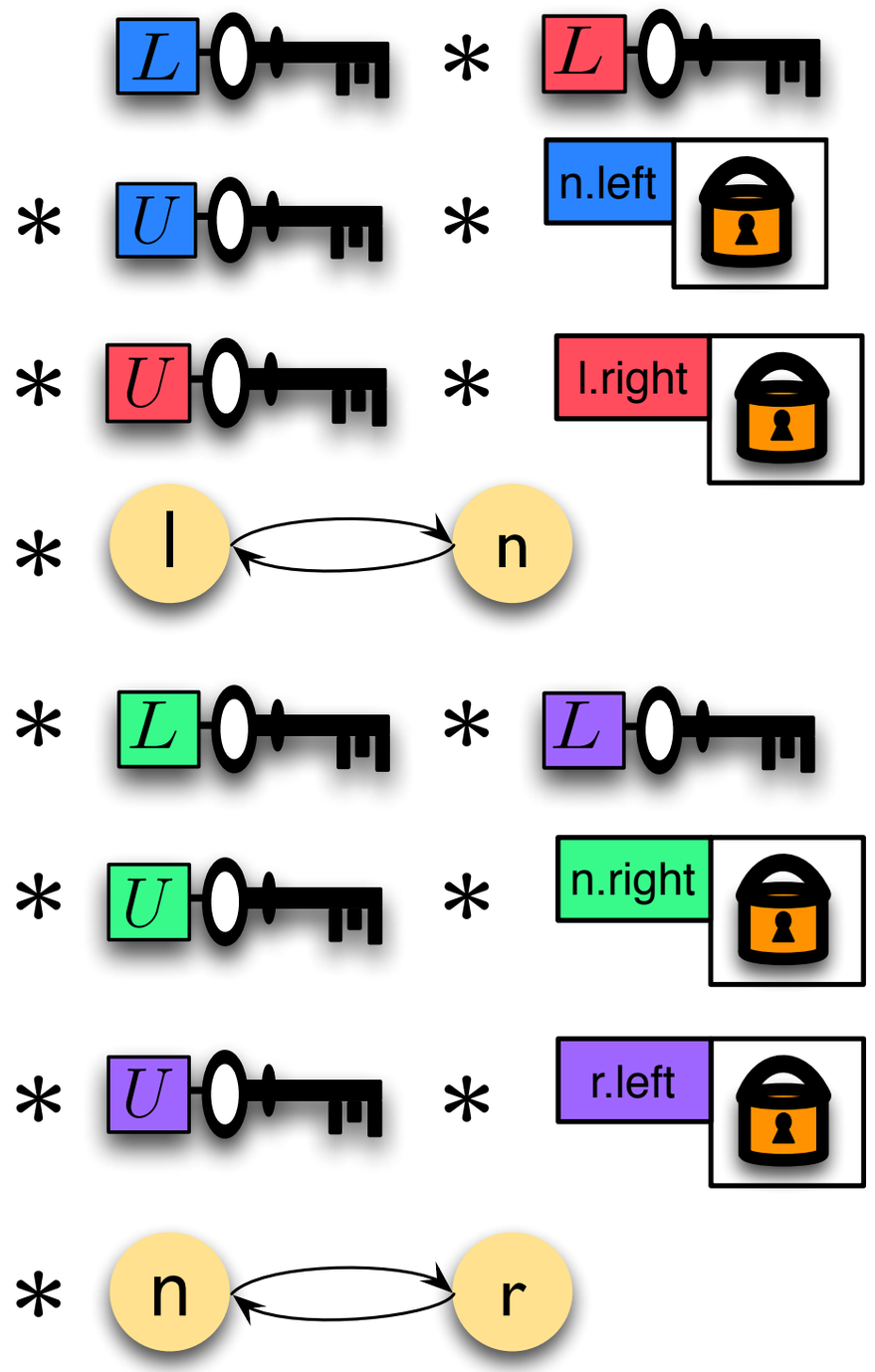


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
[l.right]:=r;
[r.left]:=l;
```

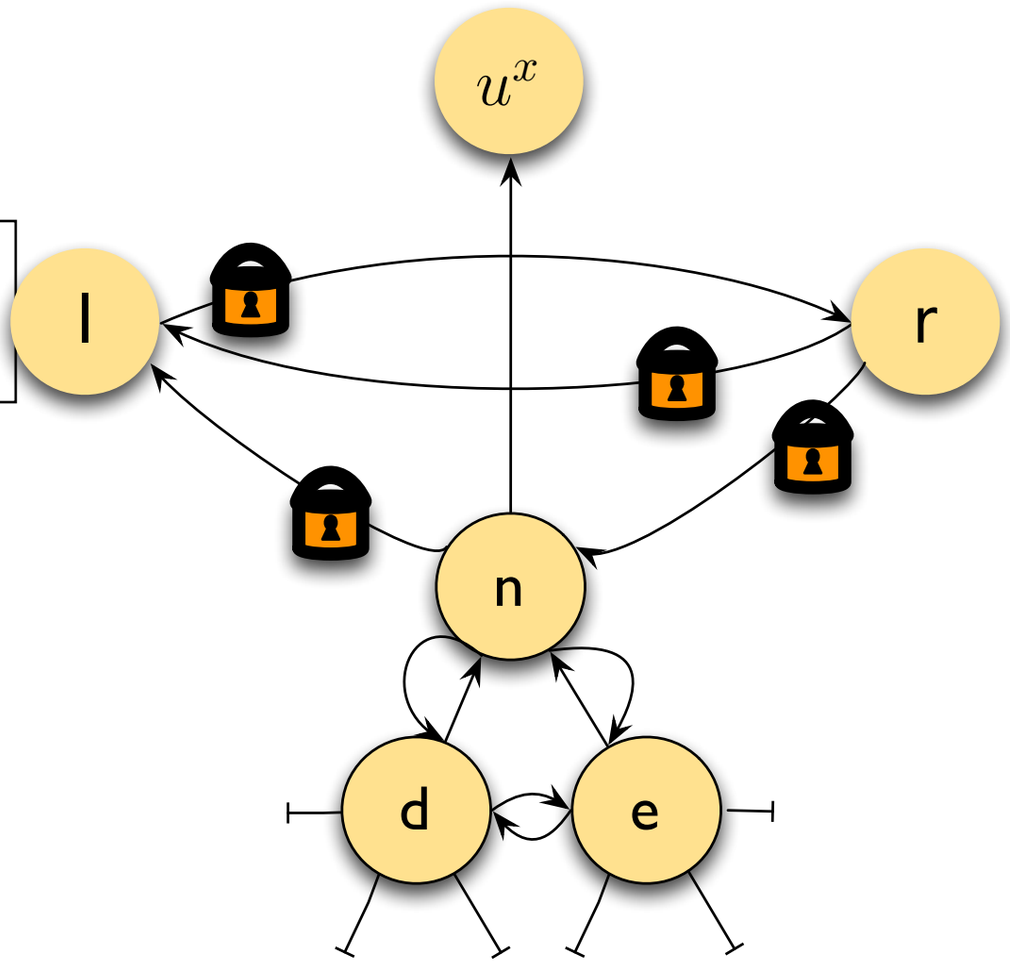


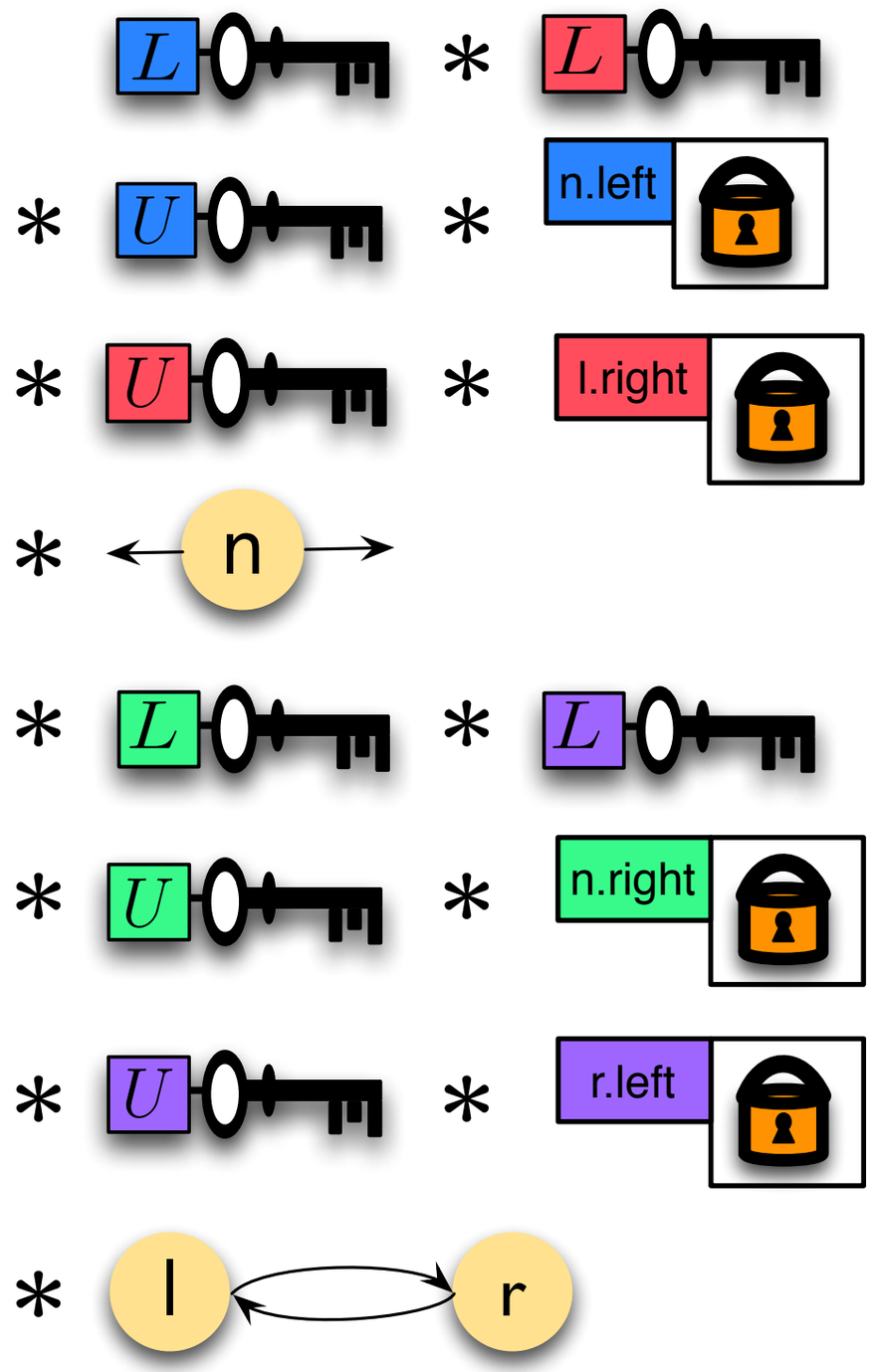


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
[l.right]:=r;
[r.left]:=l;
```

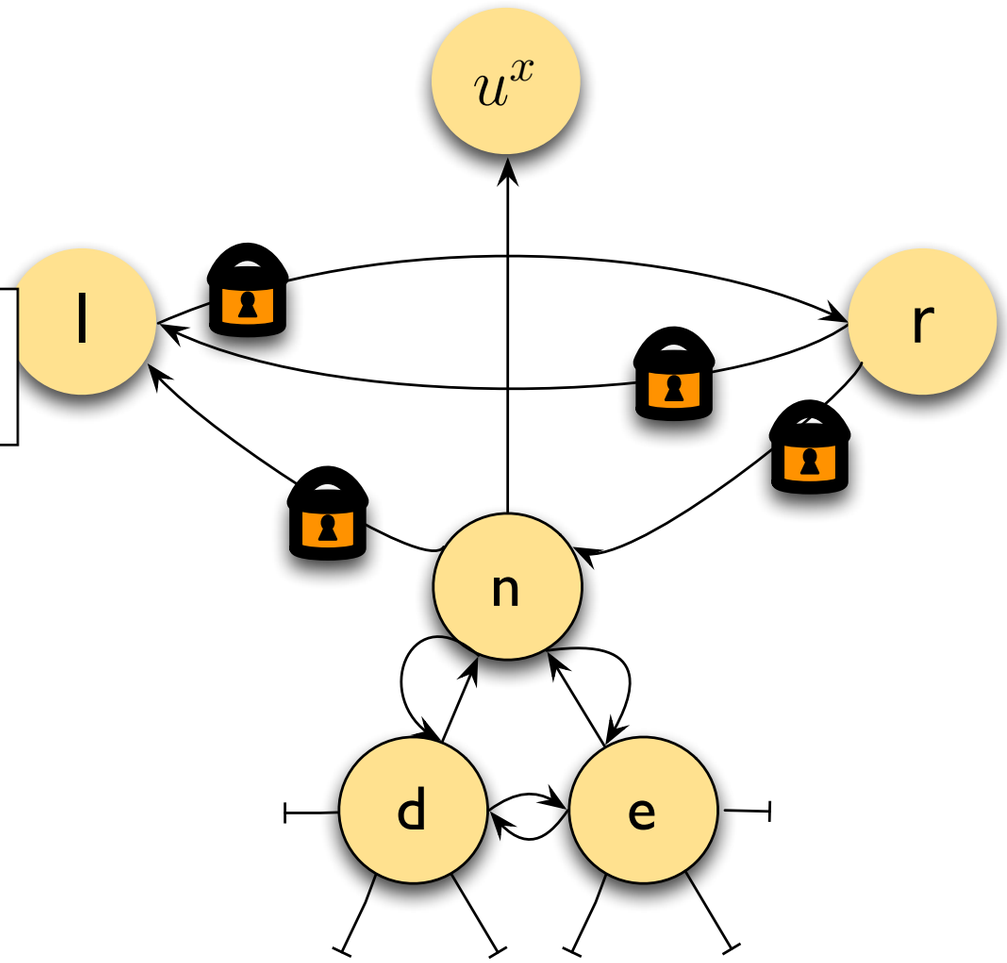


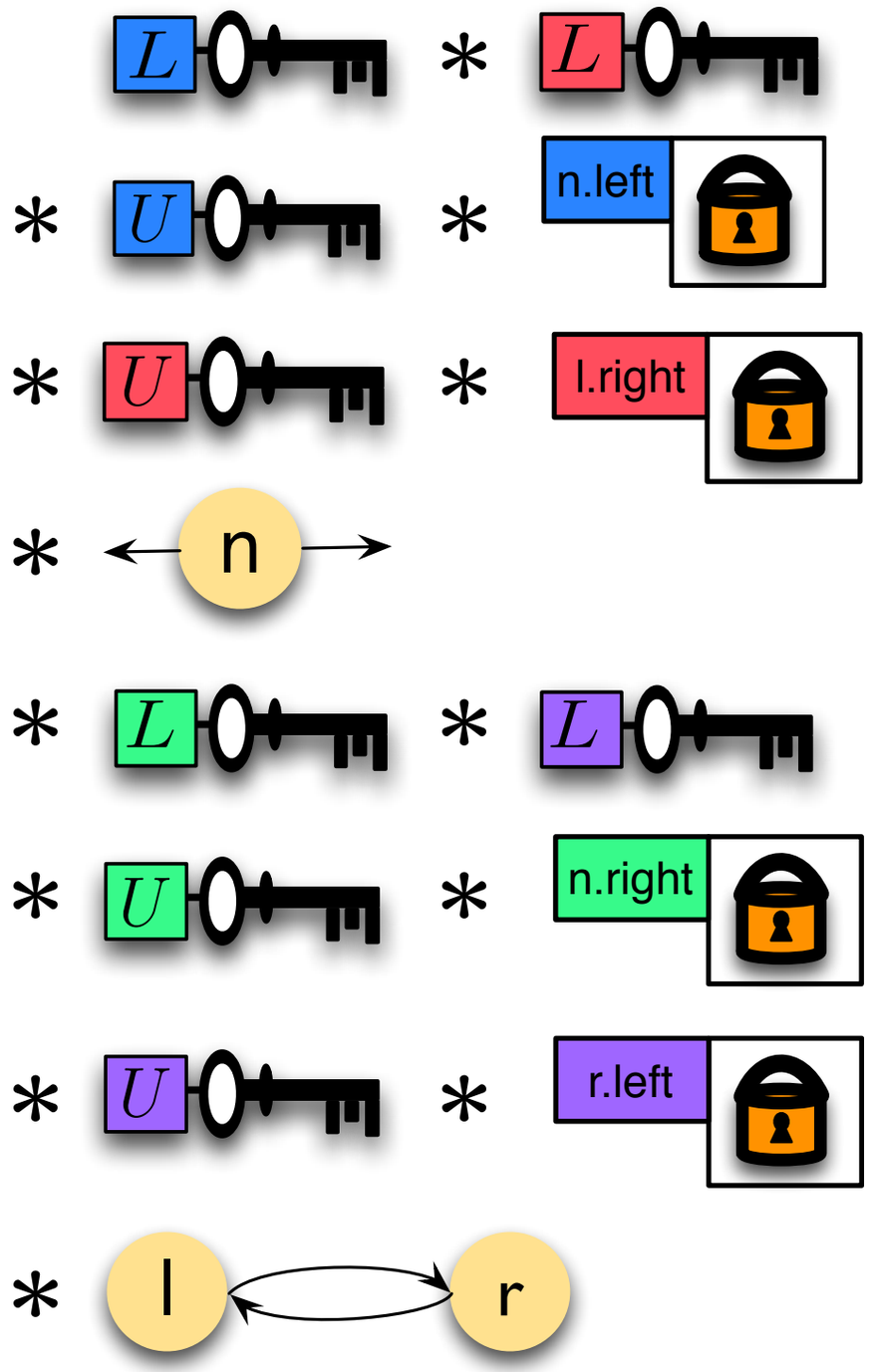


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```

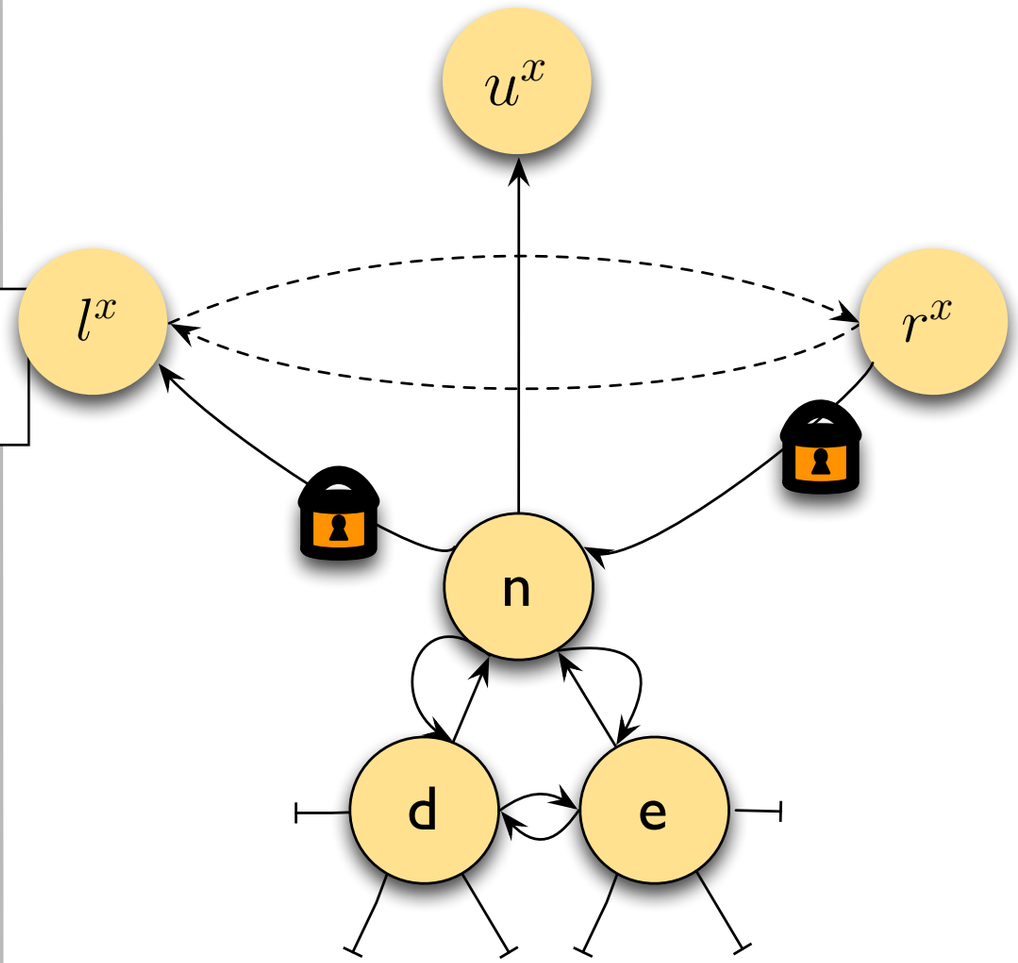


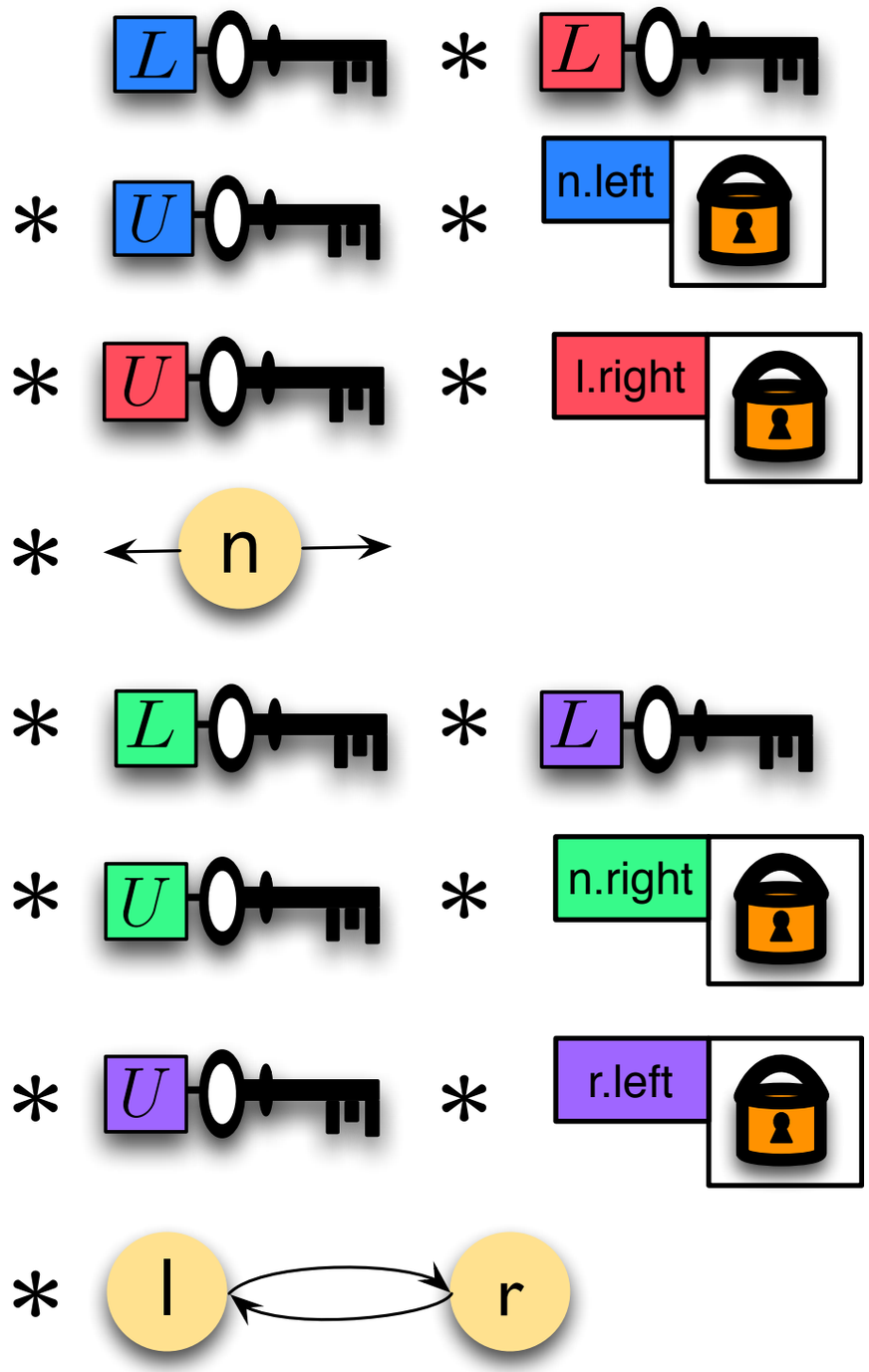


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```

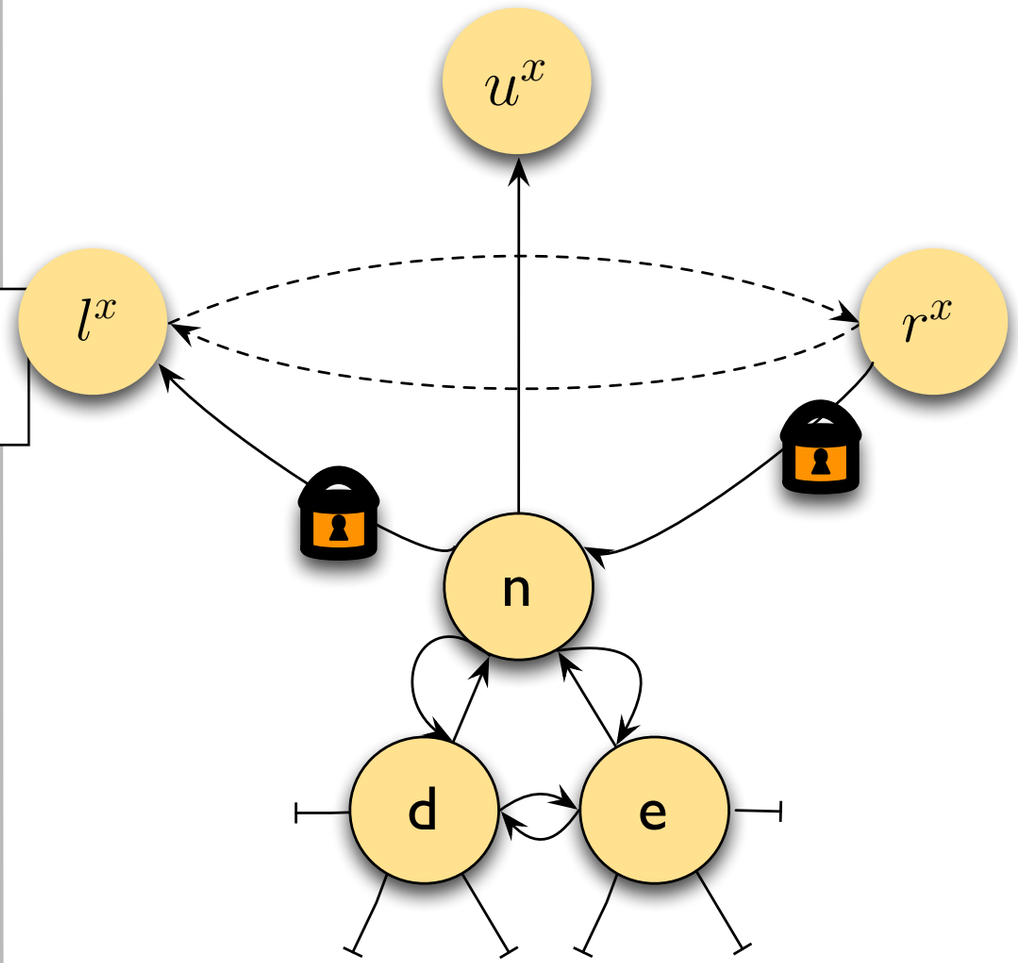


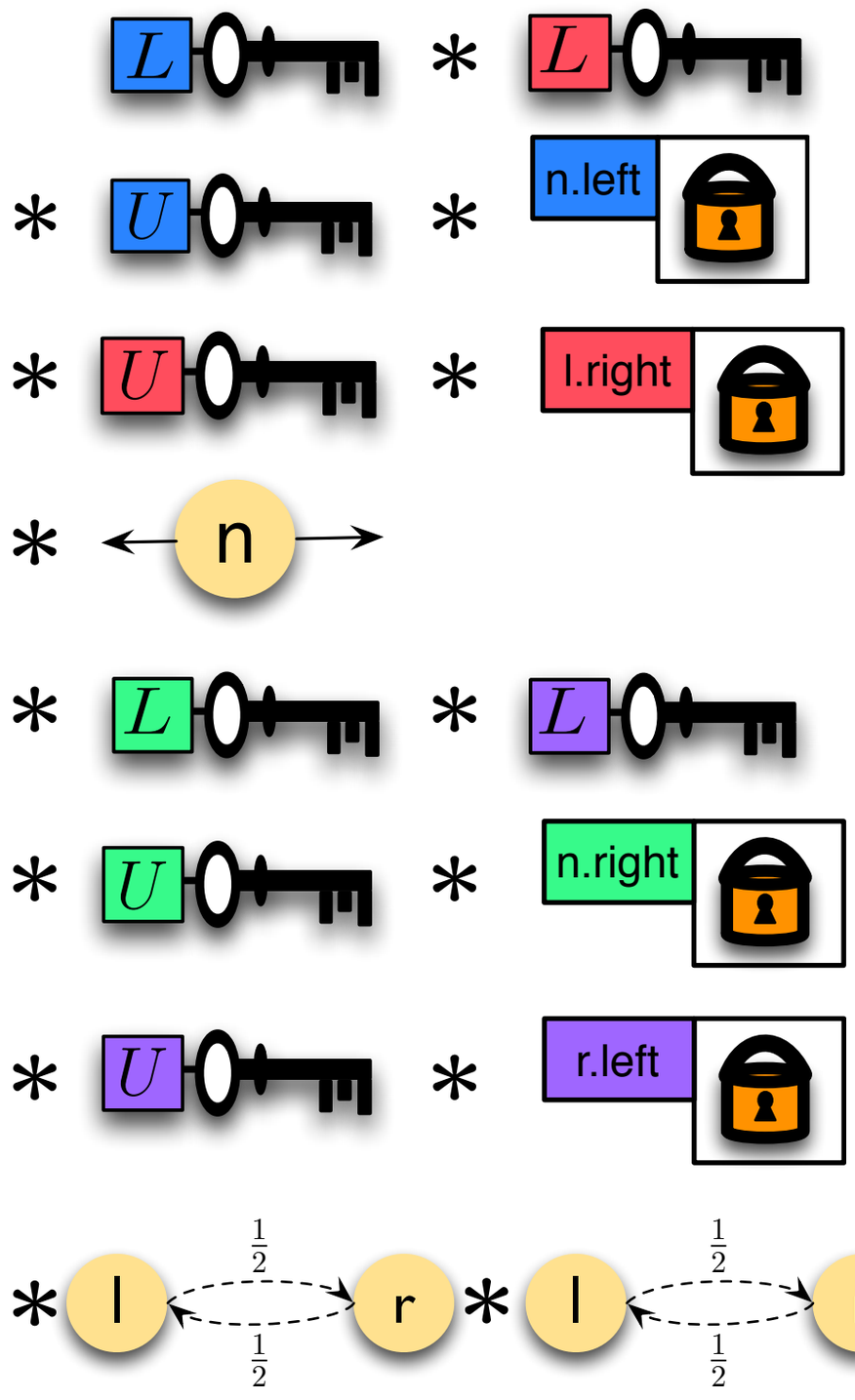


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```

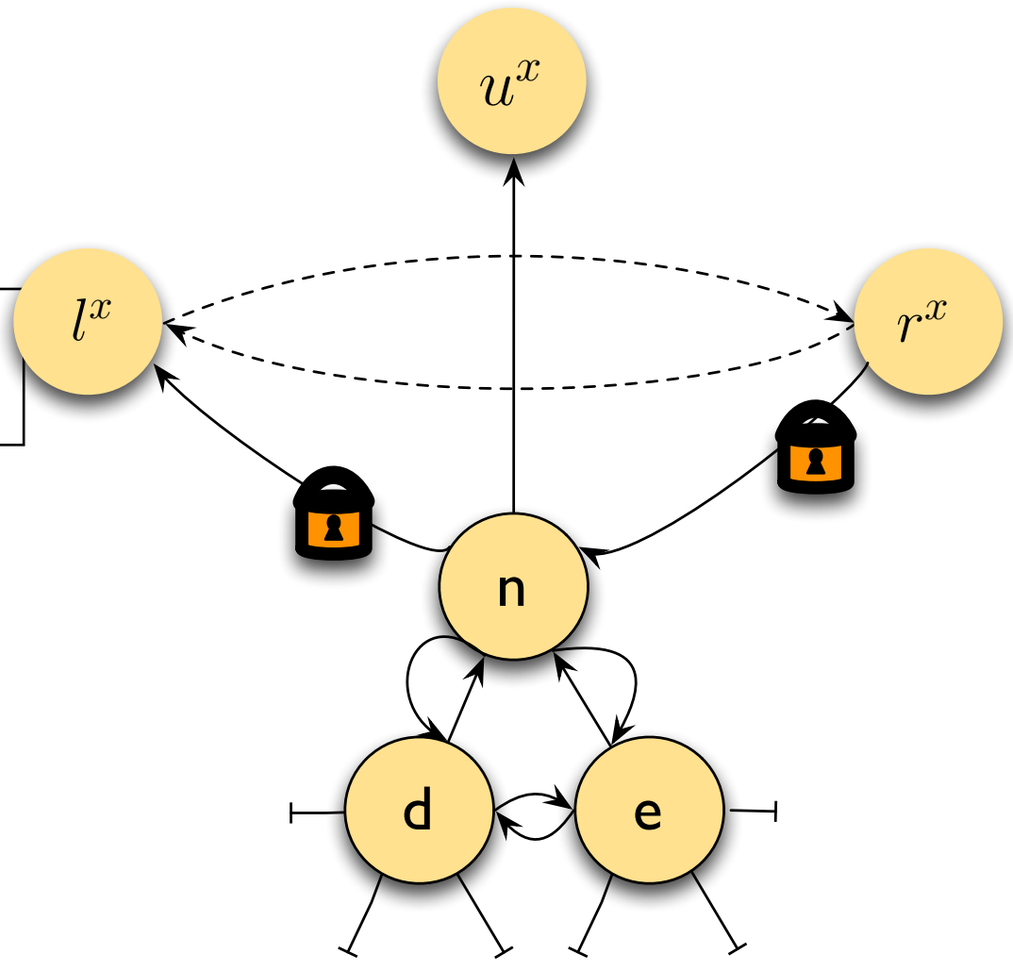


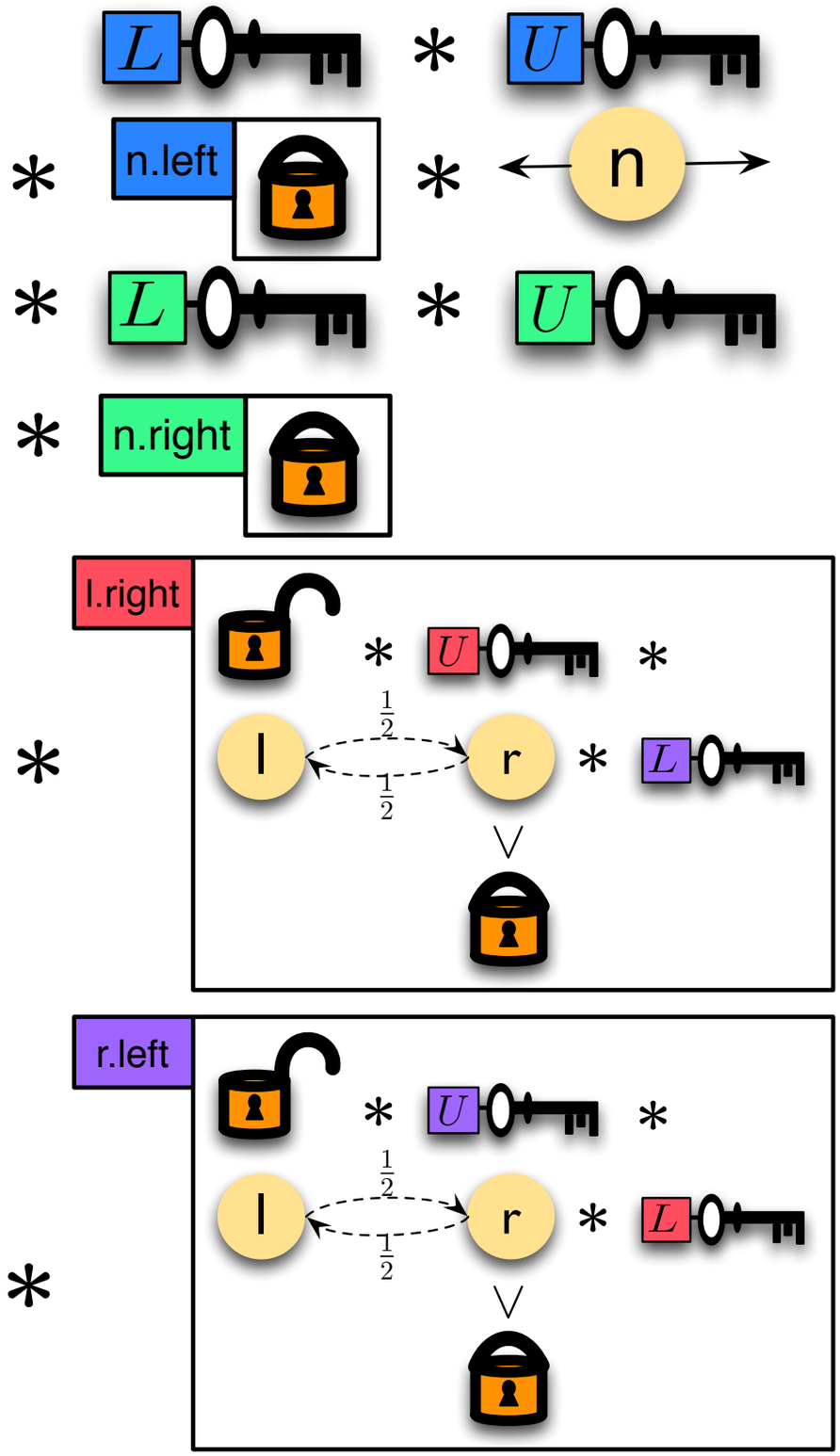


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```

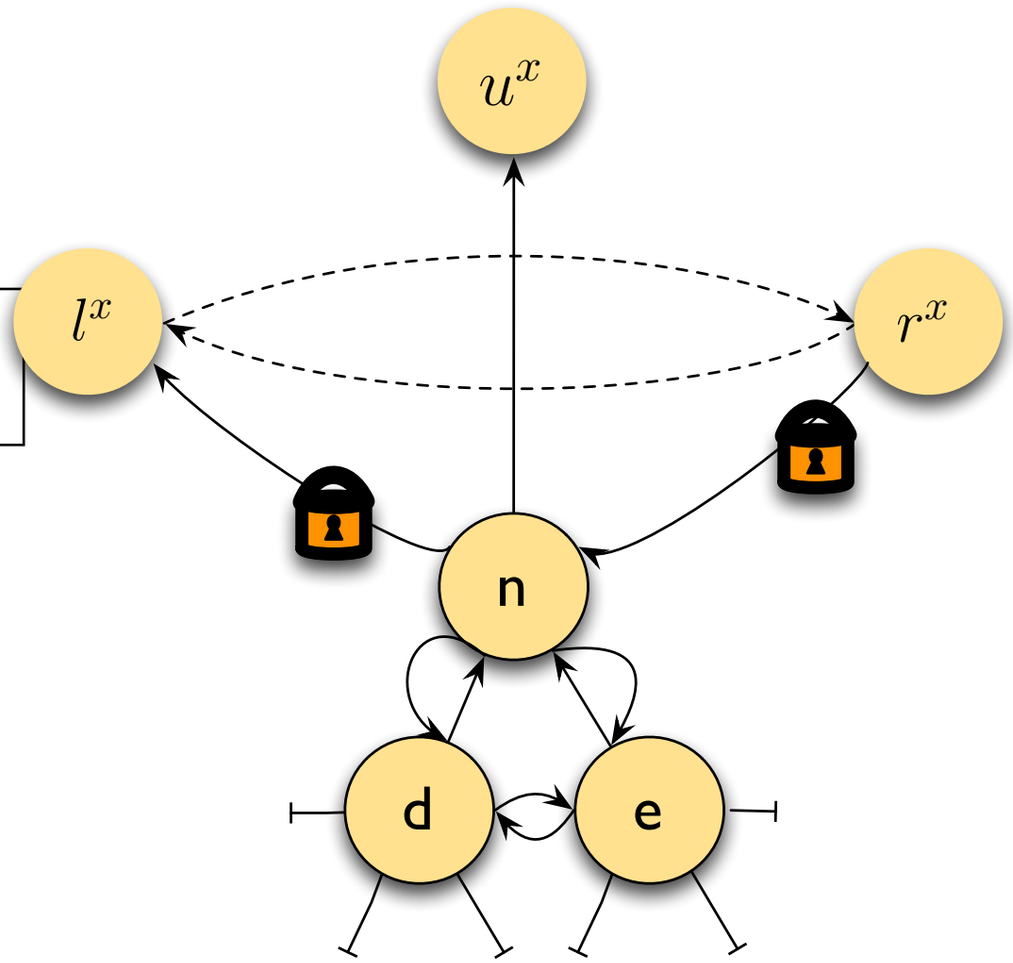


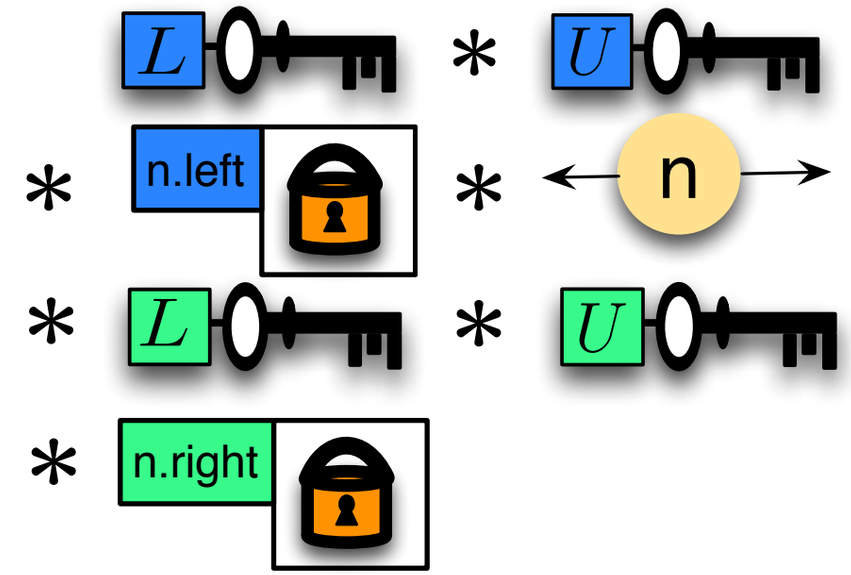


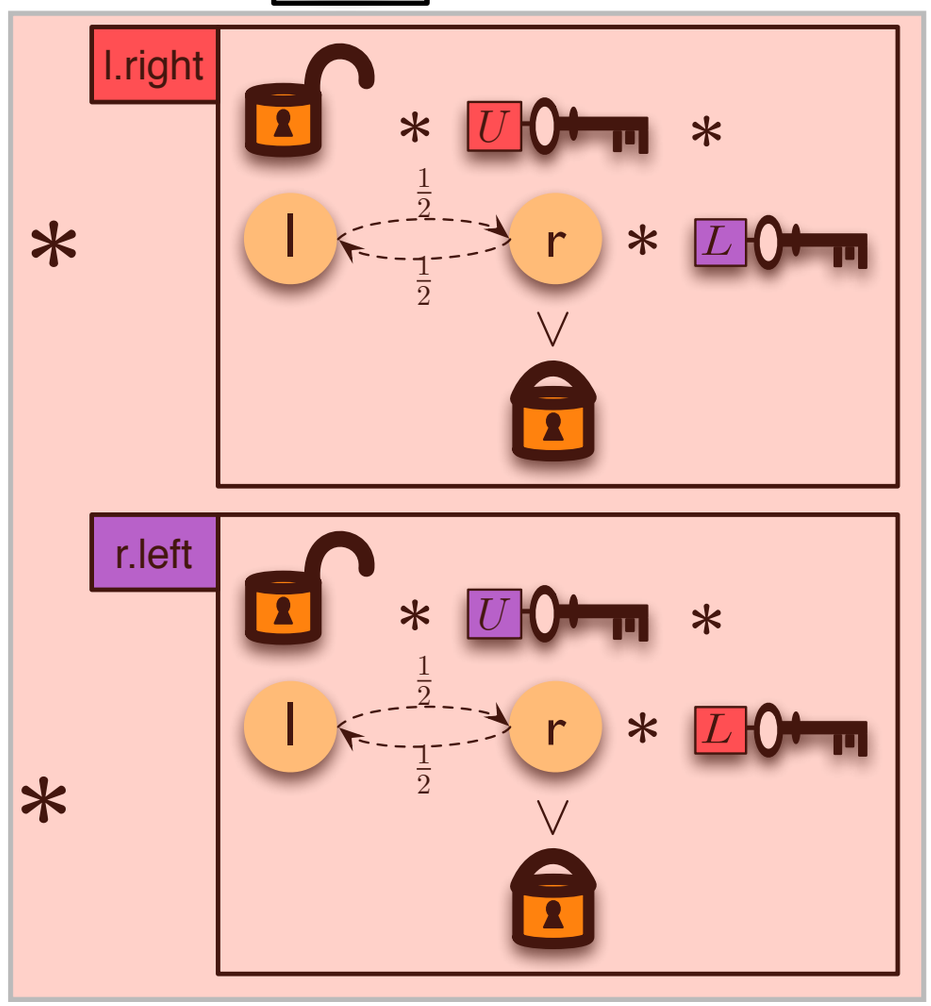
Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```



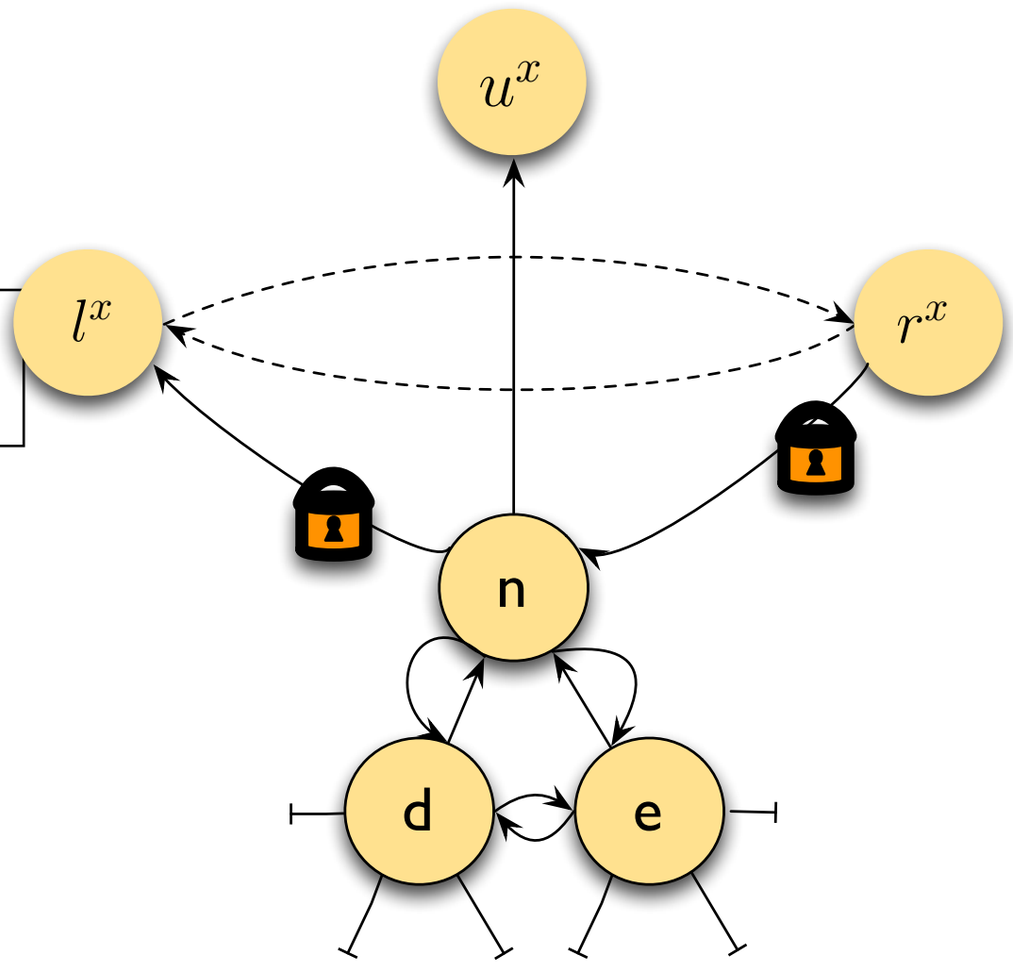


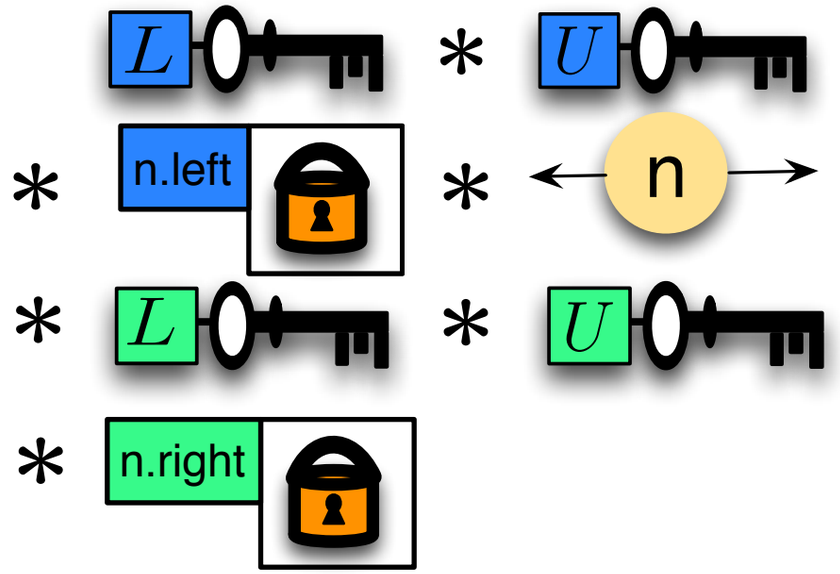


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

```
unlock(l.right);
unlock(r.left);
```



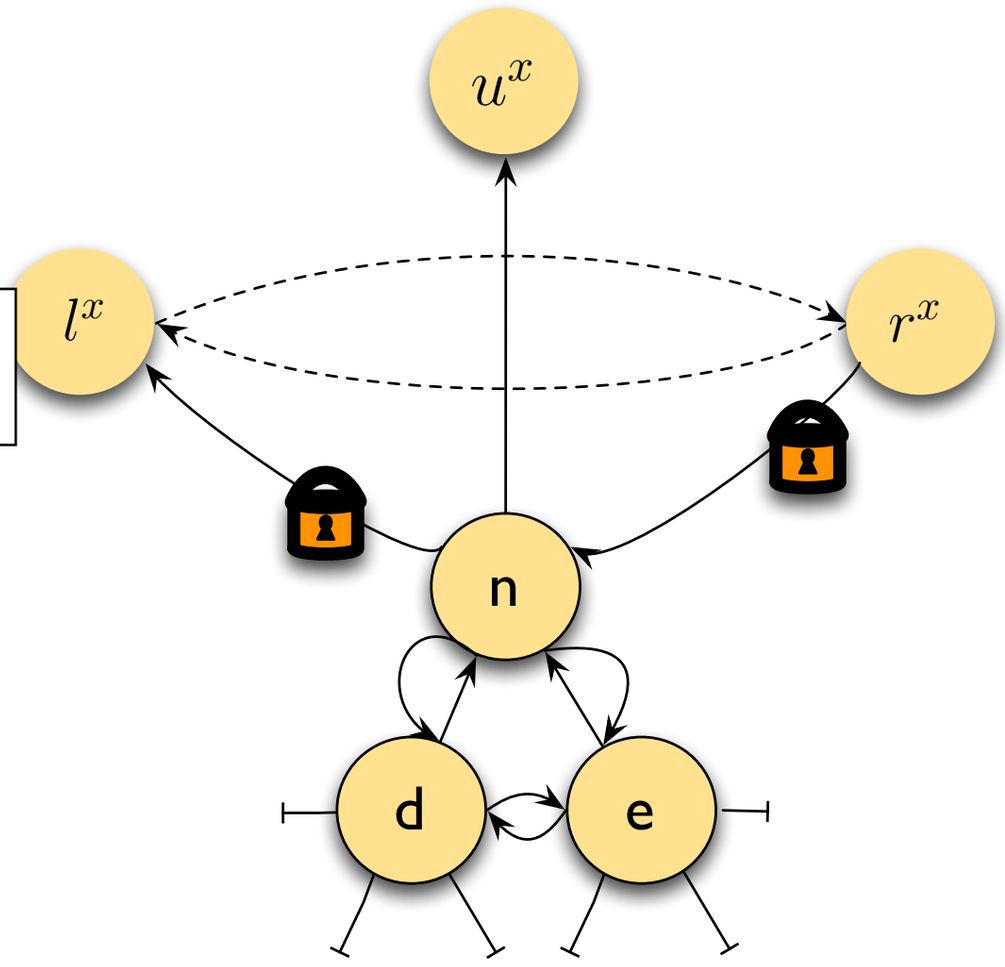


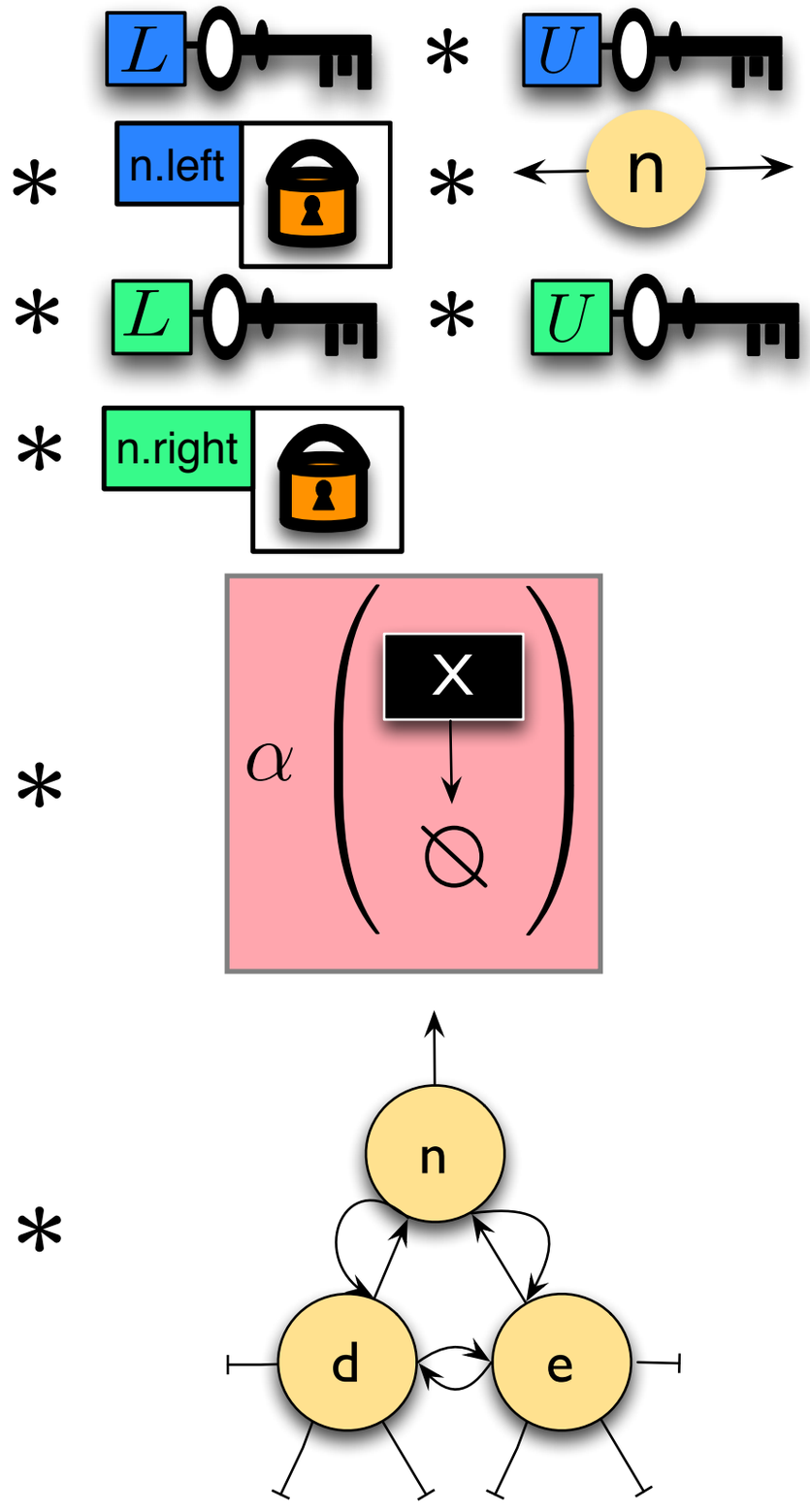
$*$ Crust $(r^x, l^x)\,(l^x, u^x, r^x)$

Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

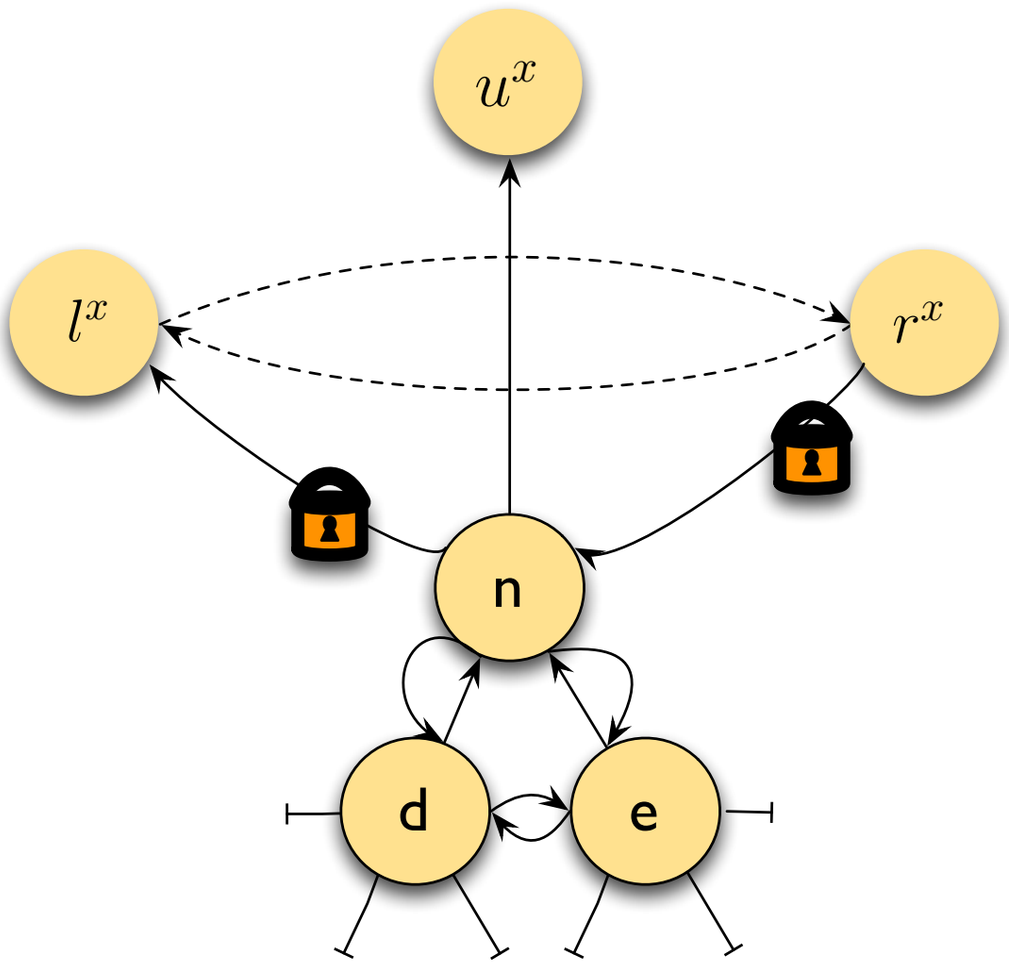
```
unlock(l.right);
unlock(r.left);
```

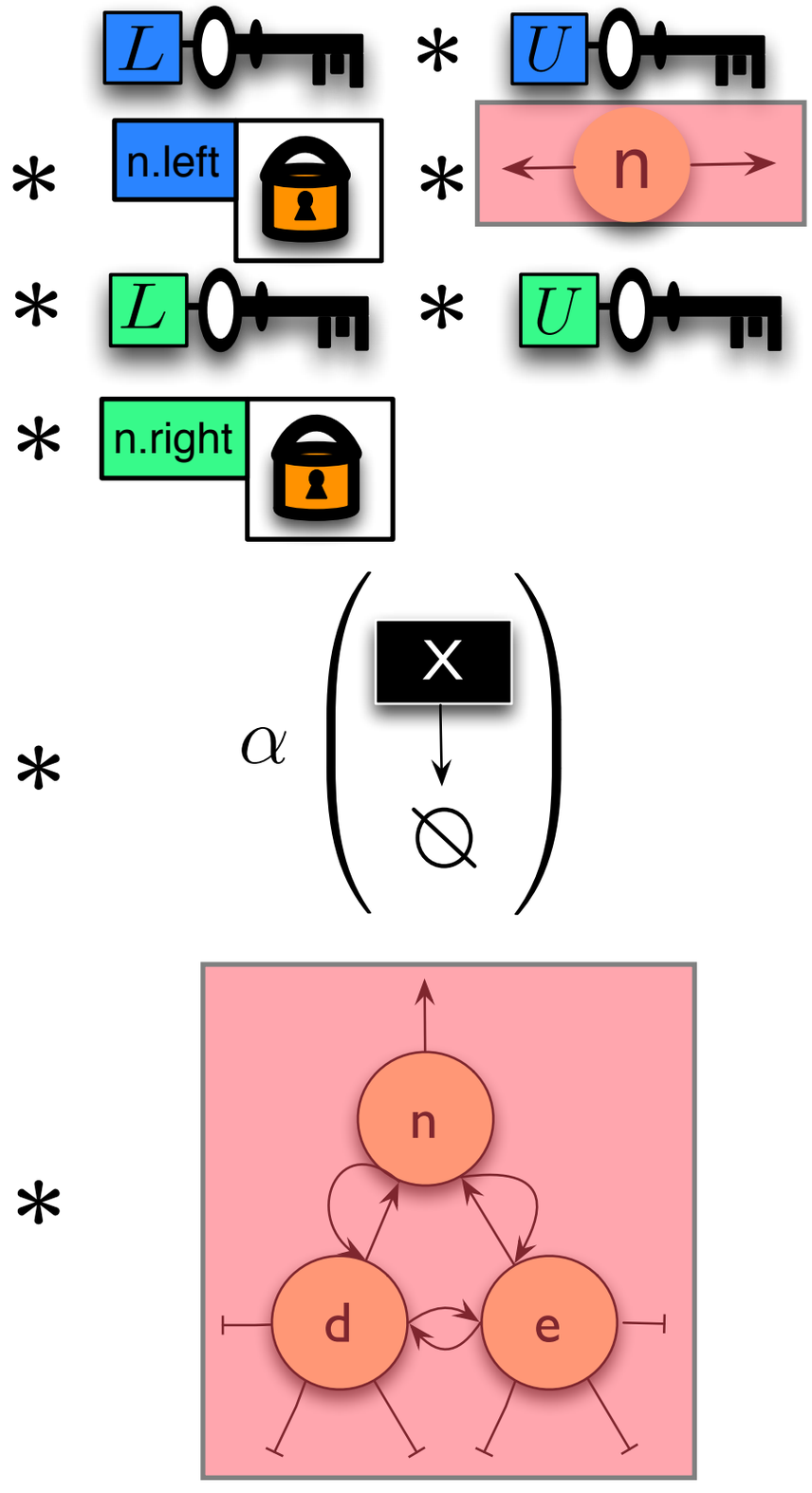




Refinement (Axiomatic Correctness)

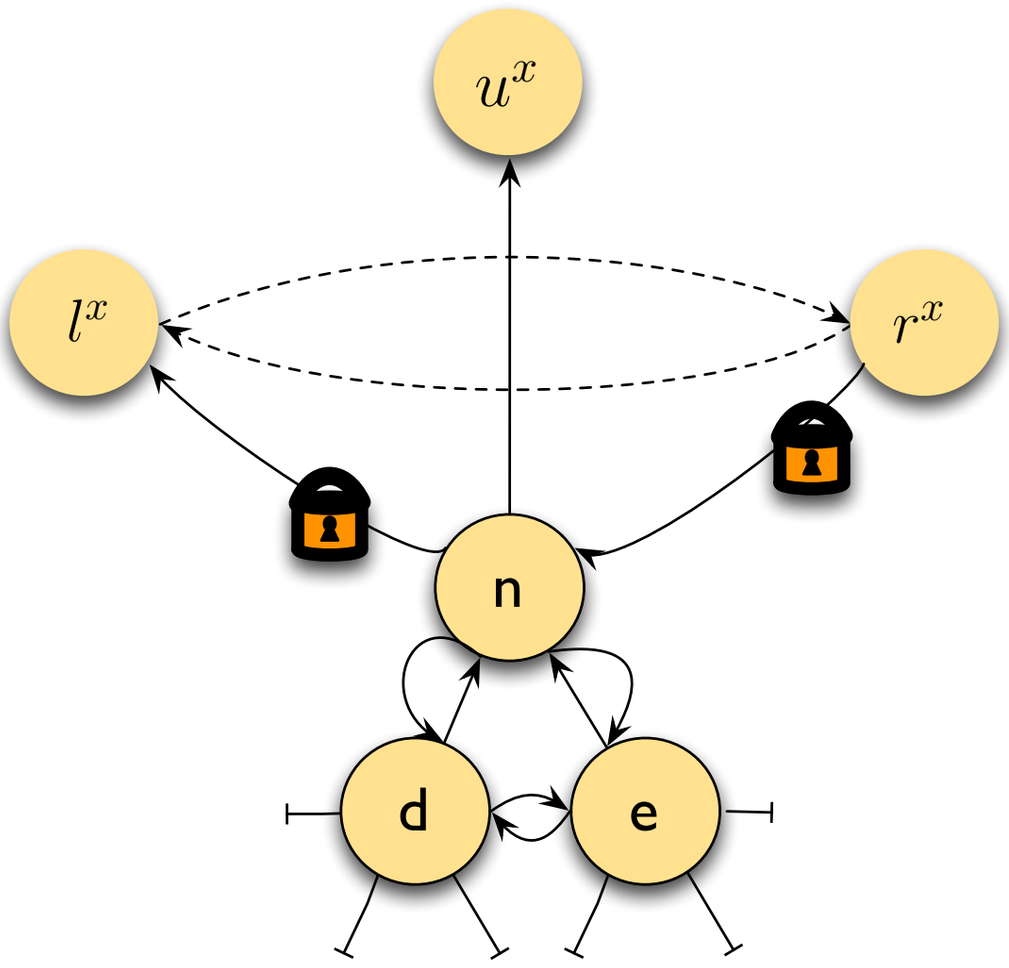
```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then      [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

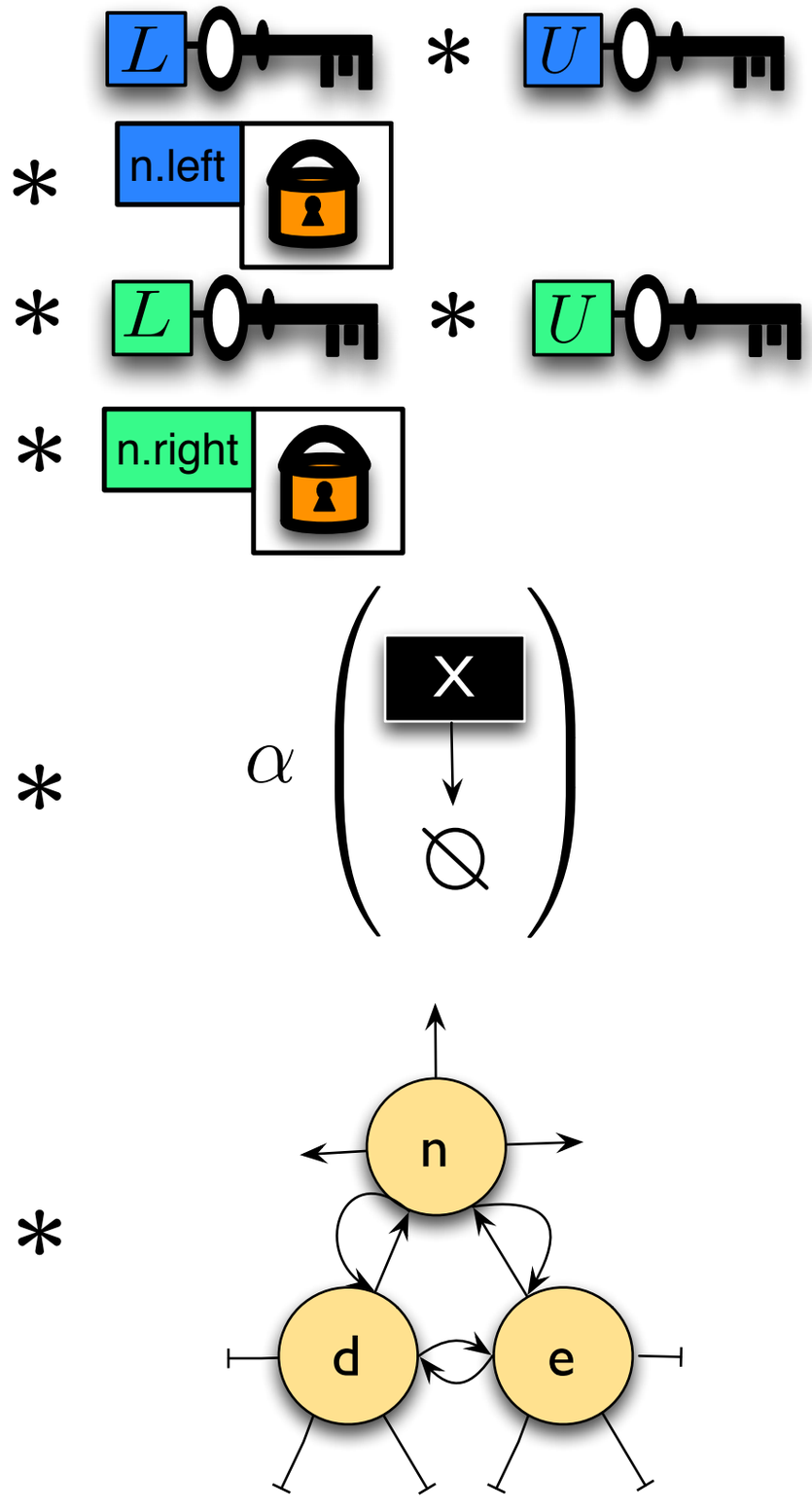




Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    else if u ≠ null then lock(u.lastL) ;
    unlock(ul) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

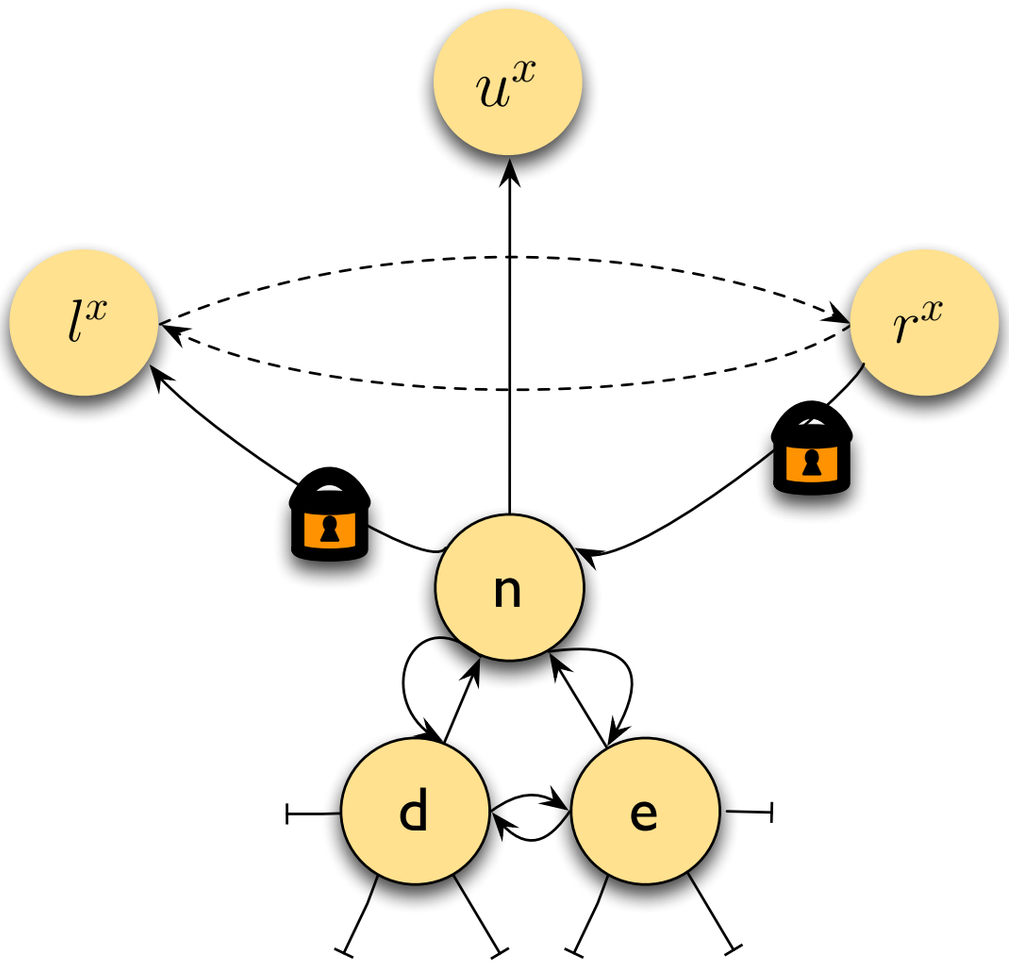


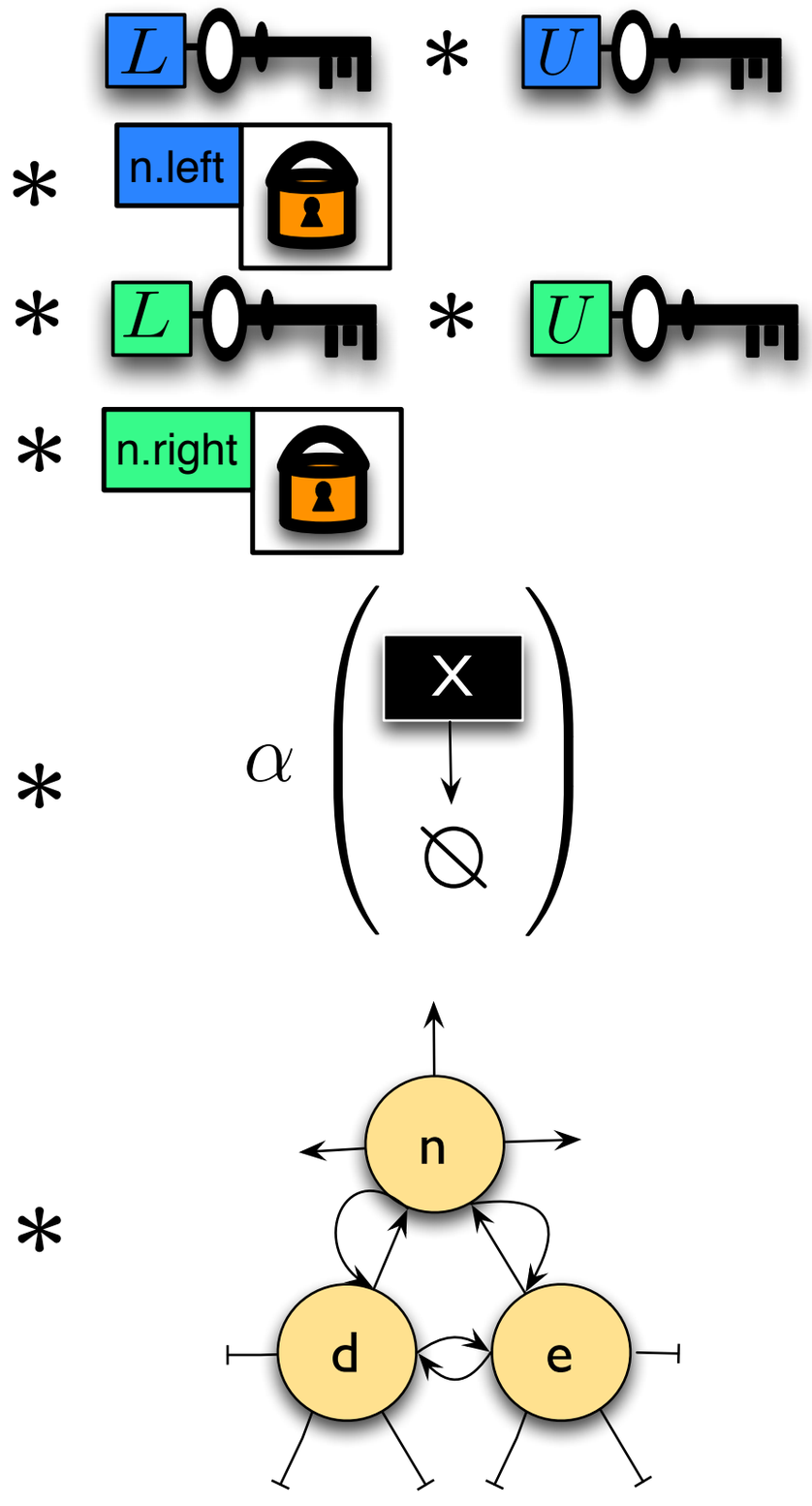


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

disposeTree(n)

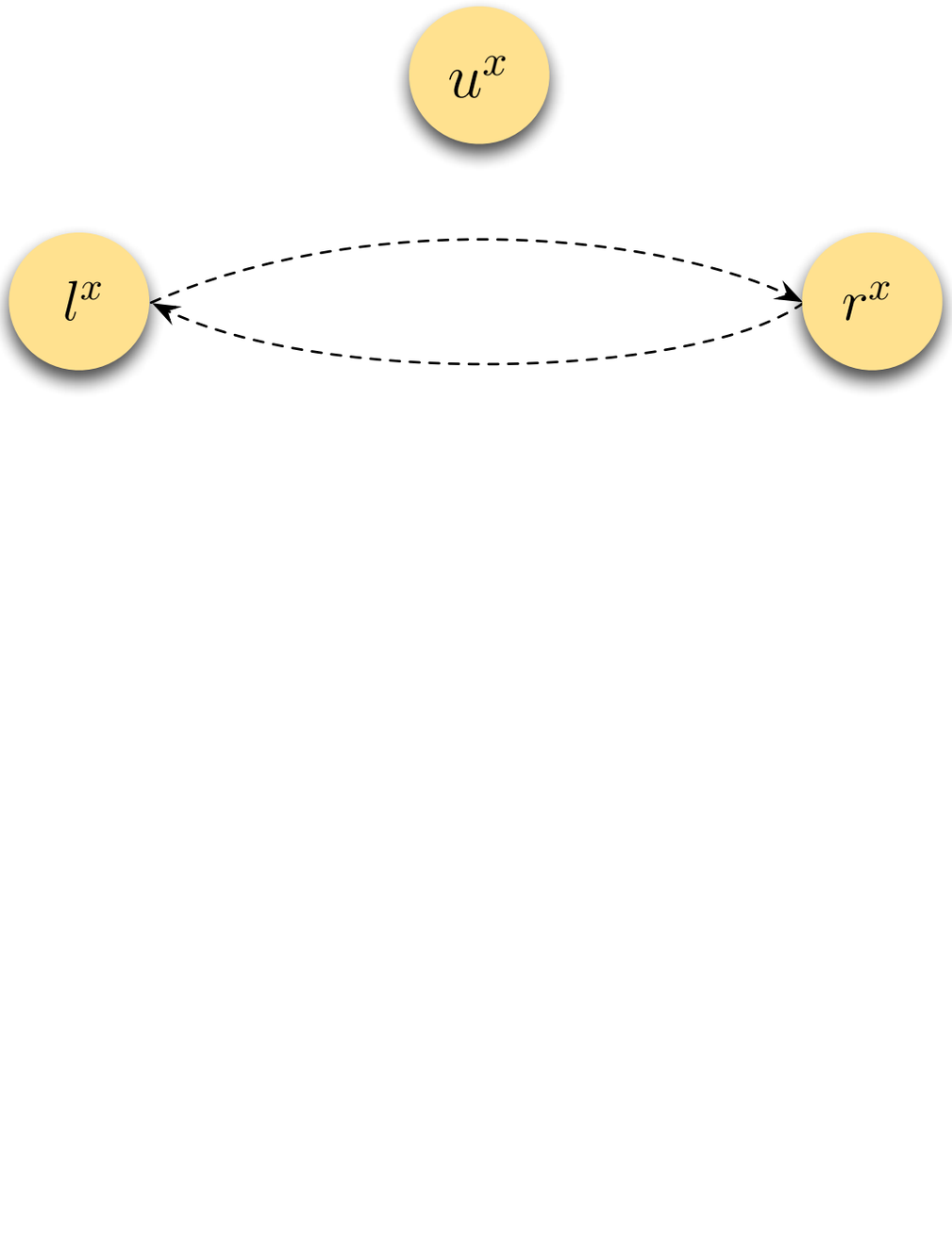


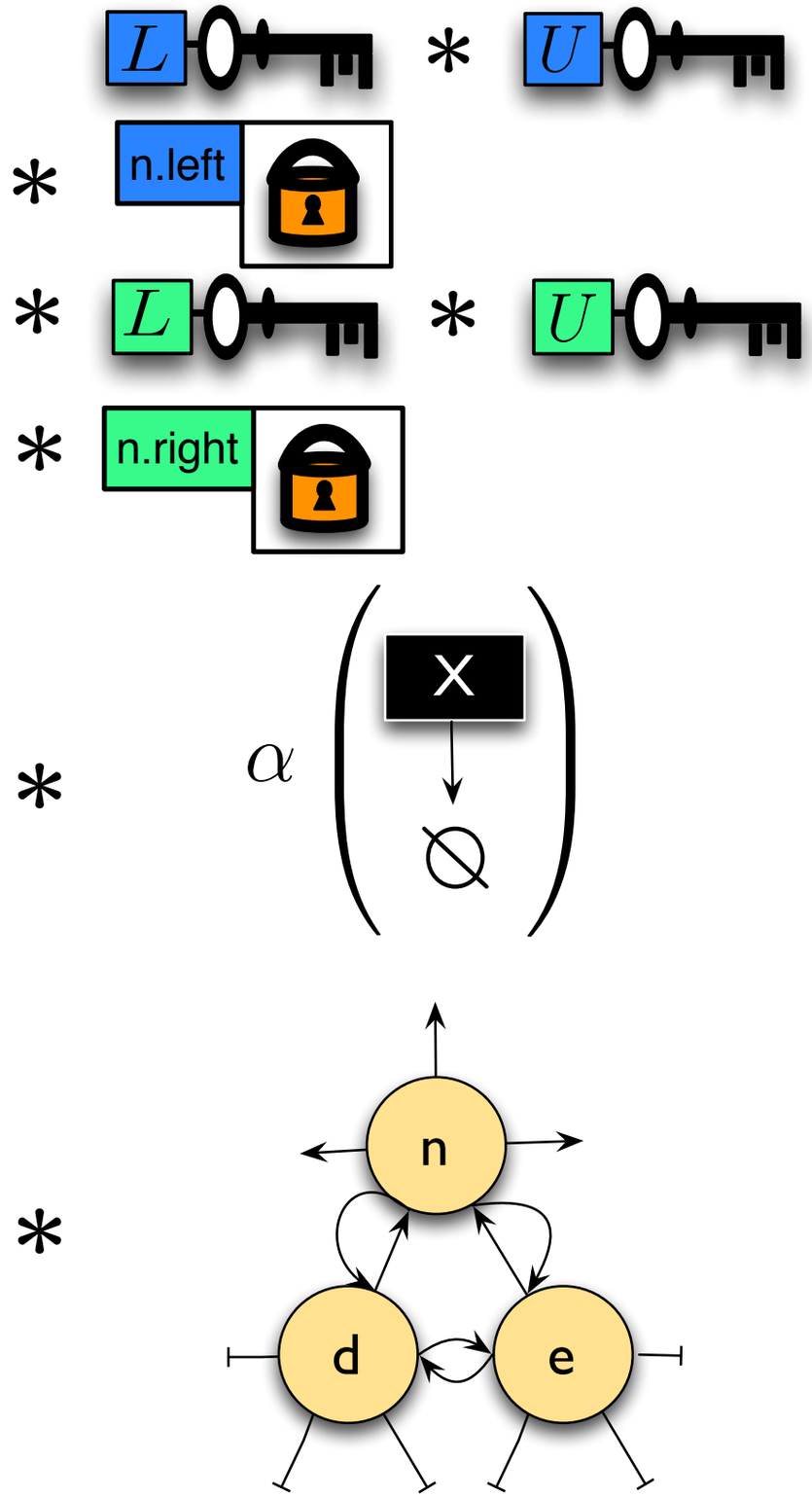


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

disposeTree(n)

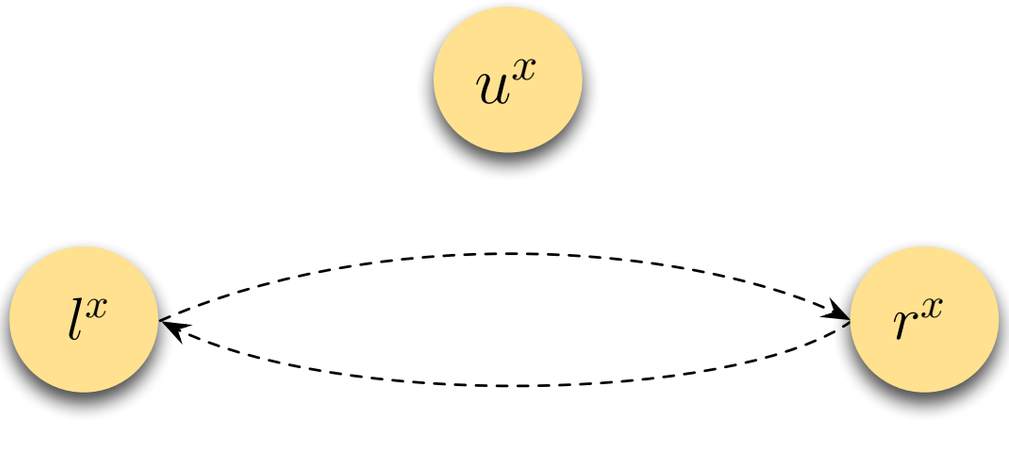


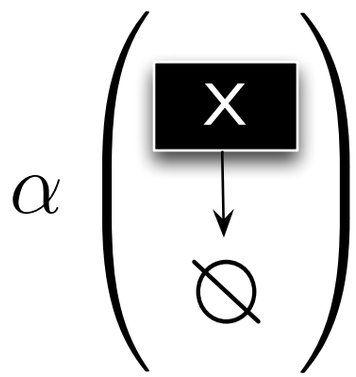


Refinement (Axiomatic Correctness)

```
proc deleteTree(n){
  local l,u,d,r,ul in
    u := [n.up] ; d := [n.first] ; ul:= [n.upL] ;
    //Acquiring the necessary locks.
    lock(ul) ;
    lock(n.leftL) ; l:= [n.left] ;
    if l ≠ null then lock(l.rightL)
    else if u ≠ null then lock(u.firstL) ;
    lock(n.rightL) ; r:= [n.right] ;
    if r ≠ null then lock(r.leftL) ;
    //Pointer Swinging.
    if l ≠ null then     [l.right] := r ;
    else if u ≠ null then [u.first] := r ;
    if r ≠ null then [r.left] := l ;
    else if u ≠ null then [u.last] := l ;
    //Unlocking the acquired locks.
    if l ≠ null then unlock(l.rightL) ;
    else if u ≠ null then unlock(u.firstL) ;
    if r ≠ null then unlock(r.leftL) ;
    else if u ≠ null then unlock(u.lastL) ;
    call disposeForest(d) ;
    disposeNode(n) ;
}
```

`disposeTree(n)`





Summary

- Structural Separation Logic
  - Compositional reasoning for concurrent data structures
  - Abstract Connectivity

- Webkit-based implementation
  - Bridging the gap (refinement via interface functions)
  - Application of CAP to a real-life example
    - Limitations of CAP
    - Ongoing work: CoLoSL (Concurrent Local Subjective Logic)

Thank you for listening.

Questions?