# Program Analysis for Overlaid Data Structures

Hongseok Yang
Queen Mary University of London

(Joint work with Oukseh Lee and Rasmus Petersen)

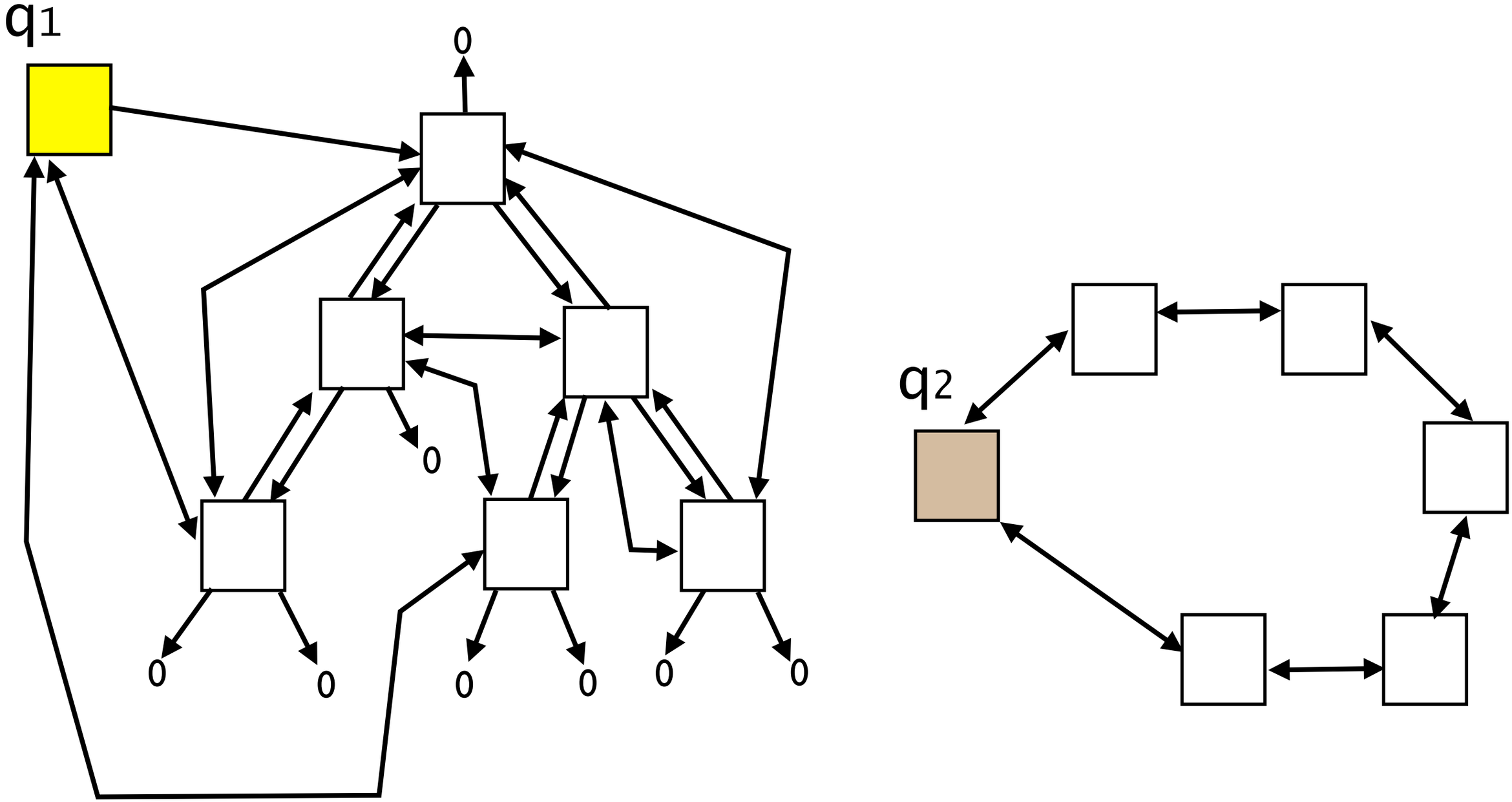
# Our goal

- Automatically verify deep heap properties (e.g., no memory leak, and preservation of shape invariant) of real-world systems code, such as Linux.
- Great progress in the past 5 years.
- Challenges:
  - Highly-shared data structures.
  - Concurrency.
  - ...

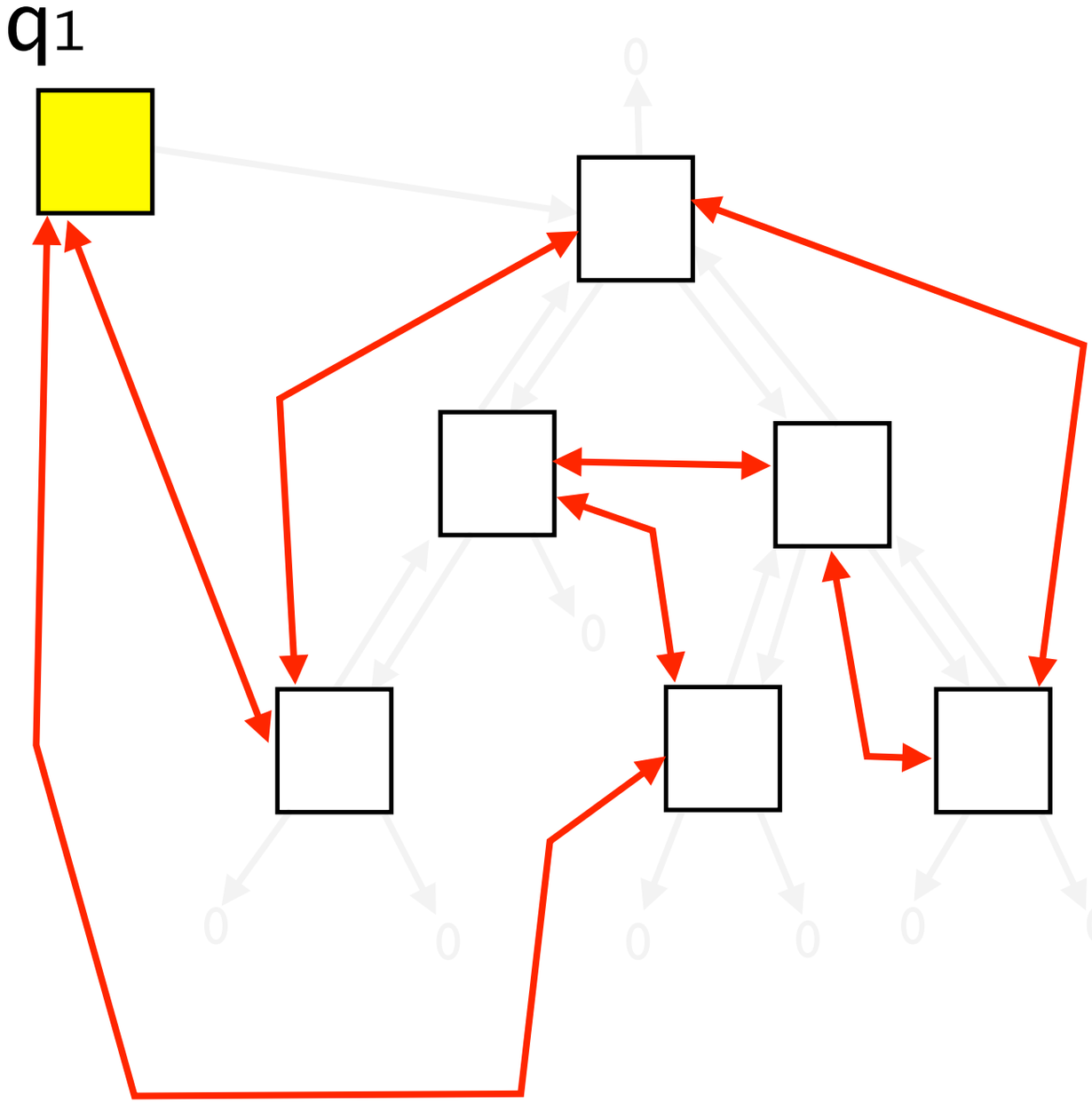
# Our goal

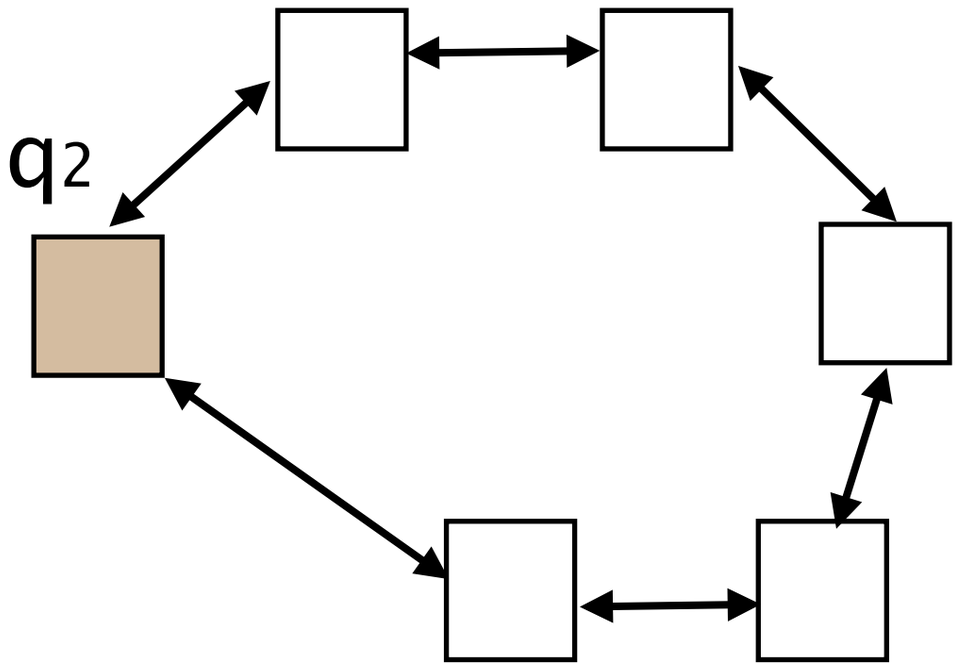
- Automatically verify deep heap properties (e.g., no memory leak, and preservation of shape invariant) of real-world systems code, such as Linux.
- Great progress in the past 5 years.
- Challenges:
  - Highly-shared data structures.
  - Concurrency.
  - ...

# Deadline IO scheduler

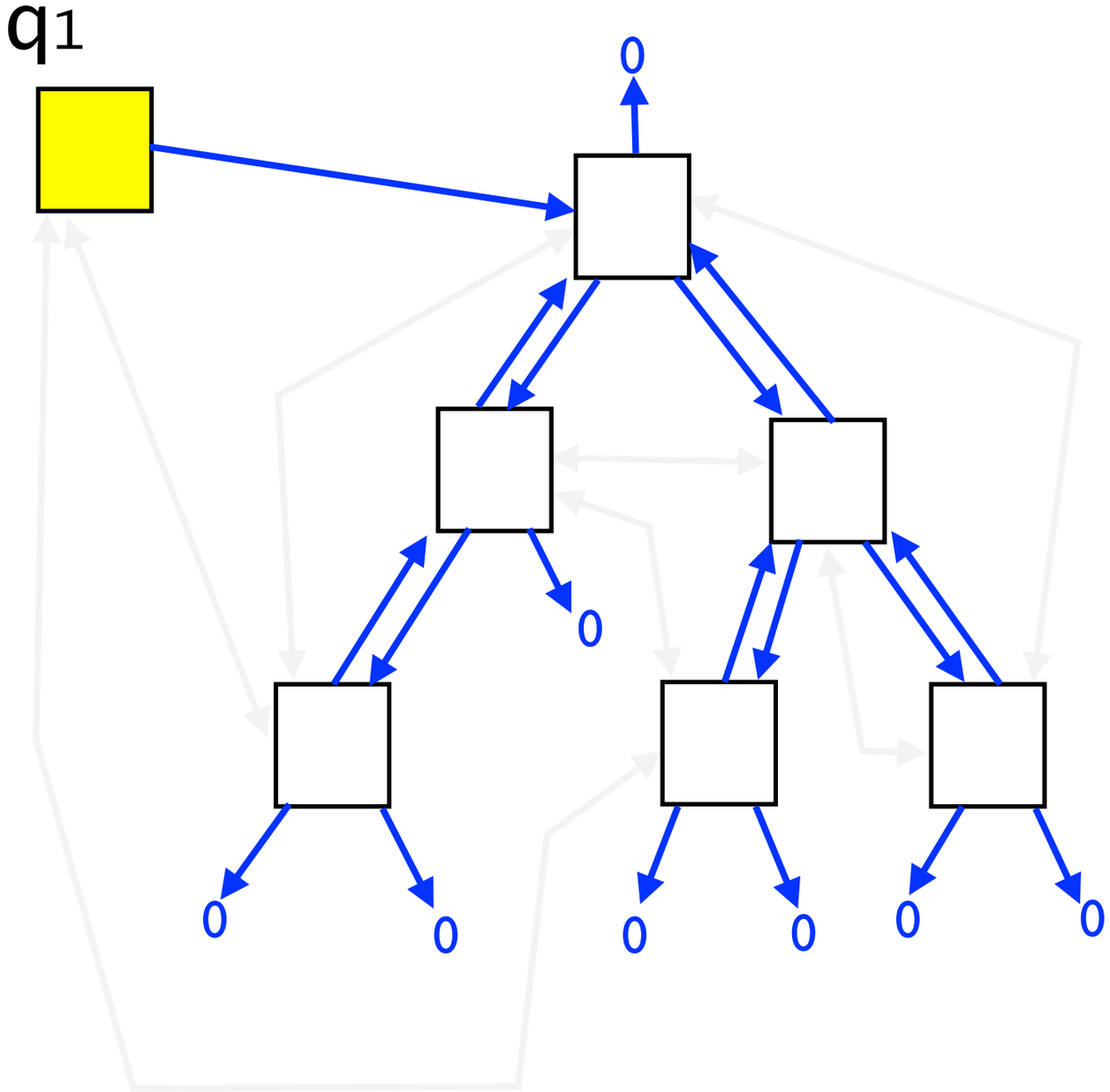


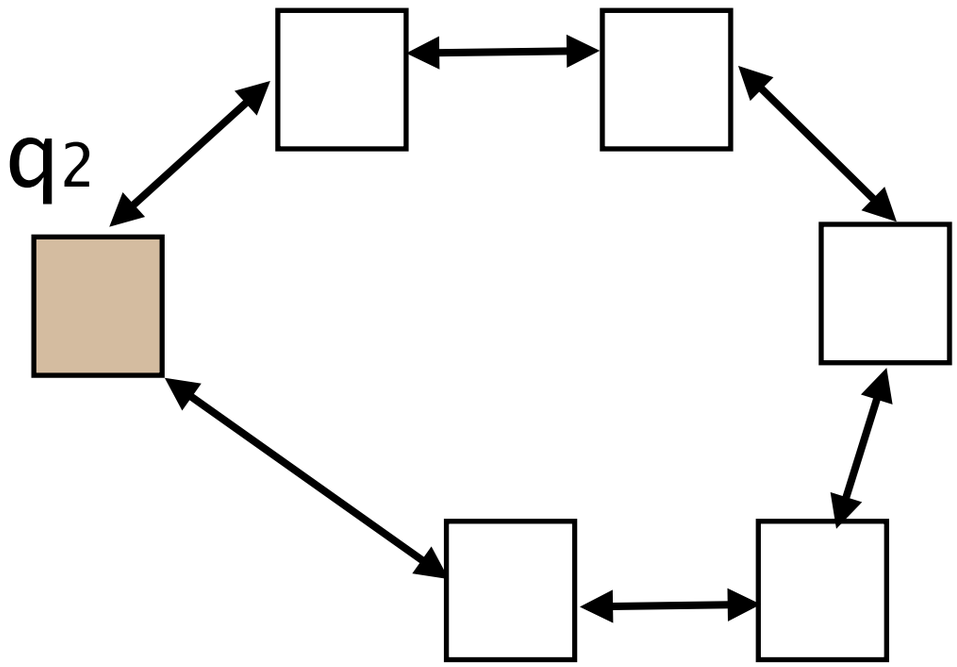
# Deadline IO scheduler



# Deadline IO scheduler

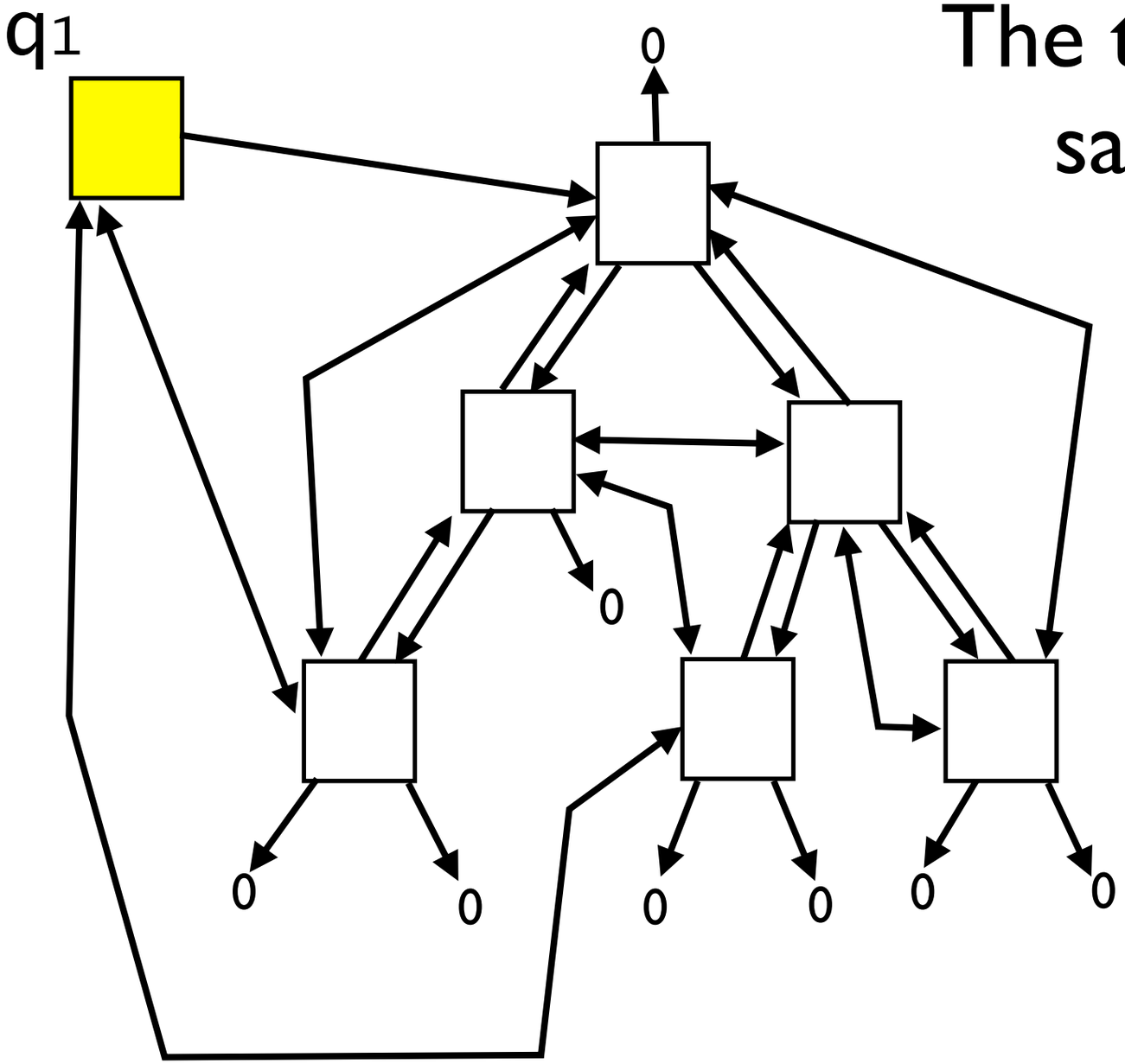


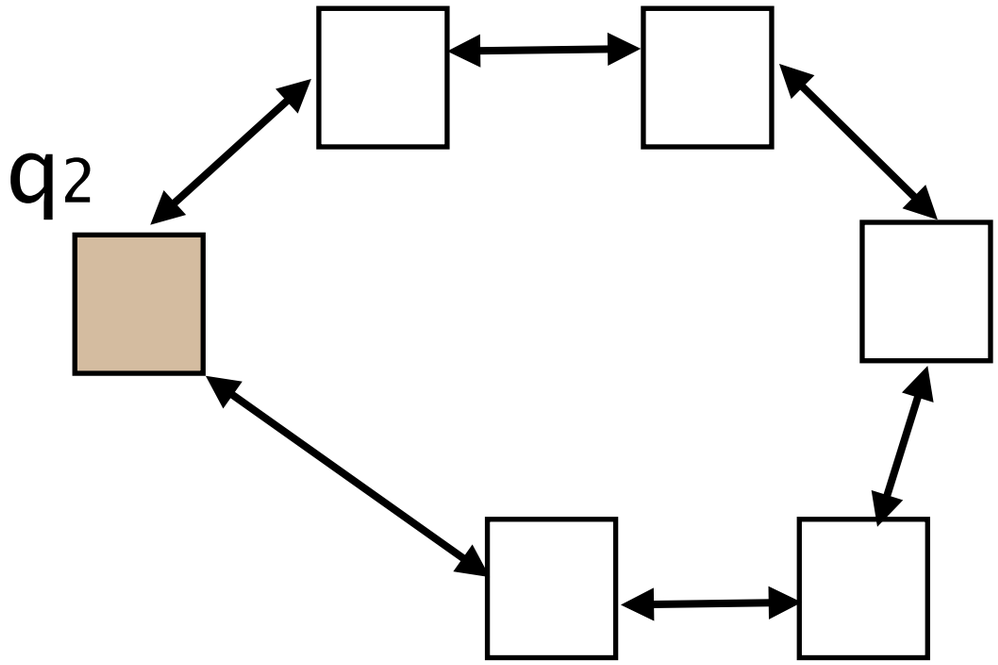
# Overlaid data structure

- One or more data structures on top of another.
- Frequently found in Linux.
- Components of an overlaid data structure (e.g., tree and list in the IO scheduler) are loosely correlated.
- The only important correlation is that the components talk about the same set of heap cells.

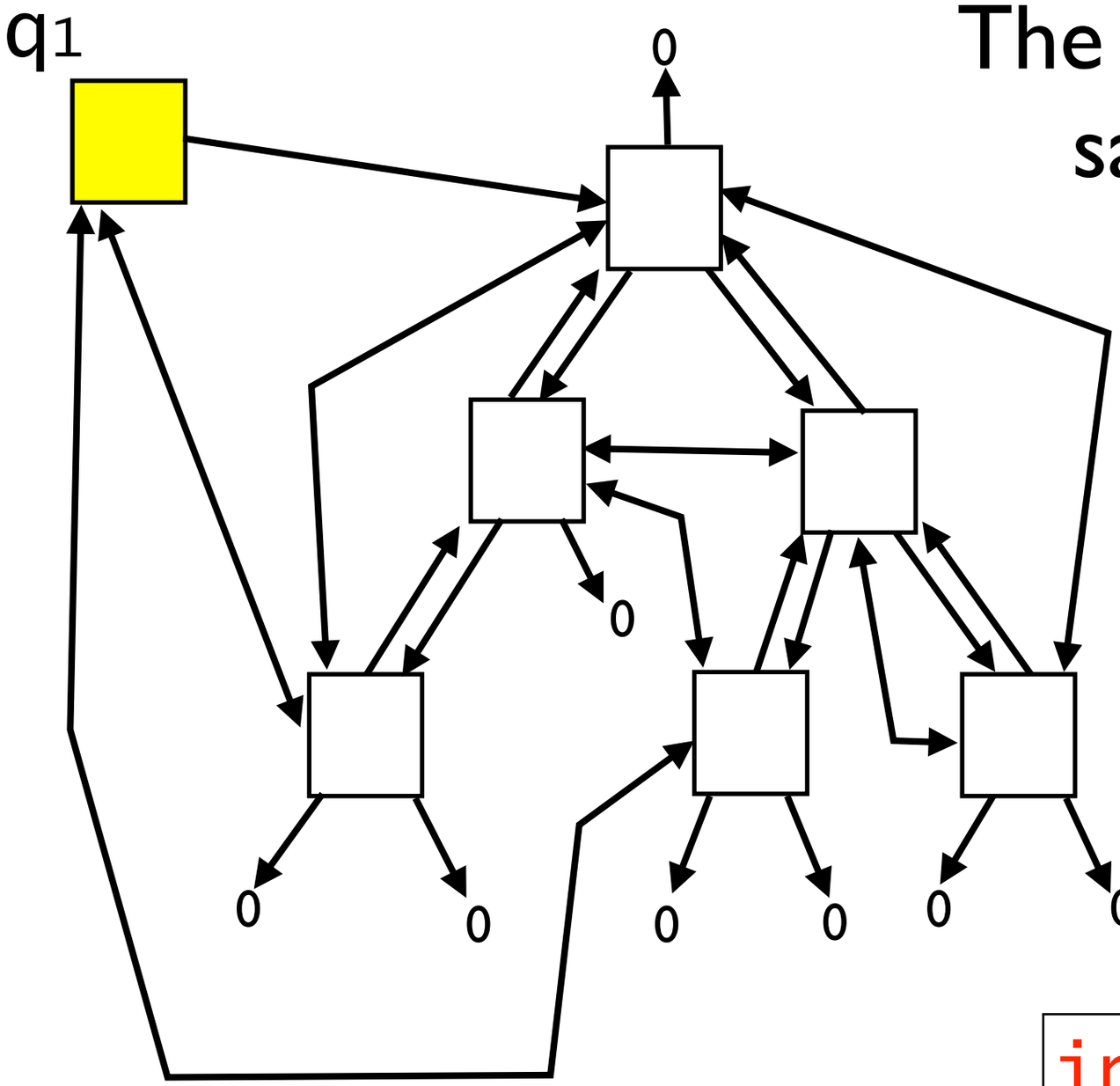
# Deadline IO scheduler



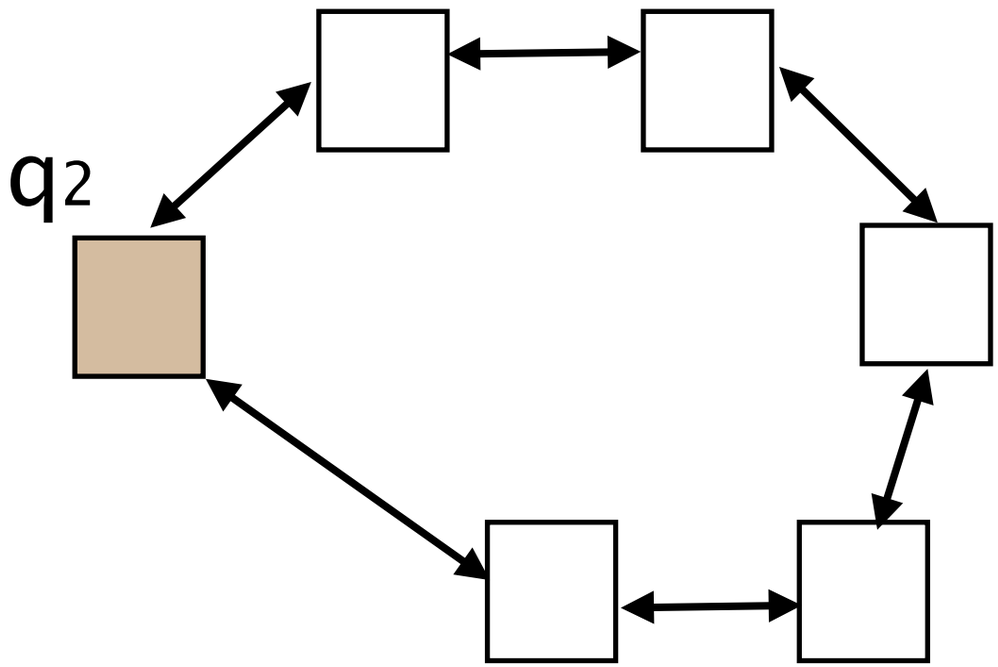

The tree and the list use the same set of heap cells.

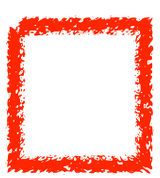


# Insert a request



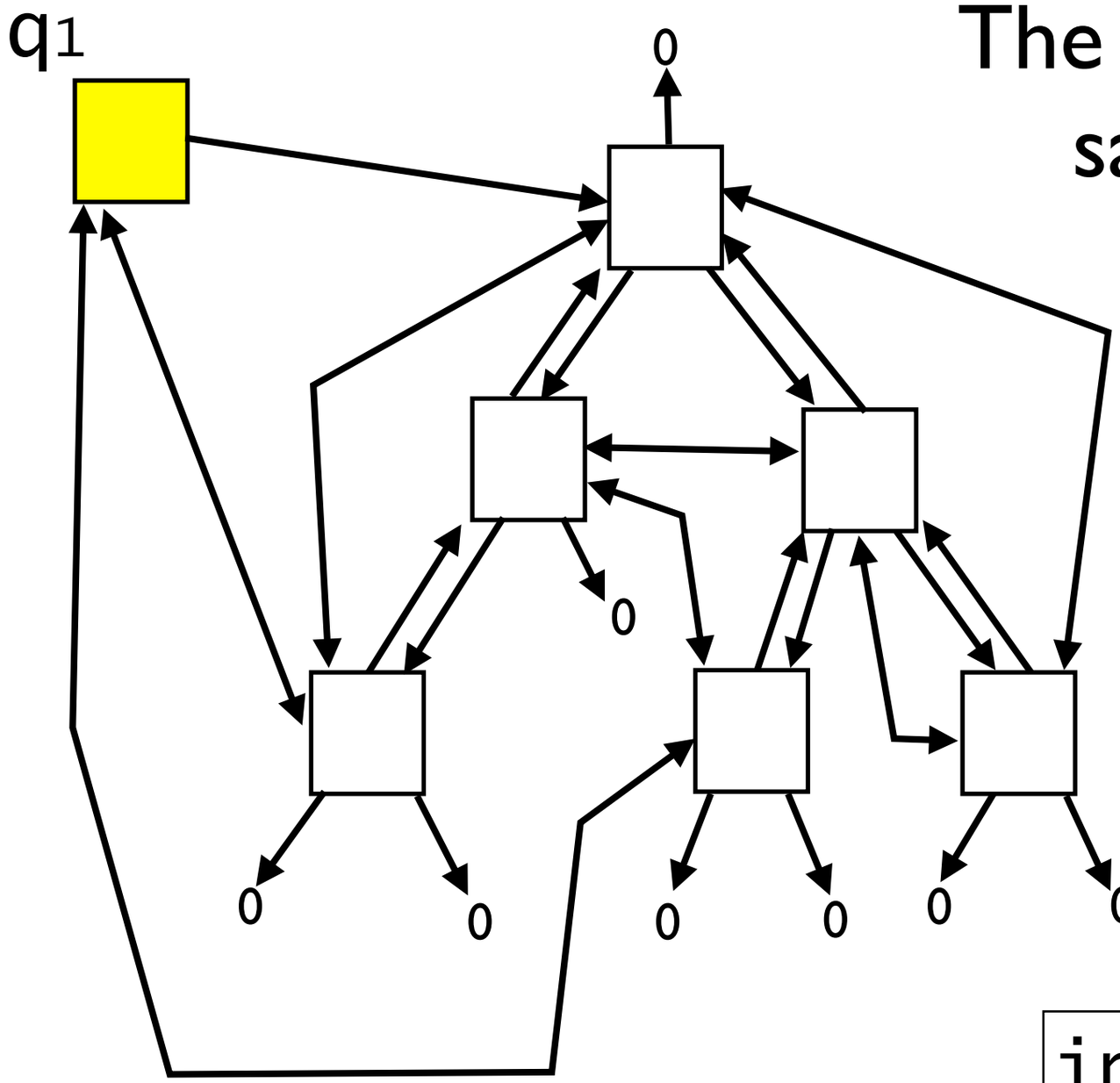

The tree and the list use the same set of heap cells.



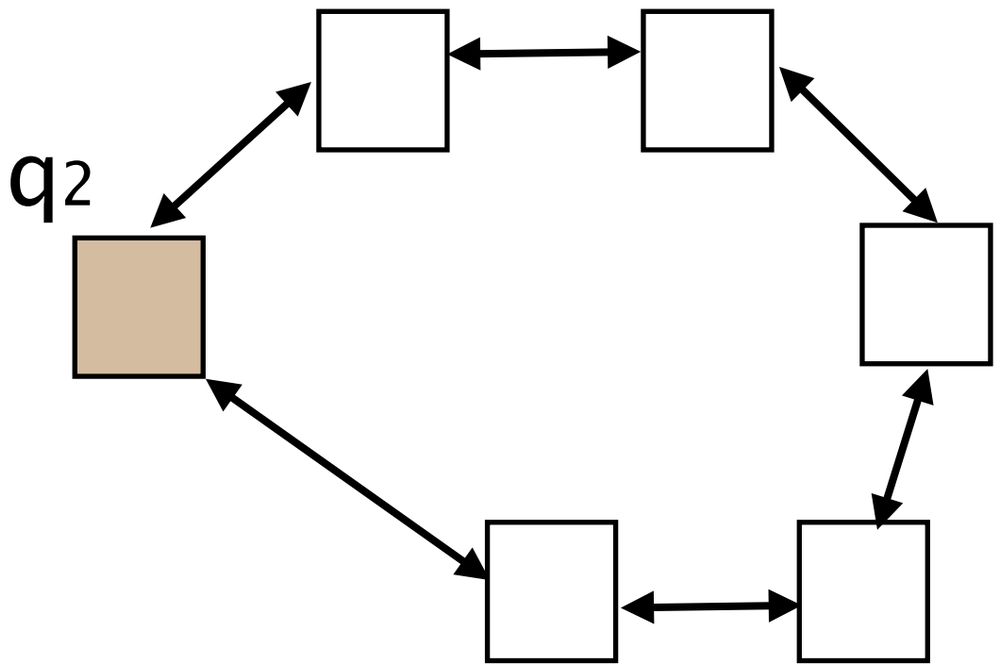




```
insert(q1,rq) {
    list_insert(q1,rq);
    tree_insert(q1,rq); }
```

# Insert a request

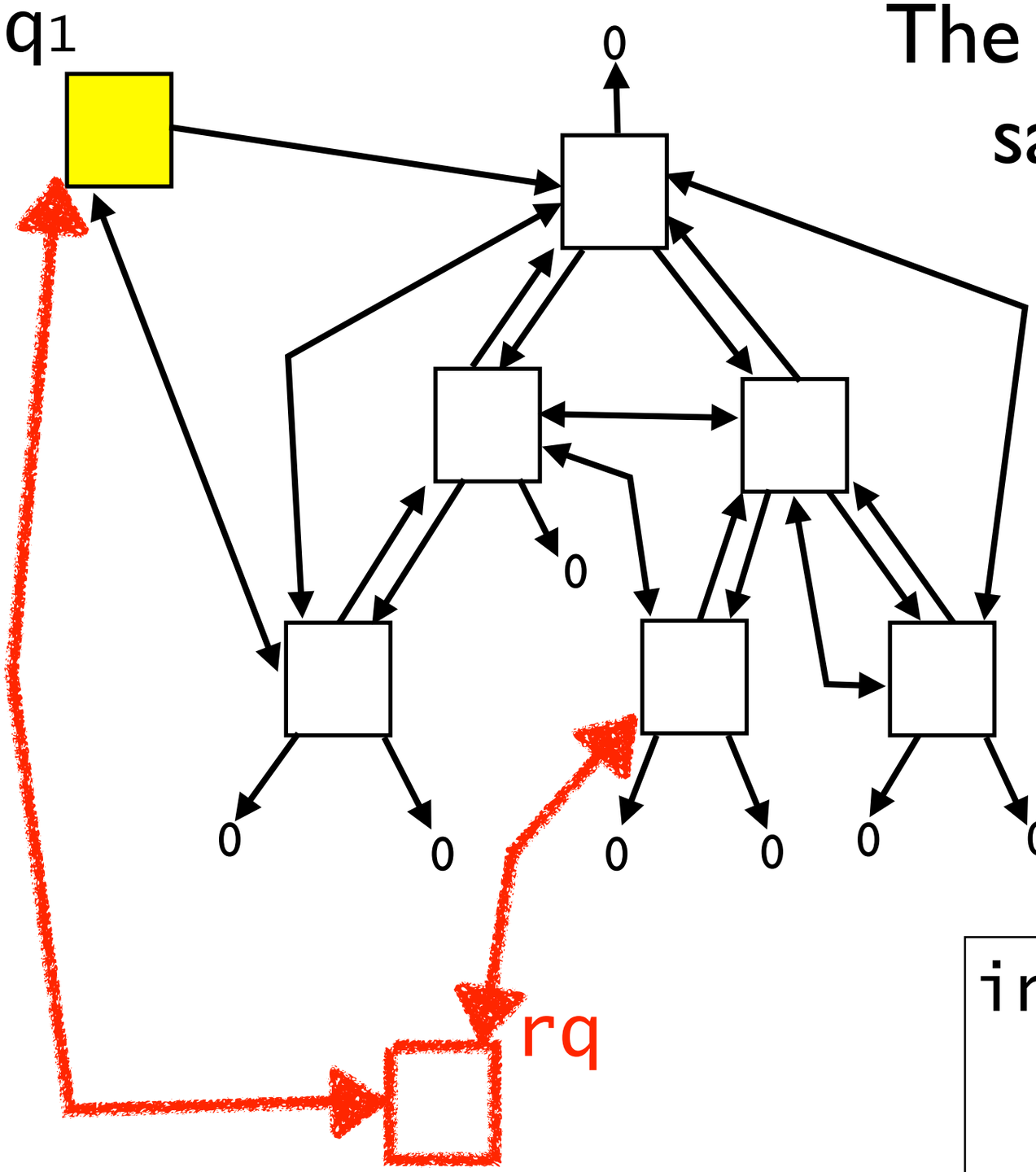



The tree and the list use the same set of heap cells.

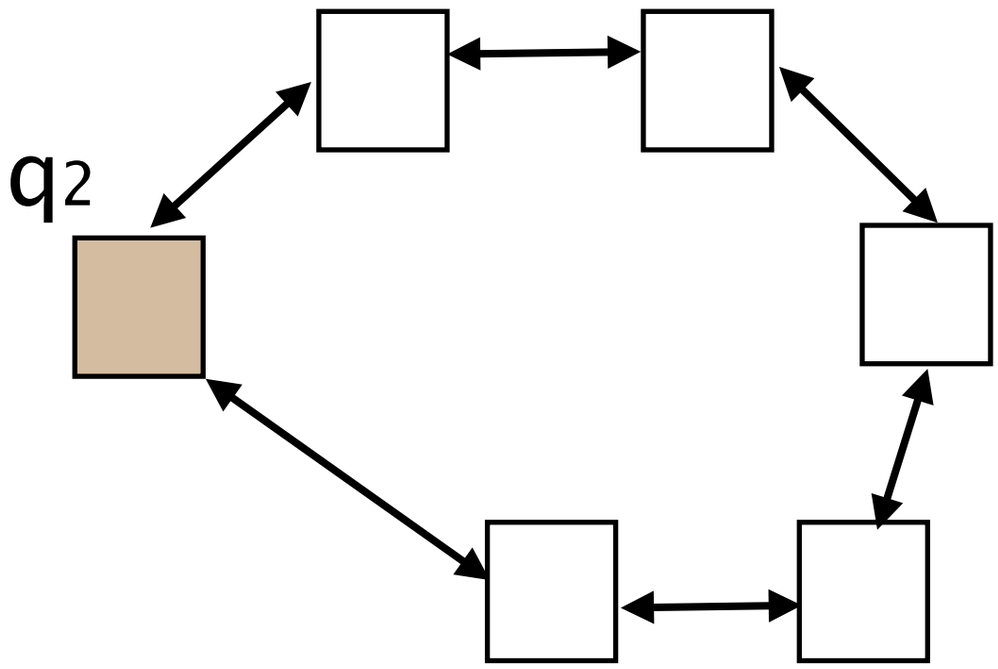



```
insert(q1,rq) {
    list_insert(q1,rq);
    tree_insert(q1,rq); }
```

# Insert a request

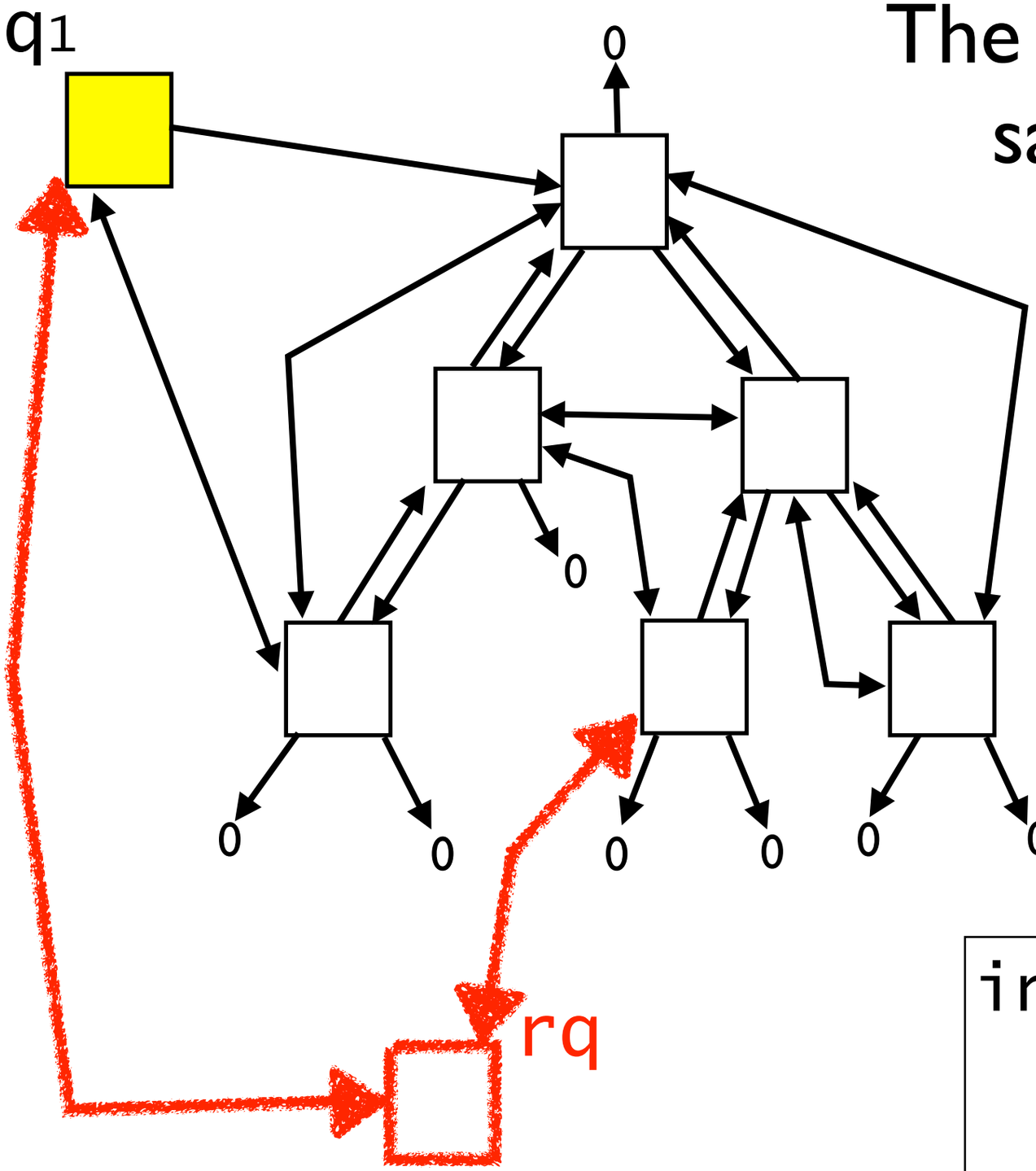



The tree and the list use the same set of heap cells.

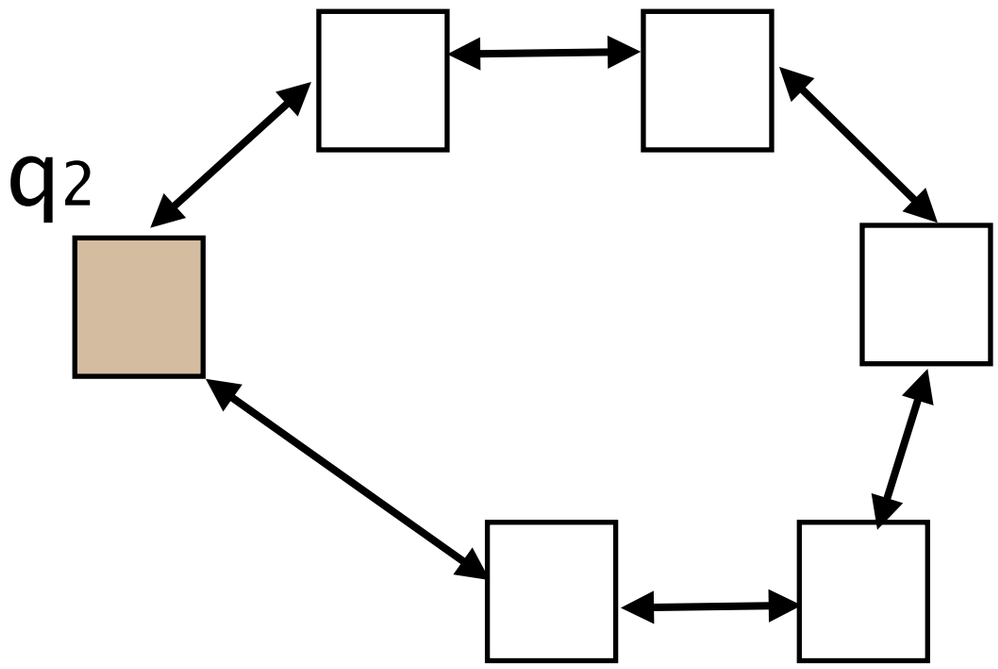



```
insert(q1,rq) {
    list_insert(q1,rq);
    tree_insert(q1,rq); }
```

# Insert a request

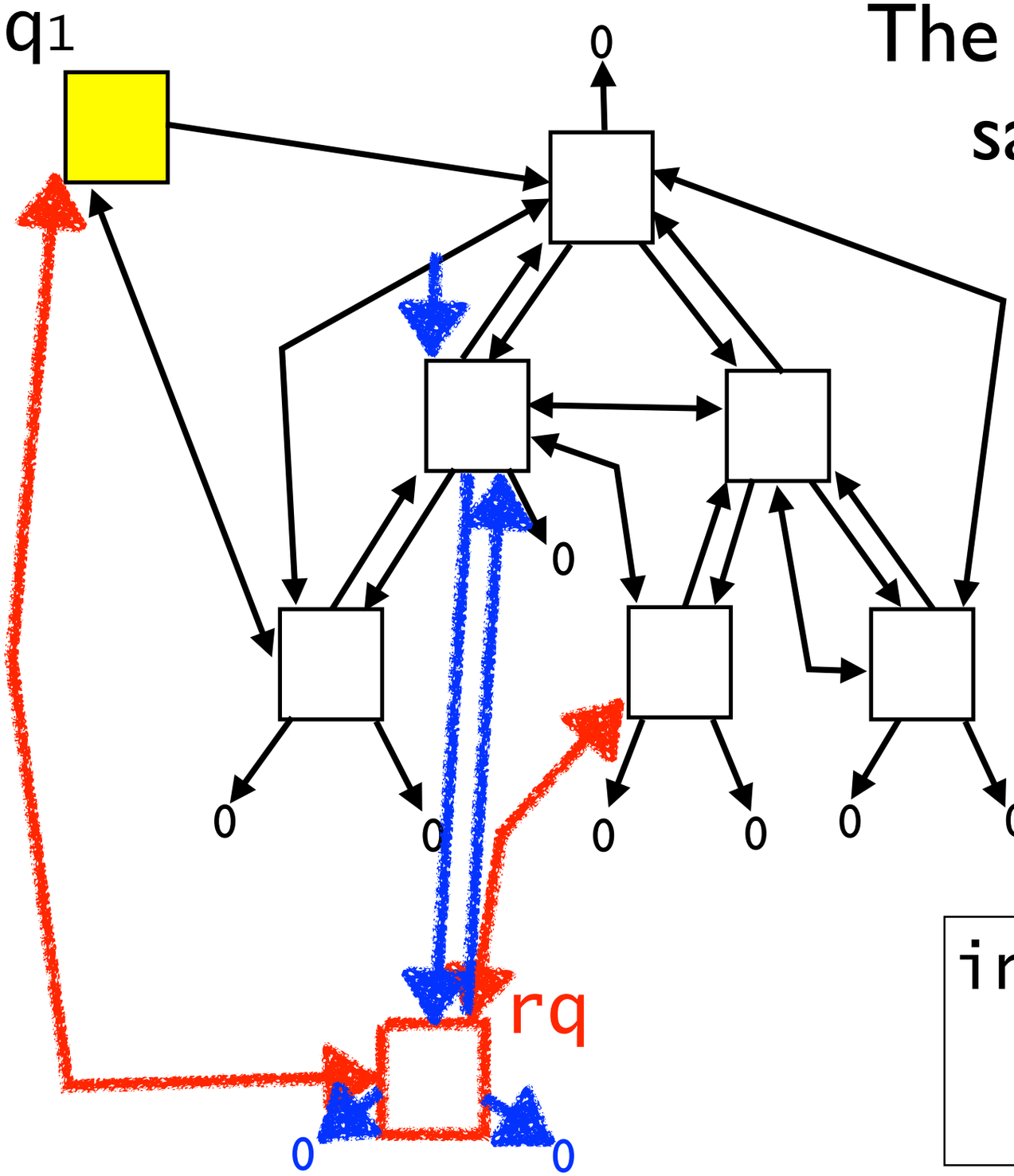



The tree and the list use the same set of heap cells.

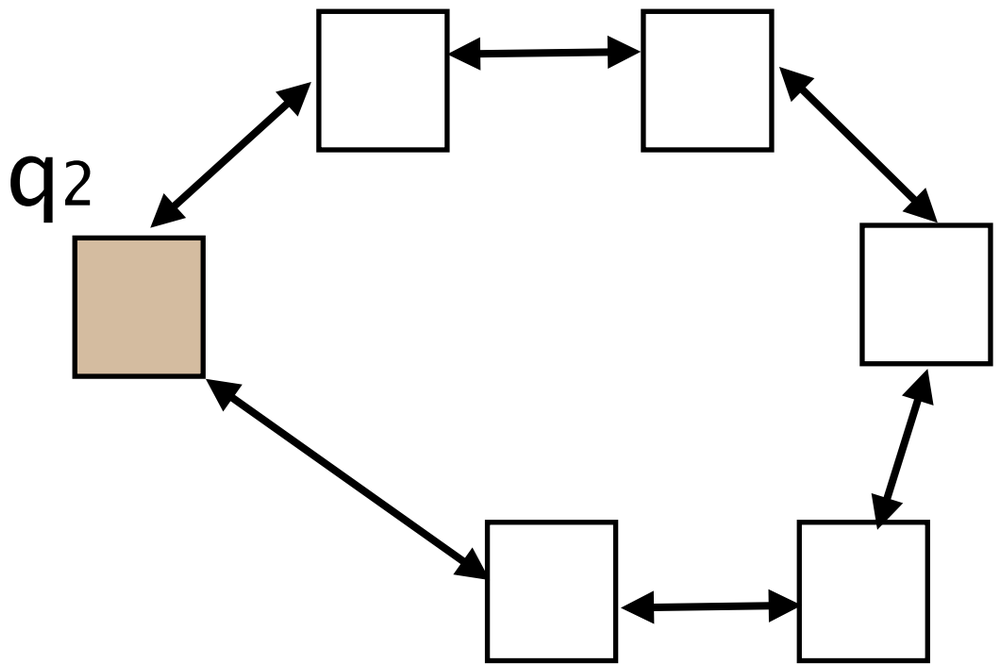



```
insert(q1,rq) {
    list_insert(q1,rq);
    tree_insert(q1,rq); }
```

# Insert a request

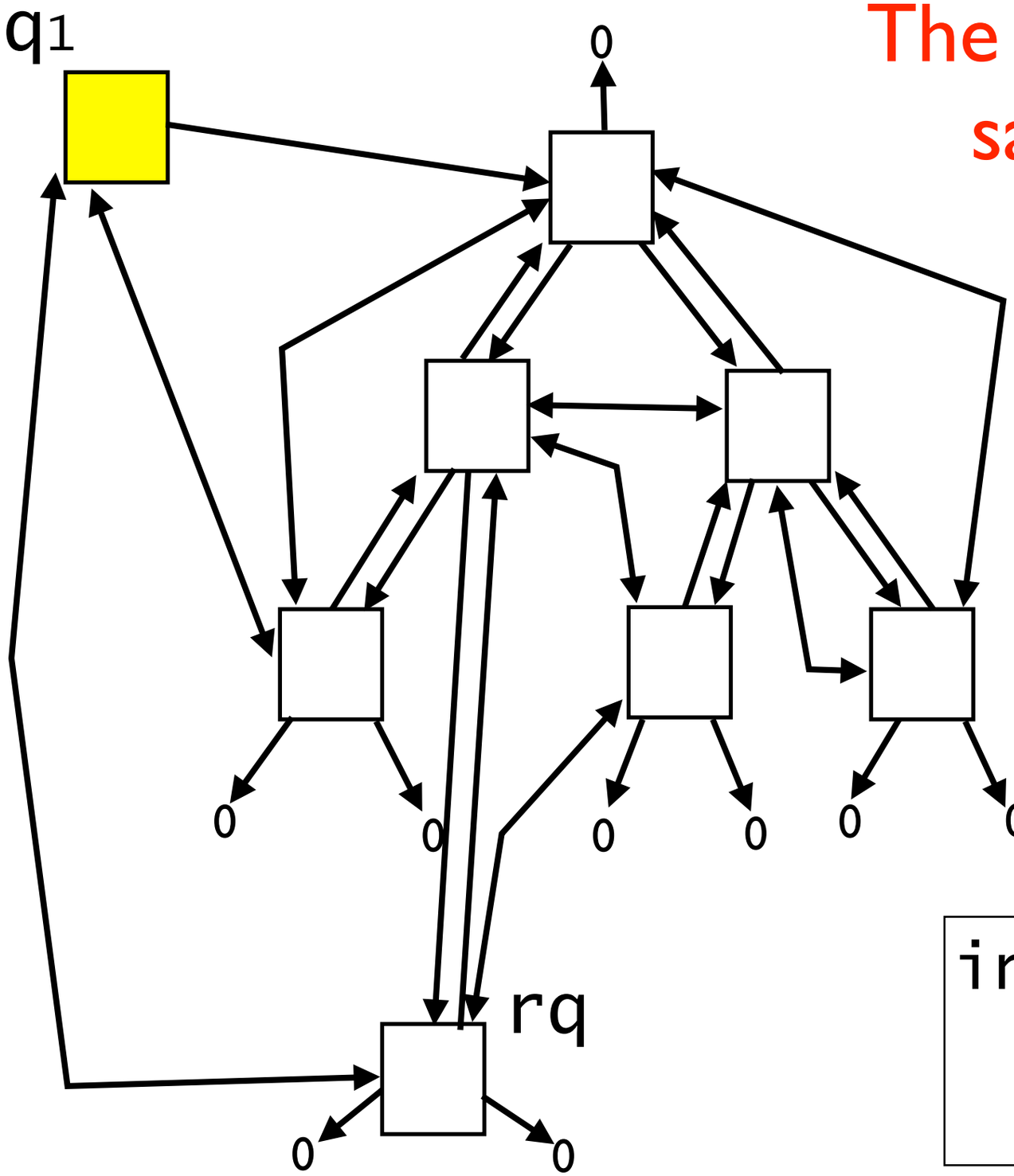



The tree and the list use the same set of heap cells.

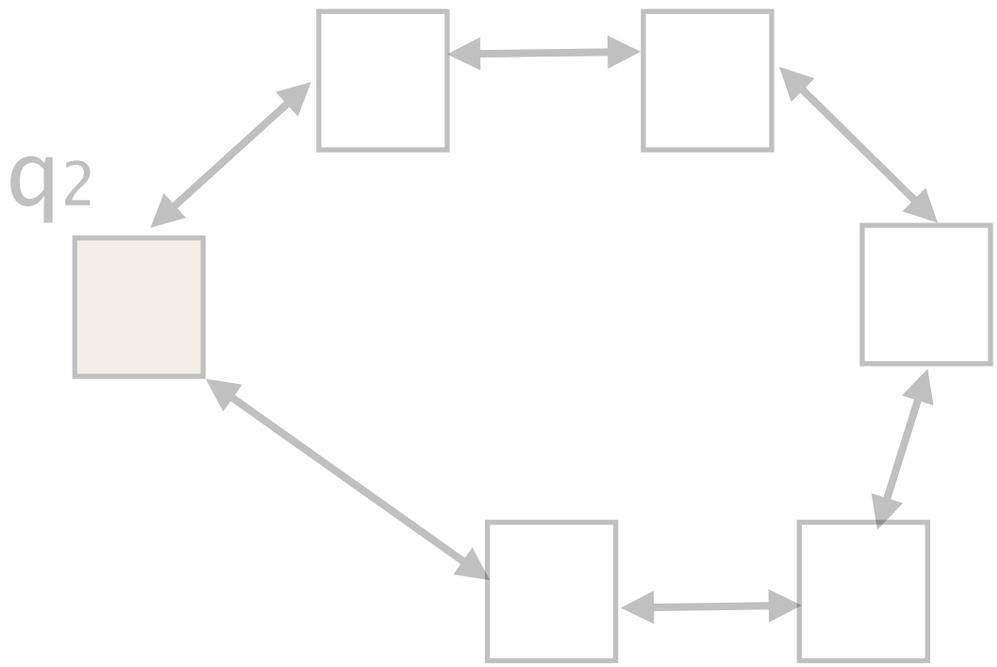



```
insert(q1,rq) {
    list_insert(q1,rq);
    tree_insert(q1,rq); }
```

# Insert a request

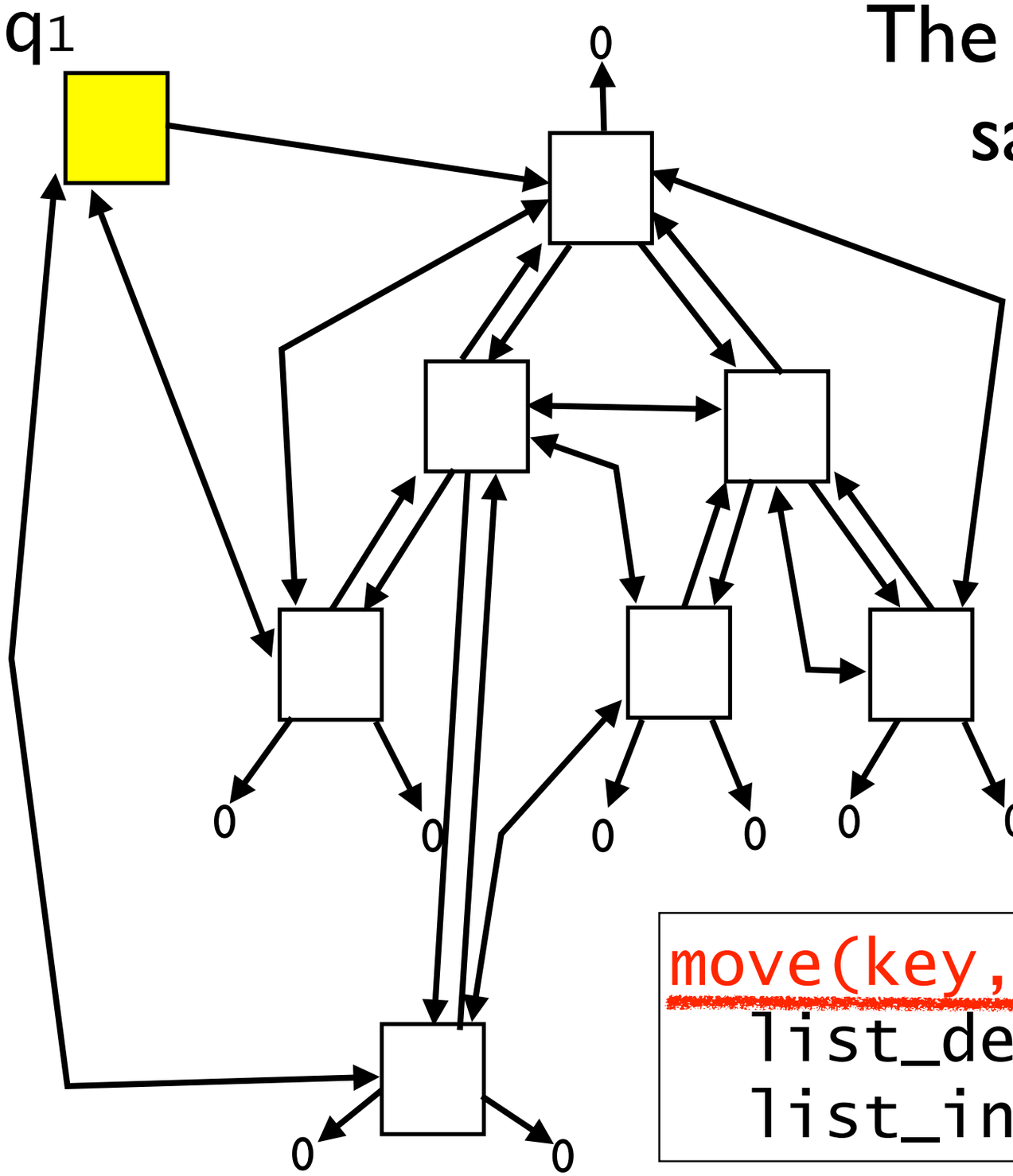



The tree and the list use the same set of heap cells.

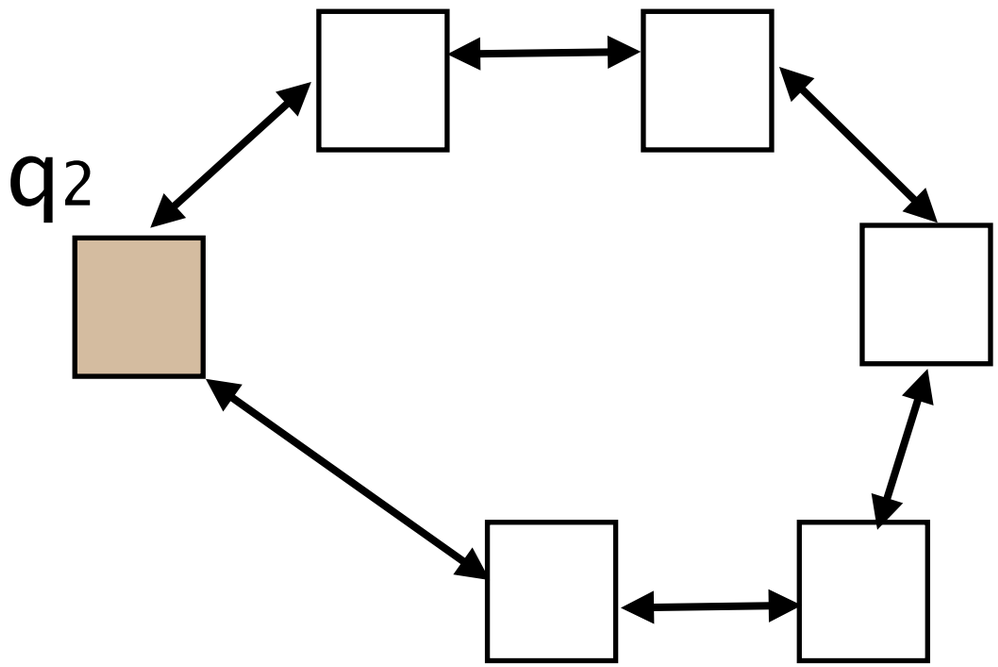



```
insert(q1,rq) {
   list_insert(q1,rq);
   tree_insert(q1,rq); }
```

# Move a request



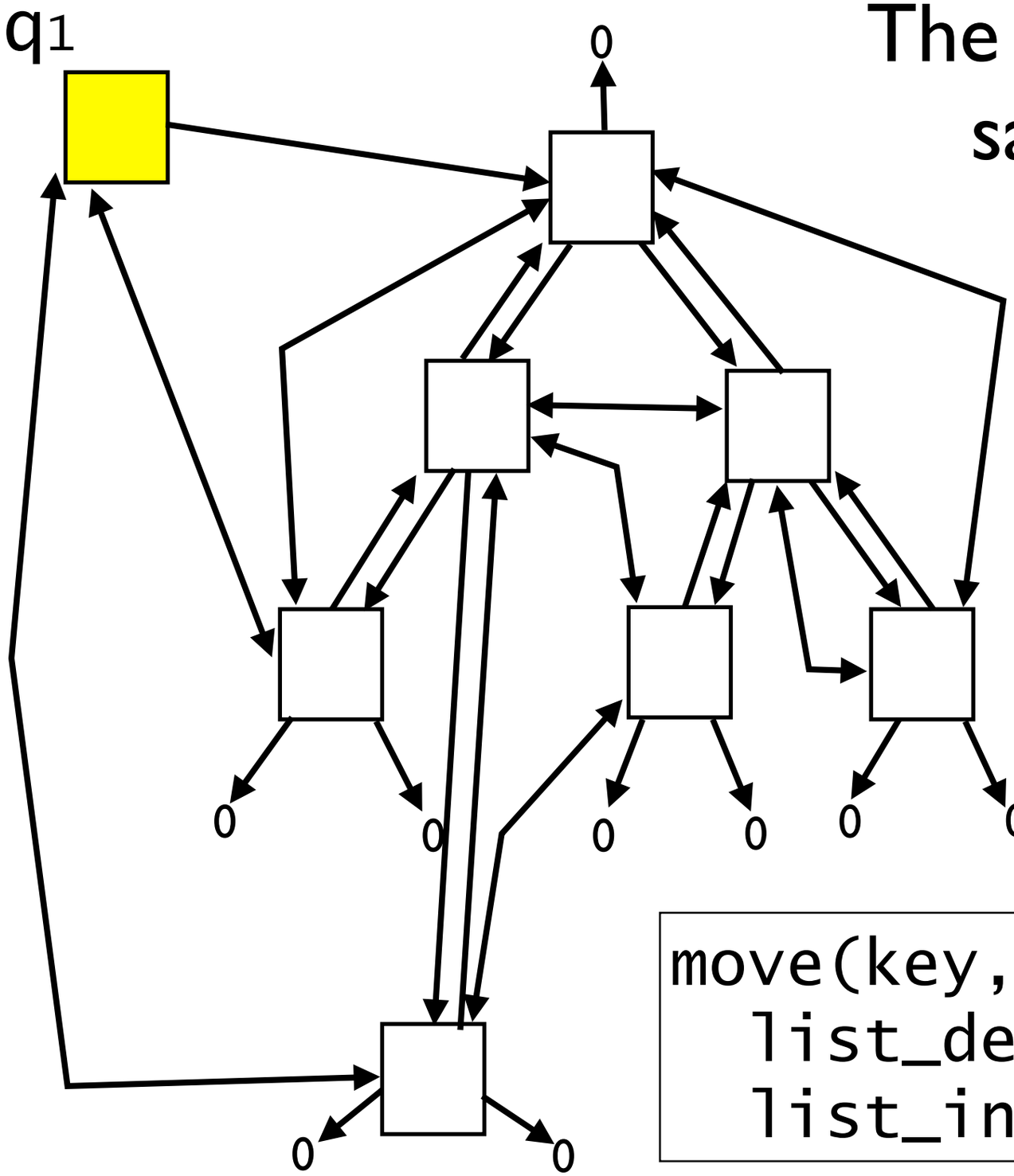

The tree and the list use the same set of heap cells.

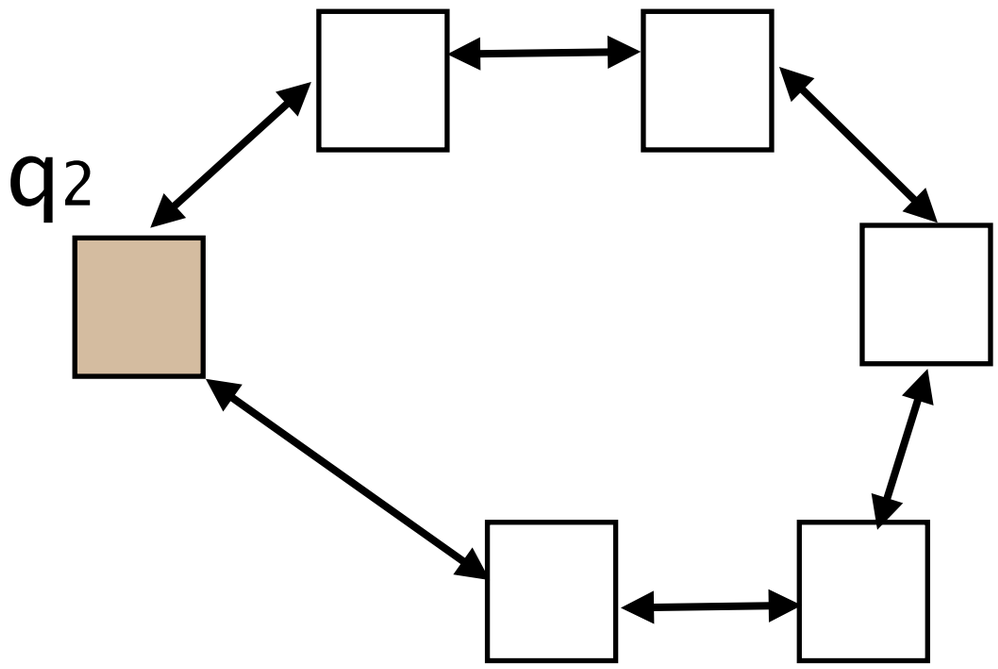



```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.

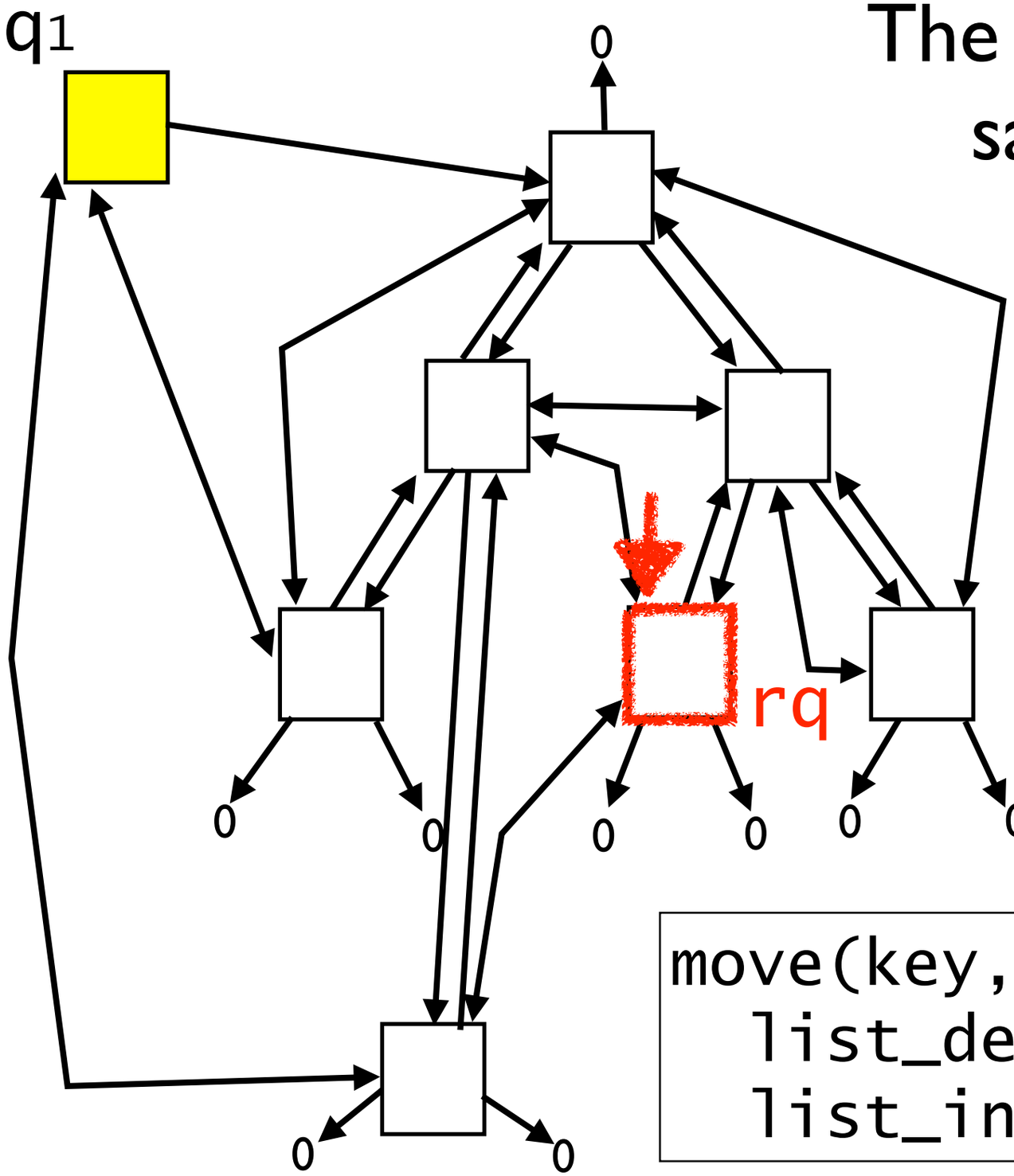


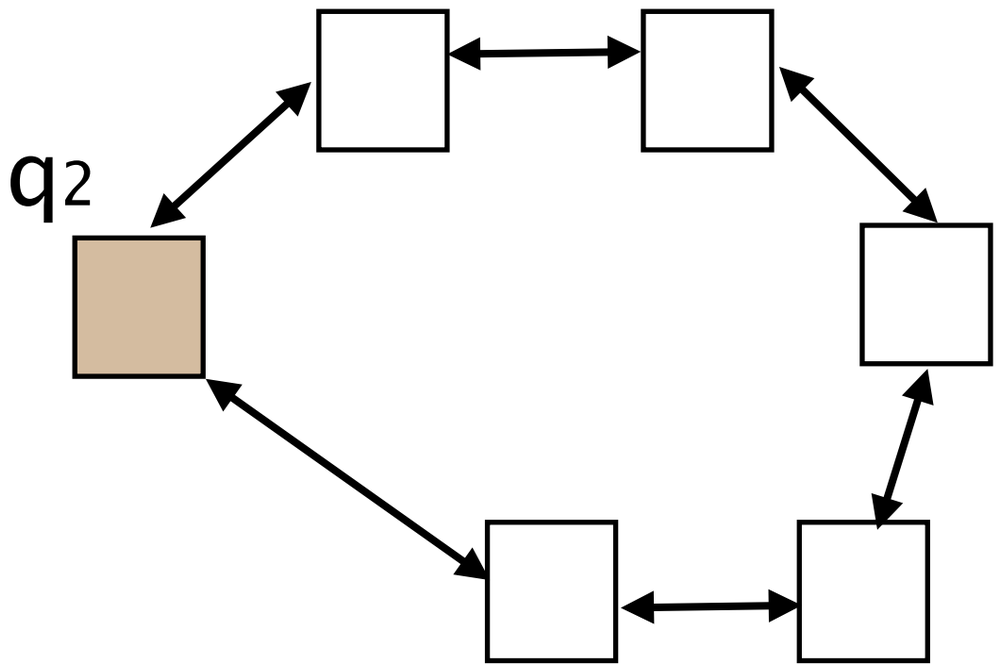




```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.

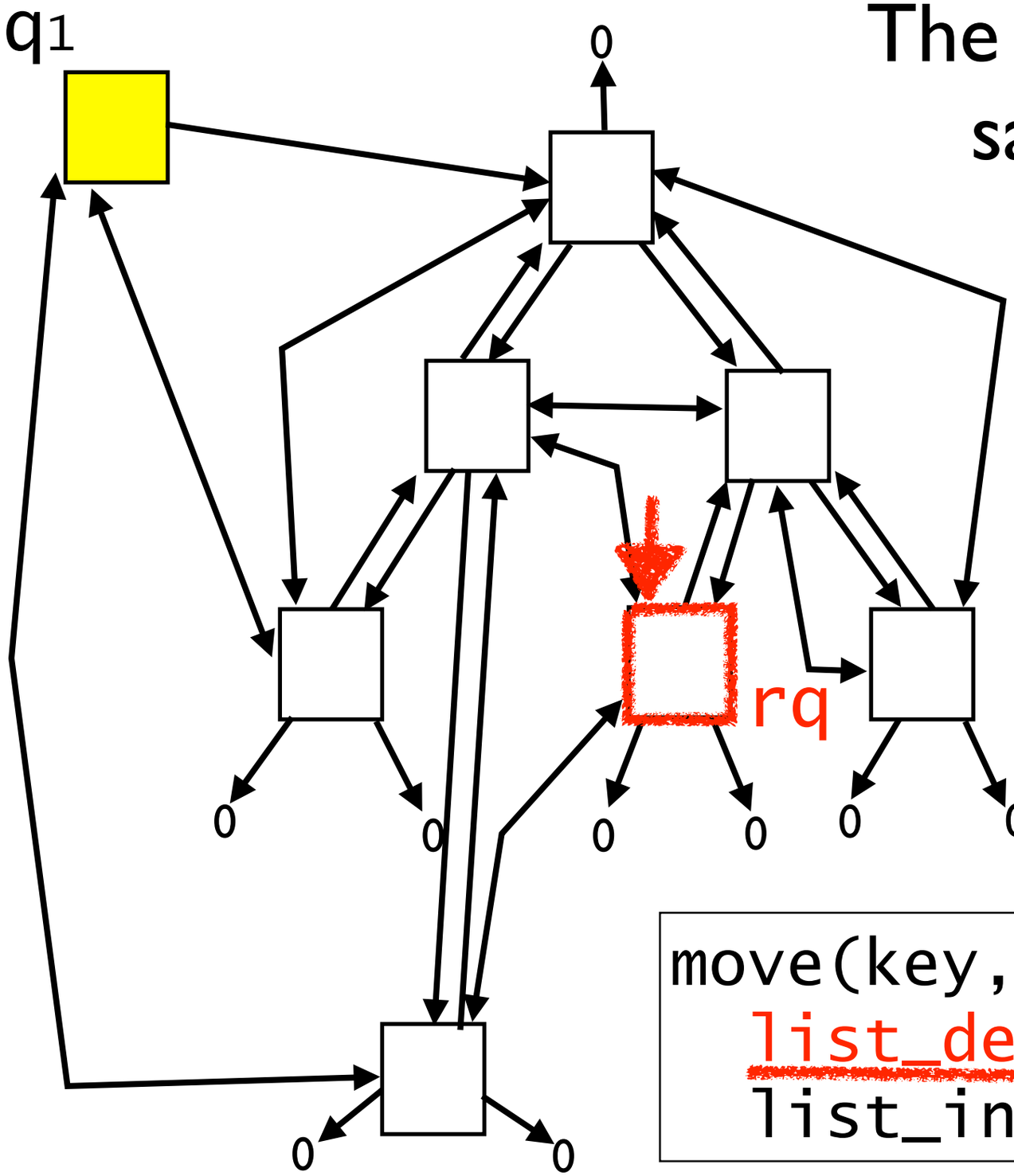


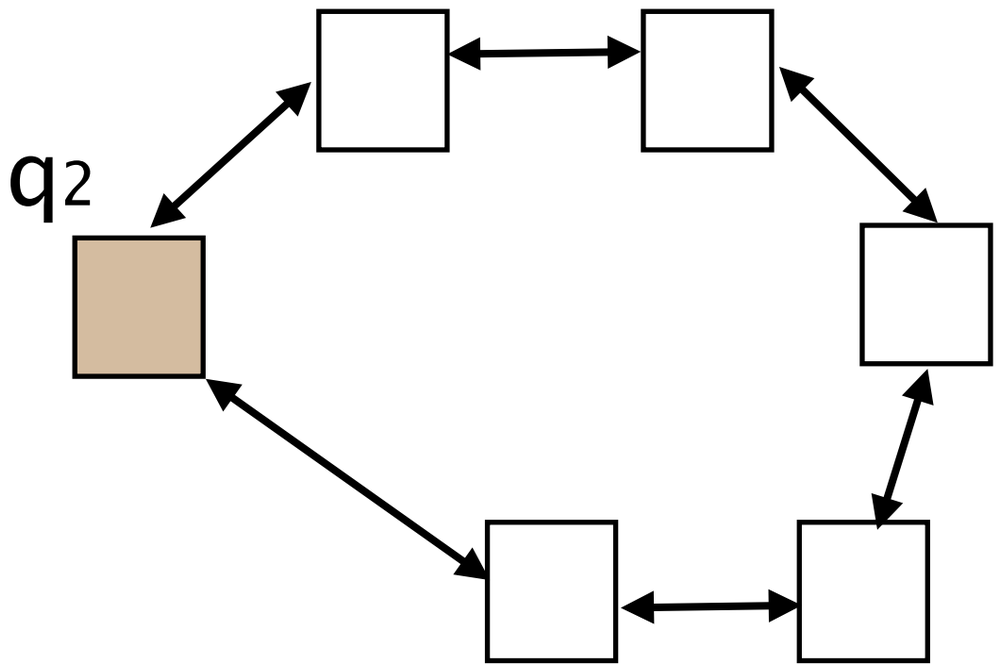




```
move(key,q1,q2) { rq=find(q1,key);
   list_del(q1,rq);tree_del(q1,rq);
   list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.

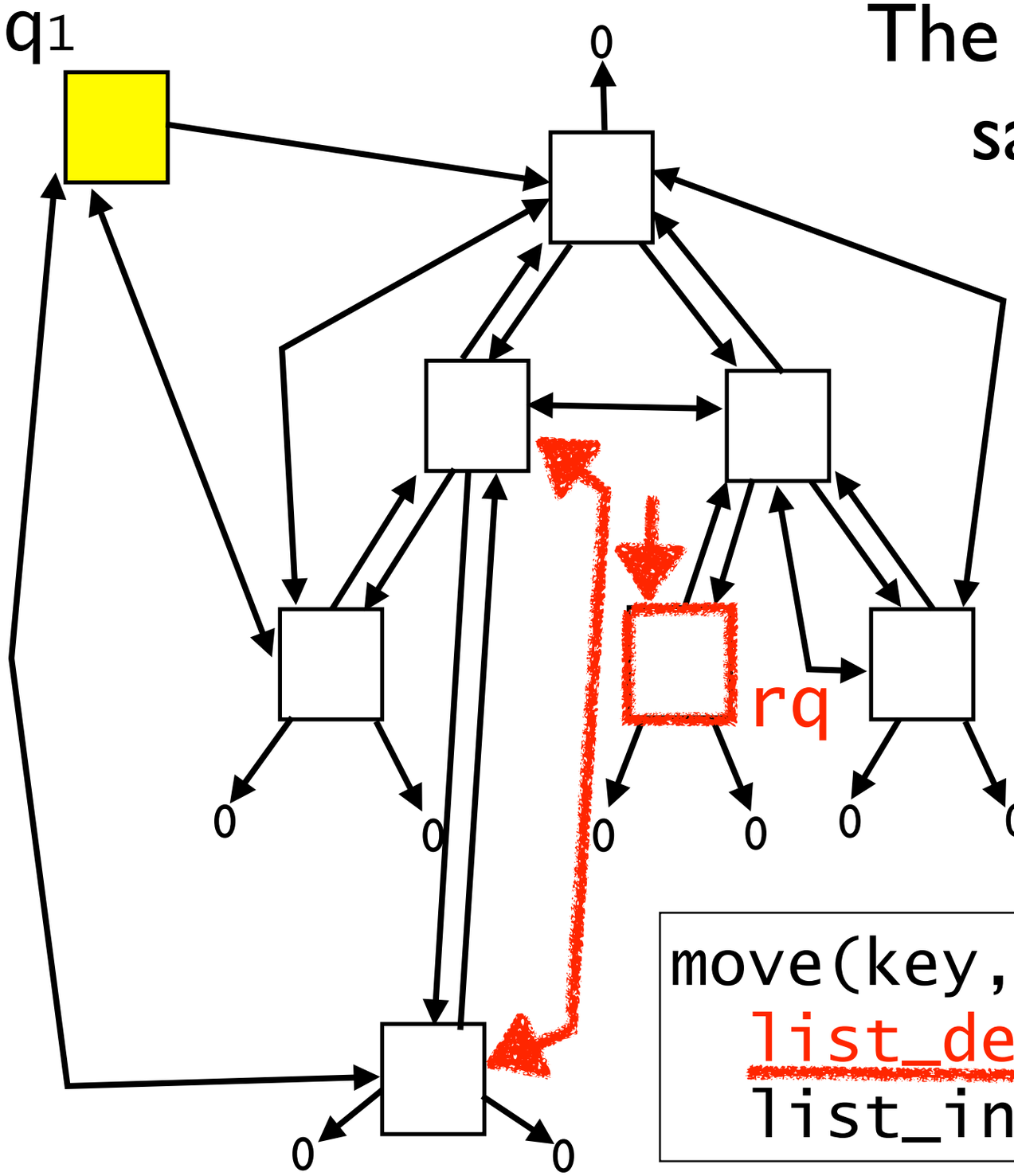


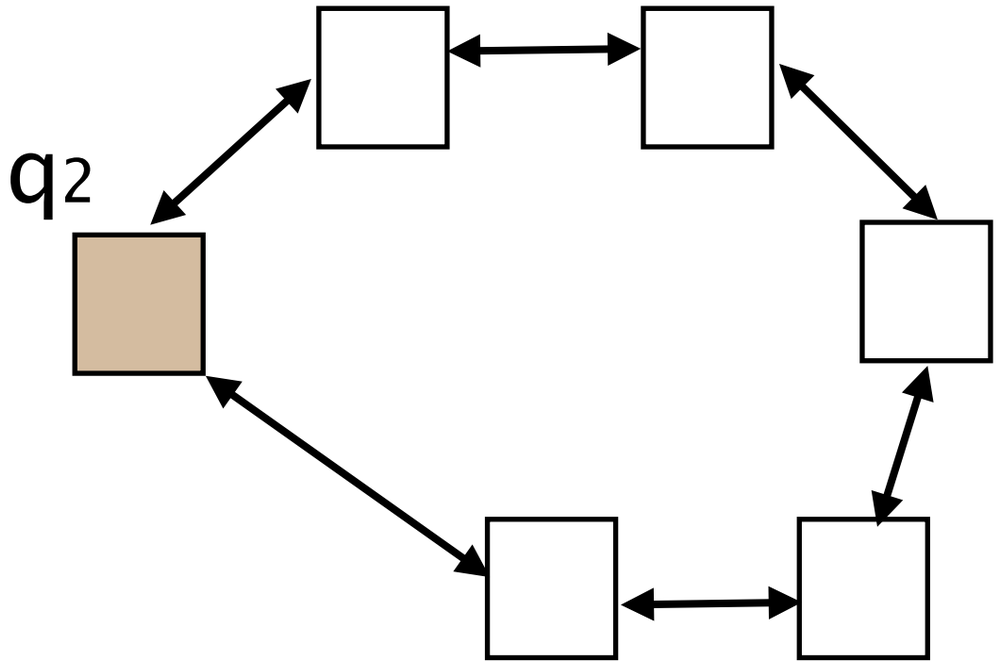




```
move(key,q1,q2) { rq=find(q1,key);
   list_del(q1,rq);tree_del(q1,rq);
   list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.

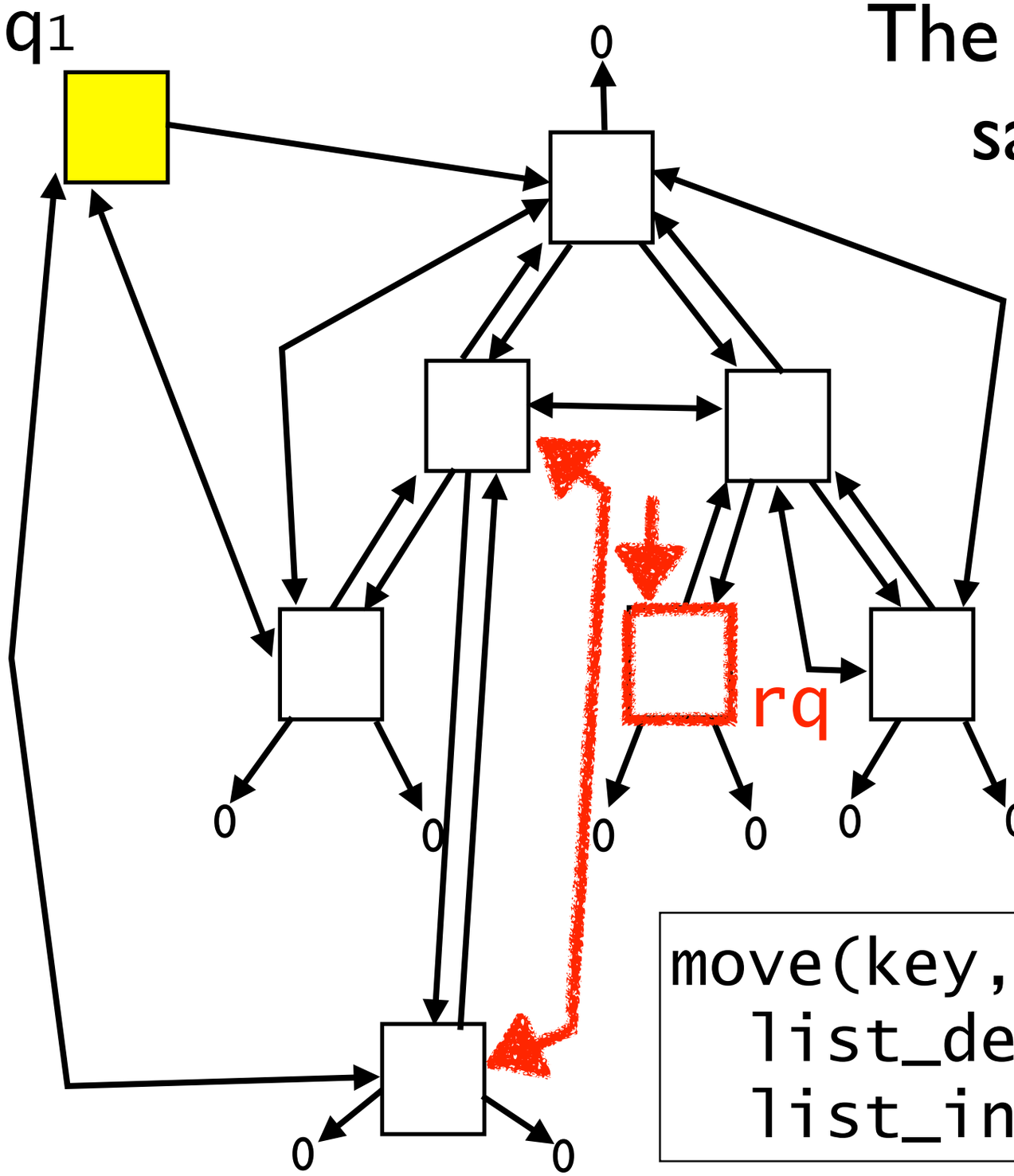


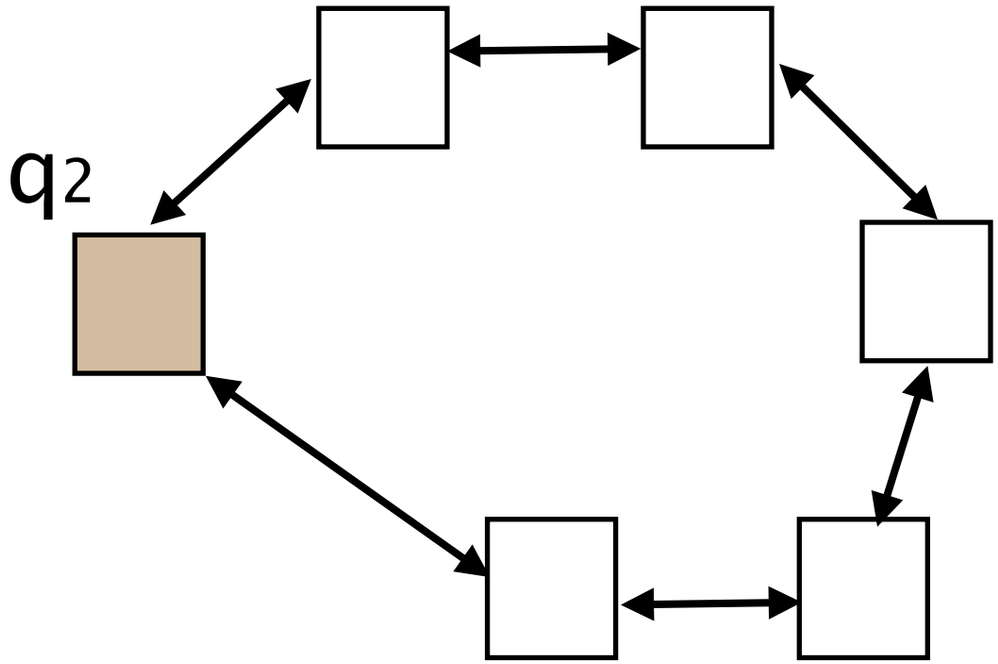




```
move(key,q1,q2) { rq=find(q1,key);
   list_del(q1,rq);tree_del(q1,rq);
   list_insert(q2,rq); }
```

# Move a request

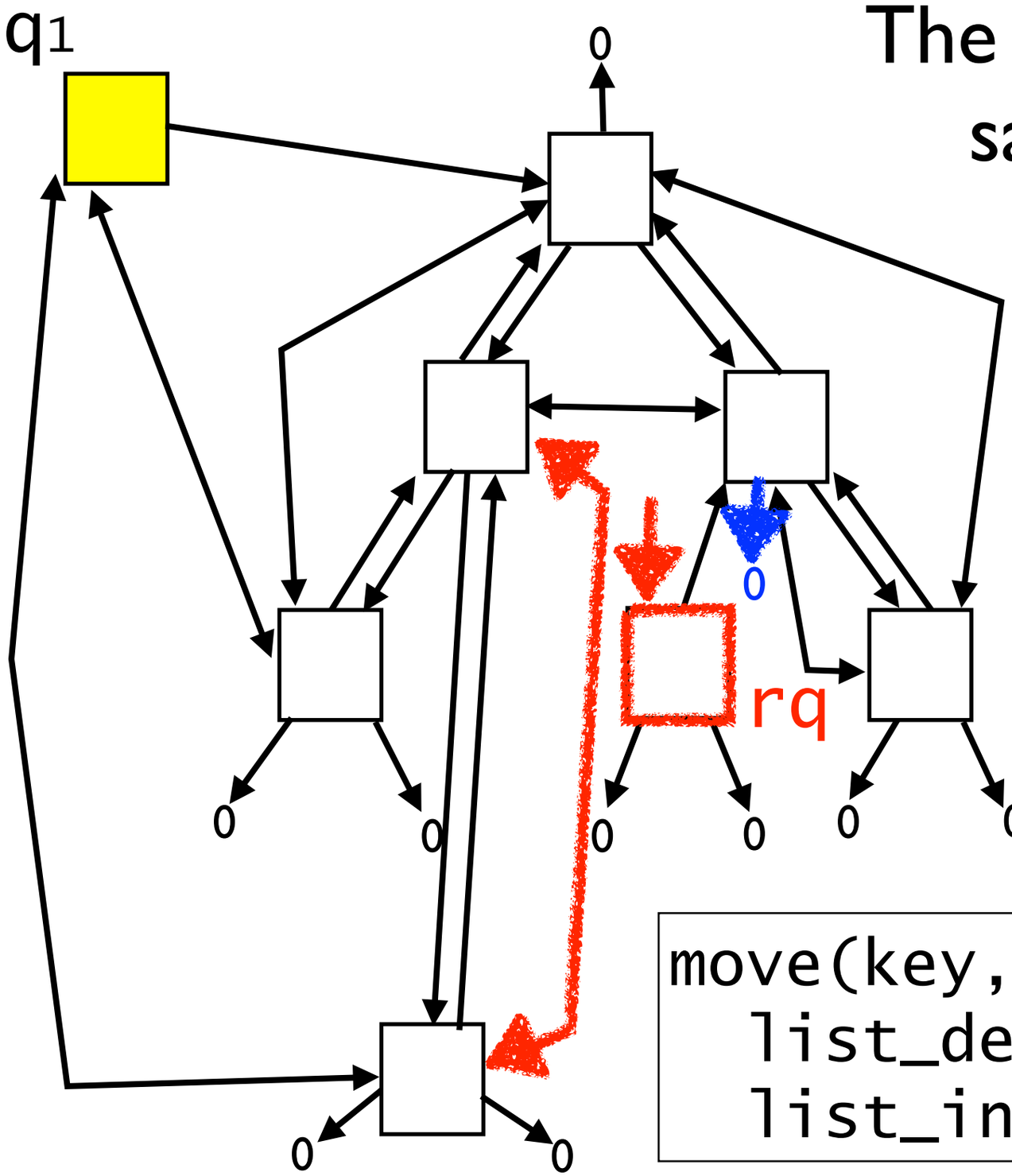
The tree and the list use the same set of heap cells.



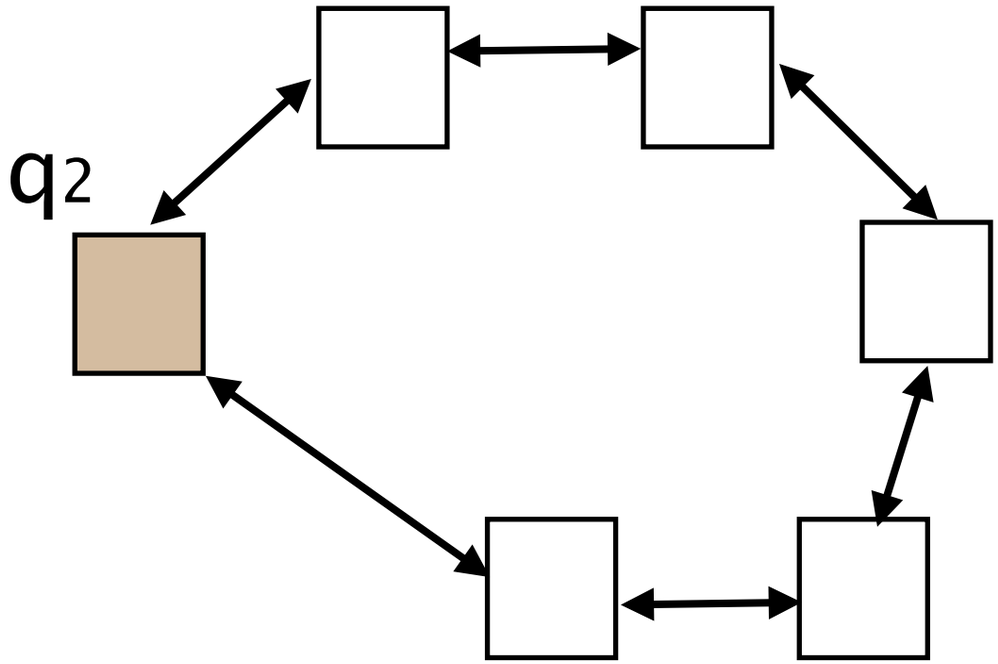




```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request



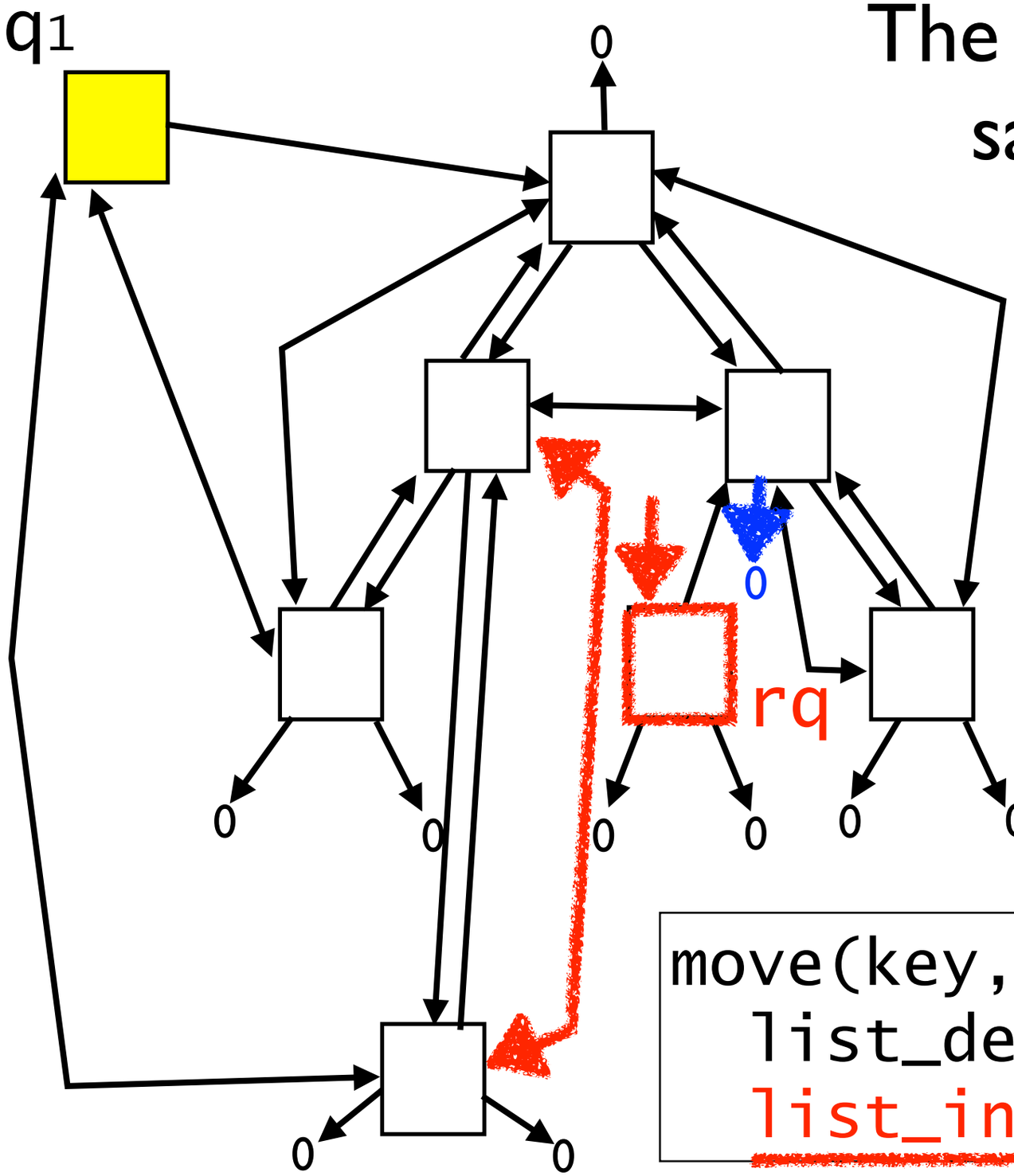

The tree and the list use the same set of heap cells.

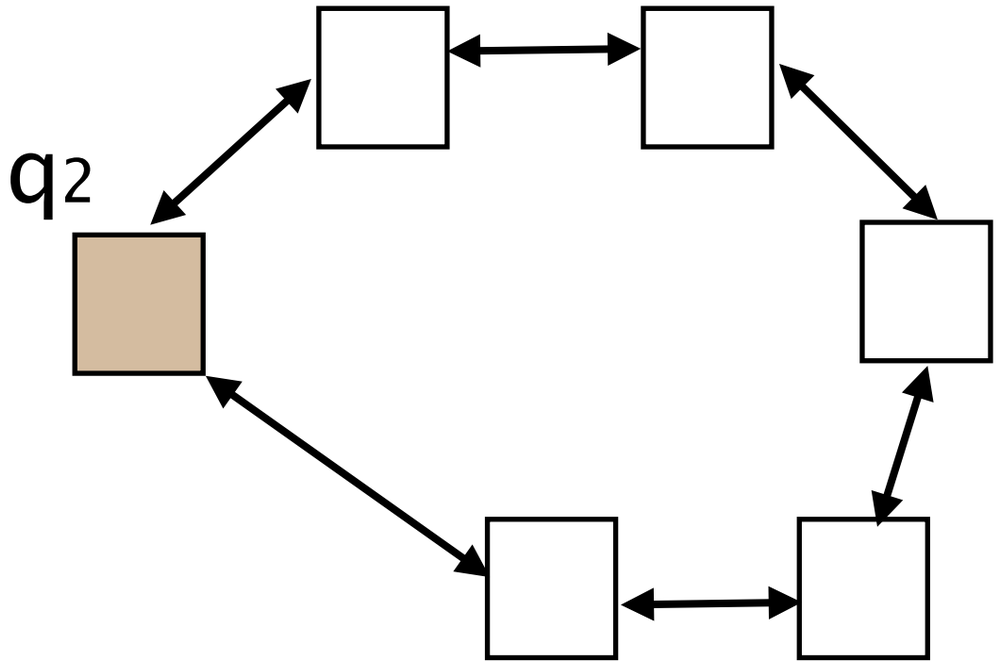



```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.

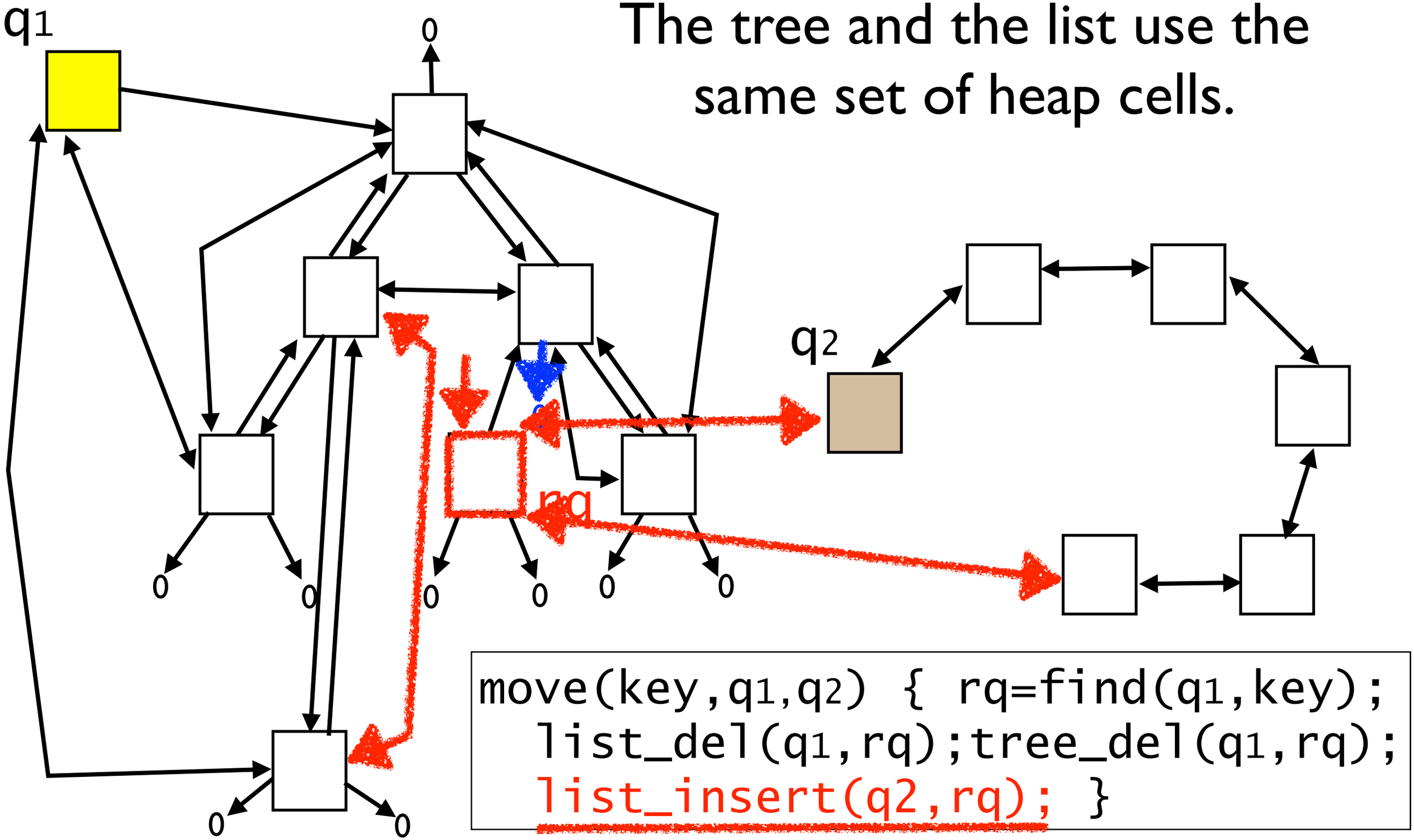



```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

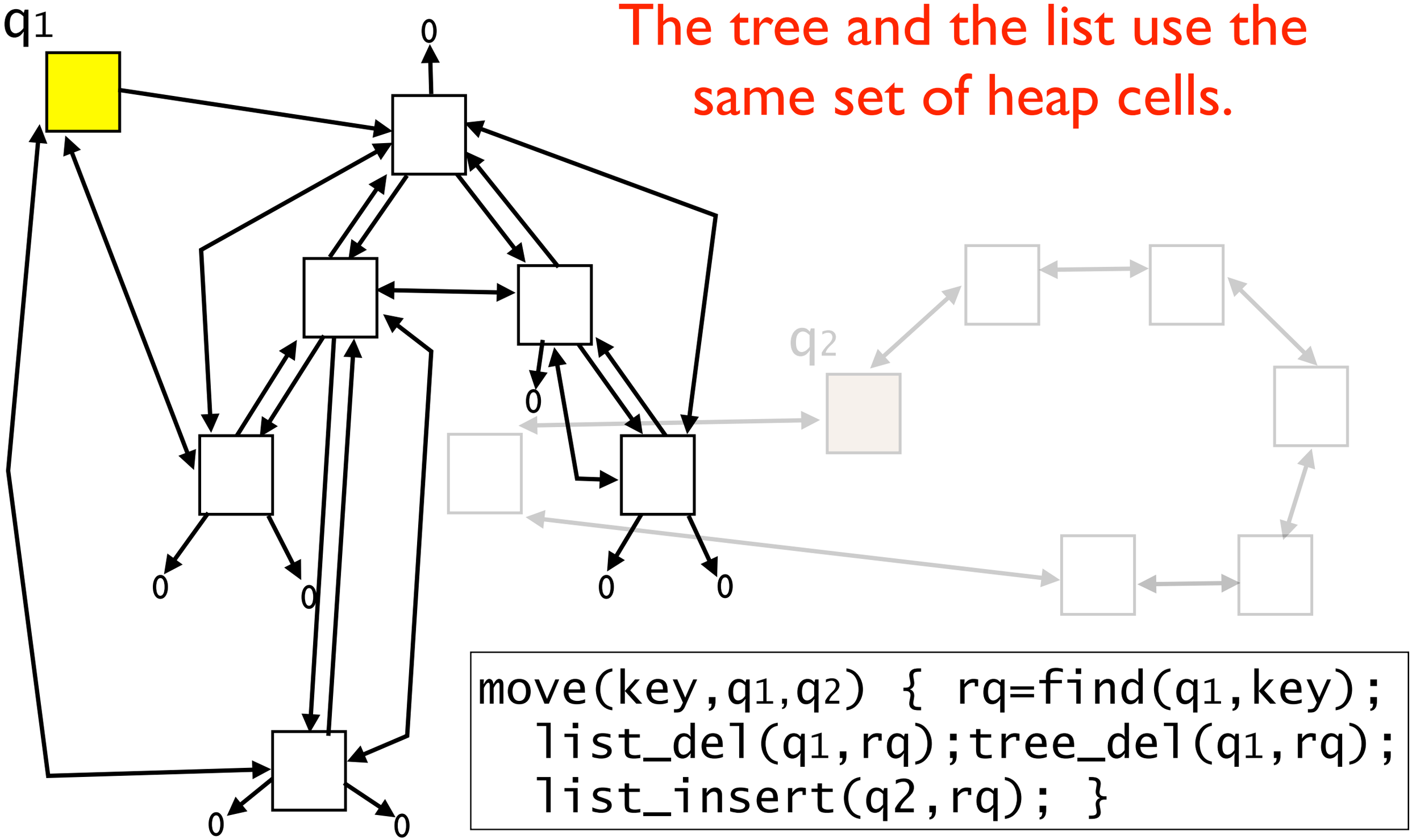
# Move a request

The tree and the list use the same set of heap cells.




```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request

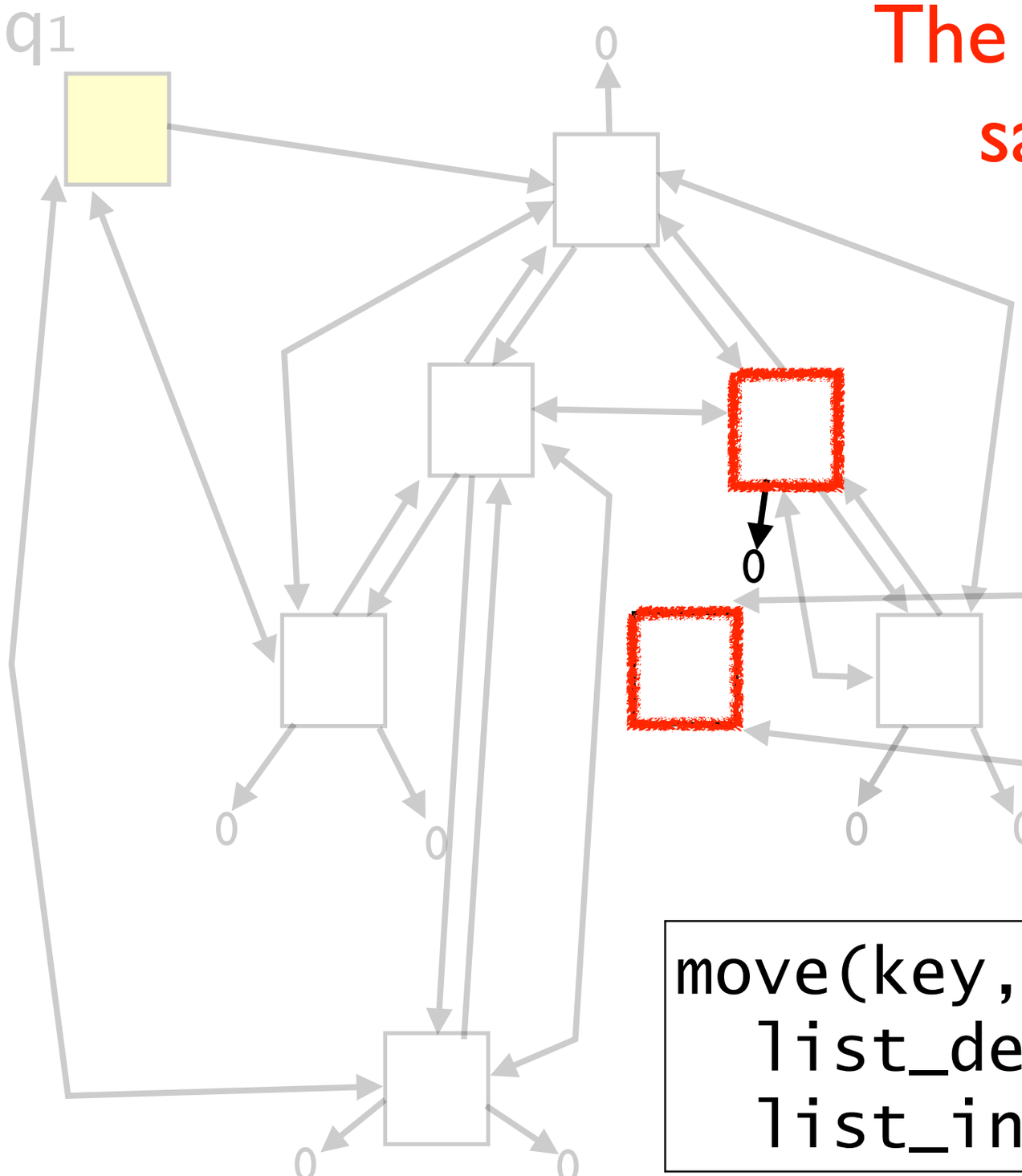


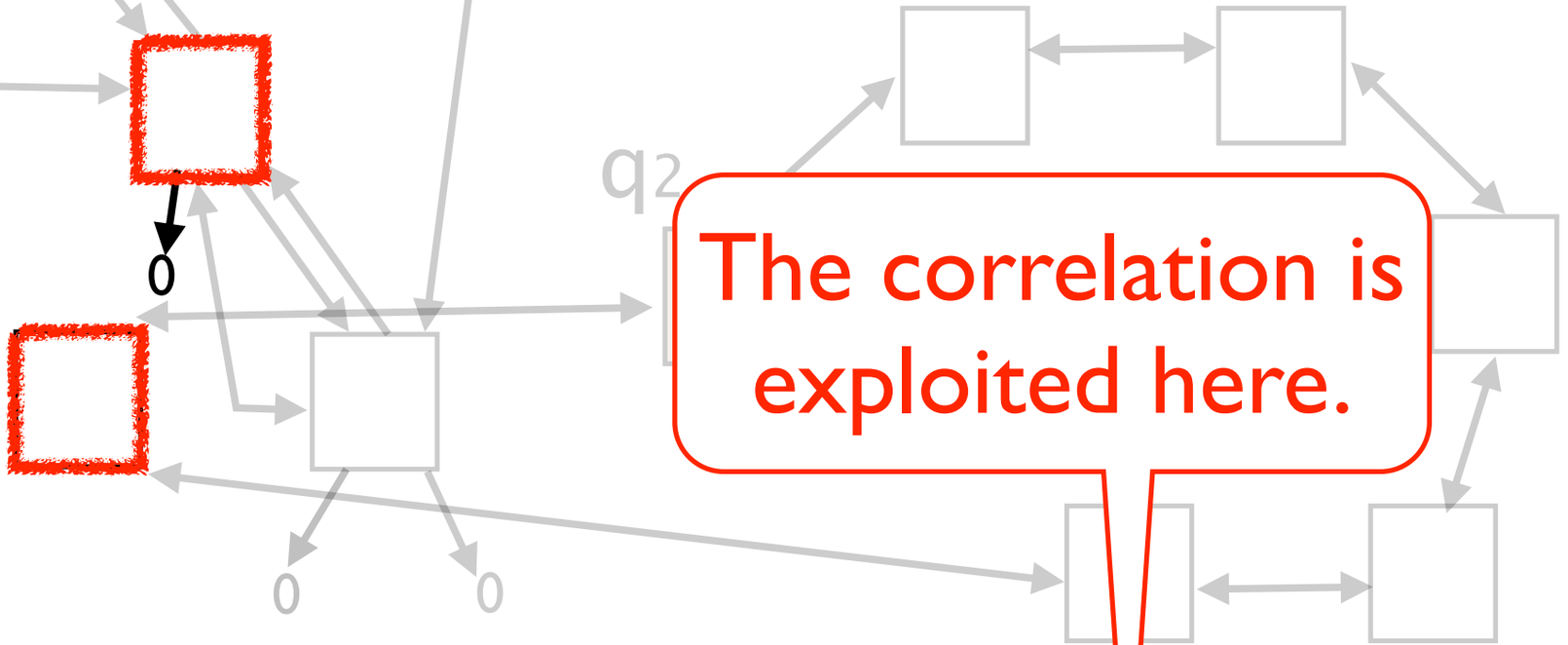

The tree and the list use the same set of heap cells.

```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Move a request

The tree and the list use the same set of heap cells.




```
move(key,q1,q2) { rq=find(q1,key);
  list_del(q1,rq);tree_del(q1,rq);
  list_insert(q2,rq); }
```

# Concrete goal

- Exploit the loose correlation, and build an efficient heap analysis for overlaid data structures.
- Verify the memory safety of the deadline IO scheduler.
  1. Aug'09 version: Too slow. Also too imprecise.
  2. Oct'09 version: Cannot prove one move routine.
  3. Nov'10 version: Can prove the whole in < 500s.

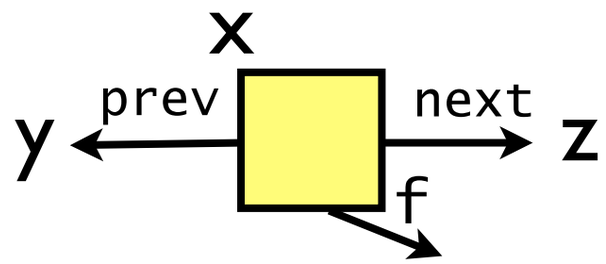
# Demo

# Design principle

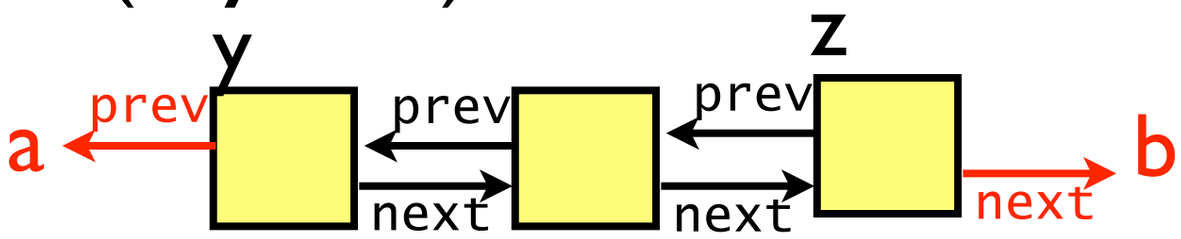
- Analyse the tree part and the list part of code as separately as possible.
- Separation buys us performance.
- Cannot be made completely separate.
- To allow communication between parts, we use ghost variables and infer ghost instructions during analysis.

# Assertions in separation logic

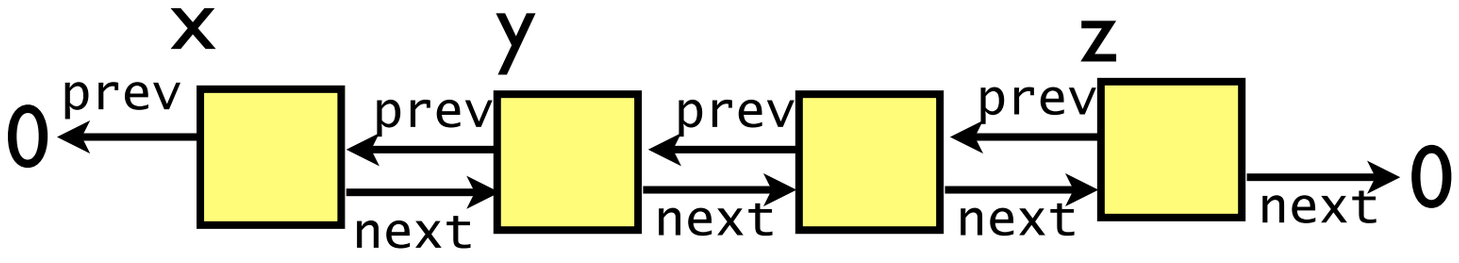
$x \mapsto \{\texttt{prev}: y, \texttt{next}: z\}$



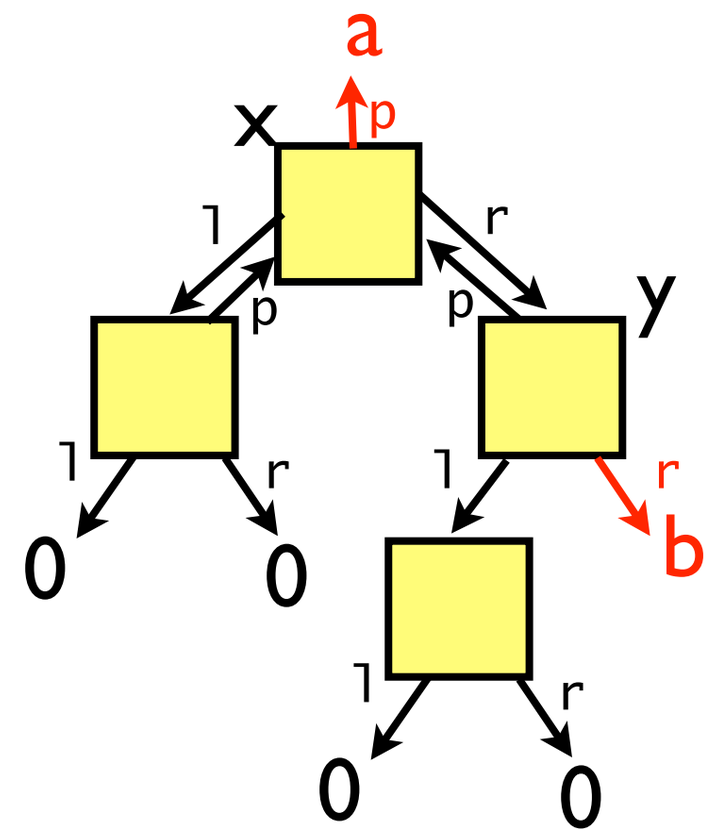

$\mathrm{ls}_{ne}(a, y, z, b)$




$x \mapsto \{\texttt{prev}: 0, \texttt{next}: y\} * \mathrm{ls}_{ne}(x, y, z, 0)$




$\mathrm{tree}_{pe}(a,x,y,b)$




Also, includes standard logical connectives $\wedge$, $\vee$, $\exists$ etc.

# Representation with *, $\wedge$ and $\alpha$

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$
$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, ∧ and α

Conjunct for tree-related properties

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, $\wedge$ and $\alpha$

Conjunct for tree-related properties

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

Conjunct for list-related properties

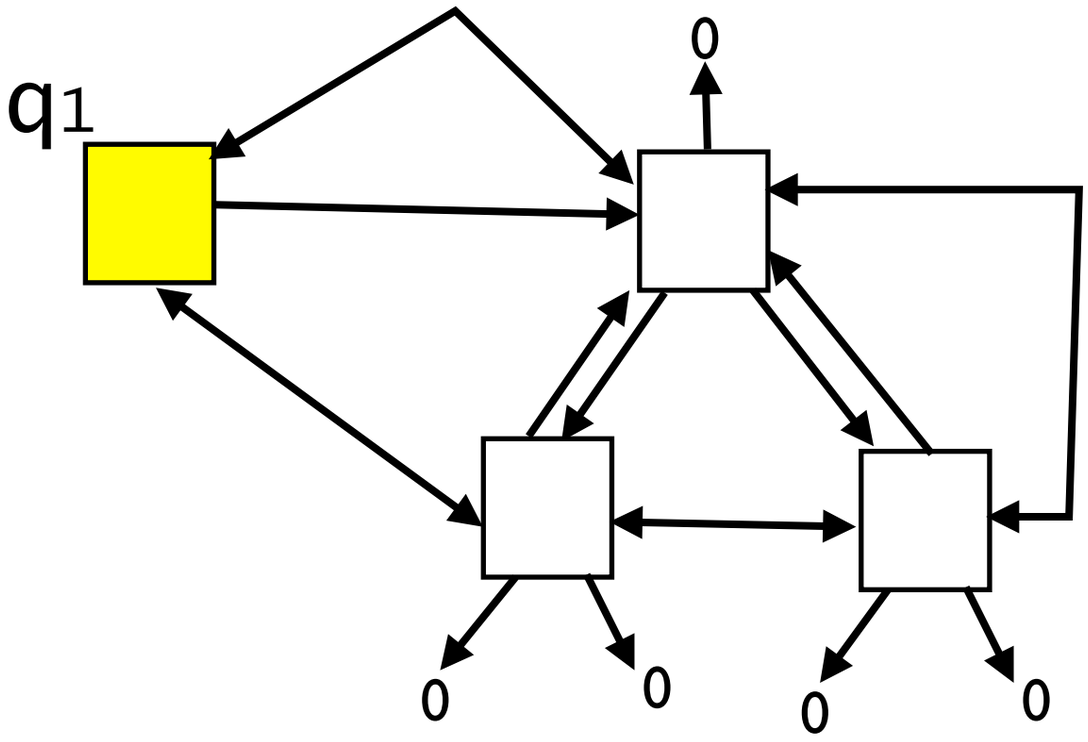
# Representation with *, ∧ and α

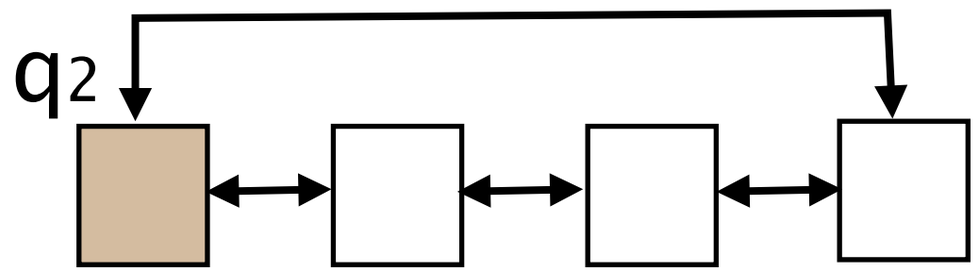
$$(q_1 \mapsto \{\texttt{root}:a'\}_{\alpha} * \mathrm{tree}_{pe}(0,a',b',0)_{\beta} * \mathrm{true}_{\gamma}) \quad \wedge$$
$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_{\alpha} * \mathrm{ls}_{pe}(q_1,c',d',q_1)_{\beta} * \mathrm{ls}_{ne}(e',q_2,e',q_2)_{\gamma})$$

# Representation with *, ∧ and α

$$\exists a', b'.\ (q_1 \mapsto \{\texttt{root}: a'\}_\alpha * \mathrm{tree}_{pe}(0, a', b', 0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$
$$(q_1 \mapsto \{\texttt{next}: c', \texttt{prev}: d'\}_\alpha * \mathrm{ls}_{pe}(q_1, c', d', q_1)_\beta * \mathrm{ls}_{ne}(e', q_2, e', q_2)_\gamma)$$
$$\exists c', d', e'$$

# Representation with *, ∧ and α

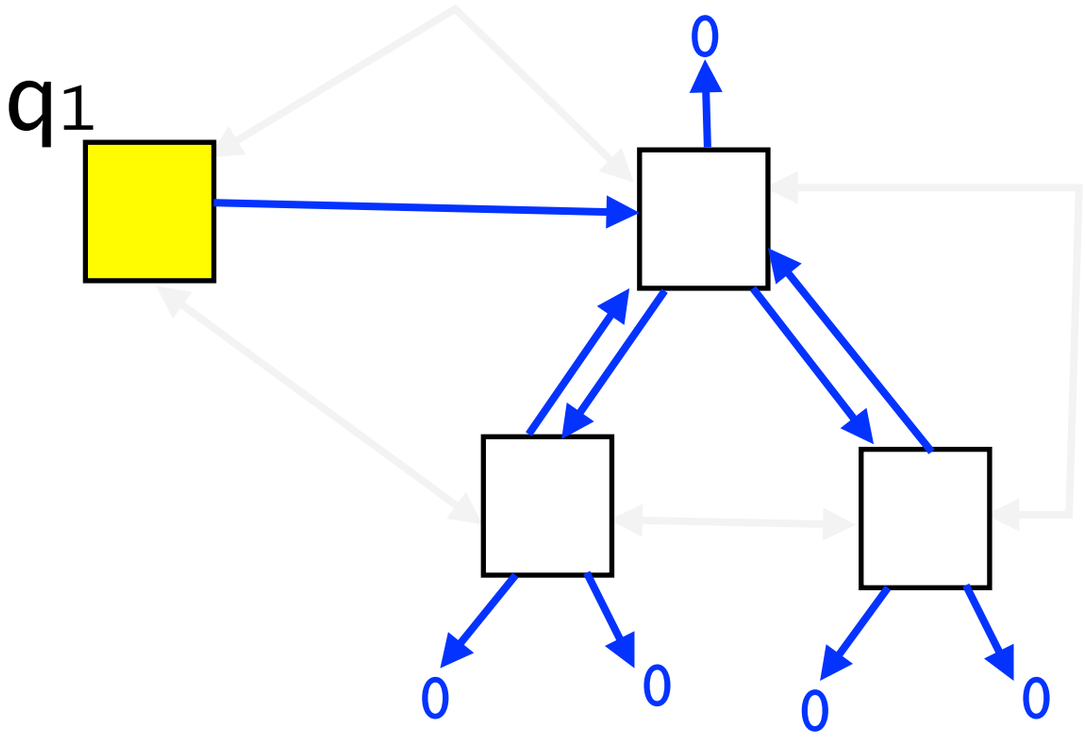


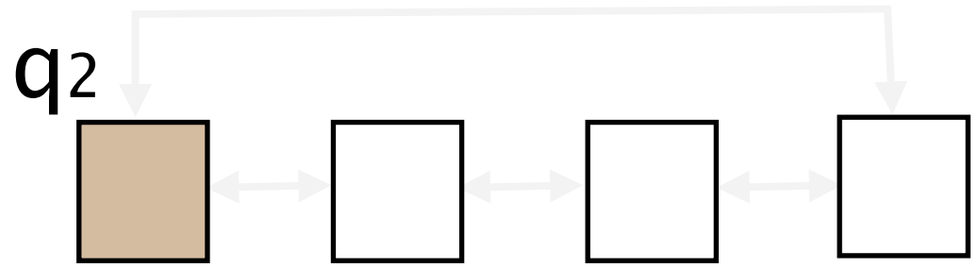




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, ∧ and α

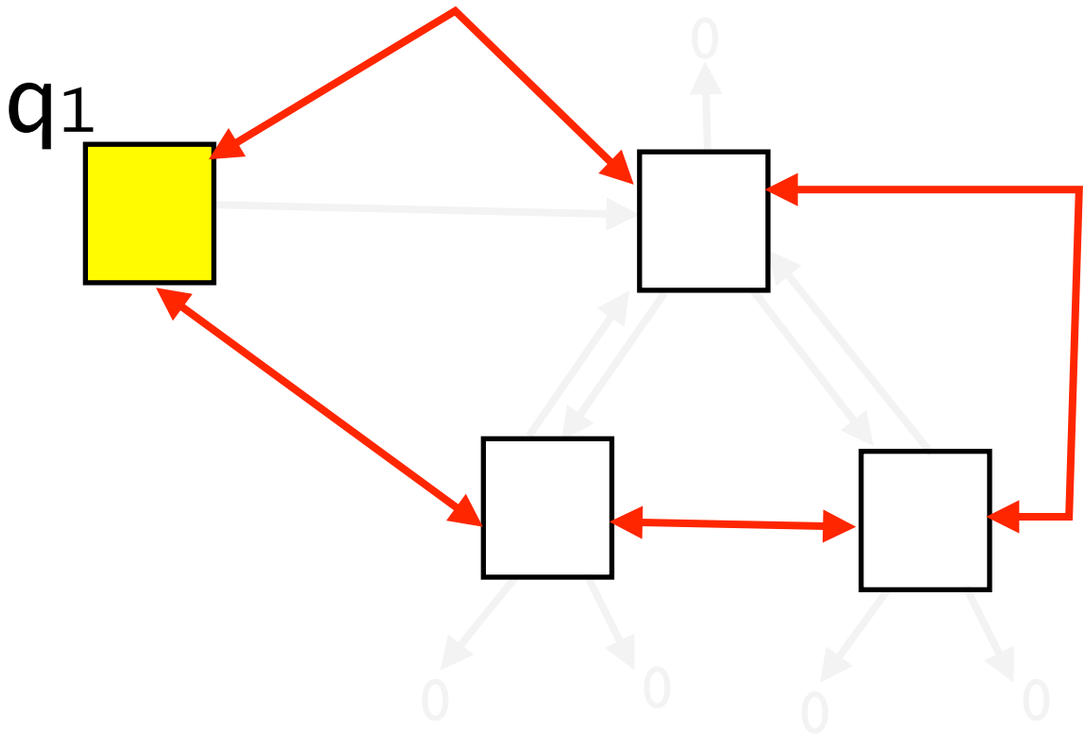


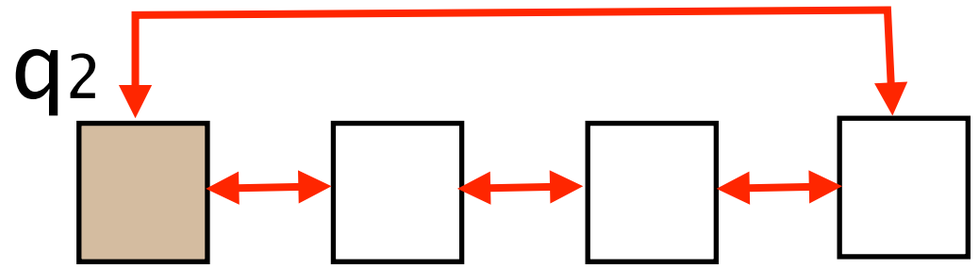




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, ∧ and α

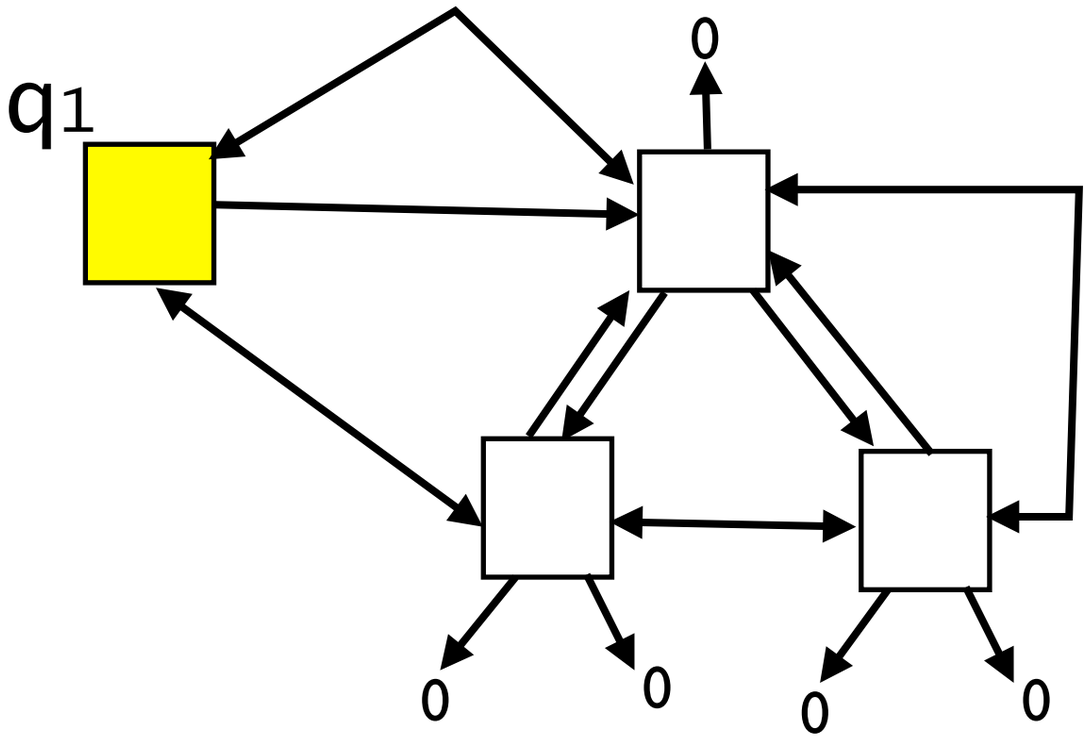


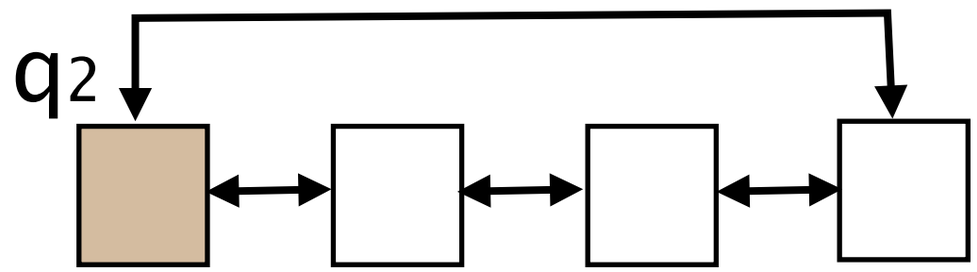




$(q_1 \mapsto \{\texttt{root:a'}\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$

$(q_1 \mapsto \{\texttt{next:c'},\texttt{prev:d'}\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$

# Representation with *, ∧ and α

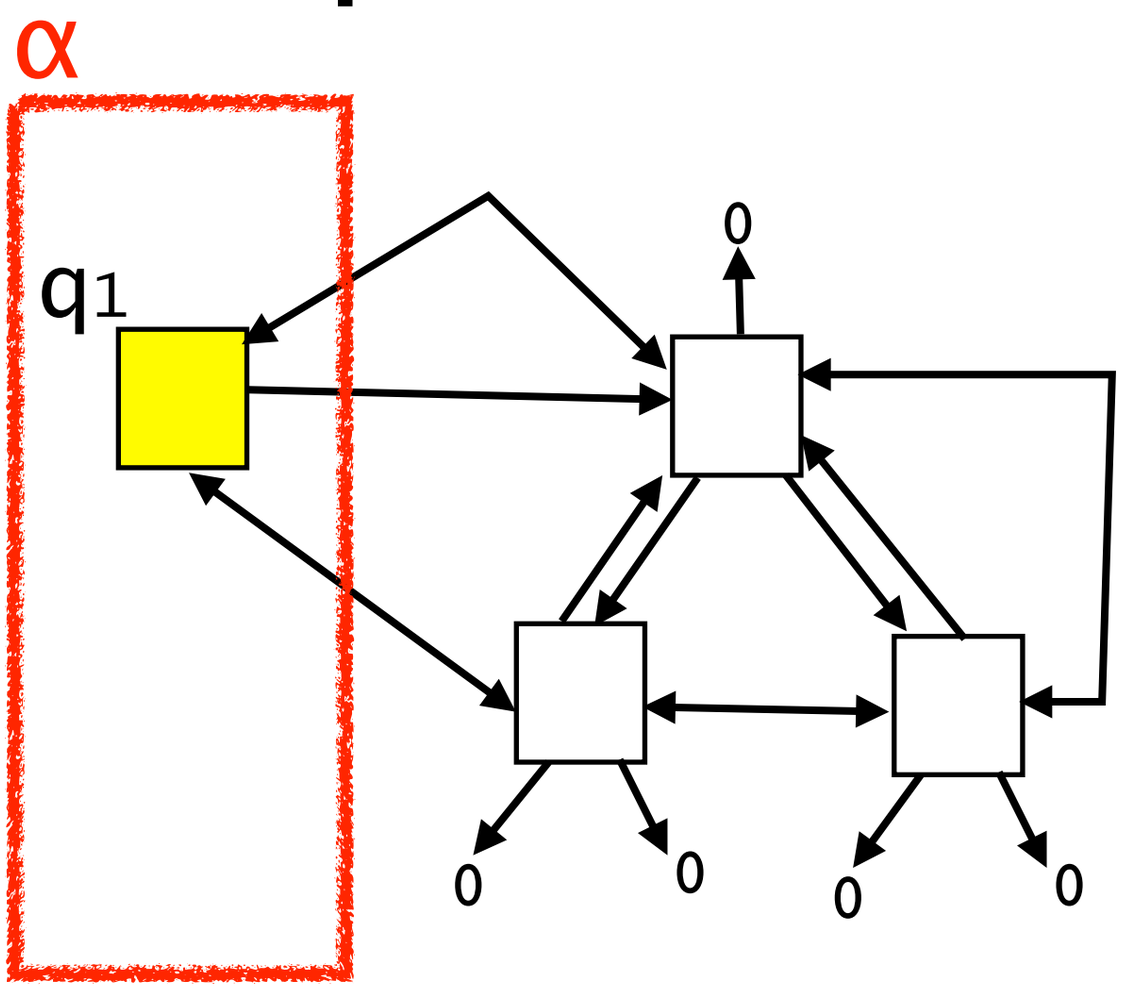


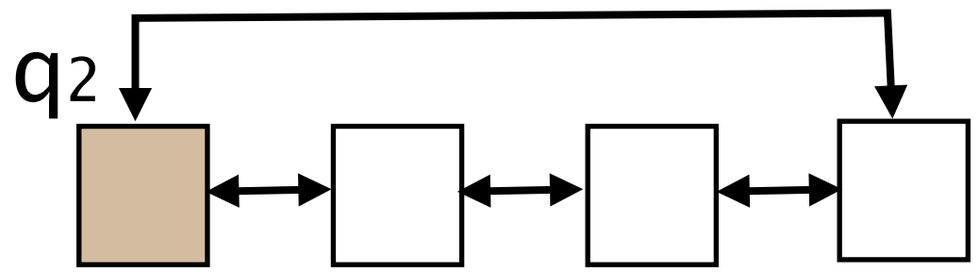




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, ∧ and α

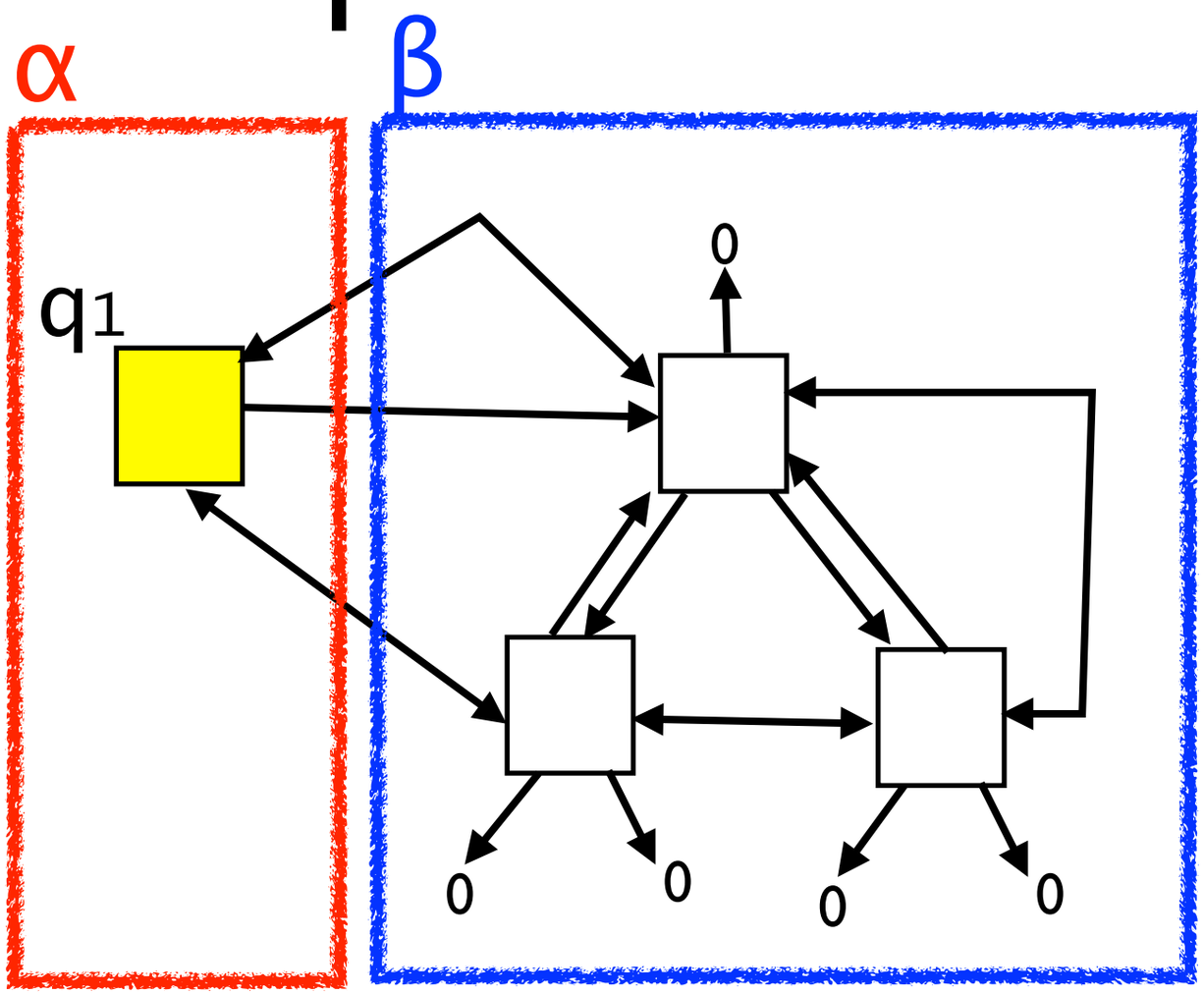


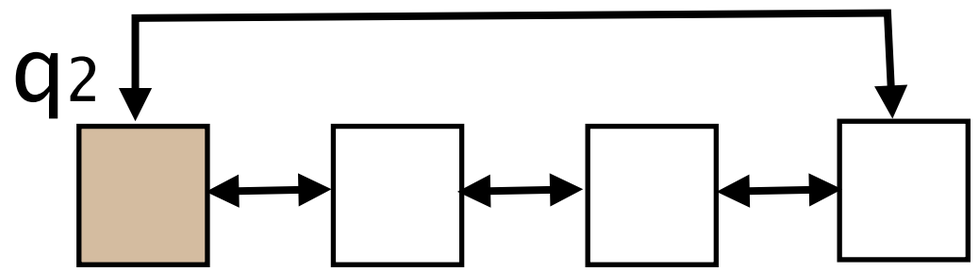




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Representation with *, ∧ and α

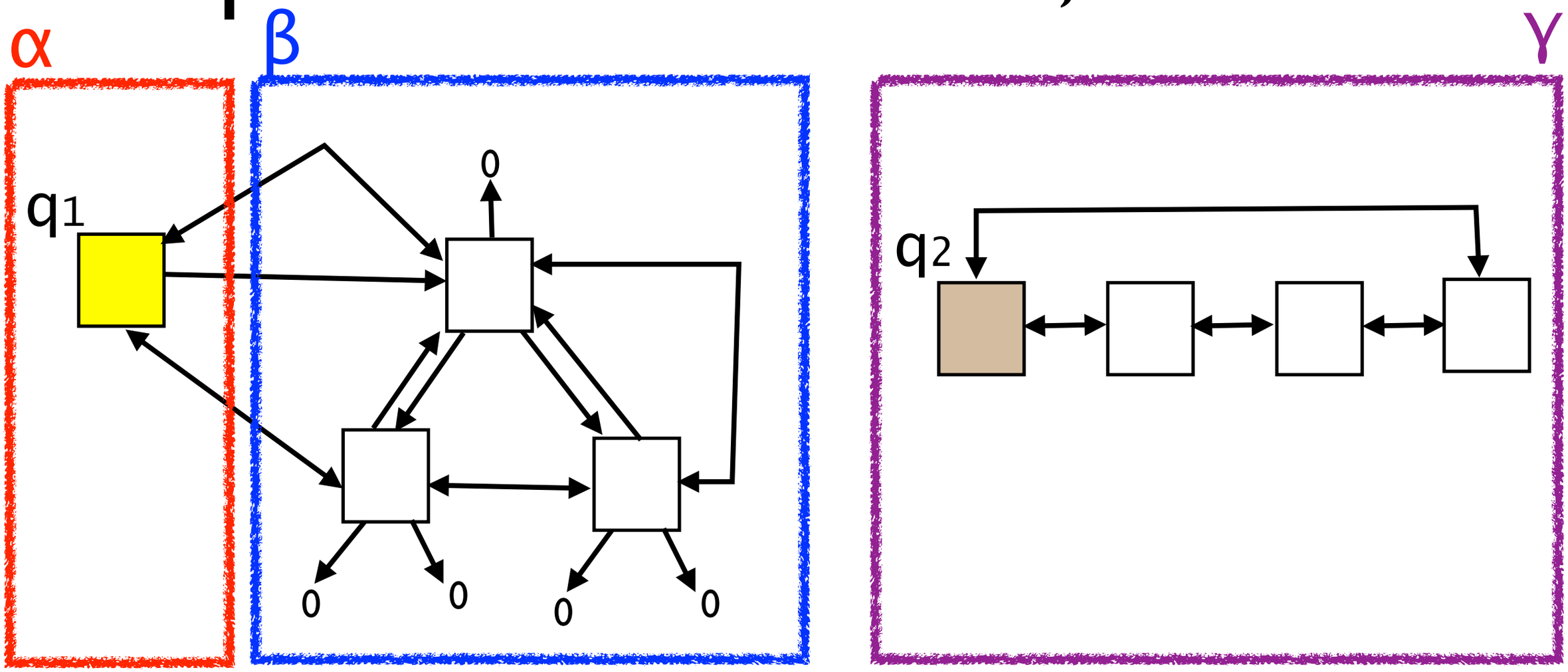



$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \underline{\text{tree}_{pe}(0,a',b',0)_\beta} * \text{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c', \texttt{prev}:d'\}_\alpha * \underline{\text{ls}_{pe}(q_1,c',d',q_1)_\beta} * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

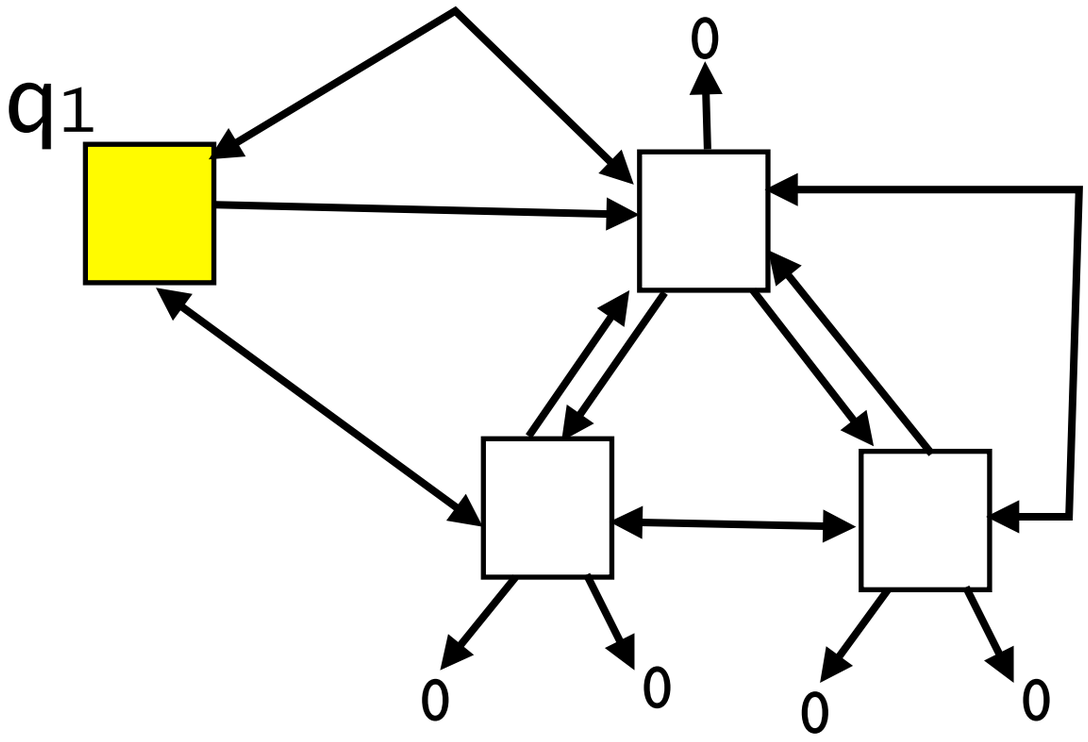
# Representation with *, ∧ and α

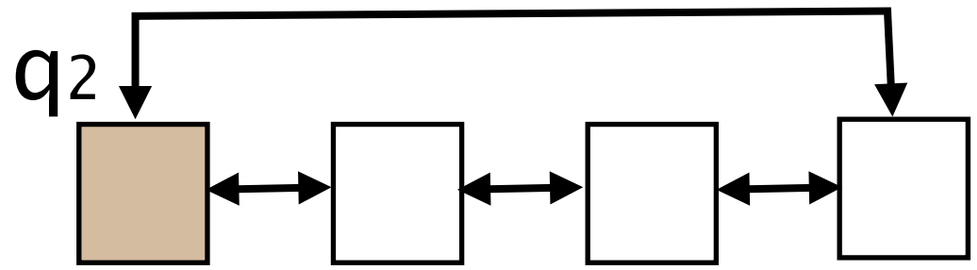



$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c', \texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation

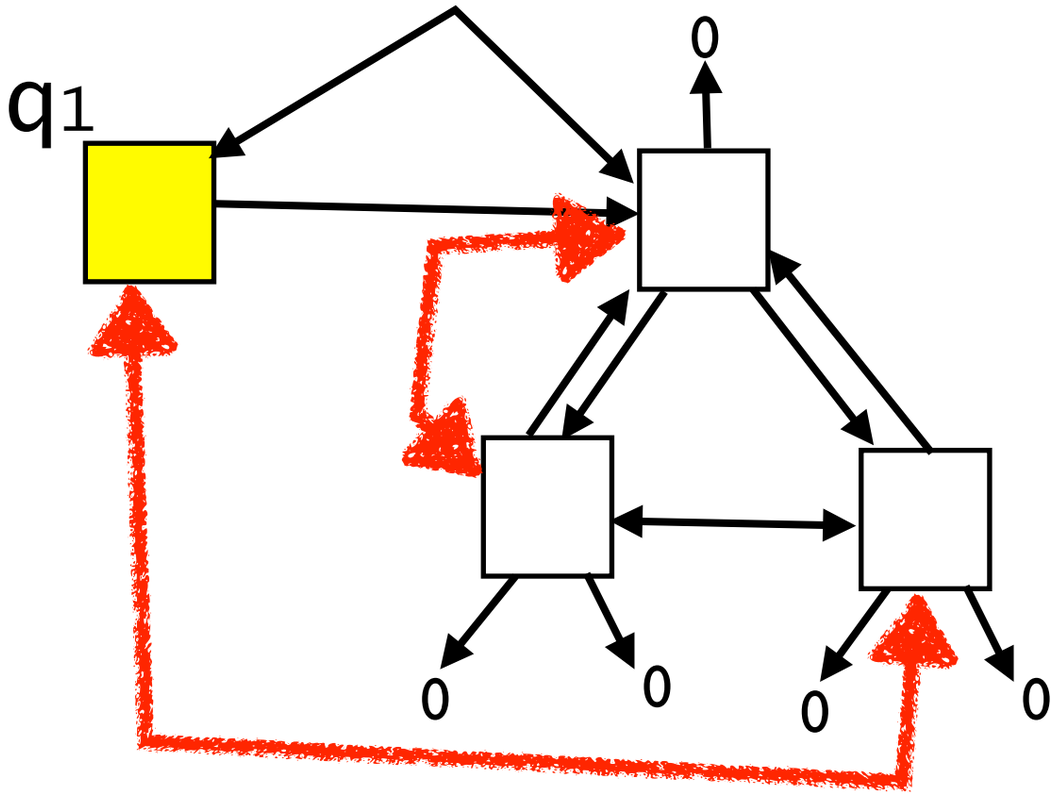


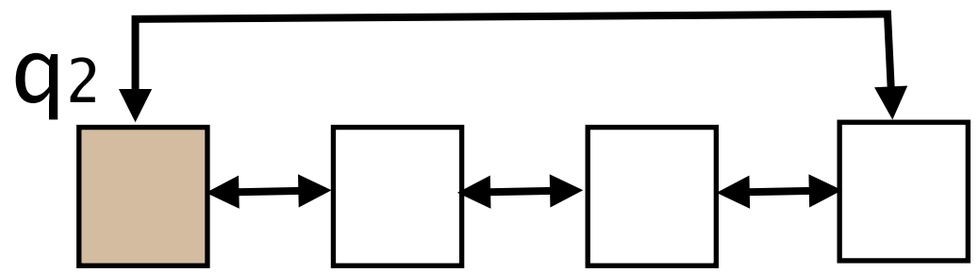




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathsf{tree}_{pe}(0,a',b',0)_\beta * \mathsf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathsf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathsf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation

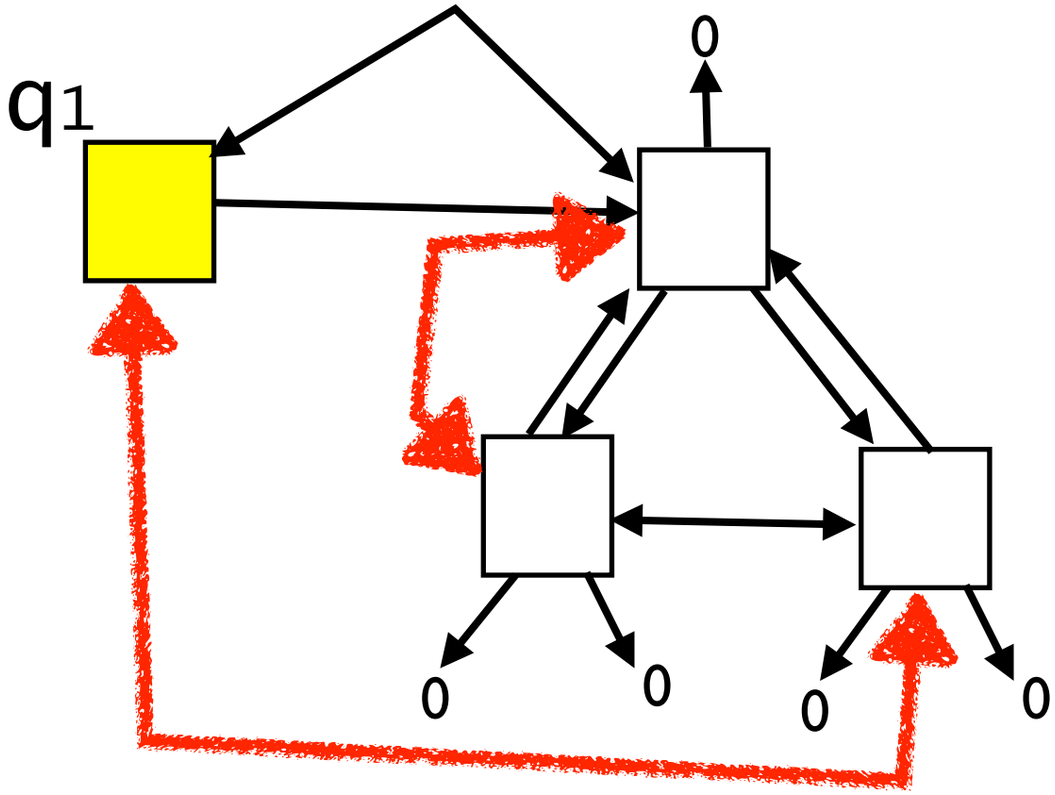


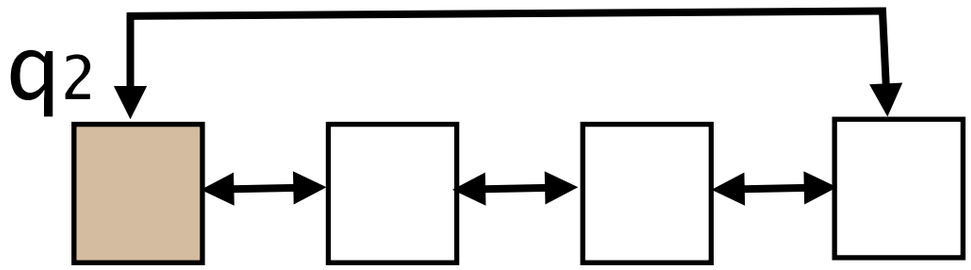




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation



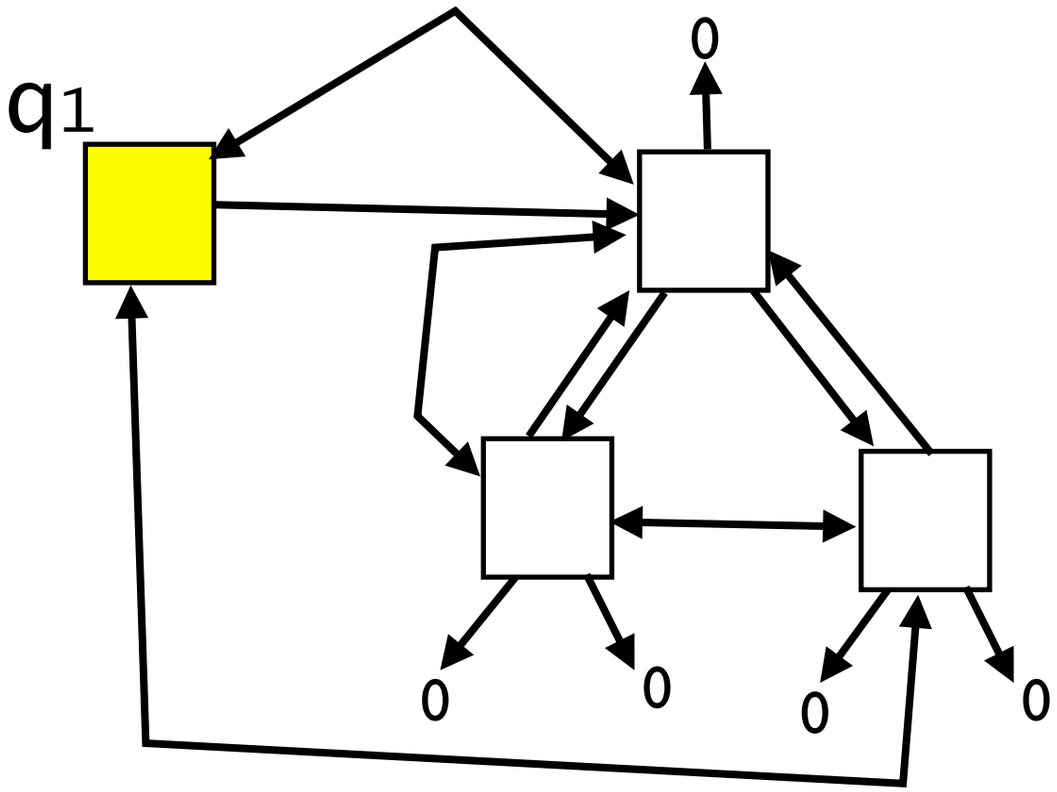



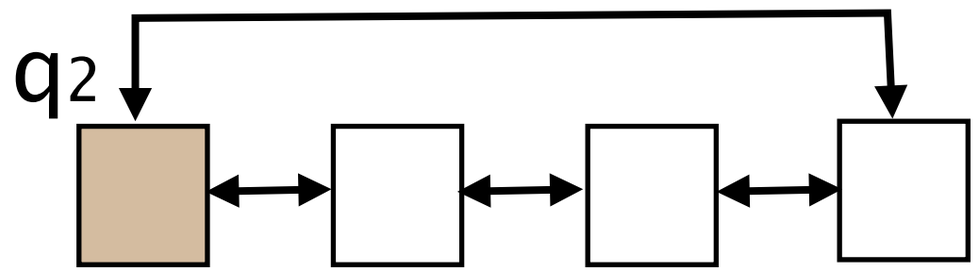

**Yes!!**

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation

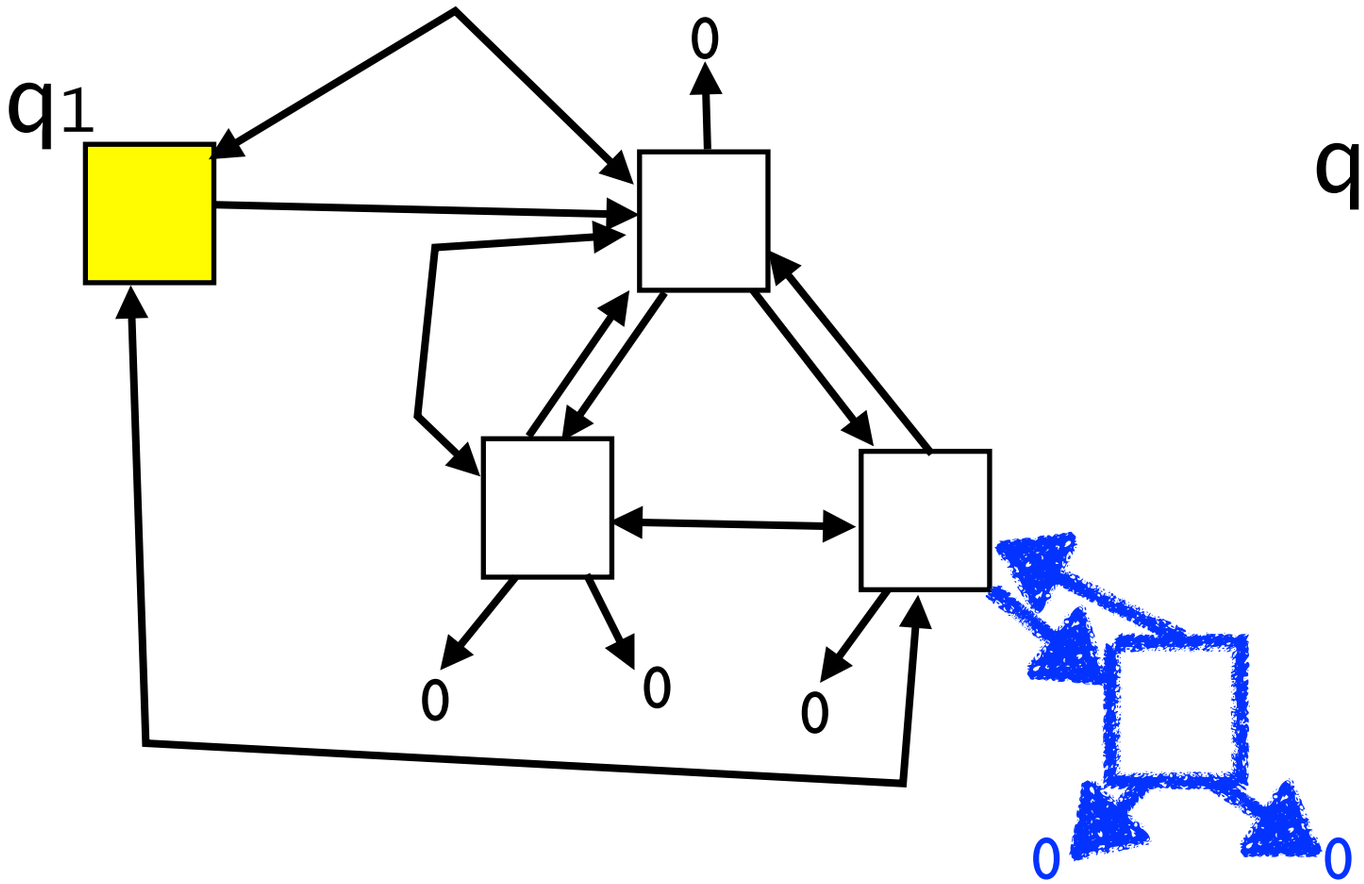


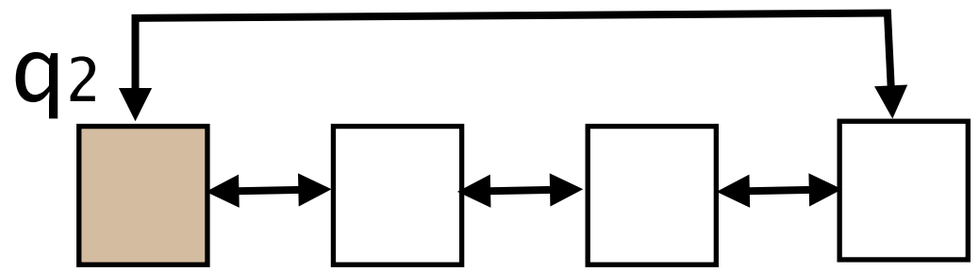




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation

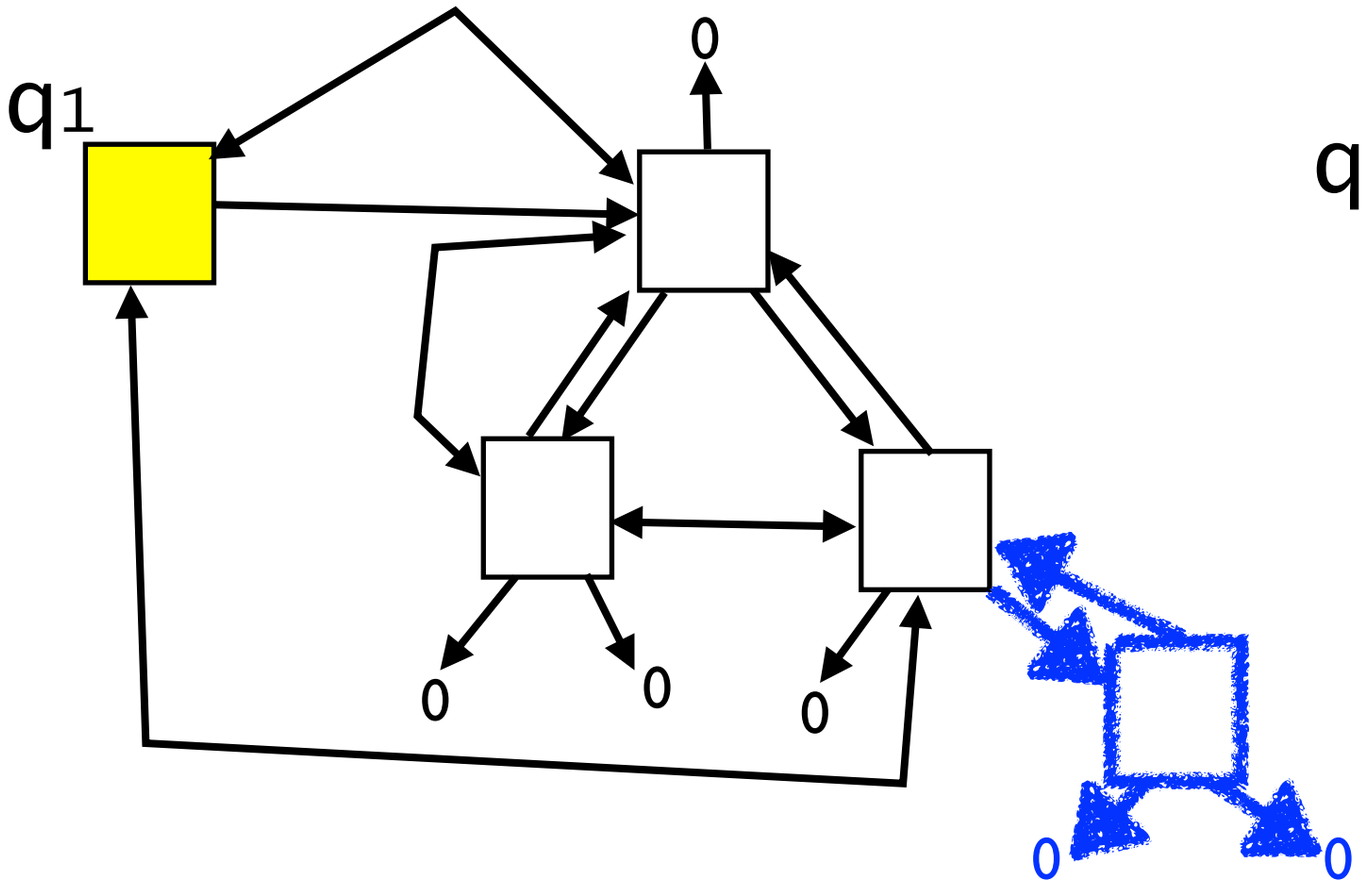


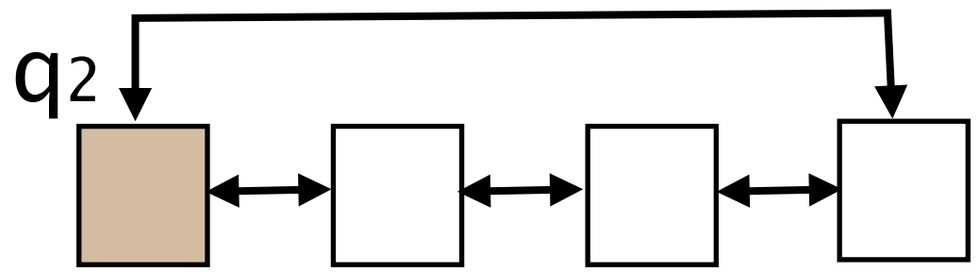




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Loose correlation




NO!!

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \underline{\text{tree}_{pe}(0,a',b',0)_\beta} * \text{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \underline{\text{ls}_{pe}(q_1,c',d',q_1)_\beta} * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Abstract domain

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$
$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

# Abstract domain

$(q_1 \mapsto \{\texttt{root}:0\}_\alpha * \text{true}_\gamma$

$\vee \quad \ldots$

$\vee\ q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$

$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma$

$\vee \quad \ldots \quad \vee \quad q_1 \mapsto \{\texttt{next}:q_1,\texttt{prev}:q_1\}_\alpha * \text{ls}_{ne}(f',q_2,f',q_2)_\gamma\ )$

# Abstract domain

$$\mathrm{Dom} \;=\; \mathrm{P}(\mathrm{TreeForm}) \times \mathrm{P}(\mathrm{ListForm}) \;\cup\; \{\top\}$$

$$\begin{array}{l}
(q_1 \mapsto \{\texttt{root}:0\}_\alpha \;*\; \mathrm{true}_\gamma \\
\quad \vee \quad \ldots \\
\quad \vee\; q_1 \mapsto \{\texttt{root}:a'\}_\alpha \;*\; \mathrm{tree}_{pe}(0,a',b',0)_\beta \;*\; \mathrm{true}_\gamma)
\end{array} \quad \wedge$$

$$\begin{array}{l}
(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha \;*\; \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta \;*\; \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma \\
\quad \vee \quad \ldots \quad \vee \quad q_1 \mapsto \{\texttt{next}:q_1,\texttt{prev}:q_1\}_\alpha \;*\; \mathrm{ls}_{ne}(f',q_2,f',q_2)_\gamma\;)
\end{array}$$

# Analysis of request move

Run tree and list analyses as separately as possible.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq);
  tree_del(q1,rq);
  list_insert(q2,rq);                    }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq);
  tree_del(q1,rq);
  list_insert(q2,rq);                    }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

```
void move(key,q1,q2) {
   request* rq=find(q1,key);
   list_del(q1,rq);
   tree_del(q1,rq);
   list_insert(q2,rq);                    }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

1. Insert `moveInfo` and `transfer`.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq);                }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

1. Insert `moveInfo` and `transfer`.

2. Do the main analysis. This can insert the unification of ghost variables.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq);                }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

1. Insert `moveInfo` and `transfer`.

2. Do the main analysis. This can insert the unification of ghost variables.

SomeTreeForm $\wedge$ ( .. * $\text{ls}_{ne}(.., q_2, ..)_Y$) from prev slide

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq);                }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

1. Insert `moveInfo` and `transfer`.

2. Do the main analysis. This can insert the unification of ghost variables.

SomeTreeForm $\wedge$ ( .. * $\mathrm{ls}_{ne}(.., q_2, ..)_\gamma$) from prev slide

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq); ϒ←ϒ ∪ δ;      }
```

# Analysis of request move

Run tree and list analyses as separately as possible.

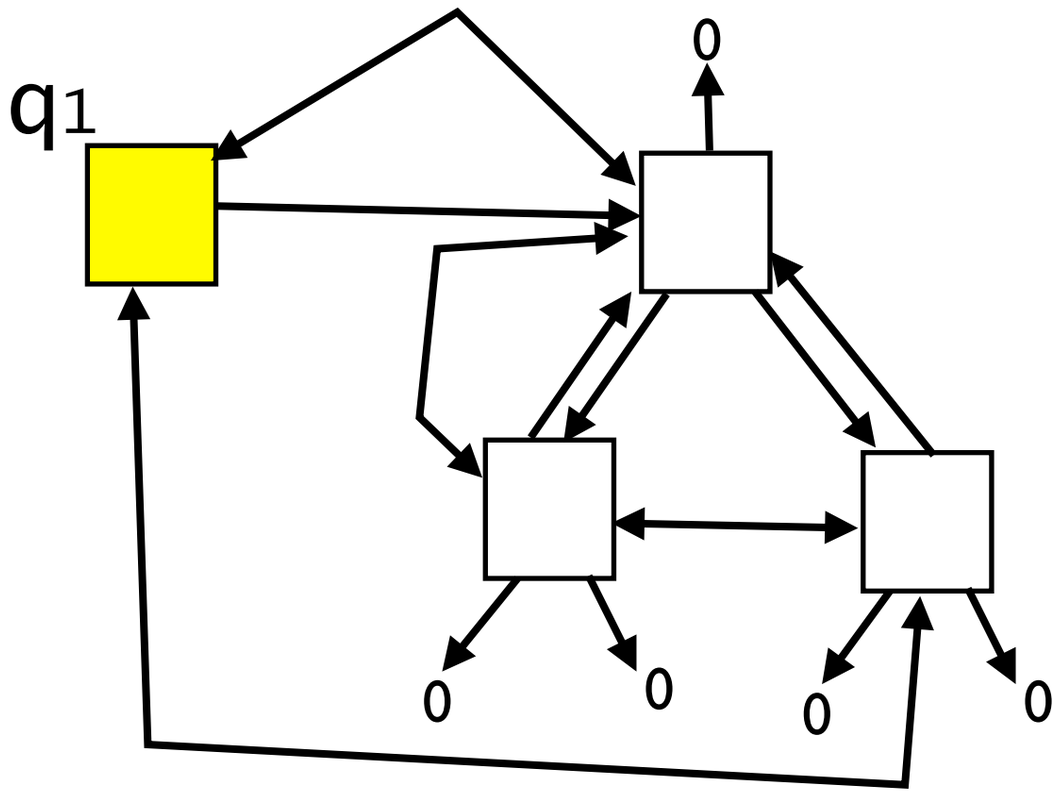
1. Insert `moveInfo` and `transfer`.
2. Do the main analysis. This can insert the unification of ghost variables.

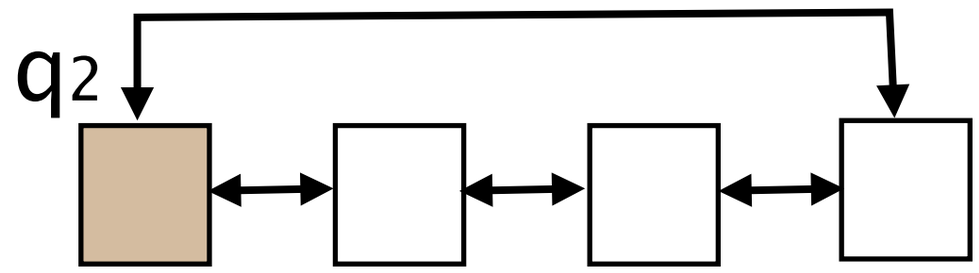
SomeTreeForm $\wedge$ ( .. * $\text{ls}_{ne}(.., q_2, ..)_\gamma$) from prev slide

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq);
  tree_del(q1,rq);
  list_insert(q2,rq);
```

```
moveInfo(2,1,rq);
transfer(rq,δ);
ϒ←ϒ ∪ δ;
}
```

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```

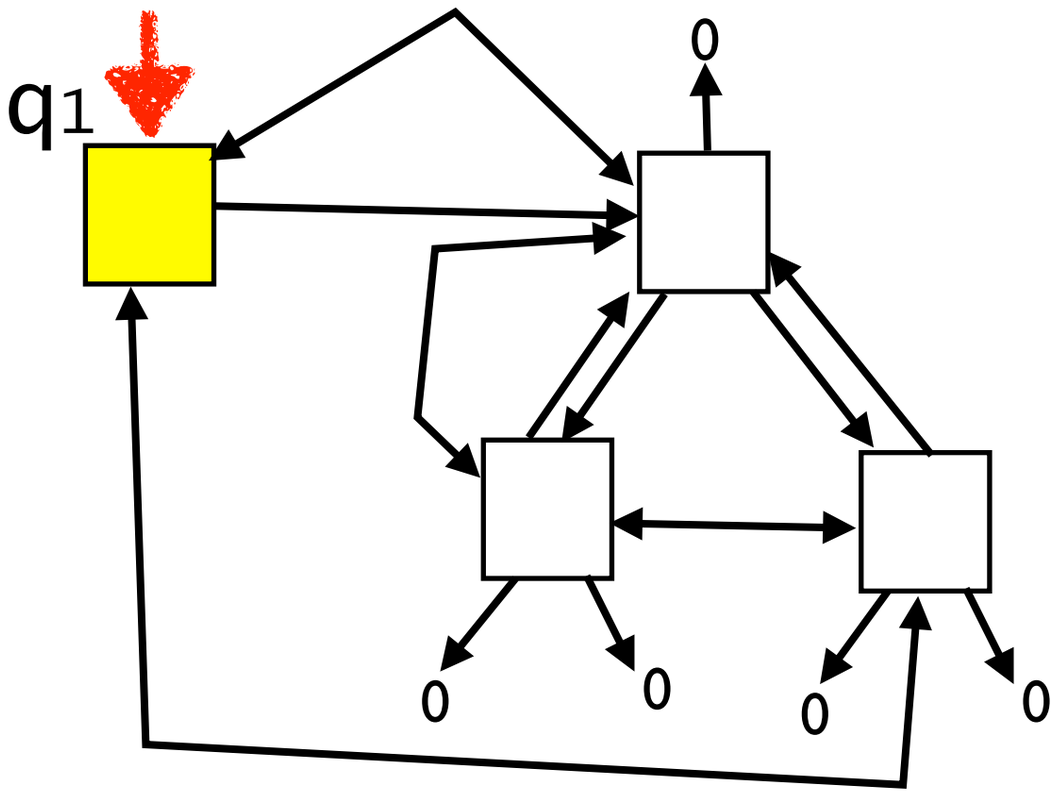


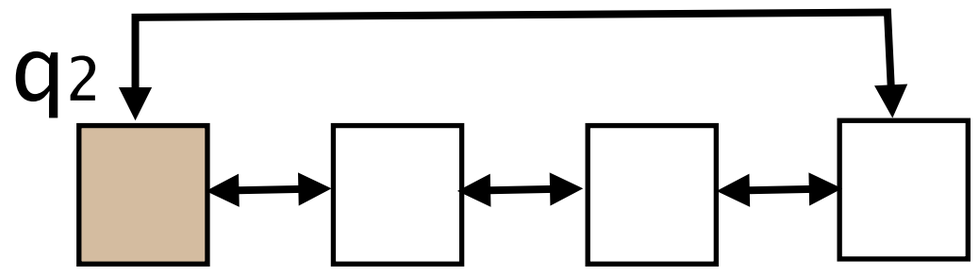




$$(q_1 \mapsto \{\texttt{root}: a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}: c', \texttt{prev}: d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```

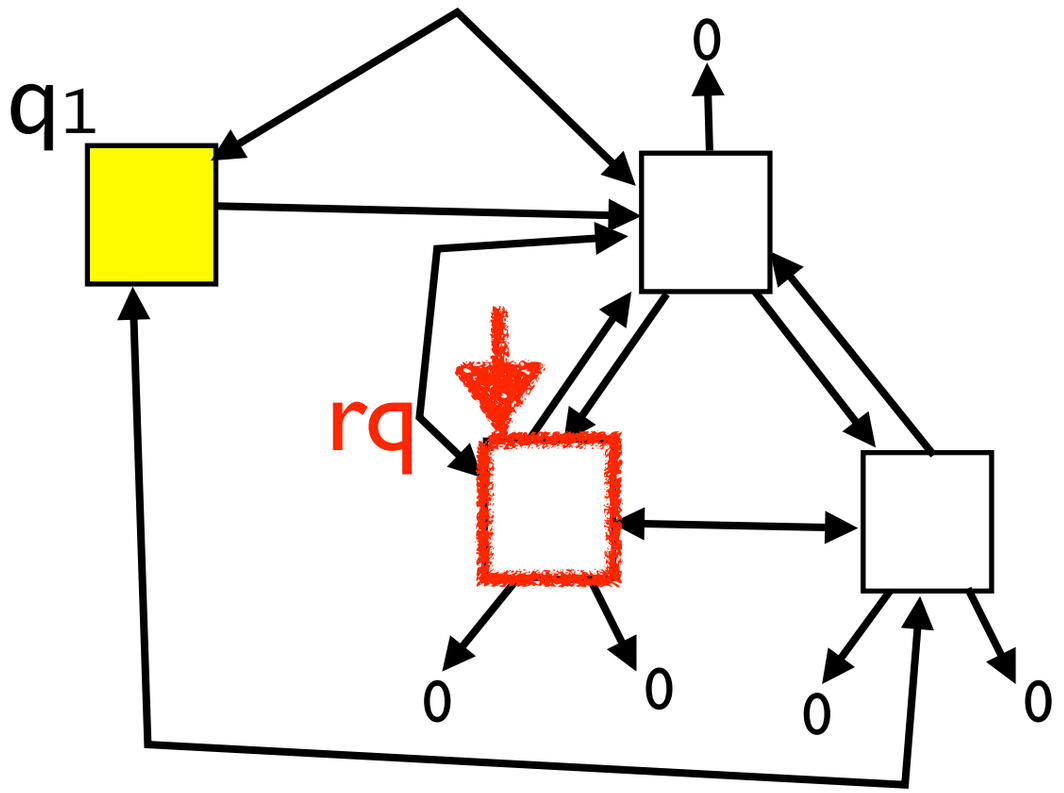


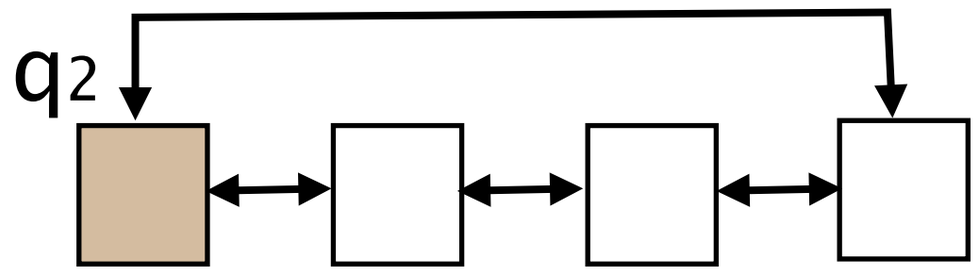




$$(q_1 \mapsto \{\texttt{root}: a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}: c', \texttt{prev}: d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```

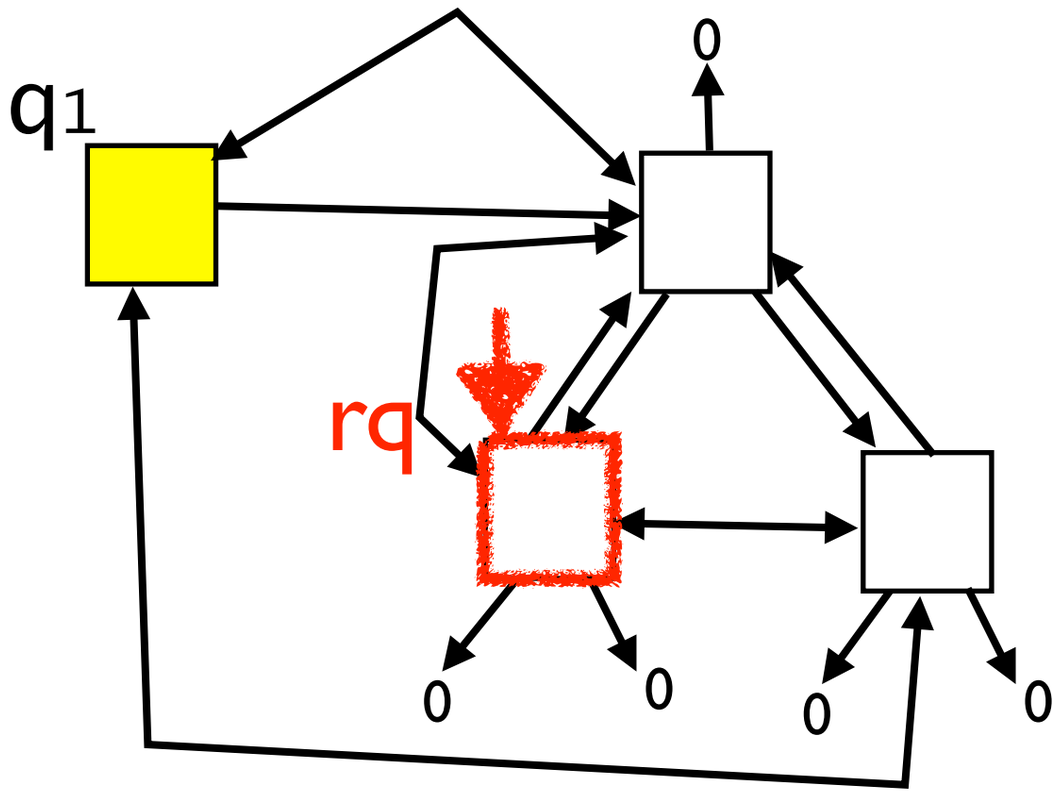


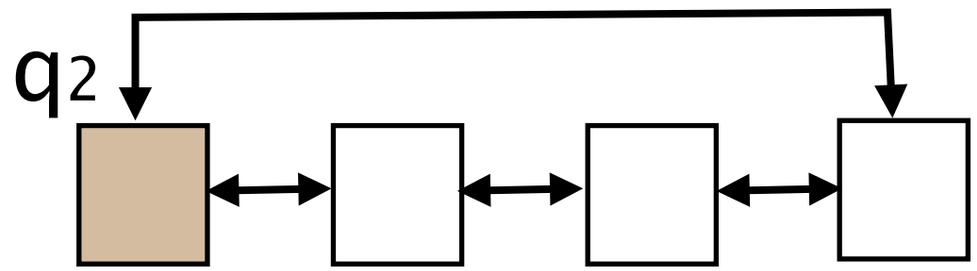




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq);  moveInfo(2,1,rq);
tree_del(q1,rq);  transfer(rq, δ);
list_insert(q2,rq);
```

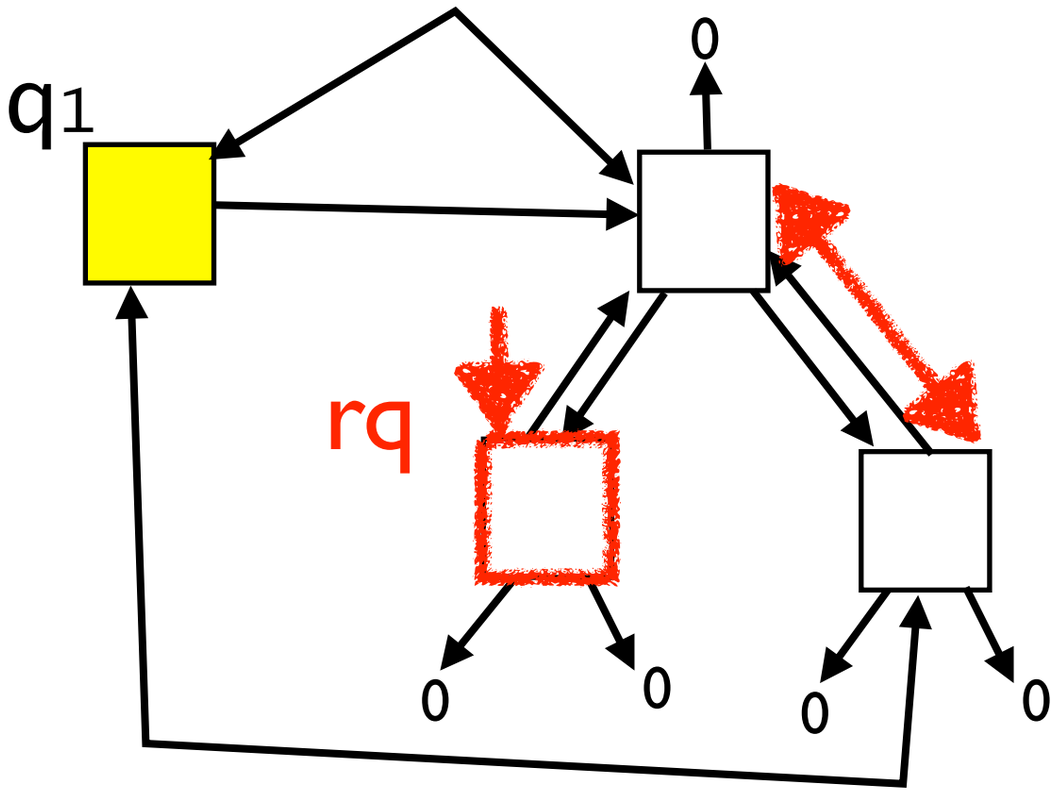


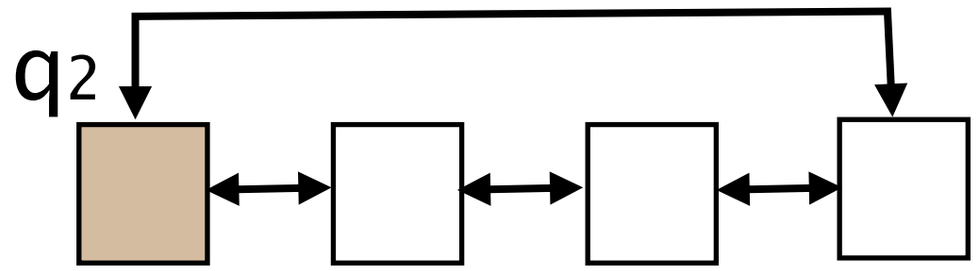




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq);  moveInfo(2,1,rq);
tree_del(q1,rq);  transfer(rq, δ);
list_insert(q2,rq);
```

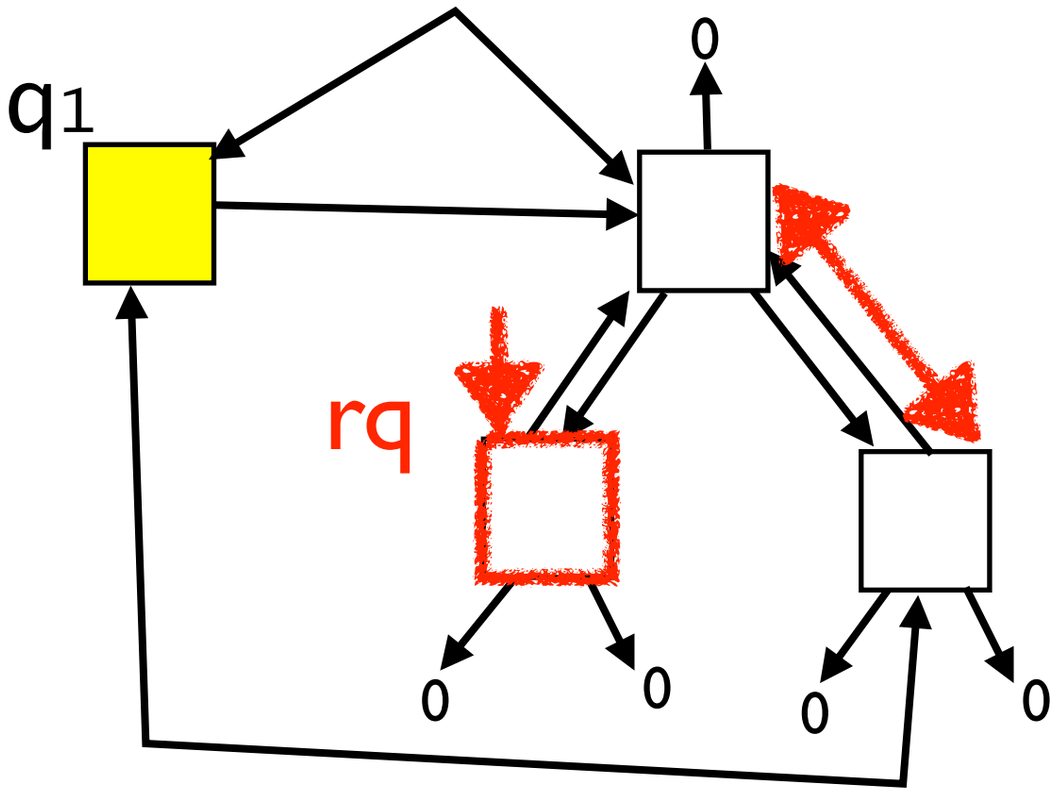


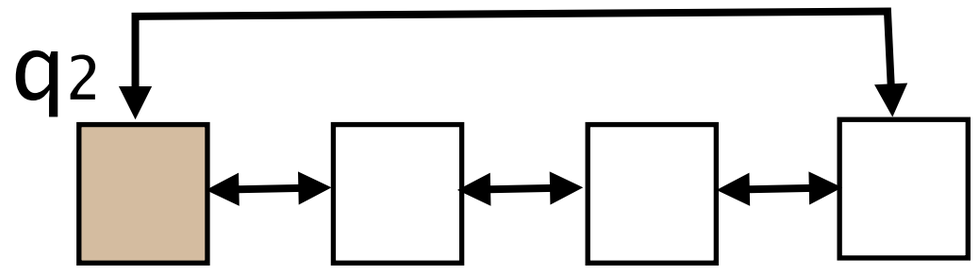




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq);  moveInfo(2,1,rq);
tree_del(q1,rq);  transfer(rq, δ);
list_insert(q2,rq);
```

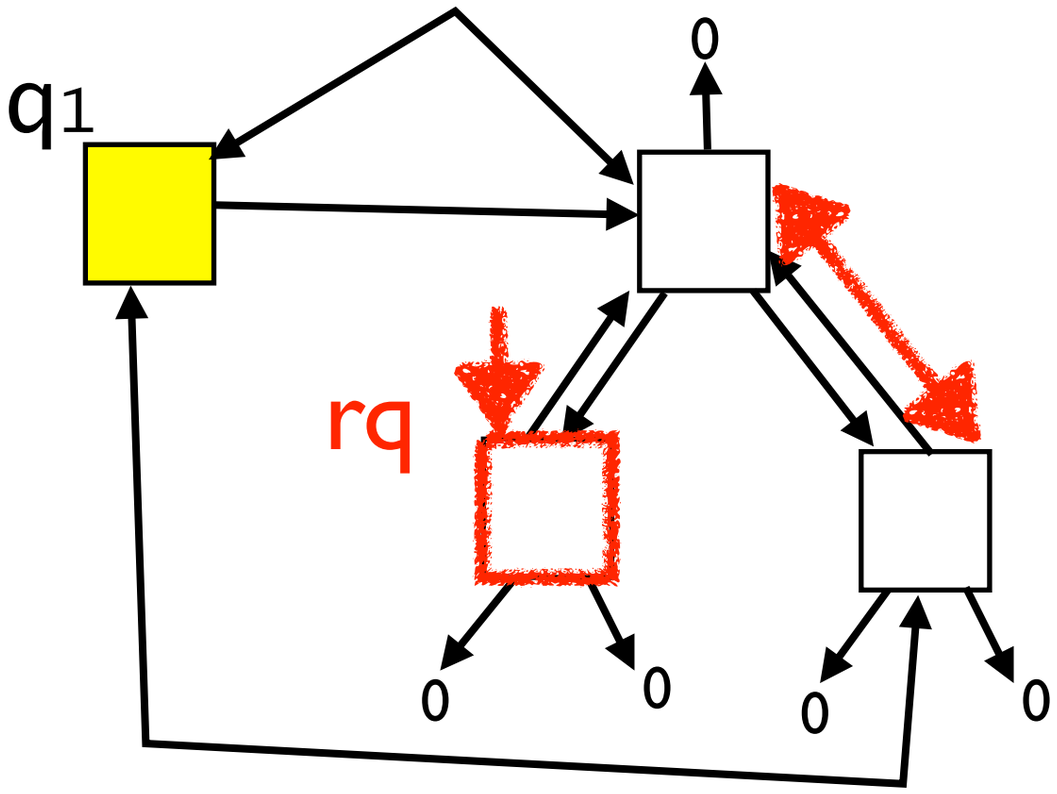


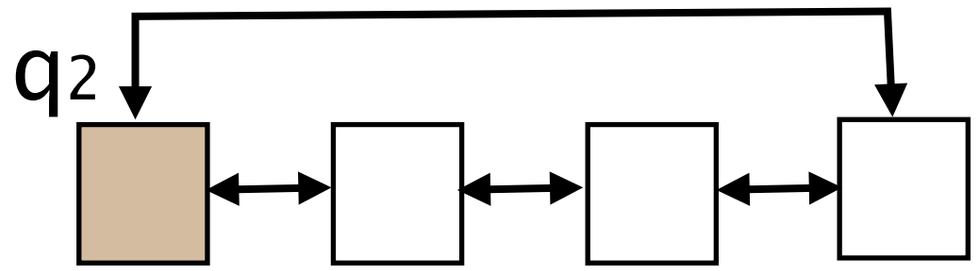




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

```
rq=find(q1,key);
list_del(q1,rq);  moveInfo(2,1,rq);
tree_del(q1,rq);  transfer(rq, δ);
list_insert(q2,rq);
```



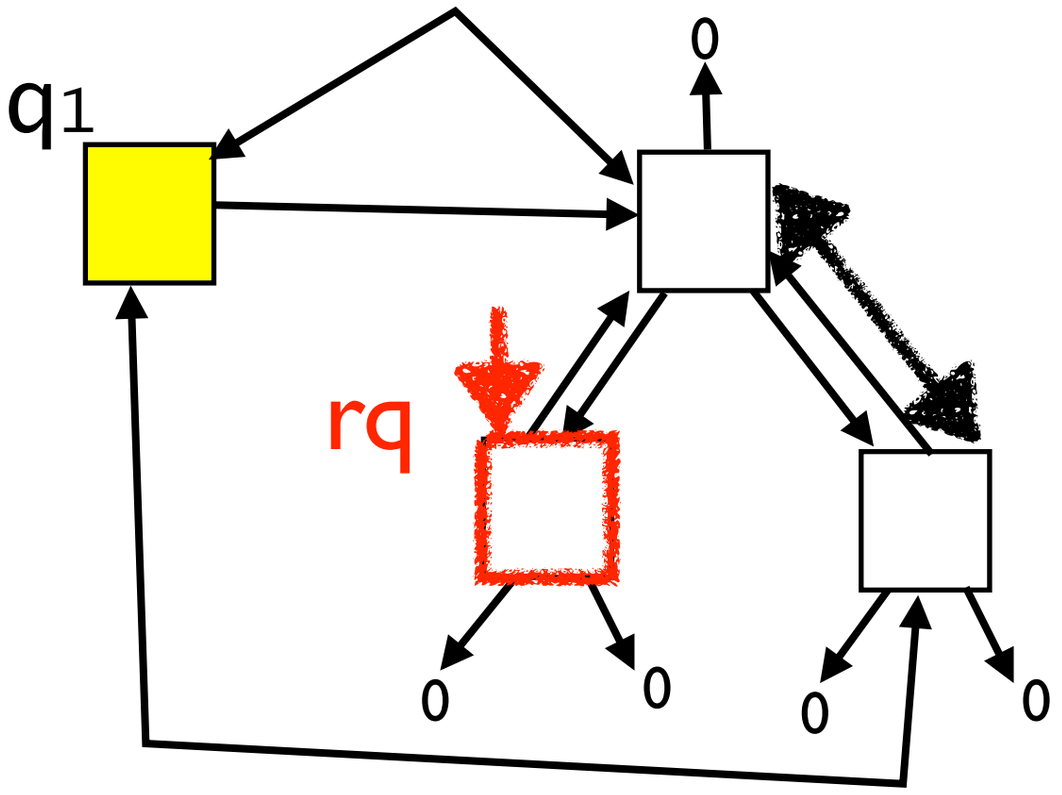



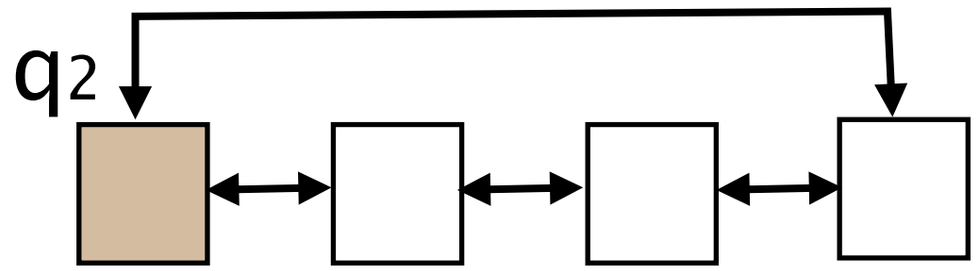

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

$$\mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{\texttt{next}:d_1',\texttt{prev}:c_1'\}_\beta$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



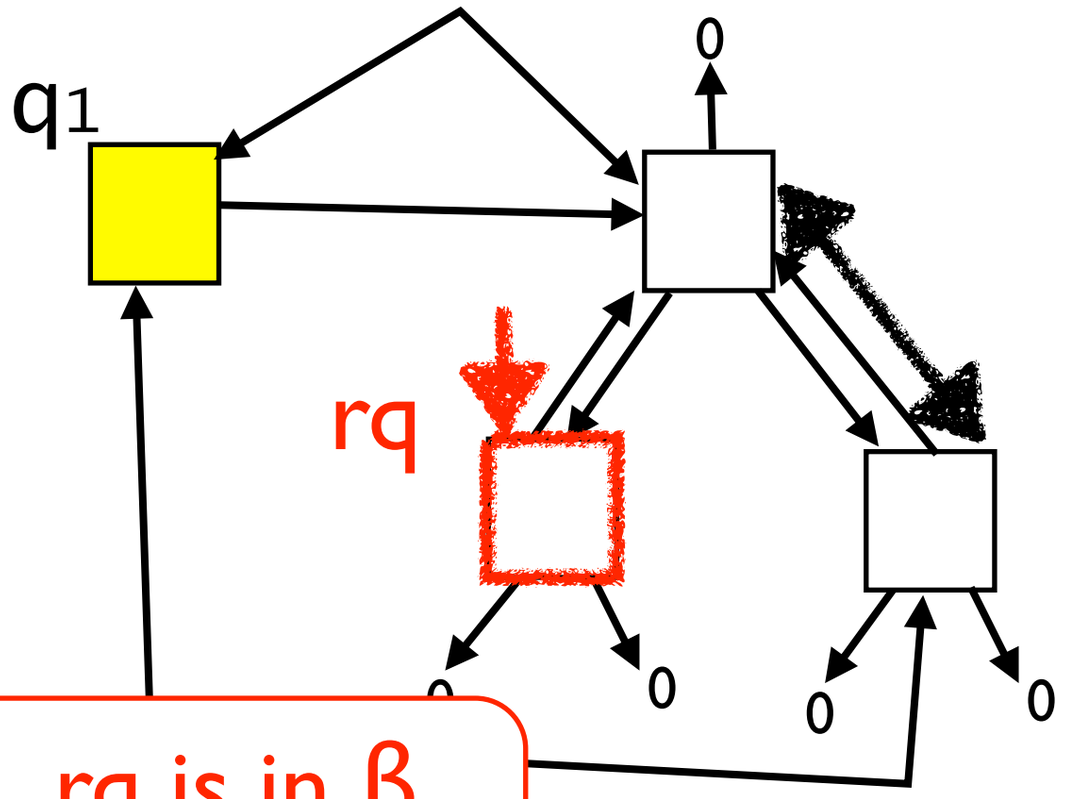



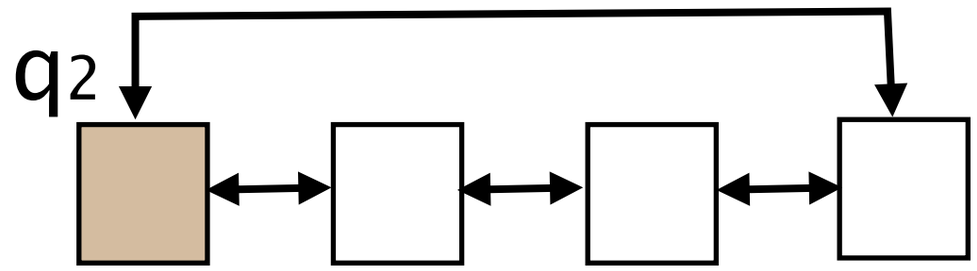

$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

$$\mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{\texttt{next}:d_1',\texttt{prev}:c_1'\}_\beta$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



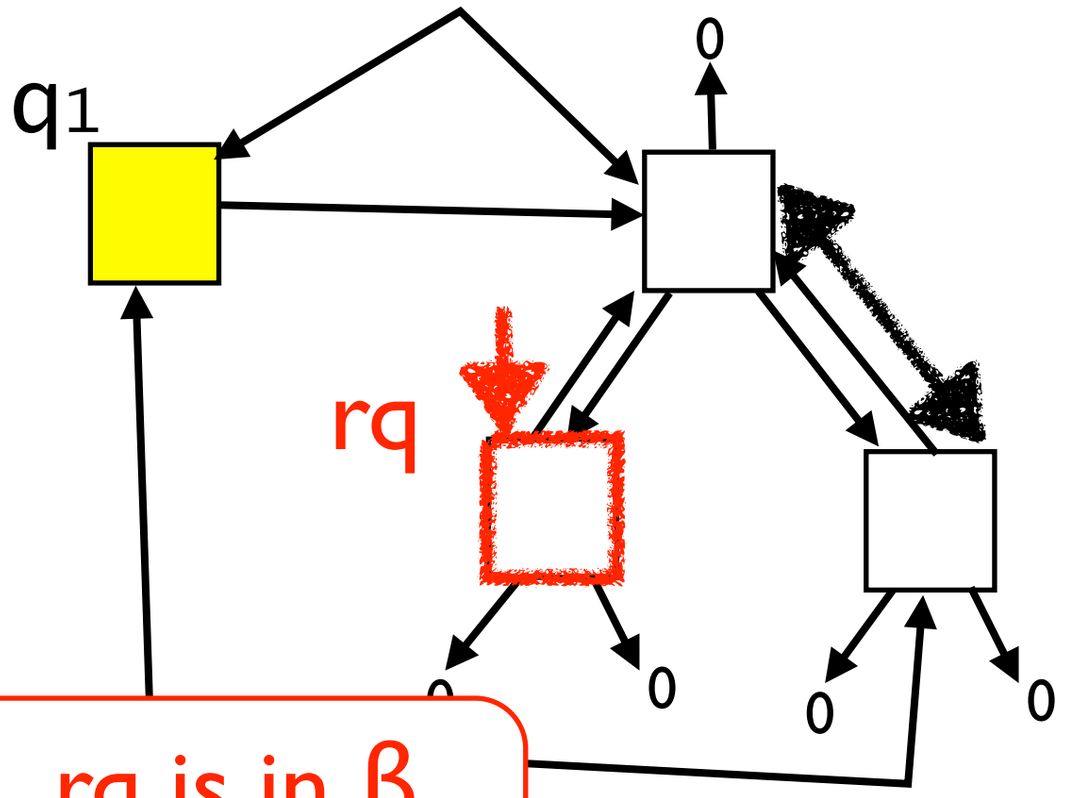

rq is in β

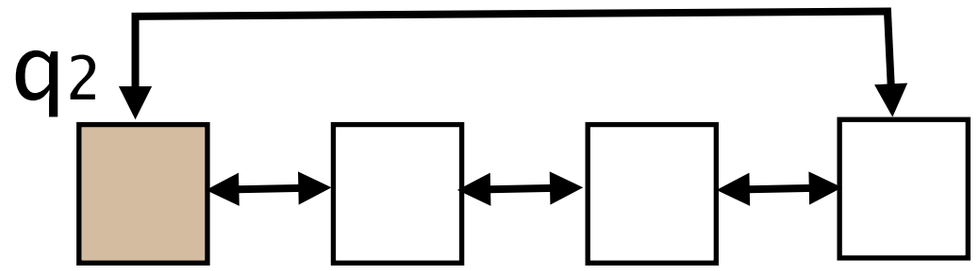
$(q_1 \mapsto \{\text{root}: a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$

$(q_1 \mapsto \{\text{next}: c', \text{prev}: d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$

$\text{ls}_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{\text{next}: d_1', \text{prev}: c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



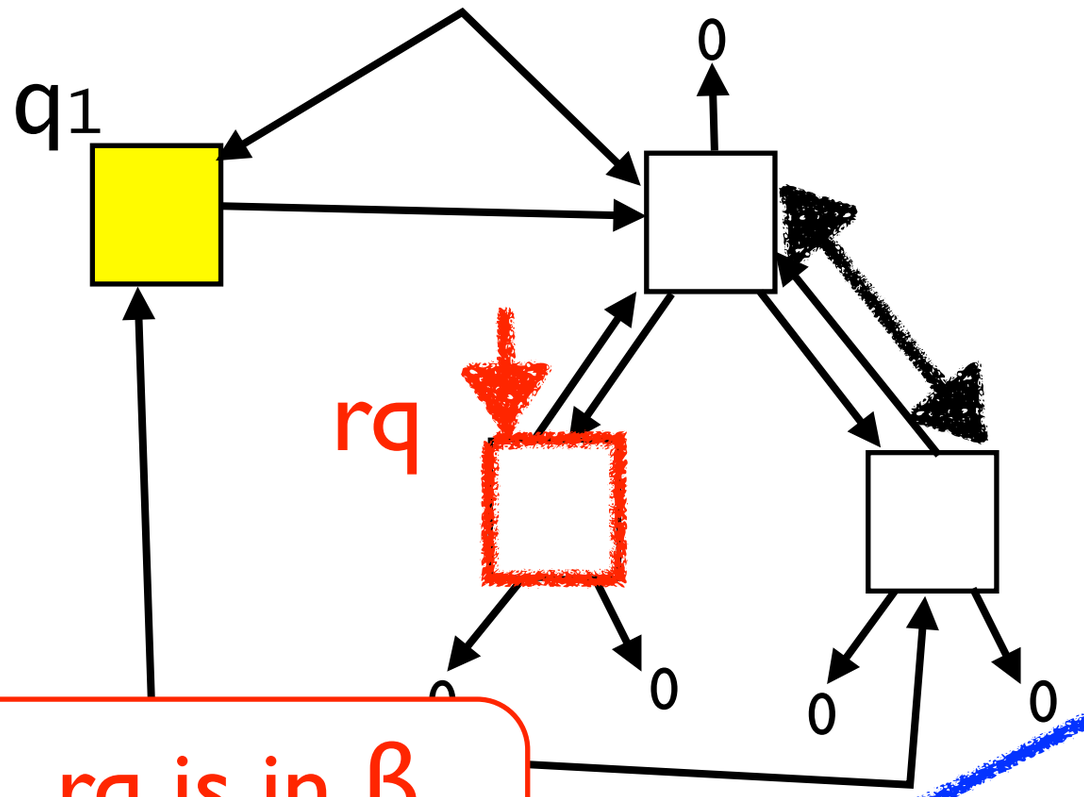

rq is in β

$(q_1 \mapsto \{\text{root}: a'\}_\alpha * \text{tree}_{pe}(0,a',b',0)_\beta * \text{true}_\gamma) \quad \wedge$

$(q_1 \mapsto \{\text{next}: c', \text{prev}: d'\}_\alpha * \text{ls}_{pe}(q_1,c',d',q_1)_\beta * \text{ls}_{ne}(e',q_2,e',q_2)_\gamma)$

$\text{ls}_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{\text{next}: d_1', \text{prev}: c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




rq is somewhere in $tree_{pe}(0,a',b',0)_\beta$

$tree_{pe}(0,a',a_1',rq)_\beta$

$* \; rq \mapsto \{p:a_1', lrev:b_1', lrev:b_2'\}_\beta$

$* \; \ldots$

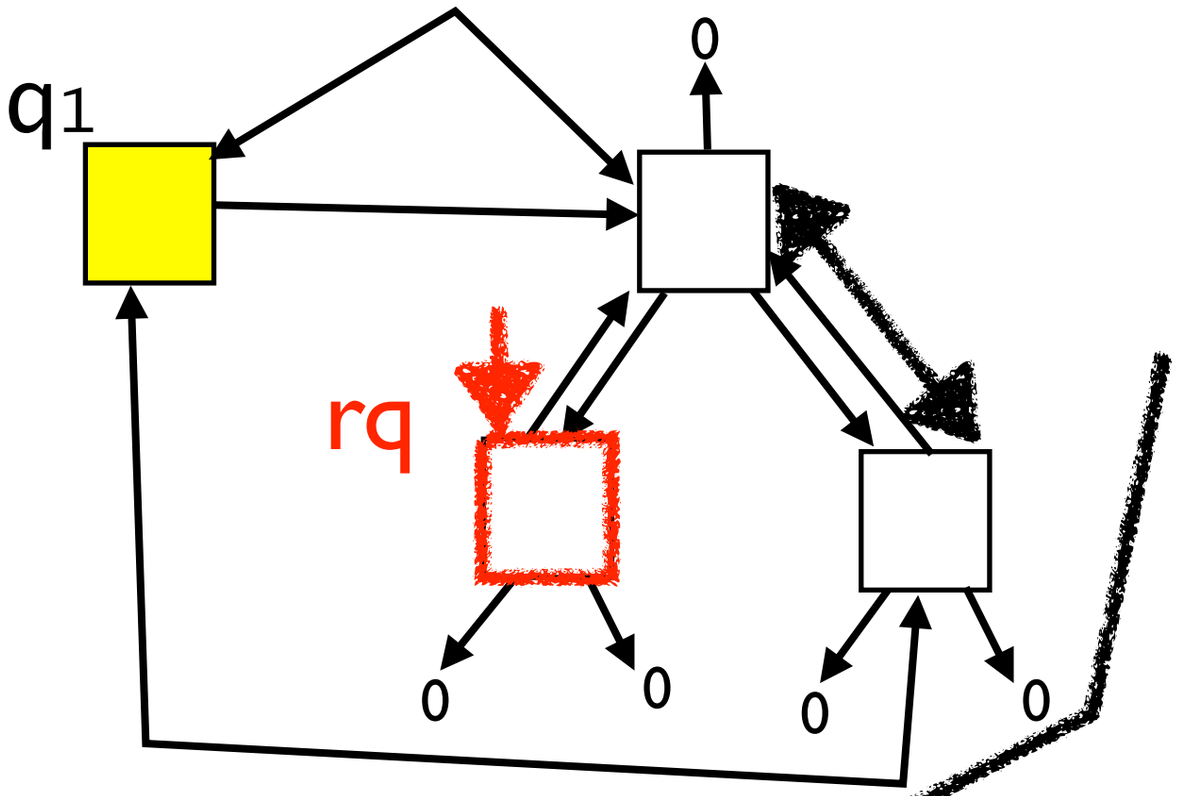
rq is in β

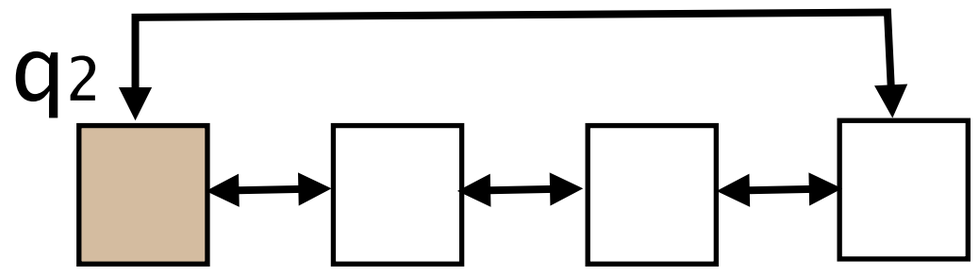
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



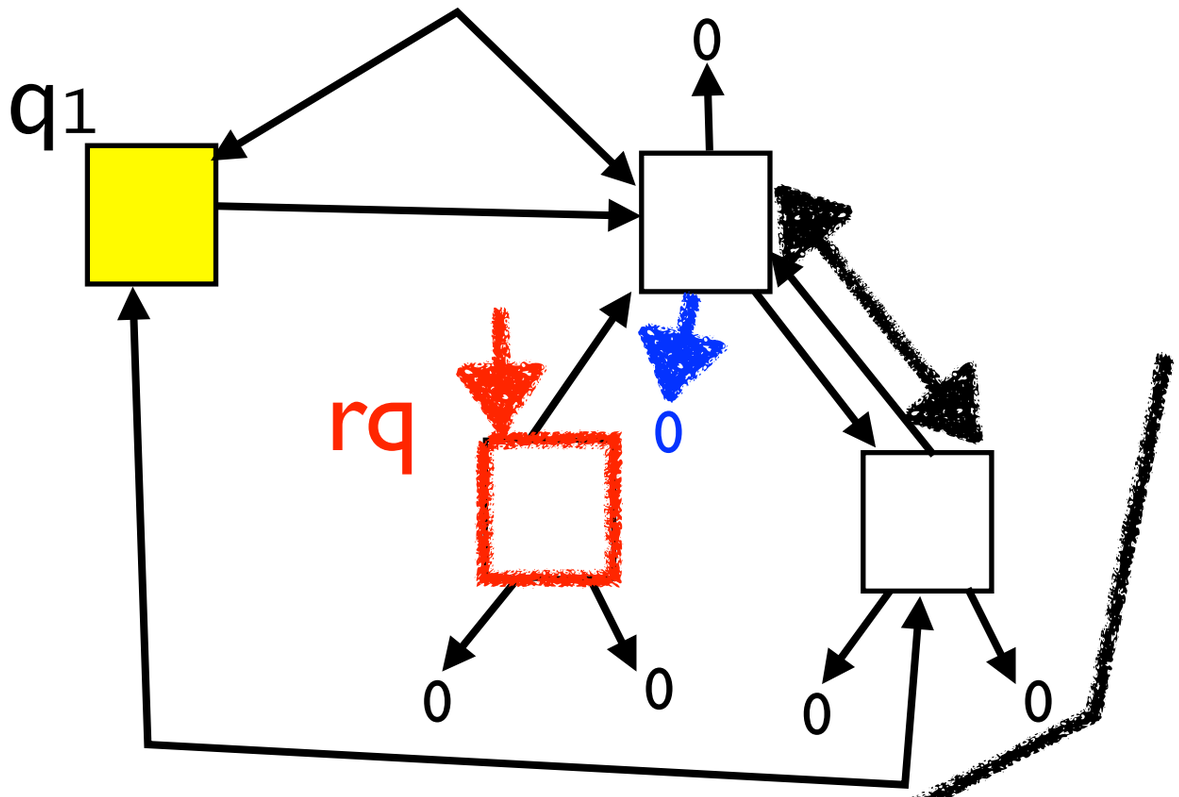

$$\begin{aligned}
&\mathrm{tree}_{pe}(0,a',a_1',rq)_\beta \\
&* rq\mapsto\{p:a_1',lrev:b_1',lrev:b_2'\}_\beta \\
&* \ldots
\end{aligned}$$

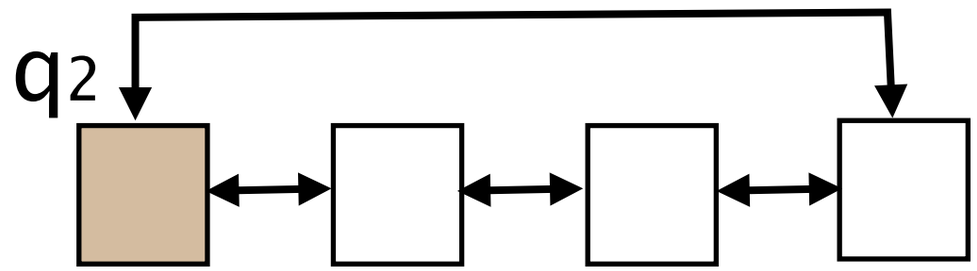
$$(q_1\mapsto\{root:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1\mapsto\{next:c',prev:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

$$\mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * rq\mapsto\{next:d_1',prev:c_1'\}_\beta$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



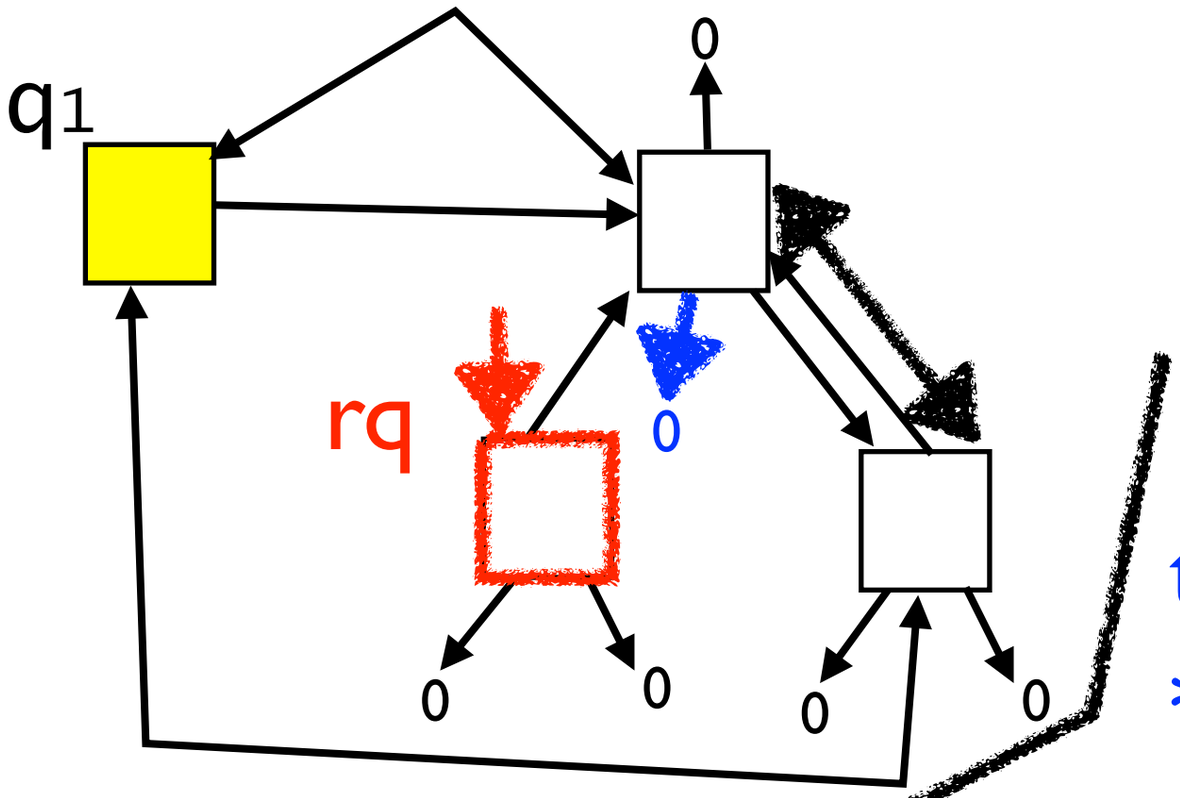

$$\begin{aligned}
&\mathrm{tree}_{pe}(0,a',a_1',rq)_\beta \\
&* rq\mapsto\{p:a_1',\mathrm{lrev}:b_1',\mathrm{lrev}:b_2'\}_\beta \\
&* \ldots
\end{aligned}$$

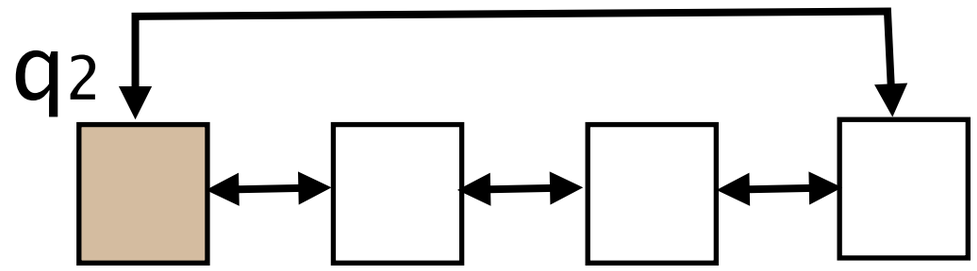
$$(q_1\mapsto\{\mathrm{root}:a'\}_\alpha * \mathrm{tree}_{pe}(0,a',b',0)_\beta * \mathrm{true}_\gamma) \quad \wedge$$

$$(q_1\mapsto\{\mathrm{next}:c',\mathrm{prev}:d'\}_\alpha * \mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathrm{ls}_{ne}(e',q_2,e',q_2)_\gamma)$$

$$\mathrm{ls}_{pe}(q_1,c',d',q_1)_\beta * rq\mapsto\{\mathrm{next}:d_1',\mathrm{prev}:c_1'\}_\beta$$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

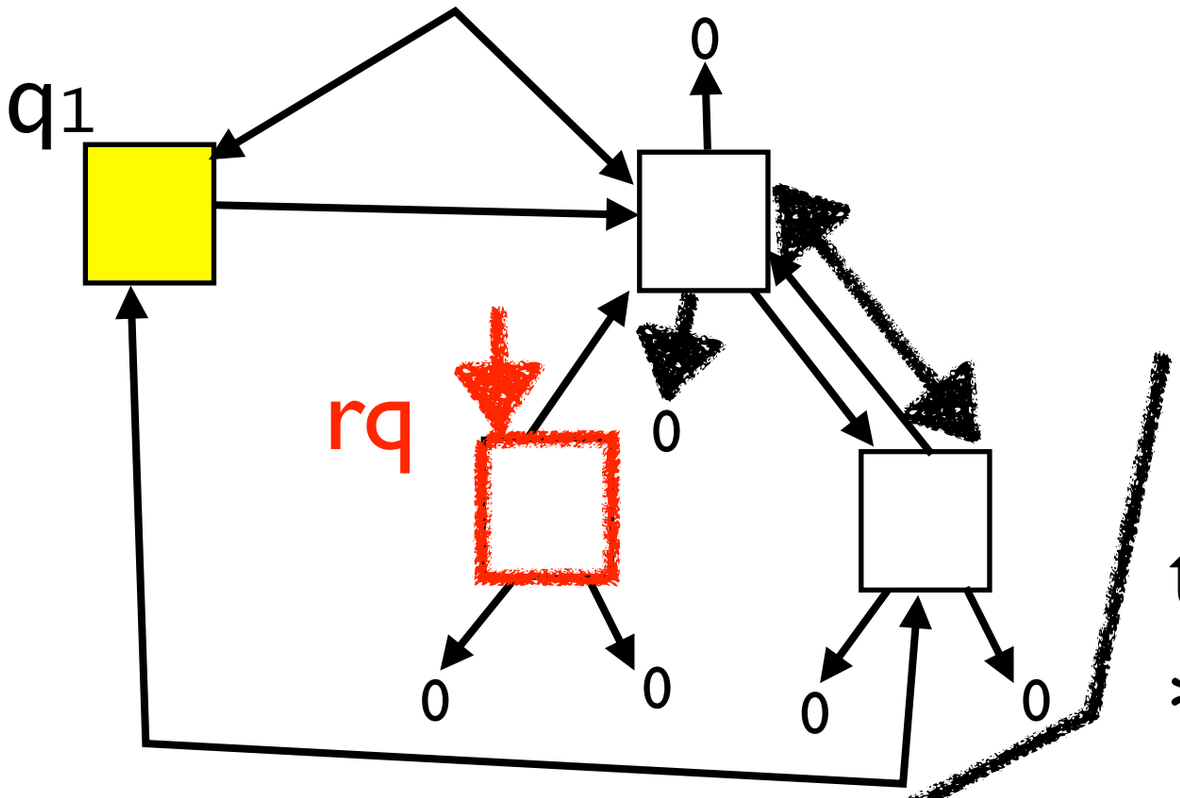
$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\beta$

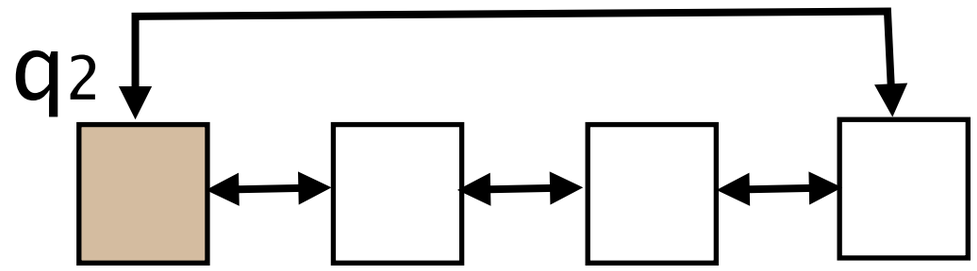
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```



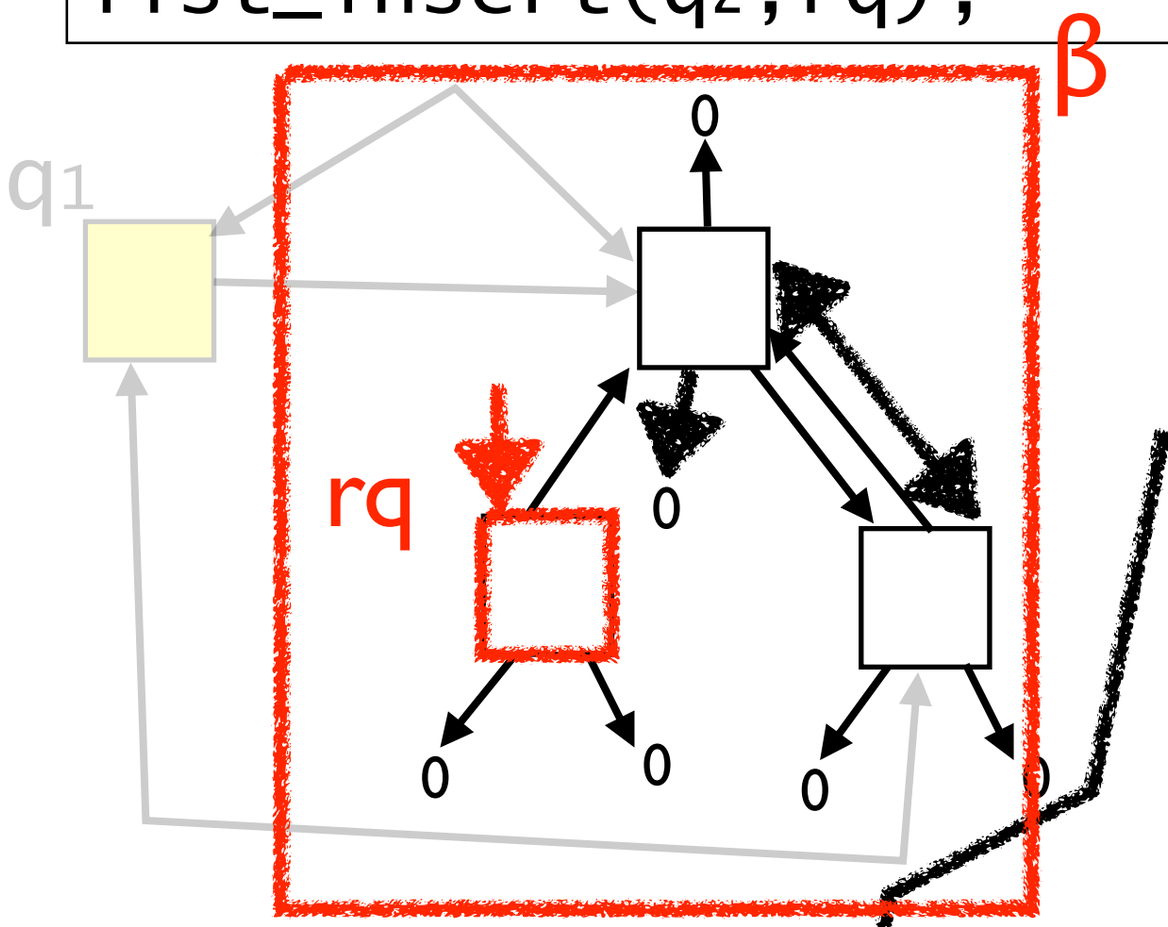

$\mathsf{tree}_{pe}(0,a',b',0)_\beta$
$* \; rq \mapsto \{p{:}a_1', {}_1\mathsf{rev}{:}b_1', {}_1\mathsf{rev}{:}b_2'\}_\beta$

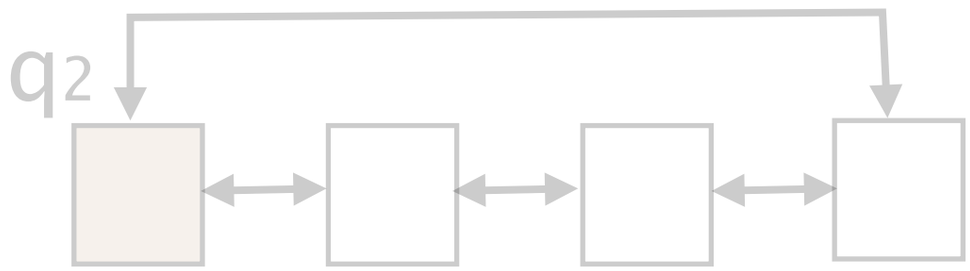
$(q_1 \mapsto \{\mathtt{root}{:}a'\}_\alpha * \mathsf{tree}_{pe}(0,a',b',0)_\beta * \mathsf{true}_\gamma) \quad \wedge$

$(q_1 \mapsto \{\mathtt{next}{:}c', \mathtt{prev}{:}d'\}_\alpha * \mathsf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathsf{ls}_{ne}(e',q_2,e',q_2)_\gamma)$

$\mathsf{ls}_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{\mathtt{next}{:}d_1', \mathtt{prev}{:}c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

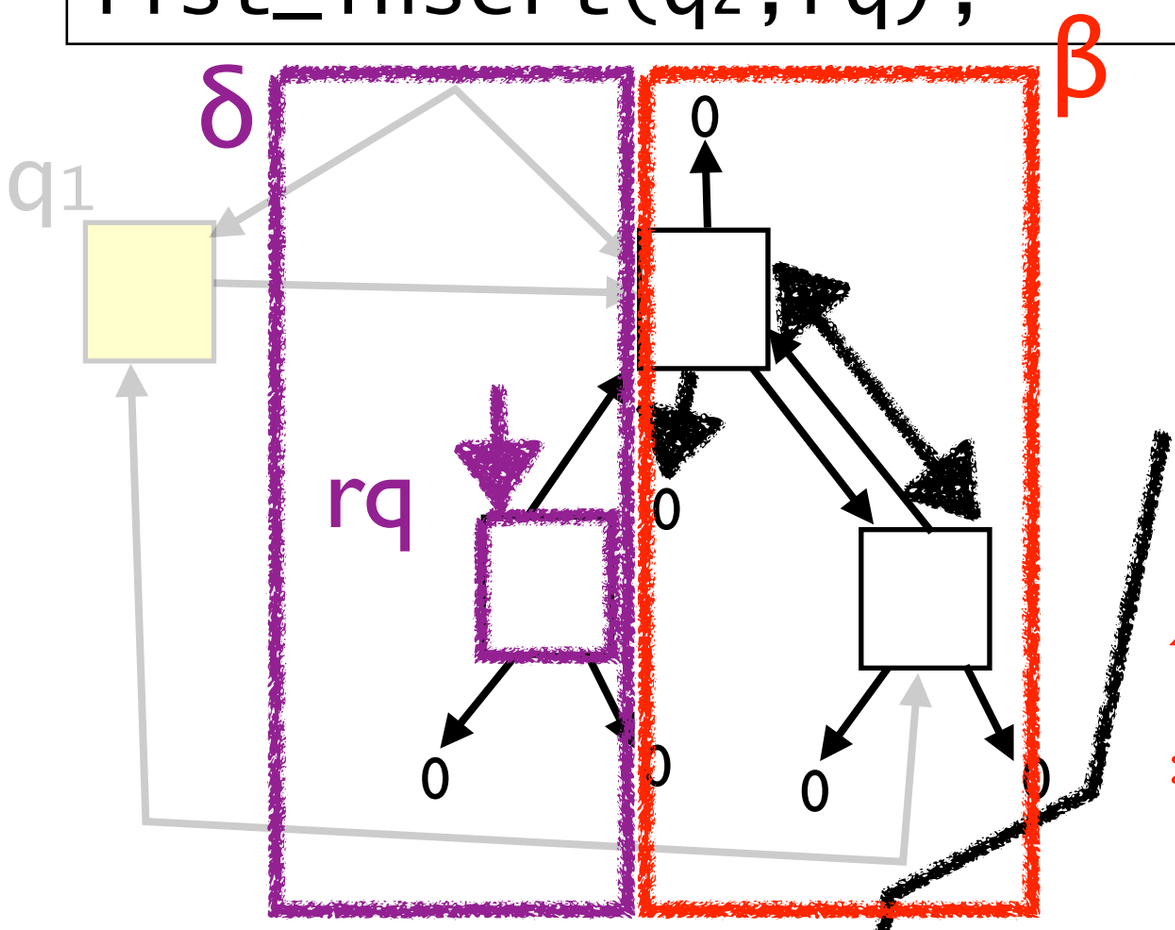
$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\beta$

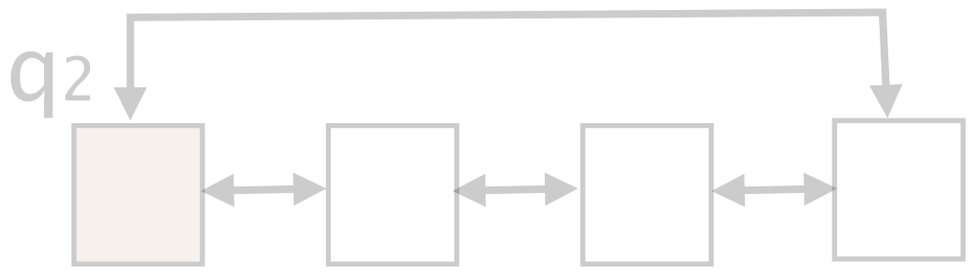
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

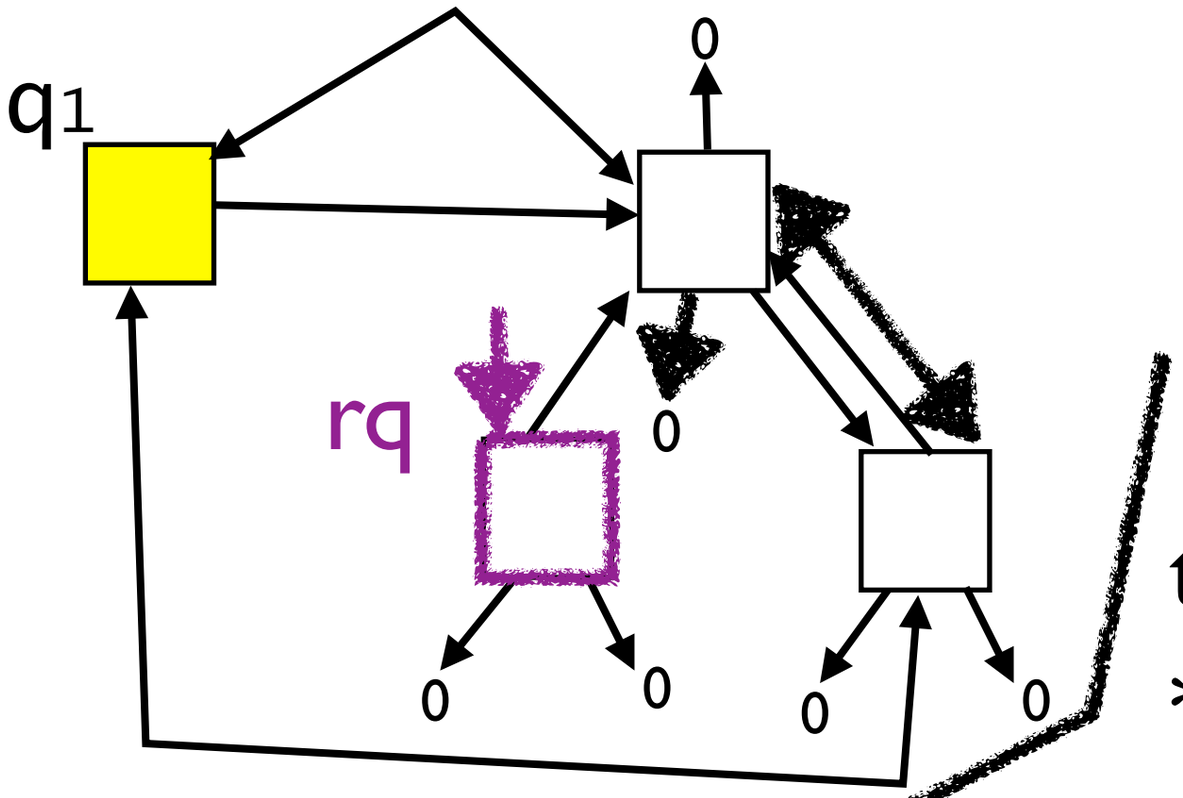
$\wedge$

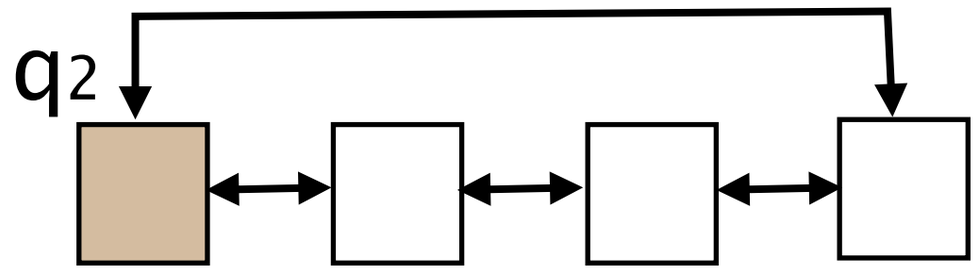
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma)$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\delta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

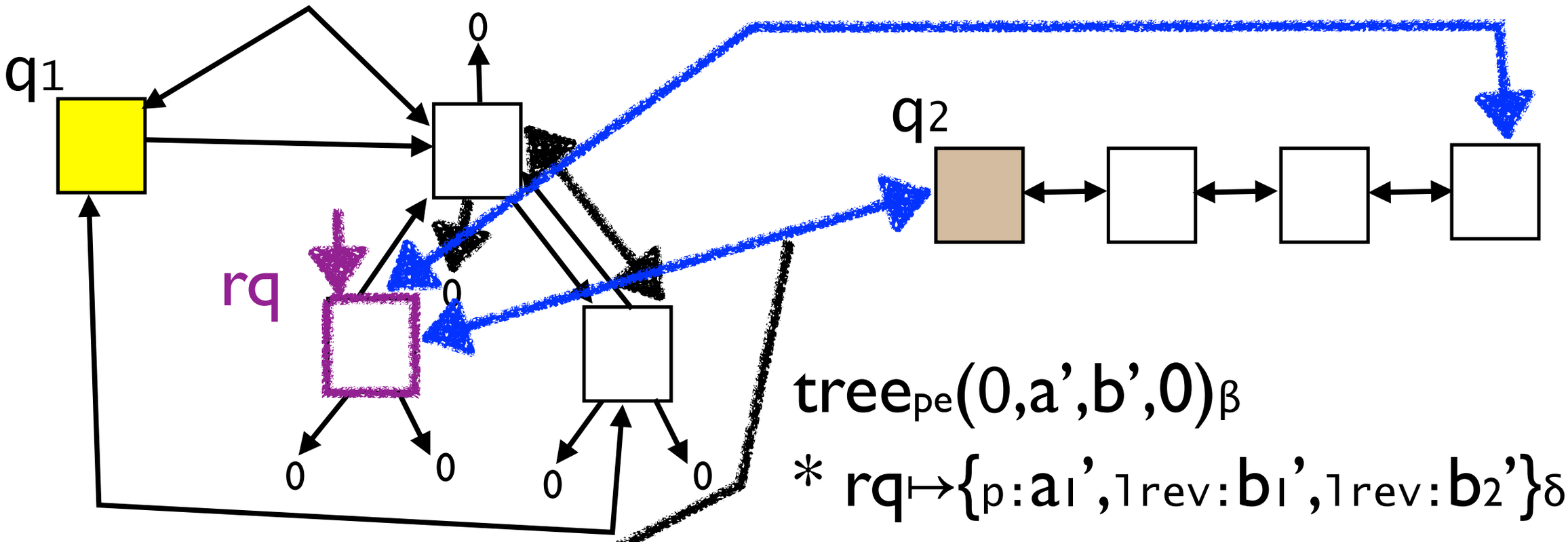
$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\delta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

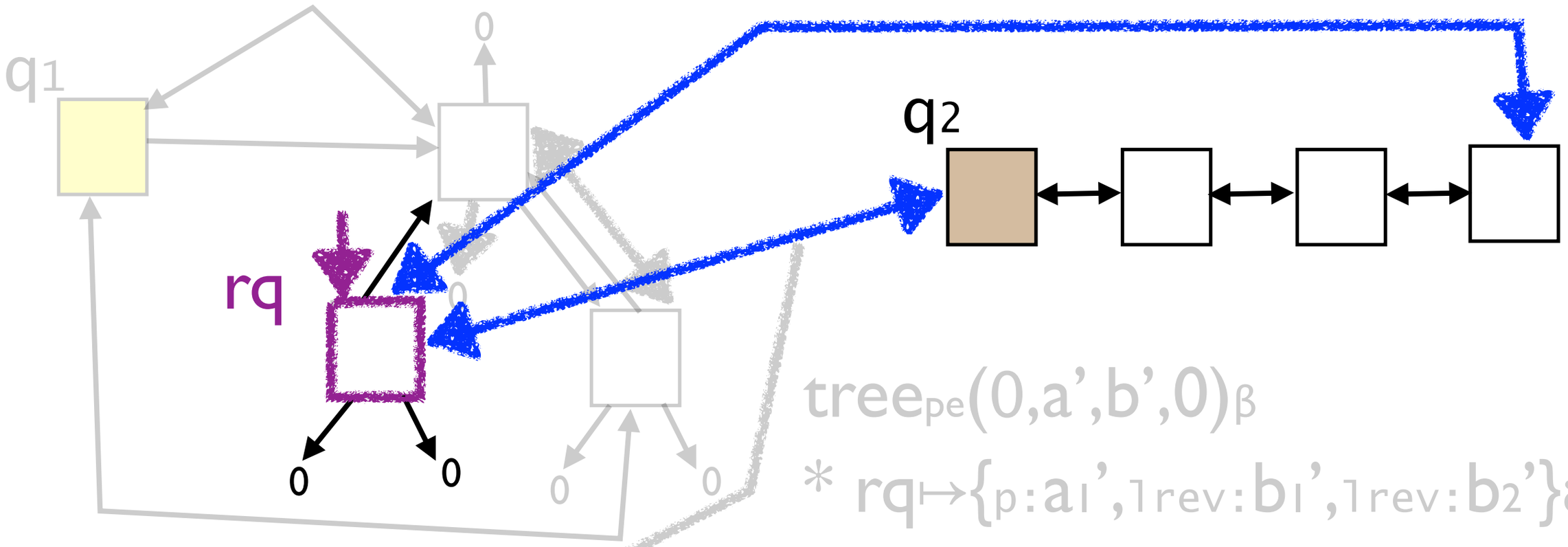
$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

$(q_1 \mapsto \{root:a'\}_\alpha * \cancel{tree_{pe}(0,a',b',0)_\beta} * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * \cancel{ls_{pe}(q_1,c',d',q_1)_\beta} * ls_{ne}(e',q_2,e',q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:d_1', prev:c_1'\}_\delta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




$tree_{pe}(0,a',b',0)_\beta$

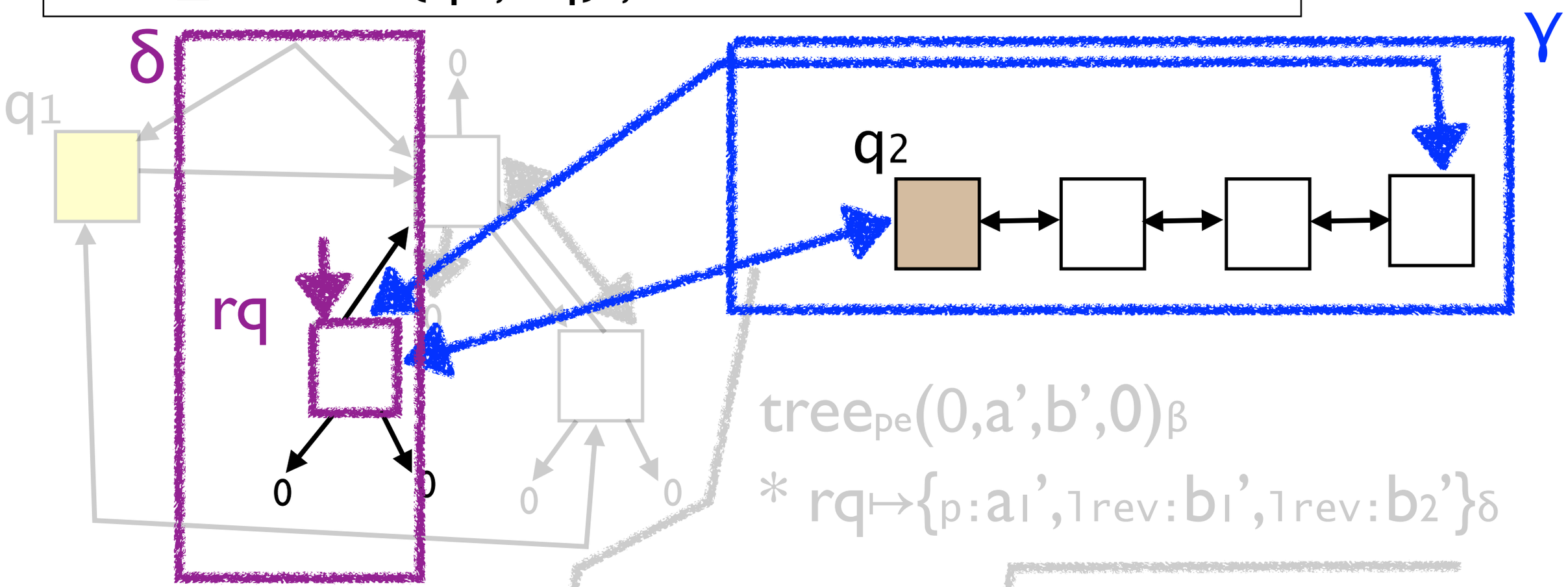
$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,e',rq)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:q_2, prev:e'\}_\delta$

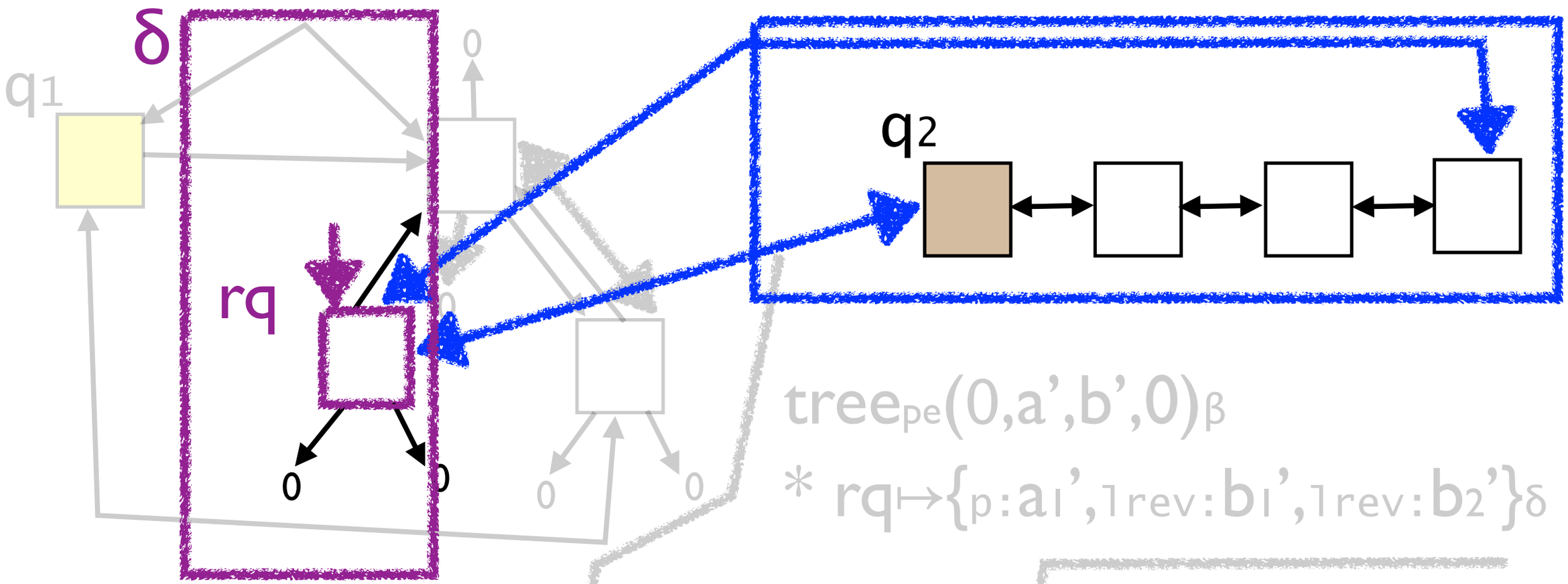
```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq);
```




Cannot be abstracted to a list, because they belong to different partitions.

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); γ←γ ∪ δ;
```

Merge γ, δ




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

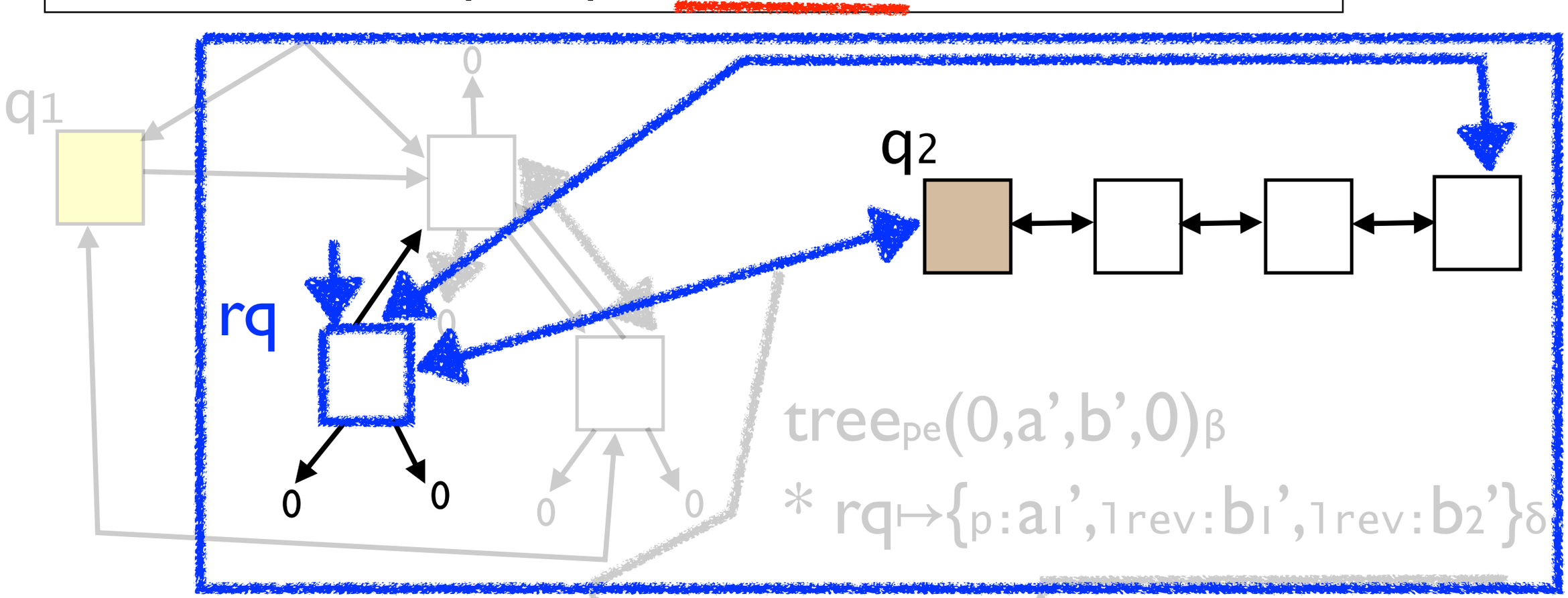
$* \; rq \mapsto \{next:q_2, prev:e'\}_\delta$

$* \; ls_{ne}(rq,q_2,e',rq)_\gamma)$

Cannot be abstracted to a list, because they belong to different partitions.

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); γ←γ ∪ δ;
```

Merge γ, δ




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', {}_1rev:b_1', {}_1rev:b_2'\}_\delta$

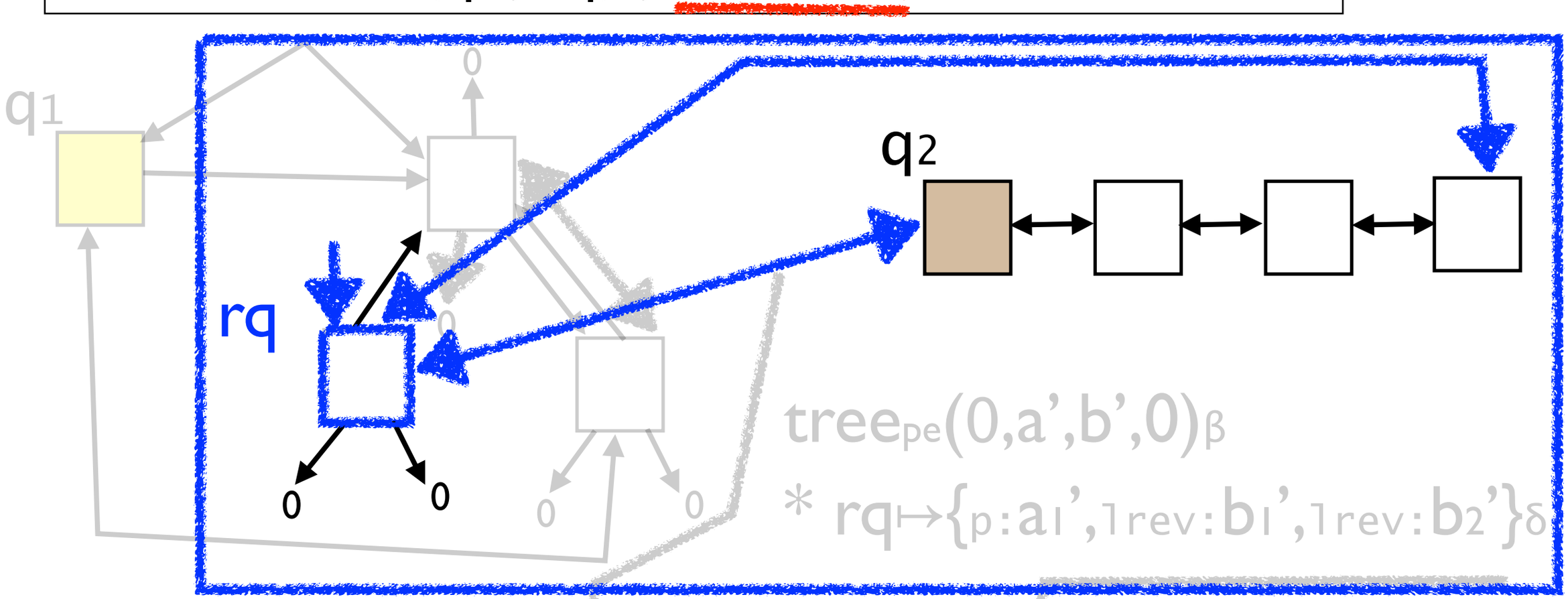
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \;\wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,e',rq)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta * rq \mapsto \{next:q_2, prev:e'\}_\gamma$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); γ←γ ∪ δ;
```

Merge γ, δ




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', \overleftarrow{next}:b_1', \overleftarrow{next}:b_2'\}_\delta$

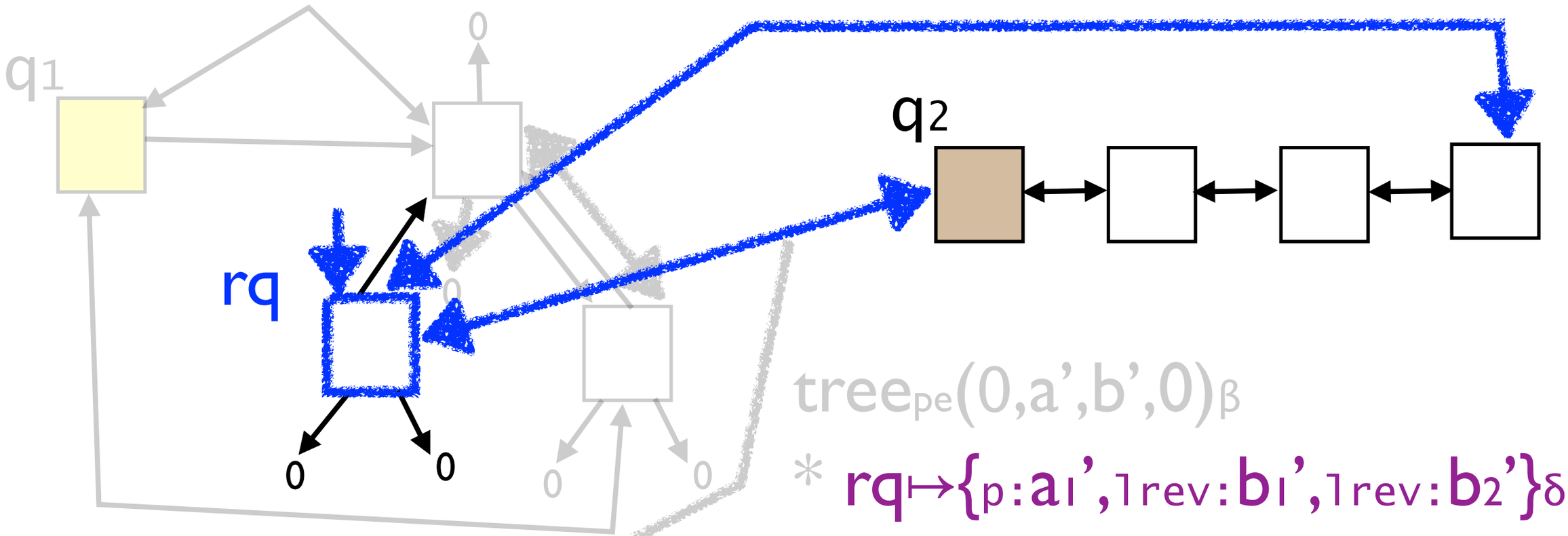
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \;\wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,rq,q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); ϒ←ϒ ∪ δ;
```

Merge ϒ, δ




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', lrev:b_1', lrev:b_2'\}_\delta$

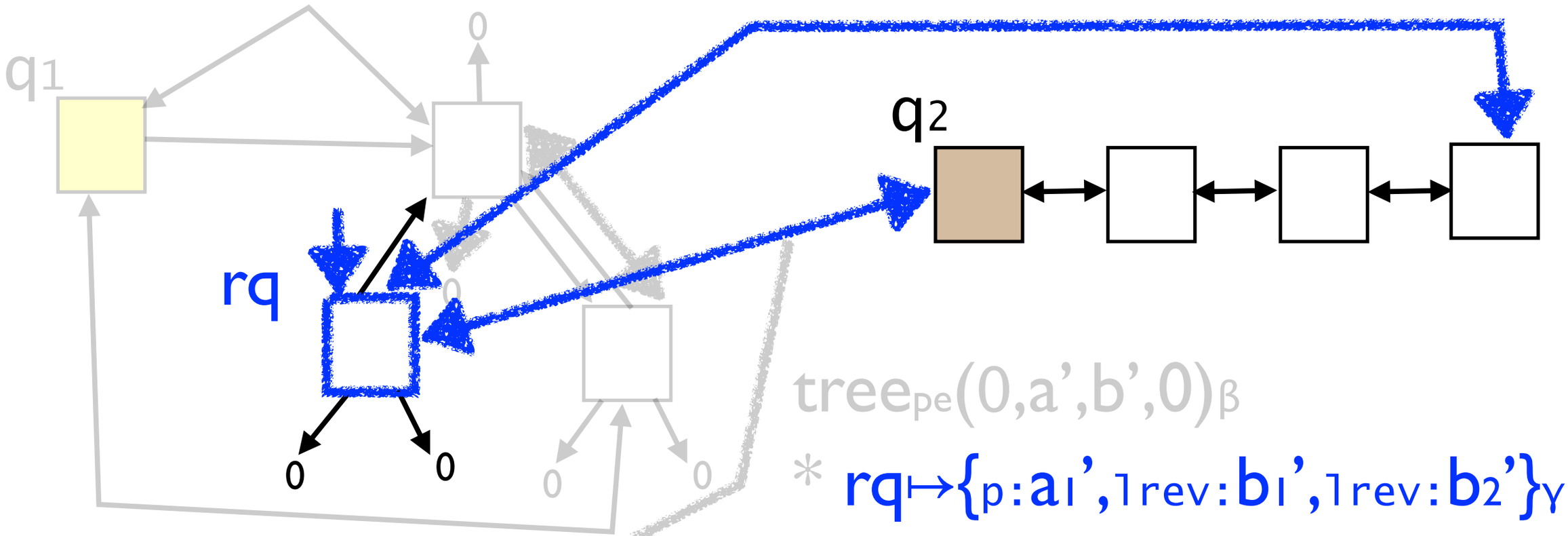
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,rq,q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); ϒ←ϒ ∪ δ;
```

Merge ϒ, δ




$tree_{pe}(0,a',b',0)_\beta$

$* \; rq \mapsto \{p:a_1', lrev:b_1', lrev:b_2'\}_\gamma$

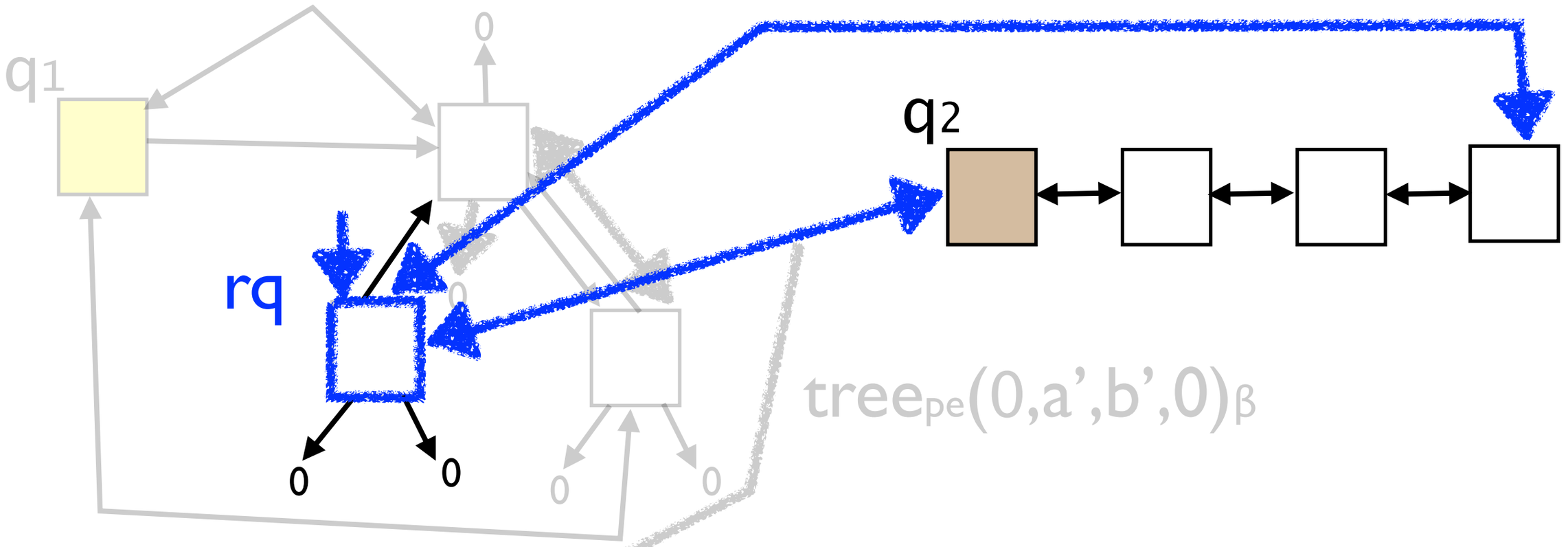
$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \quad \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,rq,q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); ϒ←ϒ ∪ δ;
```

Merge ϒ, δ



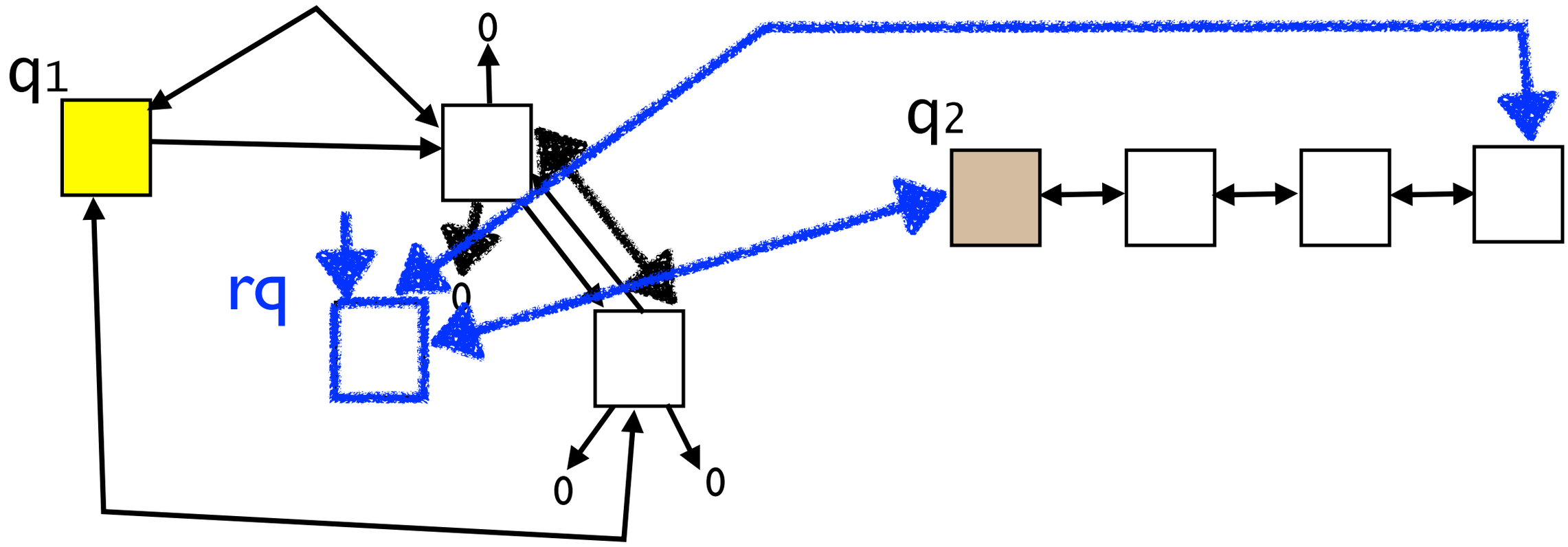

$tree_{pe}(0,a',b',0)_\beta$

$(q_1 \mapsto \{root:a'\}_\alpha * tree_{pe}(0,a',b',0)_\beta * true_\gamma) \wedge$

$(q_1 \mapsto \{next:c', prev:d'\}_\alpha * ls_{pe}(q_1,c',d',q_1)_\beta * ls_{ne}(rq,q_2,rq,q_2)_\gamma)$

$ls_{pe}(q_1,c',d',q_1)_\beta$

```
rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq, δ);
list_insert(q2,rq); ϒ←ϒ ∪ δ;
```




$$(q_1 \mapsto \{\texttt{root}:a'\}_\alpha * \mathbf{tree}_{pe}(0,a',b',0)_\beta * \mathbf{true}_\gamma) \quad \wedge$$

$$(q_1 \mapsto \{\texttt{next}:c',\texttt{prev}:d'\}_\alpha * \mathbf{ls}_{pe}(q_1,c',d',q_1)_\beta * \mathbf{ls}_{ne}(rq,q_2,rq,q_2)_\gamma)$$

# Abstract operators

$$\text{Dom} = \text{P}(\text{TreeForm}) \times \text{P}(\text{ListForm}) \cup \{\top\}$$

$$\text{Form} = \text{TreeForm} \cup \text{ListForm}$$

- $\text{moveInfo}(2,1,x) : \text{Dom} \to \text{Dom}$
- $\text{transfer}(x,\alpha) : \text{Form} \to \text{Form} \cup \{\top\}$
- $\alpha \leftarrow \beta \cup \alpha : \text{Form} \to \text{Form}$
- Also, other usual operators. Namely, abstraction function, join, and abstract transfer functions.

# moveInfo(2,1,x)

- Transfers information about the cell x from the list part to the tree part.
- moveInfo(x)( $(T_1 \vee T_2) \wedge (L_1 \vee L_2)$ ) =

  $\underline{\text{let}}\ A = \cup_i \text{collect}(x, L_i)\ \ \underline{\text{in}}\ \ (\vee_i \text{case}(x,A,T_i)) \wedge (L_1 \vee L_2)$

- collect(x,Li) computes a set of partitions containing x:

  $\text{collect}(x, L_i)=A_i \implies L_i \vdash (\text{alloc}(x) \wedge x \in \cup A_i)$

- case(x,A,Ti) materialises cell x for $T_i$:

  $\text{case}(x,A,T_i)= (\vee_j T_j') \implies (T_i \wedge \text{alloc}(x) \wedge x \in A) \vdash \vee_j T_j'$

# transfer(x,α)

transfer(x,α) : Form → Form ∪ {⊤}

- Changes the partition tag of x to α.
- transfer(x,α)(F) =
  match F with
  | ... * x↦{..}γ => ... * x↦{..}α
  | _ => ⊤
- Used to take x out of an existing partition.

# $\alpha \leftarrow \beta \cup \alpha$

$\alpha \leftarrow \beta \cup \alpha \quad : \quad \text{Form} \rightarrow \text{Form}$

- $(\alpha \leftarrow \beta \cup \alpha)(F) = (F[\alpha / \beta])$
- Implements the merging of two partitions.
- Introduced on the fly during abstraction.

# Inserting `moveInfo` and `transfer`

Use forward and backward pre-analyses to find out places to insert `moveInfo` and `transfer`.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq); ϒ←ϒ ∪ δ;      }
```

# Inserting `moveInfo` and `transfer`

Use forward and backward pre-analyses to find out places to insert `moveInfo` and `transfer`.

```
void move(key,q1,q2) {
  request* rq=find(q1,key);
  list_del(q1,rq); moveInfo(2,1,rq);
  tree_del(q1,rq); transfer(rq,δ);
  list_insert(q2,rq); ϒ←ϒ ∪ δ;      }
```

Both tree and list parts contain rq↦{...}.

# Inserting `moveInfo` and `transfer`

Use forward and backward pre-analyses to find out places to insert `moveInfo` and `transfer`.

Only the list part contains info on rq, but the tree part needs info on rq.

```
request* rq=find(q1,key);
list_del(q1,rq); moveInfo(2,1,rq);
tree_del(q1,rq); transfer(rq,δ);
list_insert(q2,rq); ϒ←ϒ∪δ;        }
```

Both tree and list parts contain rq↦{...}.

# Take-home message 1

- Components of overlaid data structures are loosely correlated.
- Can exploit this loose correlation by running multiple independent analyses with minimal communication among them.
- We realised the minimal communication using *, $\wedge$ and $\alpha$.

# Take-home message 2

- How to make two expensive analyses talk a little bit, without compromising performance too much?
- Think about using ghost variables and designing an algorithm for discovering necessary ghost commands.