# Dynamic Searchable Symmetric Encryption

# DSSE

INTRODUCTION TO CYBER SECURITY

ESTR 4306 | LIU YICUN

## Motivating Problem

The clients want a encrypted database which can update dynamically.

The clients want to do the updates without re-encrypting the whole data base.

The updates should affect the previous DB as less as possible.

Queries and updates are supposed to reveal as less information as possible.

Efficiency vs. Functionality

# Abstract Idea of "Dynamic"

Additions:

- Edge Addition: Simple. Just add edge (x, y) into the DB, noted that node x and y have already existed.
- Node Addition: Based on edge addition. Add a set of new edges (·, y) connecting node y. The client need to keep all edges connecting node y in mind.

Deletions:

- Edge Deletion: Just remove (x, y). Simple, but unrealistic for massive deletion.
- Node Deletion: Client give the name of the node y to the server. The server do the rest of the job. Server traverses (·, y) and deletes all these edges.

# Notations

1. Edge Addition: $u = (\mathsf{Add}, x, y, w)$ adds the edge $(x, y, w)$ to $G$, where $x$ and $y$ are originally not connected to each other. However, $x$ and $y$ can be originally connected to other nodes respectively.

2. Node Addition: $u = (\mathsf{Add}, \mathbf{x}, y, \mathbf{w})$ (or $u = (\mathsf{Add}, x, \mathbf{y}, \mathbf{w})$) adds the edges $(x_i, y, w_i)$ where $x_i \in \mathbf{x}$ (or $(x, y_i, w_i)$ where $y_i \in \mathbf{y}$) and $w_i \in \mathbf{w}$ to $G$, where node $y$ (or $x$) originally does not exists in $G$.

3. Edge Deletion: $u = (\mathsf{Del}, x, y, *)$ deletes the edge of the form $(x, y, \cdot)$ from $G$.

4. Node Deletion: $u = (\mathsf{Del}, x, *, *)$ (or $u = (\mathsf{Del}, *, y, *)$) deletes the set $(x, *, *)$ (or $(*, y, *)$) of edges from $G$.

# General DSSE

Bipartite graphs defined by the spaces X, Y, and W.

- $(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, M, N)$: *In the setup algorithm, the user inputs the security parameter* $\lambda$, *the size* $M$ *and* $N$ *of the spaces* $\mathcal{X}$ *and* $\mathcal{Y}$ *respectively. It outputs a key* $K$, *and an (initially empty) encrypted database* EDB, *which is outsourced to the server.*
- $((K', R), (\mathsf{EDB}', R) \leftarrow \mathsf{Query}_e((K, q), \mathsf{EDB})$:
  - ✓ $(K', \tau_q) \leftarrow \mathsf{QueryToken}(K, q)$ : Generate a query token
  - ✓ $(\mathsf{EDB}', R) \leftarrow \mathsf{Query}_e(\tau_q, \mathsf{EDB})$ : Return the result R of the query. Update EDB.
- $(K', \mathsf{EDB}') \leftarrow \mathsf{Update}_e((K, u), \mathsf{EDB})$. $u \in \{\mathsf{Add}, \mathsf{Del}\} \times \mathcal{X} \times \mathcal{Y} \times \mathcal{W}$

  $(K', \tau_u) \leftarrow \mathsf{UpdateToken}(K, u)$ $\mathsf{EDB}' \leftarrow \mathsf{Update}_e(\tau_u, \mathsf{EDB})$

## Forward Privacy

Example: The client adds a new edge ( x, y, w). The server might be able to tell that this new edge is connecting to node x, using the previous query result on x.

“Forward Privacy” was previously discussed in [SPS14, RG15], however it is captured by the leakage function when query.

Captured by $\mathcal{L}_q$ could be problematic: in our example, the criterion to determine whether forward privacy has been achieved happens in updates, but not queries!

Our definition: based on update leakage function

# Forward Privacy

- Half-forward Privacy:

  Addition of edge (x, y) doesn't reveal both x and y. Hide one end.

  - *half-forward private, if for any* $u_b = (\mathsf{Add}, x_b, y_b, w)$ *where* $b = 0, 1$, *and* $x_0 = x_1$ *or* $y_0 = y_1$, *then for any PPT distinguisher* $\mathcal{D}$, *it holds that*

$$|\Pr[\mathcal{D}(\mathcal{L}_u(u_0)) = 1] - \Pr[\mathcal{D}(\mathcal{L}_u(u_1)) = 1]| \leq \mathsf{negl}(\lambda)\,.$$

- Fully-forward Privacy:

  Addition of edge (x, y) doesn't reveal either x and y. Hide both ends.

  - *fully-forward private, if for any* $u_b = (\mathsf{Add}, x_b, y_b, w)$ *where* $b = 0, 1$, *then for any PPT distinguisher* $\mathcal{D}$, *it holds that*

$$|\Pr[\mathcal{D}(\mathcal{L}_u(u_0)) = 1] - \Pr[\mathcal{D}(\mathcal{L}_u(u_1)) = 1]| \leq \mathsf{negl}(\lambda)\,.$$

# Forward Privacy

- A more comprehensive definition of Forward Privacy:

  Based on the update leakage function $\mathcal{L}_u$

$$|\Pr[\mathcal{D}(\mathcal{L}_u(u_0)) = 1] - \Pr[\mathcal{D}(\mathcal{L}_u(u_1)) = 1]| \leq \mathsf{negl}(\lambda) .$$

- Previously defined half forward privacy can be represented as $(\mathcal{U}_1 \cup \mathcal{U}_4, p_1 + p_4)$
- Previously defined fully forward privacy can be represented as $(\mathcal{U}_1 \cup \mathcal{U}_4, 1)$

| $i$ | Sets of updates $\mathcal{U}_i$ | Possible conditions such that $p_i = 1$ (If there are no restrictions, $p_i \equiv 1$) |
|---|---|---|
| 1 | $\{(\mathsf{Add}, x, y, w)\}$ | $x_0 = x_1$ or $y_0 = y_1$ |
| 2 | $\{(\mathsf{Add}, \mathbf{x}, y, \mathbf{w})\}$ | $y_0 = y_1$ |
| 3 | $\{(\mathsf{Add}, x, \mathbf{y}, \mathbf{w})\}$ | $x_0 = x_1$ |
| 4 | $\{(\mathsf{Del}, x, y, *)\}$ | $x_0 = x_1$ or $y_0 = y_1$ |
| 5 | $\{(\mathsf{Del}, x, *, *)\}$ | $\lvert(x_0, *, *)\rvert = \lvert(x_1, *, *)\rvert$ |
| 6 | $\{(\mathsf{Del}, *, y, *)\}$ | $\lvert(*, y_0, *)\rvert = \lvert(*, y_1, *)\rvert$ |

# Forward Privacy for Any DSSE

- DSSE scheme $\mathcal{E}$. (implemented before)
- Each query has a table of PRF (deterministic) keys $K_x$ and counters $c_x$ .
- Plaintext bipartite graph $\hat{G}$ and ciphertext bipartite graph $G$

  (to be achieved by an efficient data structure).

- Dictionary (or table) $\gamma^{\mathcal{Q}}$ :

  Maps a query $q = x \in \mathcal{X}$ to a PRF key $K_x$ and counter $c_x$ .
- Idea: locally maintain the table $\gamma^{\mathcal{Q}}$ .

# Setup

1. The dictionary and the plaintext graph are set to be empty.

2. The returned K contains the key and the dictionary. Both the EDB and the plaintext graph are returned.

```
$(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda)$
1 :  $(\tilde{K}, \tilde{\mathsf{EDB}}) \leftarrow \mathcal{E}.\mathsf{Setup}(1^\lambda)$
2 :  $\gamma^{\mathcal{Q}} = \phi, \hat{G} = \phi$
3 :  $\textbf{return } K = (\tilde{K}, \gamma^{\mathcal{Q}}), \mathsf{EDB} = (\tilde{\mathsf{EDB}}, \hat{G})$
```

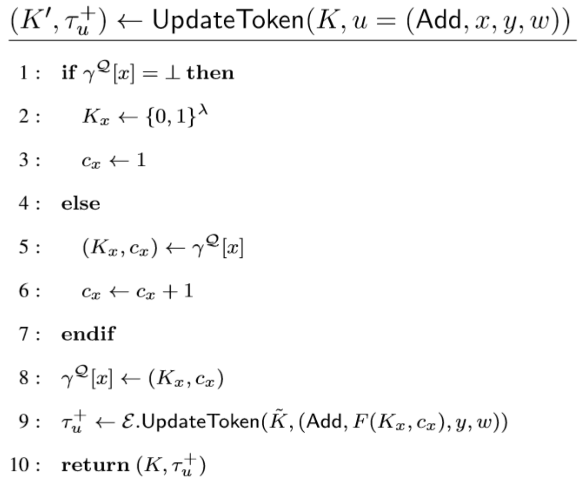
## Addition: UpdateToken

u = (Add, x, y, w ) sent by the client, means adding a new edge (x, y).

Case 1: if node x is a singleton, generate the PRF key and set counter as 1.

Case 2: if node x is not a singleton, counter + 1.

Then generate the UpdateToken for addition.



```
$(K', \tau_u^+) \leftarrow \mathsf{UpdateToken}(K, u = (\mathsf{Add}, x, y, w))$
1 :  if $\gamma^{\mathcal{Q}}[x] = \perp$ then
2 :      $K_x \leftarrow \{0,1\}^\lambda$
3 :      $c_x \leftarrow 1$
4 :  else
5 :      $(K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
6 :      $c_x \leftarrow c_x + 1$
7 :  endif
8 :  $\gamma^{\mathcal{Q}}[x] \leftarrow (K_x, c_x)$
9 :  $\tau_u^+ \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Add}, F(K_x, c_x), y, w))$
10 : return $(K, \tau_u^+)$
```

## Addition: Update

Simple, using the already generated updated token, and update the EDB.

```
1 :  $\tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Update}(\tilde{\tau}_u^+, \tilde{\mathsf{EDB}})$
2 :  return $\mathsf{EDB}$
```

# Deletion: UpdateToken

u = (Del, x, *, * ) sent by the client, means deleting all edges connecting node x.

More complex than addition, cause more edges to deal with.

Case 1: if node x is a singleton

1. Get the PRF key, know the counter. Then set dict[x] empty.
2. Generate DeleteToken for each edges
3. Encapsulate them into an UpdateToken

Case 2: if node x is not a singleton, UpdateToken = empty

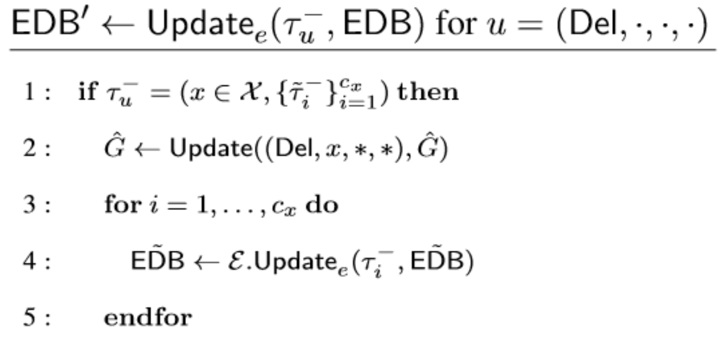
```
if $u = (\mathsf{Del}, x, *, *)$ then
    if $\gamma^{\mathcal{Q}}[x] \neq \perp$ then
        $(K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
        $\gamma^{\mathcal{Q}}[x] \leftarrow \perp$
        for $i = 1, \ldots, c_x$ do
            $\tilde{\tau}_i^- \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Del}, F(K_x, i), *, *))$
        endfor
        $\tau_u^- \leftarrow (x, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$
    else
        $\tau_u^- \leftarrow x$
    endif
```

## Deletion: Update

Here we already have a update token, which contains all deletion tokens of edges (x,*).

1. Delete all the edges in plaintext graph.

2. Delete all the edges in EDB.



$\mathsf{EDB}' \leftarrow \mathsf{Update}_e(\tau_u^-, \mathsf{EDB})$ for $u = (\mathsf{Del}, \cdot, \cdot, \cdot)$

```
1 : if $\tau_u^- = (x \in \mathcal{X}, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$ then
2 :     $\hat{G} \leftarrow \mathsf{Update}((\mathsf{Del}, x, *, *), \hat{G})$
3 :     for $i = 1, \ldots, c_x$ do
4 :         $\widetilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Update}_e(\tau_i^-, \widetilde{\mathsf{EDB}})$
5 :     endfor
```

Here we only discuss half of the situation, the other half is deleting node y, whose token generation and update procedure is very similar to the previous situation.

# Query: QueryToken

When q = x, query on a node x.

Case 1: If node x is a singleton

1. Get the PRF key and the counter in dict.
2. Generate the QueryToken and UpdateToken for each counter.
3. Encapsulate these two kinds of token into one token named QueryToken.

```
$(K', \tau_q) \leftarrow \mathsf{QueryToken}(K, q)$
if $q = x \in \mathcal{X} \wedge \gamma^{\mathcal{Q}}[x] \neq \perp$ then
    $(K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
    $\gamma^{\mathcal{Q}}[x] \leftarrow \perp$
    for $i = 1, \ldots, c_x$ do
        $\tilde{\tau}_i \leftarrow \mathcal{E}.\mathsf{QueryToken}(\tilde{K}, F(K_x, i))$
        $\tilde{\tau}_i^- \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Del}, F(K_x, i), *, *))$
    endfor
    $\tau_q \leftarrow (x, \{\tilde{\tau}_i, \tilde{\tau}_i^-\}_{i=1}^{c_x})$
```

Case 2: x is not a singleton.

QueryToken = x.

$$\tau_q \leftarrow x$$

Case 3: x doesn’t exists in previous graph. QueryToken = empty.

$$\tau_q \leftarrow \perp$$

# Query: Do Query

Why do we need the plaintext graph?
Once the query on x is done, we think that (x, *) is not secure anymore. For efficiency, we throw these edges to a plaintext graph.

1. Try to get the result directly from plaintext graph
2. Get the result form EDB, delete these edges from EDB
3. Merge the result from step 1 and 2
4. Adding the query result to the plaintext graph

```
Parse $\tau_q$ as $(x, \{\tilde{\tau}_i, \tilde{\tau}_i^-\}_{i=1}^{c_x})$
$R \leftarrow \mathsf{Query}(x, \hat{G})$
for $i = 1, \ldots, c_x$ do
    $\tilde{R} \leftarrow \mathcal{E}.\mathsf{Query}_e(\tilde{\tau}_i, \mathsf{E\tilde{D}B})$
    $\mathsf{E\tilde{D}B} \leftarrow \mathcal{E}.\mathsf{Update}_e(\tilde{\tau}_i^-, \mathsf{E\tilde{D}B})$
    $R \leftarrow R \cup \tilde{R}$
endfor
foreach $(\tilde{x}, y, w) \in R$ do
    $R \leftarrow R \setminus \{(\tilde{x}, y, w)\} \cup \{(x, y, w)\}$
    $\hat{G} \leftarrow \mathsf{Update}((\mathsf{Add}, x, y, w), \hat{G})$
endfor
return $(\mathsf{EDB}, R)$
```

# Security Proof

- Take the proof of previously constructed $(\mathcal{U}_1, p'_1)$ for illustration:

$$p'_1(u_0, u_1) = 1 \textit{ if and only if } y_0 = y_1 \textit{ and } w_0 = w_1$$

Simulator $\mathcal{S}_{\mathcal{E}}$ : which simulates the queries and updates of $\mathcal{E}$ , which is a $(\tilde{\mathcal{L}}_q, \tilde{\mathcal{L}}_u)$-CQA2 secured scheme.

Simulator $\mathcal{S}$ : which receive the leakage for random x when update.

for instance: $\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\tilde{x}, \tilde{\mathcal{L}}_u(\mathsf{Add}, \tilde{x}, y, w))$

Then pass $\tilde{\mathcal{L}}_u(\mathsf{Add}, \tilde{x}, y, w)$ to $\mathcal{S}_{\mathcal{E}}$ , which outputs a simulated token $\tau_u^+$ .

If there exists an environment which distinguish the simulation and the real scheme, then S can be constructed by this environment.

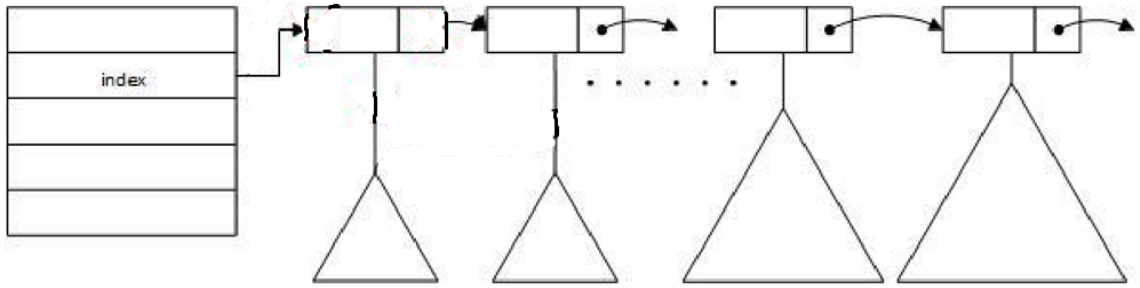
## Cascaded Triangles

Motivations: The updates should affect the previous DB as less as possible.

Previously we used linked list, trees and others.

Idea: we separate the big binary tree into many small perfect binary trees, whose size is kept in cascaded order. We call these perfect binary trees "Cascaded Triangles".

So the update will just affect a few triangles!



# Cascaded Triangles

Consists of three dictionaries $\gamma^{\mathcal{Q}}$, $\gamma^{\mathcal{D}}$ , and $\gamma^{\delta}$

Why do we need three dictionaries instead of one? Think when traverse.

Firstly, we need to store the root address of every triangles, that is the first dictionary.

Secondly, given the root, we should know what are the left and right child (if any) of the root. So the second dictionary to store the detailed information of every node is needed.

Lastly, clients are not always traverse by X, so another dictionary to store the dual root address is needed.

# Cascaded Triangles

Consists of three dictionaries $\gamma^{\mathcal{Q}}$, $\gamma^{\mathcal{D}}$, and $\gamma^{\delta}$

$\gamma^{\mathcal{Q}}$ stores the root address and heights of the triangles at the client.

$$\gamma^{\mathcal{Q}}[x] = (\mathsf{addr}_1, \ldots, \mathsf{addr}_k, h)$$

$\gamma^{\mathcal{D}}$ stores the address of the dual tuples, for dual retrieving.

$\gamma^{\delta}$ maps the address to a tuple (a, b).

$$a = \overline{a} = (x, y, w) \qquad b = (\mathsf{chd}_0, \mathsf{chd}_1)$$

$\gamma^{\mathcal{D}}$ and $\gamma^{\delta}$ are stored at the servers' side.

Requirement of Heights: $h_1 \leq h_2 < h_3 < \ldots < h_k$, $h \in \{0,1,2\}^{\lceil \lg n_q \rceil}$

# Adding a Node

First add it as a root node of a new tree.

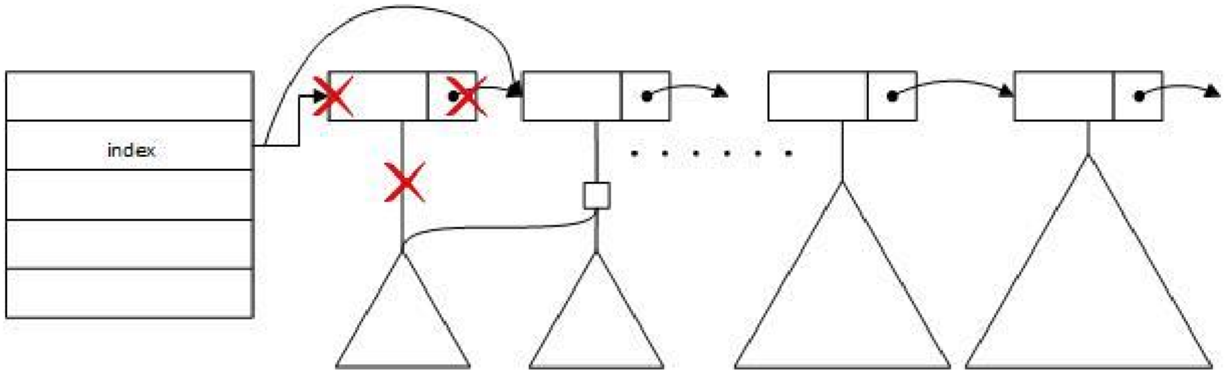
If two shortest trees have the same height, combine them.

$\mathsf{Carry}(h+1) = 1$ , add $(a, b)$ where $a = (x, y, w)$ and $b = (\mathsf{addr}_1, \mathsf{addr}_2)$ to a random address addr

$$\gamma^{\mathcal{Q}}[x] \leftarrow (\mathsf{addr}, \mathsf{addr}_3, \ldots, \mathsf{addr}_k, h+1),$$

$\mathsf{Carry}(h+1) = 0.$ $a = (x, y, w)$ and $b = (\perp, \perp)$

$$\gamma^{\mathcal{Q}}[x] \leftarrow (\mathsf{addr}, \mathsf{addr}_1, \ldots, \mathsf{addr}_k, h+1).$$

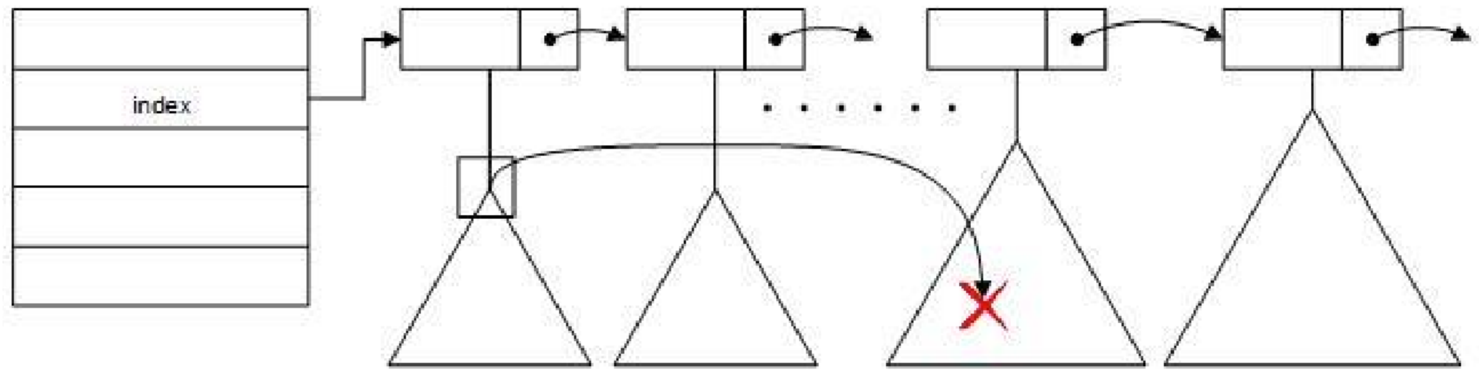


# Deleting a Node

Firstly, replace that node with the root of the shortest tree.

If it is a singleton, simply remove the  corresponding linked list node.

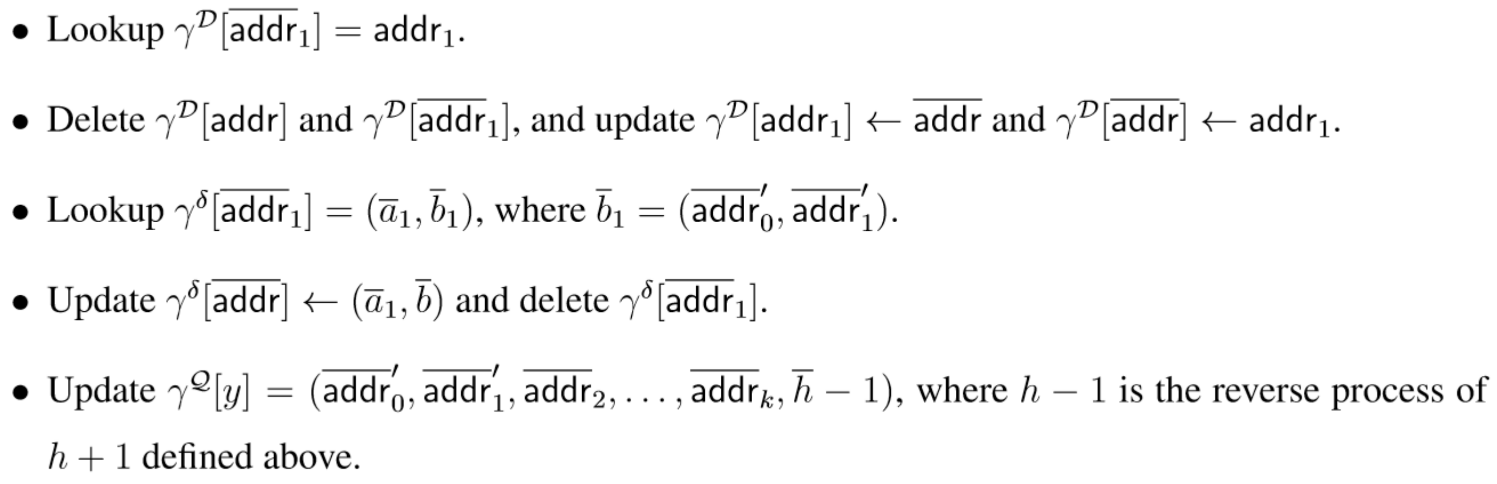
Otherwise, update the linked list so it points to one of its child.



# Deleting a Node

If we want to delete node x:
Firstly, we traverse the set (x, *, *) using the traverse algorithm.
Delete all the traverse nodes, based on the address of the root node.

Then, use traverse result of x to find the duals in triangles based on y.
Replace these nodes (in the middle of some triangles) with the smallest triangles.



- Lookup $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}_1}] = \mathsf{addr}_1$.
- Delete $\gamma^{\mathcal{D}}[\mathsf{addr}]$ and $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}_1}]$, and update $\gamma^{\mathcal{D}}[\mathsf{addr}_1] \leftarrow \overline{\mathsf{addr}}$ and $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}}] \leftarrow \mathsf{addr}_1$.
- Lookup $\gamma^{\delta}[\overline{\mathsf{addr}_1}] = (\overline{a}_1, \overline{b}_1)$, where $\overline{b}_1 = (\overline{\mathsf{addr}}_0', \overline{\mathsf{addr}}_1')$.
- Update $\gamma^{\delta}[\overline{\mathsf{addr}}] \leftarrow (\overline{a}_1, \overline{b})$ and delete $\gamma^{\delta}[\overline{\mathsf{addr}_1}]$.
- Update $\gamma^{\mathcal{Q}}[y] = (\overline{\mathsf{addr}}_0', \overline{\mathsf{addr}}_1', \overline{\mathsf{addr}_2}, \ldots, \overline{\mathsf{addr}_k}, \overline{h} - 1)$, where $h - 1$ is the reverse process of $h + 1$ defined above.

# Encrypted Version

Main idea:

Split G into two graphs, one is the ciphertext graph and plaintext graph.

1. Now we only stored the ciphertext in tuples of $\gamma^{\mathcal{Q}}[x]$ .

2. Using two secret keys to access a = (x, y, w) and b = (ch0, ch1) . The keys are independently generated for each x.

3. The $\gamma^{\mathcal{Q}}[x]$ additionally stores the two secret keys associated to x.

# Encrypted Version: Queries

Differences form non-encrypted version:

1. When retrieving of the root address, the two secret keys are also retrieved.

2. The clients sends the result of the retrieval back to the server. The server used the first key to get part of the query result. And use the second key to get the subtrees.

3. The server also returns the previous result in plaintext graph.