CellTree: Implementation Documentation

Anasuya Acharya
Bar-Ilan University

Akash Trehan
Microsoft

Manoj Prabhakaran
IIT Bombay

March 15, 2021

1 Implementation Details : $\mathbf{CT_0}$

As a proof-of-concept, we have a Python implementation of instances running a CellTree as per the model described in $\mathbf{CT_0}$. This version, much like a blockchain, allows storage of immutable, static data in cells, though this can be extended to allow for more diverse functionality. This section discusses the details of the implementation.

Design Outline

The CellTree architecture has two types of entities: a *Host* and a *Client*. The Client refers to a stateless external entity that uses the CellTree as a service. A Host, on the other hand, is part of the tree serving as a *worker* in crews for one or many nodes. It has addresses and keys associated with it, stores cell information and maintains all the related data structures that facilitate assimilation.

The **Client** has an interface through which it can query nodes in the tree for their data. It uses the READ procedure to do so. This in turn accesses the `discover` module to find the crew of the node from the CellTree, the `fetch` module to read the cell and poa, the `crewSign` and `exec` modules to verify the validity of the cell data. `discover` also uses `fetch` internally which in turn uses a `communication` module. Further functionality may be implemented when the need for such arises for applications hosted on the tree.

The **Host** as a part of the CellTree has associated with it a public-private key pair, a hash address and is bound to a network address. For each node whose crew it is part of, it runs *worker* threads in parallel. These interact with other hosts with workers of the same crew and maintain the cell. For each worker the host needs to maintain a list of its crew members, the cell, a merkle multi-tree, a poa store, the previous assimilation data. The latter three, along with the UUlist are maintained by the `store` module. The `rootward` and `leafward` modules use these, schedule and call procedures ROOTWARDPROPAGATION and LEAFWARDPROPAGATION as specified in $\mathbf{CT_0}$. A worker may initiate the `selectCell` module to run a protocol with other crew members to get from an empty – cell to a static – cell. This internally calls the EVOLVE function which in turn uses the `exec` module. If a child node of a crew is yet to be initialized, a worker may initiate the `selectCrew` protocol. This involves all the supervising crews of the that node. The host also interfaces with the clients on discovery or read requests through the `fetch` module. All these modules internally use the `communication`, `hash`, and `crewSign` modules to run protocols and verify data.

Cell

The current implementation supports the *empty-cell*, the *static-cell* and *ledger-cell*. The code for these cell types is organized as follows:

- CellInterfaces.py - This file contains classes `CellData` and `NuclearData` that are interfaces outlining the functions that the cell-data and nuclear-data of any cell type must have. Other cell types inherit from these and populate these functions with the required functionality. This file also has classes `EmptyCData` and `EmptyNData` that implement these interfaces for the cell-data and nuclear-data respectively for the *empty-cell*.
- Cell.py - This file contains classes `Cell` and `Nucleus`. The `Cell` class represents a cell. It has instance variables *cData* and *nuc* that hold the cell-data and nucleus respectively. When the cell is being populated, the type of *cData* is resolved to `EmptyCData`, `StaticCData` or `LedgerCData` depending on what the cell type is. *nuc* holds an instance of type `Nucleus`. The class `Nucleus` contains instance variables *nData*, *strChkCell*, *strChkNext* and *strChkPrev*. The latter three are strings that hold the nuclear codes chkCell, chkNext and chkPrev respectively. The former holds nuclear-data whose type is resolved similarly to that of cell-data.
- StaticCell.py - This file contains classes `StaticCData` and `StaticNData` that implement the interfaces `CellData` and `NuclearData` respectively for the *static-cell*. There is also the class `StaticCell` with the function MAKESTATICCELL that takes in the path of a file storing the static cell-data and creates and

returns a static cell with the appropriate nuclear-data and nuclear code strings. This is used in `selectCell`.

- LedgerCell.py - This contains `LedgerCData` and `LedgerNData` that implement the respective interfaces for *ledger-cell*. The `LedgerCell` class contains MAKELEDGERCELL that takes a previous (either empty or ledger) cell and a path to new block data and creates and returns a ledger cell with the appropriate cell-data, nuclear-data and nuclear code strings.

The `Cell` and `Nucleus` also have methods to convert the instances to a string and back for use in the `store` and `communication` module.

Organization

The data for an execution of the CellTree $\mathbf{CT_0}$ is organized as follows:

A separate directory 'addr' contains files 'addr.txt' and 'rootCrew.txt'. The former contains the hash-address, public-key, IP address and port information for each *host*, as described earlier. The latter contains the hash-addresses of only those hosts that form the root crew of $\mathbf{CT_0}$.

A *client* is a stateless entity that accesses the CellTree. It uses information in the 'addr' directory and also locally stores a configuration file 'config.txt' containing the value of ℓ. This is required in `discover` for $\mathbf{CT_0}$. The client also has a 'crews' directory with files named in the form: '<nodeID>.txt'. Each such file contains a list of hash-addresses of the hosts that act as workers for the crew of <nodeID>. These are populated using `discover` and used in READ. Lastly, its 'var' directory contains a 'log.txt' file.

A *host* has the following directory structure:

- 'config.txt' - This contains configuration information for the host with identifiers, network addresses and the path to the 'addr' directory.
- 'wrkrconfig.txt' - This is the default configuration information for used by the host when it is acting as a *worker* in a node. This is used if a custom configuration is not specified.
- 'key' - This directory contains key information of the host:
  - 'host.pem' - contains the password protected private-key
  - 'pubkey.txt' - contains the corresponding public-key
- 'crews' - This stores member information of crews that the host is not part of in a manner similar to what is described for client.
- 'var' - This contains a 'log.txt' file. It also contains other files such as those with inputs to `selectCell` or `selectCrew` with new cell data or new crew member sugestions respectively.
- 'wrk' - This directory contains 'wrkrList.txt' that stores a list of all nodes in the CellTree whose crew this host is a *worker* in. It also contains sub-directories named '<nodeID>' that contains files with state information for that <nodeID>. Each such sub-directory contains the following files:
  - members.txt - list of hash-addresses of all hosts that are a worker of that crew.
  - cell.txt - the current cell that the node contains.
  - mmTree.txt - the merkle multi-tree for that node.
  - poa.txt - the dictionary of assimilated cells and their Proof-of-Assimilation from along with the assimilation-root. This is updated in `leafward` and accessed during READ.
  - assimData.txt - the list of Assimilation-Signals to be used as input to `rootward`.

The code for the instances (hosts and clients) consists of:

- 'main.py' - This is the program that starts off the execution. This takes in a command line argument specifying weather a Client or a Host needs to be spun off. If it is a Host, an additional unique identifier is required specifying the directory where all its related files are stored.
- 'CTClnt.py' - This has the class `Clnt` that contains code for a *client*, the functionality of which is as described before. It accesses its files through an instance of `ClntConfig` in 'Config.py'. It provides a terminal interface through which it takes read commands using 'Shell.py'.
- 'CTHost.py' - This has the class `Host` that contains code for a *host*. It accesses 'wrkrList.txt' through an instance of `Config` (from 'Config.py) and for each *worker*, it initiates a thread running an instance of class `Wrkr`. `Wrkr`, in the same file, takes its information through `WrkrConfig` (from 'Config.py') and performs all the functions of a worker of the node, taking user commands through the terminal using 'Shell.py'.
- 'Shell.py' - This contains class `Shell` that allows both host and client to have an interactive terminal interface with users that may enter commands.
- 'Config.py' - This contains `Config`, `WrkrConfig`, `ClntConfig` which initializes information that connects the hosts, worker threads, and clients respectively with the directory files. It also contains class `CrewInfo` that is used by these classes to access files with crew member information.
- 'Modules.py' - This contains the class `Modules` and `ClntModules` that contains an instance for all the

Static Data: The nuclear data ndata is simply a hash of the cellular data cdata using a collision-resistant hash function.

$$\mathsf{chkCell}(\mathsf{cell}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{cell.nuc.ndata} = \text{hash}(\mathsf{cell.cdata}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkPrev}(\mathsf{nuc}, \mathsf{nuc}^{(-)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(-)} = \mathsf{nuc}_0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkNext}(\mathsf{nuc}, \mathsf{nuc}^{(+)}) = \mathsf{false}$$

Blockchain Ledger: cdata is an array of blocks (|cdata| refers to the number of blocks in the array and cdata[i] refers to the i^{th} block). ndata is a triple (n, blkHash, chnHash) where n is the number of blocks in the ledger and the other two items are hash values (the last block and the root in a Merkle-*chain*).

$$\mathsf{chkCell}(\mathsf{cell}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{cell.nuc.ndata} = (n, \gamma_n, \text{hash}(\text{hash}((\ldots\text{hash}(\\ & \text{hash}(\gamma_1, \gamma_2), \gamma_3), \ldots,), \gamma_{n-1}), \gamma_n)))) \\ & \text{where } n = |\mathsf{cell.cdata}| \text{ and } \gamma_i = \text{hash}(\mathsf{cell.cdata}[i]) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkNext}(\mathsf{nuc}, \mathsf{nuc}^{(+)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(+)}.\mathsf{ndata.chnHash} = \text{hash}(\mathsf{nuc.ndata.chnHash}, \\ & \mathsf{nuc}^{(+)}.\mathsf{ndata.blkHash})), \\ & \quad \mathsf{nuc}^{(+)}.\mathsf{ncode} = \mathsf{nuc.ncode} \text{ and} \\ & \quad \mathsf{nuc}^{(+)}.\mathsf{ndata.n} = \mathsf{nuc.ndata.n} + 1 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkPrev}(\mathsf{nuc}, \mathsf{nuc}^{(-)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(-)} = \mathsf{nuc}_0, \mathsf{nuc.ndata.n} = 1 \\ & \text{and } \mathsf{nuc.ndata.chnHash} = \\ & \quad \mathsf{nuc.ndata.blkHash} \\ \mathsf{nuc.chkNext}(\mathsf{nuc}^{(-)}, \mathsf{nuc}) & \text{otherwise} \end{cases}$$

Figure 1 Examples of nucleus families

modules for each worker and the client respectively. They also initiate the listener and scheduler threads for protocols for the modules that require it.

- 'CT0<module-name>.py - Contains the code for the classes for worker and client (if required) as listed in Appendix **??** for the module <module-name>. These are populated as required for performing the functionality of $\mathbf{CT_0}$.
- 'Procedures.py' - This has `Procedures` with the RootwardPropagation, LeafwardPropagation and Evolve functions with references to the different modules; and the `ClntProcedures` class with the Read procedure. All of these are static methods and their algorithms conform strictly to that described in Section **??**.

Modules

In the implementation, the various modules are coded to perform the following functions:

`hash` **module.** This is the class `HashDigest` in the file 'HashDigest.py'. It has the functions:

- generate(message): returns the SHA256 hash of the message passed
- generateNonce(): returns a nonce value
- getCumulativeHash(node): returns the cumulative hash of the node of the merkle multi-tree that is passed
- mEval(mTree): returns the cumulative hash of the root of the merkle tree mTree

`crewSign` **module.** This exists in the file 'CrewSign.py'. It has the classes `CrewSign` and `ClntSign` for the worker and client respectively:

- `CrewSign`.wrkrSign(message): returns a signature of that host on the message along with the public key
- `CrewSign`.wrkrVerify(sign, message): checks if the signature (sign) verifies to the message using the public key encoded into sign
- `CrewSign`.sign(message, mode): in the 'init' mode, returns a crew-sign dictionary with elements for all members of crew. In the 'resp' mode, returns wrkrSign(message). Uses `communication` to run an in-crew protocol
- `CrewSign`.listener(): thread that responds to messages in the crew-sign protocol with sign(message, 'resp')
- `CrewSign`.verify(sender, message, sign): verifies the crew signature on message according to the public key dictionary and a threshold value
- `CrewSign`.toString(signature): turns a crew signature dictionary into string for communication
- `CrewSign`.parse(stringSign): parses a string into a crew signature dictionary
- `CrewSign`.sendMessage(connections, receiver, tag, message): uses `communication` to send message and

tag to all receivers ass indicated in receiver using connections

- `CrewSign`.getMessage(tag): returns `crewSign` messages with the indicated tag

- `ClntSign`.wrkrVerify(sign, message): checks if the signature (sign) verifies to the message using the public key encoded into sign
- `ClntSign`.verify(sender, message, sign): verifies the crew signature on message according to the public key dictionary and a threshold value
- `ClntSign`.toString(signature): turns a crew signature dictionary into string for communication
- `ClntSign`.parse(stringSign): parses a string into a crew signature dictionary

`communication` **module.** This exists in the file 'Communication.py'. It has the classes `Communication` and `ClntCommunication` for the worker and client respectively:

- Class `msgPacket`: An instance of this type stores all data to be communicated (sender and receiver address, channel, tag, message, verification data)
- Class `Communication`:
  - readConfig(con) - uses `Config` to read the host addresses and keys
  - initCommQueues() - initializes the communication queues structure
  - addCommQueue(wrkr) - adds a communication queue for each channel (one for each module using the `communication` module) for the host acting as a worker in the nodeID indicated in wrkr
  - connectTo(wrkrcon, nodeID) - initializes socket connections with all crew members of the node in nodeID
  - disconnect(cnxns) - disconnects the socket connections for the set of connections in cnxns
  - listen() - listens for incoming socket connections
  - thrd(target, args) - starts a thread with function target and arguements args
  - addToMessageQueue(conn, addr) - a separate thread for each connection that listens and adds incoming messages to the communication queue
  - sendMessage(cnxns, wrkrcon, receiver, sender, channel, tag, message) - creates a message packet with the arguements passed and calls communicate to send it to all connections in cnxns
  - getMessage(wrkrcon, channel, tag, source) - returns a message from the indicated channel with the given tag and from the sender as given in source
  - getInCrewResponse(wrkrcon, channel, tag) - returns all messages in the channel with the given tag coming from fellow crew members of nodeID indicated in wrkrcon
  - getNWInfo(address, wrkrcon) - given a hash-address, return the network address of host
  - communicate(textMsg, cnxns) - sends the message packet to all connections in cnxns
- Class `clntMsgPacket`: Creates a message packet with similar structure to `msgPacket` but with parameters in accordence with what a client will send and recieve.
- Class `ClntCommunication`:
  - connectTo(nodeID) - creates and returns all socket connections to hosts in the crew of nodeID
  - getNWInfo(address) - given a hash-address, returns the network address of host
  - sendMessage(cnxns, channel, tag, messageStruct, destination) - creates a client message packet and sends it to all connections in the list cnxns
  - getMessage(cnxns, channel, tag, source) - returns the first message with tag received from cnxns
  - disconnect(cnxns) - disconnects the socket connections listed in cnxns

`fetch` **module.** This exists in the file 'CT0Fetch.py'. It has the classes `Fetch` and `ClntFetch` for the worker and client respectively:

- `Fetch`.sendMessage(cnxns, receiver, tag, message) - uses `communication` to send message to cnxns
- `Fetch`.getMessage(tag) - uses `communication` to receive message from the fetch channel with given tag
- `Fetch`.getCrew(nodeID, queryCrewID) - initiates the protocol for asking nodeID for the members in queryCrewID and returns the set
- `Fetch`.getCrewResponse(nodeID) - returns the set of members in nodeID if known
- `Fetch`.getCellResponse(root) - returns the latest version of `cell` and `poa` with given assimilation root
- `Fetch`.listener() - listens and responds to client getCell and getCrew requests

- `ClntFetch`.sendMessage(cnxns, tag, message, destination) - uses `communication` to send message to all connections in cnxns
- `ClntFetch`.getMessage(cnxns, tag, source) - uses `communication` to return message from the fetch channel with given tag
- `ClntFetch`.getCell(nodeID, POAroot) - initiates the getCell protocol to ask nodeID for an assimilated version of their `cell` and `poa` with the assimilation root as POAroot
- `ClntFetch`.getCrew(nodeID, queryCrewID) - initiates the getCrew protocol to ask nodeID for the members in crew queryCrewID

`discover` **module.** This exists in the file 'CT0Discover.py'. It has the classes `Discover` and `ClntDiscover` for the worker and client respectively:

- `Discover`.discover(nodeID) - uses `fetch`.getCrew() iteratively to find and store in 'crews' all members of the crew of nodeID
- `Discover`.discoverFetch(nodeID, rootID) - returns members of nodeID if present in 'crews' or runs discover() and then returns the members of nodeID

- `ClntDiscover`.discover(nodeID) - uses `fetch`.getCrew() iteratively to find and store in 'crews' all members of the crew of nodeID
- `ClntDiscover`.discoverFetch(nodeID, rootID) - returns members of nodeID if present in 'crews' or runs discover() and then returns the members of nodeID

`rootward` **module.** This is the class `Rootward` in the file 'CT0Rootward.py'. It has the functions:

- sendMessage(cnxns, receiver, tag, message) - uses `communication` to send message to connections in cnxns
- getMessage(tag) - uses `communication` to receive message from channel rootward with given tag
- listener() - collects all received assimilation signals and puts them into assimData structure
- scheduler() - invokes ROOTWARDPROPAGATION at time epoch as in $\mathbf{CT_0}$
- receive(nodeID) - returns from assimData structure the assimilation signal from nodeID
- monitored() - returns the list of nodeIDs in the monitored subtree
- send(uuList, mNode) - sends assimilation signal to parent using sendMessage

`leafward` **module.** This is the class `Leafward` in the file 'CT0Leafward.py'. It has the functions:

- sendMessage(cnxns, receiver, tag, message) - uses `communication` to send message to connections in cnxns
- getMessage(tag, sender) - uses `communication` to receive message from channel leafward with given tag from sender
- listener() - collects the received proof-of-assimilation and invokes LEAFWARDPROPAGATION
- scheduler() - does nothing
- receive() - returns the poa just received
- pickLeaves(rootID) - returns nodeIDs to which the extended poa needs to be sent
- send(POAset) - sends to all appropriate descendant nodes their respective poa

`selectCell` **module.** This is the class `SelectCell` in the file 'CT0SelectCell.py'. It has the functions:

- sendMessage(cnxns, receiver, tag, message) - uses `communication` to send message to connections in cnxns
- getMessage(tag) - uses `communication` and returns message with given tag from the selectCell channel
- getData(path) - reads cell data from path and creates a new cell
- newCell(cell, mode) - in 'init' mode, initiates the selectCell protocol with the cell created from getData(). In the 'resp' mode, returns the result of EVOLVE on the cell received
- listener() - listens and responds to hosts that initiate the selectCell protocol

`selectCrew` **module.** This is the class `CreateNode` in the file 'CT0CreateNode.py'. It has the functions:

- sendMessage(cnxns, receiver, tag, message) - uses `communication` to send message to connections in cnxns
- getMessage(tag) - uses `communication` and returns message with given tag from the createNode channel
- newNode(child, path, mode) - runs the protocol for creating a new node: achieves consensus on a set of members for the new crew
- selectCrew(stringCrewInfo) - a sub-protocol that asks new potential workers if they want to be part of the new crew and returns the result
- memberResponse(nodeInfoString) - creates a worker thread as a member of the new crew and initializes parameters
- joinCrewListener() - listens and responds to requests to become a member of the crew of a new node
- crewListener() - listens and responds to workers of the same crew that participate in the createNode protocol

`exec` **module.** This exists in the file 'CT0Execute.py'. It has the class `Execute` that has the function `exe(tag, strCode, args)`. It takes in a tag (chkCell, chkPrev, chkNext), a code string (strCode) and arguements (args) to pass to the function in the code string, and returns a true or false value.

`moduleManager` **module.** This exists in the file 'CT0ModuleManager.py'. It has the class `ModuleManager` that remains unimplemented.

`store` **module.** All classes in this module exist in 'CT0Store.py'. It has the following structure:

- Class `Store`: Handles all data structures and internal storage related to the information that a crew needs to store. It has instances of the following:

  - *currentCell* - `Cell` type containing the latest value of the cell in the node
  - *uuList* - `UUList` type storing all unassimilated updates
  - *mmTree* - `MerkleMultiTree` type storing the merkle multi-tree
  - *poaStore* - `POAStore` type with a dictionary of assimilated cells with their proof-of-assimilation (poa)
  - updateCell(cell) - changes the value of *currentCell* to the cell passed
  - listener() - thread that triggers garbage collection on command

- Class `UUList`: Unassimilated Updates List
  - *uuList* - list storing all unassimilated updates
  - add(nuc) - adds newly evolved cell's nucleus to *uuList*
  - flush() - returns all elements of *uuList* and empties it
  - toString() - returns string version of UUList for communication
  - parse(stringUUlist) - converts a string stringUUlist to an instance of `UUList`

- Class `MNode`: Merkle Node
  - *nodeID* - of the CellTree node whose nucleus is being stored
  - *nuc* - nucleus of the cell in the subtree that has been assimilated
  - *nonce* - used to hash the nucleus contents
  - *lChildCuHash* - cumulative hash of the left child merkle node
  - *rChildCuHash* - cumulative hash of the right child merkle node
  - updateL(lPtr) - populates the *lChildCuHash* from blank to the value passed
  - updateR(rPtr) - populates the *rChildCuHash* from blank to the value passed
  - toString() - returns a string representation of the merkle node
  - parse(strNode) - converts a string representation of the merkle node to an instance of `MNode`

- Class `MerkleMultiTree`: Merkle Multi-Tree of Merkle Nodes
  - *nodeDict* - dictionary of instances of `MNode` indexed by their cumulative hash
  - *lastRoot* - cumulative hash node with of the latest assimilated version of that node's nucleus
  - getCumulativeHash(node) - returns the cumulative hash of the node
  - parse(stringTData) - populates the Merkle multi-tree structure from a string representation of the same
  - writeToFile(path) - writes the string representation of the Merkle multi-tree to a file specified by path
  - readFromFile(path) - reads and parses the Merkle multi-tree from a file specified in path
  - toString() - returns a string representation of all the contents in `MerkleMultiTree`
  - updateMMTree(listOfNodes) - adds the nodes in listOfNodes to the *nodeDict* and accordingly updates the *lastRoot*
  - cleanMMTree(rootNodeList) - deletes the list of outdated nodes given in rootNodeList and calls the garbage collection function
  - collectGarbage() - a garbage collection function that deletes all nodes in the multi-tree without any parent, unless its nodeID is that of the current node
  - getTree(leaves, nodeID, POA) - this works in 2 modes: if the POA is specified, it returns the merkle tree node list that is rooted at the leaf node of the POA's merkle path. If no POA is passed, it returns the merkle tree rooted at *lastRoot*
  - concat(mTree1, mTree2) - if mTree2's root is part of mTree1's leaves, returns a merkle tree with root as mTree1's root and including all nodes in mTree2

- Class `AssimData`: Store of all Assimilation Signals
  - *assimSignalDict* - dictionary of assimilation signals indexed by monitored nodeID
  - readFromFile(path) - reads and parses an *assimSignalDict* from file specified in path
  - writeToFile(path) - writes to the file specified in path a string version of the *assimSignalDict*
  - toString() - returns a string version of the *assimSignalDict*
  - parse(text) - populates the *assimSignalDict* from a string version of the same
  - update(newData) - replaces old assimilation signal entries with ones specified in newData
  - append(text) - converts the text passed into an assimilation signal and adds it to *assimSignalDict*

- Class `POA`: Proof of Assimilation `poa`
  - *merklePath* - dictionary of instances of `MNode` that form a merkle path from the assimilation root to the node, indexed by cumulative hash
  - *sign* - `crewSign` on the node contents of the assimilation root
  - checkPOA() - returns `true` if the *sign* verifies to the assimilation root of *merklePath*

  - toString() - returns a string representation of the `poa`
  - parse(stringPOA) - populates an instance of `POA` from the string representation of the same

- Class `POAStore`: Store of all assimilated `cell` and `poa` that are not outdated
  - *cellHistory* - dictionary of `cell` and `poa` with assimilation root, indexed by `cell`
  - readFromFile(path) - reads from file given in path and populates *cellHistory*
  - writeToFile(path) - writes to the file given in path a string representation of the *cellHistory*
  - toString() - returns a string representation of the *cellHistory*
  - parse(strCellHistory) - populates the *cellHistory* from a string version of the same
  - add(cell, root, POA) - adds an entry to the *cellHistory* with the given parameters
  - delete(cell) - deletes all `poa` for the given cell, along with the cell itself, from *cellHistory*
  - exists(cell, root, POA) - returns `true` if the tuple passed exists in *cellHistory*
  - search(cell, root) - returns a `poa` for the given cell with the specified assimilation root