Q1) In a distributed system, processing elements in a linked list structure typically requires one core to handle the initial elements sequentially, leaving the remainder to another core, which leads to unbalanced workloads. By using JoinLists, however—due to their tree-like structure—you can divide elements (and thus computation) more evenly across cores. In a MapReduce framework, this advantage is particularly valuable: the list of files can be split so each core processes a roughly equal portion of data. This aligns with MapReduce's core principle—allowing independent mapping operations—thereby optimizing workload balance and boosting overall performance.

Q2) Say we have a list of accounts for some social media site, with each account represented as a tag-value pair of the user's handle (String), along with a list of the handles for all accounts followed by the user (List<String>). Then, say we want to determine which of the users on the platform follow a specific account. We would pass each account to a mapper function which would perform a search algorithm on the list of accounts followed, to determine if the specific account is present. The mapper function would then return a tag-value pair of a boolean representing if the search was successful or not, along with the handle of the account. The shuffle function would then, as usual, group the pairs based on tag. The reducer function would then return the pair associated with the 'true' tag unchanged, ignoring the 'false' tag-value pair.

Formal function signatures:

```
input :: List<Tv-pair<String, List<String>>>
mapper ::(Tv-pair<String, List<String>>
      -> List<Tv-pair<Boolean, String>>)
reducer :: (Tv-pair<Boolean, List<String>>
      -> Tv-pair<Boolean, List<String>>))
```