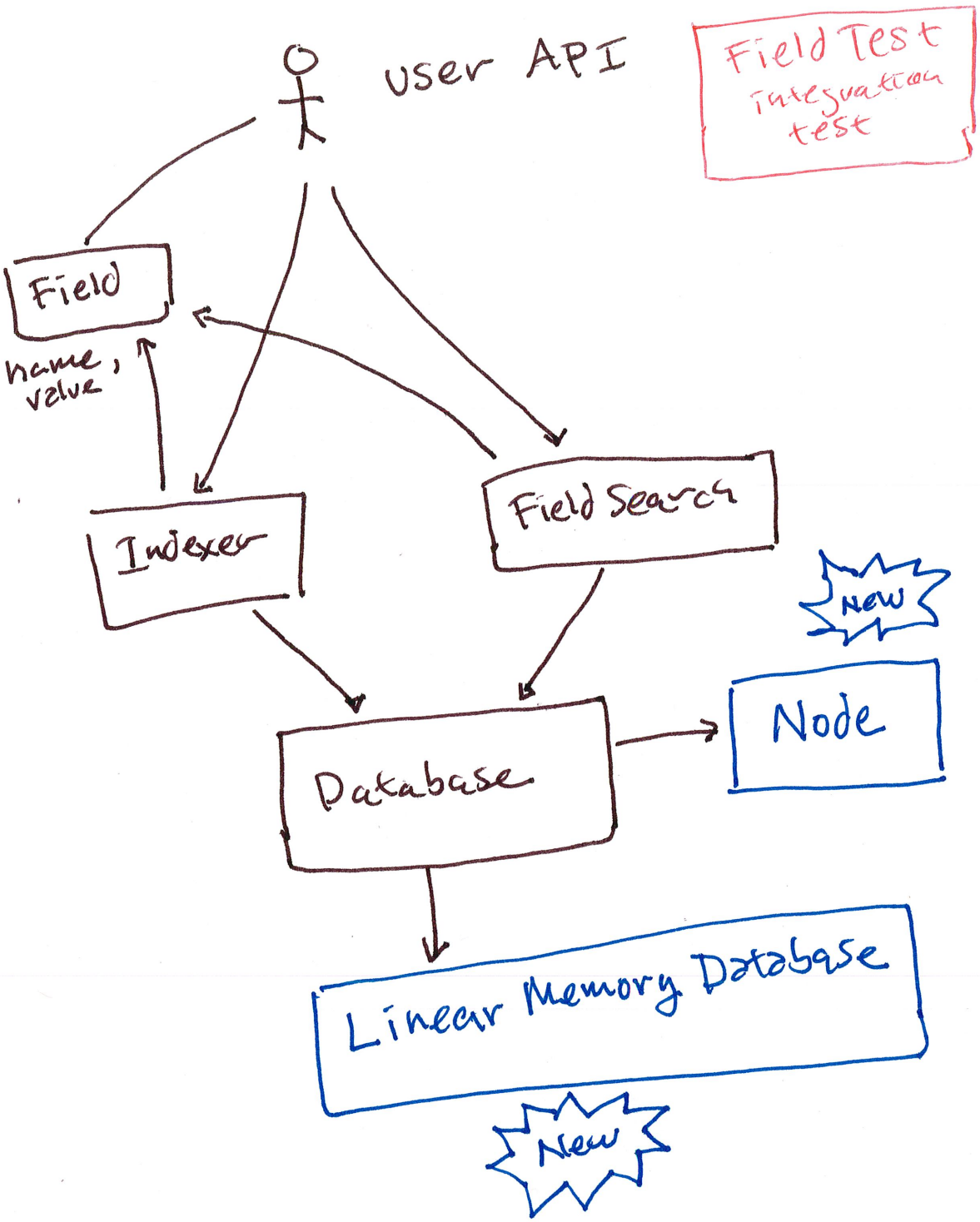
Iteration #1 of Search Engine (review)



Field $\geq$ Field $\in$ Field ~ regex

Field $\neq$ Field max

Store on Disk

Ideas for improvement

(1) Storage of values

(2) Storage of keys

For (1), we need variable-size areas - like (length, byte[]), plus disk address

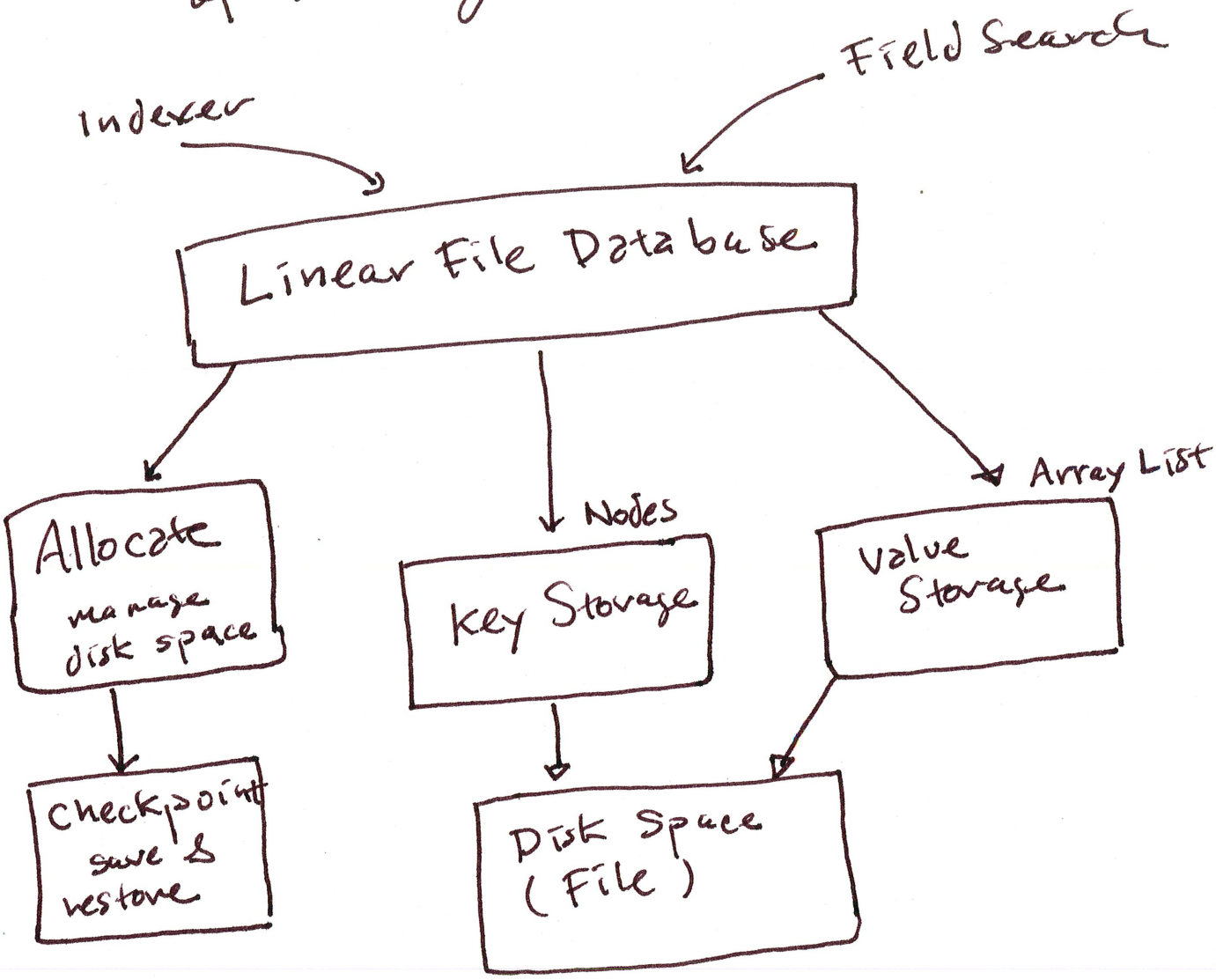
For (2), maybe we need more than one index (for different kinds of search)

For both (1) & (2), might need

- init()
- clear()
- find free area (allocate())
- free allocated area

Iteration #2 of
Search Engine

— Store data in Files instead of memory



revise Node to use addresses of ArrayList

+ maybe a StartUp Class to initiate everything?

CheckPoint Class

save ( )
restore ( )

save — can take a big Java object, convert to byte array, write into a file
(filename is a constant)

restore — reads file, converts bytes back into original object

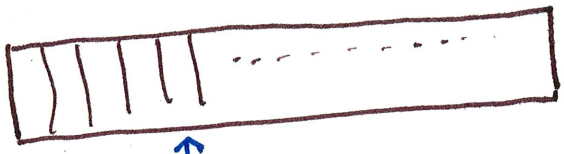
- use Field.convert( ), Field.revert( )
- unit test is easy
- special case:
  first-time, file does not exist

Allocate Class

allocate ( )

free (area number)

Idea array of bits or booleans



← can expand if needed

at index i:

true or 1 ⇒ area i in use

false or 0 ⇒ area i is free

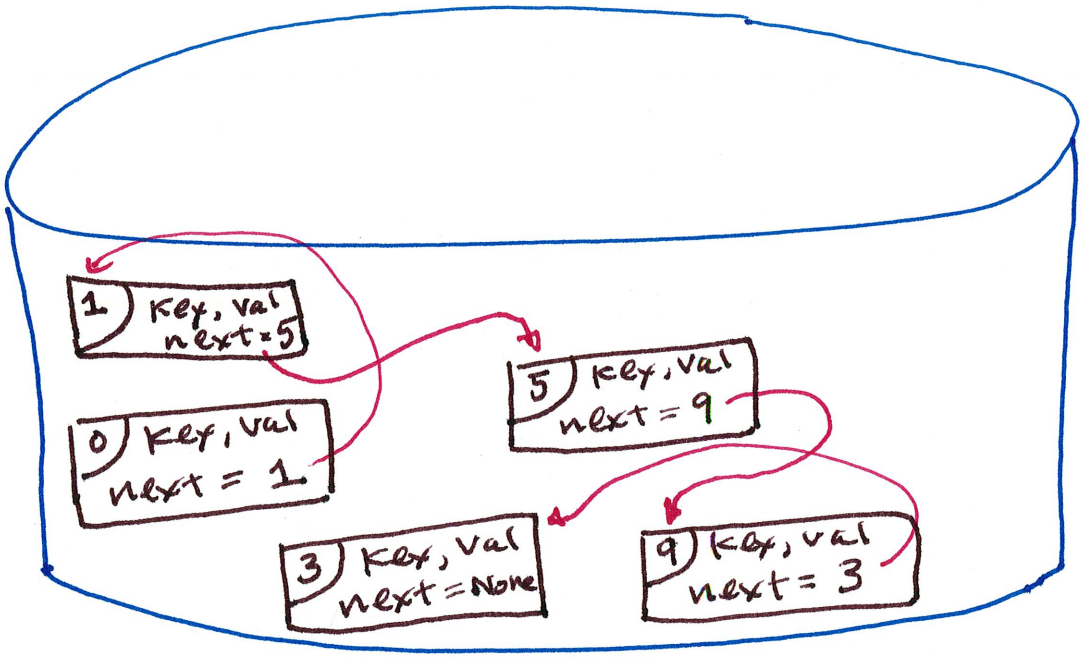
Each index position refers to a disk area of 1024 bytes

- use Java BitSet to do the work
- use CheckPoint to load, save the BitSet object

Key Storage Class

Idea:

Linked List of Node objects, but each Node has area on disk (1024 bytes)



Note: area 0 is always first in list
(key & value are "fake")

Key Storage Continued

get (area #) — read Node

put (area #, Node)

add (Node)

del (Node)

- uses DiskSpace Class for actual file read/write
- converts byte[] to Node
- converts Node to byte[]

} use Field. convert revert

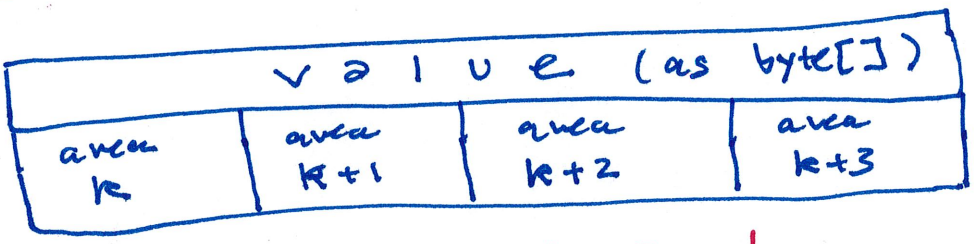
- uses Allocate to get new area for add
- uses Allocate to free area for del
- for unit test: new method — return ArrayList<Node> of current list on disk

Value Storage Class

idea:
"value" stands for an array of identifiers (returned by FieldSearch, modified by Indexer)

Problem value could be larger than 1024 bytes — so one value uses multiple areas!

Solution try



hmmm... flawed solution!

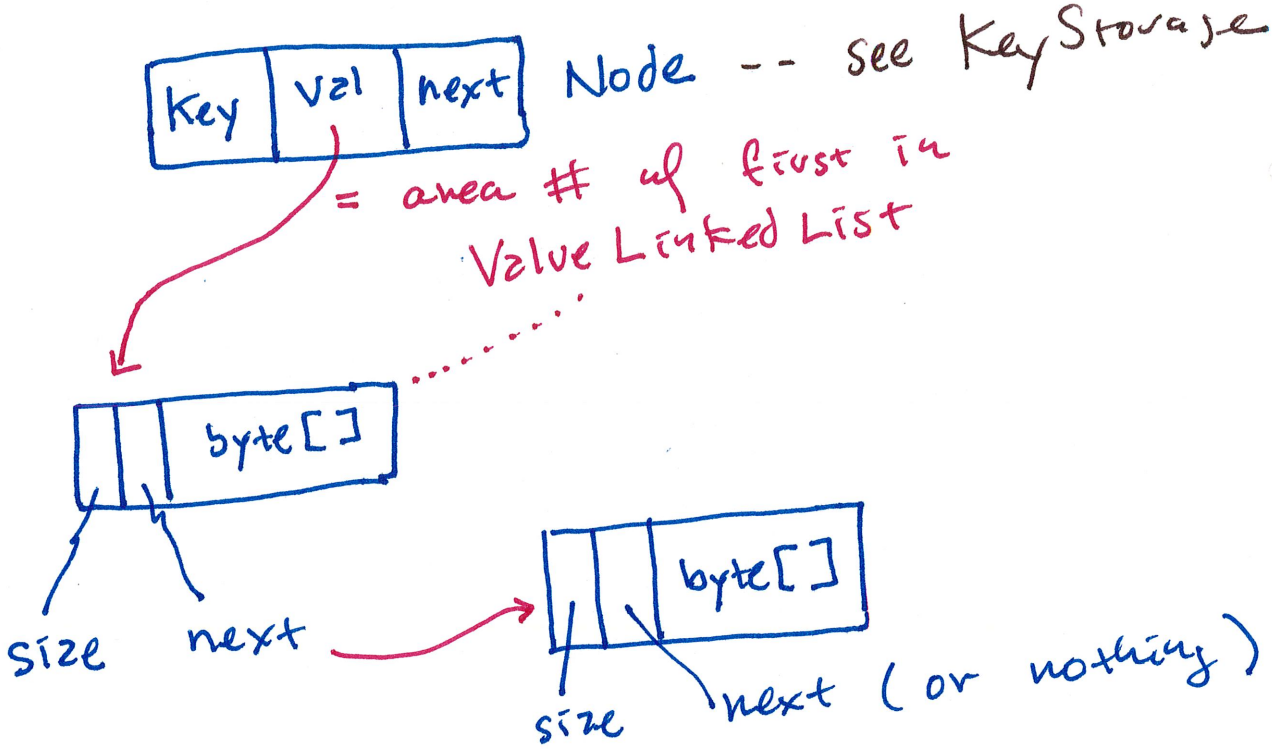
(1) finding free area #'s k, k+1, ... k+m not so easy.

(2) when reading from disk, who knows how many areas to get?

Value Storage continued

Alt Solution

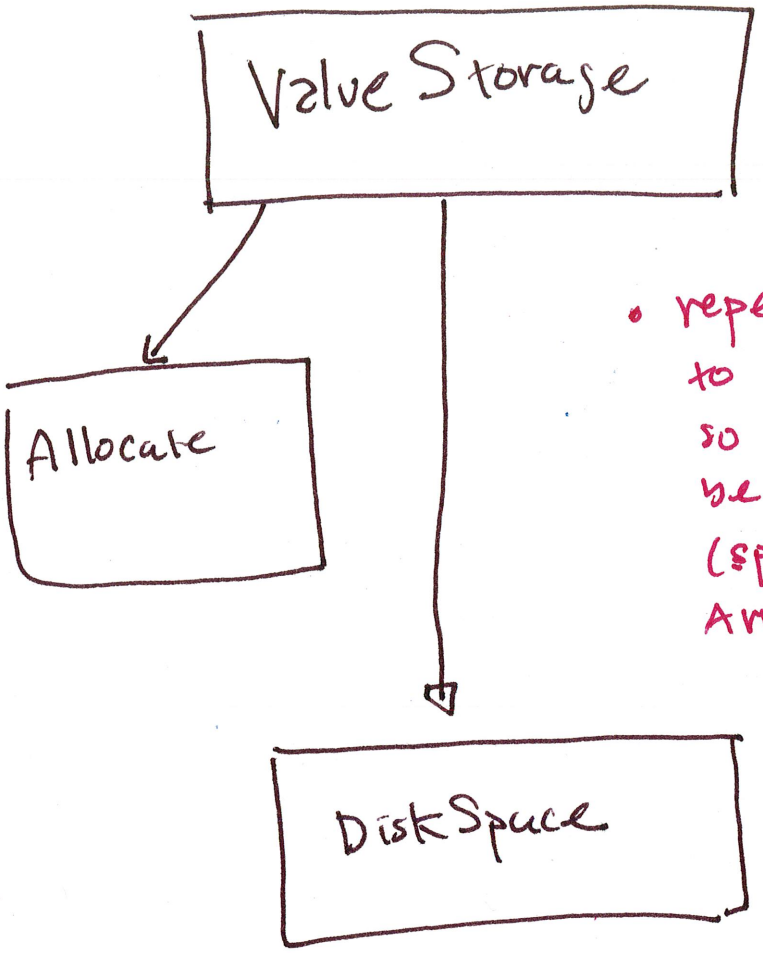
Linked List of Areas



methods:

load( ) — read entire ValueLinkedList, convert into ArrayList of Identifiers

store( ) — write entire ArrayList of Identifiers



- repeated calls to Allocate to obtain free areas, so that LinkedList can be created to contain (split up) the byte[] for ArrayList of Identifiers

- repeated calls to DiskSpace to write into the areas

- repeated calls to Allocated to delete a linked list's areas

DiskSpace Class

writeArea (areaNo, byte[])

readArea (areaNo) returns byte[]

Use Java's RandomAccessFile

Note:

- random reads cannot go beyond end of file!
- random write starting at current end of file increases file size.
- ⇒ constructor needs to learn if file exists, create it if needed, learn current size

Node Class

rewrite / refactor

data members

byte [ ] Key currently this is Field object converted to byte [ ]

- ~~ArrayList<String> Identifiers~~

long ValueArea where the ValueStorage area starts

methods:

add (Identifier)

del (Identifier)

getIdentifiers ( ) return ArrayList of Identifiers (calls ValueStorage)