File Structures

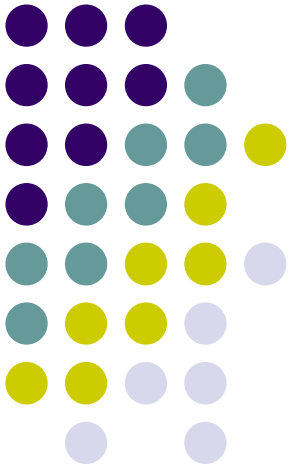
06. Organizing Files for Performance

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU



Slides are based on the textbook.

datalab

data science laboratory

Outline

- 6.1 Data compression
  - skipped
- 6.2 Reclaiming space in files
- 6.3 Finding things quickly
  - An Introduction to internal sorting and binary searching
- 6.4 Keysorting

Modifying a file

- Scenario
  - modify a record in a variable-length record file, where the new record is longer than the original record
    - should keep RID intact
  - Method1
    - append the new record to the end of the file, and then put a pointer from the original record space to the extension of the record
    - slower than it was originally
  - Method2
    - rewrite the whole record at the end of the file, leaving a hole at the original location of the record
    - the file contains waste space

Space reclaiming

- Is the file organization deteriorated?
  - Record addition
    - No deterioration
  - Record updating and deleting
    - Yes!
- Space reclaiming is needed when
  - record updating
    - can be treated as a record deletion followed by a record addition
  - record deletion (we focus on this)
    - want to reuse the space

Record deletion and storage compaction (1/2)

- Storage compaction
  - Look for places in a file where there is no data at all and recover this space
  - Record deletion => empty spaces occur in files
- How to recognize records as deleted?
  - to place a special mark in each deleted record

```
Ames|John|123 Maple|Stillwater|OK|74075|...
Morrison|Sebastian|9035 South Hillcrest|Forest Village|OK|78420|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...
```

```
Ames|John|123 Maple|Stillwater|OK|74075|...
*|Sebastian|9035 South Hillcrest|Forest Village|OK|78420|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...
```

After compaction

```
Ames|John|123 Maple|Stillwater|OK|74075|...
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...
```

Record deletion and storage compaction (2/2)

- how to reuse the space from the deleted record
  - Storage compaction approach does not reuse the space for a while
    - Programs must include logic for ignoring records with deleted marks
    - Benefit: easily undelete a record
  - the reclamation of the space from the deleted records at once
    - a simple compaction
      - a file copy program that skips over deleted records
    - how often to run the storage compaction program
      - based on # of deleted records
      - based on the calender

Dynamic Storage reclamation

- Storage compaction
  - The simplest and most widely used of the storage reclamation methods
  - In some applications wants to reuse the space from deleted records as soon as possible
- dynamic storage reclamation
  - to make record reuse happen more quickly
    - know immediately if there are empty slots in the file
    - jump directly to one of those slots if they exist

Case 1 . Deleting fixed-length records (1/2)

- How to reuse the space from deleted records
  - 1. deleted records must be marked in special way
  - 2. we could find the space which deleted records occupied

- Much slower approach when working with fixed-length records
  - Search sequentially through a file before adding a file
    - If deleted record is found → reuse the space
    - If not, the new record can be appended at the end

Case 2 . Deleting fixed-length records (1/2)

- To make record reuse quickly, we need
  - (1) a way to know immediately if there are empty slots in the file
  - (2) a way to jump directly to one of those slots if they exist

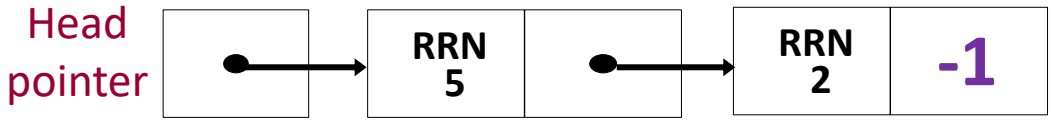
- Solution
  - Linked lists or Stacks for avail list
    - avail list : a list that is made up of deleted records

Detour: Linked lists and Stacks

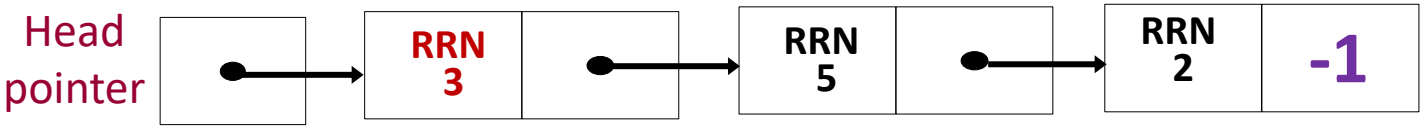
- Linked Lists
  - the available space list made up of deleted records

List of length 2:

- Stacks
  - handle a list as a stack
    - All insertions and removals of nodes take place at one end of the list



Before



After adding the New node

Linking and Stacking deleted records (1/4)

- Placing the related records on stack
  - Meet two criteria for rapid access to reusable space
    - know immediately if there are empty slots in the file

    : The top of stacks contains the end-of-list value → no empty slots
    - jump directly to one of those slots if it exists

    : the stack top contains a valid node reference → reuse it
- Where do we keep the stack
  - A separate list (perhaps in a separate file)
  - Embedded within the data file

Linking and Stacking deleted records (2/4)

- Embedded within the data file
  - arranging and rearranging the links used to make one available record slot point to the next
    - deleted records are marked by replacing the first field with an asterisk
    - second field of deleted record points to next record
  - pointing is not done with pointers(byte offset), but RRNs. Why?
    - The reason: working with fixed-length records in a disk file

Linking and Stacking deleted records (3/4)

- Example
  - Files with seven records (RRNs 0-6)

List head (first available record) => -1

0	1	2	3	4	5	6
Edwards...	Bates...	Wills...	Park….	Maters...	Browns...	Chavez

  - Deleting record 3

List head (first available record) => 3

0	1	2	3	4	5	6
Edwards...	Bates...	Wills...	*-1	Maters...	Browns...	Chavez

  - Deleting record 5

List head (first available record) => 5

0	1	2	3	4	5	6
Edwards...	Bates...	Wills...	*-1	Maters...	*3	Chavez

Linking and Stacking deleted records (4/4)

- Example (cont'd)
  - Deleting record 1

List head (first available record) => 1

0	1	2	3	4	5	6
Edwards...	*5	Wills...	*-1	Maters...	*3	Chavez

  - After adding new records
    - 1st on slot RRN1, 2nd on Slot RRN5, 3rd on Slot RRN3

List head (first available record) => -1

0	1	2	3	4	5	6
Edwards...	1st new rec	Wills...	3rd new rec	Maters...	2nd new rec	Chavez

Implementing fixed-length record deletion

- Place deleted records on a linked avail list
  - Treat avail list as a stack
  - need a suitable place to keep the RRN of the first available record on the avail list
    - place the first available record in a header record at the start of the file

Case 2. Deleting variable-length records

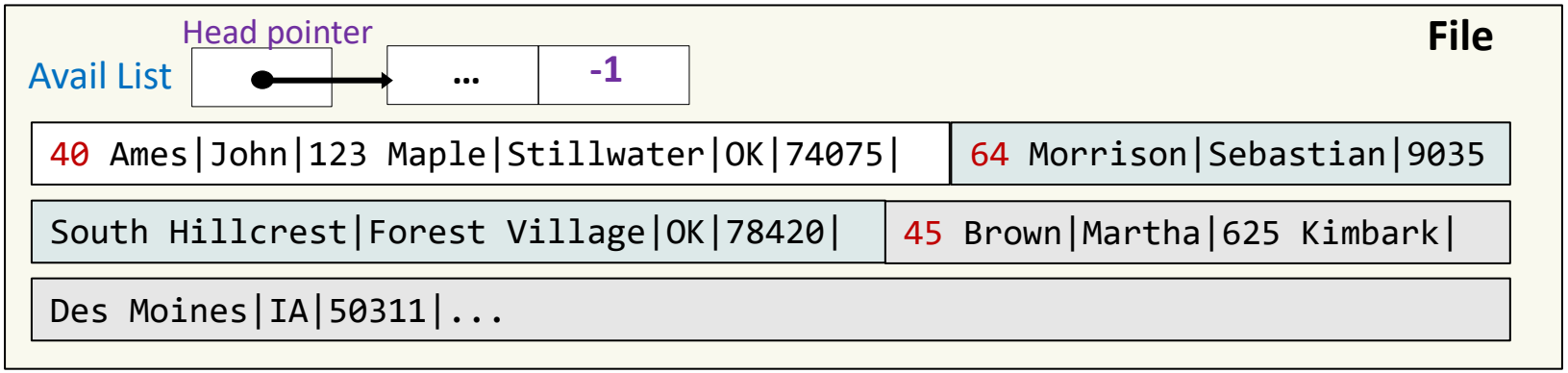
- Supporting record reuse through an avail list
  - link the deleted records together into a list
  - an algorithm for adding newly deleted records to the avail list
  - an algorithm for finding and removing records from the avail list

An avail list of variable-length records (1/2)

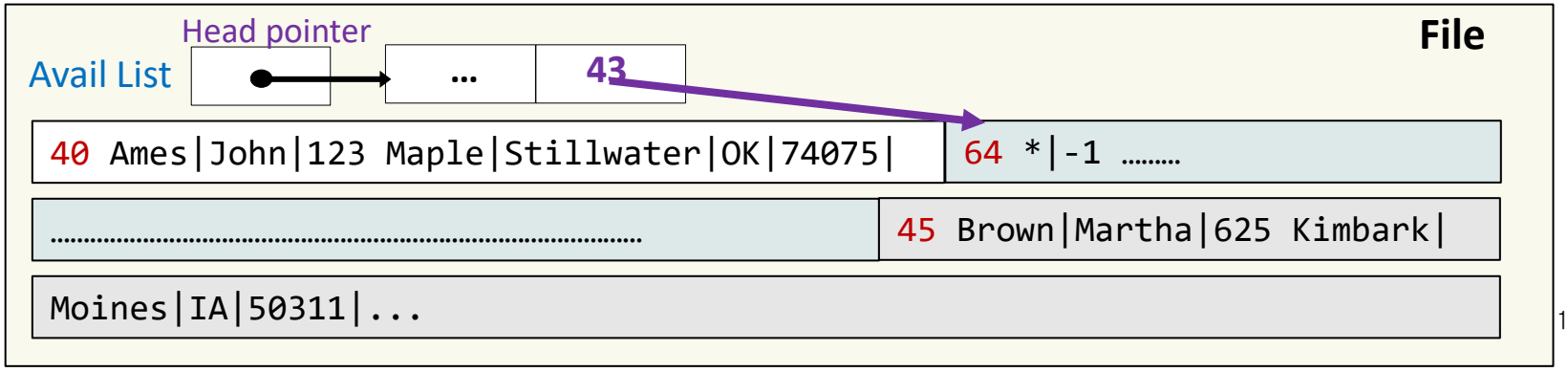
- What kind of file structure do we need?
  - the length of each record by placing a byte count of the record contents at the beginning of each record
  - E.g) VariableLengthBuffer
- How to implement deletion
  - place a single asterisk in the first field, followed by a link field pointing to the next deleted record on the avail list
    - a link field: contain the byte offset
      - cannot use RRN for links

An avail list of variable-length records (2/2)

- Example
  - A file with three records

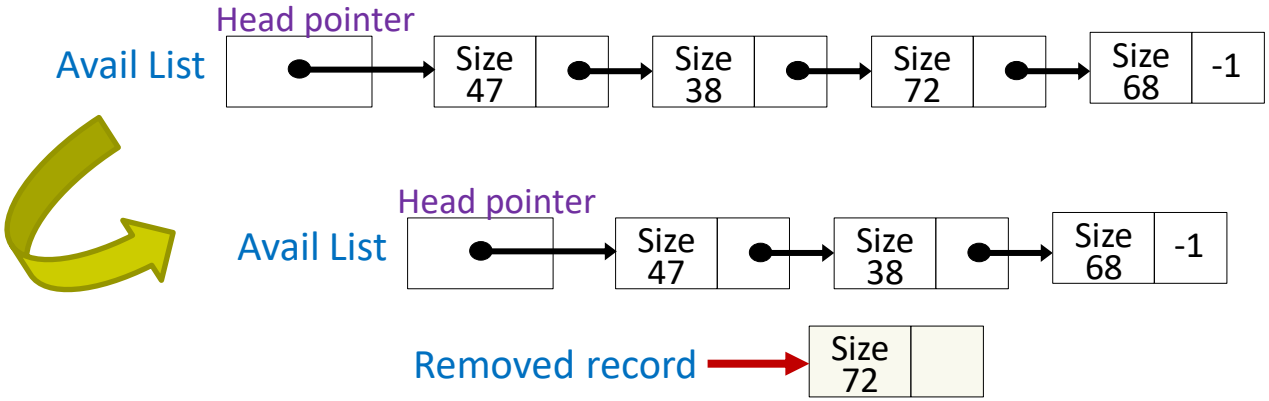


  - After deleting the second records



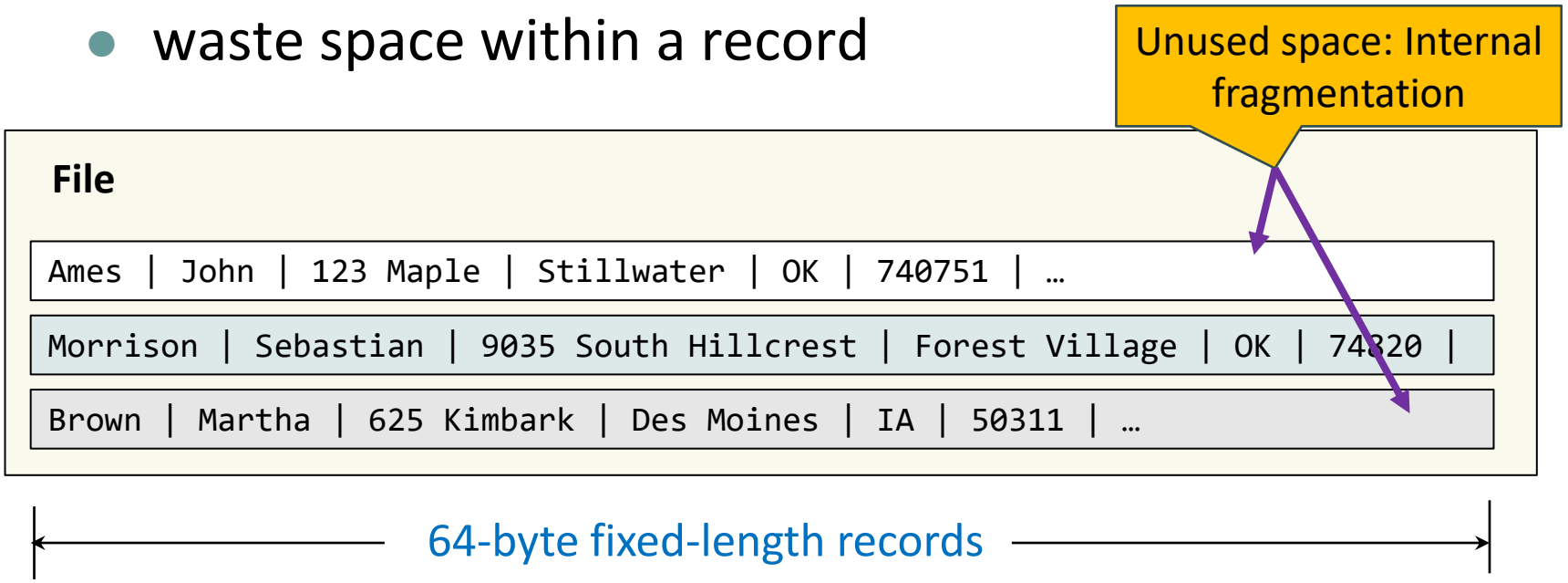
Adding and removing avail list records

- Avail list
  - Problem: cannot access the avail list as a stack
    - The reason: the record slots on the avail list differ in size
  - need to search through the avail list for a record slot that is the right size
    - define right size as "big enough"
    - a record that requires 55 bytes is to be added
      - 72(big enough) → remove it from the avail list



Internal fragmentation (fixed-length record)

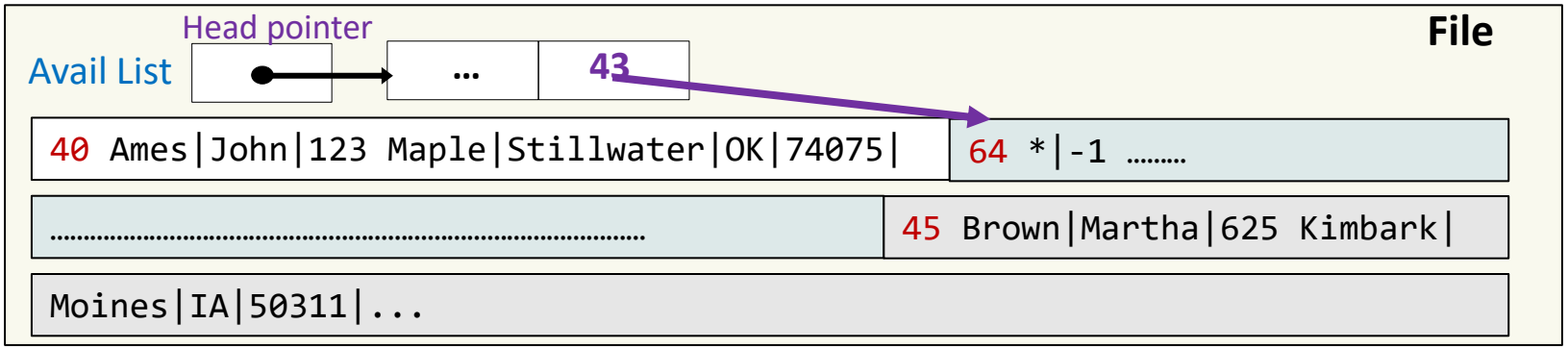
- Internal fragmentation (in fixed-length record)
  - waste space within a record



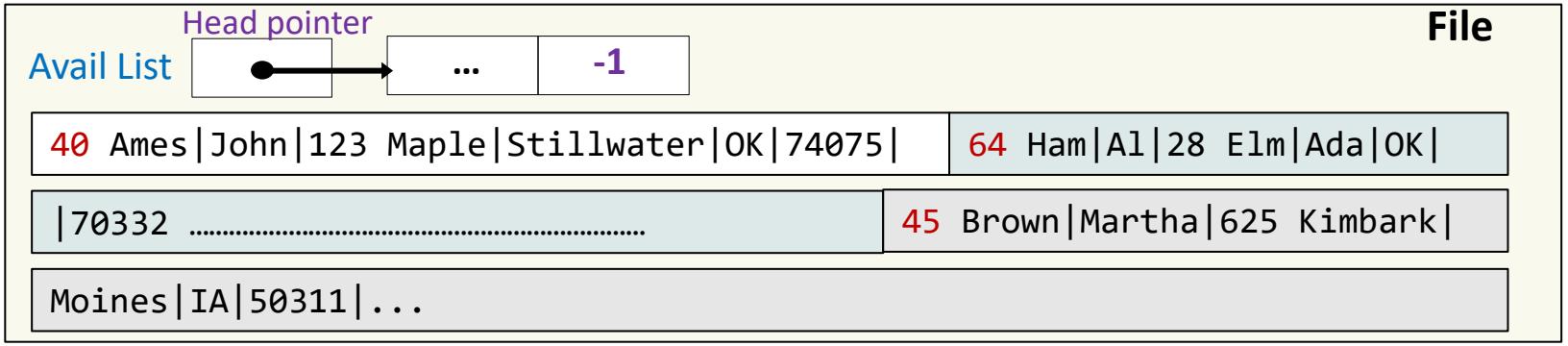
  - variable-length records
    - minimize wasted space by doing away with internal fragmentation

External Fragmentation (variable length records) (1/4)

- Simple approach
  - After deleting the second records

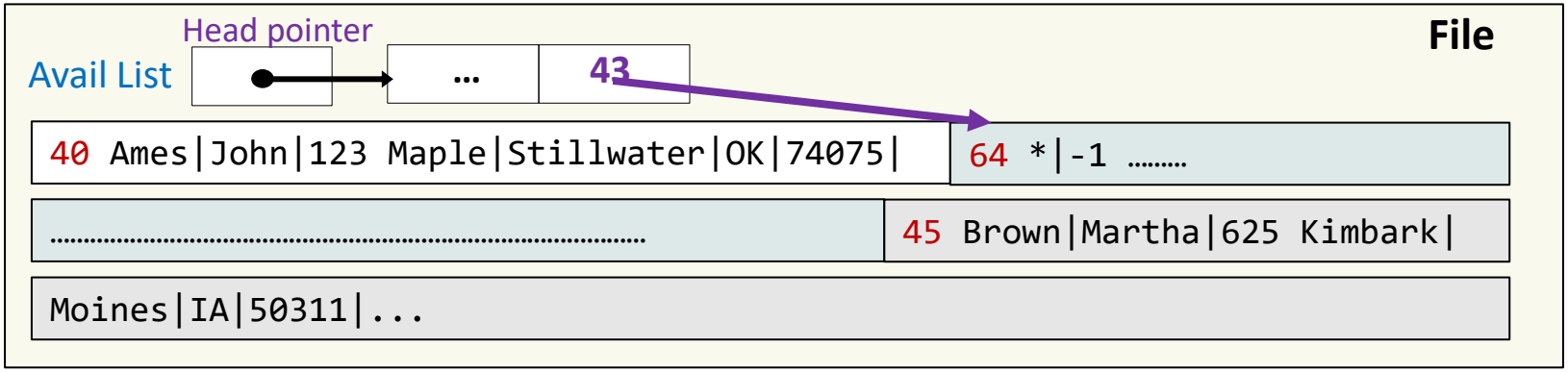


  - Adding new record: Ham|Al|28 Elm|Ada|OK|70332
    - keeping the 64-byte record slot intact → fragmentation

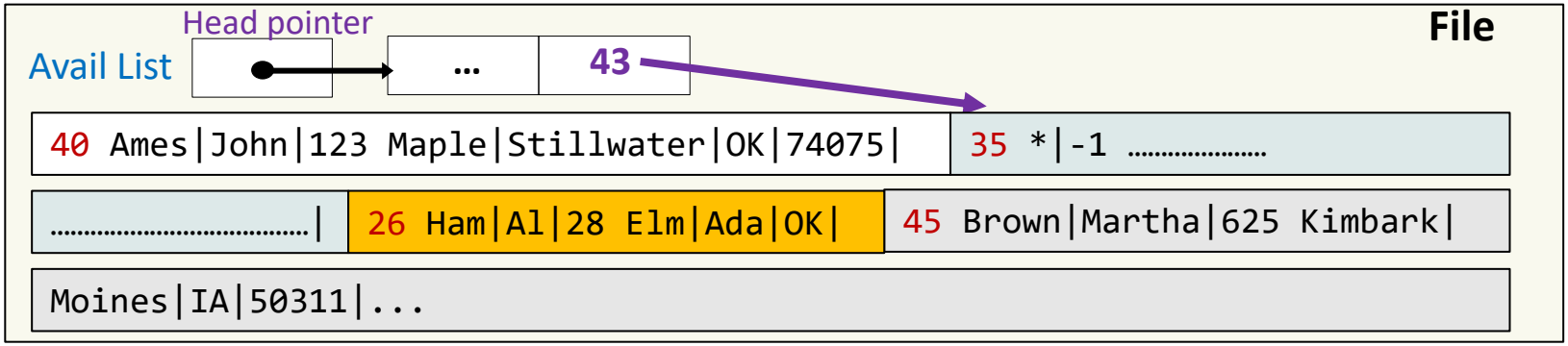


External Fragmentation (variable length records) (2/4)

- Another approach
  - After deleting the second records

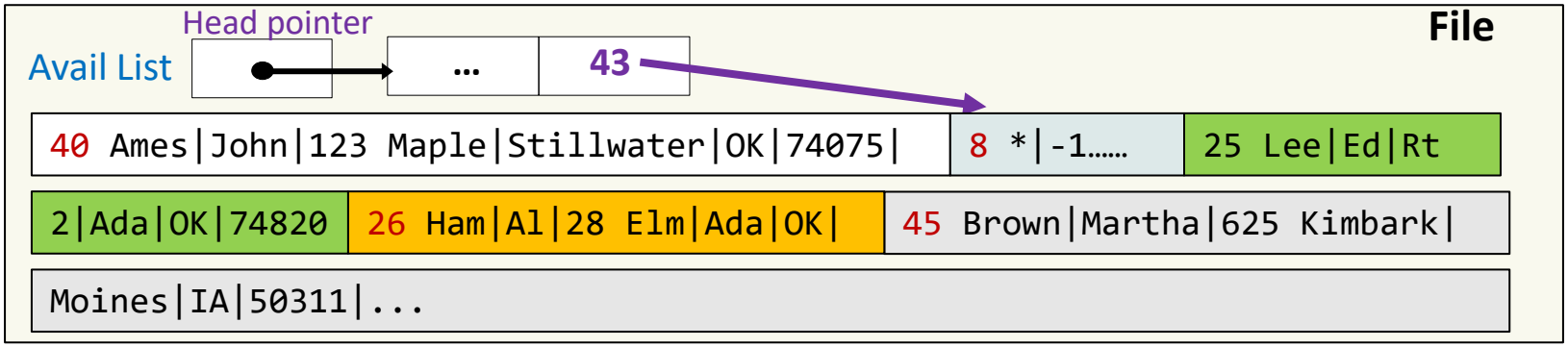


  - Adding new record: Ham|Al|28 Elm|Ada|OK|70332|
    - Break the 64-byte record two parts: new rec and avail list



External Fragmentation (variable length records) (3/4)

- Another approach (cont'd)
  - Adding new record: `Lee|Ed|Rt 2|Ada|OK|74820|`



External Fragmentation (variable length records) (4/4)

- External fragmentation
  - too small space that the probability to reuse it is close to zero
    - Too fragmented to be reused
  - three possible solutions
    - storage compaction
      - regenerate the file when external fragmentation becomes intolerable
    - coalescing the holes
      - Combining adjacent record slots to make a single, larger record slot
    - minimizing fragmentation by adopting placement strategy

Placement strategies (1/2)

- ways to add and remove variable-length records from an avail list

- need to remove a record slot from the avail list (to add a record to the file)

Placement strategies (2/2)

- find a record slot that is big enough
  - First-fit
    - select the first available record slot
    - suitable when lost space is due to internal fragmentation
  - Best-fit
    - select the available record slot closest in size
    - avail list in ascending order
    - suitable when lost space is due to internal fragmentation
  - Worst-fit
    - select the largest record slot
    - avail list in descending order
    - suitable when lost space is due to external fragmentation

Outline

- 6.1 Data compression
  - skipped
- 6.2 Reclaiming space in files
- 6.3 Finding things quickly
  - An Introduction to internal sorting and binary searching
- 6.4 Keysorting

Turning point

- Searching and Sorting data in a memory
  - Searching and sorting algorithms → making many comparisons
  - Fast, if the size of data < the size of memory

- search a file for a particular piece of information and sorting a file
  - Goal: minimize the number of disk accesses

Finding things in simple field and record files

- File has fixed length records
  - RRN lets us to compute the record's byte offset and jump to it using direct access
- Access by key
  - Implies a sequential search
  - Have to look through the entire file

Search by guessing: Binary search

- Assumption: a file is sorted
- an algorithm for binary searching on a file of fixed-sized records

```
int BinarySearch (FixedRecordFile &file, RecType &obj,  KeyType &key)
{
    int low = 0;
    int high = file.NumRecs() - 1;
    while (low <= high)
    {
        int guess = (high - low) / 2;
        file.ReadByRRN (obj, guess);
        if (obj.key() == key) return 1; // record found
        if (obj.key() < key) high = guess - 1; // search before guest
        else low = guess + 1; // search after guestt
    }
    return 0;
}
```

Classes for binary search

```
Class KeyType
{
public:
    int operator == (KeyType &);
    int operator < (KeyType &);
};

class RecType
{
public:
    KeyType key();
};

class FixedRecordFile
{
    public:
    int NumRecs();
    int ReadByRRN (RecType &record,
        int RRN);
};
```

- KeyType and RecType
  - the minimum definition to allow a successful completion of BinarySearch
- A class FixedRecordFile
  - NumRecs()
    - return # of records in the FixedRecordFile
  - ReadByRRN()
    - read the record at a specific RRN and unpack it into a RecType object

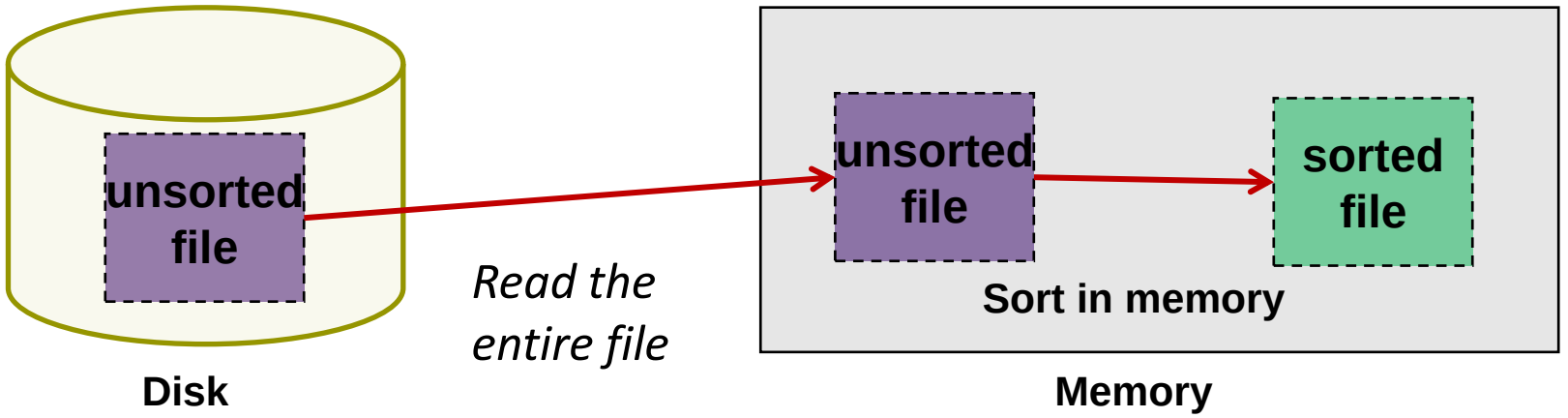
Binary search versus sequential search

- to make use of binary searching, we need to sort a list on the basis of a key
  - a binary search: $O(log_2 n)$
  - a sequential search: $O(n)$

- sorting is a very important part of file processing
  - internal sorting
  - external sorting

Sorting a disk file in RAM

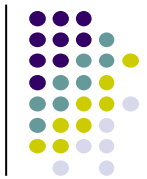
- If applying sorting directory to data stored on a disk
  - A lot of seeking, and rereading of data
- Alternative
  - read the entire file from disk to memory
  - use internal sort (=sort in memory)
    - UNIX sort utility uses internal sort



The limitations of sort then binary search (1/3)

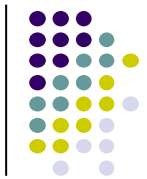
- problem1: binary searching requires more than one or two accesses
  - a binary search requires [log n] comparisons
    - $O(log_2 1{,}000) = 9.5$ accesses per request
    - $O(log_2 100{,}000) = 16.5$ accesses per request
    - Not a negligible cost
  - Alternatives
    - access records by RRN rather than by key → singe access
      - the use of index structures

The limitations of sort then binary search (2/3)



- problem2: keeping a file sorted is very expensive
  - encounter difficulty when we add a record
  - Tradeoff : search benefit v.s. insert overhead
  - accumulate record additions in a transaction file and made in a batch merge → merge sort
  - Better solutions:
    - not involve reordering of the records in the file when a new record is added
      - the use of simple indexes, can involve hashing
    - or, associated with data structures that allow for more rapid, efficient reordering of the file
      - use of tree structures, B-tree
      - to keep the file in order

The limitations of sort then binary search (3/3)



- problem3: an internal sort works only on small files
  - if the file is large, then need external sorting
  - develop a variation on internal sorting called a keysort

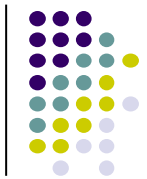
Outline

- 6.1 Data compression
  - skipped
- 6.2 Reclaiming space in files
- 6.3 Finding things quickly
  - An Introduction to internal sorting and binary searching
- 6.4 Keysorting

Keysorting

- Keysort
  - a variation on internal sorting
    - to avoid the sorting of records in a file
  - sort a file in RAM
    - sorted thing is “key” only , called tag sort
    - do not need to read the whole file into RAM during the sorting process
- Sorting procedure
  - 1. Read only the keys into memory
  - 2. Sort the keys
  - 3. Rearrange the records in file by the sorted keys

Descriptions of the method

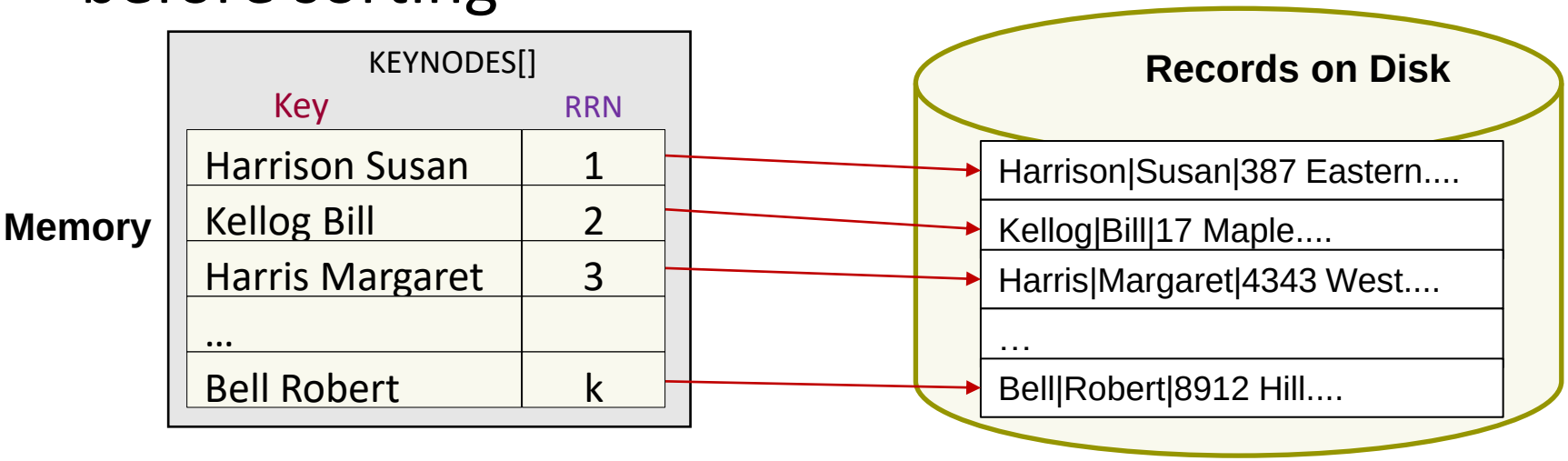


- Assume that we are dealing with a fixed-length record file

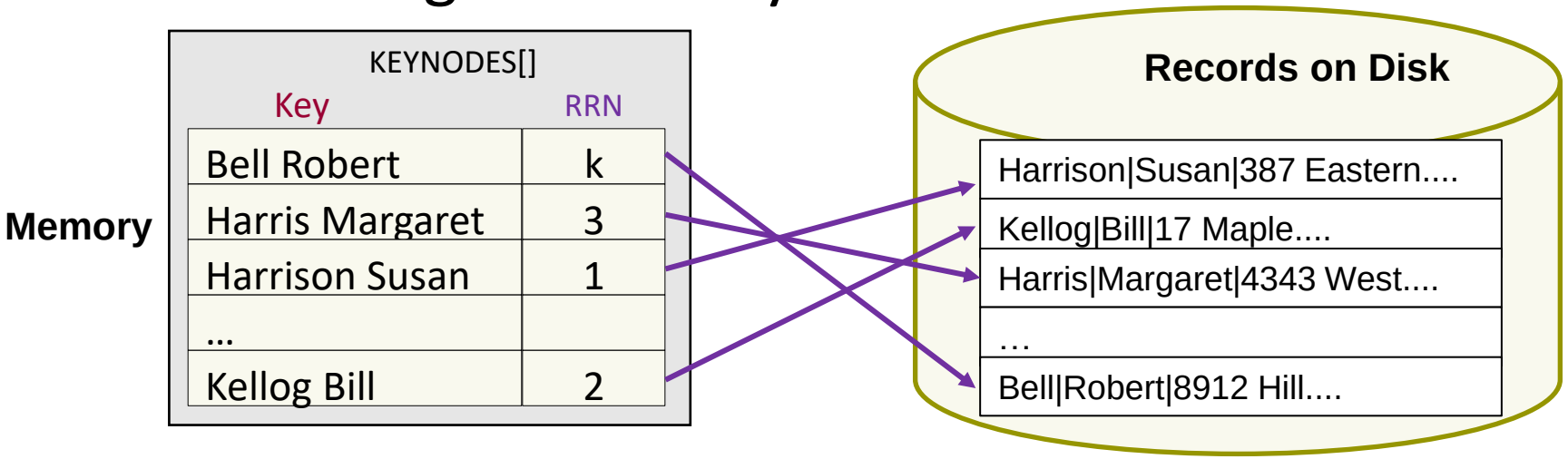
```
class FixedRecordFile
{public:
    int NumRecs();
    int ReadByRRN (RecType &record, int RRN);
  //additional methods required for keysort
    int Create (char *fileName);
    int Append (RecType &record);
};

class KeyRRN
{public:
    KeyType KEY; int RRN;
    KeyRRN();
    KeyRRN (KeyType key, int rrn);
};

int Sort (KeyRRN[], int numKeys);
```

Conceptual view of keysorting (1/2)

- before sorting

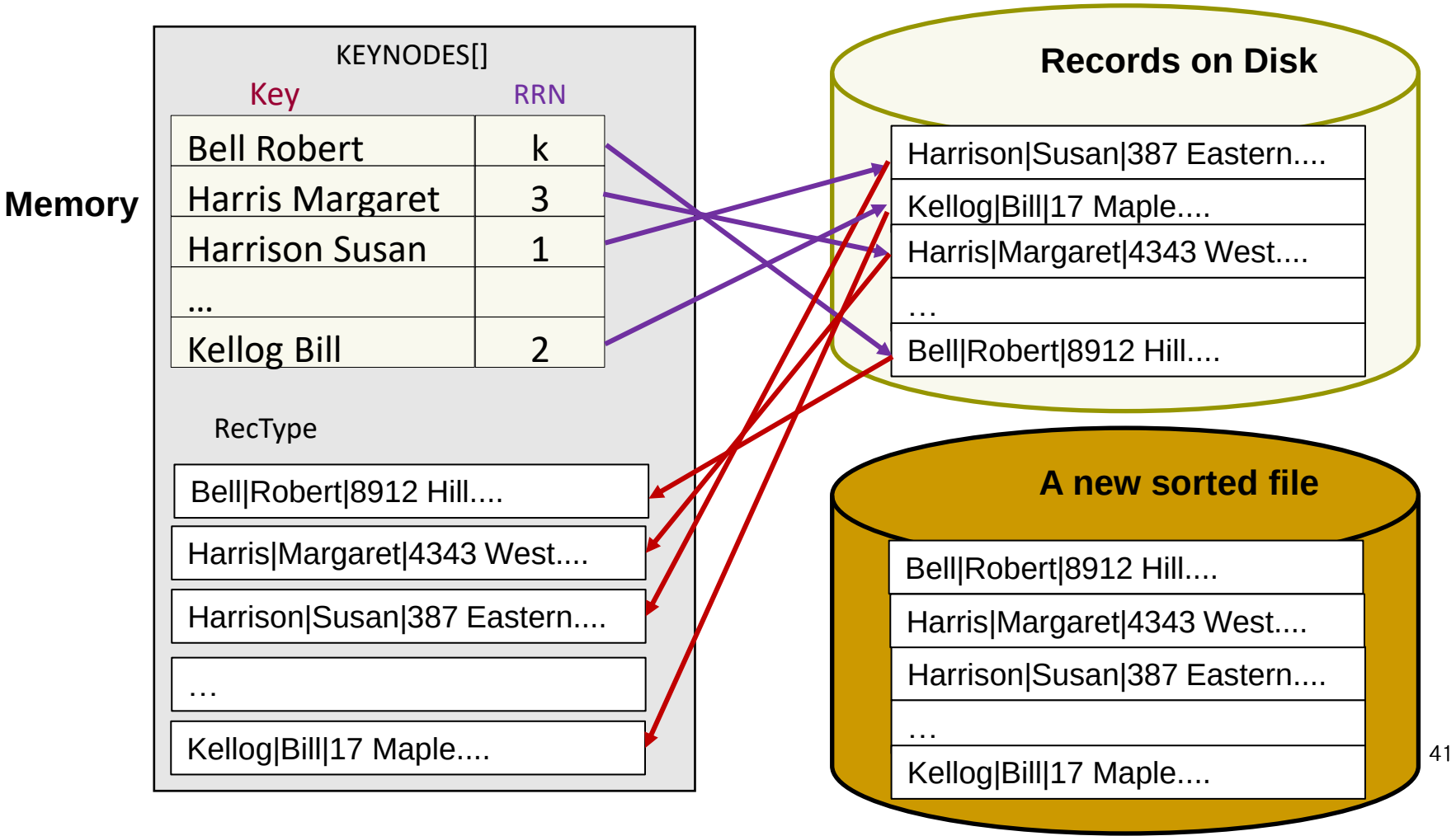


- after sorting in memory



Conceptual view of keysorting (2/2)

- Writing to a new file in the order of KEYNODE[]



Algorithm for keysort (1/2)

- Read the (key, RRN) pairs into an array of KeyRRN objects
- **Sort the KEYNODES[] array according to the KEY field**
- reorganize the file according to new ordering

Algorithm for keysort (2/2)

```
int KeySort(FixedRecordFile & inFile, char * outFileName)
{
    RecType obj;
    KeyRRN * KEYNODES = new KeyRRN[inFile.NumRecs()];
    // read file and load keys
    for (int I = 0; I < inFile.NumRecs(); I++)
    {
        inFile.ReadByRRN(obj, I);//read record I
        KEYNODES[I] = keyRRN(obj.Key(), I); // put key and RRN into KEYNodes
    }
    sort (KEYNODES, inFile.NumRecs()); // sort Keys
    FixedRecordFile outFile;    // file to hold records in key order
    OutFile.Create(outFileName);
    // write new file in key order
    for (int j = 0; j < inFile.NumRecs(); j++)
    {
        inFile.ReadByRRN(obj, KEYNODES[j].RRN); // read in key order
        outFile.Append(obj);    // write in key order
    }
    return 1;
}
```

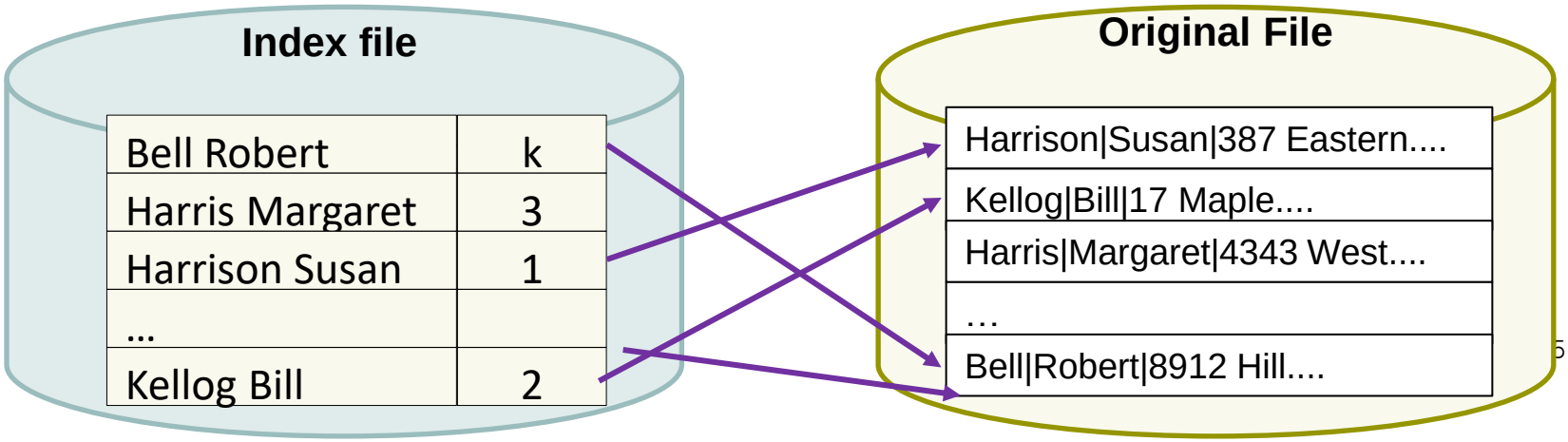
Limitations of the keysort method

- never desirable to read entire records twice
  - One read for building array
  - Another read for writing out the new sorted file

- not reading through the input file sequentially
  - Seek in the input file to the record whose KEYNODES[i].RRN
    - require many random seeks into the input file
  - the getting-something-for-nothing aspect of keysort

Another solution: why bother to write the file back?

- the problem of rearranging all the records in the file
  - what if we skip to write out a sorted version of the file?
    - simply write out a copy of the array of canonical key nodes
    - create a second kind of file, an index file
    - output an index to the original file
- One of our favorite categories of solutions to computer science problems
  - If looking like a bottleneck, consider skipping it altogether

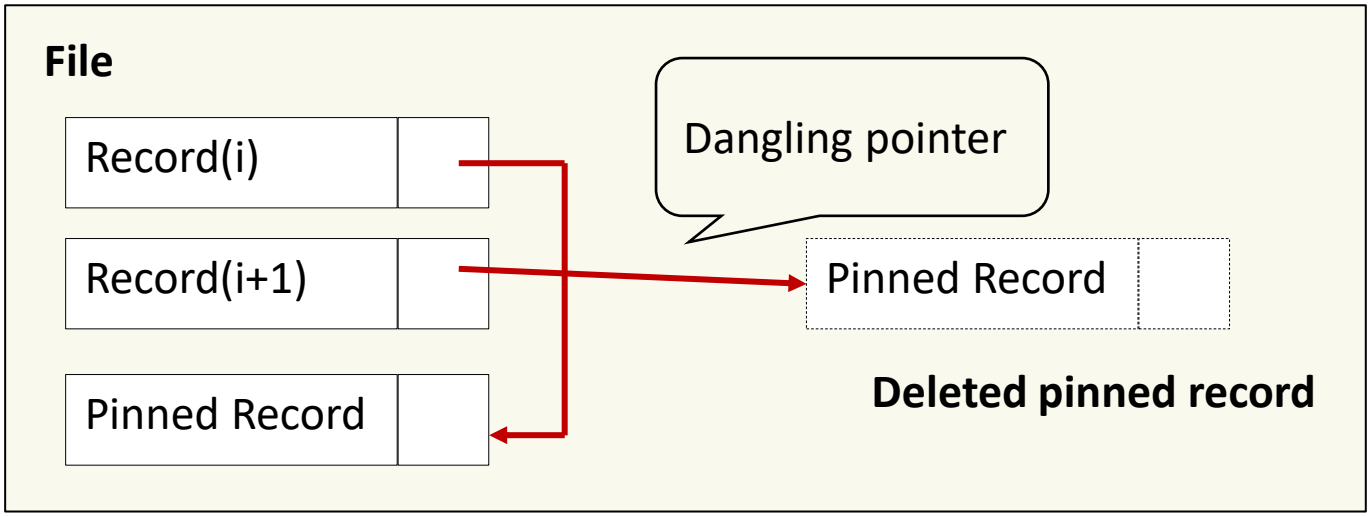


Pinned records (1/2)

- An avail list of deleted record slots
  - Created by linking all of the available slots together
  - linking is done by writing a link field into each deleted record that points to the next deleted record
    - a link field: have a pointer, physical locations
- Pinned records
  - Records that are referenced to physical location of themselves by other records

Pinned records (2/2)

- A File with pinned records



- to use an index file
  - to keep the sorted order of the records, while keeping the actual data file in its original order
  - need to take a close look at the use of indexes

Q&A