Chapter 3
20 January 2011

Objectives

- An outline of a program to manage flat files of records.
- A simple sorting algorithm.
- A look ahead to improved methods for sorting and searching.
- A look ahead to more sophisticated approaches for abstracting computations to make programs more general and more reliable.
- A binary search algorithm, the first “mature" algorithm covered in this text.

A Flat File

Basically like a spreadsheet;

rows are individual records;

columns are fields common to each row.

NAME	OFFICE	PHONE	CLASS
Herbert	2A41	789.0123	390
Lander	2A47	789.7890	146
Winthrop	3A71	780.4667	611
Marion	2A13	789.1536	750
Adams	3A56	789.8702	522
Smith	2A46	789.5275	101
Axolotl	3A22	789.2749	102
Zokni	3A39	789.9111	350
Igen	2A21	789.6050	240
Nem	2A89	789.3383	790

A Flat File (2)

A simple data layout but not very general specifically because it is simple...

Original Data

NAME	OFFICE	PHONE	CLASS
Herbert	2A41	789.0123	390
Lander	2A47	789.7890	146
Winthrop	3A71	780.4667	611
Marion	2A13	789.1536	750
Adams	3A56	789.8702	522
Smith	2A46	789.5275	101
Axolotl	3A22	789.2749	102
Zokni	3A39	789.9111	350
Igen	2A21	789.6050	240
Nem	2A89	789.3383	790

Sorted by Office

NAME	OFFICE	PHONE	CLASS
Marion	2A13	789.1536	750
Igen	2A21	789.6050	240
Herbert	2A41	789.0123	390
Smith	2A46	789.5275	101
Lander	2A47	789.7890	146
Nem	2A89	789.3383	790
Axolotl	3A22	789.2749	102
Zokni	3A39	789.9111	350
Adams	3A56	789.8702	522
Winthrop	3A71	780.4667	611

Sorted by Name

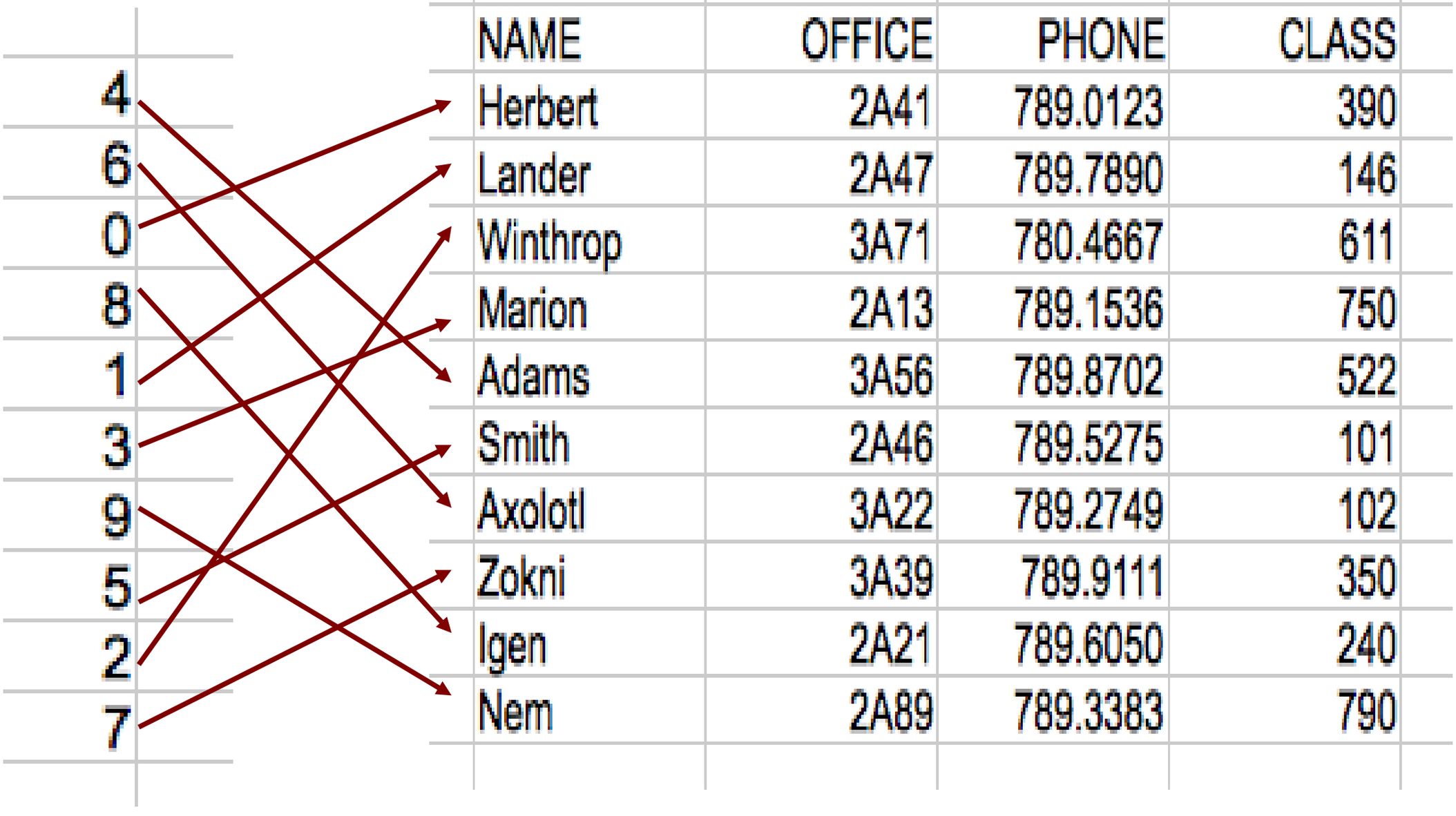
NAME	OFFICE	PHONE	CLASS
Adams	3A56	789.8702	522
Axolotl	3A22	789.2749	102
Herbert	2A41	789.0123	390
Igen	2A21	789.6050	240
Lander	2A47	789.7890	146
Marion	2A13	789.1536	750
Nem	2A89	789.3383	790
Smith	2A46	789.5275	101
Winthrop	3A71	780.4667	611
Zokni	3A39	789.9111	350

Sorted by Class

NAME	OFFICE	PHONE	CLASS
Smith	2A46	789.5275	101
Axolotl	3A22	789.2749	102
Lander	2A47	789.7890	146
Igen	2A21	789.6050	240
Zokni	3A39	789.9111	350
Herbert	2A41	789.0123	390
Adams	3A56	789.8702	522
Winthrop	3A71	780.4667	611
Marion	2A13	789.1536	750
Nem	2A89	789.3383	790

A Flat File (3)

Often we have to *index* the file into a sorted order

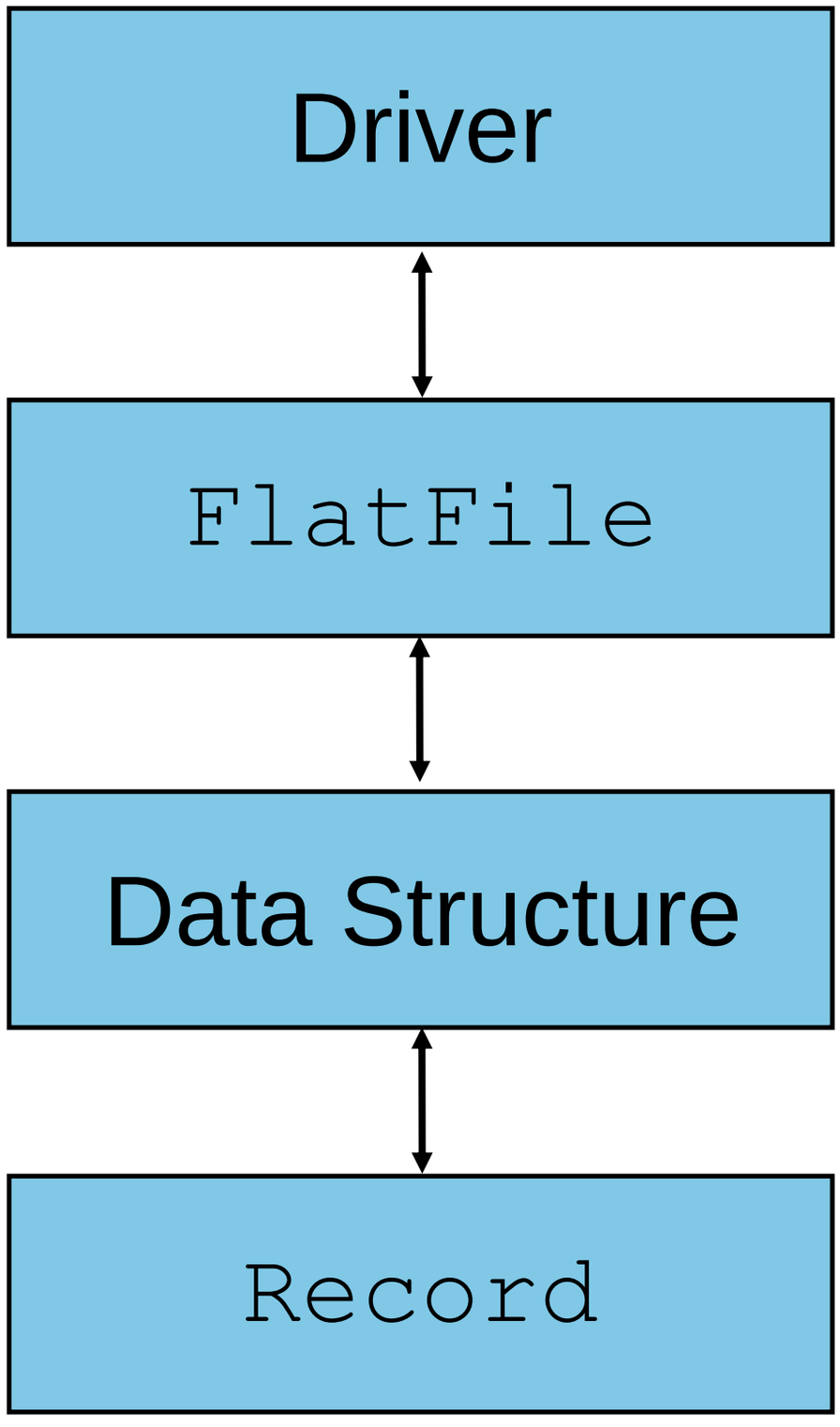


	NAME	OFFICE	PHONE	CLASS
4	Herbert	2A41	789.0123	390
6	Lander	2A47	789.7890	146
0	Winthrop	3A71	780.4667	611
8	Marion	2A13	789.1536	750
1	Adams	3A56	789.8702	522
3	Smith	2A46	789.5275	101
9	Axolotl	3A22	789.2749	102
5	Zokni	3A39	789.9111	350
2	Igen	2A21	789.6050	240
7	Nem	2A89	789.3383	790

A Flat File – Methods Needed (3)

- Read the entire file from disk into a structure (like an `ArrayList`)
- Display the entire structure
- Sort the entire structure (which requires the ability to compare)
- Search the structure for an entry with a given *key*

The Program Hierarchy



Code that opens and closes files and invokes the `FlatFile` application

This code implements the flat file view to the user

In this case, an `ArrayList` inside the `FlatFile` code

The abstract data type (ADT) that is the data payload

UML for the `FlatFile` Class

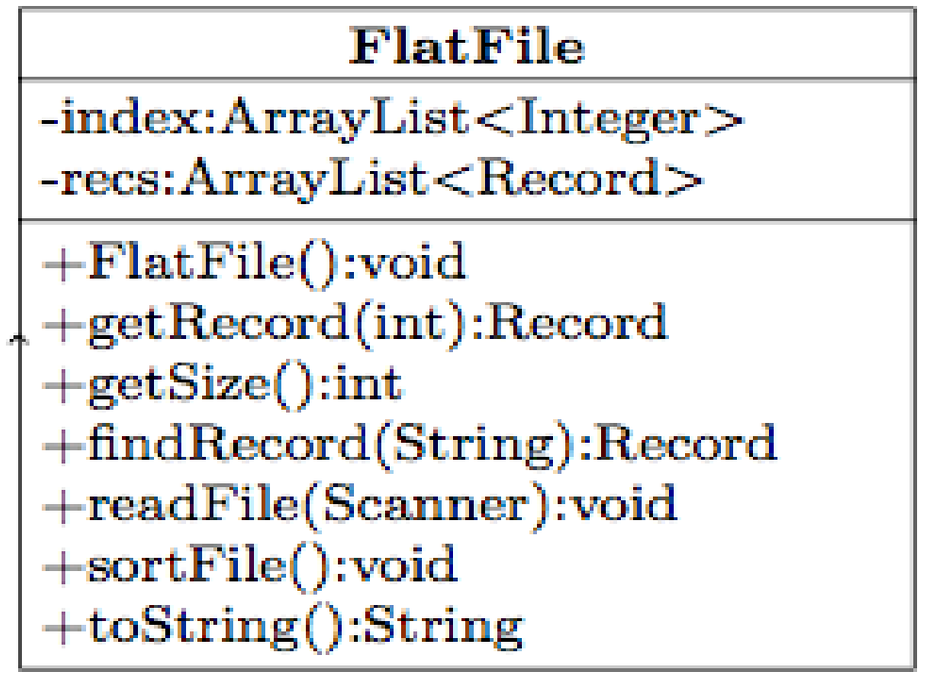


Figure 3.2 The UML diagram for the `FlatFile` class

In a more serious application, we would have a “real” data structure instead of the `ArrayList`

UML for the `Record` Class

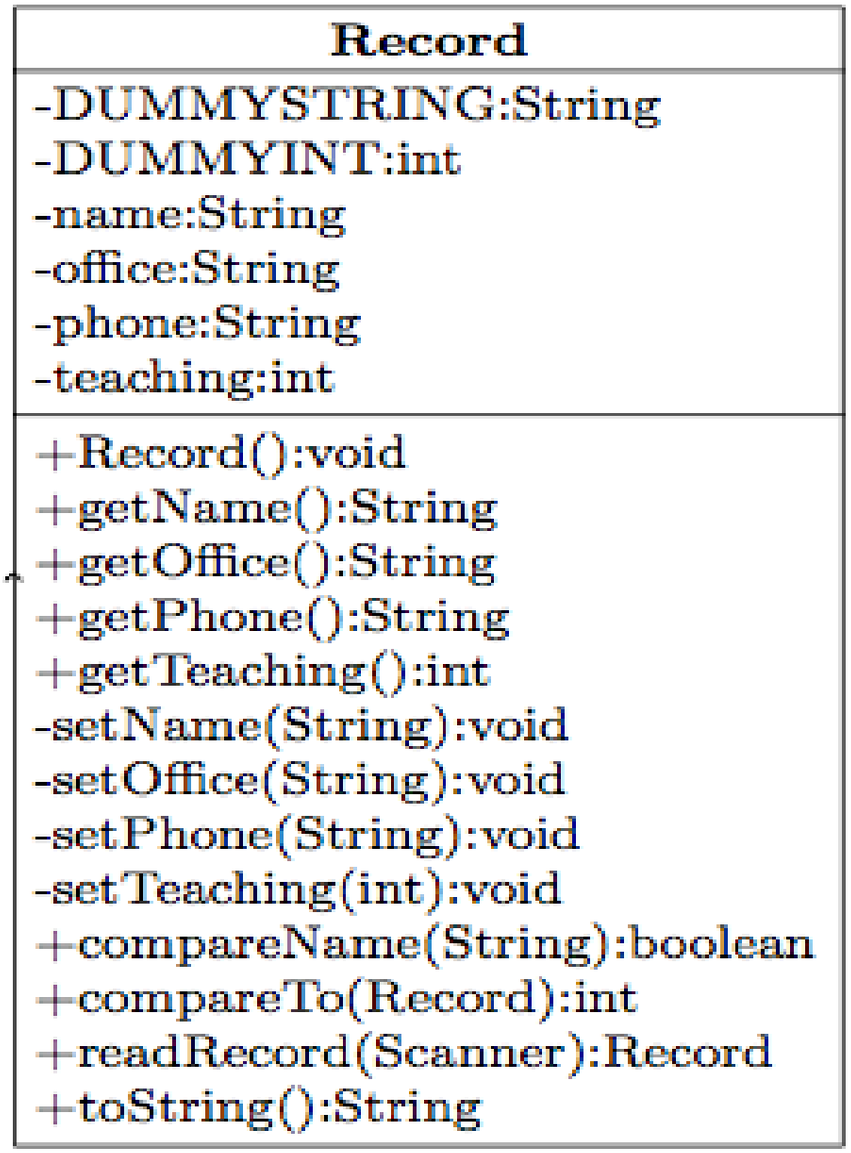


Figure 3.1 The UML diagram for the `Record` class

Input Data

In this case, a simple ASCII file

Herbert 2A41 789.0123 390

Lander   2A47 789.7890 146

Winthrop 3A71 789.4667 611

Marion   2A13 789.1536 750

Adams   3A56 789.8702 522

Smith   2A46 789.5275 101

Axolotl 3A22 789.2749 10

Zokni   3A39 789.9111 350

Igen   2A21 789.6050 240

Nem   2A89 789.3383 79

Output Data

For this example, we output log information as well as the output data,

Including the data as read in ...

```
Main:  create and write out the empty file

Main:  empty file was created
Main:  read the data
Main:  the data has been read
Main:  write out the file
FlatFile: (idx,rec) (0,0) Herbert    2A41  789.0123   390
FlatFile: (idx,rec) (1,1) Lander     2A47  789.7890   146
FlatFile: (idx,rec) (2,2) Winthrop   3A71  789.4667   611
FlatFile: (idx,rec) (3,3) Marion     2A13  789.1536   750
FlatFile: (idx,rec) (4,4) Adams      3A56  789.8702   522
FlatFile: (idx,rec) (5,5) Smith      2A46  789.5275   101
FlatFile: (idx,rec) (6,6) Axolotl    3A22  789.2749   102
FlatFile: (idx,rec) (7,7) Zokni      3A39  789.9111   350
FlatFile: (idx,rec) (8,8) Igen       2A21  789.6050   240
FlatFile: (idx,rec) (9,9) Nem        2A89  789.3383   790
```

Output Data (2)

And the data as we manipulate it ...

```
Main:  done with the write
Main:  sort the file
Main:  after sorting, write the file
Main:  write out the file
FlatFile: (idx,rec) (0,4) Adams      3A56  789.8702   522
FlatFile: (idx,rec) (1,6) Axolotl    3A22  789.2749   102
FlatFile: (idx,rec) (2,0) Herbert    2A41  789.0123   390
FlatFile: (idx,rec) (3,8) Igen       2A21  789.6050   240
FlatFile: (idx,rec) (4,1) Lander     2A47  789.7890   146
FlatFile: (idx,rec) (5,3) Marion     2A13  789.1536   750
FlatFile: (idx,rec) (6,9) Nem        2A89  789.3383   790
FlatFile: (idx,rec) (7,5) Smith      2A46  789.5275   101
FlatFile: (idx,rec) (8,2) Winthrop   3A71  789.4667   611
FlatFile: (idx,rec) (9,7) Zokni      3A39  789.9111   350

Main:  done with the write
Main:  find the data item 'Buell'
Main:  record Buell not found
Main:  done with first search
Main:  find the data item 'Smith'
Main:  found record 'Smith      2A46  789.5275   101'
Main:  done with first search
Main:  done with main
```

Figure 3.4 Output data for the indexed flat file example

The `FileUtils` Class

In all later examples, we will use a special FileUtils class to open and close files, do proper testing that files have been open, etc. This allows us to write and test these methods once and then use them and allows us to avoid having to

a) remember all the details every time

b) clutter up the regular code with file housekeeping detail

`FlatFile` Processing

Among other things, we have to sort data

Physical sorting (as with books on a bookshelf) we do only if

a) we don't have a lot of data to move around;

b) the records will be accessed only in one way;

c) the data will be fairly static so we can move it once only.

Most of the time we will sort to create an index

This requires indirect referencing

The Result of Indexing

Sorted order subscript	Physical order subscript	Last Name
0	4	Adams
1	6	Axolotl
2	0	Herbert
3	8	Igen
4	1	Lander
5	3	Marion
6	9	Nem
7	5	Smith
8	2	Winthrop
9	7	Zokni

Figure 3.5 Indexed file

Sorting with Bubblesort

- Not the best sorting method (some would say it's so bad it should never be taught)
- But it is simple, and it works ok on very small sets of data.
- With each outer loop, we push “the next smallest” element to the front of the list

```
for( i from 0 to last-1 )
{
  for( j from i+1 to last )
  {
    if(record(j) < record(i))
    {
      swap record(i) and record(j)
    }
  }
}
```

Sorting with Bubblesort

- On a list of N items, the inner loop is executed

  $(N-1) + (N-2) + \ldots + 2 + 1 = (N)(N-1)/2 = (1/2)(N^2 - N)$ times

- Efficient sorts would take C (N log N) loops

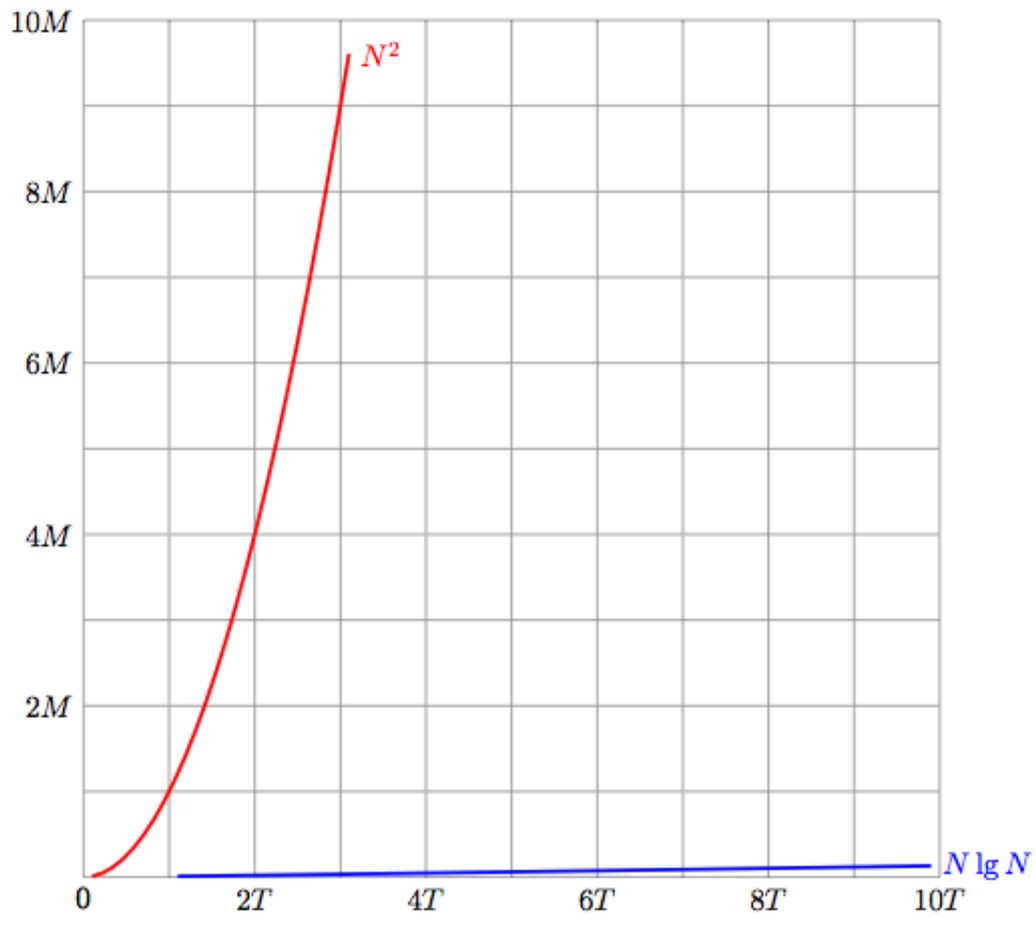


Figure 3.6 N^2 graphed against $N \lg N$ ($T = 10^3$, $M = 10^6$)

Searching with Linear Search

- If we have no information on how the data in the list is organized, we search by walking through the list looking for the item we want.

- On average, linear search of a list of N items requires that we look at N/2 items before we find what we are looking for

Binary Search

- Binary search is our first “mature” algorithm, and is ubiquitous in computing
- Once we have sorted the data, we can do binary search
- Binary search takes log N steps, not N/2 steps

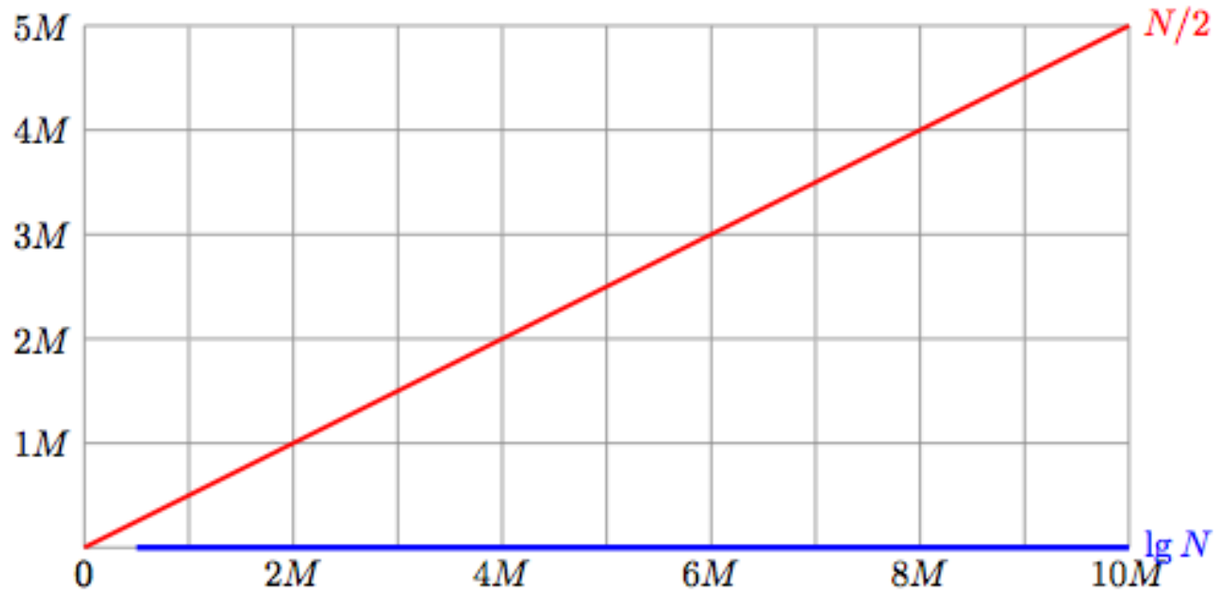


Figure 3.7 $N/2$ graphed against $\lg N$ ($M = 10^6$)

Binary Search

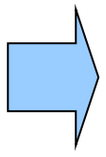
Let's look for Henry *Clay* in a sorted list

- At each step, we compare the midpoint of the list against our search key (*Clay*)
- If the key is smaller than the midpoint, then it lies in the first half of the list (if it is there at all).
- If the key is larger than the midpoint, then it lies in the second half of the list (if it is there at all)
- So with each test we can cut the size of the list in half and then do the test again

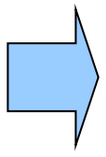
Binary Search

Searching for *Clay*

0	Adams
1	Buchanan
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington



0	**Adams**
1	**Buchanan**
2	**Fillmore**
3	**Harrison**
4	**Jackson**
5	**Jefferson**
6	**Johnson**
7	**Lincoln**
8	**Madison**
9	**Monroe**
10	**Pierce**
11	**Polk**
12	**Taylor**
13	**Tyler**
14	**Van Buren**
15	**Washington**

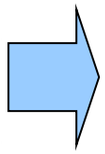


0	**Adams**
1	**Buchanan**
2	**Fillmore**
3	**Harrison**
4	**Jackson**
5	**Jefferson**
6	**Johnson**
7	**Lincoln**
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington

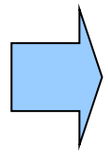
Binary Search

Searching for *Clay* … and not finding it

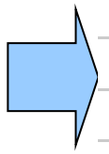
0	**Adams**
1	**Buchanan**
2	**Fillmore**
3	**Harrison**
4	**Jackson**
5	**Jefferson**
6	**Johnson**
7	**Lincoln**
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington



0	**Adams**
1	**Buchanan**
2	**Fillmore**
3	**Harrison**
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington



0	**Adams**
1	**Buchanan**
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington

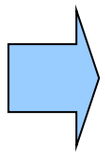


0	Adams
1	Buchanan
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	Monroe
10	Pierce
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington

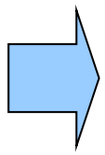
Binary Search

Searching for *Pierce* … and finding it

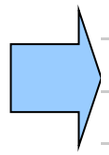
0	**Adams**
1	**Buchanan**
2	**Fillmore**
3	**Harrison**
4	**Jackson**
5	**Jefferson**
6	**Johnson**
7	**Lincoln**
8	**Madison**
9	**Monroe**
10	**Pierce**
11	**Polk**
12	**Taylor**
13	**Tyler**
14	**Van Buren**
15	**Washington**



0	Adams
1	Buchanan
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	**Monroe**
10	**Pierce**
11	**Polk**
12	**Taylor**
13	**Tyler**
14	**Van Buren**
15	**Washington**



0	Adams
1	Buchanan
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	**Monroe**
10	**Pierce**
11	**Polk**
12	Taylor
13	Tyler
14	Van Buren
15	Washington



0	Adams
1	Buchanan
2	Fillmore
3	Harrison
4	Jackson
5	Jefferson
6	Johnson
7	Lincoln
8	Madison
9	Monroe
10	**Pierce**
11	Polk
12	Taylor
13	Tyler
14	Van Buren
15	Washington

Things to Come In This Course

- How to store, manage, manipulate, and retrieve data efficiently
- How to structure data so that “the next item” is also “the next item to be processed”
- How to sort efficiently
- How to search efficiently
- How to store data that changes all the time
- How to structure data so that the structure implies an organization of the data
- How to implement generic data payload elements in a data structure so the data structure program doesn't have to be rewritten for each type of data payload