File Structures

Ch11. Hashing

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

Slides are based on the textbook.

datalab

data science laboratory

Outline

- 11.1 Introduction
- 11.2 A Simple Hashing Algorithm
- 11.3 Hashing Functions and Record Distribution
- 11.5 Collision Resolution by Progressive Overflow
- 11.4 How Much Extra Memory Should Be Used?
- 11.6 Storing More Than One Record per Address: Buckets
- 11.7 Making Deletions
- 11.8 Other Collision Resolution Techniques
- 11.9 Patterns of Record Access

Introduction

- Motivation
  - Hashing is a useful searching technique, can be used for implementing indexes
    - Improves searching time
- Hash vs other index
  - Sequential search : $O(N)$
  - Binary search : $O(log_2 N)$
  - B(B+) Tree index : $O(log_k N)$ where k records in an index node
  - Hashing : $O(1)$

What is hashing

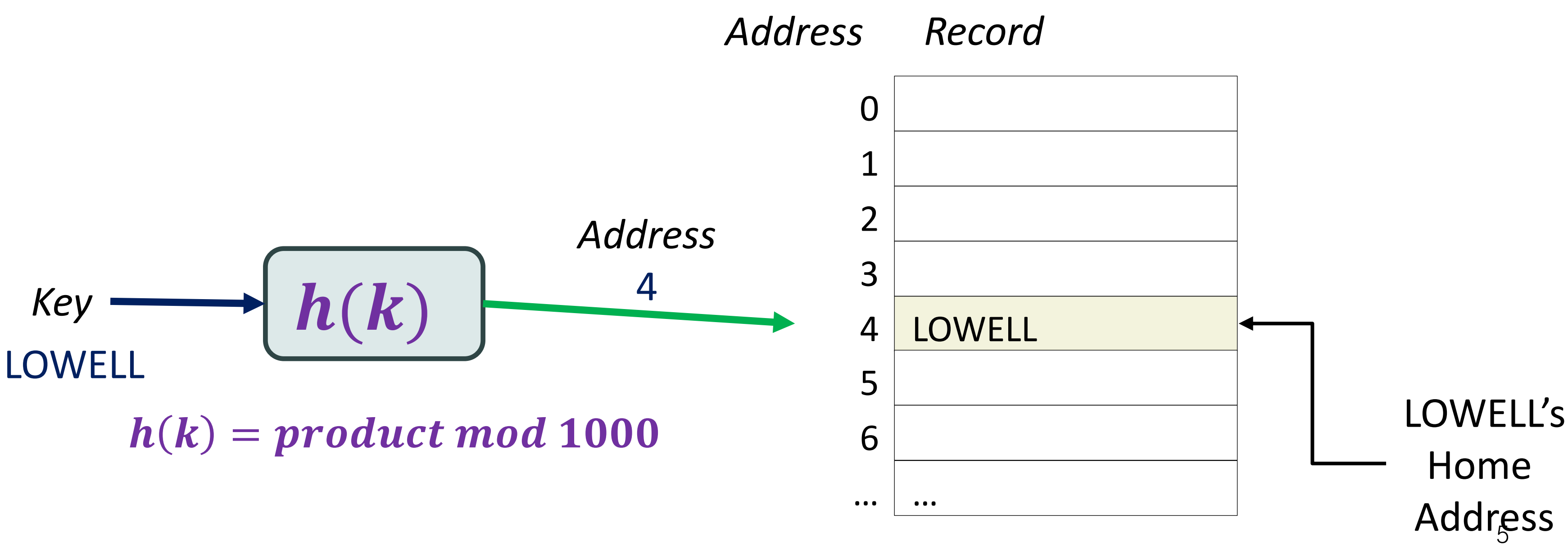
- Idea
  - To discover the location of a key by simply examing the key → need to design a hash function
- Hash function : $h(k)$
  - Transforms a key *k* into an address
  - An address space is chosen beforehand
  - Example
    - U is a set of all possible keys, h is hashing function

$$h: U \rightarrow \{0,1,\dots,999\}$$

A Simple Hashing Scheme

Name	ASCII code for First Two Letters	Product	Home Address
BALL	66 65	66*65 = 4,290	290
LOWELL	76 96	76*96 = 6,004	004
TREE	84 82	84*82 = 6,888	888

$$h(LOWELL) = 004$$



$$h(k) = product\ mod\ 1000$$

Hashing differs from indexing

- With hashing
  - There is no obvious connection between the key and address
  - **hashing is referred to as randomizing**
- Two different keys may be translated to the same address
  - Two records may be sent to the same place in the file

→ **collision**

Collision (1/2)

- Collision
  - Situation in which a record is hashed to an address that does not have sufficient room to store the record
  - Perfect hashing : impossible!
  - Different key, same hash value (Different record, same address)

$$\mathrm{h(LOWELL)} = 4$$

$$\mathrm{h(OLIVER)} = 76 * 79 \bmod 1000 = 6004 \bmod 1000 = 4$$

Collision (2/2)

- Solutions
  - Spread out the records
    - By choosing a good hash function
  - Use extra memory
    - Increase the size of the address space
    - Ex: reserve 5,000 available addresses rather than 1,000
  - Put more than one record at a single address
    - Use of buckets

A Simple Hashing Algorithm (1/4)

- Step 1. Represent the key in numerical form
  - If the key is a string : take the ASCII code
  - If the key is a number : nothing to be done
  - e.g. **LOWELL**

L	O	W	E	L	L	6 blanks					
76	79	87	69	76	76	32	32	32	32	32	32

A Simple Hashing Algorithm (2/4)

- Step 2 . Fold and Add
  - Fold
    - 76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32
  - Add parts into one integer
    - Suppose we use 15bit integer expression, 32767 is limit
      - 7679 + 8769 + 7676 + 3232 + 3232 + 3232 = 33820 > 32767 (overflow!)
      - Make sure that each successive sum is less than 32768
    - Largest addend : 9090 (‘ZZ’)
      - 32767-9090 = 23677

→ Choose largest allowable result : 19937 (prime number)

A Simple Hashing Algorithm (3/4)

- Step 2 . Fold and Add (cont'd)
  - Ensure no intermediate sum exceeds using 'mod'
    - (7679 + 8769) mod 19937 = 16448
    - (16448 + 7676) mod 19937 = 4187
    - (4187 + 3232) mod 19937 = 7419
    - (7419 + 3232) mod 19937 = 10651
    - (10651 + 3232) mod 19937 = 13883

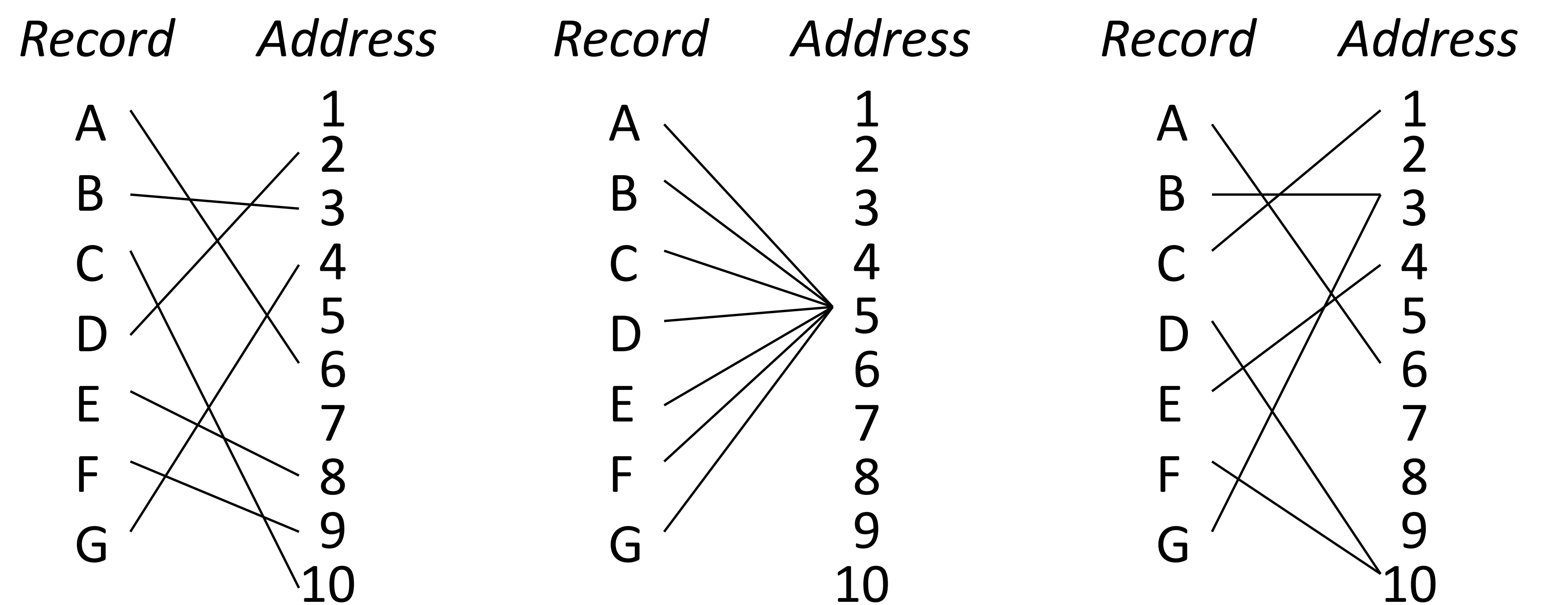
A Simple Hashing Algorithm (4/4)

- Step 3. Divide by size of the address space
  - a = s mod n
    - a : home address
    - s : the sum produced in step 2
    - n : the number of addresses in the file
  - Assume that we use the 100 addresses 0-99 for our file
    - a = 13883 mod 100 = 83

Hashing Functions and Record Distributions

- Distributing records among address

Best: Uniform (no synonyms) | **Worst** (all synonyms) | **Acceptable: Random** (a few synonyms)



Uniform distributions are extremely rare.
Random distributions are acceptable and more easily obtainable.

Some other hashing methods

- Better-than-random
  - Examine keys for a pattern
  - Fold parts of the key
  - Divide the key by a prime number
- When the better-than-random methods do not work ----- *randomize!*
  - Square the key and take the middle
    - Key 453 → $453^2 = 205\ 209$ → take 52
  - Radix transformation
    - 453(base 10) → 382 (base 11)

Outline

- 11.1 Introduction
- 11.2 A Simple Hashing Algorithm
- 11.3 Hashing Functions and Record Distribution
- 11.5 Collision Resolution by Progressive Overflow
- 11.4 How Much Extra Memory Should Be Used?
- 11.6 Storing More Than One Record per Address: Buckets
- 11.7 Making Deletions
- 11.8 Other Collision Resolution Techniques
- 11.9 Patterns of Record Access

Progressive Overflow (1/6)

- Collision Resolution by Progressive overflow (= **linear probing**)

- Insert a new record with key k
  - 1. Go to the home address of k: h(k)
  - 2. If free, place the key there
  - 3. Otherwise, try the next position until an empty one is found
  - 4. If no more next space - wrapping around

Progressive Overflow (2/6)

- Insertion example

Key k	Home address: h(k)
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

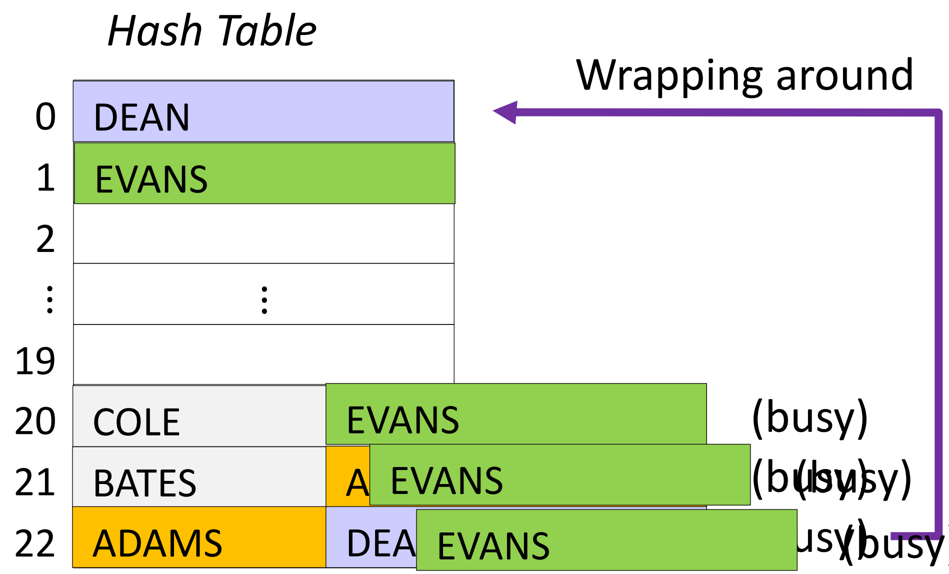


Table Size =23

Progressive Overflow (3/6)

- Searching a record with key k
  - 1. Go to the home address of k : h(k)
  - 2. If k is in home address, done
  - 3. Otherwise, try the next position until
    - key is found or
    - empty space is found or
    - home address is reached
    - (in the last 2 cases : the key is not found)

- Worst case
  - When the record does not exist and the file is full

Progressive Overflow (4/6)

- Searching example
  - For “EVANS” probes places
    - 20, 21, 22, 0, 1 → finding at record 1
  - For “KWON”, if h(KWON)=22, probes
    - 22, 0, 1, 2 (empty) → KWON not exists
  - For “SMITH”, if h(SMITH)=19 probes
    - 19 → SMITH not in the table

Hash Table

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Table Size =23

Progressive Overflow (5/6)

- Search Length
  - Number of accesses required to retrieve a record

$$\boldsymbol{Avg.\ search\ length\ = \frac{sum\ of\ search\ lenghts}{num\ of\ records}}$$

  - Example

Hash Table

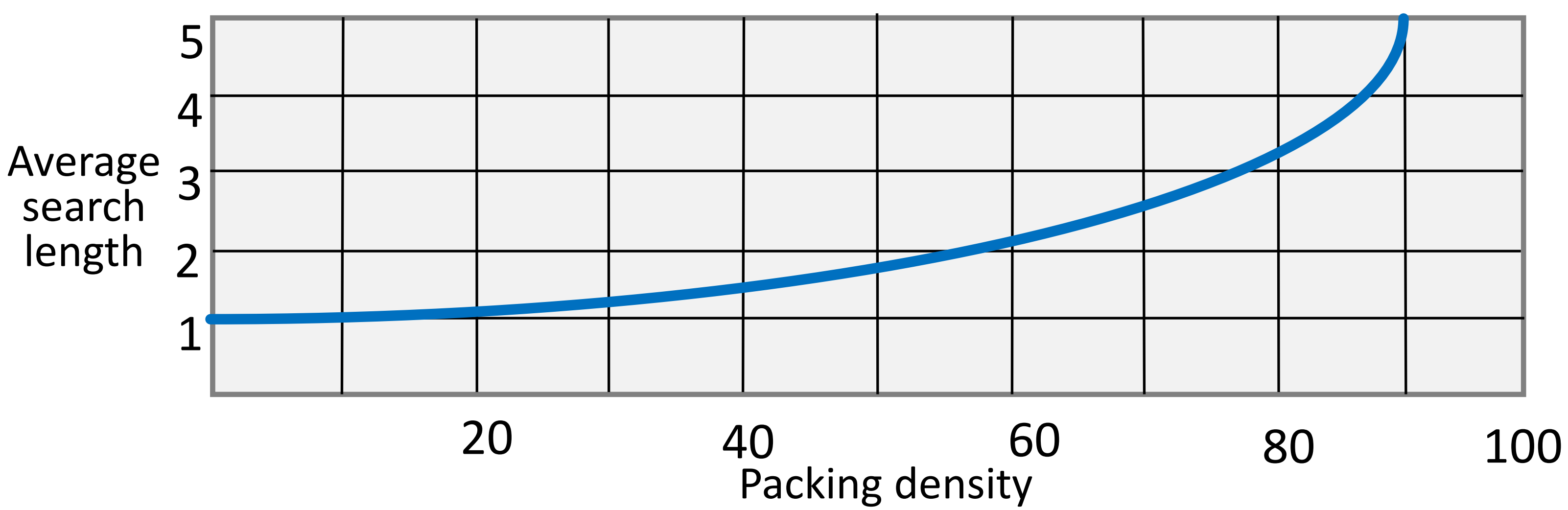
0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Key k	Search Length
COLE	1
BATES	1
ADAM	2
DEAN	2
EVANS	5

$$\boldsymbol{Avg.\ search\ length\ = \frac{11}{5} = 2.2}$$

Progressive Overflow (6/6)

- Average search length
  - With perfect hashing function
    - average search length = 1
  - <u>no greater than 2.0</u>
    - are generally considered acceptable



Outline

- 11.1 Introduction
- 11.2 A Simple Hashing Algorithm
- 11.3 Hashing Functions and Record Distribution
- 11.5 Collision Resolution by Progressive Overflow
- 11.4 How Much Extra Memory Should Be Used?
- 11.6 Storing More Than One Record per Address: Buckets
- 11.7 Making Deletions
- 11.8 Other Collision Resolution Techniques
- 11.9 Patterns of Record Access

Predicting record distribution (1/2)

- Assume a random distribution for the hash function
  - N: number of available addresses
  - r: number of records to be stored
- p(x)
  - the probability that a given address will have x records assigned to it
    - $p(x) = \frac{r!}{(r-x)!x!}\left[1 - \frac{1}{N}\right]^{r-x}\left[\frac{1}{N}\right]^{x}$
    - For N and r large enough, then, (poisson function)

$$p(x) \sim \frac{(r/N)^{x} e^{-(r/N)}}{x!}$$

Predicting record distribution (2/2)

$$p(x) \sim \frac{(r/N)^x e^{-(r/N)}}{x!}$$

- Example: N=1,000 r=1,000
  - $p(0) \sim \frac{(1)^0 e^{-1}}{0!} = 0.368$ $p(1) \sim \frac{(1)^1 e^{-1}}{1!} = 0.368$
  - $p(2) \sim \frac{(1)^2 e^{-1}}{2!} = 0.184$ $p(3) \sim \frac{(1)^3 e^{-1}}{3!} = 0.061$
- For N addresses
  - The expected number of addresses with x records:

$$\boldsymbol{N \cdot p(x)}$$

Reducing collision by using more addresses

- How to reduce collisions by increasing the number of available addresses
  - The more records are packed, the more likely a collision will occur

- Definition
  - Packing Density = $\dfrac{\#\ of\ records}{\#\ of\ spaces} = \dfrac{r}{N}$
  - Example: 500 records to be spread over 1000 addrs.
    - Packing density= 500/1000 = 0.5 = 50 %

Predicting collisions (1/4)

$$p(x) \sim \frac{(r/N)^x e^{-(r/N)}}{x!}$$

- 1. How many addresses go unused?
  - More precisely, What is the expected number of address with no key mapped to it?

$$\boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{0}) = 1000 \cdot \frac{(0.5)^0 e^{-0.5}}{0!} = 1000 \cdot 0.607 = 607$$

- 2. How many addresses have no synonyms?
  - More precisely: What is the expected number of address with only one key mapped to it?

$$\boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{1}) = 1000 \cdot \frac{(0.5)^1 e^{-0.5}}{1!} = 1000 \cdot 0.303 = 303$$

Predicting collisions (2/4)

- 3. How many addrs. Contain 2 or more synonyms?
  - What is the expected number of addrs. with two or more keys mapped to it?

$$\boldsymbol{N} \cdot (\boldsymbol{p}(\mathbf{2}) + \boldsymbol{p}(\mathbf{3}) + \cdots) = \boldsymbol{N}(\mathbf{1} - (\boldsymbol{p}(\mathbf{0}) + \boldsymbol{p}(\mathbf{1})) = 1000 = 1000 \cdot 0.09 = 90$$

Predicting collisions (3/4)

- 4. Assuming that only one record can be assigned to an address,
  - how many overflow records are expected ?
  - Two approaches
    - 1. Naive

$$\mathbf{1} \cdot \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{2}) + \mathbf{2} \cdot \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{3}) + \mathbf{3} \cdot \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{4}) + \cdots = 107$$

    - 2. Expected # of overflow pages = (# records) – (expected # of non overflow records)

$$\begin{aligned} &\boldsymbol{r} - (\boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{1}) + \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{2}) + \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{3}) + \cdots) \\ &= \boldsymbol{r} - \boldsymbol{N}\big(\mathbf{1} - \boldsymbol{p}(\mathbf{0})\big) = \boldsymbol{N} \cdot \boldsymbol{p}(\mathbf{0}) - (\boldsymbol{N} - \boldsymbol{R}) \\ &= \mathbf{607} - \mathbf{500} = \mathbf{107} \end{aligned}$$

Predicting collisions (4/4)

- 5. What is the expected percentage of overflow records?

$$107/500 = 0.214 = 21.4\%$$

Packing Density(%)	% overflow records (synonyms as % of recs.)
10	4.8
20	9.4
30	13.6
40	17.6
50	21.4
60	24.8
70	28.1
80	31.2
90	34.1
100	36.8

Outline

- 11.1 Introduction
- 11.2 A Simple Hashing Algorithm
- 11.3 Hashing Functions and Record Distribution
- 11.5 Collision Resolution by Progressive Overflow
- 11.4 How Much Extra Memory Should Be Used?
- 11.6 Storing More Than One Record per Address: Buckets
- 11.7 Making Deletions
- 11.8 Other Collision Resolution Techniques
- 11.9 Patterns of Record Access

Hashing with buckets (1/2)

- This is a variation of hashed files in which more than one record/key is stored per hash address.

- bucket
  - A block of records corresponding to one address in the hash table

- The hash function gives the Bucket Address.

Hashing with buckets (2/2)

- Example: for a bucket holding 3 records, insert the following keys

Key k	Home Address
Green	30
Hall	30
Jenks	32
King	33
Land	33
Marx	33
Nutt	33

Bucket Address	*Bucket Contents*		
0			
1			
…			
30	Green…	Hall…	
31			
32	Jenks…		
33	King…	Land…	Marks…
…	…	…	…

(Nutt… is an overflow record)

Effects of Buckets on Performance (1/2)

- We should slightly change some formulas:

$$Packing\ Density = \frac{\#\ of\ records}{\#\ of\ spaces} = \frac{r}{b \cdot N}$$

- compare the following two alternatives:
  - 1. Storing 750 data records into a hashed le with 1,000 addresses, each holding 1 record.
    - PD= 750/1000=0.75
  - 2. Storing 750 data records into a hashed le with 500 bucket addresses, each bucket holding 2 records.
    - PD= 750/(500*2) = 0.75

Effects of Buckets on Performance (2/2)

- Estimating the probabilities as defined before

	p(0)	p(1)	p(2)	p(3)	p(4)
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.5 (b=2)	0.223	0.335	0.251	0.126	0.047

- b=1
  - Number of overflow records= $r - N(1 - p(0)) = 222$
- B=2
  - Number of overflow records: 141

$$
\begin{aligned}
& N[1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \cdots] \\
& = r - N \cdot p(1) - 2 \cdot N[p(2) + 2 \cdot p(3) + \cdots] \\
& = r - N[p(1) + 2[1 - p(0) - p(1)] \\
& = r - N\big(2 - 2p(0) - p(1)\big) \\
& = 750 - 500(2 - 2 * 0.112 - 0.224) = 140.5 \approx 141
\end{aligned}
$$

Bucket Implementation (1/2)

Collision counter =< bucket size

0	///	///	////	////	////

An empty bucket

2	JONES	ARNSWORTH	////	////	////

Two entries

5	JONES	ARNSWORTH	STOCKTON	BRICE	TROOP

Two entries

Bucket Implementation (2/2)

- Initializing and Loading
  - Creating empty space
  - Use hash values and find the bucket to store
  - If the home bucket is full, continue to look at successive buckets

- Problems when
  - No empty space exists
  - Duplicate keys occur

Outline

- 11.1 Introduction
- 11.2 A Simple Hashing Algorithm
- 11.3 Hashing Functions and Record Distribution
- 11.5 Collision Resolution by Progressive Overflow
- 11.4 How Much Extra Memory Should Be Used?
- 11.6 Storing More Than One Record per Address: Buckets
- 11.7 Making Deletions
- 11.8 Other Collision Resolution Techniques
- 11.9 Patterns of Record Access

Making Deletions (1/3)

- Deletions in a hashed file have to be made with care

Key k	Home address: h(k)
ADAMS	5
JONES	6
MORRIS	6
SMITH	5

Hash Table

⋮	⋮
4	///////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Making Deletions (2/3)

- Delete ‘Morris’
  - If `MORRIS' is simply erased, a search for `SMITH' would be unsuccessful

Key k	Home address: h(k)
ADAMS	5
JONES	6
MORRIS	6
SMITH	5

Hash Table

⋮	⋮	
4	///////	← empty slot
5	ADAMS	
6	JONES	
7	///////	← empty slot
8	SMITH	
⋮	⋮	

  - Search for `SMITH’
    - would go to home address (position 5) and when reached 7 it would conclude `SMITH' is not in the file

Making Deletions (3/3)

- IDEA: use TOMBSTONES
  - replace deleted records with a marker indicating that a record once lived there
  - A search must continue when it finds a tombstone
    - but can stop whenever an empty slot is found.
  - Insertions should be modified to work with tombstones

Hash Table

⋮	⋮	
4	///////	← empty slot
5	ADAMS	
6	JONES	
7	#########	← tombstone
8	SMITH	
⋮	⋮	

Effects of Deletions and Additions on Performance

- The presence of too many tombstones increases search length.
- Solutions to the problem of deteriorating average search lengths:
  - 1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address.
  - 2. Complete reorganization: re-hashing.
  - 3. Use a different type of collision resolution technique

Other Collision Resolution Techniques

- Double hashing
  - avoid clustering with a second hash function for overflow records
- Chained progressive overflow
  - each home address contains a pointer to the record with the same address
- Chaining with a separate overflow area
  - move all overflow records to a separate overflow area
- Scatter tables
  - Hash file contains only pointers to records (like indexing)

Patterns of Record Access

- Pareto Principle (80/20 Rule of Thumb)
  - 80 % of the accesses are performed on 20 % of the records!
    - The concepts of “the Vital Few and the Trivial Many”
    - 20 % of the fisherman catch 80 % of the fish
    - 20 % of the burglars steal 80 % of the loot
- If we know the patterns of record access ahead, we can do many intelligent and effective things!
  - Sometimes we can know or guess the access patterns
  - Very useful hints for file syetems or DBMSs
- Intelligent placement of records
  - fast accesses
  - less collisions

Q&A