Module 10

Storing Relations in Files

Objectives

- Learn about issues affecting data retrieval when relations are stored on disk
- Learn how hashing and indexing structures are created to speed up data retrieval

95.305

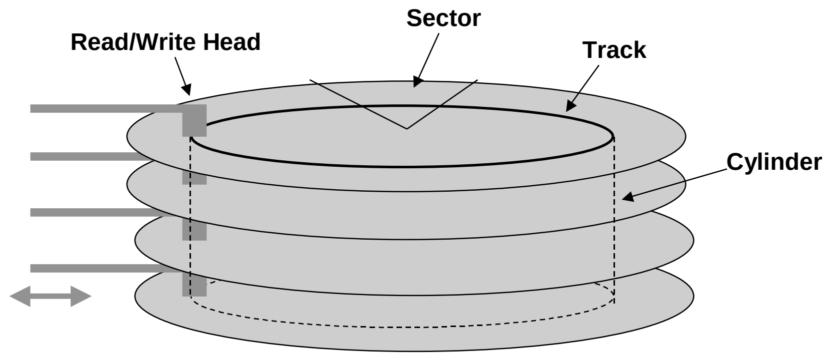
Topics

- **Disk Storage and Buffering**
- **Blocks and Data Retrieval**
- **Hashing**
- **Indexing and B trees**

References

- **Elmasri & Navathe Chapters 4, 5**

Disk Storage



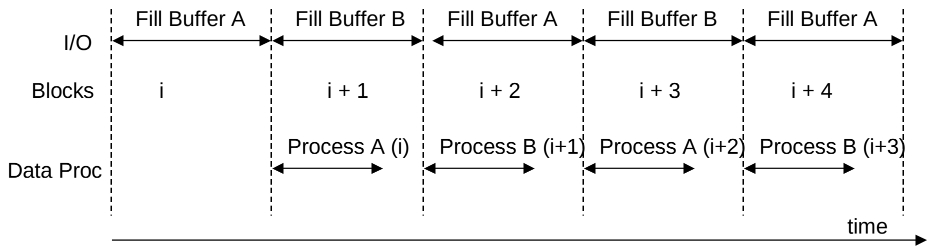
- Data access time =
  Seek time + Latency Time + Block Transfer Time
- Typically 15-60 msec, but block transfer time is only about 1-2 msec
- Times are slow compared to CPU so must optimize

Disk Blocks

- Disks are organized into blocks during formatting
- Typical block size 512 - 4096 bytes
- Disks are random access devices in that blocks can be access randomly and there contents read sequentially
- Reading and Writing contiguous blocks is quite fast (1-2 msec) but non contiguous block access is slow requiring seeking (moving heads) and latency (waiting for sector to spin under head).

- Any strategy which minimizes seeking and latency is helpful for database performance

Double Buffering

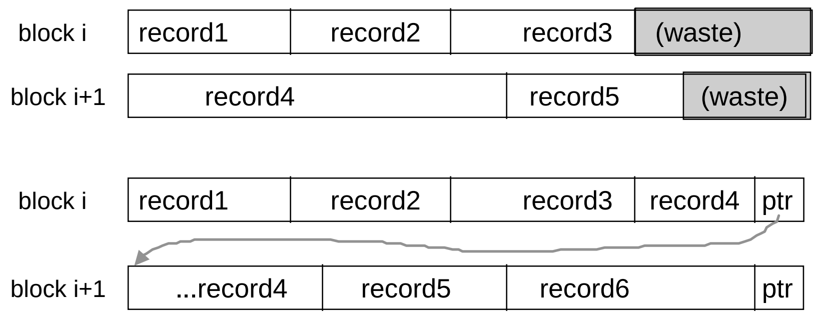


- **Double buffering allows blocks to be read and processed contiguously**
- **Processor must be faster than disk (usually is)**

Records, Files and Blocks

- **Data, such a tuples are usually stored as a <u>record</u>**
- **A <u>File</u> is a collection of records**
- **Can store one relation per file, or many relations in a file**
- **File records are written to disk <u>blocks</u>**
- **Blocks are the unit of transfer from Disk to Memory**
- **(Records can be shorter or longer than blocks)**

Spanned and Unspanned Record Organization



- **Unspanned vs. Spanned blocks**

Organizing File Blocks

- **option 1) Store file records in contiguous file blocks**
  - **-makes sequential reading easy**
  - **-makes updating the file difficult**
- **option 2) Store blocks with a pointer to where the next block resides on disk**
  - **-makes sequential reading difficult**
  - **-makes updating the file easy**
- **option 3) compromise -store file as clusters of contiguous blocks**

Processing Files

- **Imagine processing a query which looks for an employee whose name is 'John Smith'.**
- **If nothing is known structurally about the file, and it's organization into blocks, we have to do a linear search through the file blocks**
- **It is extremely helpful if we know in which block a piece of data resides**
- **(Possibly the single most important organizational factor affecting performance)**

Processing a Query

- **e.g.** Select name, address
  From employee
  Where SSN = 321321321
- **To process this query we must search the employee table (file) for the record with** SSN = 3212321321
- **Requires reading blocks into main memory and checking the** SSN **field of each record**
- **If** SSN **is the key we might expect many searches to be based on the key so it would be helpful if we can determine the block on disk based on the key**
- **File organization can affect this greatly**

File Organization: Heap File (Pile)

- Records are stored in the order they are inserted
- Makes insertion very easy
  - -read last block of file
  - -modify block
  - -write block back to disk
- Retrieval is difficult
  - -must search linearly through block looking for record (on average 1/2 the blocks)
- Deletion leads to fragmentation
  - -search (linearly) for record to be deleted
  - -read block to memory
  - -delete record from block (leaving wasted space)
  - -write block back to disk
  - -periodically clean up fragmentation

Ordered or Sorted Files

- File records are stored in the order of some record field (attribute)
- The primary key is possibly a good attribute to pick
- Retrieving the records in the order of the order-key is easy
- Retrieving a random record based on the value of the order key is also faster (logarithmic rather than linear time)

Binary Search of Ordered File

- **Assumptions**
- **File is stored as b blocks 1, 2, ... ,b**
- **Records are ordered by ascending value of their ordering-key**
- **Disk addresses of file blocks are available in file header**

...Binary Search of Ordered File

```
retrieve( K )
l := 1;  u := b;
while (u >= l) do {
   i := (l+u) // 2;
  read block i of the file into buffer
  if (K < ordering_key value of first record in block) {
    u := i-1;
    else ( if K > ordering_key field value of the last record in block) {
                    l := i + 1;
                   else if ( record with ordering_key = K is in buffer) {
                              return (record with ordering_key = K)
                              else return( NOT_FOUND )
                              }
           }
   }
 }
```

...Binary Search of Ordered File (Recursive)

```
retrieve( K ) {
return ( retrieve(1, b, K)  ) }

retrieve(min, max, K) {
   i := (min + max) // 2;
   read block i of file into buffer;
   if (K < ordering_key of first record in block)
         {return( retrieve(min, i, K) ) };
   if (K > ordering_key of last record in block)
         {return( retrieve(i, max, K) ) };
   return( linearlyRetrieve(i, K) );
}
```


Ordered Files -other operations

- **Ordered retrieval, or random search, based on non-ordering-key field requires sorting file, or linear search**
- **Insertions and deletions are expensive because blocks must retain their physical ordering**
- **Ordered files are rarely used in databases by themselves, they typically have an additional access path based on indexing**

Organization based on Hashing

- Idea: compute the location of the data based on the values of some of the attributes
- Compute a hash function on the specified hash attribute
- Hash function returns the address of the block where the data would reside (if it were present)
- The block identified by the hash function can be read into main memory and searched linearly for the data

Hashing

- Let K = set of all possible search key values
- Let {K1, ... ,Kn} be the search key values in the database
- Let B be a set of buckets
- A hash function h() maps key values to buckets

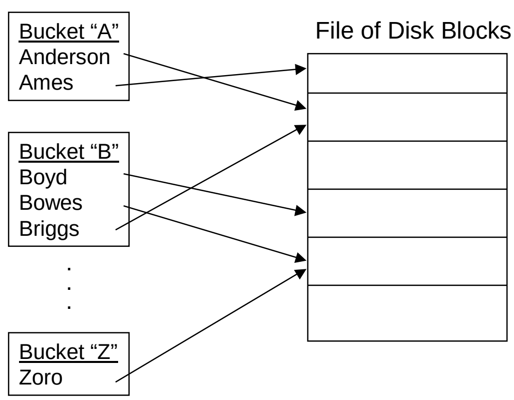
h(Ki) is some bucket in B

Properties of Good Hash Functions

- Bad: all actual keys map to the <u>same</u> bucket
- Ideal: all actual keys map to a different bucket (but wasteful)
- Good: each bucket has the same number of key values (uniform distribution)
- Good: on average each bucket will have nearly the same distribution (randomly uniform)

example simple Hash Function

- Consider hashing employee names to bucket based on simple hash function h(name)
- h(name) = position in alphabet of first letter in name



example simple Hash Function

- **Consider hashing employee names to bucket based on simple hash function** h(name)
- h(name) **= position in alphabet of first letter in name**

- **Problems with this hash function**

  number of buckets is fixed at 26

  some letters are more popular than other so it's probably not very uniform

popular Hash Function

- **For** b **buckets**
- **Let** C1, ... ,Cm **be the binary representation of the m characters of the key value (e.g. characters of employee's name)**
- h(name) = (C1 + C2 + ... + Cm) mod b

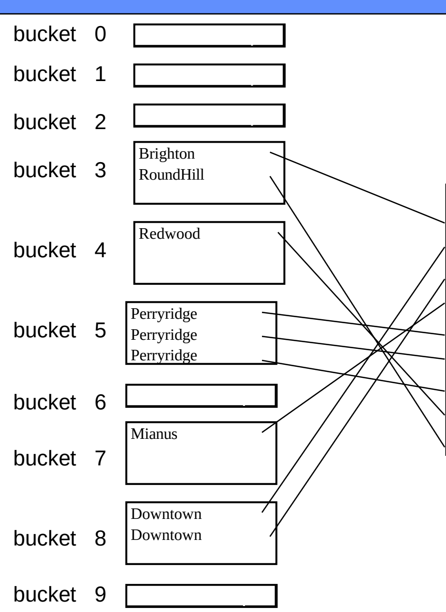
Example (Static Hashing)

- **Show the hash buckets if the branch name are hashed into 10 buckets based on the hash function**

h(branch) = (bin(C1) + bin(C2) + ... + bin(Cm)) mod 10

Branch	Account	Customer	Balance
Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Static Hashing



Branch	Account	Customer	Balance
Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Problem with Static Hashing

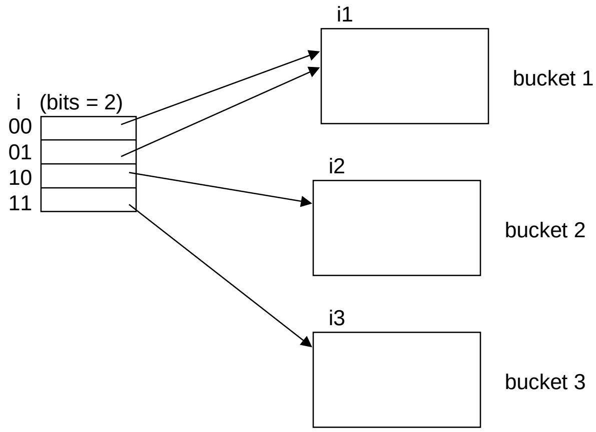
- Previous example was static hashing
- Problem: the number of buckets must be decided ahead of time
- Selecting an appropriate number of buckets is difficult if we must anticipate how data will grow and shrink over time

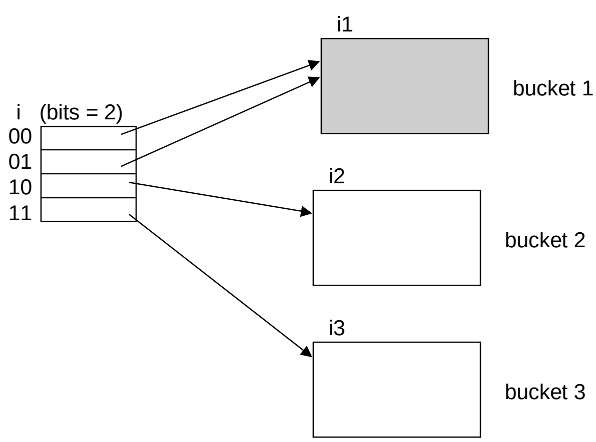
- Solution: use extendible hashing which grows and shrinks the number of buckets with database size

Extendible Hashing

- Idea: let h() map key values onto a 32-bit binary number
- e.g. h("Brighton") = 0010 1101 1111 1011 0010 ...

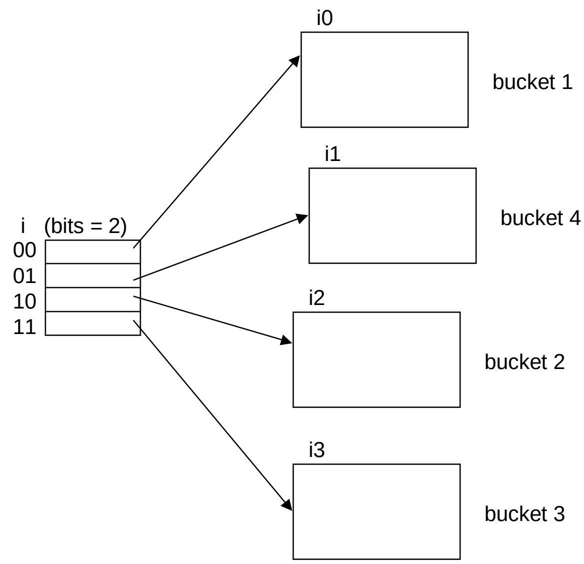
- If we use only the first two bits of the hash value we can distinguish among 4 buckets
- As these buckets become full, use another bit position -this doubles the number of buckets
- As buckets become empty, throw them away and use fewer bits of the hash value

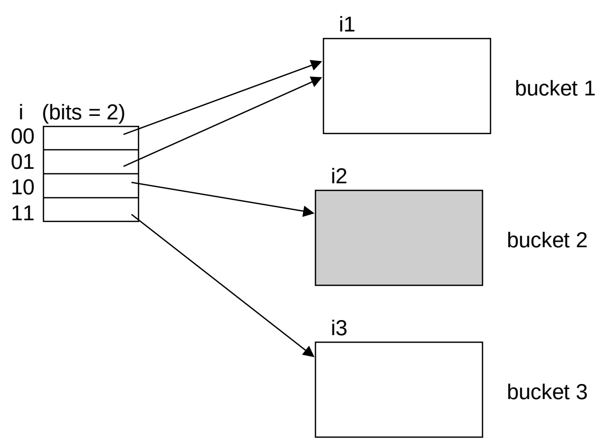




- **Suppose bucket 1 becomes full**

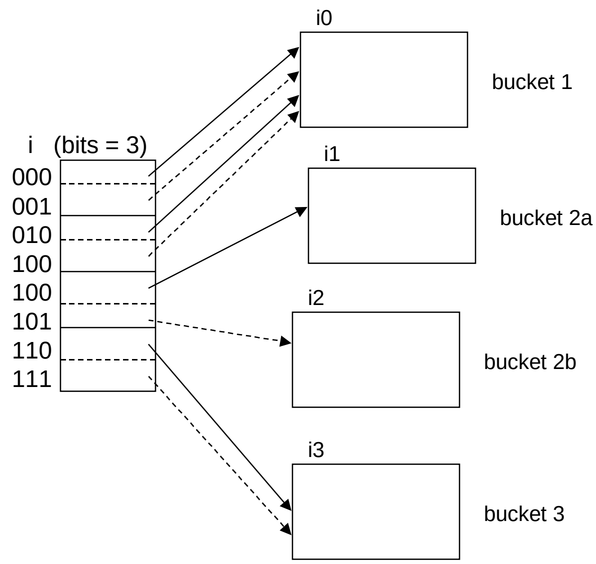
- **Create a new bucket 4**
- **make half the bucket 1 pointers point to bucket 4**
- **rehash all of bucket 1**
- **split buckets again if necessary**





- **Suppose instead bucket 2 becomes full**

- Split bucket 2 into 2a, 2b
- bucket 2 has no duplicate pointer so increase i to 3 bits
- create new pointer to bucket 2b
- rehash bucket 2
- duplicate pointers to other buckets



Example (Extendible Hashing)

- Create an extendible hashing structure for this branch table by starting with an empty file and inserting the records one at a time
- Use buckets which can hold two records

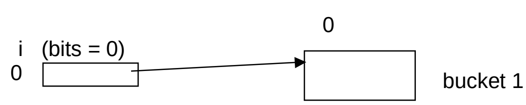
Branch	Account	Customer	Balance
Brighton	217	Green	750
Downtown	101	Johnson	500
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Redwood	222	Lindsay	700
Round Hill	305	Turner	350
Clearview	117	Throggs	295

Example (Extendible Hashing)

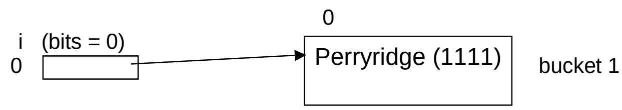
- **hash function for branch name**
- **h(branch)**

Branch	**h(branch)**
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Clearview	1101 0101 1101 1110 0100 0110 1001 0011
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1000 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	1011 0101 1010 0110 1100 1001 1110 1011
RoundHill	0101 1000 0011 1111 1001 1100 0000 0001

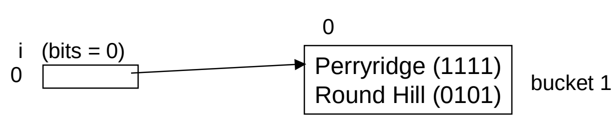
- **start with empty file**



- **insert (“Perryridge”, 102, “Hayes”, 400)**
- **h(“Perryridge”) = 1111 ...**

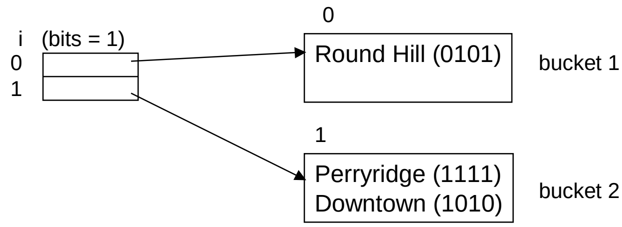


- **insert (“RoundHill”, 305, “Turner”, 350)**
- **h(“RoundHill” = 0101 ...**



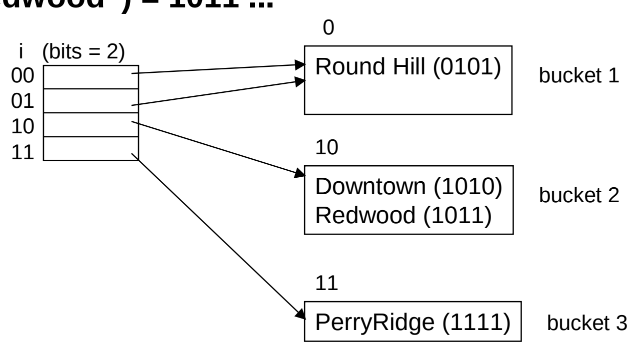
- **insert (“Downtown”, 101, “Johnson”, 500)**
- **h(“Downtown”) = 1010 ...**

- **insert ("Downtown", 101, "Johnson", 500)**
- **h("Downtown") = 1010 ...**

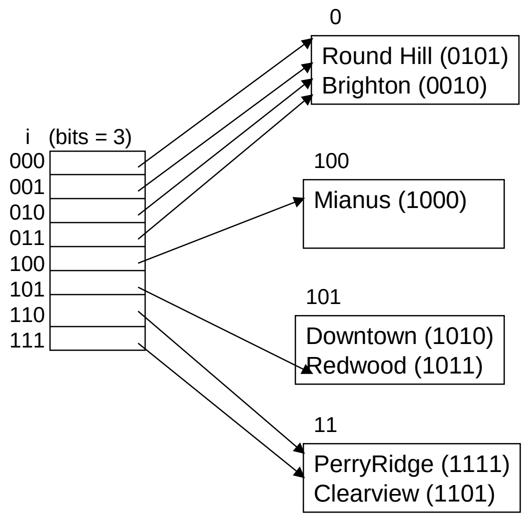


- **insert ("Redwood", 222, "Lindsay", 700)**
- **h("Redwood") = 1011 ...**

- **insert ("Redwood", 222, "Lindsay", 700)**
- **h("Redwood") = 1011 ...**



- **insert ("Mianus", 215, "Smith", 700)**
- **insert ("Brighton", 217, "Green", 750)**
- **insert ("Clearview", 117, "Throggs" 295)**



Indexing

- **An index for a database is like a card catalog for a library**
- **One card catalog orders books by author Another orders books by subject**
- **Creates the illusion that the books are stored certain order, even though they are physically stored in another**

- Imagine a query like
  Select book_title
  From books
  Where author = “Dickens”
- Solution 1: start at one end of the library and start walking, examining every book until you find those by “Dickens”
- Solution 2: Go to the card catalog, look up “Dickens” and write down the locations (call numbers) of the books by “Dickens”, finally go to the locations to retrieve the books

- Each index is associated with a particular <u>search key</u> (which is not necessarily a candidate key)
- If the file is sequentially ordered the index whose search key specifies the same order is called the <u>primary index</u>
- Index-Sequential files:
  files that have a sequential order and a primary index (good for sequential and random access)

Dense Index

Brighton
Downtown
Mianus
Perryridge
Redwood
Round Hill

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

- **Index for every Search Key value**

Sparse Index

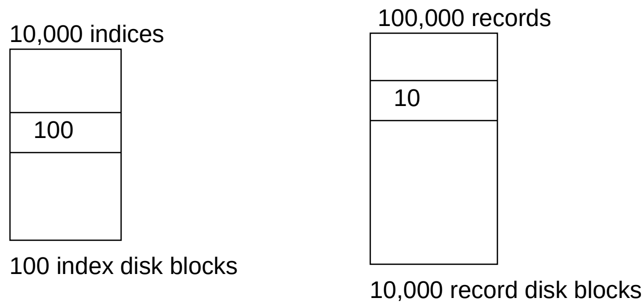
Brighton
Mianus
Redwood

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

- **Index for some Search Key values**
- **e.g. one index per disk block**

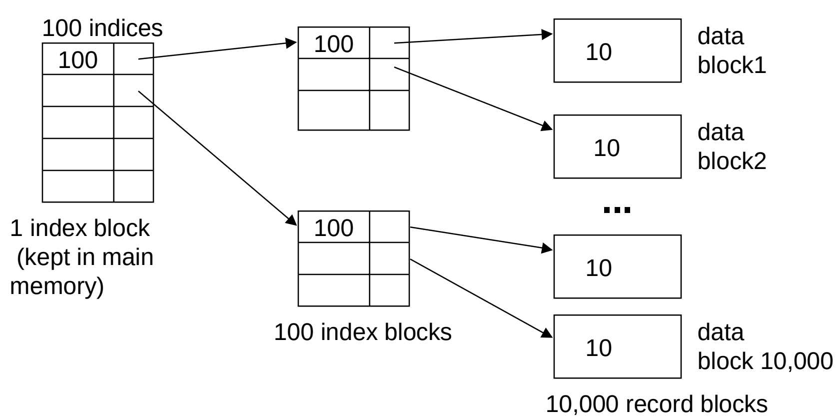
Large Indices

- e.g. 100,000 records, 10 per disk block
- Consider sparse indexing, 1 index per disk block
  - 10,000 indices, 100 indices per disk block



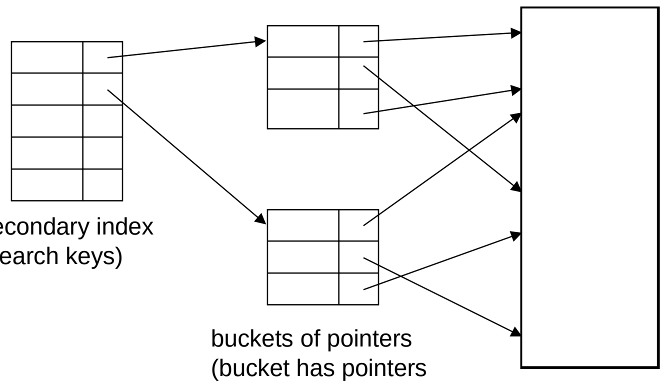
- To find index block $\log_2(100) = 7$ block reads
- To find data from index = 1 block read
- 8 block reads total

Large Indices: Sparse index on the index



- To find correct index to data: 1 block read
  To access data from index: 1 block read
- total 2 block reads
- updates may now be trickier

Pseudo-sequential Ordering with Secondary Index



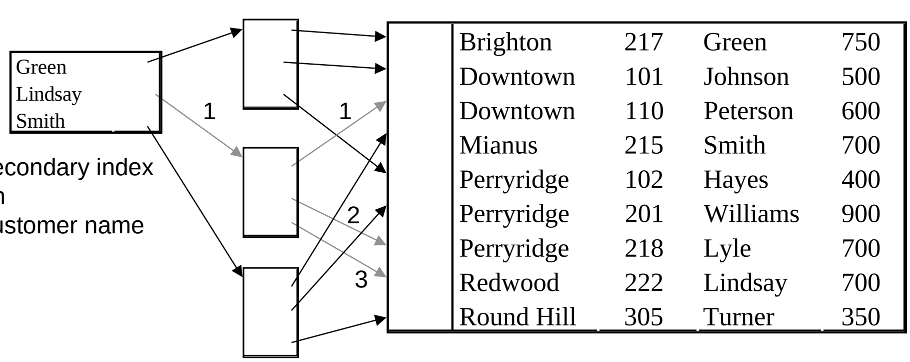
secondary index
(search keys)

buckets of pointers
(bucket has pointers
to all records with same
search key values, or
proper range in sparse
key case)

record blocks

- **note: sequential access on primary search key is easy since records are stored in that order**

Pseudo-sequential Ordering with Secondary Index



Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

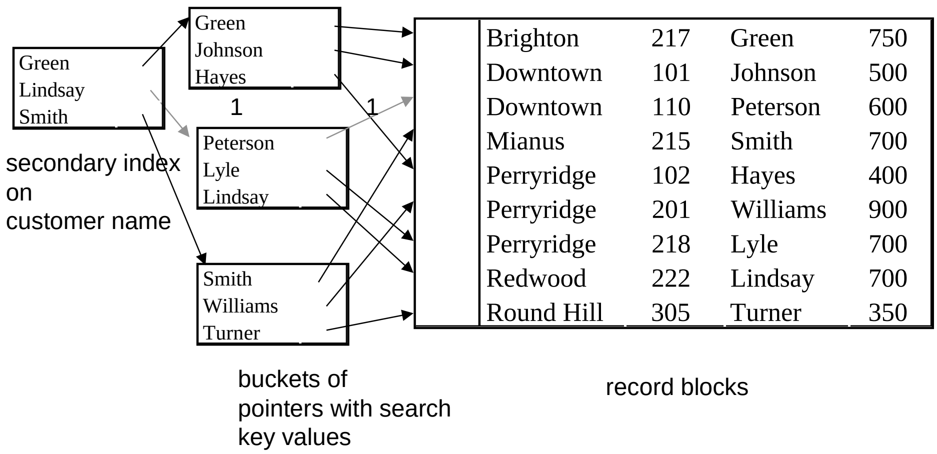
secondary index
on
customer name

buckets of
pointers

record blocks

- **Finding records for "Peterson" requires 1 + 3 block reads**

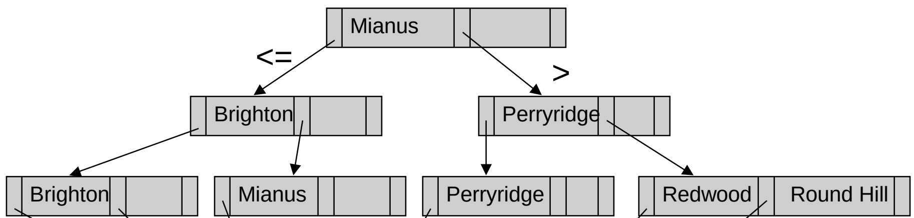
...Pseudo-sequential Ordering with Secondary Index



- **Finding "Peterson" records requires only 1 + 1 block reads if search key value is stored with the pointer in the buckets**

- **Index sequential file organization degrades as the files grow (and has frequent insertion, deletions)**
- **Alternative is to use a multi-level search tree to maintain indices and to try and keep the tree balanced**
- **B+ trees are widely used for such file structures**

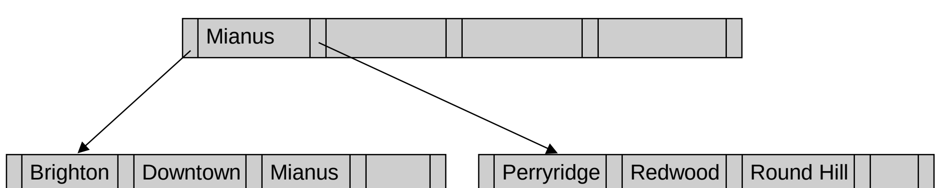
B+ Tree (n = 3 pointers)



Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

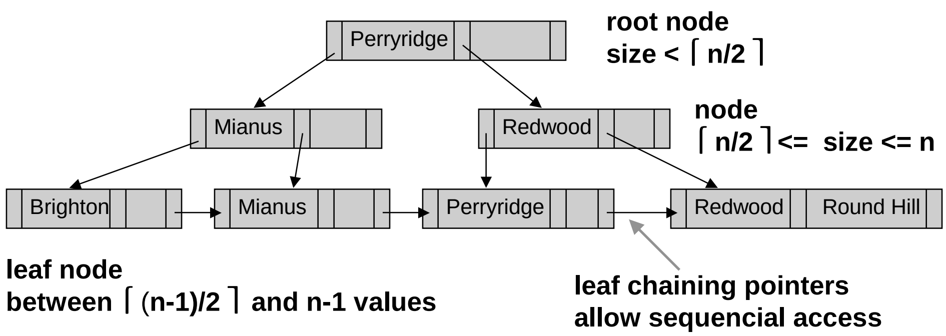
For secondary keys these would point to buckets of pointers to actual records

B+ Tree (n = 5 pointers)



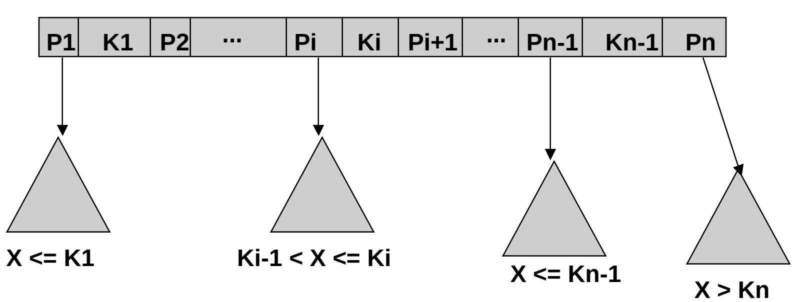
Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Anatomy of a B+ Tree (n pointers)



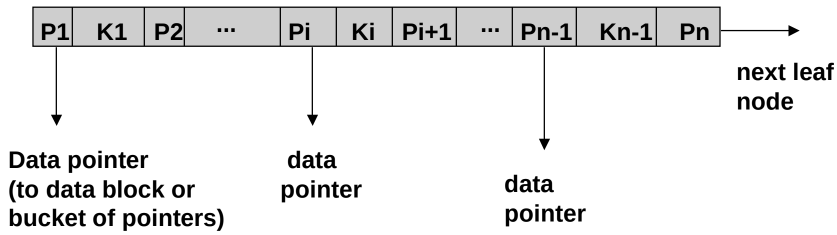
- **Good size for n? -size of a disk block**
- **Leaf nodes form a dense search key index**
- **Leaf node contains search key values K_1, ... ,K_{n-1} in sorted order and P_1, ... ,P_n pointers**
- **Leaf node key values don't overlap**

B+ Tree Internal Node



- **Each internal node (except root) has at least** ceiling(n/2) **pointers**
- **Note some books reverse the convention and have Ki-1 <= X < Ki, which may cause some confusion**

B+ Tree Leaf Node



- **Each leaf node has at least** floor(n/2) **pointers**
- **All leaf nodes are at the same level in the tree**

Processing a Search Query

- **To search for data using a B+ tree, follow pointers from root node to leaf node according to search key value.**
- **Search path length <=** $\log_{\lceil n/2 \rceil}(|K|)$

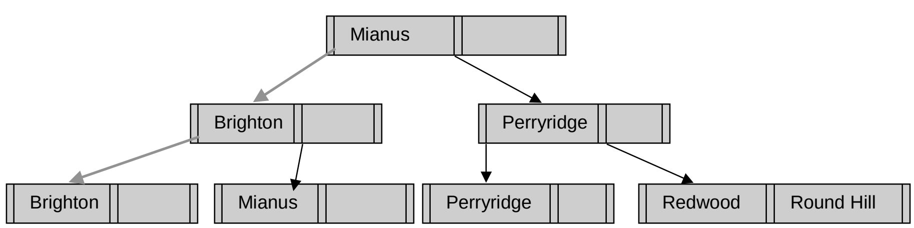
  for at set of search key values K (quite short!)

B+ Trees

- B+ trees are kept <u>balanced</u>, paths from root to leaf is <u>always the same</u>
- Insertions and deletions, therefore, require <u>splitting</u> and <u>merging</u> of nodes to ensure that the tree remains balanced

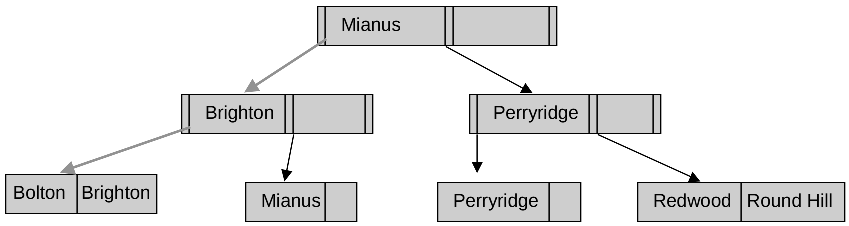
- Exercise: Figure out how to do insertions and deletions so that the tree always stays balanced

Insertions and Deletions



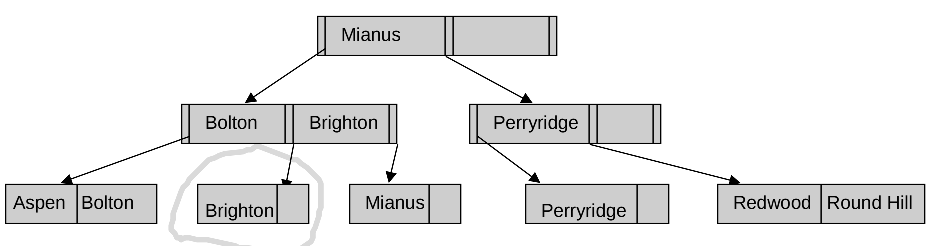
- Insert tuple <“Bolton”, ... >

Insertions and Deletions



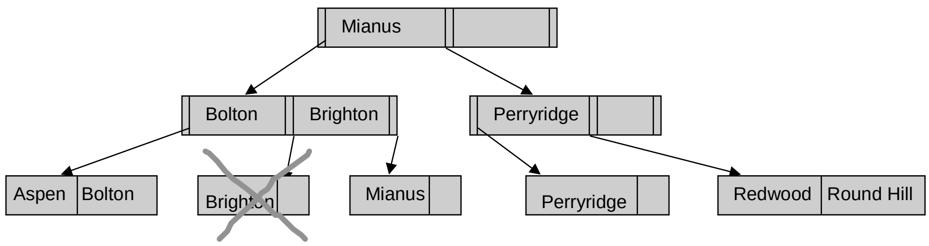
- **Insert tuple <“Aspen”, ... >**

Insertions and Deletions



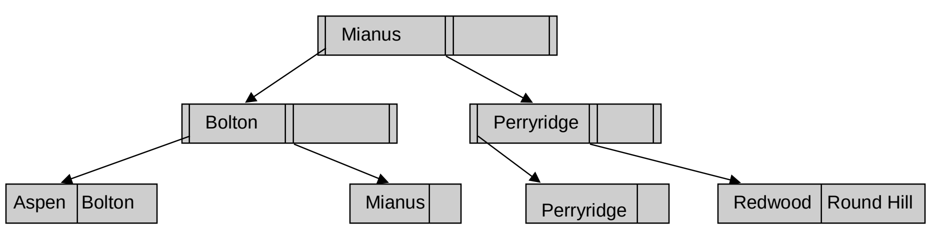
- **Inserting tuple <“Aspen”, ... > requires a new leaf node**
- **Note the internal node must also be updated**
- **Next delete tuple <“Brighton”, ... >**

Insertions and Deletions



- **Delete tuple <“Brighton”, ... >**

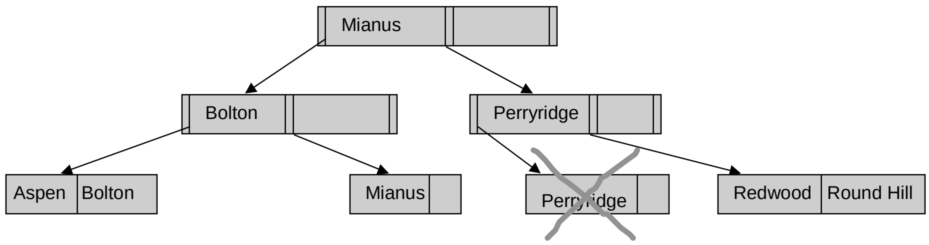
Insertions and Deletions



- **Delete tuple <“Brighton”, ... >**

- **Next delete <“Perryridge”, ... >**

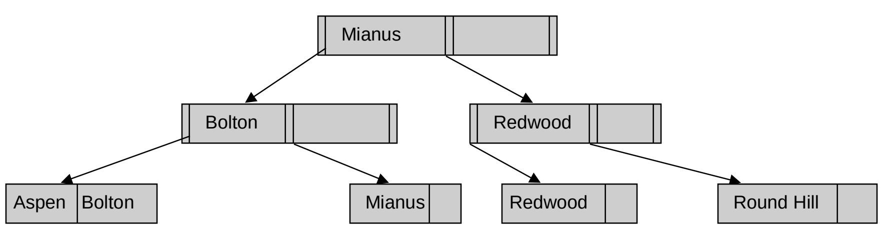
Insertions and Deletions



- **delete <“Perryridge”, ... >**

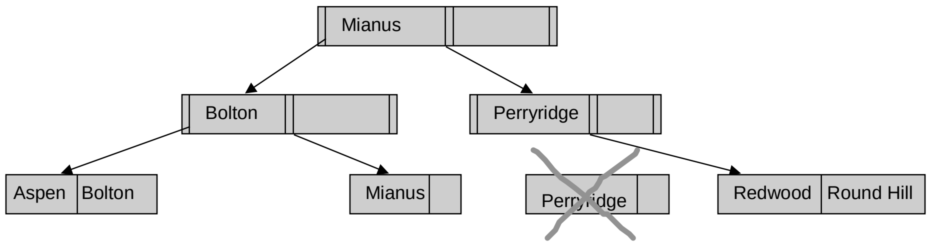
  Notice “Perryridge” internal node will have too few pointers when the leaf node is removed

Insertions and Deletions



- **delete <“Perryridge”, ... >**

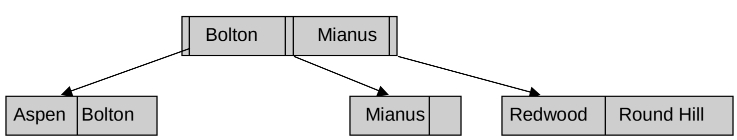
Insertions and Deletions



- **delete <"Perryridge", ... >**

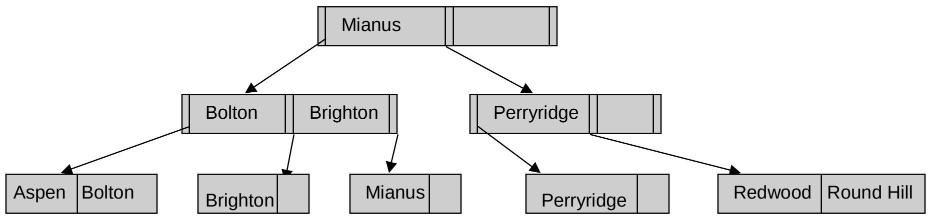
  Alternatively, we can notice the combining the two internal nodes "Bolton" and "Perryridge" will not cause us to exceed the number of available pointers

Insertions and Deletions



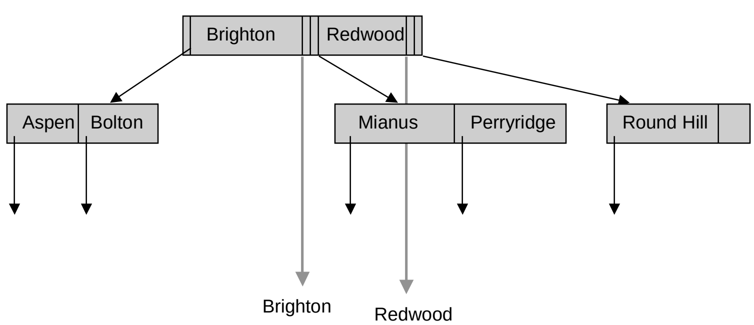
- **delete <"Perryridge", ... >**
- **In this case we are actually able to delete the root and shorten the height of the tree (reducing the block reads by one)**

B+ trees and B trees



- Notice in B+ trees search key values can occur more than one (in the internal nodes and in the leaf nodes)
- B trees are a modification of this by providing a direct data pointer at the first occurrence of a search key value

B trees



- B trees uses one additional pointer per key
- While it appears that fewer nodes are needed the internal nodes of a B-tree are bigger so fewer entries might fit in a disk block
- This can lead to actually more levels in a B-tree compared to a B+-tree

Comparing Methods like Hashing and Indexing

- **Criteria**
- **<u>Access time</u> -time to find data**
- **<u>Insertion time</u> -time to find data position, insert data and update index**
- **<u>Deletion time</u> -time to find data, delete data, and update index**
- **<u>Space Overhead</u> -how much space does the indexing structure take up**

Indexing vs. Hashing --Equality Search

- Select A1, A2, ... ,An
  From R
  Where Ai = c
- **Look up c in index structure or hash c to appropriate bucket (Ai is search key)**
- **indexing: time = log(|R|)**
- **hashing: time = constant**

...Indexing vs. Hashing ---Range Search

- Select A1, A2, ... ,An
  From R
  Where Ai <= c1 and Ai >= c2

- **Indexing: loop up c1 in and chain through file (index) until c2**
- **Hashing: no simple notion of what the next bucket is**
- **Possible solution: find order preserving hash function**
  (e.g. position of first letter of c in alphabet)
- **Good order preserving hash functions are not easy to find**

Multiple Search Key Access

- Select balance
  From deposit
  Where branch = “Ottawa South” and
  customer = “Smith”

- **1) index on branch, examine records to see if customer = “Smith”**
- **2) index on customer, examine records to see if branch = “Ottawa South”**
- **3) use index on branch to find pointer to “Ottawa South” branches, use index on customer to find customer “Smith” pointers, then take intersection of pointers**