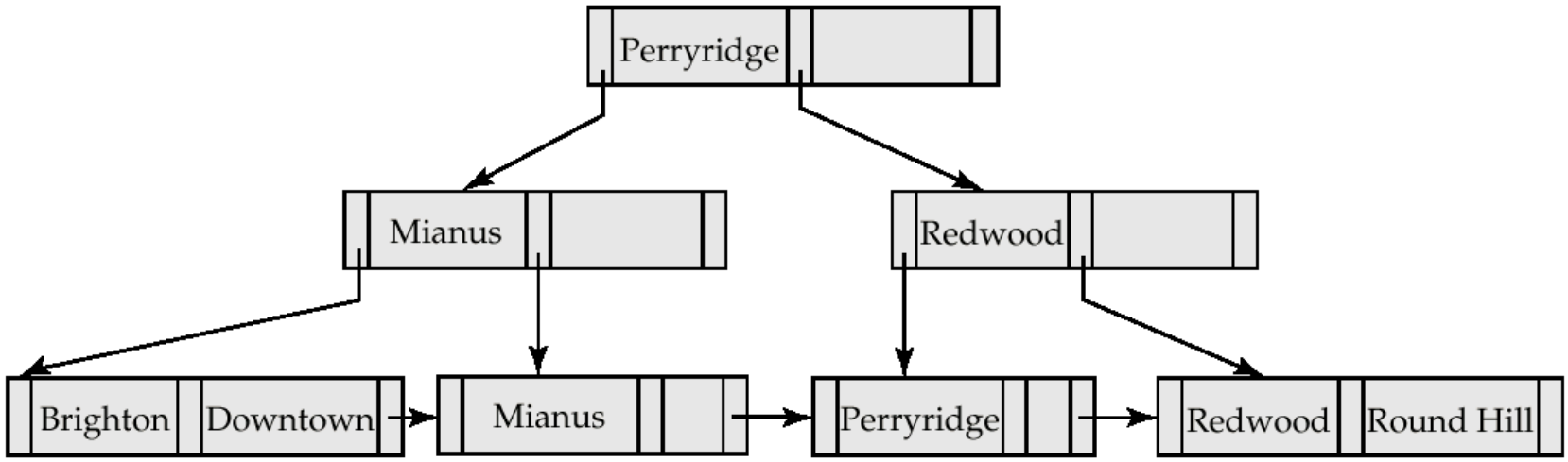
B+ TREE

B^+-Tree Index Files

- ایندکسهای درخت B+ نوعی از ایندکس چندسطحی هستند.
- اشکال فایلهای ایندکس اسپارس و چگال متداول:
  - نیاز به سازماندهی مجدد به صورت دوره ای میباشد.
- مزیت درخت B+:
  - با تغییرات کوچک محلی، بطور خودکار خودش را بازسازی میکند.
- اشکال B+ سربار فضا، درج و حذف است.
- مزایای روش از معایب آن بیشتر است

Example of a B^+-tree

- ساختار B+ با پترامتر n مشخص میشود که تعداد مقادیر و اشاره گرهایی است که یک گره میتواند داشته باشد.



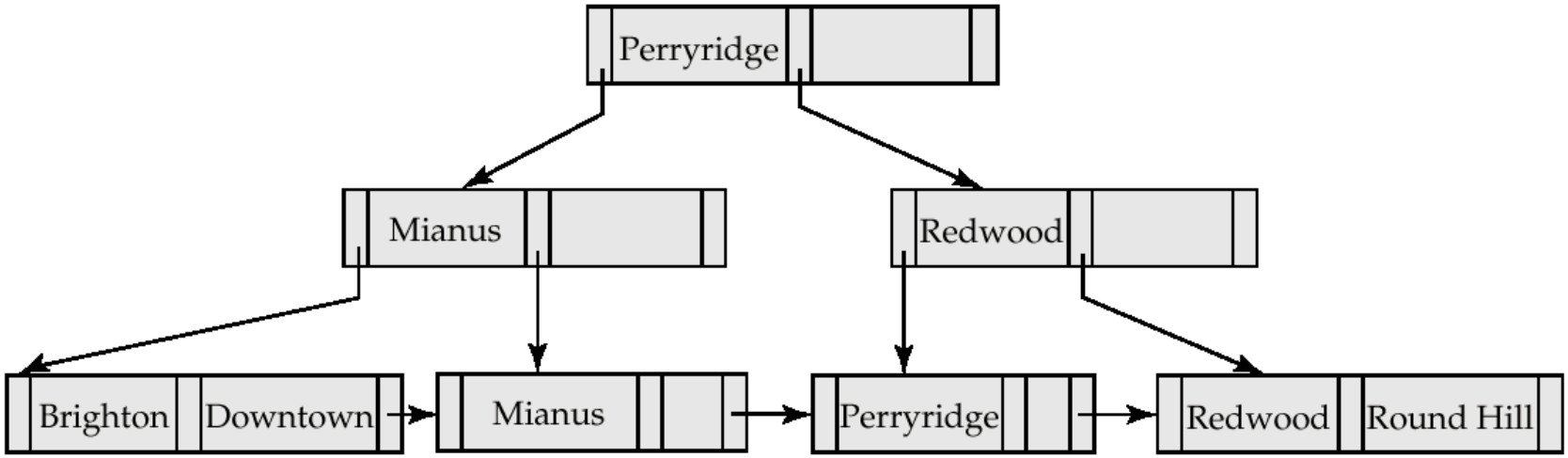
B^+-tree for *account* file ($n = 3$)

Observations about B^+-trees

- هر نود معمولا یک بلاک دیسک است. بلاکهایی که از نظر منطقی نزدیک هستند الزاما ازلحاظ فیزیکی نزدیک نیستند.
- مقدار n معمولا با مقادیر زیر تعیین می شود:
  - Block size
  - Search key size
  - Pointer size

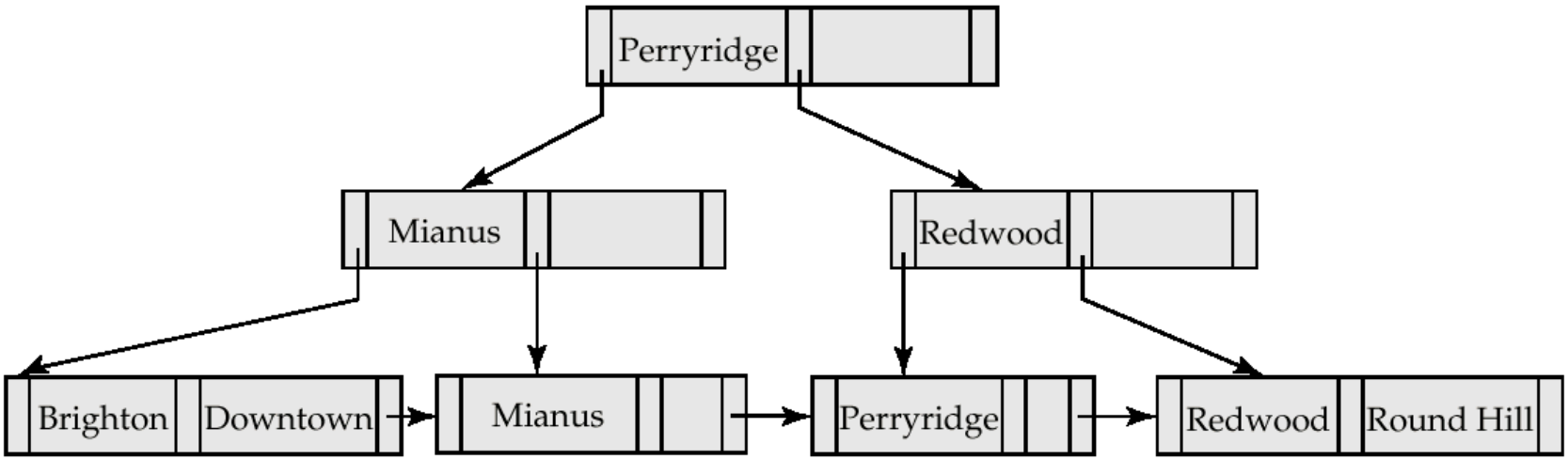
Example of a B^+-tree

- همه مسیرها از ریشه به برگ طول مشابهی دارند.
  - تفاوتی بین زمان جستجوی ایندکس در بهترین و بدترین حالت وجود ندارد.
  - از این لحاظ متفاوت با ساختار hash است.



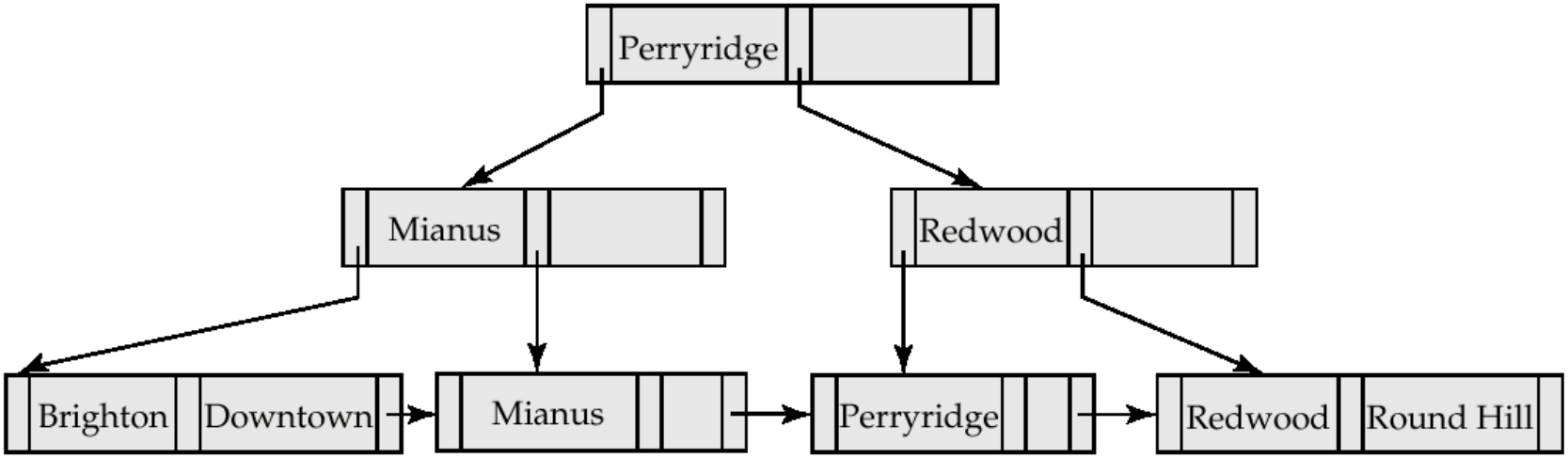
Example of a B^+-tree

- هر گره غیر ریشه حداقل نیمی از آن پر است.
  - Between $\lceil n/2 \rceil$ and n pointers.
  - Between $\lceil n/2 \rceil - 1$ and $n - 1$ search key values.

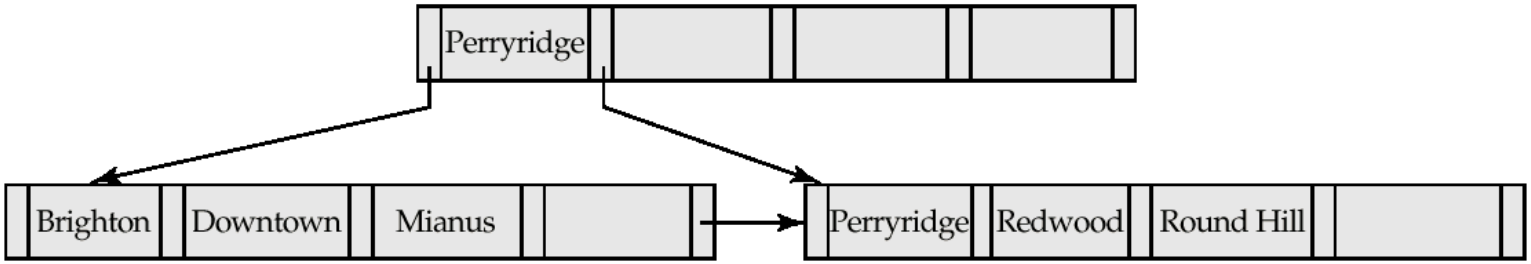


Example of a B^+-tree

- ریشه حالت خاص است:
  - اگر ریشه برگ نباشد میتواند بدون توجه به n دارای دو فرزند باش.
  - اگر رشه برگ باشد و گره دیگری در درخت وجود نداشته باشد می تواند فقط یک مقدار داشته باشد و هیچ فرزندی نداشته باشد.



Another Example



B$^+$-tree for *account* file ($n = 5$)

B^+-Tree Node Structure

- ساختار گره (برگ یا داخلی):

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

- K_iمقادیر کلید جستجو به ترتیب است.

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

- P_iاشاره گری است به:
  - فرزندان که میتواند در گره های غیر برگ، زیر درخت باشد یا
  - در گره های برگ رکوردها یا باکت های رکوردها باشد

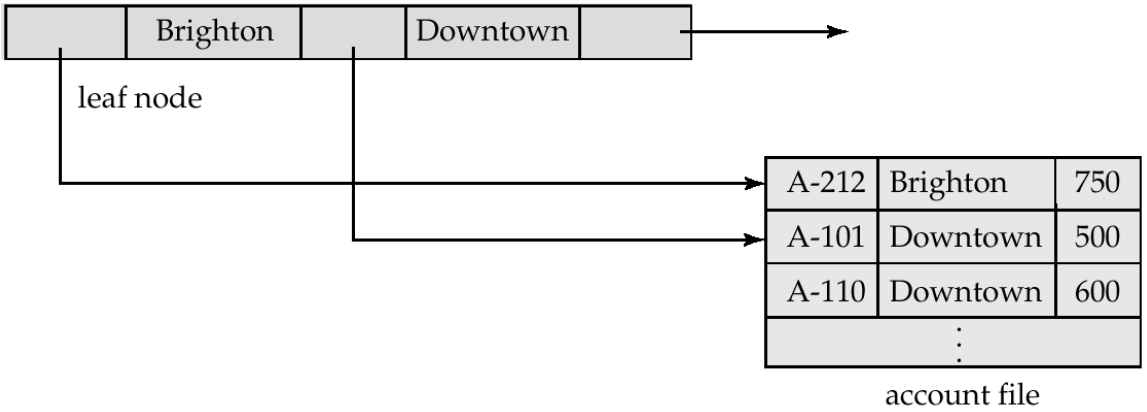
Non-Leaf Nodes in B^+-Trees

- برای گره غیر برگ
  - همه کلیدهای جستجو در زیردرختی که P1 اشاره میکند از k1 کوچکتر هستند.
  - برای $2 \leq i \leq n-1$ همه کلیدهای جستجو در زیردرختی که Pi اشاره میکند مقدارشان بزرگتر یا مساوی K_{i-1} و کمتر از K_i است.
  - همه کلیدهای جستجو در زیردرختی که P_n اشاره میکند بزرگتر از K_{n-1} هستند.

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

Leaf Nodes in B^+-Trees

- برای هر گره برگ:
  - اشاره گر P_i که $1 <= i <= n-1$ به یک رکورد با کلید K_i یا به باکتی از اشاره گرها به رکوردها اشاره میکند که کلید همه آنها K_i است.
  - P_n به برگ بعدی در دنباله کلید جستجو اشاره میکند.



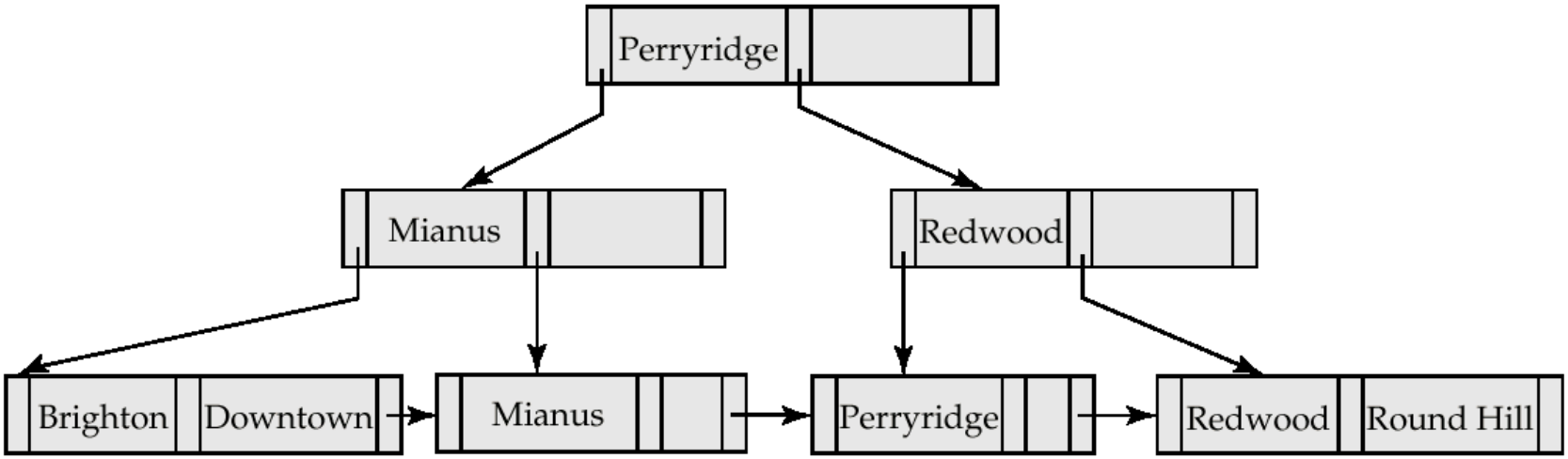
Queries on B^+-Trees

- جستجو در درخت B+
- یافتن همه رکوردها با مقدار کلید جستجوی k:

- Start with the root node
  - Examine the node for the smallest search-key value > k.
  - If such a value exists, call it is K_i, then follow P_i to the child node
  - Otherwise if $k \geq K_{n-1}$, then follow P_n to the child node.
- If the node is not a leaf, repeat the above procedure on that node.
- Eventually reach a leaf node.
  - If for some i, key $K_i = k$, follow pointer P_i to the desired record or bucket.
  - Otherwise no record with search-key value k exists.

Queries on B^+-trees

- Search examples:
  - Downtown
  - Perryridge
  - Newberry
  - All values between Mianus and Redwood (range query)



Queries on B^+-Trees (Cont.)

- در پردازش پرس و جو مسیری در درخت از ریشه به برگ جستجو می شود.
- اگر K مقدار کلید جستجو در فایل باشد طول مسیر برابر $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$. است.
- گره به اندازه بلاک دیسک است (معمولا 4 کیلوبایت) بنابراین n معمولا حدود 100 است. (با فرض این که هر ورودی ایندکس 40 بایت باشد.
- برای 1 میلیون مقدار کلید جستجو و n=100 حداکثر $log_{50}(1{,}000{,}000) = 4$ گره دستیابی می شود.
- برخلاف این، در درخت باینری با 1 میلیون مقدار کلید جستجو حدود 20 نود یا بلاک دستیابی می شود.

Updates on B^+-Trees: Insertion

- درج: گره برگی که کلید جستجو باید درآن باشد پیدا می شود

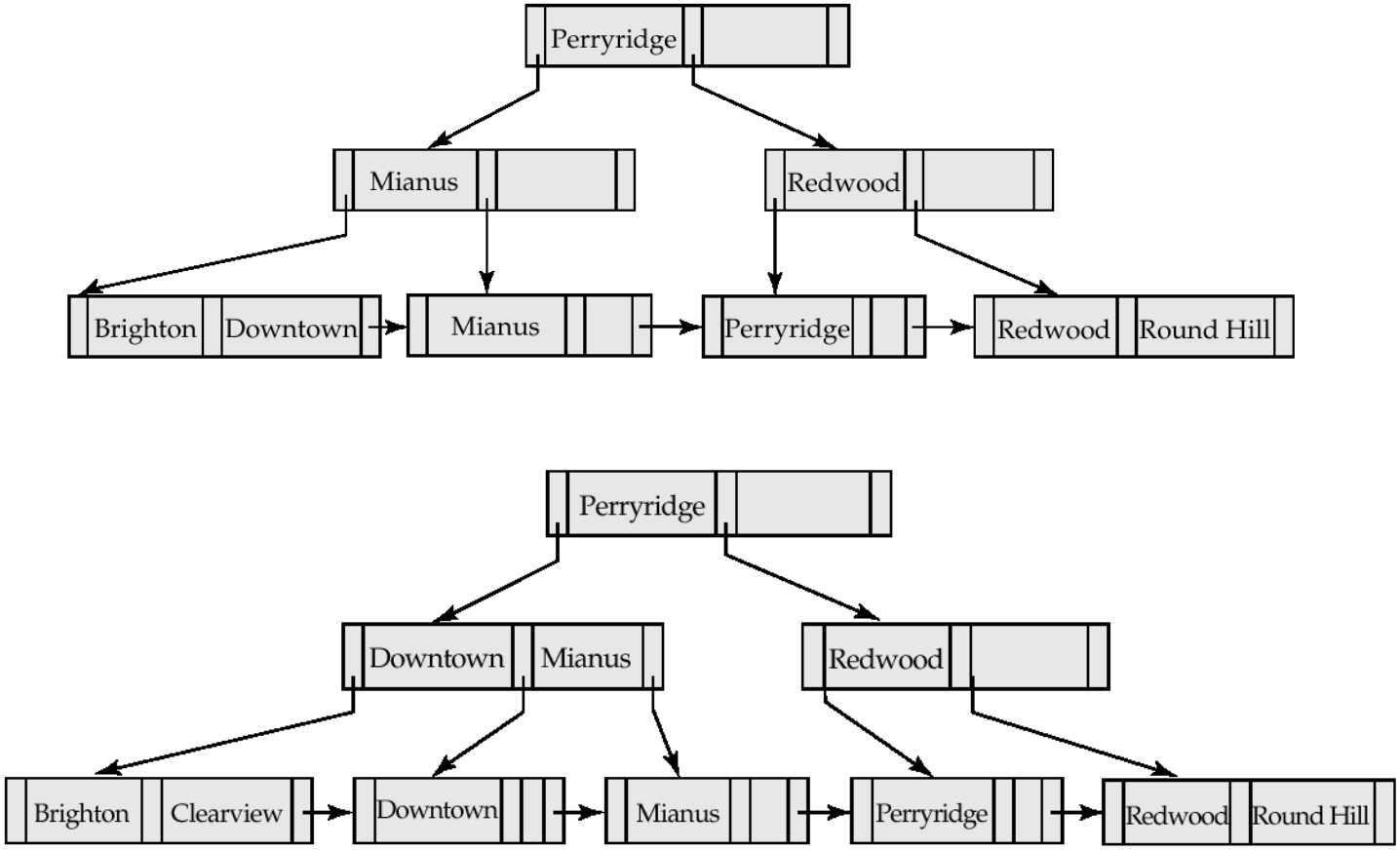
- If the search-key value is already in the leaf node:
  - If the index is on a candidate key field then report an error.
  - The record is added to data file, and
  - If necessary, a pointer is inserted into the bucket (secondary index on a non-candidate key).

- If the search-key value is not in the leaf node:
  - Add the record to the data file, and
  - Create a bucket if the index is a secondary index.
  - If there is room in the leaf node, insert the (key-value, pointer) pair.
  - Otherwise, split the node as discussed in the next slide.

Updates on B^+-Trees: Insertion (Cont.)

- Splitting a (leaf) node:
  - Examine the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.
  - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - Let the new node be *p,* and let *k* be the least key value in *p*.
  - Insert (*k*,*p*) in the parent of the node being split (recursively).
  - If the parent is full, split it and propagate the split further up.
- Splitting nodes continues up the tree until a node is found that is not full.
- In the worst case the root node is split increasing the height of the tree.

Updates on B$^+$-Trees: Insertion (Cont.)



B$^+$-Tree before and after insertion of "Clearview"

Updates on B^+-Trees: Deletion

- حذف:

- Find the record to be deleted, remove it from the data file and from the bucket (if necessary).
- If there are no more records with the deleted search key then remove the search-key and pointer from the appropriate leaf node in the index.
- If the node is still at least half full, then nothing more needs to be done.
- If the node has too few entries, i.e., if it is less than half full, then one of two things will happen:
  - merging, or
  - redistribution

Updates on B^+-Trees: Deletion

- If the entries in the node and a sibling fit into a single node, then the two are merged into one node:
  - Insert all search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

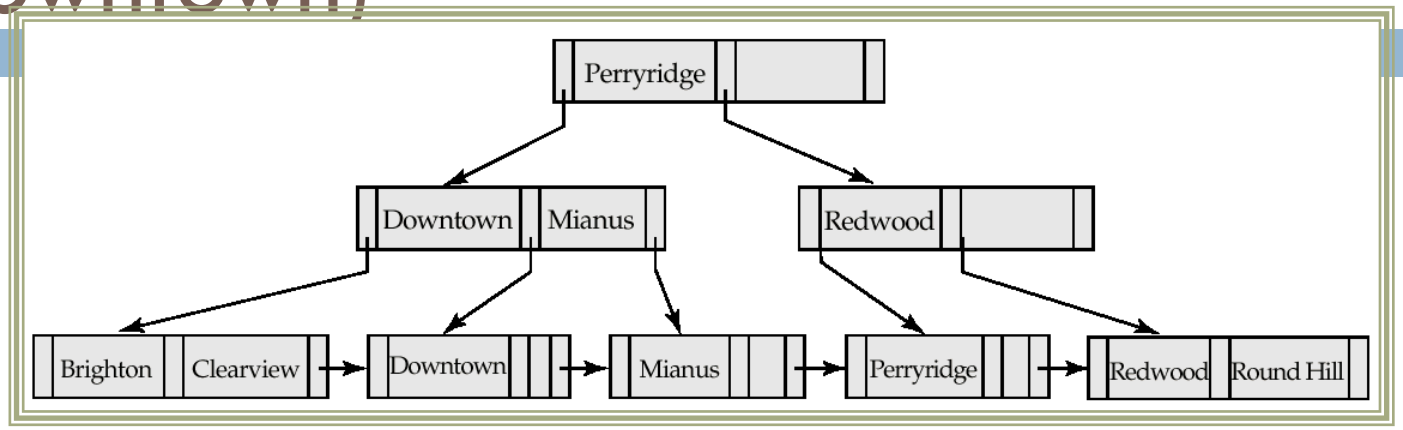
- Otherwise redistribution occurs:
  - Move a pointer and search-key value to the node from a sibling so that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent node.

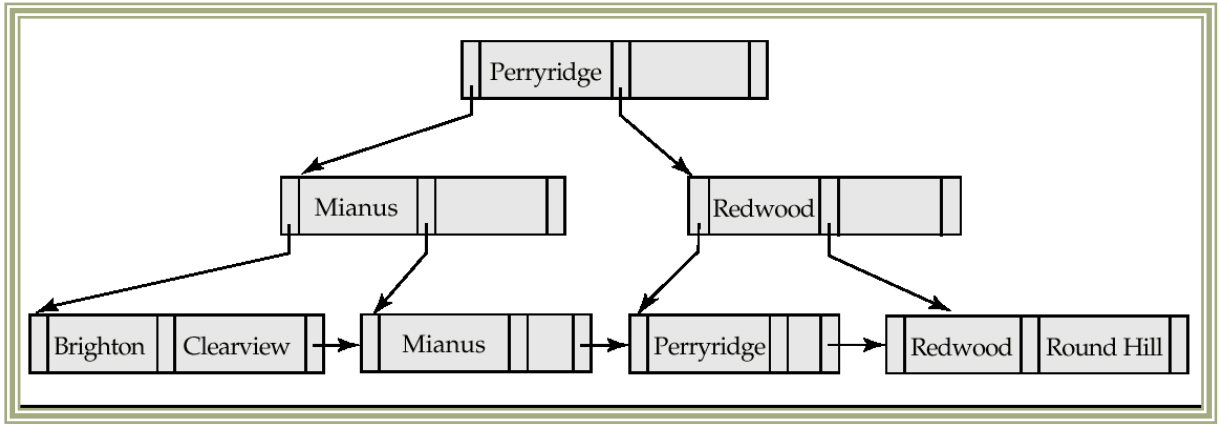
Updates on B^+-Trees: Deletion

- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

- Note that node deletions will cascade upwards until either:
  - a node with $\lceil n/2 \rceil$ or more pointers is reached
  - redistribution with a sibling occurs, or
  - the root is reached.

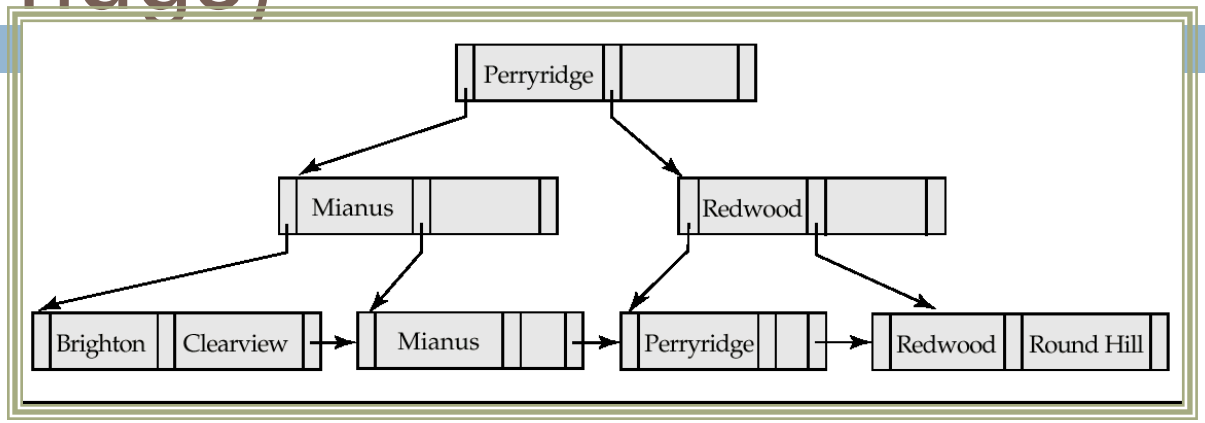
Example of B+-Tree Deletion (Downtown)

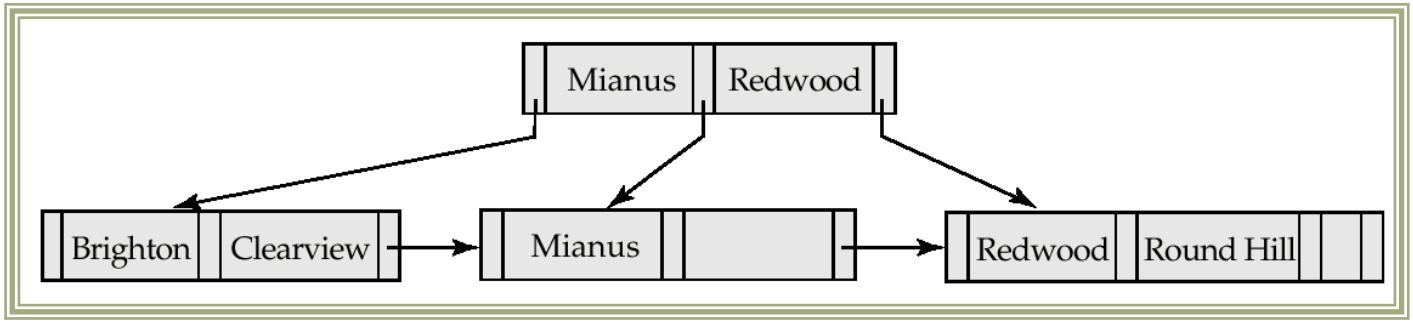




- Node with "Downtown" becomes underfull (actually empty, in this special case) and merged with its sibling.
- Note that the removal of the leaf node did not result in its parent having too few pointers, so the deletions stopped.

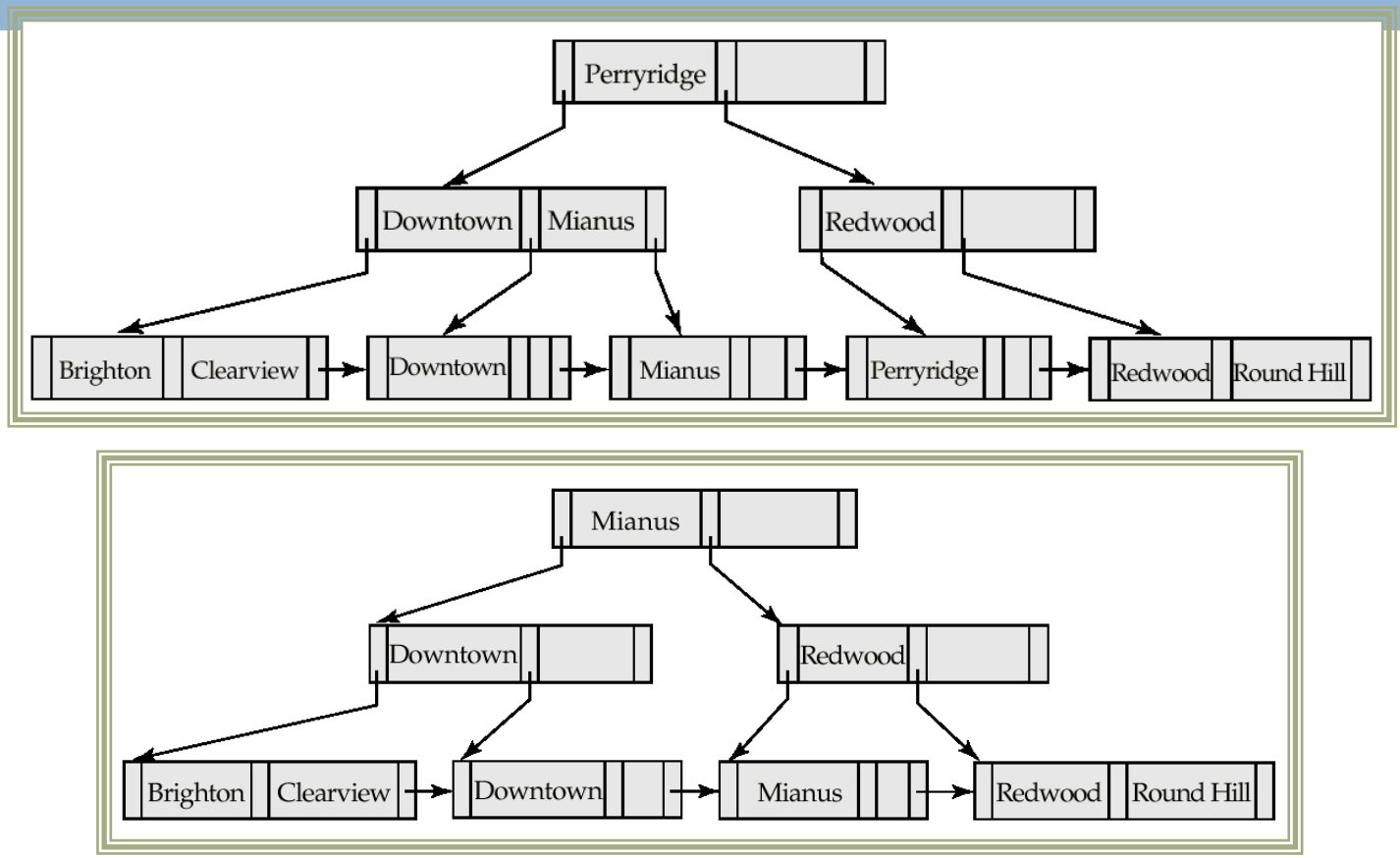
Example of B$^+$-Tree Deletion (Perryridge)





- Node with "Perryridge" becomes under full and merged with its sibling.
- As a result the "Perryridge" node's parent became under-full, and was merged with its sibling (and an entry was deleted from their parent).
- Root node then had only one child, and was deleted.

Example of B$^+$-tree Deletion (Perryridge)



- Parent of leaf containing "Perryridge" became under-full, and borrowed a pointer from its left sibling.
- Search-key value in the root changes as a result.

Static Hashing

- اولین بخش ساختار hash مجموعه ای از باکتها است.

- باکت واحد ذخیره اطلاعات است که شامل یک یا چند رکورد است.
  - باکت میتواند بلاک دیسک باشد.
  - ممکن است شامل بیش از یک بلاک باشد.
  - هر باکت یک آدرس دارد.

Static Hashing

- بخش دوم ساختار hash، تابع hash است.
- مشابه ایندکس تابع hash براساس کلید جستجو است.
- تابع h تابعی از مجموعه ای از مقادیر کلید جستجو K به مجموعه ای از آدرسهای بلاک B است.
  - برای یافتن، درج و حذف رکوردها استفاده می شود.
  - معمولا محاسبه آن سریع و آسان است.
  - معمولا محاسبات برروی شکل باینری کلید جستجو انجام می شود.
- ممکن است کلیدهای جستجوهای مختلف به باکت مشابهی نگاشت شوند. بنابراین برای یافتن یک رکود باید کل باکت جستجو شود.

Example of Hash File Organization

- Consider a hash file organization of the *account* table (next page):
  - 10 buckets (more typically, this is a prime number)
  - Search key is *branch-name*

- Hash function:
  - Let the binary representation of the *I* th character be integer *i*
  - Return the sum of the binary representations of the characters modulo 10

    h(Mianus) is (13+9+1+14+21+19) = *77* mod 10 = *7*

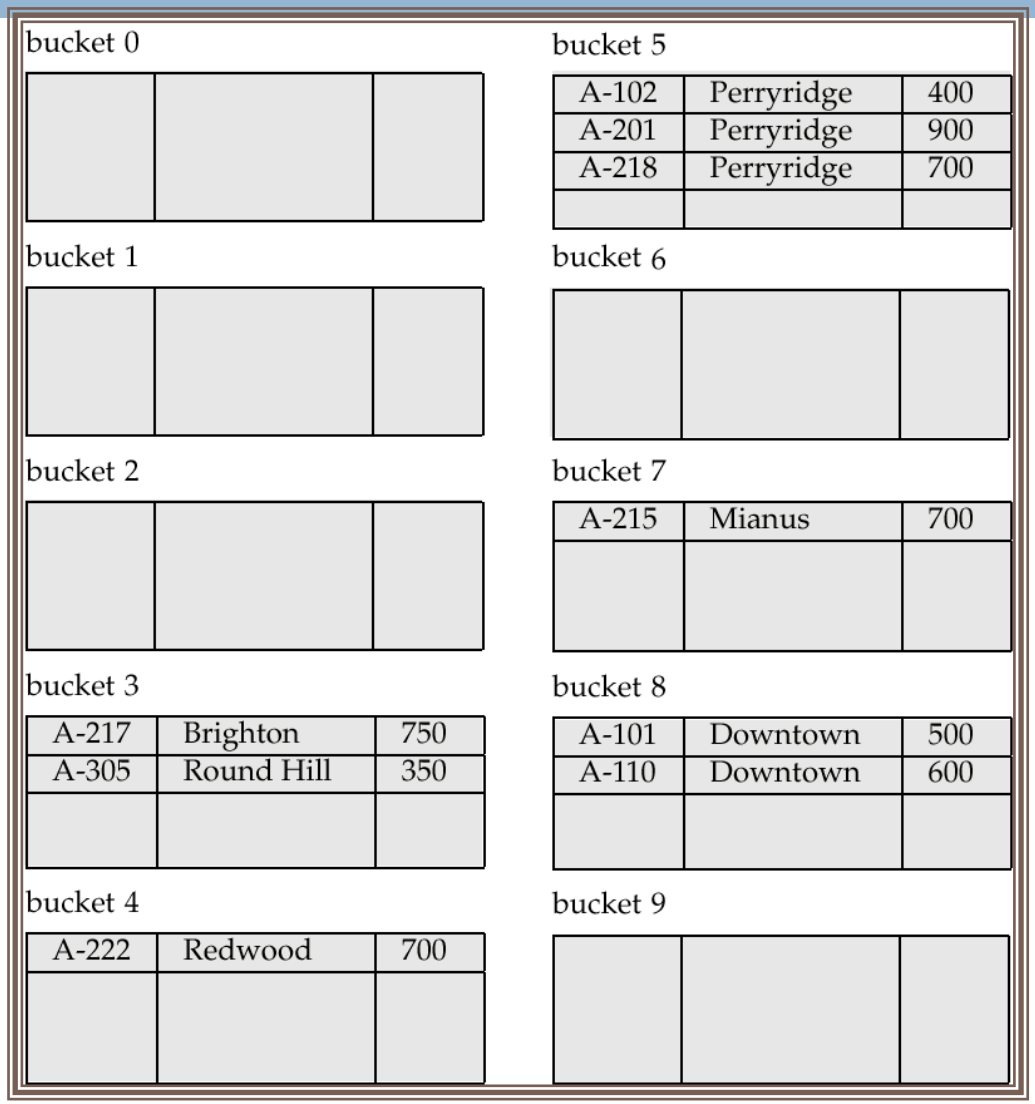
    h(Perryridge) = *5*

    h(Round Hill) = 3 h(Redwood) = 4

    h(Brighton) = 3 h(Downtown) = 8

- Inserting 1 Brighton, 1 Round Hill, 1 Redwood, 3 Perryridge, 1 Mianus and 2 Downtown tuples results in the following assignment of records to buckets.

Example of Hash File Organization



Effective Hashing

- اهداف:
  - دسترسی سریع به رکوردها
  - فضای هرز کم
- فاکتورهای تحقق اهداف
  - داشتن تابع hash مناسب
  - داشتن تعداد مناسبی از باکتها نه زیاد و نه کم

Hash Functions

- یکنواختی: در هر باکت تعداد مشابهی از مقادیر قرار بگیرد.
- تصادفی بودن: بدون توجه به توزیع مقادیر کلید جستجو در فایل، به طور متوسط تعداد رکوردهای مشابهی به هر باکت نسبت داده شود.
  - زمانی که مقدار کلید جستجو کسر کوچکی از رکوردها باشد.
- بدترین حالت: همه مقادیر کلید به یک باکت مشابه نگاشت شوند. بنابراین زمان دستیابی متناسب با مقدار کلید می شود.