Lesson 11: Data Storage
Data Access

Contents

- Part 1: Storage of Data
  - Fixed & Variable Records
  - Sequential Files
  - Multi-table Clustering
- Part 2: Indexing
  - Concepts of Indexing
  - Dense & Sparse Indices, Multilevel Indices
  - B+ Tree Indices
- Part 3: Hashing
  - Static Hashing
  - Hash Functions
  - Dynamic Hashing
  - Extendible Hash Structure

Storing Data

Database in Files

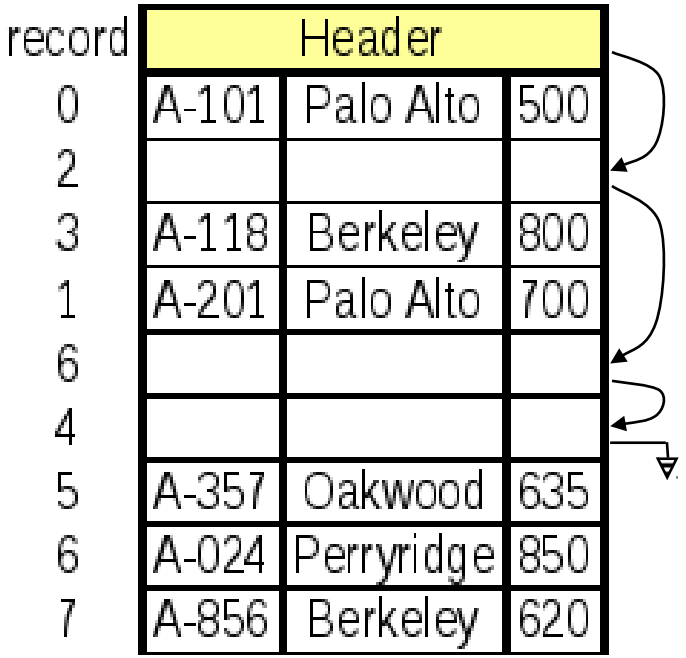
- The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields.
- Simple approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

    This case is easiest to implement; will consider variable length records later
- Fixed-Length Records (of size *n*)
  - Store record *i* starting from byte $n * (i - 1)$
  - Record access is simple but records may cross disk blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - do not move records, but link all free records on a *free list*

record	account_id	customer_city	balance
0	A-101	Palo Alto	500
2	A-102	Bromfield	450
3	A-118	Berkeley	800
1	A-202	Palo Alto	700
6	A-201	Stanford	700
4	A-249	Auckland	550
5	A-357	Oakwood	635
6	A-024	Perryridge	850
7	A-856	Berkeley	620

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation:
  - Reuse space for normal attributes of free records to store pointers.
  - No pointers stored in in-use records
- Use items in the free list when inserting records



Variable-Length Records

- Variable-length records are rare and can arise in database systems in several ways:
  - Storage of multiple record types in a file
  - Record types that allow variable lengths for one or more fields

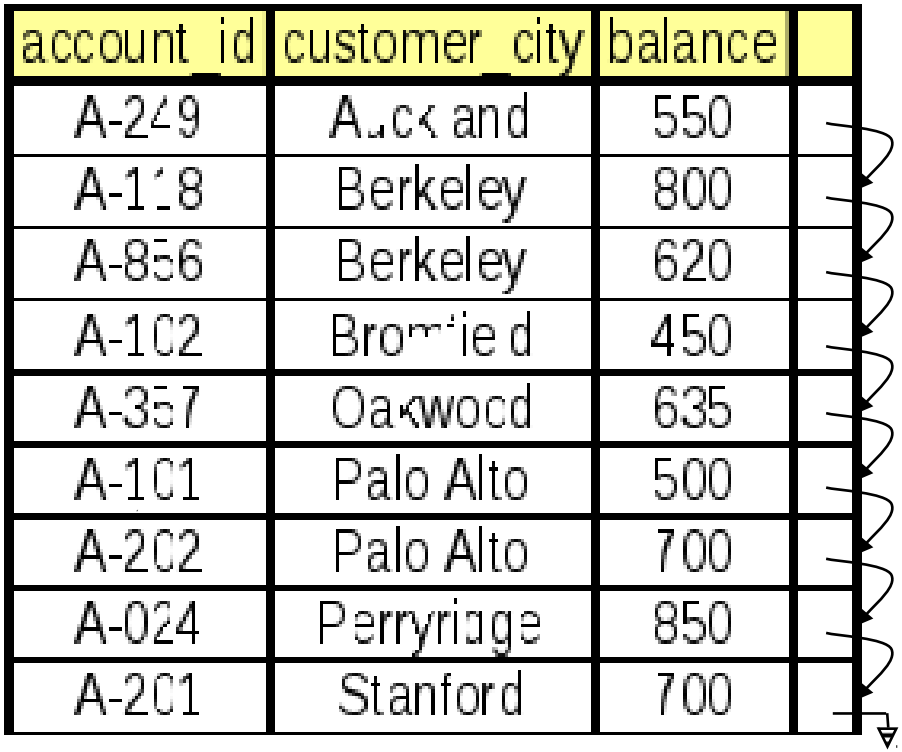
- Variable-Length Records: Slotted Page Structure
  - File is a set of pages
  - Slotted page header contains:
    - number of record entries
    - end of free space in the block
    - location and size of each record
    - Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record
    - instead they should point to the entry for the record in header.

Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record
  - the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multi-table clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Sequential File Organization

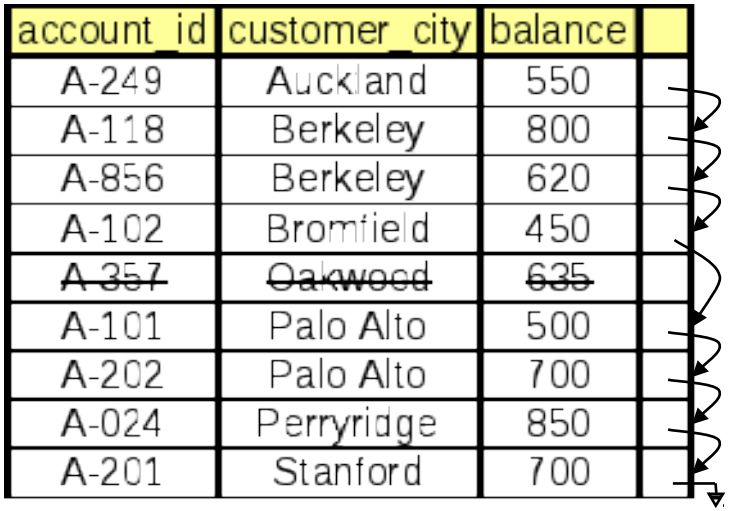
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key
  - Ordered sequential files allow for efficient search but are **difficult to maintain**



account_id	customer_city	balance
A-249	Rock and	550
A-118	Berkeley	800
A-856	Berkeley	620
A-102	Brownfield	450
A-357	Oakwood	635
A-101	Palo Alto	500
A-202	Palo Alto	700
A-024	Perryridge	850
A-201	Stanford	700

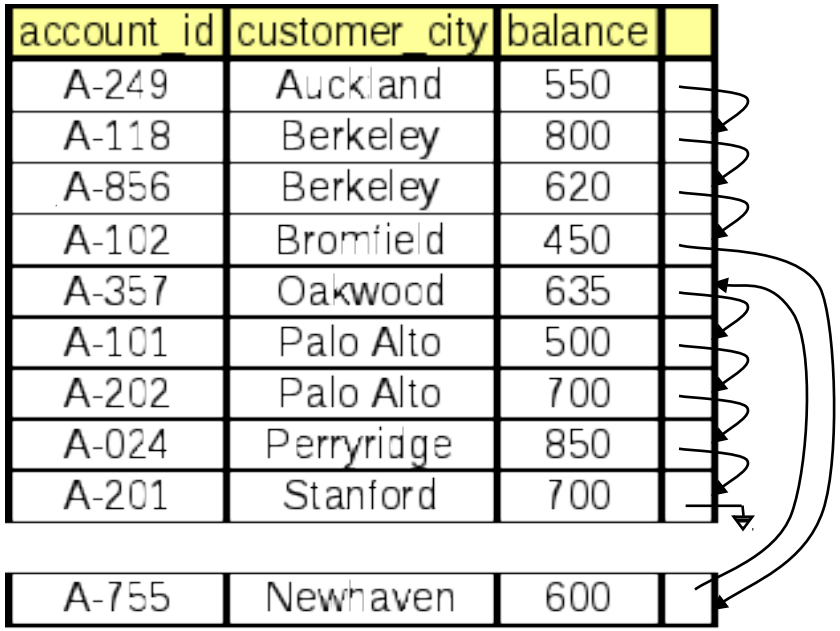
Sequential File Organization (Cont.)

- Deletion
  - use pointer chain to skip the deleted tuple



account_id	customer_city	balance
A-249	Auckland	550
A-118	Berkeley	800
A-856	Berkeley	620
A-102	Bromfield	450
~~A-357~~	~~Oakwood~~	~~635~~
A-101	Palo Alto	500
A-202	Palo Alto	700
A-024	Perryridge	850
A-201	Stanford	700

- Insertion – locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated



account_id	customer_city	balance
A-249	Auckland	550
A-118	Berkeley	800
A-856	Berkeley	620
A-102	Bromfield	450
A-357	Oakwood	635
A-101	Palo Alto	500
A-202	Palo Alto	700
A-024	Perryridge	850
A-201	Stanford	700
A-755	Newhaven	600

- Need to reorganize the file from time to time to restore sequential order

Multi-table Clustering File Organization

- Store several relations in one file using a multi-table clustering file organization
  - Relations *depositor* and *customer*

customer_name	account_id
Johnson	A-101
Johnson	A-202
Johnson	A-301
Hayes	A-102

customer_name	customer_street	customer_city
Johnson	12 Alma St.	Palo Alto
Hayes	22 Main St.	Bromfield

  stored in one file
  - Good for queries involving *depositor* ⋈ *customer*, and for queries involving one single customer and his accounts
  - Bad for queries involving only customer

Johnson	12 Alma St.	Palo Alto
Johnson	A-101	
Johnson	A-202	
Johnson	A-301	
Hayes	22 Main St.	Bromfield
Hayes	A-102	

- Results in variable size records
- Can add pointer chains to link records of a particular relation

Data Dictionary Storage

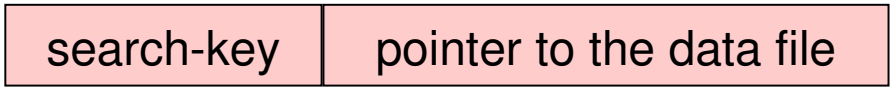
- Data dictionary (also called system catalog) stores metadata (data about data):
  - Information about relations
    - names of relations
    - names and types of attributes of each relation
    - names and definitions of views
    - integrity constraints
  - User and access-rights information, including passwords
  - Views and their definitions
  - Physical file organization information
    - How relation is stored (sequential/hash/…)
    - Physical location of relation
    - Indices (see later)
- Catalog structure
  - Relational representation on disk
- A possible catalog representation
  - a set of relations

Relation_metadata = (*relation_name, number_of_attributes, storage_organization, location*)
Attribute_metadata = (*attribute_name, relation_name, domain_type, position, length*)
User_metadata = (*user_name, encrypted_password, group, ACL*)
Index_metadata = (*index_name, relation_name, index_type, index_attributes*)
View_metadata = (*view_name, definition*)

Indexing

Basic Concepts of Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form



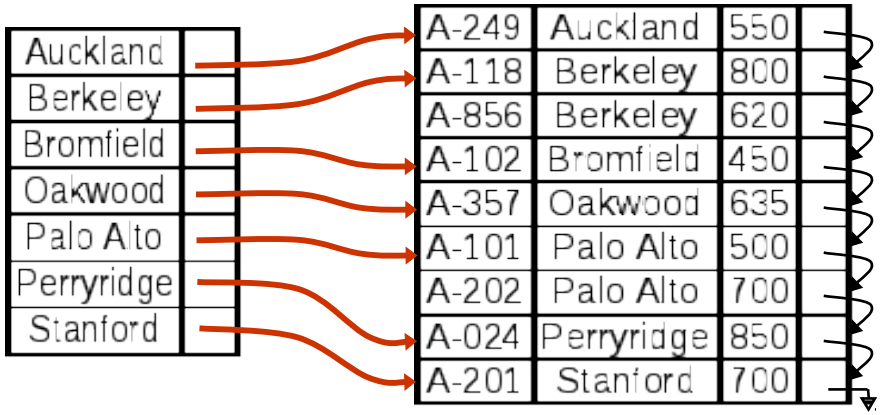
- Index files are typically ***much smaller*** than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hashed indices:** search keys are distributed uniformly across "buckets" using a "hash function"
- Index Evaluation Metrics
  - Access types supported efficiently. E.g.,
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
  - Access time
  - Insertion & Deletion time
  - Space overhead

Ordered Indices

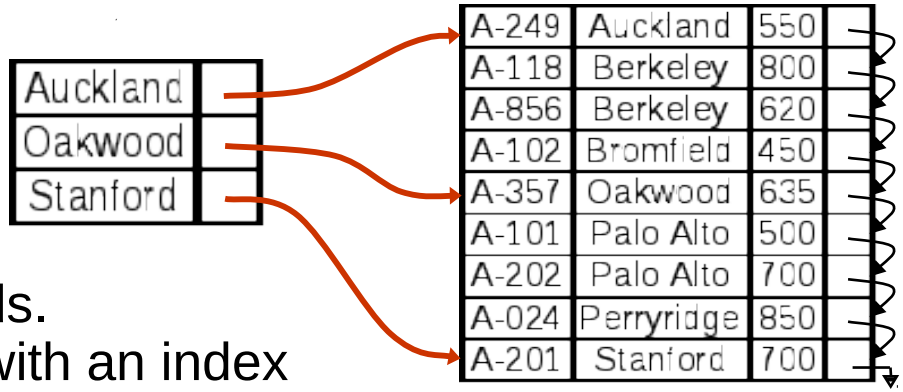
- In an **ordered index**, index entries are stored sorted on the search key value.
  - E.g., author catalog in library
  - Index can be searched by iterated bisection
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index
- **Index-sequential file**: ordered sequential file with a primary index

Dense & Sparse Index Files

- Dense index – contains a record for every search-key value in the data file

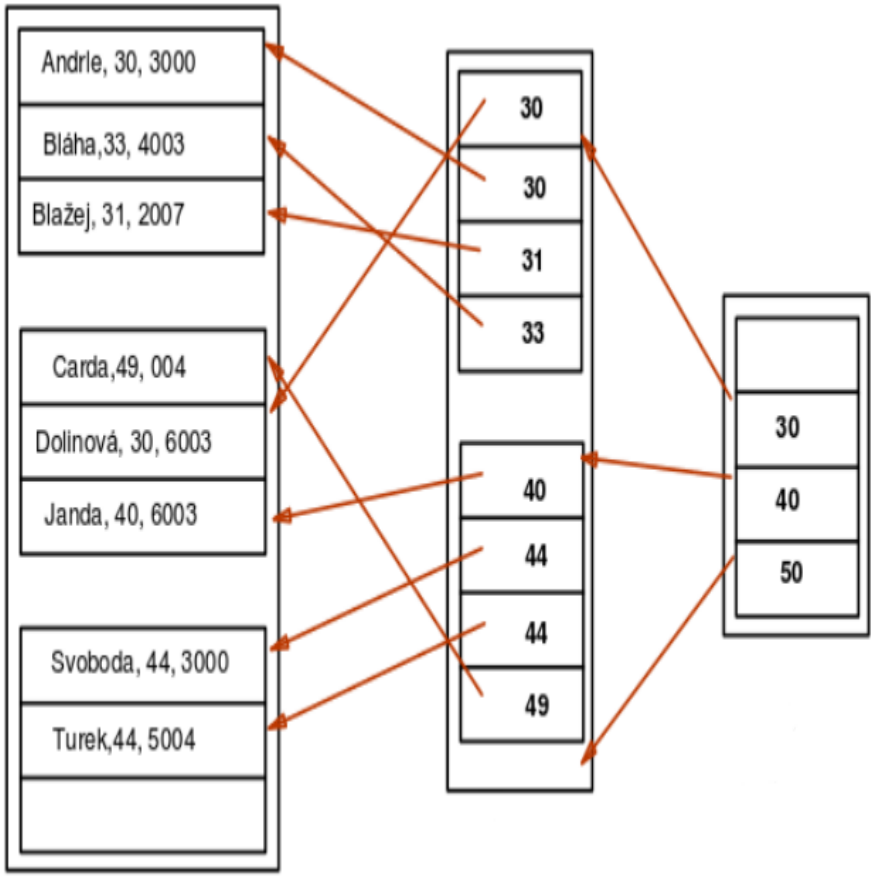


- Sparse Index – contains index records for only some search-key values
  - Applicable when data records are ordered on search-key
  - To locate a record with search-key value *K* we:
    - find index record with largest search-key value < *K*
    - search file sequentially from the record to which the index record points
- Sparse compared to dense
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records.
  - **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block



Multilevel Index

- If primary index does not fit in memory, access becomes expensive
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

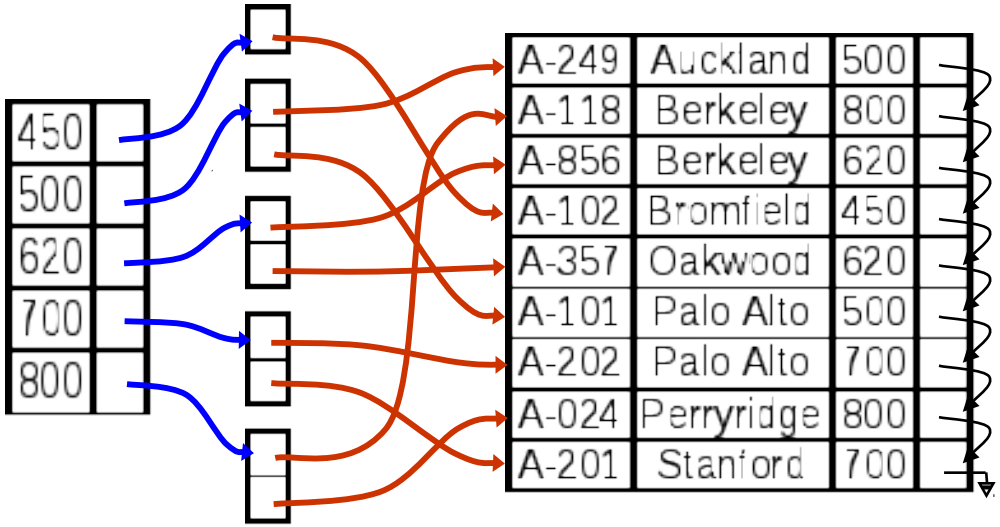


Index Update

- Deletion
  - If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also
  - Single-level index deletion:
    - **Dense indices** – deletion of search-key: similar to file record deletion.
    - **Sparse indices** –
      - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
      - If the next search-key value already has an index entry, the entry is deleted instead of being replaced
- Insertion
  - Single-level index insertion:
    - Perform a lookup using the search-key value appearing in the record to be inserted.
    - **Dense indices** – if the search-key value does not appear in the index, insert it.
    - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
      - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- Multilevel deletion and insertion algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (not necessarily the search-key of the primary index) satisfy some condition.
  - Example 1: In the *account* relation stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value
  - Secondary indices have to be dense
  - Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records
- **BUT**: Updating indices imposes overhead on database modification
  - when a file is modified, **every** index on the file must be updated
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds
    - versus about 100 nanoseconds for memory access

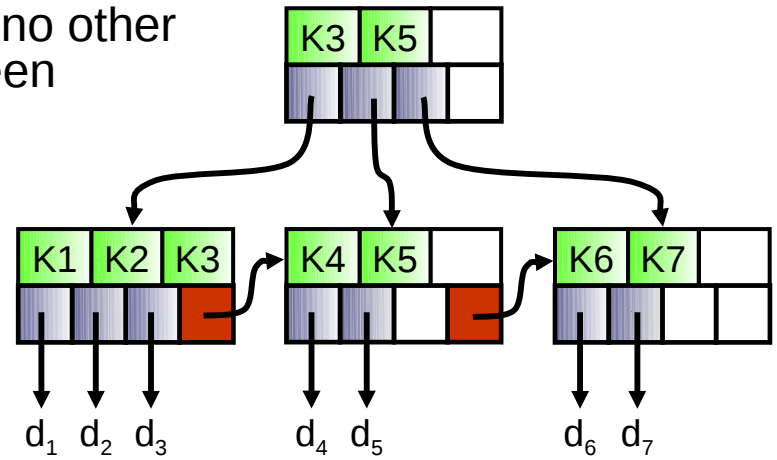
B+ Tree Index Files

B+-tree indices are an alternative to index-sequential files

- Disadvantage of index-sequential files
  - Performance degrades as file grows – too many overflow blocks
  - Periodic reorganization of entire file is required
- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions
  - Reorganization of entire file is not required to maintain performance
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead,
  - space overhead
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

B+ Tree Index Files

- B+ tree is a data structure type of tree which represents sorted data in a way that allows for efficient retrieval, insertion, and removal of records identified by a key. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index "block" or "node"
- B+ tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children, where n is called tree **order** or **branching factor**
    - ***n*** depends on key size and blocks size; usually $n \approx 100$
  - A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children
    - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values



B+ tree with $n=3$

B+ Tree Node Structures

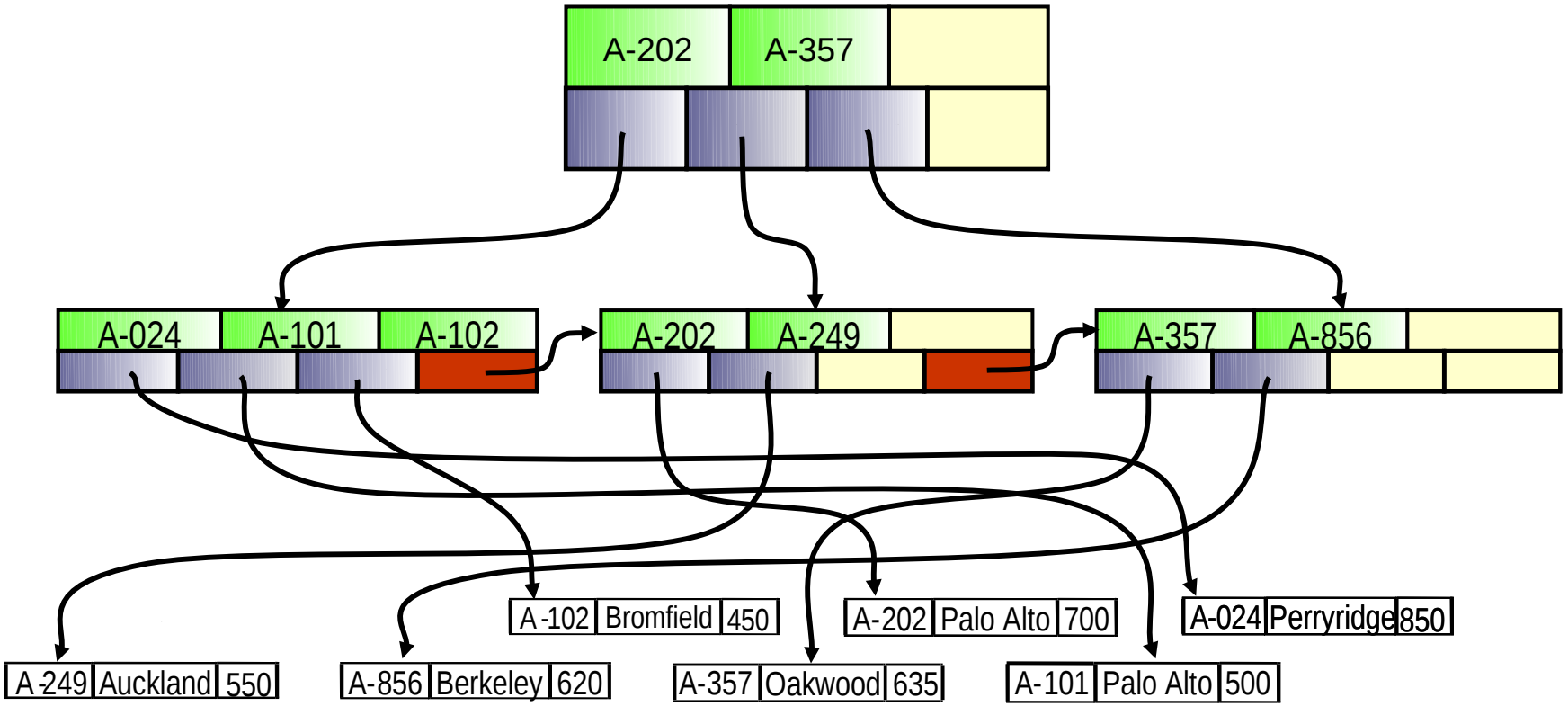
- Typical non-leaf node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
  - The search-keys in a node are ordered

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

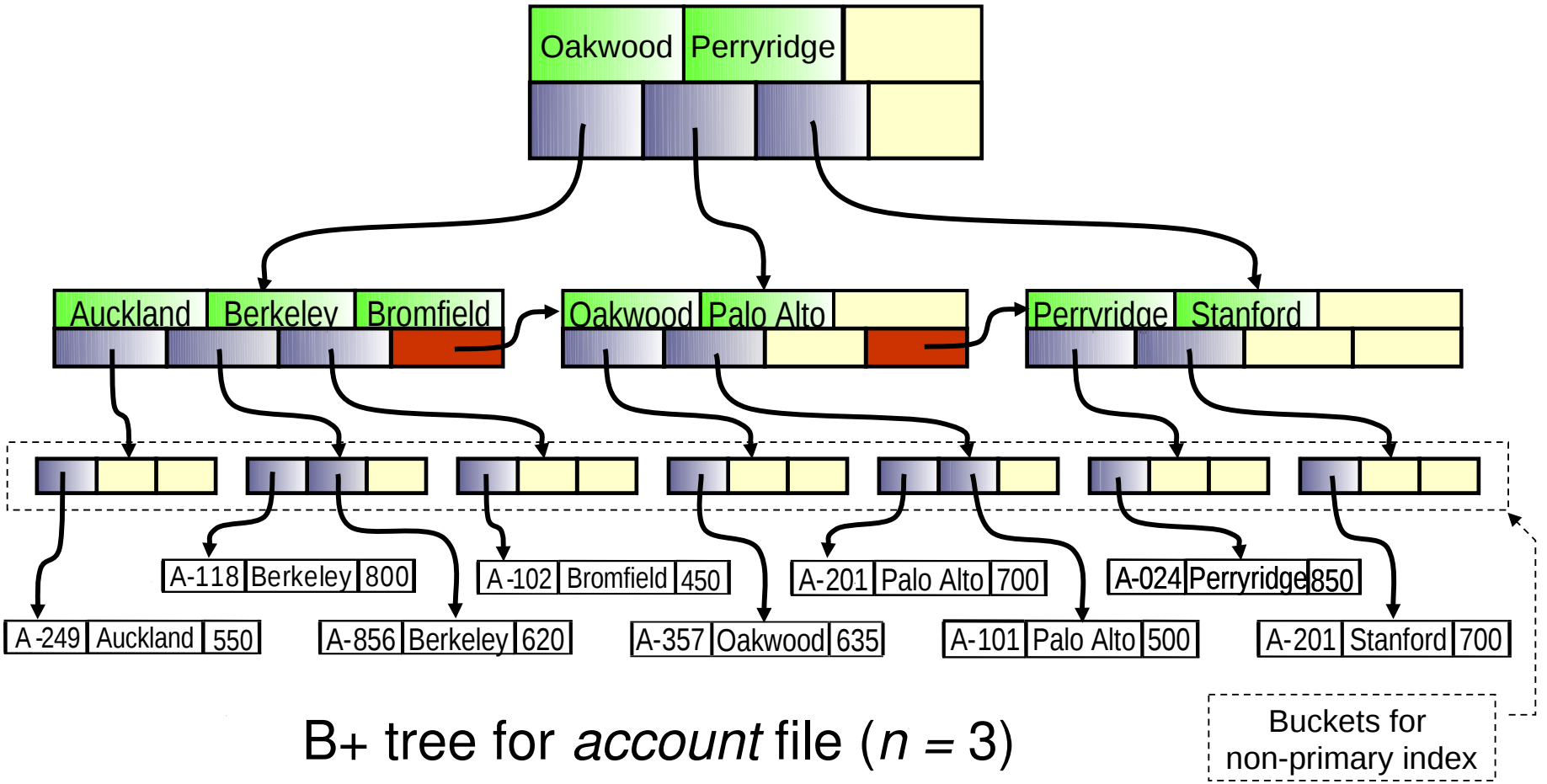
- Non-leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \le i \le n - 1$, all the search-keys in the subtree to which P_i points have key values λ where $K_{i-1} \le \lambda < K_i$
  - All search-keys in the subtree to which P_n points have values $\ge K_{n-1}$
- Properties of leaf nodes
  - For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i.
    - Only need bucket structure if search-key does not form a primary key
  - If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
  - P_n points to next leaf node in search-key order

Example of a B+ Tree for Primary Index



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

Example of a B+ Tree for Non-primary Index



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

Observations about B+ trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close
- The non-leaf levels of the B+ tree form a hierarchy of sparse indices.
- The B+ tree contains a relatively small number of levels
  - Level below root has at least $2* \lceil n/2 \rceil$ values
  - Next level has at least $2* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Searches on B+ Trees

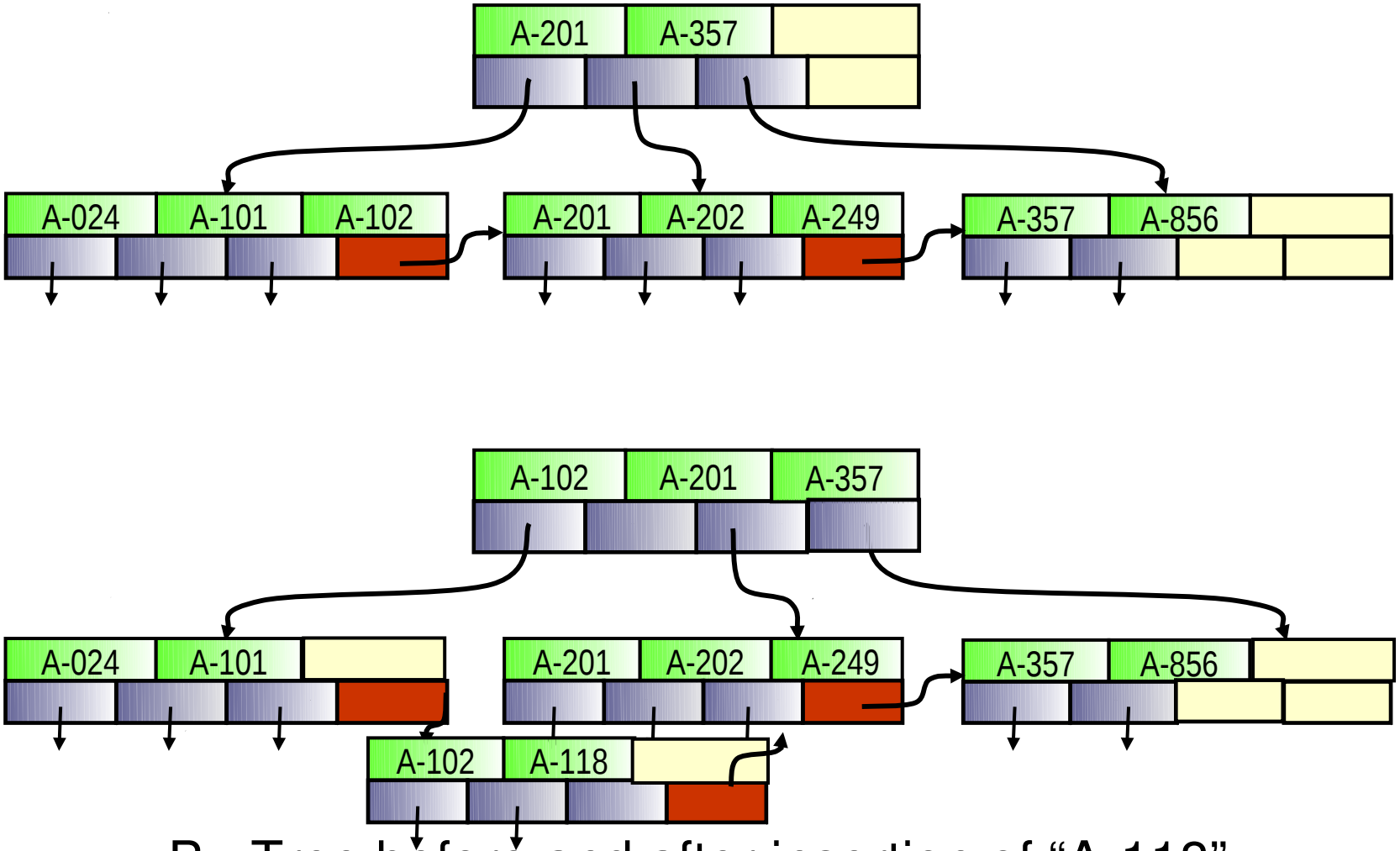
- Find all records with a search-key value of *k.*
  1. *N=root*
  2. Repeat
     1. Examine *N* for the smallest search-key value > *k.*
     2. If such a value exists, assume it is K_i. Then set $N = P_i$
     3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

     Until *N* is a leaf node
  1. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  2. Else no record with search-key value *k* exists.
- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block
  - typically 4 kilobytes and *n* is typically around 100 (≈ 40 bytes per index entry)
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
  - Contrast this with a balanced binary tree with 1 million search key values – around 20 nodes are accessed in a lookup
  - Above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+ Trees: Insertion

- Principal algorithm
  1. Find the leaf node in which the search-key value would appear
  2. If the search-key value is already present in the leaf node
     1. Add record to the file
     2. If necessary add a pointer to the bucket.
  3. If the search-key value is not present, then
     1. add the record to the main file (and create a bucket if necessary)
     2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
     3. Otherwise, split the node
- Splitting a leaf node:
  - Take 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.
  - If the parent is full, split it and **propagate** the split further up.
  - Splitting of nodes proceeds upwards till a node that is not full is found.
    - In the worst case the root node may be split increasing the height of the tree by 1

Updates on B+-Trees: Insertion Example

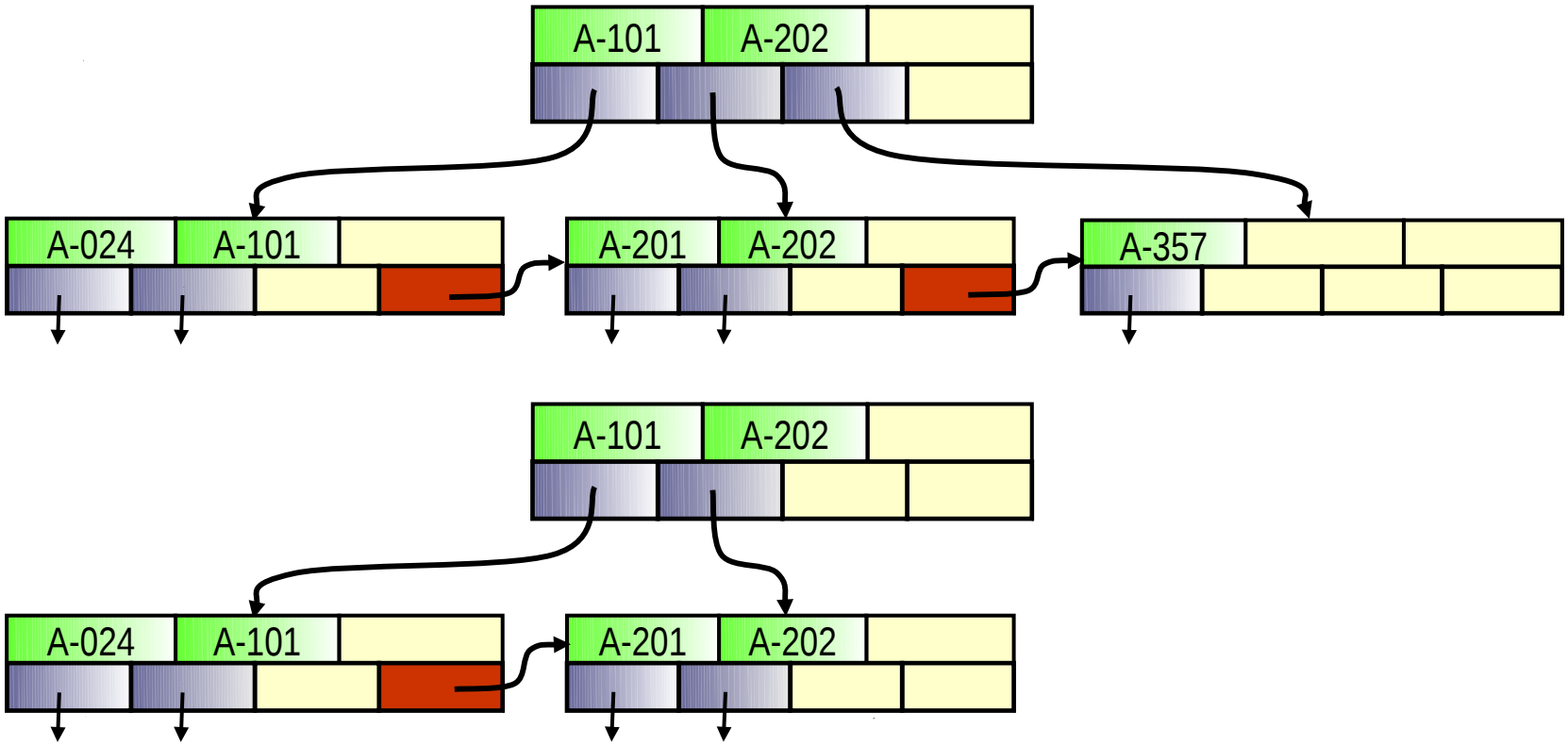


B+ Tree before and after insertion of “A-118”

Updates on B+-Trees: Deletion

- Find the record to be deleted
  - remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node
  - if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal
  - and the entries in the node and a sibling fit into a single node, then **merge siblings**:
    - Insert all the 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, if the node has too few entries due to the removal
  - and the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
    - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
    - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
  - If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

B+ Tree Deletion Example



Before and after deleting “A-357”

- For detailed recursive procedures on B+ tree updates, see literature
  - E.g., Silberschatz A., Korth H. F., Sudarshan S.: *Database System Concepts*

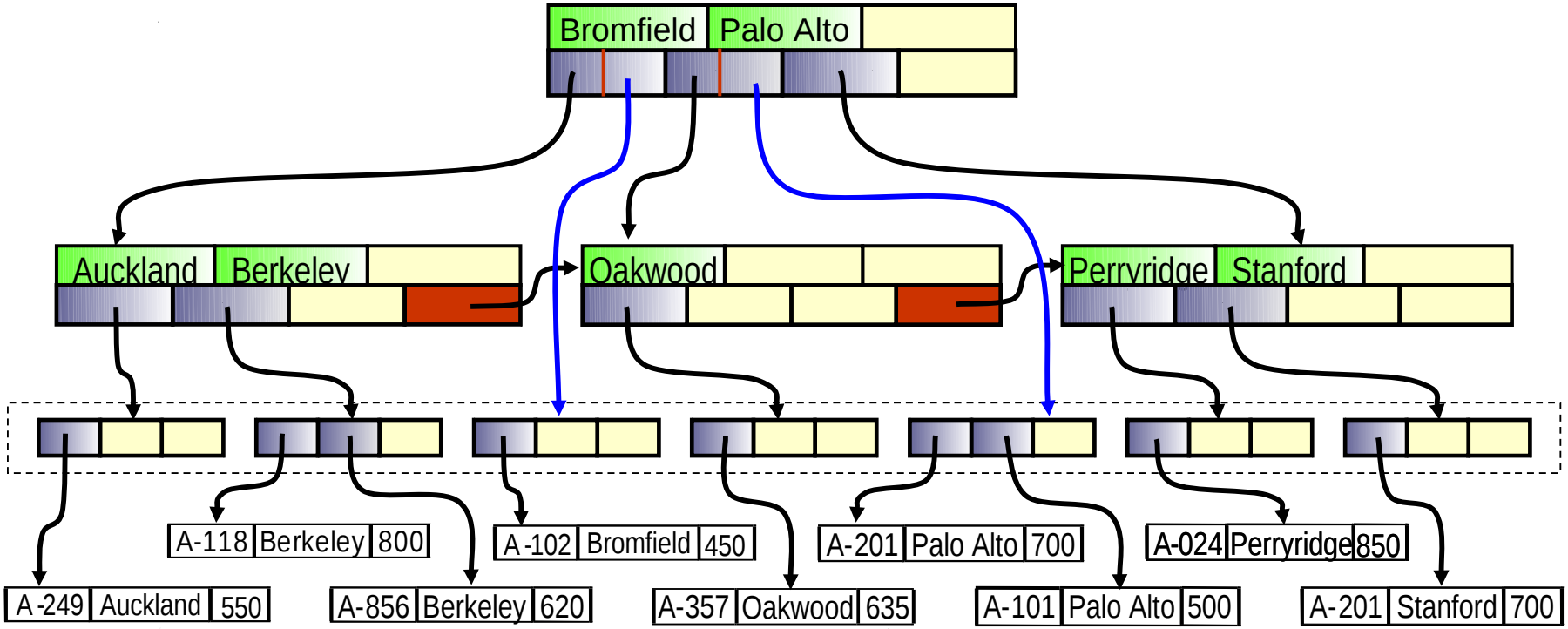
B+ Tree File Organization

- B+ Tree File Organization is a combination of the B+ tree indiex and data into one file
- The leaf nodes in a B+ tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+ tree index
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries.

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once
  - eliminates redundant storage of search keys.
- Search keys in non-leaf nodes appear nowhere else in the B-tree
  - an additional pointer field for each search key in a non-leaf node must be included.
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages

B-Tree Index File Example



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

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:
  select *account_number*
  from *account*
  where *branch_name* = “Perryridge” **and** *balance* = 1000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch_name* to find accounts with branch name Perryridge; test *balance = 1000*
  2. Use index on *balance* to find accounts with balances of $1000; test *branch_name* = “Perryridge”.
  3. Use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained
- Indices on Multiple Keys
  - **Composite search keys** are search keys containing more than one attribute
    - E.g. (*branch_name, balance*)
  - Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
    - $a_1 < b_1$, or
    - $a_1 = b_1$ and $a_2 < b_2$

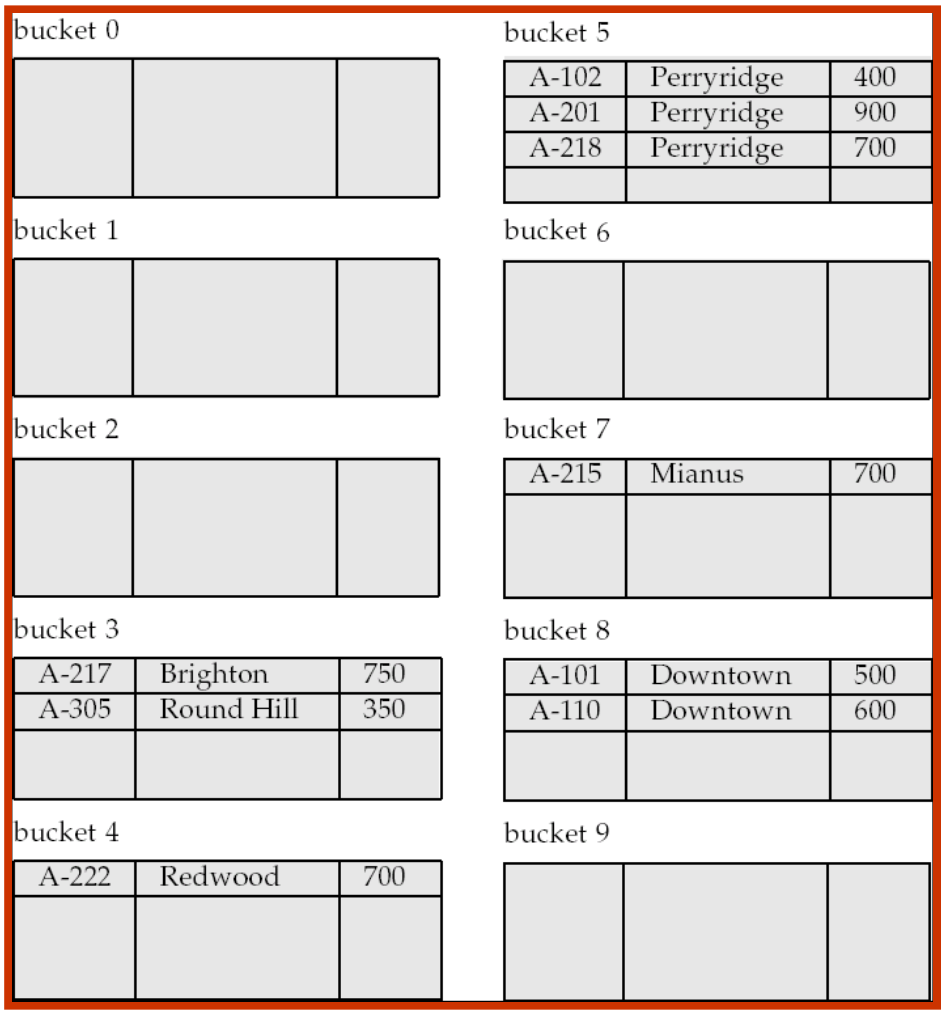
Hashing

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*
  - $B = h(k)$, where *k* is a search-key value
- Hash function is used to locate records for access, insertion, and deletion
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

- Hash file organization of *account* file, using *branch_name* as key
  - There are 10 buckets,
  - The binary representation of the *i*th character is assumed to be the integer *i.*
  - The hash function returns the sum of the binary representations of the characters modulo 10
    - E.g.
      h(Perryridge) = 5
      h(Round Hill) = 3
      h(Brighton) = 3

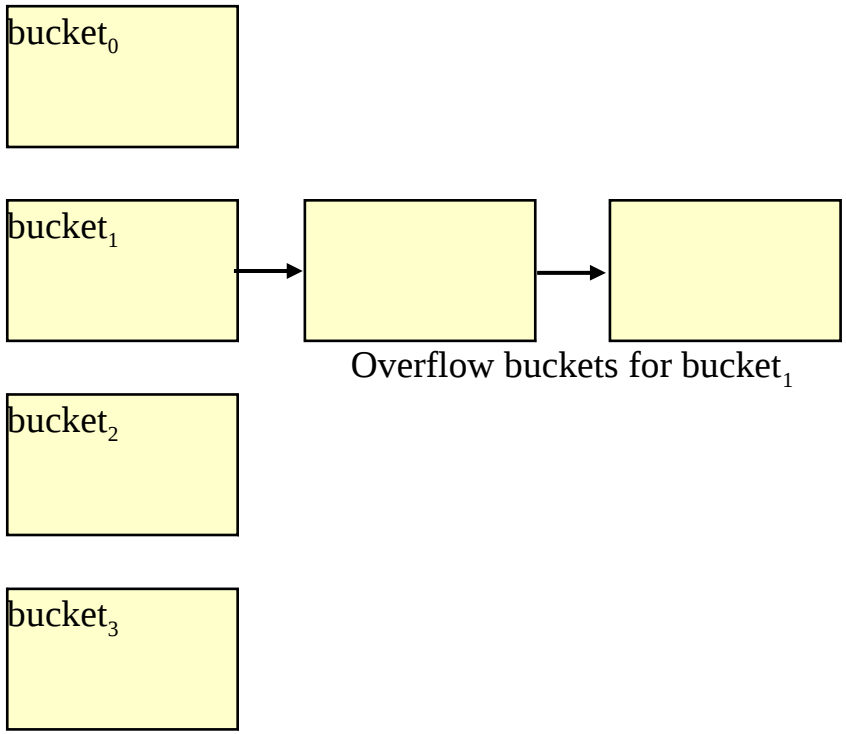


Hash Functions

- Worst hash function maps all search-key values to the same bucket
  - This makes access time proportional to the number of search-key values in the file and brings little benefit
- An ideal hash function is **uniform**
  - Each bucket is assigned the same number of search-key values from the set of *all* possible values
  - Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated
  - It must handled by using ***overflow buckets***
- Overflow chaining
  - The overflow buckets of a given bucket are chained together in a linked list



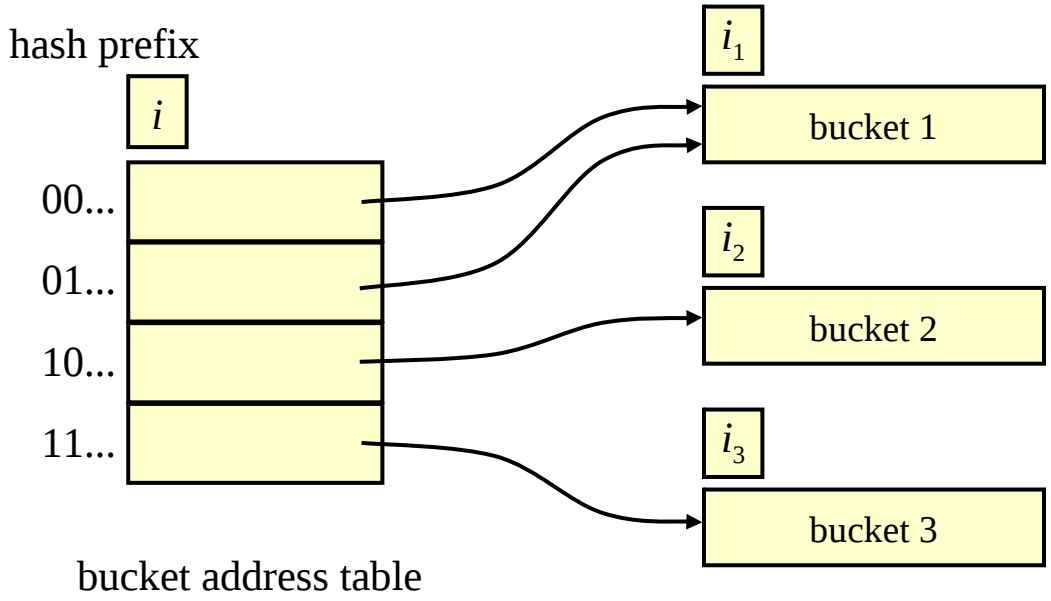
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfilled).
  - If database shrinks, again space will be wasted.
- Possible solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendible hashing** – one form of dynamic hashing
  - Hash function generates values over a large range – typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \le i \le 32$.
    - Bucket address table size = 2^i. Initially $i = 0$
    - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to merging and splitting of buckets

General Extendible Hash Structure



- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits.
  - To locate the bucket containing search-key K_j:
    - Compute $X = h(K_j)$ and use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket

Use of Extendible Hash Structure

- To **insert** a record with search-key value K_j
  - Follow look-up procedure and locate the bucket, say *j*
  - If there is room in the bucket *j* insert record in the bucket.
  - Else the bucket must be split and re-attempt insertion
- To split a bucket *j* when inserting record with search-key value K_j:
  - If $i > i_j$ (more than one pointer to bucket *j*)
    - allocate a new bucket *z*, and set $i_j = i_z = (i_j + 1)$
    - Update the second half of the bucket address table entries originally pointing to *j,* to point to *z*
    - remove each record in bucket *j* and reinsert (in *j* or *z*)
    - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket *j*)
  - If *i* reaches some limit *b*, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment *i* and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above

Extendible Hash Structure – Simplistic Example

- Suppose bucket capacity = 1 record
- First two records with keys k_1 and k_2

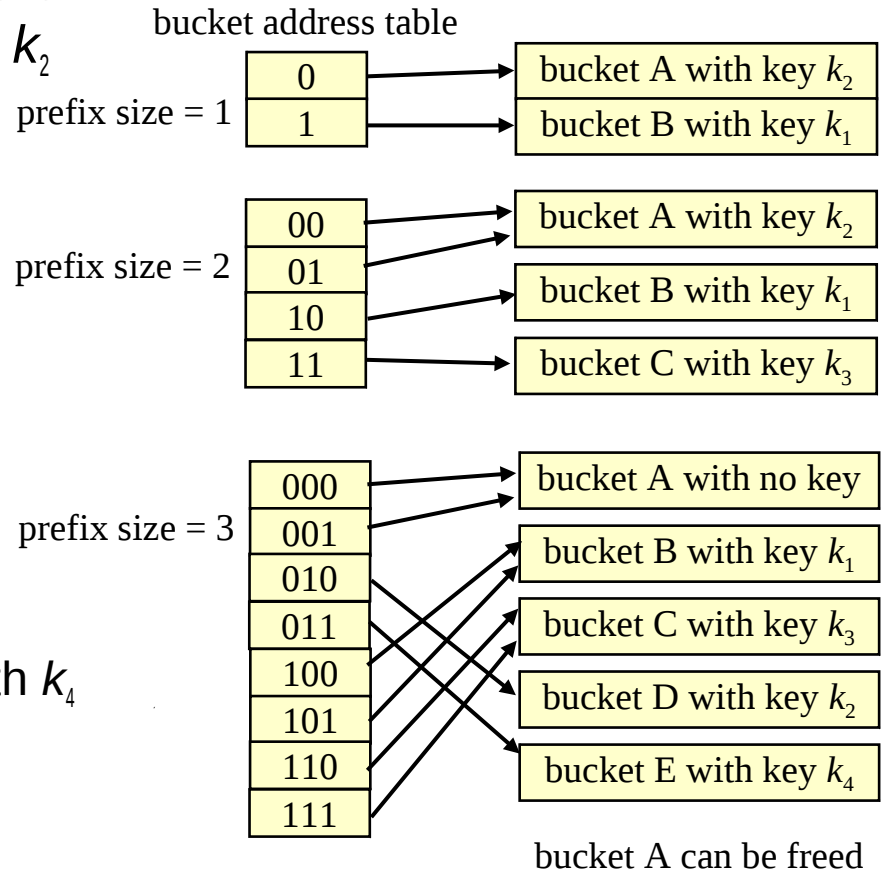
  $h(k_1)$ = 100100

  $h(k_2)$ = 010110
- Third record with k_3 comes

  $h(k_3)$ = 110110
- Record 4 with k_4 comes

  $h(k_4)$ = 011110

  causes bucket A to overflow
  and must be split into A and D
  Re-insertion attempt of record with k_4
  into D causes overflow again
  into D and E and further bit
  has to be added



Deletion in Extendible Hash Structure

- To delete a key value
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Merging of buckets can be done
    - can merge only with a "*buddy*" bucket having same value of i_j and same i_j–1 prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

- Please read other examples in the slides provided on the course web!

Extendible Hashing vs. Other Schemes

- Benefits of Extendible hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of Extendible hashing:
  - Extra level of indirection to find desired record
  - Bucket address table may become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B+ tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B+ trees

End of Lesson 11

Questions?