COMP90050
Advanced Database Systems

THE UNIVERSITY OF MELBOURNE

Winter Semester, 2023

Lecturer: Farhana Choudhury (PhD)

Week 2 part 4-5

Core Concepts of Database management system

Database performance metrics

- Efficiency/speed
- Effectiveness
- Security & Reliability

Efficiency

- Hardware
  - Disks and I/O bandwidth
  - Main memory
  - Type of architecture
- Software/ DB tuning
  - Types of DB
  - Indexing
  - Query optimisation

Effectiveness

- Concurrent users
- Transactions

Reliability

- Crash recovery
- Fault tolerance
- Data duplication

What needs to be efficient

- DBMS must support:
  –insert/delete/modify record
  –read a particular record (specified using record id)
  –scan all records (possibly with some conditions on the records to be retrieved), or scan a range of records

A key choice to make

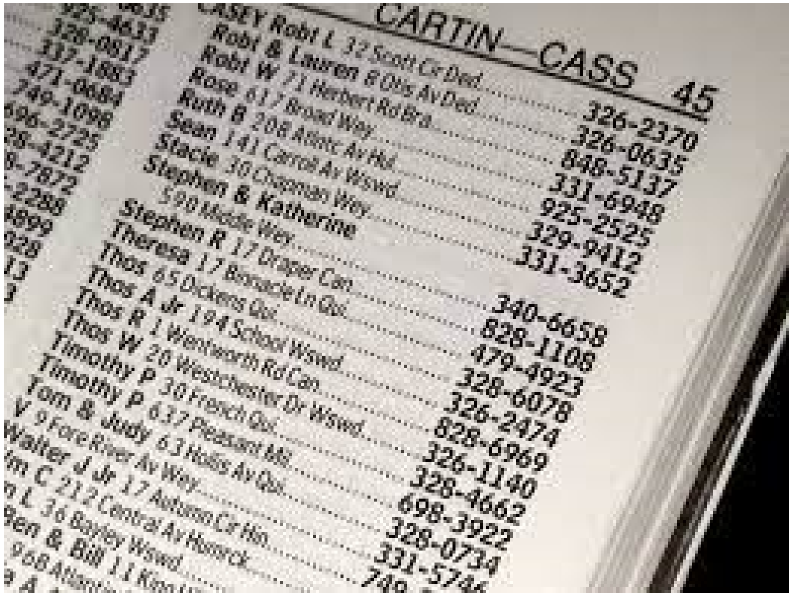
- **DBMS admin** generally **creates indices** to allow almost direct access to individual items
- These are also good during join operations
  - if there is a join condition that restricts the number of items to be joined in a table

Topics

- What is indexing
- Different types of indexes
- Usage of indexes
- Indexing in practice

Indexing is Critical for Efficiency

- Indexing mechanisms used to speed up access to desired data in a similar way to look up a phone book or dictionary
- **Search Key** - attribute or set of attributes used to look up records/rows in a system like an ID of a person

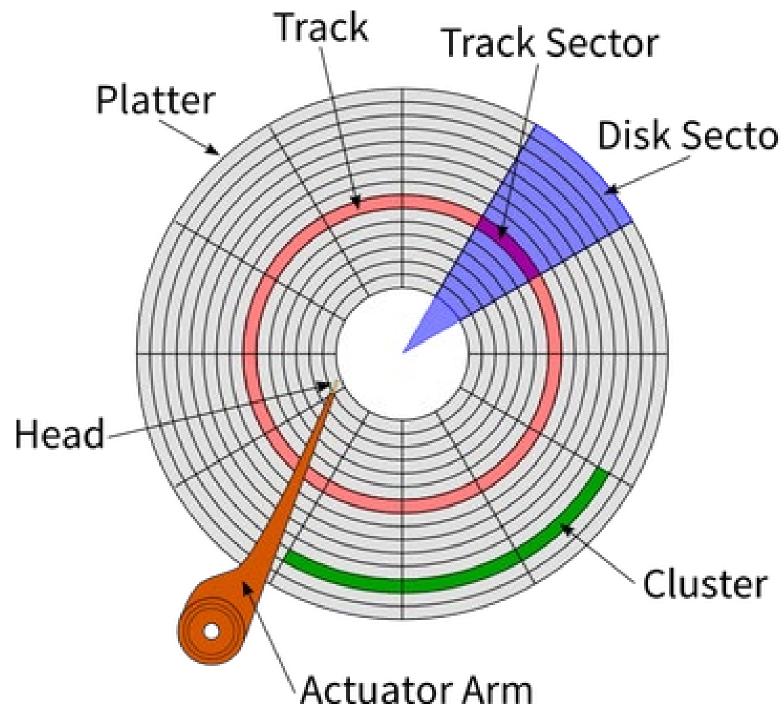


Indexing is Critical for Efficiency

- An **index file** consists of records (called **index entries**) of the form **search-key, pointer to where data is**
- Index files are typically much smaller than the original data files and many parts of it are already in memory

What becomes faster?

- Disk access becomes faster through:
  - records with a specified value in the attribute accessed with minimal disk accesses
  - or records with an attribute value falling in a specified range of values can be retrieved with a single seek and then consecutive sequential reads



Indexing is Critical for Efficiency

- Insertion time to index is also important
- Deletion time is important as well
- No big index rearrangement after insertion and deletion
- Space overhead needs to be considered for the index itself
- No single indexing technique is the best. Rather, each technique is best suited to particular applications.

Indexing is Critical for Efficiency

- Two basic kinds of indices based on search keys:
  - **Ordered indices:** search keys are stored in some order
  - **Hash indices:** search keys are distributed hopefully uniformly across "buckets" using a "function"

Ordered Index

The records in the indexed file may themselves be stored in some sorted order

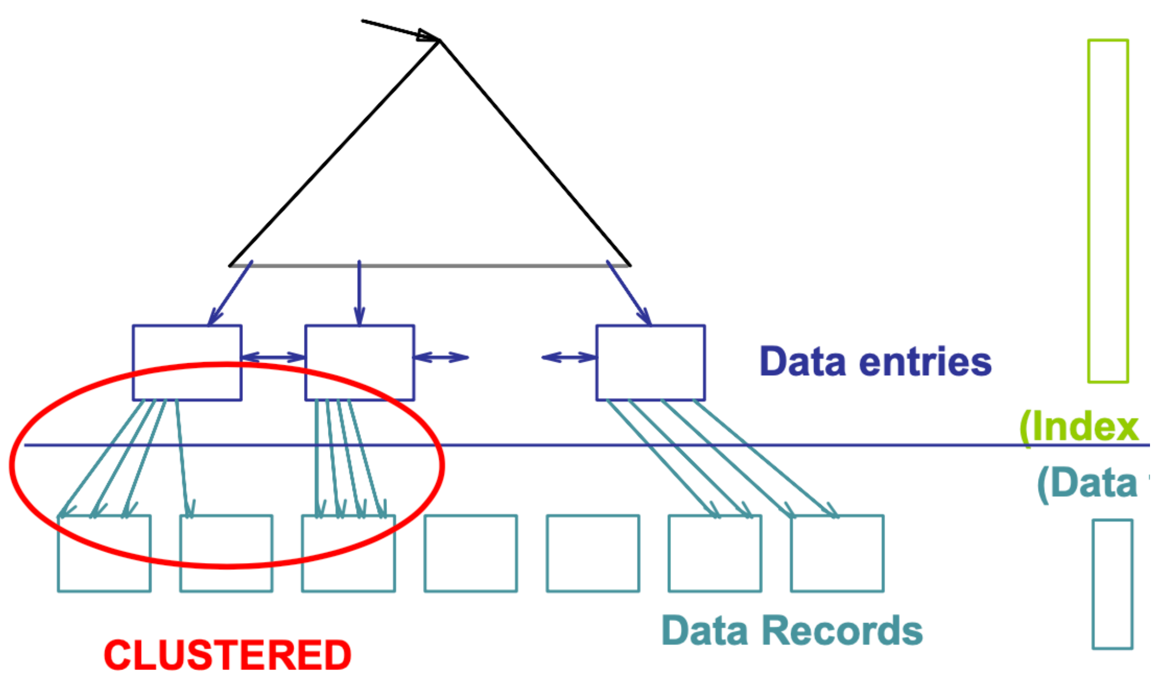
- **Clustering index/ Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - The search key of a primary index is usually but not necessarily the primary key

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	

Ordered Index

The records in the indexed file may themselves be stored in some sorted order

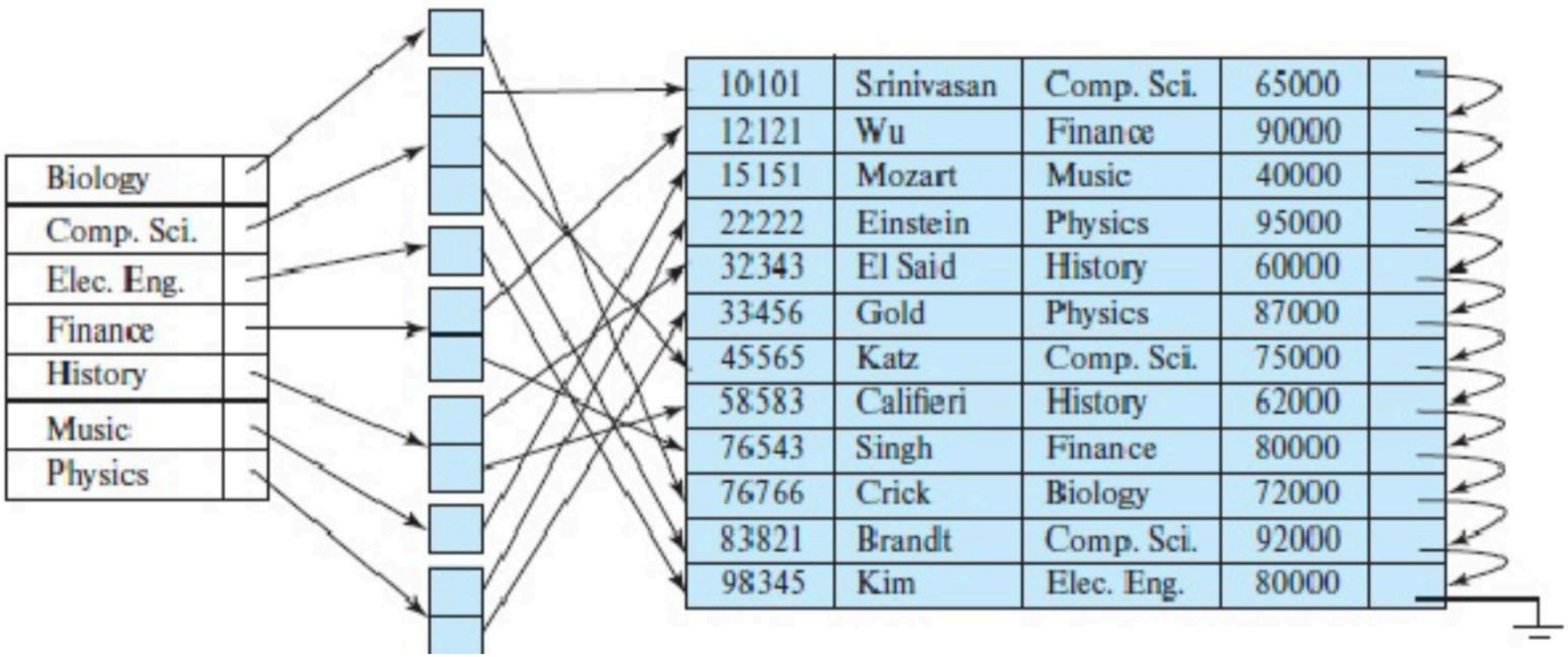
- **Clustering index/ Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file



Ordered Index

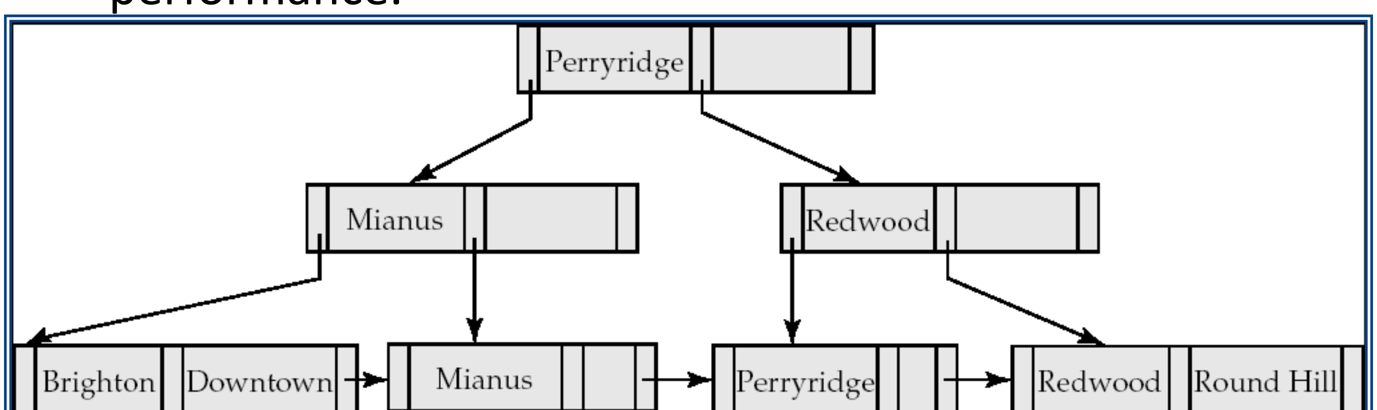
- **Non-clustering index/ Secondary index**: an index whose search key specifies an order different from the sequential order of the file

Secondary indices improve the performance of queries that use keys other than the search key of the clustering index.



The most popular in DBMS: B+trees

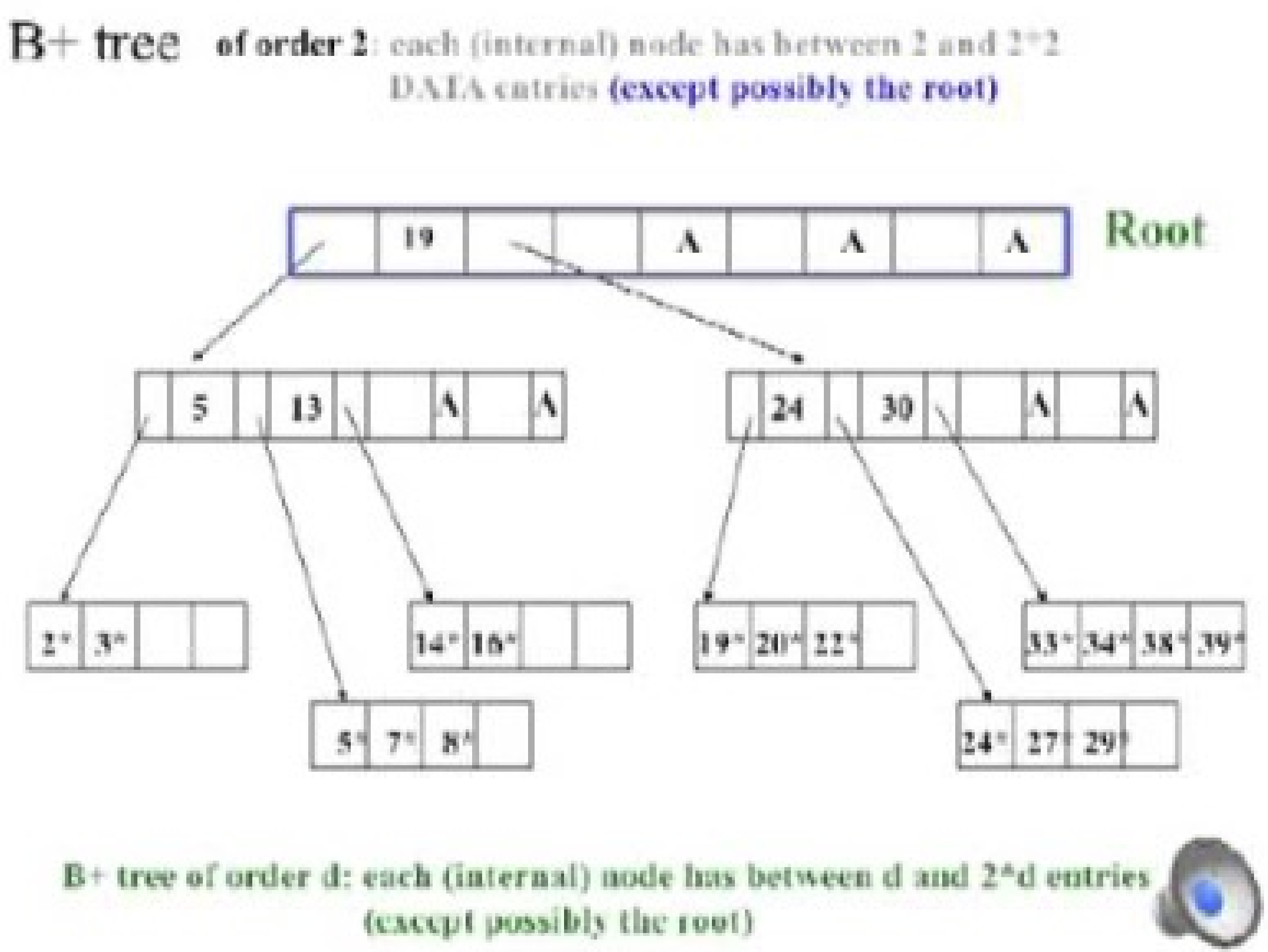
- Why need them:
  - Keeping files in order for fast search ultimately degrades as file grows, since many overflow blocks get created.
  - So binary search on ordered files cannot be done.
  - Periodic reorganization of entire file is required to achieve this.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.



B+trees Contd

- Similar to Binary tree in many aspects but the fan out is much higher
- Disadvantage of B^+-trees:
  - Extra insertion and deletion overhead and space overhead
- Advantages of B^+-trees outweigh disadvantages for DBMSs
  - B^+-trees are used extensively

B+tree structure



Video link: https://youtu.be/CYKRMz8yzVU

How is a B+tree defined?

- It is similar to a binary tree in concept but with a fan out that is defined through a number n
- All paths from root to leaf are of the same length (depth)
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- 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.

A Single Node

- Typical node

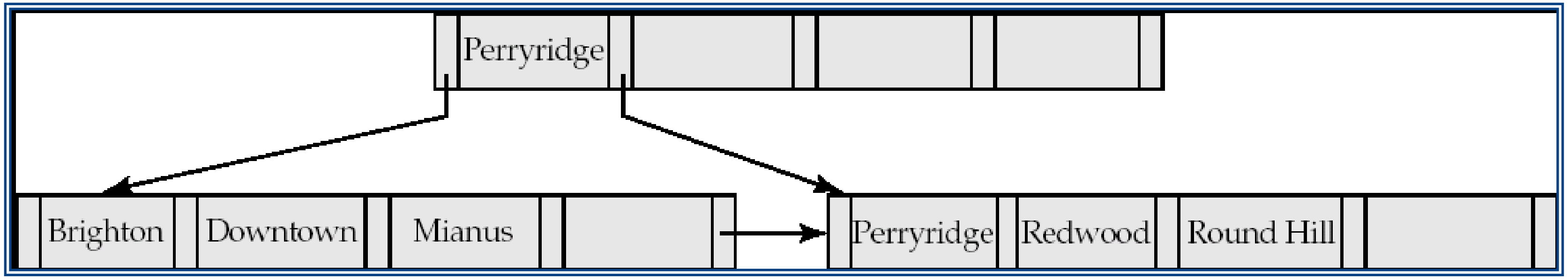
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

  - 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

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

NOTE: Most of the higher level nodes of a B+tree would be in main memory already!

Example B+tree



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

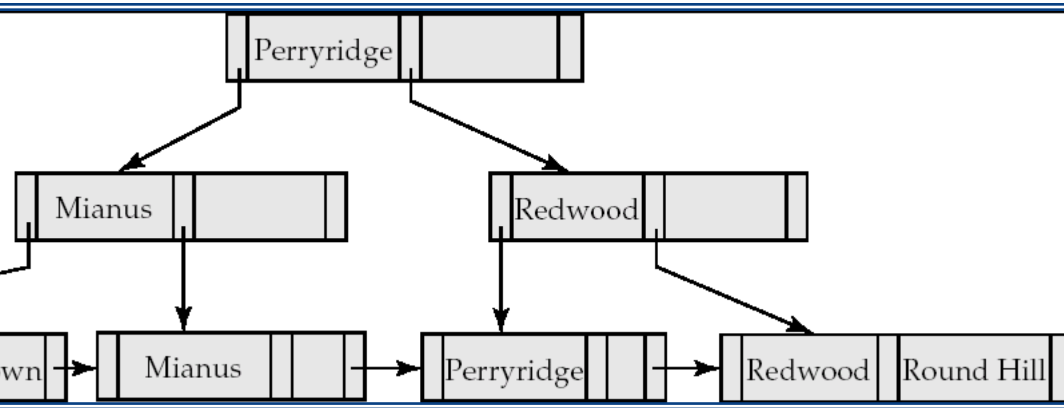
- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).
- Root must have at least 2 children.

How to run a query fast then?

- Finding all records with a search-key value of *k.*
  1. *N=root initially*
  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 $N = P_n$. Follow pointer.

     Until N is a leaf node
  3. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  4. Else no record with search-key value *k* exists.

Example



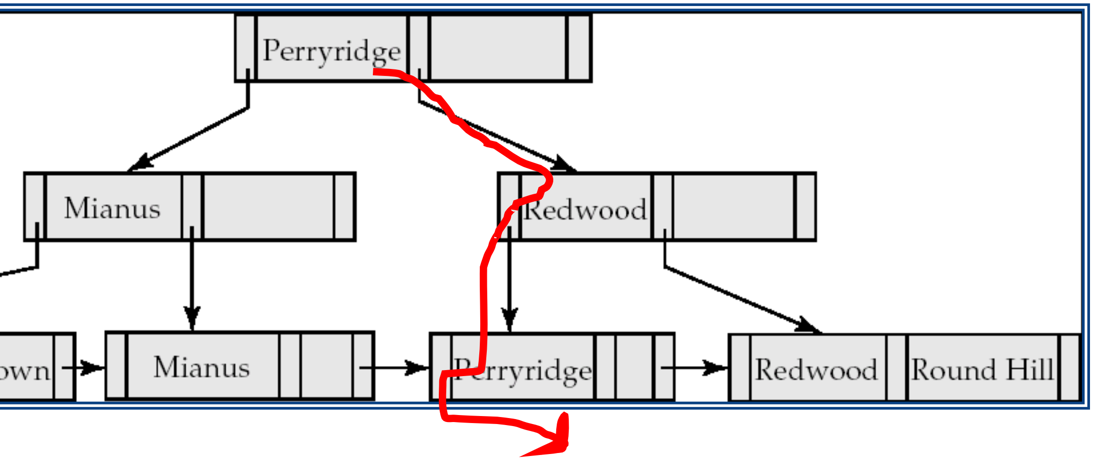
Recall: running a query on B+trees?

- Finding all records with a search-key value of *k.*
  1. *N=root initially*
  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 $N = P_n$. Follow pointer.

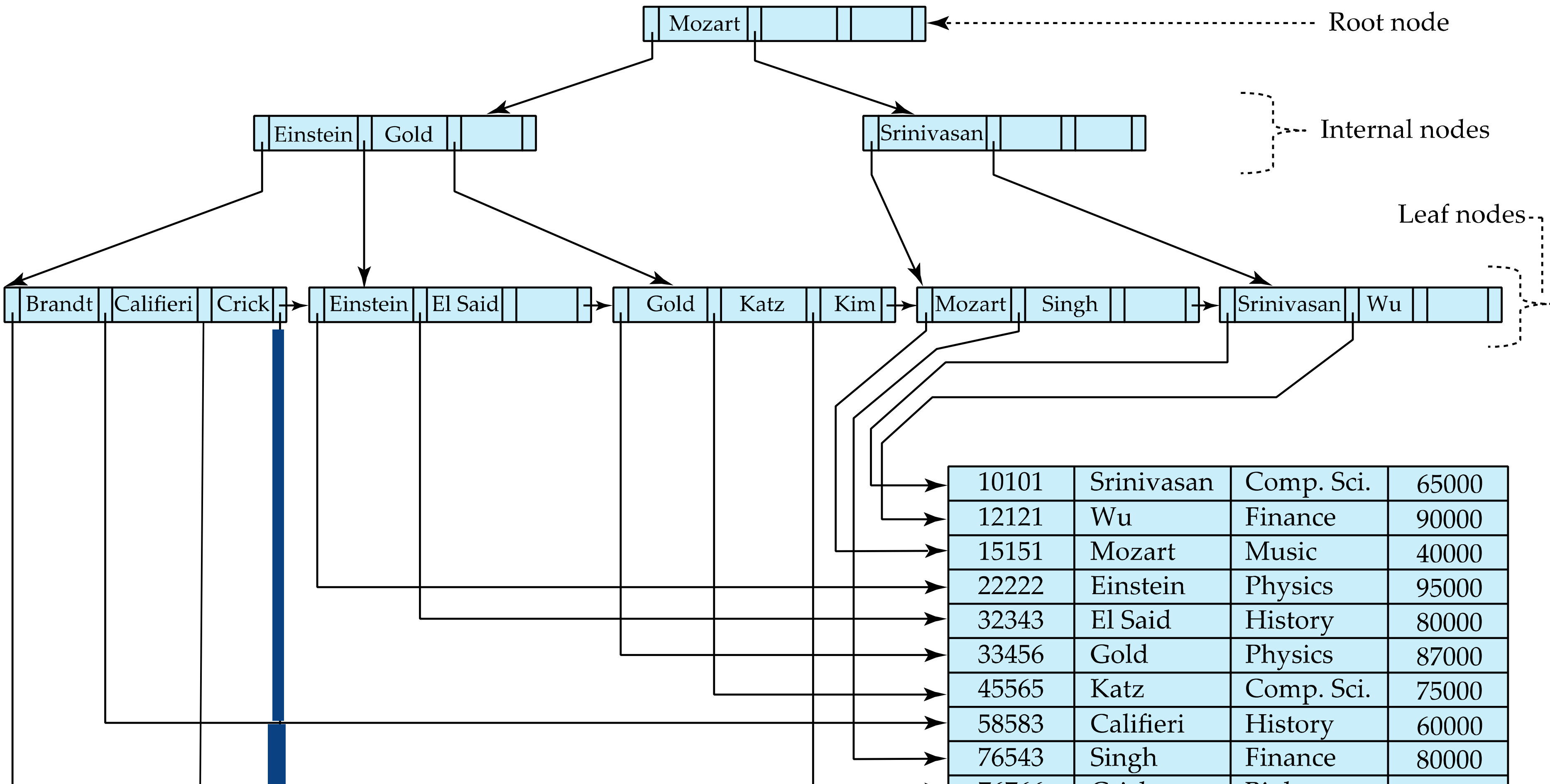
     Until *N* is a leaf node
  3. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  4. Else no record with search-key value *k* exists.

Example

Perryridge



A full view of a B⁺-Tree

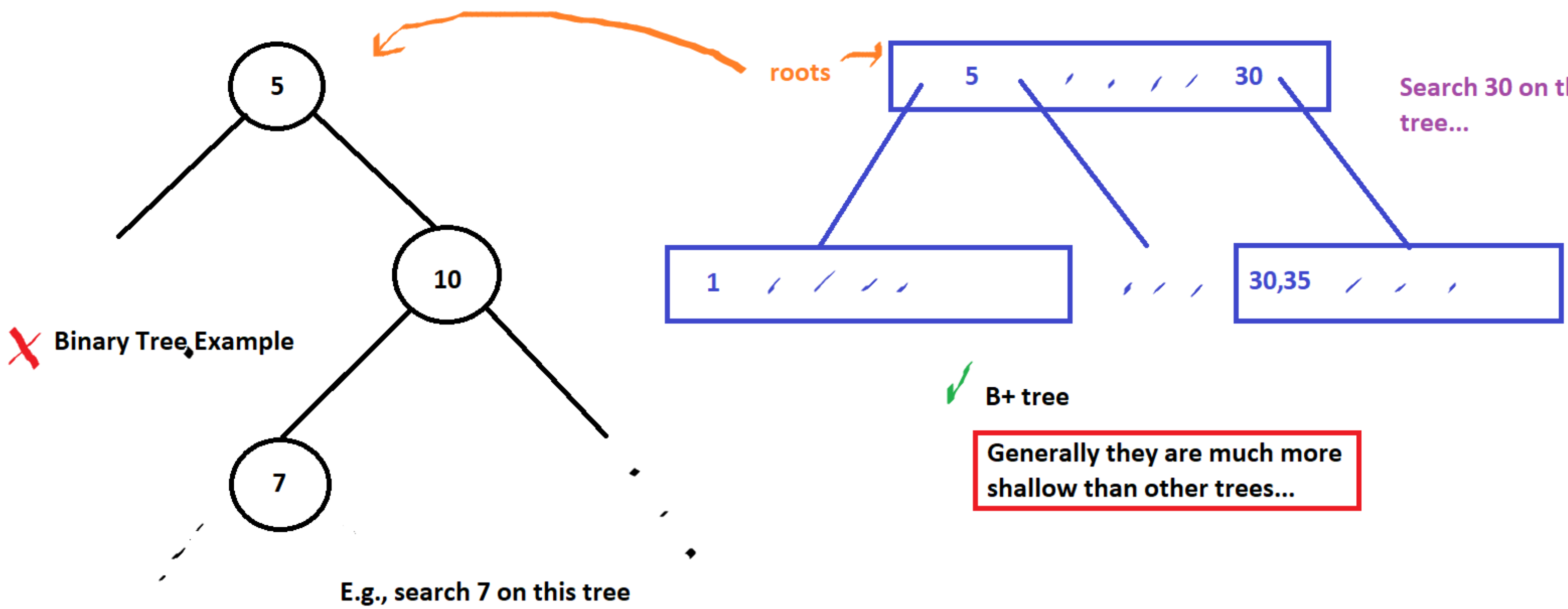


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Metrics

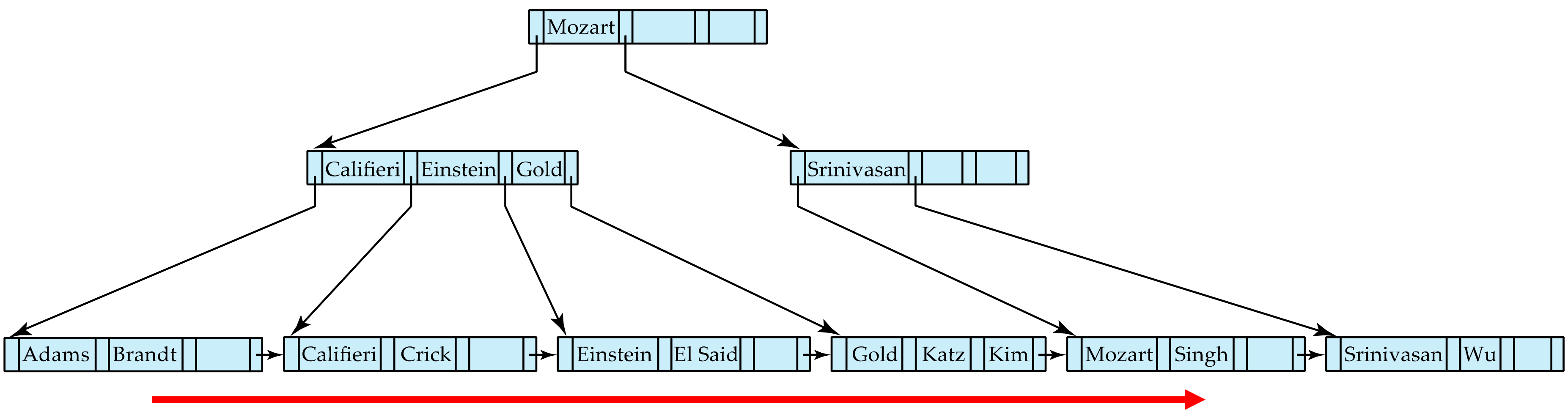
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ and **it would be balanced**
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and ***n* is typically around 100** (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - $log_{50}(1{,}000{,}000)$ = **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

A Quick Visual Comparison

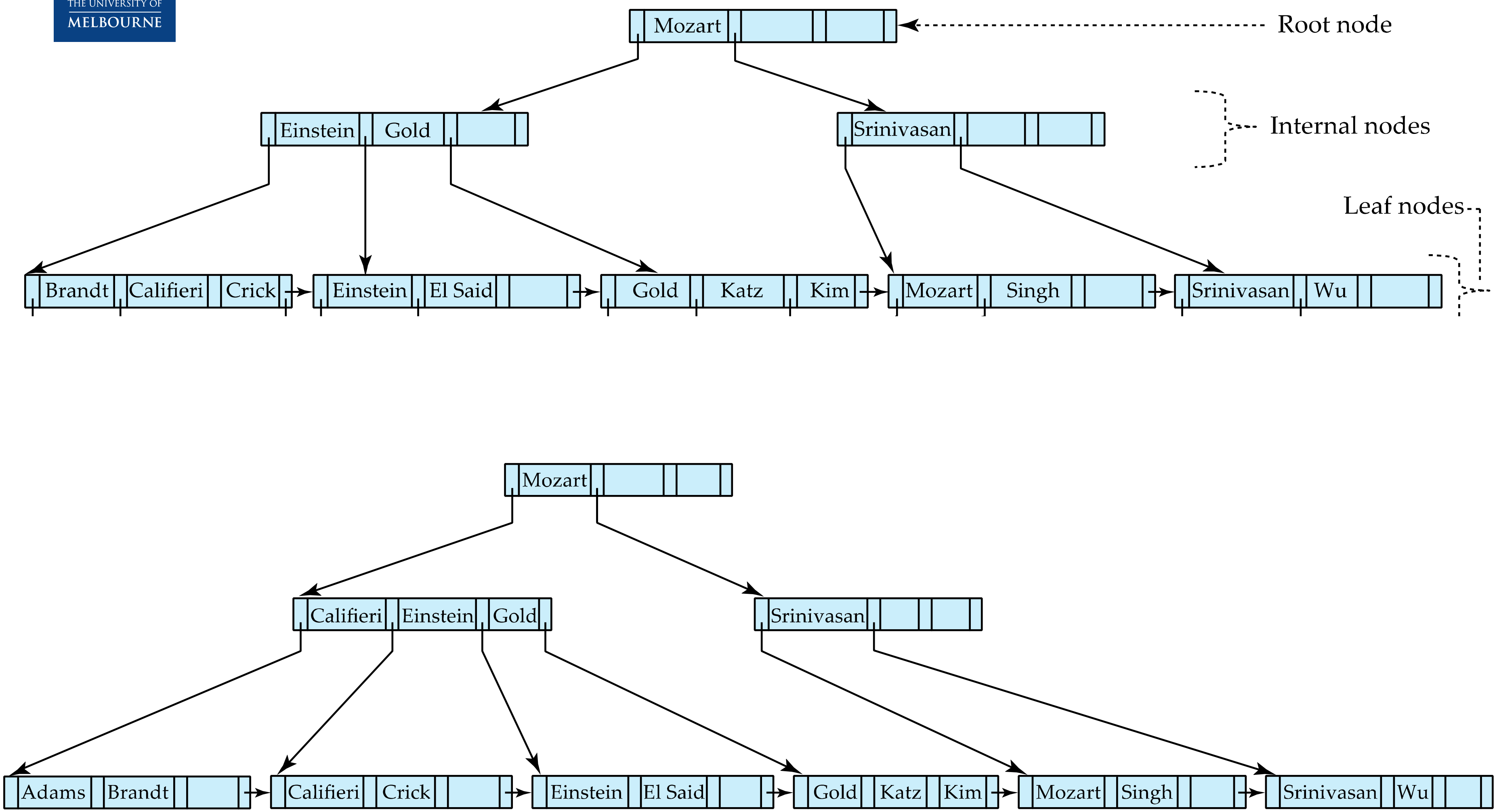


Range Queries on B$^+$-Trees

Range queries find all records with search key values in a given range

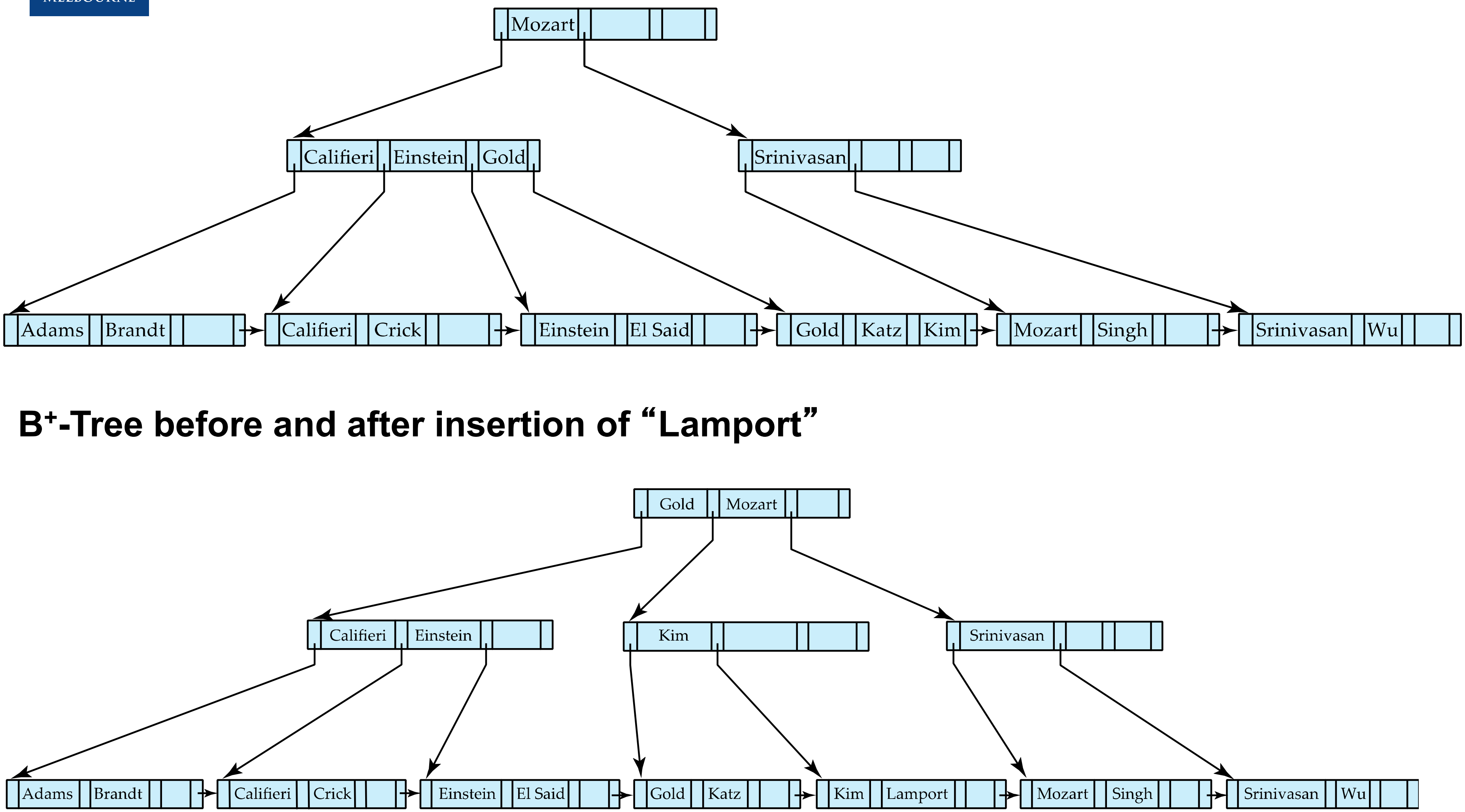


B+-Tree Insertion



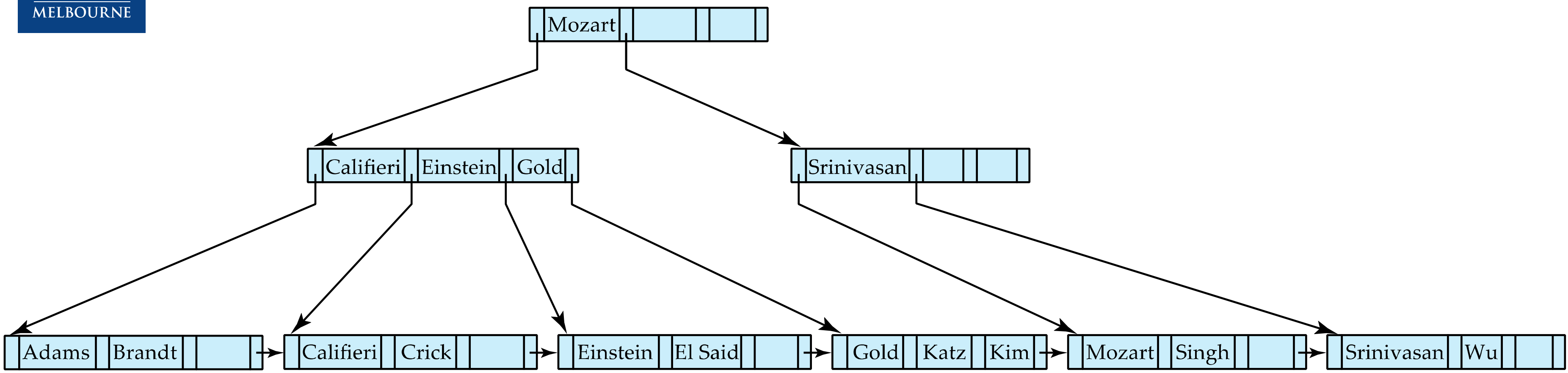
B+-Tree before and after insertion of "Adams"

B+-Tree Insertion

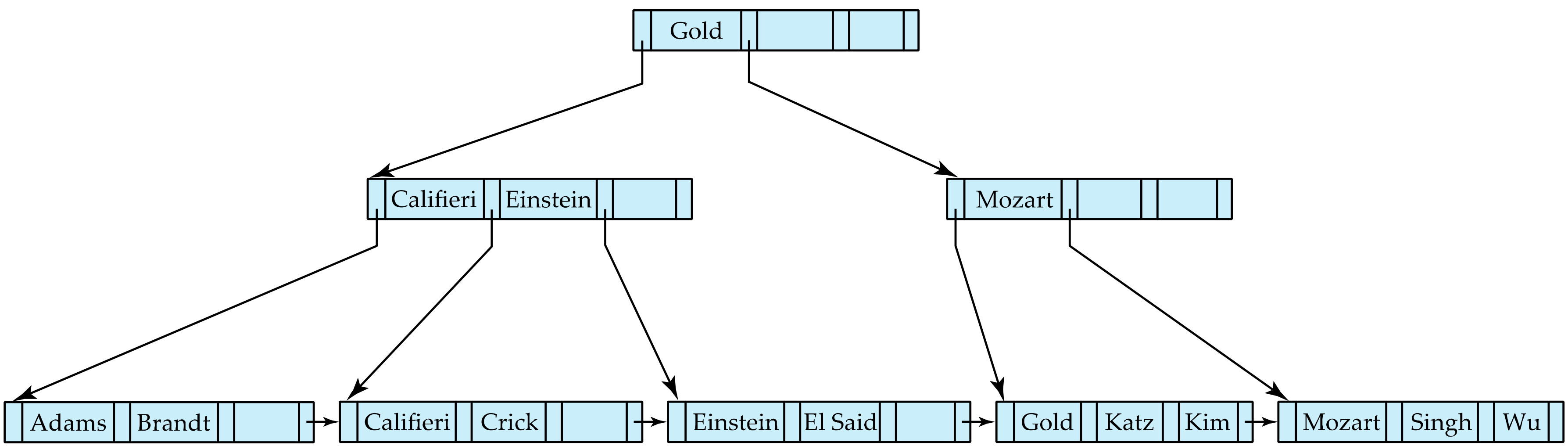


B+-Tree before and after insertion of "Lamport"

Examples of B+-Tree Deletion

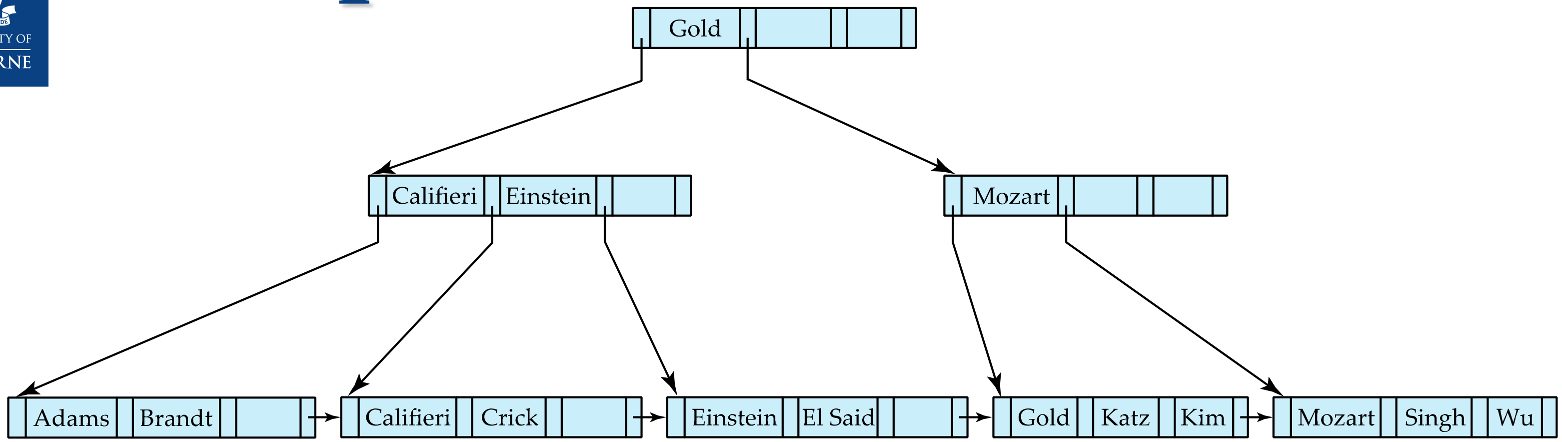


Before and after deleting "Srinivasan"

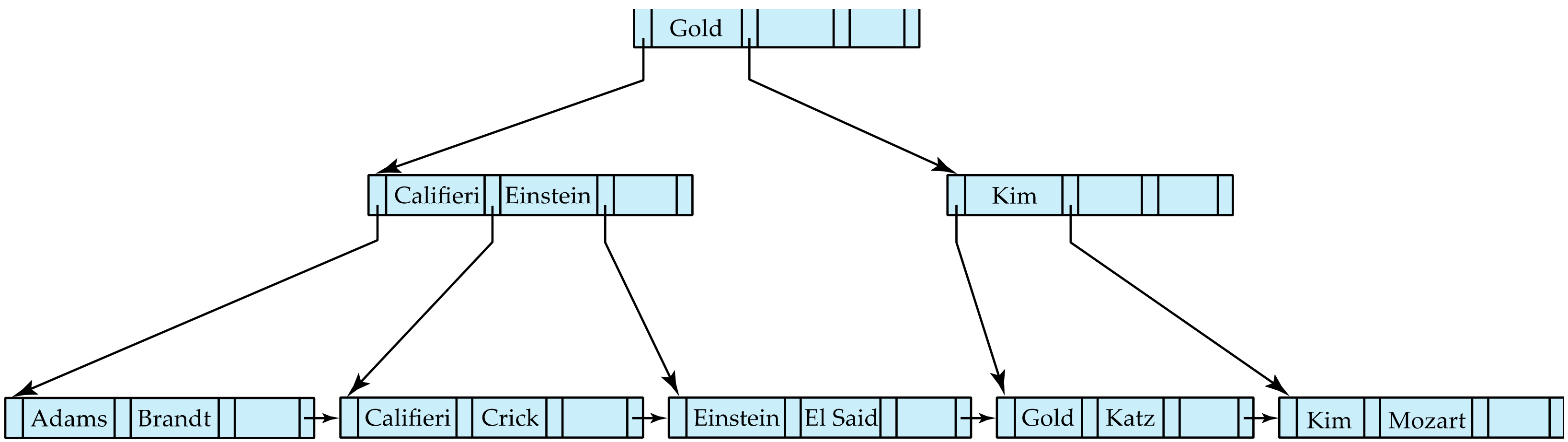


Deleting "Srinivasan" causes **merging** of under-full leaves

Examples of B+-Tree Deletion



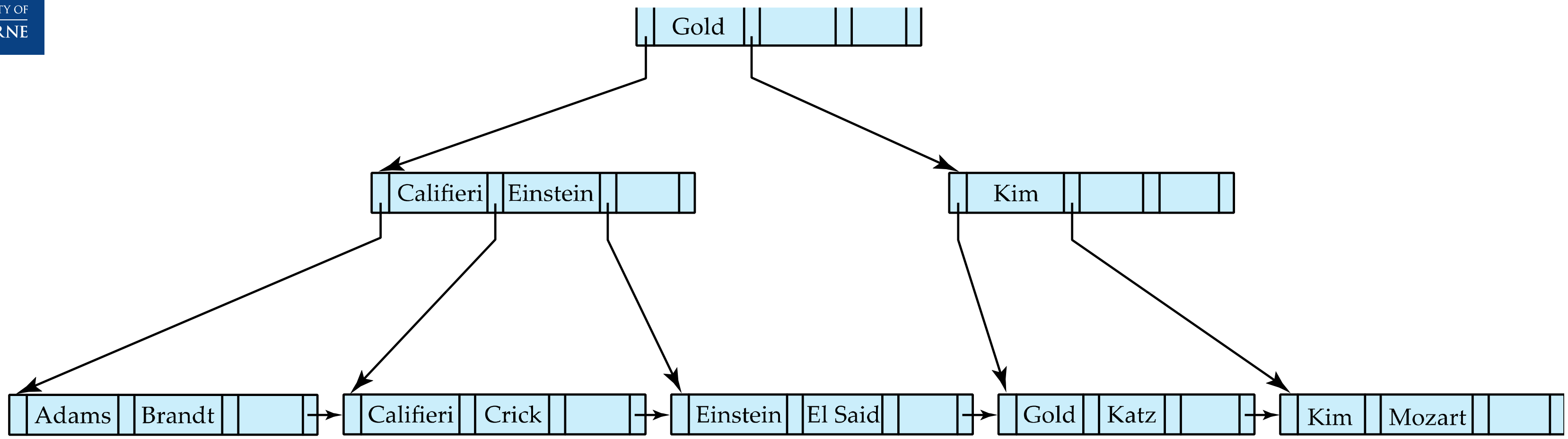
Before and after deleting "Singh" and "Wu"



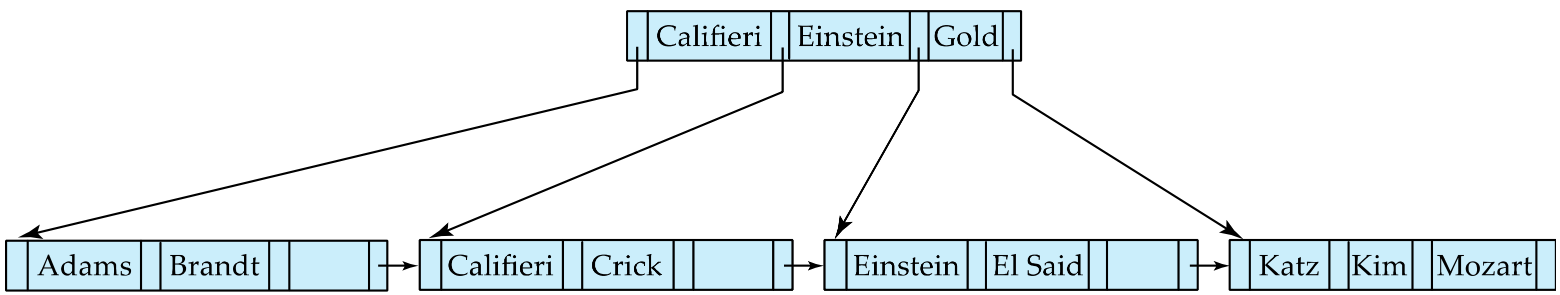
Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling

Search-key value in the parent changes as a result

Example of B+-tree Deletion



Before and after deletion of “Gold”



Node with Gold and Katz became underfull, and was merged with its sibling

Parent node becomes underfull, and is merged with its sibling

- Value separating two nodes (at the parent) is pulled down when merging

Root node then has only one child, and is deleted

B^+-Tree File Organization

B^+-Tree File Organization:

- Leaf nodes in a B^+-tree file organization store records, instead of pointers to children
- Helps keep data records clustered (ordered) even when there are insertions/deletions/updates
- Insertion and deletion or records are handled in the same way as insertion and deletion of entries in a B^+-tree index.