Database Management Syatems

Indexing and Hashing

Topics

- Indexing
  - Linear indexing
  - Tree indexing
- B+Tree
- Multiple Indexing
- Hashing

Indexing

Motivation

- Some file operations are very slow
- Example: Reading all records in order of an attribute may take several hours in a large file
- Sorting files can speed up file operations but still there are some problems

Problems with Pile Files

- Finding a record (T_F) Finding next record in an order (T_N) and Deleting a record (T_D) are very slow.
- In these operation, on average, half of the records are read because the records are not stored with any order.
- As a solution we may consider putting records in some order (sorting the file). But sorting files does not solve the problem.

Problems with Sorted Sequential Files

- Sorting large files need extra memory and is slow

- Files will not remain sorted after new insertions

- Search using binary search needs $Log_2 n$ file access which is slow in large files.
  - Example: for a file with 16,000,000 records, 24 file access is needed

Problems with Sorted Sequential Files

- Yet a more important problem with sorted files is that the file is sorted according to only one attribute
- Searches with other attributes need other copies of the file
- Multiple copies of records may cause inconsistency in data

Problems with Sorted Sequential Files

ID	Name
4567	David
2345	John
5678	Ema
1234	Stephan
3456	Sophia

Pile File

ID	Name
1234	Stephan
2345	John
3456	Sophia
4567	David
5678	Ema

Sorted by ID

ID	Name
4567	David
5678	Ema
2345	John
3456	Sophia
1234	Stephan

Sorted by Name

Indexing

- Indexes are lookup tables for finding records quickly
- The simplest index is a list in order of the key values (linear indexing)

Case 1: Linear Indexing

- Linear indexing is a sorted list of keys and record locations
- Search is done in index list before going to the main file
- Linear indexing is suitable for small files

Example: Linear Indexing

ID	Name
4567	David
2345	John
5678	Ema
1234	Stephan
3456	Sophia

Data File

ID (Key)	Location
1234	3
2345	1
3456	4
4567	0
5678	2

Index File

Searching in an Indexed File

- Given a key value do:
  - Search the index list using binary search
  - Getting the location go to the block and read the record (s+r+btt)

- If the index list is in the memory, the search is fast

Insertion into an Indexed File

- Insertion is done at the end of the data file
- Add new key value to the index file. Then the index is updated to be sorted again

Example: Insertion into Indexed Files

ID	Name
4567	David
2345	John
5678	Ema
1234	Stephan
3456	Sophia
3825	Emily

Data File

ID (Key)	Location
1234	3
2345	1
3456	4
3825	5
4567	0
5678	2

Shift Down

Index File

Case 2: Tree Indexing

- If the index list is larger than the memory, the search should be done in the file
- Searching the index file will be slow if it is a binary search ($\text{Log}_2 n$)
- Tree indexes are used for faster search

B+Trees

- A tree with
  - Several children at each node
  - All leaves are at the same level

Nodes of a B+Tree

1. **Internal Nodes**

   Have key values and pointers to child nodes

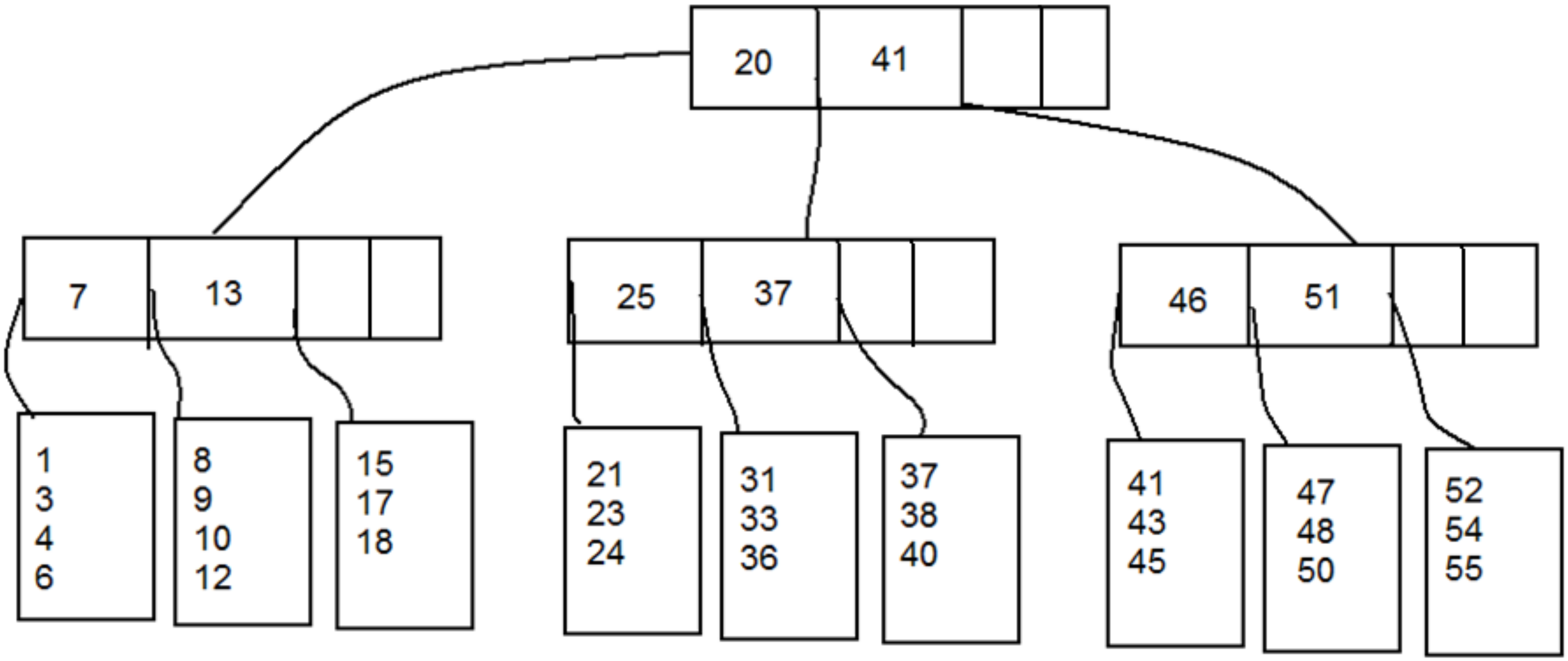
2. **Leaf nodes**
   1. Records if it is a primary index
   2. Keys and record locations if it is a secondary index

All nodes except the root should be at least half full

Structure of a B+Tree

- If the attribute value is less than a key in internal node, it is stored at its left side leaf node
- Otherwise the attribute is compared with the next key in the internal node.

Example B+Tree



Operations on a B+Tree

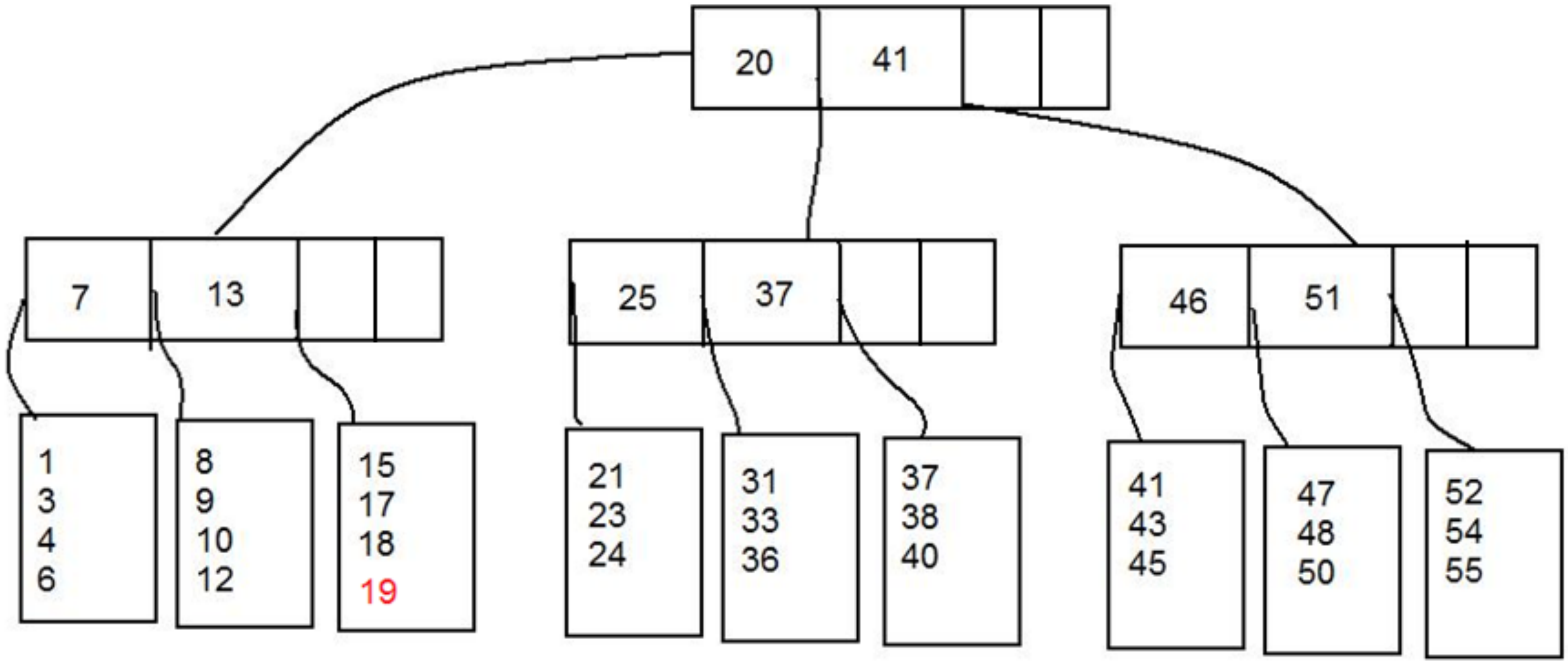
- Insertion
  - Insertion is done from the leaf nodes and the tree is updated. Nodes may split
- Deletion
  - Deletion is done from leaf nodes.
  - Nodes may be merged after deletion

Insertion

- Algorithm
  - Using the key value of the data item, search the tree to a leaf node.
  - Insert the new data if the leaf node has enough space
  - Split the leaf node if there is no place to insert new data
  - Update tree

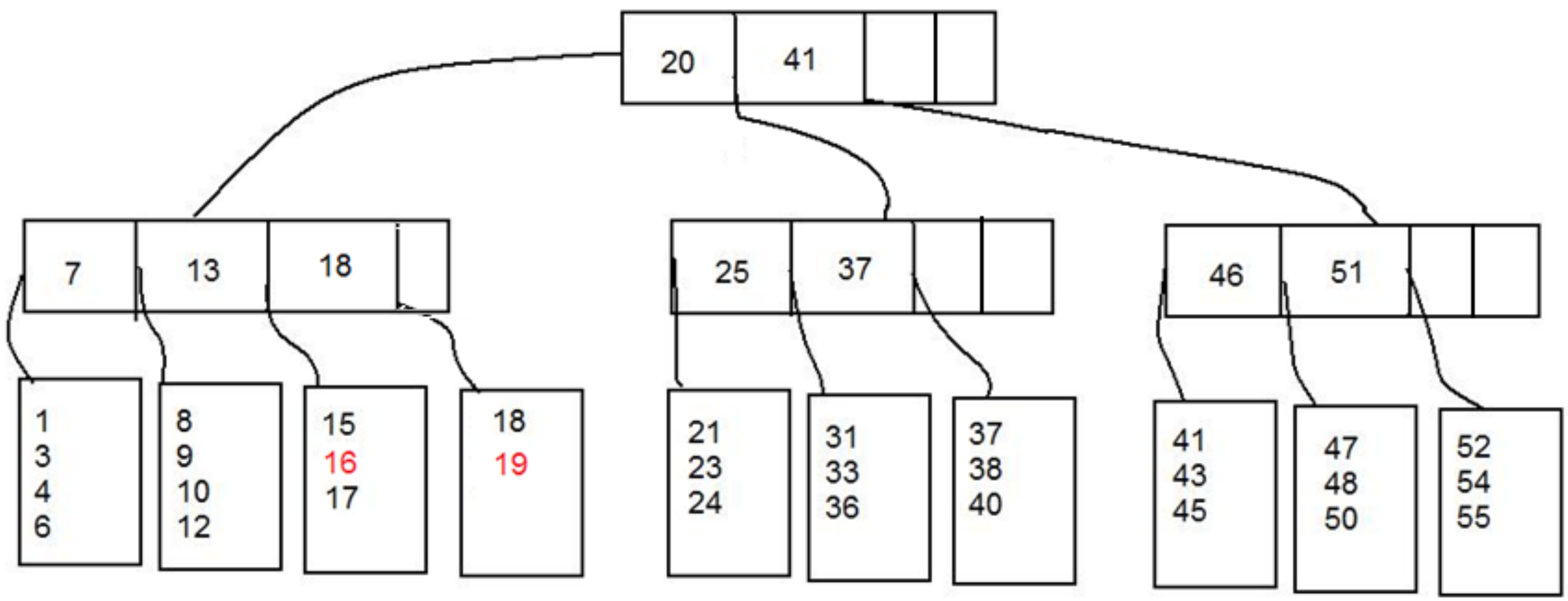
  Note: All nodes except the root should be at least half full

Insertion: Example



Insert 19

Insertion: Example

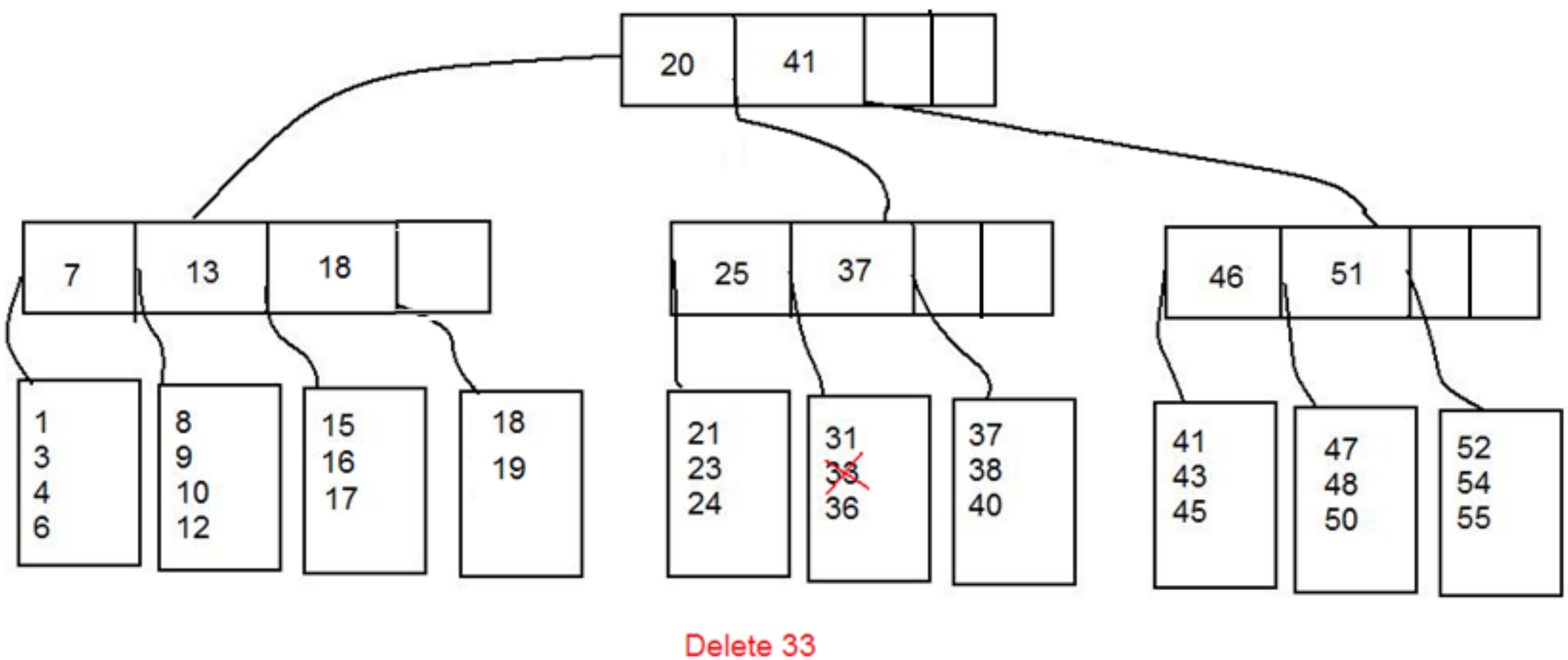


Insert 16

Deletion from a B+Tree

- Algorithm
  - Find the leaf node containing the data item
  - Remove the data item

Example: Deletion



Hashing

Hashing

- Motivation: The number of file access in an indexed file is as many as the tree height (3 or 4 for example)

- Hashing methods provide a quick access to the records (1 or 2 file access)

Definitions

- Hash function: A function that returns the location of a record given its key value.
  - Example: f(25)=1, f(1)=3

5	A
25	K
27	E
1	R
7	G
3	H
19	Z

Hash Function

- Hash functions do not use any list or index.
- Therefore, hash functions include no file access.
- After finding the record location using a hash function, we go to the file and read the record.

How Do Hash Functions Work?

- Hash functions get a key value and find the record location by doing some arithmetic on it.

- Generally hash functions find the remainder of the key value by a constant **N**, then multiply it by a constant like **a**, and add another constant like **b**

- E.g. **Hash(key) = (key MOD N)*a+b**
- E.g. N=10, a=2, b=3 ➔ Hash(25)=5*2+3=13

Definition

- **Hash table**: The data file having the records is called a hash table.

- Hash tables are created using the location values returned from the hash functions.

Creating Hash Table

- Compute the location of the record using hash function.
- Put the record at the position returned from the hash function.

Example Hash Table

- Use Key Mod 10 to create the hash table.

12	A
25	K
14	E
1	R
7	G
3	H
19	Z
36	N

Data File

1	R
12	A
3	H
14	E
25	K
36	N
7	G
19	Z

Hash Table

Collision Problem

- The hash function may generate the same values for different keys.

  Example: Keys 12 and 32 generate same results with hash function :: key mod 10

- This is called the collision problem

Solutions for the Collision Problem

1. Bucketing: Use buckets as large as n records at each hash table entry
2. Chaining: Records with the same hash values are chained in a linked list using an overflow area or dynamic links

Bucketing

- A Bucket is a group of records.
- Each entry in the hash table is a bucket.
- Therefore each entry can hold several records.
- It is difficult to decide about the bucket size.
- Large buckets are wasteful.

Bucketing

1	R	11	T
12	M	22	B
3	H		
14	U		
5	A	25	K

Chaining

- If the number of records in collision is larger than the bucket size, bucketing fails.
- This problem is because the bucket size is fixed (static)
- Dynamic buckets, which grow with the number of records in collision, are possible.
- Dynamic buckets are created using linked lists

Chaining using Overflow Area

1	R	
22	B	
3	H	
14	U	
5	A	
25	K	
11	T	
12	M	

Combining Bucketing and Chaining

- Bucketing can be used with chaining for better performance.
- If a bucket is the same size of a block, file I/O operations will be more efficient (the unit of I/O operation is a block)
- The buckets are connected using linked lists if collisions happens.

Sample Data

Student ID	Student Name	Department
132	A	CENG
141	B	CENG
155	C	ECE
176	D	CENG
162	A	ECE
134	E	IE
145	H	IE
112	B	CENG
114	T	CENG
125	H	ECE
133	U	ECE
147	P	CENG
118	M	IE
129	F	CENG
119	R	IE

Bucket Size and Hash Function

- For this example we used
  - Student ID as key value
  - Key MOD 10 as hash function
  - Bucket size = 2

Hash Table

141	B	CENG
132	A	CENG
162	A	ECE
133	U	ECE
134	E	IE
114	T	CENG
155	C	ECE
145	H	IE
176	D	CENG
147	P	CENG
118	M	IE
129	F	CENG
119	R	IE

112	B	CENG

125	H	ECE

Questions?