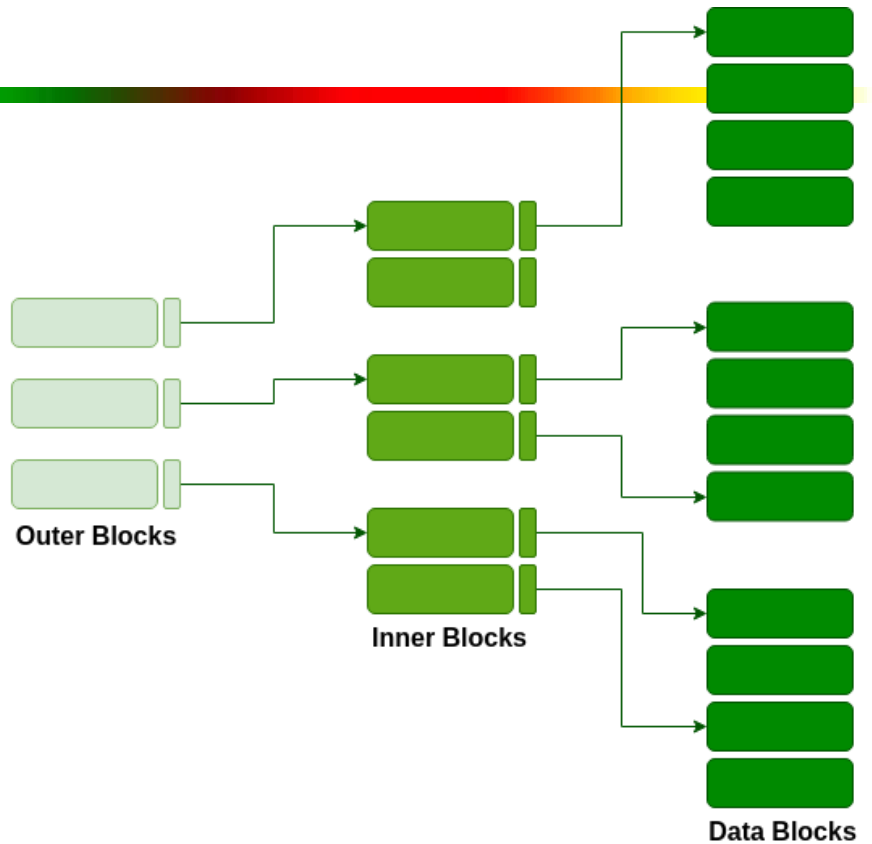
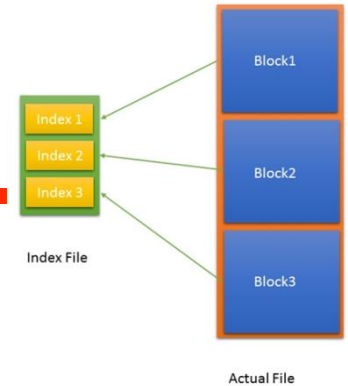


INDEXING



Courtesy:
Silberschatz, Korth and Sudarshan, Database System Concepts
Anjali Jivani

WHY INDEXING?



- Access structures to speed up retrieval of records
- Indexes can be of two types (1) ordered (2) hashed
 1. Ordered indices based on sorted values of the indexing field.
 2. Hashed indices based on the hashing function which directly gives the relative address of the record
- 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 time
- Deletion time
- Space overhead

ORDERED INDICES



Ordered Indices:

- Has a search key associated with it which is stored in sorted order
- The records in the data file itself are sometimes stored in a sorted order
 - The attribute on which the file is sorted is known as its **clustering index** (also called **the primary index**)
 - The primary index need not be the primary key though it is preferred
 - There can be **only one clustering index**
 - The indices other than clustering – attributes on which file is not stored in a sorted fashion are called non-clustering indices or secondary indices (can have **any no. of secondary indices**)

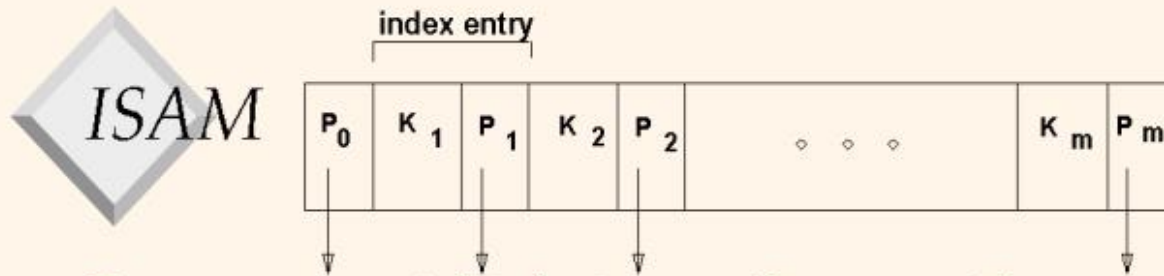
ISAM



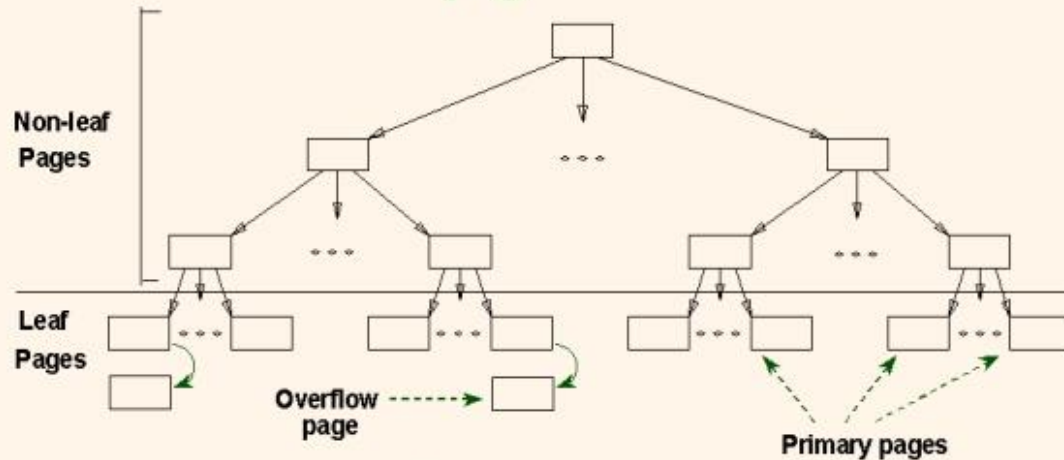
ISAM (Indexed Sequential Access Method)

- It is the process/software that handles the data retrieval in index-sequential files.
- It is one of the oldest schemes used in database systems.
- It is a way or method of indexing data files on a search key.
- To implement ISAM it is to be loaded in the memory first. (COBOL supported ISAM technology).
- The files which are indexed using ISAM are known as indexed sequential files.
- Data files which have a clustering index are called index-sequential files.
- These files are stored in a sequential order of the search key and have an index file (containing index entries) like –
<search key, address>.
- The address here refers to the physical address of the record with the search key value.

ISAM



❖ Repeat sequential indexing until sequential index fits on one page.



❖ Leaf pages contain data entries.

11

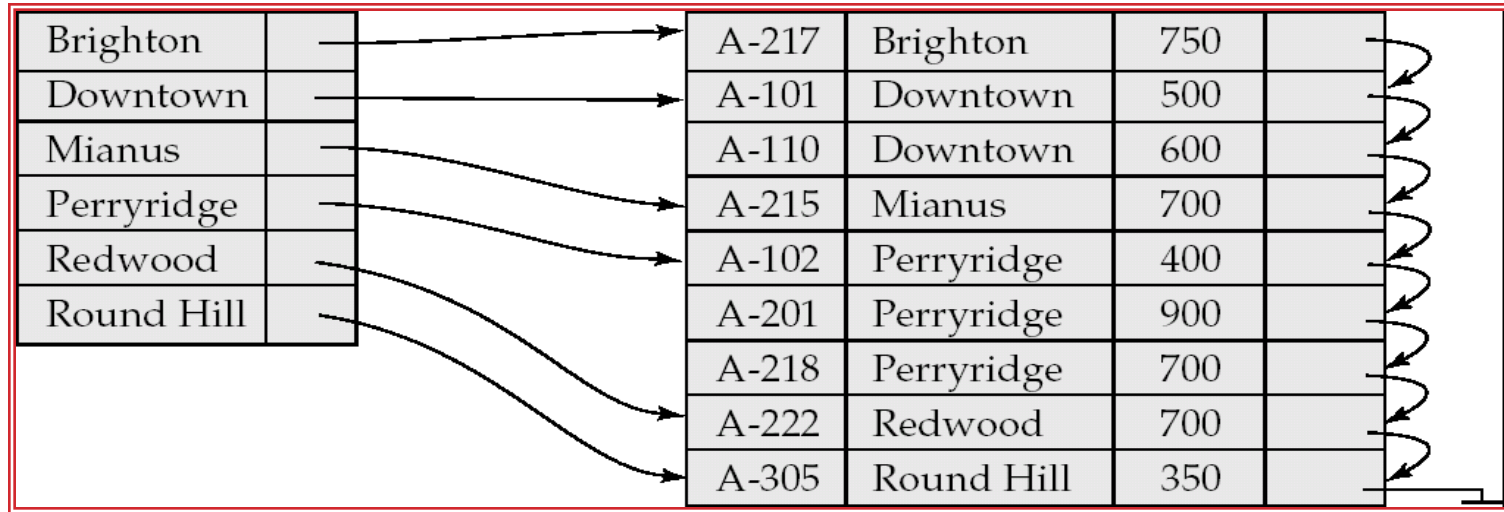
- The ISAM index looks like a B+ Tree structure.
- Unlike B+ tree however, ISAM is a static structure. The data records in the data file have to be sorted on some key field (primary key generally – not necessary) and then stored on the disk.
- These sorted records are then indexed using a multi-level index.

ISAM



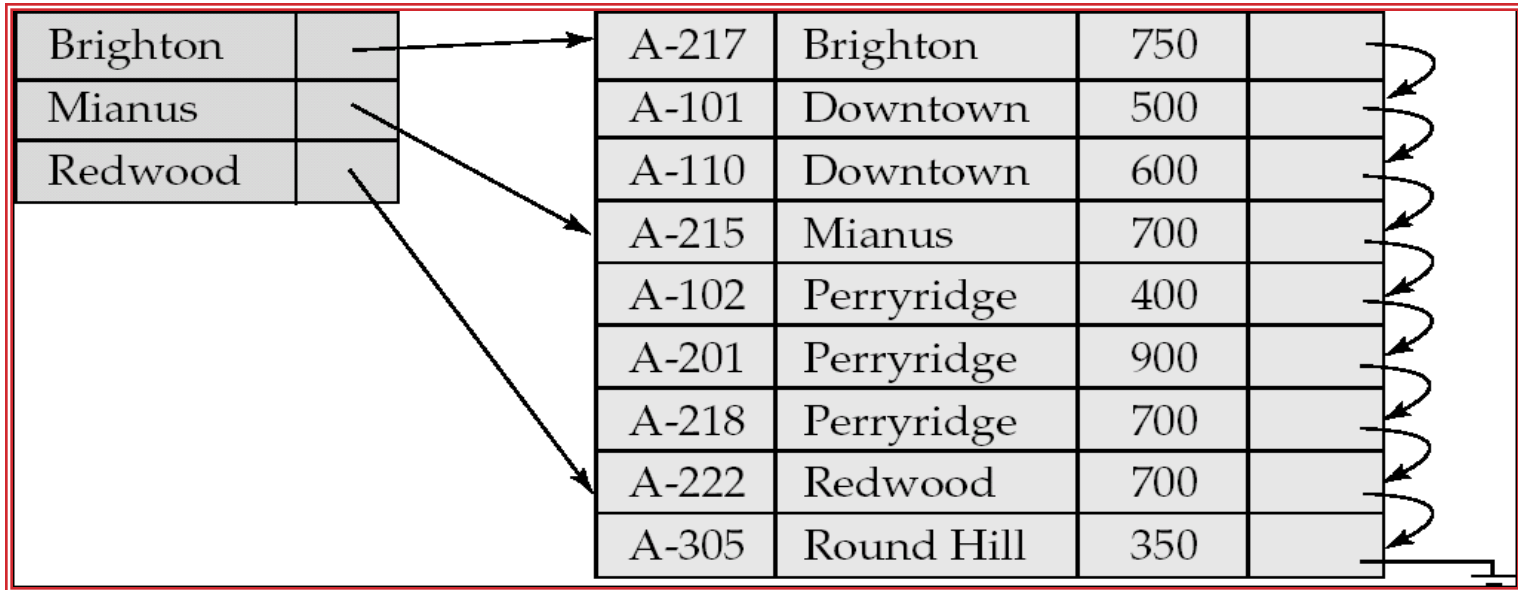
- For every insert in this file, if there is not enough space, an overflow page is added at the appropriate position to insert the record.
- When the overflow records increase, the speed of indexing and searching reduces. The data file is to be re-indexed in such cases.
- Re-indexing means all the data records are again sorted on the key and stored sequentially in the physical memory. After that the multi-level index is re-created.
- ISAM is for static files, B+ Trees is for dynamic files.
- There are three alternatives for index entry:
 1. Just the <search key> (in case of hashing)
 2. <search key, address> search key is unique i.e. rollno, empid, etc.
 3. <search key, list of addresses> search key is not unique i.e. salary, name, etc.

DENSE INDEX



- One entry in the index file for every distinct value of the search key.
- If the search key is not unique then the pointer points to the first record with that value. The rest of the records are stored sequentially after the first record if the file is index-sequential
- If the file is not index-sequential then one entry per record for all duplicate values of the search key also
- The index file itself can become too large – more than one block in memory
- Fast access possible

SPARSE INDEX



- An index entry for only few records
- Can be created only if the index is a clustering one – file is stored in sequential order of the search key
- To locate a record we need to find an index entry with a largest value that is less than or equal to the search key value we are looking for. Start at that record in the file and then sequential search to actually read the record we want
- Speed is slower than dense index but occupies less space

INDEX 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 is 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). (Mianus replaced by Perryridge)
 - If the next search-key value already has an index entry, the entry is deleted instead of being replaced. (Brighton, Downtown all three records deleted)

Dense Index

Brighton		A-217	Brighton	750	
Downtown		A-101	Downtown	500	
Mianus		A-110	Downtown	600	
Perryridge		A-215	Mianus	700	
Redwood		A-102	Perryridge	400	
Round Hill		A-201	Perryridge	900	
		A-218	Perryridge	700	
		A-222	Redwood	700	
		A-305	Round Hill	350	

Sparse Index

Brighton		A-217	Brighton	750	
Mianus		A-101	Downtown	500	
Redwood		A-110	Downtown	600	
		A-215	Mianus	700	
		A-102	Perryridge	400	
		A-201	Perryridge	900	
		A-218	Perryridge	700	
		A-222	Redwood	700	
		A-305	Round Hill	350	

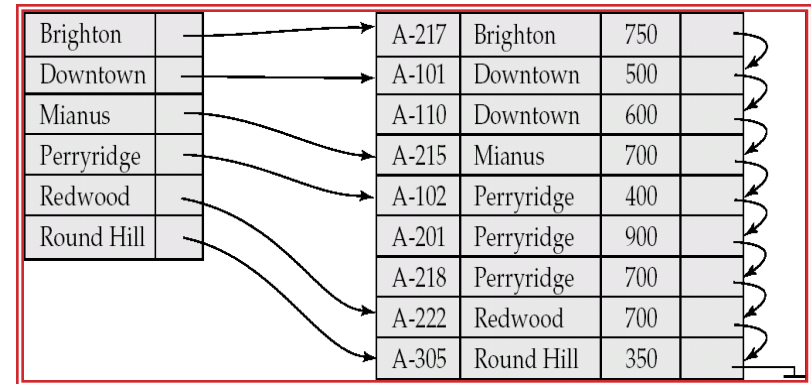
INDEX INSERTION

■ Single-level index insertion:

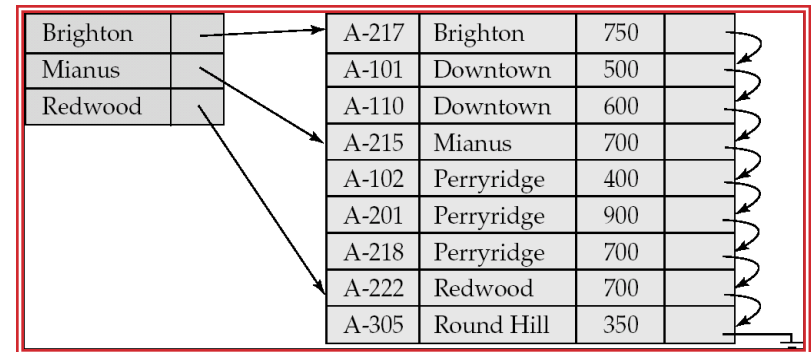
- Perform a lookup using the search-key value appearing in the record to be inserted.
- Dense indices – if the search-key value does not appear in the index, insert it.
- Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.

■ Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Dense Index

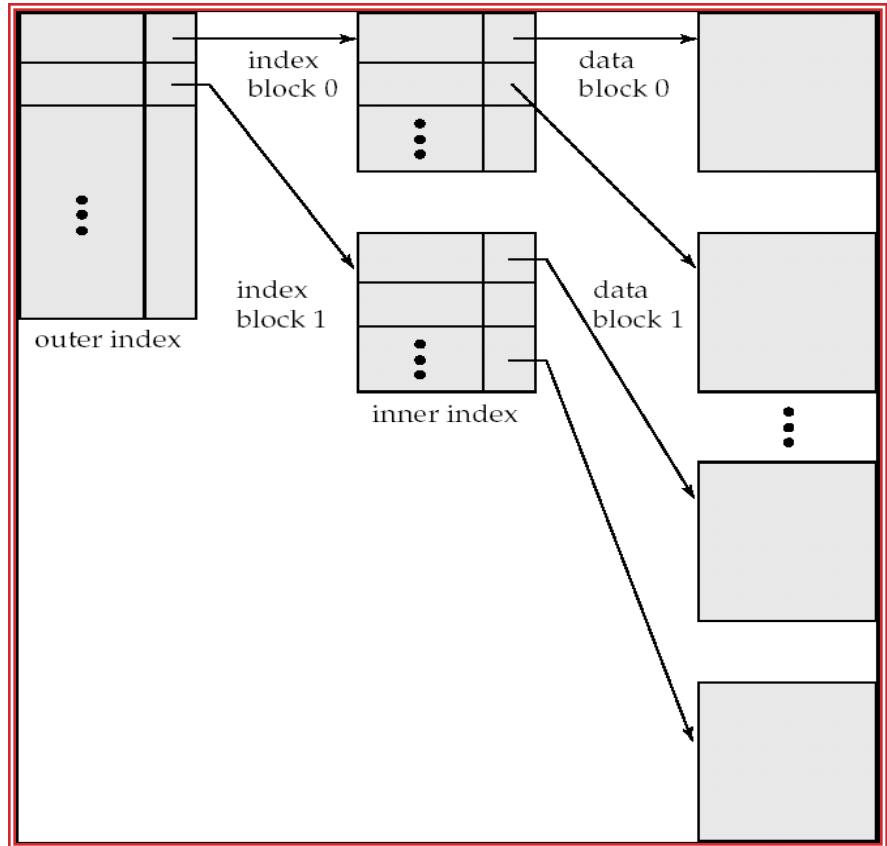


Sparse Index

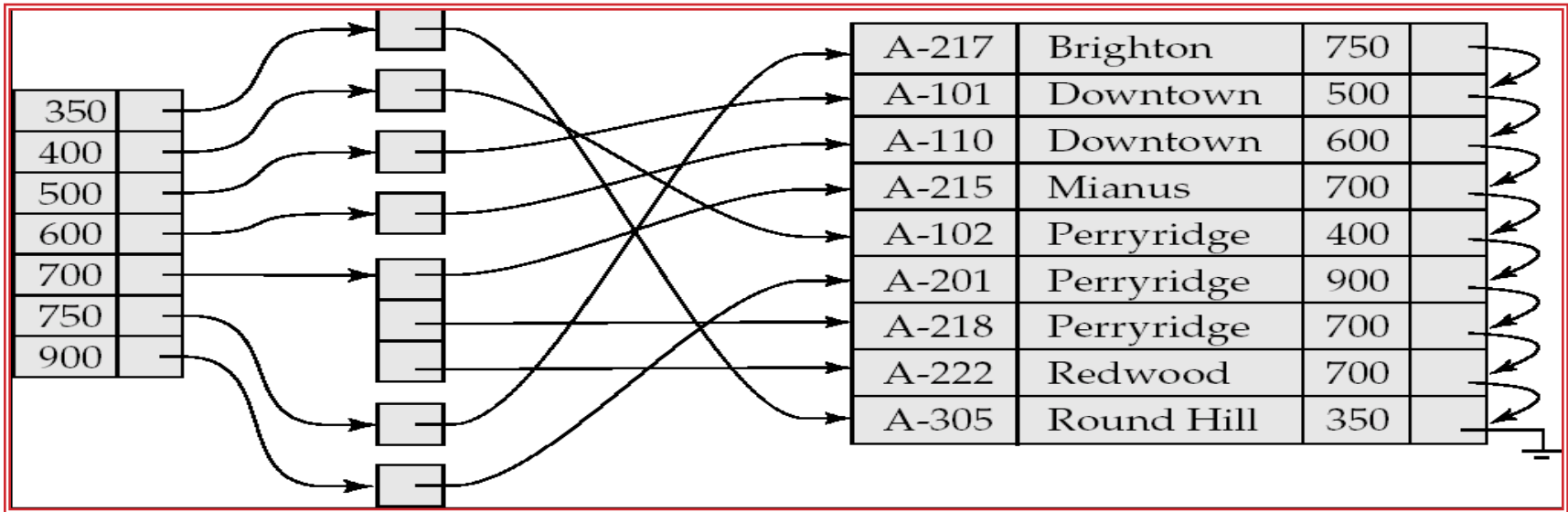


DENSE/SPARSE INDEX

- 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.
- Generally one entry points to one block for space optimization



SECONDARY INDEX



- Index created on non-distinct columns/attributes
- **Secondary indices have to be dense** as the actual file is not stored in sequential order of the secondary index (the above file is sequentially stored on the branchname field and not the balance field)
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Hence if a file consists of 'n' attributes then the number of secondary indices = $2^n - 1$

PRIMARY INDEX VERSUS SECONDARY INDEX

PRIMARY INDEX

Index on a set of fields that includes the unique primary key and is guaranteed not to contain duplicates

Requires the rows in data blocks to be ordered on the index key

There is only one primary index

There are no duplicates in the primary key

SECONDARY INDEX

Index that is not a primary index and may have duplicates

Does not have an impact on how the rows are actually organized in data blocks

There can be multiple secondary indexes

There can be duplicates in the secondary index

Visit www.PEDIAA.com

B Tree / B+ Tree an Alternative

B and B⁺-tree indices are an alternative to indexed-sequential files.

Advantages:

- B and B⁺ trees automatically reorganize themselves with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.
- Advantages of B⁺-trees outweigh disadvantages, and they are used extensively.

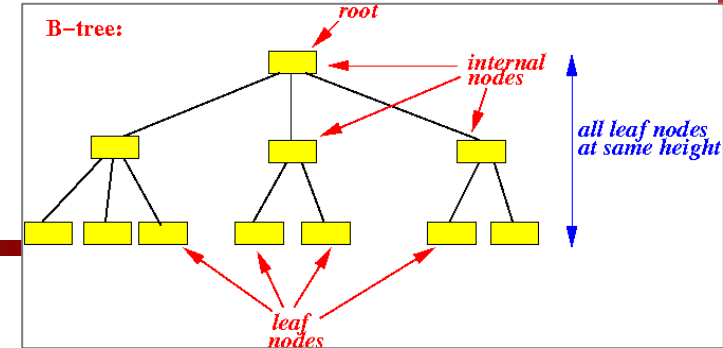
Disadvantages:

- Performance of indexed-sequential files degrades as file grows, since many overflow blocks get created.
- Periodic reorganization of entire file is required.
- Disadvantage of B and B⁺-trees is extra insertion and deletion overhead, space overhead.

B Tree / B+ Tree Comparison

B Trees	B+ Trees
Record pointers exist at all levels of the tree	Record pointers exist only at leaf level nodes
One node can store less keys as compared to B+ tree	One node can store more keys (as there are no record pointers in non-leaf nodes)
Height of tree for say n number of keys is high	Height of tree for n number is less than B tree
As height is more searching can take more time	Height is less and all record pointers are at leaf node
Keys are stored only once	Some keys may be duplicated
Sequential access becomes difficult	Sequential access is fast as all the leaf nodes are connected and have keys in sequence

B Trees



- A B-Tree is a **height balanced search tree**.
- A B-Tree of order m satisfies the following properties:
 - The root node has at least 2 children (if it is not empty or is not a leaf node)
 - All nodes other than root have at least $\text{ceil}(m/2)$ children (i.e. links) and $\text{ceil}(m/2)-1$ keys
- All leaf nodes are at the same level
- Structure of a non-leaf (internal) and leaf node in a B-Tree are same e.g. a 3-4 B-Tree (order 4)



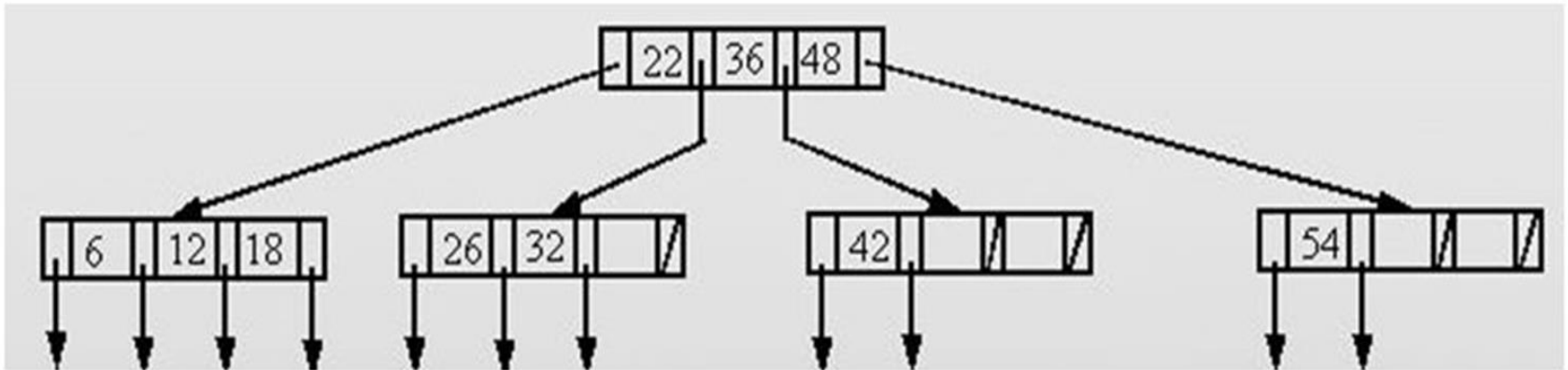
- n – block pointers, $n-1$ search keys, $n-1$ record pointers
- At least $\text{ceil}(4/2) = 2$ links, $\text{ceil}(4/2)-1 = 1$ key at each node
- Each node size is approximately \leq size of block (page in main memory)

Therefore,

$$n(\text{BP}) + (n-1)K + (n-1)\text{RP} \leq \text{size of block}$$

B Trees (no. of keys at each level)

Example: A B-tree of order 4



- | | | | | |
|-----------|---------|-------|-----------|-----------------|
| ➤ Level 0 | 4 BP | n | 3 Keys | $(n-1)$ keys |
| ➤ Level 1 | 4x4 BP | n^2 | 4x3 Keys | $n(n-1)$ keys |
| ➤ Level 2 | 16x4 BP | n^3 | 16x3 Keys | $n^2(n-1)$ keys |
| ➤ Level 3 | 64x4 BP | n^4 | 64x3 Keys | $n^3(n-1)$ keys |

B Trees

➤ Level 0	4 BP	n	3 Keys	$(n-1)$ keys
➤ Level 1	4x4 BP	n^2	4x3 Keys	$n(n-1)$ keys
➤ Level 2	16x4 BP	n^3	16x3 Keys	$n^2(n-1)$ keys
➤ Level 3	64x4 BP	n^4	64x3 Keys	$n^3(n-1)$ keys

The total no. of keys for a B-Tree of order $n(4)$ and height $h(3)$:

$$= (n-1) + n(n-1) + n^2(n-1) + n^3(n-1)$$

$$= (n-1) [1+n+n^2+n^3]$$

$$= n-1+n^2-n+n^3-n^2+n^4-n^3$$

$$= n^4-1$$

$= n^{(h+1)}-1$ Now here height of tree is 3 and root level starts at level 0 so we can put $(h+1)$ in place of 4 because $(h=3)$

Total no of search keys possible in a tree of order n and height h would be equal to: $n^{(h+1)}-1$

B Trees (Keys at each level)

Height	Keys	$n^l(n-1)$ where l is level
➤ Level 0	$(n-1)$ keys	$n^{l=0}(n-1)$ keys
➤ Level 1	$n(n-1)$ keys	$n^{l=1}(n-1)$ keys
➤ Level 2	$n^2(n-1)$ keys	$n^{l=2}(n-1)$ keys
➤ Level 3	$n^3(n-1)$ keys	$n^{l=3}(n-1)$ keys

At each level the maximum no. of search keys possible would be: $n^l(n-1)$, where l is the level (assume root to be at level 0).
How many minimum search keys possible at each level?

Height	max. BP	max. Keys	$\text{ceil}(n/2)=m$	$\text{ceil}(n/2) - 1 = m-1$
Level 0	4	$(n-1)$	m (2 for root node)	$m-1$ (1 for root node)
Level 1	4x4	$n(n-1)$	m^2	$m(m-1)$
Level 2	16x4	$n^2(n-1)$	m^3	$m^2(m-1)$
Level 3	64x4	$n^3(n-1)$	m^4	$m^3(m-1)$

B+ Trees

- Structure of a non-leaf (internal) node in a B+ Tree: e.g. a 3-4 B+ Tree (order 4):



- Structure of a leaf in a B+ Tree:



- In a B+ Tree the data record pointers are always stored in the leaf nodes.
- For every access, the path accessed is from the root to the respective leaf (same length).
- Since there are no record pointers in the internal nodes, the number of key values that can be stored in one node is more than that of a B Tree.
- Since more keys can be stored, the height of a B+ Tree as compared to a B Tree is less.
- Order (n) of an internal node : n – block pointers, n-1 search keys,

$$n(\text{BP}) + (n-1)K \leq \text{size of block}$$

B+ Trees in practice



- Typical order: 200 (8Kb page/40 bytes per index entry).
 - Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 Mbytes



- The BP (block pointers) in the leaf nodes point to the next leaf and the key values in the leaf nodes are sorted.
- In case of sequential access, the leaf nodes can be directly accessed.
- The order of internal nodes and the order of the leaf nodes are not necessarily same. This is because the size of a block pointer and size of a record pointer is not same.
- In a non-leaf node, if order is n then keys are $n-1$.
- In a leaf node, the number of record pointers and no. of keys are same and so if the order of a leaf node is n_{leaf} then the no. of keys in a leaf node is also n_{leaf} .

Order (n_{leaf}) of a leaf node is:

$$n_{\text{leaf}}(\text{RP}) + n_{\text{leaf}}(\text{K}) + \text{BP} \leq \text{size of block}$$

Examples of B / B+ Trees

Suppose in a B+ tree, the search key = 9 bytes, block size = 512 bytes, record pointer = 7 bytes and block pointer = 6 bytes. Find the order p of an internal node and order p_{leaf} of a leaf node.

$$p \cdot 6 + (p-1) \cdot 9 \leq 512$$

$$6p + 9p - 9 \leq 512$$

$$15p \leq 512 + 9$$

$$p \leq 521/15$$

$$p \leq 34.73$$

$$\mathbf{p = 34 \text{ (order of internal node)}}$$

$$9p_{\text{leaf}} + 7p_{\text{leaf}} + 6 \leq 512$$

$$p_{\text{leaf}} \leq 506/16$$

$$p_{\text{leaf}} \leq 31$$

$$\mathbf{p_{\text{leaf}} = 31 \text{ (order of leaf node)}}$$

Examples of B / B+ Trees

If a file consists of 5 attributes, then the number of secondary indices that can be constructed on that file is:

1. 1
2. 5
3. 32
4. 31

Ans. 4. 31

Secondary indices are built on non ordering fields. Hence if a file consists of 'n' attributes then the number of secondary indices = $2^n - 1$

$$\begin{aligned} 2^n - 1 &= 2^5 - 1 \\ &= 32 - 1 \\ &= 31 \end{aligned}$$

Examples of B / B+ Trees

Let 'm' be the number of primary indices on a file and 'n' be the number of clustering indices on other file, then which of the following is true?

1. $m \leq 1$ and $n \leq 1$
2. $m \leq 1$ and $n \geq 1$
3. $m \geq 1$ and $n \leq 1$
4. $m \geq 1$ and $n \geq 1$

Ans. 1. $m \leq 1$ and $n \leq 1$

There is at the most only one primary index on a file and there can be at the most only one clustering index on a file.

Examples of B / B+ Trees

Which of the following is true?

1. Both sparse and dense indices consume same space
2. Sparse index consumes more space than dense index
3. Dense index consumes more space than sparse index
4. Can't say

Ans. 3. Dense index consumes more space than sparse index

The maximum number of records that can be indexed by a B tree of degree 3 and height 3 is:

1. 80
2. 27
3. 26
4. None of these

Ans. 1. 80

$$n^{h+1} - 1 = 3^{3+1} - 1 = 3^4 - 1 = 81 - 1 = 80$$

Examples of B / B+ Trees

The record pointer, key field and block pointer of a B tree are 8 bytes, 10 bytes and 6 bytes respectively. Calculate the order of the tree if the block size is 1 kbyte.

1. 44
2. 43
3. 24
4. None of these

Ans. 2. 43

Suppose the order is n (n block pointers)

$$(n-1)8 + (n-1)10 + 6n \leq 1024$$

$$8n-8+10n-10+6n \leq 1024$$

$$24n \leq 1024 + 18$$

$$24n \leq 1042$$

$$n \leq 43.41$$

Order 43.

Examples of B / B+ Trees

Consider a B tree of degree 'n' with height 'h'. Calculate the maximum number of records that can be indexed by the above tree [assume root is at level '0']

1. $n^h - 1$
2. n^h
3. $n^{h+1} - 1$
4. None of these

Ans. 3. $n^{h+1} - 1$

If the order is n, then (n links, n-1 keys)

Number of records at level '0' = n-1

Number of records at level '1' = $n(n - 1)$

Number of records at level 'h' = $n^h (n - 1)$

Total = $(n - 1) (1 + n + \dots + n^{h-1} + n^h)$

$$= (n - 1) (n^{h+1} - 1) / (n - 1)$$

$$= n^{h+1} - 1$$

Examples of B / B+ Trees

In a B-Tree, the block size is 512 bytes, search key is 2 bytes, block pointer size is 6 bytes and record pointer is 8 bytes. Every node contains $n-1$ record pointers, $n-1$ keys and n block pointers. Find out the maximum number of keys that can be accommodated in a level 0 B-Tree.

a. 32 b. 31 c. 30 d. none

Ans. b. 31

$K = 2$ bytes, $rp = 8$ bytes $bp = 6$ bytes

$$(n-1)rp + (n-1)k + nbp \leq 512$$

$$(n-1)8 + (n-1)2 + 6n \leq 512$$

$$8n - 8 + 2n - 2 + 6n \leq 512$$

$$16n \leq 512 + 10$$

$$n \leq 522/16$$

$n \leq 32$ (block pointers), therefore $n-1 \leq 31$ (search keys)

Examples of B / B+ Trees

In a B-Tree, the block size is 512 bytes, search key is 2 bytes, block pointer size is 6 bytes and record pointer is 8 bytes. Every node contains $n-1$ record pointers, $n-1$ keys and n block pointers. Find out the average number of keys that be accommodated if the fill factor of B-Tree is 67%.

a. 20 b. 21 c. 31 d. none

Ans. 20

(See previous example)

$n \leq 32$ (block pointers), therefore $n-1 \leq 31$ (search keys)

$$0.67 * 31 = 20.77$$

Examples of B / B+ Trees

Consider a B+ Tree with a node size of 512 bytes, search key size is 5 bytes and block pointer size is 10 bytes. Find out the fan-out of the tree.

a. 22 b. 34 c. 35 d. 36

(Fan-out – No. of children (links) i.e. block pointers)

Ans. b. 34

If n is no. of block pointers, (n-1) no. of search keys,

$$10n + (n-1)5 \leq 512$$

$$10n + 5n - 5 \leq 512$$

$$15n \leq 517$$

$$n \leq 517/15$$

$$n \leq 34.46 \text{ (block pointers)}$$

Examples of B / B+ Trees

Which of the following statements is/are true?

S1: Primary index are clustering index

S2: Secondary index are non-clustering index

S3: Primary index are dense

S4: Secondary index are sparse

- a. S1,S2,S4
- b. b. S1,S2
- c. c. Only S3
- d. d. None

Ans. b. S1,S2

Examples of B / B+ Trees

In a B+ tree, size of each node is same as size of disk block which is 8 KB. Search key size is 64B and disk pointer size is 16B. If the file contains 10^6 search key values then, a look up (read the record) requires how many blocks to be read from the disk, assuming the root node for its heavy access is in buffer.

- a. 2 b. 3 c. 4 d. 5

Ans. b. 3

$$64(n-1) + 16n < 8192$$

$$64n + 16n < 8256$$

$$n < 8256/80$$

$$n < 103.2$$

$$n = 103 \text{ (pointers) approx. } 100$$

$$\text{keys} = 102(n-1)$$

100 at the root (buffer)

$$100 \times 100 \text{ at level 1 (10000)}$$

$$10000 \times 100 \text{ at level 2 (1000000 i.e. } 10^6)$$

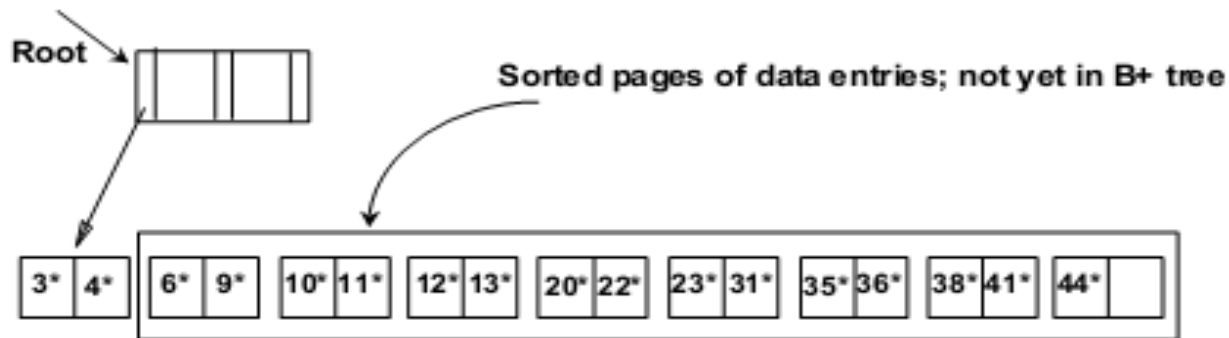
Since the root is in the buffer, two disk access to find the address and one to read the record from the address – 3.

Examples of B / B+ Trees

Bulk Loading of a B+ Tree



- ❖ If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- ❖ Bulk Loading can be done much more efficiently.
- ❖ *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Examples of B / B+ Trees

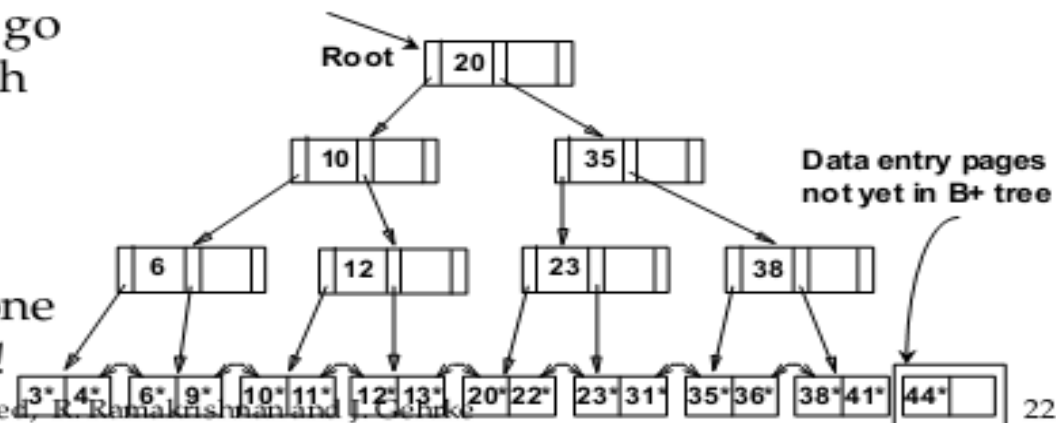
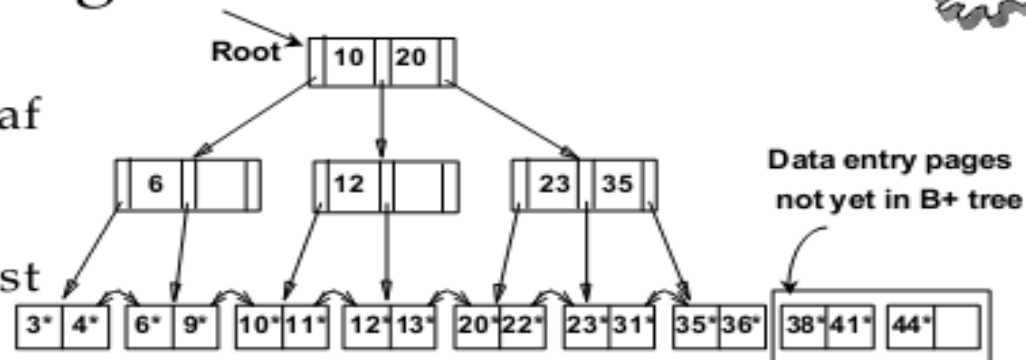
Bulk Loading (Contd.)



- ❖ Index entries for leaf pages always entered into right-most index page just above leaf level.

When this fills up, it splits. (Split may go up right-most path to the root.)

- ❖ Much faster than repeated inserts, especially when one considers locking!



Example of Bulk Loading

1. Sort the data in ascending/descending order.
2. As per the leaf node order, fill the leaf nodes from left to right with the sorted values. There are two options
 - a. Fill the first leaf node completely and then go to the next leaf...all leaf nodes filled with max. keys possible
 - b. Fill the leaf nodes to about 67% capacity (helps when new values are to be inserted).
3. Now insert the values in the internal nodes as per B+ tree insertion technique.

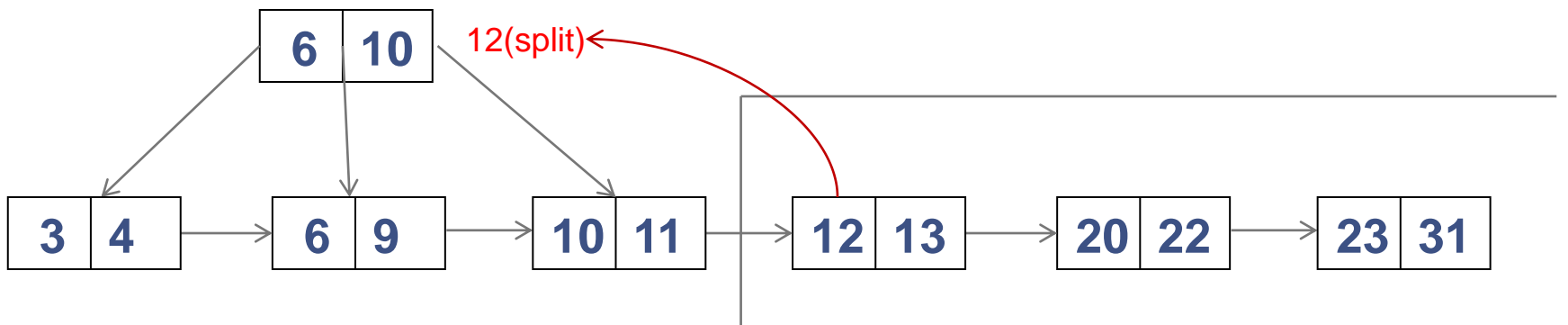
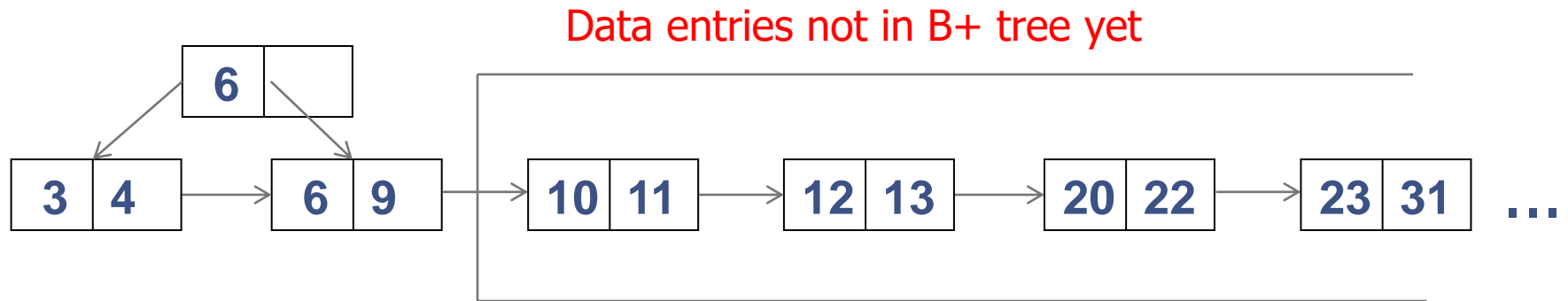
e.g. if the keys to be inserted are 20, 32, 3, 22, 10, 9, 11,
and the order is 3 then,

3, 9, 10, 11, 20, 22, 32, (sorted). Fill the leaf nodes.

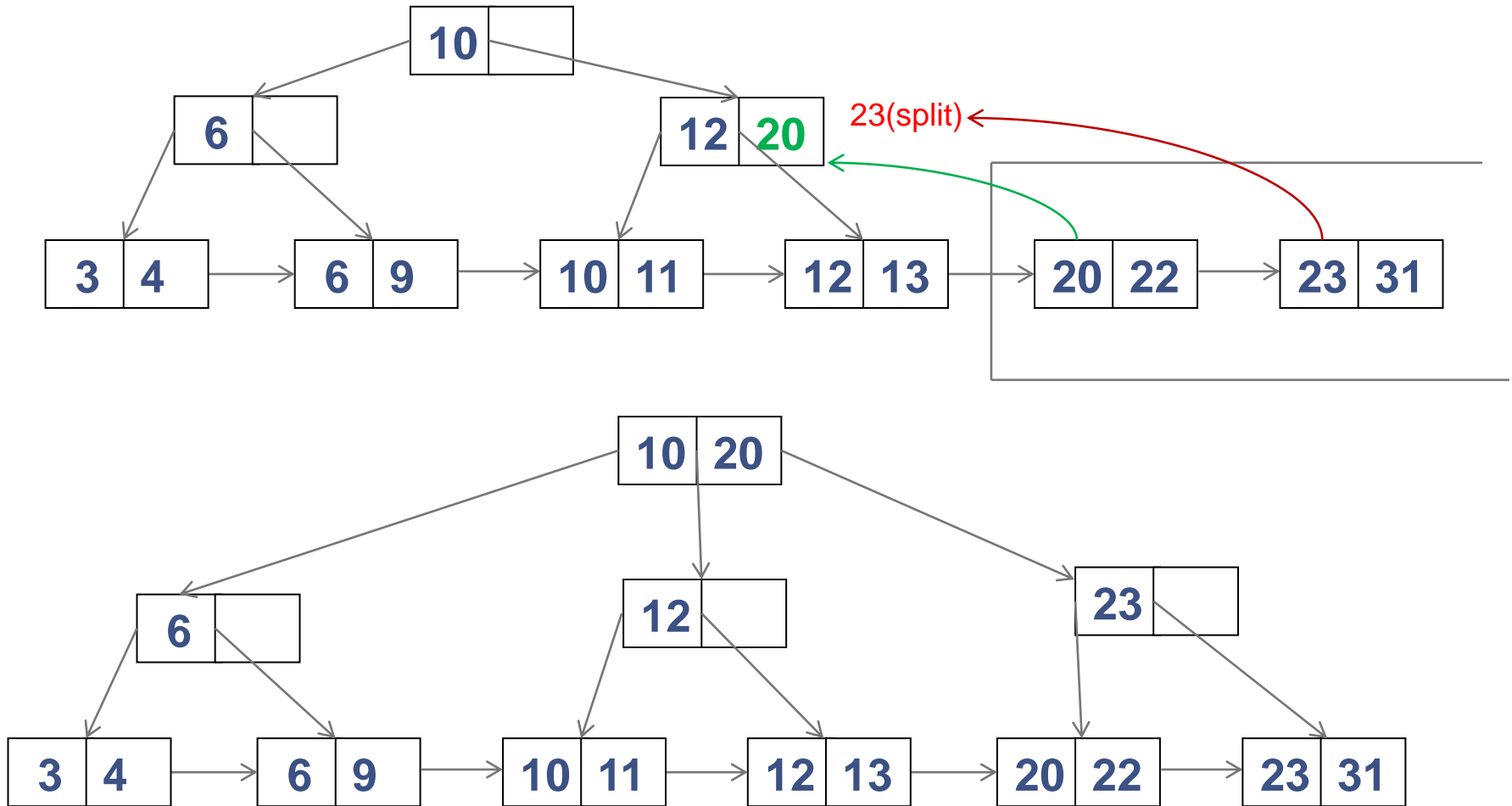


Example of Bulk Loading

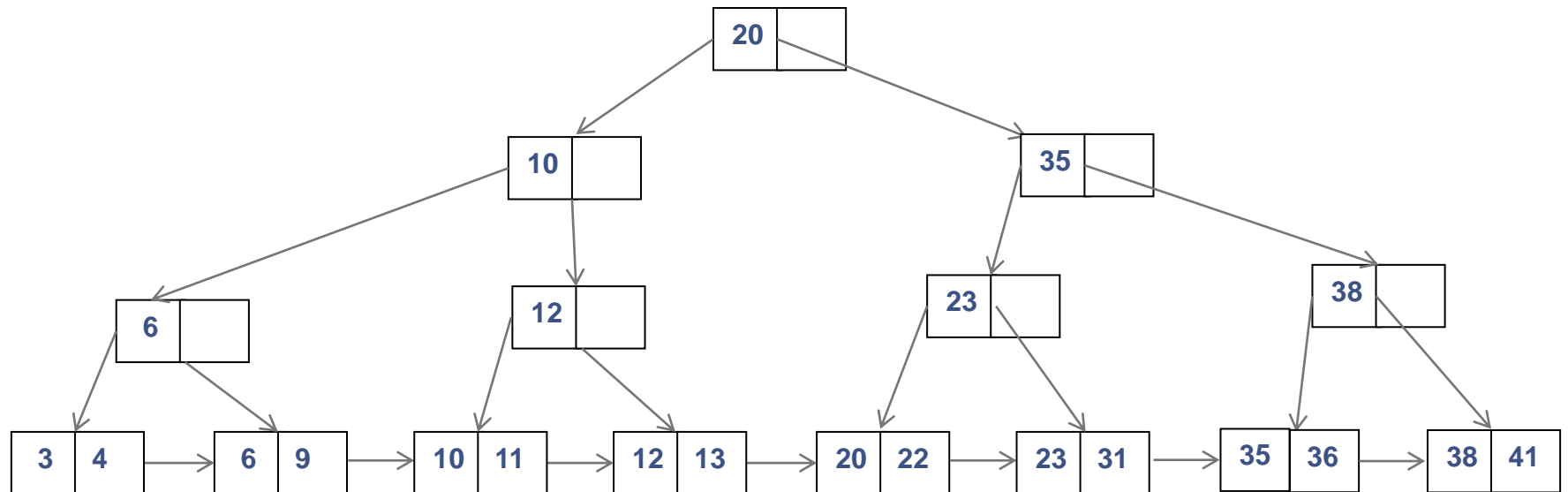
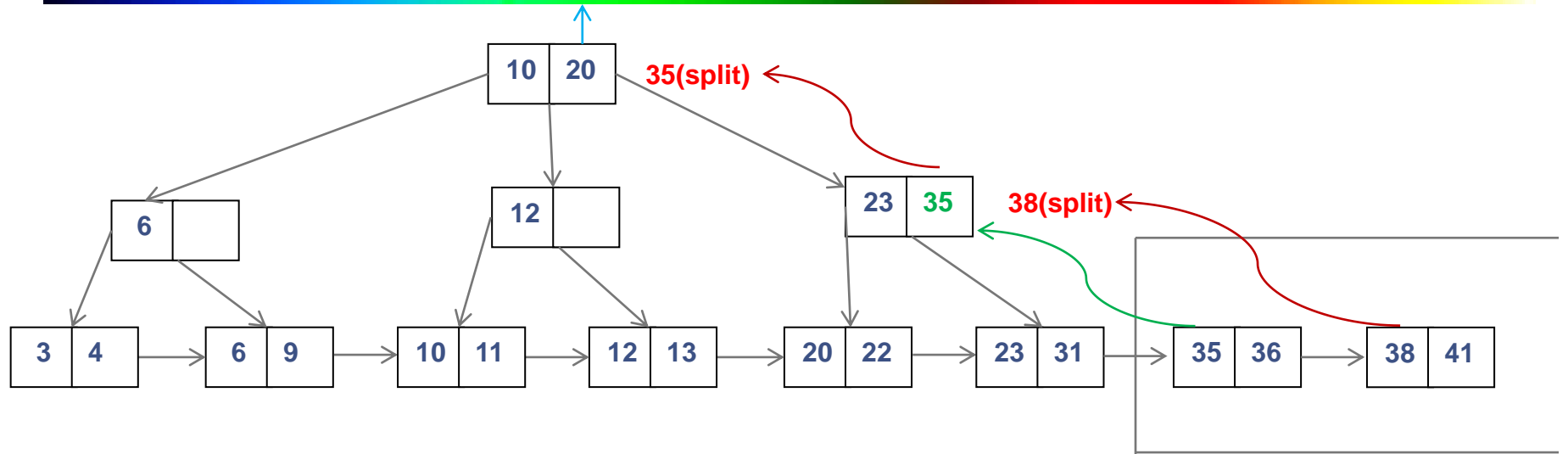
The values are: 3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 23, 31, 35, 36, 38, 41




Example of Bulk Loading



Example of Bulk Loading





Please watch the following video for more details on Bulk Loading:
<https://www.youtube.com/watch?v=ITJy2MXI4lY>