Storage

Physical Data Organization

- Physical database representation. (different from traditional files)
  - To store a file consisting of records, each record having an identical format.
  - A record format consists of a list of field names, with each field processing a fixed number of bytes and having a fixed data type.
- Operations on a record
  - Insert, delete, modify a record.
  - Find a record with a particular value in a particular field or a combination of values in a combination of fields. (look up)

- The complexity of organizing a file for storage depends on the particular combination of these operations that we intend to perform on the file. (here, we don't consider files with variable length records)
- External (secondary) storage: disk storage.
  - A file system using disk storage usually divides the disk into equal sized physical blocks.
  - A file is stored among one or more blocks, with one or more records stored in each block.
- Some blocks are devoted to be a header: a collection of bytes, usually at the beginning of the block, used for special purposes. Ex, contain information connecting the block to other blocks.

- Basic external storage operation is the transfer of a block   from secondary to main memory or vice verse.
  (a block access)
- Records become <u>pinned</u> because they may exist pointers to them somewhere in the database.
- File organization is less flexible for pinned records, because   those records cannot be moved.
- When delete a record, must be careful --> <u>dangling pointer</u>.

- **The simplest organization (Unordered Files)**
  - Records are packed from the beginning of blocks, immediately following the header.
  - Put a count of the number of records in the header.
  - To search a record --> linear search until findor EOF.
  - Record insertion is efficient.
  - Reading the records in order of a particular field requires sorting the file records.
  - The **heap** file organization. (simply list them in as many blocks as they require.)

- The blocks used for a heap may be linked by pointers, or a table of their addresses.

1) Insert a record: add at the end of the block.
2) Delete a record: uses a deletion bit to indicate.(garbage collection)

- Goal
  - Find alternative file organizations that allow arbitrary lookups without scanning more than a small fraction of the file.
  - Avoid using too much extra space, and make insertion and deletion (operations) easy.

- **Ordered Files (Sequential Files)**
  - File records are kept sorted by the values of an ordering field.
  - Insertion is expensive; records must be inserted in the correct order.
  - A binary search can be used to search for a record on its ordering field value.
  - Reading the records in order of the ordering field is quite efficient.

Hashed Files

- Divide the records of a file among buckets; each consists of one or more blocks of storage.
- The hash function *h*: input (a value for the key of the field), output (an integer, which indicates the number of the buckets in which the record with key value v is to be found)

1) Treat the key value as a sequence of bits. (Do permutation)
2) <u>Divide</u> the sum by the number of buckets, and the remainder as the bucket number.

   h = |key| mod 5;

- Bucket directory contains B pointers of B buckets. (stored in the main memory)
- Each bucket contains a pointer the first block if it exists; otherwise, a null pointer exists.
- Each block also has a header: pointer to the next block (in the same bucket), status of records inside the block (empty or non-empty).
- Each record has a deletion bit.

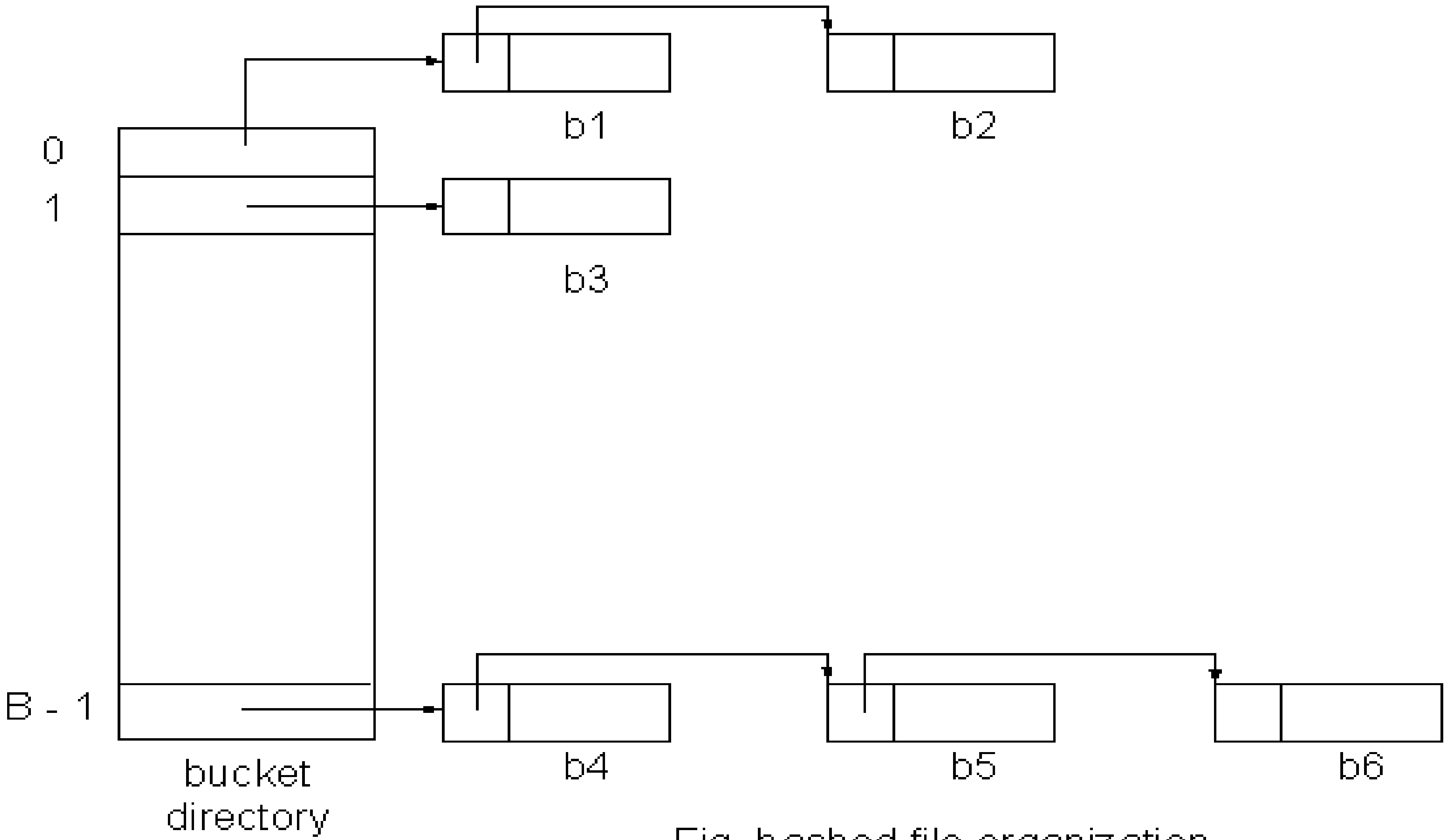


Fig. hashed file organization

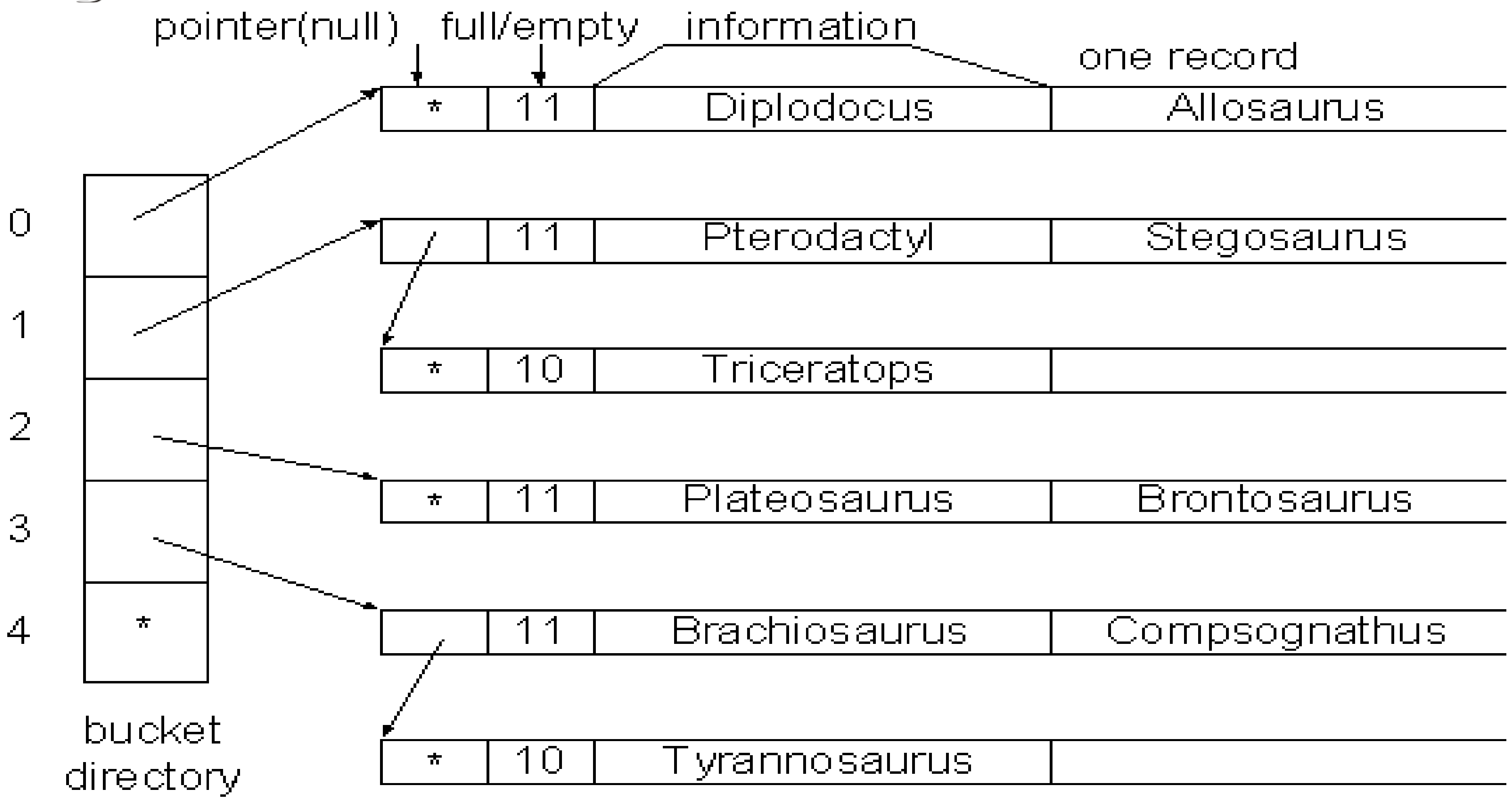


Fig. A hashed File

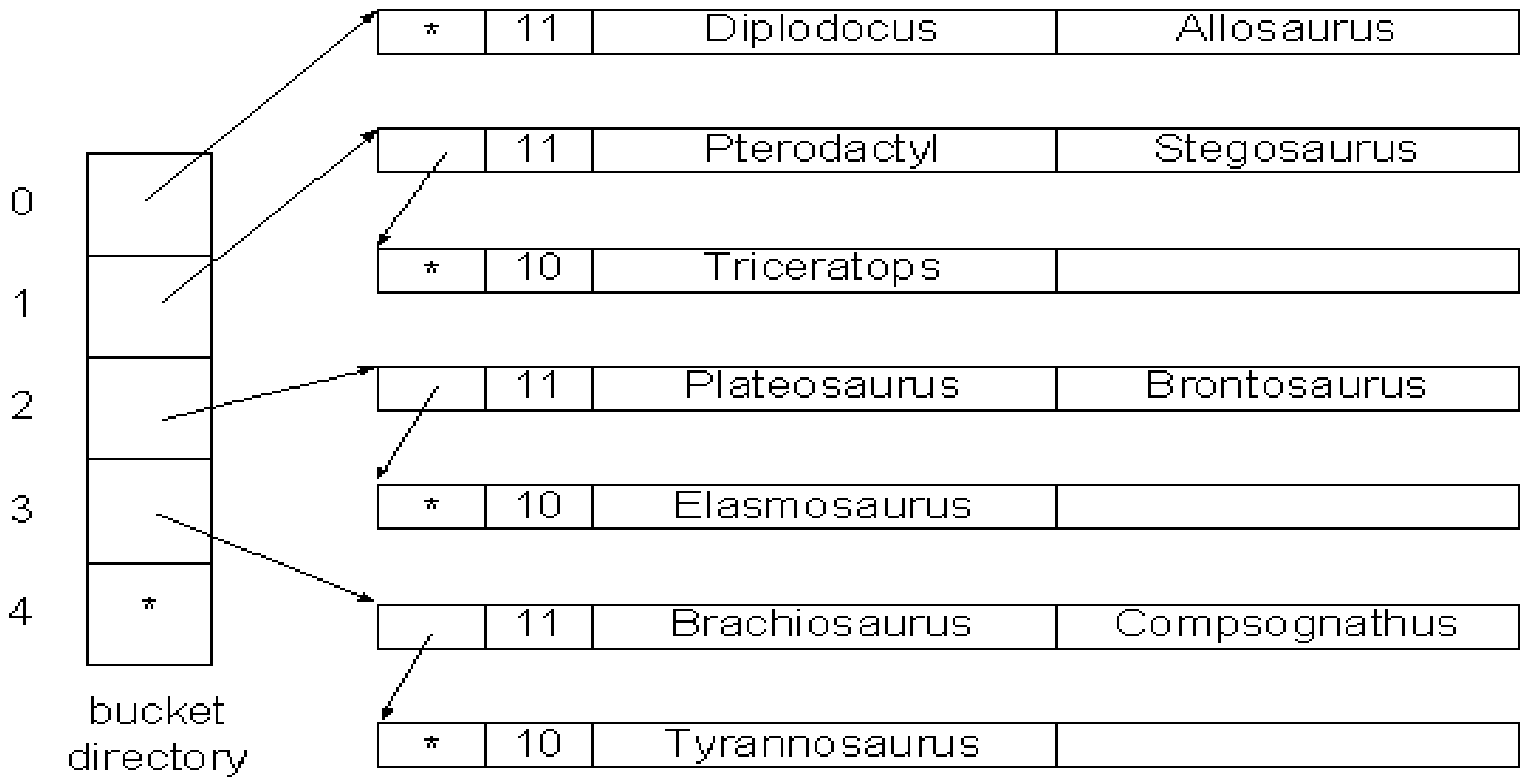

0
1
2
3
4
*
bucket directory
* | 11 | Diplodocus | Allosaurus
11 | Pterodactyl | Stegosaurus
* | 10 | Triceratops
11 | Plateosaurus | Brontosaurus
* | 10 | Elasmosaurus
11 | Brachiosaurus | Compsognathus
* | 10 | Tyrannosaurus

Fig. Insert (Elasmosaurus)

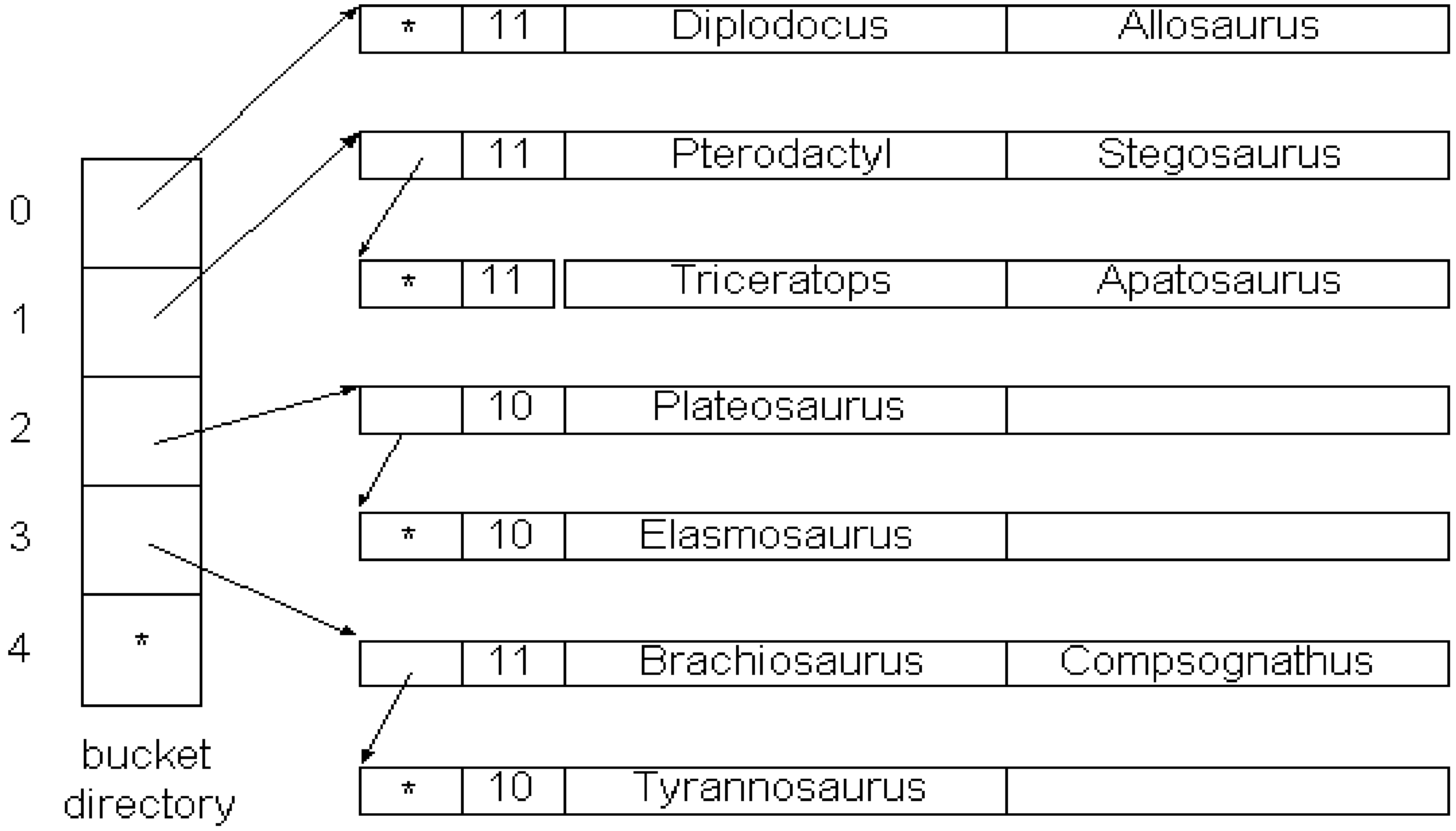


Fig. modify Brontosaurus to Apatosaurus

Operations

1) Lookup.
2) Modification.
   - If some field to be modified is part of the key, treat this modification as a deletion followed by an insertion.
3) Insertion.
   - Lookup first.
   - Find the first empty subblock in the blocks for bucket h(v).
   - Link old and new blocks together.
4) Deletion.
   - Lookup first.
   - Set the deletion bit or (simply move this record and set record status)

- Records become pinned because there may exist pointers to them somewhere in the database. We can never move those record; otherwise, dangling pointers occur.
- Hashed File: Fastest of all methods.
  - However, ordered access on the hash key is quite inefficient.
  - As a file grows, access slows(buckets get large).
  - The number of buckets is fixed -> as a file grows, high collision.
- How to choose bucket size, block size, hash function? (such that low collision is achieved)

Indexed Files

- Access via a key.
- ISAM: Indexed Sequential Access Method.
- Stored records of a file by their key values. (sort the values first)
- A key may contain more than one field. (we still can sort a key with several fields)
- If it does not exist, we know <u>when to stop</u> searching.
- A sorted file --> create an index file.
- The index is called an <u>access path</u> on the field.

- An index (key value v, block address b): the first record in the block with address b has key value v.
- Given a key value v1 for the file being indexed, find the record (v2, b) in the index such that v2 ≦ v1 and either (v2, b) is the last record in the index, or the next record (v3, b') has v1 ≦ v3. (we say v2 cover v1 in this case)
- Search an index

1) linear search.
2) binary search.

- Advantages of a sorted index file (over a hashed file)

1) for the hashed organization, it is very difficult to process or list records in the order of their values.
2) for range queries, a sorted file is better.

- Disadvantages of a sorted organization

1) slower than a hashed file.
2) need space for the index. (index must be kept in order after each kind of operation.)

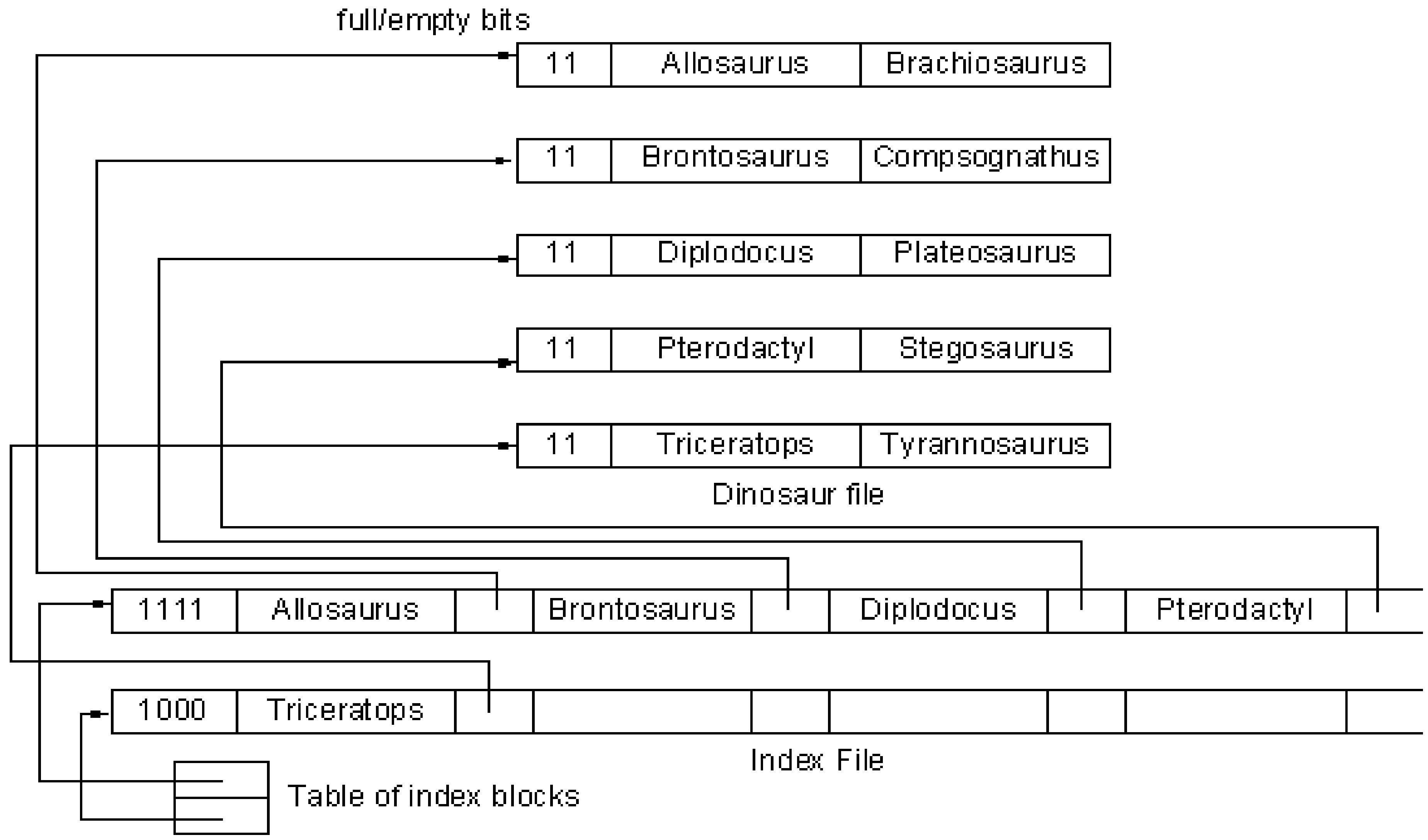
Operations on an **index file (unpinned records)**

1) Lookup. Find v2 cover v1.
2) Modification.
   – If the modification changes the key, treat the operation as an insertion and deletion.
3) Insertion.
   – Lookup first.
   – Put the new record in its correct place in Block Bi, keeping the records stored and moving records with key values greater than v1 to the right, to make room for the new record.
   – May have to modify index file entry when it is B1.
   – Change the full/empty information in the header of block Bi.

- When Bi is full
  a) Bi is split into two half-empty blocks. Or
  b) shift excess records from Bi to Bi+1, and shift all records in Bi+1 to the right one position.
  - Change the empty/full information in the header of Bi+1.
  - Modify the index record for Bi+1 to reflect the new key value in its first record.
  - If Bi+1 does not exist, because i = k, or Bi+1 exists but is full, we must get a new block, which will follow Bi in the order.
  - Place the excess record from Bi into the new block, and <u>insert a record for the new block in the index file</u>.

4) Deletion.

- Lookup first.
- <u>Move</u> any records to its right one subblock left to close the gap and adjust the full/empty bits in the header.
- If the block is complete empty, return it to the file system, and delete the record for that block in the index.
- If the deleted record was the first in the block, modify the index record for that block.



full/empty bits
11 | Allosaurus | Brachiosaurus
11 | Brontosaurus | Compsognathus
11 | Diplodocus | Plateosaurus
11 | Pterodactyl | Stegosaurus
11 | Triceratops | Tyrannosaurus
Dinosaur file
1111 | Allosaurus | Brontosaurus | Diplodocus | Pterodactyl
1000 | Triceratops
Index File
Table of index blocks

Fig. Sorted, indexed dinosaur file (unpinned records)

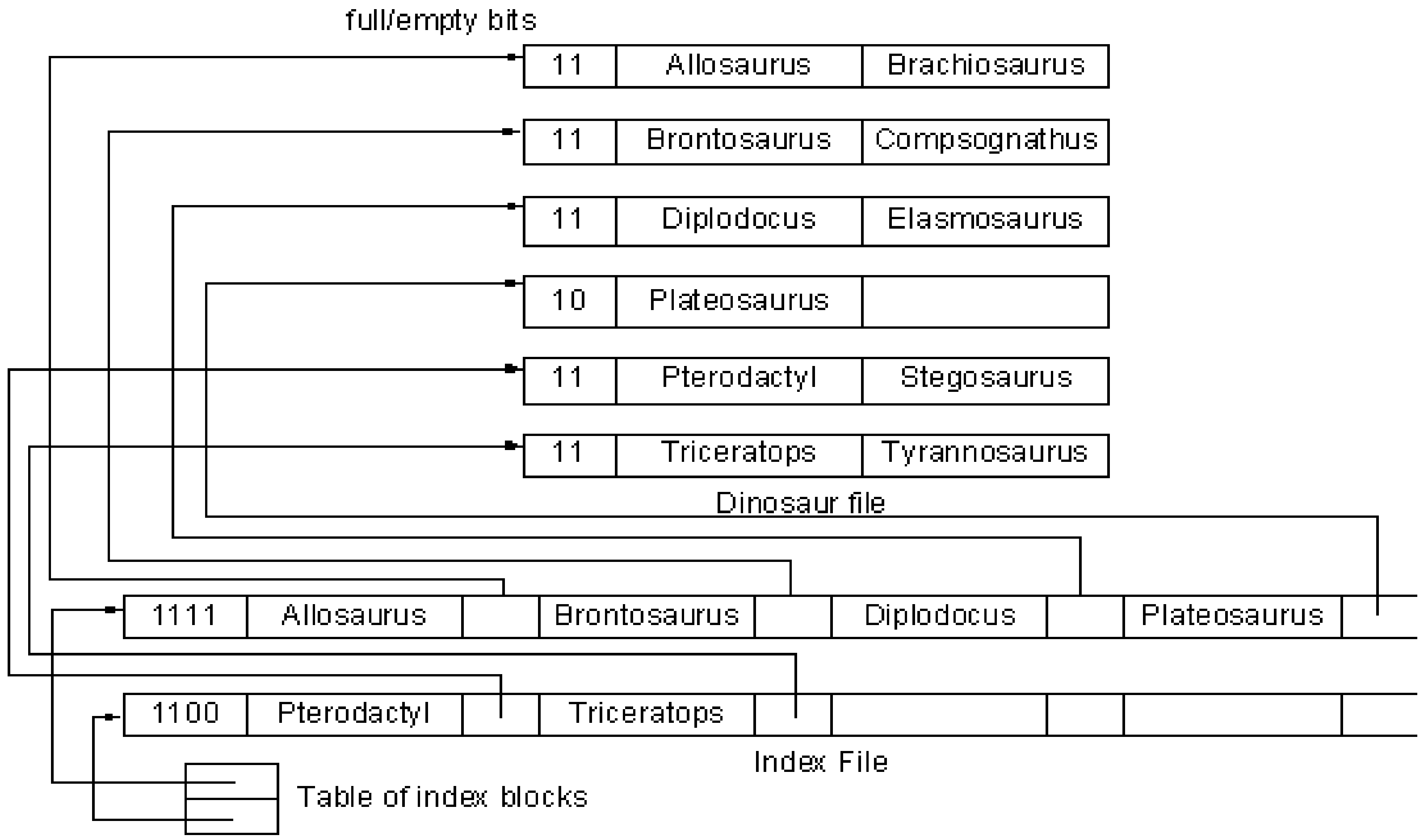


Fig. Insert Elasmosaurus (unpinned records)

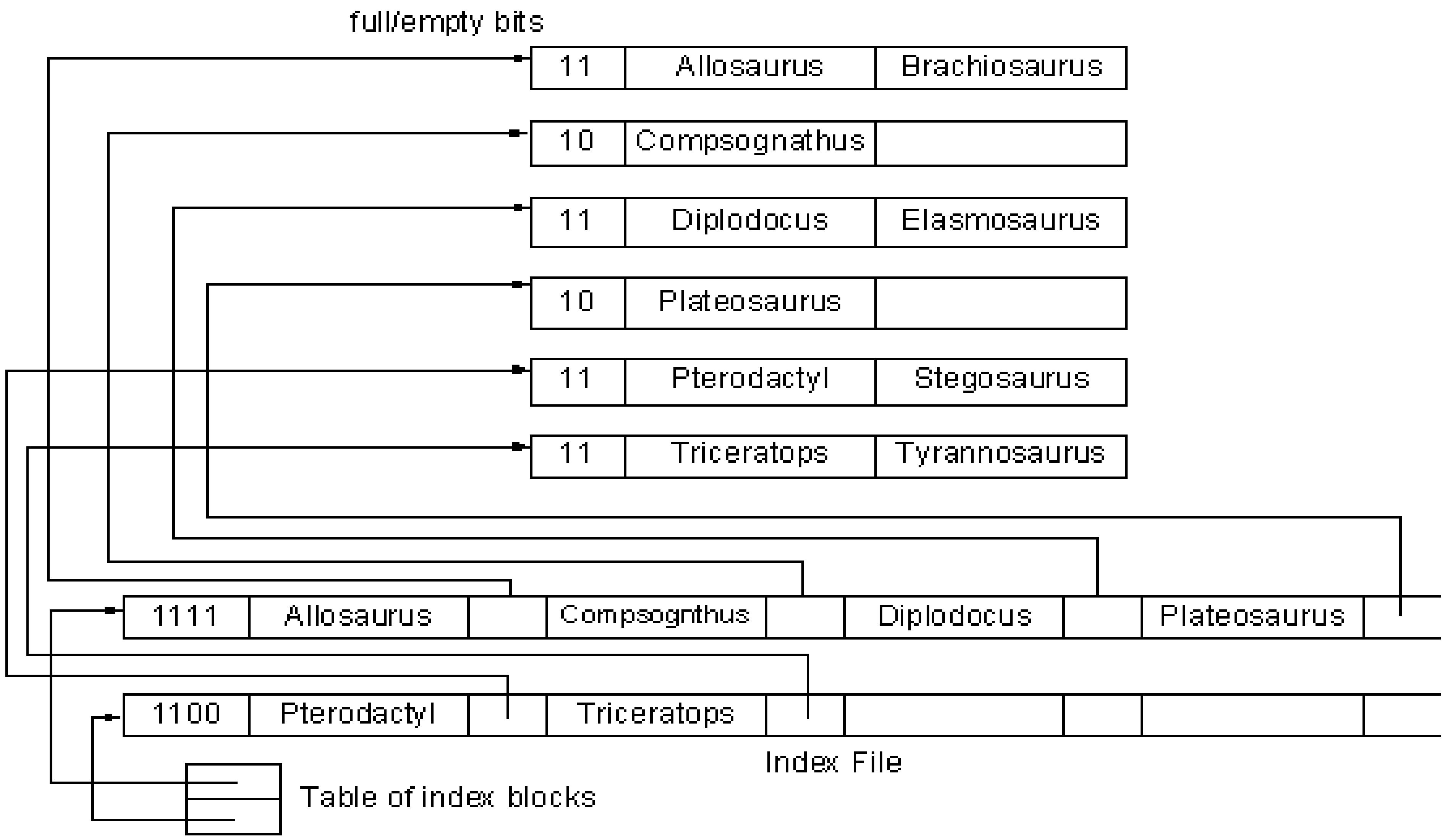


Fig. (a) modify Brontosaurus to Apatosaurus (unpinned records)

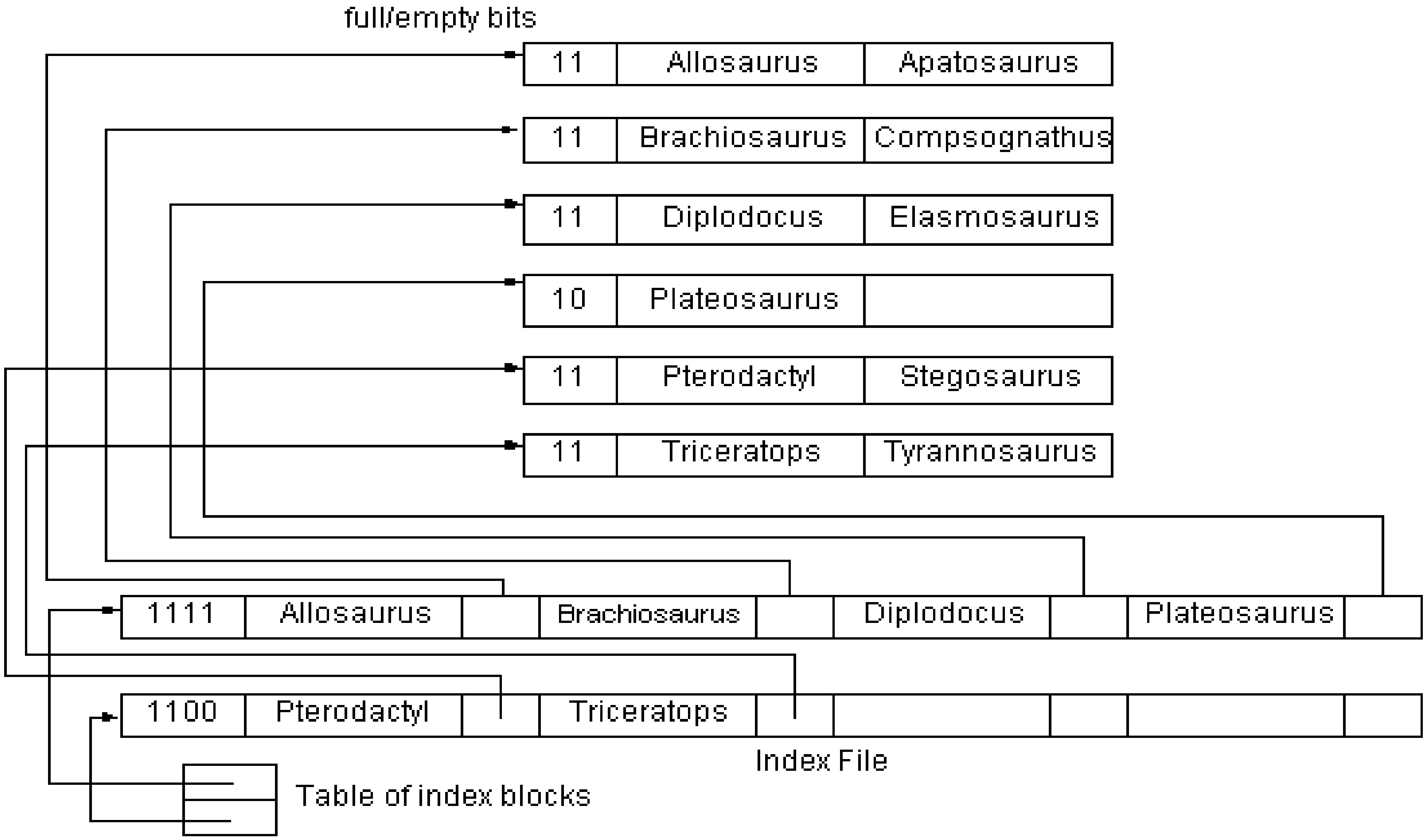


Fig. (b) modify Brontosaurus to Apatosaurus (unpinned records)

An Organization for Sorted Files with Pinned Records

- We view each block of the main file as the first block of a bucket.
- As records are inserted, additional blocks will be added to the bucket, and new blocks are chained by series of pointers extending from the original block for that bucket.
- We also create an empty block to begin a bucket that will hold any records that precede the first record of the first block in the initial file.
- The index never changes in this organization, and the first records of each block of the initial file determines the distribution of records into buckets forever.

- The first index has pointer only without key value.(an empty block)
- Operations

1) Initialization.
   - Sort the file and distribute its records among blocks.
   - Fill each block to less than its capacity to make room for expected growth and avoid long chains of blocks in one bucket.
   - Get one additional block to the head of the bucket for those records inserted later that precede all records of the initial file.

2) Lookup
   - Find v2 that covers v1.
   - If v1 is less than the first key value of the index file (note that the second index record has the first key value), then the desired index record is the first record.
   - Scan this block and any blocks of the bucket chained to it.
3) Modification.

4) Insertion.
   - Lookup first.
   - Scan the blocks of the bucket to find the first empty place.
   - If no empty subblock exists, get a new block and place a pointer to it in the header of the last block of the bucket. (chain together) (without moving any record)

5) Deletion.
   - Lookup first.

   a) Set the full/empty bit to 0. or
   b) For pinned record concerned, set full/empty bit to 1 and deletion bit (in this record itself) to 1. (without moving any record)

- Problem: there is no ordering inside each bucket.
- Therefore, link the buckets in order.
- Put one more pointer in each header, and link the first block of successive buckets.
- To save space, replace the null pointer at the end of this chain for each bucket by a pointer to the first block of the next bucket.
- A single bit (empty-full status bit, 1/0) in the header tells whether the pointer is to the next bucket or to the next block of the same bucket.
- Records are not placed in a bucket in a sorted order after the initialization.
- To examine records in sorted order, we add a pointer in each record to the next record in sorted order.

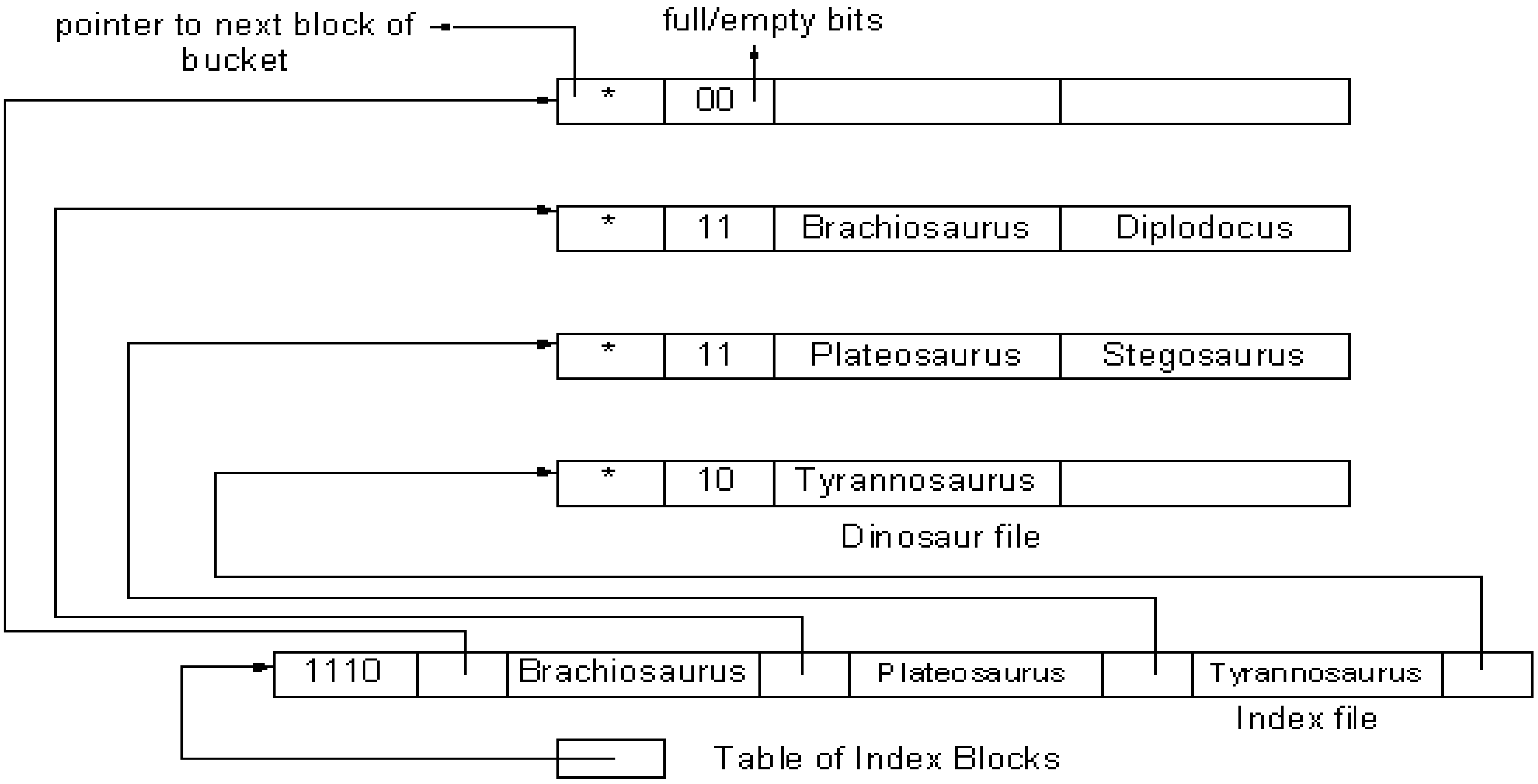
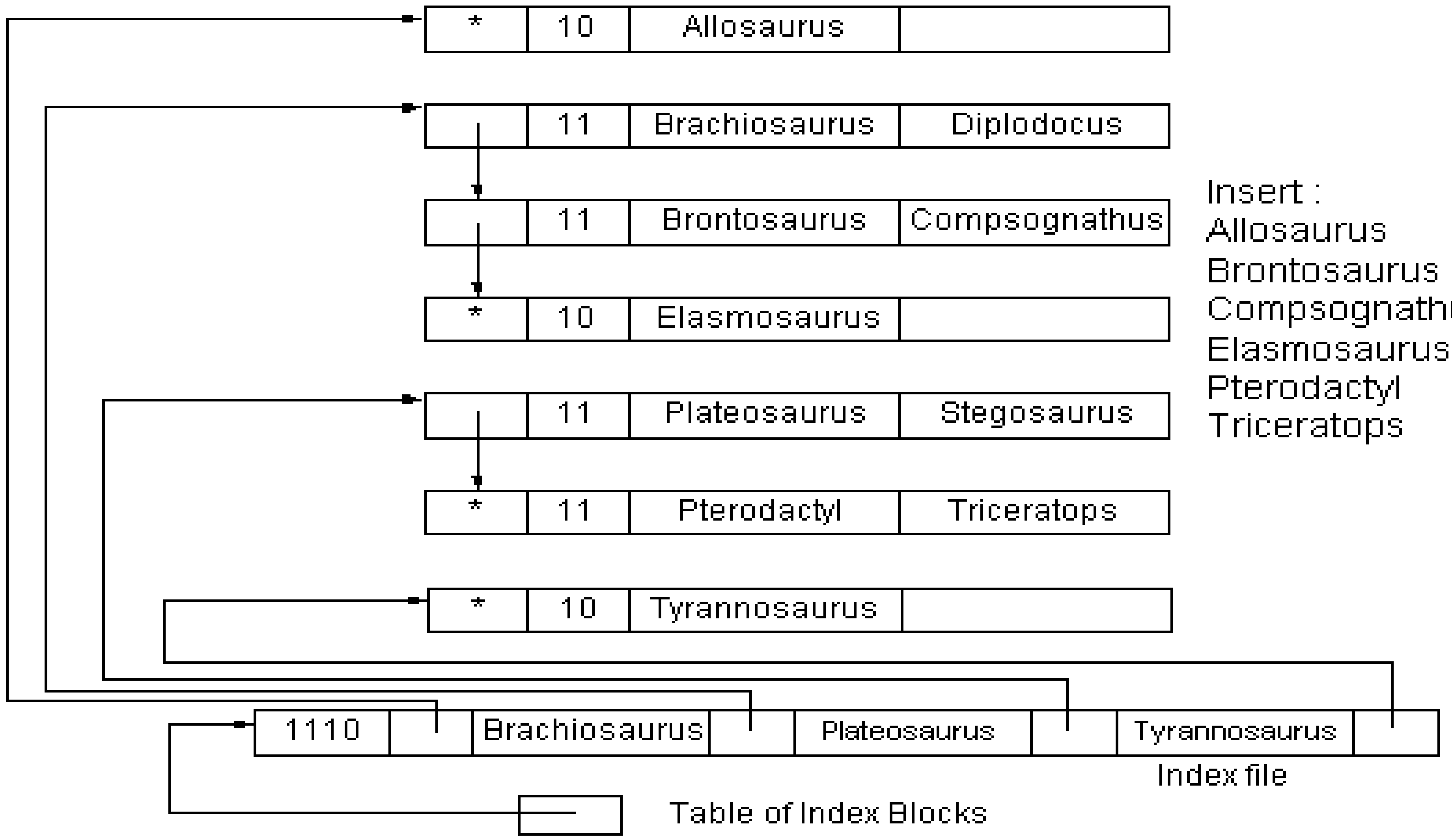


Fig. Initial file organization (pinned records)



Insert :
Allosaurus
Brontosaurus
Compsognathus
Elasmosaurus
Pterodactyl
Triceratops

Fig. Insertion (pinned records)

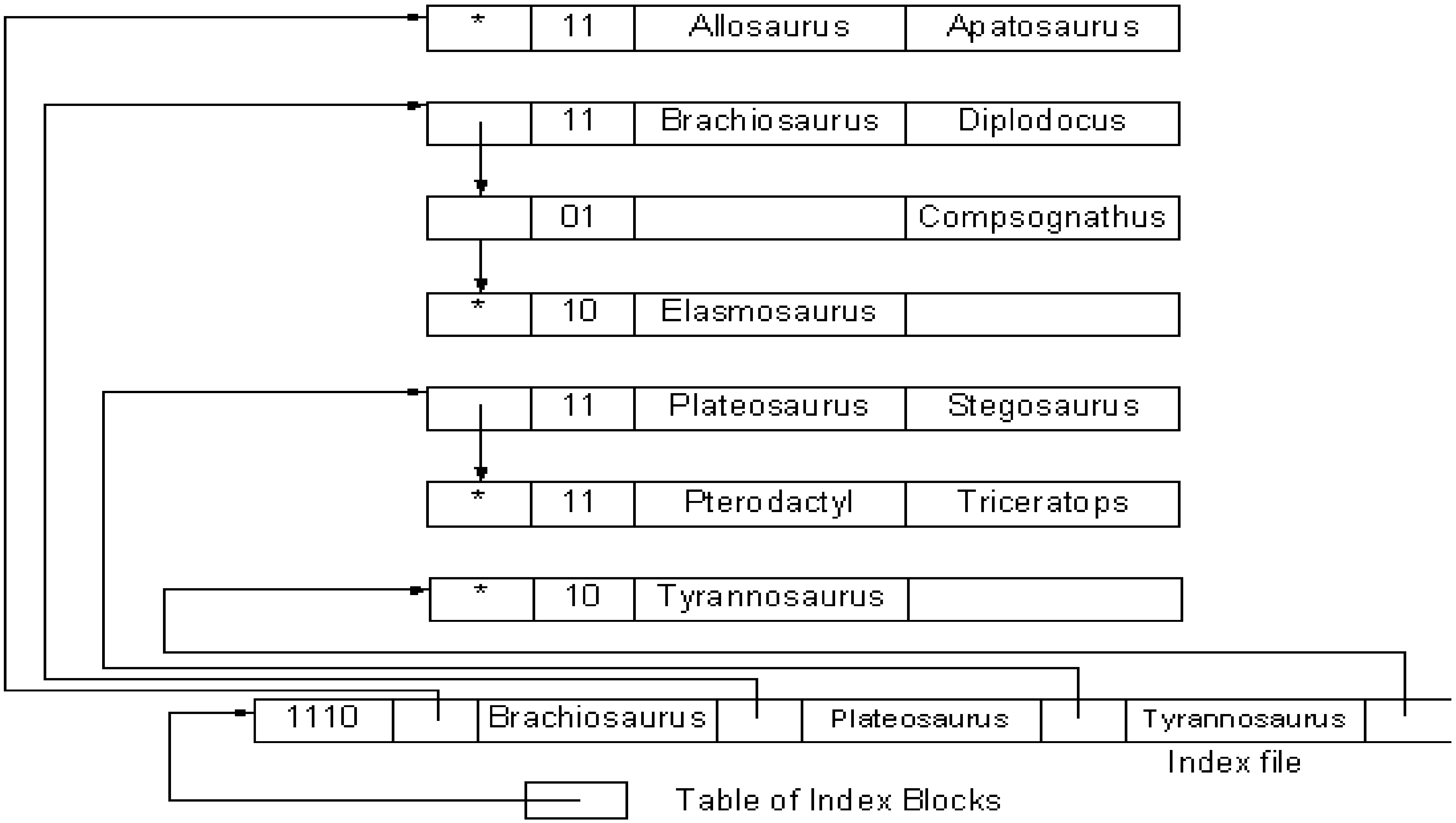


Fig. modify Brontosaurus to Apatosaurus (pinned records)

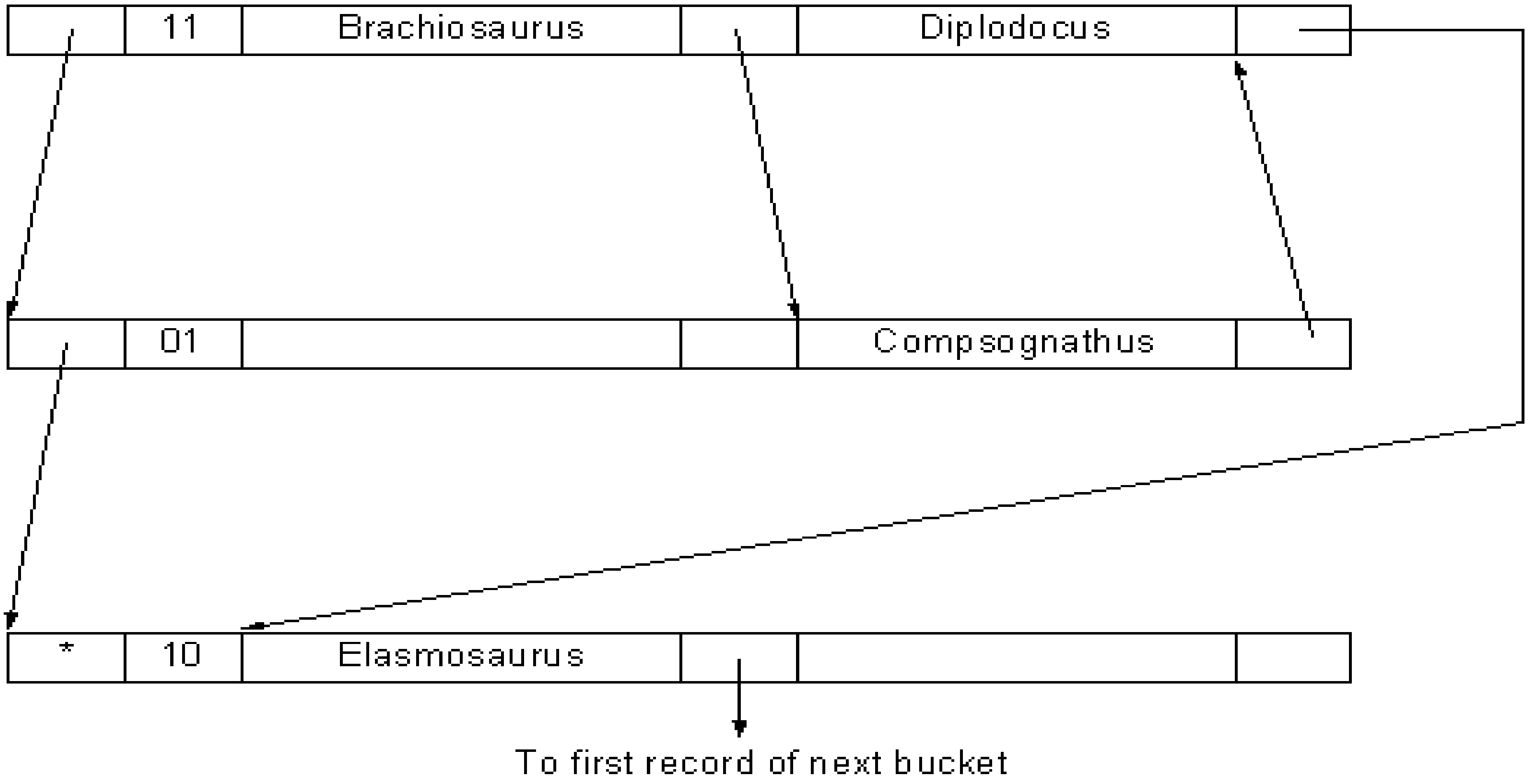
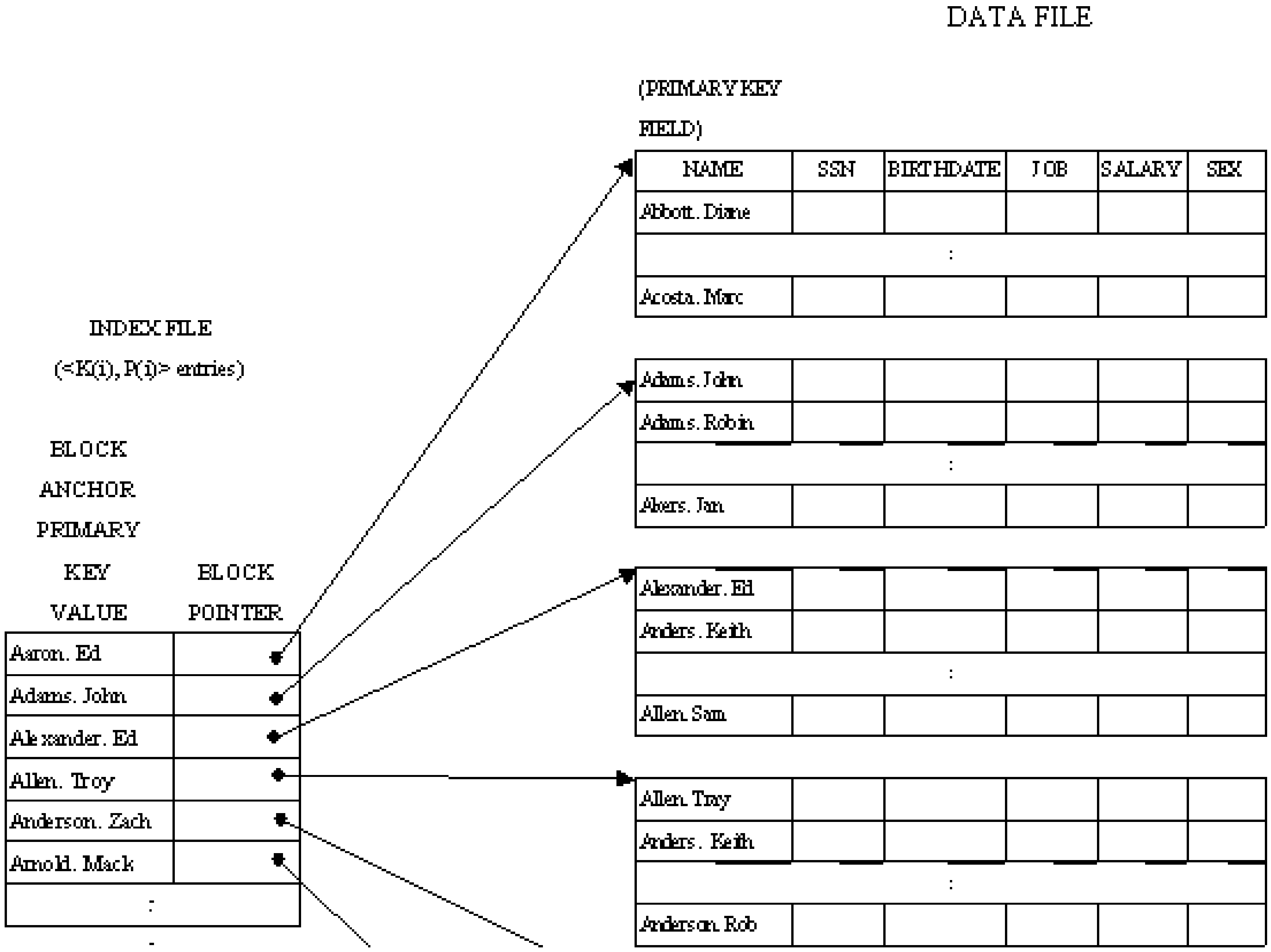


Fig. A bucket with pointers to indicate sorted order

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file.
  - The data file is ordered on a key field.
  - Includes one index entry for each block in the data file; the index entry has the key field value for the first record in the block.
- Clustering Index
  - Defined on an <u>ordered</u> data file.
  - The data file is ordered on a <u>non-key</u> field.
  - Includes one index entry for each distinct value of the field; the index entry points to the first block that contains records with that field value.

- Secondary Index
  - Defined on an unordered data file.
  - Can be defined on a key field or a non-key field.
  - Include one entry for each record in the data file; hence, it is called a dense index. (vs. sparse index)
- The basic idea behind clustering is to try and store records that are logically related physically close together on the disk.
- A file with an index on every field is called fully inverted.
- Dense and non-dense indexing:
  - Non-dense: does not contain an entry for every stored record in the indexed file. (less storage; but can not perform existence tests on the bases of the index alone)



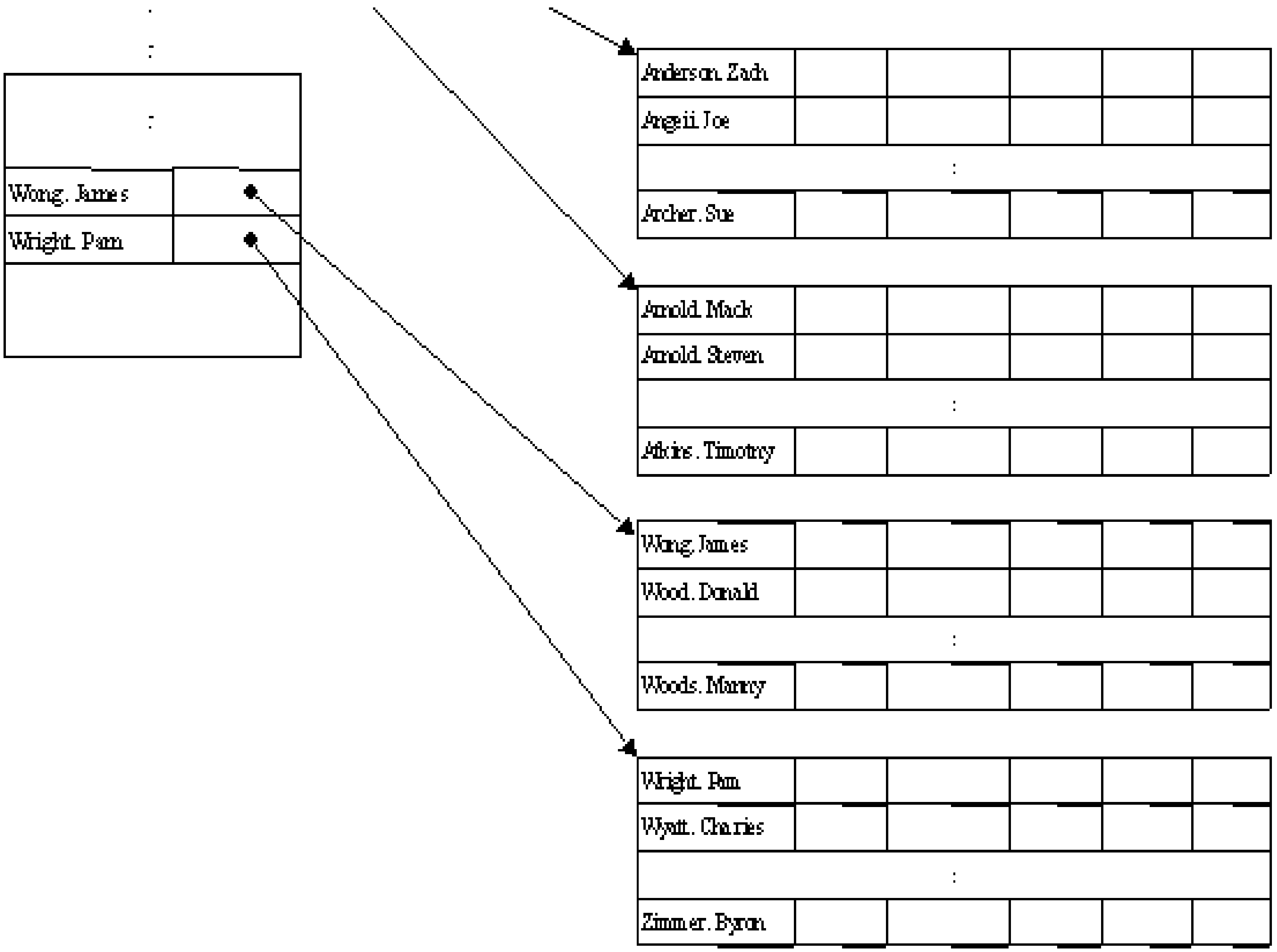
BLOCK ANCHOR PRIMARY KEY VALUE	BLOCK POINTER
Aaron, Ed	●
Adams, John	●
Alexander, Ed	●
Allen, Troy	●
Anderson, Zach	●
Arnold, Mack	●
:	

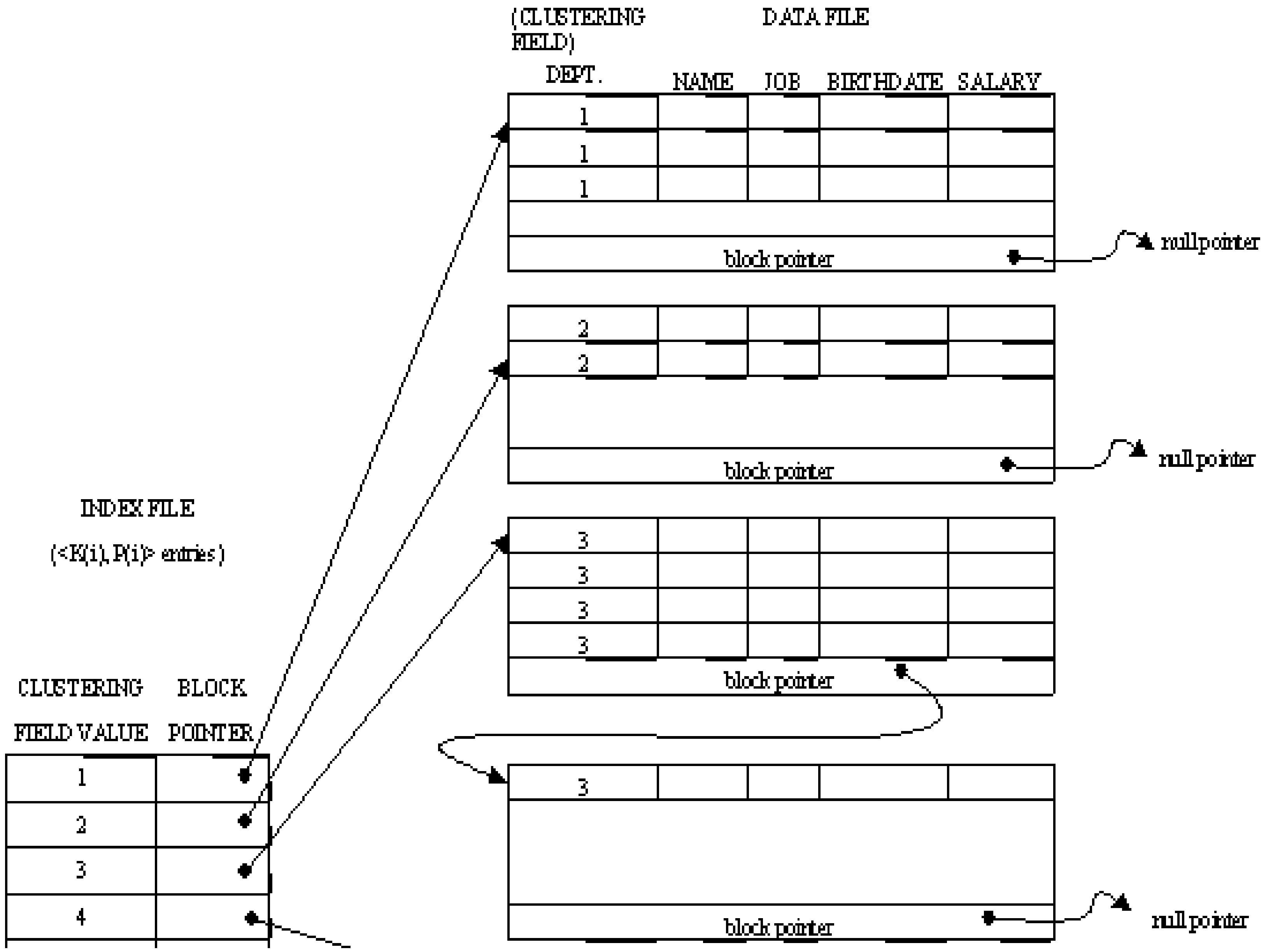
NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
Abbott, Diane					
:					
Acosta, Marc					

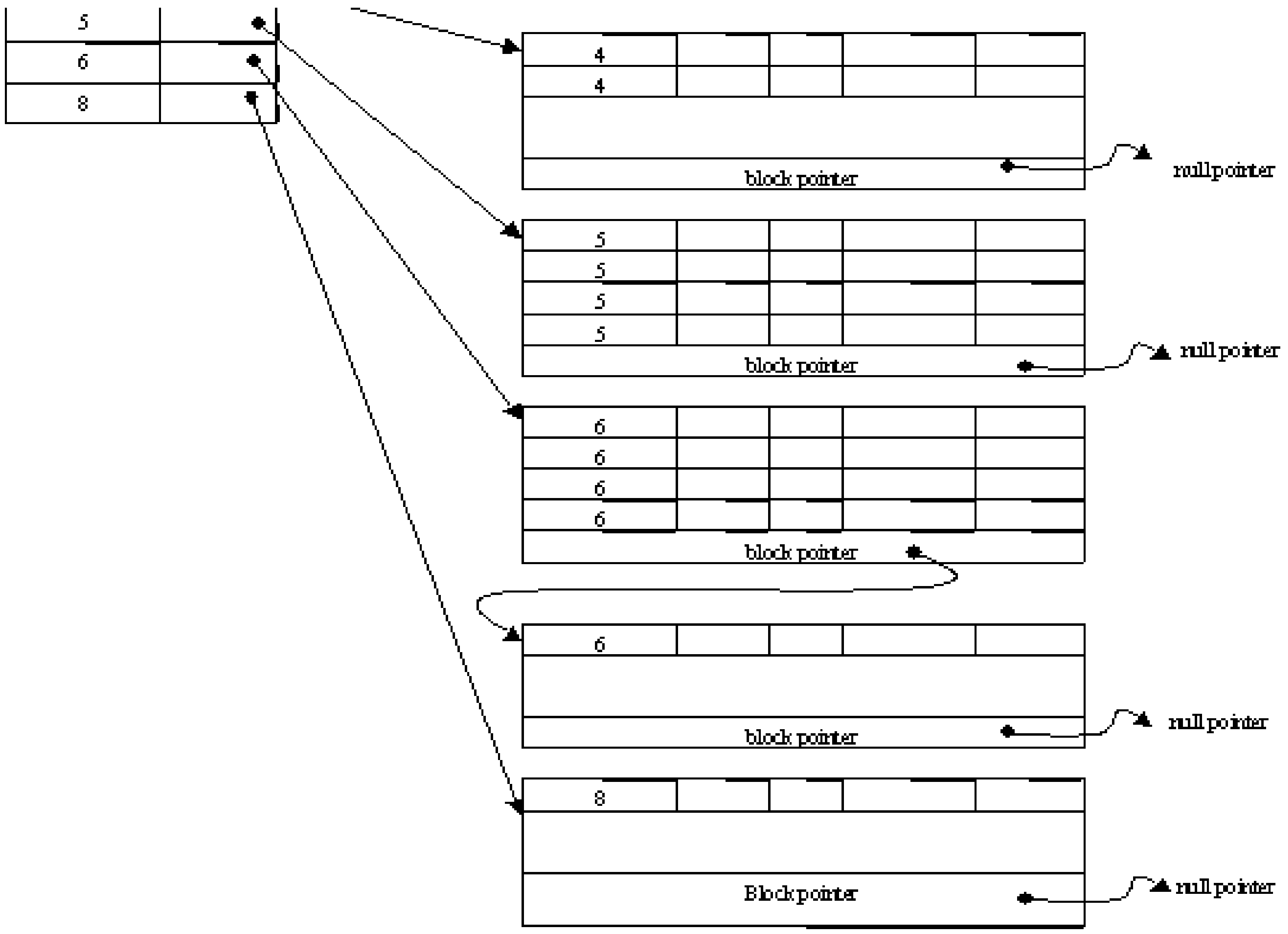
Adams, John					
Adams, Robin					
:					
Akers, Jan					

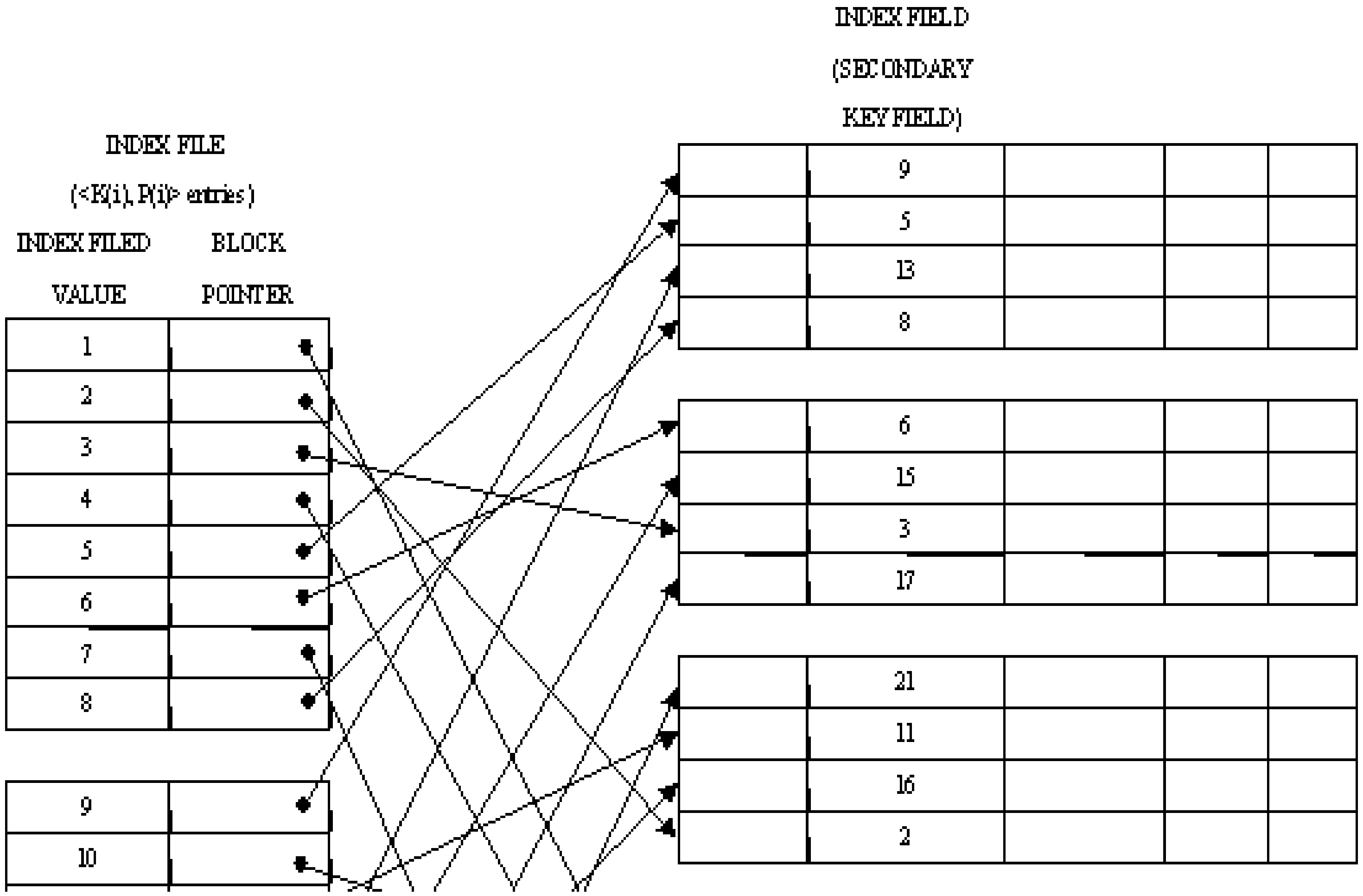
Alexander, Ed					
Anders, Keith					
:					
Allen, Sam					

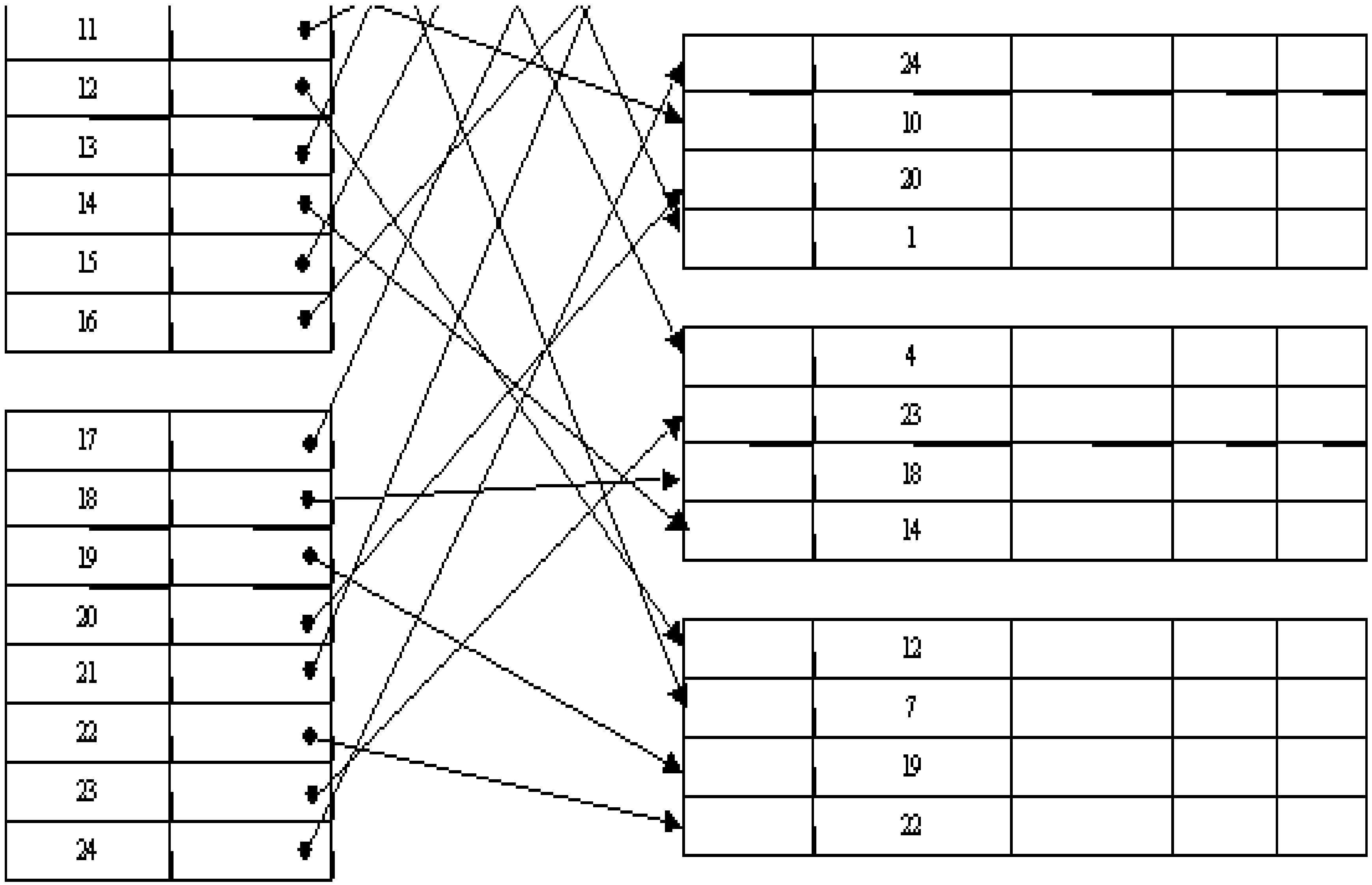
Allen, Troy					
Anders, Keith					
:					
Anderson, Rob					


Wong. James
Wright. Pam
Anderson. Zach
Angeli Joe
Archer. Sue
Arnold. Mack
Arnold. Steven
Atkins. Timothy
Wong. James
Wood. Donald
Woods. Manny
Wright. Pam
Wyatt. Charles
Zimmer. Byron


(CLUSTERING FIELD)
DATA FILE
DEPT.
NAME
JOB
BIRTHDATE
SALARY
1
1
1
block pointer
null pointer
2
2
block pointer
null pointer
INDEX FILE
(<K(i), P(i)> entries)
3
3
3
3
block pointer
CLUSTERING FIELD VALUE
BLOCK POINTER
1
2
3
4
3
block pointer
null pointer


5
6
8
4
4
block pointer
null pointer
5
5
5
5
block pointer
null pointer
6
6
6
6
block pointer
6
block pointer
null pointer
8
Block pointer
null pointer


INDEX FIELD
(SECONDARY
KEY FIELD)
INDEX FILE
(<K(i), P(i)> entries)
INDEX FILED
VALUE
BLOCK
POINTER
1
2
3
4
5
6
7
8
9
10
9
5
13
8
6
15
3
17
21
11
16
2


11
12
13
14
15
16
17
18
19
20
21
22
23
24
24
10
20
1
4
23
18
14
12
7
19
22

Trees

- Because a single-level index is an ordered file, we can create a primary index to the index itself.
- An index is like a file with unpinned records. (can move around)
- An index of an index file. (more efficient but need space).
-
- A hierarchy of indexes. (a tree)
- Several disk units --> several cylinders inside a disk unit --> tracks --> blocks.
- Good: database can exist for a long time without reorganization of storage.
- When the first level index exceeds the capacity of a block, we add another level of index.

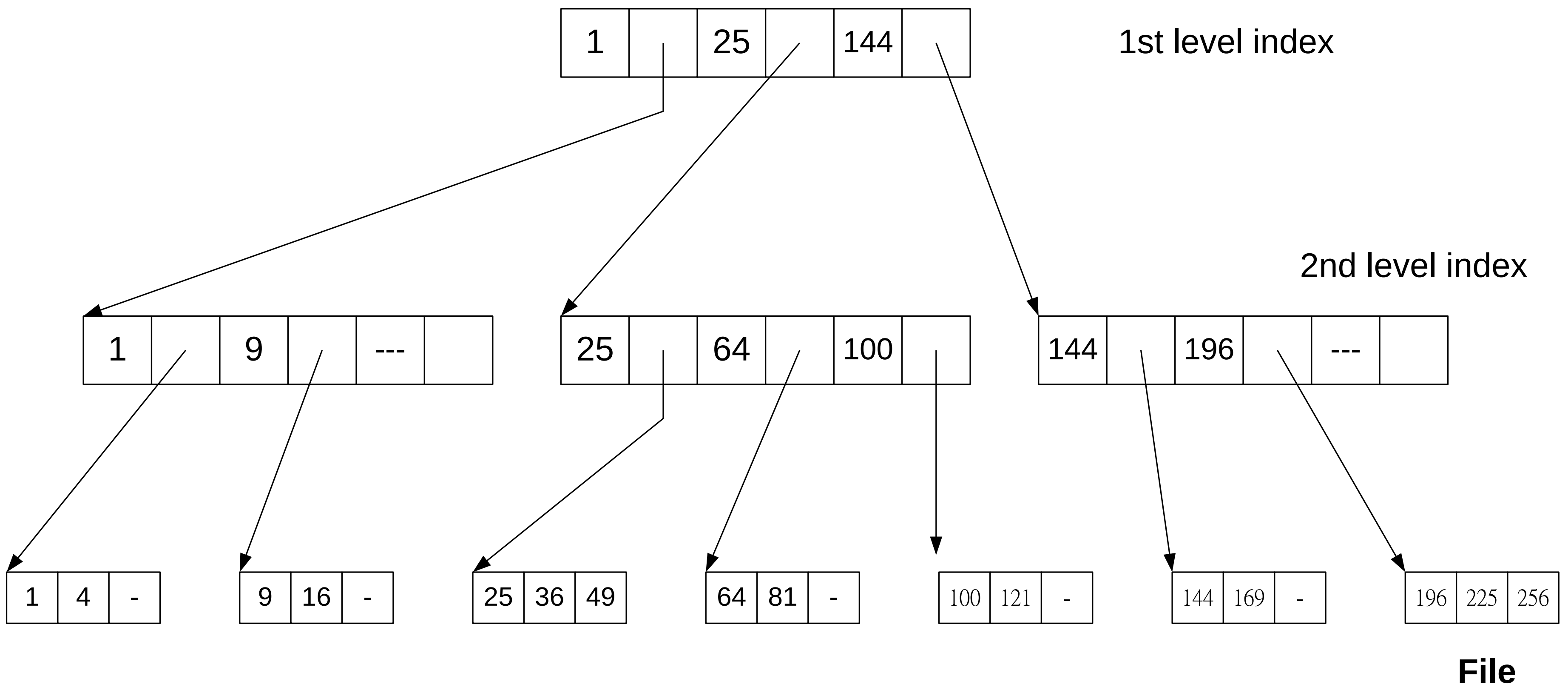
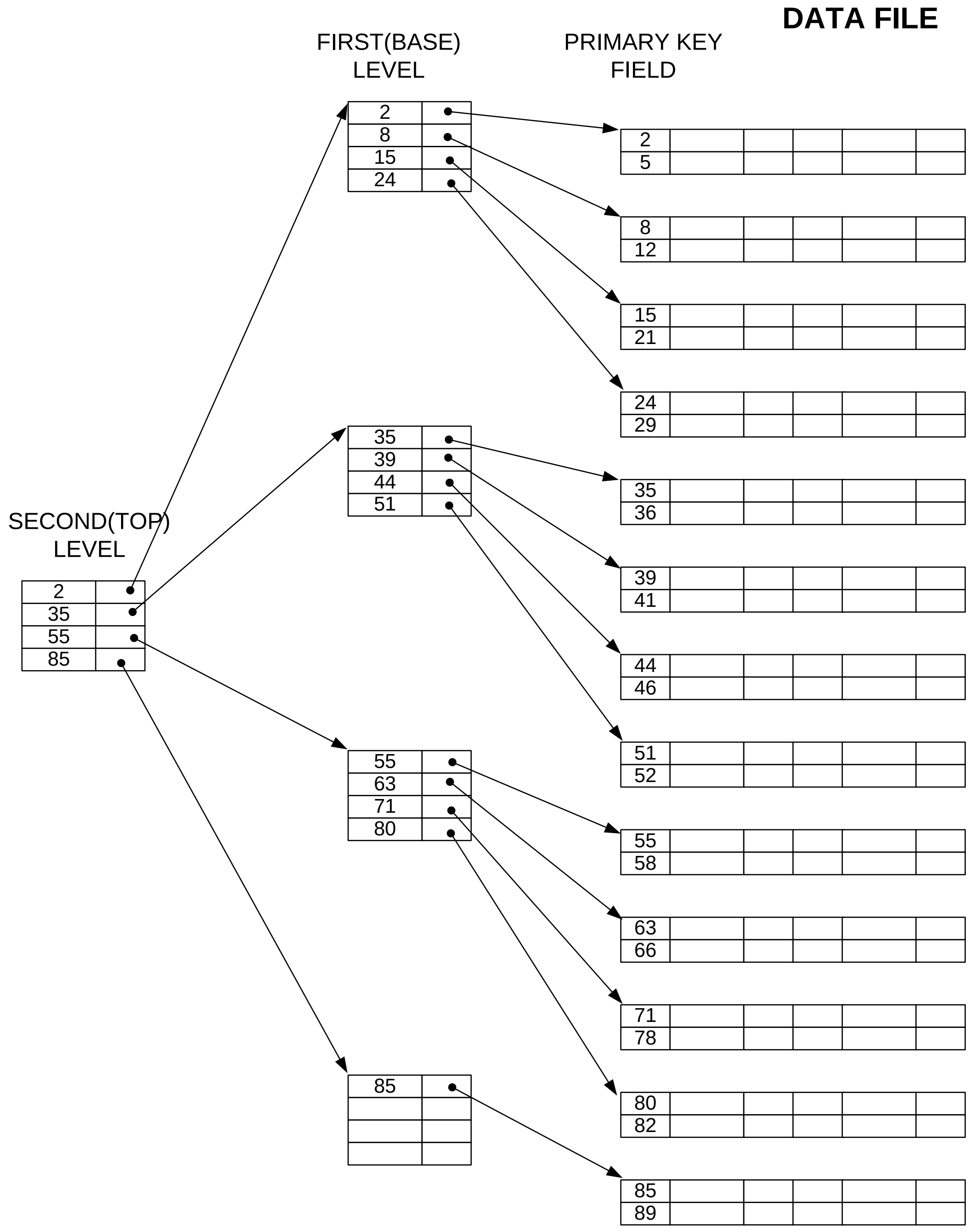


Fig. index Hierarchy



TWO-LEVEL INDEX

B-tree (balanced tree):

- A tree structure with an unspecified number of levels.
- The tree is balanced.
- Each node is kept between half-full and completely full!!
- Each path from the root to the leaf is of the same length.
- Records of the main file are unpinned, so the blocks of the main file as part of the B-tree.
  (pinned approach is also OK, in this case, B-tree contains pointers to those pinned records)
- Do not use space as efficient as possible (it wastes space). However, conceptual simplicity.
- Use particular <u>insertion/deletion strategy</u> that ensure no node, expect possibly the root, is less than half full.

Formal Definition of B-Tree Properties

- Given these definitions of order and leaf, we can formulate a precise statement of the properties of a B-tree of *order m*:
- The maximum number of pointers that can be stored in a node is called the order of the B-tree.
  1) Every block has a maximum of *m* descendents.
  2) Every block, except for the root and the leaves, has at least $\lceil m/2 \rceil$ descendents.

3) The root has at least two descendents (unless it is a leaf).

4) All the leaves appear on the same level.

5) A nonleaf block with *k* descendents contains *k-1* records.

6) A leaf block contains at least $\lceil m/2 \rceil - 1$ records and no more than *m-1* records.

- In index blocks of a B-tree, the key value in the first record is omitted, to save space.
- During lookup, all key values less than the value in the second record of a block are seemed to be covered by the first key value.

Operations

1) Lookup.
   - Search until it is a leaf.
2) Modification.
   - Deletion then insertion.
3) Insertion.
   a) If there are fewer than *m-1* records in B, simply insert the new record in sorted order in the block.
   – The new record can never be the first in block B, unless B is the leftmost leaf.

b) If there are already *m-1* records in B, create a new block B1; promote the middle record in block B (include m-1 records and new record) so that insertion can continue on the recursive ascent back up the tree; and then divide $\lceil (m-1)/2 \rceil$ records into B, $\lceil (m-1)/2 \rceil - 1$ records into B1. (ex. Insert 32)

– If the effects recursively ripple up to the root, we split the root, and create a new root with two children.

4) Deletion.

– The steps involved in deleting records from a B-tree can be summarized as follows: (ex. delete 64 and 81)

1. If the record to be deleted is not in a leaf, swap it with its immediate successor, which is in a leaf.
2. Delete the record.
3. If the leaf now contains at least the minimum number of records, no further action is required.

4. If the leaf now contains one too few records, look at the left and right siblings.
   a. If a sibling has more than the minimum number of records, redistribute.
   b. If neither sibling has more than the minimum, concatenate the two leaves and the median record from the parent into one leaf.
5. If leaves are concatenated, apply steps 3-6 to the parent.
6. If the last record from the root is removed, then the height of the tree decreases.

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree.
- In a B+-tree, all pointers to data records exists at the leaf-level nodes.
- A B+-tree can have levels (or higher capacity of search values) than the corresponding B-tree.

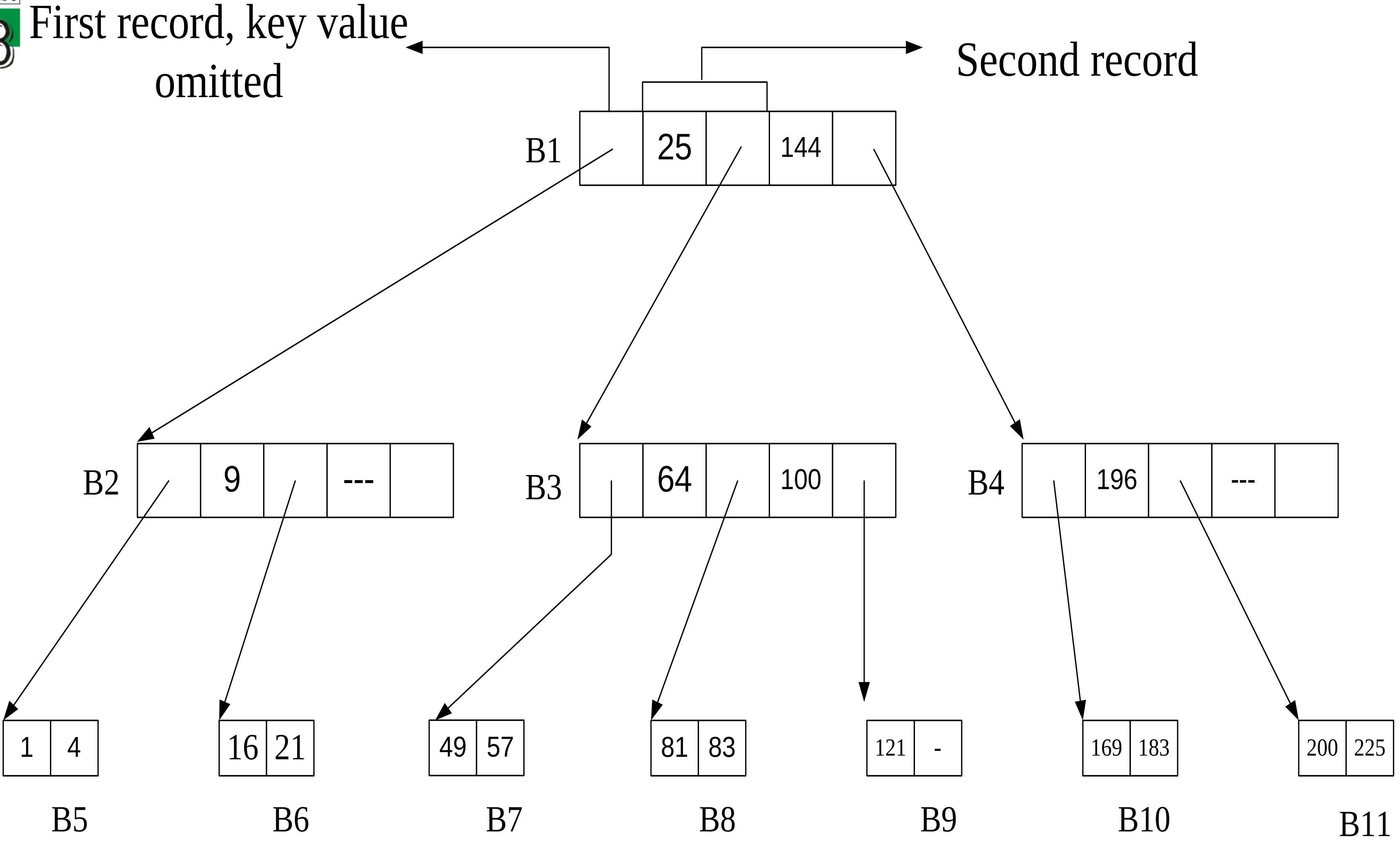


Fig. Initial B - tree (order = 3)

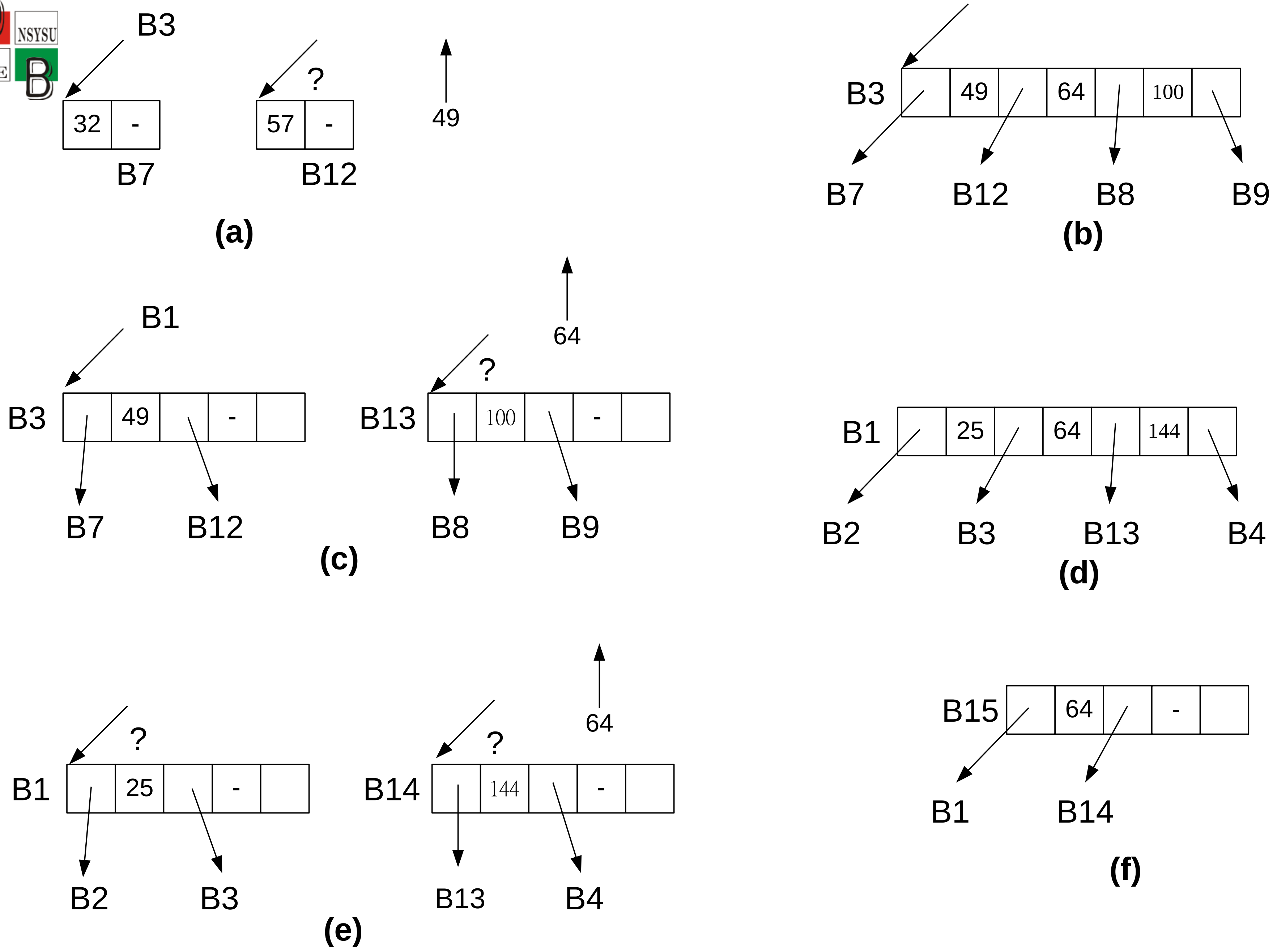


Fig. Insert 32

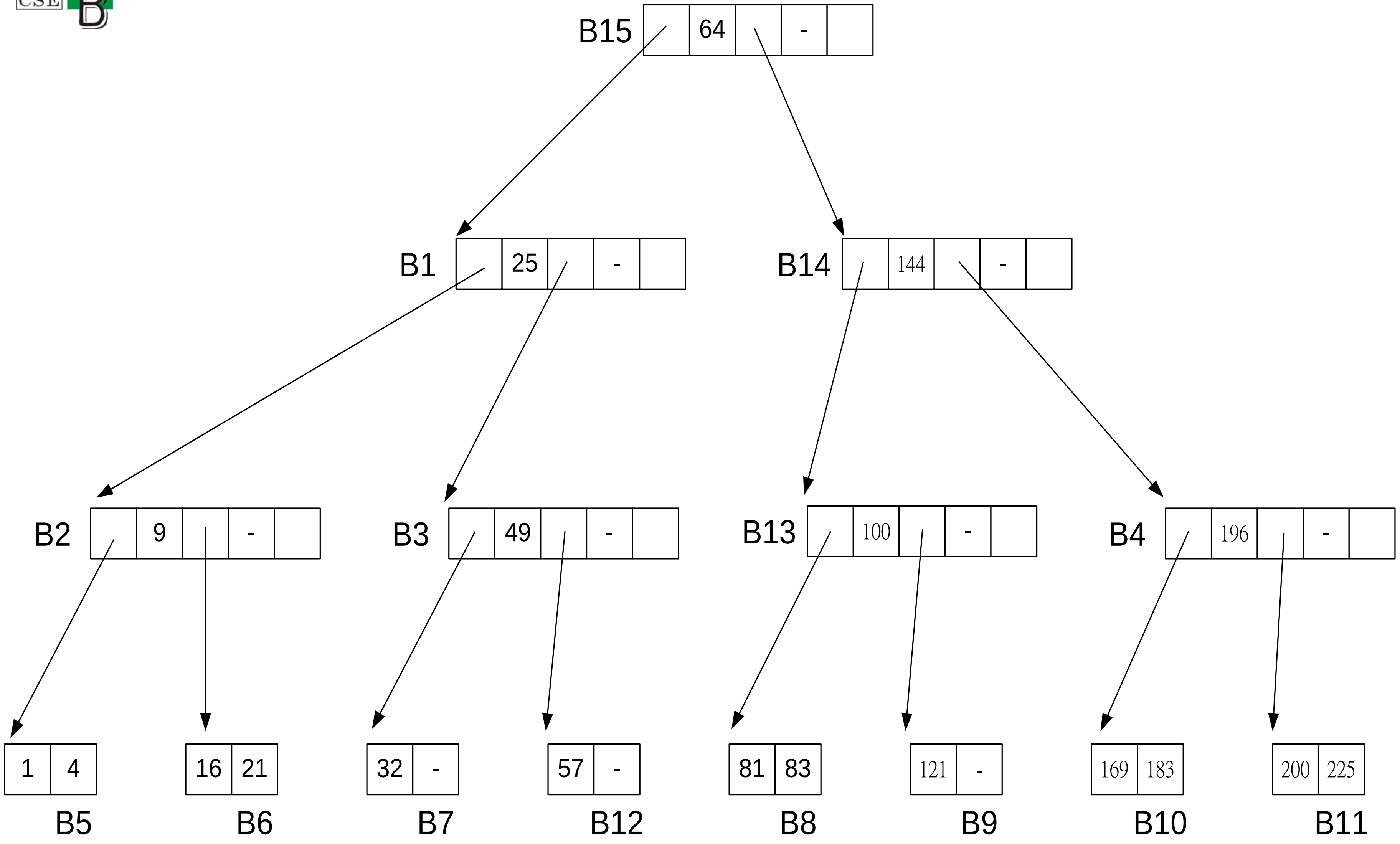
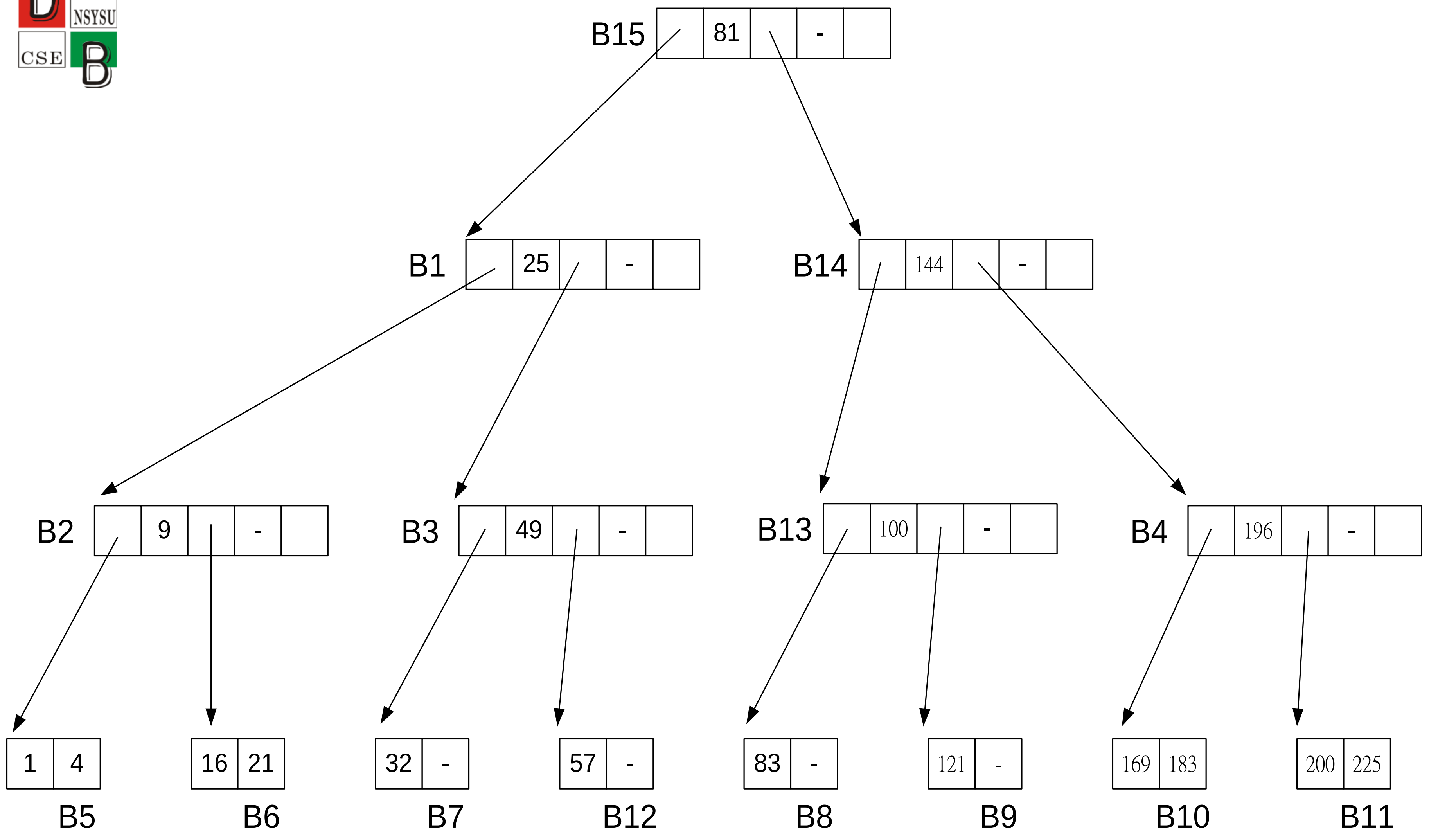
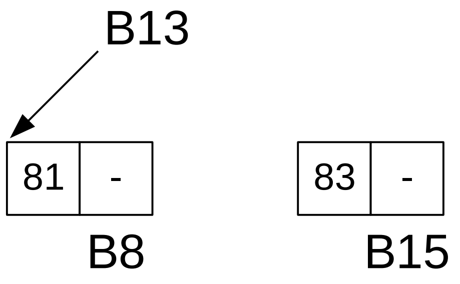


Fig. B-tree after insertion of 32

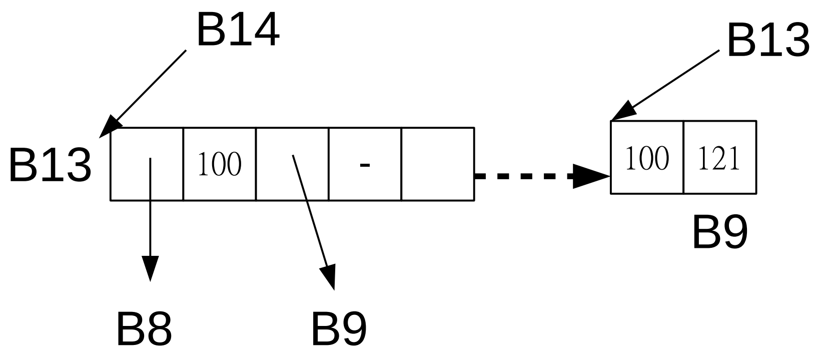


swap with immediate successor.
swap 64 in B15 and 81 in B8, and then delete 64.

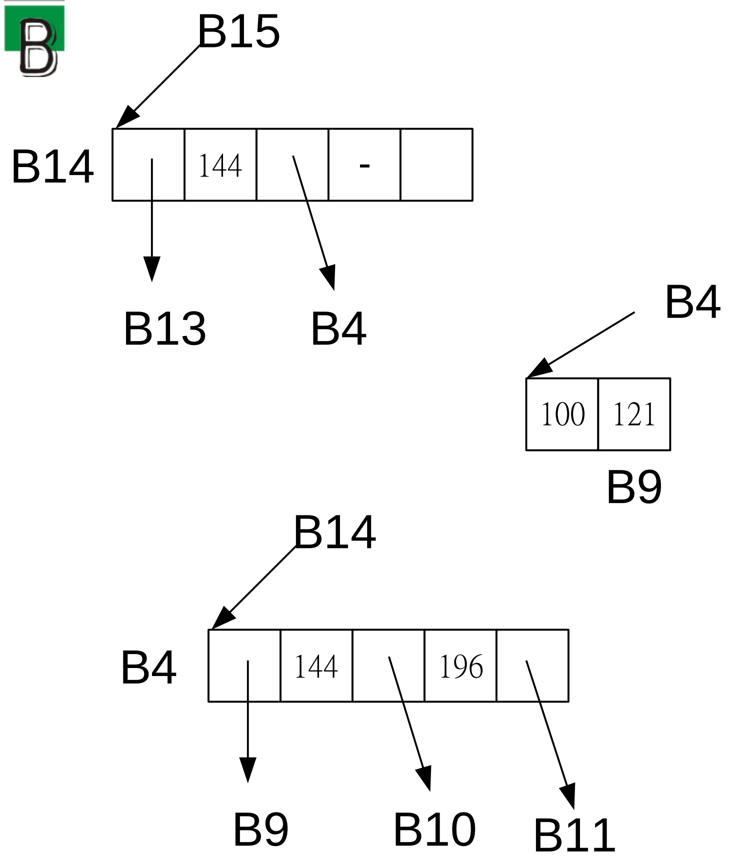
Fig. B-tree after deletion of 64



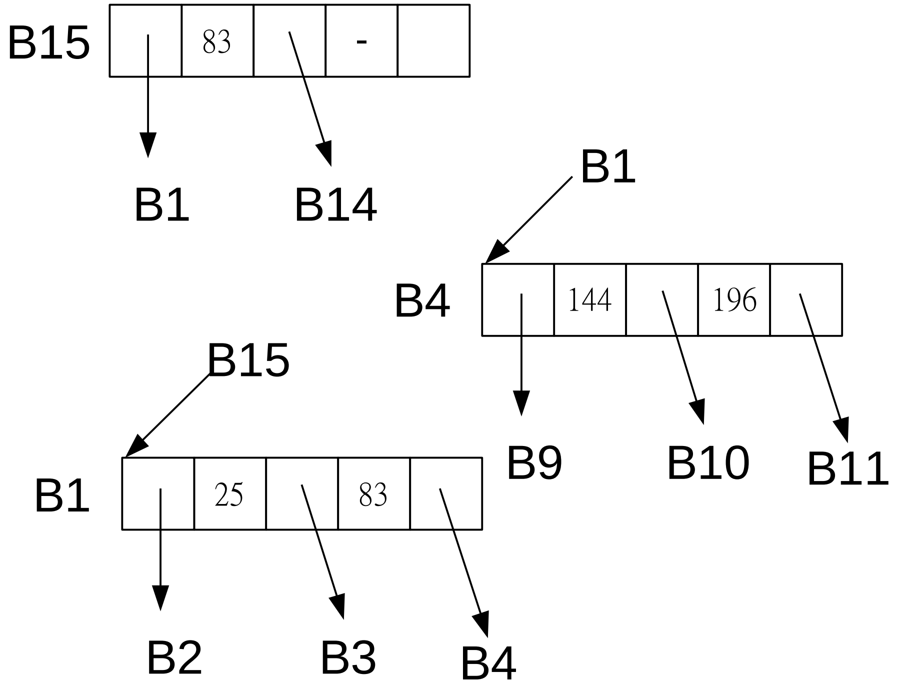
(a) swap 81 and 83, and then delete 81



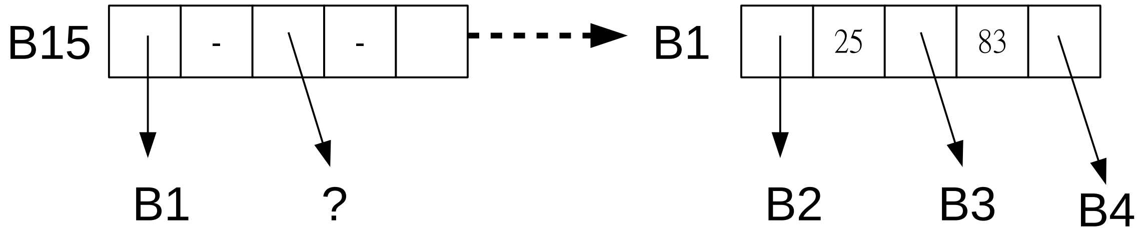
(b) underflow occurs in B8, it cannot be addressed by redistribution. Concatenate B8, B9 and B13 into one block B9, and then delete B8.



(c) underflow occurs in B13, it cannot be addressed by redistribution. Concatenate B4, B13 and B14 into one block B4, and then delete B13.



(d) underflow propagates upward. B14 has underflow. Again we cannot redistribute. we concatenate B1, B14 and B15 into one block B1, and then delete B14.



(e) Since the root B15 contains no keys, it is absortbed into the new root B1.

Fig. delete 81

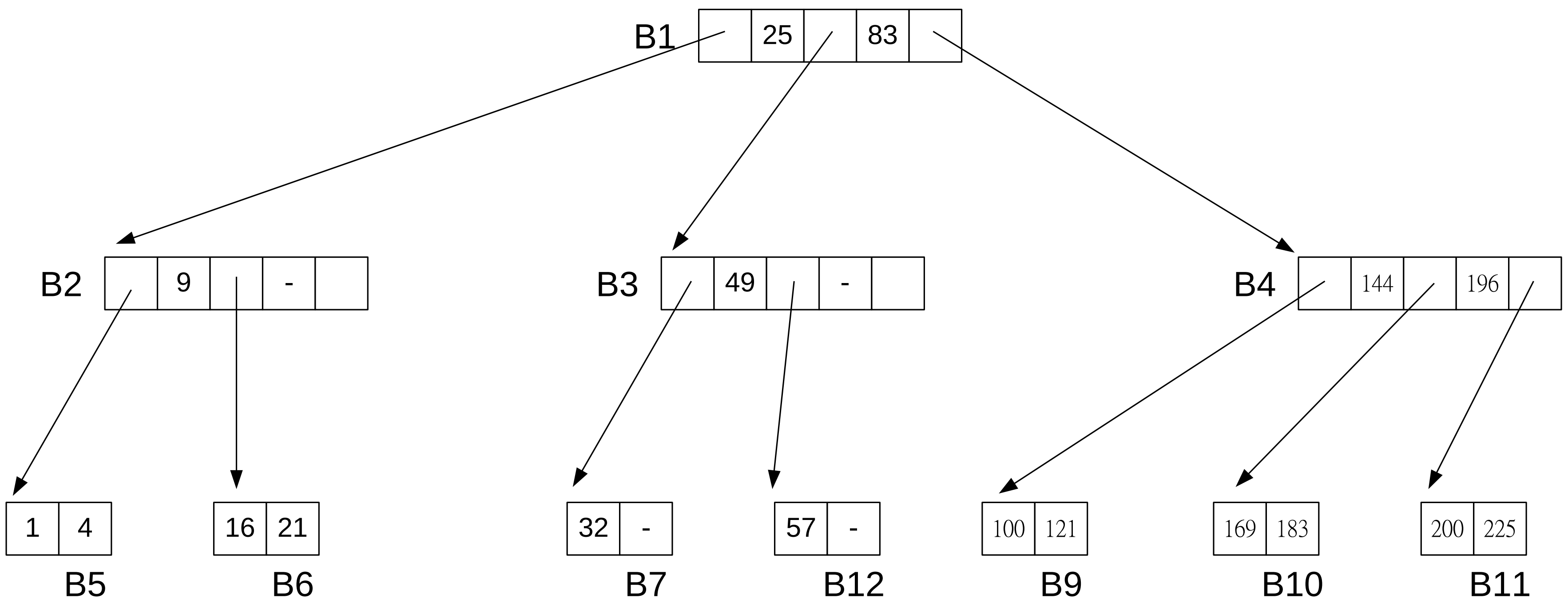
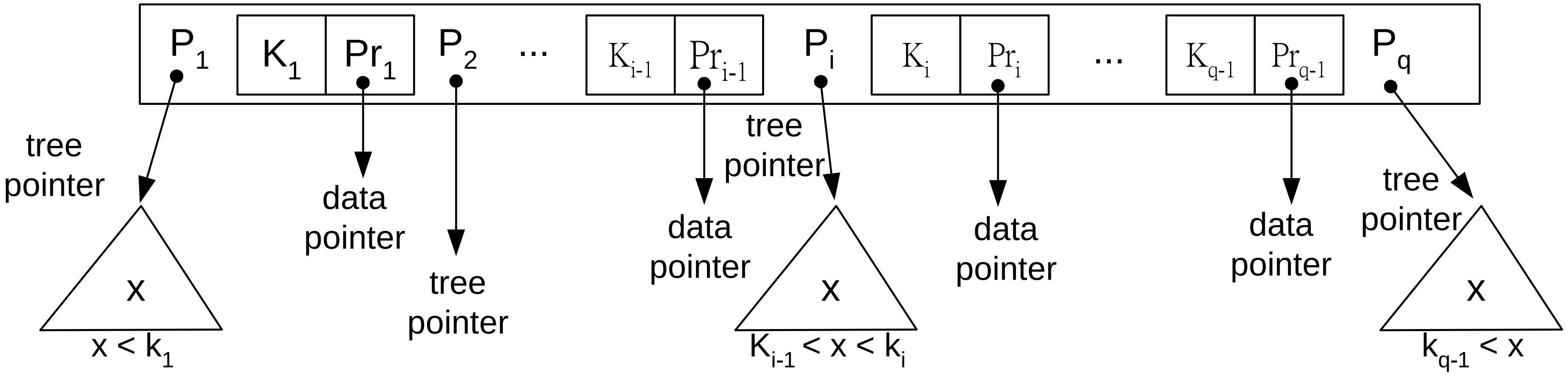
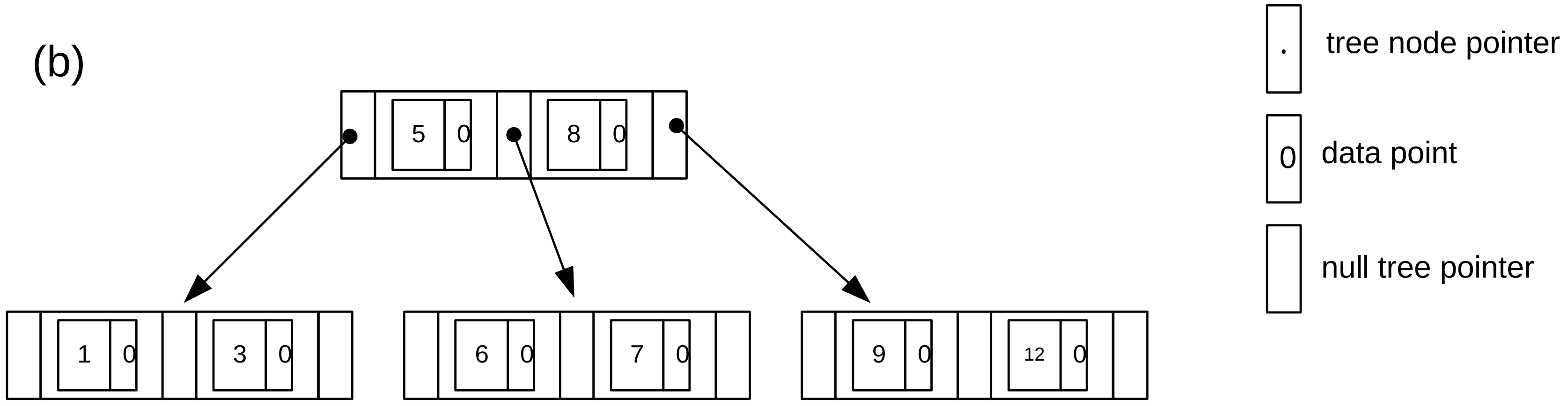


Fig. B-tree after deleting 81

(a)



(b)



B -tree structures.(a) A node in B-tree with q - 1 search values.
(b) A B-tree of order = 3. The values were inserted in the order 8,5,1,7,3,12,9,6.

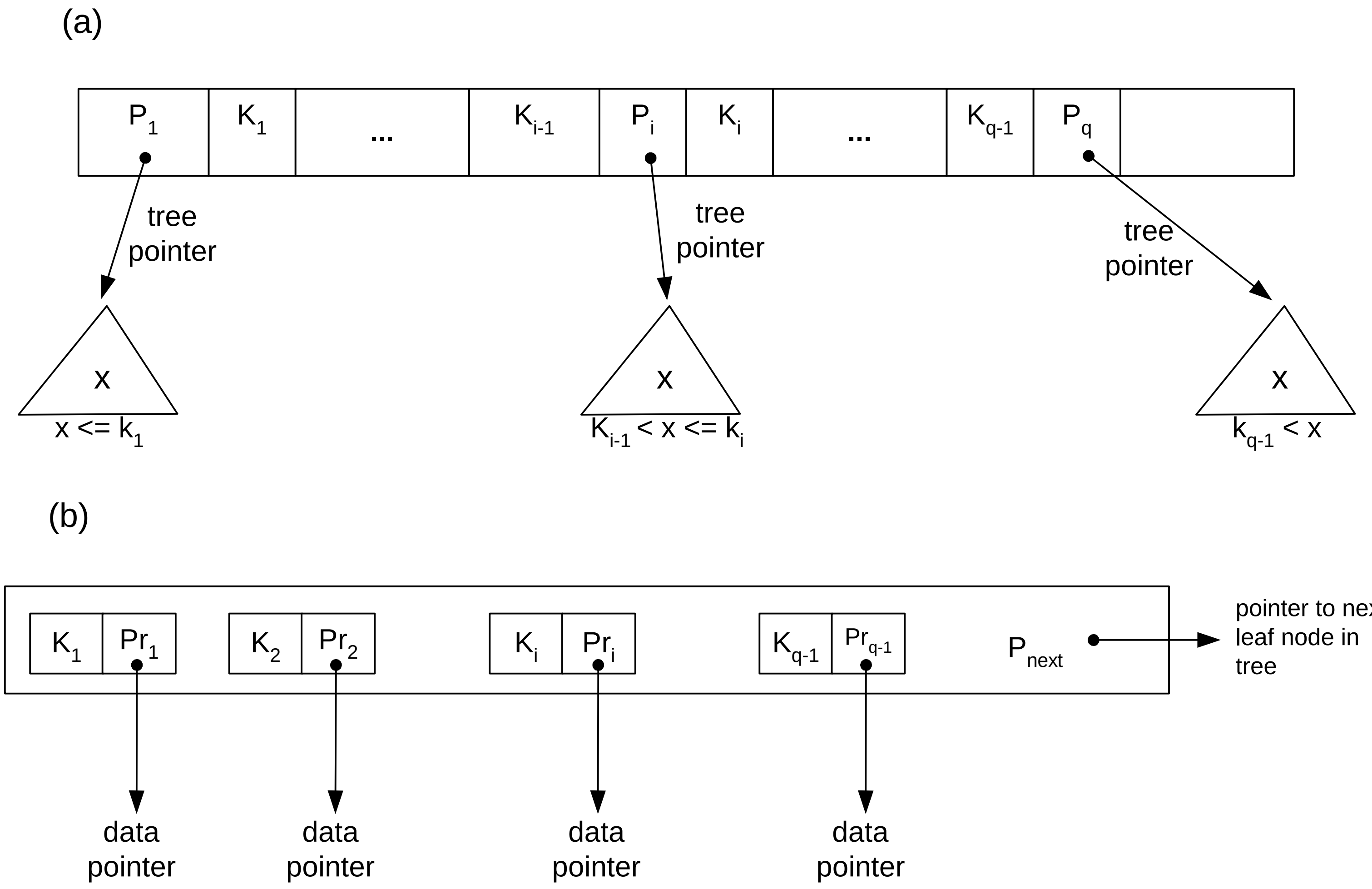
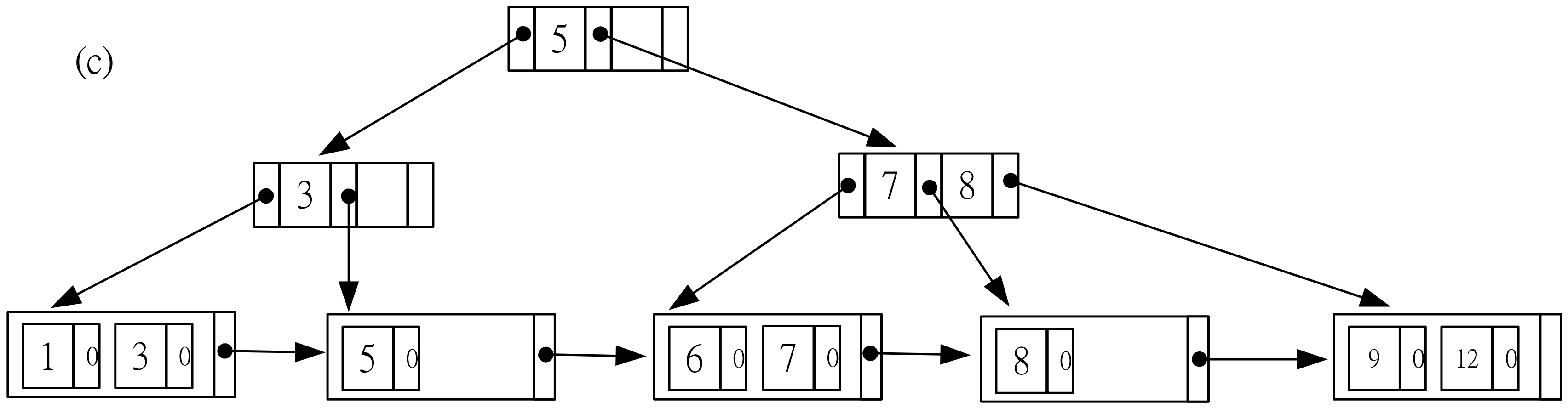


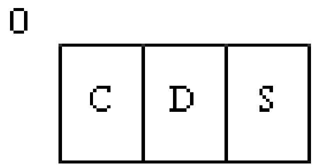
Figure Illustrating the nodes of a B+-tree. (a) Internal node of a B+-tree with q -1 search values. (b) Leaf node of a B+-tree with q -1search node.



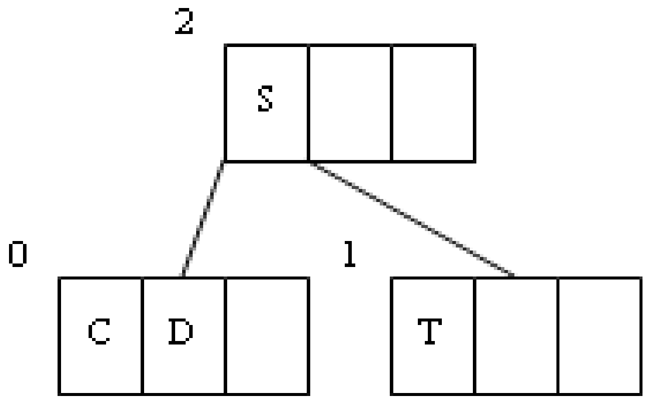
(c)order = 3. The values were inserted in the order 8,5,1,7,3,12,9,6.

Growth of a B-tree (order = 4).

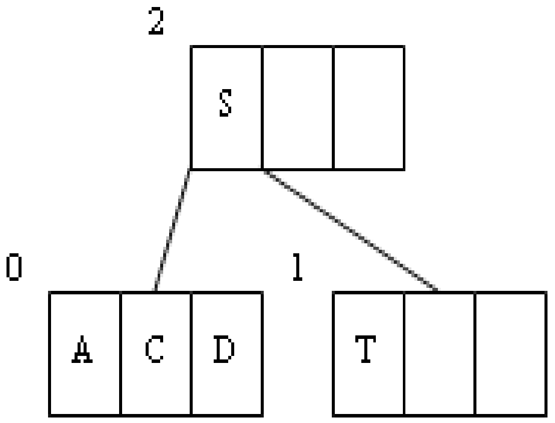
Insertion of C, S, and D into the initial page



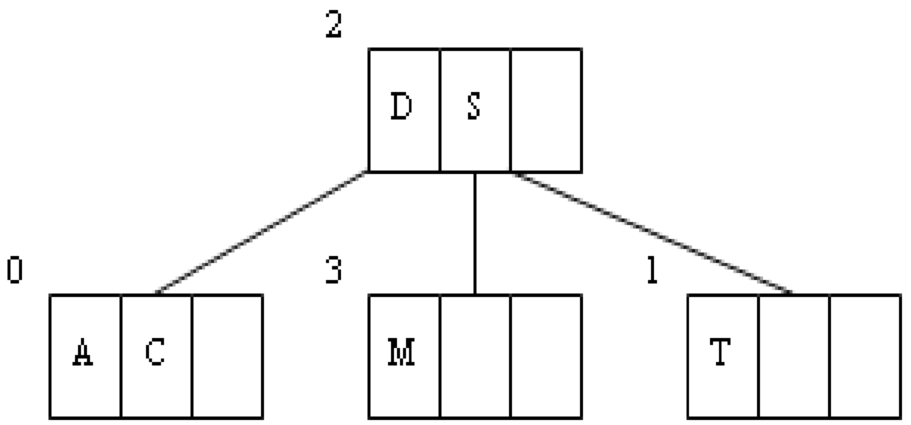
Insertion of T forces the split
and the promotion of S:



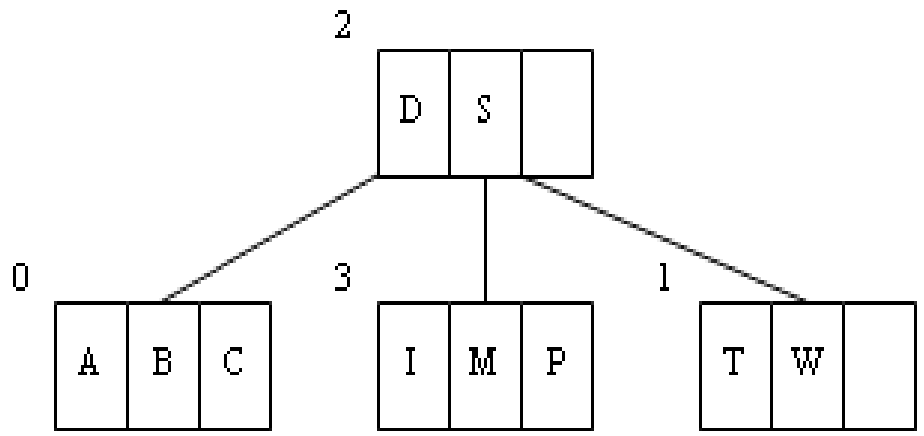
A added without incident:



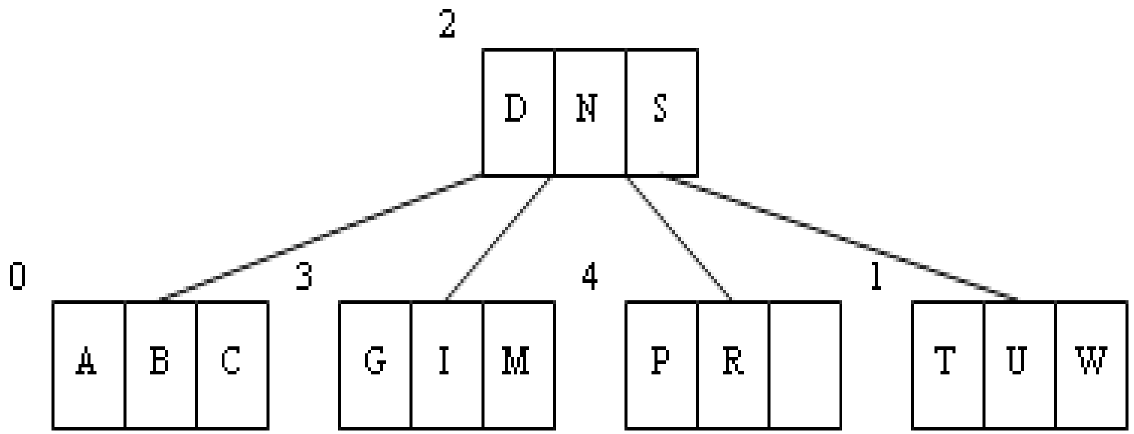
Insertion of M forces another split and the promotion of D:



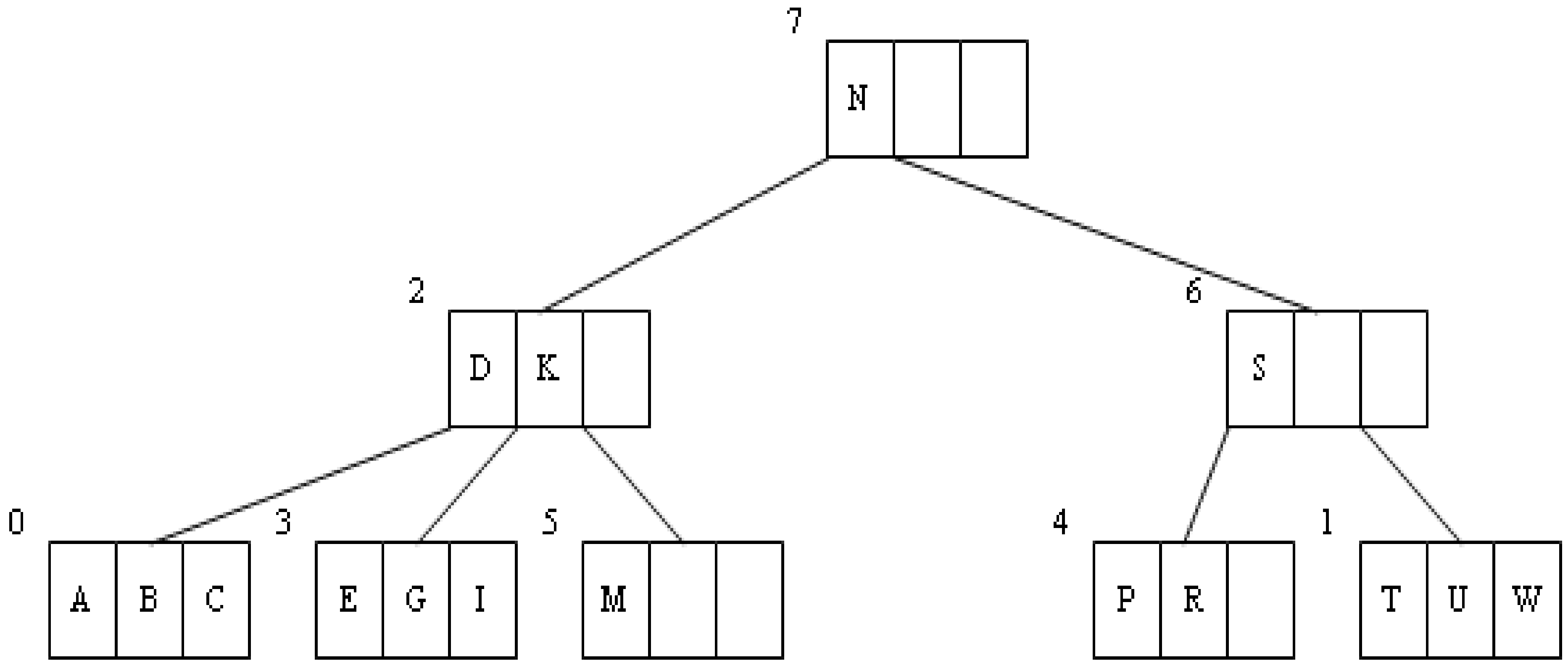
P, I, B, and W inserted into existing pages:



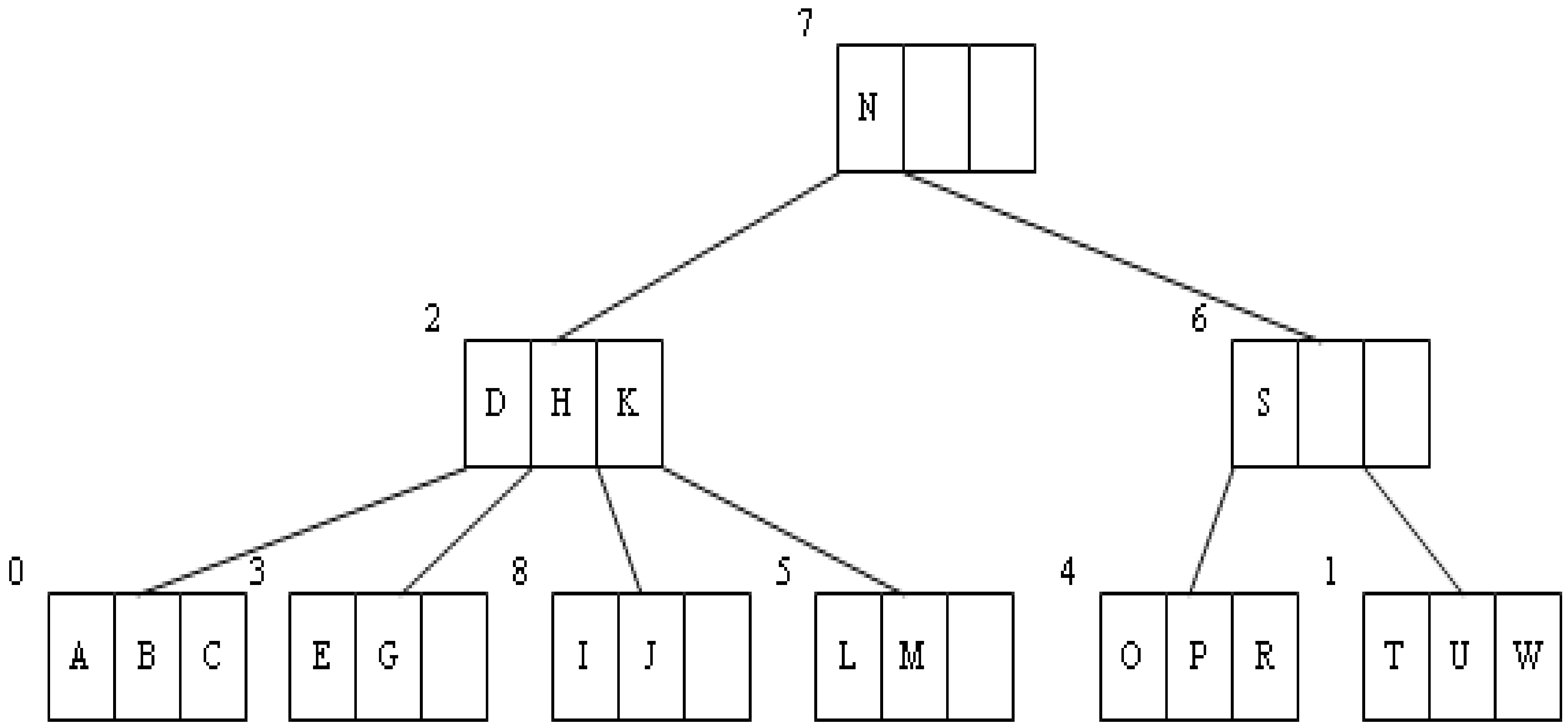
Insertion of N causes another split, followed by the promotion of N. G, U, and R are added to existing pages:



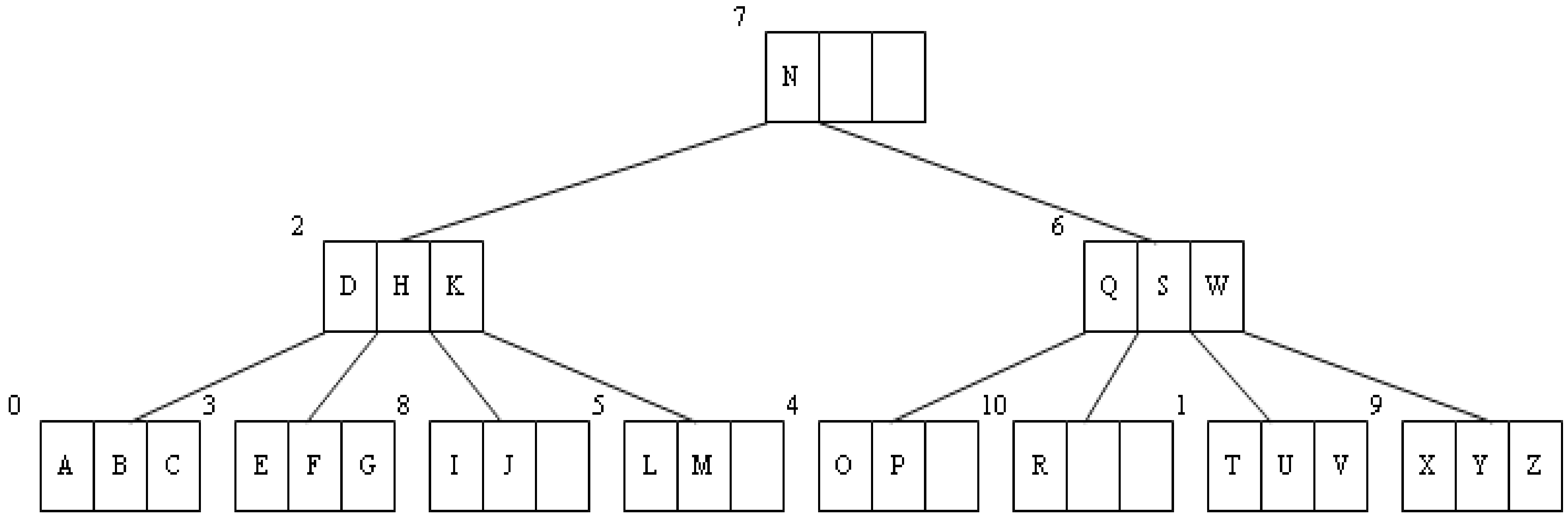
Insertion of K causes a split at leaf level, followed by the promotion of K. This causes a split of the root. N is promoted to become the new root. E is added to a leaf.



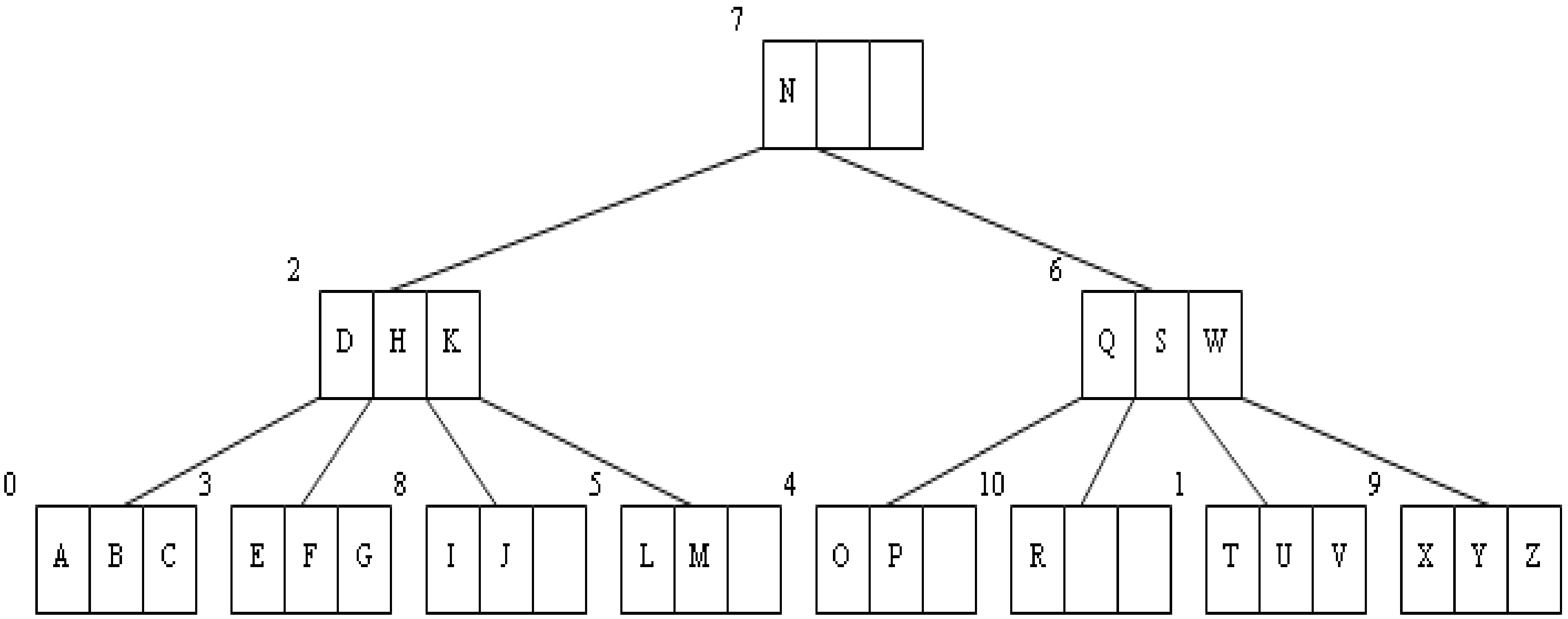
Insertion of H causes a leaf to split. H is promoted. O, L, and J are added:



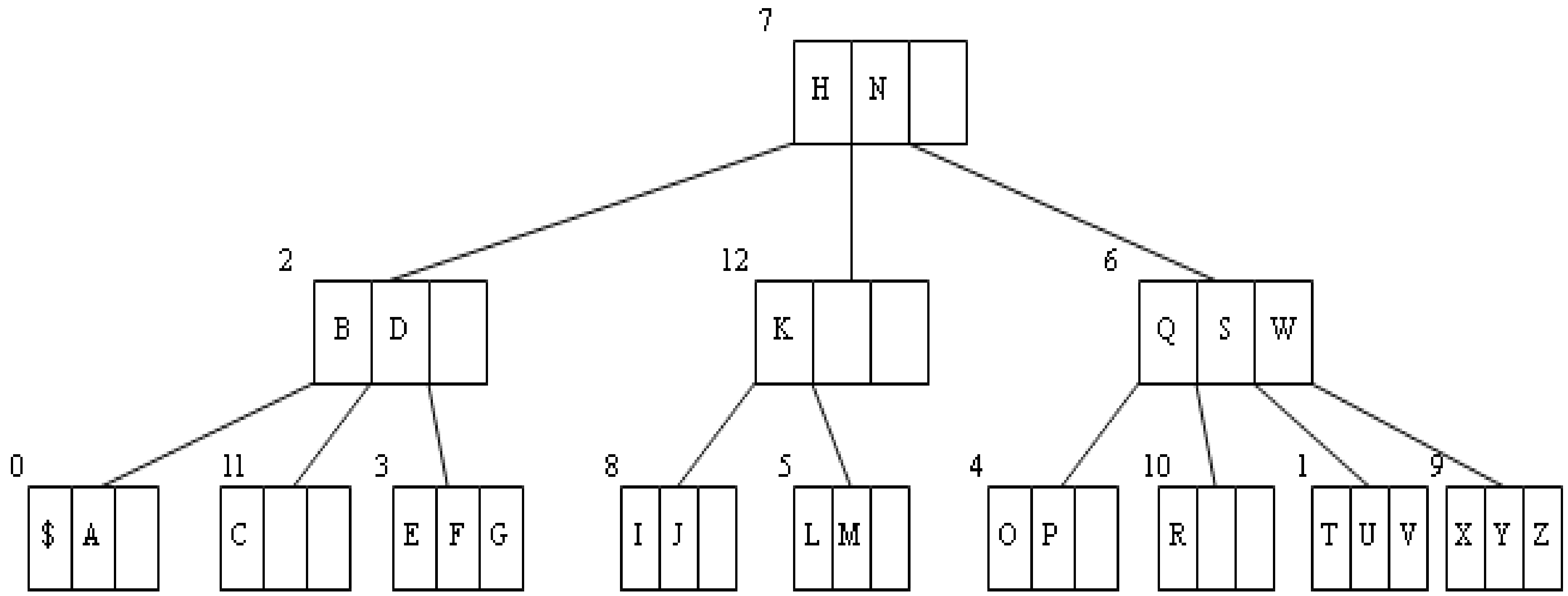
Insertion of Y and Q force two more leaf splits and promotions. Remaining letters are added:



Before inserting $:

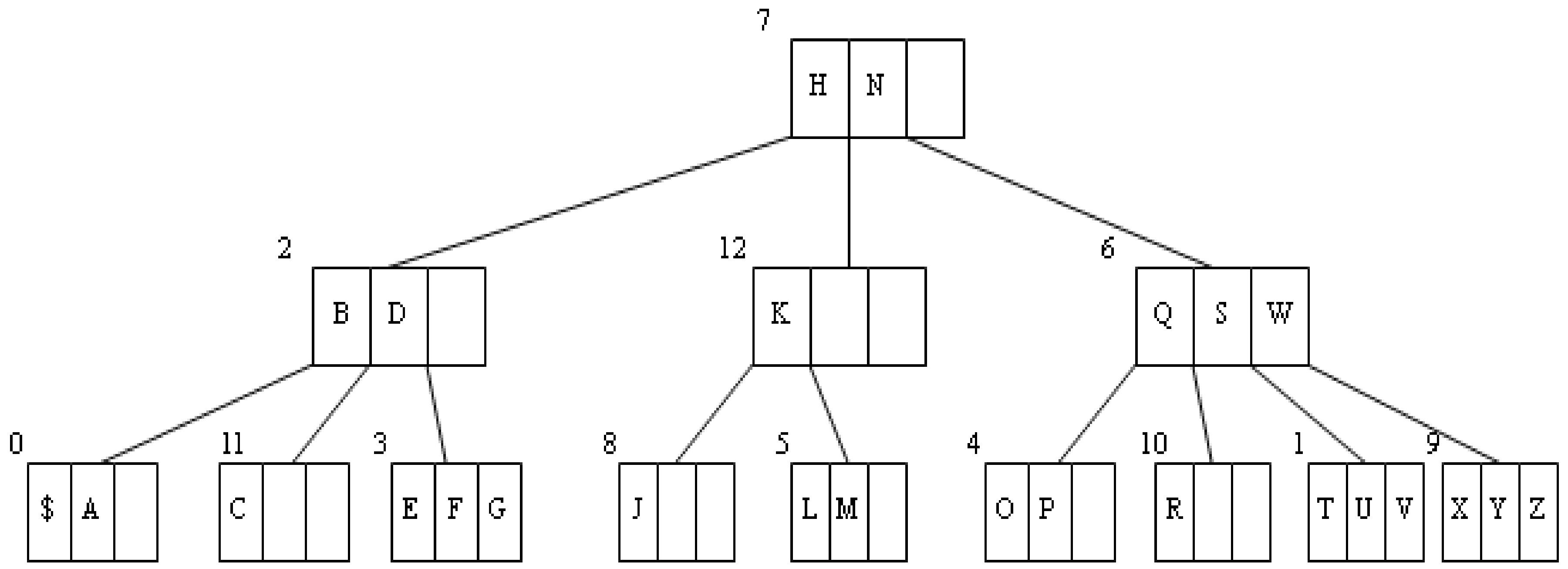


After inserting $:



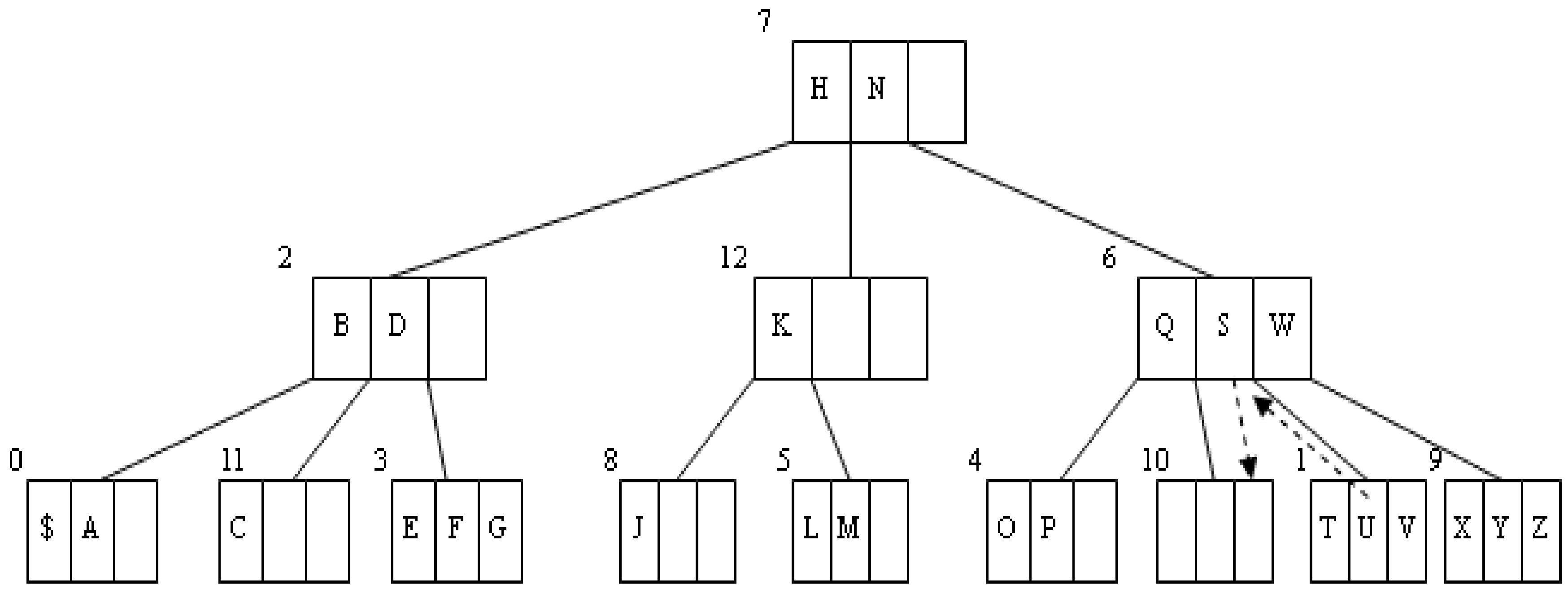
After deleting I:

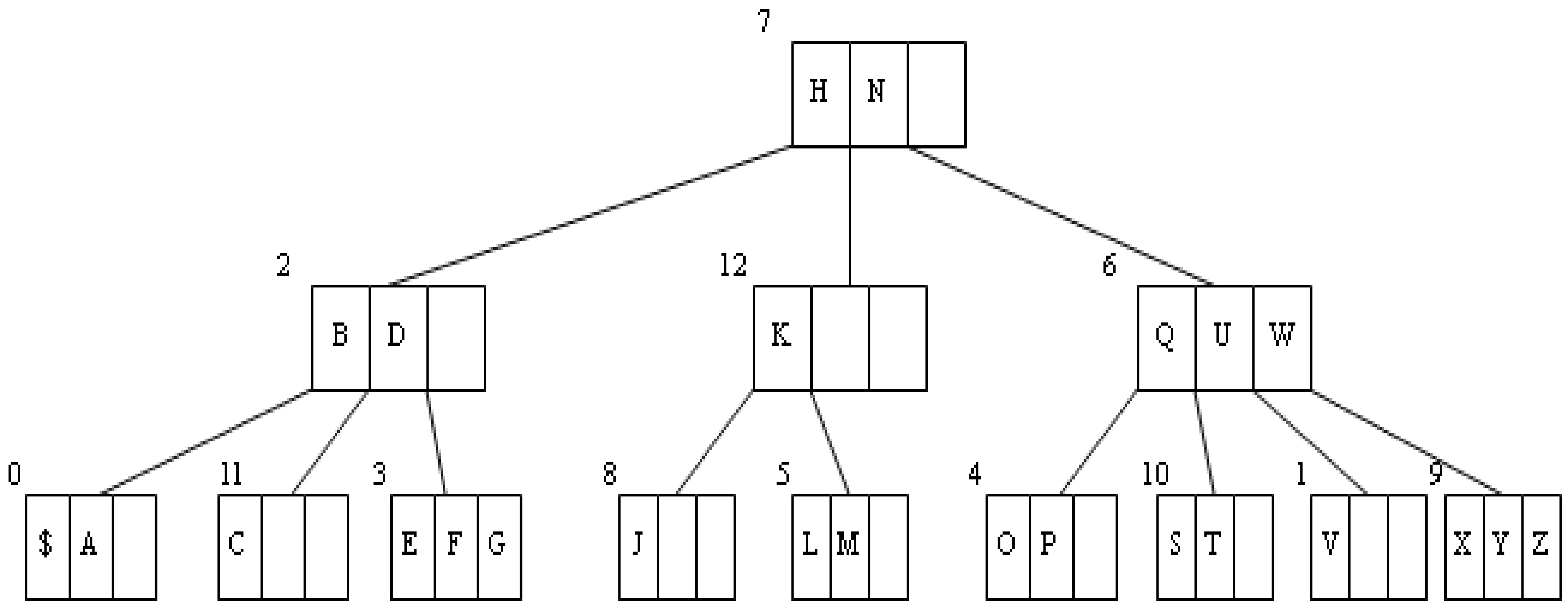
Since Page 8 has key J, key I can be removed without recognition.

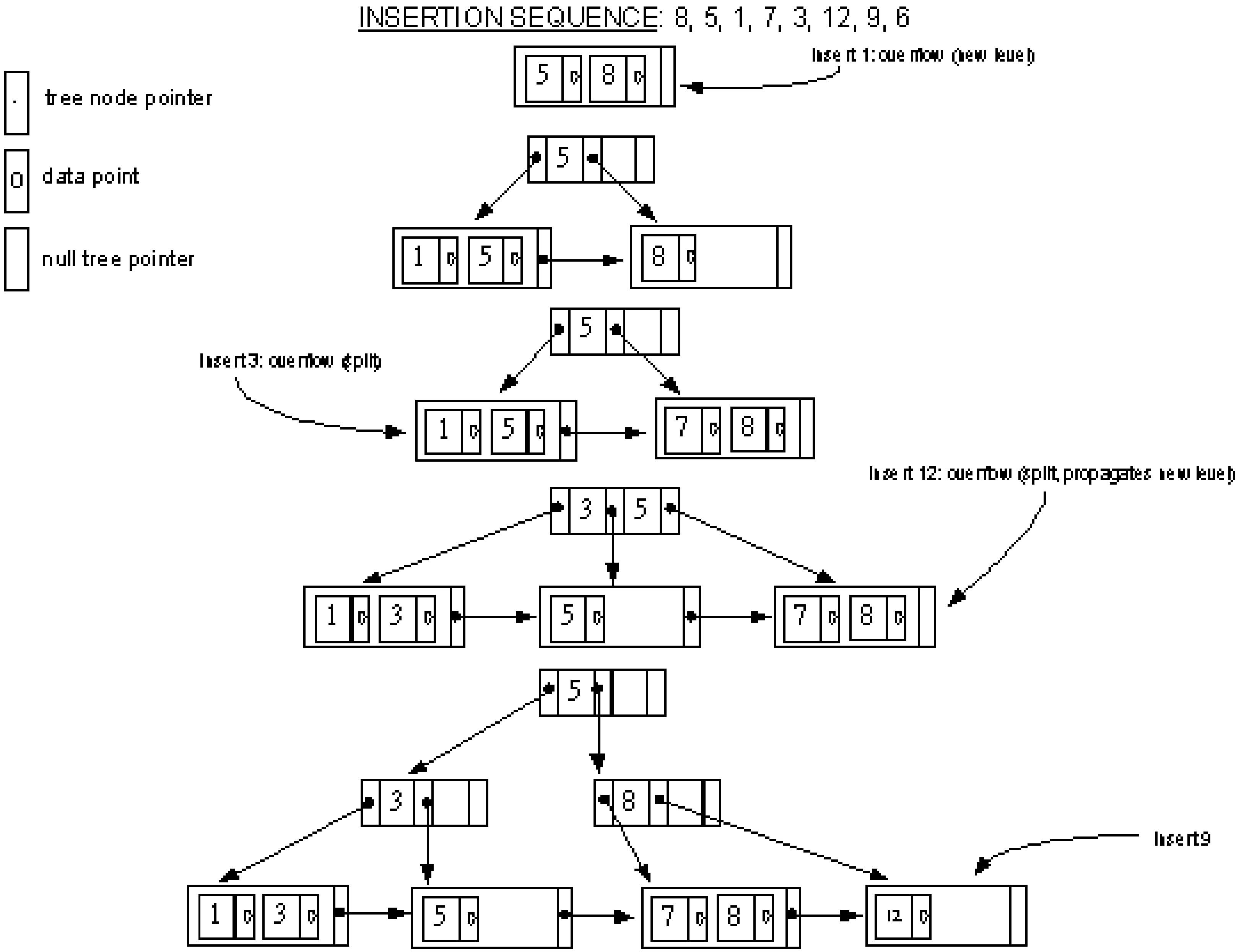


Delete R:

Underflow occurs in page 10. Redistribute keys among pages 1, 6, and 10 to restore balance between leaves.




7
H
N
2
B
D
12
K
6
Q
U
W
0
$
A
11
C
3
E
F
G
8
J
5
L
M
4
O
P
10
S
T
1
V
9
X
Y
Z


INSERTION SEQUENCE: 8, 5, 1, 7, 3, 12, 9, 6
tree node pointer
data point
null tree pointer
Insert 1: overflow (new level)
Insert 3: overflow (split)
Insert 12: overflow (split, propagates new level)
Insert 9
5 8
5
1 5
8
5
1 5
7 8
3 5
1 3
5
7 8
5
3
8
1 3
5
7 8
12

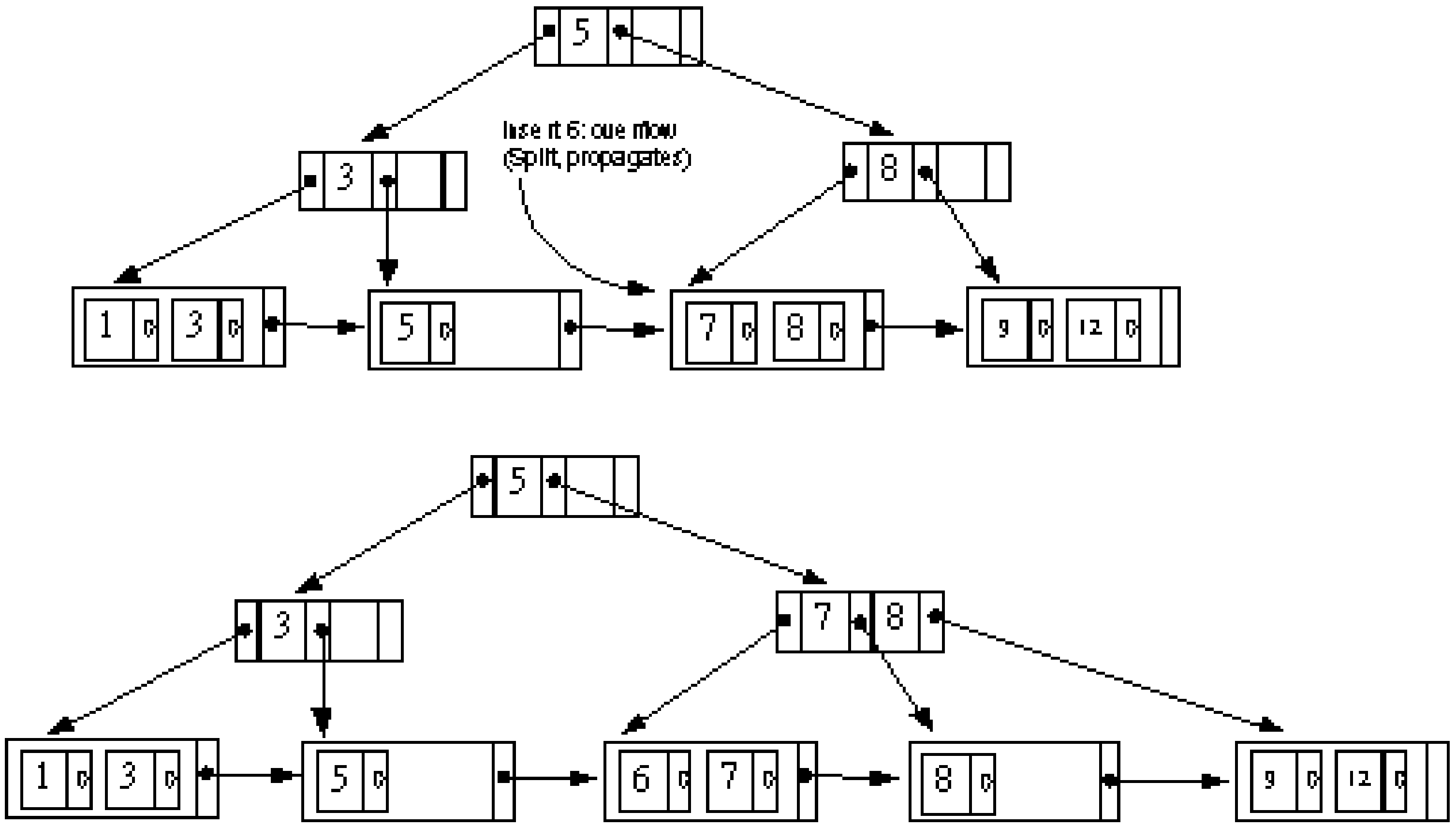
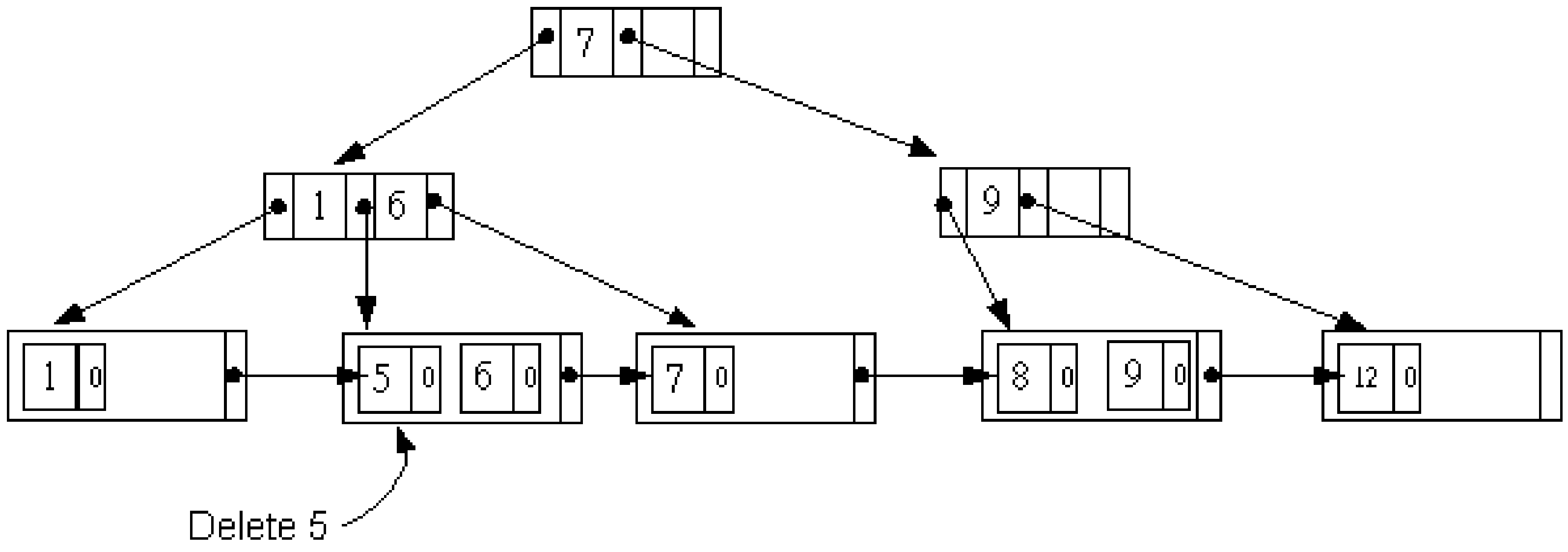
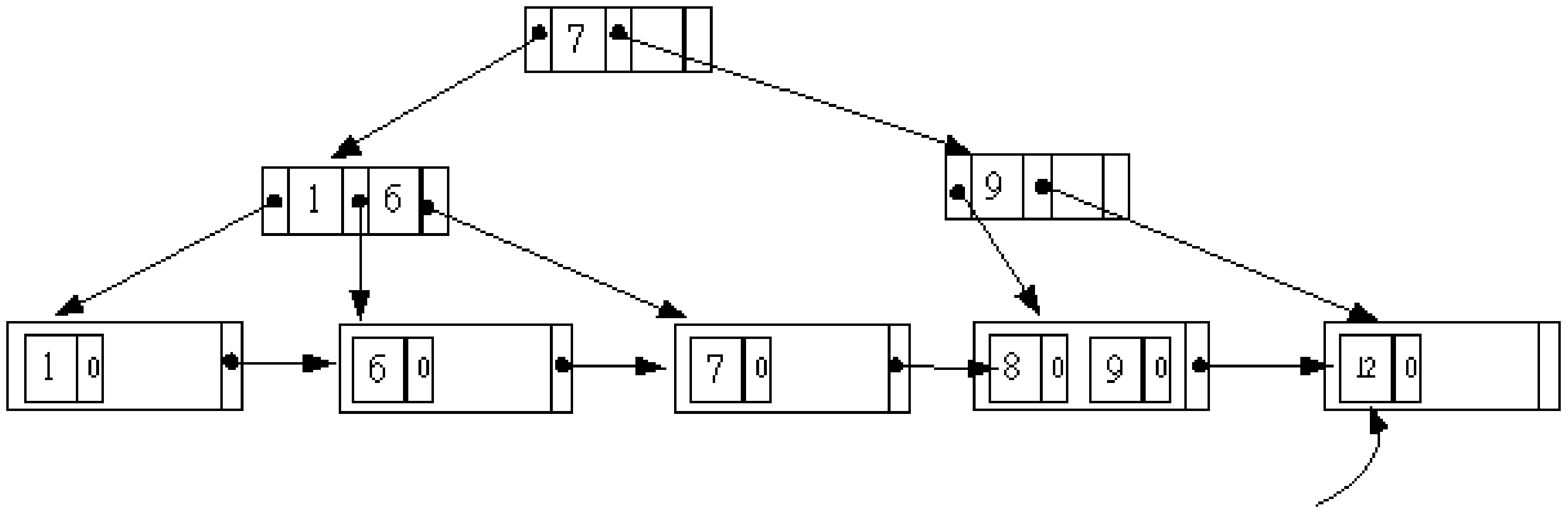


Figure 6.12 An example of insertion in a B^+-tree with $p = 3$ and $P_{leaf} = 2$

DELETION SEQUENCE:5, 12, 9





Delete 12: underflow (redistribute)

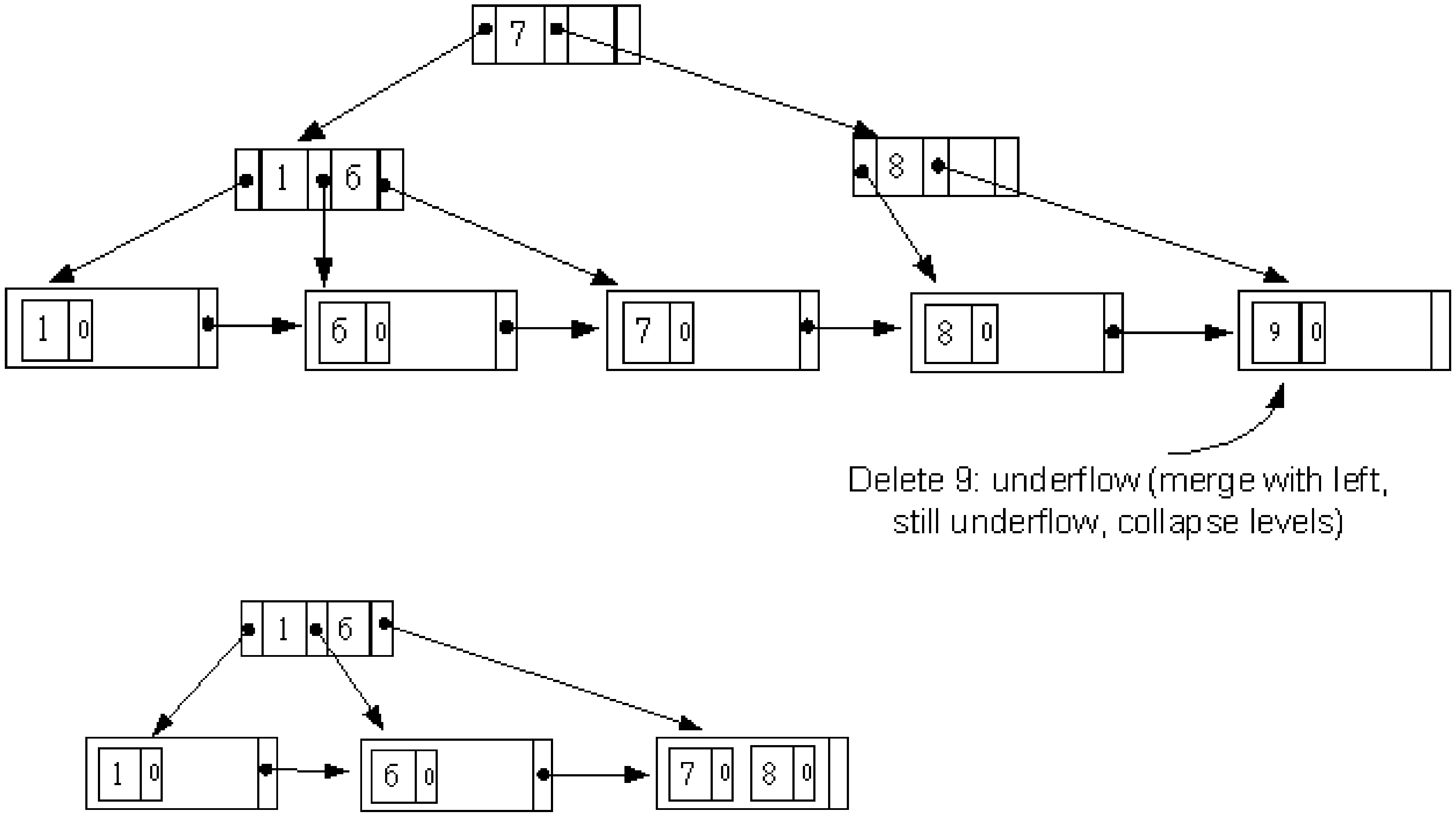

7
1
6
8
1 0
6 0
7 0
8 0
9 0
Delete 9: underflow (merge with left, still underflow, collapse levels)
1
6
1 0
6 0
7 0
8 0

Figure 6.13 An example of deletion from a B+-tree

A Dense Index

- Consists of index (v, p) for each key value v in the main file, where p is a pointer to the main file record having key value v (a certain block).
- Good for unsorted file; allow records to appear in a random order.
- Lookup, delete and modify operations are easy.
- Insertion: add the record after the end of the last record and insert a pointer to that record in the dense index file.
- Always require two more accesses for the main file.
- But has a compact storage organization.

A comparison of hashed organization and index organization.

Hash	Index
Fast	less fast
no sequential access	sequential access
organize search space. (n buckets)	organize data
not suitable for range query	good for range query
must consider collision problem (long search inside a bucket)	
space for hash table (value, pointer)	space for index table
mapping the key space into the address space.	

Hashing Techniques for Expandable Files

- Fixed table size in hashing function causes the problem.
- When too many records mapped into the same bucket --> collision, long search. (consider overflow-handling)
- Then, it is better to reorganize the hash table --> increase bucket number and change hash function. (static table size)
- The hash functions used for mapping the key space into the address space use the table size.

- If we need to add records beyond the capacity of the table --> reorganize the table --> unload all data --> reenter them plus new data into a table with a different size and associated hashing function. --> large address space.
  (i.e., must extend hash function range)
  Therefore, it wastes time.
- How about change the table size dynamically? (dynamic table size)
- B-tree: key --> B-tree (pointer) --> access database.
- Hashing: key --> h() location in hash table --> location on disk.
- Dynamic hashing: key --> h() pseudo key (random #;index) --> h'() location in hash table --> location on disk.