B+-Tree is a multi way search tree which is used in implementing Indexed Sequential Access Method of retrieving data. B+-Tree is used in many Data Base Management Systems where a large data file requires index [48]. Teaching the concepts of file structures in Computer Science require the understanding of B+-Tree management. Using B+-Tree, records can be accessed both sequentially and randomly. B+-Tree structure is a complex structure and understanding the concept is possible only if we go through the whole process of insertions, deletions and searching. We have designed and developed a CAI package to simulate the operations insertion, deletion and searching with a set of data items in first go. Later, learners are allowed to visualize the process of insertions, deletions and searching with their own data.

1. Introduction.

One method of retrieving records from a data base is using an index. Index is a table (or directory) contains the key values and the location of records. This method is known as Indexed Sequential Access Method (ISAM) and it is introduced by IBM in early 1960's [48]. In ISAM, index is searched for the key value and if found, the corresponding record pointer is used to retrieve the record. If a data base is very large, then it is not possible to store the index in the primary memory and it is to be stored on a secondary storage device. Whenever a data base is updated, index is to be updated. Other than updation, most frequent operation

performed on an index is searching. Since index is available on a secondary storage device, this search falls under external searching. Many methods are available to manage index. Generally, efficiency of an ISAM depends on the storage space required to store the index and the number of disk accesses required to search a key value in the index. B+-Tree is known as multi way search tree because each node of B+-Tree contains more than two key and pointer pairs. B+-Trees are stored on direct access storage devices. The maximum number of key and pointer pairs in a node of a B+-Tree is known as the order of B+-Tree. Size of B+-Tree node is generally depends on the size of the block used in reading/writing data. By choosing the size of the block as the size of the node, we can read or write the node in a single disk access. However, it is not necessary to take the block size as the node size. Order of a B+-Tree will be decided using the length of the key. If the key length is smaller, then the order of B+-Tree will be more. This results in a B+-Tree of minimum levels. Number of disk accesses depend on the number of levels in a B+-Tree. If we could create a B+-Tree with less number of levels, then the number of disk accesses required to search a key will be less. Nodes in a B+-Tree are divided into two types namely non terminal nodes (or internal nodes) and terminal nodes (or leaf nodes). Root node is the first internal node in the B+-Tree. Leaf nodes contain the key values and the corresponding record addresses and they are linked in the

order of key values. This linkage of leaf nodes facilitate sequential access of key values. Internal nodes (or non terminal nodes) do not contain the record addresses. These nodes contain a pointer to the internal node or leaf node in the next level. Internal nodes facilitate to search down to locate the key value and its corresponding address. This means, all key values must appear in leaf nodes. Generally, the root node of a B+-Tree will be kept always in primary memory to reduce the number of disk accesses[21].

The format of internal node is C H K_1P_1 K_2P_2... K_jP_j, Where C is a counter that indicates total number of key values in the internal node (i.e. C=j), H is an integer that indicates height of the node, K_i, i=1...C are key values and P_i, i=1...C are pointers[48]. For each leaf node, H=1. Parent of leaf nodes have H=2. Parents of those parents have H=3, and so on. Root node contains the largest value of H which indicates the height of the B+-Tree. K_i in each node is the largest value in the node pointed to by P_i.

The leaf node format is C H K_1A_1 K_2A_2 ... K_jA_j P, where K_i, i=1...C are key values and A_i, i=1...C are addresses of records in the file[48]. P is a pointer to the next leaf node which contains next larger key value. C and H are used similarly as in internal nodes.

2. B+-Tree Properties.

In general, a B+-Tree of order m has the following properties :

1. The root has at least two children.
2. Each internal node has no more than m children.
3. Each internal node which is not a root has at least [m/2] children.
4. Each leaf node has at least [m/2] key and record address pairs and all leaf nodes must appear at same level.

3. Capacity of B+-Tree.

Consider a B+-Tree of order m and height h. Assume that root of this B+-Tree is at level 0. B+-Tree property is root node must have at least two children. Each of those two must have at least [m/2] children. Each of those children must have also at least [m/2] children and so on. The following table gives minimum number of nodes that must be present at each level.

Level Number	Minimum Number of Nodes
0	1
1	2
2	2*[m/2]
3	2*[m/2]*[m/2]
4	2*[m/2]*[m/2]*[m/2]
..	.....

Total number of internal nodes is sum of all the internal nodes from level number 0 to level number h-1. The children of level h-1 are the leaf nodes. Thus, there will be minimum $2*[m/2]^{(h-1)}$ leaf nodes.

Consider a B+-Tree of order 8 with height 5. Minimum number of internal nodes are $1 + 2 + (2*4^1) + (2*4^2) + (2*4^3)=171$. The minimum number of leaf nodes are $(2 * 4^4) = 512$.

4. B+-Tree Operations.

Operations that are performed on a B+-Tree are insertion, deletion and search.

In insertion operation, a key value and its record address are inserted into B+-Tree, if it is not present in any one of the leaf nodes. B+-Tree retains its properties after the completion of this operation. During insertion, the key value and its record address will go to one of the leaf nodes and the other nodes in the B+-Tree may or may not change.

In deletion operation, key value will be deleted, if it is present in one of the leaf nodes. B+-Tree retains its properties after completion of this operation. When deletion is performed, key value and its record address are removed from the leaf node and the internal nodes are modified accordingly, if required.

Search operation is to find whether a key value is present in the B+-Tree or not.

5. Insertion of a key in B+-Tree.

Insertion of a key in B+-Tree requires searching through the indices (from the root node to the leaf node) until a specific leaf node is encountered where the key is to be inserted. Searching of this leaf node either produces the key value and address of the corresponding record, if the key is present or returns the position where the key value and record address are to be inserted into the leaf node.

This process is easy to understand because, the key value and the record address are just to be inserted into the leaf node and it is not required to make any other changes in the B+-Tree. However, insertion could require node splitting and changes to every level of B+-Tree, if a leaf node is full. Number of splits are generally depends on the set of keys that are to be inserted.

5.1 Algorithm for insertion :

Assume that key value (say KEY) is to be inserted in a B+-Tree of order m having height h.

Step 1 : Search for a leaf node where KEY is to be inserted.

1.1 Search down to a leaf node from the root node for KEY. While searching through the nodes of different levels from the root node, if KEY is the largest value in the node then replace the existing largest value with KEY.

1.2 If KEY is present, then return an error message about duplication.

Step 2 : Add KEY to the leaf node.

2.1 If leaf node is full then go to step 3.

2.2 Insert KEY in the leaf node at the appropriate place.

2.3 Change the key count of the leaf node.

2.4 Return.

Step 3 : Split the leaf node.

3.1 Get a new leaf node.

3.2 Leave keys from positions 1,2,...[m/2] in original node and place remaining keys in new node.

3.3 Insert KEY in appropriate node.

3.4 Update other fields in both the nodes. i.e links between leaf nodes and key counts.

Step 4 : Update the parent node.

4.1 If the parent node is not full then update the parent node and return.

4.2 While parent node is not a root node repeat steps 4.2.1 through 4.2.3.

4.2.1 Split the parent node (like leaf node).

4.2.2 Change parent node to its parent node in the next higher level.

4.2.3 If parent node is not full then update the parent node and return.

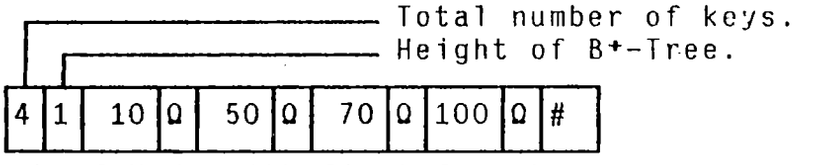
4.3 Split the parent node and make a new root node. Update the root node with appropriate values.

4.4 Return.

5.2 Illustration of above algorithm with an example :

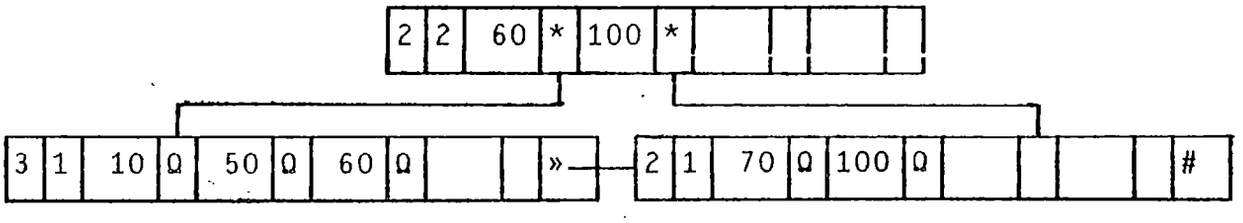
Consider an example of inserting the set of keys 100, 10, 50, 70, 60, 61, 101, 11, 12, 52, 53, 54, 55, 80, [illegible] in a B+-Tree of order 4. In this example for our understanding, we use * as pointer to a node in the next level, Ω as record pointer, » as pointer to leaf node and # as null pointer.

Assume that initially B+-Tree is empty. Since B+-Tree is empty, first four keys will be inserted in the leaf node as shown below.



a. Leaf Node with first four keys.

Next key to be inserted is 60. The leaf node, where 60 is to be inserted is full, so split this leaf node. After splitting, B+-Tree takes the following form.

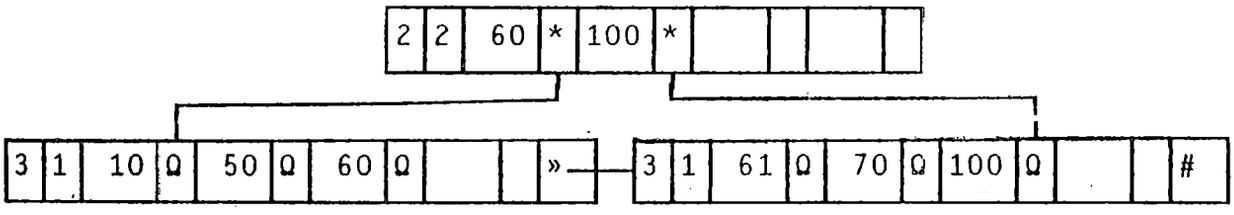


b. After insertion of 60

At this stage, B+-Tree consists of two levels. Root node is at level one containing two keys 60 and 100. Leaf nodes are at level two, containing 3 and 2 keys respectively. First leaf node is pointing to the second leaf node.

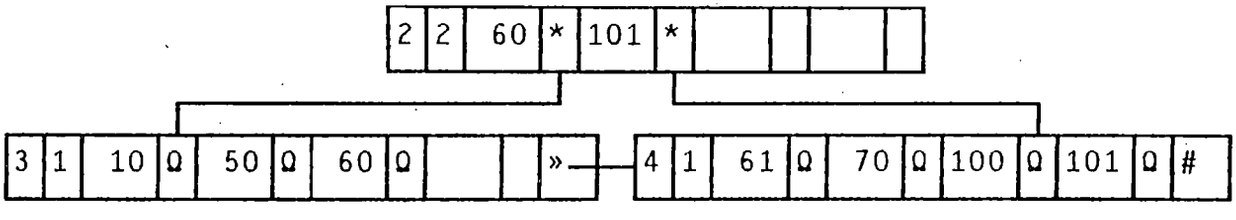
Next key that is to be inserted is 61. It is required to search for a leaf node where this key is to be inserted. We search the root node to get a pointer which points to the node in the next level. First key of root node is 60 and it is less than 61, so it is skipped and compared with the next

key 100. Since 61 is less than 100, 61 will be searched in the node corresponding to the pointer associated with the key 100. 61 is not present in this leaf node and the leaf node is not full, so it will be inserted in this node.



c. B+-Tree after insertion of 61

Inserting 101 needs searching down to leaf node. When the root node is searched for 101, 101 is greater than the largest key (i.e. 100). This requires replacing 100 with 101 and continue searching the node in the next level using the pointer corresponding to 101 in the root node. Remaining steps are carried out as in the previous case.



d. B+-Tree after inserting 101

Figure e shows the status of B+-Tree after inserting keys 11 and 12. Figure f shows the status of B+-Tree after the insertion of keys 52, 53, 54 and 55. Consider the case of inserting 80. 80 is to be inserted in the last leaf node in the second level and it is full. We split the leaf node and 80 is inserted appropriately. Here parent node is to be updated. Since parent node is root node and it is full, we split this node and make a new root node.

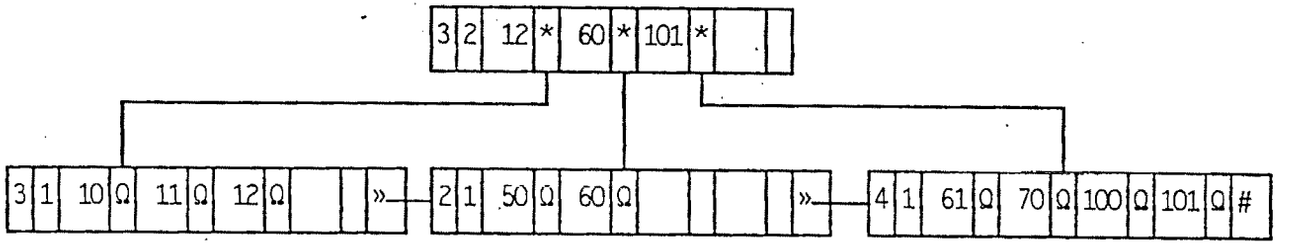


Fig.e. B+-Tree after inserting 11 and 12.

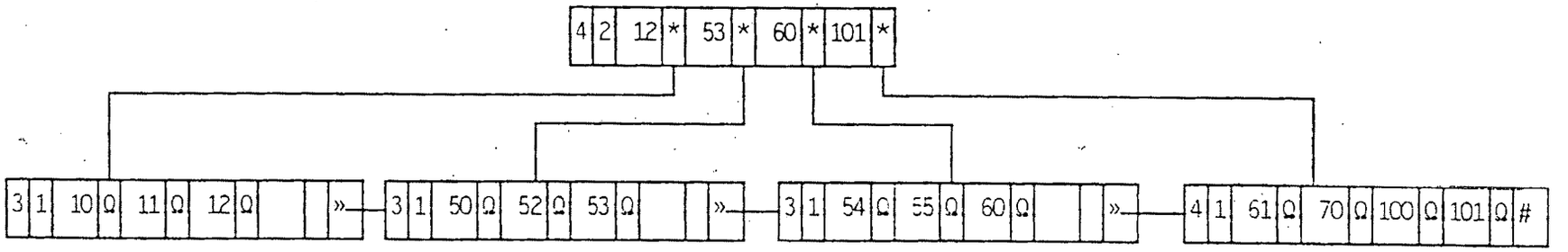


Fig.f. B+-Tree after inserting 52, 53, 54 and 55.

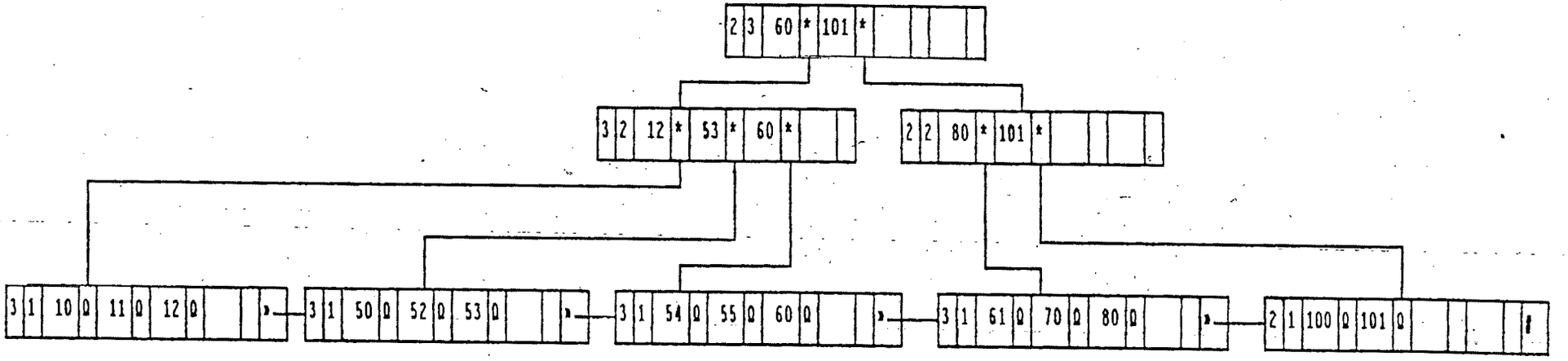


Fig.g. B+-Tree after inserting 80.

Figure g is the new B+-Tree after inserting 80 which is containing 3 levels. The level number in the root node is 3 which is the highest level in the B+-Tree.

6. Deletion of a key from B+-Tree.

Deleting a key from B+-Tree requires searching through all levels of nodes until a specific leaf node is found. The key value and record address are deleted from that leaf node, if the given key value is found. If the key value that is to be deleted is the largest key value in that leaf node, then all internal nodes are to be updated accordingly and additional process of maintaining the B+-Tree structure, is to be carried out. In insertion, either key value is added to the leaf node or leaf node is split. Where as in deletion, key value is to be deleted and if required, leaf nodes are to be joined and the parent nodes are to be updated.

6.1 Algorithm for Deletion :

Assume that a key value (say KEY) is to be deleted from a B+-Tree of order m and height h.

Step 1 : Search for KEY.

1.1 Search down to a leaf node for KEY.

1.2 If KEY is not found then return.

Step 2 : Remove KEY from leaf node.

2.1 Remove KEY and address from the leaf node in which it was found.Decrement the key count by 1.

2.2 If key count is greater than or equal to [m/2] and if the key just deleted is not the largest key in that leaf node, then return.

2.3 If key count is greater than or equal to [m/2] and if key just deleted is the largest key in that leaf node, then update the parent nodes accordingly with the next largest key.

2.4 If the key count is less than [m/2], then go to step 3.

Step 3 : Adjust leaves.

3.1 If there is no node present right to this leaf node goto step 3.4.

3.2 If the key count of the node right to this leaf node (say RNODE) is greater than [m/2] then transfer first key of RNODE to the current leaf node; otherwise goto step 3.3.

3.2.1 Change key count of both nodes.

3.2.2 Replace the key in the parent node with the largest key.

3.2.3 Return.

3.3 Merge RNODE with the leaf node.

3.3.1 Change key count of the leaf node accordingly.

3.3.2 Replace the key in its parent node with the largest key value.

3.3.3 Delete the key value corresponding to RNODE in its parent node (i.e. perform step 4.) and return.

3.4 If the key count of the node left to this leaf node (say LNODE) is greater than [m/2] then transfer last key of LNODE to the current leaf node; otherwise goto step 3.5.

3.4.1 Change key count of both nodes.

3.4.2 Replace the key in the parent node with the largest key.

3.4.3 Return.

3.5 Merge LNODE with the leaf node.

3.5.1 Change key count of the leaf node accordingly.

3.5.2 Replace the key in its parent node with the largest key value.

3.5.3 Delete the key value corresponding to LNODE (i.e. perform step 4.) and return.

Step 4 : Adjust Internal Nodes.

If this internal node is the only one node in that level then make this node as root node and return.

4.1 If there is no node present right to this internal node goto step 4.4.

4.2 If the key count of the node right to this internal node (say RNODE) is greater than [m/2] then transfer first key of RNODE to the current internal node otherwise goto step 4.3.

4.2.1 Change key count of both nodes.

4.2.2 Replace the key of parent node with the largest key.

4.2.3 Return.

4.3 Merge RNODE with current internal node.

4.3.1 Change key count of the internal node accordingly.

4.3.2 Replace the key in its parent node with the largest key value.

4.3.3 Delete the key value corresponding tc RNODE in its parent node by calling Adjust Internal Nodes.

4.4 If the key count of the node left to this internal node is greater than [m/2] then transfer last key of LNODE to the current internal node otherwise goto step 4.5.

4.4.1 Change key count of both nodes.

4.4.2 Replace the key in the parent node with the largest key.

4.4.3 Return.

4.5 Merge LNODE with the current internal node.

4.5.1 Change key count of the current internal node accordingly.

4.5.2 Replace the key in its parent node with the largest key value.

4.5.3 Delete the key value corresponding to LNODE in its parent node by calling Adjust Internal Nodes.

6.2 Illustration of above algorithm with an example :

Consider the B+-Tree of order 4 and height 3 as shown in Fig. h. We consider the case of deleting the key 54. Key 54 is found in the third leaf node and it will be deleted. Deletion of this key does not require any other changes, since the key count is 3 which is greater than 2. Fig. i is the B+-Tree after deleting the key 54.

We try to delete 53 from the B+-Tree shown in Fig. i. 53 is present in the second leaf node. This key is to be deleted. Since it is the largest key in the node, we replace this value in the parent node with the next largest key (i.e.52). No other changes are required to the corresponding higher level nodes, because 52 is not the largest key in the parent node. Fig. j is the B+-Tree after the deletion of 53.

Consider the key to be deleted is 50. 50 is present in the second leaf node. If 50 is deleted, then the key count of this leaf node becomes 1 which is less than 2. There is a node present right to this leaf node. We transfer first key of that right node to the current node and we change the key counts of both the nodes. Update the parent node accordingly. Fig. k is the B+-Tree after deleting 50. If 52 is deleted, then key count of leaf node which is containing 52 becomes one. Node right to this leaf node will be merged with this node and accordingly the parent node will be updated. Fig. l is the B+-Tree after the deletion of 52.

Let us consider the case of deleting 100. If 100 is deleted from the leaf node, the key count becomes 1. There is no node present right to this leaf node. There is a node present on the left side with key count as 3. So we transfer the largest key from left node to the current leaf node and accordingly parent nodes are updated. Fig. m is the B+-Tree after the deletion of 100. Let us consider the case of deleting 80. If 80 is deleted, keycount becomes 1 which is less than 2. There is no node present on the right side.

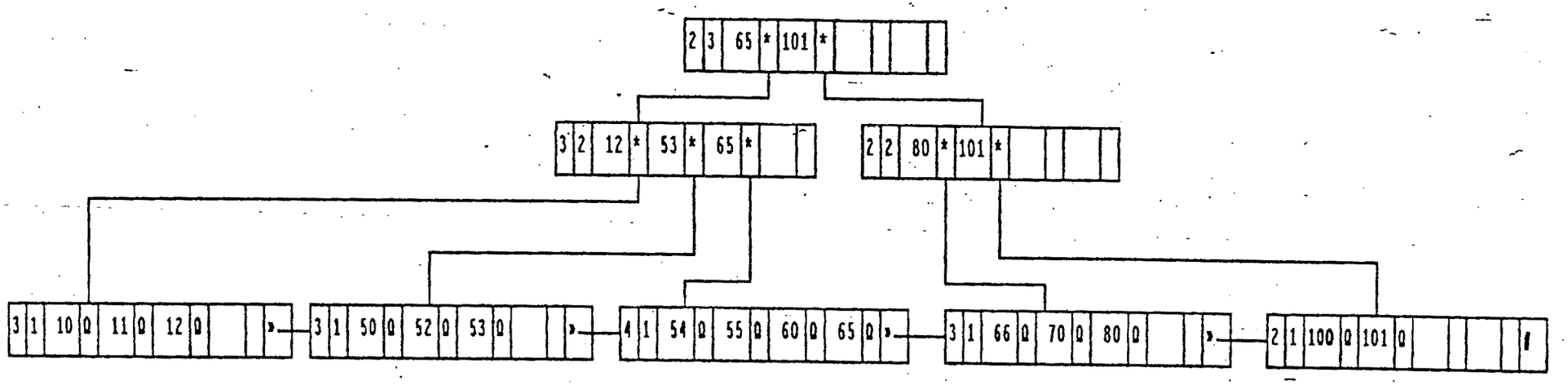


Fig.h. B+-Tree

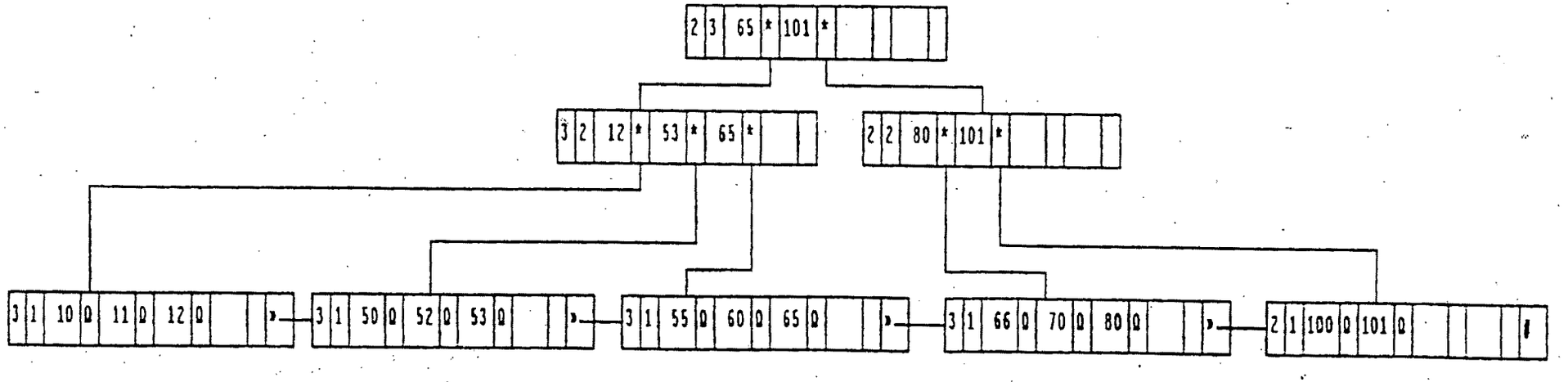


Fig.i. B+-Tree after deleting 54.

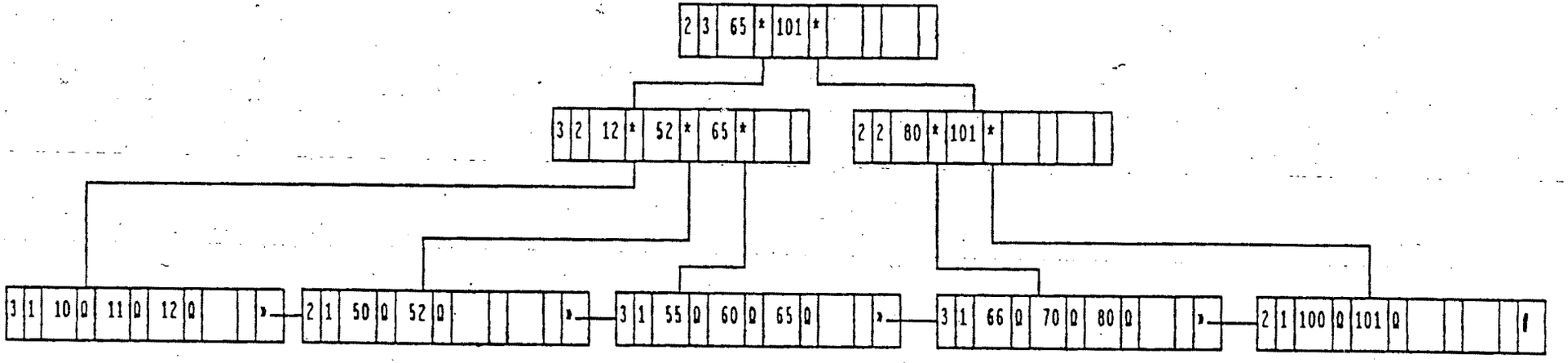


Fig.j. B+-Tree after deleting 53.

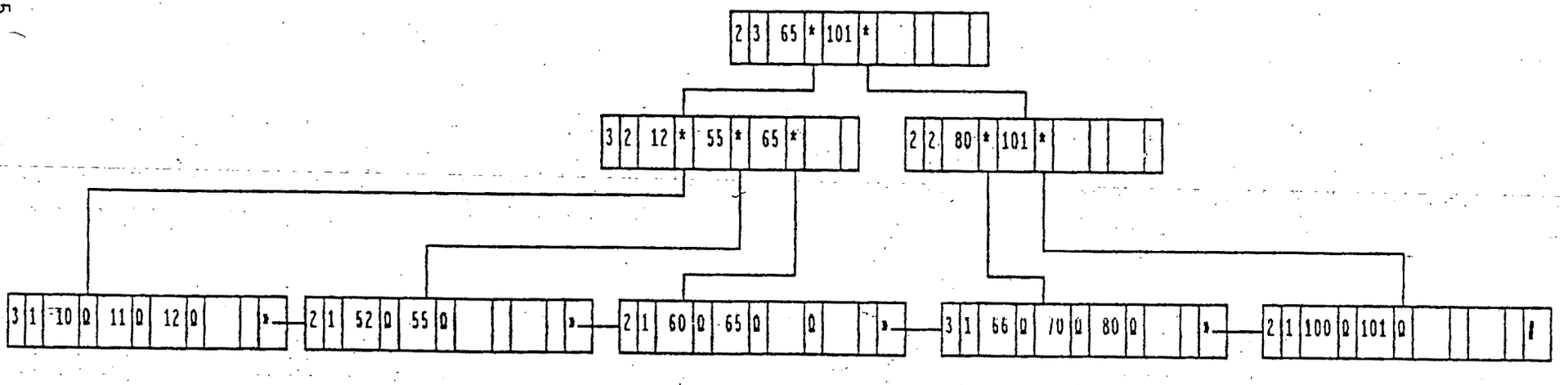


Fig.k. B+-Tree after deleting 50.

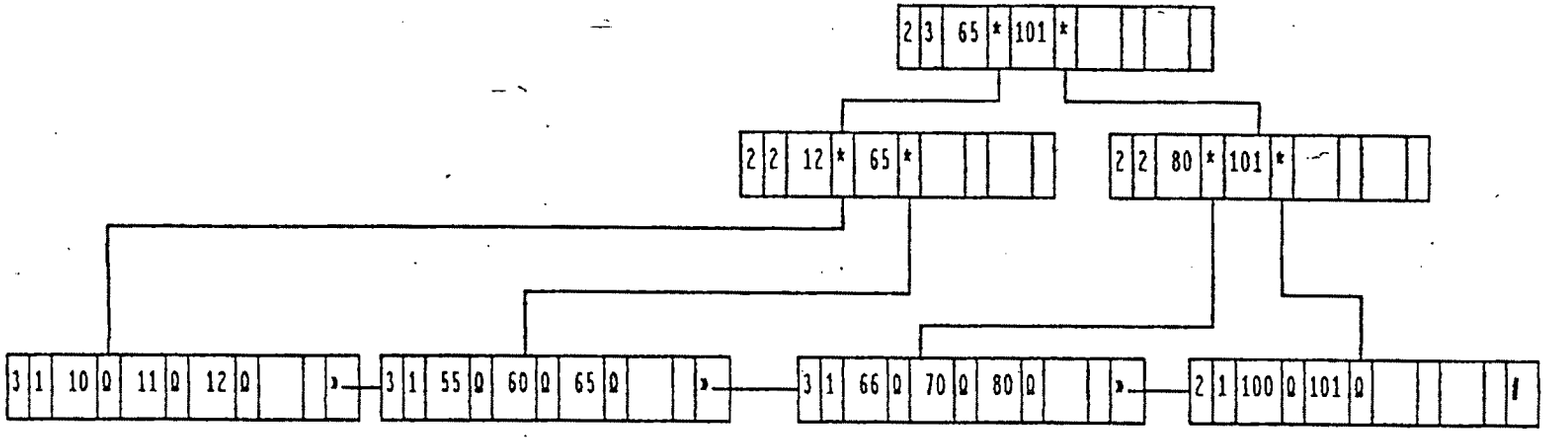


Fig.l. B+-Tree after deleting 52.

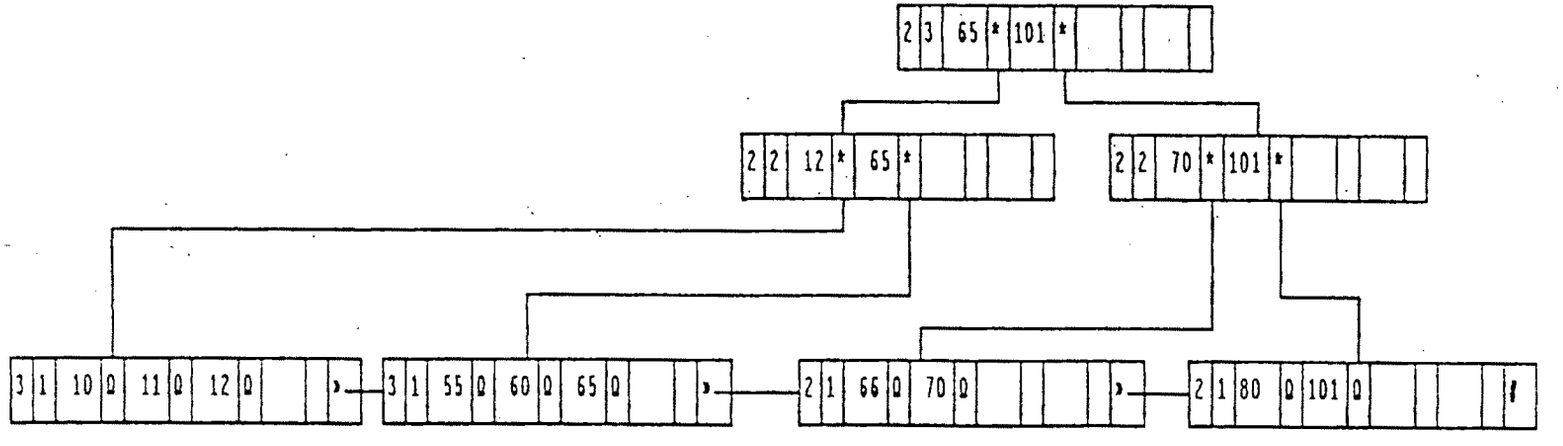


Fig.m. B+-Tree after deleting 100.

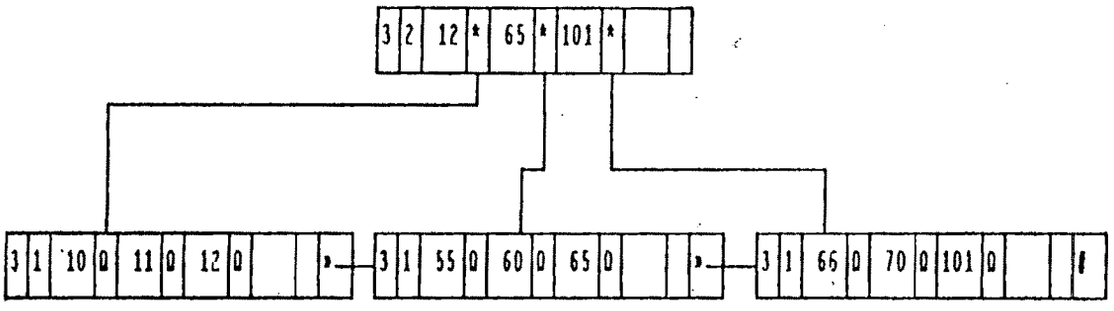


Fig. n. B+-Tree after deleting 80.

There is a node present on the left side. We merge this node with the left node and there by the parent node will be updated. Key count of parent node is 1 and there is no node right to this node. This node will be merged with the left node and it becomes the root node since it is the only node present in that level.

7. Search.

Assume that B+-Tree is of order m. Start with root node and search for the key value in this node. This search leads to the node in the next level. Searching this node leads to a node in the next level. Continue this process until a leaf node is encountered. Search the leaf node for the key value.

8. Software Implementation.

From the algorithms and examples discussed above, it has become clear that the structure and the operations of a B+-Tree are not that simple for a student to understand the complexities. These concepts become clear only when more number of examples are solved. Solving examples is very tedious task, because each time a new B+-Tree is to be drawn. The intrinsic understanding of a B+-Tree concepts demand a lot of imagination. To overcome these difficulties, we have developed a CAI package to teach B+-Tree structure and its operations. We have divided this package into four different components to make it easy for the user to select one of the topics. These topics include help on B+-Tree, B+-Tree insertion, B+-Tree deletion and B+-Tree search. Each of the operations of B+-Tree is simulated and interactions are provided to make the learning process more interesting with

on-line help. The imaginary things on each operation are picturised with animation to make the subject matter more clearer to all.

8.1 Help on B+-Tree :

We have provided help facility which includes B+-Tree definition, order of B+-Tree, levels of B+-Tree, structure of non terminal nodes and terminal nodes, B+-Tree properties and B+-Tree operations. This help can be invoked by pressing F1 key and user can browse through the help using PgUp/PgDn keys.

8.2 B+-Tree Insertion :

In this part, we have simulated an environment to show each case of insertion algorithm. We start with an empty B+-Tree of order 4 and we insert a set of keys. Whenever a key is inserted, the steps required are shown separately in a window. User gets sufficient time to examine each step taken by the system to insert a particular key. The key set is chosen in such a way that it covers all the cases of insertion namely inserting a key in leaf node without making changes to any part of the B+-Tree, changing the largest key in internal nodes with the new key, splitting a leaf node, updation of internal nodes and finally, the increase in the number of levels or creation of a new root node. User is shown B+-Tree insertion with predefined key set and later users are allowed to use their own key set for insertion. To show the whole process, screen is divided into two main windows. In one window, B+-Tree is shown and in other window the steps that are being taken to insert a key are displayed.

Figures 5.1 to 5.7 are the sample screen layouts which are showing the B+-Tree insertion process.

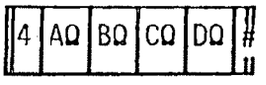
8.3 B+-Tree Deletion :

In this option, each case of deletion operation are shown as in B+-Tree insertion. Keys are chosen in such a way that they cover all cases namely deleting a key from leaf node without changing any other node in B+-Tree, deleting the largest key from the leaf node, deleting a key from the leaf node which was half full, transferring the key from right node to the current leaf node, merging the right node with the current leaf node, transferring the key from the left node to the current leaf node, merging the left node with the current leaf node, adjusting the keys between two internal nodes as in leaf nodes, updation of parent nodes and finally the decrease in the levels of B+-Tree. We have provided the user with sufficient time to understand the steps that are being carried out to delete a key from B+-Tree. User is shown the whole process with predefined key set and later, users are allowed to use their own keys for deletion. As in insertion, screen is divided into two main windows to show the B+-Tree and the steps.

8.4 B+-Tree Search :

Though it is easy to understand search mechanism, our aim in providing this option is to make the user to understand the number of probes that are required to reach the leaf node.

B Plus Tree Insertion




```
Key is not present, so  it is to be inserted in the Leaf Node.
Current Leaf Node is full, so split this leaf node in to two Leaf Nodes.

Order : 4    No. of Levels : 1    Current Key  : E
Press any key to continue ....
```

Fig. 5.1. Before splitting the leaf node.

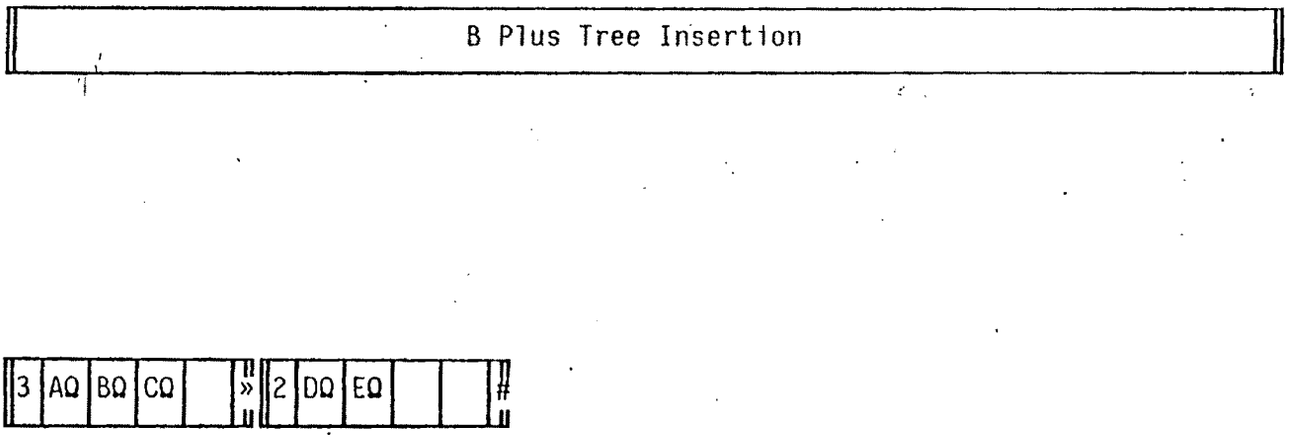


Fig. 5.2. After splitting the leaf node and before forming root node.

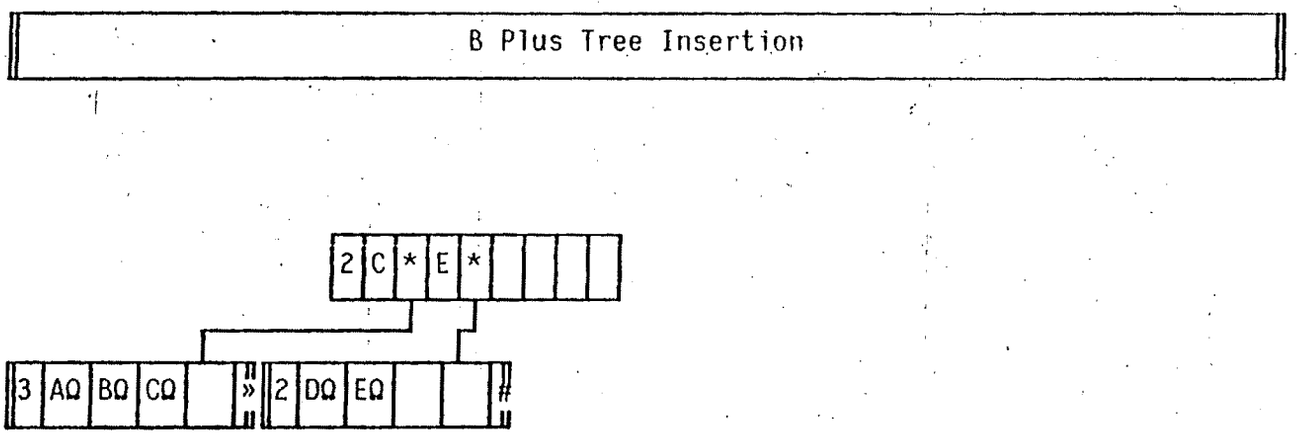


Fig. 5.3. B⁺-Tree with two levels. First step in inserting the key 'F'.

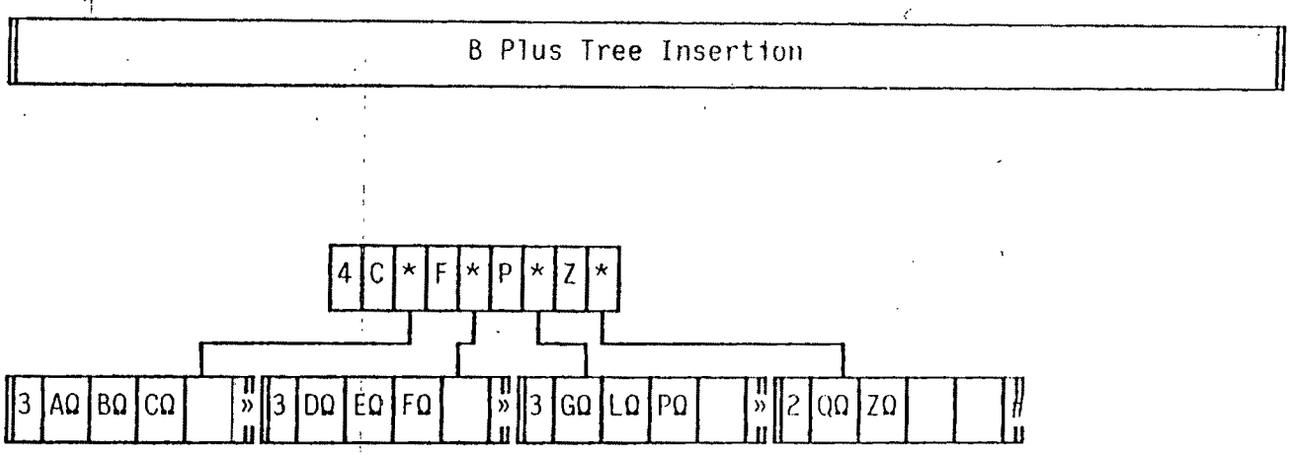
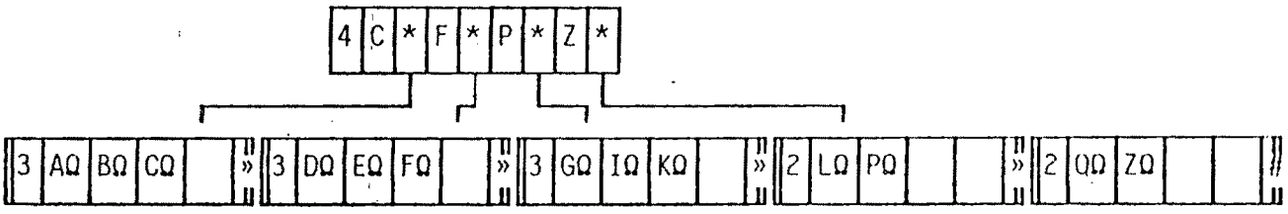


Fig. 5.4. B⁺-Tree with 4 leaf nodes. First step in inserting the key 'K'.

B Plus Tree Insertion



Update the corresponding root node.

Order : 4 No. of Levels : 2 Current Key : I

Press any key to continue ....

Fig. 5.5. After splitting the leaf node when 'I' is inserted.

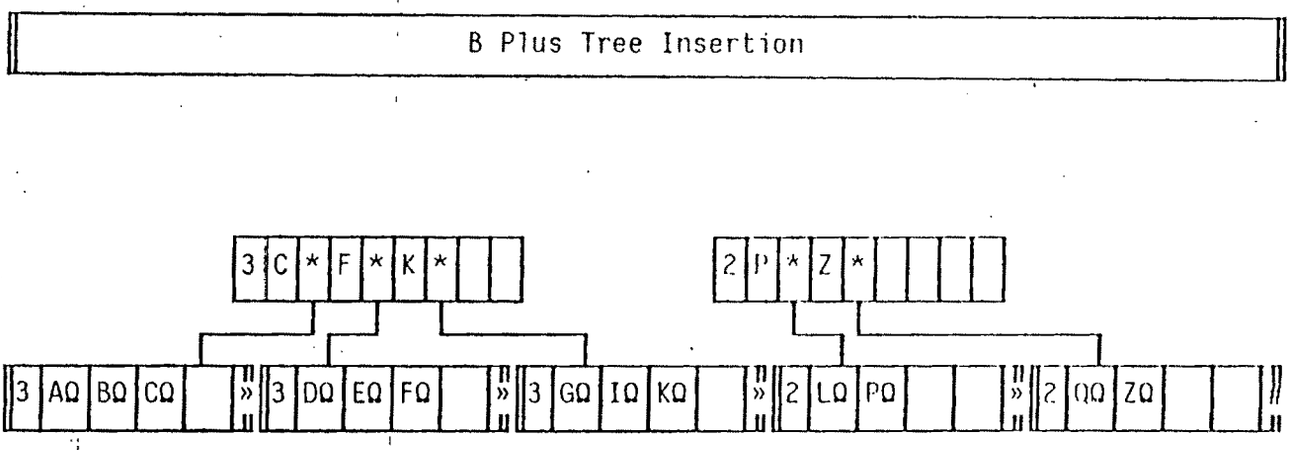


Fig. 5.6. After splitting the root node.

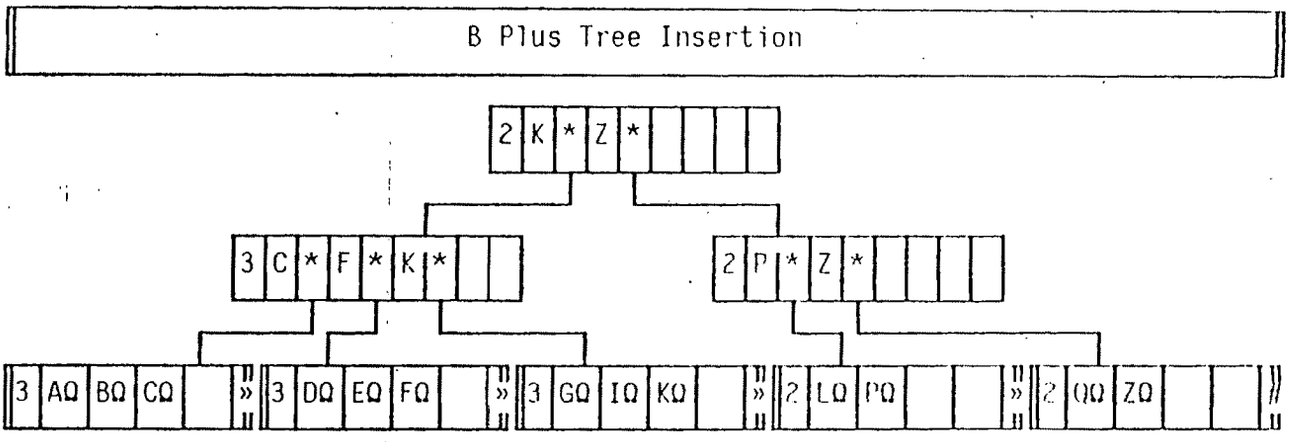


Fig. 5.7. B⁺–Tree with THREE levels and FIVE leaf nodes.

9. Advantages.

9.1 User gets good understanding about the concept of multi way search.

9.2 More number of examples with different combinations would enhance the understanding.

9.3 Difference between the non terminal nodes and terminal nodes become clearer.

9.4 The relation between the order of a B+-Tree and levels in a B+-Tree is known.

9.5 Different cases that are to be considered in both insertion and deletion are known.

9.6 The necessity of keeping root node of B+-Tree in primary memory is made clearer.

9.7 Thorough understanding of whole process may lead to new ideas in terms of reducing storage space.

10. Limitations of CAI package.

The following limitations are due to the space that is available on the screen.

10.1 Order of a B+-Tree is taken as 4. User is not allowed to change the order of B+-Tree and it is not decided using the length of the key which is usually done.

10.2 Maximum three levels are shown with maximum 5 leaf nodes are shown in the last level.

10.3 In internal nodes and leaf nodes level number is not shown.

10.4 Only upper case alphabets are considered as keys.

10.5 The total number of keys that are to be inserted is not decided and it depends on the sequence of keys that are to be inserted and the number of leaf nodes.

10.6 Key pointer pairs in a leaf node are taken together to accommodate more number of leaf nodes.