B+ Trees

What is a B+ tree?

A B+ tree consists of one or more blocks of data, called nodes, that are linked together by pointers. The tree has a root node, interior nodes, and leaf nodes. The root node may have one or more children, and the interior and leaf nodes may have multiple keys and pointers. The keys and pointers are organized in an ordered list, and the tree is balanced by ensuring that all the leaves are at the same level and each node is at least half full. The B+ tree also has a next sibling pointer in the leaf nodes to allow for fast iteration through contiguous blocks of values. This property is useful for efficient range queries, where the block containing the first value and the subsequent sibling blocks can be quickly scanned until the last value is found. To search for or insert an element into the tree, the root node is loaded, the adjacent keys that the searched-for value is between are found, and the corresponding pointer to the next node in the tree is followed. This process is repeated until the desired value is found, or it is determined that the value is not present.

How are B tree and B+ tree similar?

Both B trees and B+ trees are self-balancing tree data structures that are used to store and retrieve data efficiently. They are both characterized by the fact that all the leaves are at the same level and each node is at least half full. Both B trees and B+ trees are useful when dealing with large data sets that cannot fit in main memory and require disk seek operations, as they can reduce the number of disk seeks needed.

How are B tree and B+ tree different?

There are several differences between B trees and B+ trees. One of the main differences is that B+ trees only store data pointers in the leaf nodes, while B trees can store data pointers in both the interior and leaf nodes. This means that B+ trees can fit more keys in the interior nodes, making them more efficient for searching and inserting data. B+ trees also have linked leaf nodes, which allows for efficient linear scans and range queries. B trees do not have this property, as they require a traversal of every level in the tree to perform a linear scan.

Anatomy of B+ Trees

At the core of a B-tree are nodes, which come in two types: internal nodes and leaf nodes. Internal nodes contain keys and pointers to their child nodes. Leaf nodes, on the other hand, store the actual data. Here’s a closer look at the components:

1. **Root Node**: This is the topmost node in a B-tree. It serves as the starting point for any search, insert, or delete operation. The root can have as few as one key if it is the only node in the tree.
2. **Internal Nodes**: These nodes contain keys and pointers to other nodes. The keys act as separators that divide the tree into subtrees, each subtree containing values that fall within a certain range. For instance, in a node with keys [20, 50], all values in the left subtree are less than 20, values in the middle subtree are between 20 and 50, and values in the right subtree are greater than 50.
3. **Leaf Nodes**: These are the bottommost nodes of the tree that contain the actual data or values. Leaf nodes do not have any children, and all leaf nodes are at the same level, ensuring the tree remains balanced.
4. **Keys**: Each key in a node represents a value used to divide the tree into smaller subtrees. The number of keys in a node determines the node’s degree, and each node can have a varying number of keys within the allowed minimum and maximum limits.
5. **Pointers**: These are references to the child nodes. Each internal node has one more pointer than the number of keys, directing the search process towards the appropriate subtree.

The balancing property of B-trees ensures that the tree remains shallow, even with large datasets. This property minimizes the number of disk reads required to access data, as fewer nodes need to be traversed. The self-balancing nature of B-trees, achieved through operations like node splitting and merging, maintains optimal performance during data insertions and deletions.

Operations in B+ Trees

B+ tree data insertion

To insert data into a B+ tree, the following steps are generally followed:

1. Start at the root node and find the adjacent keys that the value to be inserted is between.
2. Follow the corresponding pointer to the next node in the tree.
3. If the node is a leaf node, insert the value into the node.
4. If the node is not a leaf node, repeat the process until a leaf node is reached.
5. If the leaf node is full, split it into two nodes and insert the value into the appropriate node.
6. If the root node splits, create a new root node with two children.

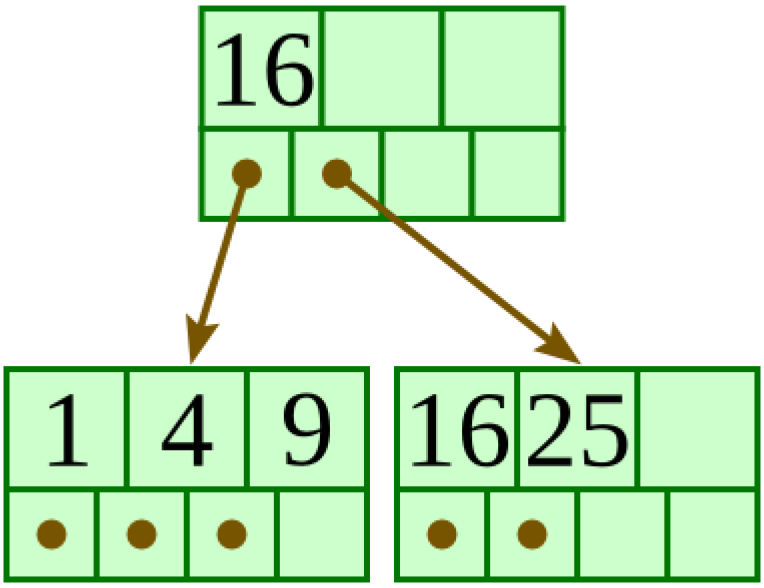
Example of some data insertion:

Example source: CSci 340: B+-trees (cburch.com)

Let's consider the following B+ tree, where we have inserted **1, 4, 9, 16, and 25** already. We will insert **20, 13, 15, 10, 11, and 12** respectively.

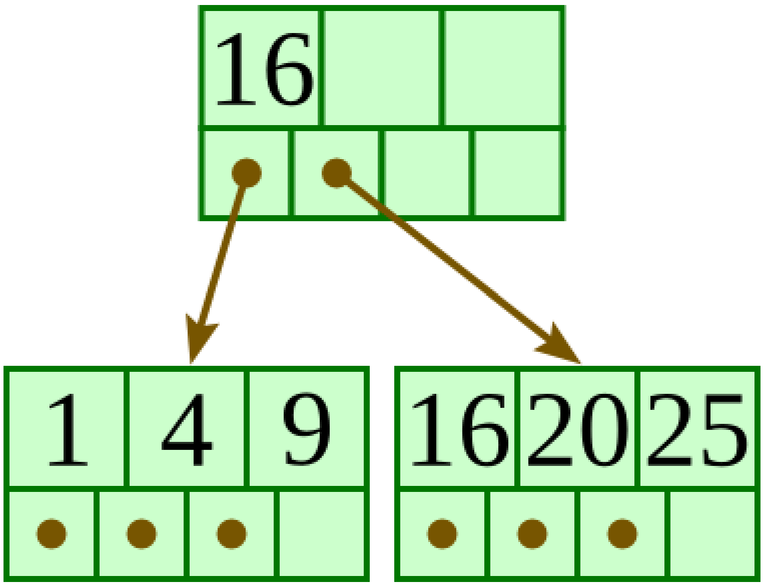
Maximum capacity of a node or maximum number of keys that a node can contain is 3, hence the order is 4. So, minimum capacity is 1. That means, a node has to have at least 1 key.

> Initial tree:



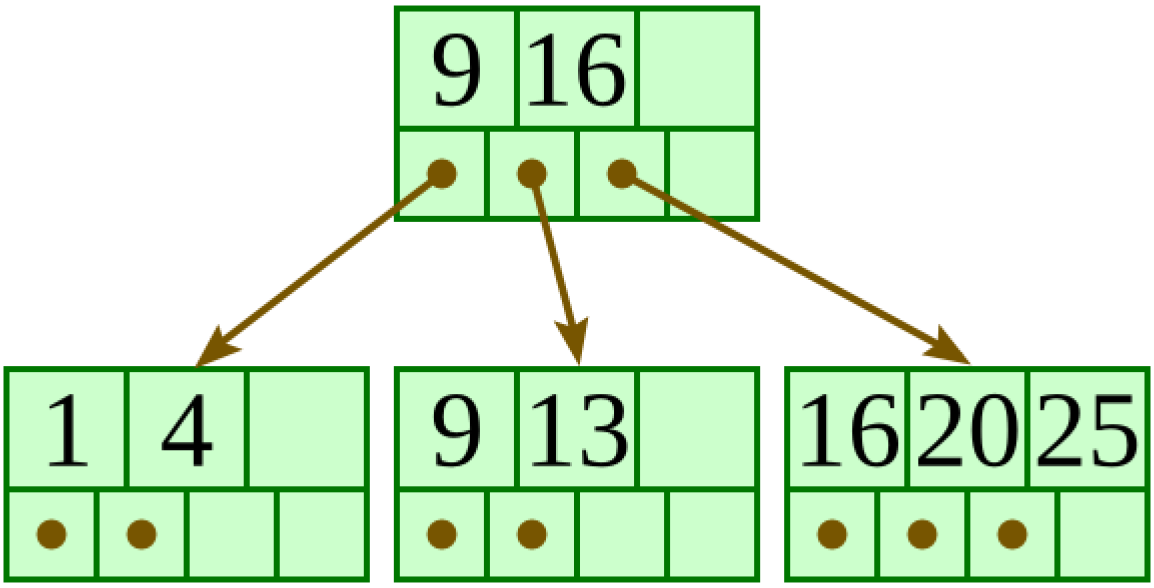
Initial tree

> Insert 20:



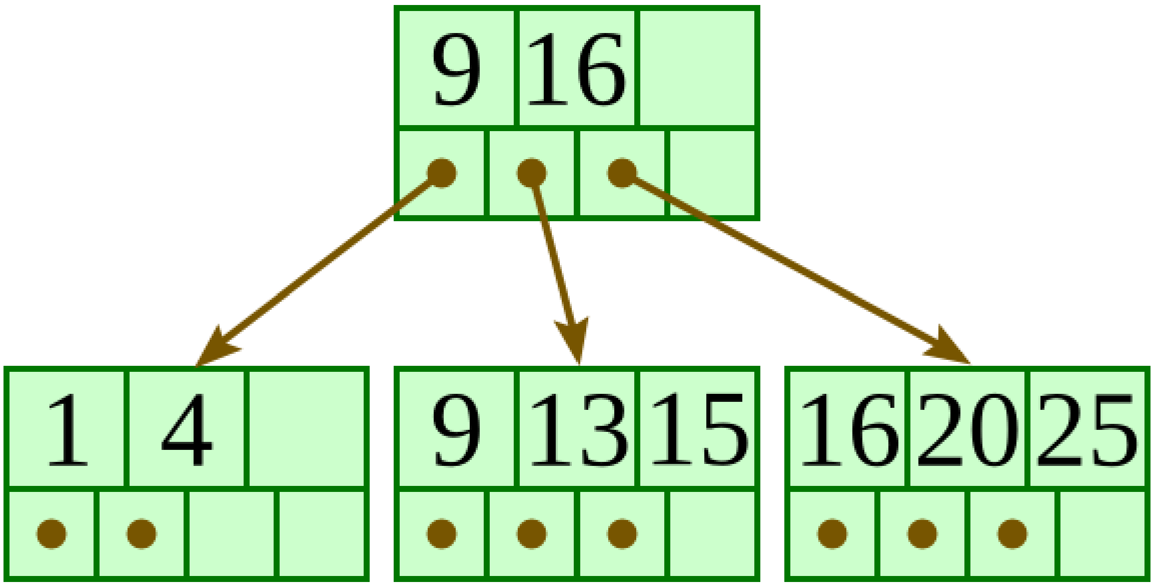
After inserting 20

> Insert 13:



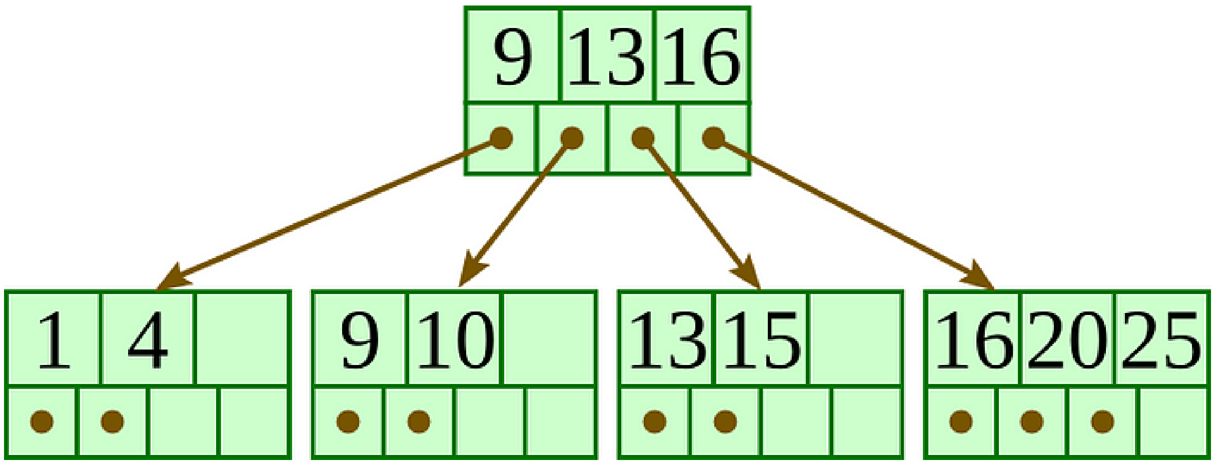
After inserting 13

> Insert 15:



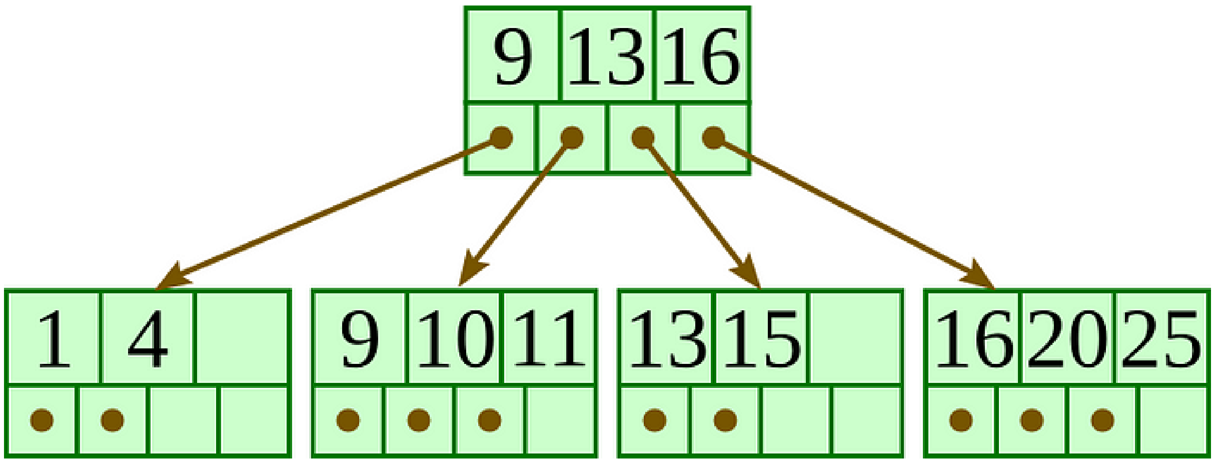
After inserting 15

> Insert 10:



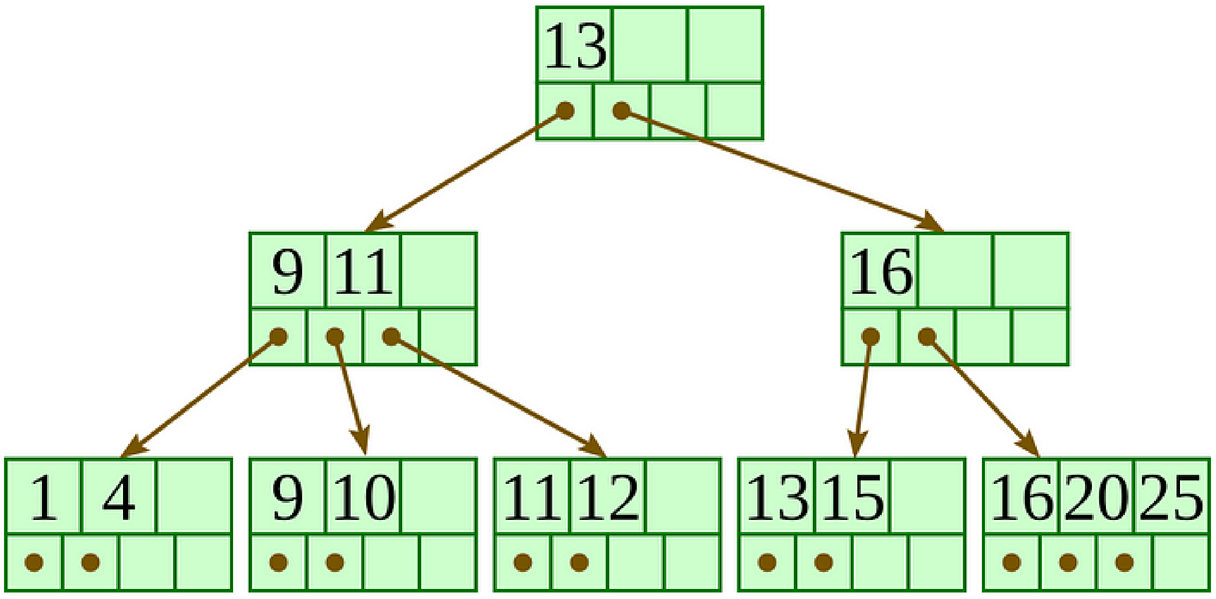
After inserting 10

> Insert 11:



After inserting 11

> Insert 12:



After inserting 12

B+ tree data deletion

To delete data from a B+ tree, the following steps are generally followed:

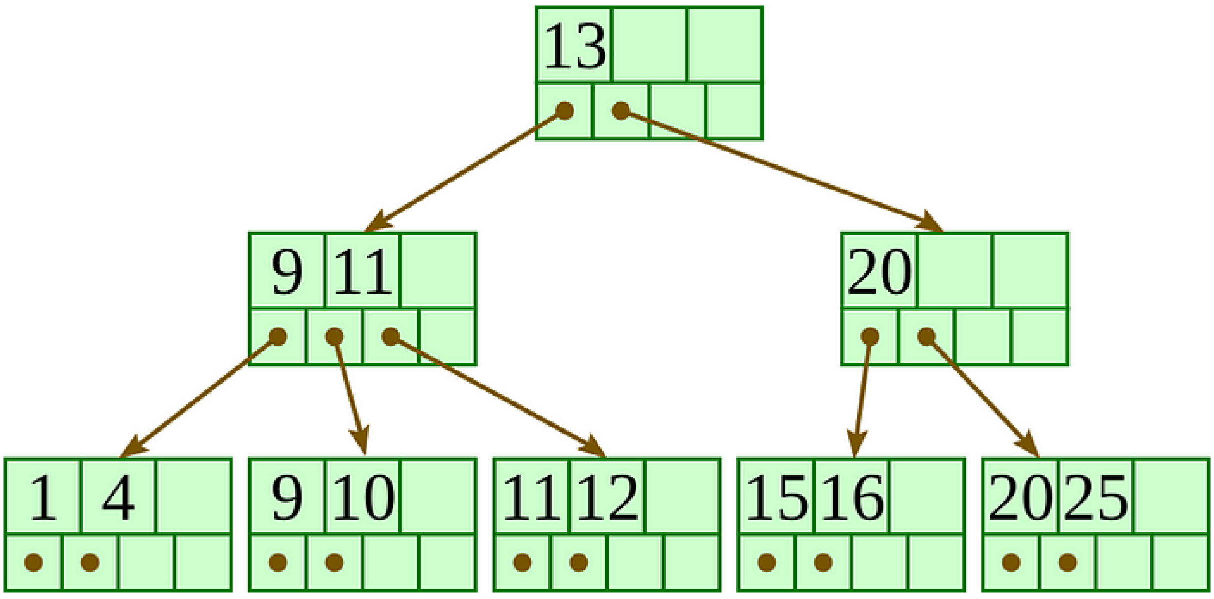
1. Start at the root node and find the adjacent keys that the value to be deleted is between.
2. Follow the corresponding pointer to the next node in the tree.
3. If the node is a leaf node, delete the value from the node.
4. If the node is not a leaf node, repeat the process until a leaf node is reached.
5. If the leaf node has fewer than the minimum number of keys required, merge it with one of its siblings or borrow a key from a sibling.
6. If the root node has only one child, delete the root node and make the child the new root.

Example of some data deletion:

Example source: CSci 340: B+-trees (cburch.com)

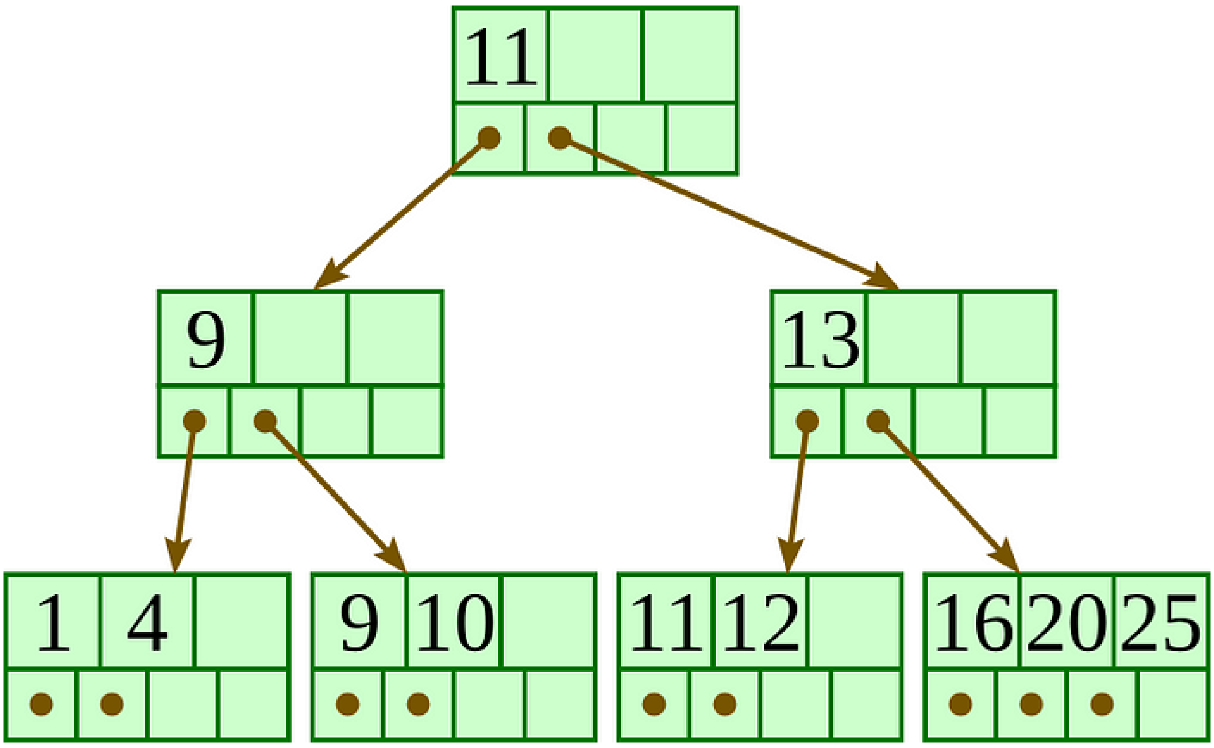
Let's delete 13, 15, and 11 from above built B+ tree.

> Delete 13:



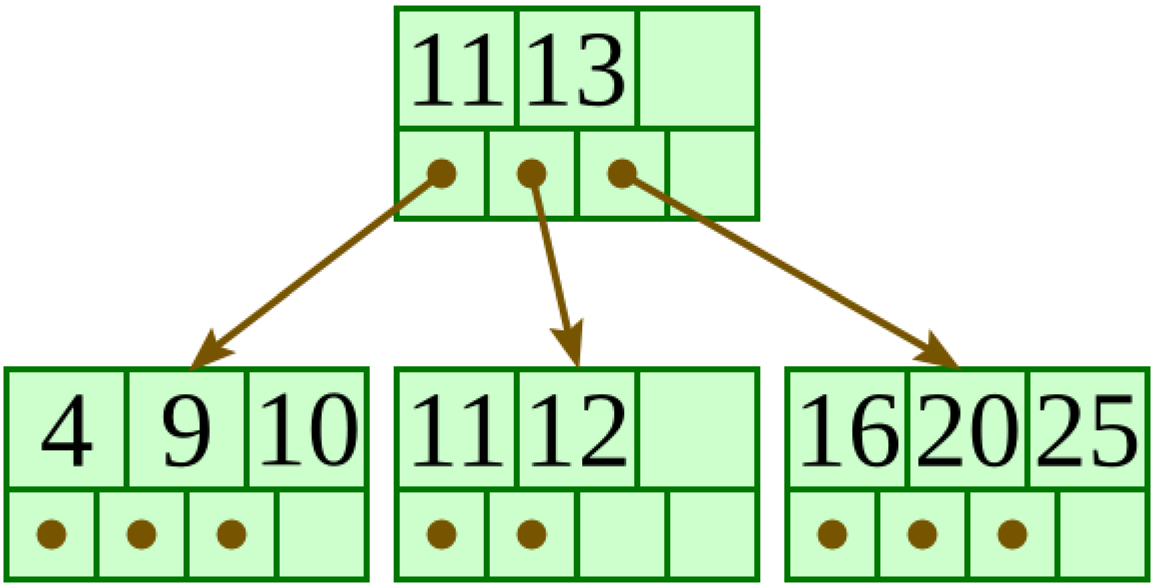
After deleting 13

> Delete 15:



After deleting 15

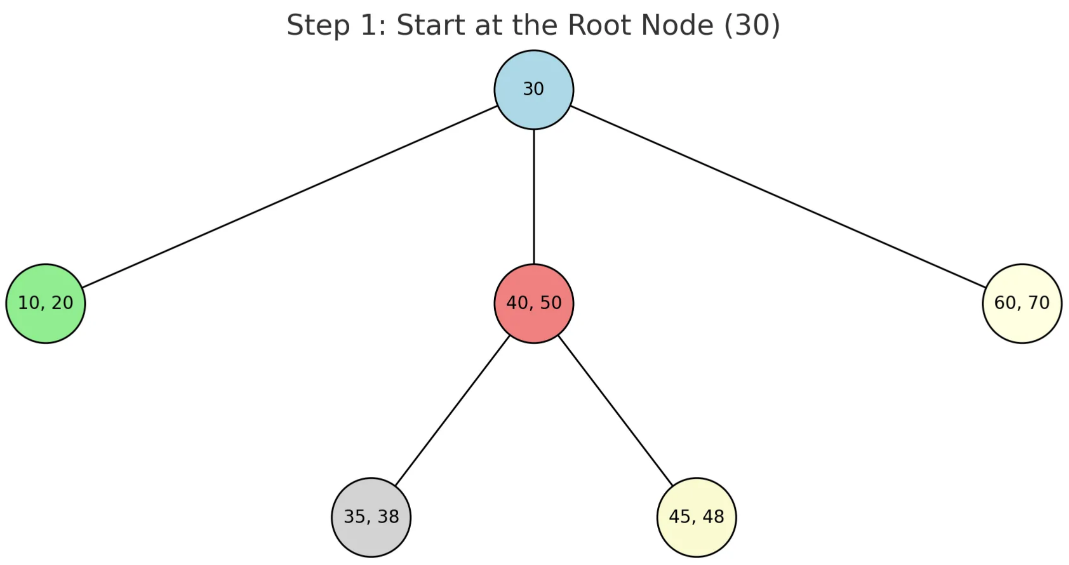
> Delete 1:

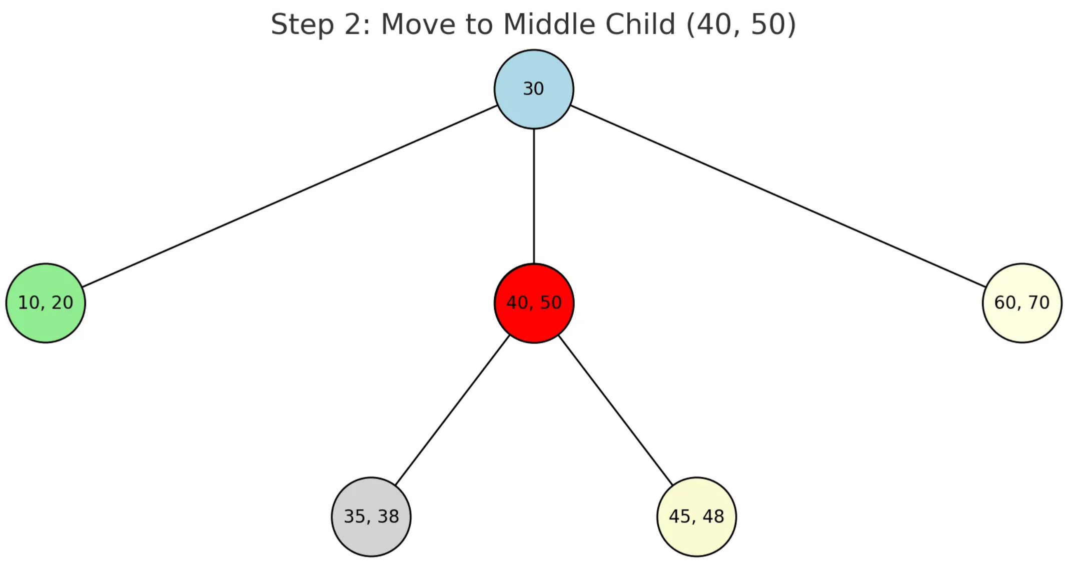


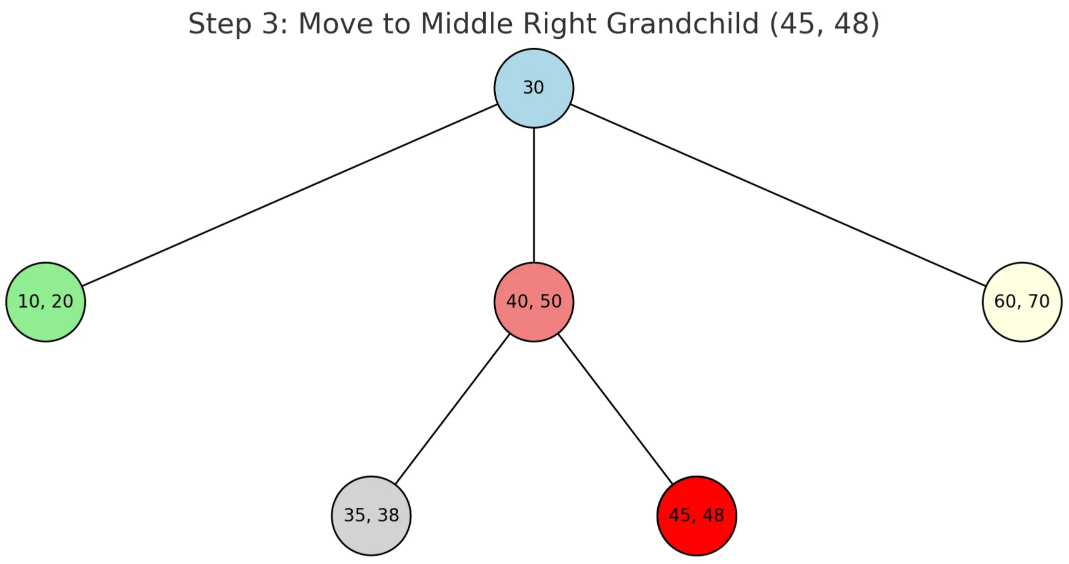
After deleting 1

Searching in B+ Trees

The primary strength of B-trees lies in their ability to optimize search operations. This efficiency is achieved through a structured hierarchy and balanced distribution of keys, which significantly reduces the number of comparisons needed to find a particular value.


Step 1: Start at the Root Node (30)
30
10, 20
40, 50
60, 70
35, 38
45, 48


Step 2: Move to Middle Child (40, 50)
30
10, 20
40, 50
60, 70
35, 38
45, 48


Step 3: Move to Middle Right Grandchild (45, 48)
30
10, 20
40, 50
60, 70
35, 38
45, 48

When searching for a key in a B-tree, the process begins at the root node. At each node, a binary search is performed on the keys within the node to determine the path to follow. Here's a step-by-step look at how the search process works:

1. **Start at the Root Node**: The search begins at the root node. For example, if we are searching for the key `45` in our B-tree, we start by comparing `45` with the key(s) in the root node.
2. **Binary Search within the Node**: If the node contains multiple keys, a binary search is used to quickly locate the range in which the key falls. For instance, in a node with keys [30], `45` is greater than `30`, so we move to the next child node, which is the middle child in this case.
3. **Traverse to the Child Node**: Based on the result of the comparison, the search moves to the appropriate child node. This process continues recursively. For our key `45`, after the root node, we move to the middle child containing [40, 50].
4. **Repeat the Process**: The binary search and traversal steps are repeated at each node until the key is found or the search reaches a leaf node. In our example, `45` lies between `40` and `50`, so we move to the middle child of this node, eventually finding the key `45`.
5. **Balanced Tree Structure**: The balanced nature of B-trees ensures that all leaf nodes are at the same level, making the search path predictable and efficient. This balance minimizes the number of nodes that need to be accessed, reducing disk I/O operations.
6. **Efficiency through Height Reduction**: B-trees are designed to remain shallow even with a large number of keys. This shallow height, combined with the efficient binary search within nodes, ensures that the search operation is logarithmic in nature, typically O(log n), where `n` is the number of keys.

This structured approach allows B-trees to handle large datasets effectively, making them a popular choice for database indexing and file systems. The ability to quickly narrow down the search path and minimize disk accesses is crucial for performance, especially in systems where data retrieval speed is critical.

Advantages of B-Trees:

Balanced Structure: B-trees maintain a balanced structure, ensuring that all leaf nodes are at the same depth. This balance minimizes the height of the tree, resulting in efficient search, insert, and delete operations. By keeping the tree shallow, B-trees reduce the number of disk accesses required, which is particularly beneficial in database systems.

Efficient Disk Usage: B-trees are designed to minimize disk I/O operations, which are often the bottleneck in data retrieval processes. By storing multiple keys in each node and keeping the tree balanced, B-trees ensure that most operations can be performed with a minimal number of disk reads and writes. This efficiency is critical for large-scale data management systems.

Dynamic Growth: B-trees can grow dynamically as new data is inserted. They accommodate growth by splitting nodes and promoting keys, ensuring that the tree remains balanced regardless of how much data is added. This flexibility makes B-trees well-suited for applications where the volume of data is expected to increase over time.

Range Queries: B-trees support efficient range queries, allowing for the retrieval of all keys within a specified range. This feature is particularly useful in database indexing, where range queries are common. The ability to quickly locate and return a set of keys within a given range makes B-trees ideal for tasks such as finding all records within a certain date range or price range.

Robustness: B-trees are resilient to changes in data and maintain their performance characteristics even under heavy load. They handle insertions, deletions, and updates efficiently, ensuring that the tree remains balanced and operations remain fast. This robustness makes B-trees a reliable choice for critical data management applications.

Applications of B-Trees:

Database Indexing: B-trees are widely used in database management systems (DBMS) to index data. Indexes improve query performance by allowing the DBMS to quickly locate the rows that match query conditions. B-trees are used for primary, secondary, and composite indexes, making them a fundamental component of database optimization.

File Systems: Modern file systems use B-trees to manage directories and file metadata. The balanced nature of B-trees ensures quick access to files and efficient storage management. For example, the Btrfs and ReiserFS file systems use B-trees to organize and access file data.

Multilevel Indexing: B-trees are used in multilevel indexing schemes to manage large datasets. By providing a hierarchical structure, B-trees enable efficient indexing of data across multiple levels, reducing the number of I/O operations required to access data. This application is common in large databases and data warehouses.

Caching and Memory Management: B-trees are used in caching and memory management systems to keep track of cached data and manage memory allocation efficiently. The balanced structure of B-trees ensures quick access to frequently used data, improving system performance.

Conclusion

In summary, B trees and B+ trees are both self-balancing tree data structures that are used to store and retrieve data efficiently. They have some similarities, such as the fact that all the leaves are at the same level and each node is at least half full. However, they also have some important differences, such as the fact that B+ trees only store data pointers in the leaf nodes and have linked leaf nodes, while B trees can store data pointers in both the interior and leaf nodes and do not have linked leaf nodes. Both B trees and B+ trees are useful when dealing with large data sets that cannot fit in main memory and require disk seek operations.

5.11 Introduction to Indexing

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

There are two basic kinds of indices:

- **Ordered indices**. Based on a sorted ordering of the values.
- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

Index Evaluation Metrics

- **Access types**: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

An attribute or set of attributes used to look up records in a file is called a **search key**.

5.12 Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.

Clustering indices are also called **primary indices**; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key.

Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary** indices.

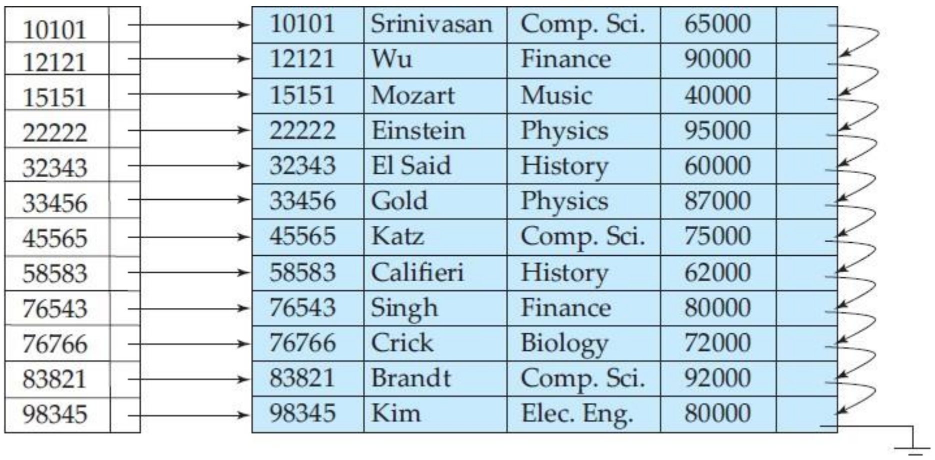
Primary key ordered-> Primary / Clustering index
Non-primary key order -> Secondary / Non-Clustering index

Dense and Sparse Indices

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

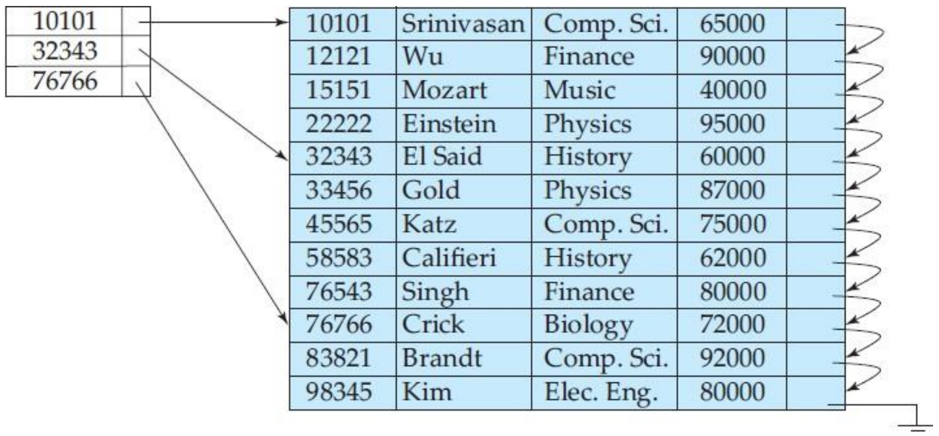
There are two types of ordered indices that we can use:

- **Dense index**: In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.
- **Sparse index**: In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Dense index



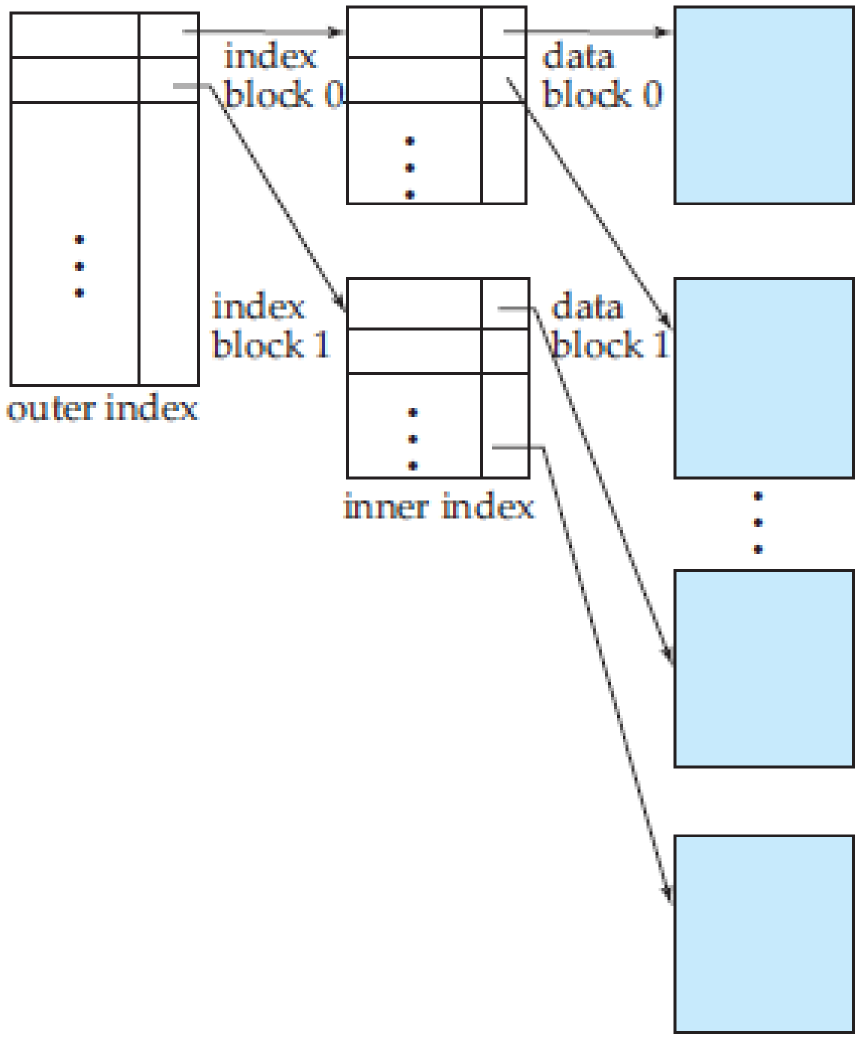
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Sparse index

Multilevel Indices

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

We treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in below figure. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.



Two level sparse index

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

We first describe algorithms for updating single-level indices.

- **Insertion**. First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:
  - **Dense indices:**

    1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:

       a. If the index entry stores pointers to all records with the same search key value, the system adds a pointer to the new record in the index entry.

       b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
  - **Sparse indices:** We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.
- **Deletion**. To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:
  - **Dense indices:**

    1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
    2. Otherwise the following actions are taken:

       a. If the index entry stores pointers to all records with the same search key value, the system deletes the pointer to the deleted record from the index entry.

       b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.
  - **Sparse indices:**

    1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
    2. Otherwise the system takes the following actions:

       a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

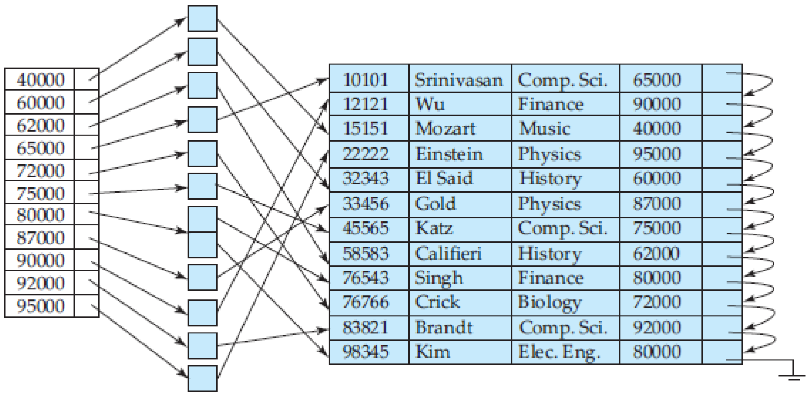
       b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

Below figure shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

5.13 B+ Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.

The **B+ tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.

A B+ tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each non leaf node in the tree has between *n*/2 and *n* children, where *n* is fixed for a particular tree.

Structure of a B+ Tree

A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 11.7 shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

We consider first the structure of the leaf nodes. For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i. Pointer P_n has a special purpose that we shall discuss shortly.

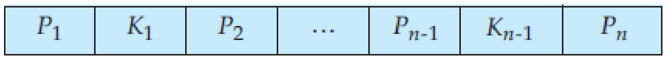
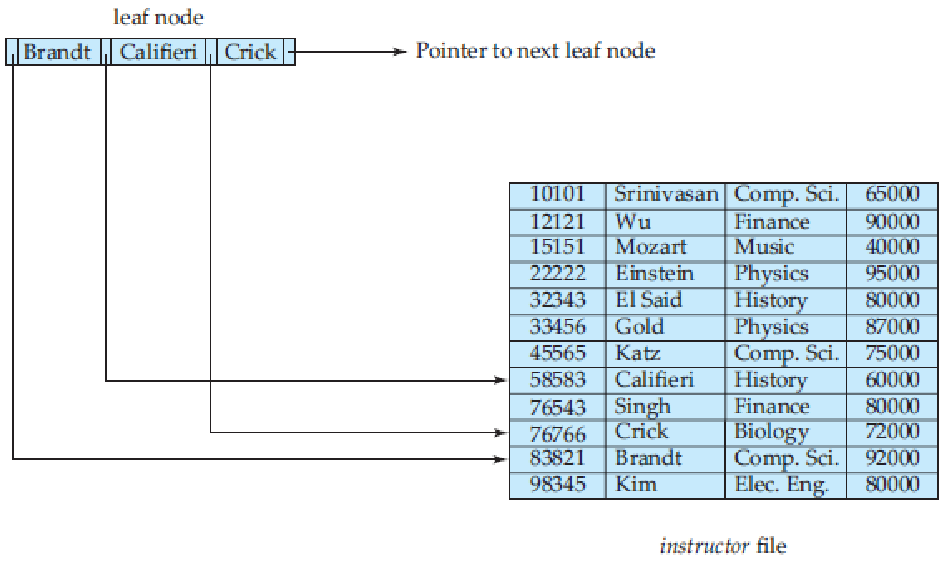


Figure 11.7 Typical node of a B$^+$-tree.

Figure 11.8 shows one leaf node of a B$^+$-tree for the *instructor* file, in which we have chosen n to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n-1)/2 \rceil$ values. With $n = 4$ in our example B$^+$-tree, each leaf must contain at least 2 values, and at most 3 values.



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

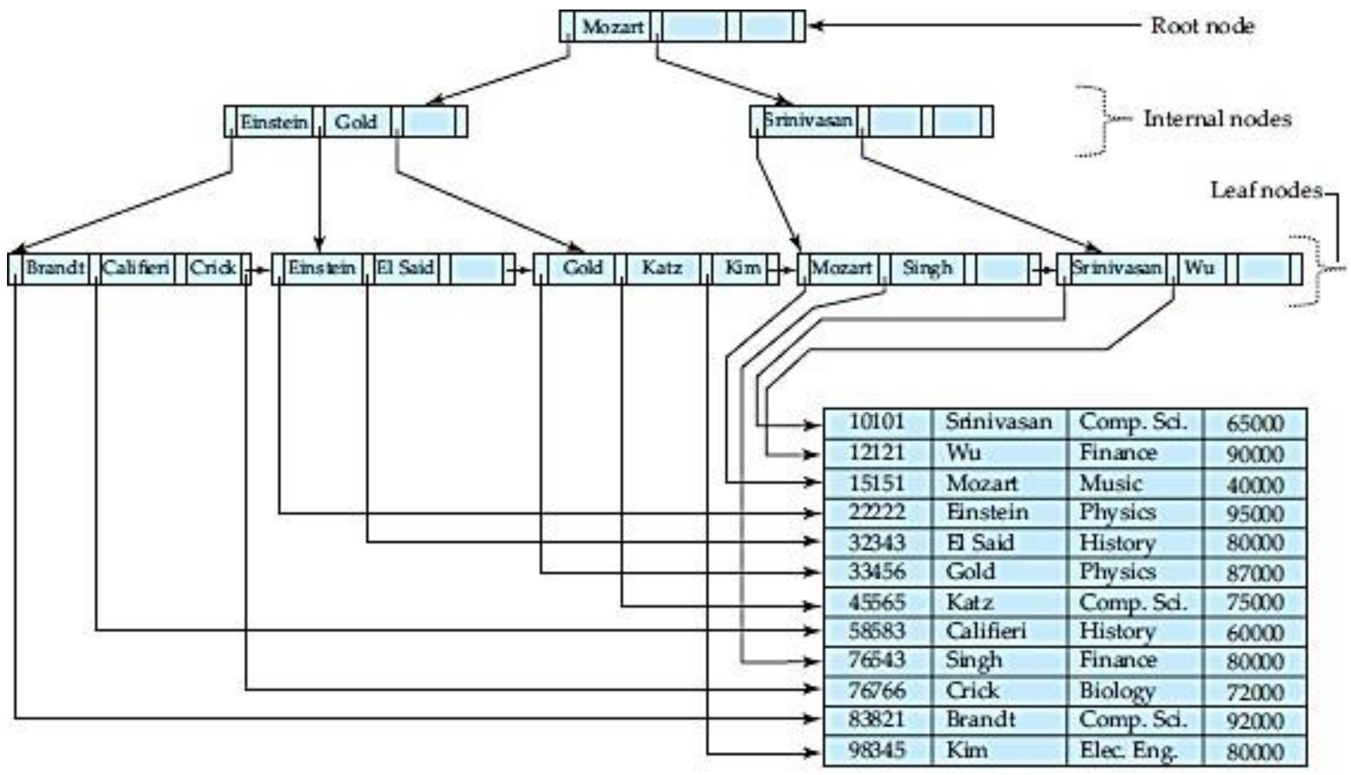
Figure 11.8 A leaf node for *instructor* B$^+$-tree index ($n = 4$).

The nonleaf nodes of the B$^+$-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must* hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as internal nodes.

Let us consider a node containing m pointers ($m \leq n$). For $i = 2, 3, \ldots, m-1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B$^+$-tree, for any n, that satisfies the preceding requirements.

Figure 11.9 shows a complete B$^+$-tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.



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

Figure 11.9 B$^+$-tree for *instructor* file ($n = 4$).

Queries on B$^+$ Trees

Let us consider how we process queries on a B+-tree. Suppose that we wish to find records with a search-key value of *V*.

Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest i such that search-key value K_i is greater than or equal to V. Suppose such a value is found; then, if K_i is equal to V, the current node is set to the node pointed to by P_{i+1}, otherwise $K_i > V$, and the current node is set to the node pointed to by Pi. If no such value Ki is found, then clearly $V > K_{m-1}$, where P_m is the last non null pointer in the node. In this case the current node is set to that pointed to by P_m. The above procedure is repeated, traversing down the tree until a leaf node is reached.

At the leaf node, if there is a search-key value equal to V, let Ki be the first such value; pointer Pi directs us to a record with search-key value Ki. The function then returns the leaf node L and the index i. If no search-key with value V is found in the leaf node, no record with key value V exists in the relation, and function find returns null, to indicate failure.

```
function find(value V)
/* Returns leaf node C and index i such that C.P_i points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that $V \leq C.K_i$
        if there is no such number i then begin
            Let $P_m$ = last non-null pointer in the node
            Set C = $C.P_m$
        end
        else if ($V = C.K_i$)
            then Set C = $C.P_{i+1}$
        else C = $C.P_i$ /* $V < C.K_i$ */
    end
    /* C is a leaf node */
    Let i be the least value such that $K_i = V$
    if there is such a value i
        then return (C, i)
        else return null ; /* No record with key value V exists*/
```

Updates on B$^+$ Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly.

Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for "Adams" into the B$^+$-tree of Figure 11.9. Using the algorithm for lookup, we find that "Adams" should appear in the leaf node containing "Brandt", "Califieri", and "Crick." There is no room in this leaf to insert the search-key value "Adams." Therefore, the node is *split* into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting "Adams". The search-key values "Adams" and "Brandt" are in one leaf, and "Califieri" and "Crick" are in the other. In general, we take the n search-key values (the $n - 1$ values in the leaf node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B$^+$-tree structure. In our example, the new node has "Califieri" as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B$^+$-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

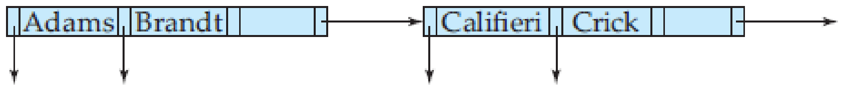


Figure 11.12 Split of leaf node on insertion of "Adams"

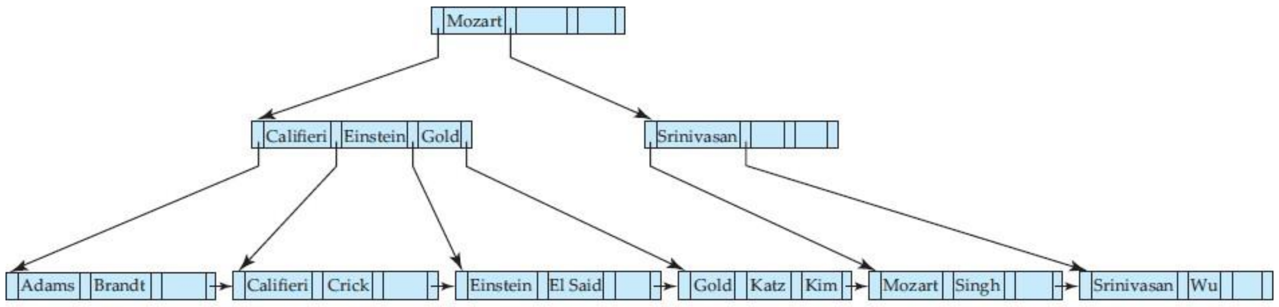


Figure 11.13 Insertion of "Adams" into the B$^+$-tree of Figure 11.9.

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n − 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n − 1 key values already, split it */
            Create node L′
            Copy L.P_1 ... L.K_{n−1} to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L′.P_n = L.P_n; Set L.P_n = L′
            Erase L.P_1 through L.K_{n−1} from L
            Copy T.P_1 through T.K_{⌈n/2⌉} from T into L starting at L.P_1
            Copy T.P_{⌈n/2⌉+1} through T.K_n from T into L′ starting at L′.P_1
            Let K′ be the smallest key-value in L′
            insert_in_parent(L, K′, L′)
        end

procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than K
            Insert P, K into L just after T.K_i
        end
```

```
procedure insert_in_parent(node N, value K′, node N′)
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K′, N′    /* N and N′ are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K′, N′) in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K′, N′)
            Insert (K′, N′) into T just after N
            Erase all entries from P; Create node P′
            Copy $T.P_1 \ldots T.P_{\lceil n/2 \rceil}$ into P
            Let $K'' = T.K_{\lceil n/2 \rceil}$
            Copy $T.P_{\lceil n/2 \rceil+1} \ldots T.P_{n+1}$ into P′
            insert_in_parent(P, K″, P′)
        end
```

Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete “Srinivasan” from the B$^+$-tree of Figure 11.13. The resulting B$^+$-tree appears in Figure 11.16. We now consider how the deletion is performed. We first locate the entry for “Srinivasan” by using our lookup algorithm. When we delete the entry for “Srinivasan” from its leaf node, the node is left with only one entry, “Wu”. Since, in our example, $n = 4$ and $1 < \lceil (n-1)/2 \rceil$, we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for “Wu” can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing “Srinivasan”. (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value “Srinivasan” and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < \lceil n/2 \rceil$ for $n = 4$, the parent node is underfull. (For larger n, a node that becomes underfull would still have some values as well as pointers.)

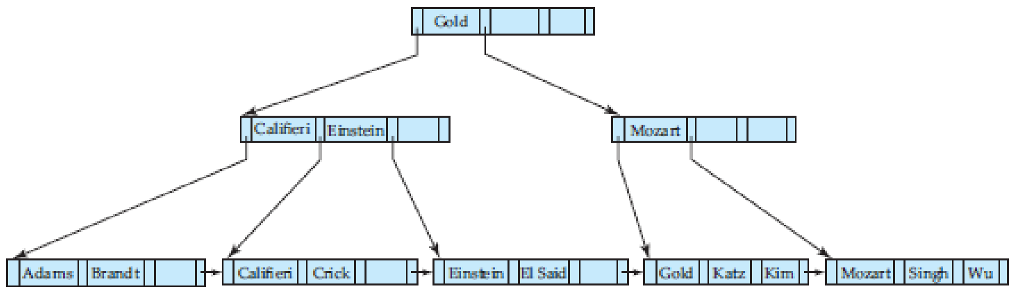


Figure 11.16 Deletion of "Srinivasan" from the B⁺-tree of Figure 11.13.

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all ($K_i$, $P_i$) pairs in N to N'; set $N'.P_n = N.P_n$
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that $N'.P_m$ is the last pointer in N'
                    remove ($N'.K_{m-1}$, $N'.P_m$) from N'
                    insert ($N'.P_m$, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by $N'.K_{m-1}$
                end
                else begin
                    let m be such that ($N'.P_m$, $N'.K_m$) is the last pointer/value
                        pair in N'
                    remove ($N'.P_m$, $N'.K_m$) from N'
                    insert ($N'.P_m$, $N'.K_m$) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by $N'.K_m$
                end
            end
            else ... symmetric to the then case ...
        end
    end
```

5.14 Hash Organization

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** – A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.

Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

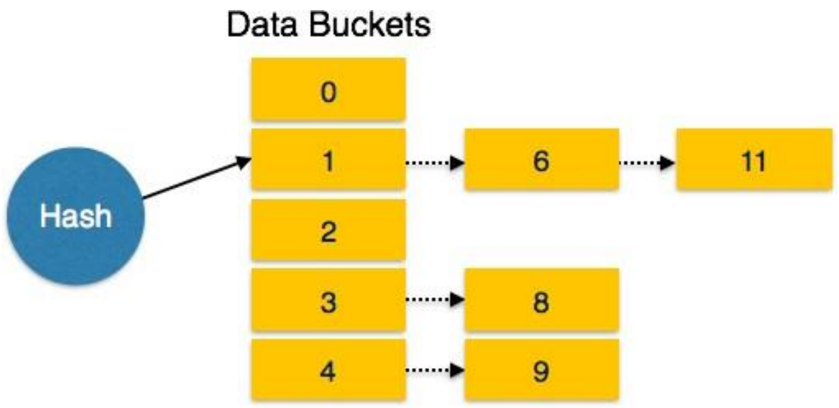
  Bucket address = h(K)

- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** – This is simply a search followed by a deletion operation.

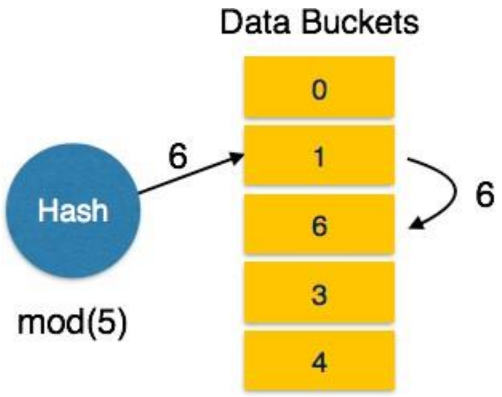
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



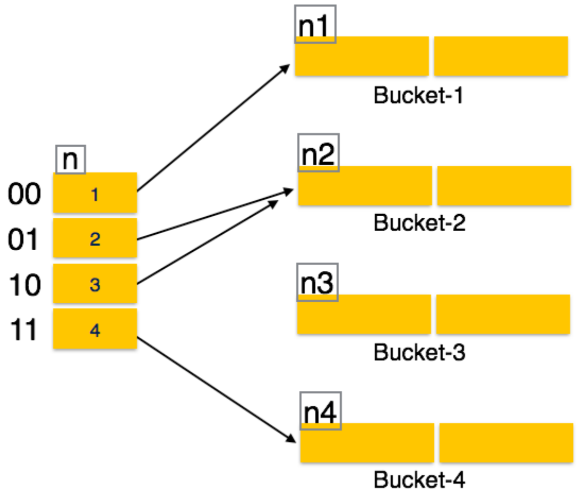
- **Linear Probing –** When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



5.15 COMPARISON OF THREE FILE ORGANIZATIONS

Refer text book

Database Management Systems, 3/e, Raghurama Krishnan, Johannes Gehrke, TMH

Chapter 8 File Organizations & Indexes

Pages from 232 to 236.

Review Questions

1. Explain about the measures that are to be considered for comparing the performance of various file organization techniques.
2. Explain in detail B+ tree file organization.
3. Write short notes on: i) Primary index ii) Clustered index iii) Secondary index.
4. Explain various anomalies that arise due to interleaved execution of transactions with suitable examples.
5. What is static hashing? What rules are followed for index selection?
6. Define transaction and explain desirable properties of transactions.
7. What is database Recovery? Explain Shadow paging in detail.
8. Explain about Conflict Serializability and view serializability.
9. Explain the following a) Concurrent executions, b) Transaction states.

References:

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J. Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.