Introduction:

A B-tree is a data structure used in computer science for organizing and storing data in a balanced tree-like format.

The "B" in B-tree stands for "balanced," which refers to the fact that the tree is kept balanced by maintaining a constant maximum and minimum number of children for each node

Key Features:

Balanced Structure:

B-trees are self-balancing tree data structures, meaning they maintain a balanced structure as nodes are added or removed. This ensures the following:

* Height of the tree is low
* Low height results in less traversal and this optimises the search
* The leaf nodes of the btree are on the same level

Node structure and order:

A node of the B-tree can contain n number of keys n+1 pointers to the children. Here n is defined as the order of the tree which dictates the maximum number of keys in a B-tree. An list is used to store the keys and children

Node structure:

It includes the number of keys in the node

Array of keys

Array of pointer to the children

Leaf flag indicating whether it is leaf node or not

Creation of a Node:

Input: (True or False) to Find whether a Node is leaf Node

Output: a pointer to a newly created node

1.Allocate Memory dynamically to node

2. Allocate Memory to the keys array such that size is 2t-1 where t is the Maximum of children(user input)

3. Allocate Memory to the children pointer array with the size of 2t

4. Initialise the number of keys to 0

5. The leaf status of the node should be set to the input

6. Return the pointer

Time complexity for creating a node:

Time complexity for creating a node:

It solely depends on the Max number of children which provides the size of the memory allocation so it is O(n)

INSERTION OF KEY TO A NODE:

When inserting a node into a B-tree, several conditions need to be considered to maintain the B-tree properties. These conditions ensure that the resulting tree remains balanced and that the keys are properly ordered:

* Tree is empty:

Allocate a root node and insert the key

* Not empty and Not Full:

Check whether it is a leaf node or not:

If it is a leaf node:

Iterate backwards and check whether

a key present in the node is greater

than the given number, if greater

move the key one space higher and

create a spot for the number

finally, change the number of keys to n+1

If not a leaf node:

Iterate backwards until you a key

Which is less than the number provided or until you reach the boundary

Check if the child node at the index i+1 is full, if full split the node at median of the sorted keys and push the median as parent key and remaining keys as children

Else insert the key in the child node

If the Node is full:

* + Insert the elements in the increasing order and split and move the median upwards
  + Make the elements on the right side of the median as right child and left side elements as left child
  + If the upper node is not full insert the nodes in the increasing order
  + If it is full, repeat the process

Time complexity:

Best case scenario:

When there is no split operation and searching the node takes O(1)

The algorithm takes O(1) for insertion

Worst case scenario:

When there are multiple split operations and searching takes O(log n) and split operations take O(n)

The algorithm for insertion takes O(log n)

Searching Algorithm:

Input: A pointer to root node , number to be searched

Output: A pointer to node containing the number if found else a NULL pointer

Algorithm:

* Loop through the keys in the node until n(size of array) and while num is greater than value in key
* If number equals the value in key, return the node
* If number is not in any of the keys, then move to child of the key whose value is greater than num similar to binary search tree
* On traversal if the pointer reaches the leaf and the element is not found return NULL

Time Complexity:

Best Case:

If the element is found in the current without the need for traversal and it is present at the current key then it is O(1)

Worst Case:

If the element traverses all the keys in the node -O(m)

and also traverses the height of the tree - O(log n)  
Then the time complexity is O(log n.m)

DISPLAY ALGORITHM:

Input: A pointer to the root node (temp)

Output: elements of the btree is printed

Algorithm:

* Initialise i to 0
* Iterate through each key in the node till n(number of keys:
  + 1. If temp is not a leaf node, repeat the process for the child node at index i
    2. Print the value present in that index

Time complexity:

If the tree contains only the root node then the time complexity is O(n)

If it contains h levels in the tree and the nodes are not compeletely filled then the time complexity is O(log n)

Where n being the number of keys

DELETION OF A KEY IN A NODE:

Input: pointer to the root(temp)and the number to be deleted

Output: indicate whether deleted or not

Algorithm:

* If temp is NULL,return 0
* Else search if the number is present in the array of keys,if yes:
  + 1. Check whether it is a leaf node:
       - 1. If the number of keys in children >minimum number of keys, then

Let i be the index of the element in the keys array then all the elements from i+1 through left by 1 space

(b)If on deleting an element, the number of keys becomes less than min no of keys,then

2. look for the largest key in the left sibling (only if it has greater than minimum number of keys)and then transfer the sibling key to the parent node and transfer the parent key to the node(current) and the delete the num

3.if 2 failed, look for the minimum key in the right sibling (only if the number of keys > minimum number of keys) and repeat the similar procedure

4. If both 2 and 3 failed, then merge the current node and the sibling and the parent key and delete the num and balance the tree

ii) If not a leaf node:

Then find the predecessor that is the maximum value in left sibling and replace value with the number to be deleted in the parent node only if number of keys >minimum of number of keys in lst

Then find the successor that is the minimum value in right sibling and replace value with the number to be deleted in the parent node only if number of keys >minimum of number of keys in lst

If both 2 and 3 failed, then merge both the siblings with the key and delete the key and reduce the height of the tree

TIME COMPLEXITY ANALYSIS FOR DELETION:

Best case scenario:

When the node is found near the root node the time complexity is O(log n)

Worst case scenario:

When the node is a leaf node and found in deeper parts

The time complexity is O(logn)

Applications of B-tree:

B-trees are widely used in various applications where efficient insertion, deletion, and retrieval of large datasets are essential.

Some common applications of B-trees include:

Database Systems: B-trees are extensively used in database management systems (DBMS) to implement indexes, particularly for primary and secondary keys. They enable efficient searching, inserting, updating, and deleting records in large datasets.

File Systems: B-trees are used in file systems to organize and manage disk blocks efficiently. They facilitate quick access to files and directories by maintaining balanced tree structures on disk.

File Transfer Protocols: B-trees are employed in file transfer protocols like FTP and HTTP to index files and directories on servers. This allows for fast and efficient retrieval of files during file transfers.

Web Browsers: B-trees are utilized in web browsers to store and manage bookmarks, history, and cache data. They enable quick access to web pages and resources, enhancing the browsing experience.

Distributed Systems: B-trees are utilized in distributed systems to maintain distributed indexes and metadata. They help in organizing and searching data efficiently across multiple nodes in the system.

Geospatial Databases: B-trees are used in geospatial databases to index spatial data such as maps, locations, and geographic information systems (GIS) data. They support spatial queries and operations efficiently.

Compiler Design: B-trees are employed in compilers and interpreters to implement symbol tables, which store information about variables, functions, and other identifiers in a program. They facilitate fast symbol lookup during compilation and execution.

Operating Systems: B-trees are used in operating systems for various purposes such as managing process control blocks, maintaining file system structures, and organizing system resources.

Caching Systems: B-trees are utilized in caching systems to organize and manage cached data efficiently. They enable quick retrieval and eviction of cached items based on access patterns and cache policies.

Network Routing: B-trees are employed in network routing tables to store and search routing information efficiently. They help in determining the optimal paths for data packets in computer networks