11/14/2018

Chen Wang

Fudan University

Data Structure and Algorithms

Project Documentation

Table of Contents

[I. General structure of project 3](#_Toc529999128)

[II. Theoretical analysis of Red black tree 3](#_Toc529999129)

[1. Binary Search Tree 3](#_Toc529999130)

[1) Definition 3](#_Toc529999131)

[2) Order relation 4](#_Toc529999132)

[2. Operations on Binary Search Tree 4](#_Toc529999133)

[1) Searching 4](#_Toc529999134)

[2) Insertion 4](#_Toc529999135)

[3) Deletion 5](#_Toc529999136)

[4) Traversal 5](#_Toc529999137)

[3. Self-balancing Binary Search Tree 6](#_Toc529999138)

[1) Overview of Self-balancing Binary Search Tree 6](#_Toc529999139)

[2) Implementation of Self-balancing Binary Search Tree 7](#_Toc529999140)

[3) Application of Self-balancing Binary Search Tree 7](#_Toc529999141)

[4. Red black Tree 7](#_Toc529999142)

[1) Overview of Red black Tree 8](#_Toc529999143)

[2) Terminology of Red black Tree 8](#_Toc529999144)

[3) Properties of Red black Tree 8](#_Toc529999145)

[5. Operations on Red black Tree 9](#_Toc529999146)

[1) Diagram notes 9](#_Toc529999147)

[2) Insertion 10](#_Toc529999148)

[3) Removal 11](#_Toc529999149)

[6. Proof of asymptotic bounds 14](#_Toc529999150)

[III. Theoretical analysis of B+ tree 15](#_Toc529999151)

[1. N-ary Tree 15](#_Toc529999152)

[1) Types of k-ary trees 15](#_Toc529999153)

[2) Properties of k-ary trees 15](#_Toc529999154)

[2. B-Tree 15](#_Toc529999155)

[1) Overview of B-tree 15](#_Toc529999156)

[2) Variants of B-tree 16](#_Toc529999157)

[3) B-tree usage in database 17](#_Toc529999158)

[4) Advantages of B-tree usage for databases 19](#_Toc529999159)

[5) Technical description 19](#_Toc529999160)

[6) Best case and worst-case heights 20](#_Toc529999161)

[3. Algorithms of B-Tree 21](#_Toc529999162)

[1) Search 21](#_Toc529999163)

[2) Insertion 21](#_Toc529999164)

[3) Deletion 21](#_Toc529999165)

[4) Sequential access 23](#_Toc529999166)

[5) Initial construction 23](#_Toc529999167)

[4. B+ Tree 24](#_Toc529999168)

[5. Algorithms in B+ Tree 24](#_Toc529999169)

[1) Search 24](#_Toc529999170)

[2) Prefix key compression 25](#_Toc529999171)

[3) Insertion 25](#_Toc529999172)

[4) Deletion 25](#_Toc529999173)

[5) Bulk-loading 25](#_Toc529999174)

[IV. Specification and details of the requirements 26](#_Toc529999175)

[V. Implementation structure 26](#_Toc529999176)

# General structure of project

The aim of the project is to implement the two data structures, red black tree and the B+ tree. In order to utilize the trees that we have implemented and to examine whether we have correctly realized all the required work, an English-Chinese dictionary is used as the user of the data structure. Red-black tree is a kind of binary search tree and also a kind of self-balancing search tree, while B+ tree is not a binary search tree when still being self-balancing search tree.

The general structure and functionality of Red black tree and B+ tree is presented as the following sections and the detail implementation is described in the following sections.

# Theoretical analysis of Red black tree

## Binary Search Tree

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Several variants of the binary search tree have been studied in computer science; this article deals primarily with the basic type, making references to more advanced types when appropriate.

### Definition

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another.

Frequently, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records. The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

* When inserting or searching for an element in a binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found.
* The shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.
* After a long-intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys, which grows much faster than .
* There has been a lot of research to prevent degeneration of the tree resulting in worst case time complexity of .

### Order relation

Binary search requires an order relation by which every element (item) can be compared with every other element in the sense of a [total preorder](https://en.wikipedia.org/wiki/Total_preorder). The part of the element which effectively takes place in the comparison is called its *key*. Whether duplicates, i.e. different elements with same key, shall be allowed in the tree or not, does not depend on the order relation, but on the application only.

In the context of binary search trees a total preorder is realized most flexibly by means of a [three-way comparison](https://en.wikipedia.org/wiki/Three-way_comparison) [subroutine](https://en.wikipedia.org/wiki/Subroutine).

## Operations on Binary Search Tree

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

### Searching

Searching a binary search tree for a specific key can be programmed recursively or iteratively. We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found after a *null* subtree is reached, then the key is not present in the tree.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also, in case of equality search down to the leaves in a direction specified by the user. A binary tree sort equipped with such a comparison function becomes stable.

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree’s *height* (see tree terminology). On average, binary search trees with *n* nodes have height. However, in the worst case, binary search trees can have height, when the unbalanced tree resembles a linked list (degenerate tree).

### Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair as its right or left child, depending on the node’s key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

The operation requires time proportional to the height of the tree in the worst case, which is time in the average case over all trees, but time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root’s, it is then compared with the key of the root’s left child. If its key is greater, it is compared with the root’s right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node’s right or left child, depending on its key: if the key is less than the leaf’s key, then it is inserted as the leaf’s left child, otherwise as the leaf’s right child.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

### Deletion

When removing a node from a binary *search* tree it is mandatory to maintain the in-order sequence of the nodes. There are many possibilities to do this. However, the following method which has been proposed by T. Hibbard in 1962 guarantees that the heights of the subject subtrees are changed by at most one. There are three possible cases to consider:

* Deleting a node with no children: simply remove the node from the tree.
* Deleting a node with one child: remove the node and replace it with its child.
* Deleting a node with two children: call the node to be deleted *D*. Do not delete *D*. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node *E*. Copy the user values of *E* to *D*. If *E* does not have a child simply remove *E* from its previous parent *G*. If *E* has a child, say *F*, it is a right child. Replace *E* with *F* at *E*’s parent.

Deleting a node with two children from a binary search tree. First the leftmost node in the right subtree, the in-order successor *E*, is identified. Its value is copied into the node *D* being deleted. The in-order successor can then be easily deleted because it has at most one child. The same method works symmetrically using the in-order predecessor *C*.

In all cases, when *D* happens to be the root, make the replacement node root again. Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node’s in-order successor is its right subtree’s left-most child, and a node’s in-order predecessor is the left subtree’s right-most child. In either case, this node will have only one or no child at all. Delete it according to one of the two simpler cases above.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an [unbalanced](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) tree, so some implementations select one or the other at different times.

Runtime analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus, in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

### Traversal

Once the binary search tree has been created, its elements can be retrieved in-order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it’s recursively accessed. As with all binary trees, one may conduct a pre-order traversal or a post-order traversal, but neither are likely to be useful for binary *search* trees. An in-order traversal of a binary search tree will always result in a sorted list of node items (numbers, strings or other comparable items).

Traversal requires time, since it must visit every node. This algorithm is also , so it is asymptotically optimal.

Traversal can also be implemented iteratively. For certain applications, e.g. greater equal search, approximative search, an operation for *single step (iterative) traversal* can be very useful. This is, of course, implemented without the callback construct and takes on average and in the worst case.

## Self-balancing Binary Search Tree

In computer science, a self-balancing (or height-balanced) binary search tree is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.

These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.

The red-black tree, which is a type of self-balancing binary search tree, was called symmetric binary B-tree and was renamed but can still be confused with the generic concept of self-balancing binary search tree because of the initials.

### Overview of Self-balancing Binary Search Tree

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. A binary tree with height *h* can contain at most nodes. It follows that for any tree with *n* nodes and height *h*:

And that implies:

In other words, the minimum height of a binary tree with *n* nodes is , rounded down; that is, .

However, the simplest algorithms for BST item insertion may yield a tree with height *n* in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a linked list with *n* nodes. The difference in performance between the two situations may be enormous: for *n* = 1,000,000, for example, the minimum height is .

If the data items are known ahead of time, the height can be kept small, in the average sense, by adding values in a random order, resulting in a random binary search tree. However, there are many situations where this randomization is not viable.

Self-balancing binary trees solve this problem by performing transformations on the tree (such as tree rotations) at key insertion times, in order to keep the height proportional to . Although a certain overhead is involved, it may be justified in the long run by ensuring fast execution of later operations.

Maintaining the height always at its minimum value is not always viable; it can be proven that any insertion algorithm which did so would have an excessive overhead. Therefore, most self-balanced BST algorithms keep the height within a constant factor of this lower bound.

In the asymptotic (“Big-O”) sense, a self-balancing BST structure containing *n* items allows the lookup, insertion, and removal of an item in worst-case time, and ordered enumeration of all items in time. For some implementations these are per- operation time bounds, while for others they are amortized bounds over a sequence of operations. These times are asymptotically optimal among all data structures that manipulate the key only through comparisons.

### Implementation of Self-balancing Binary Search Tree

Popular data structures implementing this type of tree include:

* 2-3 tree
* AA tree
* AVL tree
* B-tree
* Red-black tree
* Scapegoat tree
* Splay tree
* Weight-balanced tree

### Application of Self-balancing Binary Search Tree

Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self- balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Self-balancing BSTs have better worst-case lookup performance than hash tables (compared to ), but have worse average-case performance ( compared to ).

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than merge sort, quicksort, or heapsort, because of the tree-balancing overhead as well as cache access patterns.)

Self-balancing BSTs are flexible data structures, in that it’s easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

## Red black Tree

### Overview of Red black Tree

A red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in time, where *n* is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in time.

Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a classic (uncolored) binary search tree. In many cases, the additional bit of information can be stored at no additional memory cost.

### Terminology of Red black Tree

A red–black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as text fragments or numbers.

The leaf nodes of red–black trees contain no data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save execution time, sometimes a pointer to a single sentinel node (instead of a null pointer) performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.

Red–black trees, like all binary search trees, allow efficient in-order traversal (that is: in the order Left–Root–Right) of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree of *n* nodes, having the least possible tree height, results in search time.

### Properties of Red black Tree

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red-black tree:

1. Each node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

Some definitions: the number of black nodes from the root to a node is the node’s black depth; the uniform number of black nodes in all paths from root to the leaves is called the black-height of the red-black tree.

These constraints enforce a critical property of red–black trees: *the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf*. The result is that the tree is roughly height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst case, unlike ordinary binary search trees.

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red–black tree *T*, let *B* be the number of black nodes in *property 5*. Let the shortest possible path from the root of *T* to any leaf consist of *B* black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore, ignoring any black NIL leaves, the longest possible path consists of *2\*B* nodes, alternating black and red (this is the worst case). Counting the black NIL leaves, the longest possible path consists of *2\*B-1* nodes.

*The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes*. Since all maximal paths have the same number of black nodes, by property 5, this shows that *no path is more than twice as long as any other path*.

## Operations on Red black Tree

Read-only operations on a red–black tree require no modification from those used for binary search trees, because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number ( or amortized ) of color changes (which are very quick in practice) and no more than three tree rotations (two for insertion). Although insert and delete operations are complicated, their times remain .

### Diagram notes

1. The label **N** will be used to denote the current node in each case. At the beginning, this is the insertion node or the replacement node and a leaf, but the entire procedure may also be applied recursively to other nodes (see case 3).
2. **P** will denote **N**’s parent node, **G** will denote **N**’s grandparent, **S** will denote **N**’s sibling, and **U** will denote **N**’s uncle (i.e., the sibling of a node’s parent, as in human family trees).
3. In between some cases, the roles and labels of the nodes are shifted, but within each case, every label continues to represent the same node throughout.
4. In the diagrams a blue border rings the current node **N** in the left (current) half and rings the node that will become **N** in the right (target) half. In the next step, the other nodes will be newly assigned relative to it.
5. Red or black shown in the diagram is either assumed in its case or implied by those assumptions. White represents either red or black, but is the same in both halves of the diagram.
6. A numbered triangle represents a subtree of unspecified depth. A black circle atop a triangle means that black-height of that subtree is greater by one compared to a subtree without this circle.

### Insertion

Insertion begins by adding the node in a very similar manner as a standard binary search tree insertion and by coloring it red. The big difference is that in the binary search tree a new node is added as a leaf, whereas leaves contain no information in the red-black tree, so instead the new node replaces an existing leaf and then has two black leaves of its own added.

What happens next depends on the color of other nearby nodes. There are several cases of red–black tree insertion to handle:

* **N** is the root node, i.e., first node of red–black tree
* **N**’s parent (**P**) is black
* **P** is red (so it can’t be the root of the tree) and **N**’s uncle (**U**) is red
* **P** is red and **U** is black

Note that:

* Property 1 (every node is either red or black) and Property 3 (all leaves are black) always holds.
* Property 2 (the root is black) is checked and corrected with case 1.
* Property 4 (red nodes have only black children) is threatened only by adding a red node, repainting a node from black to red, or a rotation.
* Property 5 (all paths from any given node to its leaves have the same number of black nodes) is threatened only by adding a black node, repainting a node, or a rotation.

**Case 1:** The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

**Case 2:** The current node’s parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

*Note:* In the following cases it can be assumed that **N** has a grandparent node **G**, because its parent **P** is red, and if it were the root, it would be black. Thus, **N** also has an uncle node **U**, although it may be a leaf in case 4.

*Note:* In the remaining cases, it is shown in the diagram that the parent node **P** is the left child of its parent even though it is possible for **P** to be on either side. The code samples already cover both possibilities.

**Case 3:** If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent **G** becomes red to maintain property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes). Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate Property 2 (The root is black) if it is the root or Property 4 (Both children of every red node are black) if it has a red parent. To fix this, the tree’s red-black repair procedure is rerun on **G**.

Note that this is a tail-recursive call, so it could be rewritten as a loop. Since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur.

**Case 4, step 1:** The parent **P** is red but the uncle **U** is black. The ultimate goal will be to rotate the current node into the grandparent position, but this will not work if the current node is on the “inside” of the subtree under **G** (i.e., if **N** is the left child of the right child of the grandparent or the right child of the left child of the grandparent). In this case, a left rotation on **P** that switches the roles of the current node **N** and its parent **P** can be performed. The rotation causes some paths (those in the sub-tree labelled “1”) to pass through the node **N** where they did not before. It also causes some paths (those in the sub-tree labelled “3”) not to pass through the node **P** where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this step has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to step 2.

**Case 4, step 2:** The current node **N** is now certain to be on the “outside” of the subtree under **G** (left of left child or right of right child). In this case, a right rotation on **G** is performed; the result is a tree where the former parent **P** is now the parent of both the current node **N** and the former grandparent **G**. **G** is known to be black, since its former child **P** could not have been red without violating property 4. Once the colors of **P** and **G** are switched, the resulting tree satisfies property 4 (both children of every red node are black). Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through **G** before, and now they all go through **P**.

Note that inserting is actually in-place, since all the calls above use [tail recursion](https://en.wikipedia.org/wiki/Tail_recursion).

In the algorithm above, all cases are called only once, except in Case 3 where it can recurse back to Case 1 with the grandparent node, which is the only case where an iterative implementation will effectively loop. Because the problem of repair in that case is escalated two levels higher each time, it takes maximally ​iterations to repair the tree (where is the height of the tree). Because the probability for escalation decreases exponentially with each iteration the average insertion cost is practically constant.

### Removal

In a regular binary search tree when deleting a node with two non-leaf children, we find either the maximum element in its left subtree (which is the in-order predecessor) or the minimum element in its right subtree (which is the in-order successor) and move its value into the node being deleted. We then delete the node we copied the value from, which must have fewer than two non-leaf children. (Non-leaf children, rather than all children, are specified here because unlike normal binary search trees, red-black trees can have leaf nodes anywhere, which are actually the sentinel Nil, so that all nodes are either internal nodes with two children or leaf nodes with, by definition, zero children. In effect, internal nodes having two leaf children in a red-black tree are like the leaf nodes in a regular binary search tree.) Because merely copying a value does not violate any red-black properties, this reduces to the problem of deleting a node with at most one non-leaf child. Once we have solved that problem, the solution applies equally to the case where the node we originally want to delete has at most one non-leaf child as to the case just considered where it has two non-leaf children.

Therefore, for the remainder of this discussion we address the deletion of a node with at most one non-leaf child. We use the label **M** to denote the node to be deleted; **C** will denote a selected child of **M**, which we will also call “its child”. If **M** does have a non-leaf child, call that its child, **C**; otherwise, choose either leaf as its child, **C**.

If **M** is a red node, we simply replace it with its child **C**, which must be black by property 4. (This can only occur when **M** has two leaf children, because if the red node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would violate property 5.) All paths through the deleted node will simply pass through one fewer red node, and both the deleted node’s parent and child must be black, so property 3 (all leaves are black) and property 4 (both children of every red node are black) still hold.

Another simple case is when **M** is black and **C** is red. Simply removing a black node could break Properties 4 (“Both children of every red node are black”) and 5 (“All paths from any given node to its leaf nodes contain the same number of black nodes”), but if we repaint **C** black, both of these properties are preserved.

The complex case is when both **M** and **C** are black. (This can only occur when deleting a black node which has two leaf children, because if the black node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would have been an invalid red–black tree by violation of property 5.) We begin by replacing **M** with its child **C** – we recall that in this case “its child **C**” is either child of **M**, both being leaves. We will *relabel* this child **C** (in its new position) **N**, and its sibling (its new parent’s other child) **S**. (**S** was previously the sibling of **M**.) In the diagrams below, we will also use **P** for **N**’s new parent (**M**’s old parent), **SL** for **S**’s left child, and **SR** for **S**’s right child (**S** cannot be a leaf because if **M** and **C** were black, then **P**’s one subtree which included **M** counted two black-height and thus **P**’s other subtree which includes **S** must also count two black-height, which cannot be the case if **S** is a leaf node).

*Note*: In order for the tree to remain well-defined, we need every null leaf to remain a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child **N**, it is easy to see that the property is satisfied. If, on the other hand, **N** would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

We can perform the steps outlined above with the following code, where the function replace\_node substitutes child into n’s place in the tree. For convenience, code in this section will assume that null leaves are represented by actual node objects rather than NULL (the code in the *Insertion* section works with either representation).

*Note*: If **N** is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling delete\_case1() on its parent (the node that we delete, n in the code above) and deleting it afterwards. We do this if the parent is black (red is trivial), so it behaves in the same way as a null leaf (and is sometimes called a ‘phantom’ leaf). And we can safely delete it at the end as n will remain a leaf after all operations, as shown above. In addition, the sibling tests in cases 2 and 3 require updating as it is no longer true that the sibling will have children represented as objects.

If both **N** and its original parent are black, then deleting this original parent causes paths which proceed through **N** to have one fewer black node than paths that do not. As this violates property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

**Case 1:** **N** is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

*Note*: In cases 2, 5, and 6, we assume **N** is the left child of its parent **P**. If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.

**Case 2:** **S** is red. In this case we reverse the colors of **P** and **S**, and then rotate left at **P**, turning **S** into **N**’s grandparent. Note that **P** has to be black as it had a red child. The resulting subtree has a path short one black node so we are not done. Now **N** has a black sibling and a red parent, so we can proceed to step 4, 5, or 6. (Its new sibling is black because it was once the child of the red **S**.) In later cases, we will relabel **N**’s new sibling as **S**.

**Case 3:** **P**, **S**, and **S**’s children are black. In this case, we simply repaint **S** red. The result is that all paths passing through **S**, which are precisely those paths *not* passing through **N**, have one less black node. Because deleting **N**’s original parent made all paths passing through **N** have one less black node, this evens things up. However, all paths through **P** now have one fewer black node than paths that do not pass through **P**, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is still violated. To correct this, we perform the rebalancing procedure on **P**, starting at case 1.

**Case 4:** **S** and **S**’s children are black, but **P** is red. In this case, we simply exchange the colors of **S** and **P**. This does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths.

**Case 5:** **S** is black, **S**’s left child is red, **S**’s right child is black, and **N** is the left child of its parent. In this case we rotate right at **S**, so that **S**’s left child becomes **S**’s parent and **N**’s new sibling. We then exchange the colors of **S** and its new parent. All paths still have the same number of black nodes, but now **N** has a black sibling whose right child is red, so we fall into case 6. Neither **N** nor its parent are affected by this transformation. (Again, for case 6, we relabel **N**’s new sibling as **S**.)

**Case 6:** **S** is black, **S**’s right child is red, and **N** is the left child of its parent **P**. In this case we rotate left at **P**, so that **S** becomes the parent of **P** and **S**’s right child. We then exchange the colors of **P** and **S**, and make **S**’s right child black. The subtree still has the same color at its root, so Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes) are not violated. However, **N** now has one additional black ancestor: either **P** has become black, or it was black and **S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node.

Meanwhile, if a path does not go through **N**, then there are two possibilities:

It goes through **N**’s new sibling **SL**, a node with arbitrary color and the root of the subtree labeled **3** (s. diagram). Then, it must go through **S** and **P**, both formerly and currently, as they have only exchanged colors and places. Thus, the path contains the same number of black nodes.

It goes through **N**’s new uncle, **S**’s right child. Then, it formerly went through **S**, **S**’s parent, and **S**’s right child **SR** (which was red), but now only goes through **S**, which has assumed the color of its former parent, and **S**’s right child, which has changed from red to black (assuming **S**’s color: black). The net effect is that this path goes through the same number of black nodes.

Either way, the number of black nodes on these paths does not change. Thus, we have restored Properties 4 (Both children of every red node are black) and 5 (All paths from any given node to its leaf nodes contain the same number of black nodes). The white node in the diagram can be either red or black, but must refer to the same color both before and after the transformation.

Again, the function calls all use tail recursion, so the algorithm is [in-place](https://en.wikipedia.org/wiki/In-place_algorithm).

In the algorithm above, all cases are chained in order, except in delete case 3 where it can recurse to case 1 back to the parent node: this is the only case where an iterative implementation will effectively loop. No more than loops back to case 1 will occur (where is the height of the tree). And because the probability for escalation decreases exponentially with each iteration the average removal cost is constant.

Additionally, no tail recursion ever occurs on a child node, so the tail recursion loop can only move from a child back to its successive ancestors. If a rotation occurs in case 2 (which is the only possibility of rotation within the loop of cases 1-3), then the parent of the node **N** becomes red after the rotation and we will exit the loop. Therefore, at most one rotation will occur within this loop. Since no more than two additional rotations will occur after exiting the loop, at most three rotations occur in total.

## Proof of asymptotic bounds

A red black tree which contains internal nodes has a height of .

Definitions:

* = height of subtree rooted at node
* = the number of black nodes from to any leaf in the subtree, not counting if it is black - called the black-height

**Lemma:** A subtree rooted at node has at least 2 b h ( v ) − 1 {\displaystyle 2^{bh(v)}-1} internal nodes.

Proof of Lemma (by induction height):

Basis:

If has a height of zero then it must be *null*, therefore. So: 2 b h ( v ) − 1 = 2 0 − 1 = 1 − 1 = 0 {\displaystyle 2^{bh(v)}-1=2^{0}-1=1-1=0}

Inductive Step: such that, has at least **2 b h ( v ) − 1 {\displaystyle 2^{bh(v)}-1}**  internal nodes implies that v ′ {\displaystyle v'} such that has at least 2 b h ( v ′ ) − 1 {\displaystyle 2^{bh(v')}-1} internal nodes.

Since v ′ {\displaystyle v'} has it is an internal node. As such it has two children each of which have a black-height of either or (depending on whether the child is red or black, respectively). By the inductive hypothesis each child has at least 2 b h ( v ′ ) − 1 − 1 {\displaystyle 2^{bh(v')-1}-1} internal nodes, so v ′ {\displaystyle v'} has at least: 2 b h ( v ′ ) − 1 − 1 + 2 b h ( v ′ ) − 1 − 1 + 1 = 2 b h ( v ′ ) − 1 {\displaystyle 2^{bh(v')-1}-1+2^{bh(v')-1}-1+1=2^{bh(v')}-1}

internal nodes.

Using this lemma, we can now show that the height of the tree is logarithmic. Since at least half of the nodes on any path from the root to a leaf are black (property 4 of a red–black tree), the black-height of the root is at least . By the lemma we get: n ≥ 2 h ( root ) 2 – 1 ↔ log 2 ⁡ ( n + 1 ) ≥ h ( root ) 2 ↔ h ( root ) ≤ 2 log 2 ⁡ ( n + 1 ) . {\displaystyle n\geq 2^{h({\text{root}}) \over 2}-1\leftrightarrow \;\log \_{2}{(n+1)}\geq {h({\text{root}}) \over 2}\leftrightarrow \;h({\text{root}})\leq 2\log \_{2}{(n+1)}.}

Therefore, the height of the root is .

# Theoretical analysis of B+ tree

## N-ary Tree

In graph theory, a **k-ary tree** is a rooted tree in which each node has no more than *k* children. It is also sometimes known as a **k-way tree**, an **N-ary tree**, or an **M-ary tree**. A binary tree is the special case where *k=2*.

### Types of k-ary trees

* A **full k-ary tree** is a k-ary tree where within each level every node has either *0* or *k* children.
* A **perfect k-ary tree** is a full k-ary tree in which all leaf nodes are at the same depth.
* A **complete k-ary tree** is a k-ary tree which is maximally space efficient. It must be completely filled on every level except for the last level. However, if the last level is not complete, then all nodes of the tree must be “as far left as possible”.

### Properties of k-ary trees

* For a k-ary tree with height , the upper bound for the maximum number of leaves is k h {\displaystyle k^{h}} .
* The height of a **k-ary tree** does not include the root node, with a tree containing only a root node having a height of 0.
* The height of a tree is equal to the maximum depth D of any node in the tree
* The total number of nodes N {\displaystyle N} in a **perfect** **k-ary tree** is ∑ i = 0 h k i = k h + 1 − 1 k − 1 {\displaystyle \sum \_{i=0}^{h}k^{i}={\frac {k^{h+1}-1}{k-1}}} , while the height is

k h + 1 – 1 k – 1 ≥ N > k h – 1 k – 1 {\displaystyle {\frac {k^{h+1}-1}{k-1}}\geq N>{\frac {k^{h}-1}{k-1}}}

* By the definition of Big-Ω, the maximum depthD = h >= ⌈ log k ⁡ ( ( k − 1 ) ⋅ N + 1 ) − 1 ⌉ = O ( l o g k ( n ) ) = O ( l o g ( n ) / l o g ( k ) ) {\displaystyle D=h>=\left\lceil \log \_{k}((k-1)\cdot N+1)-1\right\rceil =O(log\_{k}(n))=O(log(n)/log(k))}

## B-Tree

In computer science, a **B-tree** is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. Unlike self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in databases and file systems.

What, if anything, the *B* stands for has never been established.

### Overview of B-tree

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre- defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self- balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree contains a number of keys. The keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: *a*1 and *a*2. All values in the leftmost subtree will be less than *a*1, all values in the middle subtree will be between *a*1 and *a*2, and all values in the rightmost subtree will be greater than *a*2.

Usually, the number of keys is chosen to vary between and , where is the minimum number of keys, and is the minimum degree or branching factor of the tree. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has keys, then adding a key to that node can be accomplished by splitting the hypothetical key node into two key nodes and moving the key that would have been in the middle to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have keys, then a key may be deleted from the internal node by combining it with its neighbor. Deleting the key would make the internal node have keys; joining the neighbor would add keys plus one more key brought down from the neighbor’s parent. The result is an entirely full node of keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters or simply with the highest branching order,.

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

B-trees have substantial advantages over alternative implementations when the time to access the data of a node greatly exceeds the time spent processing that data, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the node data are in secondary storage such as disk drives. By maximizing the number of keys within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing of the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk block or an analogous size in secondary storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage need a lager number of child nodes to improve performance.

### Variants of B-tree

The term **B-tree** may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+ tree and the B\* tree.

* In the B+ tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access.
* The B\* tree balances more neighboring internal nodes to keep the internal nodes more densely packed. This variant ensures non-root nodes are at least full instead of . As the costliest part of operation of inserting the node in B-tree is splitting the node, \*B-trees are created to postpone splitting operation as long as they can. To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. This spill operation is less costly to do than split, because it requires only shifting the keys between existing nodes, not allocating memory for a new one. For inserting, first it is checked whether the node has some free space in it, and if so, the new key is just inserted in the node. However, if the node is full (it has keys, where is the order of the tree as maximum number of pointers to subtrees from one node), it needs to be checked whether the right sibling exists and has some free space. If the right sibling has keys, then keys are redistributed between the two sibling nodes as evenly as possible. For this purpose, keys from the current node plus the new key inserted, one key from the parent node and *j* keys from the sibling node are seen as an ordered array of keys. The array becomes split by half, so that lowest keys stay in the current node, the next (middle) key is inserted in the parent and the rest go to the right sibling. (The newly inserted key might end up in any of the three places.) The situation when right sibling is full, and left isn’t is analogous. When both the sibling nodes are full, then the two nodes (current node and a sibling) are split into three and one more key is shifted up the tree, to the parent node. If the parent is full, then spill/split operation propagates towards the root node. Deleting nodes is somewhat more complex than inserting however.
* B-trees can be turned into order statistic trees to allow rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operation.

### B-tree usage in database

* **Time to search a sorted file**

Usually, sorting and searching algorithms have been characterized by the number of comparison operations that must be performed using order notation. A binary search of a sorted table with records, for example, can be done in roughly comparisons. If the table had 1,000,000 records, then a specific record could be located with at most 20 comparisons: .

Large databases have historically been kept on disk drives. The time to read a record on a disk drive far exceeds the time needed to compare keys once the record is available. The time to read a record from a disk drive involves a seek time and a rotational delay. The seek time may be 0 to 20 or more milliseconds, and the rotational delay averages about half the rotation period. For a 7200 RPM drive, the rotation period is 8.33 milliseconds. For a drive such as the Seagate ST3500320NS, the track-to-track seek time is 0.8 milliseconds and the average reading seek time is 8.5 milliseconds. For simplicity, assume reading from disk takes about 10 milliseconds. Naively, then, the time to locate one record out of a million would take 20 disk reads times 10 milliseconds per disk read, which is 0.2 seconds.

The time won’t be that bad because individual records are grouped together in a disk block. A disk block might be 16 kilobytes. If each record is 160 bytes, then 100 records could be stored in each block. The disk read time above was actually for an entire block. Once the disk head is in position, one or more disk blocks can be read with little delay. With 100 records per block, the last 6 or so comparisons don’t need to do any disk reads—the comparisons are all within the last disk block read.

To speed the search further, the first 13 to 14 comparisons (which each required a disk access) must be sped up.

* **An index speeds the search**

A significant improvement can be made with an index. In the example above, initial disk reads narrowed the search range by a factor of two. That can be improved substantially by creating an auxiliary index that contains the first record in each disk block (sometimes called a sparse index). This auxiliary index would be 1% of the size of the original database, but it can be searched more quickly. Finding an entry in the auxiliary index would tell us which block to search in the main database; after searching the auxiliary index, we would have to search only that one block of the main database—at a cost of one more disk read. The index would hold 10,000 entries, so it would take at most 14 comparisons. Like the main database, the last 6 or so comparisons in the aux index would be on the same disk block. The index could be searched in about 8 disk reads, and the desired record could be accessed in 9 disk reads.

The trick of creating an auxiliary index can be repeated to make an auxiliary index to the auxiliary index. That would make an aux-aux index that would need only 100 entries and would fit in one disk block.

Instead of reading 14 disk blocks to find the desired record, we only need to read 3 blocks. Reading and searching the first (and only) block of the aux-aux index identifies the relevant block in aux-index. Reading and searching that aux-index block identifies the relevant block in the main database. Instead of 150 milliseconds, we need only 30 milliseconds to get the record.

The auxiliary indices have turned the search problem from a binary search requiring roughly disk reads to one requiring only disk reads where is the blocking factor (the number of entries per block: entries per block in our example; reads).

In practice, if the main database is being frequently searched, the aux-aux index and much of the aux index may reside in a disk cache, so they would not incur a disk read.

* **Insertions and deletions**

If the database does not change, then compiling the index is simple to do, and the index need never be changed. If there are changes, then managing the database and its index becomes more complicated.

Deleting records from a database is relatively easy. The index can stay the same, and the record can just be marked as deleted. The database remains in sorted order. If there are a large number of deletions, then searching and storage become less efficient.

Insertions can be very slow in a sorted sequential file because room for the inserted record must be made. Inserting a record before the first record requires shifting all of the records down one. Such an operation is just too expensive to be practical. One solution is to leave some spaces. Instead of densely packing all the records in a block, the block can have some free space to allow for subsequent insertions. Those spaces would be marked as if they were "deleted" records.

Both insertions and deletions are fast as long as space is available on a block. If an insertion won’t fit on the block, then some free space on some nearby block must be found and the auxiliary indices adjusted. The hope is that enough space is available nearby, such that a lot of blocks do not need to be reorganized. Alternatively, some out-of-sequence disk blocks may be used.

### Advantages of B-tree usage for databases

The B-tree uses all of the ideas described above. In particular, a B-tree:

* keeps keys in sorted order for sequential traversing
* uses a hierarchical index to minimize the number of disk reads
* uses partially full blocks to speed insertions and deletions
* keeps the index balanced with a recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

### Technical description

* **Terminology**

The literature on B-trees is not uniform in its terminology.

Bayer & McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk & Zoellick (1992) points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys. Knuth (1998, p. 483) avoids the problem by defining the **order** to be maximum number of children (which is one more than the maximum number of keys).

The term **leaf** is also inconsistent. Bayer & McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth considered the leaf level to be one level below the lowest keys (Folk & Zoellick 1992, p. 363). There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record. Those choices are not fundamental to the idea of a B-tree.[[5]](https://en.wikipedia.org/wiki/B-tree#cite_note-5)

There are also unfortunate choices like using the variable *k* to represent the number of children when *k* could be confused with the number of keys.

For simplicity, most authors assume there are a fixed number of keys that fit in a node. The basic assumption is the key size is fixed and the node size is fixed. In practice, variable length keys may be employed (Folk & Zoellick 1992, p. 379).

* **Definition**

According to Knuth’s definition, a B-tree of order *m* is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every non-leaf node (except root) has at least ⌈*m*/2⌉ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with *k* children contains *k* − 1 keys.
5. All leaves appear in the same level.
6. Each internal node’s keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: *a*1 and *a*2. All values in the leftmost subtree will be less than *a*1, all values in the middle subtree will be between *a*1 and *a*2, and all values in the rightmost subtree will be greater than *a*2.

**Internal nodes**

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and a **minimum** of *L* children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between *L*−1 and *U*−1). *U* must be either 2*L* or 2*L*−1; therefore, each internal node is at least half full. The relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there’s room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

**The root node**

The root node’s number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than *L*−1 elements in the entire tree, the root will be the only node in the tree with no children at all.

**Leaf nodes**

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

A B-tree of depth *n*+1 can hold about *U* times as many items as a B-tree of depth *n*, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, the B-trees benefit from improved performance if they use a specialized structure.

### Best case and worst-case heights

Let *h* be the height of the classic B-tree. Let *n* > 0 be the number of entries in the tree. Let be the maximum number of children a node can have. Each node can have at most keys.

It can be shown (by induction for example) that a B-tree of height *h* with all its nodes completely filled has entries. Hence, the best-case height of a B-tree is:

Let be the minimum number of children an internal (non-root) node can have . For an ordinary B-tree, .

Comer (1979, p. 127) and Cormen et al. (2001, pp. 383–384) give the worst-case height of a B-tree (where the root node is considered to have height 0) as

## Algorithms of B-Tree

### Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search reduces its field of view to the child pointer (subtree) whose range includes the search value. A subtree’s range is defined by the values, or keys, contained in its parent node. These limiting values are also known as separation values. Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

### Insertion

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum allowed number of elements, then there is room for the new element. Insert the new element in the node, keeping the node’s elements ordered.
2. Otherwise the node is full, evenly split it into two nodes so:
3. A single median is chosen from among the leaf’s elements and the new element.
4. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
5. The separation value is inserted in the node’s parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is *U*−1. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number *U*−1 of elements into two legal nodes. If this number is odd, then *U*=2*L* and one of the new nodes contains (*U*−2)/2 = *L*−1 elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If *U*−1 is even, then *U*=2*L*−1, so there are 2*L*−2 elements in the node. Half of this number is *L*−1, which is the minimum number of elements allowed per node.

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining *U*−2 elements into two legal nodes, without adding a new element. This requires *U* = 2*L* rather than *U* = 2*L*−1, which accounts for why some textbooks impose this requirement in defining B-trees.

### Deletion

There are two popular strategies for deletion from a B-tree.

* Locate and delete the item, then restructure the tree to retain its invariants, **OR**
* Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy. There are two special cases to consider when deleting an element:

* The element in an internal node is a separator for its child nodes
* Deleting an element may put its node under the minimum number of elements and children

The procedures for these cases are in order below.

* **Deletion from a leaf node**

1. Search for the value to delete.
2. If the value is in a leaf node, simply delete it from the node.
3. If underflow happens, rebalance the tree as described in section “Rebalancing after deletion” below.

* **Deletion from an internal node**

Each element in an internal node acts as a separation value for two subtrees, therefore we need to find a replacement for separation. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is still greater than the separator. Both of those elements are in leaf nodes, and either one can be the new separator for the two subtrees. Algorithmically described below:

1. Choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
2. The previous step deleted an element (the new separator) from a leaf node. If that leaf node is now deficient (has fewer than the required number of nodes), then rebalance the tree starting from the leaf node.

* **Rebalancing after deletion**

Rebalancing starts from a leaf and proceeds toward the root until the tree is balanced. If deleting an element from a node has brought it under the minimum size, then some elements must be redistributed to bring all nodes up to the minimum. Usually, the redistribution involves moving an element from a sibling node that has more than the minimum number of nodes. That redistribution operation is called a **rotation**. If no sibling can spare an element, then the deficient node must be **merged** with a sibling. The merge causes the parent to lose a separator element, so the parent may become deficient and need rebalancing. The merging and rebalancing may continue all the way to the root. Since the minimum element count doesn’t apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows:

* If the deficient node’s right sibling exists and has more than the minimum number of elements, then rotate left

1. Copy the separator from the parent to the end of the deficient node (the separator moves down; the deficient node now has the minimum number of elements)
2. Replace the separator in the parent with the first element of the right sibling (right sibling loses one node but still has at least the minimum number of elements)
3. The tree is now balanced

* Otherwise, if the deficient node’s left sibling exists and has more than the minimum number of elements, then rotate right

1. Copy the separator from the parent to the start of the deficient node (the separator moves down; deficient node now has the minimum number of elements)
2. Replace the separator in the parent with the last element of the left sibling (left sibling loses one node but still has at least the minimum number of elements)
3. The tree is now balanced

* Otherwise, if both immediate siblings have only the minimum number of elements, then merge with a sibling sandwiching their separator taken off from their parent

1. Copy the separator to the end of the left node (the left node may be the deficient node or it may be the sibling with the minimum number of elements)
2. Move all elements from the right node to the left node (the left node now has the maximum number of elements, and the right node – empty)
3. Remove the separator from the parent along with its empty right child (the parent loses an element)
   1. If the parent is the root and now has no elements, then free it and make the merged node the new root (tree becomes shallower)
   2. Otherwise, if the parent has fewer than the required number of elements, then rebalance the parent

**Note**: The rebalancing operations are different for B+ trees (e.g., rotation is different because parent has copy of the key) and B\*-tree (e.g., three siblings are merged into two siblings).

### Sequential access

While freshly loaded databases tend to have good sequential behavior, this behavior becomes increasingly difficult to maintain as a database grows, resulting in more random I/O and performance challenges.

### Initial construction

A common special case is adding a large amount of *pre-sorted* data into an initially empty B-tree. While it is quite possible to simply perform a series of successive inserts, inserting sorted data results in a tree composed almost entirely of half-full nodes. Instead, a special "bulk loading" algorithm can be used to produce a more efficient tree with a higher branching factor.

When the input is sorted, all insertions are at the rightmost edge of the tree, and in particular any time a node is split, we are guaranteed that the no more insertions will take place in the left half. When bulk loading, we take advantage of this, and instead of splitting overfull nodes evenly, split them as *unevenly* as possible: leave the left node completely full and create a right node with zero keys and one child (in violation of the usual B-tree rules).

At the end of bulk loading, the tree is composed almost entirely of completely full nodes; only the rightmost node on each level may be less than full. Because those nodes may also be less than *half* full, to re-establish the normal B-tree rules, combine such nodes with their (guaranteed full) left siblings and divide the keys to produce two nodes at least half full. The only node which lacks a full left sibling is the root, which is permitted to be less than half full.

## B+ Tree

A **B+ tree** is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

The ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories. NTFS uses B+ trees for directory and security-related metadata indexing. EXT4 uses extent trees (a modified B+ tree data structure) for file extent indexing. Relational database management systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, and SQLite support this type of tree for table indices. Key–value database management systems such as CouchDBand Tokyo Cabinet support this type of tree for data access.

The order, or branching factor, of a B+ tree measures the capacity of nodes (i.e., the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as , is constrained for internal nodes so that . The root is an exception: it is allowed to have as few as two children. For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least and at most . In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary and at most .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Node Type** | **Children Type** | **Min Number of Children** | **Max Number of Children** | **Example** | **Example** |
| Root Node (when it is the only node in the tree) | Records | 1 |  | 1–6 | 1–99 |
| Root Node | Internal Nodes or Leaf Nodes | 2 | *b* | 2–7 | 2–100 |
| Internal Node | Internal Nodes or Leaf Nodes |  | *b* | 4–7 | 50–100 |
| Leaf Node | Records |  |  | 4–7 | 50–100 |

## Algorithms in B+ Tree

### Search

The root of B+ Tree represents the whole range of values in the tree, where every internal node is a subinterval.

We are looking for a value k in the B+ Tree. Starting from the root, we are looking for the leaf which may contain the value k. At each node, we figure out which internal pointer we should follow. An internal B+ Tree node has at most children, where every one of them represents a different sub-interval. We select the corresponding node by searching on the key values of the node.

### Prefix key compression

It is important to increase fan-out, as this allows to direct searches to the leaf level more efficiently. Index Entries are only to ‘direct traffic’, thus we can compress them.

### Insertion

* Perform a search to determine what bucket the new record should go into.
* If the bucket is not full (at most entries after the insertion), add the record.
* Otherwise, *before* inserting the new record
  + split the bucket.
    - original node has items
    - new node has items
  + Move-th key to the parent, and insert the new node to the parent.
  + Repeat until a parent is found that need not split.
* If the root splits, treat it as if it has an empty parent and split as outline above.
* B-trees grow at the root and not at the leaves.

### Deletion

* Start at root, find leaf *L* where entry belongs.
* Remove the entry.
  + If *L* is at least half-full, done
  + If *L* has fewer entries than it should,
    - If sibling (adjacent node with same parent as *L*) is more than half-full, re-distribute, borrowing an entry from it.
    - Otherwise, sibling is exactly half-full, so we can merge *L* and sibling.
* If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
* Merge could propagate to root, decreasing height.

### Bulk-loading

Given a collection of data records, we want to create a B+ tree index on some key field. One approach is to insert each record into an empty tree. However, it is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. An efficient alternative is to use bulk-loading.

* The first step is to sort the data entries according to a search key in ascending order.
* We allocate an empty page to serve as the root, and insert a pointer to the first page of entries into it.
* When the root is full, we split the root, and create a new root page.
* Keep inserting entries to the right most index page just above the leaf level, until all entries are indexed.

Note :

* when the right-most index page above the leaf level fills up, it is split;
* this action may, in turn, cause a split of the right-most index page on step closer to the root;
* splits only occur on the right-most path from the root to the leaf level.

# Specification and details of the requirements

1. First insert into trees the data in the file 1\_initial.txt

2. Then delete the data in the file 2\_delete.txt

3. Add the data in the file 3\_insert.txt

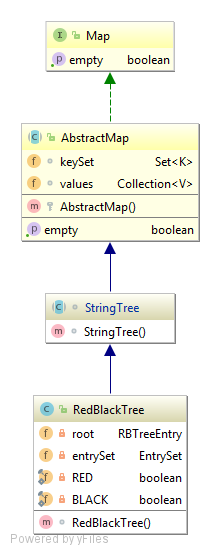
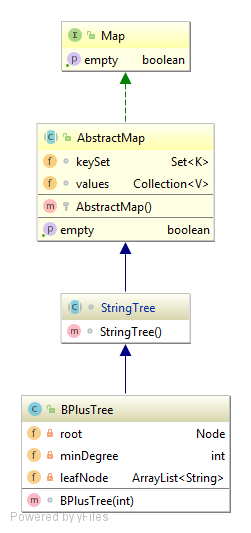
4. Query a word

5. Query some words

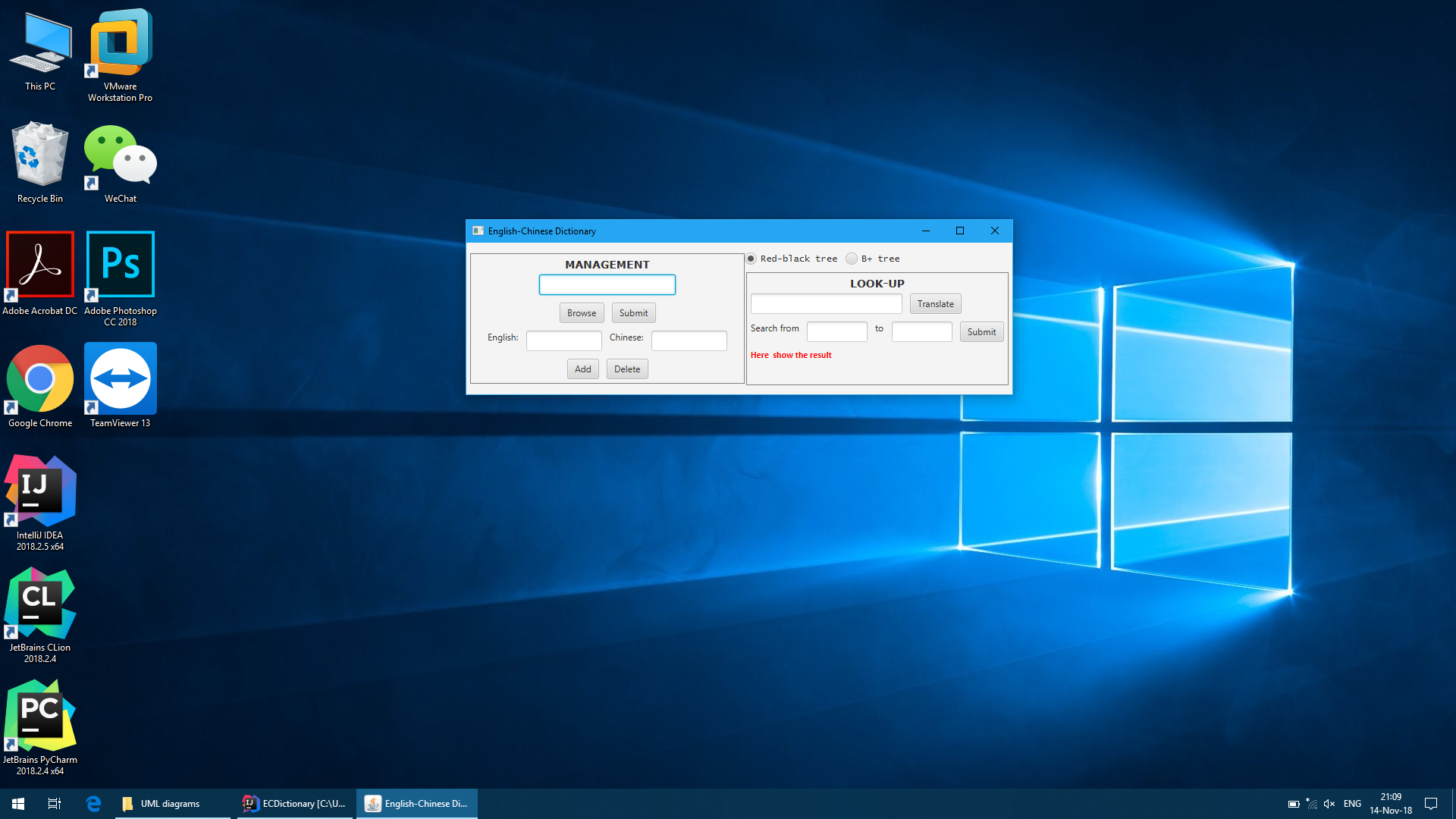
For the first three steps, after each operation on 100 pieces of data, you should record the time used. For step 4 and step 5, just give the time totally consumed for each query.

# Implementation structure

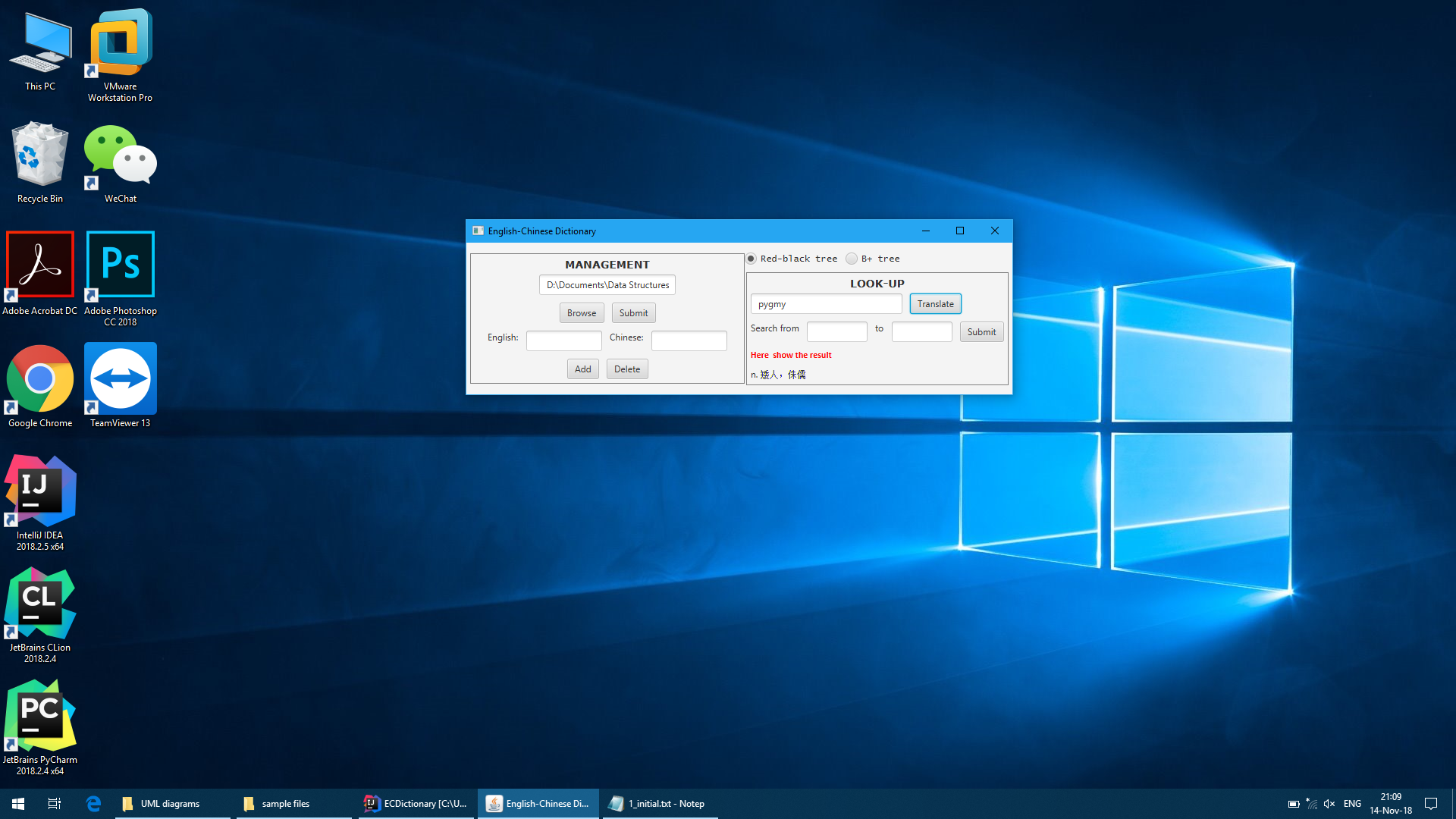
An abstract class Abstract Map is designed to better represent the data structure of the Map. The UML diagram of the two classes are shown as below, indicating the inheriting relationships between the parent and child classes. For more detailed UML diagrams including the inner classes and methods, please refer to the exported UML diagrams attached in the project folder.



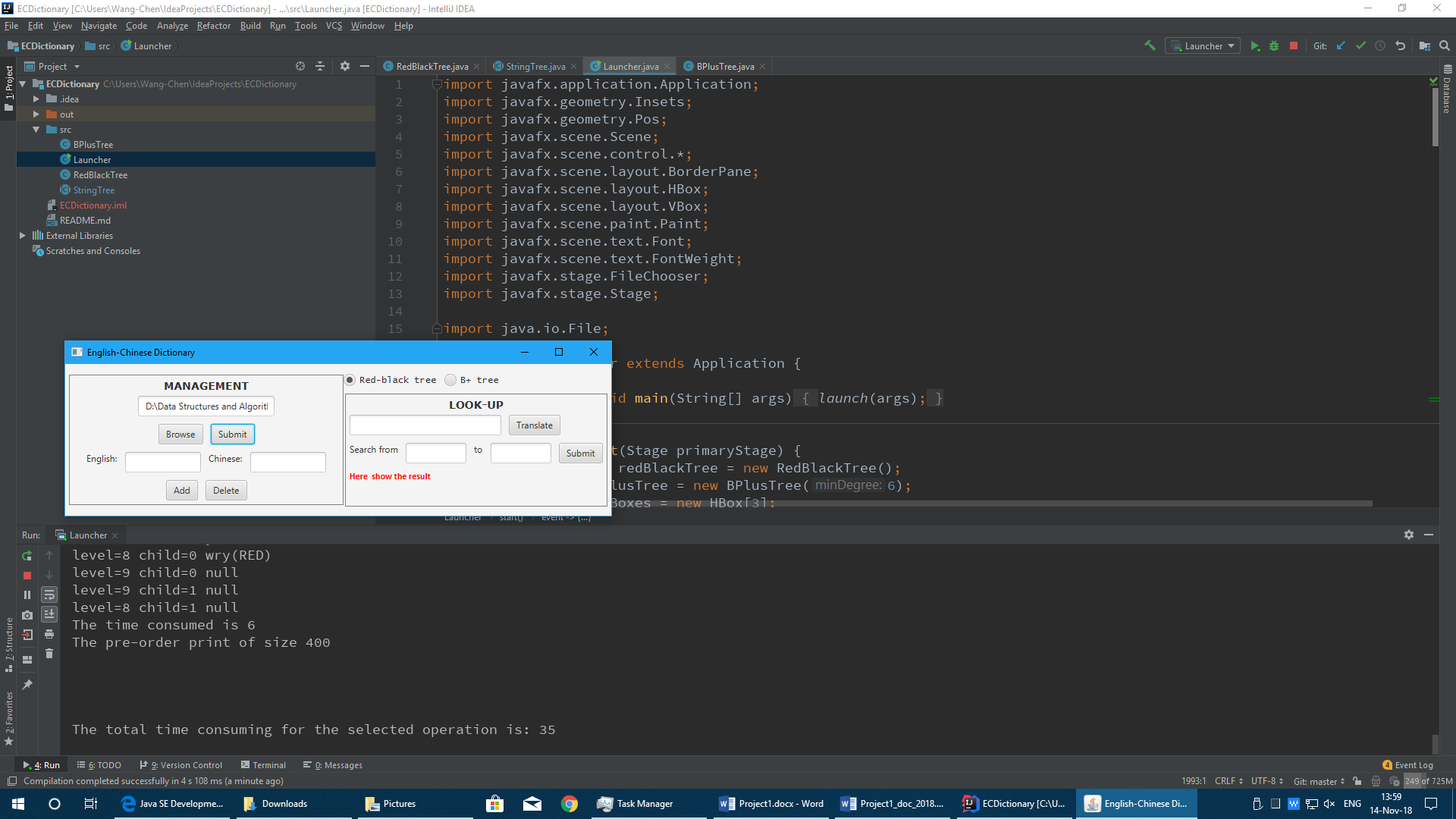
The GUI interface of my version of implementation is as follow, the functions are just as required.

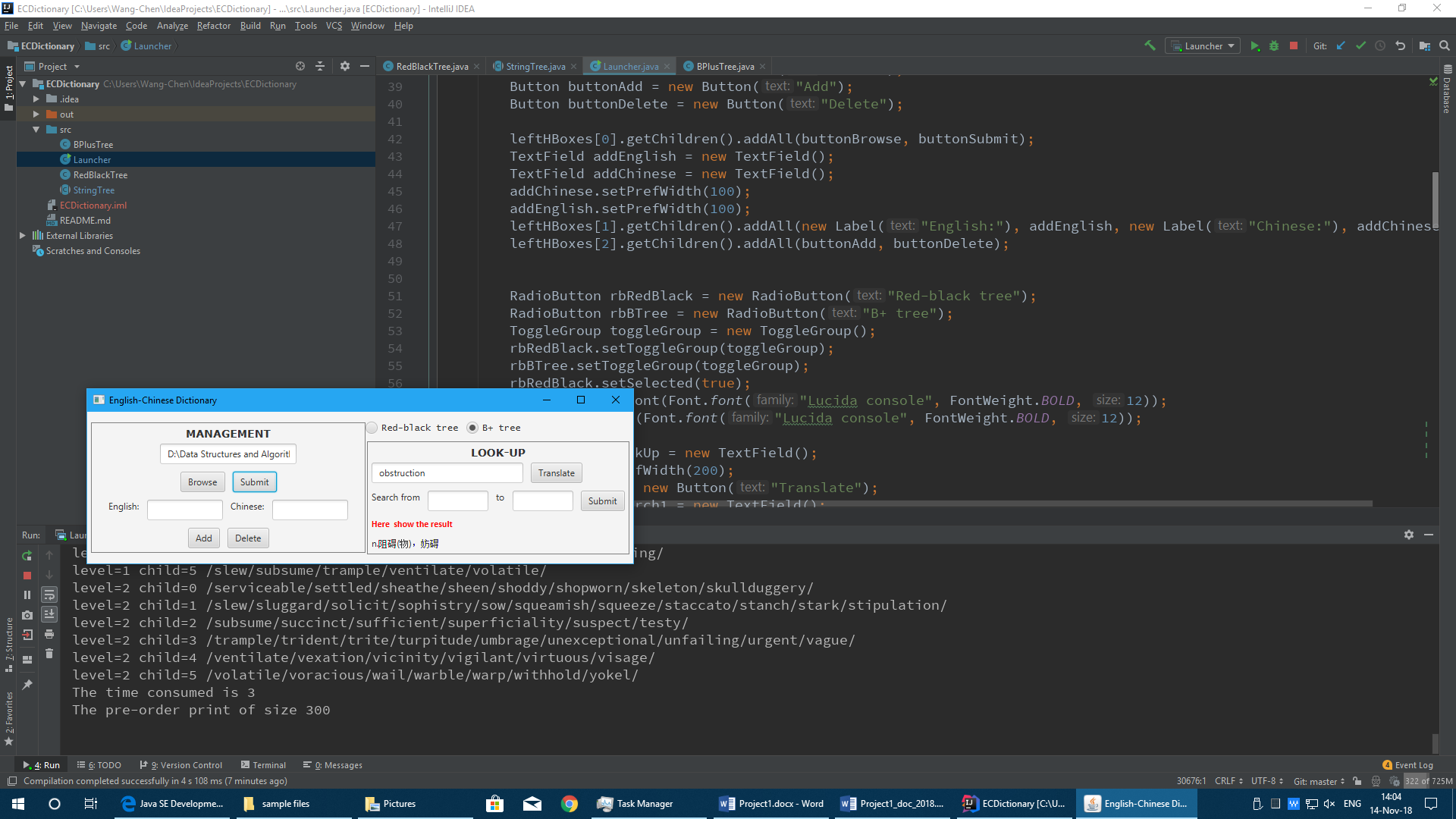


After a query, the interface displays as follow:



The time consumed for my implementation is that every 100 Red black tree insertion takes 6 milliseconds on average and every 100 B tree insertion takes 3 milliseconds on average. The screenshots are shown as below.





Based on the theoretical analysis above, the time consumption of the two trees is reasonable. They can well represent the time to be consumed by the program. Because the multi-process operating system allows multiple processes running at the same time, the timing might be unable to totally represent the performance of the program. The exact analysis of the performance of the program should be followed as the analysis shown in the previous sections.