B+ Trees

Step 1

As we have now seen, the **B-Tree** is an exceptionally optimized way to store and read data on the Hard Disk. Consequently, when saving a large dataset on the Hard Disk, it is only natural to want to use the **B-Tree** structure to take advantage of all of its perks (i.e., it keeps our data sorted and lets us query fairly fast in practice).

A common way to store a large dataset is to use a relational database. If we wanted to store a relational database using the **B-Tree** structure, we would store each data *entry* (commonly stored as a row in a database, with multiple data fields stored as columns) as a single key. However, there are a couple of aspects to working with databases that make the **B-Tree** structure not so ideal for this purpose:

1. More often than not, database entries are actually really big (each row can have tens of columns). Also, we often only want to query a single database entry at a time. Consequently, in the worst case, if we wanted to find *one* database entry in a **B-Tree**, we would need to traverse O($log_b n$) database entries to reach it. Since our databases can consist of *millions* of large rows, traversing O($log_b n$) entries can be a *huge* waste of time to only look up *one* database entry!
2. Another common task typically performed on databases is to query all the entries at once (e.g. to output everything currently being stored). If we wanted to query all the keys in a **B-Tree**, we would have to traverse *up* and *down all* the nodes in all levels of the tree (i.e. execute a pre-order traversal), thereby having to figure out exactly *where* the pointer to the node is located in memory (L1 cache, Main Memory, Hard Disk...) each time we go both *down* and *up* the tree. Potentially having to figure out where a node is located *twice* thus adds a large time overhead for an operation that definitely shouldn't need it.

Step 2

How do we take into consideration the concerns described in the previous step to transform the **B-Tree** structure into a more useful structure to store large data sets?

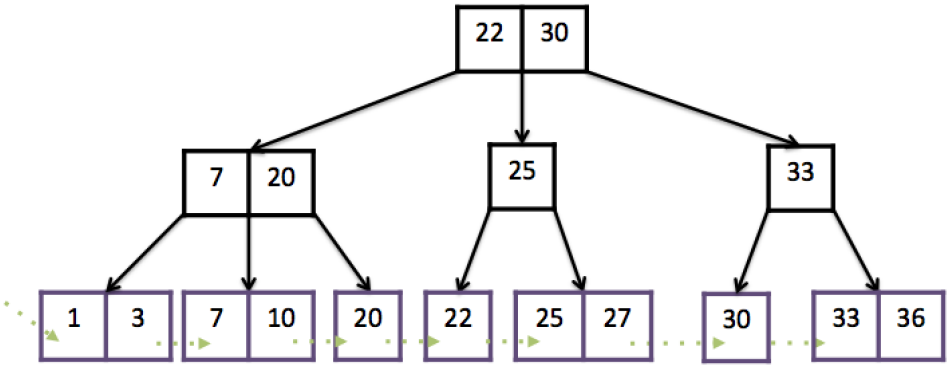
1. If we are worried about having to traverse nodes filled with other database entries before reaching the *single* database entry we queried, then why don't we store the actual full database entries *only* at the leaves? In order to still be able to traverse the tree, however, our keys above the leaves would be kept for storing *only* the single field we used to *sort* the data records. That way, we get to traverse *smaller* nodes to reach the query. By having less data to traverse, we would increase our chances that the smaller key nodes are stored *closer to* ***each other*** in memory, thereby making it more likely that they are in a *closer* ***section of memory***, such as the L1 Cache (remember, this is because the CPU determines which data to put in closer sections of memory largely based on ***spatial*** locality). By having the keys be in a closer section of memory, it will thus take less time to access the queried database entry.
2. Now that we have all the full data records guaranteed to be at the leaves, let's take advantage of this situation to create an easier way to read all the data entries *without* having to traverse up and down the tree; let's link together all the leaf nodes (i.e., implement the leaf nodes as a Linked List)! That way, we can just do a linear scan of our leaf nodes to read *all* of our database entries.

The implementation of the two ideas above in the **B-Tree** structure produces a modified structure called the **B+ Tree**. In the next step, we will see how a **B+ Tree** actually looks.

Step 3

Below is an example of **B+ Tree** with the modifications we discussed in the previous step. Note that the internal nodes outlined in **black** are filled with *keys* (i.e., the integers we use to sort the tree), and the nodes outlined in **purple** are the leaves filled with *all* the data records. Consequently, you will notice that some integers are repeated twice within the **B+ Tree** (e.g. there are two instances of the key 7: one is a *key* and another is a data record).

Also, note that we are using integer values to sort the tree *and* we are storing the integer values themselves. As a result, the *keys* are equal to the *data records*. However, if we were inserting large database entries, then the data records in the **purple** boxes would contain more information other than just the integer value.



STOP and Think: Can you see how the **purple** data records are sorted in the **B+ Tree** above?

Note: Hopefully you have noticed that the **B+ Tree** that we introduced *technically* violates one of the fundamental properties of a **tree**: the green Linked List arrows cause undirected cycles to appear in the "**tree**." As a result, this presents a bit of a conundrum because, on the one hand, the **B+ Tree** that is used *in practice* to *actually* implement databases really does use the Linked List structure at the bottom. On the other hand, *in theory*, this is no longer a **tree** and is simply a **graph** (a **directed acyclic graph**, to be exact). What are we going to do about this? Well, since we have agreed to talk about **B+ *Trees,*** for this lesson and for the purpose of this text, we are going to omit the Linked List structure at the leaves in order to keep a **B+ Tree** a valid **tree**. Do keep in mind, however, that in practice, it is common for people to still refer to using a "**B+ Tree**" structure to implement their database *with* the use of the Linked List leaf structure.

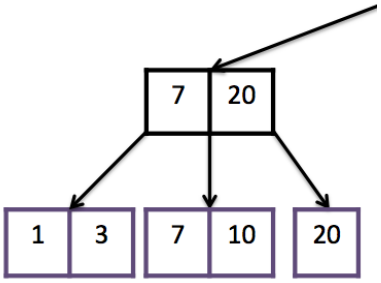
Step 4

Formally, we define a **B+ Tree** by the values *M* and *L*, where *M* is equal to the maximum number of children a given node can have, and *L* is equal to the maximum number of data records stored in a leaf node.

A **B+ Tree** of order *M* is a **tree** that satisfies the properties below:

1. Every node has at most *M* children.
2. Every internal node (except the root) has at least $\lceil M/2 \rceil$ children.
3. The root has at least two children if it is not a leaf.
4. An internal node with *k* children contains $k-1$ keys.
5. All leaves appear on the same level of the tree.
6. Internal nodes contain only search keys (i.e., no data records).
7. The smallest data record between search keys *x* and *y* equals *x*.

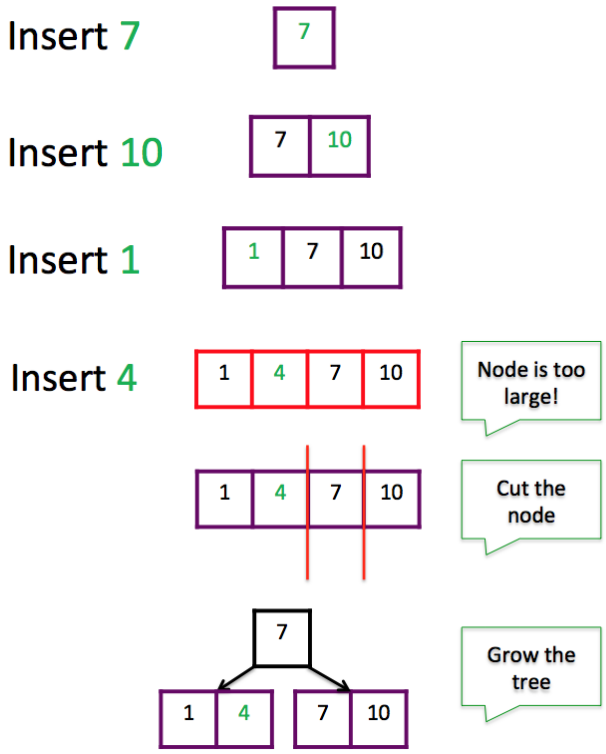
Note that the major difference between a **B+ Tree** and a **B-Tree** is expressed in properties 6 and 7 (the first 5 properties are shared between both). Also, note how property 7 is expressed in the subtree below, which was taken from the **B+ Tree** in the previous step. For example, the *data record* 7 is stored between the *keys* **7** and 20. The *data record* 20 is stored between the *keys* **20** and 'NULL.'



Step 5

How does the insert operation of a **B+ Tree** work? It is actually algorithmically quite similar to the **B-Tree** insert operation (unsurprisingly). Just a reminder: within the **B-Tree** insert operation, we always inserted keys into the leaves. Once a leaf node would overflow, we would cut the overflowing node and grow the tree *upward*. The same idea applies to a **B+ Tree**.

Let's look at an example of a **B+ Tree** in which $M = 3$ and $L = 3$ (i.e., the leaves can have a maximum of 3 data records and the internal nodes can have a maximum of 2 keys and 3 child pointers):



Note the two main differences between the **B+ Tree** insertion that happened above and the **B-Tree** insertion we studied previously:

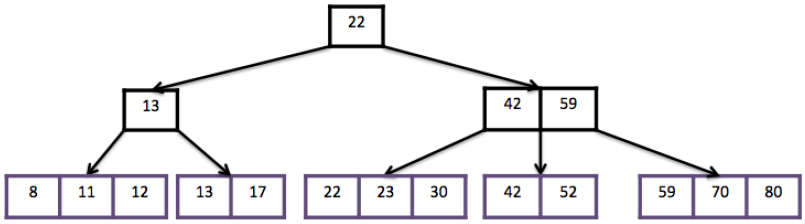
1. The node was cut around the data record 7: the *smallest* data record of the *second* half of the node. This differs from what happened in the **B-Tree** insertion, in which we cut around the data record 4: the *largest* data record of the *first* half of the node.
2. We initially inserted elements into data record nodes (the **purple** nodes). Consequently, when we grew the tree, none of our data records moved up a level. Instead, we created a *key* node (the **black** node) containing the key 7 and used it to grow the tree upward. As a result, we see two instances of the integer 7 (one is the *key* and the other is the actual *data record*).

STOP and Think: Why did we choose to cut around the the *smallest* data record of the *second* half of the node? **Hint:** Look back at properties of a **B+ Tree** on the previous step.

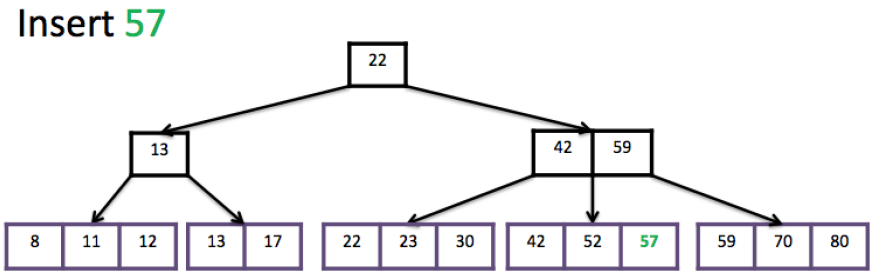
Fun Fact: A **B+ Tree** with *M = 3 and L = 3* is commonly referred to as a "**2-3 Tree**." This is because, as mentioned above, the internal nodes can have a maximum of **2** keys and the leaves can have a maximum of **3** data records.

Step 6

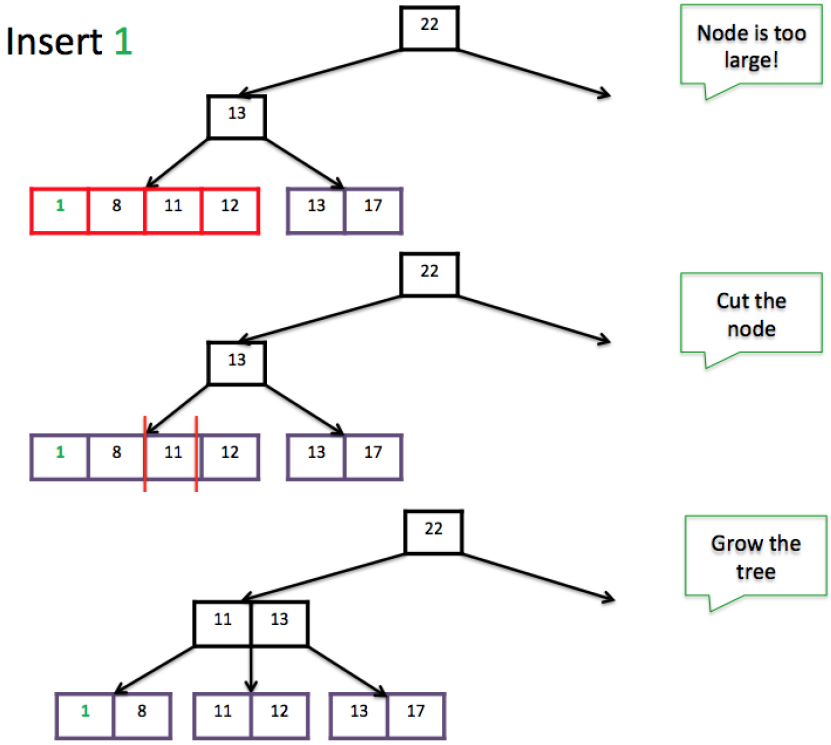
Since we have already practiced using a very similar **B-Tree** insert operation with a good number of fairly simple trees in the previous lesson, let's skip ahead and try a slightly more complicated **B+ Tree** structure, in which $M = 3$ and $L = 3$:



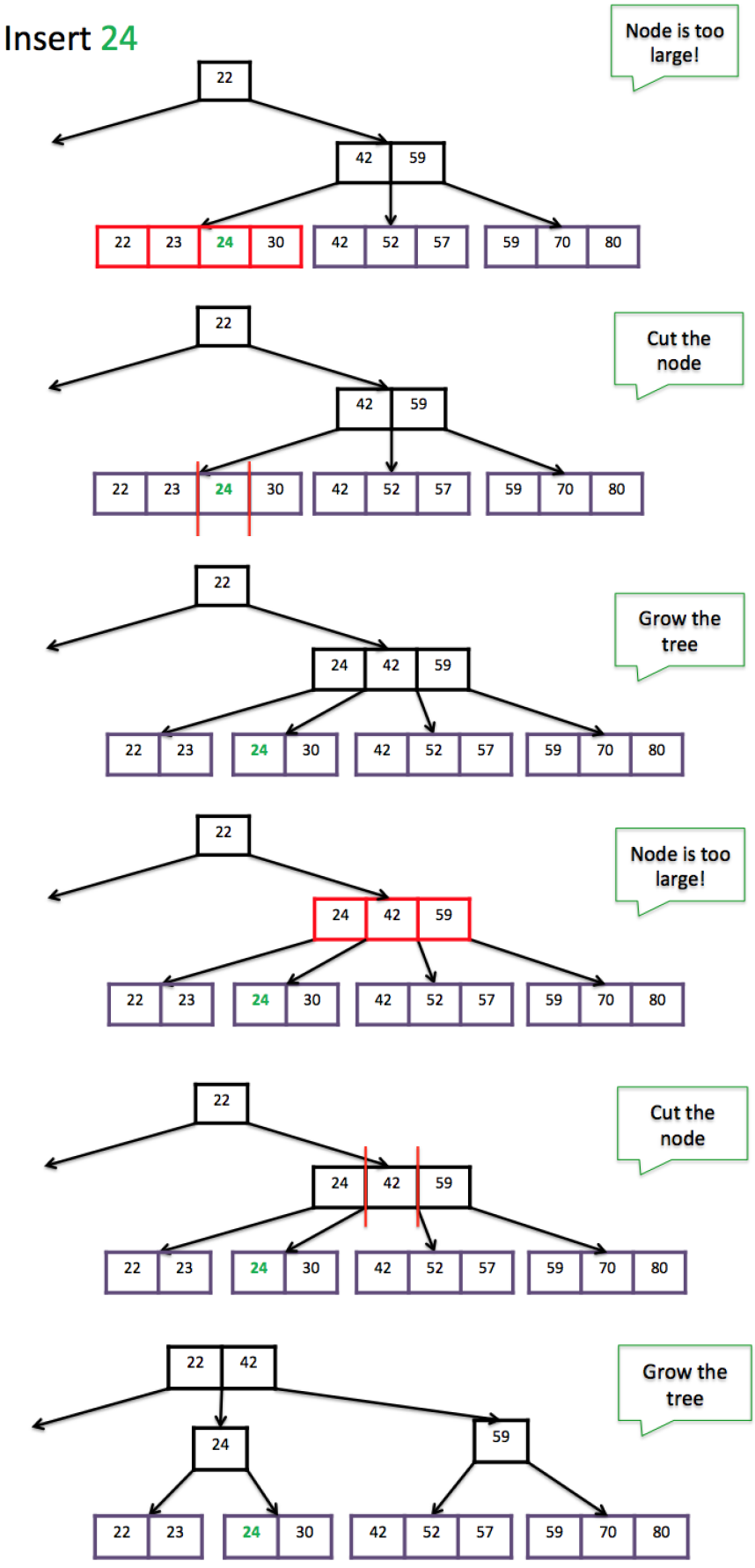
What happens when we attempt to insert the data record 57?



Note how no adjustments need to be made (because no *data record* nodes nor *key* nodes face an overflow) and data record 57 can be safely inserted. Does the same thing happen if we attempt to insert data record 1 into the root's left subtree?

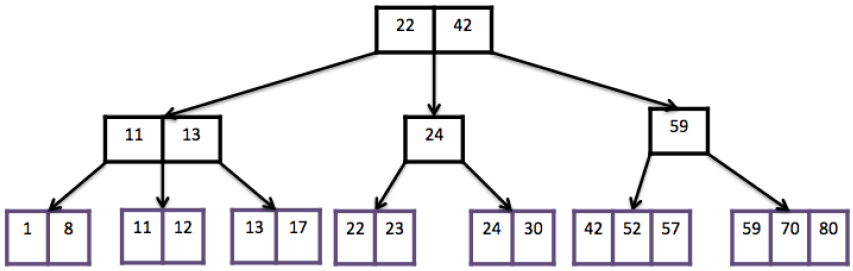


Because the inserting leaf node faced an overflow of data records, we had to resort to growing a new key node. What happens when we attempt to insert data record 24 into the root's right subtree?



Note: When growing upward directly from the leaves, we grow a *key* (i.e., we do not move the actual data record up a level)! For example, when "cutting" data record 24, a duplicate 24 appeared in the tree (i.e., the key). However, when growing upward from an internal node (i.e., a key), we do actually cut the key itself and move it up a level (i.e., a duplicate does not appear). For example, when "cutting" key 42, we actually moved the key itself up to the root.

Thus, we end up with this monstrosity below. Pay attention to how property 7—"Smallest data record between search keys *x* and *y* equals *x*"—still holds after all the insertions we have made:



Step 7

Below is pseudocode more formally describing the **B+ Tree** insertion algorithm. Note that the real code to implement insert is usually *very* long because of the shuffling of keys and pointer manipulations that need to take place.

```
insert(<key,dataRecord>): // return true upon success

    // check for duplicate insertion (not allowed)
    if dataRecord is already in tree:
        return false

    node = the leaf node into which dataRecord must be inserted
    insert <key,dataRecord> into node (maintain sorted order) and allocate space for children

    // while the node is too large, grow the tree
    if node.size > L:
        left,right = result of splitting node in half (if odd number, make right node take extra
dataRecord)
        parent = parent of node
        duplicate smallest key from right and insert into parent (maintain sorted order)
        make left and right new children of parent
        node = parent

        while node.size > M-1:
            left,right = result of splitting node in half (if odd number, make right node take
extra key)
            parent = parent of node
            remove smallest key from right and insert into parent (maintain sorted order)
            make left and right new children of parent
            node = parent

    return true
```

Note: We pass in a (*key*, *data record*) pair into the insert method in order to provide our **B+ Tree** *both* a *means of sorting* the data record (the key) *and* the *data record itself*.

Note: Duplicates of a data record are not allowed. *However*, different data records *can* share the same key. If different data records happen to share the same key, we can create a Linked List at the location of the previous key and append the new data record to that Linked List. Do not worry if this doesn't make much sense to you; we will explore this concept in more depth when we discuss Separate Chaining in the Hashing chapter. Consequently, for our current purposes, we will assume that all keys inserted into a **B+ Tree** are unique.

What is the worst-case time complexity for a **B+ Tree** insert operation?

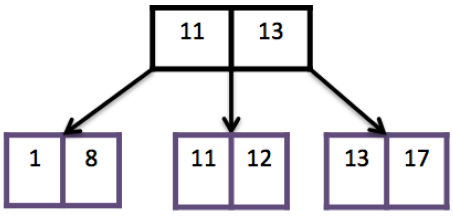
Hopefully it is intuitive to you that the **B+ Tree** insertion algorithm has the same worst-case time complexity as a **B-Tree** insert operation because, in both cases, we need to potentially traverse the entire height of the tree and reshuffle data records/keys at each level to maintain the sorted property. However, because a **B+ Tree** is expressed in terms of the variables *M* and *L* (*not b*), the worst-case

time complexity for a **B+ Tree** insertion becomes $O(M \log_M n + L)$. Why? In the worst case, we need to traverse over each of the *M* keys in an internal node, each of the *L* data records in a leaf node, and each of the $O(log_M n)$ levels of the tree.

The worst-case time complexity *with regards to memory accesses* is also $O(M \log_M n + L)$.

Step 8

EXERCISE BREAK: What is the value of the ***new key*** node that will be generated after inserting the data record 9 into the **B+ Tree** below, in which *M* = 3 and *L* = 2 (i.e., the leaves can have a maximum of 2 data records and the internal nodes can have a maximum of 2 keys/3 child pointers)?

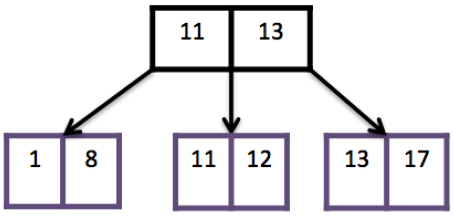


To solve this problem please visit https://stepik.org/lesson/31576/step/8

Step 9

EXERCISE BREAK: What is the value of the ***new root*** node after inserting the data record 9 into the **B+ Tree** below, in which *M* = 3 and *L* = 2?

Note: This is the same **B+-Tree** as in the previous step (we are not trying to trick you).



To solve this problem please visit https://stepik.org/lesson/31576/step/9

Step 10

It should also hopefully be intuitive that finding a key in a **B+ Tree** is practically identical to a **B-Tree**'s find algorithm! Below is pseudocode describing the find algorithm, and we would call this recursive function by passing in the root of the **B+ Tree**:

```
find(<key,dataRecord>, node): // return true upon success

    if node is empty:
        return false

    if node is leaf and dataRecord is in node: // find via binary search
        return true

    // if key is smaller than smallest element in node, go to left child
    if key is smaller than node[0]:

        if node[0] has leftChild:
            return(<key,dataRecord>, node[0].leftChild)

        //else there is no way the dataRecord can be in the tree
        else:
            return false

    nextNode = largest element of node that is smaller than or equal to the key // find via binary
search

    if nextNode has rightChild:
        return(<key,dataRecord>, node[0].rightChild)

    //else there is no way the dataRecord can be in the tree
    else:
        return false
```

STOP and Think: Why do we pass in *both* the key *and* the data record as a pair? In other words, why couldn't we just search for the data record by itself?

What is the worst-case time complexity of the find operation of a **B+ Tree**? Considering that the algorithm above is practically identical to the **B-Tree** find algorithm, it should hopefully be intuitive that the time complexity will be very similar. In the worst case, we have to traverse the entire height of the balanced tree, which is $O(\log_M n)$. Within each internal *key* node, we can do a binary search to find the element, which is $O(\log_2 M)$. Within each *leaf* node, we can do a binary search to find the element, which is $O(\log_2 L)$. Thus, the worst-case time complexity to find an element simplifies to $O(\log_2 n + \log_2 L)$.

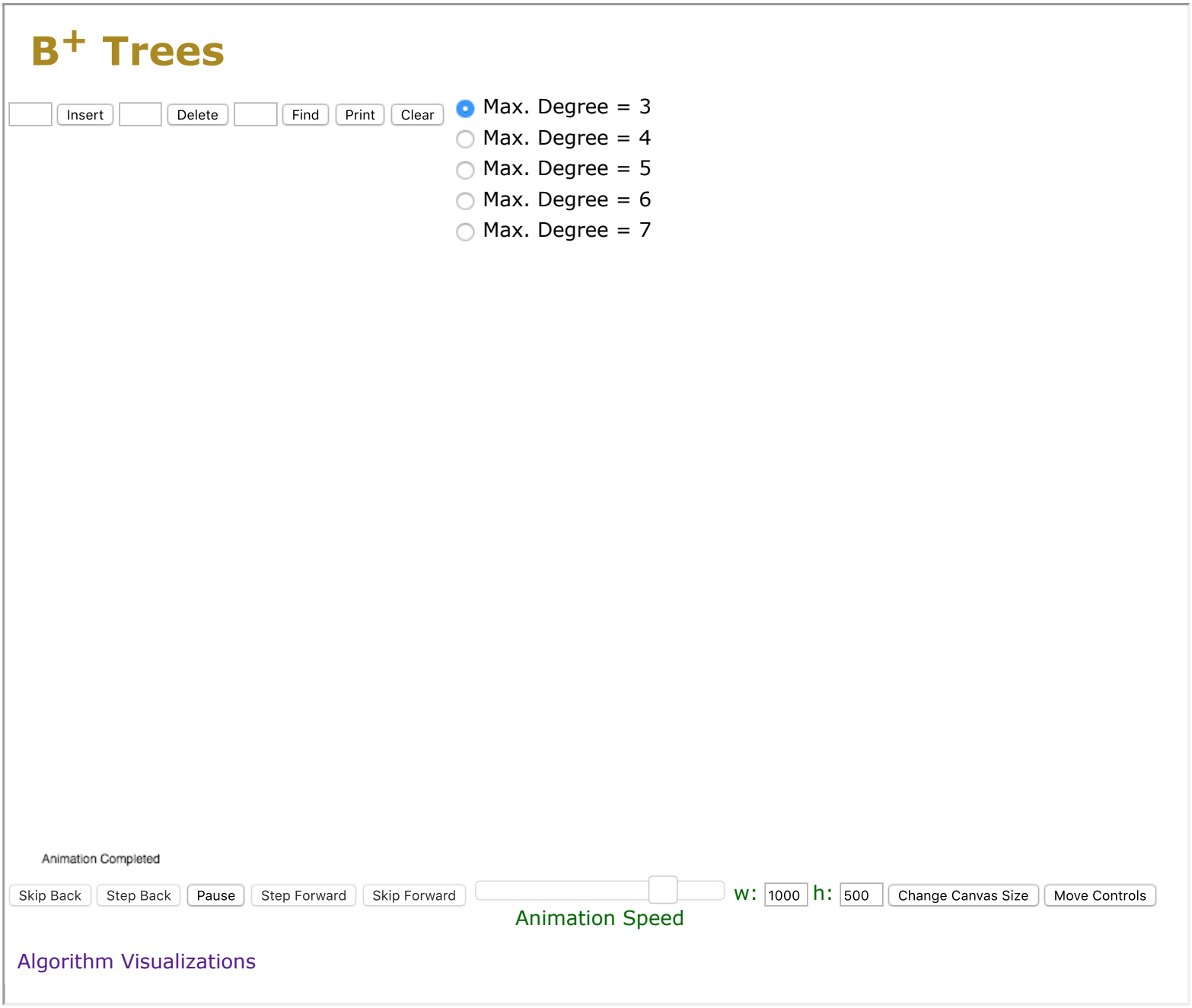
However, what is the worst-case time complexity of the find operation *with respect to memory access*? At each level of the tree *except* the leaves, we need to read in the entire node of size O(*M*). At the level of the leaves, we need to read in the entire node of size O(*L*). Consequently, the worst-case time to find an element *with respect to memory access* is $O(M \log_M n + L)$.

Step 11

As you have probably guessed by now, deleting a data record in a **B+ Tree** is *also* very similar to deleting a key in a **B-Tree**. The major difference is that, because data records are *only* stored in the leaves, we will always be deleting from the leaves. However, problems start to arise when we need to deal with an existing key in the **B+ Tree** that all of a sudden may or may not have its respective data record in the tree (depending on whether or not it was deleted). Consequently, a lot of merging of leaf nodes and key redistributions begin to happen. Therefore, we will omit formal discussion of the deletion algorithm for a **B+ Tree** in this text. Nonetheless, we recommend playing with the visualization below, created by David Galles at the University of San Francisco, in order to see how deleting different data records can result in different techniques to handle rebalancing the **B+ Tree**.

It is important to note, however, that the worst-case time complexity of the delete operation for a **B+ Tree** turns out to be the same as the insert operation, $O(M \log_M n + L)$. Why? In the worst case, we need to traverse over all *M* keys in an internal node, all *L* data records in a leaf node, and all $O(\log_M n)$ levels of the tree, similarly to insert.

Note: The "Max Degree" (maximum number of children) option is equal to *M*.



Step 12

EXERCISE BREAK: Which of the following statements regarding **B+ Trees** are true? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/31576/step/12

Step 13

As we have now seen, the **B+ Tree** is actually very similar to the **B-Tree**. Both are self-balancing trees that have logarithmic insertion, find, and delete operations. However, because of the two changes that we made to the **B-Tree**—storing data records *only* at the leaves and implementing the leaves as a Linked List (something that we admittedly ignored for the sake of the keeping the **B+ Tree** a valid tree—the **B+ Tree** *actually* becomes extremely useful to store large data sets. It is actually so useful, that it *really is* used to implement common database languages and file systems!

For example, relational database systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, and SQLite support the use of a **B+ Tree** to efficiently index into data entries. Also, your very own computer's operating system is likely using **B+ Trees** to organize data on your hard drive! NTFS, the file system used by all modern Windows operating systems, uses **B+ Trees** for directory indexing, and ext4, a file system very commonly used in a wide variety of Linux distributions, uses **extent trees** (a modified version of the **B+ Tree**) for file extent indexing.

This concludes our discussion of tree structures, but as you may have noticed, in this section, we had already started to deviate from the traditional definition of a "tree" by introducing the Linked List structure at the leaves of a **B+ Tree** that violate the formal "tree" definition. In the next chapter, we will finally expand our vision and generalize the notion of "nodes" and "edges" without the restrictions we placed on our data structures in this chapter. Specifically, in the next chapter, we will discuss **graphs**.