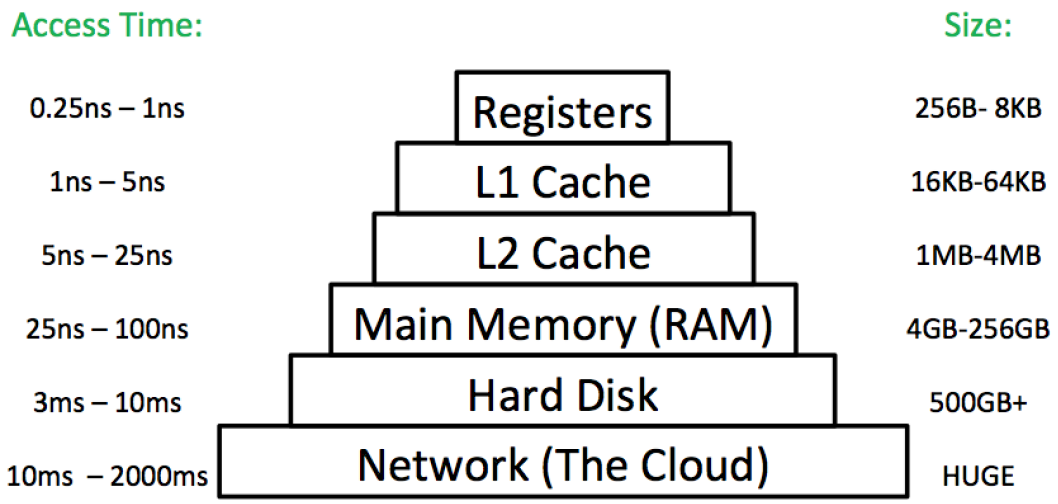
B-Trees

Step 1

As we have been introducing different tree structures and analyzing their implementations, we have actually been making an assumption that you probably overlooked as being negligible: we have been assuming that all node accesses in memory take the same amount of time. In other words, as we traverse an arbitrary tree structure and visit a node, we assume that the time it takes to "visit" a node is the same for every node. In practice, is this assumption accurate?

It turns out that, unfortunately, this assumption can't *possibly* be accurate based on the way memory is organized in our computers. As you might already know (and if you don't, we'll bring you up-to-speed), memory in a computer is based on a hierarchical system as shown below:



The basic idea behind the hierarchy is that the top layers of memory—colloquially referred to as the "closest" memory—take the shortest time to access data from. Why? Partly because they are much smaller and therefore naturally take less time to traverse to find the data.

Going back to tree traversals, consequently, if there happens to be a pointer to a node that we need to traverse that points to memory in an L2 cache, then it will take longer to traverse to that node than it would take to traverse to a node that is located in an L1 cache. Also, since pointers can point practically anywhere, this situation happens more often than you may think. How does the CPU (the Central Processing Unit) generally determine which data is placed where in memory **and** how can we take advantage of that knowledge to speed up our tree traversal *in practice*?

Note: It is a common to ask: "Why don't we just (somehow) fit our *entire* tree structure into a fast section of memory—such as an L1 cache—to ensure that we have fast constant accesses across the entire tree?" The problem with doing this is that the sections of memory that are fast (such as the caches) are actually quite small in size. Consequently, as our data structures grow in size, they will not be able to fit in such a small section of memory. You might subsequently wonder: "Well, why don't we just make that fast section of memory bigger?" Unfortunately, the bigger we make the memory, the slower it will be, and we don't want slower memory!

Step 2

As it turns out, the CPU determines which data to put in which sections of memory based largely based on *spatial* locality. In other words, if we are accessing data that is close to other data, the CPU will also load the close data into an L1 cache because it predicts that we might need to access the close data in the near future. For example, take a look at the for-loop below:

```
for(int i = 0; i < 10; i++) {

    a[i] = 0; //Along with a[0] initially, CPU will load a[1], a[2]... into close memory in
advance

}
```

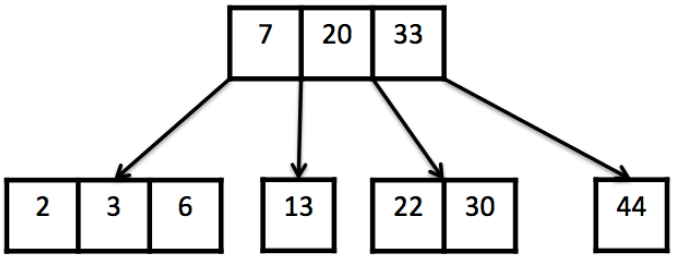
The CPU is going to try to load the entire array `a` into close memory (the L1 cache) because it predicts that we will probably need to access other sections of the array soon after (and what a coincidence, it's right)!

So, by knowing how the CPU determines which data is placed where in memory, what can this further tell us about accessing memory in tree structures? Well, it tells us that, if a node holds multiple pieces of data (e.g. an employee object that contains `string name`, `int age`, etc.), then by loading one piece of data from the node (e.g. the employee's name), the CPU will automatically load the rest of the data (e.g. the employee's age, etc.) from the same node. This implies that accessing other data that is *inside* the node structure is expected to be faster than simply traversing to another node.

Step 3

How can we take advantage of the information we just learned about spatial locality of memory to not just have a tree structure be fast *theoretically,* but also be fast *in practice*? The answer is to minimize the amount of node traversals we have to do to find data (i.e., make the trees as short as possible) **and** store as much data as possible close together (i.e., have each node store multiple keys). Why? By minimizing the amount of node traversals, we minimize the risk of having to access a section of memory that takes longer to access**.** By having a node store multiple keys, we are able to trick the CPU into loading more than one key at once into a fast section of memory (the L1 cache), thereby making it faster to find another key within the same node because it will already be in the L1 cache (as opposed to somewhere else in Main Memory).

The data structure that does the above is called a **B-Tree**. A **B-Tree** is a self-balancing tree data structure that generally allows for a node to have more than two children (to keep the tree wide and therefore from growing in height) and keeps multiple inserted keys in one node. We usually define a **B-Tree** to have "order *b*," where *b* is the minimum number of children any node is allowed to have and 2*b* is the maximum number of children any node is allowed to have. Below is an example of a **B-Tree** with *b* = 2:



In the **B-Tree** above, each integer is considered a separate key (i.e., each integer would have had its own node in a BST). Also, just like in other trees, every key in the left subtree is smaller than the current key and every key in the right subtree is greater (e.g. starting at key 7, keys 2, 3, and 6 are all smaller and key 13 is greater). Also note that, since the maximum number of children is 2*2 = 4, a node can therefore only store up to 3 keys (because pointers to children are stored *in between* keys and on the edges). More generally, we say that a node can store up to 2*b*−1 keys.

Formally, we define a **B-Tree** of order *b* to be a tree that satisfies these properties:

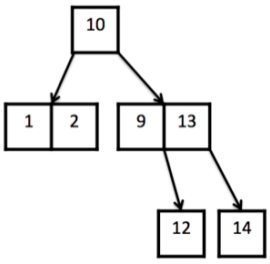
1. Every node has at most 2*b* children.
2. Every internal node (except root) has at least *b* children.
3. The root has at least two children if it is not a leaf node.
4. An internal node with *k* children contains *k*−1 keys.
5. **All leaves appear in the same level.** (We will later see how this is enforced during the insert operation)

Step 4

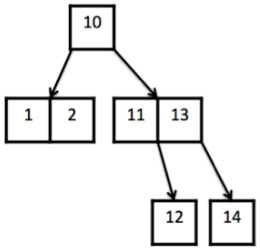
EXERCISE BREAK: Which of the following **B-Trees** below are valid for any arbitrary order *b*? (Select all that apply)

Note: Make sure to check that *all* **B-Tree** properties are satisfied for any valid tree.

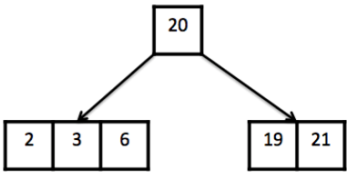
A.



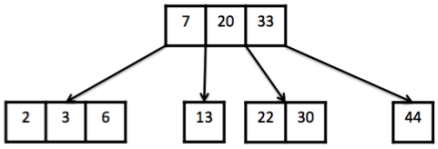
B.



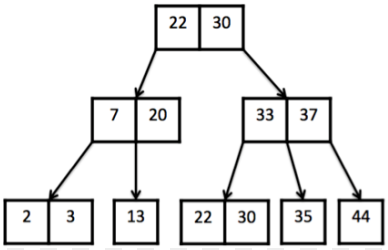
C.



D.



E.

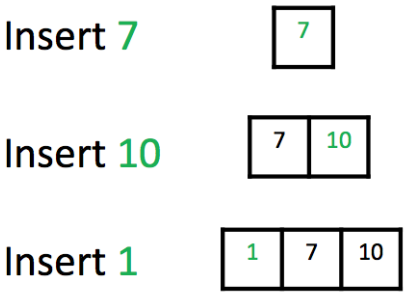


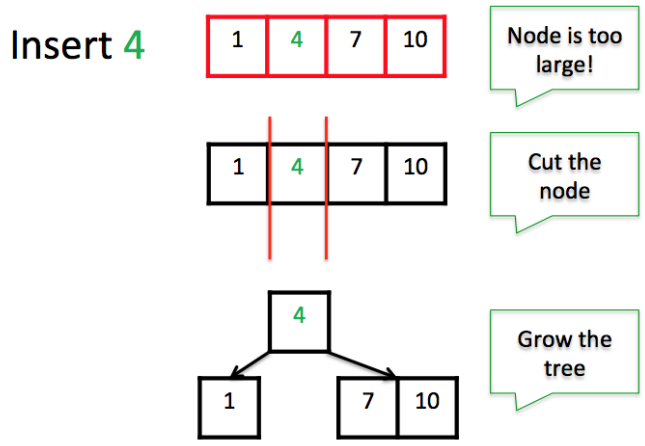
To solve this problem please visit https://stepik.org/lesson/30028/step/4

Step 5

How do we insert new elements into a a **B-Tree**? The insert operation for a **B-Tree** is a slightly intricate matter since, as we mentioned earlier, this is a *self-balancing* tree. Moreover, we have the requirement that *all* leaves *must* be on the **same** level of the tree. How do we achieve this? The secret to having the leaves constantly be on the same level is to have the tree grow *upward* instead of down from the root, which is an approach unlike any of the tree structures we have discussed thus far.

Let's look at an example for a **B-Tree** in which *b* = 2:

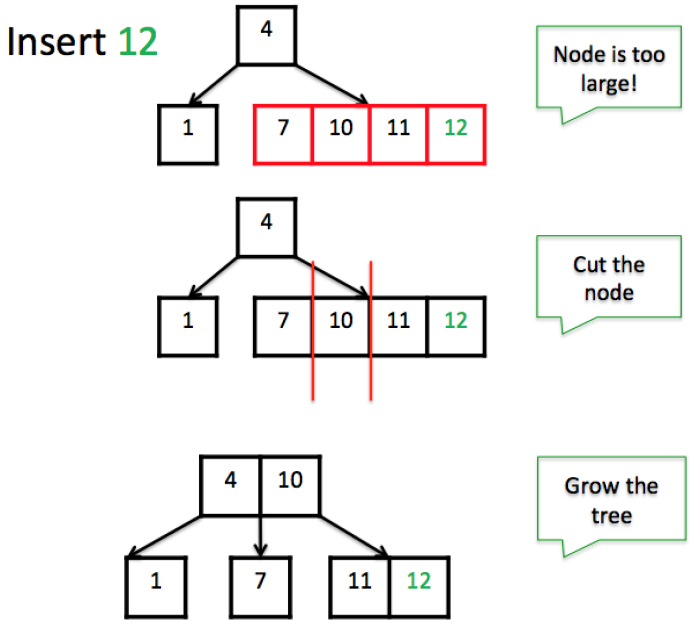




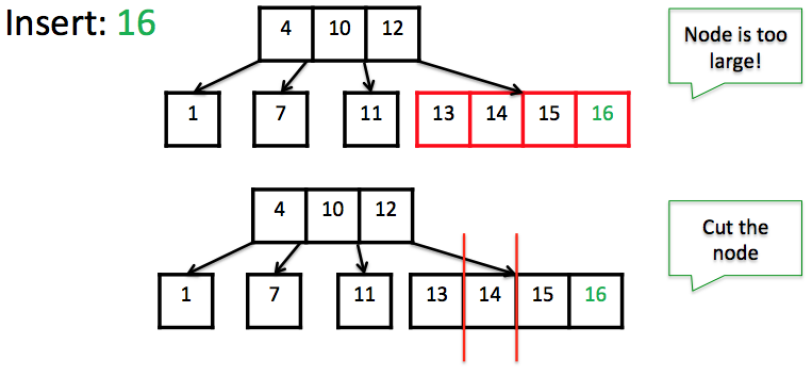
Notice how, every time we insert, we "shuffle" over the keys to make sure that the keys in our node stay properly sorted.
Also, notice that, in the first 3 insertions, there was just one node and that node was the root. However, in the last insertion, where our node grew too big, we cut the node and chose the largest key of the first half of the node to create a new root (i.e., the tree grew *upward*). By doing so, we have now kept all leaves on the same level!

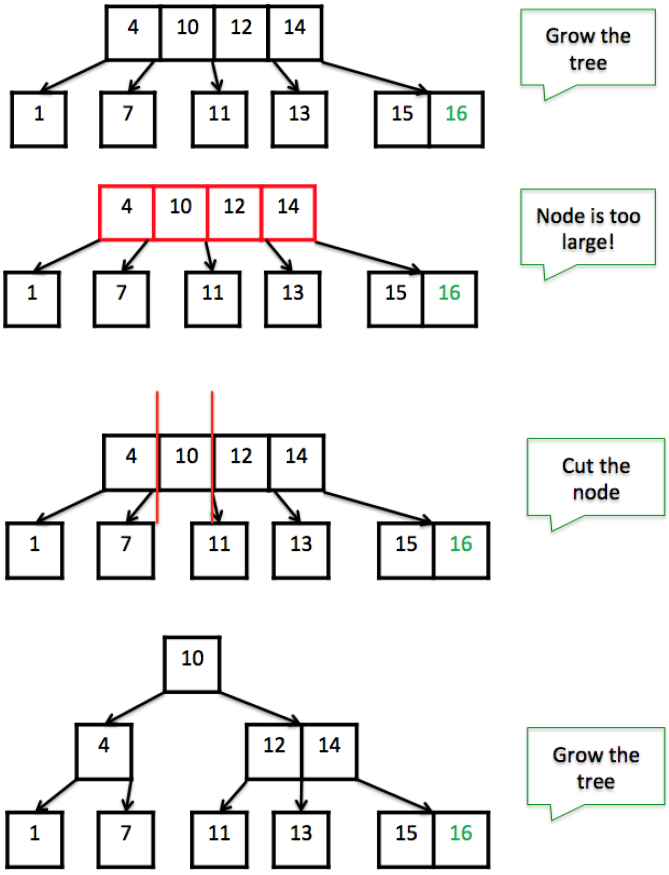
Step 6

If we continue to insert keys into the previous example, we will see that the insertion and re-ordering process generally stays the same:



Notice how, in the example above, since the root was not full, we allowed key 10 to become a part of the root. But what happens when the root becomes full *and* we need to grow upward? The same exact steps! The keys just "bubble up" the tree like so:





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): // return true upon success

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

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

    // while the extra key doesn't fit in the node, grow the tree
    while node.size > 2*b:
        left,right = result of splitting node in half
        parent = parent of node
        remove largest key from left and insert into parent (maintain sorted order)
        make left and right new children of parent
        node = parent

    return true
```

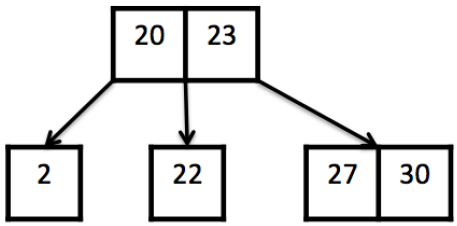
What is the worst-case time complexity for a **B-Tree** insert operation?

Well, in each insertion, we traverse the entire height of the tree to insert the key in one of the leaves, which is $O(log_b n)$. Worst-case scenario, during insertion, we need to re-sort the keys within each node to make sure the ordering property is kept; this is a $O(b)$ operation. We also need to keep in mind that in the worst case, the inserting node overflows and we therefore need to traverse all the way back up the tree to fix the overflowing nodes and re-assign pointers; this is a $O(log_b n)$ operation. Consequently, the worst-case time complexity for a **B-Tree** insert operation simplifies to $O(b\ log_b n)$.

Also, since we have been analyzing the performance of a **B-Tree** with respect to memory, we will also choose to analyze the worst-case time complexity with respect to memory (i.e., how much memory does the CPU need to access in order to insert a key). In the worst case, for each node, we need to read all the keys the node contains. Because the maximum number of keys we can store in a node is 2*b*–1, the number of keys the CPU needs to read in a node becomes $O(b)$. As mentioned before, the tree has a depth of $O(\log_b n)$, making the *overall* worst-case time complexity become $O(b \log_b n)$.

Step 8

EXERCISE BREAK: Which of the following key insertion orders could have built the **B-Tree** with *b* = 2 shown below? (Select all that apply)



To solve this problem please visit https://stepik.org/lesson/30028/step/8

Step 9

How would we go about finding a key in a **B-Tree**? Finding a key is actually very simple and similar to performing a Binary type search for each key in a given node. 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, node): // return true upon success

    if node is empty:
        return false

    if key 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, node[0].leftChild)

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

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

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

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

What is the worst-case time complexity of the find operation?

Well, worst case scenario we have to traverse the entire height of the balanced tree, which is O($log_b n$). Within each node, we can do a binary search to find the element, which is O($log_2 b$). Thus, the worst-case time to find an element simplifies to O($log\ n$). However, what is the worst-case time complexity of the find operation *with respect to memory access*? At each level of the tree, we need to read in the entire node of size O(*b).* Consequently, the worst-case time to find an element is equal to O($b\ log\ n$).

Step 10

How do we delete a key from a **B-Tree**? If we're lazy and don't care about memory management, then we can do a "lazy deletion" in which we just mark the key as "deleted" and just make sure that we check whether a key is marked as deleted or not in the find operation. However, the whole motivation behind **B-Trees** is to gain fast access to elements, partly by minimizing node traversals. Thus, if we start piling up "deleted keys," our nodes will fill up much more quickly and we will have to create unnecessary new levels of nodes. By repeatedly doing so, our find and insert operations will take longer (because the height of our tree will be larger), meaning we will essentially lose the efficiency we just spent so much time trying to obtain via clever design. Consequently, most programmers choose to invest their time into implementing a more *proper* delete method.

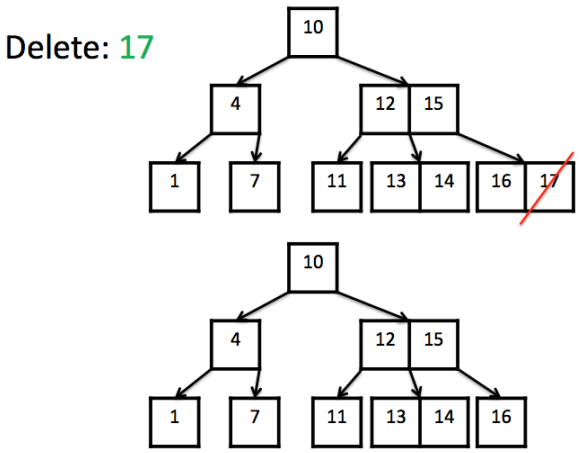
How do we implement a *proper* delete method?

The delete operation has 4 different cases we need to consider:

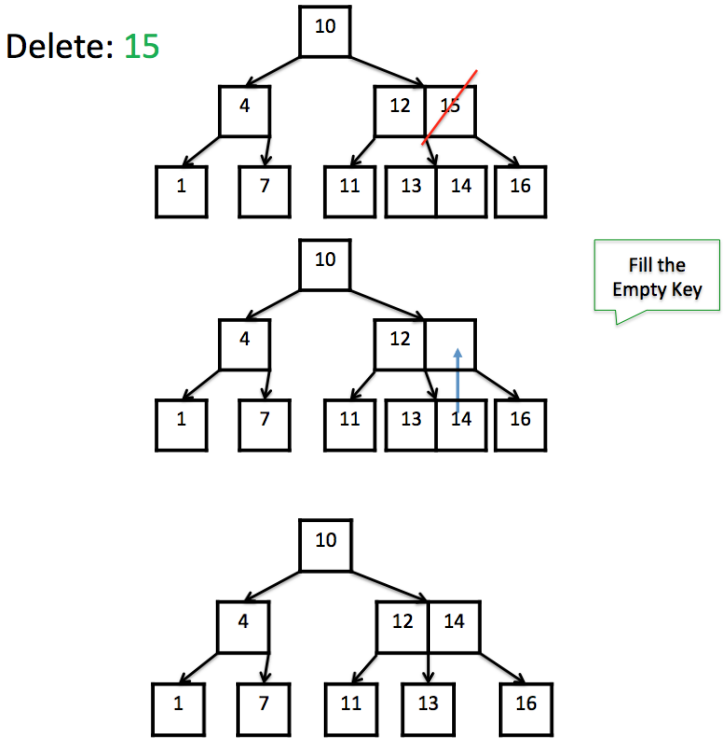
1. Delete leaf-key — no underflow
2. Delete non-leaf key — no underflow
3. Delete leaf-key — underflow, and "rich sibling"
4. Delete leaf-key — underflow, and "poor sibling"

Note: 'Underflow' is defined as the violation of the *"a non-leaf node with k children contains k−1 keys"* property.

Let's start with the easiest example, **Case 1: Delete a key at a leaf – no underflow**. Within this first case, no properties are violated if we just remove the key:



Case 2: Delete non-leaf key – no underflow. Within this second case, we face the problem of re-assigning empty key "slots" because there are pointers to leaves that need to be kept (this is different than in case 1 where we just deleted the key "slot").



When filling the empty key, we have two different options for choosing which key we want to use as a replacement:

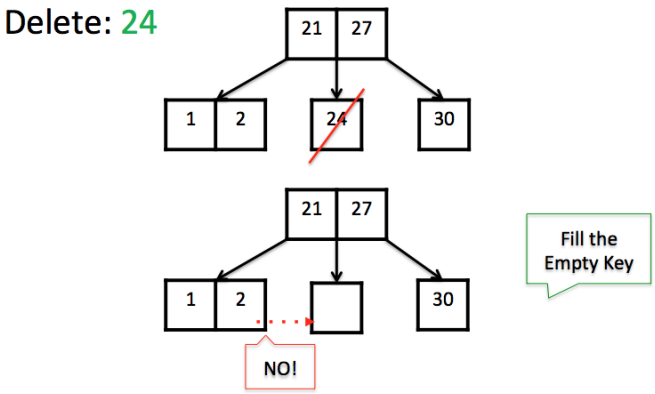
1. The largest key from the left subtree
2. The smallest key from the right subtree

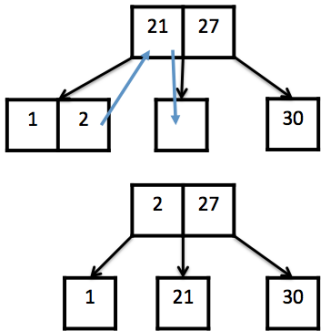
Note: If the node from which we took a key now also has an empty "slot" that needs to be filled (i.e., it wasn't a leaf), just repeat the same steps above recursively until we hit a leaf node!

Step 11

In the next two cases of the delete operation, we now face the problem of violating the *"a non-leaf node with k children contains k−1 keys"* property (i.e., we face an "underflow") and thus have to restructure the tree to avoid the violation. In other words, we can't just get rid of the "slot" as we did in the previous cases because that would lead to an internal node without enough children. In both of the following cases, we restructure the tree by taking a key from the parent node. However, the difference between the following two cases is determined by whether we also ***take*** a key from the immediate sibling or ***merge*** with a key from the immediate sibling.

Case 3: Delete leaf-key — underflow, and "rich sibling"



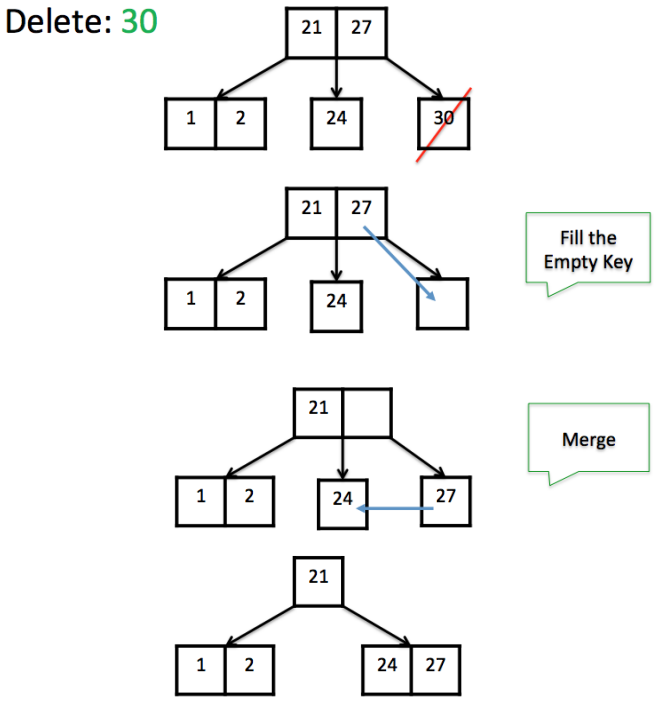


Note that, in the case above, the sibling was "rich" enough to provide a key to the parent in order for the parent to be able to give a key to the empty node.

STOP and Think: Why is it not okay to just move the sibling's largest key to the empty node (i.e., what becomes violated)?

But what happens if the empty node doesn't have an immediate sibling that can provide a key without causing more underflow?

Case 4: Delete leaf-key — underflow, and "poor sibling"



What is the worst-case time complexity of the delete operation for a **B-Tree**? It turns out that it is the same as the insert operation, O($b * log_b n$). Why? In the worst case, we need to traverse over all the keys in a node (O(b)) in all levels of the tree (O($log_b n$), similarly to insert.

Note: Some programmers choose to voluntarily violate the "underflow" conditions described above in order to simplify the deletion process. However, by doing so, we risk having a taller tree than necessary, thereby making it less efficient. Also note that the real code to implement the delete operation is *extremely* long (much longer than the code to implement the insert operation) because of the excessive shuffling of keys and pointer manipulation that needs to take place.

Step 12

To better understand the behavior of **B-Trees**, here is a visualization created by David Galles at the University of San Francisco. We recommend actually playing around with this visualization to make sure that you are comfortable with all the insertion and deletion edge cases that could happen.

Note: The "Max Degree" (maximum number of children) option is equal to 2**b*. Consequently, if you are trying to build a **B-Tree** of order *b*=3, then you would need to set "Max Degree" to 2*3 = 6.

Step 13

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/30028/step/13

Step 14

As **B-Trees** grow large, we start to expect that the CPU will need to access the Hard Disk (the slowest section of memory in our computer) every now and then because there is no way that the rest of the tree can fit in the closer and faster sections of memory. Because it is known that the Hard Disk is designed in block-like structures (to help the CPU traverse memory more efficiently), a single node in a **B-Tree** is specifically designed to fit in one "Hard Disk block" to ensure that the CPU copies over *all* the data from a single node in one access.

Moreover, since **B-Trees** have such a good system in place for efficiently reading stored data from the Hard Disk, it is very common for **B-Trees** to explicitly store their data directly on the Hard Disk so we can save the **B-Tree** and have the ability to access it later without having to re-build it from scratch. Note that, if we do this, then the worst case number of Hard Disk reads that we will need to make is equal to the height of the **B-Tree**!

Databases and filesystems are common for storing large amounts of relational data on the Hard Disk. Consequently, the optimizations done behind **B-Trees** are extremely relevant for these systems as well. Therefore, in the next lesson, we will look at a variant of the **B-Tree**, called a **B+-Tree**, that organizes data slightly differently to help better implement systems such as databases.