CSC 370

Activity Worksheet:
B Plus Trees

Dr. Sean Chester

Fall 2022

Notes

In this worksheet, you will get extra practice modifying B+-Trees. There are two sections to the worksheet: in the first section, you will only perform insertions and lookups; in the second section, you will have a mixture of insertions and deletions.

Questions

1. Section 1 - B Plus Tree Insertions and Lookups: For all of these problems, assume that a node has capacity for four keys and that all keys are unsigned integers.

   (a) Draw the B Plus Tree that results from inserting into an empty B Plus Tree all keys between 0 and 24 in ascending order. Show your work.

   Solution:

(b) How many I/O's are required to look up key 7 and all keys in the closed range [7, 16]?

Solution:

(c) Provide an insertion order for all keys between 0 and 24 that produces a shorter B Plus Tree than in part a. Show the resulting tree.

2. Section 2 - B Plus Tree Deletions: Recall that all non-root nodes of a B+-Tree must use at least half of their capacity for keys. A deletion can simply remove a key from a leaf if that leaves at least half of the capacity used. Otherwise, it can either steal a key from a sibling or merge with a sibling. For the questions below, use the following B Plus Tree.

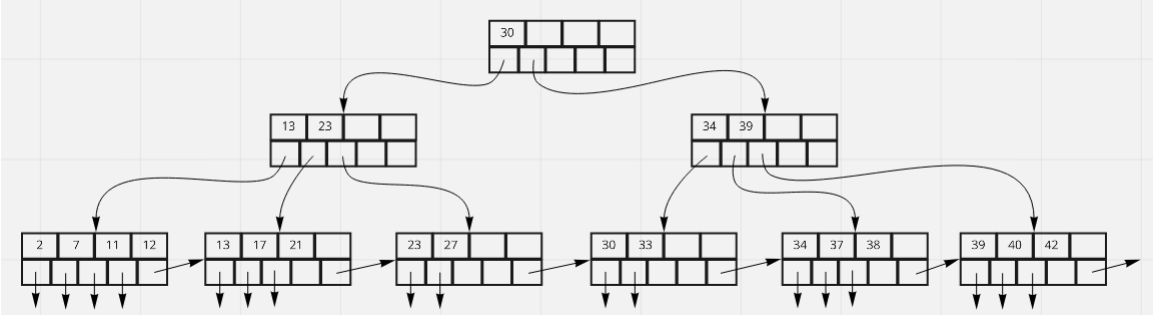


Figure 1: B Plus Tree

Draw the B Plus Tree that results from the following sequence of deletions:

(a) Delete Key 13

Solution:

(b) Delete Key 33

Solution:

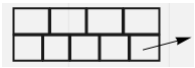
(c) Delete Key 27

Solution:

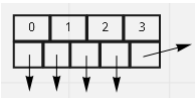
Solutions

Question 1A

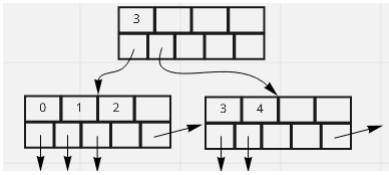
To begin with, we only have one empty node as the root:



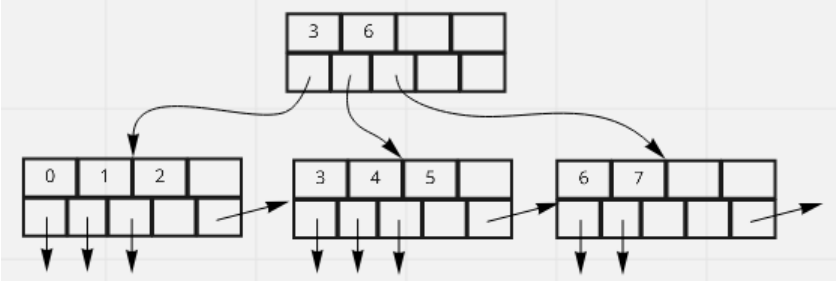
We can add the first keys with no difficulty. We perform a lookup which of course returns the only leaf node. While there is space, we simply add the new keys, resorting if necessary (it's not, in this case). Each insertion requires 1 read I/O and 1 write I/O. At that stage, we have:



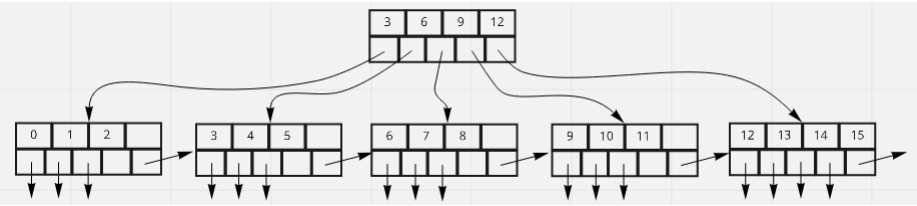
When we try to add the fifth key, there isn't any spare capacity. So, we split this node into two, putting the first three values in the left leaf and the last two values in the right leaf. We also set this node's next pointer to be the new node and the new node's next pointer to be whatever the original leaf's next pointer pointed to (standard insertion into a linked list). Generally we would update the parent, but this is the root node (i.e., it has no parent). Thus, we also need to create a parent/root node which will have the first key of the second leaf as its lone key. In the end, after 1 read I/O and 3 write I/O's (plus 2 new block allocations), we have the following tree:



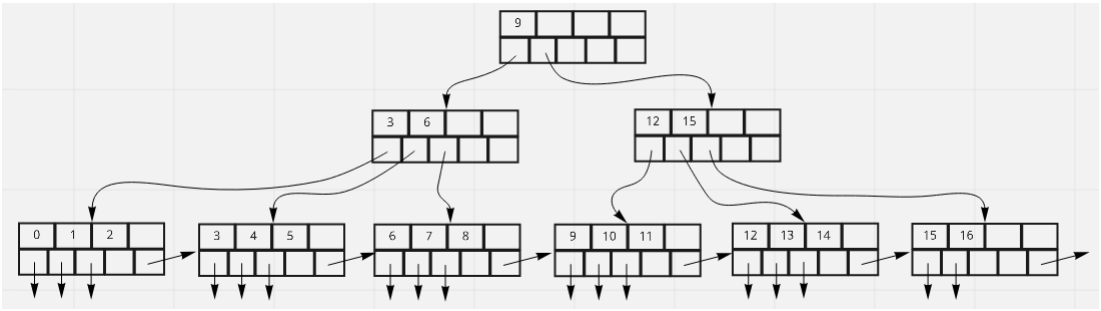
The next two keys can be added easily with 2 read I/O's to locate the second leaf and 1 write I/O because there is spare capacity in that leaf for the insertion. When we add key 7, however, we need to split the second leaf. As before, we allocate a new block, put the three smallest values in the left child and the two largest values in the right child, update next pointers in the linked list, and copy the leftmost key of the right child into the parent who is then linked to the new node. In total, we read 2 I/O's, write 3 I/O's (and allocate 1 new block) to produce the following tree:



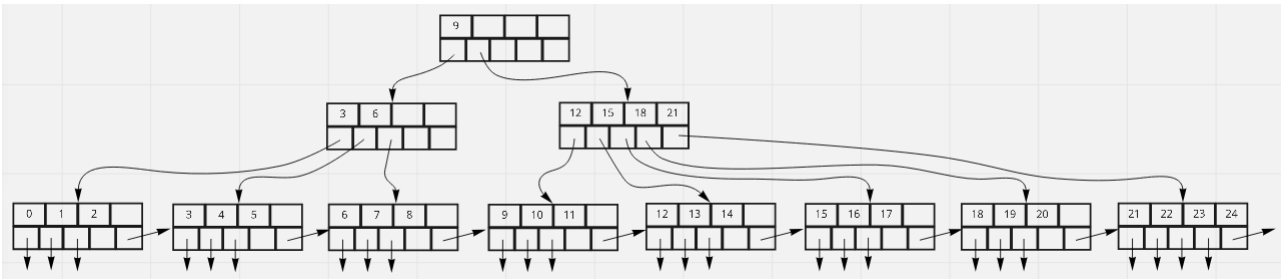
The next several insertions follow analogously until we have inserted the first sixteen keys. At that point we have the following tree:



Adding key 16 overflows the rightmost leaf node, but this time when we split it, there is no room in the parent. So, we need to split the parent as well. When we split the parent, we give the left node half the first half of the pointers and we give the right node the other half of the pointers. We promote the middle key up to their parent. In this case, they don't have a parent because we just split the root, so we create a new parent node, thereby increasing the height by 1. In total, we incurred 2 read I/O's, 5 write I/O's (and 2 new block allocations) to arrive at the following tree:



The remaining insertions follow the patterns that we have already seen. Ultimately, we end up with the following tree:



Question 1B

To lookup key 7, we:

- Load the root into memory (1 read I/O).
- Scan the list of keys to find the smallest key i that is greater than equal to 7, then following pointer i. In this case i = 0.
- We load the node into memory at the address of the pointer (1 read I/O).
- Scan the list of keys to find the smallest key j that is greater than equal to 7, then following pointer j. In this case j = 2 (there are no such keys).
- We load the leaf into memory at the address of the pointer (1 read I/O).
- Observing that this is a leaf, we scan the keys to see if any of them are 7. In this case, we indeed find it.

Thus, the lookup cost 3 read I/O's. (A fourth may be necessary to access the actual tuple if we need it.)

To lookup keys in the range [7, 16], we:

- Perform a lookup on key 7, which (as shown above) costs 3 read I/O's.
- Scan the remaining keys to see if any are greater than our upper bound 16. They are not, so we add them all to our result.
- We follow the leaf node's next pointer to grab the next leaf in the linked list (1 read I/O).
- Scan all keys in the new leaf until we find one ¿16. There are none, so we repeat the steps above.
- Eventually we reach the node with key 17 (the first key ¿16). We abort at that point.

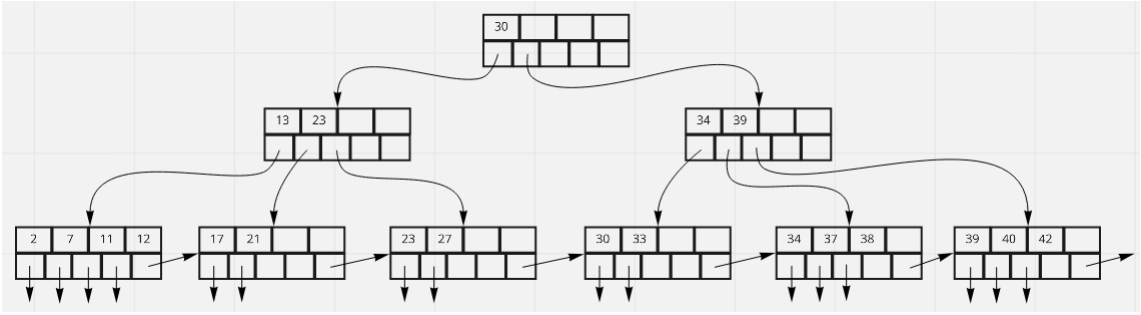
Thus, the range query costs 3 read I/O's for the initial lookup, plus another 3 read I/O's on the linked list to retrieve all keys not greater than 16. The total is 6 read I/O's.

Question 1C

We can see that this is not possible and the tree above already has the minimum height. A B+-tree tree of height 2 with a fan-out of 5 can store at most 5*(5-1)=20 keys.

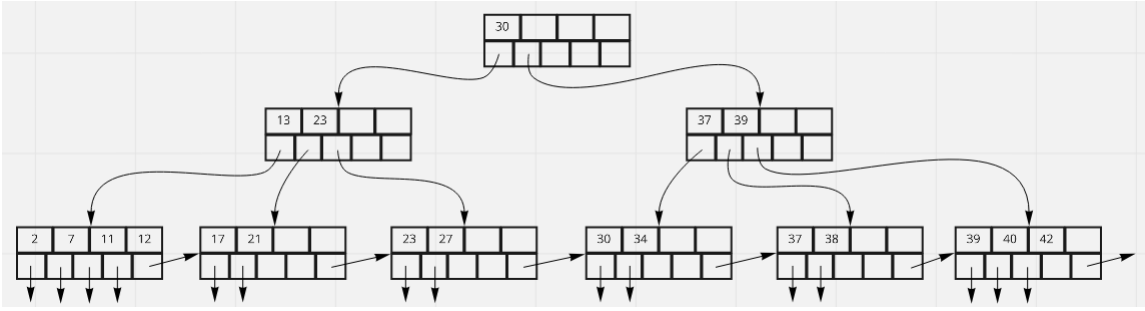
Question 2A

This is an easy key to delete, because after locating it in and then removing it from the second leaf there will still be more than half of the key positions occupied. There is no need to update the parent, because the key 13 still correctly splits the content of the first leaf from those of the second leaf. We do need to shift the keys in the leaf one position left though to maintain a contiguous range. After 3 read I/O's and 1 write I/O, we have:



Question 2B

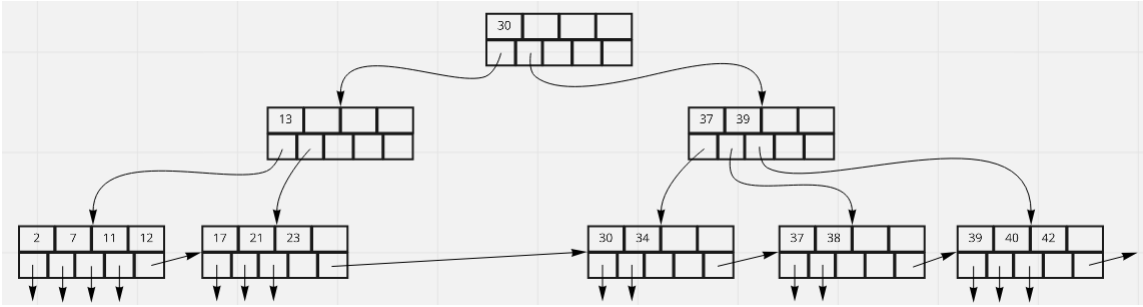
We first locate key 33 with 3 read I/O's in the fourth leaf. However, we cannot simply delete it, because then the leaf would be under-capacity. We follow its next pointer with 1 extra read to find its successor sibling. We see that the sibling can donate its lowest key; so, we replace 33 with 34, and shift the keys of the sibling leaf left by one position. In this case, we also have to update the parent, because the key 34 is no longer greater than all keys in the original child. Thus, we end up modifying and writing back 3 blocks. After 4 read and 3 write I/O's, we have the following tree:



Question 2C

This is pretty much the worst possible case (and is quite rare in practice). We locate key 27 in the third leaf after 3 read I/O's. However, we cannot delete the key without falling under-capacity. Moreover, we cannot borrow from the left sibling even after incurring the cost of 1 I/O to check. (We could theoretically redistribute the first leaf's keys, but we don't incur more than 1 read to check siblings, so we don't know this.)

So, we have to collapse the node into its sibling that we already read. This leads to the following intermediate state:



However, now the directory node (parent) is under capacity. We incur an extra I/O to read its sibling using the next pointer in the root (the parent's parent). However, again, we cannot borrow a key from the sibling. Thus, we must collapse these two nodes as well. Observe that this will leave the root with zero keys; so, it is deleted entirely, its key is stolen by the child, and the tree shrinks to a height of 2. After a total of 5 read I/O's and 2 write I/O's (and 3 blocks freed), we have the following final tree: