

*Goals:*

- Motivate the design of an efficient search tree DS
- Specify the objectives of such a DS
- Conceptually understand B-trees and their various supporting operations
- Reason about the effectiveness of B-trees
- Appreciate the practical applications of B-trees

### Problem 1. B-Trees

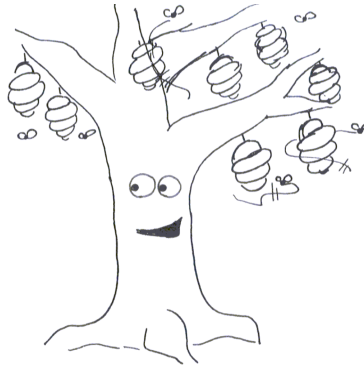


Image credit: [Jeremy Fineman](#)

In the first lecture you were asked whether the fastest way to search for data is to store them in an array, sort them and then perform binary search. The answer to that question is — surprisingly — no! You have seen Binary Search Trees (BSTs), where we design a DS that always maintains data in an (hierarchically) ordered manner. This will allow us to avoid the overhead of sorting before we search. However, you also learnt that unbalanced BSTs can be terribly inefficient for insertion, deletion and search operations (*why?*).

In this week's lecture, you were introduced to the idea of self-balancing BST such as an AVL-tree (other variants exist!). Today, we will learn about **B-trees**, which is another (very important!) self-balancing tree DS for maintaining ordered data. It is important to note that the 'B' in B-tree *does not* stand for "Binary" so don't get confused between B-Trees and BSTs. They are not the same!

# 1 $(a, b)$ -trees

Before we talk about B-trees, we first introduce its family (generalized form) which are called  $(a, b)$ -trees. In an  $(a, b)$ -tree,  $a$  and  $b$  are parameters where  $2 \leq a \leq (b+1)/2$ . Respectively,  $a$  and  $b$  refer to the minimum and the maximum number of children an internal node in the tree (i.e. non-root, non-leaf) can have.

The main differences between binary trees and  $(a, b)$ -trees can be summarized in the following table.

Binary trees	$(a, b)$ -trees
Each node has at most 2 children	Each node can have more than 2 children
Each node stores exactly one key	Each node can store multiple keys

## 1.1 Rules

*Pro-Tip:* Review these rules by concurrently referring to the  $(2, 4)$ -tree example in Figure 1.

Rule 1  $(2, 4)$   $(1, 2)$   
 $(a, b)$ -child Policy"

Node type	#Keys		#Children		Node type	#keys		#Children	
	Min	Max	Min	Max		Min	Max	Min	Max
Root	1	$b-1$	2	$b$	Root	1	3	2	4
Internal	$a-1$	$b-1$	$a$	$b$	Internal	1	3	2	4
Leaf	$a-1$	$b-1$	0	0	Leaf	1	3	0	0

**Table 1:** For each type of node, we can either define this rule in terms of the number of children or equivalently in terms of the number of keys permitted.

$$n-1 \text{ keys} \quad b-1+b$$

With the exception of leaves, realize that the number of children is always one more than the number of keys (see Rule 2).

$$b-1+b = n \quad b = \frac{n+1}{2}$$

$$2b-1 = n \quad 2b \geq n+1$$

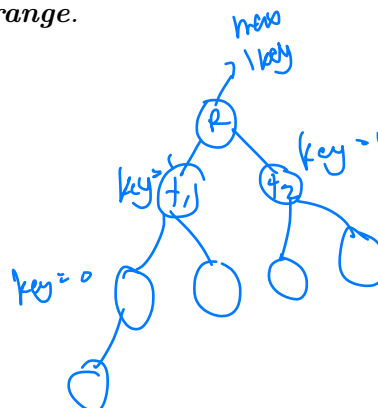
Rule 2 "Key-ordering"

A **non-leaf node** (i.e. root or internal) must have one more child than its number of keys. This is to ensure that all value ranges due to its keys are covered in its subtrees. We shall henceforth refer to the permitted range of keys within a subtree to be its **key range**.

Specifically, for a non-leaf node with  $k$  keys and  $k+1$  children:

- Its keys in sorted order are  $v_1, v_2, \dots, v_k$
- The subtrees due to its keys are  $t_1, t_2, \dots, t_{k+1}$

Then:



- First child  $t_1$  has key range  $\leq v_1$
- Final child  $t_{k+1}$  has key range  $> v_k$
- All other children  $t_i$  where  $i \in [2, k]$  has key range  $(v_{i-1}, v_i]$

### Rule 3 “Leaf depth”

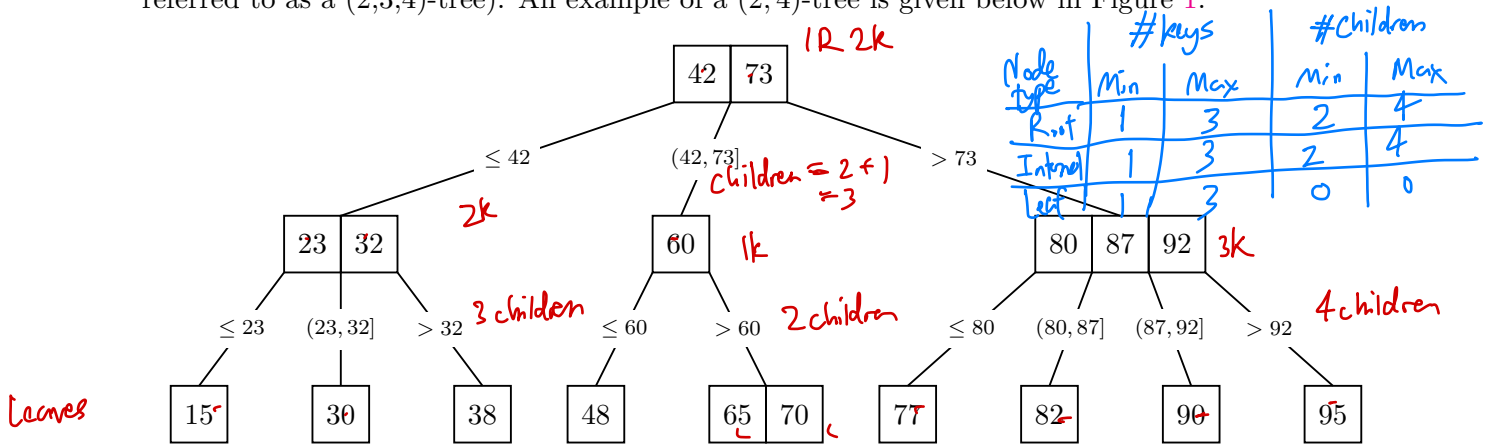
$(1, 2)$ -tree

All leaf nodes must all be at the same depth (from root).

**Test yourself:** Given these rules, is a BSTs just an  $(a, b)$ -tree where  $a = 1$  and  $b = 2$ ? Why or why not?

Yes?

**B-trees** are simply  $(B, 2B)$ -trees. That is to say, they are a subcategory of  $(a, b)$ -trees such that  $a = B$  and  $b = 2B$ . For instance, when  $B = 2$ , we have a  $(2, 4)$ -tree. (FYI: this is sometimes referred to as a  $(2, 3, 4)$ -tree). An example of a  $(2, 4)$ -tree is given below in Figure 1.



**Figure 1:** A  $(2, 4)$ -tree storing 18 keys. The range of keys contained within a subtree is indicated by the label on the edge from its parent.

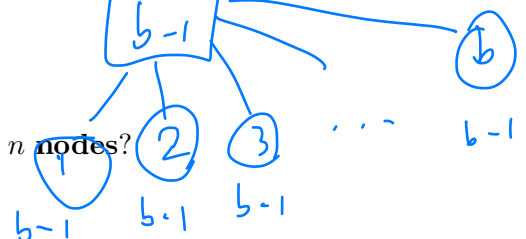
**Problem 1.a.** Is an  $(a, b)$ -tree balanced? If it is, which rule(s) ensures that? If not, why?

**Problem 1.b.** What is the minimum and maximum height of an  $(a, b)$ -tree with  $n$  keys?

$$\log_b n \leq \text{height} \leq \log_a n$$

**Problem 1.c.** Write down the pseudocode for searching an  $(a, b)$ -tree. What data structure should we use for storing the keys and subtree links in a node?

**Problem 1.d.** What is the cost of searching an  $(a, b)$ -tree with  $n$  nodes?

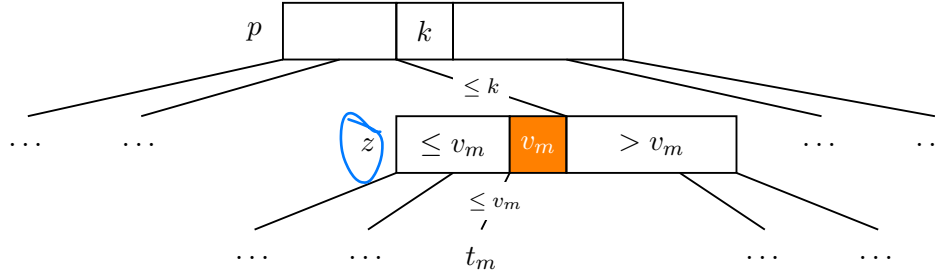


## 1.2 split operation

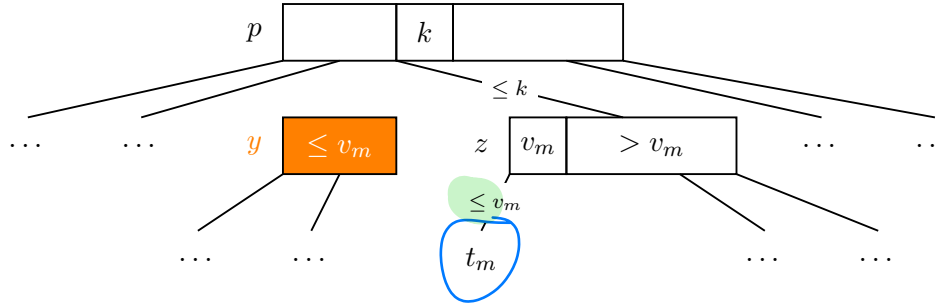
By definition, the three rules (Section 1.1) must hold before and after any operation that  $(a, b)$ -trees support. To ensure this, we need to take care of any potential violations that can occur across all operations. It should be obvious that **insertion** and **deletion** poses risks of violating the rules since they alter membership.

We will first look at **insertion**. This operation clearly risk nodes growing beyond permissible bounds. To address this, we will sometimes need to **split** large nodes into smaller ones. Firstly, keys in the violating node  $z$  are divided into two parts using the median index in its keylist (think partition step in QuickSort). We shall refer to the key at the median index as the split key. This ensures that each part contains half of  $z$ 's keys along with their associated subtree links. The split key is then inserted into  $z$ 's parent. Specifically, split operation on a non-root node  $z$  with  $b$  keys is outlined in the following algorithm:

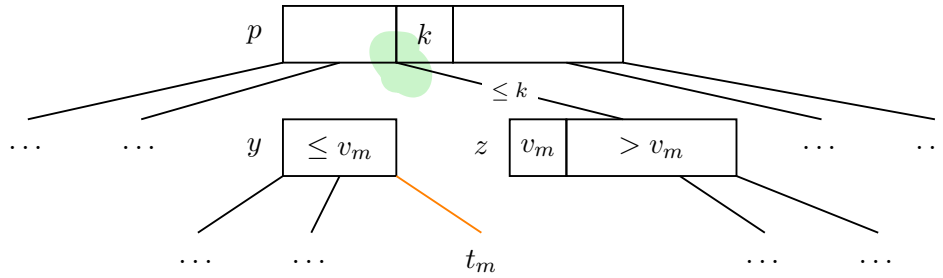
1. Find the median key  $v_m$  where the number of keys in  $z$  that are  $\leq v_m$  (i.e. LHS) is  $\lfloor (b-1)/2 \rfloor$  and the number of keys  $> v_m$  (i.e. RHS) is  $\lceil (b-1)/2 \rceil$ . Since by definition  $b \geq 2a$ , the sizes of each half can be respectively rewritten as  $\geq \lfloor (2a-1)/2 \rfloor = a$  and  $\geq \lceil (2a-1)/2 \rceil = a-1$ . Therefore both LHS and RHS each have at least  $a-1$  keys, satisfying Rule 1.



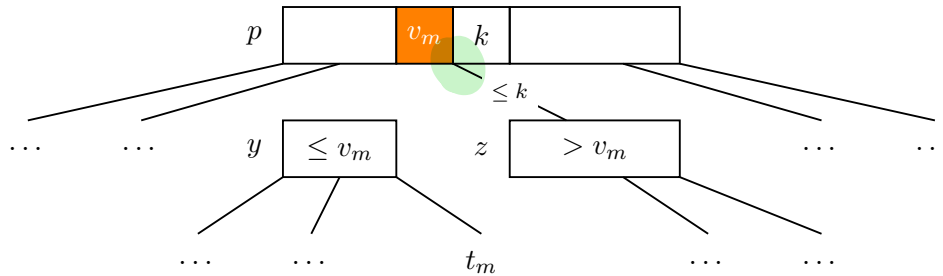
2. In  $z$ , separate all keys  $\leq v_m$  as well as their associated subtree links to form a new adjacent node  $y$ .



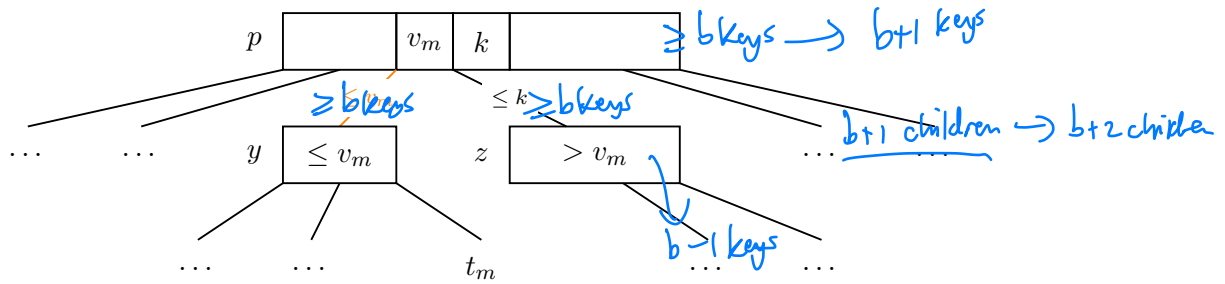
3. Since  $y$  will be missing a final child corresponding to key range  $> y_{m-1}$ , assign it to be  $t_m$ , the subtree associated with  $v_m$ . Remove the the association between  $t_m$  and  $v_m$ .



4. Move key  $v_m$  from  $z$  to  $p$  ( $z$ 's parent). This newly inserted key should immediately precede the key in  $p$  which is associated to node  $z$ .



5. Add  $y$  as a child of  $p$  by associating it with newly inserted  $v_m$  in  $p$ . We can do this because all keys in  $y$  are guaranteed to be  $\leq v_m$  by virtue of the split.



6. Return  $(y, v_m, z)$ .

**Test yourself:** Why do we have to give one key to the parent as 'offering'? *Hint: Recall Rule 1.*

You should convince yourself that such a split operation on a non-root node  $z$  is safe (i.e. post-conditions satisfy all rules) so long as it satisfies the following conditions:

1.  $z$  contains  $\geq 2a$  keys, because after splitting
  - LHS has  $\geq a - 1$  keys
  - RHS has  $\geq a$  keys

2.  $z$ 's parent contains  $\leq b - 2$  keys

Test yourself: What happens if after the split operation, the parent node has  $\geq b$  keys?

The algorithm outlined before applies to splitting a non-root node  $z$ , but what happens when  $z$  is actually the root node? What's different now is that there is no parent to offer the split key  $v_m$  to! Hence following the same procedure, now we simply create a new node, offer it the split key  $v_m$  and then promote it to become the new root node. Recall from Rule 1 that we have made it legal to have a root with one key — the purpose of that is to simply support this root-splitting operation!

Specifically, the root-splitting operation can be outlined as follows:

- |           |   |
|-----------|---|
| Old steps | 1. Pick median key $v_m$ as split key   |
|           | 2. Split $z$ into LHS and RHS using $v_m$   |
|           | 3. Create a new node $y$  |
|           | 4. Move LHS split from $z$ to $y$   |
| <hr/>     |   |
| New steps | 5. Create a new empty node $r$  |
|           | 6. Insert $v_m$ into $r$  |
|           | 7. Promote $r$ to new root node   |
|           | 8. Assign $y$ and $z$ to be the left and right child of $r$ respectively            |
|           | 9. Assign previous subtree $t_m$ associated with $v_m$ to be the final child of $y$ |

Problem 1.e. Come up with an example each for splitting an internal node, and for splitting a root node.

### 1.3 Insertion

Alike BSTs, new items are also inserted only in the leaves of an  $(a, b)$ -tree. With the **split** operation defined, it is now easy to specify the full insertion procedure of an item  $x$  into the tree:

1. Start at node  $w = \text{root}$
2. Repeat until  $w$  is a leaf:
  - 2.1. If  $w$  contains  $b - 1$  keys,
    - 2.1.1. Split  $w$  into  $y$  (LHS) and  $z$  (RHS) using split key  $v_m$
    - 2.1.2. Examine split key  $v_m$  returned by the split operation
      - If  $x \leq v_m$ , set  $w = y$

- Else, set  $w = z$

2.2. Search for  $x$  in the keylist of  $w$

- If  $x$  is larger than all the keys, set  $w$  to be the final child
- Else, look for the first key  $v_i \geq x$  and set  $w$  to be  $t_i$  (subtree with key range  $\leq v_i$ )

3. Add  $x$  to  $w$ 's keylist

Notice in the procedure outlined above, we preemptively split any nodes we encounter along the way which are at maximum key capacity (i.e.  $b - 1$ ). This is known as a *proactive* approach. With such an approach, we are guaranteed that the parent of the node to be split will not have too many keys after the addition of the split key. As opposed to the proactive approach, the *passive* approach is a lazier strategy whereby the new item is first inserted and then we check the parent for violation, potentially having to perform splitting all the way up to the root node.

**Problem 1.f.** Prove that:

- If node  $w$  is split, then it satisfies the preconditions of the split routine
- Before  $x$  is inserted into  $w$  (at the last step), there are  $< b$  keys in  $w$  (so key  $x$  can be added)
- All three rules of the  $(a, b)$ -tree are satisfied after an insertion

**Problem 1.g.** What is the cost of inserting into an  $(a, b)$ -tree with  $n$  nodes?

3. depth doesn't change

2.

1.

or

we ensure  $w$  has  $b-1$  keys

↓  
split → confirm  $< b$

## 1.4 merge and share operations

Deletion in an  $(a, b)$ -tree risks having nodes shrink too small. In order to support deletion in an  $(a, b)$ -tree without violating rules afterwards, we need to introduce two different operations on nodes, each for handling a separate scenario after removing a key from the tree:

Scenario	Operation	Algorithm
<ul style="list-style-type: none"> <li>• <math>y</math> and <math>z</math> are siblings</li> <li>• Have <math>\leq b - 1</math> keys together</li> </ul>	$\text{merge}(y, z)$	<ol style="list-style-type: none"> <li>1. In parent, delete key <math>v</math> (the key separating the siblings)</li> <li>2. Add <math>v</math> to the keylist of <math>y</math></li> <li>3. In <math>y</math>, merge in <math>z</math>'s keylist and treelist</li> <li>4. Remove <math>z</math></li> </ol>
<ul style="list-style-type: none"> <li>• <math>y</math> and <math>z</math> are siblings</li> <li>• Have <math>(b - 1, 2b]</math> keys together</li> </ul>	$\text{share}(y, z)$	<ol style="list-style-type: none"> <li>1. <math>\text{merge}(y, z)</math></li> <li>2. Split newly combined node using the regular <b>split</b> operation</li> </ol>

## 1.5 Deletion

Now it is easy to delete an item  $x$  from the tree:

1. Start at node  $w = \text{root}$
2. Repeat until  $w$  contains  $x$  or  $w$  is a leaf:
  - 2.1. If  $w$  contains  $a - 1$  keys and  $z$  is a sibling of  $w$ 
    - If  $w$  and  $z$  together contain  $\leq b - 1$  keys,
      - i.  $\text{merge}(w, z)$
      - ii. Set  $w$  to be the newly merged node
    - Else if  $w$  and  $z$  together contain  $(b - 1, 2b]$  keys,
      - i.  $\text{share}(w, z)$
      - ii. Set  $w$  to be the newly split node whose key range contains  $x$
  - 2.2. Search for  $x$  in the keylist of  $w$ 
    - If  $x$  is larger than all the keys, set  $w$  to be the final child
    - Else, look for the first key  $v_i \geq x$  and set  $w$  to be  $t_i$  (subtree with key range  $\leq v_i$ )
3. Check for  $x$  in  $w$ 's keylist,



- If it exists, remove it and return success
- Else, return failure

Just like insertion, notice in the deletion procedure outlined above, we also employed a proactive approach to **merge/share** any nodes we encounter along the way which are at minimum key capacity (i.e.  $a - 1$ ). With such an approach, we are guaranteed that the parent of the node to undergo **merge/share** will not be left with too few keys after the removal of the key separating the siblings. On the other hand, the *passive* approach for deletion first deletes the target key, carry out **merge/share** on the node (if necessary) and then check the parent for violation, potentially having to perform **merge/share** all the way up the tree.

**Problem 1.h.** Come up with an example each for merge and share operations, as well as a key deletion. Prove that for a merge or share, the necessary preconditions are satisfied before the operation, and that the three rules of an  $(a, b)$ -tree are satisfied afterwards. Thereafter, prove that after a delete operation, the three rules of an  $(a, b)$ -tree are satisfied. What is the cost of deleting from an  $(a, b)$ -tree?

## Problem 2. External Memory (or why we care about B-Trees)

In general, large amounts of data are stored on disk. And searching big data is where efficiency is most important and  $(a, b)$ -trees shine. In general, data is stored on disk in blocks, e.g.,  $B_1, B_2, \dots, B_m$ . Each block stores a chunk of memory of size  $B$ . You can think of each block as an array of size  $B$ .

Note that when you want to access a memory location in some block  $B_j$ , the entire block is read into memory. You can assume that your memory can hold  $M$  blocks at a time. Transferring a block from disk to memory is expensive: it requires spinning up the harddrive platter, moving the arm to the right track (using, e.g., the elevator algorithm!), and reading the disk while the platter spins. By contrast, reading the block once it is stored in memory is almost instantaneous (by comparison), e.g., many, many orders of magnitude faster. (See Table 2)

To estimate the cost of searching data on disk, let us only count the number of blocks we have to move from disk to memory. (We can ignore the cost of accessing a block once it is in memory. If we run out of space in memory (e.g., we need more than  $M$  blocks), then one of the blocks we do not need should be kicked out. For today, let's assume that never happens.

CPU caches are housed within the CPU which means they are blazing fast memory units. This is why they sit at the top of the memory hierarchy. CPU caches are organized in levels, with L1 being the fastest and L3 being the slowest. To keep our discussions simple, we will be ignoring them today.

Notice that accessing the disk, especially if it is a magnetic disk (as is typical for 100TB disks), is 100,000 times slower than accessing memory. Just as a numerical example, if 90% of your data is

Memory unit	Size	Block size	Access time (clock cycles)
L1 cache	64KB	64B	4
L2 cache	256KB	64B	10
L3 cache	up to 40MB	64B	40–74
Main memory	128GB	16KB	200–350
(Magnetic) Disk	Arbitrarily big	16KB	20,000,000 (An SSD is only 20,000)

**Table 2:** Some typical numbers (from the Haswell architecture data sheet)

found in the L1 cache, 8% of your data is found in the L2 cache, and 2% of your data is in main memory, that means that you will be spending 57% of your time waiting on main memory. If you have to get any data from disk, you will be spending 99% of your time waiting for the disk!

We will be focusing on data stored on disk, and counting the number of block transfers between disk and main memory.

**Problem 2.a.** Assume your data is stored on disk. Your data is a sorted array of size  $n$ , and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a *linear search* for an item? Leave your answer in terms of  $n$  and  $B$ .

**Problem 2.b.** Assume your data is stored on disk. Your data is a sorted array of size  $n$ , and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a *binary search* for an item? Leave your answer in terms of  $n$  and  $B$ .

Now, imagine you are storing your data in a B-tree. (Notice that you might choose  $a = B/2$  and  $b = B$ , or  $a = B/4$  and  $b = B/2$ , etc., depending on how you want to optimize.)

Notice that each node in your B-tree can now be stored in  $O(1)$  blocks, for example, one block stores the key list, one block stores the subtree links, and one block stores the other information (e.g., the parent pointer, pointers to the two other blocks, and any other auxiliary information).

**Problem 2.c.** What is the cost of searching a keylist in a B-tree node? What is the cost of splitting a B-tree node? What is the cost of merging or sharing B-tree nodes?

**Problem 2.d.** What is the cost of searching a B-tree? What is the cost of inserting or deleting in a B-tree?

**Problem 3.** (Extra challenge problems)

**Problem 3.a.** Imagine you have two  $B$ -tree  $T_1$  and  $T_2$ , where every element in  $T_1$  is less than every element in  $T_2$ . How could you merge these into a single  $(a, b)$ -tree. (Remember, the resulting tree has to satisfy the three rules of an  $B$ -tree.) What is the cost of this operation?

**Problem 3.b.** Conversely, what if you have an  $(a, b)$ -tree  $T$  and a key  $x$  and want to divide it into two trees  $T_1$  and  $T_2$  where every element  $\leq x$  is in tree  $T_1$  and every element  $> x$  is in tree  $T_2$ ? How would you do this efficiently? What is the cost of this operation?