CS2040S: Data Structures and Algorithms

Recitation 3

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)

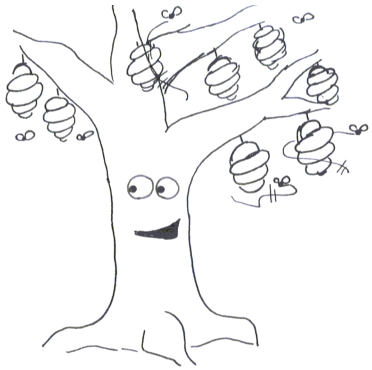


Image credit: Jeremy Fineman

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 learnt from Binary Search Trees (BSTs) that if we can design a DS which always maintains data in sorted order, then we can avoid the overhead to sort them every time before we search. However, as you have also learnt, unbalanced BSTs can be horribly 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!). In this week's recitation, we will look at **B-trees** which is another (very important!) type of 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 confuse yourself!

1 (a, b)-trees

Before we talk about B-trees, we have to first introduce its family (generalized form) which are called (a, b)-trees. In an (a, b)-tree, a and b are parameters where $a \leq b/2$. 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 ***"(a, b)-child Policy"***

- *The* ***root node*** *has at least* 2 *children and at most* b *children*
- *An* ***internal node*** *has number of children at least* a *and at most* b

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 for its keys are covered in its subtrees.*

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

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

Then:

- *For every key* w *in subtree* t_1 *(first child):* $w \leq v_1$
- *For every key* w *in subtree* t_{k+1} *(last child):* $w > v_k$
- *For every key* w *in tree* t_i *where* $i \in [2, k]$*:* $v_{i-1} < w \leq v_i$

Rule 3 ***"Leaf depth"***

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

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, a $(2, 4)$-tree is a B-tree (*FYI:* this is often referred to as a 2—3—4-tree). An example of a $(2, 4)$-tree is given below in Figure 1.

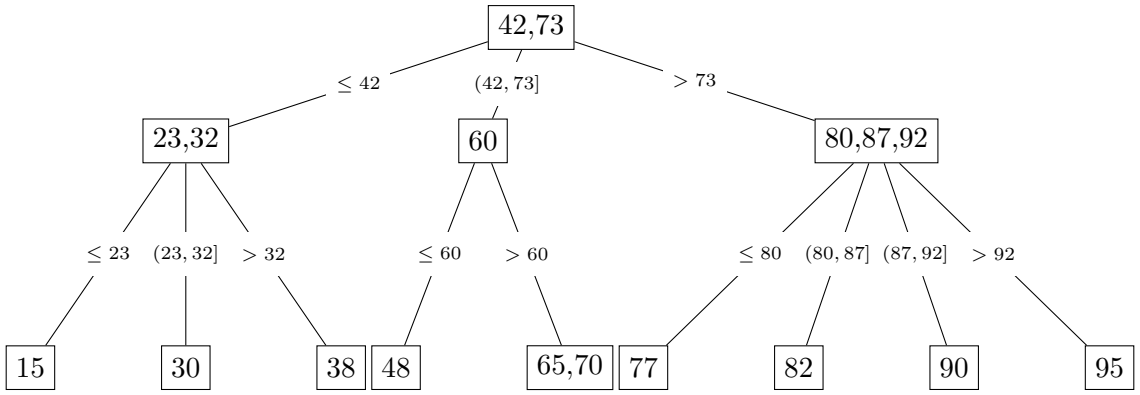


Figure 1: A $(2, 4)$-tree storing 18 keys. The range of keys contained within a subtree is indicated by the label on the edge from its parent.

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

Problem 1.b. What is the minimum and maximum height of an (a, b)-tree with n keys? What guarantees does this give us?

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?

1.2 `split` operation

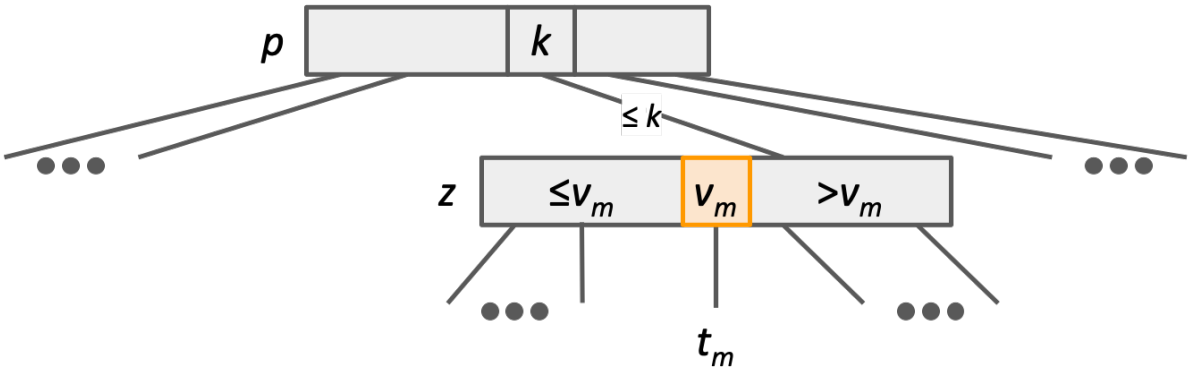
By definition, the three rules (Section 1.1) must hold before and after any operation that (a, b)-trees support. To ensure this, we need to take care of any potential violations that can occur across all operations. It should be obvious that `insertion` and `deletion` poses risks of violating the rules 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. A node z can be safely split if it satisfies the following conditions:

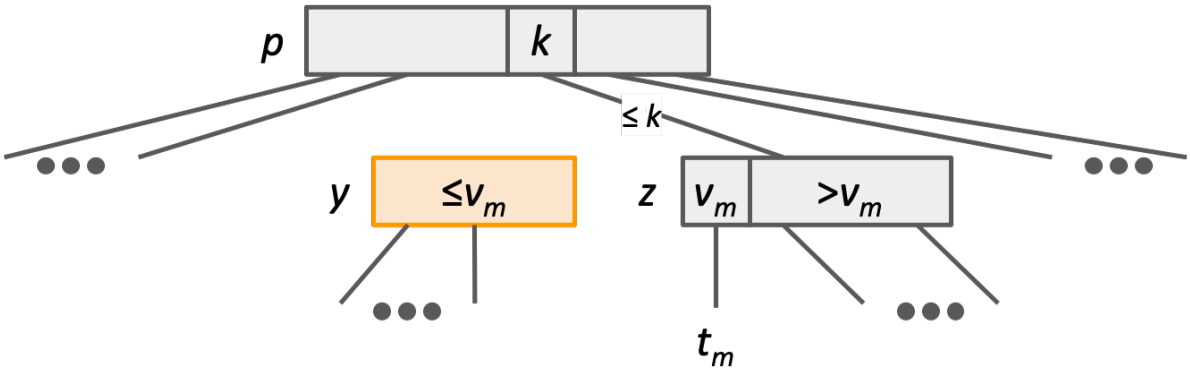
- z contains b keys
- z's parent contains $< b$ keys (doesn't apply if z is root)

A `split` operation redistributes out keys so as to resolve the violation of a node. Firstly, keys in the violating node z are divided into two parts using the median key (think partition step in QuickSort). This ensures that each part contains half of z's keys along with their associated subtree links. The median key is then inserted into z's parent. Specifically, split operation on a node z is outlined in the following algorithm:

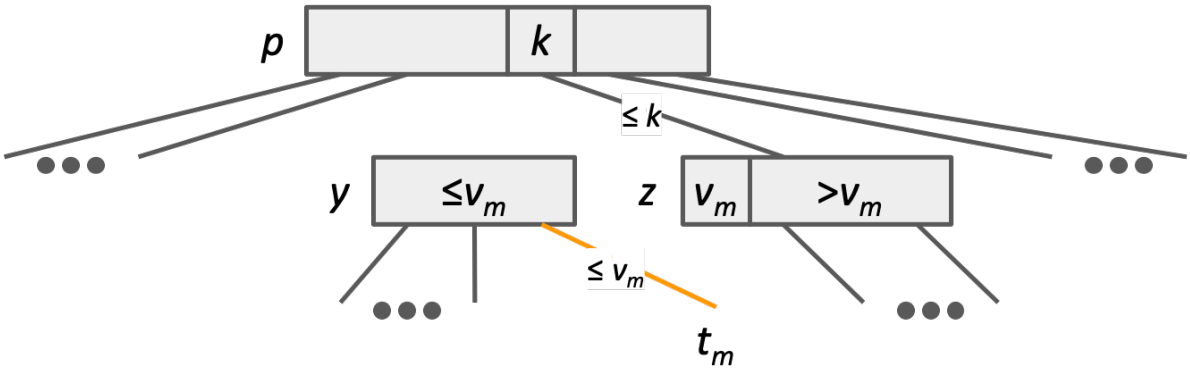
1. Find the median key v_m where the number of keys in z that are $\leq v_m$ is $\lceil b/2 \rceil$ and the number of keys $> v_m$ is $\lfloor b/2 \rfloor$. Notice that both $\lceil b/2 \rceil$ and $\lfloor b/2 \rfloor$ are at least a, by assumption (since $b \geq 2a$)



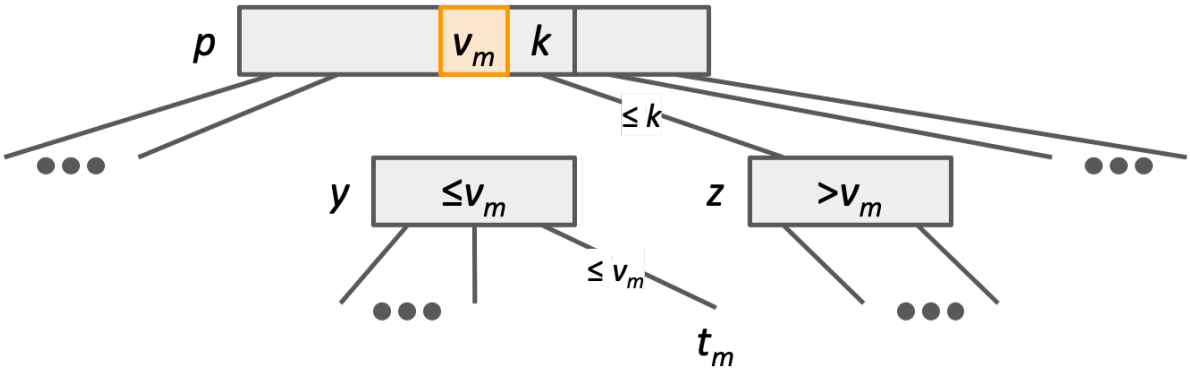
2. For each key $v_i < v_m$ and their associated subtrees t_i, move them from z into a new node y



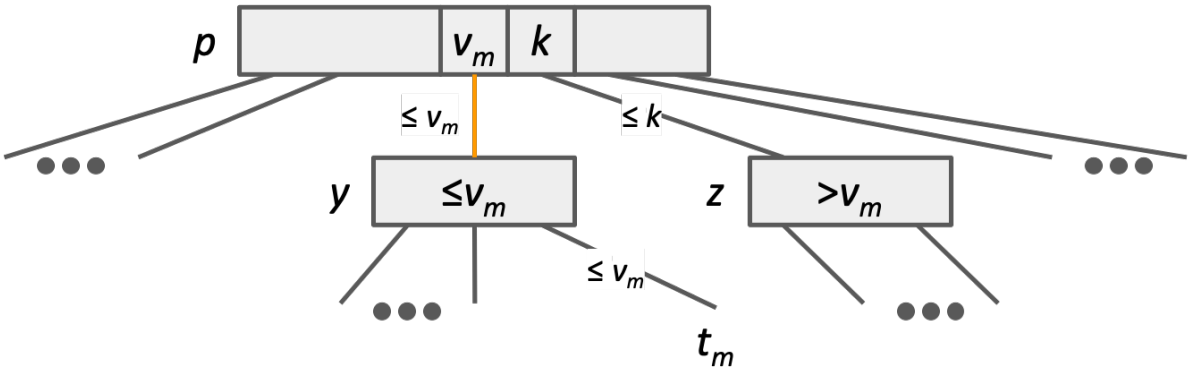
3. Since y will be missing a final child, move subtree t_m from z and add it as the final subtree in y



4. Move key v_m from z and to z's parent p (we know we can do this because p has $< b$ keys). 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



6. Return (y, v_m, z).

There is one corner case not covered by this split procedure — splitting the root. For splitting the root z, follow exactly the same procedure, creating a new node y and moving half the keys from z to y; however, before inserting the key and link into the parent, first create a new root node r; then insert the split key v_m as the only key in the key list at r, and add links to both y and z.

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

1.3 Insertion

Now it is easy to insert an item x into the tree:

1. Start at node $w = root$.
2. Repeat until w is a leaf:
   - If w contains b keys,
     (a) Split w into y and z
     (b) Examine the key v_m returned by the split operation
     (c) – If $x \leq v_m$, then set $w = y$
         – Else, set $w = z$
   - Else,
     (a) Search for x in the keylist of w
     (b) – If x is larger than all the keys, then set $i = k + 1$ where k is the number of keys in w
         – Else, find i such that v_i in w is the smallest key $\geq x$
     (c) Set w to the child linked to by t_i
3. Add x to the key list at w

Problem 1.e. 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.f. What is the cost of searching an (a, b)-tree with n nodes? What is the cost of inserting into an (a, b)-tree with n nodes?

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
If y and z are siblings and have $\leq b$ keys together, and v is the key separating them in their parent, then we can merge them together	`merge(y, v, z)`	1. Delete key v from the parent 2. Add v to the key list of y 3. Copy the key list and subtree link list of z to y 4. Remove z from tree
If y and z are siblings and have $> b$ keys together but $\leq 2b$ keys, and v is the key separating them in their parent, then we can share keys between them	`share(y, v, z)`	1. Merge y and z 2. Split them using the regular `split` operation

1.5 Deletion

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

1. Start at node $w = root$.
2. Repeat until w contains the key to delete or w is a leaf:
   (a)
   - If w contains (exactly) a keys, and z is a sibling of w, and w and z together contain $\leq b$ keys, then merge w and z
   - Else if w contains (exactly) a keys, and z is a sibling of w, and w and z together contain $> b$ keys (but $\leq 2b$ keys), then share keys between w and z. Update w if the key we are searching for moved to z.

   .

   (b) Search for x in the keylist of w. If x is not a key in w, then find i such that v_i is the smallest key $\geq x$ in the keylist at w; if x is larger than all the keys, set $i = k + 1$. Set w to the child linked to by t_i.

3. Delete x from the key list at w, if it exists in the list.

Problem 1.g. 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)

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

The important thing is that when you want to access a memory location in some block B_j, the entire block is copied into memory. You can assume that your memory can hold M blocks at a time. Transfering 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 1)

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.

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 1: Some typical numbers (taken from the Haswell architecture data sheet)

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

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 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?