Overview
In lecture, we saw that AVLs are efficient and useful when we need to keep our data ordered. However, the downside is that the amount of data we can store in an AVL tree is bounded by the size of main memory. When we have a large amount of data, we may not be able to load it all in the main memory. In this case, we want to use B-trees. B-trees are a versatile tree-based data structure, typically used to store large amounts of data in disks.

B-tree of Order m:
Rules of a B-tree of order m:
- All keys within a node are ordered.
- All non-leaf nodes have one more child than keys.
- Root can have: \[2, m\] children and \([1, m-1]\) keys
- All other internal nodes have \([\text{ceil}(m/2), m]\) children
- All non-root nodes have \([\text{ceil}(m/2)-1, m-1]\) keys

Exercise 1.1: What are the possible values of \(m\) if the following BTree is of order \(m\)?

\[
\begin{array}{c}
4 \quad 9 \quad \boxed{17} \\
1 \quad 3 \quad 6 \quad 8 \quad 24 \quad 25 \quad 33 \quad 35 \quad 56 \quad 80
\end{array}
\]

To get our lower bound, we look at the nodes with most children or most elements. We see that we have 2 nodes that have 3 children, and the most elements in a node is 2. Thus we need \(m\) to be at least 3 otherwise the nodes with 3 children/2 elements will not be legal.

To get our higher bound, we look at the node(s) with the least number of elements/children. In our example, we have a node with only 1 element (10), thus we need \(\text{ceil}(m/2)-1 \leq 1\). \(M=3\) and \(m=4\) both satisfy this condition, but \(m=5\) makes \(\text{ceil}(5/2)-1 = 2\) > 1, thus we can’t have \(m=5\)!

In conclusion:
\(m = 3\) or 4

Exercise 1.2: Which B-tree is a valid B-tree? (there is only one valid!) Why are the others invalid (what rules do they break)?

From left to right:
1) INVALID: all leaf nodes must be on the same level.
2) INVALID: since there is a node with 4 elements FGHI, \(m \geq 5\), but then the minimum number of elements is \(\text{ceil}(5/2)-1 = 2\), so we can’t have a node with just one element D.
3) INVALID: node JM MUST have exactly 3 children, not 2.
4) VALID: can be a B-tree of order \(m=4\) or \(m=5\).

Inserting Into a B-tree
Recall from lecture:
1. Always insert the new element into a leaf node.
2. If we have an overflow in a node after insertion; meaning it has more than the maximum number of elements allowed: we must split the node by throwing up the middle element to the parent node.

Given the B-tree below: we want to insert the following elements in this order \([16, 17, 18, 19, 20]\)

\[
\begin{array}{c}
4 \quad 9 \quad 12 \\
1 \quad 3 \quad 6 \quad 8 \quad 10 \quad 11 \quad 13 \quad 14 \quad 15 \quad 24 \quad 25 \quad 33 \quad 35 \quad 56 \quad 80 \quad 82
\end{array}
\]
**Exercise 2:** Suppose the B-tree above is of order 5, how will it look after inserting the elements? How many splits were performed?

```
4 9 12 15 18
/   |   |   |
1 3 6 8 10 11 13 14 16 17 19 20
```

This branch same as before

3 splits were performed

**B-tree Class Implementation:**
This part of the worksheet will provide an overview of the code structure of the `btree` class you will implement in the coding part of this lab. Each element in a `btree` is of type `DataPair`. Elements are ordered in the `btree` nodes according to the key in the DataPair. Each `BTreeNode` has one boolean member variable `is_leaf`, and two vectors: `elements` and `children`. Visually, a `BTreeNode` will look like:

**Exercise 3.1:** Let \( K_1 = 1, K_2 = 5 \), and \( K_3 = 10 \) in the above illustration of a BTreeNode. Suppose we are searching for the element with key = 7, which child pointer in the BTreeNode should we follow? Give its index in the `children` vector.

index 2 in children vector:

5 < 7 < 10,
So we need child between 5 and 10:
(1 is index of 5 and since 5 < 7 we need child on the right side of the 5, thus index 2)

**Exercise 3.2:** Suppose we need to split the BTreeNode above (so we will throw up the element \{\( K_2, V_2 \)\}), how would we split the children vector between the newly created BTreeNode after the split? Draw the resulting BTreeNode after the split:

Index of \{\( k_2 \, V_2 \)\} elem is \( i = 1 \),

Parent node:
- elements \{\( \{k_2,V_2\}\)\}
- Children vector of size 2, pointing to children 1 and 2:
  1. Elements= \{\( \{k_1,V_1\}\)\}
     - Children= frist two elements of the previous children vector
  2. elements= \{\( \{k_3,V_3\}\)\}
     - Children= last two elements of the previous children vector

As your TA and CAs, we’re here to help with your programming for the rest of this lab section! 😊