CS 225

Lab_btree : Belligerent B-trees

Welcome to Lab B-tree!

Overview

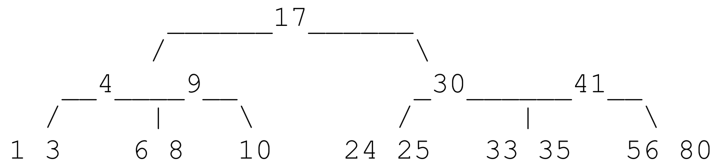
In lecture, we saw that AVLs are efficient and useful when we need to keep our data ordered. However, the downside is that the amount of the data we can store in an AVL tree is bounded by the size of main memory. When we have a large amount of data, we may not be able to load it all in the main memory. 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:

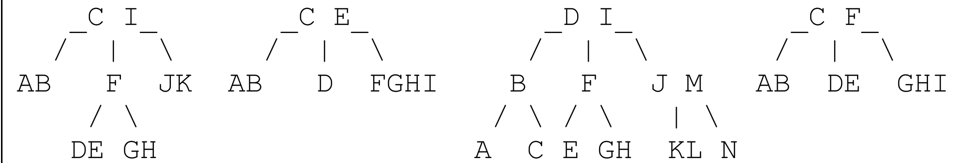
Rules of a B-tree of order **m**:

- 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 **[ceil(m/2), m] children**.
- All non-root nodes have **[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**?



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)?

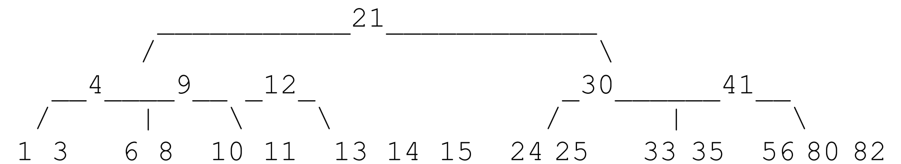


Inserting Into a B-tree

Recall from lecture:

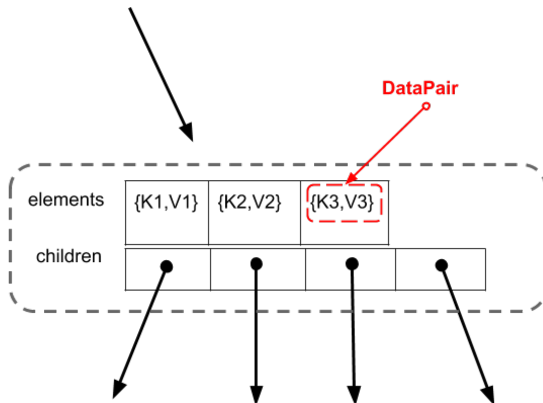
1. Always insert the new element into a leaf node.
2. If we have an ***overflow*** in a node after insertion; meaning it has more than the maximum number of elements allowed: we must ***split*** the node by throwing up the middle element to the parent node.

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



Exercise 2: Suppose the B-tree above is of **order 5**, how will it look after inserting the elements? How many 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 **K1 = 1, K2 = 5**, and **K3 = 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.

Exercise 3.2: Suppose we need to split the BTreeNode above (so we will throw up the element **{K2, V2}**), how would we split the children vector between the newly created BTreeNodes after the split? Draw the resulting BTreeNodes after the split:

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