TUTORIAL 9 B-trees

B-trees are like BSTs, except:

- Each node has a size property: the **number of keys** stored in the node.
- The tree has a **depth property**: all leaves must be at the same depth.

B-tree with degree *t = 1* are called 2-3 trees. *There is a special modification of 2-3-tree, called 2-3-4-tree, which allows to have 4 keys in a node.*

Let's consider B-trees with degree ***t* = 2** *(by Cormen 50+ is a good number for real-life implementations)*. Root should have 1..3 keys, other nodes: 1..3 keys.

Let's consider and an example at the whiteboard:

```
root node, 1..3 keys          ______17______
                             /              \
inner, 2..3 keys        __4____9_        20____30____41__
                       /     |   \      /    |    |      \
leaves, same level  1 2 3   7 8   12   18   24   33    56__80
```

Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom.

Let's ***search for key=7***.

1) Starting with the **root**, is key=17? No, let's look into subtree.
2) Is key > 17? No, use the **left subtree**.
3) Next node `(4, 9)`. Is key = 4? No.
4) Is key > 4? Yes, so **move to next key**.
5) Is key = 9? No
6) Is key > 9? No, so use subtree to the left of 9.
7) Next node `(7, 8)`. Is key=7? **Yes. Found.**

What about ***searching for key = 11***?

Starts the same but when evaluating the node `(4, 9)` we find that key > 9 and there are no more keys in that node to check. We therefore go to the right-most subtree. Then when we evaluate that key<12, we look to the left sub-tree of 12 which is a leaf. This tells us that 11 is not in the tree.

Insertion

All insertions start at a **leaf node**. Firstly, search in the tree to **find the leaf node** where the new key should be added. Insert the new element into that node with the following steps:

1. If the node contains **fewer than the maximum** legal number of keys, then there is room for the new element. Insert the new key in the node, keeping the node's keys ordered.
2. Otherwise the node is full (**OVERFLOW**), evenly **SPLIT it into two nodes so**:
    1. A single **median** is chosen from among the leaf elements and the new element.
    2. **Keys less than the median are put in the new left node** and values greater than the median are put in the new right node, with the **median acting as a separation value**.

Based on http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut04.txt, https://en.wikipedia.org/wiki/B-tree

3. **The separation value is inserted in the node's parent**, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage.

Example #1

Consider ***inserting 6*** into the above tree.

```
              ______17______
             /              \
        __4_____9__          20____30____41__
       /       |    \       /     |     |      \
    1 2 3   6 7 8    12    18    24    33     56__80
```

Remember that if a key is inserted **into a node with 1** (for root) **or 2 existing keys**, the insertion is easy. We just put the new key into the correct position so that the **order property** still holds. If the destination node has **3 keys already** then inserting the new node will violate the size property.

Consider ***inserting 5*** now. When we insert 5, new node (5, 6, 7, 8) has too many keys (**OVERFLOW**). We resolve the problem by doing a **SPLIT**.

In this case we form the node (5) and the node (7, 8) and give them the parent (6) – as it is a median. We insert the subtree.

```
   __6__
  /     \
 5       7__8
```

into node (4, 9) which was the original parent of (5,7,8). The final tree is:

```
              ______17_____________
             /                     \
        __4_____6___9_          20____30____41___
       /        |    |  \       /     |     |     \
    1 2 3       5    7 8  12   18    24    33     56_80
```

Example #2

Starting with an empty tree, insert the following keys into the tree:

```
5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 31
```

```
          (5)
---------------------------------------------------------------------
          (5,16)
---------------------------------------------------------------------
          (5,16,22)
---------------------------------------------------------------------
               (16)
              /    \
          (5)    (22, 45)
---------------------------------------------------------------------
               (16)
              /    \
          (2, 5)   (22, 45)
---------------------------------------------------------------------
               (16)
              /    \
      (2, 5, 10)   (22, 45)
```

Based on http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut04.txt, https://en.wikipedia.org/wiki/B-tree

```
---------------------------------------------------------------------
            (16)
           /    \
     (2, 5, 10)  (18, 22, 45)
---------------------------------------------------------------------
            (16 , 22)
           /    |   \
    (2, 5, 10) (18)  (30, 45)
---------------------------------------------------------------------
            (16 , 22)
           /    |   \
    (2, 5, 10) (18)  (30, 45, 50)
---------------------------------------------------------------------
          ( 5,  16 , 22 )
         /     |   |    \
      (2) (10, 12) (18)  (30, 45, 50)
---------------------------------------------------------------------
                16
               /   \
          ( 5 )     ( 22, 45 )
         /    \       /   |     \
      (2) (10,12) (18) (30,31) (50)
```

Deletion

There are two popular strategies for deleting keys from a B-tree.

1. Locate and delete the item, then restructure the tree, **OR**
2. Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring.

Let's consider ***first*** strategy. All possible cases for rebalancing can be found here.
Consider the following tree:

```
          ______17_______________
         /                      \
    ___4____6____9_      20____30____41___
   /      |    |   \      /    |    |      \
1 2 3     5   7_8  12    18   24   33    56_80
```

Delete 2

```
          ______17_______________
         /                      \
    ___4____6____9_      20____30____41___
   /      |    |   \      /    |    |      \
  1 3     5   7_8  12    18   24   33    56_80
```

Deleted 4

4 isn't in a leaf node, so we would need a key to replace 4 to remain subtree `(1, 3)`. We can find the **predecessor** (or successor) for 4 and swap the elements. In this case 4's predecessor is 3.

Delete 12

There is a problem with the size property. The node used to hold 12 will have no key (which is an illegal size). The problem is called **UNDERFLOW**, and such node is called ***deficient***.

Can we solve this problem by **BORROWING (TRANSFER)** a key from a sibling?

Based on http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut04.txt, https://en.wikipedia.org/wiki/B-tree

In this case the sibling `(7, 8)` has 2 keys so it can spare one. But notice that if we shifted 8 over to the node 12 we would have the following subtree

```
      _9_
     /   \
    7     8
```

which would violate the ordering property. So instead we **ROTATE**. We shift the 9 from the parent into 12's old position and the 8 from `(7, 8)` into the hole left from moving 9.

```
      _8_
     /   \
    7     9
```

NB this can be done from either sibling (left or right) so if we were to delete 5, we could rotate 3,4 and 5 or else 7,6 and 5.

Delete 24

Rotation in this case won't work because both direct siblings are 2-nodes (1 key). They have no extra keys to spare. If we borrowed from either of them, they would underflow. In this case we don't borrow but we **MERGE**. That's where the property of key count is important*: (**t-1**, … , 2*t-1)* keys per node.

```
                _______17_______________
               /                        \
         ___4_____6_____8_          __20_____30_____41___
        /      |     |    \        /     |       |       \
       1  3    5     7     9      18    ~~24~~     33       56_80
```

We combine the **node 24** with one of its 2-node siblings and the **key from their parent** which divides them. In this case, one choice is to combine **18, 20 and 24**. We delete node 24 from this and attach the combined node as the subtree of the parent. Here is the resulting tree:

```
                _______17_______________
               /                        \
         ___4_____6_____8_          ____30_____41___
        /      |     |    \        /       |        \
     1  2  3   5     7     9     18_20     33       56_80
```

Notice that we only need to merge when the sibling is a 2-node so the new resulting node is always a 2-node. Notice also that we could merge with either sibling.

In other situation, this can cause the parent p to **underflow**. When this occurs, we resolve the underflow by borrowing from or merging with a sibling of p.

Delete 60

Consider new tree example:

```
                _______31_________
               /                  \
          __6____             _____78 __
         /       \           /          \
     1_2_3        7_8     ~~60~~          100
    /  | | \     / | \  /  \          /   \
```

60 cannot borrow from 100. 100 is already a 2-node. 60 cannot borrow from `(7, 8)`. They are not siblings. So 60 is deleted and **100 (sibling) merges with 78 (parent)** and now the parent (78) underflows.

Based on http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut04.txt, https://en.wikipedia.org/wiki/B-tree

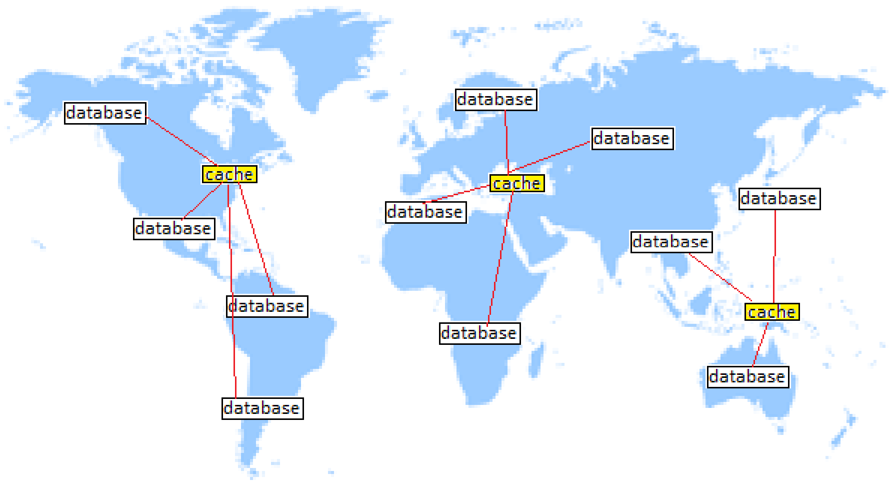
```
               ________31__________
              /                    \
         __6____                ___?___
        /       \                      \
   1_2_3         7_8               78_100
  /  | | \      / | \             /  |    \
```

Now **?** could borrow from node **6** (sibling) if it were a 3-node or 4-node. BUT because it is a 2-node 6 is merged with 31 (parent) resulting in:

```
              ?
             /
         __6_______31__
        /       |       \
   1_2_3       7_8      78_100
```

Since 31 came from a 2-node it now underflows. Because it is the root, it is simply removed from the tree and `(6, 31)` becomes the new root.

Discussion



Consider the map. Assume you are given an infrastructure of social network, where data is distributed in shards worldwide. You access data by **keys** throw the **caching** servers (some frequently used data, e.g. frequently viewed posts, popular users) which also stores **routing information**. Work with your neighbor(s) for 5 minutes to propose your version which data structures will you chose to enable this infrastructure. Present your results at whiteboard.

Practice

Take the implementation provided in http://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/BTree.java.html

Update it:

- Make node capacity tunable from constructor.
- Implement traversal method:
    - Public methods `traverse(visitor)`

Based on http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut04.txt, https://en.wikipedia.org/wiki/B-tree