Exercise 1

```
Provide an implementation of the B-Tree data structure. Note that you are required to
choose how to implement the Node structure. Recall that different choices may have
different performances both in terms of computational complexity and in terms of I/O
complexity. Motivate your choice.
Provide, moreover, a script that verifies all implemented functionalities.
```

We implemented the basic functionalities of a B-Tree.

Search

Since we have n nodes of b keys, te search time complexity is **O(log n/ log B)** . This can also be seen as *O(f(d) * log_B(n))* By hint we can imagine choosing at each step 1 out of B paths until an element is found or a leaf is reached. This hints us towards a logarithmic complexity.

Insert

We know that the time complexity of an insert in a generic B-Tree is

$$O((f(b) + g(b))logn/logB)$$

Where **f(b)** and **g(b)** both depend on the Node implementation. Note that f(b) is the time required to check if a key is in a node and g(b) is the time required to manage an overflow that might occur.

In our implementation we used sorted arrays (in concept, implemetation uses python lists) to store our keys.

So the time to search in a sorted array is **O(log(b))**, which is our f(b). The overflow management, g(b) takes time **O(b)** asyntotically, since a node need to be splitted and a new one needs to be created. The median finding is constant, **O(1)**.

Delete

O((f(b) + g(b)) log n / log b) This time, in the delete, g(b) is the time required to manage the underflow, by transfer or fusion.

I/O complexity

Until now we have measured in units of number of operations. This measure is appropriate if the whole data structure can be stored in *cache*. But for large data structures not all of the data can be stored in cache. In this case portions of the data structure must be stored in *external memory*, which is usually a disk. Accessing data on a disk is much slower then

accessing data in cache. For large data structures the appropriate measuring unit for time is number of *disk transfers*.

Memory in the computer is organized in a hierarchy. The RAM is generally called the *main memory*. The *secondary memory* is generally the external memory device, the disk. Our goal is to minimize the number of disk transfers needed to perform a query or update. Even for modestly sized data structures algorithms can be more efficient if we consider memory hierarchy and accesses.

Analyzing the performance of an ADT by counting the number of disk transfers is called *I/O complexity*.

Let d be such that a node can fit exactly in a disk block, so the block contains the d items and the d+1 pointers. So this means that for each level we need 1 block transfer. This also means that the I/O complexity is O(log_B(n)) for searches and inserts/delete. Note that to fit a node in a disk block we might want to avoid data structures that are implemented in a non-compact way, like linked lists.

Why we chose an ordered array to implement the BTree ?

An ordered array is a very compact data structure (and rather easy to use). Using maps and liked lists might result in a structure not as compact, that would not minimize the I/O complexity, since a node would not fit entirely in a disk block.

In general the underlying data structure must assure that f(b) and g(b) are not *much larger* than ceil((b - 1) / 2) or a, otherwise the advantage of height reduction is lost in the process of searching for a key or managing underflows/overflows.