Problem 1

Usage

In order to use the **BTree** implementation you have to install `numpy`. You can install it using the command `pip install numpy` or using the line `pip install -r requirements.txt` that will download and install the same version of the library used in developement phase.
To test the data structure we have provided a script test.py that tests all the functionalities of a BTree by calling the standard methods of the ADT **MutableMapping**.

Solution description

Data structure description and analysis of I/O complexity

We have chosen to optimize the I/O complexity instead of the computational one, since the BTrees are used to achieve low I/O complexity. In order to do that, our choice was to implement the **Node** of the BTree in such a way that every single node can be stored in just one block of memory.
Therefore, we have chosen to use a contiguous sorted array to store the items into the Node. Doing this we are sure that for each operation (Search, Insert and Delete) we have $O(log(n)/log(B))$ (where n is the number of the items in the tree and B is the maximum number of items that fit into a memory block) block transfers because every Node is stored in a single block even into the internal memory.
This second constraint is important for the I/O complexity. In fact if you use another data structure for the items, instead of a contiguous array, you can still have a Node that fits into a block of internal memory, but you could lose this property in external memory, because of **paging**.

In order to have a Node stored in a contiguous part of the memory we have encapsulated the Node into a numpy object of type `numpy.dtype`. Using a self-made dtype you can create a type that is similiar to a **C struct**. In fact, the Node is a structure that contains a numpy.array of a `dtype(key, value)` (**elements**), a numpy.array of pointers that contains the references to the children of the node (**children**), an attribute size that keeps the logic size of the Node (**size**) that is the number of items into the node) and the reference to the parent of the node (**parent**).
After having done this you can be sure that the Node is **C_CONTIGUOUS** checking the flag of the numpy object `npobject.flags("C_CONTIGUOUS")`. If this value is True, it means that the array is stored as a contiguous block of memory.

Compute the order

Now, when you want to use a BTree you can choose the types of the key and the value, for example you can use a BTree(int, int) or a BTree("U16", float64). Accordingly when we initialize the BTree we need to compute the order **d** based on the types that the user passes.
The **BLOCK_DIMENSION** can be changed into the file btree.py.
This computation is made by the formula:

$$Remaining\ dim = Block\ dim - node\ dim - size\ dim$$

$$Order = (Remaining\ dim + pair\ dim)\ //\ (pair\ dim + node\ dim)$$

Where *size_dim* is the dimension in memory of the variable size, *node_dim* is the dimension in memory of a reference (that depends on the python version that you are using) and *pair_dim* is the dimension in memory of a pair **(key,value)** of the type passed by the user.

Computational complexity

Since a sorted array was used to implement a node, the time complexity for the main operations is:

1. Add element : $O((log(d) + d) * (log(n)/log(d-1)))$
   where log(d) is the time needed for the research in a node (f(b)).
   (d) is the time needed to handle an overflow g(b) because whe need to do d insert operations.
   (log(n)/log(d-1)) is the height of the tree.
2. Remove element : $O((log(d) + d) * (log(n)/log(d-1)))$
   the only term that is different is g(b) that is the time to
   handle an underflow that depends if it is handled with a fusion
   $O(d)$ or a transfer $O(1)$.
3. Search an element : $O(log(d) * (log(n)/log(d-1)))$

- n is the number of elements of the tree
- d the maximum number of children of a node
- $log(d)$ the time needed to search an item in a node
- $log(n)/log(d)$ is the height of the BTree