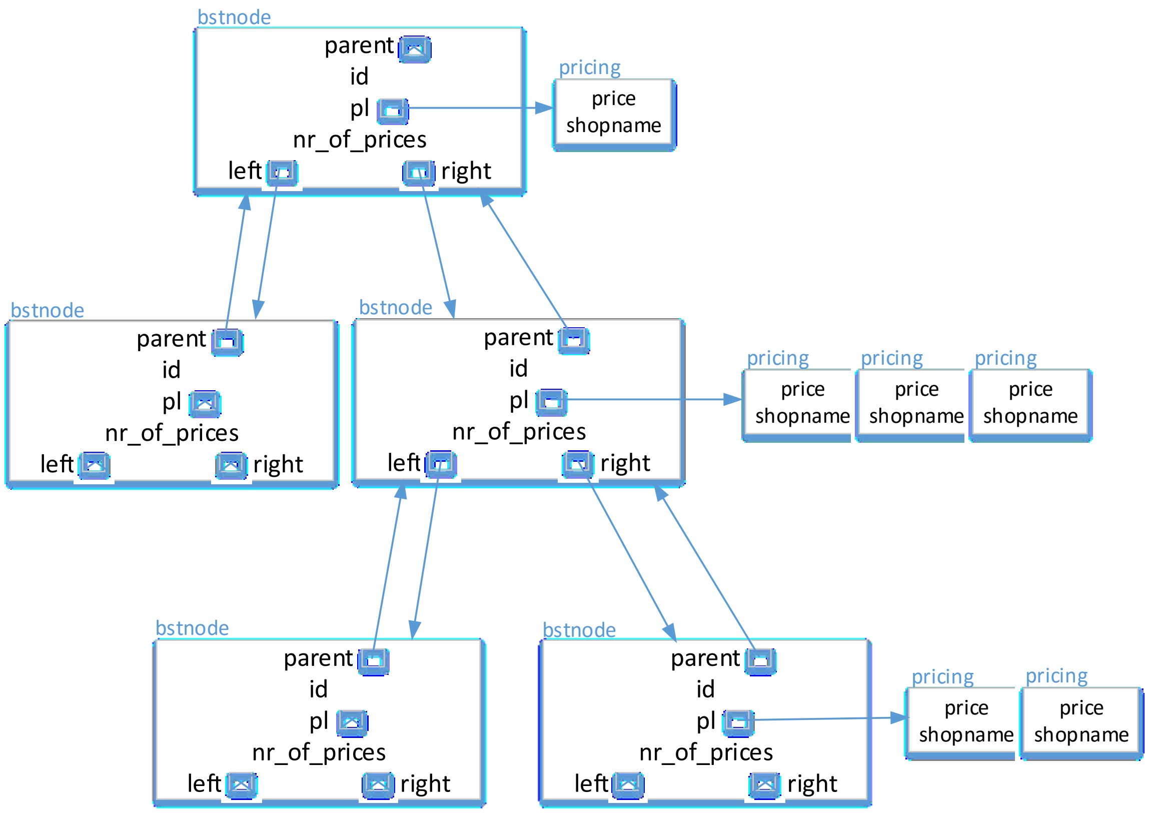
Exercise: Programming in C

In this exercise we will develop a system which can store the current price of products (identified by their barcode), and does this for multiple supermarkets. Since we wish to search quickly for a product, we chose to store the products in a binary search tree, sorted by their unique barcode. As value we keep an array (with elements in this array stored in order of adding them – a new price gets stored at the end of the array) of on the one hand the current price of this product, and on the other the name of the supermarket where you can get that price. The array (and associated memory) is extended whenever adding elements, or shrunk whenever removing elements. Schematically this data structure can be pictured as follows:



Figuur 1 – Binary Search Tree for keeping track of product prices

We will use the C language to implement this.

In PricingBST.h you will find predefined structures (pricing and bstnode). For simplicity we assume a barcode can be represented as an integer (id). *pl* denotes the pricelist of a node. In Main.c you can find test code.

We ask you to implement following functions:

```
bstnode* createNode(int id)
```

- This function creates a *bstnode* with given *id* and returns a pointer to the created *bstnode* or NULL if errors occurred.

`void freeNode(bstnode** node)`

- This function frees all memory taken up by the bstnode that is given as parameter. Note that this does not free any bst nodes linked to this node.

`void freeBST(bstnode** root)`

- Function which recursively frees the binary search tree with given root.

`void displayNode(FILE* output, bstnode* node, int treedepth)`

- This function prints the properties of the bstnode that is given as parameter, to file *output* (note that this will be used to pass standard output pointers for printing to the console – see main.c for examples). Parameter *treedepth* keeps track of at what depth we currently are in the binary search tree. Example output given below:

```
BSTNode at address: 006D6880 at tree depth: 2
Left child: 00000000
Right child: 00000000
Parent: 006D68C8
ID: 239876
        Pricelistings:
        Price: 20.300000
        Shop: Colruyt
        Price: 22.800000
        Shop: Delhaize
        Price: 21.700000
        Shop: Carrefour
```

`bstnode* findnodeBST(bstnode* root, int id)`

- This help function searches for a product with given id in the binary search tree with given root. If found a pointer to this bstnode is returned, if not found we return NULL

`void insertnodeBST(bstnode** root, int id)`

- Inserts a new node with given ID to the BST with given root (take into account that the tree can be empty). If a product with given id is already in the binary search tree, nothing needs to be done and you can exit the function.

`void updatePrice(bstnode* root, int id, char* shop, double price)`

- This function performs an update of the price (parameter price) in the supermarket (parameter shop) of product with barcode id in the binary search tree with given root. If not product with such a barcode is present in the binary search tree, the function does nothing. If no price of this store is known yet for this product, then price and store name needs to be added to the array of price / shopnames of this product (take a deep copy of the shop name). If a price for this store / product combination is already known, then update the old price with the new price.

```
void displayBSTInOrder(FILE* output, bstnode* root, int
treedepth)
```

- This function prints recursively and in order of small to big (in terms of barcodes) all nodes of the binary search tree with given root, to file output. Parameter treedepth denotes at which depth we currently are in the binary search tree (to keep track of during the recursion).

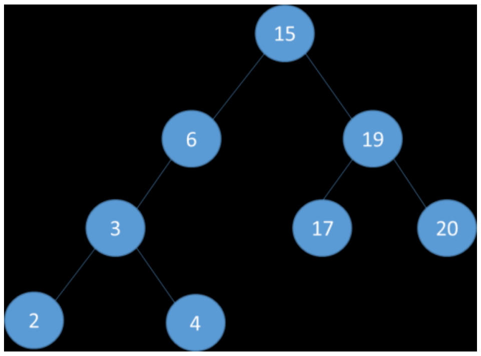
Now we implement two helper functions to assist in supporting removing elements from the binary search tree:

```
void swapNodeData(bstnode** node1, bstnode** node2)
```

- This help function, to be used in `removenodeBST`, swaps all data (exceptions are the parent, left and right pointers) of two bstnodes.

```
bstnode* findSuccessorNode(bstnode* node)
```

- This help function, to be used in `removenodeBST`, finds the node that follows the node given in the parameter and returns a pointer to that node. If no node follows the given node (i.e. de given node has the highest barcode / key in the binary search tree) you return NULL.
- Search for a node following a given node, can be done as follows:
  - If the node has a right child, then the follower of a node is the smallest (i.e. leftmost) element from the binary search (sub)tree with root that right child. E.g. follower of 15 is 17 in below example.
  - If the node has no children, then the follower of that node is found by following the parent-pointers until the start-node (parameter node) no longer is a member of the right subtree of the parent. Example follower 4: you follow the parent-pointer to node 3, which has 4 as child in the right subtree. You follow the parent-pointer from 3 to 6. This does not have 4 as a child in the right subtree, so you found the follower. 2nd example: follower of 17 is 19.



```
void exportToFile(bstnode* root, char* outputFileName)
```

- This function prints the binary search tree with given root in order to the file with given output file name. Use already implemented functions (displayBSTInOrder)

```
void printFile(FILE* stream, char* fileName)
```

- This function prints the file with given filename to given output stream (`stdout, stderr` are also `FILE*`). This function is used to check the correctness of `exportToFile` (see main).
-

```
void removenodeBST(bstnode** root, int id)
```

- This function removes the product with given id from the binary search tree with given root, and frees all memory taken by this node. Follow the pseudocode below:

```
Find given node in the BST
    If not found, return
    If found
        If node has no children
            Remove node
        If node has 1 child
            Let the child take the position of the to be removed
            node and remove the node
        If node has 2 children
            Find the follower of the to be removed node
            (findSuccessorNode).
            Swap the data of the to-be-removed node with the
            follower node (swapNodeData).
            call removenodeBST on the follower node.
```

In main.c these functions are called allowing you to check your implementation (inclusive exceptions). Use this to test if your implementation works correct. As development environment we used Visual Studio Community edition (a free version of Microsoft's development environment).