**Programming Homework 3: B trees**

# introduction

The library's book lending system stores data on all books in the library. For each book, the system saves its catalogue number (int), location on which shelf the book is placed (int), and a Boolean value indicating whether the book is currently in the library (true) or borrowed (false).

All of the above information is stored in a balanced AVL binary search tree (as learned in Data Structures A) organized by the catalogue number (int) of the books. Every time you search/borrow/return a book, you search for the book in a tree and find/update the information as required.

The search operation requires loading (many) vertices (nodes) from computer memory - which is relatively expensive. Therefore, you were asked by the library management to help upgrade the tree search operation (and in any case all other operations) by reducing the number of vertices that must be loaded from memory.

# Requirements

To do this, you need to rearrange the data in a B tree of degree m (m is an odd number). The tree will support insert and search operations.

## Exercise files

The following is a list and description of the files given to you:

* main.cpp: File containing the main program: You need to add your details at the top of the file.
* LibraryDataStructure.h: An abstract class representing the abstract data type
* AVL.h: The class that represents the old data structure (AVL tree) and supports the abstract LibraryDataStructure data type.
* Library.h, Library.cpp: A class that represents the library. The class receives a data structure (old or new) and uses it to manage the library's operations (borrowing/returning books and printing the contents of the data structure).
* File with a list of books data to insert into the data structure

The purpose of these files is to assist you and save you problems with the output format, but the given parts can also be changed, as needed.

In addition to the data files, you will need to implement and submit the following classes:

* Book.h, Book.cpp: A class representing a book
* BTree.h: A class representing the new data structure B-tree and supporting the abstract data type LibraryDataStructure

In this exercise you have to:

1. Build a class called Book to represent a book
2. Build a class called BTree to represent a B-tree of degree m and insert the books into it according to their catalogue numbers
3. Compare searching for a value in tree B with searching for the same value in the AVL tree that contains the same values (the original tree used in the library). The comparison must show that searching the B-tree does require access to fewer vertices.

Here is guidance and details for each stage of the work:

**Stage 1**

Building the **Book** class to represent a book

The fields in the class will be:

* Book catalogue number (int)
* Book shelf number (int)
* Boolean field indicating whether the book is in the library or borrowed (bool)

Add at least the following functions to the class:

* A constructor that receives a catalogue number and a shelf number and initializes a book accordingly. (Assumes the new book is in the library).
* getKey: A function that returns the catalog number of the book (the function is required for the operation of the old and new data structure)
* borrowBook – a function for borrowing a book –

The function takes a catalogue number, if the book is in the library, the function makes its status borrowed and prints the message in the following format:

*Successfully found the book 0 on shelf 4 and borrowed it*

And if the book is already borrowed, the function prints the message in the following format:

*Error! The book 0 is already borrowed!*

* returnBook - Book return function –

The function takes a catalogue number. If the book is borrowed, the function makes its status "in the library" and prints the message in the following format:

*Successfully returned the book 0 to shelf 4*

And if the book is already in the library,the function prints the message in the following format:

*Error! The book 0 is not borrowed!*

* comparison operators to compare two books as needed (comparison by catalogue number field). (>,<,>=,<=,==,!=)
* operator << Input of book data by format: Catalog Number Shelf number (New book: available=true). An example of an input line that you can enter to for a new book

*0 4*

* Operator >> Print the book data on one line according to the following format: catalogue number, shelf number and word **available** if available in the library or **borrowed** if the book is currently borrowed. For example:

{0, 4, available}

**Step 2**

Construction of a class called BTree to represent a B-tree of degree m.

1. Since the type of records stored in the tree is not known in advance, we define the class as template. In practice, we will use the tree to store objects (**records**) of type **Book**:

template<class T>

class BTree

{

private:

class BNode We define an internal auxiliary class to represent a single vertex (node) in a tree.

{ All of it can be set to public

public:

};

public:

};

1. Defines the BNode internal class that represents a single vertex in a tree

The fields in the class will be:

* m – The degree of the tree. (Odd number, no need to check).
* Array of objects (records) of size m. the actual size, m, will be determined at runtime, when we define a new tree. Pay attention! In a B tree of degree , each vertex has at most records, but since insertion may have a temporary state of a vertex with records that causes the vertex to split, we will reserve more space in the array.
* An array of length m+1 of pointers to children (see note in previous bullet point)
* Pointer to the parent vertex
* The number of records stored at the actual vertex
* Number of sons that the vertex actually has

Add at least the following functions to the BNode class:

* A constructor that accepts (as a parameter) the tree degree (m) and initializes all fields accordingly.
* destructor
* isLeaf – A Boolean function that returns true when the vertex is a leaf (the number of children is 0).
* insert(T record) – A function for inserting a record to the appropriate place in the records array
* remove(T record) – A function that deletes a record from the records array and moves all records accordingly.
* printKeys()–function for printing the keys of the records stored at the vertex (note! this function will use the getKey function we defined in **Book**) according to the format:

1. After building the BNode class the fields in theBTree class will be:

* m– Tree degree (odd number–no need to check this, default 5)
* root – BNode pointer to the root of the tree.

Add to the BTree class the following public functions:

* A constructor that takes as a parameter the tree degree (m) and initiates the root to point to nullptr.
* destructor
* insert (T record) – A function for inserting a record to the appropriate place in the tree.
* inorder()– A function that scans the tree and prints the keys of the records stored at its vertices in order. To illustrate the scan, the function must print the vertex keys each time it visits it.Pay attention! In B-tree, we will visit each inner vertex more than once!Therefore, an inner vertex will be printed more than once.
* search(int key)–A function that takes a key to a record and returns a pointer to the record (of type T\*) if it exists or a nullptr if not exists. The function prints out how many vertices it scanned during the search in the following format:

The search involved scanning 5 nodes

Guidance-

You can add recursive auxiliary private functions to the BTree class– for example:

* void clear (BNode\* current) – A recursive function that performs the tree deletion using the post-order method
* void inorder (BNode\* current) - A recursive function that prints the keys of the records stored at the vertices of the tree in order.
* BNode\* findAddNode (BNode\* current,T record) - A recursive function that finds the vertex in the tree to which the new record should be added. If the tree is empty, it returns a nullptr.
* Void split (BNode\*current) – A recursive function to update the tree structure if a record is inserted to a vertex that is already filled.
* T\* search (BNode\* current,int key,int& counter) - recursive function to search for a record given its key. The last parameter will be used to count how many vertices we scanned. Note that it is passed by reference.

**Step 3**

Use the main function provided to check that your implementation is correct and to make sure that searches require scanning for fewer vertices than using AVL.

**Good luck!!**