Problem

The C++ programming language is widely used in database management systems server code for its modernity, ease of error handling, and execution speed. For example, C++ is used in Microsoft's MySQL, MongoDB, and SQLServer. One of the most important data structures in DBMS is the B-tree or B+tree for indexing. A B-Tree is a self-balancing tree that maintains sorted data and guarantees searches, sequential access, insertions, and deletions in log time. Like a binary search tree, for any given node, the left child is less than the node and the right child is greater than. However, in a B-Tree, each node contains multiple elements and children pointers. Multiple elements in a node makes B-Tree optimized for disk retrieval, which allows DBMS servers to efficiently navigate table rows based on index keys. B-Trees are also good for locality resulting in a relatively high cache hit ratio, suggesting it is an efficient structure for main memory use cases as well. Despite its universality in DBSM, there is no standard template library container for B-trees or any tree. The aim of this project is to implement a B-tree and measure its efficiency against associative containers in the STL. Additionally, we will use the B-Tree structure to index a database and create a SQL interface.

Background

There are a couple reasons why B-Trees are not included in the STL. Containers are defined by the interface and observable characteristics, not the underlying structure. Associative containers such as `std::map` and `std::set` implemented internally as red-black trees, but the tree functionality is not exposed. So although trees are used internally to optimize key lookup containers, there are no STL tree containers. It is difficult to have a general tree container that satisfies all use cases. For example, trees can be traversed in many different ways. Since STL containers are externally represented as "sequences", one interaction mechanism would need to be determined. Other issues include the number of children per node, pointer overhead, and algorithms. Would a tree template include BFS or DFS for searching? Because of these complexities, it has been seen fit to allow programmers to implement their own B-tree based on their needs.

When there is a large amount of data, data needs to be stored on a disk, not main memory. At an average of 15 milliseconds, disk access time is relatively high so it is important to reduce the number of disk reads. By comparison, access time to RAM is about 5 microseconds. The B-Tree structure is optimized for this exact purpose because the number of elements stored on a node (determined by the order of the tree, *m*) can be correlated to disk block size. Since B-Trees are short and wide, the number of nodes, and thus blocks, to visit are reduced when searching for elements. This means typical B-trees have a single digit height even with millions of keys.

Goals

Our goal for this project was to build a skeleton for a possible btree container class, and understand if that container could outperform current STL containers, like a map, vector for example, for accessing and maintaining an ordered sequence. We believe that a structure like this is important to the c++ STL container interface because of the way that C++ has been used to build open-source databases, which rely on btrees and b+ trees in order to store data within that database.

Therefore, our primary goal was to design a btree that could maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time.

In order to demonstrate the way that a btree can be used in c++, we also designed a user interface to support the main methods of the b-tree that would allow a user to query the data that is ultimately stored in the btree in-memory.

Non-Goals (i.e. things that we wanted to do but couldn't because of time)

B-Trees have been extended into a B+Tree for DBMS. B+trees are essentially a B-Tree with an additional level of linked leaves with copies of keys and data (or data pointers). The linked nodes allow ordered access of the records and guarantee all data pointers are present only at the leaf nodes. Moreover, this structure has simpler inserts and deletes. Creating a B+Tree was out of the scope of this project because of time constraints and complexity.

In designing our SQL interface, we needed a parser to be able to read the command line arguments that the user uses to interact with the database table constructed from the input file. While this does check for some simple errors like missing primary key values, or a few clear syntax errors, it isn't robust to deal with all of these errors. To be able to use the query tool effectively, the user must follow the syntactic instructions as specified.

The `SELECT` function of the interface is limited by primary key. It is possible to allow lookups for each column if a btree was instantiated for each column, with that column item as the primary key. Since the primary focus of this project was on the btree template and the efficiency of the data structure in comparison with other containers, additional btrees were not implemented for each column.

The C++ Standard Library has three sets of design requirements for containers: container interface, allocator interface, and iterator requirements. We provided functionality needed following STL style to implement a btree with insert, search, and delete functionality. The iterator and allocator interface requirements were pushed to V2.0 because of time constraint. Details about the iterator design issues are discussed in the next section.

Design

Containers in C++ are defined as "a holder object that stores a collection of other objects… and manages the storage space for its elements, and provides member function to access them". Following that definition, it made sense to us to look at the way that container classes are defined to them ultimately design our own interface for our btree.

A key difference between our btree container class and the other current STL containers is that the former contains pointers to another container i.e. "nodes" that then point to the information that is stored in the structure, whereas the latter stores the elements in the container itself. Therefore, the process of designing a tree relies on the design of these nodes themselves.

As we were creating this structure with the idea that it would be used for a database implementation, it was clear that each node in the tree has to contain a "key", i.e. a value that is used to maintain a sorted order in the tree, and then an index which points to other data associated with that key that doesn't necessarily need to be stored within that tree. Separating the key from the rest of the data was a design decision that we made in order to minimise the number of bytes of information that were being stored in that tree (especially as they were all allocated on the heap) and as they hold no importance to how the tree maintains its structure.

The difference between a btree and binary tree is that a btree can contain more than 2 children, and store more than 1 key of information. This then further informed our design decision to include other containers within our btree in order to store that information. In fact, for each node there are vectors allocated within that node that store all the different keys of the tree, and another vector that is referred to as children that store pointers to other nodes that are further down the tree. The choice of container had to take into consideration the role of indices between the keys and children elements because of the sequential order of the tree. For example, if there is a key at index i in the keys structure, all keys in the node at index i + 1 in the children structure had to be greater than the value of the key at index i. For that reason, the choice of the container had to be able to maintain and track indices of values, which led to arrays, lists and vectors as viable options for that structure. The next consideration to choosing this structure was that while there are order properties to btrees that allow it to be kept shallow (as discussed above), the number of elements within the key and children structures at any moment is not constant and has to change. To avoid memory access and reallocation issues that come with arrays, and the complexity of linear search to find elements in a list, we decided to use vectors as the containers that would hold the key and children values in a node.

Like current STL containers, we wanted to allow for the most flexibility in the number of types that are supported as elements within that class, and so we followed their example of implementing the container as a class template. This means that when a btree is constructed, it must pass in a specified type. However, one clear issue with creating a template class is that because it needs to keep the order sorted, in its construction it must also be passed a comparator function. The tree also requires a print function in order to meaningfully present the data in the tree. This actually follows from a similar instantiation of an ordered map in c++, where for user-defined types a comparator struct must be passed into the constructor of the map. The specified type of the btree is also passed to the constructor of all subsequent nodes of the tree, to specify the type of keys that are being held in the node. It is important to note that the type of the tree cannot be changed once constructed, and that follows logical sense as you cannot compare data of different types.

The STL conventions for a container include the following design patterns that allow each container to work with the generic algorithms class in c++:

1. begin & end
2. size
3. insert
4. clear
5. erase

To begin the design of our btree, we focused on the following 3 methods:

1. insert
2. erase
3. search

While these methods use iterators to perform the majority of these methods in the STL container classes, for time sake, we decided not to implement explicit iterators that could be called from outside the tree but rather focus on developing the actual algorithms themselves that allowed for these operations to be performed correctly. This is something that we hope to be able to implement in a V2 of the btree.

Another key design decision that we made is that the btree will only take an element of the specified type as an input, rather than a node itself. This is because of the way that the tree rebalances itself, in which it splits and merges together nodes depending on the value of the key being inserted to ensure that the constraints of the tree are maintained. This also differentiates our btree container class to the standard STL classes which rely on iterators for their insert and search methods. As a btree iterator would ultimately only point to nodes of a tree, rather than the actual values that the node holds, it would require a more complex implementation that would rely on an iterator of a node as well.

The other side of the design problem was a user interface that would ultimately be able to interact with this btree in order to easily query the data that the btree holds. The user is first prompted to input a file, which will then be parsed and converted into a structure that can be sequentially sorted based on the first column value. The values in the first column become the keys that are then stored in the nodes of the btree. For each insert into the btree, a unique index is created and stored with the key in the node in a struct called BNodeKey. That index then becomes the key in a map that is used to store the values of the other columns in order to minimise the amount of information that is stored in the btree which is redundant to its structure. The use of this map ultimately replaces the b+tree that would otherwise store the other column values. In addition, as we built this interface for a datafile storing the top hit songs per fortnight, our current parser expects a datestamp as the primary key for the column, but this is something that we are hoping to change in the future. This then means that the btree implemented by the database user interface will construct a btree using a `chrono::system_clock::time_point` as its type.

To be able to interact with the data, we designed a command line interface that follows SQL syntax as shown below:

```
COMMANDS:

  1. SELECT <PK>, ..., <columnN> FROM table [WHERE <PK> = x]
  2. INSERT (<PK>, ..., <columnN>) (<PK>, ..., <valueN>)
  3. DELETE WHERE PK = x
```

In a usual SQL database the user is able to query the table based on any of the table's columns; however, for time constraints we implemented a database that is only queryable on its primary key (or first column of the table) because that is how the data is sequentially sorted in the btree. This means that the primary key must be in the SELECT, INSERT, or DELETE command, but it can also retrieve other column information if specified in the table using the map container that uses the index of values in the btree. We believe that following this SQL syntax would make it easier for a user who may have prior knowledge of SQL to more easily interact with our own database.

Results

The following graphs show the insertion and search times for our B Tree container implementation, map and set. The STL map and set are built using red-black trees. This type of tree uses three pointers and a bit of information per element, whereas a B Tree stores a number of elements per node, which reduces the memory allocation of pointers. Furthermore, usually the performance of these different algorithms is based on the number of cache misses, which are extremely expensive and costly. Due to the simplicity of our comparison function (simply a less than), and the low locality of reference of the node-based structure, our container saves on performance time in comparison to a Red-Black tree, which is more prone to cache-misses. This is clear in the following graphs, as our search time was faster on the whole than map and set and our insert time was fairly similar.

Figure 1: B Tree vs. Map vs. Set Insertion

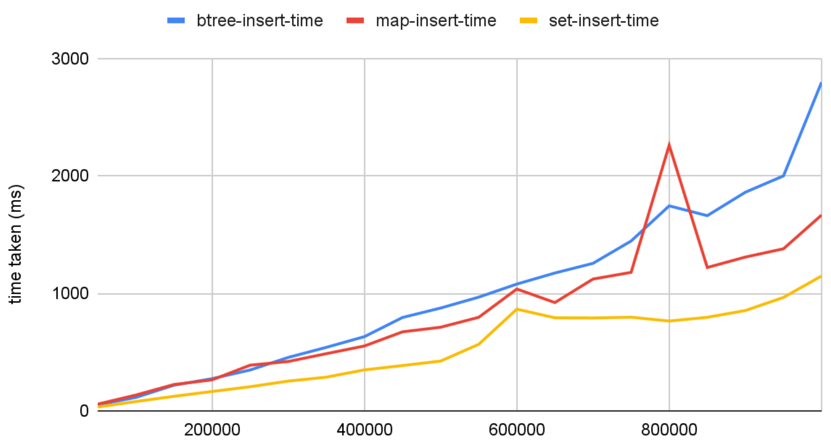
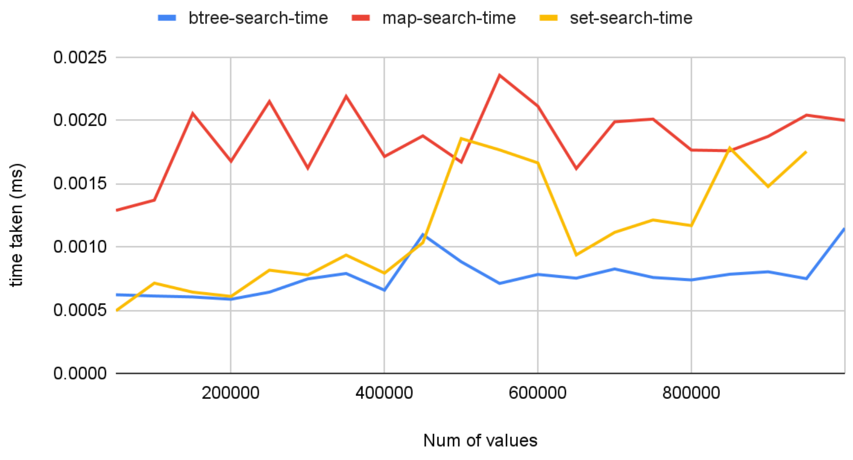


Figure 2: B Tree vs. Map vs. Set Search Time



Conclusion

Alexander Stepanov, the creator of STL said in an interview that "If I were designing STL today, I would have a different set of containers. For example, an in-memory B*-tree is a far better choice than a red-black tree for implementing an associative container." We set out to build a B Tree container to better implement different database functions such as search and insert. However, our results prove that beyond purely a container for a database, this container improves on search time and can take up less memory than the current map and set STL.