B-Tree Implementation in C++

Roxanne Farhad Carrie Hay Kellar Serena Killion

Design in C++ Final Project

Motivation

- C++ is widely used in database management systems
- B-Trees are one of the most important data structures for DBMS
- No standard library container representing B-Trees

Why is there no STL tree container already?

- Containers defined by interface and observable characteristics not underlying structure
  - `std::map and std::set` are internally implemented as red-black trees but tree functionality is not exposed

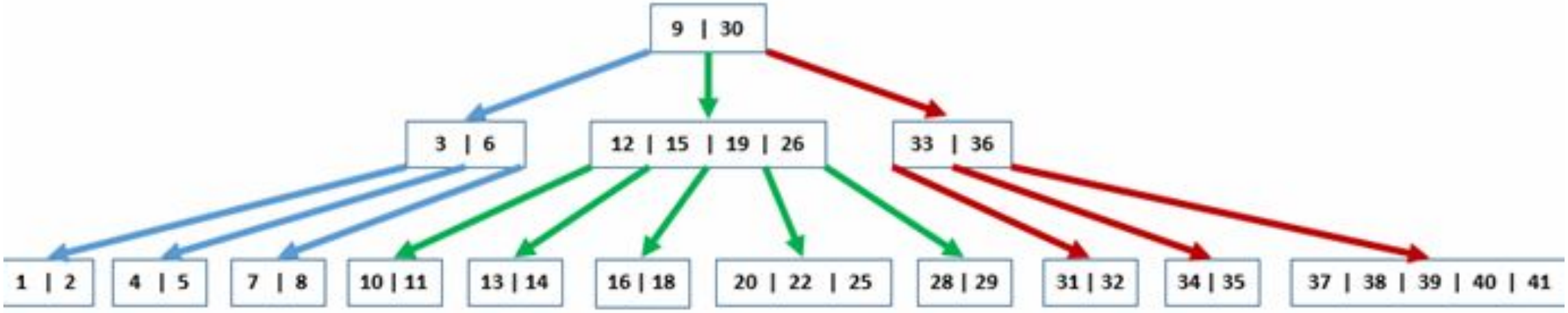
- Difficult to have a general tree container that satisfies everyone
  - How to decide on iterator?
  - Is the number of children per node fixed?
  - How much pointer overhead per node?
  - Which algorithms to include?

Why B-Tree?

- specialized m-way search tree which is a BST with multiple elements in a node
- sorted data structure is optimal for searching, inserting, and deleting
- get all keys using inorder traversal
- natural to find closest lower or greater elements or range queries
- all operations guaranteed to work in log time
    - hash tables has constant average time, but table resizing is costly

Type	Insertion	Deletion	Lookup
Unsorted Array	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(n)	O(log(n))
B-tree	O(log(n))	O(log(n))	O(log(n))

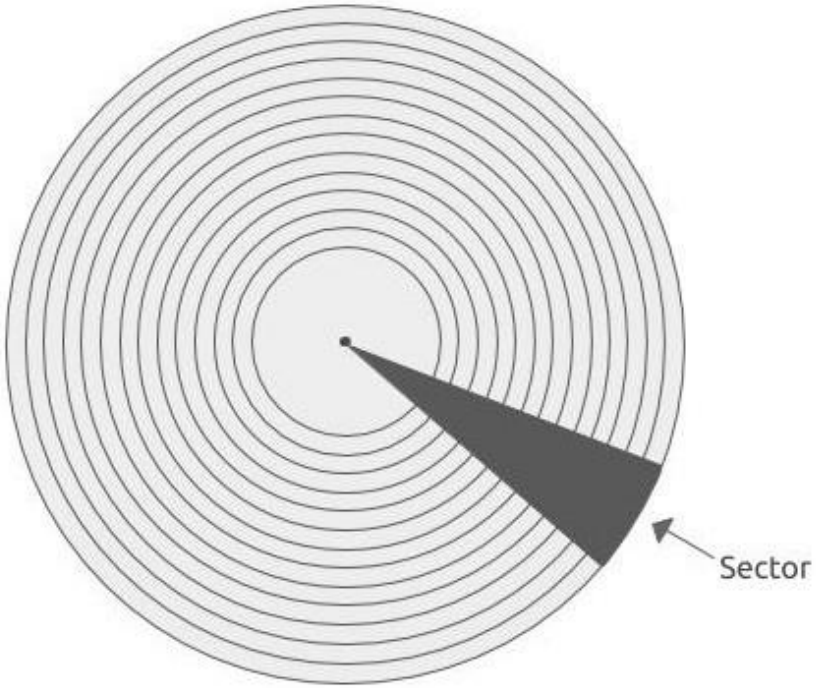
Properties of B-Tree



- Ordered, keys within node are ordered too
- Non leaf nodes must have n-1 keys and n child pointers
- All leaf nodes are at same level
- All non-root nodes have between [m/2]-1 keys and 2m-1 keys
- Non-leaf root node must have at least 2 children

Application to Database Indexing

- B-Trees are optimized for data stored on disk
- Average read operator of a sector: 15 ms
- Node size usually correlates to disk block size
- Reduces number of disk risks ➜ faster access



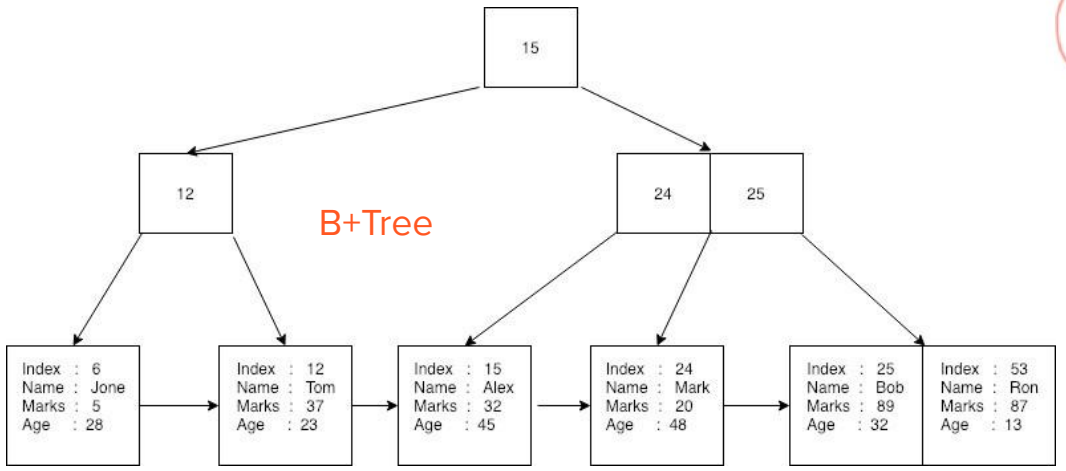
From a practical point of view, B-trees, therefore, guarantee an access time of less than 10 ms even for extremely large datasets.

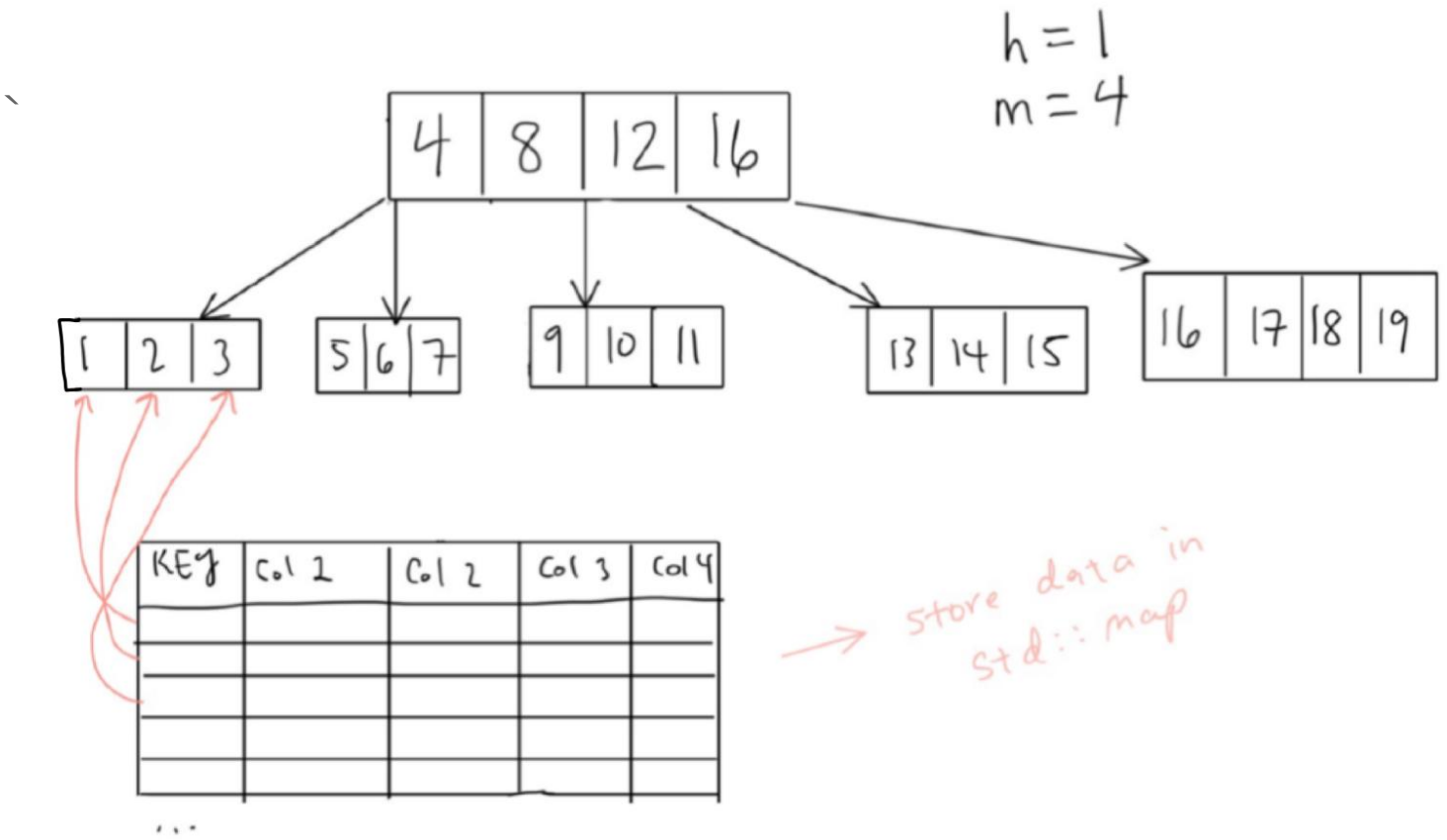
—Dr. Rudolf Bayer, inventor of the B-tree

Our Implementation

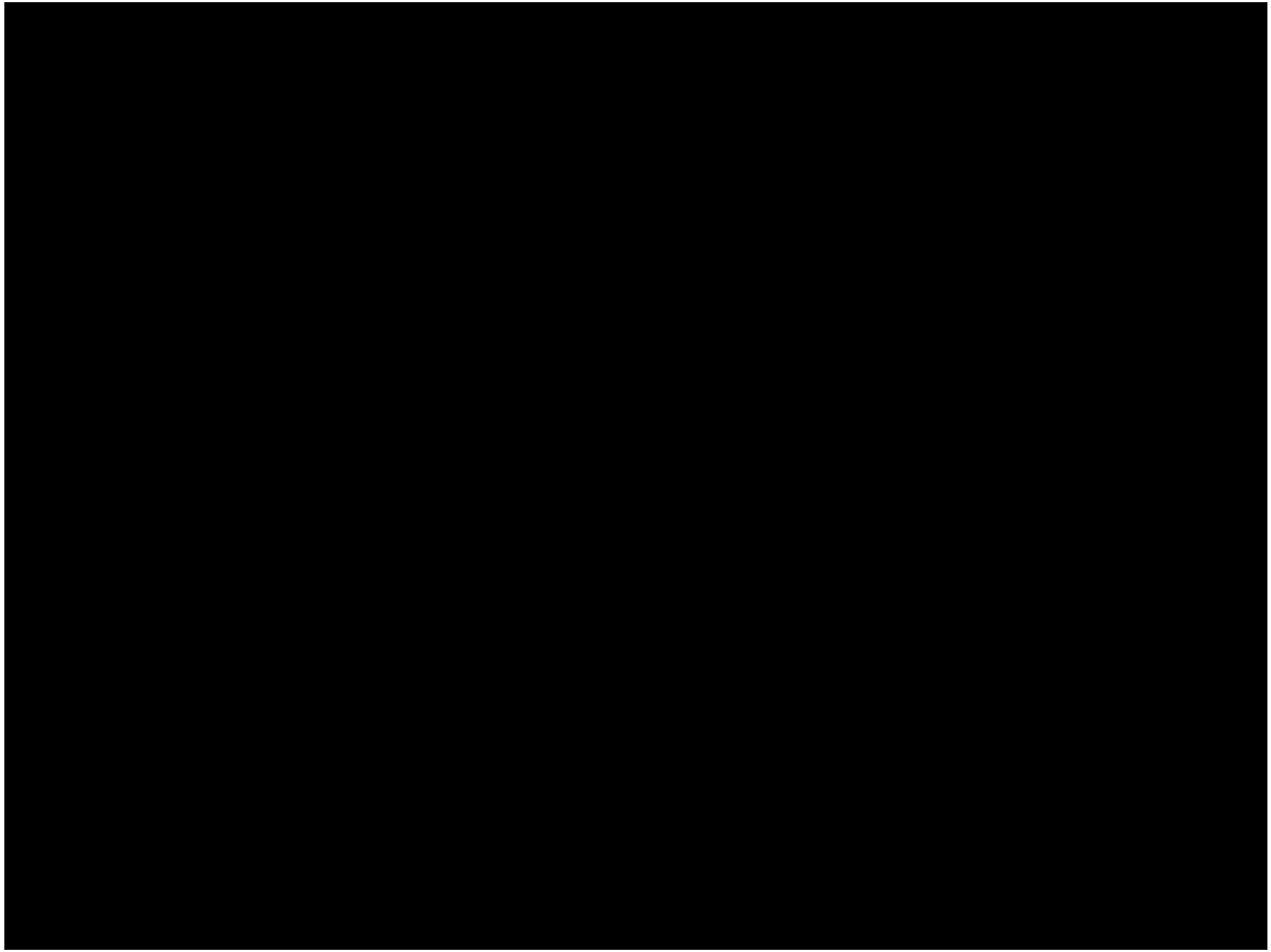
Many DBMS use b+tree, a b-tree with an additional level of linked leaves with copies of key and data records.

We wanted to focus on the b-tree so for the real-world data we stored the key (date) in the nodes and the key-values in a `std::map`.





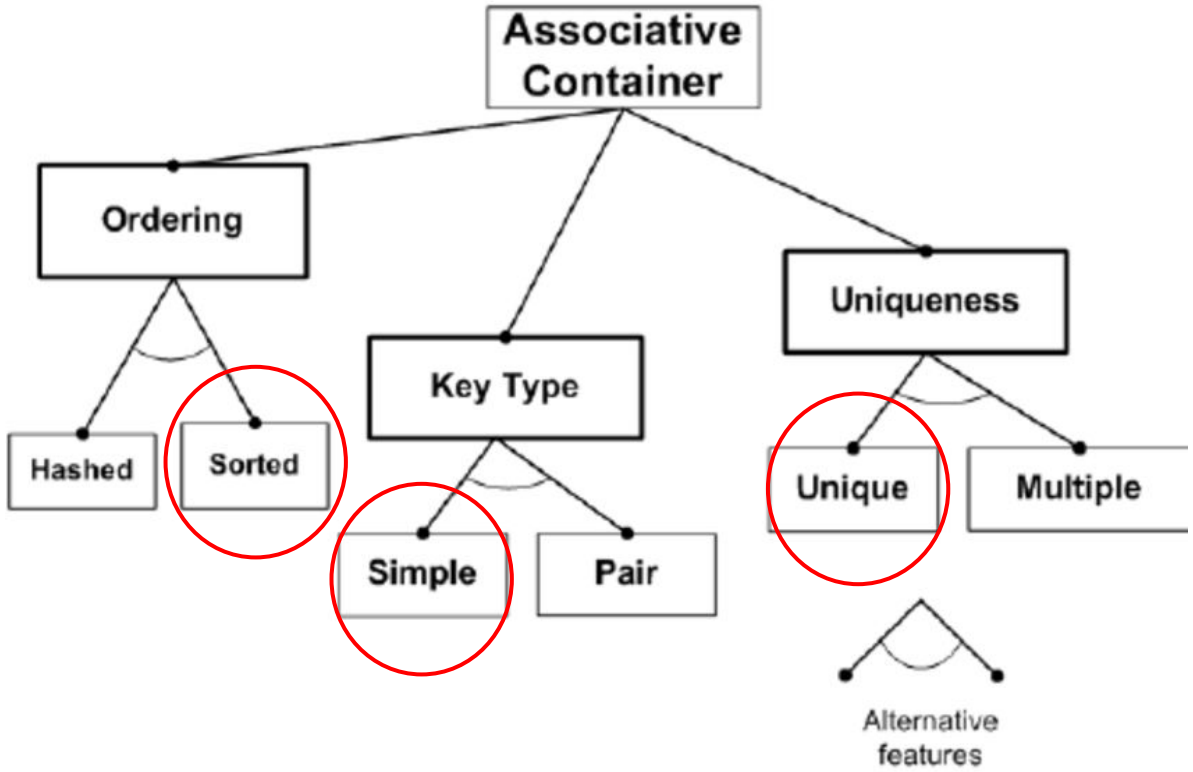
Demo



Designing BTree

- Create a btree based on the STL container standard for associative containers
- 3 main parts to a container:
  - Following STL shared properties (focus)
  - Allocators (V2)
  - Iterators (V2)
- Implemented in class templates

Feature diagram for Associative Container



BTree & BNode class

```
template<typename T>
class BTree {
   using btree_type = BTree<T>;
   private:
        BNode<T> *root;
        int m; /* pre-defined degree of tree */
        int (*compare)(T, T); /* comparative function for tree (as templated) */
        void (*printKey)(T); /* Function used to print items in the tree. */
        int index;
        int elemCount;

   public:
        BTree( int, int (*)(T, T), void (*)(T) ); /* constructor with defined type */
        ~BTree( ); /* destructor */

        void clear(); /* clears the tree */
        int insert(T); /* inserts a key into the tree */
        T remove(T); /* removes a key from tree */
        int size(); /* returns the size of the tree */
        bool empty(); /* returns bool if the tree is empty or not */

        BNodeKey<T>* search(T); /* returns the node associated with key */
        void traverse(); /* traverses the tree */
};
```

```
template<typename T>
class BNode
{
   using BNodeType = BNode<T>;
   public:
        bool isLeaf;
        int m; /* order of the tree */
        int size; /* refers to the number of keys */
        std::vector< BNodeKey<T> *> *keys;
        std::vector<BNodeType *> *children;

   private:
        int (*compare)(T, T);
        void (*printKey)(T);

   public:
        BNode(int m, int (*)(T, T), void (*)(T)); /* constructor */
        ~BNode( ); /* destructor */
        void print( );
        unsigned insertKey(T k, int index); /* inserts the key into node */
        unsigned insertChild(int i, BNodeType *x);
        void removeKey(int i, int j);
        void removeChild(int i, int j);
};
```

User Interface Design

```
COMMANDS:

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

1. **Parse** a data file from the user to create a table
   a. The PK of each row is inserted into the btree and a unique index corresponding to that PK is returned
   b. The index of the PK is then inserted into a map and corresponds to a vector of string values that represent the associated data for that row.
2. **Select** grabs the index from the btree and then finds the resulting vector from the map using that index.
3. **Delete** returns the index of the key in the node while deleting it from the btree, and then uses that index to remove the values from the map.

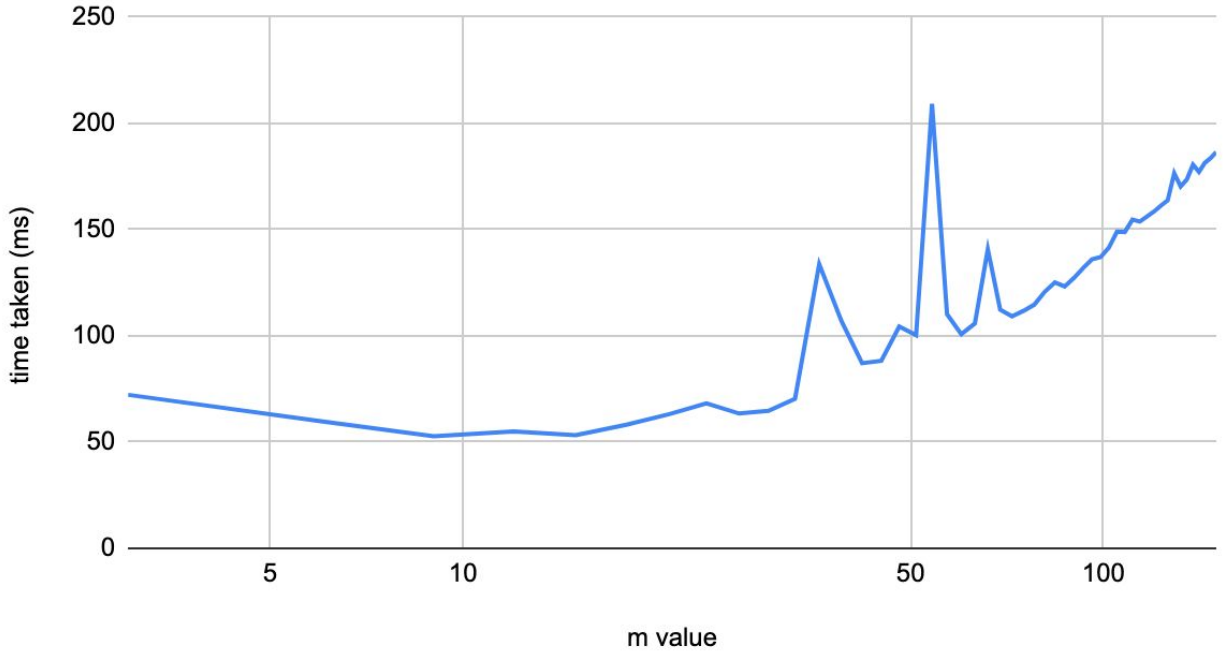
```
SELECT * FROM data
+--------------------------+----------+-------------+---------+----+--------------------------+--------------+
|artist                    |date      |last-week    |peak-rank|rank|song                      |weeks-on-board|
+--------------------------+----------+-------------+---------+----+--------------------------+--------------+
|Ricky Nelson              |04/08/1958|             |1        |1   |Poor Little Fool          |1             |
+--------------------------+----------+-------------+---------+----+--------------------------+--------------+
|Ricky Nelson              |11/08/1958|1            |1        |1   |Poor Little Fool          |2             |
+--------------------------+----------+-------------+---------+----+--------------------------+--------------+
```

Tests

- As we are designing an associative STL container, we wanted to compare our functions to current maps and sets that use red-black trees for their operations.
- Tested insert by timing how long it would take each containers to insert a range from 50,000 to 1 million random values.
- Tested search by choosing 10 random values within a range of 50,000 to 1 million values, and averaged the time it took each container to find the value.

Measurement Results

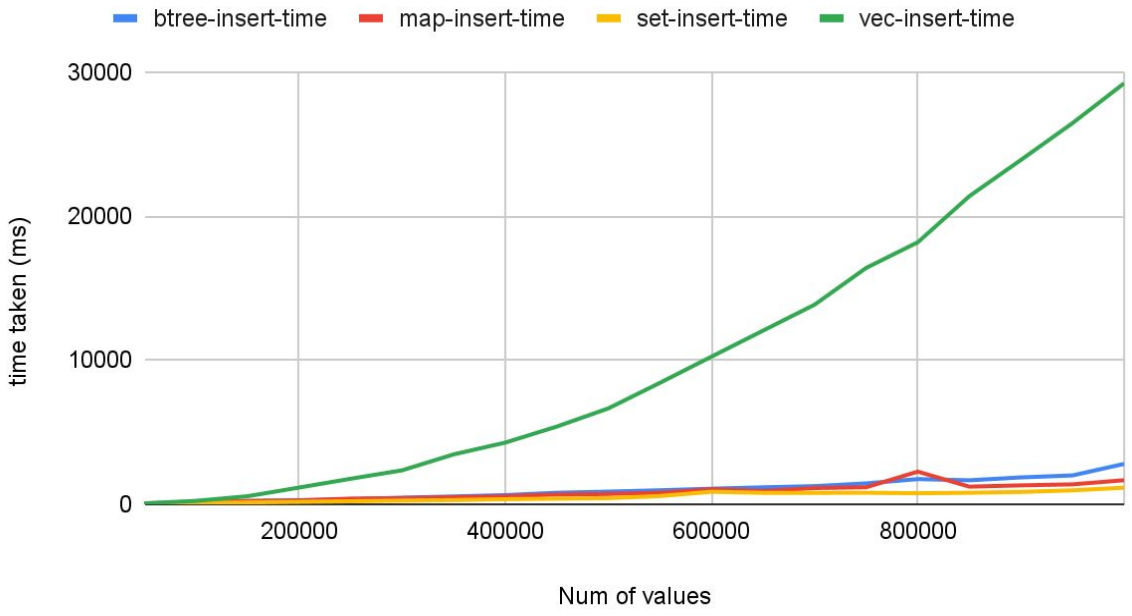
- m: minimum degree
- insertion of 50,000 random telements
- larger value of m ➔ wider, shorter tree



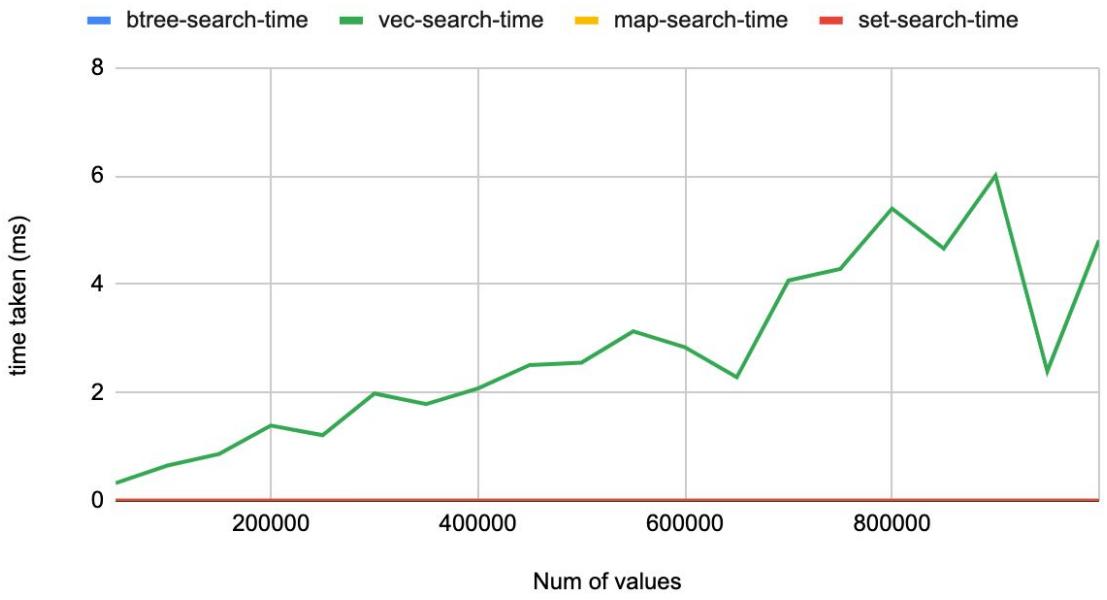
Measurement Results

- B-Tree inserts and searches faster than Vector

insert( ), m = 12



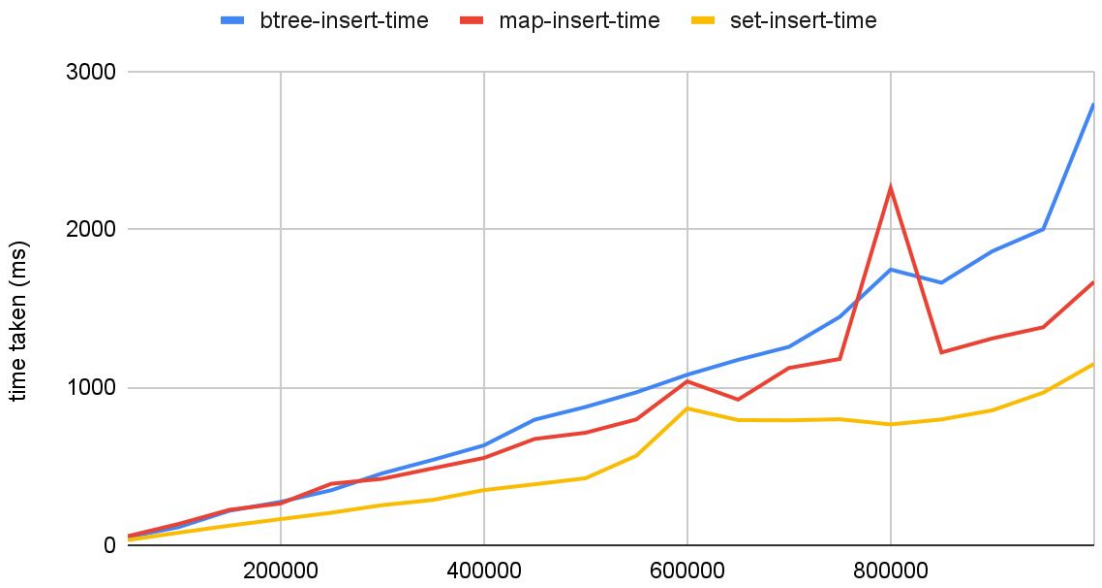
search( ), m = 12



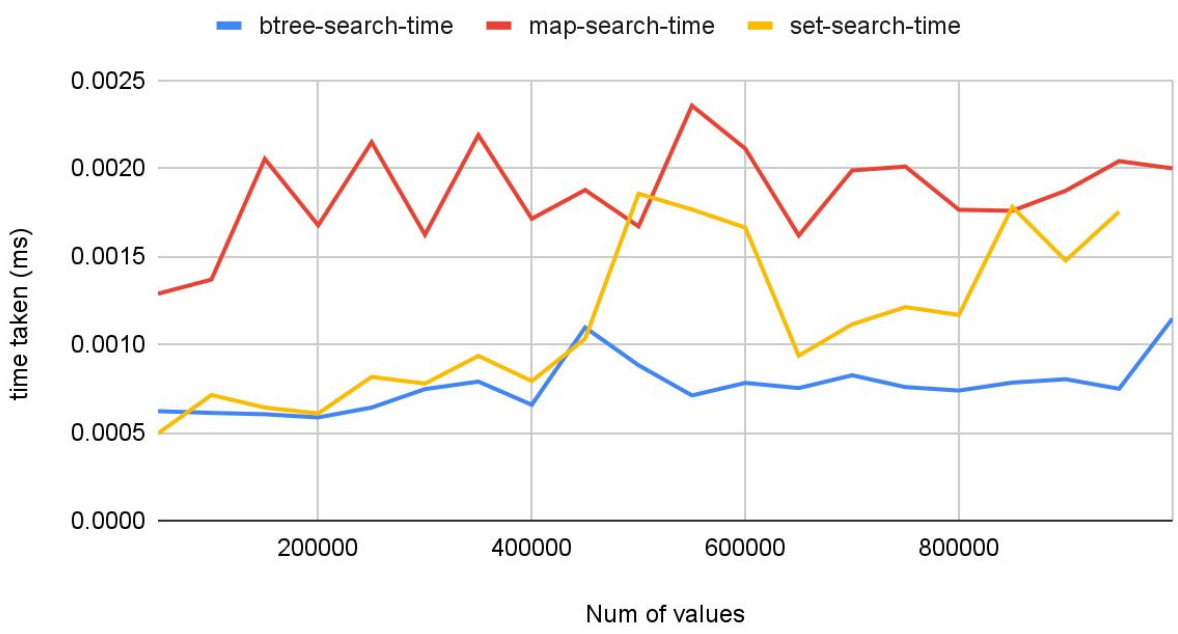
Measurement Results

- B-Tree faster at searching, roughly similar for inserting

insert( ), m = 12



search( ), m = 12



Conclusion

“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.”*

- Alexander Stepanov

V.20

- Implement B+ Tree
- SQL interface parser
- Multi-lookups
- Iterator and allocator implementations