Proposed Boost B-tree Library

Disk-resident ordered associative containers for Boost

Beman Dawes
May 17, 2011

Today’s Objectives

- Introduce B-trees (view from 36,000 feet)
- Learn how to use the proposed B-tree library
- Review status of the library
- Get feedback on the library

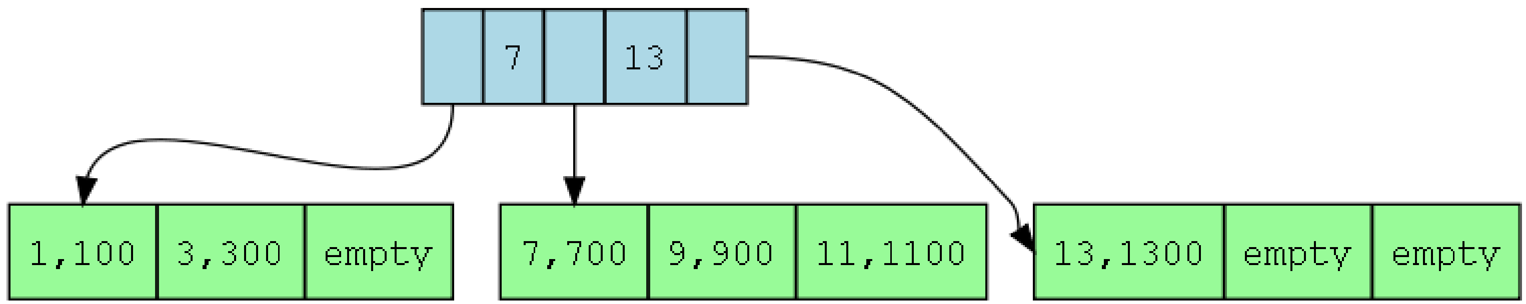
B-tree basics (1 of 2)

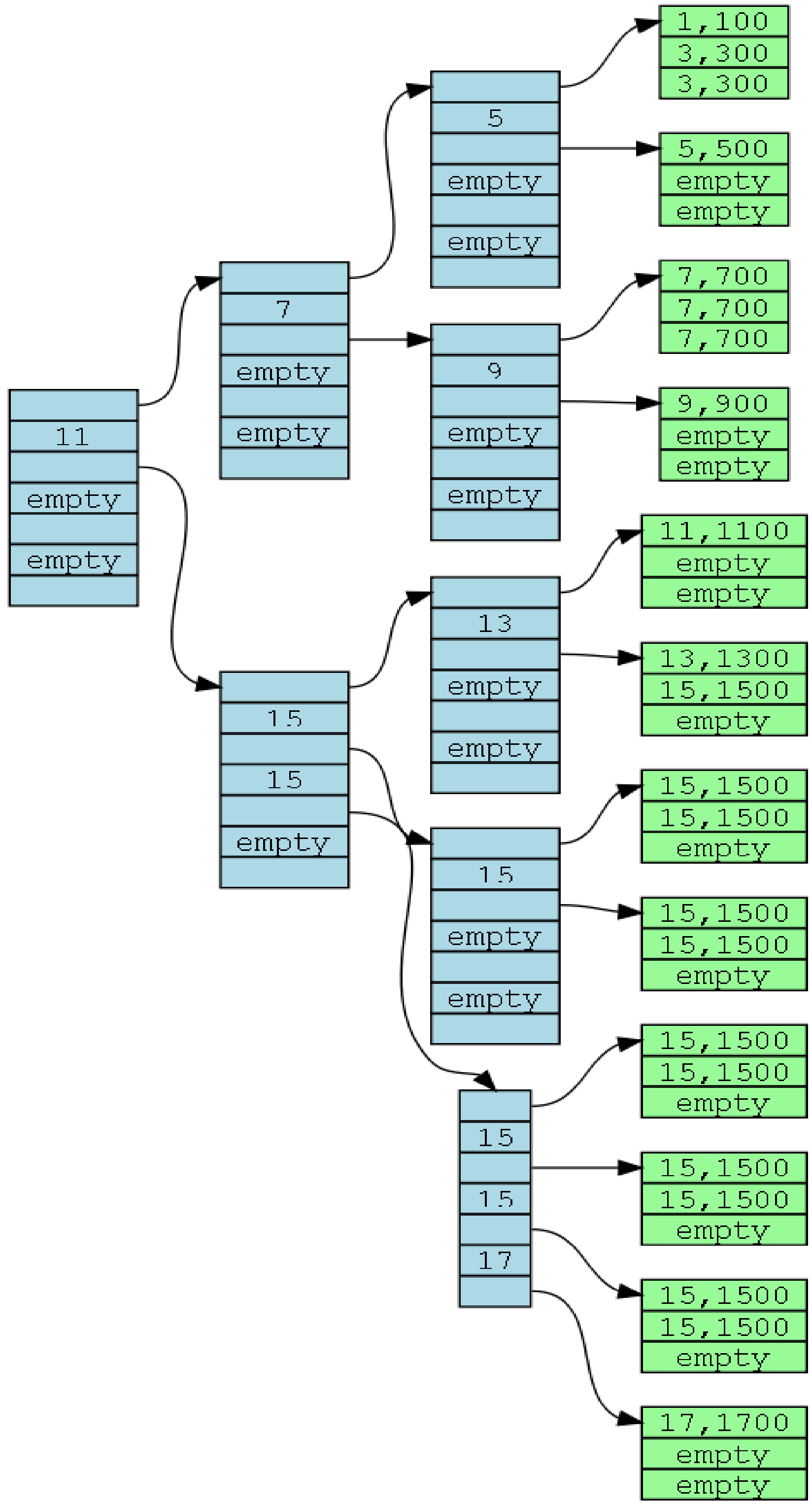
- A B-tree is a **balanced multi-way tree**
- Every leaf node is the same distance from the root node.
- Nodes have a fixed number of elements, some of which may be empty.

  (It is not unusual for nodes to have hundreds of elements.)

B-tree basics (2 of 2)

- Elements are ordered within nodes, nodes are ordered so that elements are ordered for the entire tree.
- Complexity of insert, erase, and search is $O(\log_2 n)$ for comparisons
- Insert, erase, and search touch $O(h)$ nodes. h is the height of the tree; about $\log_m(n)$ where m is the average entries per branch node. h is typically a single digit number.


7
13
1,100
3,300
empty
7,700
9,900
11,1100
13,1300
empty
empty


11
empty
empty
7
empty
empty
15
15
empty
5
empty
empty
9
empty
empty
13
empty
empty
15
empty
empty
15
15
17
1,100
3,300
3,300
5,500
empty
empty
7,700
7,700
7,700
9,900
empty
empty
11,1100
empty
empty
13,1300
15,1500
empty
15,1500
15,1500
empty
15,1500
15,1500
empty
15,1500
15,1500
empty
15,1500
15,1500
empty
15,1500
15,1500
empty
17,1700
empty
empty

Bottom line

- The B-tree's characteristics make it ideal for **disk-resident** indexes in general and **ordered associative containers** in particular.

- No other serious contenders for general disk-resident search.

- B-trees provide the indexing data structure for virtually all file systems, relational database search mechanisms, NoSQL structured data systems, and other disk-resident indexes.

Bottom line

- The B-tree's characteris… s make it ideal for **disk-resident** indexes in ger… and **ordered associative container** … rticular.

… disk-

… re for … se search … tems, and

B-trees are the technique of choice for disk-resident ordered associative containers

boost
C++ LIBRARIES

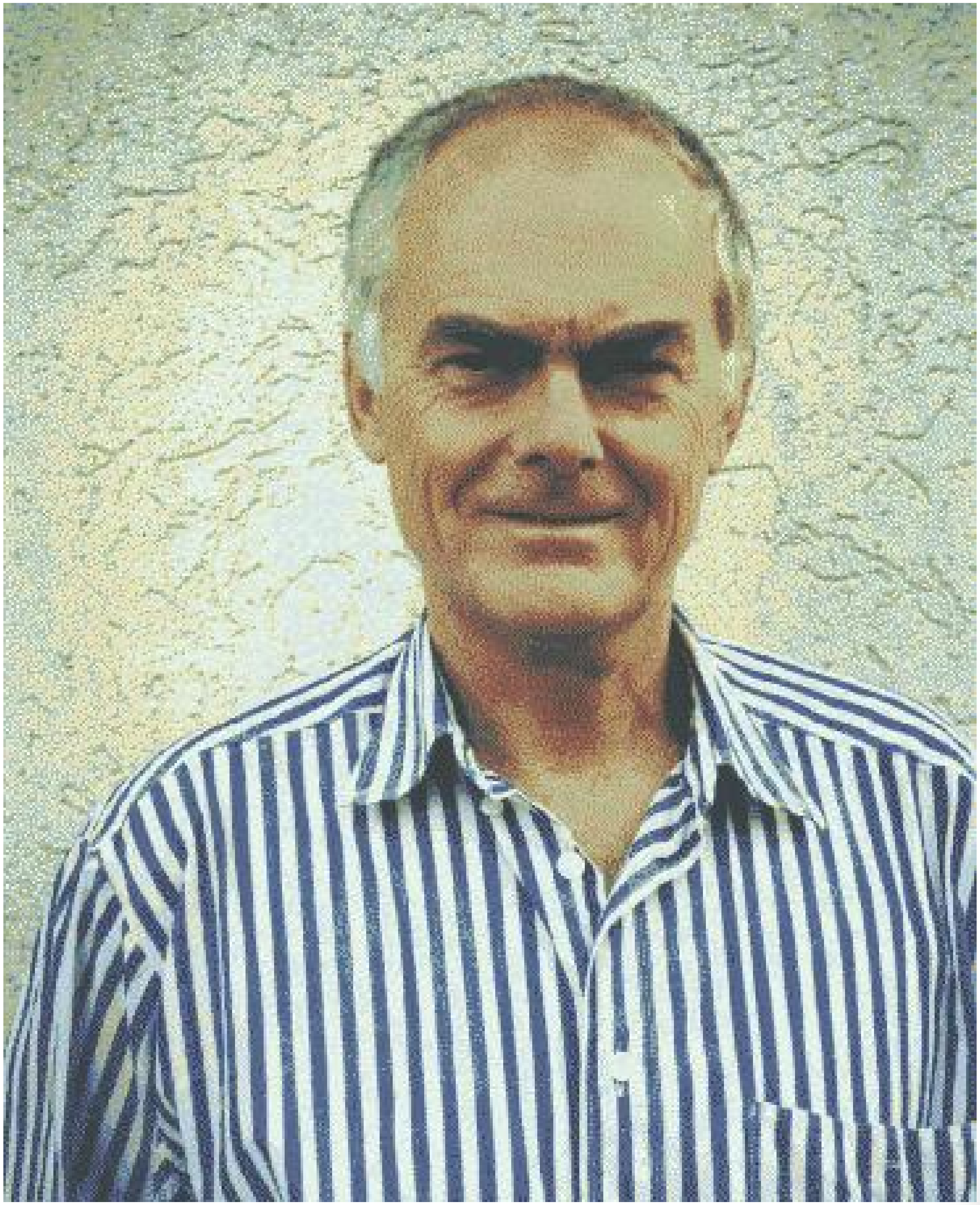
B-tree background

- B-trees were invented by Rudolf Bayer; he never said what the B stands for. Comer and others say B-trees should be called Bayer trees in his honor.

- Bayer also invented the Red-Black tree for in-memory search, and the Universal B-tree for disk-resident multi-dimensional (e.g. spatial) search.

- *Organization and Maintenance of Large Ordered Indexes*, Bayer and McCreight, 1972, was the original paper.

- *The Ubiquitous B-Tree*, Douglas Comer, 1979, is a great read



Proposed Boost B-tree Library

Proposed Boost B-tree library

- Provides **disk-resident ordered associative containers** as similar to standard library ordered associative containers **<u>as reasonable</u>**
- The important headers are **<boost/btree/set.hpp>** and **<boost/btree/map.hpp>**
- They provide classes **btree_set**, **btree_multiset**, **btree_map**, and **btree_multimap** plus scaffolding
- Variable length (e.g. character strings) keys and/or mapped values are supported

Proposed Boost B-tree library (con't)

- The library provides additional functionality in `<boost/btree/support>` that may be useful but can also just be ignored
- Data files are portable by default, assuming Key and T are types that yield portable data objects
- Performance is highly tunable
- Based on 30 years of experience with a C library I wrote and maintained

```cpp
typedef set<int> set_type;
set_type s;

s.insert(5);
s.insert(3);
s.insert(1);

for (set_type::iterator it = s.begin();
                        it != s.end(); ++it)
{
  cout << *it << '\n';
}

//  Output: 1
//          3
//          5
```

```
typedef btree_set<int> set_type;
set_type s("set.btr", flags::truncate);

s.insert(5);
s.insert(3);
s.insert(1);

for (set_type::iterator it = s.begin();
                        it != s.end(); ++it)
{
  cout << *it << '\n';
}

//  Output: 1
//          3
//          5
```

What is Boost B-tree **NOT** good for?

- Serialization; a B-tree isn't a Boost.Serialization replacement

- Sequential data; a B-tree is an associative container - ordinary files are a better choice for disk-based sequence containers.

- In-memory data; a Red-Black tree or a hash table (e.g. standard library associate containers) are usually better choices for in-memory use

Diffs from std associative containers

- Template parameters
- More stringent requirements on **`Key`** and **`T`**
- Iterator invalidation rules closer to unordered associative containers than plain old associative containers
- [multi]map `value_type` is `map_value<>` rather than `std::pair<>`
- `iterator` and `const_iterator` are the same type, and it is a constant iterator
- Some member functions missing, some added

Template parameters

```
// std::
template <class Key, class T,
  class Compare = less<Key>,
  class Allocator = allocator<pair<const Key, T>
  > >
class map;

// boost::btree::
template <class Key, class T,
  class Traits = default_endian_traits,
  class Compare = btree::less<Key> >
class btree_map;
```

Requirements on `Key, T`

- **<u>Types Key, T must work for binary input/output.</u>**
  - They must be trivially copyable types (3.9, ¶9) so that objects can be memcopy'ed.

    In other words, `std::is_trivially_copyable<>` must be true for the type
  - Objects of the type must be self-contained, and be position and process independent

Requirements on `Key, T`

- **Types Key, T must w k for binary input/output.**
  - They must be trivia pyable types (3.9, ¶9) so that objects ca emcopy'ed.
  - <> must be true
  - contained, and be position and process independent

`std::string` can't be used – it isn't trivially copyable

Variable length keys / mapped values

`std::pair<Key, T>` as a map/multimap `value_type` doesn't work if the size of Key varies

Instead, the `value_type` is `class map_value`:

```
template <class Key, class T>
class map_value
{
public:
  const Key& key() const;
  const T& mapped_value() const;
  std::size_t size() const; // dynamic size in bytes
};
```

```
typedef btree_map<int, long> map_type;
map_type bt_map("bt_map.btr", flags::truncate);

bt_map.emplace(2, -2);
bt_map.emplace(3, -3);
bt_map.emplace(1, -1);

for (map_type::iterator it = bt_map.begin();
     it != bt_map.end(); ++it)
{
  cout << "   " << it->key() << " --> "
       << it->mapped_value() << '\n';
}

// Output: 1 --> -1
//         2 --> -2
//         3 --> -3
```

```
typedef btree_map<strbuf, strbuf> map_type;
map_type bt_map("bt_map.btr",flags::truncate);

bt_map.emplace("eat","comer");
bt_map.emplace("drink", "beber");
bt_map.emplace("be merry", "ser feliz");

for (map_type::iterator it = bt_map.begin();
     it != bt_map.end(); ++it)
{
  cout << "  \"" << it->key() << "\" --> \"“
       << it->mapped_value() << "\"\n";
}

// Output: "be merry" --> "ser feliz"
//         "drink" --> "beber"
//         "eat" --> "comer"
```

```
class strbuf {
public:
  strbuf();
  strbuf(const char* s);
  strbuf(const strbuf& s);
  strbuf& operator=(const char* s);
  strbuf& operator=(const strbuf& s);

  std::size_t  size() const;
  const char*  c_str() const;

  bool operator==(const strbuf& rhs) const ;
   // other relationals...
 private:
   boost::uint8_t  m_size;  // std::strlen(m_buf) for speed
   char            m_buf[max_size+1]; // '\0' terminated
 };

 std::ostream& operator<<(std::ostream& os, const strbuf& x);

 inline std::size_t dynamic_size(const strbuf& x){return x.size();}
 template<> struct has_dynamic_size<strbuf> : public true_type{};
```

bt_map.btr hex dump

```
00000000  bbbb bbbb 0100 0100 0000 0000 0000 0003 ;;;;............
00000010  0000 0080 0000 0004 0000 0001 0000 0001 ................
00000020  0000 0001 0000 0002 0000 0000 0000 ffff ................
00000030  ffff 626f 6f73 742e 6f72 6720 6274 7265 ..boost.org btre
00000040  6500 0000 0000 0000 0000 0000 0000 0000 e...............
00000050  0000 0000 0000 0000 0000 0000 0000 0000 ................
00000060  0000 0000 0000 0000 0000 0000 0000 0000 ................
00000070  0000 0000 0000 0000 0000 0000 0000 0000 ................

00000080  0000 002f 0862 6520 6d65 7272 7900 0973 .../.be merry..s
00000090  6572 2066 656c 697a 0005 6472 696e 6b00 er feliz..drink.
000000a0  0562 6562 6572 0003 6561 7400 0563 6f6d .beber..eat..com
000000b0  6572 0000 0000 0000 0000 0000 0000 0000 er..............
000000c0  0000 0000 0000 0000 0000 0000 0000 0000 ................
000000d0  0000 0000 0000 0000 0000 0000 0000 0000 ................
000000e0  0000 0000 0000 0000 0000 0000 0000 0000 ................
000000f0  0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Design decisions for btree_map/multimap `value_type`

- `map_type` design options
  - `map_type<Key, T>`
  - `pair<const Key*, const T*>`
- `map_type` as `value_type` even if neither `Key` or `T` is variable length

Modifying mapped value

The btree_map/multimap implementation has to know when a node changes, so that nodes only get written to disk when changed. Non-constant iterators and mapped value types don't allow for that.

```
itr->mapped_value() = new_value;  // error
```

The fix is to provide:

```
iterator update(iterator itr,
  const mapped_type& x);
```

And then write:

```
bt.update(itr, new_value);
```

Missing functions

```
size_type    max_size() const;
T&           operator[](const key_type& k);
const T&     at(const key_type& k) const;
T&           at(const key_type& k);
```

Added functions (partial)

```
void open(const path& p,
  flags::bitmask flgs = flags::read_only,
  size_t node_sz = default_node_size)
  // node_sz ignored if existing file
void   flush();
void   close();
bool   is_open() const;
size_t node_size() const;
size_t max_cache_size() const;
void   max_cache_size(size_t m);
```

Use cases

1. As an ordered associative data storage container; the actual app data is the key itself (set) or the mapped value (map)

2. As an associative index into another file; the actual app data lives in that other file. Typically the B-tree is a map/multimap, with the mapped value being a record ID or file offset into the other file.

   If your design for a B-tree app seems to be fighting apparent B-tree limitations, consider (2) rather than (1)

String database class

```
template <class String, class ID>
class string_db
{
public:
  typedef String  string_type;
  typedef ID      id_type;  // 0 indicates none

  string_db(const path& p, flags f);

  id_type  insert(const string_type& s);
  id_type  erase(const string_type& s);
  bool     erase(id_type id);
  id_type  find(const string_type& s) const;
  const string_type&  find(id_type id) const;
};
```

Tuning Performance

- Maximum cache size
- Node size
- Preload option
- Pack optimization
- Traits

Maximum cache size

The implementation caches nodes that are no longer in use. The maximum size of this cache can be controlled by the user:

```
void max_cache_size(std::size_t m);
```

Cache size: btree_map vs std::map (5,000,000 calls - 3,160,338 size)

cache size	Insert	find	iterate	erase
32	33.55 sec 8.66 ratio	23.18 sec 6.50 ratio	0.07 sec 0.37 ratio	37.49 sec 8.48 ratio
2100	23.55 sec 6.10 ratio	21.74 sec 6.02 ratio	0.08 sec 0.41 ratio	34.44 sec 7.97 ratio
4200	13.79 sec 3.53 ratio	16.34 sec 4.56 ratio	0.08 sec 0.40 ratio	25.90 sec 5.81 ratio
8400	4.35 sec 1.12 ratio	3.62 sec 1.00 ratio	0.04 sec 0.23 ratio	4.13 sec 0.93 ratio

boost C++ LIBRARIES

Node size

- `btree_*` constructors take an optional 3rd argument specifying the node size.

- 4096, the default, is clearly the sweet spot for Windows NTFS disks, and seems OK for Linux. But some other value might be much better for other operating systems or file systems. 512 was optimal for floppies and older file or operating systems.

- **libs/tools/bt_time.cpp** can be used to experiment.

Node size: btree_map vs std::map (5,000,000 calls - 3,160,338 size)

Node size	insert	find	iterate	erase
512	5.54 sec 1.39 ratio	6.08 sec 1.65 ratio	0.07 sec 0.38 ratio	6.25 sec 1.41 ratio
1024	4.83 sec 1.20 ratio	5.25 sec 1.43 ratio	0.06 sec 0.29 ratio	5.33 sec 1.23 ratio
2048	4.30 sec 1.08 ratio	4.03 sec 1.12 ratio	0.05 sec 0.26 ratio	4.29 sec 0.97 ratio
4096	4.45 sec 1.13 ratio	3.67 sec 1.03 ratio	0.04 sec 0.22 ratio	4.16 sec 0.92 ratio
8192	5.20 sec 1.34 ratio	3.42 sec 0.93 ratio	0.04 sec 0.22 ratio	4.45 sec 1.01 ratio
16380	5.84 sec 1.50 ratio	2.70 sec 0.75 ratio	0.04 sec 0.22 ratio	4.59 sec 1.06 ratio

Preload option

- `flags::preload` set in the `btree_*` constructor or open 2nd argument causes preloading of an existing B-tree file. This can cause a marked application speedup. Your mileage may vary.

Pack optimization

- If a node fills on an insert, and all prior inserts have been ordered, the insert is done on a new node. This in contrast to normal node fill behavior, which splits the node into two nodes.
- That results in 100% node utilization, rather than the usual 75% utilization
- On the 3,160,338 element timing test file, size dropped from 34,299,904 to 25,440,256
- Currently only implemented for leaf nodes

Traits

```
struct default_endian_traits
{
  typedef integer::ubig32_t  node_id_type;
  typedef integer::ubig16_t  node_size_type;
  typedef integer::ubig16_t  node_level_type;
  static const
    BOOST_SCOPED_ENUM(integer::endianness)
    header_endianness
        = integer::endianness::big;
    };
```

File Header

Each Boost.btree file begins with a header record.

Contents:

- Tree infrastructure data (node size, root node id, flags, etc.)
- User convenience data (c-str, user ints, etc.)
- Robustness data (record type, class UUID, etc.)

B-tree variations

- B-trees have many variations
- The B+-tree (pronounced “B plus tree”) stores key, mapped-value pairs in leaf nodes, as usual, but only keys in branch nodes.

  The Boost library, and most other practical B-tree implementations, use B+-trees, but refer to themselves with the generic B-tree name.
- Erase on empty rather than merge on half-full.
- Be warned that the Wikipedia articles on B-tree variants contain quite a few errors, as do other web based B-tree postings.

Variations NOT applied

- Sequence links; Slower inserts, erases (actual tests), difficult to establish paternity chain for insert, erase, particularly with non-unique containers
- Branch key prefix compression; KISS
- Branch key suffix compression; KISS
- Node full element redistribution on insert ; KISS

Library status

- Available via Github. See https://github.com/Beman/Boost-Btree/raw/master/README
- Some docs but very incomplete
- Build and test setups, via Jamfiles and a VC++ 10 setup
- Example and timing programs
- Tested on Windows with VC++ 8/9/10 and GCC/MinGW 4.4/4.5, and on Ubuntu Linux 11.4 with GCC 4.4
- No C++0x features yet

Testing with `stl_test.cpp`

- Performs operations on a `std::map<int32_t, int32_t>` and a `btree_map<int32_t, int32_t>`, and verifies the results are identical

- Lots of program options to set test conditions

- Has run as many as 5 billion operations without failure

Support headers, mini-libraries, and full-blown libraries

- Proposed Endian library
- Simple fixed size c-string holder - fixstr.hpp
- Simple holder for variable length c-strings - strbuf.hpp
- random_string.hpp

boost C++ LIBRARIES

Development targets

- Formal review (Version 0.9)
- First Boost release (1.0)
- B-tree of B-trees (Version 1.1)
- Shared modification by other threads and/or processes (Version ?)

Help wanted!

- Preview comments
- Feedback from early adopters
- Independently developed test program

Discussion

- Comments?
- Missing functionality?
- Concerns or issues?

Thank You!

```
cd wherever
git clone ^
  git://github.com/Beman/Boost-Btree.git xbtree
svn export –force ^
  http://svn.boost.org/svn/boost/trunk xbtree
```

Windows:

```
cd xbtree
bootstrap
cd libs\btree\test
..\..\..\bjam
```

POSIX-like:

```
cd xbtree
./bootstrap.sh
cd libs/btree/test
../../../bjam
```