COMP3171/COMP9171 13s2 Assignment 4: The B-Tree

(Version 1.0, Sat 5 Oct 2013 12:47:19 EST)

WORTH: 14 marks **DUE: Friday, 1 November 2013 at 23:59:59**

Developed by Jerry Cain of Stanford University and adapted by Jingling Xue.

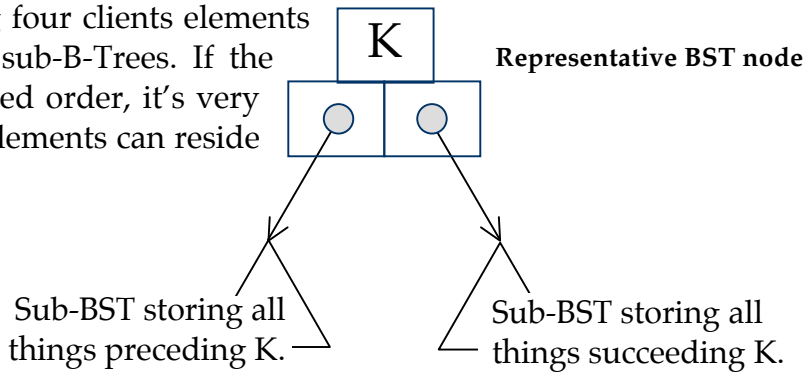
We've learned that C++ considers the preservation of type information to be much more important than C ever does. With that understanding, C++'s support for generic containers and algorithms is much more sophisticated, and you should practise writing template containers in C++—afterwards, you'll be in a better position to compare and contrast the two languages, because you'll have written serious, industrial-strength data structures in C++. In general, it's hard to write quality generic code no matter what language you're using!

1. Aims

In this assignment you will get a chance to implement a reasonably complex, generic C++ container, together with proper iterators. You will practice working with template classes and everything this entails. You will write a bi-directional iterator from scratch, use iterator adaptors to provide reverse iterators. You will also employ lots of the C++ features you have learnt about in this course. You will also have a second chance to practice working move semantics if you didn't get it right in Assignment 2.

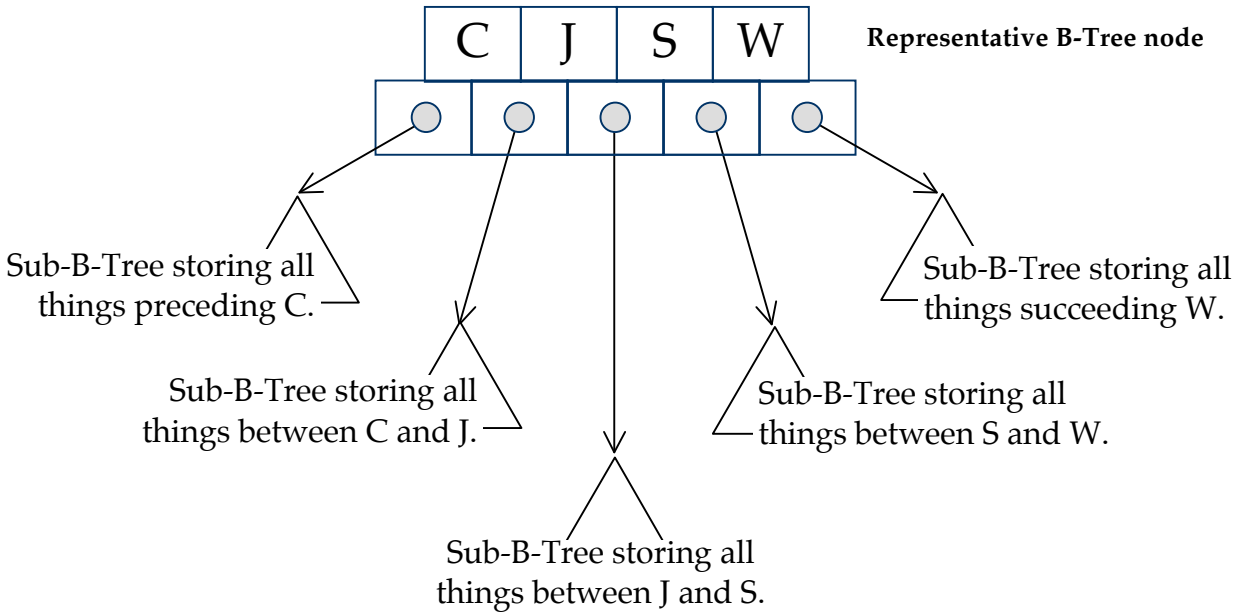
2. The B-Tree Container

B-Trees are generalised binary search trees, in that a binary-search-tree-like property is maintained throughout the entire structure. The primary difference—and it's a big one—is that each node of the tree has many children, from a handful to thousands. Actually, it's the number of client elements stored within the node that determines the number of children. In the same way that a single element partitions a binary search tree into two sub-trees, a B-Tree node storing four clients elements partitions the B-Tree into five sub-B-Trees. If the four elements are stored in sorted order, it's very easy to constrain what type of elements can reside in each of the five sub-B-Trees.



Check out the B-Tree node we've drawn on the next page. Note that C, J, S, and W split the rest of the B-Tree

into sub-B-Trees that store elements A through B, D through I, K through R, T through V, and X through Z. This B-Tree property is maintained throughout the entire structure. Think of each node as a mother with quintuplets instead of twins.



The above picture implies that the stored elements are characters, but any element capable of comparing itself to others can be stored in a B-Tree. And the decision to store four keys per node was really arbitrary. While all nodes will store the same number of elements (except along the fringe—sometimes there just aren't enough elements to fill up the fringe of the B-Tree), the actual node size can be tweaked to maximise performance. In practice, the number is typically even, and is chosen so that the total amount of memory devoted to one node is as close to a memory page size as possible. That way, all of your elements are likely to be right next to each other in memory, and the operating system maximises the number of elements in the cache at any one time.

We're not going to burden you with those optimisations though. Your task is to implement a generic B-Tree template capable of storing any type whatsoever. Here's a very brief look at the template interface. You'll need to provide implementations for the constructor and destructor, the **`insert`** operation, the two versions of **`find`** (which are precisely the same, save the **`const`** versus non-**`const`** business), and the output operator **`<<`**. You will also need to provide full, explicit copy control using *value semantics*, i.e., no shared pointees.

```
template <typename T>
class btree {
 public:
  btree(size_t maxNodeElems = 40);
  btree(const btree<T>&);
  btree(btree<T>&&);

  btree<T>& operator=(const btree<T>&);
  btree<T>& operator=(btree<T>&&);
  friend ostream& operator<< <> (ostream&, const btree<T>&);

  // typedef's for iterators and declarations
  // for begin(), end(), rbegin(), rend() and
  // cbegin(), cend(), crbegin() and crend() go here

  iterator find(const T&);
  const_iterator find(const T&) const;
  pair<iterator, bool> insert(const T&);

  // make sure your destructor does not leak memory
  ~btree();

 private:
  // your internal representation goes here
};
```

The **`btree.h`** header file tells you everything you need to know about what you're implementing and what you aren't. You must implement what's outlined above, and in doing so, you should write clean, compact C++ code that you'd be proud to show mum and dad.

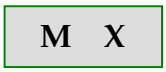
3. Implementation Details

3.1 Insertion

Remember that you are not required to balance the B-Tree.

We're leaving the details of the concrete representation up to you, but the details of insertion are complicated enough that you deserve a few illustrations. Once you have insertion down, search is more or less straightforward.

For the purposes of illustration assume that nodes are engineered to store up to four characters. Initially, a B-Tree is empty, but after inserting two elements, the state of the tree would look as follows:



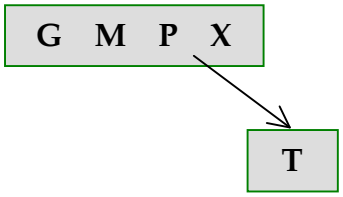
Because there are only two elements in the tree, we needn't give birth to any child nodes. In fact, it won't be until we saturate the root node above with two more elements before the addition of new elements will mandate the allocation of a child.

To insert the letter P at this point would require that the X be shifted over to make room. The node is designed to store a total of four elements, so the P really belongs in between the M and the X:

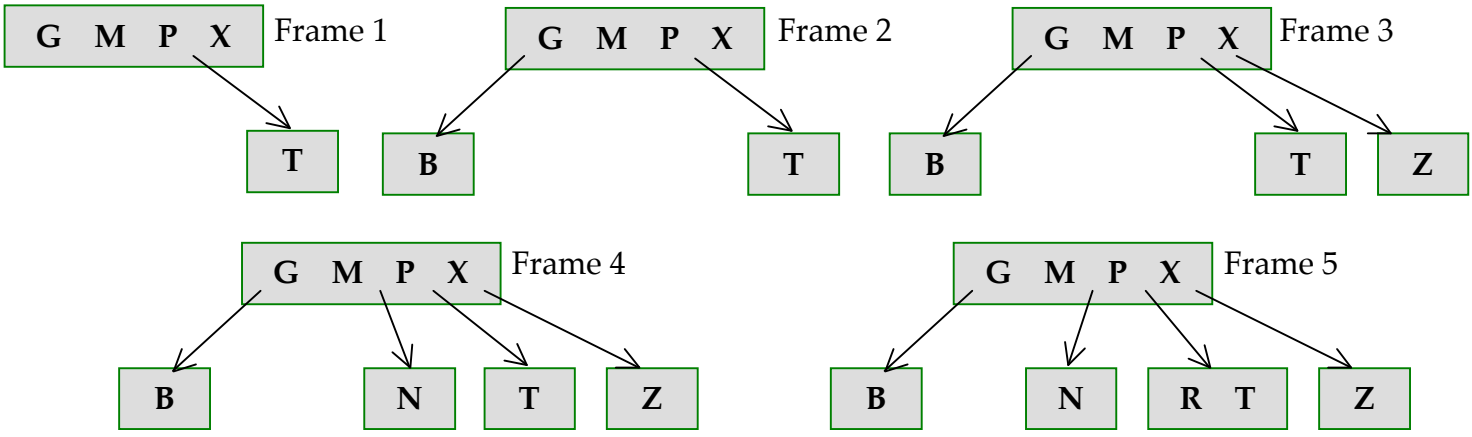
Throw in a G for good measure, and voilà, you have a saturated node of size four:

At this point, the node is structured this way for the lifetime of the B-Tree. But now that it's completely full, it will need to give birth to child nodes as needed.

Consider now the insertion of the letter T. Since five's a crowd, a new node—one capable of storing all types of characters ranging from Q through W—is allocated, and a pointer to this node is recorded as being 'in between' elements P and X of the root:

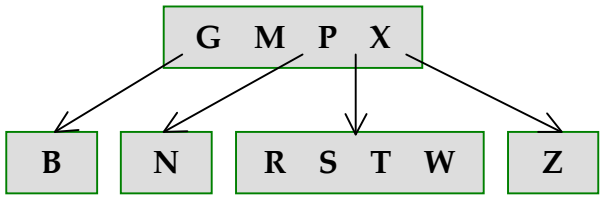


Curious how the tree grows after we insert a B, a Z, an N, and then an R? Check out the following film in slow motion:

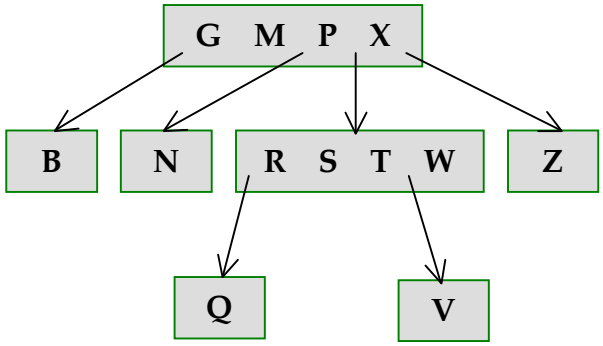


Note that each of the new nodes was allocated because the first element inserted into it was required to reside there. T had to go in the node stemming between P and X because anything alphabetically in between P and X needs to be placed there. B is inserted before G, Z after X and N between M and P. Finally R is T's sibling.

As more and more characters are inserted, it's likely that even the child nodes will saturate, and when that happens, each of the children starts its own family. Consider the insertion of S and W into the Frame-5 B-Tree above. R and T will allow S and W to join them:



The insertion of Q and V at this point would require the introduction of third-level nodes, as with:



In theory, there's no limit whatsoever to how deep the tree can get. At the risk of repeating ourselves and seeming monotonous, let us say some important things again:

The tree above stored four elements per node, but B-Trees can store up to any predefined maximum. The default value of the constructor argument implies that you should store a default of 40 elements per node. Of course, the code you write should work for any value greater than 0.

The tree above stored characters, but the B-Tree is templated, so we could have just as readily used a B-Tree of strings or doubles or records. Of course, the code you write shouldn't be character-specific, but general enough to store any type whatsoever. That's where templates come in.

The underlying implementation of the B-Tree is for you to decide, but the implementation certainly *must make use of a multiply-linked structure in line with the drawings above*.

Failure to do so is likely to result in a nice, round mark for this deliverable (0!). How do you store the client elements and the child nodes? That's your job.

The **`insert`** function returns a pair that is a class template provided in the C++ STL. The first element in the pair is an iterator positioned at the element inserted. The second element is a Boolean value indicating whether the insertion is successful or not. The second element is false if the element to be inserted is found already in the B-Tree and true otherwise.

3.2 Searching

The **`find`** function returns an iterator positioned at the element found in the B-Tree. If the element being searched is not found in the B-Tree, an iterator that is equal to the return value of **`end()`** is returned.

3.3 Copy Control

All copy-control members must be implemented correctly. The move constructor and move-assignment operators are implemented in a few lines each. For a moved-from btree object, you should know by now how to put it in a state correctly according to the C++11 requirement.

3.4 Iterators

You are required to implement a bonafide bi-directional iterator type capable of doing an **inorder walk** of the B-Tree. An inoder traversal will allow all values in a B-Tree to be printed in the increasing order of their values, where the comparison operator is defined by the type of the elements in the B-Tree. Since the iterators are bi-directional they will also support the reverse inorder traversal. *You MUST implement your iterators as an external class (**`btree_iterator`**), failure to do so will result in a final mark of 0 for this deliverable.*

The iterators will come in **`const`** and non-**`const`** forms, just like they would for any built-in STL container. The iterators should respond to the **`*`**, **`->`**, **`++`**, **`--`**, **`==`** and **`!=`** operators and be able to march over and access all of the sorted elements in sequence, from 1 to 32767, from "aadvark" to "zygote", etc., as well as support backward movement. The challenge is working out how **`++/--`** behave when a neighboring element resides in another part of the B-Tree.

Your iterators should be appropriately declared as bi-directional. You should implement and test the complete bi-directional iterator functionality. This includes pre/post incre-

ment and decrement. The post-increment/decrement operations should return a copy of the value pointed to before the increment/decrement is performed.

It can be argued that the non-**`const`** iterators to a B-Tree are dangerous, since changing the values of elements is likely to break the B-Tree invariant. This is indeed the case, but you still have to implement this feature to develop some practical experience and we will assume the responsibility. Your non-**`const`** iterator will be tested, e.g., increment every value in the B-Tree by one. It is important that you get to practice writing a non-**`const`** iterator.

In addition, you are to implement **`const`** and non-**`const`** forms of reverse iterators. These reverse iterators should be returned by the **`rbegin()`** and **`rend()`** B-Tree member functions. You should use iterator adaptors to implement your reverse iterators. *Once you have managed to make the* **`const`** *and non-***`const`** *forms of the "forward" iterators work for you, it takes usually a few extra minutes to implement the reverse iterators by using iterator adaptors.*

You can now implement the C++11 begin and end member functions, **cbegin(), cend(), crbegin()** and **crend().** *This can be done in a few minutes by implementing them in terms of the member functions that you have already implemented for the "forward" and reverse iterator.*

3.5 Output Operator

You are to implement the output operator for the B-Tree class. This operator should just print the values of the nodes in the B-Tree in breadth-first (level) order. The values should be separated by spaces. No newline is to be output. For example, the output operator would print the following when called on the last version of the tree in 3.1:

```
G M P X B N R S T W Z Q V
```

You may assume that the types stored in your B-Tree themselves support the output operator. Note that we will use the output operator in combination with the iterator traversal to ensure that your B-Tree structure is above-board correct. This means that you *must* construct your B-Tree exactly as indicated, any other approach, including any balancing scheme, is likely to lead to failed tests.

4. Testing the B-Tree

We've provided a simple C++ program that'll exercise your B-Tree just enough to start believing in it when it passes. It's hardly exhaustive, as it only builds B-Trees of integers. It does, however, do a good job of confirming that **`find`** and **`insert`** do their jobs, and it does an even better job of showing you how quickly B-Trees support the insertion of

roughly 5,000,000 random integers in less than 90 seconds. Try that with a hash table or a binary search tree and you'll see it flail in comparison. This is cool and you should think so too! Exciting!

Apart from this test program, a couple of other simple tests are provided.

You are responsible for making sure your B-Tree is bullet proof, and if some bug in your code isn't flagged by our tests, that's still your crisis and not ours. You should try building your own test programs to make sure that everything checks out okay when you store doubles, strings, and structures. This is a great opportunity to try some unit testing, because the class you're testing is very small and easily stressed.

Your code will be compiled together with a test file containing the **`main`** function and with these compiler flags:

```
% g++ -Wall -Werror -O2 –std=c++11 ...
```

No matter where you do your development work, when done, be sure your deliverable works on the school machines.

5. Getting Started

You are provided with a stub that contains the files you need to get started on this deliverable. The stub is available on the subject account. Login to CSE and then type something like this:

```
% mkdir -p cs3171/btree
% cp ~cs3171/soft/13s2/btree.zip .
% unzip btree.zip
% ls
% more README
```

In the stub, you should find a fully documented **`btree.h`** file, a **`btree.tem`** stub file, **`btree_iterator.h/tem`** stub files, and some test files designed to exercise the B-Tree and make sure it's working. You'll see a **`Makefile`** in the mix as well.

You are free to introduce any new classes you like, e.g., a node class. *To make marking easier please place any new classes in the submission files.* Take the time to do this thing right, since it's really at the heart of what C++ generics are all about. You should want to do these things the right way.

You are **not** allowed to change the names of the **`btree/btree_iterator`** classes. Neither can you change the public interface of the **`btree`** class template as defined in

btree.h, of course, you will agument it with the appropriate iterator related types and operations.

Please note that the supplied files don't yet compile since **btree.tem**, etc., are empty.

Our implementation with generous comments consists of about 400 lines of C++ code.

6. Marking

This deliverable is worth **14**% of your final mark.

Your submission will be given a mark out of 100 with a 80/100 automarked component for output correctness and a 20/100 manually marked component for code style and quality.

As always, failure to implement a proper B-Tree representation and/or an external, iterator class, will earn you a total mark of 0 for this deliverable.
As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions. However, the style marks will be awarded for writing C++ code rather than C code.

As for the performance of your solution, we are lenient. When compared to a straightforward implementation, your solution is expected not to run >20X slower.

7. Submission

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command (from within your `del4` directory):

```
% give cs3171 btree btree.h btree.tem btree_iterator.h \
btree_iterator.tem
```

Note that you are to submit specific files only, this means that these are the only files you should modify. You should also ensure they work within the supplied infrastructure.

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons to convince the LIC otherwise. Any submission after the due date will attract a reduction of 10% per day to the maximum mark. A day is defined as a 24-hour day and includes weekends and holidays. Precisely, a submission x hours after the due date is considered to be *ceil*(x/24) days late. No submissions will be accepted more than five days after the due date.

Plagiarism constitutes serious academic misconduct and will not be tolerated. CSE implements its own plagiarism addendum to the UNSW plagiarism policy. You can find it here:

http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html

Further details about lateness and plagiarism can be found in the Course Outline.