cs280-a2-blist

This assignment gives you a chance to implement an API for your first container (a.k.a. ADT). The container is a modified double-linked list that has an interface that is a small subset of the std::list interface. (Everyone has implemented several linked-lists in the past so that aspect of the assignment is trivial.) There are a few enhancements that you will make to the typical linked-list interface. The task is to implement a templated class named BList which clients can use as a container of any type. We call it a BList because the nodes in the list are similar to nodes in a BTree, which we will discuss later in the semester. It's also kind of like a hybrid between a linked-list and an array in that it appears to be a linked-list of arrays. The arrays can be sorted or unsorted.

The essential difference from a "standard" linked-list is that each node can contain more than one data item. A standard linked-list has a one-to-one mapping of items to nodes. This means that a BList that has 2 items per node will require fewer nodes than a list with nodes containing only 1 item per node. A BList with 4 items per node will require even fewer nodes, and so on. This can provide a significant performance increase while only requiring some additional complexity in the code.

The BList class will also have the capability to maintain a sort order when using the insert method. The partial class definition is provided in **Blist.h** .

Notes

1. Several public methods may throw a `BListException` exception. This exception class is already defined and implemented for you. There are only two kinds of exceptions for this assignment: out of memory exceptions (`E_NO_MEMORY`) and invalid index (`E_BAD_INDEX`). The `E_DATA_ERROR` is reserved for future use.
2. If there is no memory to allocate a node, you should throw an exception with the code set to `E_NO_MEMORY`. You must catch the `std::bad_alloc` thrown by new and throw a `BListException`. Do not use `std::nothrow`. If you don't know what that is, you are probably not using it.
3. If an invalid index is provided to the remove method or the subscript operators, an exception should be thrown with the code set to `E_BAD_INDEX`.
4. The only class that you can use from the STL is `std::string` (or possibly `std::stringstream` for formatting error messages.) You cannot use any generic algorithms or functors. You may not even need `std::string`.
5. When a node contains 0 items (due to removing items), you must remove the empty node from the list and delete it.
6. Since this is a templated class, you don't know exactly what types are going to be put into the list. During the insert method and the find method you need to compare items. Be sure to only use the less than operator and the equality operator for comparisons. All data is guaranteed to support these two operators. There is no guarantee that the data supports any other operators.
7. Make sure to test all of your code. If there is a method in the class that the driver does not call, it won't be instantiated and may fail during the grading. Also, pay attention to details like self-assignment; make sure you can deal with those kinds of things.
8. Here's how you deal with the exception if there is no more memory available.

```
BNode *node = 0;
try
{
  node = new BNode; // might throw a std::bad_alloc or something else
}
catch (const std::exception& e)
```

```
{
  throw(BListException(BListException::E_NO_MEMORY, e.what()));
}
```

9. Finally, consult the driver for many more details. Almost every question you may have will be answered by looking at the examples in the driver.

FAQ

If I'm keeping the list sorted using the insert and the client calls the push_back method, they will mess up the sort. Do I have to re-sort the entire list after they call push_back?

No. The client (driver) will only call insert or push_back/push_front. It will never mix them so you don't have to worry about that. When the insert method is used, you can assume that all of the items have been placed into the list with insert. The same is true for the push_back/push_front methods; they only work with unsorted lists. (Hint: Put a flag in your class that is set if insert is called and is unset if one of the other methods is called.)

I'm getting weird errors from the compilers when I try to build my code. The errors say something about a bad token '<'. What's wrong?

You are probably using Visual Studio and including the implementation file twice. Remember, it's a templated class, so you have to include the implementation in the header file. Since it's already included in the header, adding it to the project or on the command line causes the code to be added twice. See the command line posted for an example of correct usage.

I'm making a function that returns a `BNode *` but the compiler is complaining that it doesn't know what a BNode is. What's wrong?

Answer: You probably forgot to use the keyword typename with the return value. Here's an example of how you would specify a function called make_node that creates a node:

```
template <typename T, unsigned Size>
typename BList<T, Size>::BNode *BList<T, Size>::make_node(const T& Item) const
```

The keyword `typename` above is probably what's missing. Since BList is a templated class, the compiler doesn't know the the type of T yet, so it can't look for BNode. Since it can't look it up, it assumes that BNode is a member variable of BList, which doesn't make sense when the compiler expects to find a type. (It is the return-type afterall). In order to tell the compiler that BNode is a type and not a member variable, use the typename keyword.

In the Splitting Examples diagram 2 shows:

[10,12] <=> [20,22], insert 15, [10,empty] <=> [12,15] <=> [20,22]

but logically this 'seems' to be an ideal solution:

[10,12] <=> [20,22], insert 15, [10, 12] <=> [15,empty] <=> [20,22]

Is there a reason we are inserting in this way?

There are a few reasons why we are doing it this way. One of the reasons is the Blist structure is modeled after the B-tree data structure. In a B-tree (which we will discuss later in the semester), the only way to insert a node into the tree is to split an existing node into two equal halves. The primary reason for this is so that we don't have any "under-full" nodes. An under-full node is a node that has more than half of the slots empty. This wastes far too much space and can slow the search algorithms as they must traverse many nodes that only have a few elements in them. You can find more information about B-trees here: B-tree[1]. Also, B-trees are always sorted, so there is no push_front/push_back functionality.

[1] https://en.wikipedia.org/wiki/B-tree

Of course, there are some slight optimizations that could be made. You would trade wasted space for speed of inserting, as in your example above that shows adding a node between two existing nodes without having to split and copy elements. In the end, this is just the policy that I chose to implement inserting. This will ensure that everyone will create the exact same lists. Also realize in our assignment, we are not doing the reverse operation: joining two adjacent nodes into a single node as we delete elements in the nodes. We are just removing elements until the node is empty, and then removing the node. A B-tree would join two adjacent nodes, if all the elements can fit into a single node. You should NOT merge any nodes for this assignment.

I'm not exactly sure how/when to split nodes during the call to the insert method. It seems that there is more than one way to distribute the values among the nodes.

Yes, there is more than one way. We are going to follow the technique used by B-trees. This means that when a new node is required, a "split" will occur. This will ensure that every new node is at least 50% full. Here's the partial output from test3_4 showing the split during the insert of the fifth element (97):

A full node containing 4 elements:

```
List: 56 59 79 87
Node   1 ( 4): 56 59 79 87
```

Inserting the value '97' causes the node to split, with the two smaller values in the 'left' node and the two larger values in the 'right' node. The inserted value (97) is placed in the third spot in the right node. This is the result after the split and insertion:

```
List: 56 59 79 87 97
Node   1 ( 2): 56 59
Node   2 ( 3): 79 87 97
```

The reason we split and don't just create a new node with 97 in it is explained in the previous question above.

If you look in the sample driver, you'll see that you can print out the list after each insertion. This will generate a lot more output, but it will let you see the list after each step. I've generated some output for you to diff with for these 3 insertion tests: out3_2[2] out3_4[3] out3_8[4]

What is the best way to approach this assignment?

The best way to approach this assignment is the same way you should approach all assignments: Break the problem down into manageable pieces. This is just another way of saying you should have lots of helper functions. Inserting an item into a BList is significantly more work than inserting an item into a regular linked-list. This is because there is a lot more to do. In a regular linked-list, you just find the location where the item belongs, allocated a new node, and hook up some pointers. It's only a few lines of code.

However, inserting into a BList requires more work that isn't necessary in a regular list:

- You first have to figure out which node should contain the item.
- You may not even need to allocate a new node if there is room in an existing node.
- You may have to shift several elements over to accommodate the new item.
- A node may be full and may need to be split into two nodes, which requires a new node to be allocated, then half of the data needs to be copied into the new node.
- Other work may need to be done.

As you can see, there may be several steps required to insert a new element into the BList. You may want to make helper functions to perform each of those steps. For example, you may want helper functions that:

- Given an index (logical index), find which node contains the element.
  - This function may even return two pieces of information: a pointer to the node, and an index within the node of where the item is (or belongs).

[2] data/faq_samples/out3_2.txt
[3] data/faq_samples/out3_4.txt
[4] data/faq_samples/out3_8.txt

- Split a node into two nodes where each node contains half of the items.
- Given a node and a value, insert the value into the proper position in the node. This will mean shifting of the existing elements.
- Any other steps that need to be repeated.

Realize that subscripting locates elements, not nodes. With regular linked-lists where each node contains one element, we can use the term nodes and elements interchangeably. However, given this diagram below:

```
[17,22,28,33] <=> [43,51,57,  ]
```

These are the values of subscripting the list:

```
blist[0] is 17
blist[1] is 22
blist[2] is 28
blist[3] is 33
blist[4] is 43
blist[5] is 51
blist[6] is 57

blist.size() is 7
```

Testing

As always, testing represents the largest portion of work and insufficient testing is a big reason why a program receives a poor grade. (My driver programs takes much longer to create than the implementation of the linked-list itself.) A sample driver program for this assignment is available. You should use the driver program as an example and create additional code to thoroughly test all functionality with a variety of cases. (Don't forget stress testing.)

Files Provided

The interface[5] and a minimal skeleton implementation[6]

Sample driver[7] containing loads of test cases.

Random number generator interface[8] and implementation[9].

Diagrams[10] to help your understanding.

There are a number of sample outputs in the **data** folder e.g.,

- all outputs (64-bit)[11]
- individual test files (64-bit) in **data/individual-tests**
- individual test files (32-bit) in **data/individual-tests-32bit**

Compilation:

These are some sample command lines for compilation. GNU should be the priority as this will be used for grading.

[5] code/BList.h
[6] code/BList.cpp
[7] code/driver-sample.cpp
[8] code/PRNG.h
[9] code/PRNG.cpp
[10] docs/diagrams.pdf
[11] data/output-sample-LP64.txt

GNU g++: (Used for grading)

```
g++ -o vpl_execution driver-sample.cpp PRNG.cpp \
    -Werror -Wall -Wextra -Wconversion -std=c++14 -pedantic -Wno-deprecated
```

Microsoft: (Good to compile but executable not used in grading)

```
cl -Fems driver-sample.cpp PRNG.cpp \
   /WX /Zi /MT /EHsc /Oy- /Ob0 /Za /W4 /D_CRT_SECURE_NO_DEPRECATE
```

Deliverables

You must submit your program files (header and implementation file) by the due date and time to the appropriate submission page as described in the syllabus.

BList.h

The header files. No implementation is allowed by the student, although the exception class is implemented here. The public interface must be exactly as described above.

BList.cpp

The implementation file. All implementation goes here. You must document this file (file header comment) and functions (function header comments) using Doxygen tags as usual. Make sure you document all functions, even if some were provided for you.