Project 3 - 2-3 Trees

Due: Please refer to Blackboard

Objectives

- Implement a data structure (2-3 Trees)
- Implement an iterator for a data structure
- Utilize functors to control the behavior of a data structure
- Use a balanced tree to speed up lookup times in large sets of data
- Write code to a specification that someone else will use
- Leverage some of the more advanced features of Makefiles

Introduction

By now you have had experience creating and using “linear” data structures. Although these types of data structures are relatively simple to design and implement, one of their shortcomings finding values can require traversing the entire collection, which can take a really long time. This is an undesirable quality, because time is money. A common optimization method for searching large collections of data (e.g. databases, filesystems) is to create indexes over certain metadata.

For this project, you will be creating indexes over a collection of songs in a music library. These indexes will allow you to quickly retrieve songs that have a particular artist, album, genre, etc. without you having to search through the entire collection. Each index shall be implemented as a 2-3 tree that stores pointers to the songs in the music library in some sorted order. You will also implement an iterator to facilitate traversal over your index.

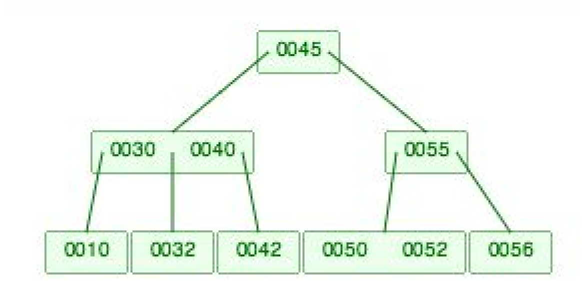
Similar to project 1, you will be implementing your 2-3 tree to a specification. As before, you must write solid code and test it extensively to ensure it works correctly.

In this project, you will also gain a deeper understanding of some of the conveniences of Makefiles and compilers that obviate programmers from having to correctly list out prerequisites and recipes for all of the components that comprise their project. These conveniences will be especially useful for this project, since it might contain more files than other class projects you have worked on.

2-3 Trees

Introduction

A 2-3 tree, like a binary search tree, is great for keeping data in sorted order. However, unlike a binary search tree, a 2-3 tree is capable of keeping itself balanced. All leaves in the tree are at the same level, and the height of the tree is always logarithmic in terms of the number of nodes in the tree. All nodes in the tree are either leaves with one or two keys, or are internal nodes with one key and two children, or internal nodes with two keys and three children. Nodes with 1 key are referred to as *2-nodes*, and nodes with 2 keys are referred to as *3-nodes*. Below is an example of a 2-3 tree:



In the tree above, the nodes with 10, 32, 42, 45, 55 and 56 are 2-nodes, and the nodes with 30/40, and 50/52 are 3-nodes. Notice the ordering of nodes with respect to their descendants, siblings and ancestors. You can play around with 2-3 trees on your own to get the hang of how they work with this interactive visualization. While this is technically a B-Tree visualization, keeping the max degree set to 3 gives you the same behavior as a 2-3 tree.

The 2-3 tree you will be implementing must be able to support two operations: searching and inserting. These behaviors will be controlled by a template parameter - `Compare`. This parameter is a functor (fancy name for "class with an overloaded function call operator") that is called to compare two data elements (or an attribute with some other value). Inside of your `Tree` class, to actually compare two items, you would use the following:

```
Compare()(item1, item2);
```

The empty parentheses actually construct an object of the `Compare` class, and the set of parentheses afterward with arguments actually calls the appropriate overloaded function. **The comparisons must return true if `item1` is less than `item2`, and false otherwise.**

Insertion

Insertion starts by finding the leaf at which the insertion should take place. To find the leaf at which to insert value `v`, for each node `n` you visit, you should use the following algorithm:

```
insert(n, v)
if n is a leaf, then we're done
else if n is a 2-node with key k
    if v < k, insert(n->left, v)  // Use Compare()(v, k) in your
code
    else, insert(n->right, v)
else if n is a 3-node with keys k1 and k2, where k1 < k2
    if v < k1, insert(n->left, v)
    else if v < k2, insert(n->middle, v)
    else insert(n->right, v)
```

Note that duplicates are allowed in our 2-3 trees - they just go down either the middle or right branch. Once you have found the correct leaf at which the insertion should take place, you need to actually perform the insertion, which can either be really simple, or more complex, depending on whether the leaf is a 2-node or a 3-node. If it's a 2-node, we're free to do the insertion right away. If it's a 3-node, you need to split the node, since no node can have more than 2 keys. These splits can propagate all the way to the root of the tree:

```
insertUp(n, v)
if n is a 2-node with key k
    insert v in the correct place with respect to k
else if n is a 3-node with keys k1 and k2, where k1 < k2
    find the middle key of k1, k2 and v; call it m
    split n into 2 nodes n1 and n2
    n1 gets the min and n2 gets the max of {k1, k2, v}
    insertUp(n's parent, m)
    n's parent's children to point to n1 and n2
```

You should experiment with the interactive visualization to completely understand how this works. Another great resource you might find helpful is this lecture summary from USC.

Searching

Searching in a 2-3 tree is not too different from searching in a binary search tree - it could be implemented recursively, and follows certain branches based on the relative order of the value being searched for and the value at the current node. However, you need to be careful since "duplicates" might exist in the tree. To find the first element equal to some desired value `v`, you should consider the following cases at each node `n`:

1. If `n` is empty, then `v` cannot exist in the tree, so return the `end` iterator.
2. If `n` is a 2-node with key `k`, and `v < k`, then search `n`'s left subtree for `v`
3. If `n` is a 2-node with key `k`, and `v == k`, then search `n`'s left subtree for `v`. If this search returns the `end` iterator, then return an iterator pointing to `k`
4. If `n` is a 2-node with key `k`, and `v > k`, then search `n`'s right subtree for `v`
5. If `n` is a 3-node with keys `k1` and `k2` where `k1 < k2`, and `v < k1`, then search `n`'s left subtree for `v`
6. If `n` is a 3-node with keys `k1` and `k2` where `k1 < k2`, and `v = k1`, then search `n`'s left subtree for `v`. If this search returns the `end` iterator, then return an iterator pointing to `k1`
7. If `n` is a 3-node with keys `k1` and `k2` where `k1 < k2`, and `v < k2`, then search `n`'s middle subtree for `v`
8. If `n` is a 3-node with keys `k1` and `k2` where `k1 < k2`, and `v = k2`, then search `n`'s middle subtree for `v`. If this search returns the `end` iterator, then return an iterator pointing to `k2`
9. If `n` is a 3-node with keys `k1` and `k2` where `k1 < k2`, and `v > k2`, then search `n`'s right subtree for `v`

Because our 2-3 tree can handle duplicates, we can't simply bail out of the search once we've found a data element with the value we're looking for - we need to see if there is another node to the left with a value equal to that data element.

Similar principles apply to finding the last element equal to a given key, except you would check branches to the right of where you find matches.

Iterators

Introduction

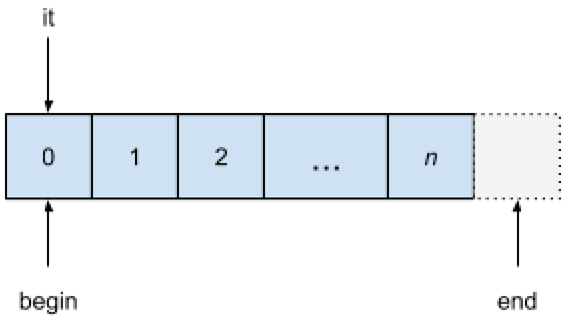
Iterators are useful for traversing through data structures. They are very similar to indexes in arrays - they know where they are in the structure, and they might know how to get to the next or previous item. Many of the STL containers have iterators associated with them. Iterators can simply be wrappers around pointers, but they could maintain additional state. Regardless of how they keep track of where they are, iterators have very simple interfaces, usually incrementing, comparison, and maybe decrementing and accessing offsets similar to indexing into arrays. Iterators are especially desirable beneficial for complex data structures because they abstract away the details of the underlying implementation.

Below is a simple `for` loop that prints out the contents of a `vector` of `int`s using an iterator:

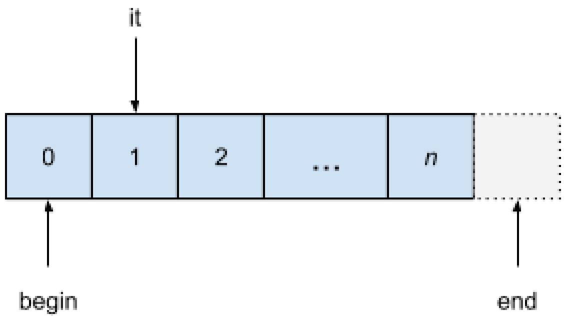
```
std::vector<int> v;
// Put some ints in v…
for (std::vector<int>::iterator it = v.begin(); it != v.end();
it++)
{
    std::cout << *it << std::endl;
}
```

The iterators returned by the `begin` and `end` functions are special - they point to the beginning of the vector and past the end of the vector, respectively. Note that the past the end iterator points to an element that isn't actually in the vector - other containers follow this same pattern. The iterator is incremented during each iteration of the for loop to advance through the vector.

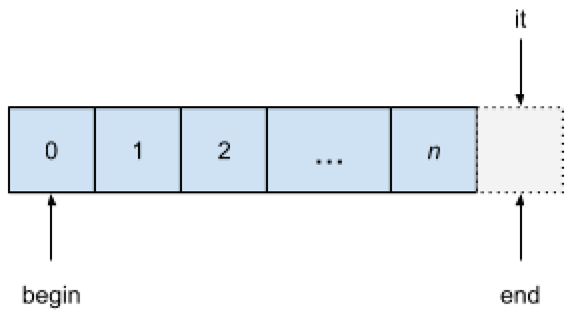
A conceptual illustration of the above iterator operating on a vector of *n* elements is shown below. The iterator starts off at the beginning of the vector:



After the iterator is incremented for the first time, the iterator points to the second item in the vector:



Eventually, the iterator advances past the end of the vector, and the loop terminates:



2-3 Tree Iterator

In general, your 2-3 tree iterator must be able to do two things: keep track of where it is in the tree, and find the successor of the data element it is currently visiting. You are free to decide how to maintain the iterator's state. When finding the successor of a data element, you should keep the following cases in mind:

1. If the current data element is the data element in a 2-node that has a right subtree, the successor is the smallest (or only) data element in the leftmost node in the node's right subtree
2. If the current data element is the data element in a 2-node that **does not** have a right subtree, the successor is located at the closest ancestor that is either:
    - A 2-node whose left subtree contains the current data element. In this case, the successor is the key in this 2-node
    - A 3-node whose left subtree contains the current data element. In this case, the successor is the smallest key in this 3-node
    - A 3-node whose middle subtree contains the current data element. In this case, the successor is the largest key in this 3-node
3. If the current data element is the smallest data element in a 3-node that has a middle subtree, the successor is the smallest (or only) data element in the leftmost node in the node's middle subtree.
4. If the current data element is the largest data element in a 3-node that has a right subtree, follow case 1.
5. If the current data element is the smallest data element in a 3-node that **does not** have a middle subtree, the successor is the largest data element in that same node.
6. If the current data element is the largest data element in a 3-node that **does not** have a right subtree, follow case 2.

You might find the interactive visualization provided above helpful to understand where to find successors.

Code Provided to You

We have provided you with the following files for this project:

main.cpp

We have provided you a very minimal driver that just creates one index over the song library. **You should expand on this file and perform your own testing.**

Song.h/Song.cpp

This is the `Song` implementation, and is not to be modified.

Library.h/Library.cpp

This is the `Library` implementation, and is not to be modified. It is just a readable collection of song metadata.

Node.h

Empty `Node` structure. These `Node`s will comprise your 2-3 tree. **You must implement the remainder of this structure.** They should, at a minimum, be capable of storing one or two data items, and have 0, 2 or 3 children. You may find it helpful to include a parent pointer in your `Node` structure, though it is not necessary.

Tree.h/Tree.cpp/TreeIterator.cpp

A largely empty tree class and its iterator. **You must implement all of the public methods declared in the class.** Although the documentation in the code briefly describes what the methods do, you should add to them any special side effects or pre/post conditions where applicable.

The `Tree` class is a templated class that takes the type of data it is storing, and the class that is to be used for comparing objects in the tree. Creating instances of this class with different `Compare` classes alters the layout of the tree.

The `insert` method of the `Tree` class should follow the algorithm given above. Based on the logic of the `Compare` class that's a template parameter, ties are broken by inserting to the right.

The `begin` method in the `Tree` class returns an iterator that points to the smallest element (i.e. smallest data item in the leftmost leaf) in the tree. The `end` method should return an iterator that indicates that it is outside of the bounds of the data structure.

The `find_first` and `find_last` methods in the `Tree` class return iterators that point to the "smallest" and "largest" elements, respectively, that have the specified key. If no item with the given key exists in the tree, these should return `end`. The `Tree` class' `find_range` method returns a pair of iterators - the first iterator points to the "smallest" data item that has the given key, and the second iterator points **past** the "largest" data item with the given key.

The overloaded stream insertion operator for the `Tree` class should print a level order traversal of the tree.

You must also implement the public methods in the iterator class. For the iterator, you must be able to determine whether or not two iterators are pointing at the same data item through the equality operators. You should also be able to dereference iterators to retrieve the data at which they are pointing. You also need to implement the logic for advancing an iterator to a data item's successor in the overloaded `operator++`.

TitleCompare.h/TitleCompare.cpp

A declaration for the functor that compares titles. Calling the overloaded function call operator on this class should return true if the first argument is less than the second, and false otherwise. **You will need to implement the comparison logic in the methods yourself.** Note that this functor should only compare titles, and no other part of the `Song` structure (i.e. no breaking ties by looking at artist, etc.).

You must also implement functors for artist, album, genre and year. Artist, album and genre should do lexicographic comparison, and year should follow numeric comparison.

Setting up Your Makefile

Makefiles are about as powerful as other programmings languages like C++, Python or Java. Here, you will learn how to leverage some of their convenient, albeit cryptic, automation features. It might help to have the GNU Make Manual handy as you work through this. The first thing we'll do is set up the variables in the Makefile. We'll start with the set of variables that tell `make` which compiler and linker to use:

```
LD = $(shell which g++)
CXX = $(shell which g++)
```

You must name these variables `LD` and `CXX`, as they are variables built into make for special purposes. Specifically, `LD` is the path to the linker to use, and `CXX` is the path to the C++ compiler to use. The right hand side of the assignments, `$(shell which g++)` tells `make` to execute the command `which g++` in the shell and return the command's output (if you're unfamiliar with the `which` command, you should use the `man` command to find out what it does).

Next, we'll define common flags for the C++ compiler (another builtin variable, `CXXFLAGS`):

```
override CXXFLAGS += -Wall
```

The `override` directive allows `make` to take a value for this variable in on the command line, and adjust its value in the Makefile. In our case, we're adding `-Wall` to the compiler flags (note the `+=` operator). If you were to run `make CXXFLAGS=-O2`, inside the Makefile after the `override CXXFLAGS += -Wall` line, the value of `CXXFLAGS` would be `-O2 -Wall`.

Now we'll define variables for the executable and objects in your project. **If you write the rest of your Makefile correctly, these two variables will be the only things you will regularly need to change in order to reuse this Makefile for other projects** (at least for this class):

```
EXE = Project3.out
OBJS = main.o Song.o Library.o TitleCompare.o
```

You will probably need to add more object files to what is listed above if you create additional `.cpp` files. Remember not to include `.cpp` files that are implementations of templates. These are `#include`d into the other files that use them.

We'll next specify the names of our *dependency files*. You're already familiar with writing out targets, dependencies and recipes in their entirety for every single object file that comprises your program. You might have found this process tedious, and you've probably run into a situation in which you've forgotten a dependency and not enough of your code compiled (if you need to run `make clean` or `make -B` every time you make a change to a single file just to get even one part of your code to compile, that's why). Instead of us humans writing out a recipe for every object file (which can get out of control for projects with hundreds or thousands of files), we're going to get the compiler to (correctly) generate these dependencies for us and store them in dependency files.

```
DEPS = $(addprefix .,$(OBJS:.o=.d))
```

Starting with the inside of the expression, `$(OBJS:.o=.d)` tells `make` to take all the values stored in the `OBJS` variable, and replace the `.o` with a `.d` in each value. The result of this operation, assuming you defined `OBJS` as above, is `main.d Song.d Library.d TitleCompare.d`. The `$(addprefix ., `*`args`*`)` function adds a `.` before each value in *`args`*. We do this to keep these dependency files from cluttering up our directory. The final result stored in the `DEPS` variable is `.main.d .Song.d .Library.d .TitleCompare.d`.

Now it's time to write the rules for the Makefile. We'll start with the rule that does the final linking:

```
$(EXE): $(DEPS) $(OBJS)
        $(LD) $(LDFLAGS) $(LDLIBS) $(OBJS) -o $@
```

This rule first says that we're going to rebuild our executable if any of our dependency or object files have been updated since we last ran `make`. The way we do that is by executing our linker (the `LD` variable), and passing the linker any flags or extra libraries we need to link with the program (builtin variables `LDFLAGS` and `LDLIBS` respectively), and our object files. The name of the output file is specified after the `-o` flag - in this case it's another special variable in `make`. `$@` is a shortcut to refer to the target name of the current rule, in this case it's the value `$(EXE)`.

Next we'll create the rule that automatically generates our dependency files:

```
.%.d: %.cpp
        @$(RM) $@; \
        $(CXX) -MM -MF $@ -MT "$(<:.cpp=.o) $@" -c $(CPPFLAGS) $<
```

There's a lot going on in this rule. If you actually want to see it in action, you can remove the `@` before the `$(RM)` in the recipe so the command gets printed. The rule (a special type of rule called a static pattern rule) says that each dependency file depends on a C++ source file that shares a similar name, disregarding the leading dot and the file extensions.

The first thing we do when building the dependency file is remove the existing dependency file if it already exists. Then, we use the compiler to generate a list of prerequisites based on all files that are `#include`d (directly or indirectly) in the `.cpp` file, and store it in a dependency file (this is the `$(CXX) -MM -MF $@` part). We also tell the compiler that the target for the dependency that it generates is to be the `.o` file and the dependency file. By default, the compiler just gives us the `.o` file as the target, but we would also like to be able to rebuild the dependency file in the event one of its `.cpp` file's prerequisite files changes. Also of note, the `$<` variable is the first prerequisite in the rule's prerequisite list.

For example, if you have a file `bob.cpp` that `#include`s `bob.h`, and `bob.h` `#include`s `jimmy.h`, a `make` rule that would accomplish the same as above (without variables) would be:

```
.bob.d: bob.cpp
        rm -f .bob.d; \
        g++ -MM -MF .bob.d -MT "bob.o .bob.d" -c bob.cpp
```

with the dependency file, `.bob.d`, containing:

```
bob.o .bob.d: bob.cpp bob.h jimmy.h
```

One more compilation rule we need is a rule describing how to build the object files. For convenience, this is another static pattern rule:

```
$(OBJS): %.o: %.cpp
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@
```

Now that all rules related to building our executable have been specified, we need to do one last thing: include the dependency files into the Makefile. This is similar to `#include`ing a header file into a C++ source file. You can include the dependency files by doing:

```
-include $(DEPS)
```

Yes - that hyphen is necessary, as it tells `make` to try to keep going and include all dependency files, even if something goes wrong with including one of them.

We can now finish the remainder of the Makefile. You should have a `clean` target, and, since we insist on including a `run` target for making grading easier, we'll add these rules to the Makefile:

```
.PHONY: clean
clean:
        $(RM) $(OBJS) $(DEPS) $(EXE)

.PHONY: run
run: $(EXE)
        ./$(EXE) $(LIBRARY_FILE)
```

The `.PHONY` targets indicate to `make` that the `clean` and `run` targets are not to be confused with files. Since the project takes a command line argument for the library file, we specify that in the `run` rule.

As promised, this Makefile is quite complex. You are encouraged to practice with it and experiment on your own and discover useful tricks of your own.

Hints, Notes and Requirements

- You will need to exercise *your* debugging skills for this project. Because of the large number of students in this class, and the complexity of this project, you will receive **very limited** help if you come to the staff with logic errors in your code.
  A great debugging strategy is to put debug print statements throughout your code such as the following to observe how your variables are changing:

```
#ifdef DEBUG
    std::cout << "Left child: " << node->left << std::endl;
    std::cout << "Right child: " << node->right << std::endl;
#endif
```

To have these statements print out, compile your program with:
`make CPPFLAGS=-DDEBUG`
If you're feeling really daring, you can also try out gdb or your favorite IDE's debugger.

- Your runtime for insertion and searching in your 2-3 tree must be *O(*lg *n)*, where *n* is the number data items in your 2-3 tree.
- All input files used in testing are guaranteed to be valid. Don't worry about the minutiae of parsing data files.
- Your 2-3 tree must be able to store arbitrary types. If your tree can only handle `Song` pointers, you should rethink your design.
- You are free to add methods and data members the `Node` struct, `Tree` class and `Tree` class' `iterator`. Feel free to add your own classes to support your data structure a well. Much of the design of this project is up to you.
- You are permitted to use **linear data structures** from the STL (`std::vector`, `std::deque`, `std::list`, `std::stack` and `std::queue`). Use of any other containers from the STL will incur significant penalties.
- Insertion into the 2-3 tree has two distinct parts - finding the leaf where the insertion should happen, and splitting nodes on the path to the root. It might help to separate this logic in your code to make it more comprehensible.
- Your code should not leak memory. Be sure you are freeing all memory that you allocate. To ensure that you are not leaking memory, run your program with valgrind.
- Level order printing can be accomplished with the help of a queue.
- Playing with the interactive visualization mentioned above will help you understand how 2-3 trees operate.
- Your driver file should be named **`main.cpp`**, your Makefile should be named **`Makefile`**, and the executable program should be named **`Project3.out`**. The files for your comparison functors for artist, album, genre and year should follow the naming scheme **`<Attribute>Compare.h`** and **`<Attribute>Compare.cpp`**, similar to **`TitleCompare.h`** and **`TitleCompare.cpp`**.

What to Submit

Follow the course project submission procedures. You should copy over all of your C++ source code (including those distributed with the project) with .cpp/.h files under the `src` directory. You must also supply a Makefile that builds your project that adheres to the instructions above.

Make sure that your code is in the `~/cs341class/proj3/src` directory and not in `~/cs341class/proj3`. In particular, the following Unix commands should work:

```
cd ~/cs341class/proj3/src
make
make run LIBRARY_FILE=library.txt
make clean
```

Don't forget the Project Submission requirements shown online. After you submit, you should check that after running the command `ls ~/cs341class/proj3` the only file listed is the `src` directory. The C++ source code must be located inside the `src` directory. Your submissions will be compiled by a script. The script will go to your `proj3` directory and run your Makefile. This is required. You will be severely penalized if you do not follow the submission instructions.