Programming Assignment 1:
Representing, Managing and Manipulating Travel Options

DUE: Saturday, Jun 26 by 11:59PM

Summary: you have been given a partially implemented C++ class called `TravelOptions` (file: `TravelOptions.h`). The class implements an ADT using singly-linked lists for which some functions are already written and some are not. Your job is to complete the unwritten functions.

Partially implemented `TravelOptions.h` and a toy driver program can be found in the `src` folder.

IMPORTANT NOTE (compiler): we will be using the `C++11` standard for all programming assignments this semester. When running `g++`, this means you want to use the `-std=c++11` flag. If you are using some other environment/IDE, spend some time to make sure it is configured to compile with the `C++11` standard.

A 2nd IMPORTANT NOTE (warmup exercises): There is a second handout called `WarmupExercises`. It is the topic of your 2nd lab and in it you apply the concepts laid out below by working through some pencil and paper exercises.

A 3rd IMPORTANT NOTE (test cases and late submissions): After the submission deadline passes (or perhaps 24 hours prior...), a full set of test cases will be released. You can then run your code on the test cases. If you are not happy with the results, you can continue to work on your implementation and take advantage of the late policy as stated in the syllabus (up to 36 hours late for a 15% penalty). Prior to the release of the test cases, it is up to you to "stress test" your code -- if you do a good job, you should pass most/all of the official test cases; otherwise, you will have learned a lesson about the importance of testing and that it is usually the programmer's responsibility, not an external entity!

Contents

Contents

Conceptual Background
- Comparing two Options
- Collections of Travel Options

Assignment Details
- Nested Types
- Data Members (private)
- Given Functions
- TODO Functions: comparison and checker functions
- TODO Functions: modification/construction operators (harder?)
- More Pre-Written Functions
- Additional Rules, Degrees of Freedom and Reminders

Recommendations

Sample Toy Driver Program

Scoring

Submission Details

Conceptual Background

Suppose you are planning a trip and have gathered a number of different options. Each option has two traits which you care about (and these are the only traits you care about -- who cares about comfort and stuff like that?):

```
    Price
    Travel time
```

Let us represent an option as an ordered-pair `<price, time>`. Now consider two options:

```
    A: <100, 120> : for $100, you can get to your destination in 120 minutes.
    B: <200, 90>  : for $200, you can get to your destination in 90 minutes.
```

Which one should you pick? In this case, it might depend on how rich, impatient and cheap you are. Why? Because, for these two options, we can see there is a tradeoff: there is a reduction in travel time by spending more than $100.

On the other hand, consider these two options C and D:

```
    C:  <100, 110> : for $100, you can get to your destination in 110 minutes.
    D:  <105, 130> : for $105, you can get to your destination in 130 minutes.
```

Option C is both cheaper ***and*** faster than option D. Thus, nobody in their right mind would select option D over option C (again, price and time are the only criteria being considered). In this situation, we say ***"C dominates D"***.

Comparing two Options

Suppose we have two options A and B: `<pA, tA>` and `<pB, tB>`. If we ask *"How does A relate to B?"*, there are four possible answers:

`equal`	A and B are identical in both price and time.
`better`	A is better/preferred over B (A dominates B): A and B are not equal/identical ***and*** option A is no more expensive than option B ***and*** option A is no slower than option B.
`worse`	A is worse than B (B dominates A): A and B are not equal/identical ***and*** option B is no more expensive than option A ***and*** option B is no slower than option A.
`incomparable`	everything else -- one option is cheaper and the other is faster

(One of your first tasks will be to write a function which takes two options and determines which of the above cases holds.)

Aside: the "dominates" relation among a set of options is a ***partial order*** which you may recall from CS151.

Collections of Travel Options

Now consider a list of options -- i.e., a list of `<price, time>` pairs. Such a list may or may not have certain properties.

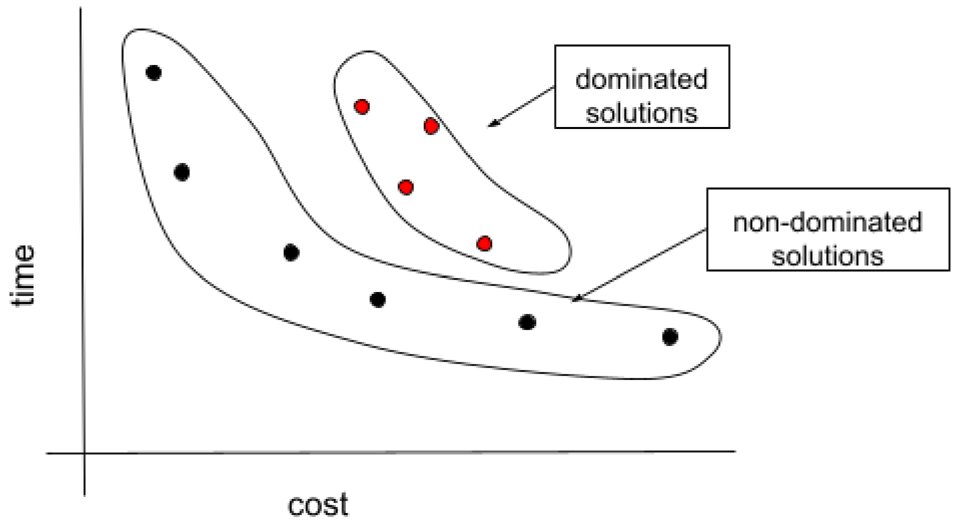
`sorted`: we will say a list of options is sorted if the elements are

- in non-decreasing order of price
- elements with identical price are ordered according to time -- i.e., time is used as a tie-breaker.
- (if two entries are identical in both price and time, of course, they just have to appear adjacent to each other).

`pareto/non-dominated:` If a collection of options has the following properties, we will say it is "pareto" or is a "pareto-frontier"[1]:

- It contains no duplicates -- i.e., it is a set
- It contains only *non-dominated* solutions -- i..e., for every option A in the collection, there does not exist another option B which dominates it.

Example: Consider the plot of a collection of 10 options below. This collection (of all ten options, both black and red) would be considered non-pareto because of the existence of the four dominated options (colored red).



If our collection included only the black options, we would say that the collection is pareto/non-dominated.

You can see that the black options form a "tradeoff curve" -- sometimes called a "pareto curve".

General Observation: In general, a list of options must be in of these "states":

- not sorted and not pareto
- sorted, but not pareto
- not sorted, but pareto

[1] For an idea of where the term "pareto" comes from: http://en.wikipedia.org/wiki/Pareto_efficiency

- both sorted and pareto

Many functions/operations have "preconditions" on the lists they operate on. Some functions operate on any old list of options; others require one or more option list to be both sorted and pareto; others may take a sorted, but not necessarily pareto list and turn it into a sorted, pareto list. And so on...

Assignment Details

You will complete the implementation of a C++ class called `TravelOptions` which uses linked lists to represent individual options. Some basic operations have been provided to you, but you will have to complete the implementation of several functions (for which "skeletons" are given).

The following sections give an overview of the class, pre-implemented functions and functions you will need to implement.

Nested Types

Type	**Description and Comments**

`Relationship`	`public enumerated type capturing the possible relationships between two options.  Values:`
	
	`{better, worse, equal, incomparable}`
	
	`comment:  return type for the compare functions`

`Node`	`private struct for singly-linked list node; represents a single option (fields:  price, time, next)`

Data Members (private)

Member	**Description and Comments**
`Node *front;`	`pointer to first node in option list (null if list is empty).`
`int _size;`	`tracks list length (note:  many of your functions will have to set/update this data member).`

Given Functions

These functions have already been implemented for you! They are ready to use.

Function	**Short Description (see banner comments for details)**
clear	empties the option list (the object still exists)
size	returns the number of options in the calling object
push_front	adds new option to front of list. Price and time of option are passed as parameters. simple utility function for building lists.
from_vec	utility function for creating an TravelOptions object from a vector of <price,time> pairs. Uses the pair class from the C++ STL. Note: this is a static function -- no calling object. See banner comment for more info..
to_vec	Inverse of from_vec. Returns pointer to vector of STL pairs populated from the calling object.
display	prints the options to the terminal (of course, in the order they appear in the list).
checksum	you are not allowed to touch this function! See banner comment if interested.

TODO Functions: comparison and checker functions

This first group of functions shouldn't be too terrible! Start with these.

Function	Short Description (see banner comments for details)	Comments	Points
compare	static function (no calling object) which takes two options (via four parameters) and determines their relationship. Returns one of the four possibilities (see enumerated Relationship type above).	Should be an easy one to knock out first! Runtime: constant	10
is_sorted	determines if calling object is sorted according to rules laid out above	Runtime: linear	10
is_pareto	determines if travel options are pareto	Runtime: quadratic	10
is_pareto_sorted	determines if options are both pareto and sorted	Runtime: linear Can't just call the two previous functions because of a runtime requirement! Still not that hard.	10

TODO Functions: modification/construction operators (harder?)

Function	Short Description (see banner comments for details)	Comments	Points
insert_sorted	inserts new option (passed via parameters) into a sorted option list.	precondition: calling object must be sorted (returns false if not). Runtime: linear	15
insert_pareto_sorted	Takes new option and, if it is not dominated by a pre-existing option, inserts it and, in turn deletes any pre-existing options which have become dominated.	precondition: list must be both sorted and pareto (returns false if not). Runtime: linear	20
union_pareto_sorted	Takes two lists (calling object and a parameter) and constructs their "pruned union" as a new list. Returns new TravelOptions object as a pointer.	preconditions: both calling object and parameter must be pareto and sorted (returns null if not) postcondition: returned object must also be pareto and sorted. In general, it will be a subset of the traditional set-union of the two lists. Runtime: Linear.	20
prune_sorted	takes sorted option list and deletes all dominated elements (if any). Of course only deletes dominated options.	precondition: calling object must be sorted (returns false if not). postcondition: calling object is both sorted AND pareto. Runtime: Linear.	20

Function	**Short Description (see banner comments for details)**	**Comments**	**Points**
join_plus_plus	Takes two option lists (calling object and a parameter). One list gives options for the first leg of a trip; the other gives options for the second. Constructs and returns a pareto-sorted option list for the entire trip.	preconditions: none really - both lists are arbitrary (not necessarily sorted; not necessarily pareto). Not as bad as it sounds... see banner comments! No runtime requirement.	20
join_plus_max	Takes two option lists (calling object and a parameter). One list gives options for traveler A and the other gives options for traveler B. Constructs and returns the pareto sorted option list for A and B traveling concurrently and so "time" is the maximum (latest) of A's time and B's time. Read banner comment for details.	preconditions: both lists expected to be pareto-sorted (null returned if not). postconditions: returned list is pareto-sorted. Runtime: linear(!) This one may take some thought! Some tips given in banner comment.	30
split_sorted_pareto	takes max_price as a parameter. Splits option list into options with price no greater than max_price and those greater than max_price. The cheap options are retained in the calling object. The expensive options are used to populate a new TravelOptions object with is returned as a pointer.	preconditions: calling object expected to be sorted and pareto (null returned if not). postconditions: both calling object and returned option list are sorted and pareto. Runtime: linear Additional Requirement: No allocation of new nodes allowed!	15

More Pre-Written Functions

These functions have been written, however, they depend on one or more of the functions that are your responsibility.

Function	**Short Description (see banner comments for details)**
`compare(Node*,Node*)`	This is a private, static function which compares the options stored in two Nodes by calling *your* compare function. You might find it convenient...
`sorted_clone`	creates a new TravelOptions object with exactly the same entries as calling object but in sorted order. Returns new options list as a pointer. Calls *your* insert_sorted function. Might be handy...especially for writing test cases.

Additional Rules, Degrees of Freedom and Reminders

You may NOT do any of the following:

- change any of the function names or signatures (parameter lists and types and return type)
- introduce any global or static variables
- use any arrays or vectors inside the `TravelOptions` class! Not for this assignment!! (You may use them as you see fit in any driver/tester programs you write).
- cheat.

You MAY do any of the following as you see fit:

- Introduce helper functions. But you should make them private.

Reminders:

- Some functions eliminate list entries (e.g., `prune_sorted`). Don't forget to deallocate the associated nodes (using the `delete` operator).

Recommendations

- Start with the easier functions.
- The assignment is very modular -- you can write and test most functions independently. In other words, don't try to implement *all* functions before doing any testing. Write a function then test and debug it before moving on.
- Do lots of testing! Write multiple driver programs which perform lots of stress tests on the `TravelOptions` class. Think carefully about boundary cases, etc.! Leverage the given functions in writing your test cases (for example, `from_vector` should make it pretty easy to hard-code some test cases -- see `toy.cpp` for a simple example).
- Test programs should be in separate files (not inside `TravelOptions.h`).
- Try to be systematic about your testing -- for example, even if you think a test case has been satisfied, do not simply overwrite that test program with another! Keep it so you can re-test as you continue to develop. Make as many independent test programs as you need (and give them meaningful names!)
- Look out for memory leaks!

Sample Toy Driver Program

As stated above, you will be writing driver programs to test your implementation. For reference a toy driver program has been given (filename: `toy.cpp`). It is just a pretty random sequence of invocations of `TravelOptions` functions, just so you have an idea of how to write a driver program. Compilation is simple:

```
g++ -std=c++11 toy.cpp
```

Scoring

The individual functions have been assigned points in the tables above. The points received will be determined via testing and include runtime tests! Some of these tests will be released prior to the submission deadline.

The point total over all functions is 180. In addition, each submission will receive up to 90 points for having made an "honest effort." This does ***not*** mean you can simply turn in the `TravelOptions.h` file as it has been given to you and expect to receive these 90 points! Similarly, **if your code does not even compile, don't expect to get many of those 90 points.** But, if you submit compilable code but end up with some functions that fail all/most test cases despite your best efforts, all is not lost!

Submission Details

Your only deliverable will be your `TravelOptions.h` file. Stay tuned for submission details (it will be done either via blackboard or gradescope...)