Problem Set 6: Tron

Assigned: April 11, 2013 Due: May 2, 2013

Design meetings: April 14 - 18

---

1 Updates.

- 4/18: Corrected description of "admissibility" in A*.
- 4/22: Added "Corner cases" section. Clarified initial rider positions.
- 4/27: Added a hyperlink to documentation guidelines.

2 Introduction.

In this problem set, you will develop a game called *Tron*. There are few constraints on how you can implement this project, and you will have greater freedom. However, this does not mean you can abandon what you have learned about abstraction, style, and modularity. Rather, this is a chance for you to demonstrate your grasp of all three.

You should begin by carefully designing your system. You will be required to attend a *design meeting* that is worth part of your score, where you will present and discuss this design with a course staff member.

On TBA, there will be a tournament where you may submit your bot to compete with other students' bots. There will be free food, and the winners will receive bragging rights and have their names enshrined eternally in the CS 312/3110 Tournament Hall of Fame.

Note that there is also a **written problem** in this problem set. It can be found at the end of this document.

2.1 Reading this document.

The non-native types discussed in this text are defined in `definitions.ml`. Constants follow the naming convention of beginning with a lowercase `c`, and having the rest of the name in all caps. For instance, `cTIME_LIMIT` specifies the time limit for the game. The reason for this mysterious `c` is that OCaml doesn't allow value names to begin with capital letters – only type constructors can do that. It may help to have these files open for reference, as you read the writeup.

Updates to this writeup will be marked in red.

2.2 Point breakdown.

1. Design meeting: 5 points
2. Game: 30 points
3. Bot: 25 points
4. A-star: 10 points

5. Documentation: 10 points
6. Written problem: 20 points

3 Rules.

Tron is played in a two-dimensional, top-down perspective. Each player begins with a team of riders, which move around a grid in straight lines. As a rider moves, it leaves a *tail* on the spaces it has traversed, which acts like a wall. When any rider either hits a tail or goes out of bounds, it is removed from the game. The players direct the movements of all their riders at once. Once a player has lost all of their riders, they lose, and their opponent wins. The goal is therefore to survive longer than your opponent.

3.1 The board.

The board is a grid whose dimensions are given by `cNUM_ROWS` and `cNUM_COLUMNS`. A tile on the grid is identified by coordinates `(col, row)`. The row index varies on the vertical axis, increasing as you move downward, and the column index varies on the horizontal axis, increasing to the right. All indices are zero-indexed. `(0,0)` corresponds to the top-left corner of the board.

3.2 Riders.

The *rider* is the basic unit controlled by a player. At any point in time, an active rider (one that has not yet crashed) is represented as a record of five fields.

- `id`: A unique integer used to refer to this rider.
- `orientation`: The direction the rider is currently moving in. This is any one of `North`, `South`, `East`, and `West`. In the absence of any commands to change orientation, the rider will continue moving in this direction.
- `tile`: The location of the rider, given by coordinates as described in the previous section.
- `modifier`: Information about what special effects are currently active on this rider. Described more in detail in the *Items* section.
- `invincibility_timer`: Described in the *Items* section as well.

3.3 Tails.

As stated, riders leave tails on every space that they move across. Tails act as walls. When a rider enters the same space as a tail, unless it has an item effect active, it crashes and is eliminated from the game. A tail itself has no particular attributes; all that needs to be recorded is the tile it is placed on.

For instance, if a rider is on square `(4,2)`, and it moves to square `(4,3)`, then a tail will be placed on `(4,2)`. Future riders will then not be allowed to enter square `(4,2)`, and those that attempt to do so will be eliminated (except under specific circumstances).

Note that the tail is not placed until *after* the rider leaves the square.

3.4 Items.

There are two items in our game.

The first is a **shield**. After a rider activates a shield, the first time it collides with a tail, rather than the rider being killed, the tail will be removed and the rider will survive. After the first such collision, the shield effect will be removed. In other words, **shields only last for one collision**. Additionally, shields do not protect riders from being killed by going out of bounds. However, shields have no time limit; they last until the first time the rider hits a tail.

The second item is **invincibility**. Once a rider activates invincibility, it cannot be killed by tails, though it will still die from going out of bounds. Invincibility *does* have a time limit, which is the constant `cINVINCIBILITY_TIME`.

For each rider, the record field `modifier` should be used to record whether it has a shield, invincibility, or both active. If invincibility is active, then `invincibility_timer` should be used to track how much time it has remaining. Note that this timer is measured in game steps, not seconds.

If a rider has both items active, then the shield cannot be lost until after the invincibility expires. In other words, the invincibility effect has priority over the shield.

3.5 Collecting and using items.

Riders do not have individual inventories; instead, the number of items possessed is an attribute of the entire team. There are separate counters for shields and invincibility. Each team begins with a number of each, as defined by the constants `cNUM_INITIAL_TEAM_X`, where `X` is either `SHIELD` or `INVINCIBILITY`.

Beyond these, there are initially a number of items placed on the field, determined by `cNUM_INITIAL_FIELD_X`. The items are placed on random squares. Riders can collect one of these items by simply moving onto the space. The team's count for that item will then be increased by one, and the item will be removed from the field.

Any rider on a team can use an item if and only if that team possesses at least one of that item at that time. The team's item count is then decreased by 1, and the effect is applied.

3.6 Winning.

When a team's riders have all crashed, that team loses and the opposing team wins. If this doesn't happen within the time limit, then the winner is the team with more riders surviving at the end. If both teams have an equal number of riders, or they both lose their last rider at the same time, then the game is a tie.

4 Game module.

The module in `game.ml` is responsible for implementing all of these rules. The game server will initially call these functions, in order.

1. `initGame`: This initializes and returns a value of type `game`. This game's state should reflect that the game has not yet begun.

The column coordinate of the initial riders should be `1/10 * cNUM_COLUMNS` for Red, and `9/10 * cNUM_COLUMNS` for Blue. The riders on one team should be spaced equidistantly from each other and also from the walls.

2. `initFieldItems`: This places items on random spaces. You should place a number of shields and a number of invincibilities, defined by the relevant constants `cNUM_INITIAL_FIELD_X`, all on different random spaces.

After these have been called, the game will begin. The server will call these functions to advance the state of the game. For all of these functions, the argument `g` is the game object that was first created.

- `handleAction`: This is called when the server receives a command from one of the AIs. The argument `act` is the command sent by the AI, and `c` is that AI's color. The game state should be updated to reflect the command that was just sent.
- `handleStatus`: This is called when the server receives a request for data from an AI. The AI can request the status of the game's riders, items (both owned and on the field), and tails. The argument `status` is the request sent by the AI. The requested information should be returned.
- `handleTime`: This is called by the server at regular intervals (the precise timing is given by `cUPDATE_TIME`, in seconds). On each call, the game should be updated by one time step, as described below.

In one game step, you should perform the following, in order.

1. Check to see if the game is over. The game is over if it has gone on for `cTIME_LIMIT` (which is in seconds), or one team has lost all its riders. If the game has ended, determine a winner. Otherwise, continue.
2. Update the positions of all riders, and place a tail on the square each rider was previously occupying. Every rider moves one space in the direction it is facing. North corresponds to up, and the other directions follow from that.
3. Remove riders that have moved out of bounds. Additionally, for each rider that has moved onto a tail, if the rider does not have a shield or invincibility active, remove that rider. If it does have a shield or invincibility, then remove the tail instead. In this case, if the rider was using a shield, remove the shield effect.
4. If two riders move onto the same space, then if one has an active item and the other does not, then only the one with the item survives (and if the item was a shield, it is removed). If one has a shield and the other has invincibility, then the one with the shield dies and the invincible one survives. In all other cases, they both die. If both die, no tail will be placed on that square, and no item will be collected from that square.
5. Of the surviving riders, check if any of them have moved onto a space with an item. For each one that has, remove the item from the field and add it to the team's count.
6. Update invincibility timers. The timer is in game steps, so simply subtract one from each active timer. If a rider's timer hits zero, then remove the invincibility effect. If that rider is on a tail when its invincibility ends, then it survives for this turn.

4.1 Corner cases.

- If more than 2 riders move onto the same space, consider them pairwise. If a rider dies from any of the pairs, then it dies.
- If two riders both with shields or both with invincibility move onto the same space with a tail, they both die and the tail is removed.
- If two riders, one with shield and one with invincibility, both move onto the same space with a tail, then the one with invincibility survives and the tail is removed.
- If one rider with a shield and one rider with no item move onto a tail, they both die and the tail is removed.

5 Communication.

Our game uses a client-server framework. What this means is that the game server and the players (clients) are running as separate processes. The server is responsible for receiving commands from the players, and updating the game state. The clients are allowed to keep track of whatever information they want, but they need to send commands to the server in order to perform actions or receive data. The protocols for this communication are given by the type `command` in `definitions.ml`.

5.1 Action commands.

These are commands that bots may send in order to cause some change in the state of the game. These commands are of type `action`. On the bot's side, they are sent by calling `send_action`. Then, on the server side, `handleAction` is called with the received command as the argument. The server should then perform the action, and return either `Success` or `Failed` based on the result.

There are two possible commands of this type.

- `ChangeOrientation(id, orientation)`: Tells the server to change the orientation of rider `id` to the new `orientation`. Fails if that id doesn't exist, or if the rider doesn't belong to the player who sent the command.
- `UseItem(id, item)`: Tells the server to use `item` on rider `id`. Fails if the id doesn't exist, or if the team doesn't have any of that item.

These commands should take effect instantaneously; the rider's state should immediately be updated to change orientation or have an active item.

5.2 Status commands.

The client is also able to send certain commands to request information from the server about the game state. These commands are of type `status`. The bot sends these requests by calling `get_status`, and the server receives them with a call to `handleStatus`, which should return the appropriate `Data`.

There are five possible commands of this type.

- `TeamItemsStatus(color)`: The server should respond with a list of numbers of items owned by the team `color`.
- `FieldItemsStatus`: The server should respond with the list of all items on the map, along with their positions.
- `TeamStatus(color)`: The server should respond with the list of all surviving riders of the team `color`, as well as the items possessed by that team.
- `TailStatus`: The server should respond with the list of positions for all tails on the map.
- `GameStatus`: The server should return all information about the game: data for both teams, as well as for field items.

6 GUI.

The GUI client for this game is written in Java, and can be found in the `gui` directory. It has already been written for you. We have supplied you with a file `gui_client.jar`, which you should run using `java -jar gui_client.jar` to start the GUI.

The GUI does not update on its own – instead, the game server is responsible for sending graphical updates to the GUI. The server will automatically listen for GUI clients to send updates to. However, if a GUI connects midway through the game, it will have missed the `InitGraphics` update. This means that you need to connect the GUI before the bots.

6.1 Sending messages.

We have provided you with a module `Netgraphics` with functions to send updates to the GUI. Of these functions, `init` and `send_updates` are both called automatically by the server, so you will not need to use these yourself.

Generally, in order to send a graphics update, you should call `Netgraphics.add_update` with the `update` type you wish to send. This buffers the update so that it will be sent to the GUI at the next time step. The one exception to this is `InitGraphics`, which needs to be sent by itself, and should be sent directly using `Netgraphics.send_update` instead.

6.2 Graphics commands.

Command	Arguments	Meaning
InitGraphics		Tells the GUI to initialize an empty board, in preparation for a new game.
UpdateInventory	color, item_data	Updates the display to show that team color has a quantity of items as indicated by item_data.
UpdateRider	id, orientation, tile	Updates the rider with id to be at location tile, facing orientation.
ModifyRider	id, modifier, flag	Applies or removes an item's visual effect on rider id. modifier is the effect to toggle. flag specifies whether to turn it on (if true) or off (if false).
PlaceItem	item, tile	Adds the specified item to the board, at location tile.
PlaceRider	id, tile, color	Adds a rider with the specified color to the board at location tile, assigning it the given id. This id will be used by the GUI to refer to this rider in the future.
PlaceTail	id, tile, color	Adds a tail with the specified color to the board, at tile. The id belongs to the rider that generated the tail.
RemoveItem	tile	Removes the item from tile.
RemoveRider	id	Removes the rider id from the board.
RemoveTail	tile	Removes the tail from tile.
Countdown	num	Displays a countdown timer on the screen, starting from num and counting down.
GameOver	result	Displays that the game as ended, and that result was the outcome.

7 A* search.

7.1 Review of Dijkstra's algorithm.

In designing your AI to play the game, you may want to be able to find shortest paths between tiles – from tile s to tile t, for instance. Dijkstra's algorithm, which you've almost certainly seen before, is the most well-known algorithm for finding such paths.

The key to the algorithm is that, at every stage, every node n has an associated value $g(n)$, which is the shortest known path from s to n; if no path is known, $g(n) = \infty$. Thus, initially, $g(n) = \infty$ for all nodes, except for $g(s) = 0$.

At every step, we choose the unexplored node with the lowest $g(n)$ to expand; for every neighbor m, we update $g(m) = g(n) + w(n, m)$, where w gets the edge weight between the two. This behavior is usually accomplished by a priority queue, with g as the priority. If we expand t, we are done, and we trace the path backwards from t back to s to reconstruct it. On the other hand, if we run out of nodes to expand, then there is no path.

7.2 Incorporating heuristics.

Dijkstra's can be quite slow in spaces with many possible paths, since it will tend to explore paths in all possible directions – even paths that seem to move farther away from the goal! For the sake of efficiency, we'd like to avoid exploring these paths, if possible. We will accomplish this by using a **heuristic**.

Recall that in Dijkstra's, we used $g(n)$, the known path cost to node n, as our priority. We will modify this slightly by defining a function h. $h(n)$ is a heuristic – an estimate of the remaining distance from n to the goal t. Then, rather than using $g(n)$ as the priority, we use $g(n) + h(n)$. Intuitively, we expand the node that we think will give the shortest path, based on the known path cost as well as our estimate.

h can be any function that is **admissible**: $h(n)$ must *never* exceed $c(n, t)$, the actual shortest path from n to the goal. In other words, h must always underestimate the true remaining distance. With an admissible heuristic, it can actually be proven that A* is both correct and optimal – it considers fewer nodes than any other search algorithm using the same heuristic.

Aside from the change in priority, the algorithm proceeds identically to Dijkstra's. In fact, Dijkstra's can be viewed as a special case of A*, with $h(n) = 0$.

7.3 Your task.

In the file `shared/a_star.ml`, we have supplied you with a functor `MakePQueue`, which can be applied to any `Set.OrderedType` to create a priority queue module. Complete the function `a_star : tile -> tile -> tile list -> (tile -> tile -> float) -> tile list` that, given a source, a destination, a heuristic function, and a set of tails (obstacles), computes the shortest path using A*. The path consists of the list of tiles, in the order they are traversed from the source to the destination. Use the values in `constants.ml` as the boundaries of the board.

The main things you will have to think about are how to keep track of each node's $g(n)$, and how to reconstruct the path once you've reached the destination. Once you've completed this part, you are free to use `a_star` anywhere else in your program. (Hint: use it.)

In particular, you should implement the functions `euclidean` and `manhattan` to test A* with. Euclidian distance is the straight line distance between two points, while Manhattan distance is the sum of the difference of the x and y coordinates of source and destination coordinates.

7.4 Karma

Come up with a heuristic that performs better than the above two heuristics. You may choose to concentrate on a certain type of graph structure if you wish. The internet is your friend. If you decide to attempt this part, come up with your heuristic and make a table or graph comparing its performance with the two given functions. Also, write a formal explanation about how you created the algorithm and how it is a better estimate.

8 Provided source code.

There are many provided code files for this assignment, most of which you will not need to edit at all. In fact, you will only need to modify files in the `game` and `team` directories, as well as the build scripts. Here is a list of all provided files, and a summary of their contents.

`build_*`	Scripts for building the game and teams.
`game/game.ml`	Stub module containing functions for handling actions, status requests, and time steps.
`game/game.mli`	Interface for `game.ml`.
`game/netgraphics.ml`	Module for sending updates to the GUI.
`game/netgraphics.mli`	Interface for `netgraphics.ml`.
`game/server.ml`	Module that starts the game and handles communication between the bots and game.
`shared/a_star.ml`	Module for A* search.
`shared/a_star.ml`	Interface for A* search.
`shared/connection.ml`	Connection helper module.
`shared/connection.mli`	Interface for connection helper module.
`shared/constants.ml`	Definitions of game constants.
`shared/definitions.ml`	Definitions of game datatypes.
`shared/thread_pool.ml`	Thread pool helper module.
`shared/thread_pool.mli`	Interface for thread pool helper module.
`shared/util.ml`	Helper functions for your general use. It may be a good idea to familiarize yourself with these functions, as they may simplify your tasks, or make debugging easier.
`team/team.ml`	Basic framework for interactions between AI clients and the game server.
`team/babybot.ml`	Stub for a basic AI.

In implementing the game, you should only need to make modifications to `game.ml`, as well as to any modules you may wish to add. You will not have to modify the server or GUI modules.

Likewise, when implementing your AI bot, you will not need to modify `team.ml`. You are free to base your bot on the provided `babybot.ml`.

8.1 Server module.

It may help to understand how the server works. At a high level, the server does the following things.

1. The server calls `Netgraphics.init`, which begins accepting connections from GUI clients.
2. The server waits for enough AI teams to connect to the game. Once they have, after a short countdown, the server begins running the game.
3. The server calls `Game.handleTime` at regular intervals, advancing the game state once every time step.
4. Whenever the server receives an action from an AI, it calls `Game.handleAction` with that command as the argument. Similarly, when it receives a status request, it calls `Game.handleStatus` with the request as the argument.
5. The server also calls `Netgraphics.send_updates` at regular intervals to update the GUI.

8.2 Game state.

Keep in mind that you are free to add whatever modules you wish. In fact, for keeping track of the game state, we strongly suggest creating a separate `State` module, rather than putting all your code in `game.ml`. In fact, it may be a good idea to create multiple modules, one for each aspect of the game state – riders, team inventories, and so on.

9 Running the game.

You will need to launch four command prompts to run the game.

9.1 Game server.

In the folder `ps6/game`, run the script `build_game.bat` to compile the game. Once this succeeds, run `game.exe`.

9.2 GUI.

In the folder `ps6/gui`, run `java -jar gui_client.jar`. At the bottom of the interface, enter the server's connection information (localhost, post 10501 by default), and press Connect.

9.3 Team.

In the folder `ps6/team`, run `build_team.bat botname` or `build_team.sh botname`, where the bot you wish to build is in the file `botname.ml`. After that, run `botname.exe localhost 10500` to start the bot. Run this in two separate command prompts, and *watch the magic happen!*

10 Your tasks.

This project consists of many parts. Make sure you spend time thinking about and designing each part before writing code. This is a large project, so start *early*. Remember, **no extensions!**

10.1 Design meeting.

The first thing you will have to do is come up with a design for your implementation of *Tron*, and meet with a course staff member to discuss it. Design review slots will be posted on CMS. If you cannot sign up for any of those time slots, contact the course staff and we will try to accomodate you.

At the meeting, you will be expected to explain the design of your system, including the data structures you will use. You will need to outline the interfaces of the modules you plan to create. In designing these interfaces, think about what functionality each module needs to have, what information needs to be stored by each one, and how the interface can be made as narrow as possible.

You will also have to discuss how you plan to manage your time to complete the project, and how work will be divided. Everyone in the group should be prepared to discuss the plan. At the end, we will give you feedback on your design.

10.2 Implementing the game.

You will then have to implement *Tron*, completing the functions in `game.ml`, and adding any additional modules that you may require. You should only add files to the `game` and `team` directories. Your implementation should conform to the specifications described in previous sections. You may use our sample bot to test your program, but you will most likely want to write some extra tests of your own.

10.3 Designing a bot.

Your next task is to implement a bot that can play *Tron*. As discussed, you may use `babybot.ml` as a model for your bot. We will be running your bot against test bots of our own, and your score will depend on your performance.

Additionally, there will be a tournament where you will be able to see your bot compete against other students' bots. There are many possible approaches – this is your chance to be creative, and to have fun coming up with a strategy.

10.4 Documentation.

Your final task is to submit a design overview document for this project. Since this project is quite large and open-ended, documentation becomes even more important.

The design overview document should cover *both* your implementation of the game, and the bot you created. For the game, you should discuss the structure and purpose of each of your modules, and your reason for creating them. For the bot, you should discuss what strategies you experimented with, and what your final approach was.

10.5 Guidelines.

Here are some things that you should keep in mind while working on this project.

- **Your design is important.** This project is quite large and complex. If you do not have a solid and complete design, you will quickly get bogged down in details when you begin to implement it. Before writing any code, you should have a very clear idea of *all* of the following.
  - Concurrency issues, and how you will address them.
  - Information that needs to be stored to fully represent the game state.
  - How to store and access that information efficiently.
  - What the interfaces of your modules will be.
  - Invariants that your modules will preserve.
  - What modules will enforce those invariants.
- **Think about how to organize your program into loosely coupled modules.** It will be difficult to debug your project unless you can develop modules that encapsulate important aspects of the game. Design your modules carefully so that you can work effectively with your partner and do unit testing of each module.
- **Preserve the relationship between game state and graphics.** Recall that updating one doesn't automatically update the other. If you are watching the game and something goes wrong, the problem could either be with the game itself, or with the GUI updates. Further, just because the GUI looks correct doesn't mean the underlying game state is necessarily correct. It would behoove you to maintain an invariant between the two.
- **Problems in the game may actually be problems with the bots.** If you are using your own bots to test, make sure their behavior is correct.
- **Implement and test the actions one at at time.** Start with the easier ones, and once you are sure those are correct, move on to the more complex ones.
- **Leave yourself enough time to write the bot.** The bot is worth almost as much as the game, so don't put it off until the end.
- **Lead TA's.** If you have specific questions that need to be resolved, direct them to Ranjay Krishna(in charge of the game and bot.), Pablo Sarmiento (in charge of the GUI.), Chris Yu (in charge of A-star.) and Ben Carriel (in charge of the Written Problem.).

11 Written Problem: Reasoning about Persistence

11.1 Functional Data Structures

In this exercise, we are going to pin down some of the differences between functional data structures and their imperative counterparts. As you have probably seen throughout the semester, there are two major differences between these two styles of data structure design:

1. Functional data structures lack destructive updates (i.e. assignments)
2. Functional data structures are automatically *persistent*, meaning they support multiple versions, whereas imperative data structures are *ephemeral*, meaning that they only support one version at a time.

With the inclusion of the `ref` cell, OCaml forfeited the first property above. We are going to explore the consequences of the second.

The way to think of persistence is that it gives you the ability to ask questions about past states of the data structure, which is very useful in practice. The persistence model we will start with is known as *partial persistence*. In this model the user can query any previous version of the data structure, but only the most recent version ever gets updated. Typical operations would be

```
val query : 'a structure -> version -> unit
val write : 'a structure -> version -> 'a -> unit
```

11.2 Preliminary Model

Many of the data structures we have encountered in the course could be thought of as bundles of nodes of some bounded size, with entries for some data. By data we mean either actual data, or a pointer to another node. You also have these operations on the structure:

```
val field : 'a structure -> 'a
val update :  'a -> 'a structure -> unit
val destroy : 'a structure -> unit
```

If we were to try to generalize this into an OCaml data type we would get something like the following:

```
module Node : Set.OrderedType = struct
  type 'a node = Node of 'a data ref
  and 'a data = Data of 'a | Pointer of 'a*'a node ref

  type 'a t = 'a node

  (* equality tested on the representation. *)
  let compare Node(d) Node(d') =
    match d,d' with
    | Data (x), Data(y) -> if x=y then 0 else -1
    | Pointer (p), Pointer(p') -> if p==p' then 0 else -1
```

```
    | _ -> -1
end
```

Note that the above would require a polymorphic `Set.OrderedType`, which isn't in the OCaml standard. We would have to implement it ourselves! Now we just need to wrap a bunch of `'a Node.t`'s into a set. The right tool for the job is a *higher-order functor*.

```
module type DATASTRUCTURE = sig
  type t (* internal data structure type *)
  type v (* internal version representation *)
  module M (* the underlying set of nodes *)
  exception Invalid_representation (* if a node gets too big *)
  val cUPPER_BOUND : int (* bound on size *)

  (* Structure level operations *)
  val query : t -> v -> unit
  val write : t -> v -> M.t -> unit

  (* Node level operations *)
  val field : 'a M.t -> 'a
  val update :  'a M.t -> t -> unit
  val destroy : 'a M.t  -> unit
end

(* functor to create the data structure *)
module DSMake = functor (M : Set.S) -> DATASTRUCTURE

(* Compose the functors to get a data structure! *)
module DataStructure = DSMake (Set.Make (Node))
```

Awesome! We have built an abstraction in OCaml that captures our intuitive notion of what a data structure is. In the following exercises, we will consider a few possible implementations and prove some facts about their properties.

11.3 Implementing Partial Persistence

Now that we have a basic framework, we can start addressing the main questions: Is it possible to implement (functional) partial persistence efficiently? CS Lords of the past answered this question affirmatively in the 80's with the restriction that the in-degree of each node is $O(1)$. In our model, we would have to include a `rep_ok` function in the `DataStructure struct`, and also come up with a good number for `cUPPER_BOUND`. Your task is to reconstruct the proof.[1]

Exercise 1:

The main idea of the proof is to add more data to the `'a node` type so that we have information when implementing the functors that build a data structure (this will complicate the code a lot). Let's go over the changes:

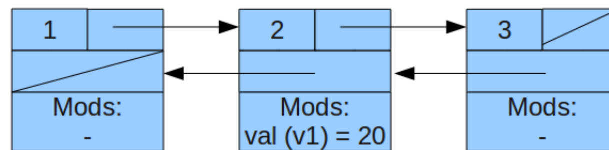
[1]Don't worry, we'll walk you through it

1. Add a "read-only" area for all of the data and pointers.
2. Add a writeable area for "back-pointers". A node n will have a back-pointer to another node m if m has a pointer to n.
3. A "log" section that has entries of the form `('a, version)`. This will store the modifications that the user will make to the data structure in the future.

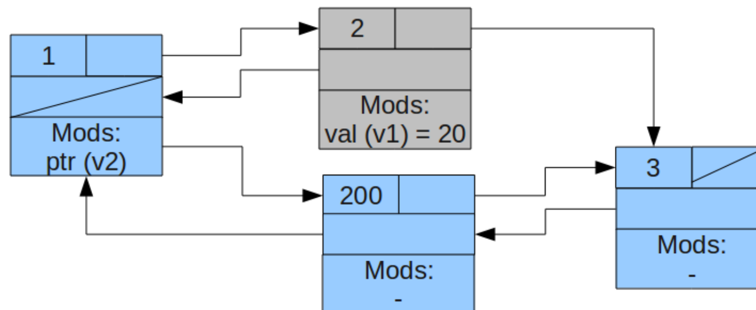
Your first task is to write a high-level English description of how one might implement these changes. Submit the following:

1. Show how you would accommodate the proposed changes by modifying the type signatures in the provided `Node` module.
2. Remember: we are shooting for an efficient implementation. Describe how you will make sure that the size of the back-pointer and log sections of each node don't get excessively large. (Hint: how does `cUPPER_BOUND` come into play?)

Below is a visual representation of how this abstraction would work for a linked-list style data structure:



The above is a partially persistent linked-list after a single update: writing the value `20` to the leftmost node.



Here is the structure after a second update: updating the pointer contained in the left-most node, with a new pointer containing the value `200`.

Exercise 2:

The above shows the intuition for how the data structure queries work. Your next task is to come up with reasonable implementations of the following operations:

1. Describe how to implement the `read` operation in the `DataStructure` module. A short English description or pseudocode will suffice.

2. Give a brief description of how to implement the `write` operation in the `DataStructure` module. (Hint: This should be clear if the mod log isn't full. If the the log is full, add a level of indirection.)

11.4 Performance Analysis

All right, now we have to actually justify to ourselves that our implementation is actually efficient and useable in the "real-world". Your `write` operation probably had some sort of recursive call in it (if not, then back to the drawing board!). We need to put a bound C on the number of recursive calls that we make, or we risk the runtime blowing up! For certain data structures like lists or trees, we know C at the onset (why? justify this to yourself).

11.4.1 Idea of the Proof

So how did we know that this framework would be effective ahead of time? When you are out in the wild designing data structures for your applications, it is important to think about why your implementation will perform well. In our case, the intuition is this:

> For every node in the old data structure, the new data structure is going to have a bunch of nodes: the current version, along with all of the old ones. This means that we will have cheap reads. Writes will also be cheap, as long as we're not filling up the log. If we do fill up the log, we know that the log will get sparser as we go deeper in the recursion.

(If your operations don't have the above properties, you should rethink your implementation!)

When you encounter any data structure that performs both frequent, cheap operations and occasional slow operations and want to analyze its performance, your first instinct should be to conduct an amortized analysis. That is the strategy we will employ.

Exercise 3:

Ok, judgment time. Discuss the performance of the partial persistence implementation using the physicists' method. In particular, provide the following two analyses:

1. **Space Complexity:** By making the proposed changes to our nodes, we have introduced a lot of overhead in our implementations. Give an asymptotic bound (i.e. Big-O notation) on the space overhead of the implementation. (Hint: Use `cUPPER_BOUND` and C.)

2. **Time Complexity:** Determine the amortized runtime of each of the structure level operations in the `DataStructure` module. Here are some things to think about to point you in the right direction:

    (a) What is the cost of writing to a node? Write out a good function for this. Then phrase your potential function in terms of the cost function. (Hint: Use your cost function to "deal with the past", and then write your potential function in terms of the cost function and "dealing with the present". What are the factors that blow up the runtime in both cases? How can you fiddle with the constants to save enough potential for the slow operations?)

(b) Termination: Does every operation terminate? Remember, the back pointers might form cycles (think about implementing a tree), how do you make sure we won't get stuck? (Hint: what happens to the potential function in the recursive step)

(c) Don't write too much! All of these questions have short (a few sentences) answers. That, plus a little algebra is all you need. If you find yourself making things complicated, go back and rethink your cost and potential functions.

Good Luck!

TO SUBMIT: The solutions to Exercises 1,2 and 3 in `.pdf` format.

12 Final submission.

You will submit:

- A `.zip` file of all files in your `ps6` directory, including those that you did not edit. Please preserve the directory structure of the original release. We should be able to unzip your submission, use the `build_game.bat` script to build the game, and use `build_game.bat` to build your bot; in other words, you should modify the build scripts to include all necessary files.
- Your documentation file, in `.pdf` format.
- Your written problem, in `.pdf` format.

Again, we should be able to unzip your submission and use the scripts to compile both your game and your bots, without errors or warnings. **Submissions that do not meet this criterion will lose points.**