YEAH A8

The Adventures of Links

Assignment overview

This is the **penultimate** 106B assignment!

This assignment is due **Friday 3/12**, and the grace period expires the following **Sunday.**

You're welcome to work in pairs on this assignment.

Assignment overview

This assignment consists of two parts:

1. Labyrinth - Using your debugger skills, can you escape a linked list labyrinth, made specifically for you?
2. DNA - Now that you're comfortable inspecting linked lists, you'll need to implement a series of functions that operate on **nucleotides** represented by linked lists!

Part 1: Labyrinth

In this first part, you'll be attempting to escape from a maze!

Part 1: Labyrinth

In this first part, you'll be attempting to escape from a maze!

To escape, you'll need to collect **three magic items:** a wand, a spellbook, and a potion.

This maze exists as a linked structure consisting of these elements:

```
struct MazeCell {
   Item whatsHere;  // Item present, if any.
   MazeCell* north; // The cell north of us, or nullptr if we can't go north.
   MazeCell* south;
   MazeCell* east;
   MazeCell* west;
};
```

Part 1: Labyrinth

In this first part, you'll be attempting to escape from a maze!

To escape, you'll need to collect **three magic items:** a wand, a spellbook, and a potion.

This maze exists as a linked structure consisting of these elements:

```
struct MazeCell {
   Item whatsHere;  // Item present, if any.
   MazeCell* north; // The cell north of us, or nullptr if we can't go north.
   MazeCell* south;
   MazeCell* east;
   MazeCell* west;
};
```

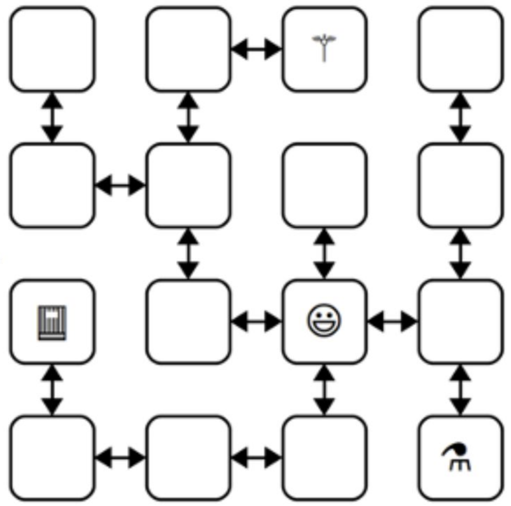
```
enum class Item {
    NOTHING, SPELLBOOK, POTION, WAND
};
```

Remember enums? Wish you could forget them? :D

Part 1: Labyrinth

Here's what an example maze might look like!

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```

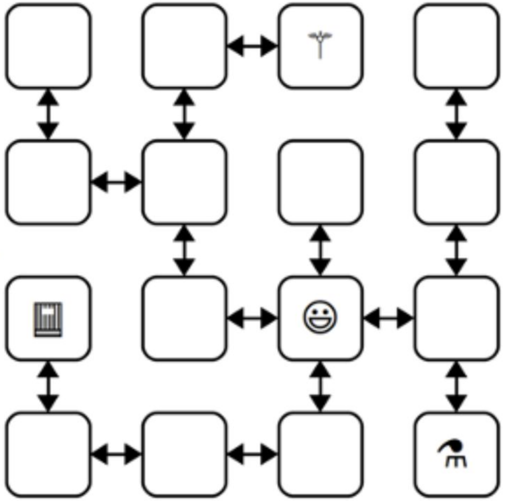


Imagine each of the **boxes** to be a **`MazeCell`** struct. Notice that not all have 4 valid arrow directions. Directions that don't lead to other cells are **`nullptr`**.

Part 1: Labyrinth

Here's what an example maze might look like!

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```



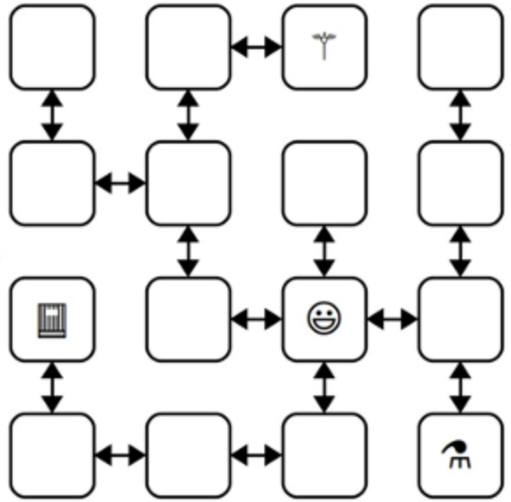
Imagine each of the **boxes** to be a **`MazeCell`** struct. Notice that not all have 4 valid arrow directions. Directions that don't lead to other cells are **`nullptr`**.

The smiley face indicates that you'll begin at a random location in the maze!

Part 1: Labyrinth

Here’s what an example maze might look like!

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```



Imagine each of the **boxes** to be a **`MazeCell`** struct. Notice that not all have 4 valid arrow directions. Directions that don’t lead to other cells are **`nullptr`**.

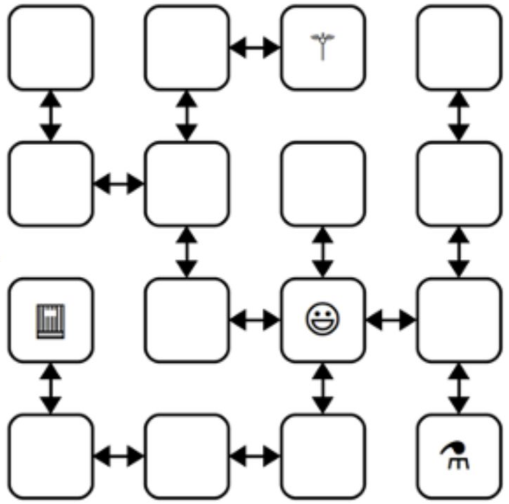
The smiley face indicates that you’ll begin at a random location in the maze!

As you can see, most **`MazeCells`** are empty, but some have the magical items in them!

Part 1: Labyrinth

Here's an example path through the labyrinth:

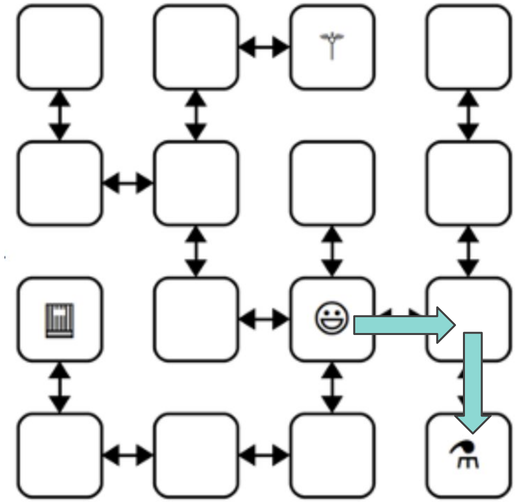
```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```



Part 1: Labyrinth

Here's an example path through the labyrinth:

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```

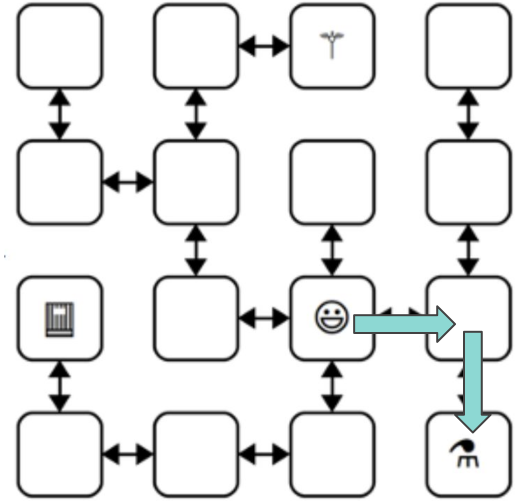


In this path, we effectively went from cell **`start`** to **`start->east->south`**.

Part 1: Labyrinth

Here's an example path through the labyrinth:

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```



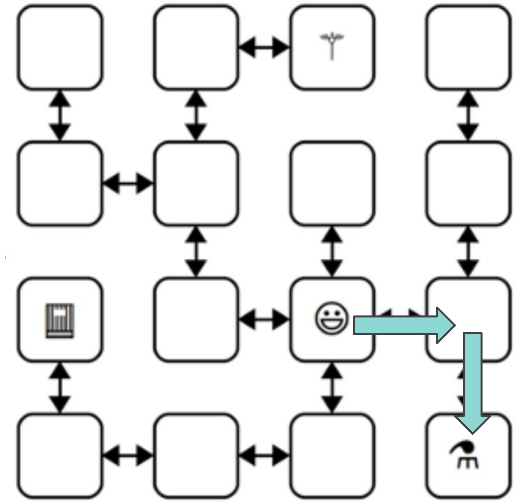
In this path, we effectively went from cell `start` to `start->east->south`.

This path was valid because we didn't go through any "walls" (i.e. cells that aren't connected), and we even found a **magic item!**

Part 1: Labyrinth

Here's an example path through the labyrinth:

```
struct MazeCell {
    Item whatsHere;
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```



In this path, we effectively went from cell **`start`** to **`start->east->south`**.

This path was valid because we didn't go through any "walls" (i.e. cells that aren't connected), and we even found a **magic item!**

You could represent this subpath as the string "ES", where each character represents a direction you went from the starting location.

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Your first task is to write the above function ^^

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Your first task is to write the above function ^^

Given a starting **`MazeCell*`** and a string that represents a path you can take through the maze (i.e. NNNSEWWSSENSWENSEEWS), return whether this path was a valid path leading to freedom!

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Your first task is to write the above function ^^

Given a starting **`MazeCell*`** and a string that represents a path you can take through the maze (i.e. NNNSEWWSSENSWENSEEWS), return whether this path was a valid path leading to freedom!

A path leads to **freedom** if it: 1- never attempts to move into a direction that's **`nullptr`** (i.e. never goes through a "wall"), and 2 - picks up all 3 magical items.

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Your first task is to write the above function ^^

Given a starting `MazeCell*` and a string that represents a path you can take through the maze (i.e. NNNSEWWSSENSWENSEEWS), return whether this path was a valid path leading to freedom!

A path leads to **freedom** if it: 1- never attempts to move into a direction that's `nullptr` (i.e. never goes through a "wall"), and 2 - picks up all 3 magical items.

You should expect to iterate through the **entire** string, not stopping early unless you're asked to go through a wall (return false).

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)
- You can assume that the **`startLocation`** is not **`nullptr`**, and that **`path`** always contains NSEW characters.

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)
- You can assume that the **`startLocation`** is not **`nullptr`**, and that **`path`** always contains NSEW characters.
- Don't assume for this part that paths are bi-directional, but we don't anticipate this being a problem, because all you need to do is follow the characters in the **`path`**.

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)
- You can assume that the **`startLocation`** is not **`nullptr`**, and that **`path`** always contains NSEW characters.
- Don't assume for this part that paths are bi-directional, but we don't anticipate this being a problem, because all you need to do is follow the characters in the **`path`**.
- You shouldn't be allocating any new cells here. Making temporary pointers is fine, but you shouldn't be using the **`new`** keyword.

Part 1: Labyrinth

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)
- You can assume that the **`startLocation`** is not **`nullptr`**, and that **`path`** always contains NSEW characters.
- Don’t assume for this part that paths are bi-directional, but we don’t anticipate this being a problem, because all you need to do is follow the characters in the **`path`**.
- You shouldn't be allocating any new cells here. Making temporary pointers is fine, but you shouldn’t be using the **`new`** keyword.
- One tricky edge case - if you find all three items, **keep looping through the string.** If you happen to **also** move through a wall, you should **return false** anway.

Part 1: Labyrinth

Any Questions?

```
bool isPathToFreedom(MazeCell* startLocation, const string& path);
```

Some notes about the problem:

- You can implement this iteratively or recursively, whichever you choose :)
- You can assume that the **`startLocation`** is not **`nullptr`**, and that **`path`** always contains NSEW characters.
- Don't assume for this part that paths are bi-directional, but we don't anticipate this being a problem, because all you need to do is follow the characters in the **`path`**.
- You shouldn't be allocating any new cells here. Making temporary pointers is fine, but you shouldn't be using the **`new`** keyword.
- One tricky edge case - if you find all three items, **keep looping through the string.** If you happen to **also** move through a wall, you should **return false** anway.

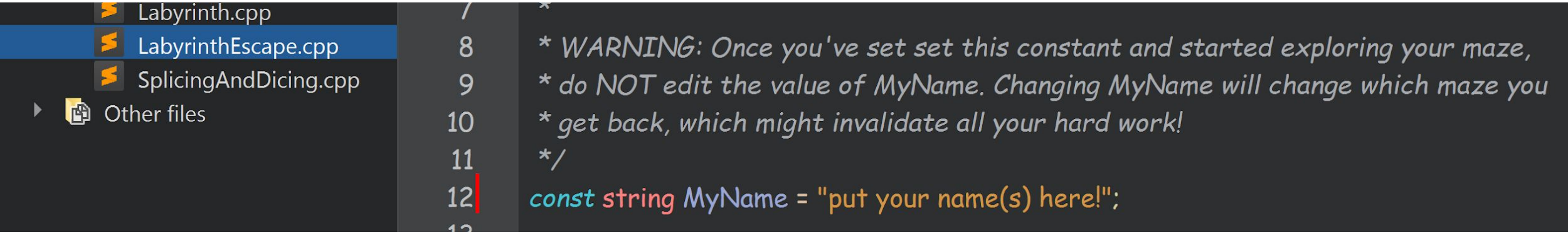
Part 1.5: Escape from the Labyrinth

Now that you’ve written a function that can determine whether a path gets you out of a labyrinth, it’s your turn to find your way out of a **personalized** one!

Part 1.5: Escape from the Labyrinth

Now that you've written a function that can determine whether a path gets you out of a labyrinth, it's your turn to find your way out of a **personalized** one!

In the file LabyrinthEscape.cpp, you'll need to put your name in the string `myName`:



```
 *
 * WARNING: Once you've set set this constant and started exploring your maze,
 * do NOT edit the value of MyName. Changing MyName will change which maze you
 * get back, which might invalidate all your hard work!
 */
const string MyName = "put your name(s) here!";
```

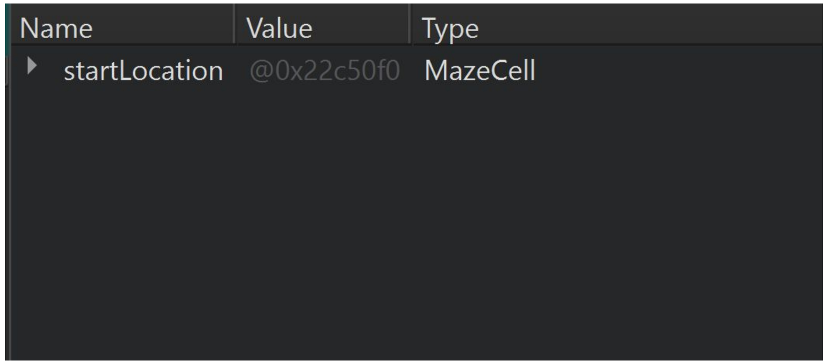
Part 1.5: Escape from the Labyrinth

Once you’ve put your name into the file, set a breakpoint on the first test in the file (for me it’s line 33). Run your code in the debugger, and you’ll be placed in a maze personally generated for your name!

Part 1.5: Escape from the Labyrinth

Once you’ve put your name into the file, set a breakpoint on the first test in the file (for me it’s line 33). Run your code in the debugger, and you’ll be placed in a maze personally generated for your name!

This is what your debugger pane should look like when you hit the breakpoint:

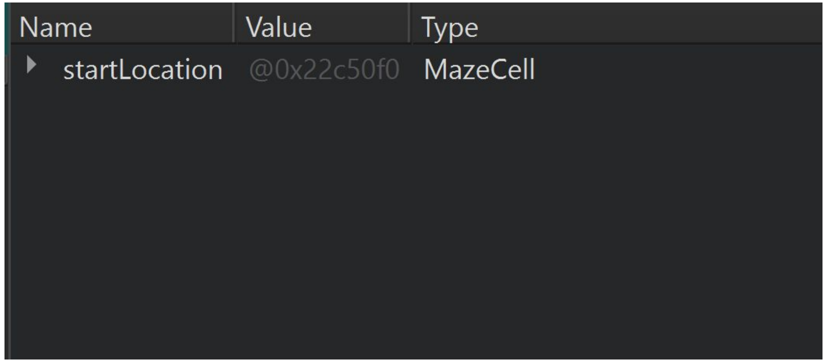


Name	Value	Type
startLocation	@0x22c50f0	MazeCell

Part 1.5: Escape from the Labyrinth

Once you've put your name into the file, set a breakpoint on the first test in the file (for me it's line 33). Run your code in the debugger, and you'll be placed in a maze personally generated for your name!

This is what your debugger pane should look like when you hit the breakpoint:

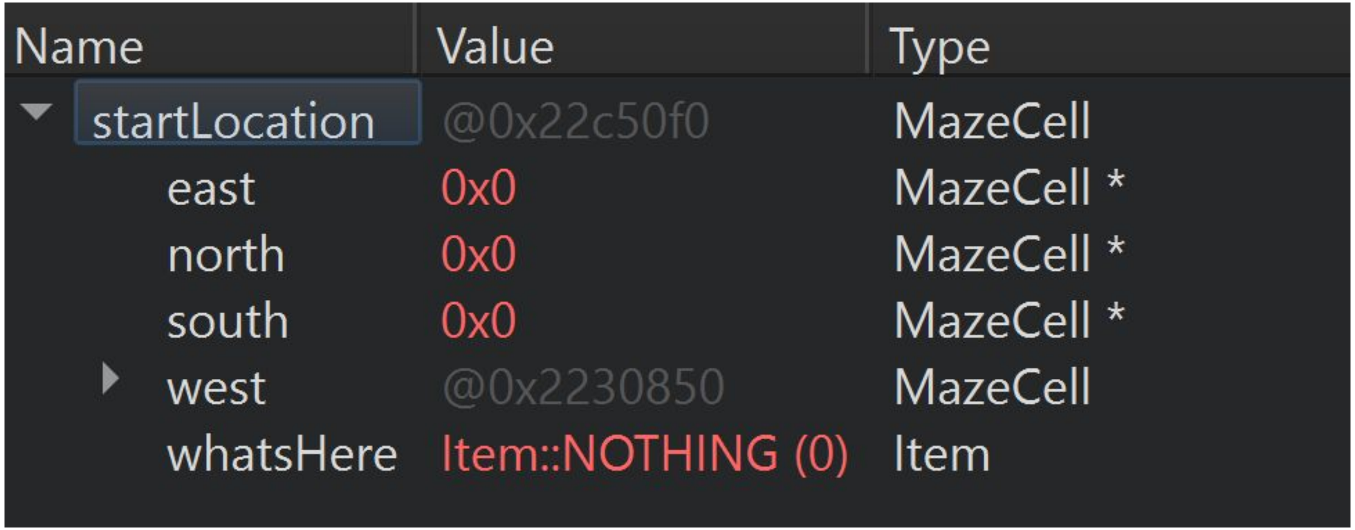


Name	Value	Type
startLocation	@0x22c50f0	MazeCell

You've been given a **`startLocation`**! Open it up, and it should behave like the **`MazeCell`**'s that you've seen before.

Part 1.5: Escape from the Labyrinth

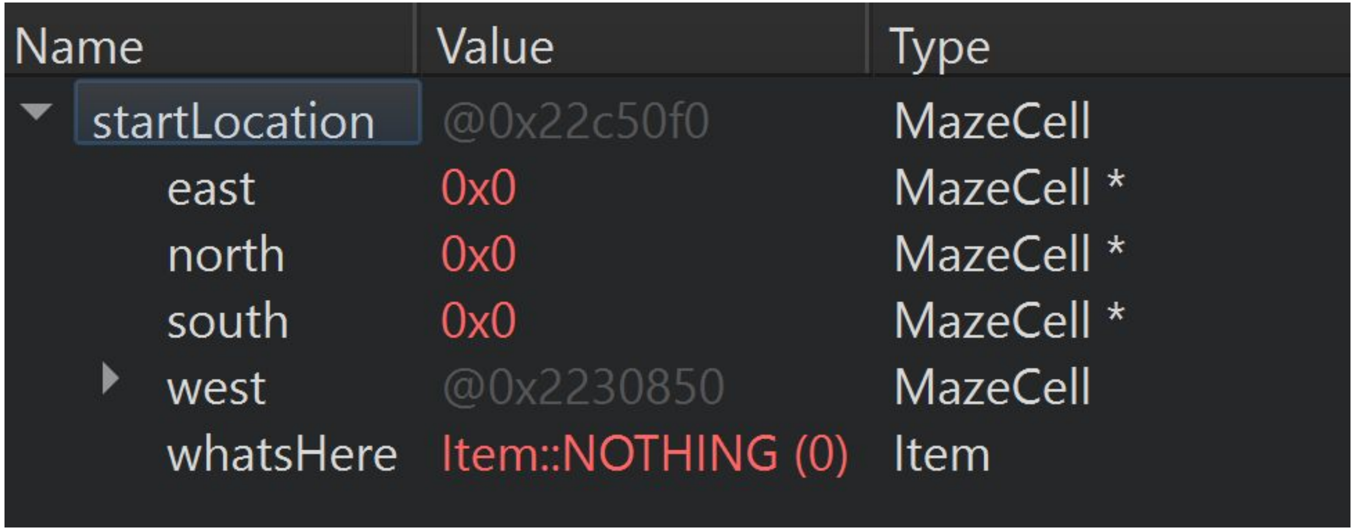
Here's what opening the startLocation yields me:



Name	Value	Type
startLocation	@0x22c50f0	MazeCell
east	0x0	MazeCell *
north	0x0	MazeCell *
south	0x0	MazeCell *
west	@0x2230850	MazeCell
whatsHere	Item::NOTHING (0)	Item

Part 1.5: Escape from the Labyrinth

Here's what opening the startLocation yields me:

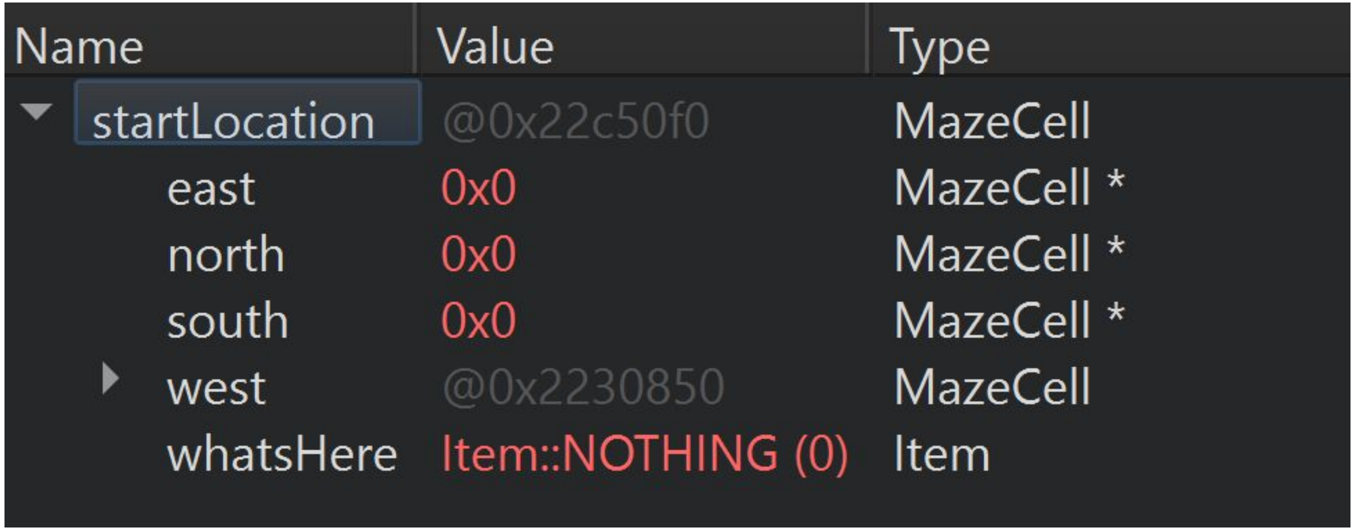


Name	Value	Type
startLocation	@0x22c50f0	MazeCell
east	0x0	MazeCell *
north	0x0	MazeCell *
south	0x0	MazeCell *
west	@0x2230850	MazeCell
whatsHere	Item::NOTHING (0)	Item

Notice that there's nothing in the **`whatsHere`** field!

Part 1.5: Escape from the Labyrinth

Here's what opening the startLocation yields me:



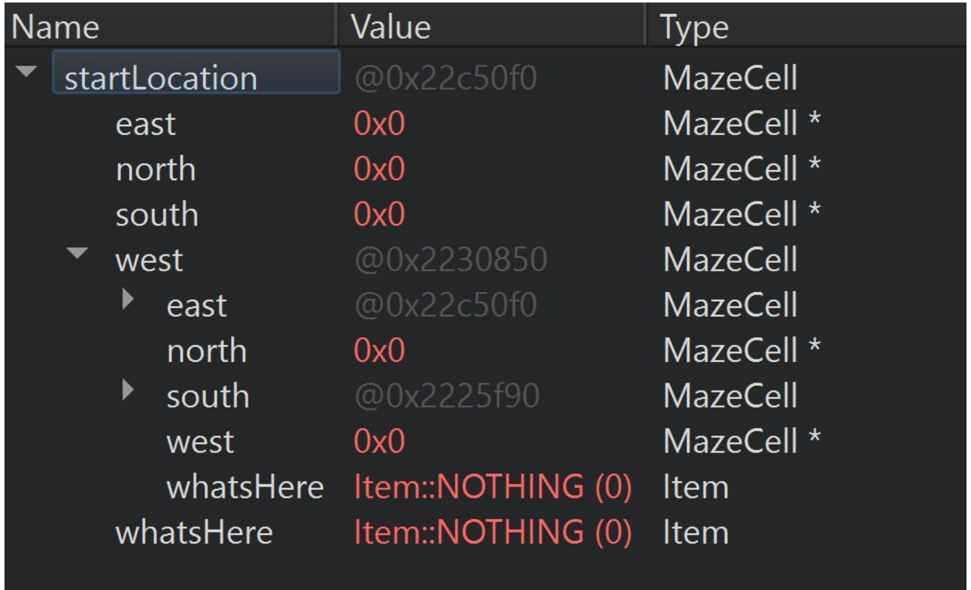
Name	Value	Type
startLocation	@0x22c50f0	MazeCell
east	0x0	MazeCell *
north	0x0	MazeCell *
south	0x0	MazeCell *
west	@0x2230850	MazeCell
whatsHere	Item::NOTHING (0)	Item

Notice that there's nothing in the **`whatsHere`** field!

Also, the only non-wall place we can go is in the west direction! Let's go there :)

Part 1.5: Escape from the Labyrinth

Here's what opening the west location yields me:

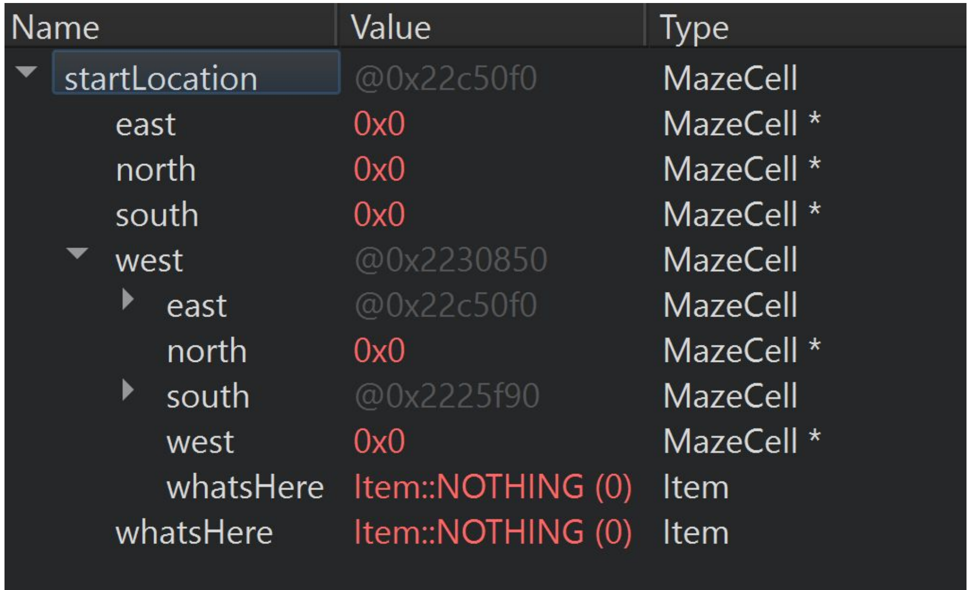


```
Name                Value                 Type
▾ startLocation     @0x22c50f0            MazeCell
    east            0x0                   MazeCell *
    north           0x0                   MazeCell *
    south           0x0                   MazeCell *
  ▾ west            @0x2230850            MazeCell
    ▸ east          @0x22c50f0            MazeCell
      north         0x0                   MazeCell *
    ▸ south         @0x2225f90            MazeCell
      west          0x0                   MazeCell *
      whatsHere     Item::NOTHING (0)     Item
    whatsHere       Item::NOTHING (0)     Item
```

Rats! Nothing here too!

Part 1.5: Escape from the Labyrinth

Here's what opening the west location yields me:

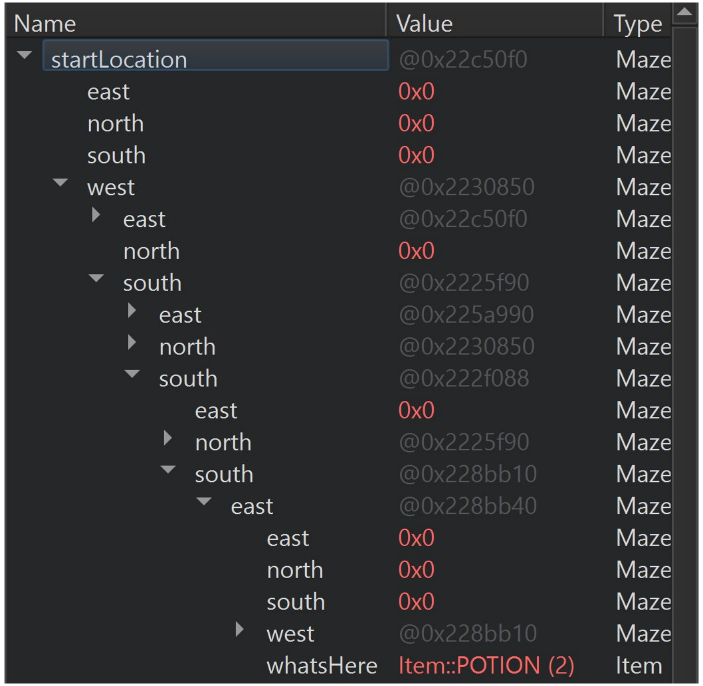


```
Name                Value                  Type
▼ startLocation     @0x22c50f0             MazeCell
    east            0x0                    MazeCell *
    north           0x0                    MazeCell *
    south           0x0                    MazeCell *
  ▼ west            @0x2230850             MazeCell
    ▶ east          @0x22c50f0             MazeCell
      north         0x0                    MazeCell *
    ▶ south         @0x2225f90             MazeCell
      west          0x0                    MazeCell *
      whatsHere     Item::NOTHING (0)      Item
    whatsHere       Item::NOTHING (0)      Item
```

Rats! Nothing here too!

Look inside the indented (west) struct. Notice that the `west->east` memory address is **identical** to the `startLocation` address. In this problem, all paths are bi-directional, so take a second to register why this must always be true.

Part 1.5: Escape from the Labyrinth

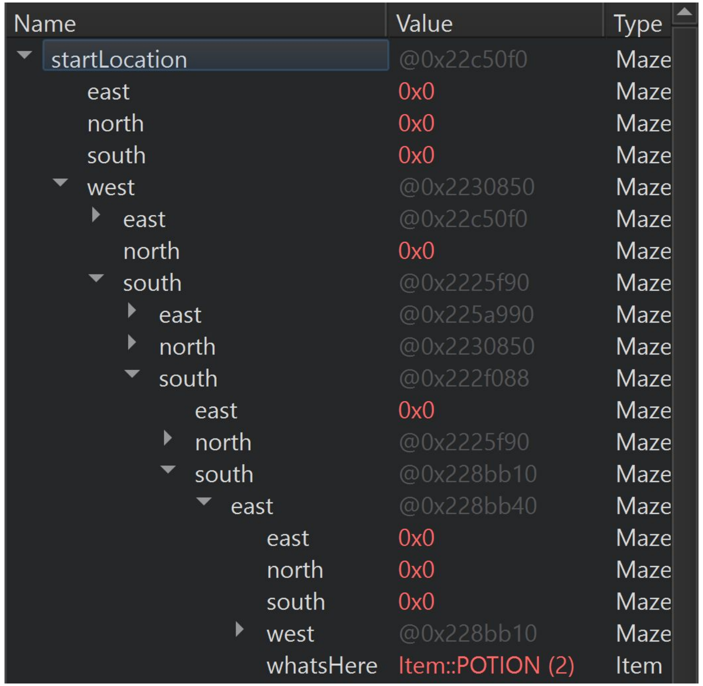


```
Name                        Value             Type
▾ startLocation             @0x22c50f0        Maze
    east                    0x0               Maze
    north                   0x0               Maze
    south                   0x0               Maze
  ▾ west                    @0x2230850        Maze
    ▸ east                  @0x22c50f0        Maze
      north                 0x0               Maze
    ▾ south                 @0x2225f90        Maze
      ▸ east                @0x225a990        Maze
      ▸ north               @0x2230850        Maze
      ▾ south               @0x222f088        Maze
          east              0x0               Maze
        ▸ north             @0x2225f90        Maze
        ▾ south             @0x228bb10        Maze
          ▾ east            @0x228bb40        Maze
              east          0x0               Maze
              north         0x0               Maze
              south         0x0               Maze
            ▸ west          @0x228bb10        Maze
              whatsHere     Item::POTION (2)  Item
```

Eventually, you'll find an item!

Quiz question: What is the string path starting at `startLocation` that find the POTION?

Part 1.5: Escape from the Labyrinth



```
Name                      Value              Type
▼ startLocation           @0x22c50f0         Maze
    east                  0x0                Maze
    north                 0x0                Maze
    south                 0x0                Maze
  ▼ west                  @0x2230850         Maze
    ▶ east                @0x22c50f0         Maze
      north               0x0                Maze
    ▼ south               @0x2225f90         Maze
      ▶ east              @0x225a990         Maze
      ▶ north             @0x2230850         Maze
      ▼ south             @0x222f088         Maze
          east            0x0                Maze
        ▶ north           @0x2225f90         Maze
        ▼ south           @0x228bb10         Maze
          ▼ east          @0x228bb40         Maze
              east        0x0                Maze
              north       0x0                Maze
              south       0x0                Maze
            ▶ west        @0x228bb10         Maze
              whatsHere   Item::POTION (2)   Item
```

Eventually, you'll find an item!

Quiz question: What is the string path starting at **`startLocation`** that find the POTION?

Answer: "WSSSE"

Part 1.5: Escape from the Labyrinth

Your job is to keep poking around this maze, recording where you’ve been / where you’re going, until you can find all 3 magic items!

Part 1.5: Escape from the Labyrinth

Your job is to keep poking around this maze, recording where you’ve been / where you’re going, until you can find all 3 magic items!

We **highly** recommend drawing lots of pictures so that you can construct an image of your maze!

Part 1.5: Escape from the Labyrinth

Your job is to keep poking around this maze, recording where you've been / where you're going, until you can find all 3 magic items!

We **highly** recommend drawing lots of pictures so that you can construct an image of your maze!

Once you have a good idea of where your items are, write out the string path that hits all 3 of your items (i.e. NNSSEWEWWWESNEENESNWS). Put that path in the variable **`ThePathOutOfMyMaze`** variable and run the tests -- if the test passes, then you've successfully escaped!

Part 1.5: Escape from the Labyrinth

Your job is to keep poking around this maze, recording where you've been / where you're going, until you can find all 3 magic items!

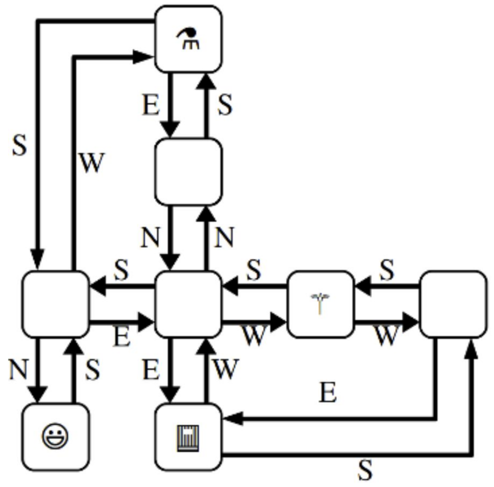
We **highly** recommend drawing lots of pictures so that you can construct an image of your maze!

Once you have a good idea of where your items are, write out the string path that hits all 3 of your items (i.e. NNSSEWEWWWESNEENESNWS). Put that path in the variable **`ThePathOutOfMyMaze`** variable and run the tests -- if the test passes, then you've successfully escaped!

Any questions?

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.

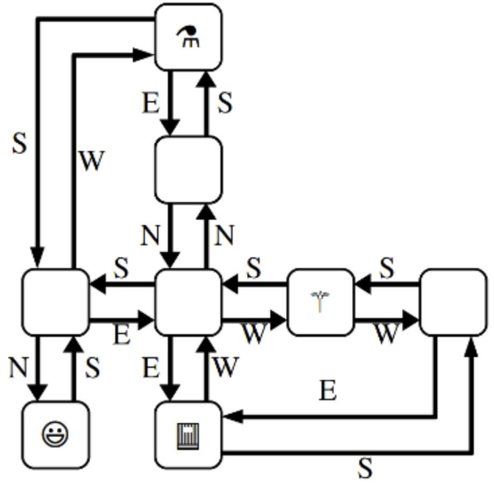


This time, the NSEW pointers may point in any direction! (for example, in one `MazeCell`, east may actually point south!)

More simply, put, **the names of the 4 pointers in the `MazeCell` have no directional meaning.**

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.



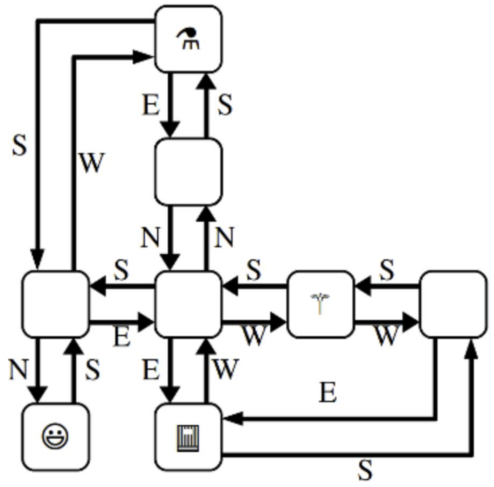
This time, the NSEW pointers may point in any direction! (for example, in one **`MazeCell`**, east may actually point south!)

More simply, put, **the names of the 4 pointers in the `MazeCell` have no directional meaning.**

Luckily, you can still find a way out of the maze! For example, the path SW finds one of the magical items!

Part 1.6: Escape from a *twisty* labyrinth

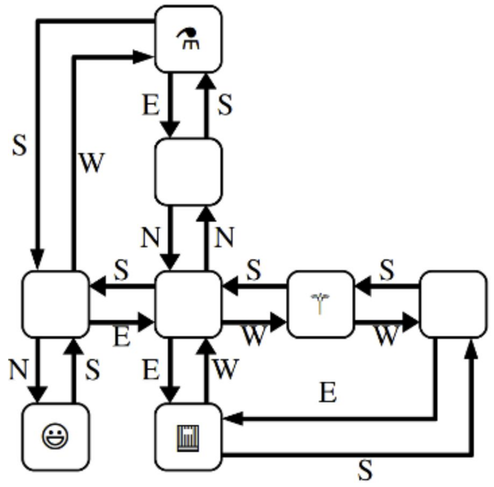
In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.



When solving this maze, we've guaranteed that the directions wont change on you (i.e. the path SW will always find the top item in this maze), but you're going to have to spend more time mapping this maze out.

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.

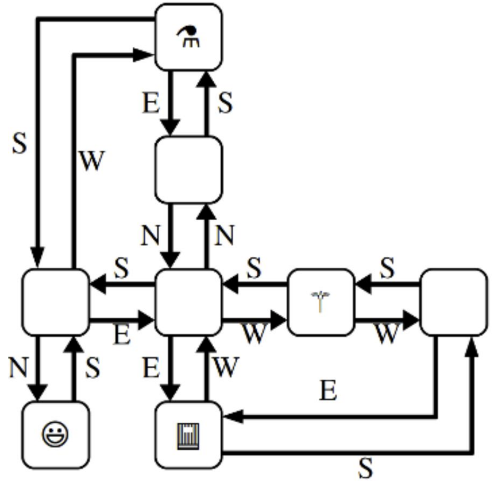


When solving this maze, we've guaranteed that the directions wont change on you (i.e. the path SW will always find the top item in this maze), but you're going to have to spend more time mapping this maze out.

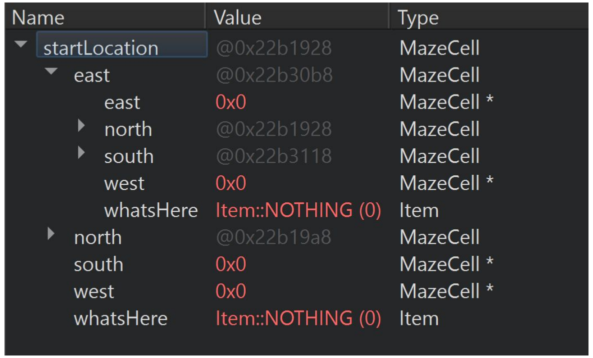
We've also guaranteed again that all paths are bi-directional, but as you can see, you're going to need to be careful to figure out which path is the way back.

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.



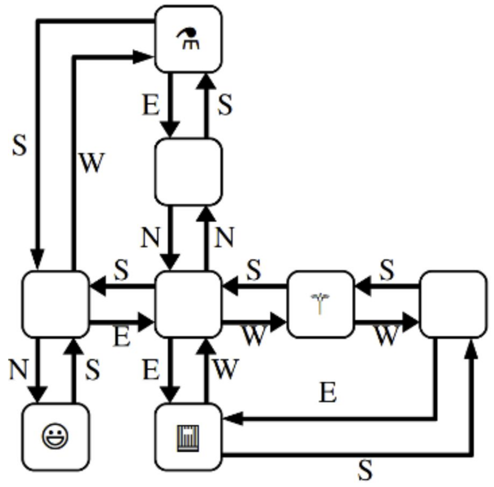
To determine whether a neighboring pointer moves you to a new location, you can look at the **memory address**



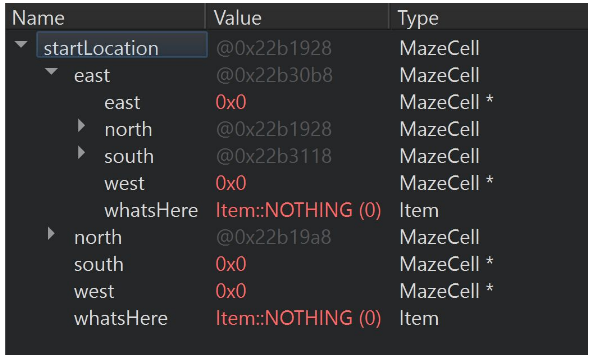
```
Name              Value                  Type
▾ startLocation   @0x22b1928             MazeCell
  ▾ east          @0x22b30b8             MazeCell
      east        0x0                    MazeCell *
    ▸ north       @0x22b1928             MazeCell
    ▸ south       @0x22b3118             MazeCell
      west        0x0                    MazeCell *
      whatsHere   Item::NOTHING (0)      Item
  ▸ north         @0x22b19a8             MazeCell
    south         0x0                    MazeCell *
    west          0x0                    MazeCell *
    whatsHere     Item::NOTHING (0)      Item
```

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.



To determine whether a neighboring pointer moves you to a new location, you can look at the **memory address**

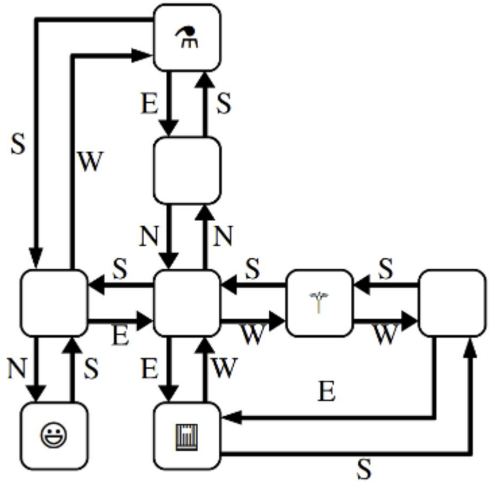


Name	Value	Type
startLocation	@0x22b1928	MazeCell
east	@0x22b30b8	MazeCell
east	0x0	MazeCell *
north	@0x22b1928	MazeCell
south	@0x22b3118	MazeCell
west	0x0	MazeCell *
whatsHere	Item::NOTHING (0)	Item
north	@0x22b19a8	MazeCell
south	0x0	MazeCell *
west	0x0	MazeCell *
whatsHere	Item::NOTHING (0)	Item

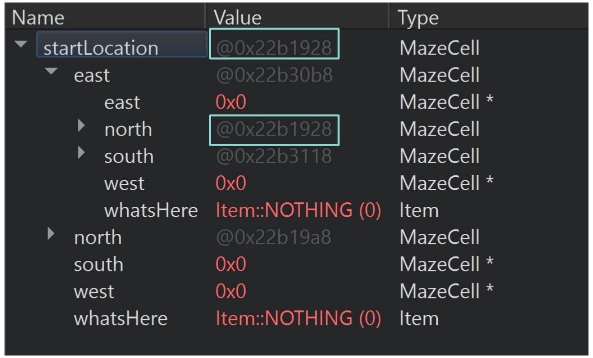
If I move **east,** which direction moves me back to the start location?

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part…. but this one's a bit different.



To determine whether a neighboring pointer moves you to a new location, you can look at the memory address

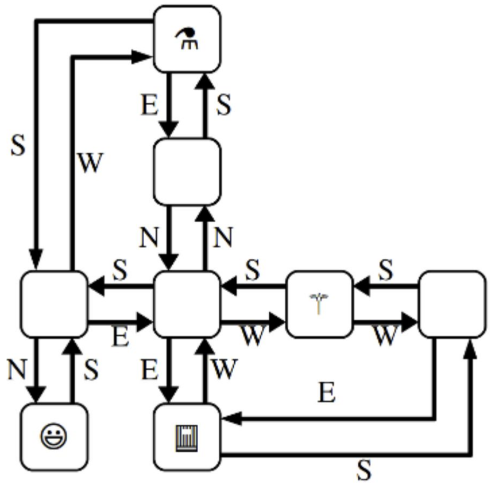


```
Name                  Value                  Type
▾ startLocation       @0x22b1928             MazeCell
  ▾ east              @0x22b30b8             MazeCell
      east            0x0                    MazeCell *
    ▸ north           @0x22b1928             MazeCell
    ▸ south           @0x22b3118             MazeCell
      west            0x0                    MazeCell *
      whatsHere       Item::NOTHING (0)      Item
  ▸ north             @0x22b19a8             MazeCell
    south             0x0                    MazeCell *
    west              0x0                    MazeCell *
    whatsHere         Item::NOTHING (0)      Item
```

If I move **east,** which direction moves me back to the start location?

Part 1.6: Escape from a *twisty* labyrinth

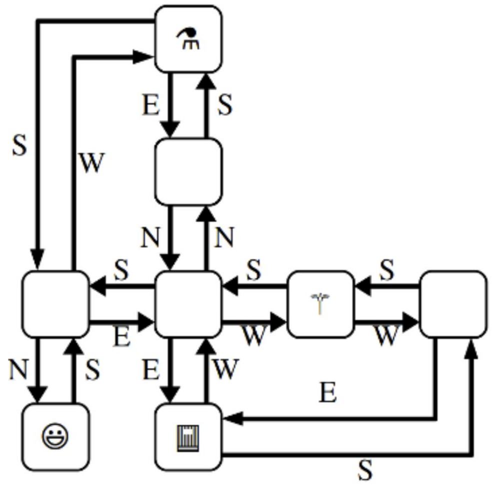
In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.



In this way, you will (slowly and carefully) figure out which links bring you to new maze locations and which ones turn you in loops. Pay attention to the memory addresses at all times because you might end up looping back to a cell that you had visited long ago!

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part.... but this one's a bit different.

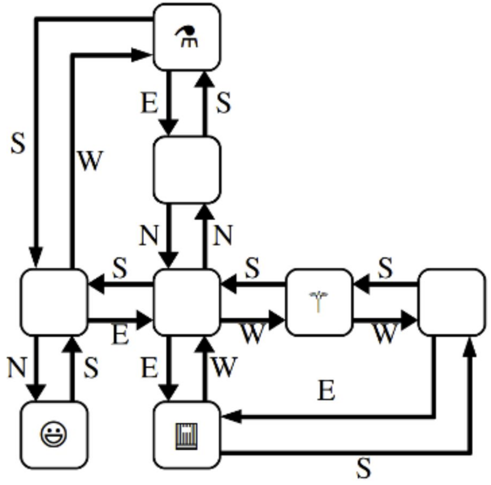


In this way, you will (slowly and carefully) figure out which links bring you to new maze locations and which ones turn you in loops. Pay attention to the memory addresses at all times because you might end up looping back to a cell that you had visited long ago!

Once you have the string path to get you out, fill it in at **`ThePathOutOfMyTwistyMaze`** and you'll be good to go!

Part 1.6: Escape from a *twisty* labyrinth

In the last portion of part 1, your job is to escape from a labyrinth like the one from the last part…. but this one's a bit different.

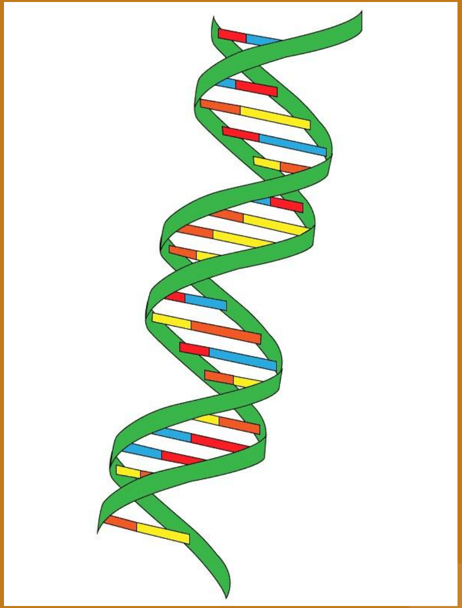


In this way, you will (slowly and carefully) figure out which links bring you to new maze locations and which ones turn you in loops. Pay attention to the memory addresses at all times because you might end up looping back to a cell that you had visited long ago!

Once you have the string path to get you out, fill it in at **`ThePathOutOfMyTwistyMaze`** and you'll be good to go!

One last point -- if you re-run your program, the memory addresses will become different, so try and do this in one run!

Part 2: DNA



Overview

We are using a **doubly** linked list to represent DNA!

Overview

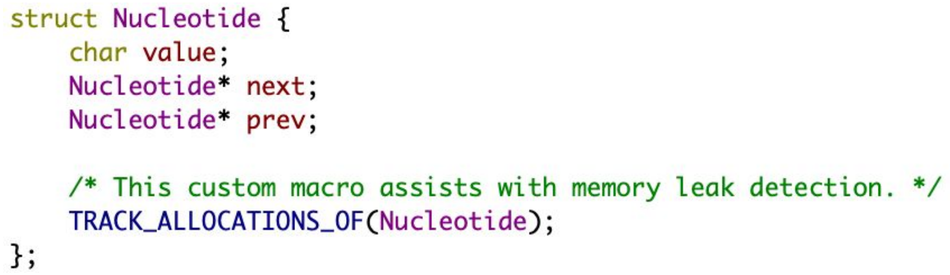
We are using a **doubly** linked list to represent DNA!

The unit we will be using is a **Nucleotide**:

Overview

We are using a **doubly** linked list to represent DNA!

The unit we will be using is a **Nucleotide**:



```
struct Nucleotide {
    char value;
    Nucleotide* next;
    Nucleotide* prev;

    /* This custom macro assists with memory leak detection. */
    TRACK_ALLOCATIONS_OF(Nucleotide);
};
```

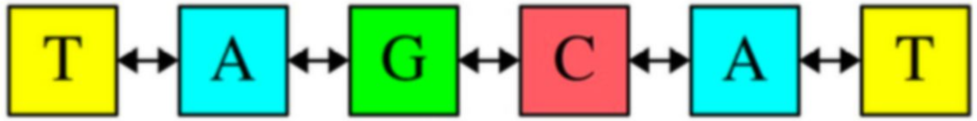
Overview

We are using a **doubly** linked list to represent DNA!

The unit we will be using is a **Nucleotide**:

An example of what a doubly linked list looks like:

Overview

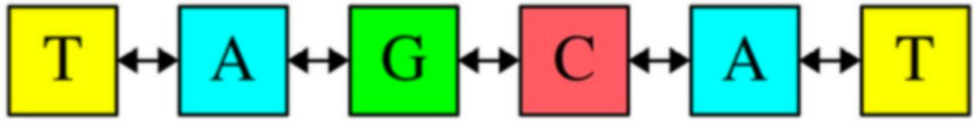


We are using a **doubly** linked list to represent DNA!

The unit we will be using is a **Nucleotide**:

An example of what a doubly linked list looks like:

Overview



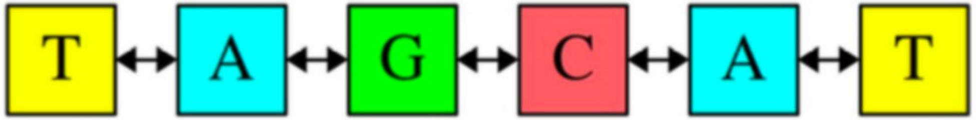
We are using a **doubly** linked list to represent DNA!

The unit we will be using is a **Nucleotide**:

An example of what a doubly linked list looks like:

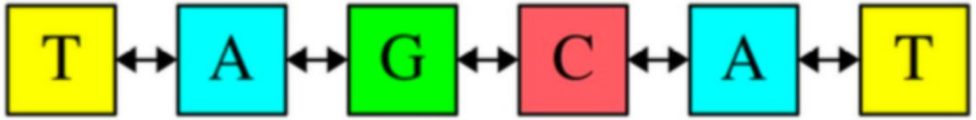
Note that the first **T** has a **nullptr** as its **prev** pointer, and the last **T** has a **nullptr** as its **next** pointer.

Part 1



You are tasked with implementing two core functions.

Part 1

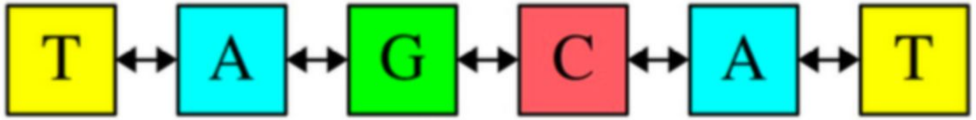


You are tasked with implementing two core functions.

Function 1: Given a pointer to the beginning of a DNA strand, free all of the associate memory.

```
void deleteNucleotides(Nucleotide* dna);
```

Part 1



You are tasked with implementing two core functions.

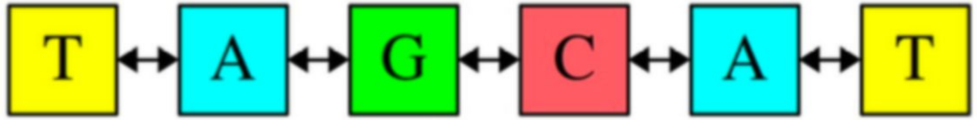
Function 1: Given a pointer to the beginning of a DNA strand, free all of the associate memory.

Function 2: Given a pointer to the beginning of a DNA strand, return a string representing the content of the strand.

Using this function on the above strand would return **TAGCAT**.

```
string fromDNA(Nucleotide* dna);
```

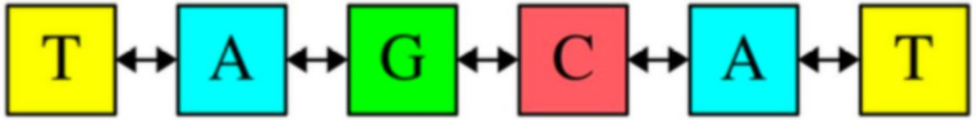
Part 2



Function 3: Given a string representing a DNA strand, build up a corresponding doubly linked list and return a pointer to the beginning of the strand.

```
Nucleotide* toStrand(const string& dna);
```

Part 2



Function 3: Given a string representing a DNA strand, build up a corresponding doubly linked list and return a pointer to the beginning of the strand.

Calling this function give **TAGCAT** would construct the strand above.

```
Nucleotide* toStrand(const string& dna);
```

Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
    - End of lecture 21.
    - Problems 6 to 9 in Section 7.

Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
  - End of lecture 21.
  - Problems 6 to 9 in Section 7.
- **NO RECURSION!**

Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
  - End of lecture 21.
  - Problems 6 to 9 in Section 7.
- **NO RECURSION!**
- Do not use other containers.

Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
  - End of lecture 21.
  - Problems 6 to 9 in Section 7.
- **NO RECURSION!**
- Do not use other containers.
- No memory allocation in part 1, yes memory allocation in part 2.

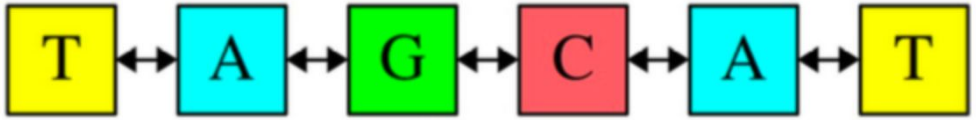
Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
    - End of lecture 21.
    - Problems 6 to 9 in Section 7.
- **NO RECURSION!**
- Do not use other containers.
- No memory allocation in part 1, yes memory allocation in part 2.
- Draw pictures! Even Keith does that :)

Part 1 && 2 Notes

- If you are having trouble with doubly linked list:
  - End of lecture 21.
  - Problems 6 to 9 in Section 7.
- **NO RECURSION!**
- Do not use other containers.
- No memory allocation in part 1, yes memory allocation in part 2.
- Draw pictures! Even Keith does that :)
- Test your code before moving on. These functions are the foundation moving forward.

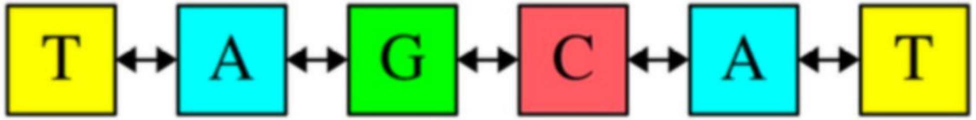
Part 3



Function 4: Given a pointer to the beginning of a DNA strand **dna**, look for a target sequence **target**.

```
Nucleotide* findFirst(Nucleotide* dna, Nucleotide* target);
```

Part 3

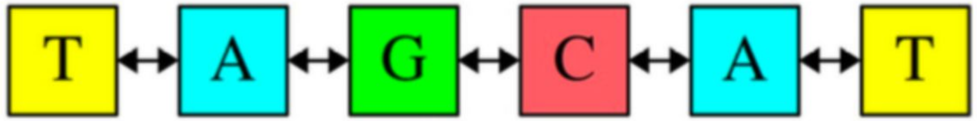


Function 4: Given a pointer to the beginning of a DNA strand **dna**, look for a target sequence **target**.

If found, return a pointer to the beginning of that sequence.

```
Nucleotide* findFirst(Nucleotide* dna, Nucleotide* target);
```

Part 3



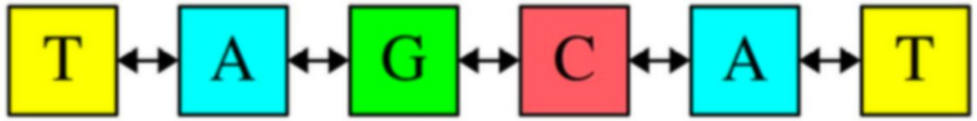
Function 4: Given a pointer to the beginning of a DNA strand **dna**, look for a target sequence **target**.

If found, return a pointer to the beginning of that sequence.

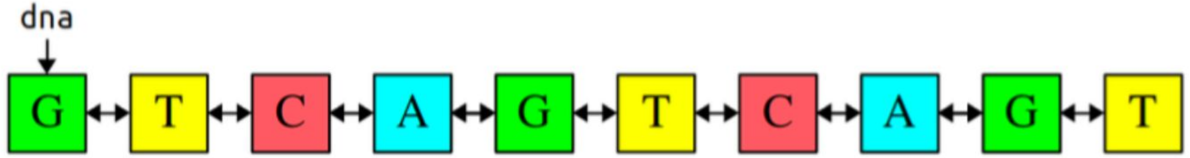
Otherwise, return a **nullptr**.

```
Nucleotide* findFirst(Nucleotide* dna, Nucleotide* target);
```

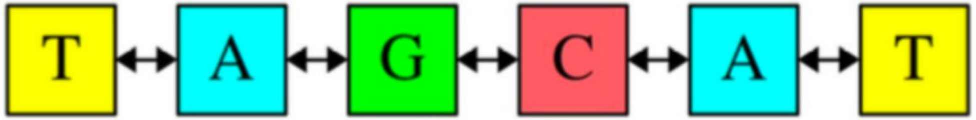
Part 3



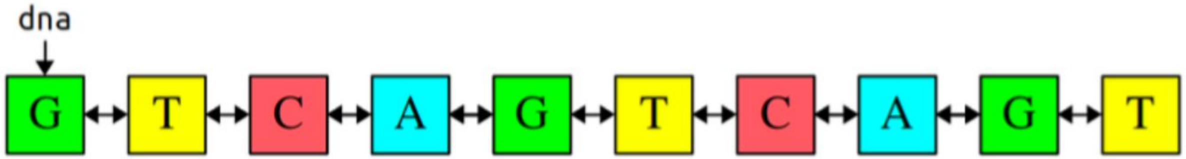
Given this dna:



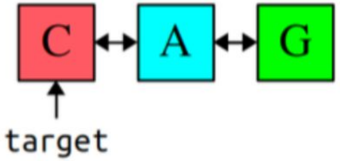
Part 3



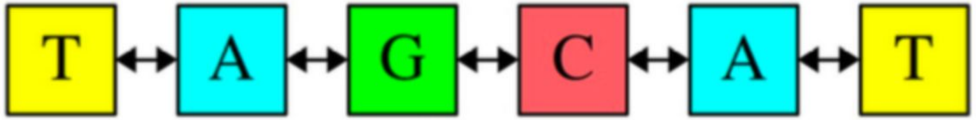
Given this dna:



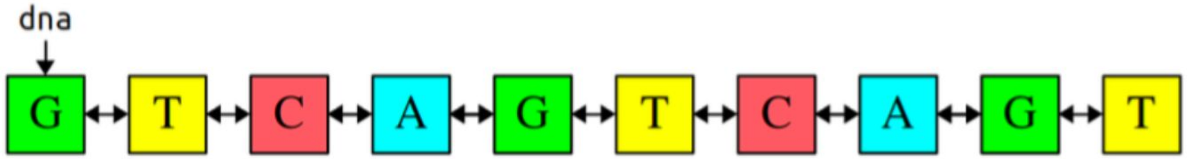
And this target:



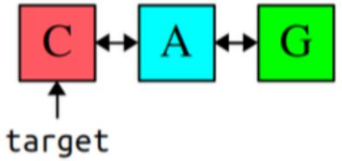
Part 3



Given this dna:

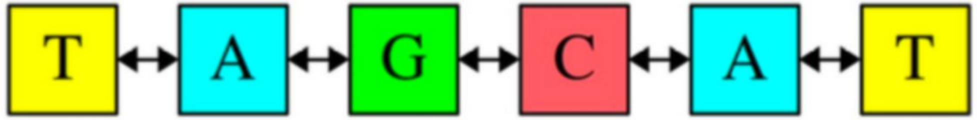


And this target:

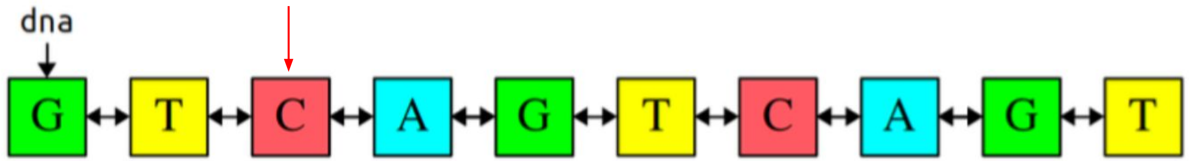


Where should the pointer we are returning be pointing to?

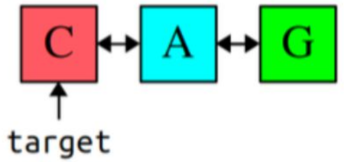
Part 3



Given this dna:



And this target:



Where should the pointer we are returning be pointing to?

Part 3 Notes

- No recursion, no container, new memory allocation, no problem :)

Part 3 Notes

- No recursion, no container, new memory allocation, no problem :)
- If the target sequence is empty, you should return a pointer to the beginning of the DNA strand.

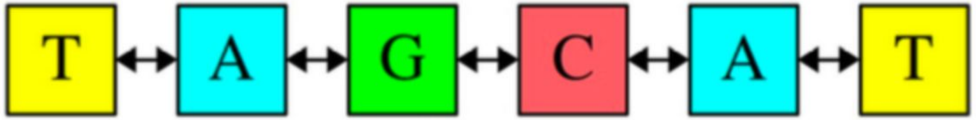
Part 3 Notes

- No recursion, no container, new memory allocation, no problem :)
- If the target sequence is empty, you should return a pointer to the beginning of the DNA strand.
- You may assume that **target** and **dna** are pointing to completely different **Nucleotide** objects

Part 3 Notes

- No recursion, no container, new memory allocation, no problem :)
- If the target sequence is empty, you should return a pointer to the beginning of the DNA strand.
- You may assume that **target** and **dna** are pointing to completely different **Nucleotide** objects
- When in doubt, pictures and well-constructed tests are your best friends!

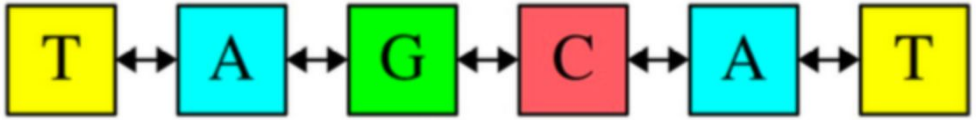
Part 4



Function 5: Given a pointer to the beginning of a DNA strand **dna** *by reference*, look for a target sequence **target**.

```
bool spliceFirst(Nucleotide*& dna, Nucleotide* target);
```

Part 4

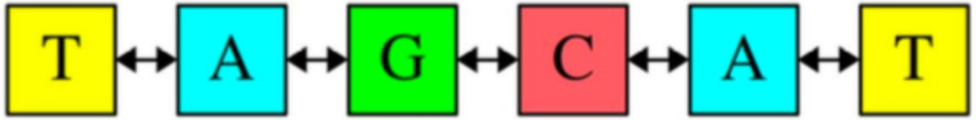


Function 5: Given a pointer to the beginning of a DNA strand **dna** *by reference*, look for a target sequence **target**.

If found, remove the sequence from that strand and return **true**.

```
bool spliceFirst(Nucleotide*& dna, Nucleotide* target);
```

Part 4



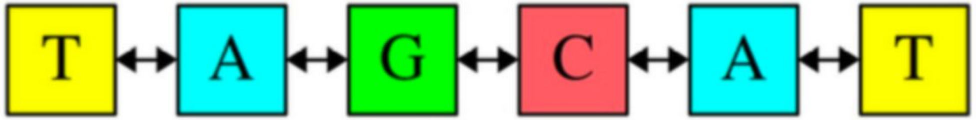
Function 5: Given a pointer to the beginning of a DNA strand **dna** *by reference*, look for a target sequence **target**.

If found, remove the sequence from that strand and return **true**.

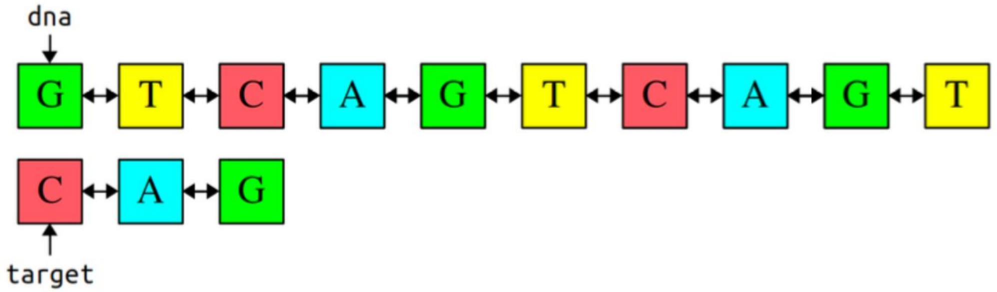
If not found, leave the strand as it is and return **false**.

```
bool spliceFirst(Nucleotide*& dna, Nucleotide* target);
```

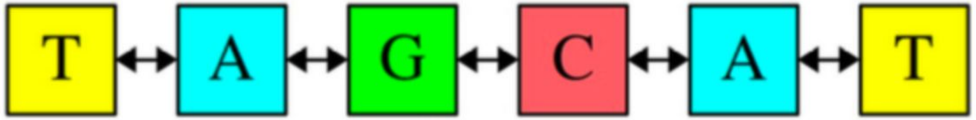
Part 4



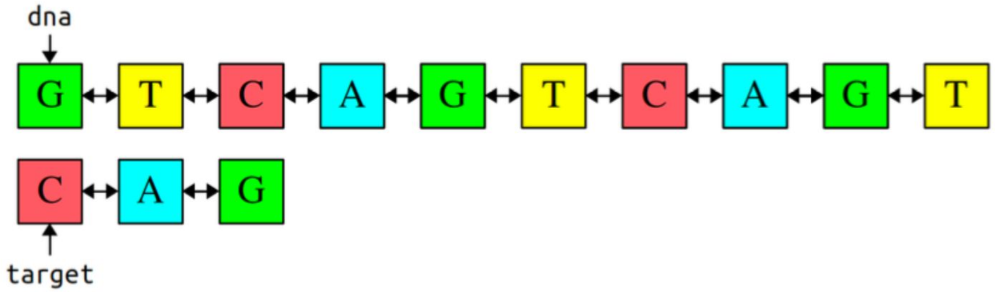
Only the first instance of the sequence should be removed:



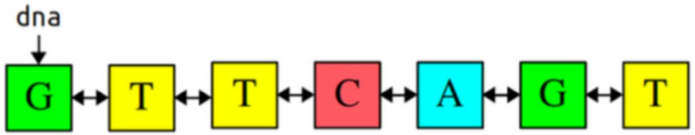
Part 4



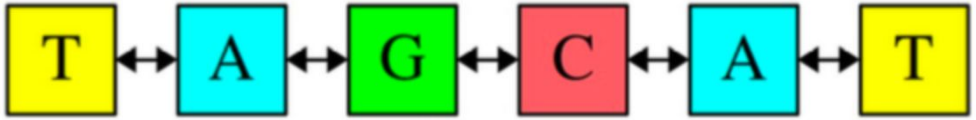
Only the first instance of the sequence should be removed:



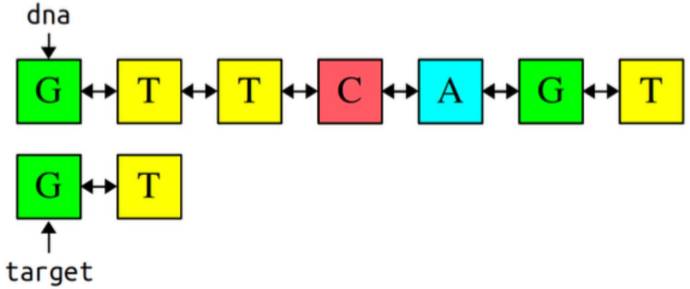
Result:



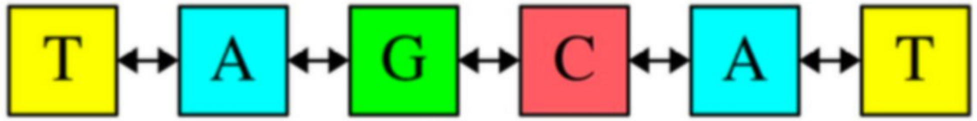
Part 4



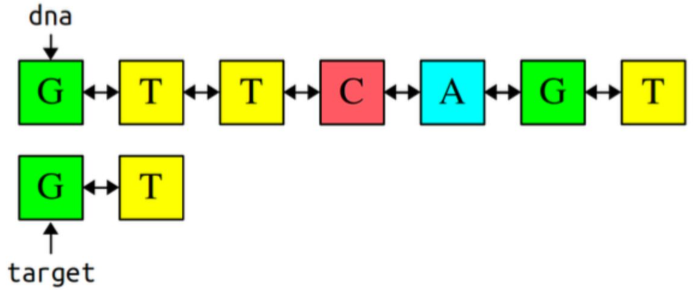
dna might be updated in the process (that is why we passed it in by reference):



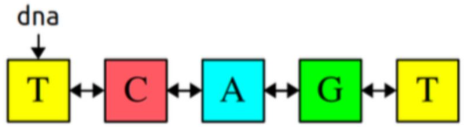
Part 4



dna might be updated in the process (that is why we passed it in by reference):



Result:



Part 4 Notes

- Utilize the earlier parts.

Part 4 Notes

- Utilize the earlier parts.
- Free the sub-strand you sliced out.

Part 4 Notes

- Utilize the earlier parts.
- Free the sub-strand you sliced out.
- Every note from before holds.