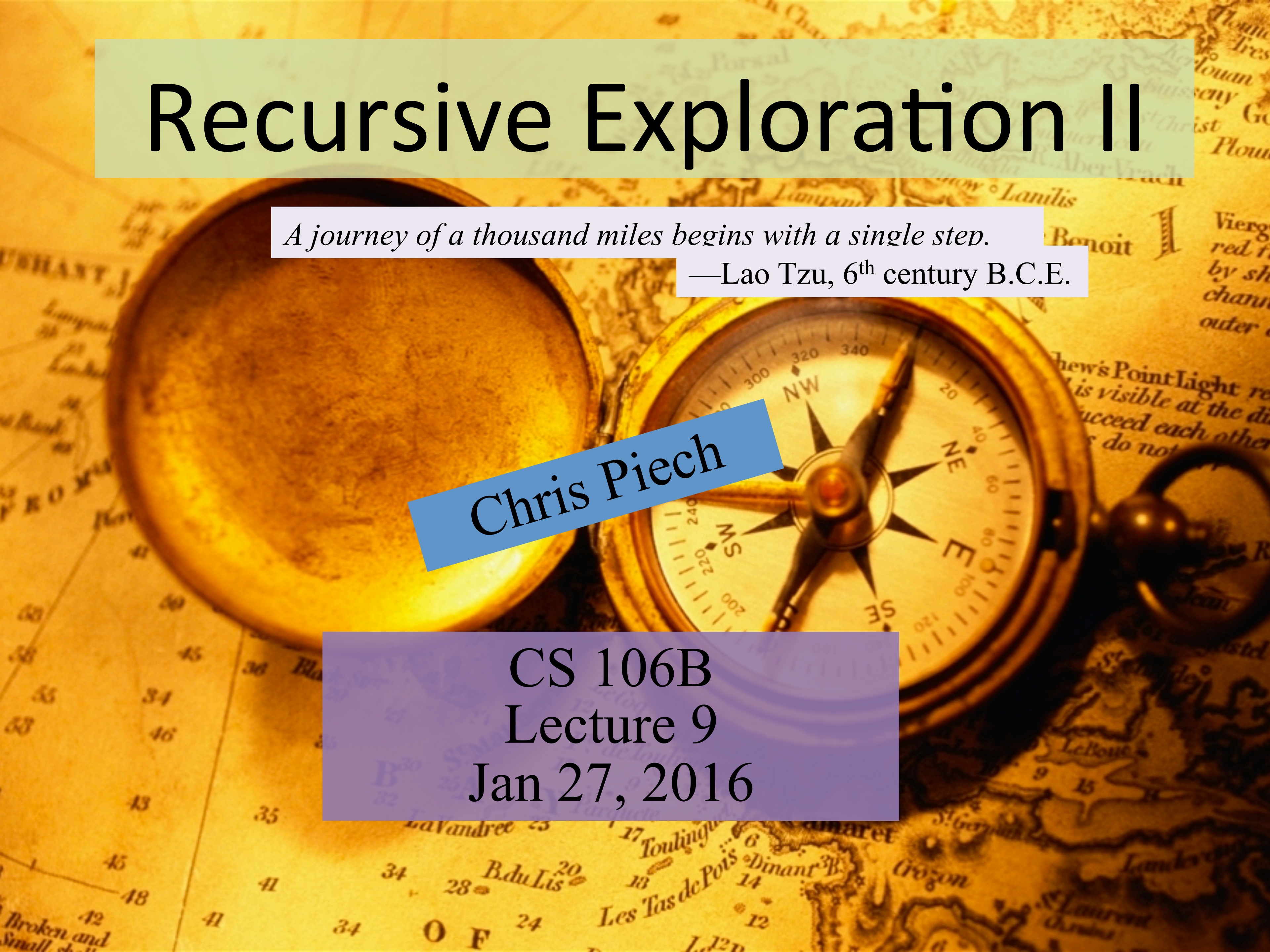


Recursive Exploration II

A journey of a thousand miles begins with a single step.

—Lao Tzu, 6th century B.C.E.

Chris Piech

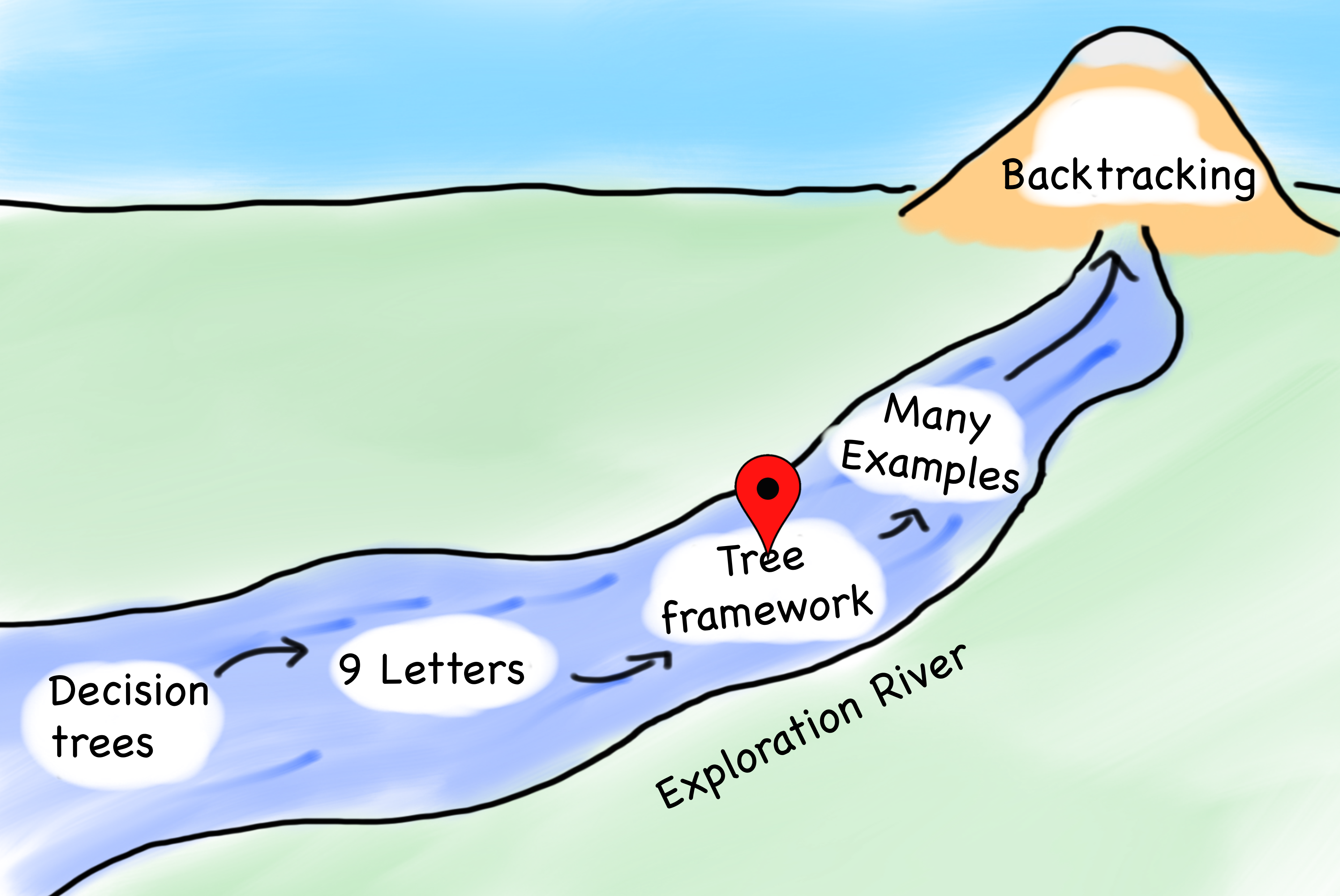
CS 106B
Lecture 9
Jan 27, 2016

One Line Change to Starter

META

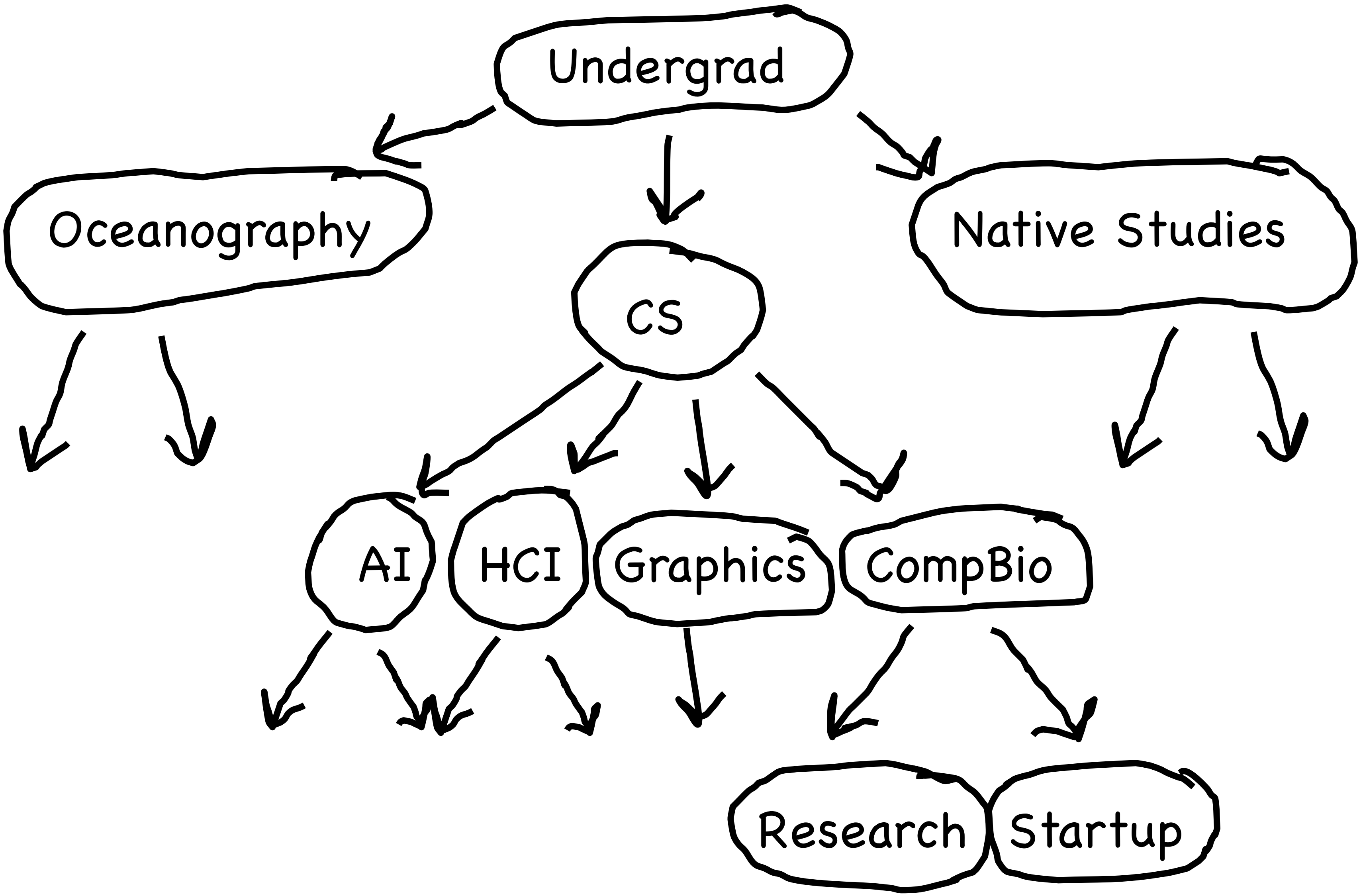
ACADEMY

Today's Route

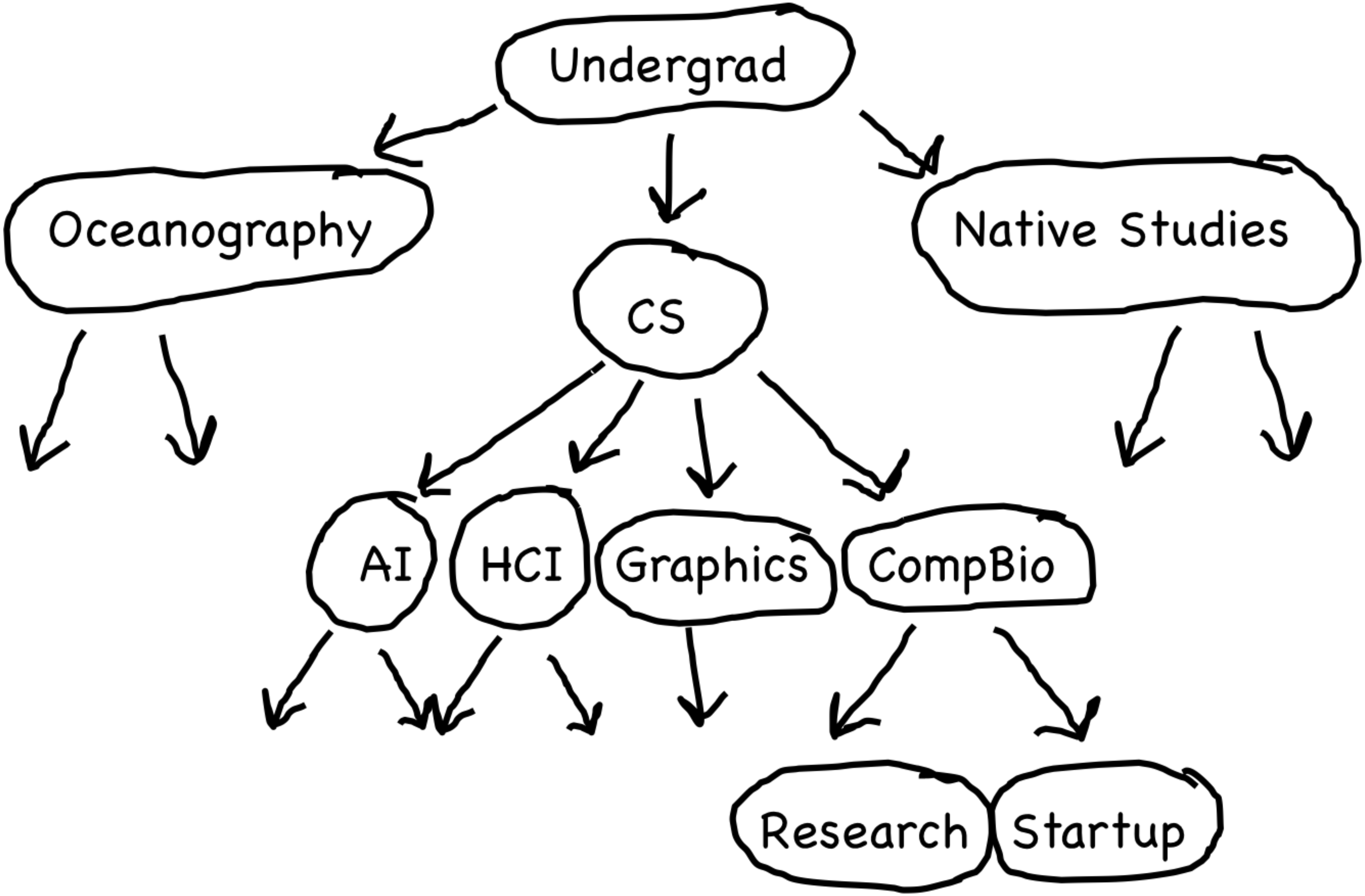


Decision Trees

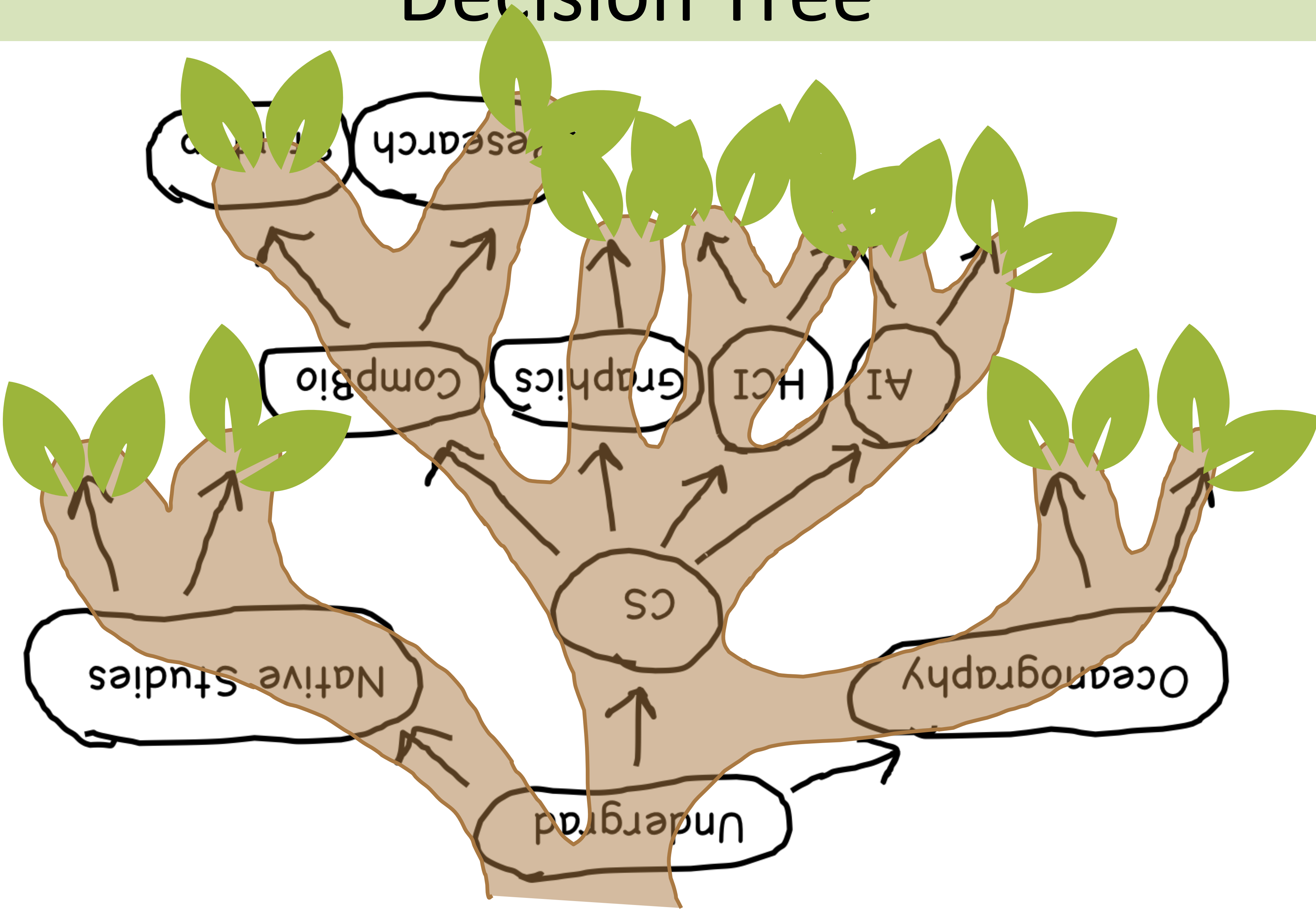
Decision Tree



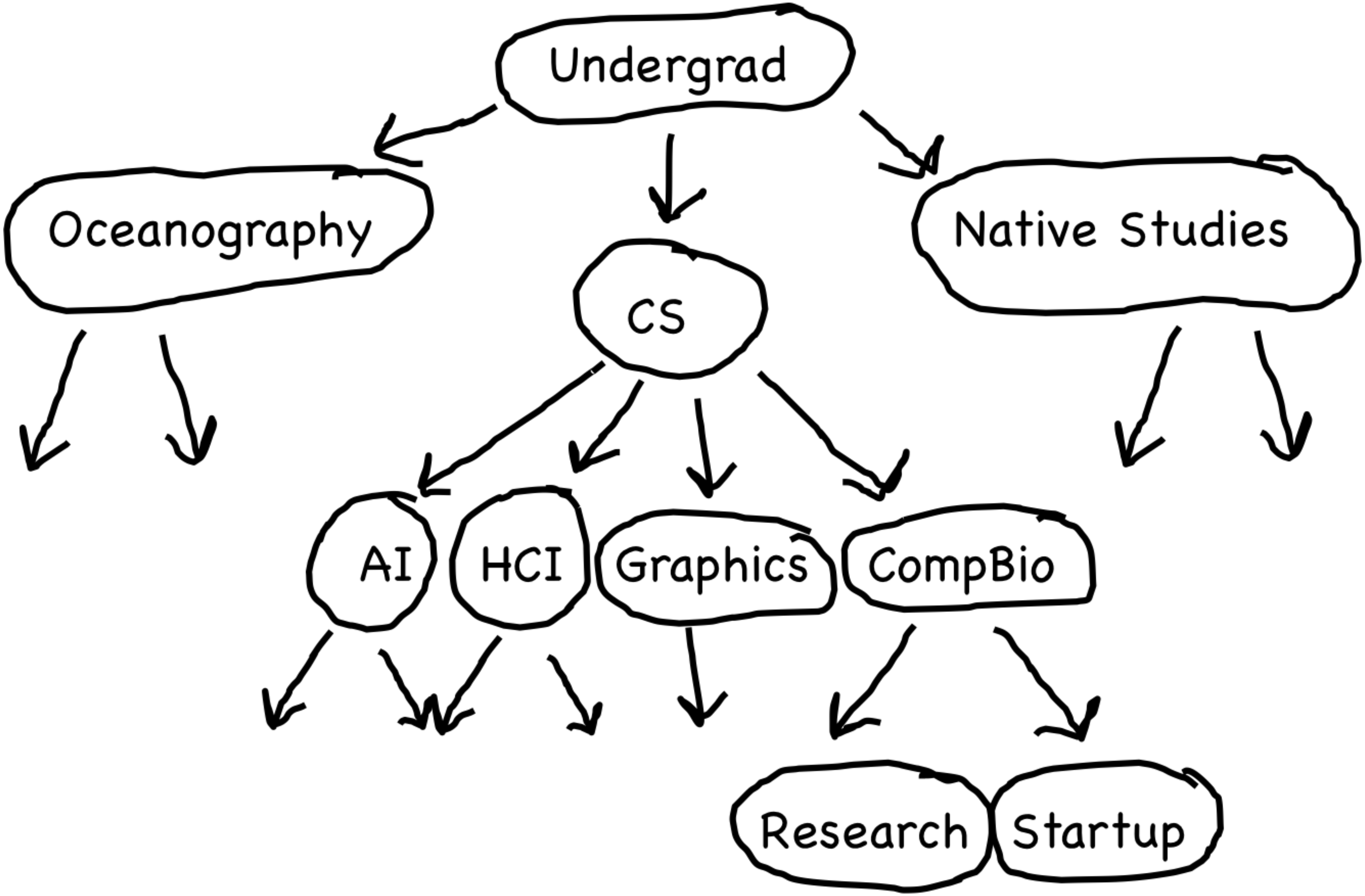
Decision Tree



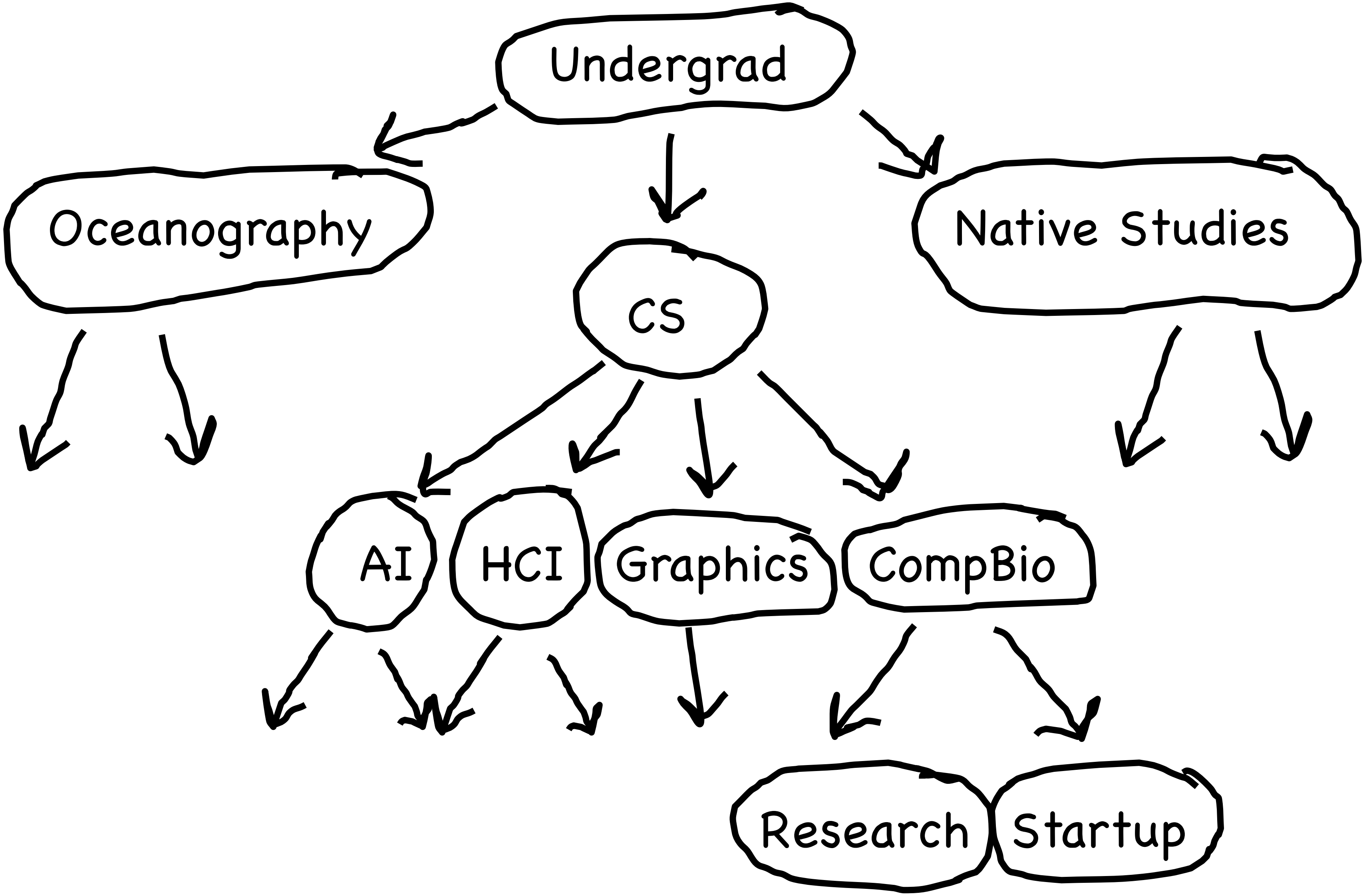
Decision Tree



Decision Tree



Decision Tree



Undergrad

Oceanography

CS

Native Studies

AI

HCI

Graphics

CompBio

Research

Startup

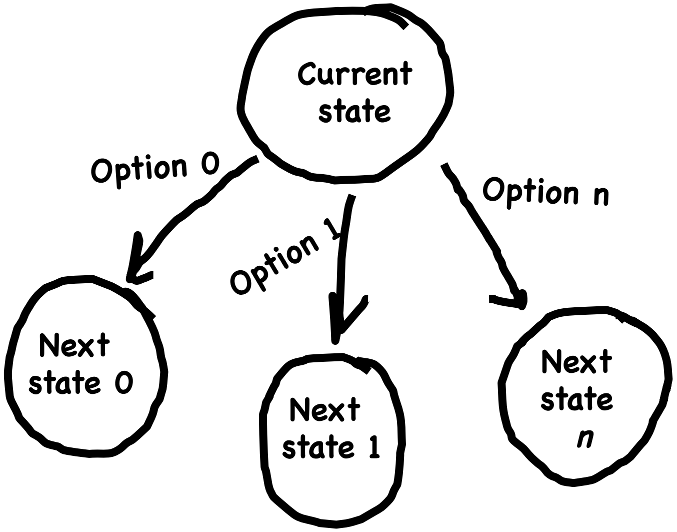
Tree Template

Tree Template

```
// a general template for working with trees
void recursiveExploration( state ) {
  if (simpleCase || foundSolution) {
     // base case
     return without recursing.
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

State is the top of a tree
And so is nextState

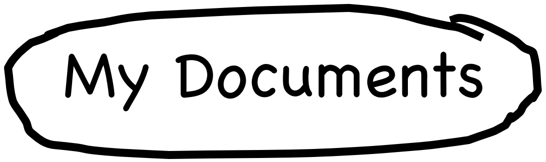
Generally you will do other work



Start Simple

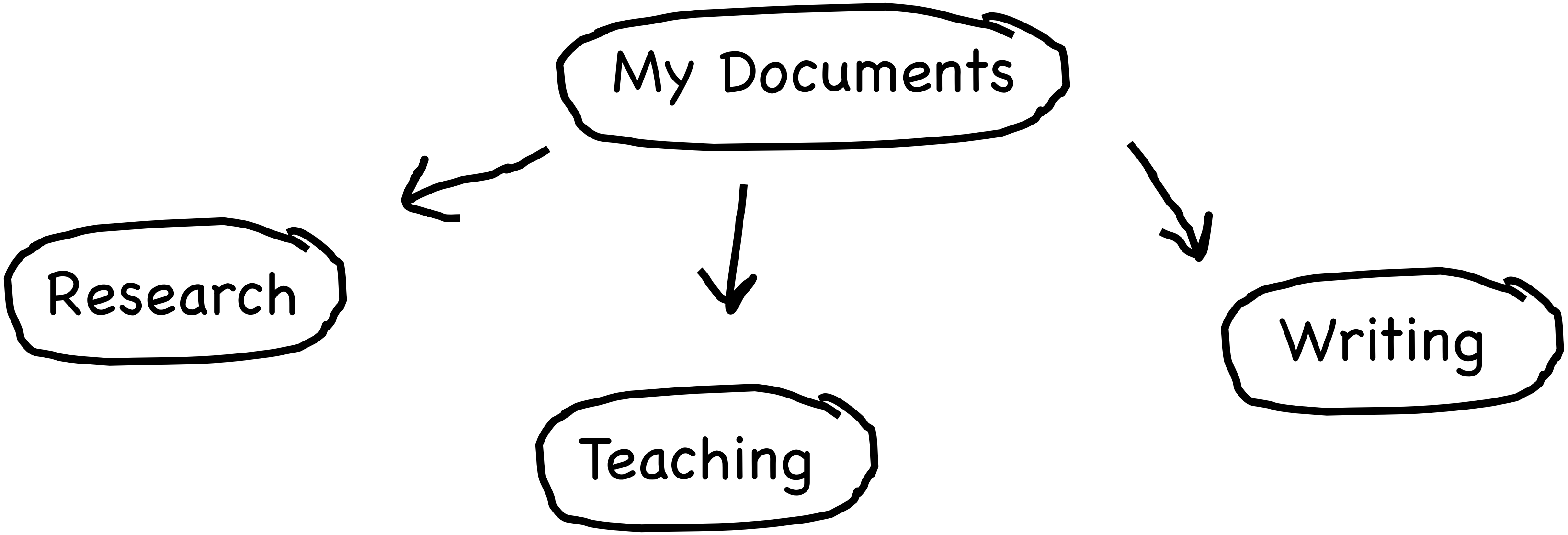
Output all Files on Computer

File System

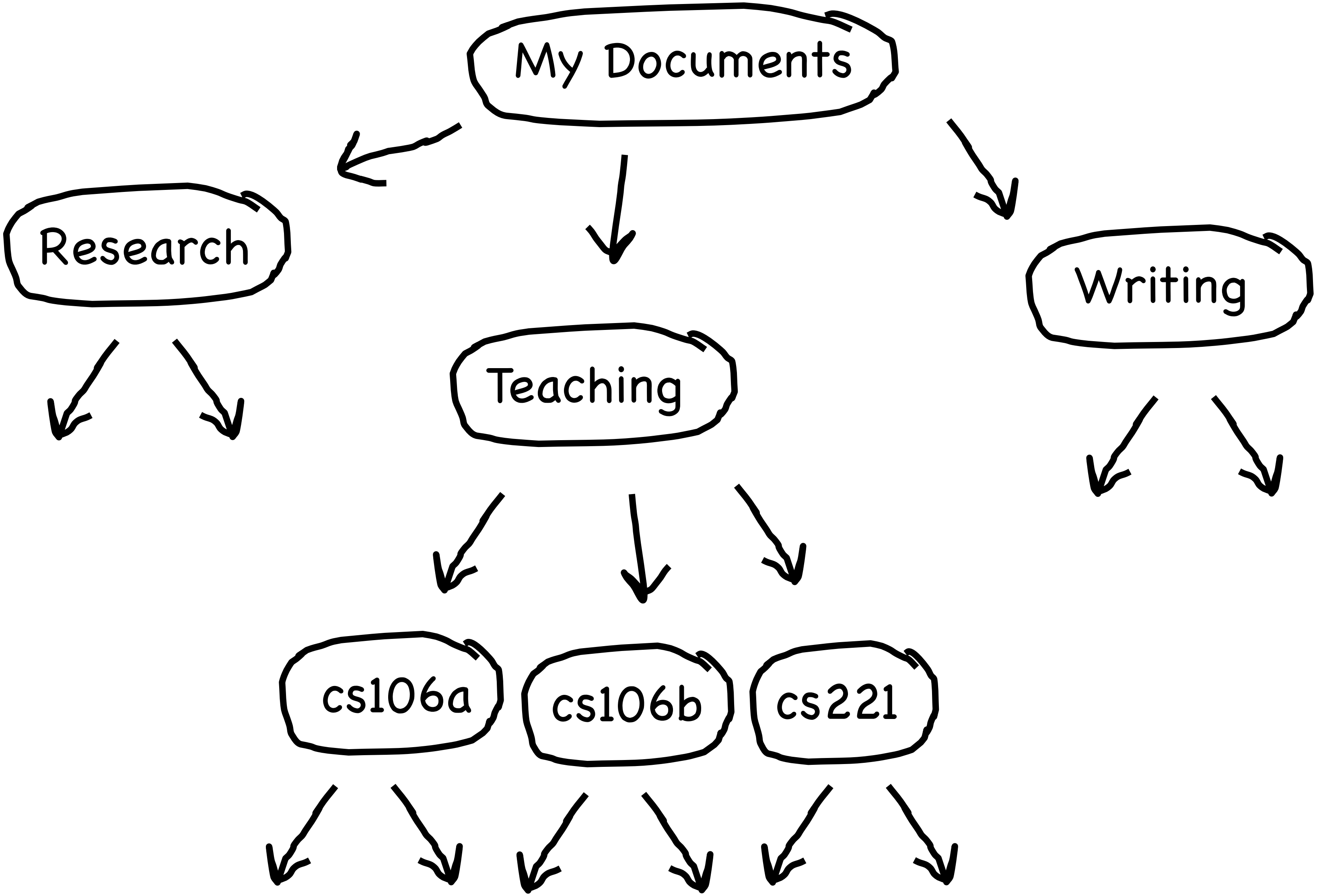


My Documents

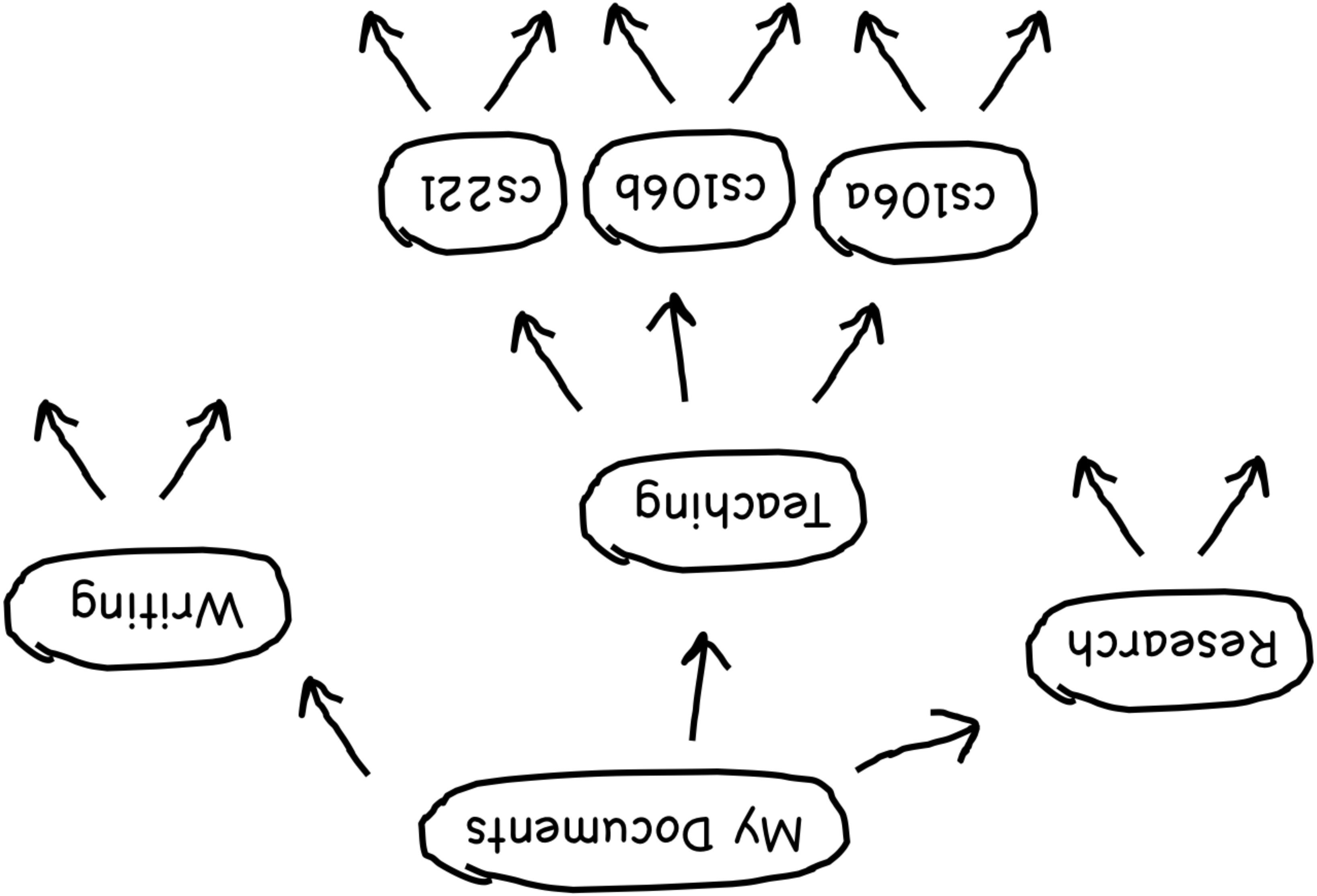
File System



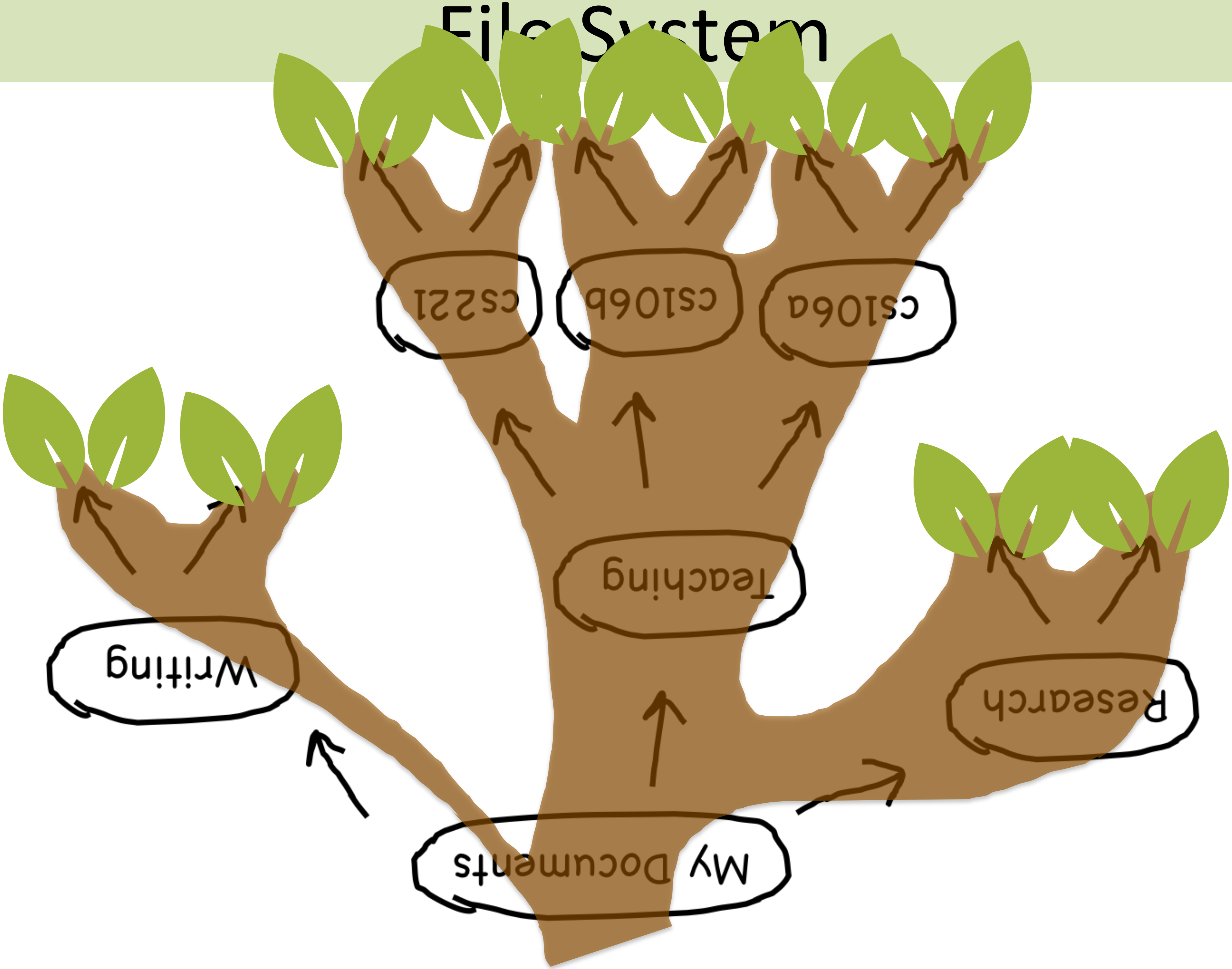
File System


My Documents
Research
Teaching
Writing
cs106a
cs106b
cs221

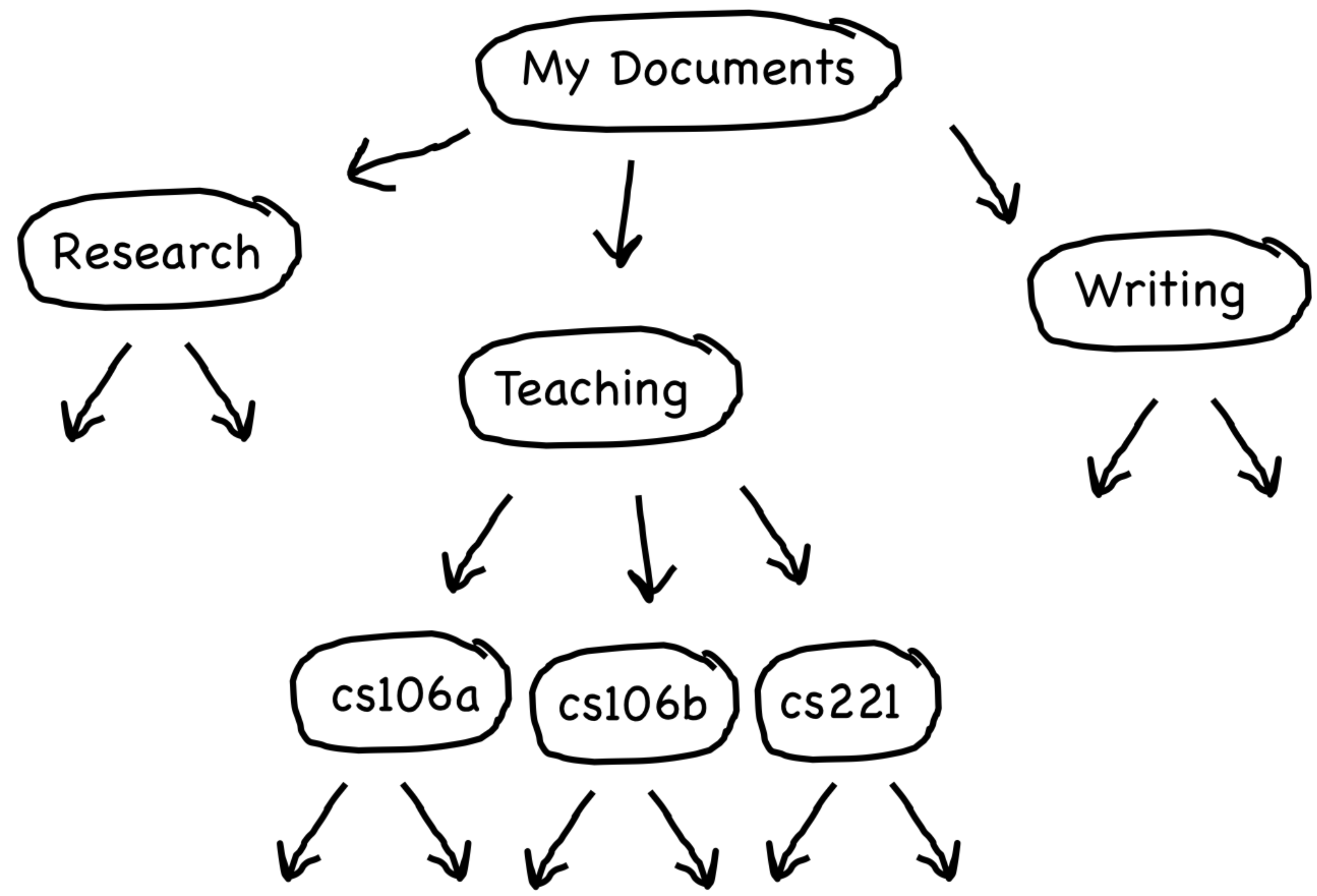
File System


My Documents
Writing
Teaching
Research
cs221
cs106b
cs106a

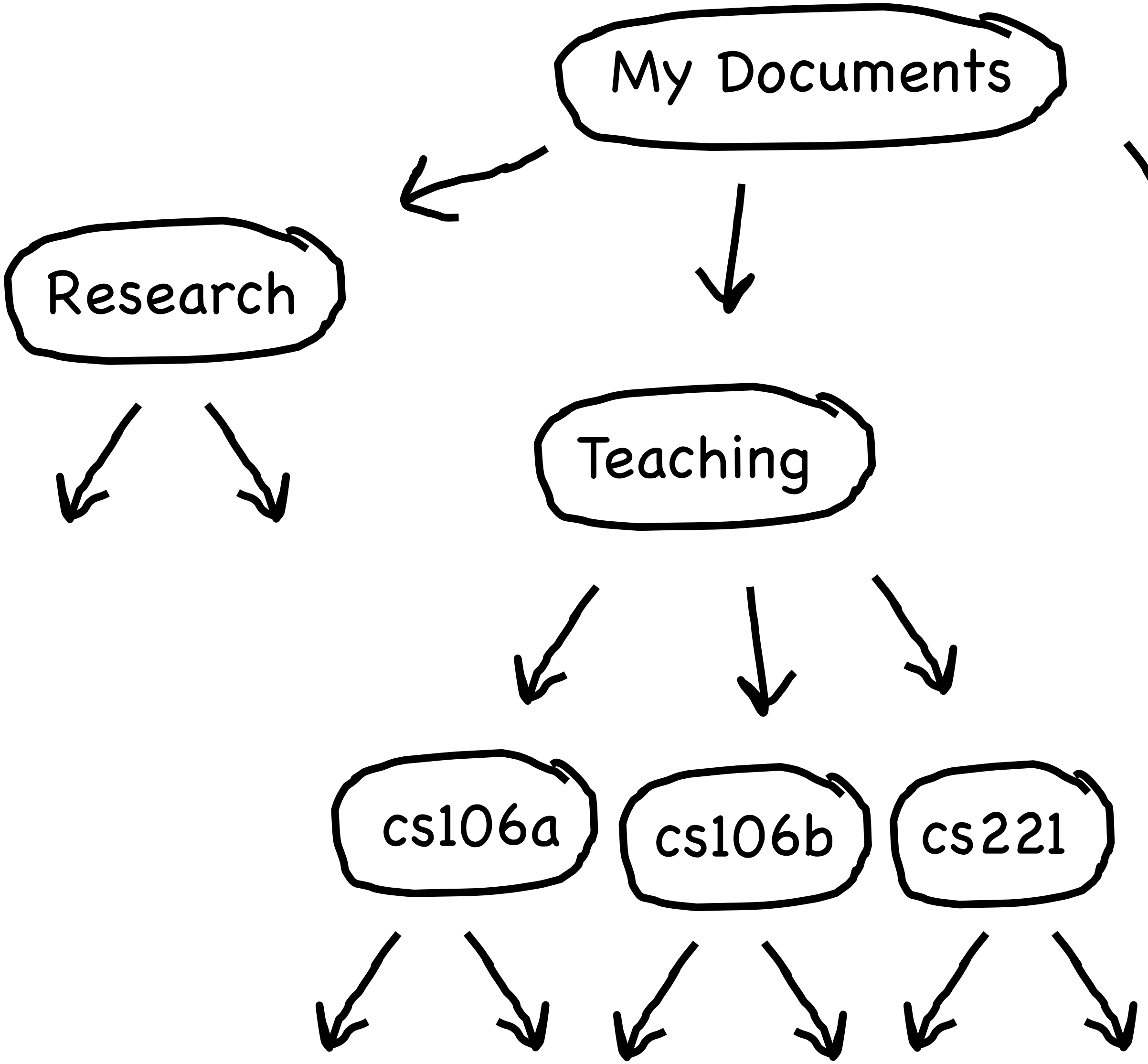
File System


cs221
cs106b
cs106a
Teaching
Writing
Research
My Documents

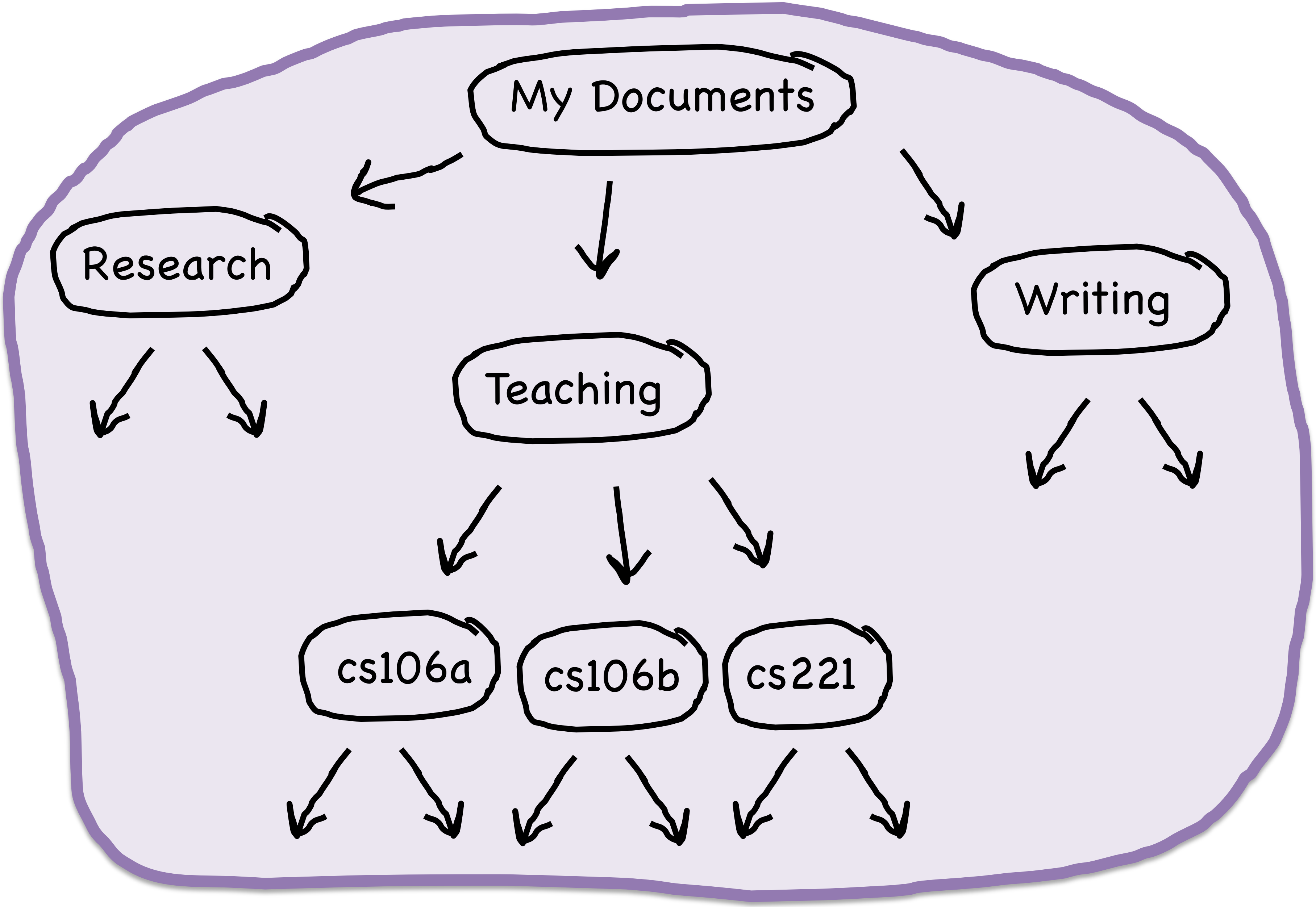
File System


My Documents
Research
Teaching
Writing
cs106a
cs106b
cs221

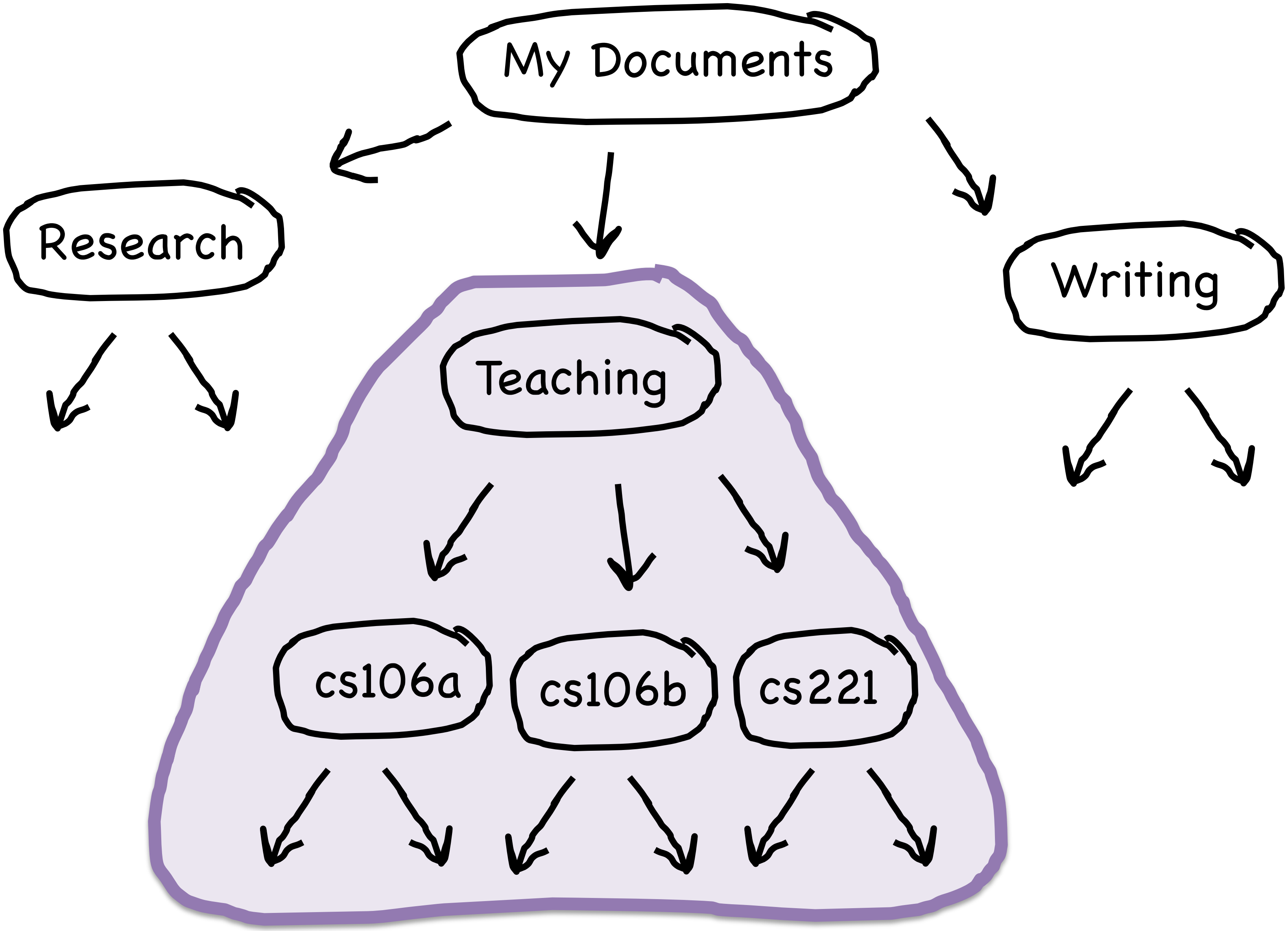
File System


My Documents
Research
Teaching
Writing
cs106a
cs106b
cs221

File System


My Documents
Research
Teaching
Writing
cs106a
cs106b
cs221

File System


My Documents
Research
Teaching
Writing
cs106a
cs106b
cs221

Tree Template

Tree Template

```
// a general template for working with trees
void recursiveExploration( state ) {
  if (simpleCase || foundSolution) {
     // base case
     return without recursing.
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (simpleCase || foundSolution) {
     // base case
     return without recursing.
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (simpleCase || foundSolution) {
     // base case
     return without recursing.
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // get all next states
     Vector<string> dirFiles;
     listDirectory(currDir, dirFiles);

     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // get all next states
     Vector<string> dirFiles;
     listDirectory(currDir, dirFiles);

     // recursive case
     for(each possible nextState from state) {
        recursiveExploration(nextState);
     }
  }
}
```

Tree Template

```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // get all next states
     Vector<string> dirFiles;
     listDirectory(currDir, dirFiles);

     // recursive case
     for(string file : dirFiles) {
        string next = currDir + “/” + file;
        recursiveExploration(next);
     }
  }
}
```

Output All Files

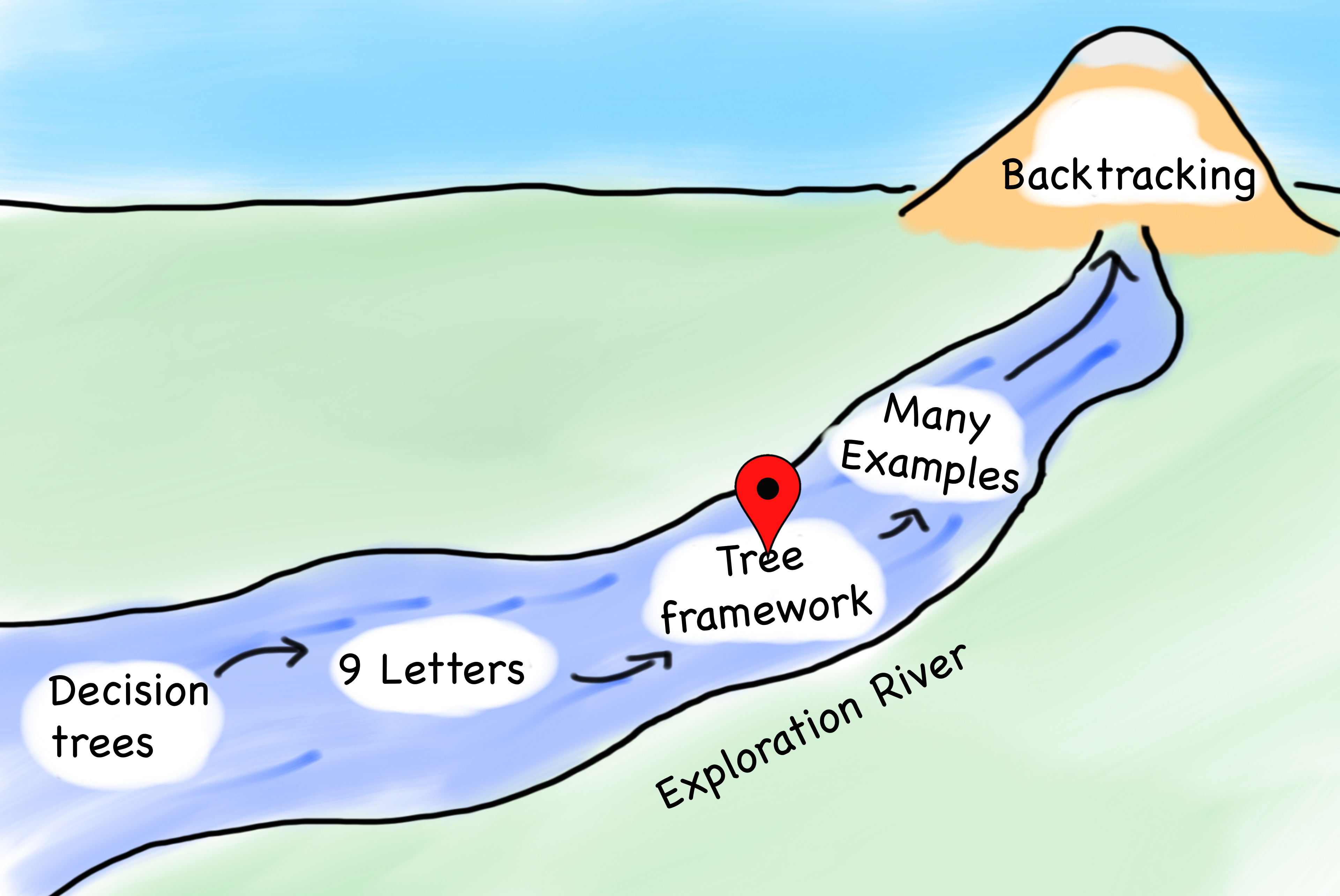
```
// a general template for working with trees
void recursiveExploration(string currDir) {
  if (!isDirectory(currDirr)) {
     // base case
     cout << currDirr << endl;
  } else {
     // get all next states
     Vector<string> dirFiles;
     listDirectory(currDir, dirFiles);

     // recursive case
     for(string file : dirFiles) {
        string next = currDir + “/” + file;
        recursiveExploration(next);
     }
  }
}
```

Recursive Exploration on the Internet

Looking For Treasure

Found Treasure Map but Lost it

Today's Route

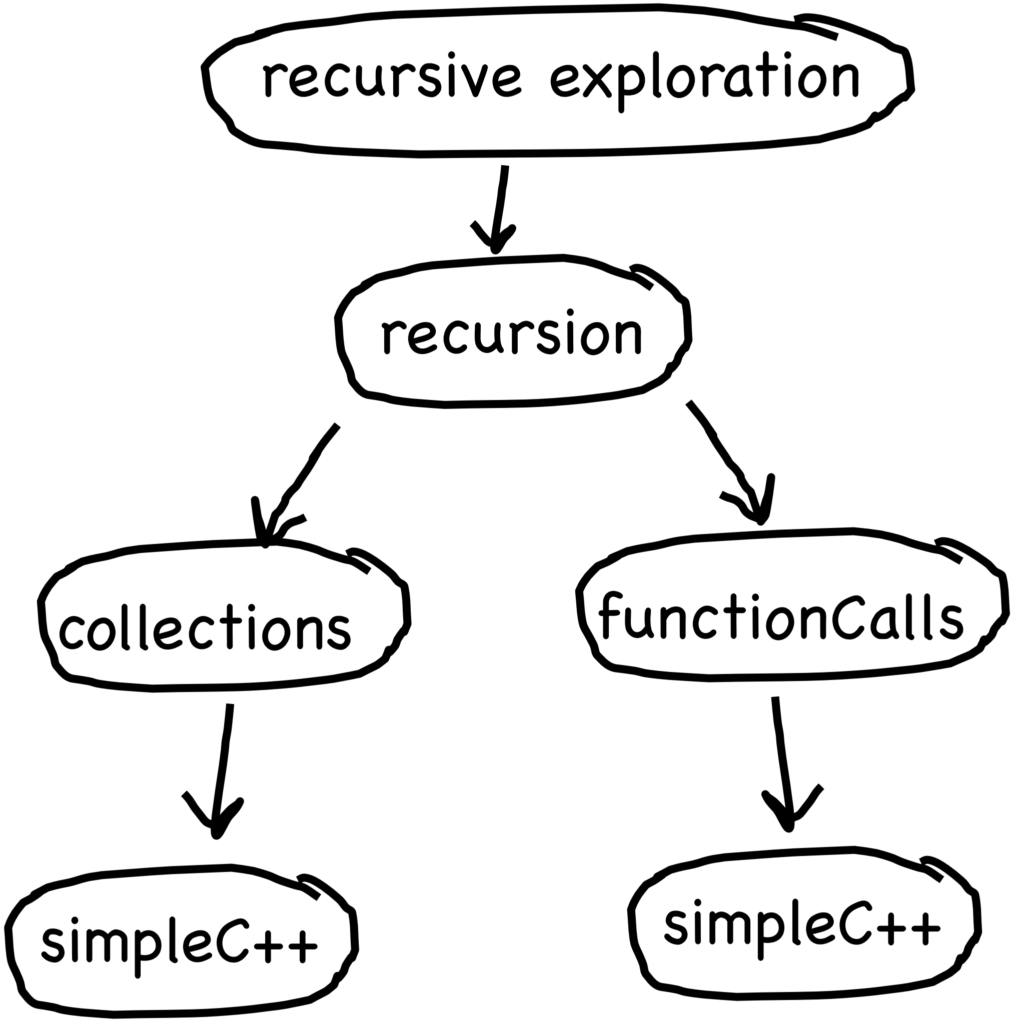


Today's Route

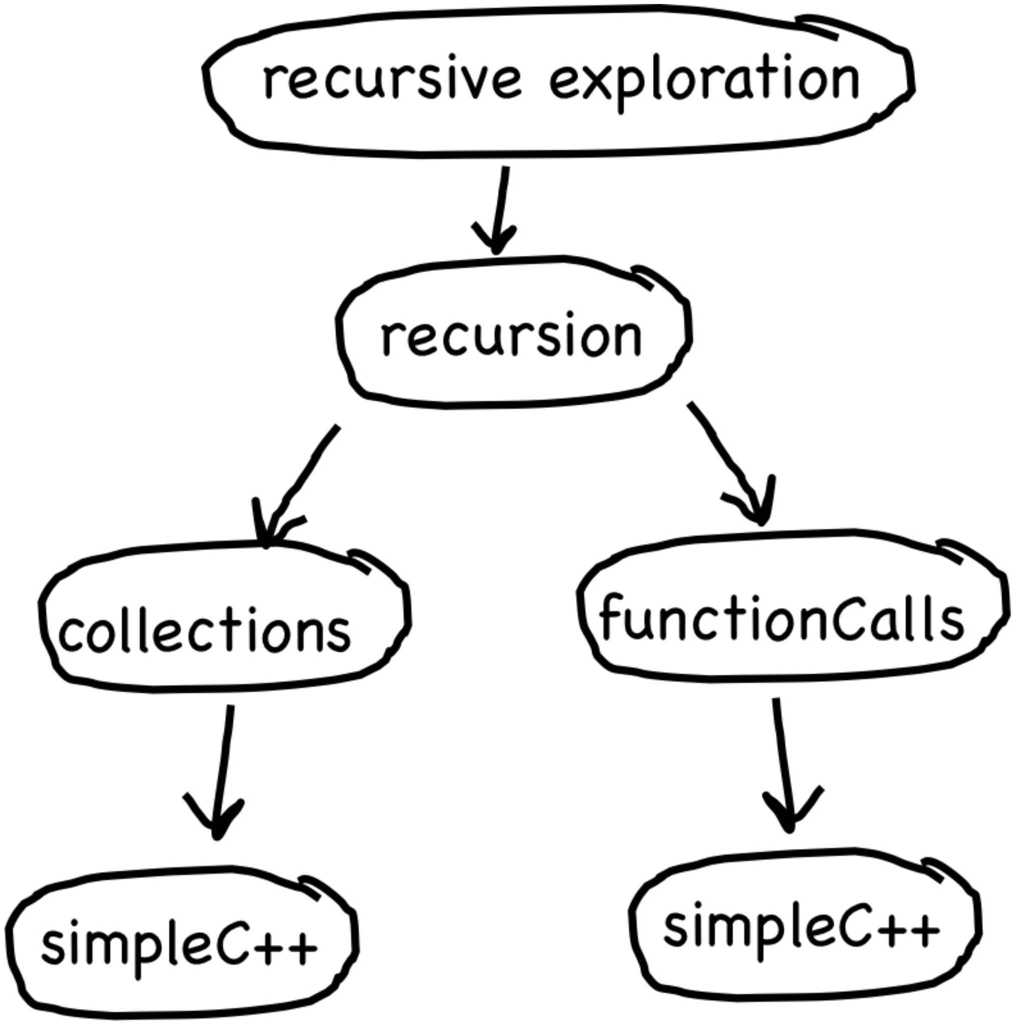
Decision trees
9 Letters
Tree framework
Many Examples
Backtracking
Exploration River

Trees are Everywhere

Prerequisites



Prerequisites



recursive exploration

recursion

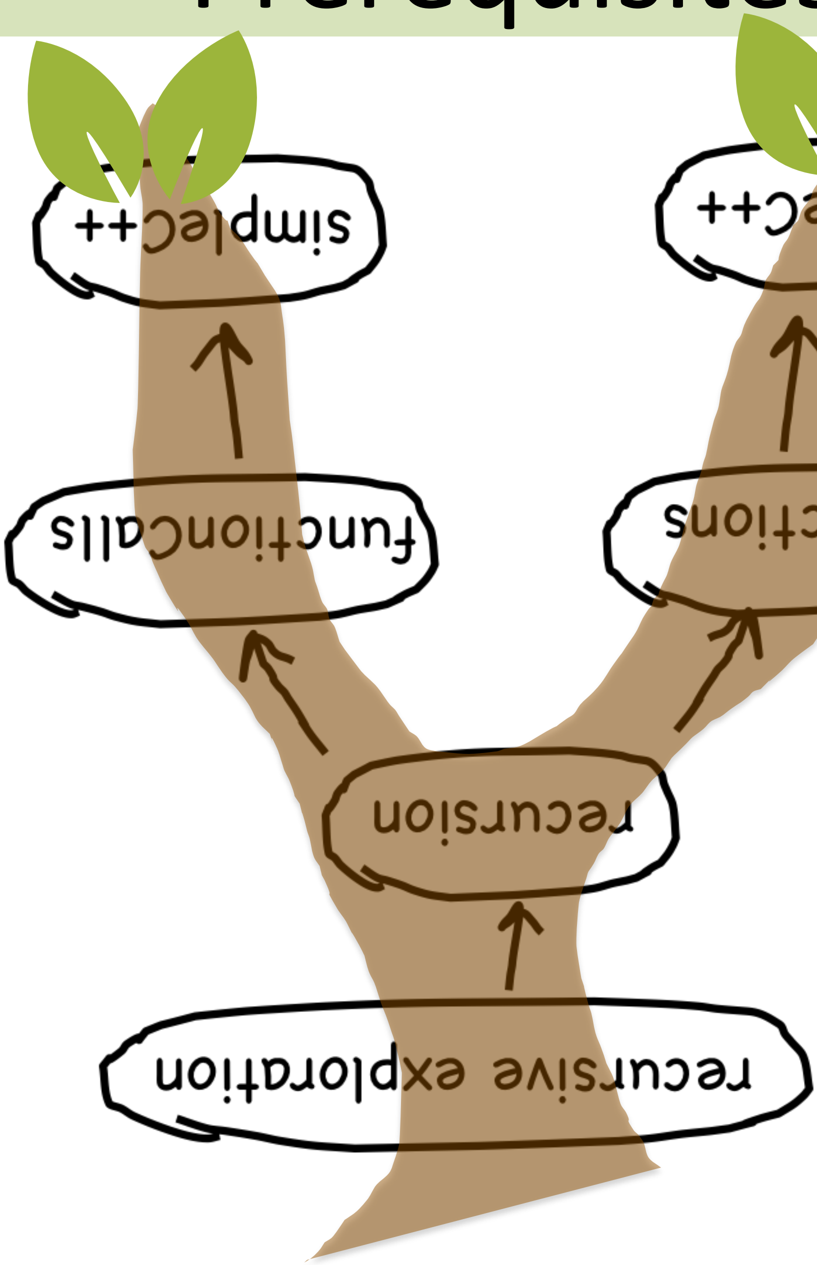
collections

functionCalls

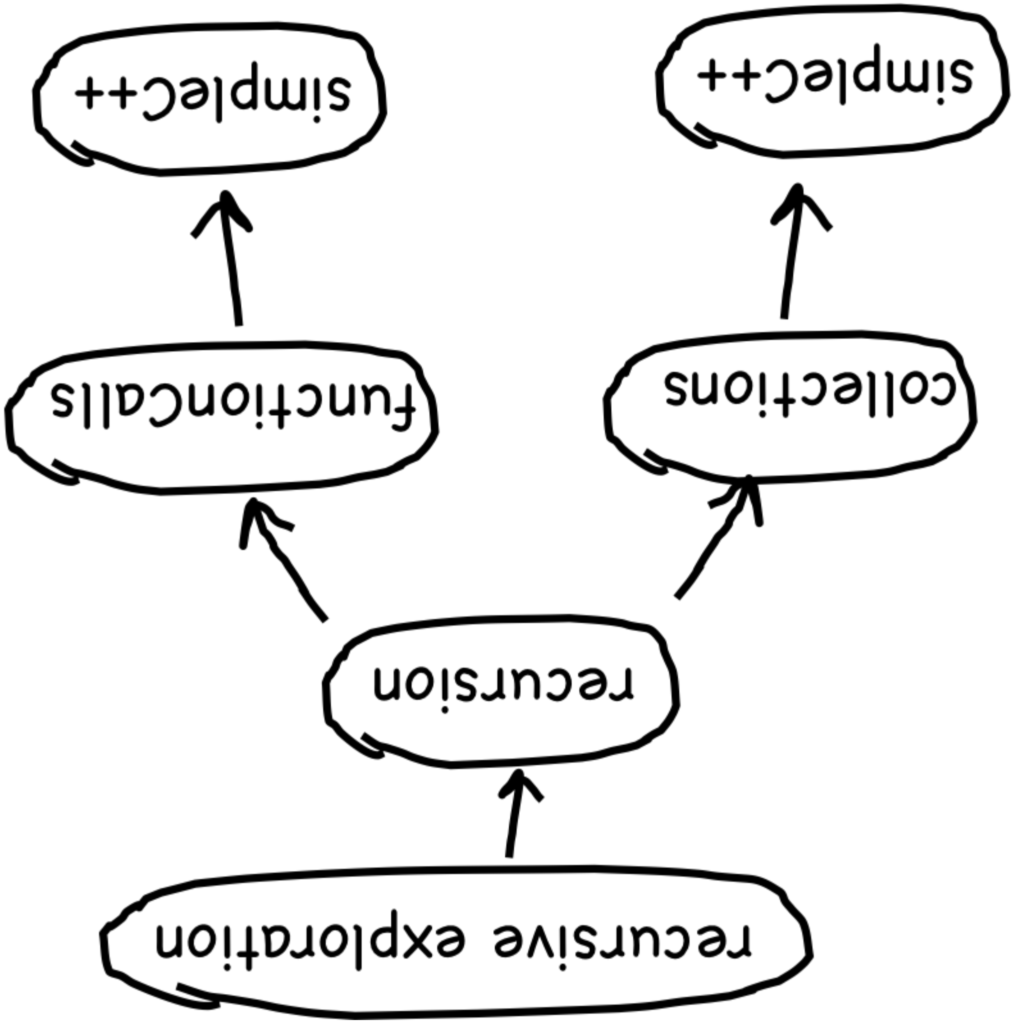
simpleC++

simpleC++

Prerequisites

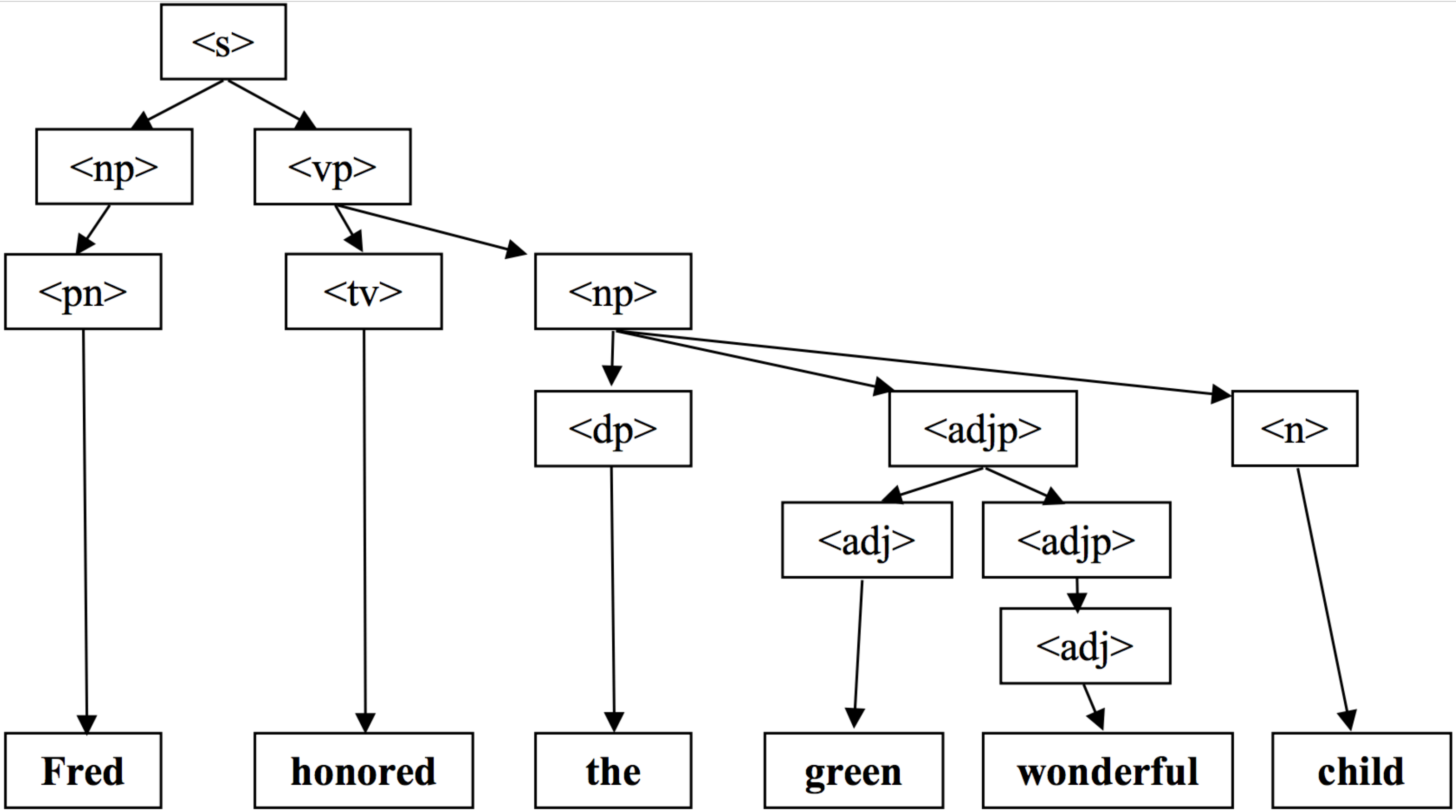


Prerequisites



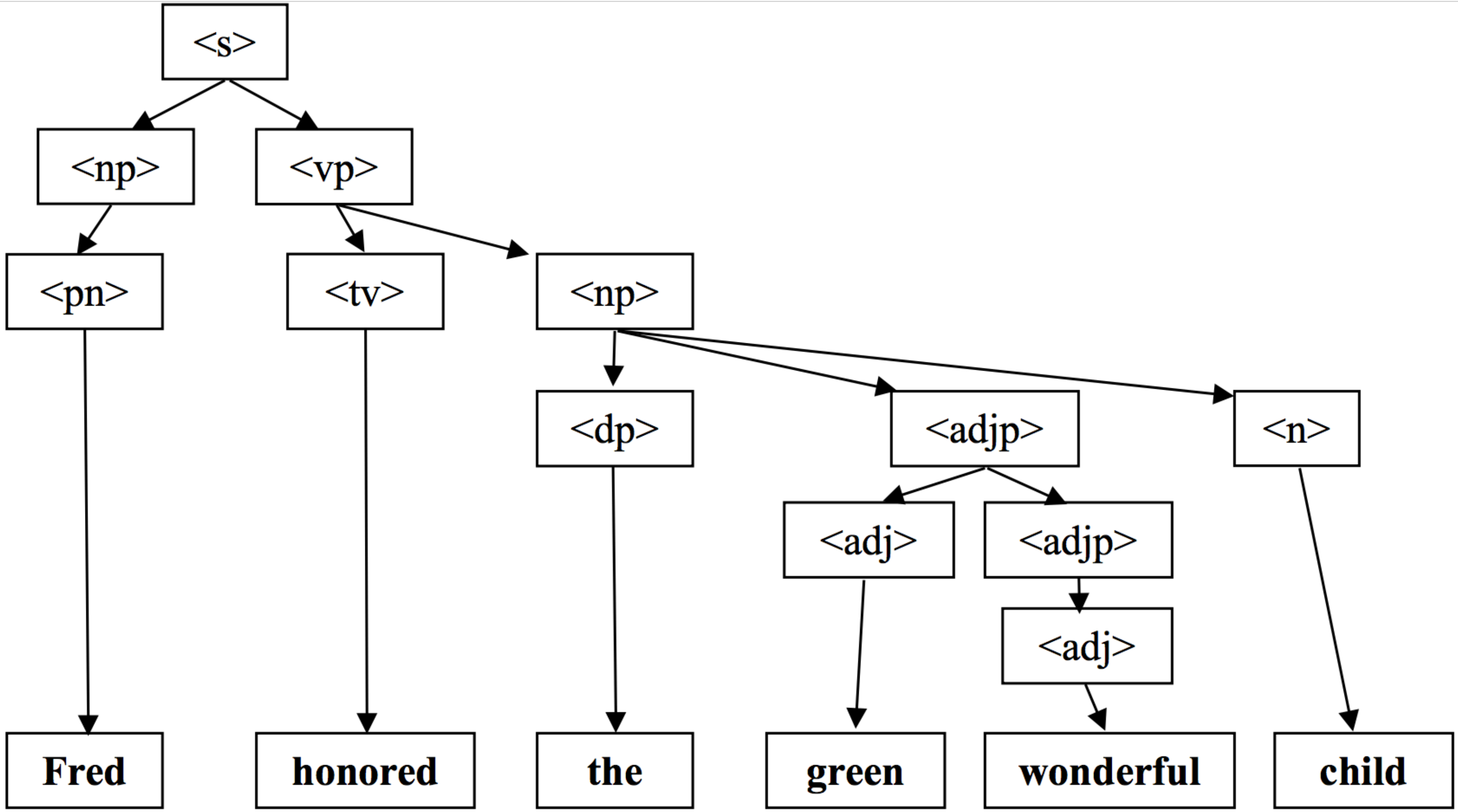
Trees are Everywhere

Syntax Tree



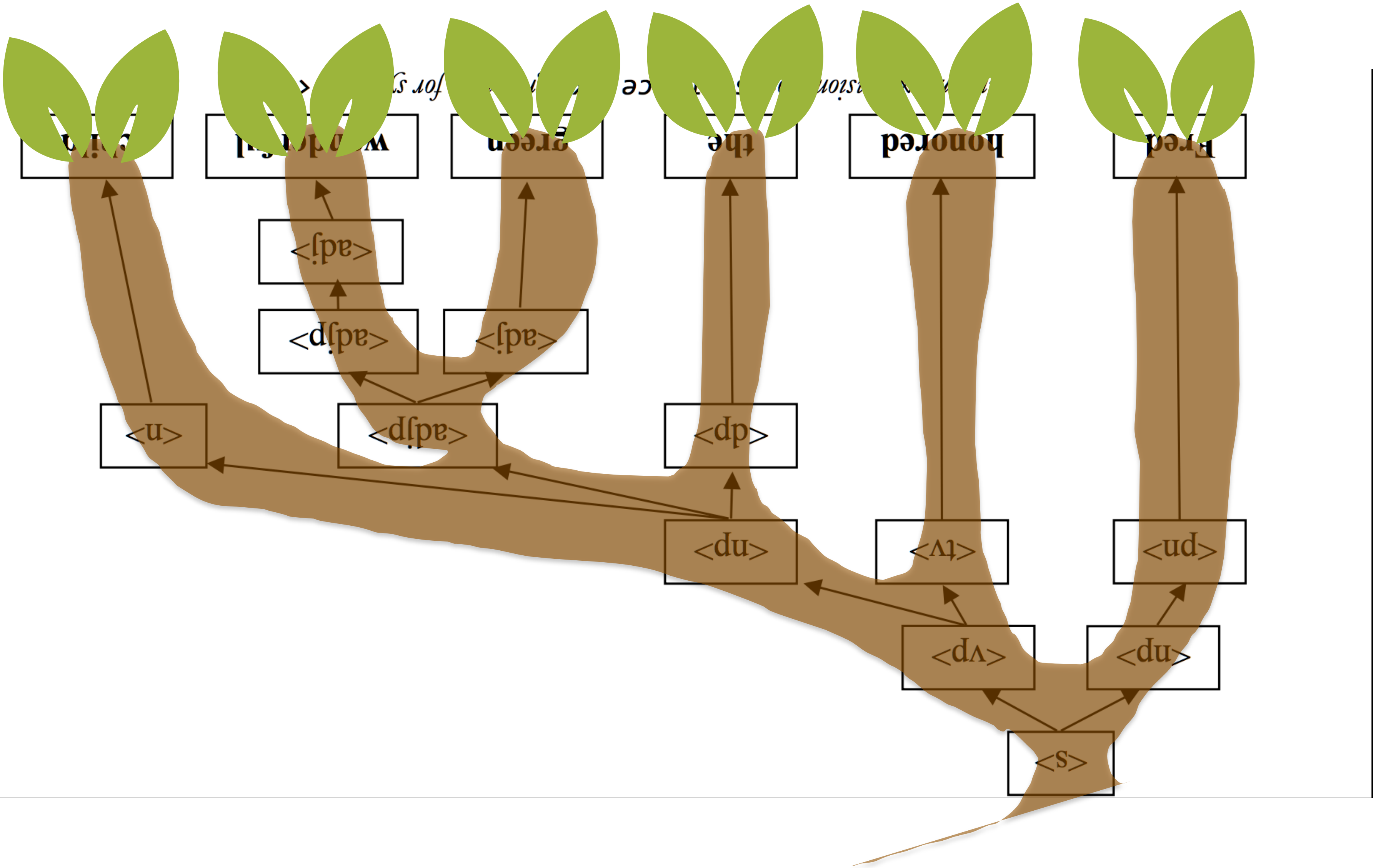
Random expansion from `sentence.txt` *grammar for symbol* `"<s>"`

Syntax Tree



Random expansion from `sentence.txt` *grammar for symbol* `"<s>"`

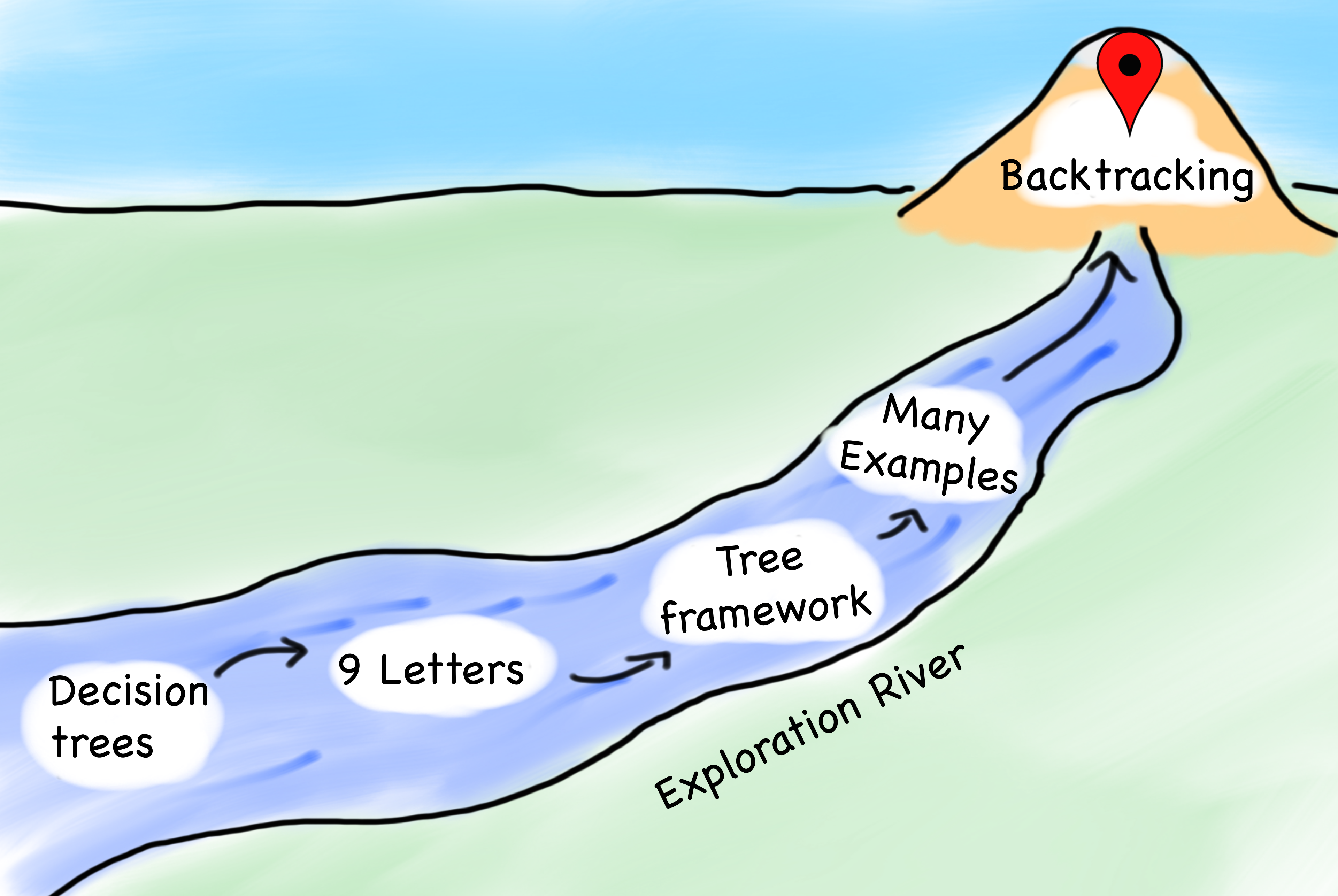
Syntax Tree



Today's Route

Decision trees
9 Letters
Tree framework
Many Examples
Backtracking
Exploration River

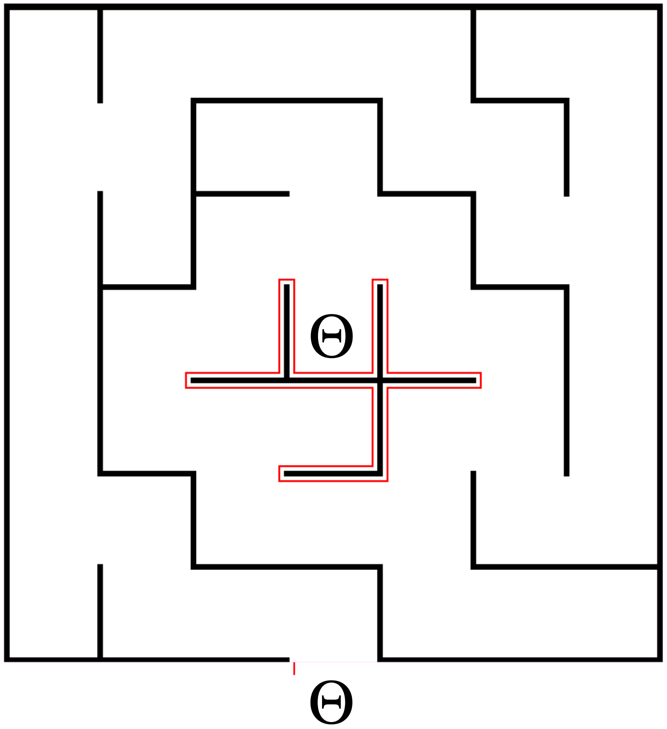
Today's Route



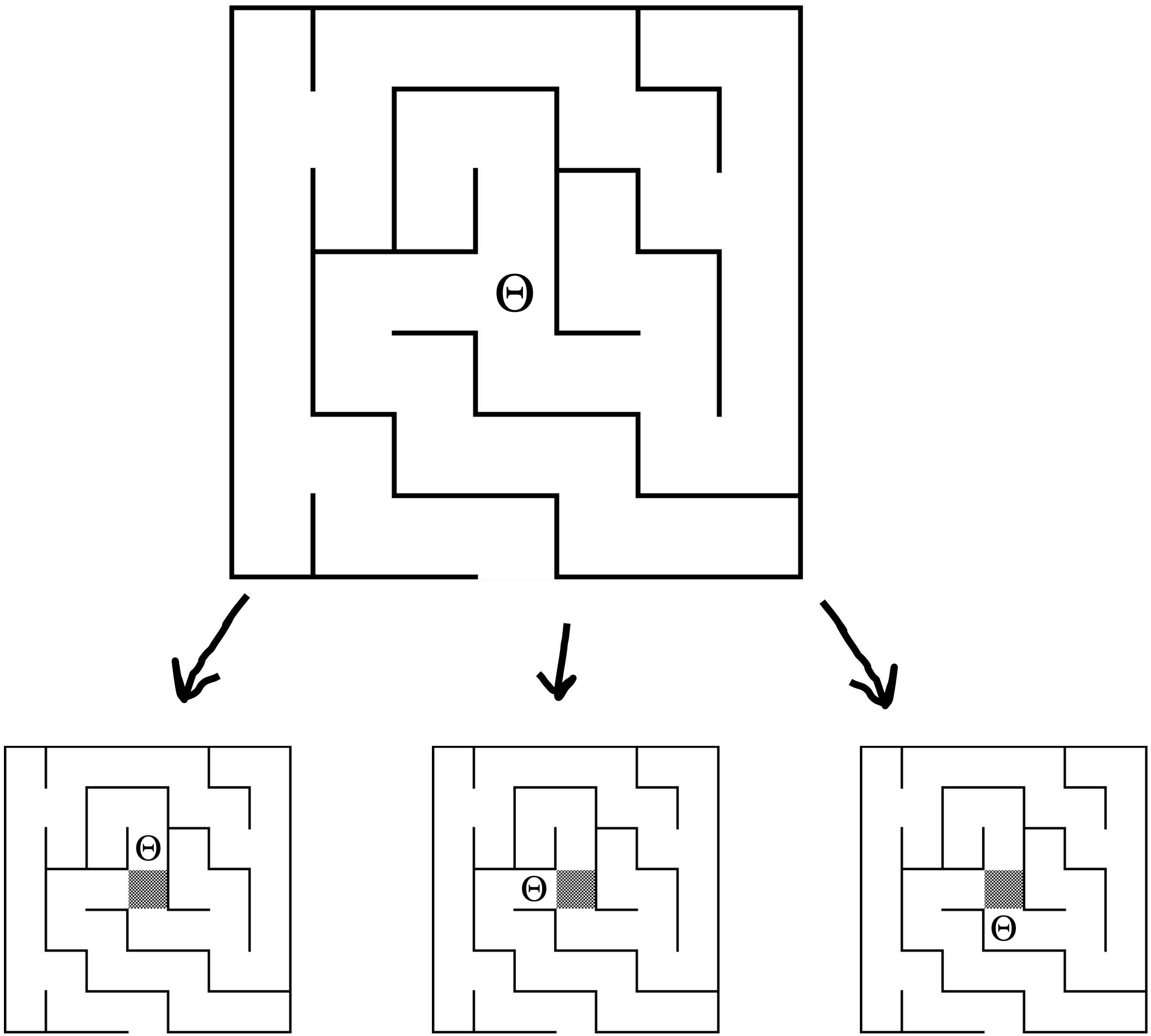
Labyrinth

Right Hand Rule

- The most widely known strategy for solving a maze is called the ***right-hand rule***, in which you put your right hand on the wall and keep it there until you find an exit.
- If Theseus applies the right-hand rule in this maze, the solution path looks like this.
- Unfortunately, the right-hand rule doesn’t work if there are loops in the maze that surround either the starting position or the goal.
- In this maze, the right-hand rule sends Theseus into an infinite loop.



Maze Decision Tree



Enumerated Types in C++

- It is often convenient to define new types in which the possible values are chosen from a small set of possibilities. Such types are called ***enumerated types***.

- In C++, you define an enumerated type like this:

```
enum name { list of element names };
```

- The code for the maze program uses **`enum`** to define a new type consisting of the four compass points, as follows:

```
enum Direction {
   NORTH, EAST, SOUTH, WEST
};
```

- You can then declare a variable of type **`Direction`** and use it along with the constants **`NORTH`**, **`EAST`**, **`SOUTH`**, and **`WEST`**.

Maze Collection

```
/*
 * Class: Maze
 * -----------
 * This class represents a two-dimensional maze contained in a rectangular
 * grid of squares.  The maze is read in from a data file in which the
 * characters '+', '-', and '|' represent corners, horizontal walls, and
 * vertical walls, respectively; spaces represent open passageway squares.
 * The starting position is indicated by the character 'S'.  For example,
 * the following data file defines a simple maze:
 *
 *       +-+-+-+-+-+
 *       |     |
 *       + +-+ + +-+
 *       |S  |     |
 *       +-+-+-+-+-+
 */

class Maze {

public:
```

Maze Collection

```
bool isOutside(Point pt);

bool wallExists(Point pt, Direction dir);

void markSquare(Point pt);

void unmarkSquare(Point pt);

bool isMarked(Point pt);
```

Maze

The SolveMaze Function

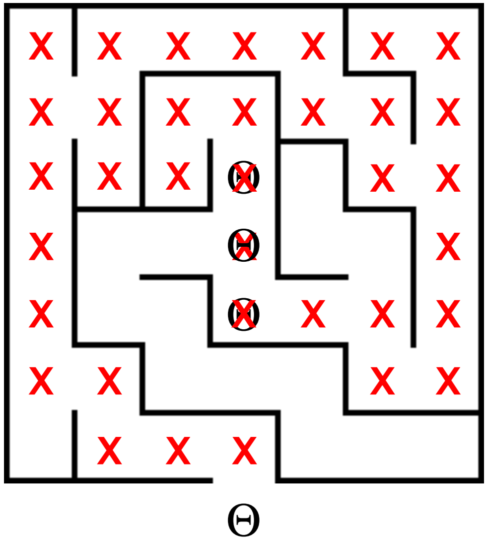
```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses recursion
 * to solve the submazes that result from marking the current square
 * and moving one step along each open passage.
 */

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

Tracing the SolveMaze Function

```
bool solveMaze(Maze & maze, Point start) {
```

```
bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjPt(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

start	dir
(3, 4)	



STOP

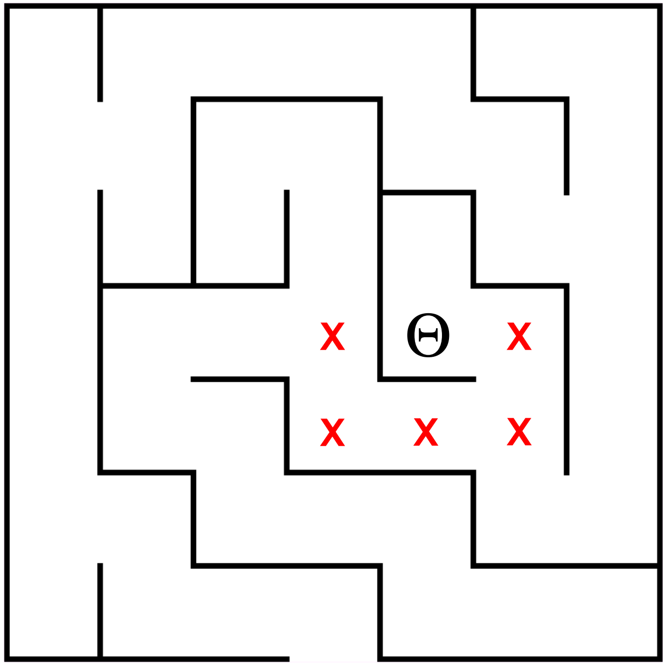
Don't follow the recursion more than one level.
Depend on the recursive leap of faith.

Reflections on Maze

- The `solveMaze` program is a useful example of how to search all paths that stem from a branching series of choices. At each square, the `solveMaze` program calls itself recursively to find a solution from one step further along the path.
- To give yourself a better sense of why recursion is important in this problem, think for a minute or two about what it buys you and why it would be difficult to solve this problem iteratively.
- In particular, how would you answer the following questions:
  - What information does the algorithm need to remember as it proceeds with the solution, particularly about the options it has already tried?
  - In the recursive solution, where is this information kept?
  - How might you keep track of this information otherwise?

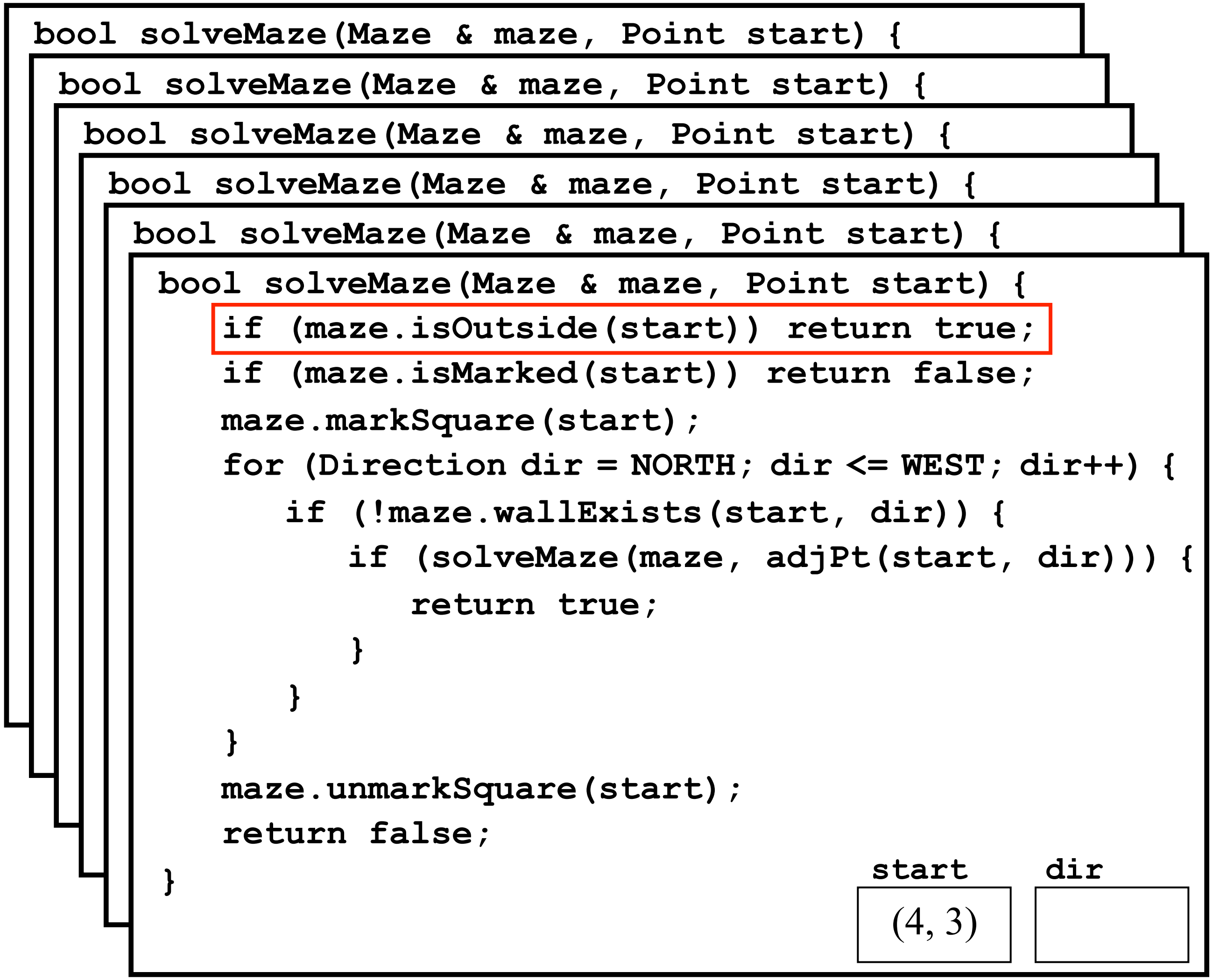
Reflections on Maze

- Suppose that the program has reached the following position:



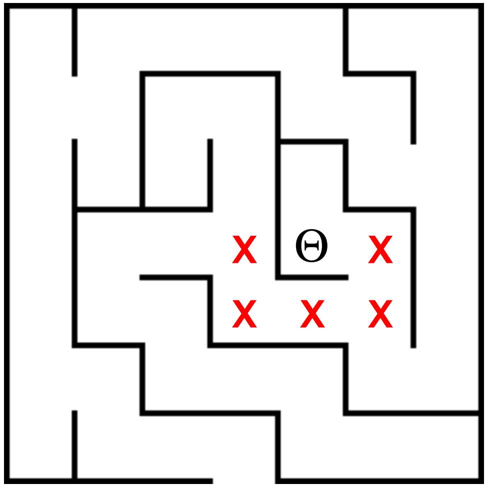
- How does the algorithm keep track of the “big picture” of what paths it still needs to explore?

Each Frame Remembers One Choice

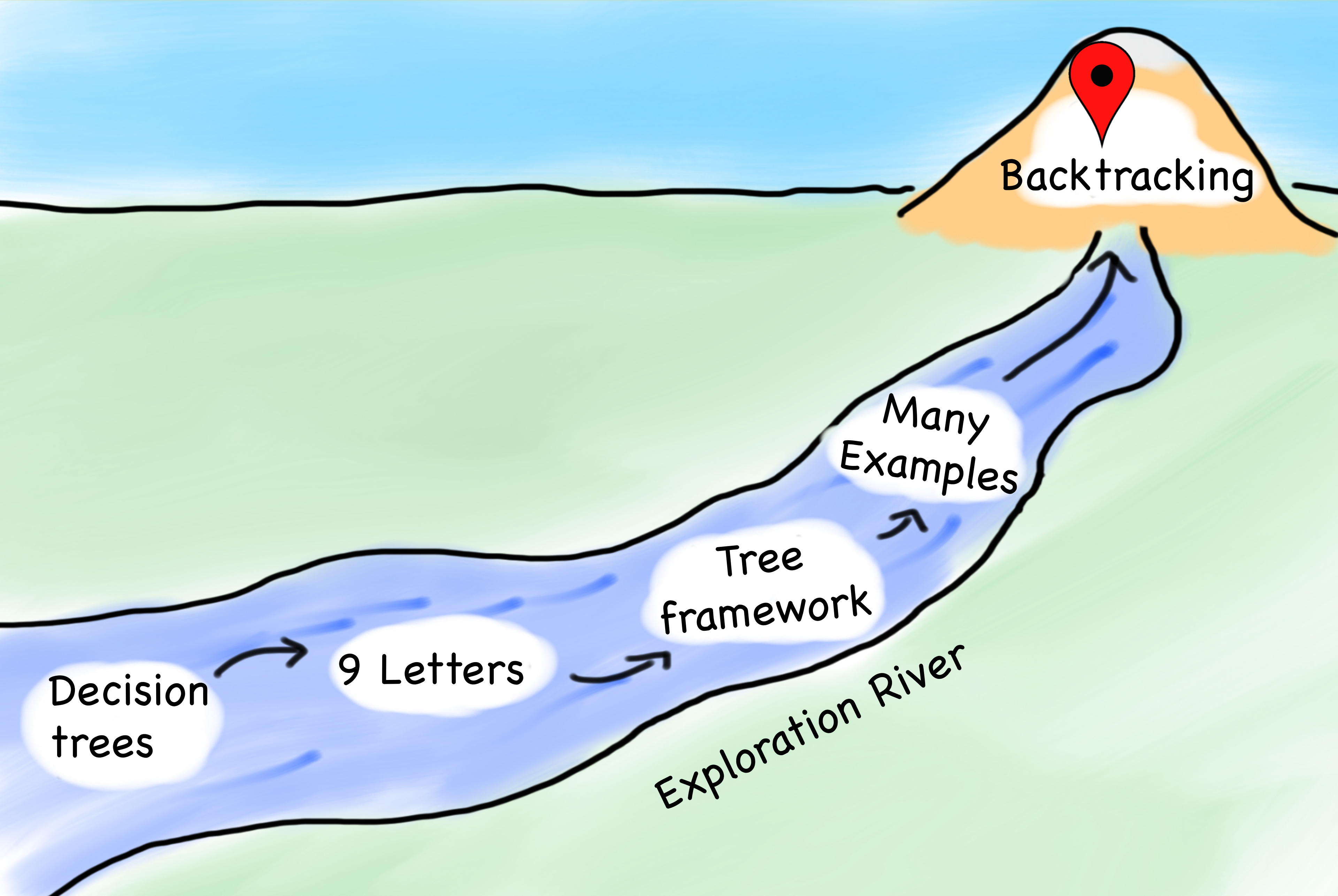


```
bool solveMaze(Maze & maze, Point start) {
 bool solveMaze(Maze & maze, Point start) {
  bool solveMaze(Maze & maze, Point start) {
   bool solveMaze(Maze & maze, Point start) {
    bool solveMaze(Maze & maze, Point start) {
     bool solveMaze(Maze & maze, Point start) {
        if (maze.isOutside(start)) return true;
        if (maze.isMarked(start)) return false;
        maze.markSquare(start);
        for (Direction dir = NORTH; dir <= WEST; dir++) {
           if (!maze.wallExists(start, dir)) {
              if (solveMaze(maze, adjPt(start, dir))) {
                 return true;
              }
           }
        }
        maze.unmarkSquare(start);
        return false;
     }
```

start	dir
(4, 3)	



Today's Route



The End