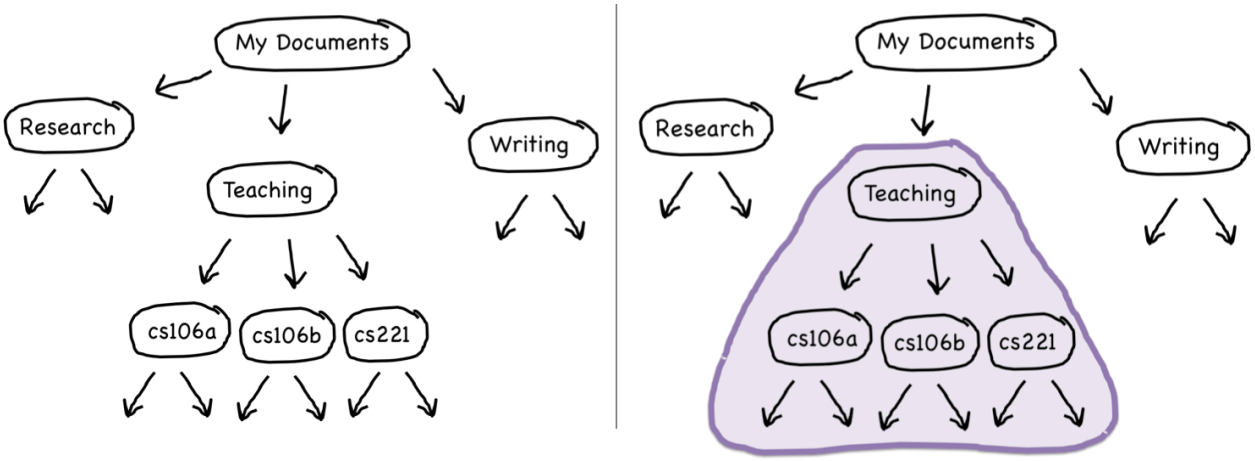
Recursion Exploration

An especially powerful flavor of recursion is when it is used to explore trees. Trees are a recursive structure where each state has a set of branches. The state that results from following a branch is also a tree. **Recursive exploration** allows us to explore an entire tree. It is also called “try everything search” or “recursive depth first search”. Given a start state, you have numerous braches to choose from. After you choose a branch, you encounter new options to choose from. Each of these options will provide new next steps and this continues until you find a solution or run out of options. This sequence of calls is called a **tree**.



In the tree above, which represents a file tree, My Documents is the top state of the tree. If you follow the branch for Teaching, the state you arrive at can also be seen as the top of a tree.

Recursion on Decision Trees

For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. In this case we call the tree on which we are operating a **decision tree**. In recursive exploration, we are looking for any path through the decision tree.

For a given state:

- If any option leads to succeeds, that's great! We're done.
- If none of the options succeed, this particular problem can't be solved from the state.

```
bool search(currentState) {
  if(isSolution(currentState)) {
    return true;
  } else {
    for(option : moves from currentState) {
      nextState = takeOption(curr, option);
      if(search(nextState)){
        return true;
      }
    }
    return false;
  }
}
```

Recursive Backtracking

There is also an important subset of recursive exploration called **backtracking**. Recursive backtracking is needed if you are not making a fresh representation of state with each recursive call and instead are ***modifying*** the state that is passed in by reference. If you make the correct set of choices, you end up at the solution. But if you make a wrong decision, you have to backtrack to a previous decision point and try a different path. As a rule of thumb, recursive backtracking requires three key steps:

- **choose**, where you set up exploring a particular path;
- **explore**, where you recursively explore that path; and
- **un-choose**, where you undo whatever you did in the first step.

Example problems

The Periodic Table Alphabet (problem courtesy of Keith Schwarz)

Have you ever wondered what English words can be spelled using just the symbols from the periodic table? You're about to find out.

The periodic table lists abbreviations used for all of the known elements—hydrogen, lithium, oxygen, carbon, molybdenum, uranium, and so forth. Each element has its own one, two, or three-letter symbol: H for hydrogen, Li for lithium, Mo for molybdenum, Cf for Californium etc. All in all, there are 118 elements, so there are 118 abbreviations.

For this problem, we're pretending that these symbols are the 'letters' of a new alphabet. Your task here is, given a **`Vector<string>`** storing all element symbols and a **`Lexicon`** of all English words, to print out all those English words that can be constructed using just the symbols from the periodic table. You should only print out words of length 11 or more, and you should retain the capitalization scheme of the atomic symbols when printing out the words.Here's a small window of the words that should be printed out:

...

```
IrReSOLuTiON
IrReSOLuTeNEsS
IrReSOLuTeNeSS
IrReSPONSiBILiTiEs
IrReSPONSiBiLiTiEs
IrReSPONSIBILiTiEs
IrReSPONSIBiLiTiEs
...
```

Ir is iridium, Re is rhenium, S is sulfur, O is oxygen, Lu is lutetium, etc.: Trust that every capitalized substring identifies some element from the periodic table. Our solution builds on a working prefix up from the empty string by recursively **exploring** every single symbol in the periodic table as a possible extension. If you construct a string that isn't even a prefix of a word, then you prune that search, recognize your dead end, and back off. But if you notice that the working string is actually a word, then as a side effect you print the string out and dig even further for a longer word beyond what you already have.

```
const int kMinWordLength = 11;

void printAllWords(const Lexicon& english, const Vector<string>& elements,
                   const string& prefix) {
    if (!english.containsPrefix(prefix)) return;
    if (english.contains(prefix) && prefix.length() >= kMinWordLength) {
        cout   << prefix << endl;
    }
    foreach (string element in elements) {
        printAllWords(english, elements, prefix + element);
    }
}

void printAllWords(const Lexicon& english, const Vector<string>& elements) {
    printAllWords(english, elements, "");
}
```

Periodic Table as Alphabet: Take II

In a previous example, we presented code to list all of those English words that can be spelled out using the symbols of the periodic table and nothing else. Here's a related problem that asks specifically whether or not the provided word can be spelled out using just the periodic table symbols. We'll assume that all of the symbols are 1, 2, or 3 letters long, that the symbols come via a **Set<string>**, and that all of the **Lexicon** methods we're familiar with are case-insensitive.

The idea is to see if the first 1, 2, or 3 letters match some symbol, and if so, to recur on all of the remaining length – 1, length – 2, or length – 3 letters to see if they can also be subdivided into periodic table elements. Here's the solution:

```
static string elementize(string str) {
  if (!str.empty()) {
    str = toLowerCase(str); // -> "he"
    str[0] = toupper(str[0]); // -> "He"
  }
  return str;
}

static bool canSpell(string word, Set<string>& symbols) {
  if (word.empty()) return true;
  int length = word.size();
  for (int i = 1; i <= min(3, length); i++) {
    if (symbols.contains(elementize(word.substr(0, i))) &&
        canSpell(word.substr(i), symbols)) {
      return true;
    }
  }
  return false;
}
```

If we want visual proof the word can be spelled, then we can accumulate the relevant symbols in a **Stack** as the successful search unwinds, and then print the serialization of the **Stack** from the call site.

```
static bool canSpell(string word, Set<string>& symbols,
                     Stack<string>& footprint) {
  if (word.empty()) return true;
  int length = word.size();
  for (int i = 1; i <= min(3, length); i++) {
    string symbol = elementize(word.substr(0, i));
    if (symbols.contains(symbol) &&
        canSpell(word.substr(i), symbols, footprint)) {
      footprint.push(symbol);
      return true;
    }
  }
  return false;
}
```

Here's a **main** function that exercises the second version of **canSpell**, and illustrates how the **footprint** can be drained and printed to standard out, knowing that the last symbol used to spell the word is buried at the bottom, and the first symbol used is at the top:

```
int main() {
  Set<string> symbols;
  addPeriodicTableElements(symbols); // assume this just works as implied
  while (true) {
    string word = getLine("Enter a word: ");
    if (word.empty()) break;
    Stack<string> footprint;
    if (canSpell(word, symbols, footprint)) {
      cout << "That can be spelled as \"";
      while (!footprint.isEmpty()) {
        cout << footprint.pop();
      }
      cout << "\"." << endl;
    } else {
      cout << "That's just not possible." << endl;
    }
  }
  return 0;
}
```

Finally, here's a test run of the above program to illustrate the output is as excepted (or at least believable).

```
Enter a word: hen
That can be spelled as
"HeN". Enter a word:
foolishness
That can be spelled as "FOOLiSHNEsS". Enter a word: partial
That can be spelled as
"PArTiAl". Enter a word:
hooligan
That can be spelled as "HOOLiGaN". Enter a word: hooliganism
That can be spelled as "HOOLiGaNISm". Enter a word: indefatigable
That's just not possible.
Enter a word: antidisestablishmentarianism
That's just not possible. Enter a word:
```