Trial Version Wondershare PDFelement

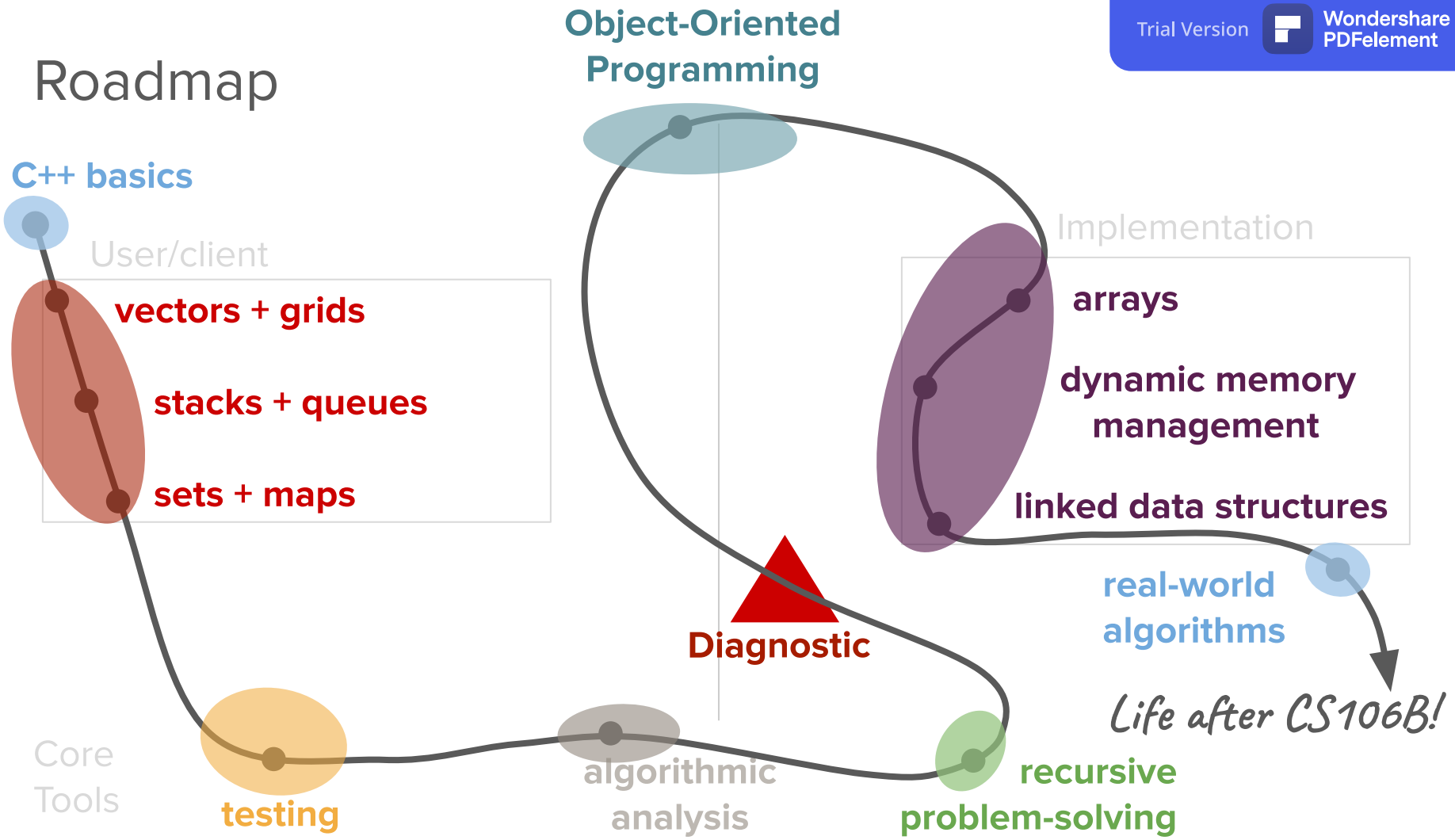
Recursive Backtracking and Enumeration

What is an example of a game that would be easy to play if you had the ability to quickly think of all possible moves/plays?

(put your answers the chat)

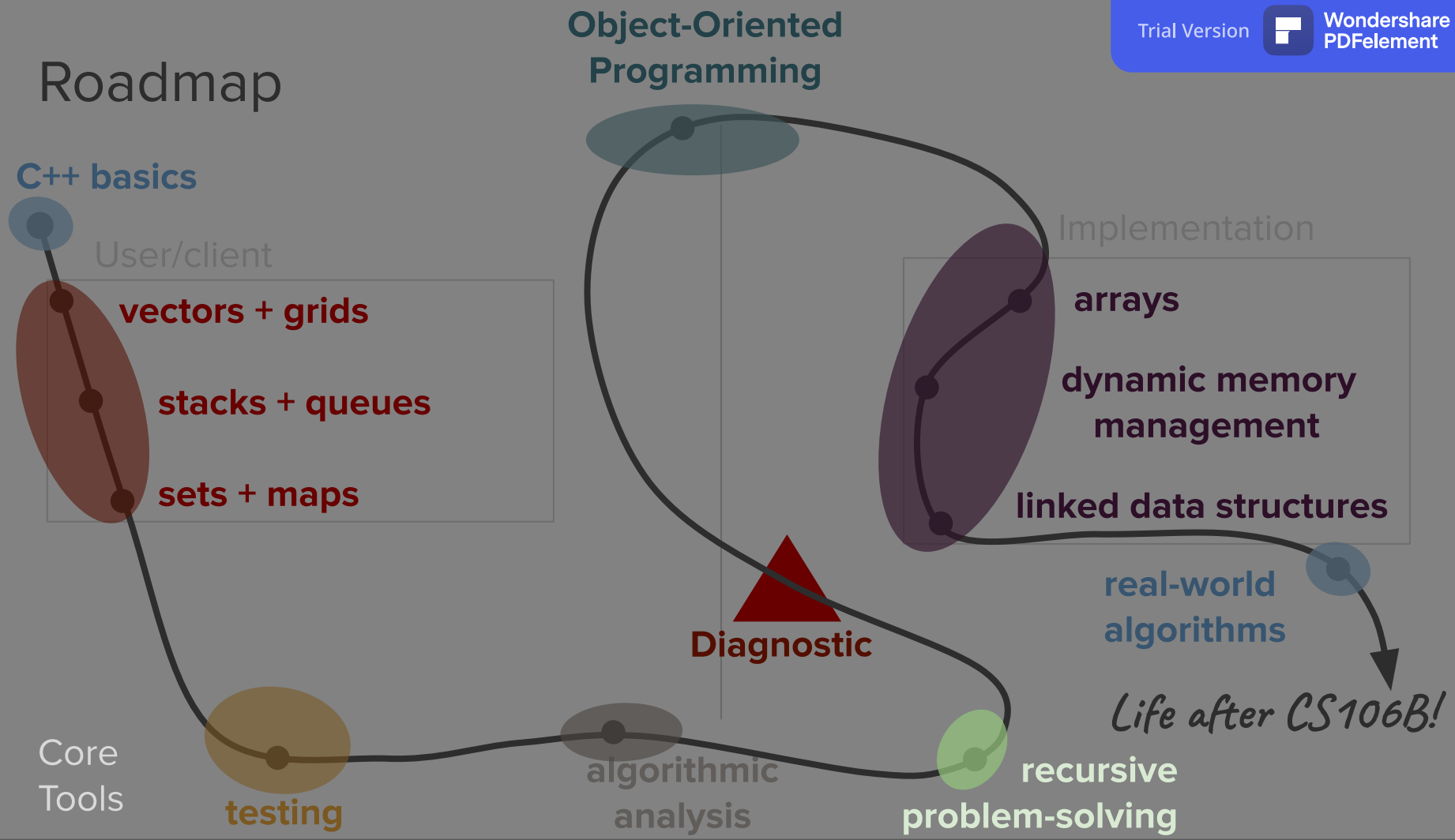
Trial Version Wondershare PDFelement

Roadmap



Trial Version Wondershare PDFelement

Roadmap



Trial Version Wondershare PDFelement

Today's question

How can we leverage backtracking recursion to solve interesting problems?

Trial Version Wondershare PDFelement

Today's topics

1. Review
2. Word Scramble
3. Shrinkable Words
4. Generating Subsets

Trial Version Wondershare PDFelement

Review

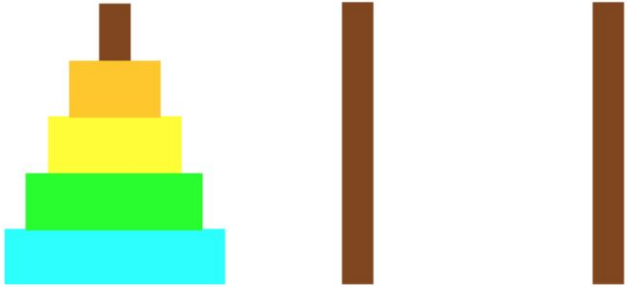
(advanced recursion patterns)

Trial Version Wondershare PDFelement

Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

Elegance (Towers of Hanoi)



source auxiliary destination

```
void findSolution(int n, char source, char dest,
char aux) {
    if (n == 1) {
        moveSingleDisk(source, dest);
    } else {
        findSolution(n - 1, source, aux, dest);
        moveSingleDisk(source, dest);
        findSolution(n - 1, aux, dest, source);
    }
}
```

```
void findSolutionIterative(int n, char source, ch
    int numMoves = pow(2, n) - 1; // total number

    // if number of disks is even, swap dest and a
    if (n % 2 == 0) {
        char temp = dest;
        dest = aux;
        aux = temp;
    }

    Stack<int> srcStack;
    for (int i = n; i > 0; i--) {
        srcStack.push(i);
    }
    cout << srcStack << endl;
    Stack<int> destStack;
    Stack<int> auxStack;

    // Determine next move based on how many moves have been made so far
    for (int i = 1; i <= numMoves; i++) {
        switch (i % 3) {
            case 1:
                if (srcStack.isEmpty() ||
                        (!destStack.isEmpty() && srcStack.peek() > destStack.peek())) {
                    srcStack.push(destStack.pop());
                    moveSingleDisk(dest, source);
                } else {
                    destStack.push(srcStack.pop());
                    moveSingleDisk(source, dest);
                }
                break;
            case 2:
                if (srcStack.isEmpty() ||
                        (!auxStack.isEmpty() && srcStack.peek() > auxStack.peek())) {
                    srcStack.push(auxStack.pop());
                    moveSingleDisk(aux, source);
                } else {
                    auxStack.push(srcStack.pop());
                    moveSingleDisk(source, aux);
                }
                break;
            case 0:
                if (destStack.isEmpty() ||
                        (!auxStack.isEmpty() && destStack.peek() > auxStack.peek())) {
                    destStack.push(auxStack.pop());
                    moveSingleDisk(aux, dest);
                } else {
                    auxStack.push(destStack.pop());
                    moveSingleDisk(dest, aux);
                }
                break;
        }
    }
}
```

Trial Version Wondershare PDFelement

Trial Version Wondershare PDFelement

Efficiency (Binary Search)

- Leverage the structure in sorted data to **eliminate half of the search space every time** when searching for an element
  - Only do a direct comparison with the middle element in the list
  - Recursively search the left half if the element is less than the middle
  - Recursively search the right half if the element is greater than the middle

- Binary search has logarithmic Big-O: `O(log N)`
  - Enables efficient performance of sets and maps

Binary Search

Input Size	Runtime (s)
1000000	0.064
2000000	0.072
4000000	0.082
8000000	0.097
16000000	0.111
32000000	0.121
64000000	0.14

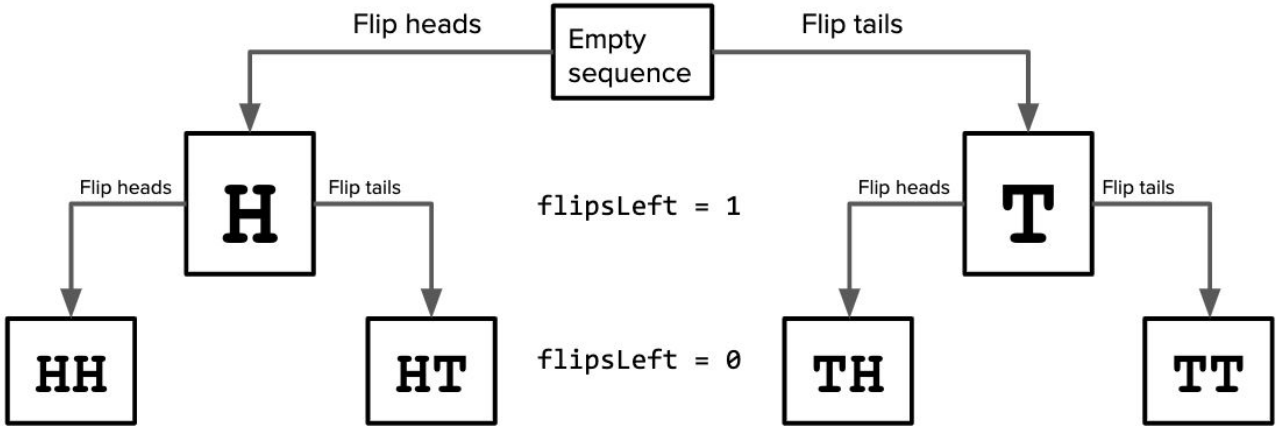
Linear Search

Input Size	Runtime (s)
10000	0.096
20000	0.189
40000	0.368
8000000	0.767
160000	1.387
320000	2.746
640000	6.154

Trial Version Wondershare PDFelement

Dynamic (Coin Sequences + Decision Trees)

- The **height** of the tree corresponds to the **number of decisions** we have to make. The **width** at each decision point corresponds to the **number of options at each decision**.
- To exhaustively explore the entire search space, we must **try every possible option for every possible decision**.



Trial Version Wondershare PDFelement

Two types of recursion

Basic recursion

- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to recursive function produces final solution

Backtracking recursion

- Build up many possible solutions through multiple recursive calls at each step
- Seed the initial recursive call with an “empty” solution
- At each base case, you have a potential solution

Trial Version Wondershare PDFelement

How can we leverage backtracking recursion to solve interesting problems?

Trial Version Wondershare PDFelement

Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - **We can generate all possible solutions to a problem or count the total number of possible solutions to a problem**
  - **We can find one specific solution to a problem or prove that one exists**
  - We can find the best possible solution to a given problem
- There are many, many examples of specific problems that we can solve, including
  - **Generating permutations**
  - **Generating subsets**
  - Generating combinations
  - And many, many more

Trial Version Wondershare PDFelement

Word Scramble

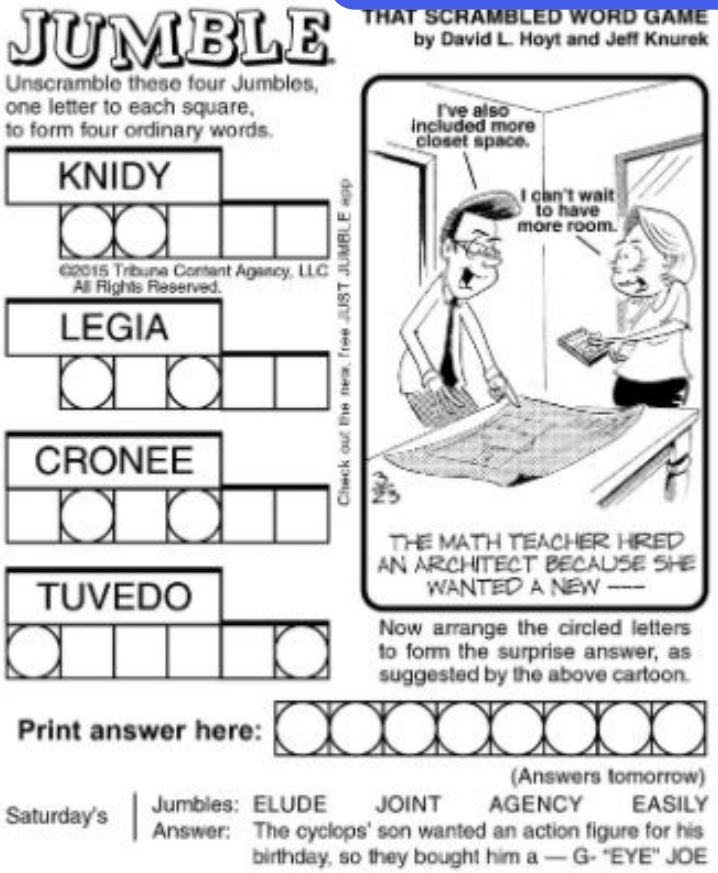
Trial Version Wondershare PDFelement

Jumble

- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.

解读

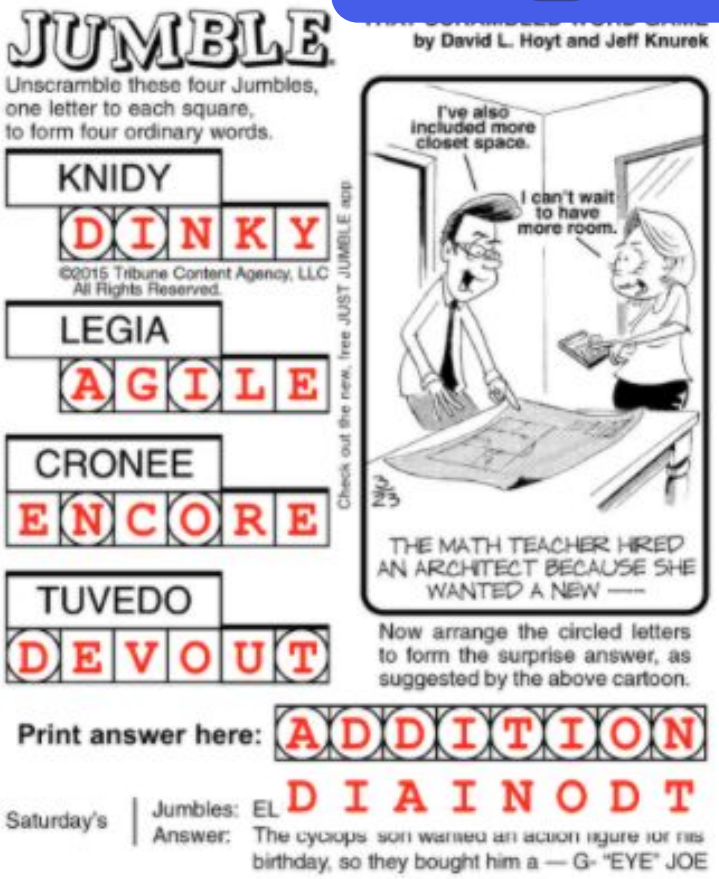
- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.



Trial Version Wondershare PDFelement

Jumble

- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.
- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.

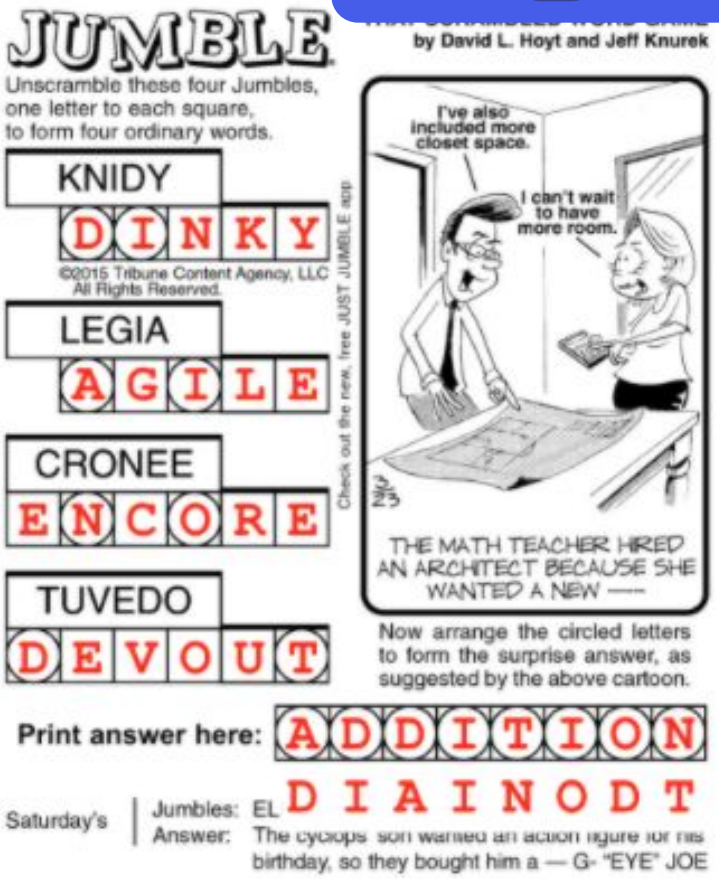


Trial Version Wondershare PDFelement

Jumble

- For some people solving puzzles like this comes pretty easily, but this is actually a pretty challenging problem!
  - For a 6-letter word, there are 6! = 720 possible arrangements of the letters

- Can we write a program to print out all the combinations to help us solve this puzzle?



Trial Version Wondershare PDFelement

Permutations

Trial Version Wondershare PDFelement

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.
- For example, permutations of the words in the motto "E Pluribus Unum" would be:
  - E Pluribus Unum
  - E Unum Pluribus
  - Pluribus E Unum
  - Pluribus Unum E
  - Unum E Pluribus
  - Unum Pluribus E

Trial Version Wondershare PDFelement

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.
- We can think of permutations as an extension of the coin flip sequences we generated yesterday.
  - Rather than having 2 fixed options (heads and tails), the components of our original sequence define the options we can use to build our new sequence.

原始序列的组成部分定义了我们可以用来构建新序列的选项

Trial Version Wondershare PDFelement

Discuss in breakouts:

What are the possible permutations of the string "saki"? What potential recursive insights about generating permutations can you glean from this example? [Time-permitting] Can you come up with a base case and recursive case for generating permutations?

Trial Version Wondershare PDFelement

Common question from lecture yesterday

- Can you solve all backtracking recursion problems with equivalent iterative solutions?
- Answer:

```
void permute4(string s) {
  for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4 ; j++) {
          if (j == i) {
              continue; // ignore
          }
          for (int k = 0; k < 4; k++) {
              if (k == j || k == i) {
                  continue; // ignore
              }
              for (int w = 0; w < 4; w++) {
                  if (w == k || w == j || w == i) {
                      continue; // ignore
                  }
                  cout << s[i] << s[j] << s[k] << s[w] << endl;
              }
          }
      }
  }
}
```

Trial Version Wondershare PDFelement

Common question from lecture yesterday

- Can you solve a valent iterative solutions?
- Answer:

```
void permute5(string s) {
  for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5 ; j++) {
          if (j == i) {
              continue; // ignore
          }
          for (int k = 0; k < 5; k++) {
              if (k == j || k == i) {
                  continue; // ignore
              }
              for (int w = 0; w < 5; w++) {
                  if (w == k || w == j || w == i) {
                      continue; // ignore
                  }
                  for (int x = 0; x < 5; x++) {
                      if (x == k || x == j || x == i || x == w) {
                          continue;
                      }
                      cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
                  }
              }
          }
      }
  }
}
```

Trial Version Wondershare PDFelement

Common

- Can you sc … t iterative solutions?
- Answer:

```
void permute6(string s) {
  for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5 ; j++) {
          if (j == i) {
              continue; // ignore
          }
          for (int k = 0; k < 5; k++) {
              if (k == j || k == i) {
                  continue; // ignore
              }
              for (int w = 0; w < 5; w++) {
                  if (w == k || w == j || w == i) {
                      continue; // ignore
                  }
                  for (int x = 0; x < 5; x++) {
                      if (x == k || x == j || x == i || x == w) {
                          continue;
                      }
                      for (int y = 0; y < 6; y++) {
                          if (y == k || y == j || y == i || y == w || y == x) {
                              continue;
                          }
                          cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
                      }
                  }
              }
          }
      }
  }
}
```

Trial Version Wondershare PDFelement

Common

- Can you sc ... t iterative solutions?
- Answer:

```
        }
      }
    }
  }
}
```

Trial Version Wondershare PDFelement

Permutations Intuition

What are all the permutations of the string "saki"?

- **"saki"**
- **"saik"**
- **"skai"**
- **"skia"**
- **"sika"**
- **"siak"**
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "lask"
- "iska"
- "isak"

A quarter of the permutations start with "s", followed by all the permutations of "aki"

Trial Version Wondershare PDFelement

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- **"aski"**
- **"asik"**
- **"aksi"**
- **"akis"**
- **"aisk"**
- **"aiks"**
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "lask"
- "iska"
- "isak"

A quarter of the permutations start with "a", followed by all the permutations of "ski"

Trial Version Wondershare PDFelement

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- **"ksai"**
- **"ksia"**
- **"kasi"**
- **"kais"**
- **"kias"**
- **"kisa"**
- "ikas"
- "iksa"
- "iaks"
- "lask"
- "iska"
- "isak"

A quarter of the permutations start with "k", followed by all the permutations of "sai"

Trial Version Wondershare PDFelement

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- **"ikas"**
- **"iksa"**
- **"iaks"**
- **"iask"**
- **"iska"**
- **"isak"**

A quarter of the permutations start with "i", followed by all the permutations of "sak"

Trial Version Wondershare PDFelement

Permutations Intuition

What are all the permutations of the string "saki"?

- "saki"
- "saik"
- "skai"
- "skia"
- "sika"
- "siak"
- "aski"
- "asik"
- "aksi"
- "akis"
- "aisk"
- "aiks"
- "ksai"
- "ksia"
- "kasi"
- "kais"
- "kias"
- "kisa"
- "ikas"
- "iksa"
- "iaks"
- "iask"
- "iska"
- "isak"

Can we formalize this intuition in a decision tree?

Trial Version Wondershare PDFelement

What defines our permutations decision tree?

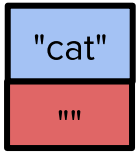
- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

- **Options** at each decision (branches from each node):
  - One option for every remaining element that hasn't been selected yet
  - **Note: The number of options will be different at each level of the tree!**

- Information we need to store along the way:
  - The permutation you've built so far
  - The remaining elements in the original sequence

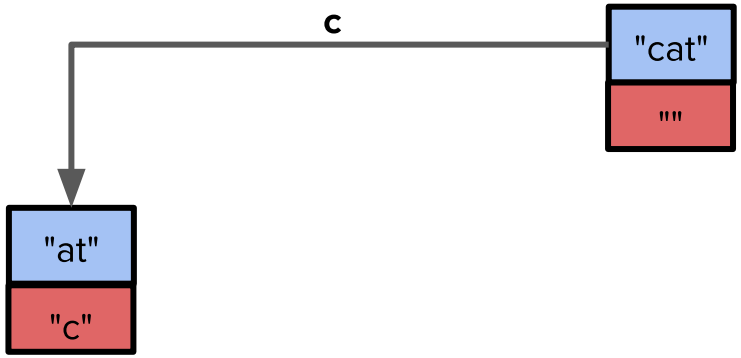
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



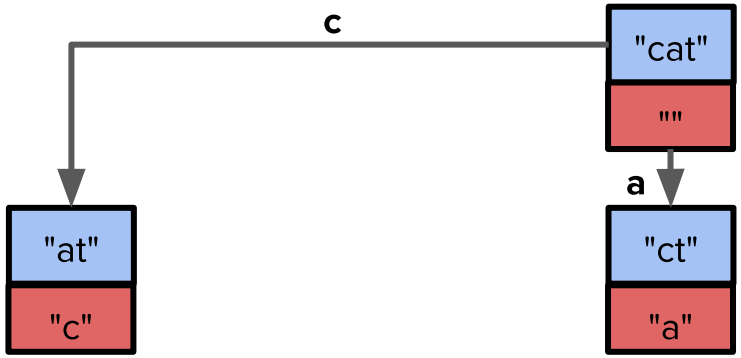
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



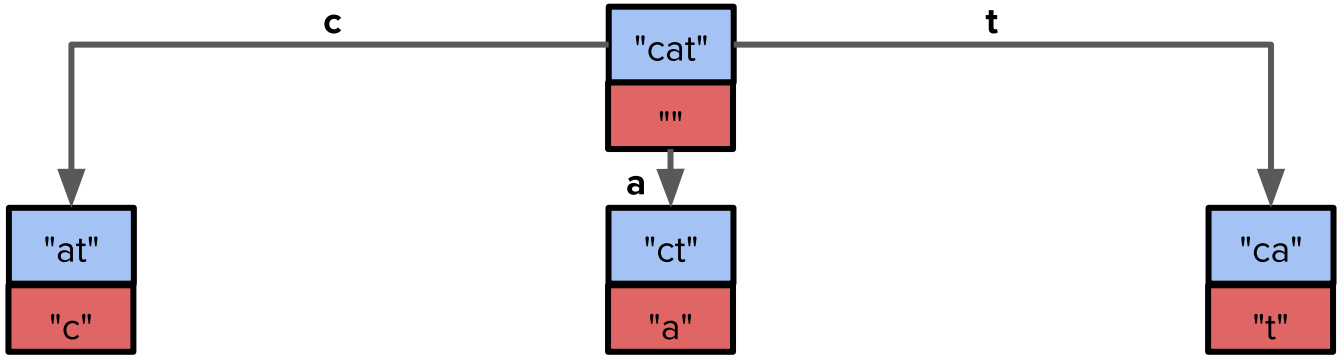
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



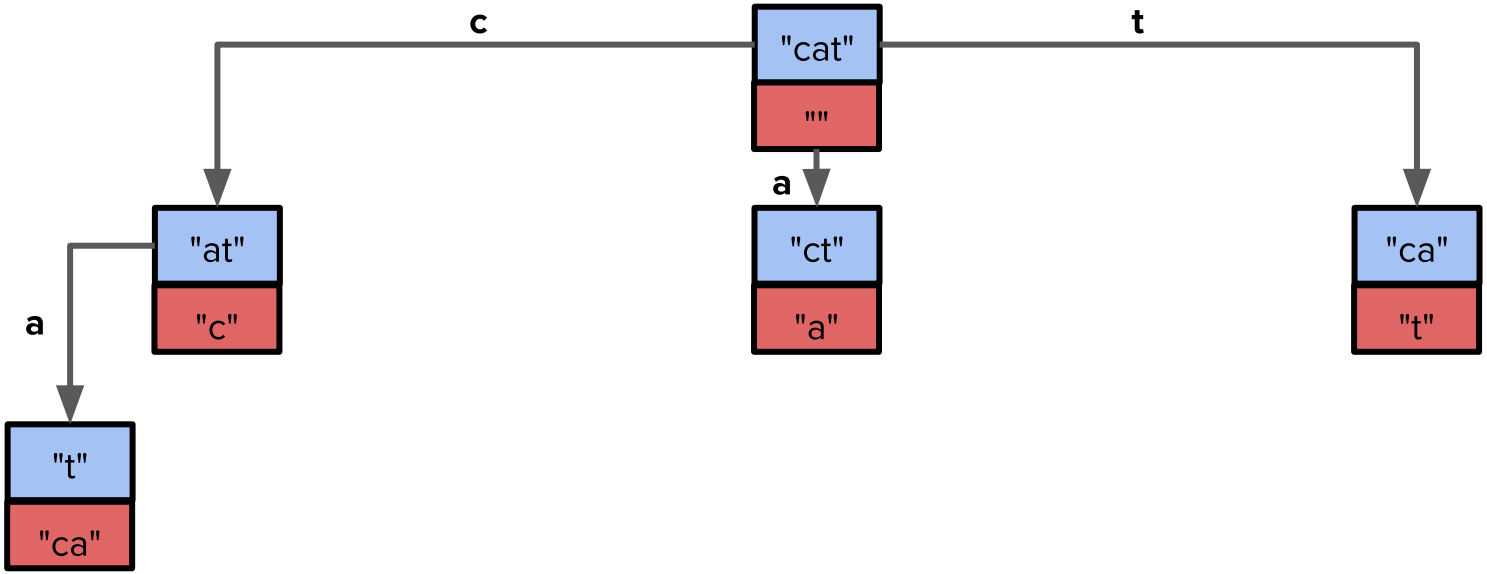
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



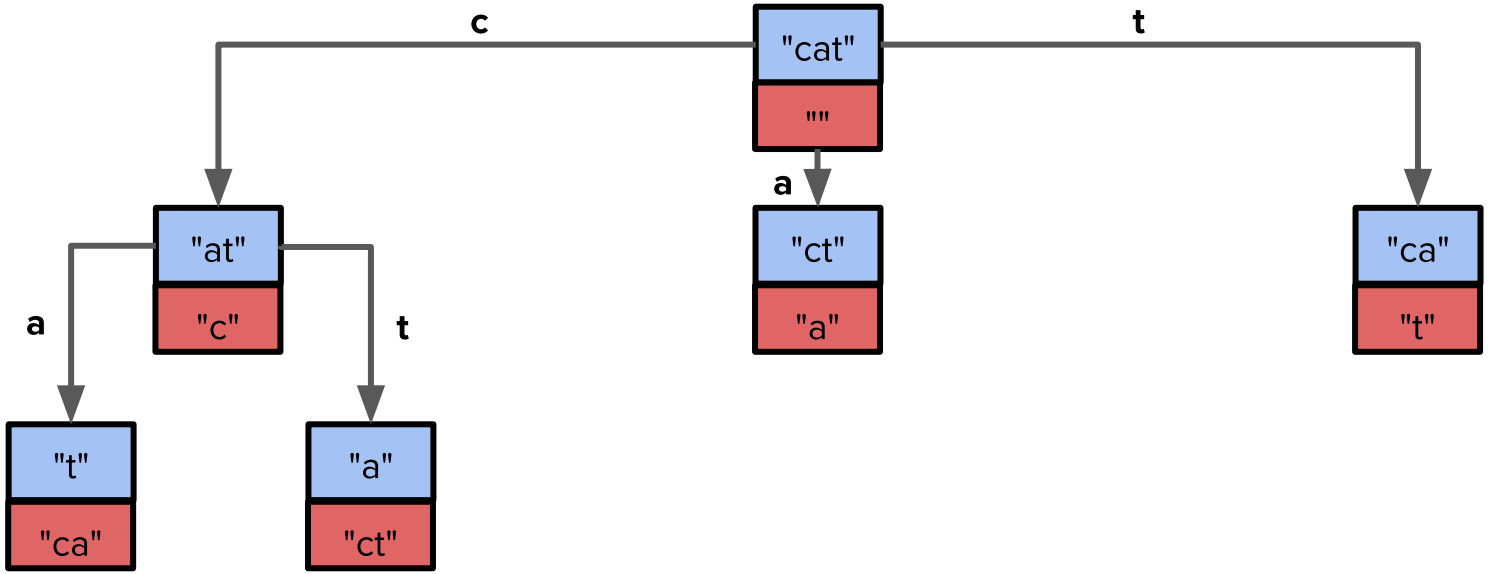
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



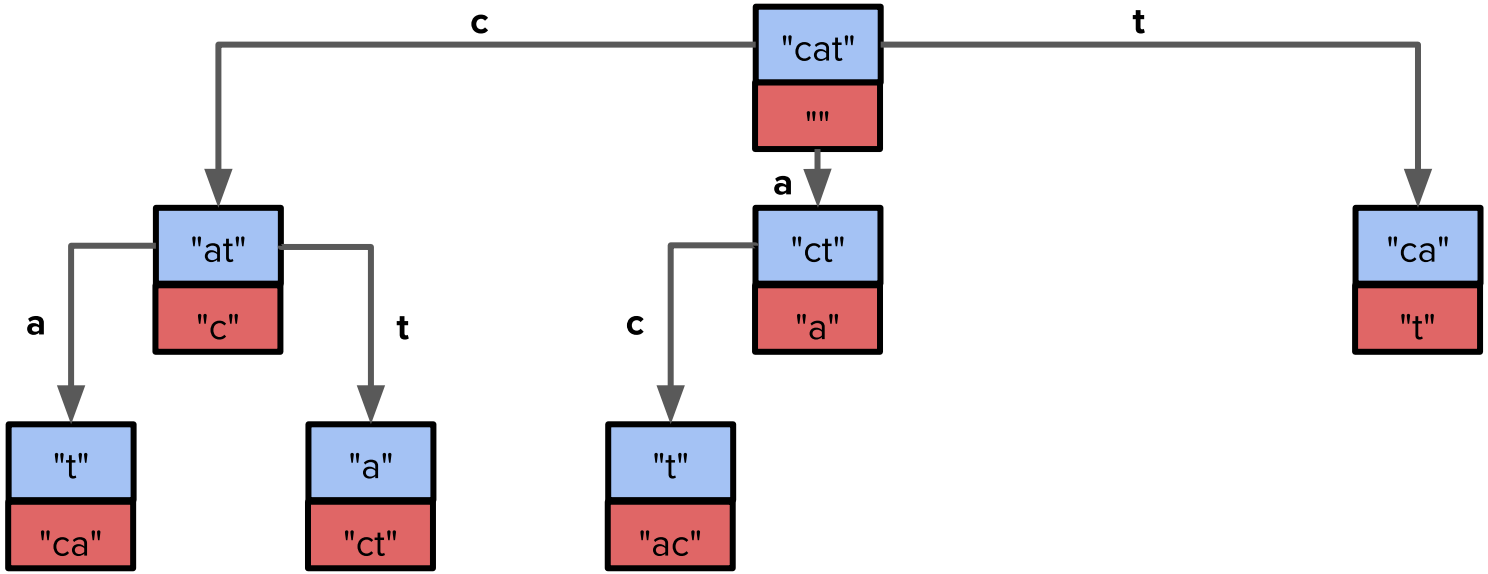
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



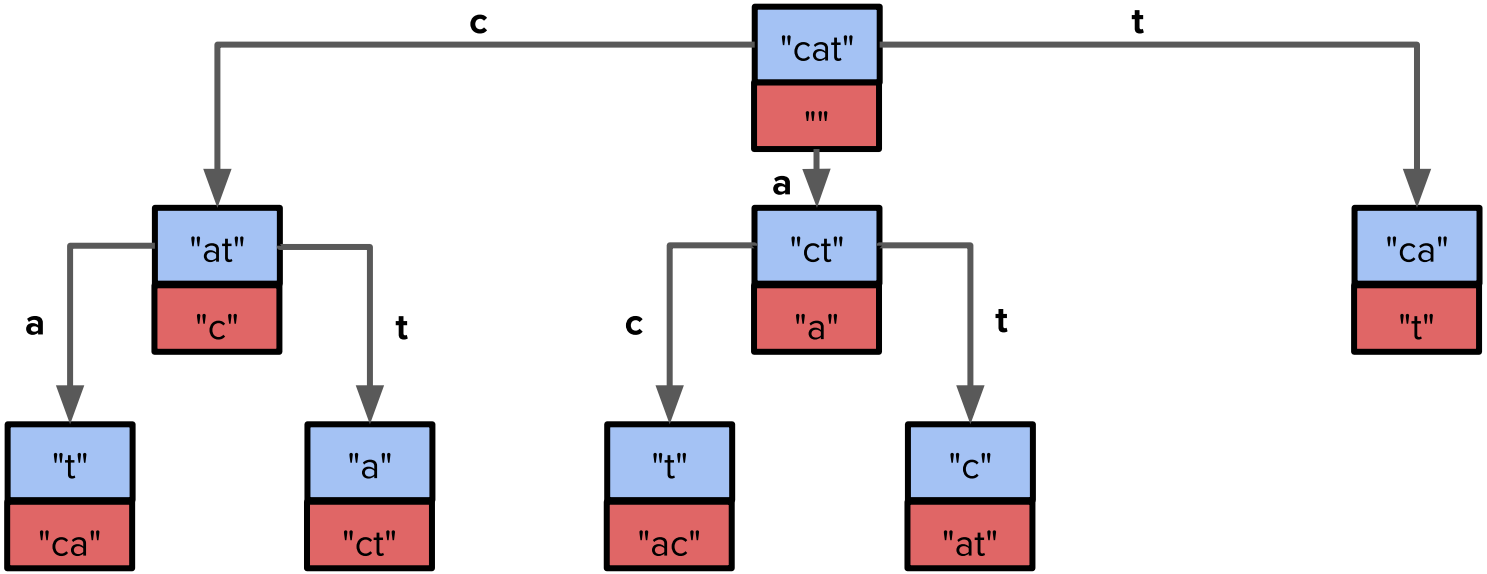
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



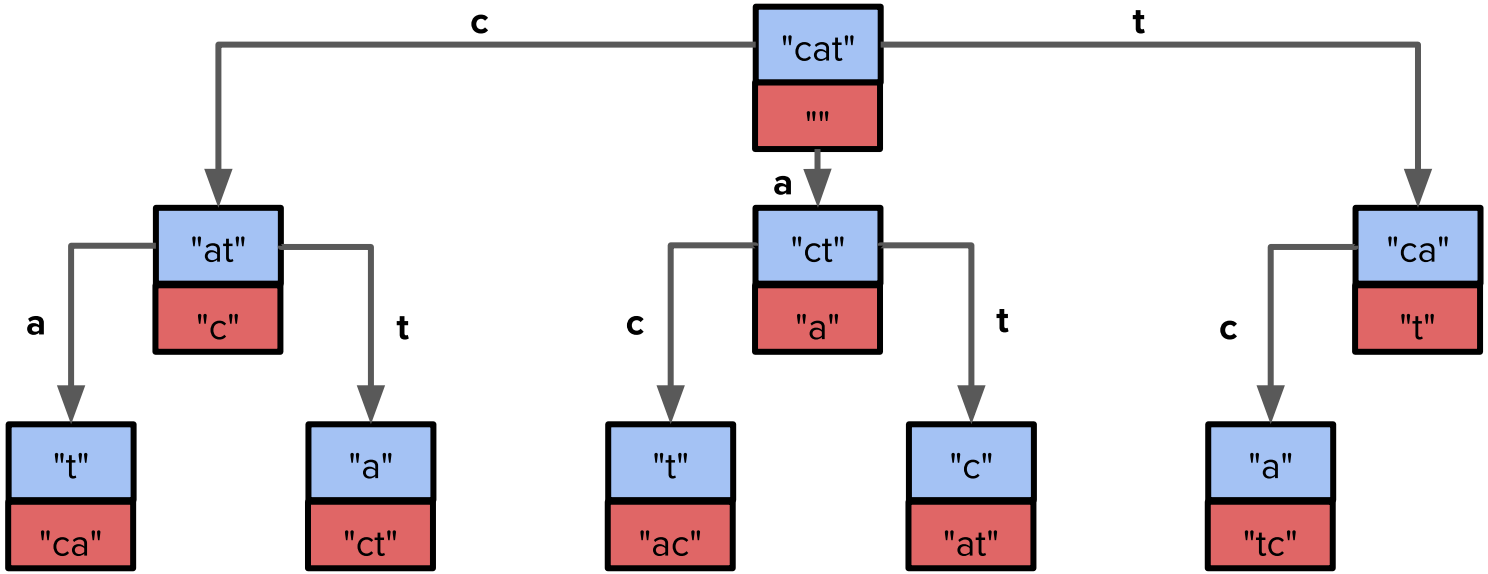
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



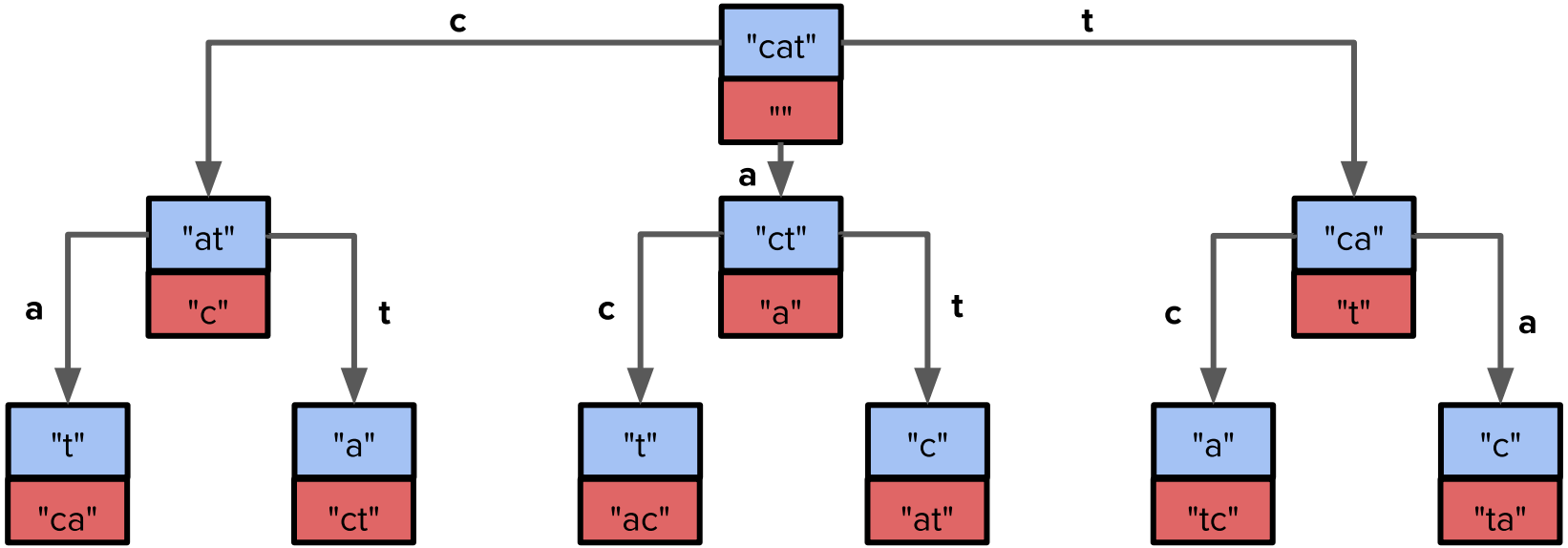
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



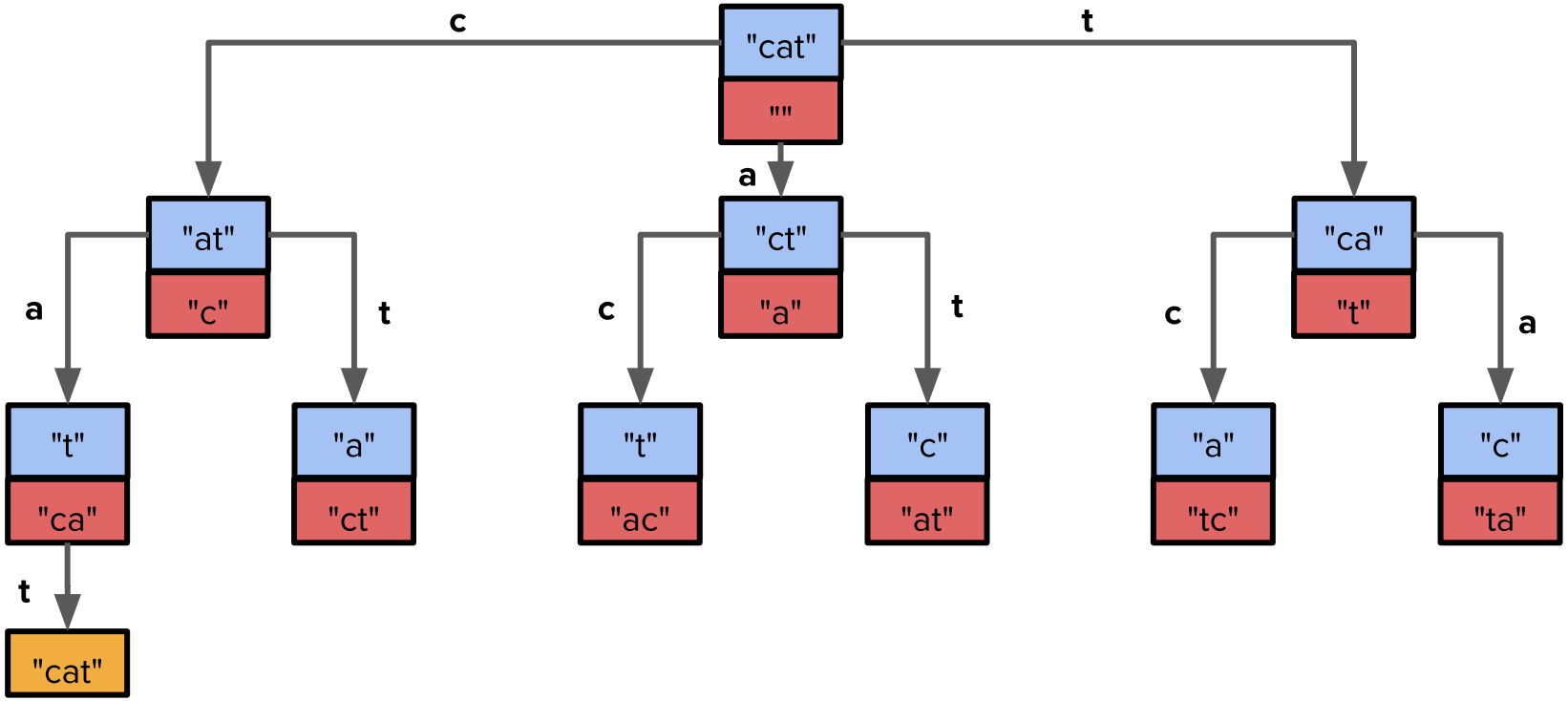
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



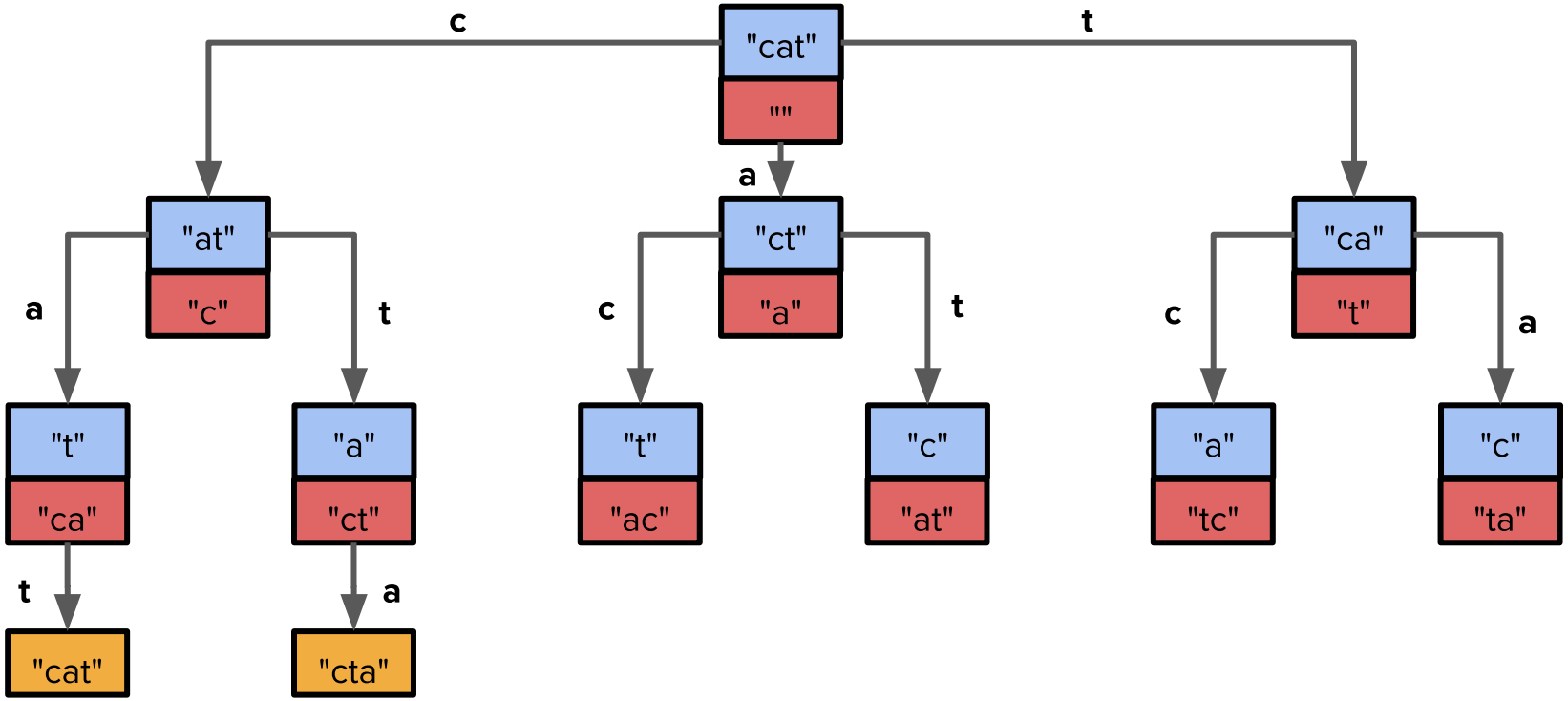
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



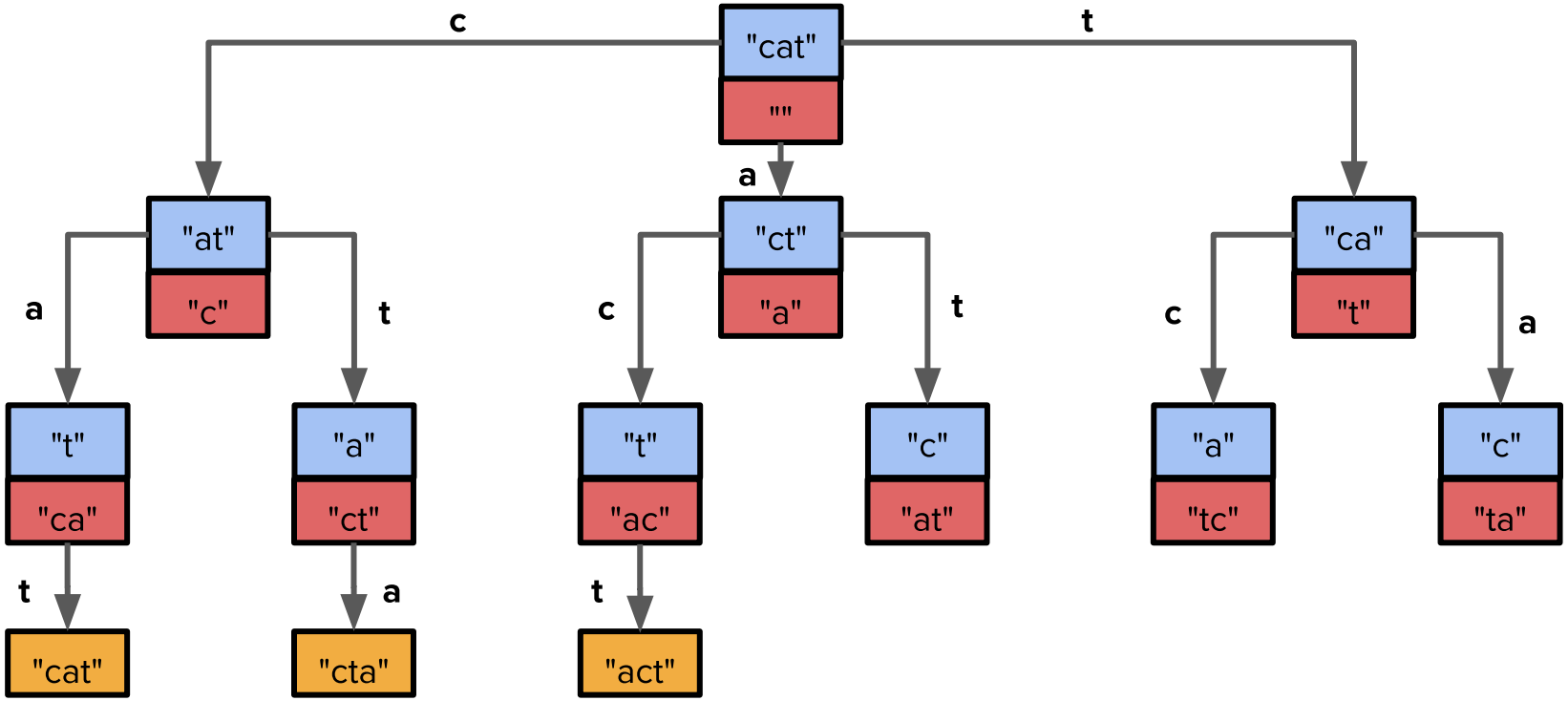
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



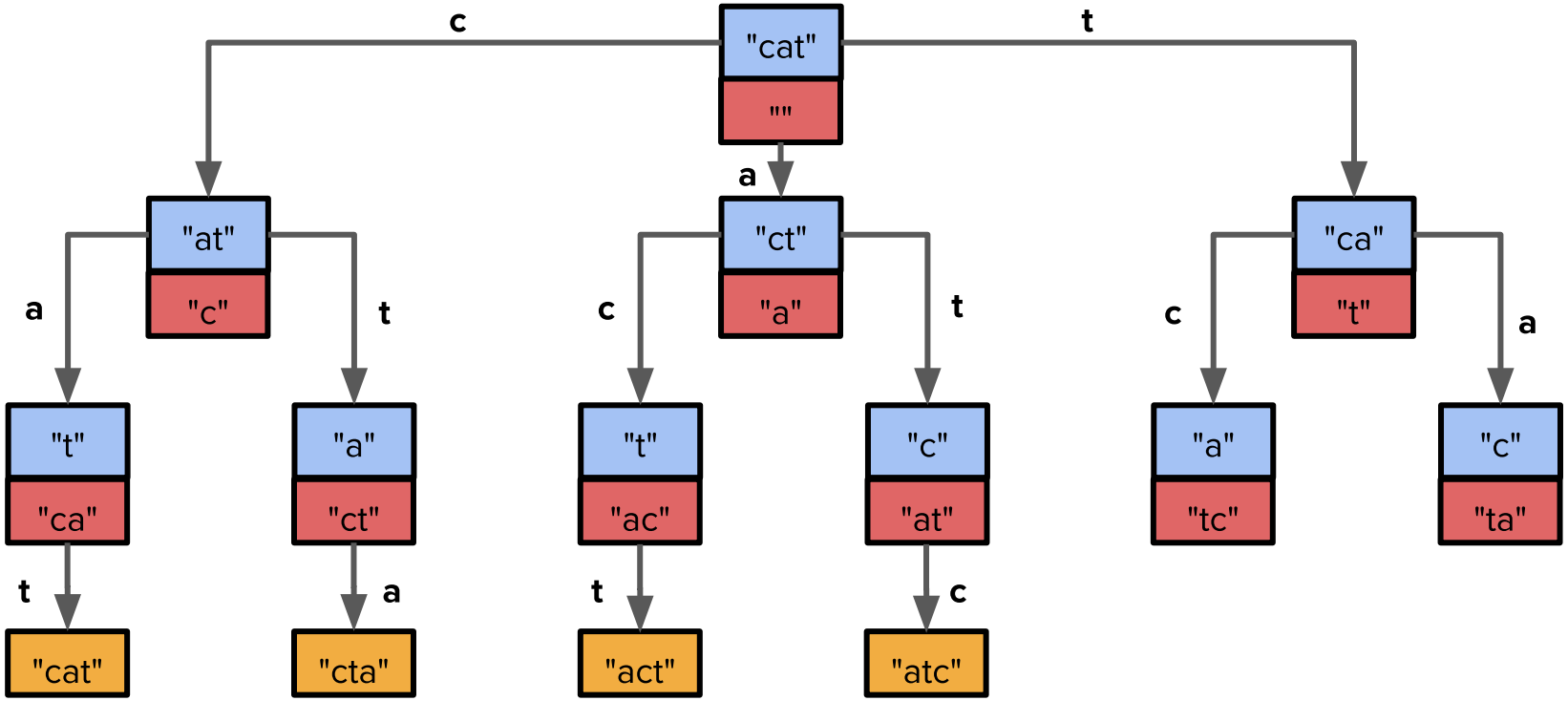
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



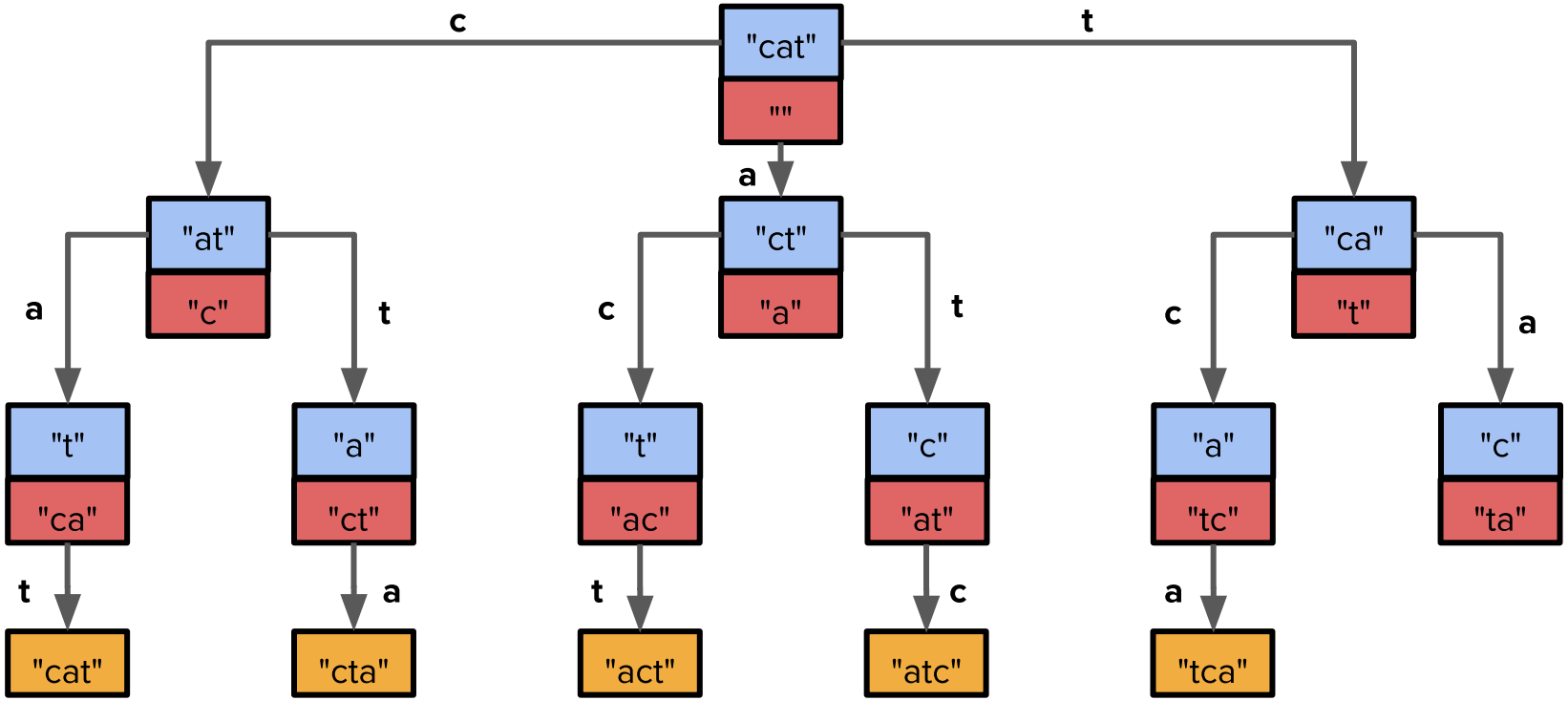
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



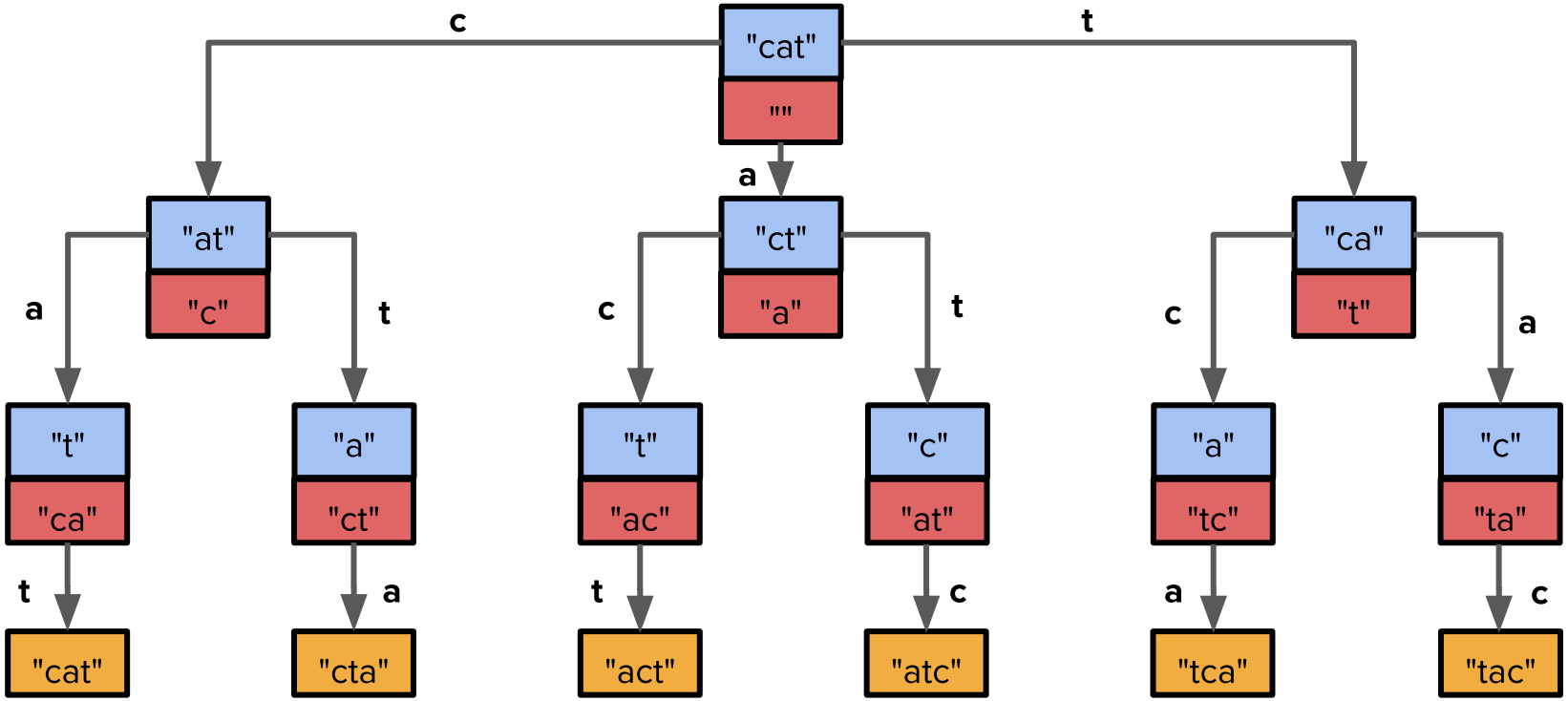
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



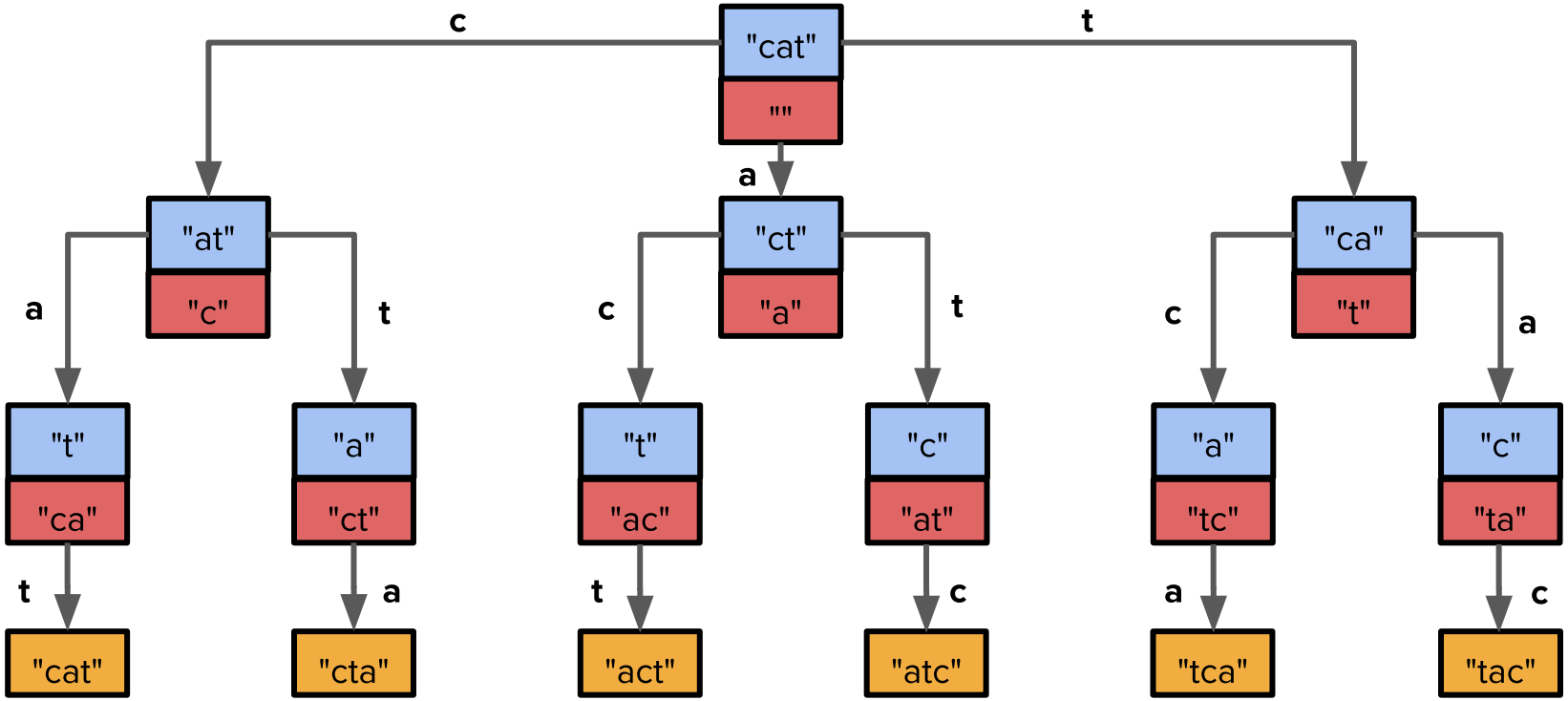
Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



Trial Version Wondershare PDFelement

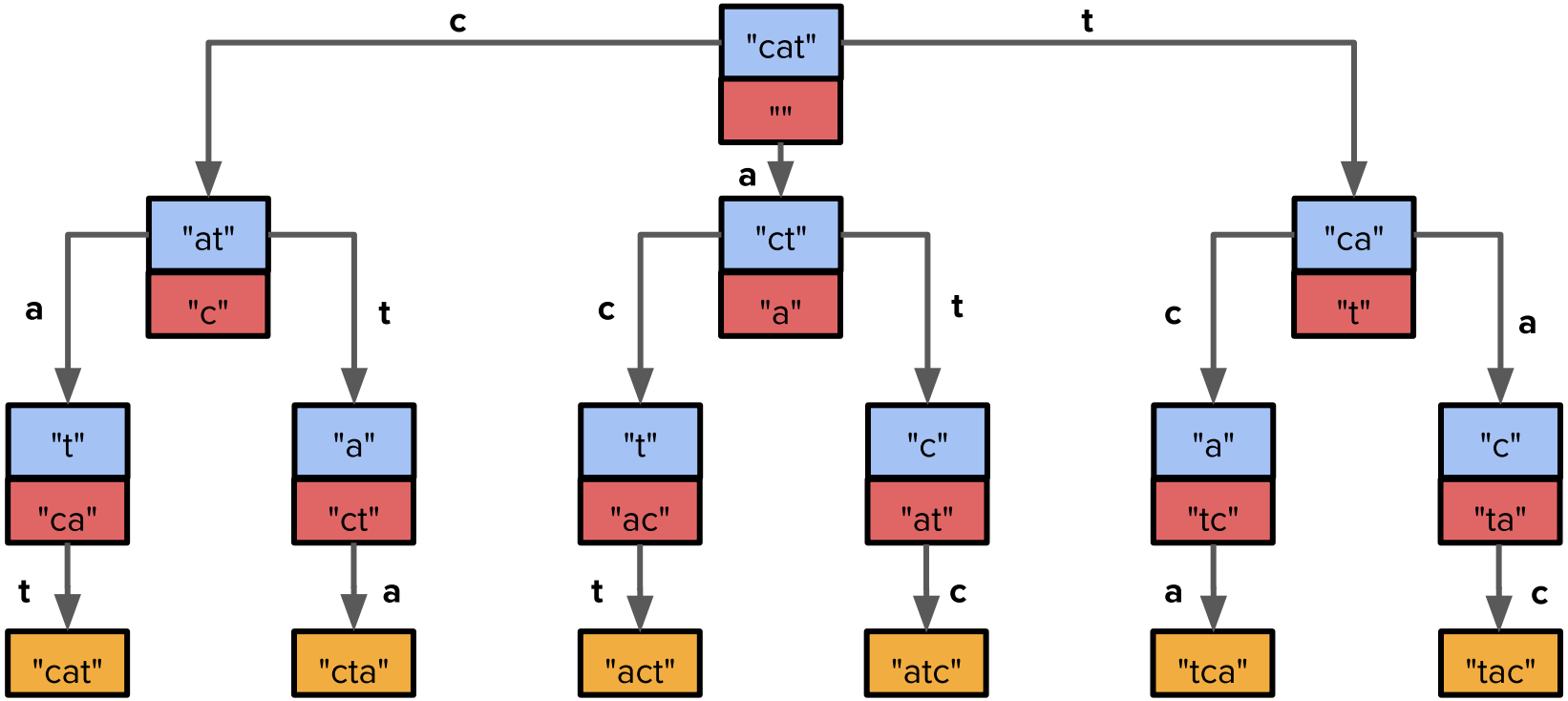
Decision tree: Find all permutations of "cat"



Base case: No letters remaining to choose!

Trial Version Wondershare PDFelement

Decision tree: Find all permutations of "cat"



Recursive case: For every letter remaining, add that letter to the current permutation and recurse!

Trial Version Wondershare PDFelement

Let’s code it!

Trial Version Wondershare PDFelement

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Trial Version Wondershare PDFelement

Permutations Code

Use of recursive helper function with empty string as starting point

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
    if (remaining.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < remaining.length(); i++) {
            char nextLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + nextLetter);
        }
    }
}
```

Trial Version Wondershare PDFelement

Permutations Code

Decisions yet to be made

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
    if (remaining.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < remaining.length(); i++) {
            char nextLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + nextLetter);
        }
    }
}
```

Trial Version Wondershare PDFelement

Permutations Code

Decisions yet to be made

Decisions already made

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
    if (remaining.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < remaining.length(); i++) {
            char nextLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + nextLetter);
        }
    }
}
```

Trial Version Wondershare PDFelement

Permutations Code

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
    if (remaining.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < remaining.length(); i++) {
            char nextLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + nextLetter);
        }
    }
}
```

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Trial Version Wondershare PDFelement

Permutations Code

```
void listPermutations(string s){
    listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
    if (remaining.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < remaining.length(); i++) {
            char nextLetter = remaining[i];
            string rest = remaining.substr(0, i) + remaining.substr(i+1);
            listPermutationsHelper(rest, soFar + nextLetter);
        }
    }
}
```

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Recursive case: Try all options for next decision

Trial Version Wondershare PDFelement

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**
  - Since we passed all our parameters by value, each recursive stack frame had its own independent copy of the string data that it could edit as appropriate
  - The "unchoose" step is **implicit** since there is no need to undo anything by virtue of the fact that editing a copy only has local consequences.

Trial Version Wondershare PDFelement

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

Trial Version Wondershare PDFelement

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

- Backtracking recursion can have **variable branching factors** at each level

Trial Version Wondershare PDFelement

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

- Backtracking recursion can have **variable branching factors** at each level

- Use of helper functions and initial empty params that get built up is common

Trial Version Wondershare PDFelement

Shrinkable Words

Trial Version Wondershare PDFelement

“What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?”

Trial Version Wondershare PDFelement

startling → starling → staring → string → sting → sing → sin → in → i

Trial Version Wondershare PDFelement

Is there really just one nine-letter word with this property?

Trial Version Wondershare PDFelement

How can we determine if a word is shrinkable?

- A **shrinkable word** is a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

- Idea: Let's use a decision tree to remove letters and determine **shrinkability**!

Trial Version Wondershare PDFelement

What defines our shrinkable decision tree?

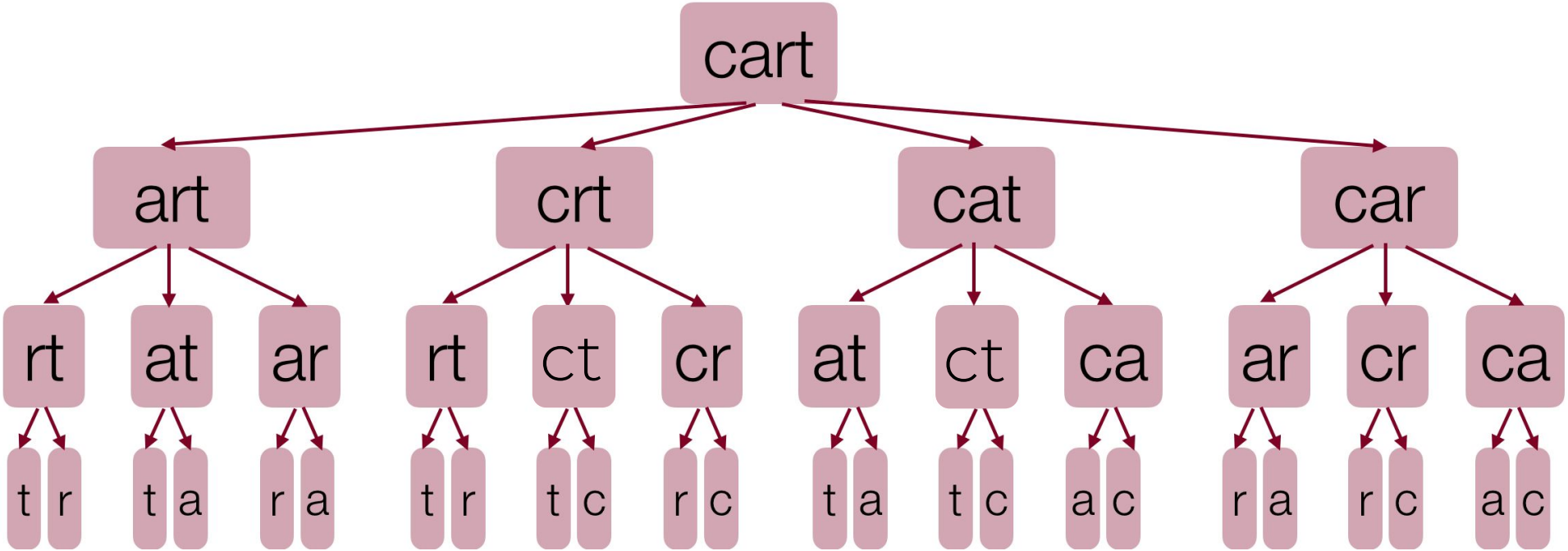
- **Decision** at each step (each level of the tree):
  - What letter are going to remove?

- **Options** at each decision (branches from each node):
  - The remaining letters in the string

- Information we need to store along the way:
  - The shrinking string

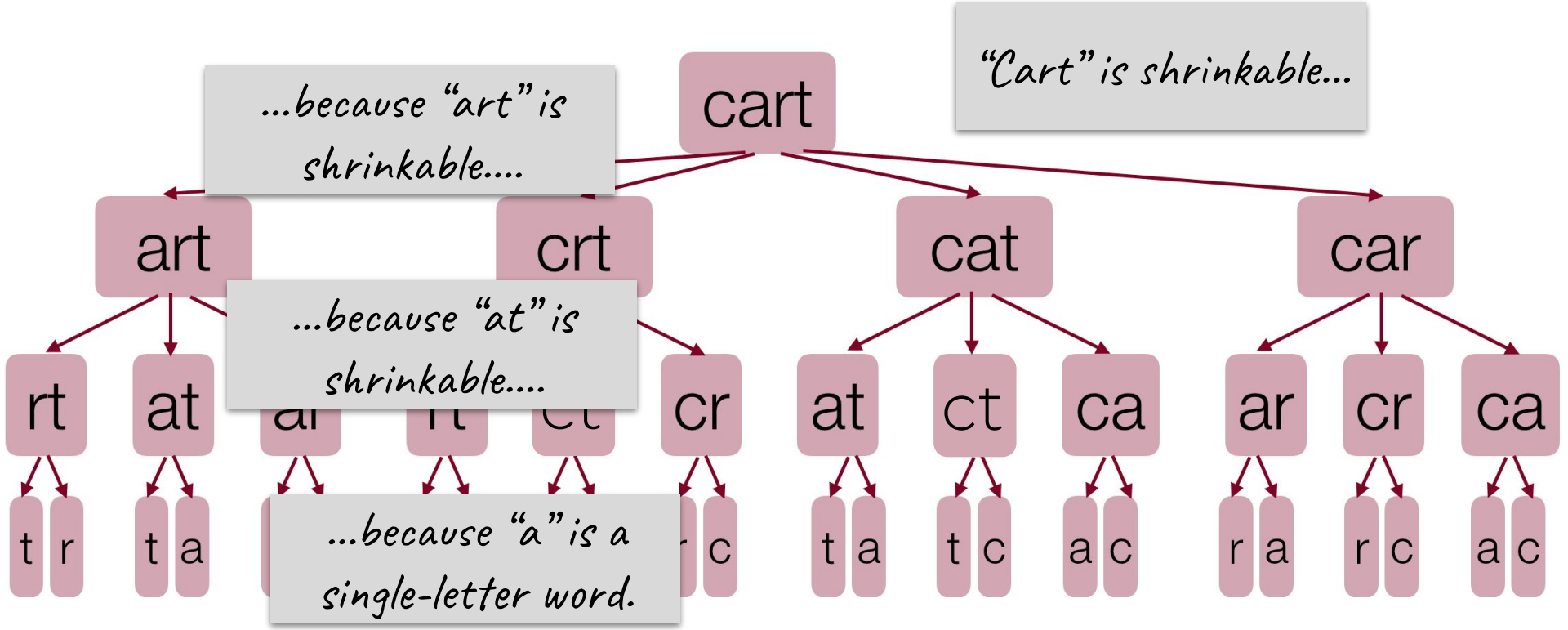
Trial Version Wondershare PDFelement

What defines our shrinkable decision tree?



Trial Version Wondershare PDFelement

What defines our shrinkable decision tree?

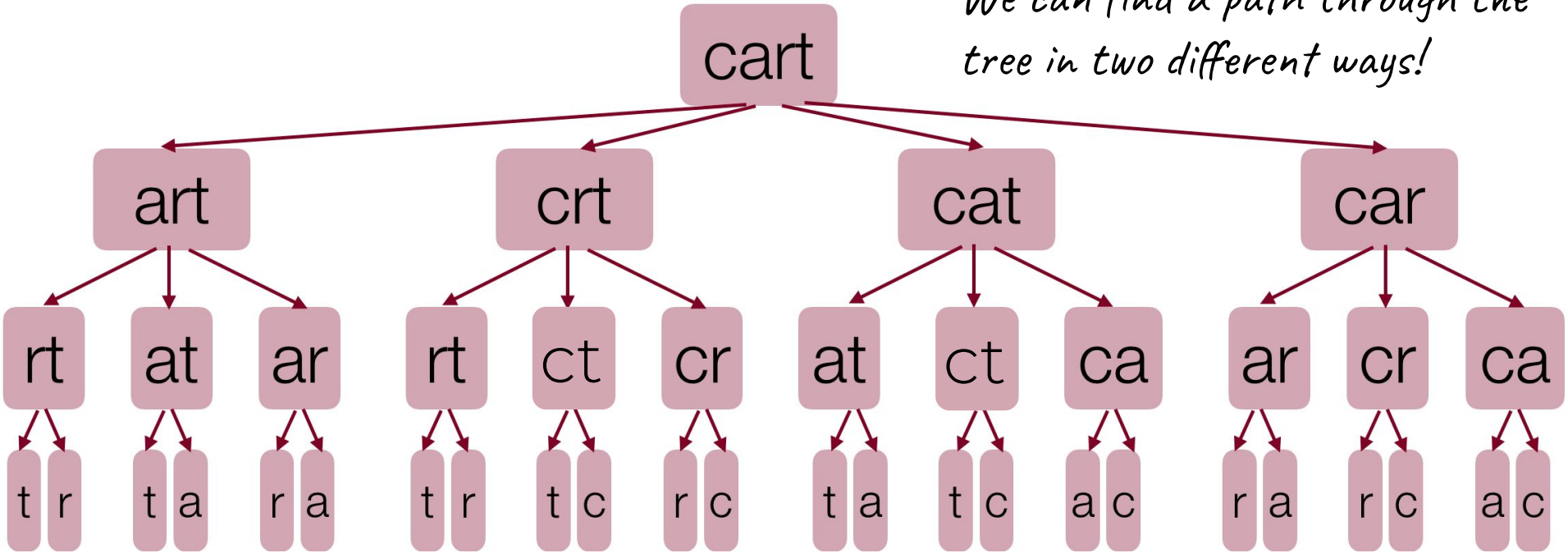


Examples from Chris Gregg and Keith Schwarz

Trial Version Wondershare PDFelement

What defines our shrinkable decision tree?

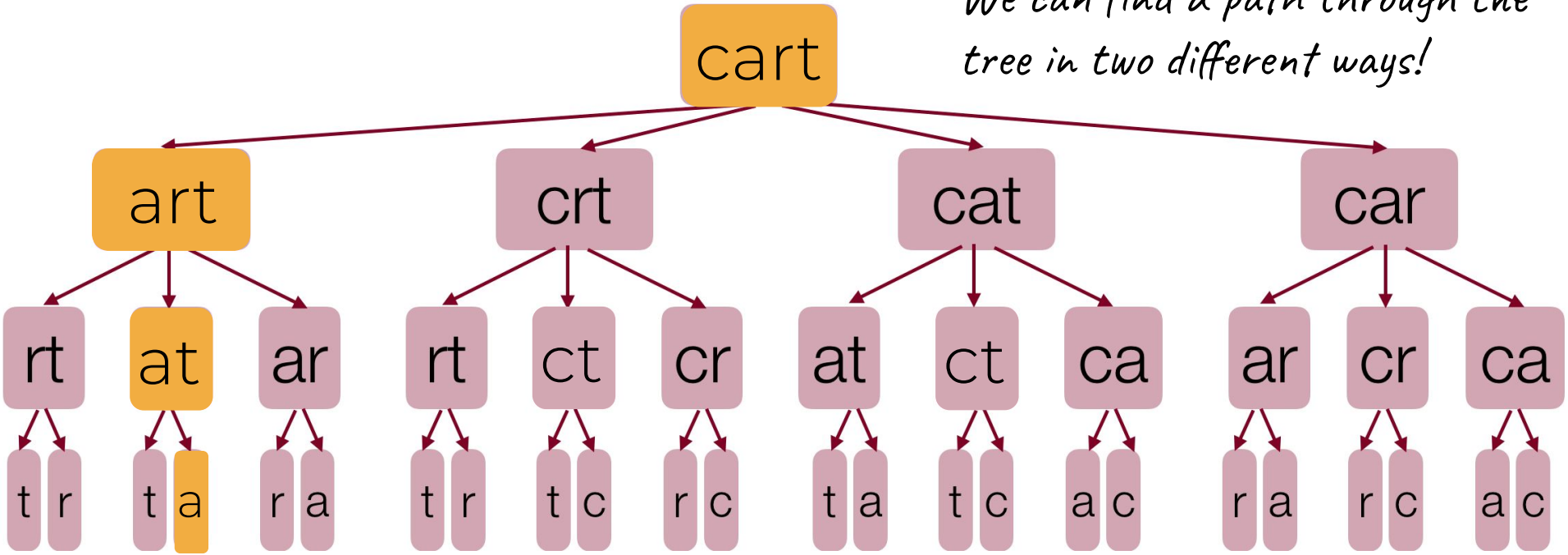
We can find a path through the tree in two different ways!



Trial Version Wondershare PDFelement

What defines our shrinkable decision tree?

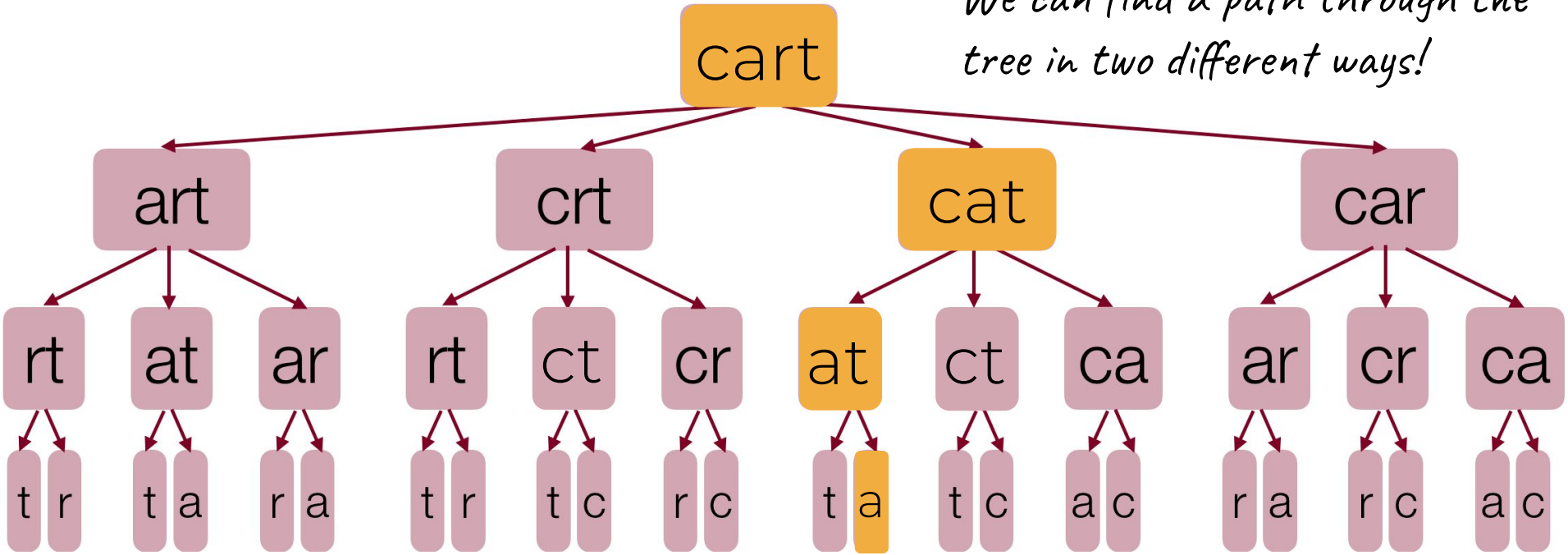
We can find a path through the tree in two different ways!



Trial Version Wondershare PDFelement

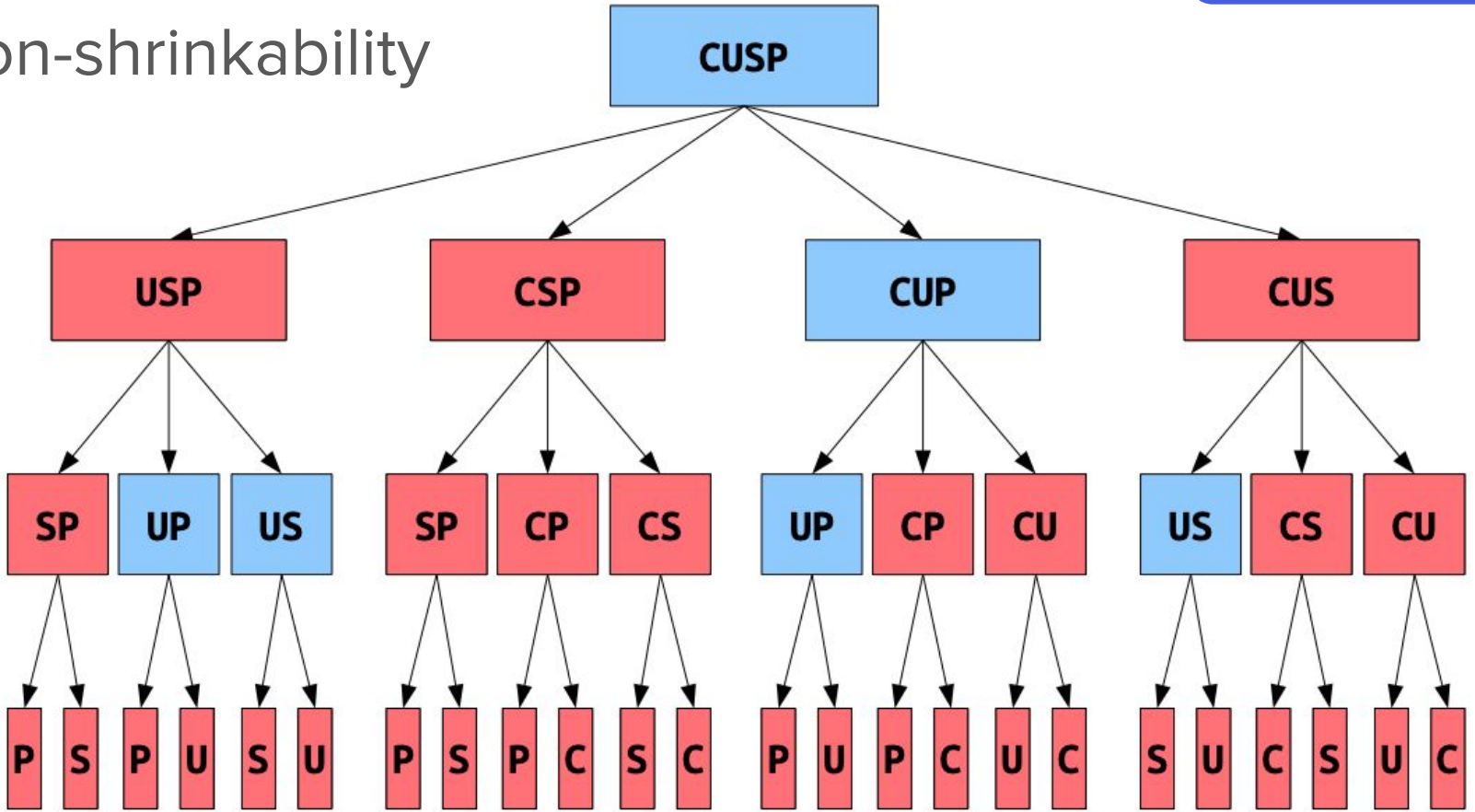
What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!



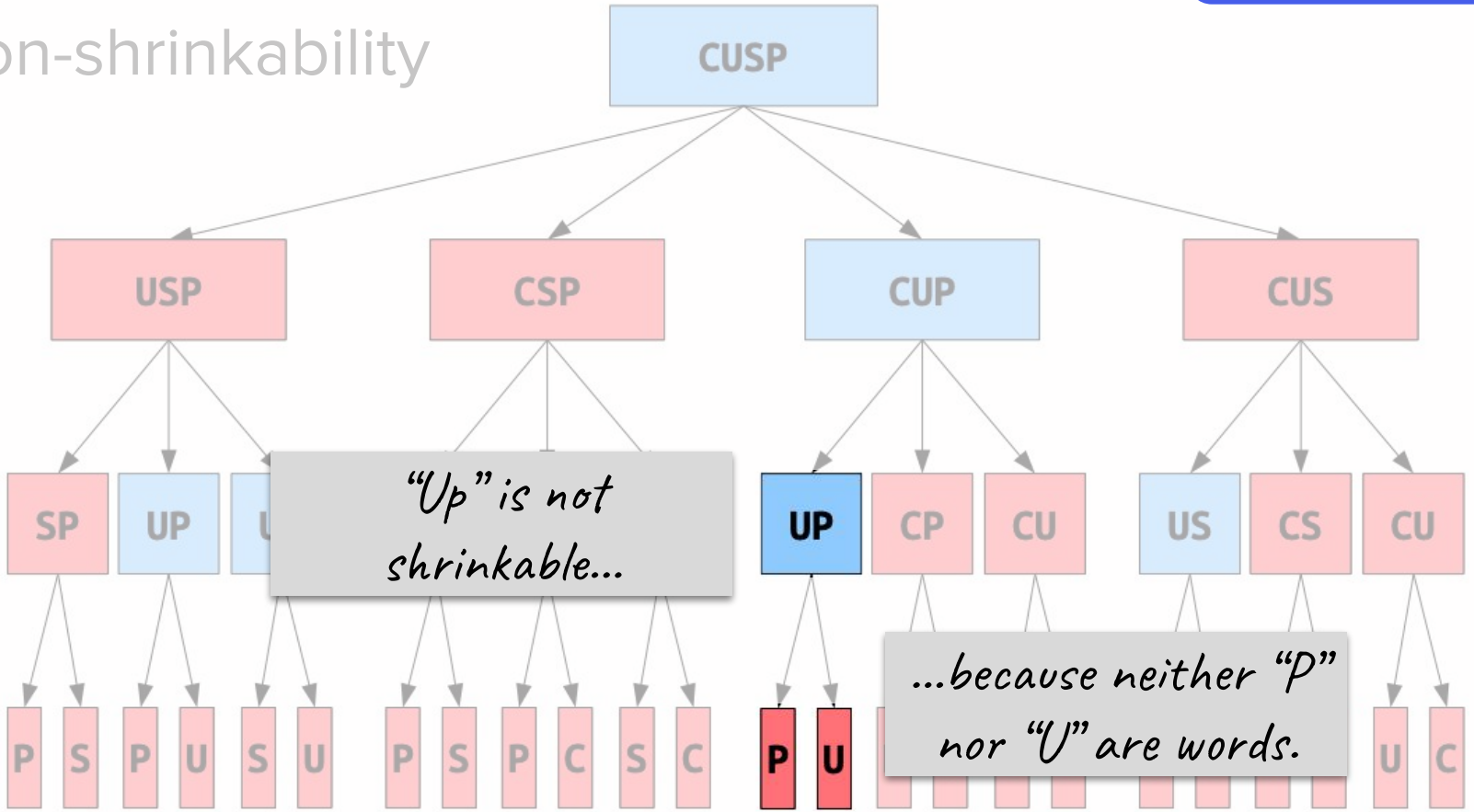
Trial Version Wondershare PDFelement

Non-shrinkability



Trial Version Wondershare PDFelement

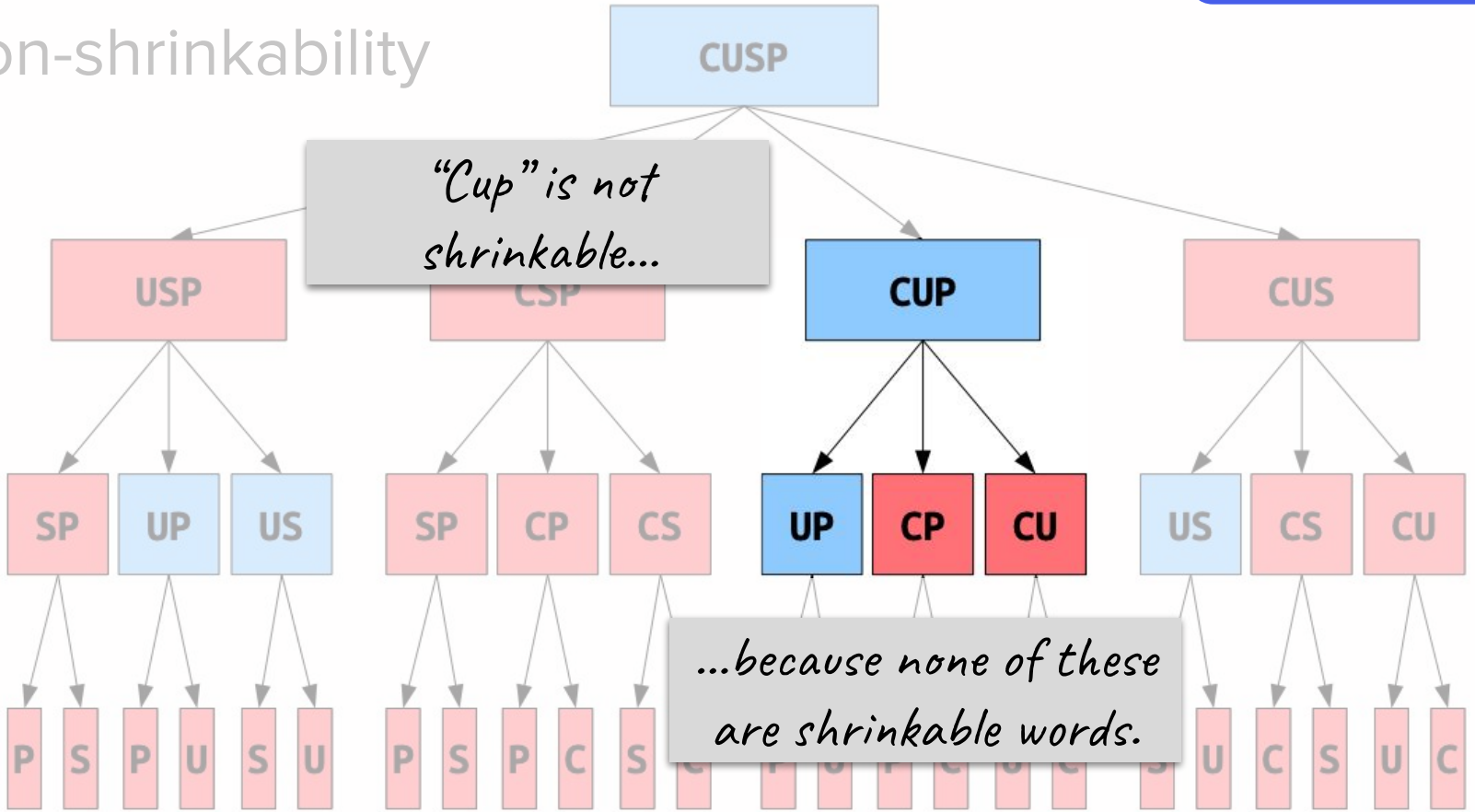
Non-shrinkability



Examples from Chris Gregg and Keith Schwarz

Trial Version Wondershare PDFelement

Non-shrinkability

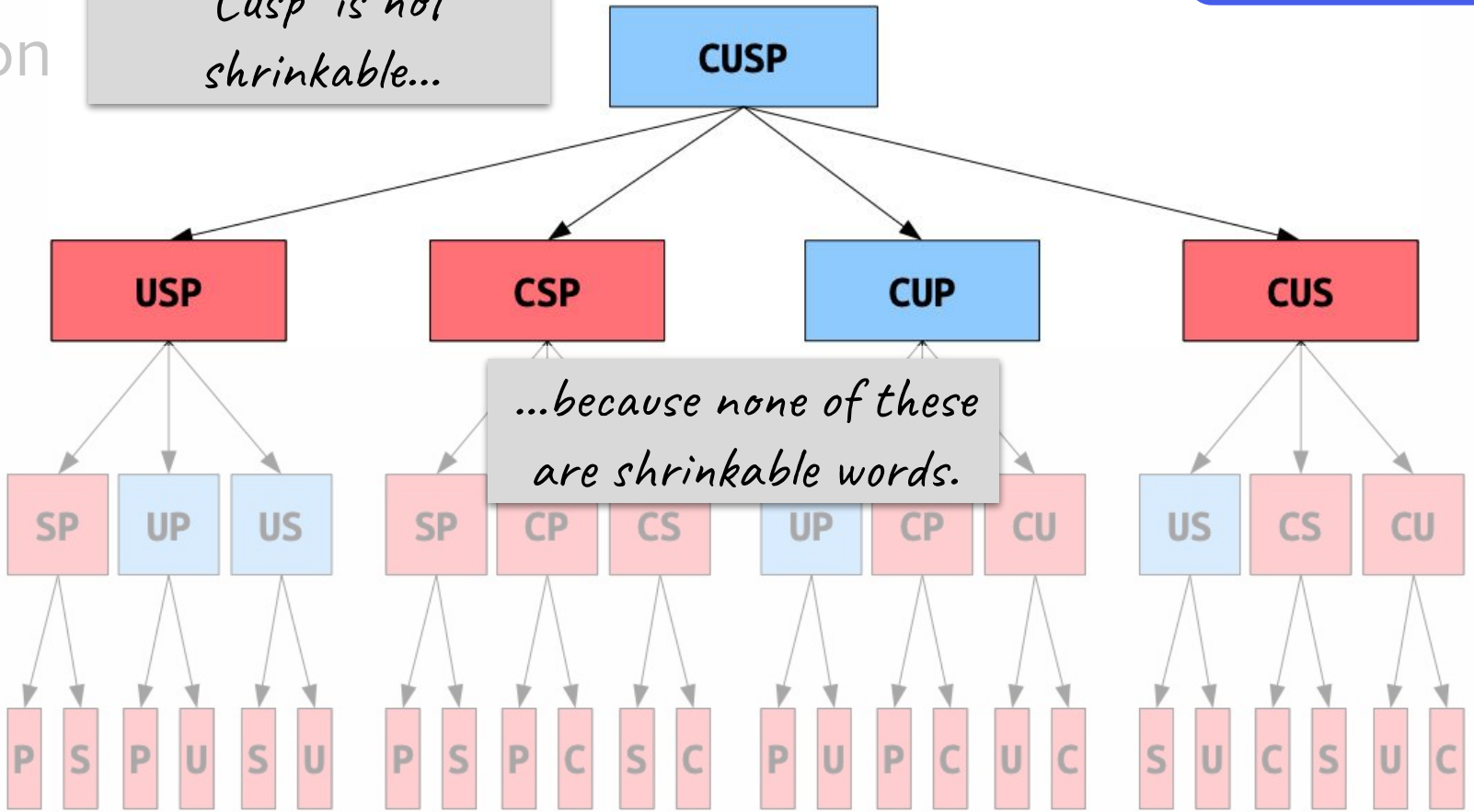


Examples from Chris Gregg and Keith Schwarz

Trial Version Wondershare PDFelement

Non

"Cusp" is not shrinkable...



...because none of these are shrinkable words.

Examples from Chris Gregg and Keith Schwarz

Trial Version Wondershare PDFelement

How can we determine if a word is shrinkable?

- **Base cases**:
  - A string that is not a word is not a shrinkable word.
  - Any single-letter word is shrinkable (A, I, and O).

- **Recursive cases**:
  - A multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.
  - A multi-letter word is not shrinkable if no matter what letter you remove, it’s not shrinkable.

Trial Version Wondershare PDFelement

Lexicon

- Lexicon is a helpful ADT provided by the Stanford C++ libraries (in **`lexicon.h`**) that is used specifically for storing many words that make up a dictionary

- Generally, Lexicons offer faster lookup than normal Sets, which is why we choose to use them when dealing with words and large dictionaries

- ```
  Lexicon lex("res/EnglishWords.txt"); // create from file
  lex.contains("koala"); // returns true
  lex.contains("zzzzz"); // returns false
  lex.containsPrefix("fi"); // returns true if there are
  any words starting with "fi" in the dictionary
  ```
  ```

Trial Version Wondershare PDFelement

Let’s code it!

Trial Version Wondershare PDFelement

Takeaways

- This is another example of **copy-edit-recurse** to choose, explore, and then implicitly unchoose!

- In this problem, we're using backtracking to **find if a solution exists**.
  - Notice the way the recursive case is structured:

```
for all options at each decision point:
    if recursive call returns true:
        return true;
return false if all options are exhausted;
```

Trial Version Wondershare PDFelement

Announcements

Trial Version Wondershare PDFelement

Announcements

- The grace period for Assignment 2 expires tonight at 11:59pm PDT.
- Assignment 3 will be released by the end of the day today.
- The Assignment 3 YEAH session will be hosted by Trip tomorrow evening at 6pm PDT. The slides and recording will be posted shortly after the session is over.
- We will be releasing practice problems and information about the diagnostic over the weekend.

Trial Version Wondershare PDFelement

Subsets

Trial Version Wondershare PDFelement

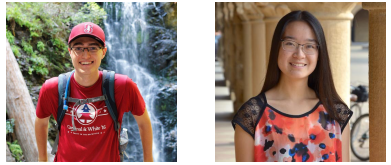
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:



Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{ }

Even though we may not care about this "team," the empty set is a subset of our original set!

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

As humans, it might be easiest to think about all teams (subsets) of a particular size.

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Another case of “generate/count all solutions” using recursive backtracking!

Trial Version Wondershare PDFelement

Discuss in breakouts:

What are the possible subsets of the choices {"c++", "python", "java", "javascript"}?

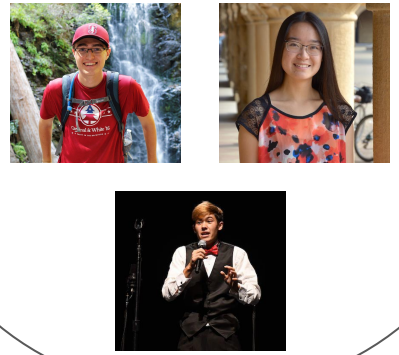
What potential recursive insights about generating subsets can you glean from this example?

[Time-permitting] Can you come up with a base case and recursive case for generating permutations?

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:



```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

For computers generating subsets (and thinking about decisions), there’s another pattern we might notice...

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}

Half the subsets contain “Nick”

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}

Half the subsets contain “Kylie”

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Half the subsets contain “Trip”

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Half the subsets that contain “Trip” also contain “Nick”

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Half the subsets that contain both “Trip” and “Nick” contain “Kylie”

Trial Version Wondershare PDFelement

Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Trial Version Wondershare PDFelement

What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

Trial Version Wondershare PDFelement

What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

Trial Version Wondershare PDFelement

What defines our subsets decision tree?

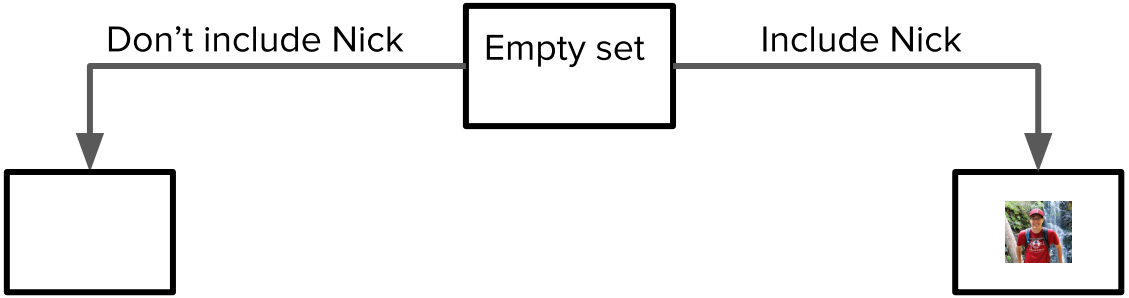
- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - The remaining elements in the original set

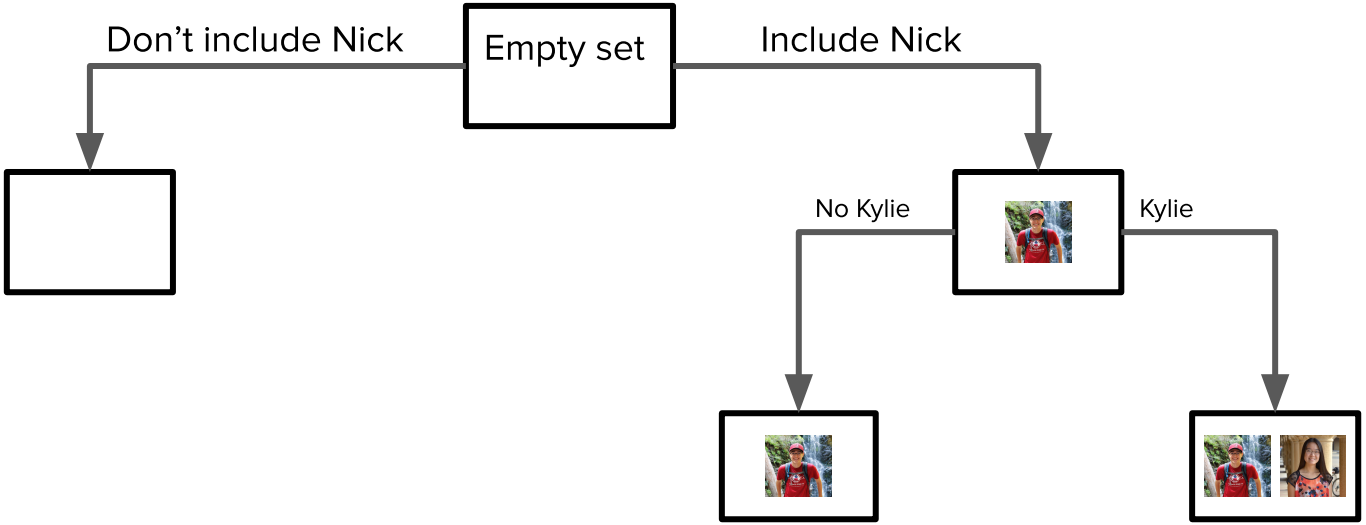
Trial Version Wondershare PDFelement

Decision tree



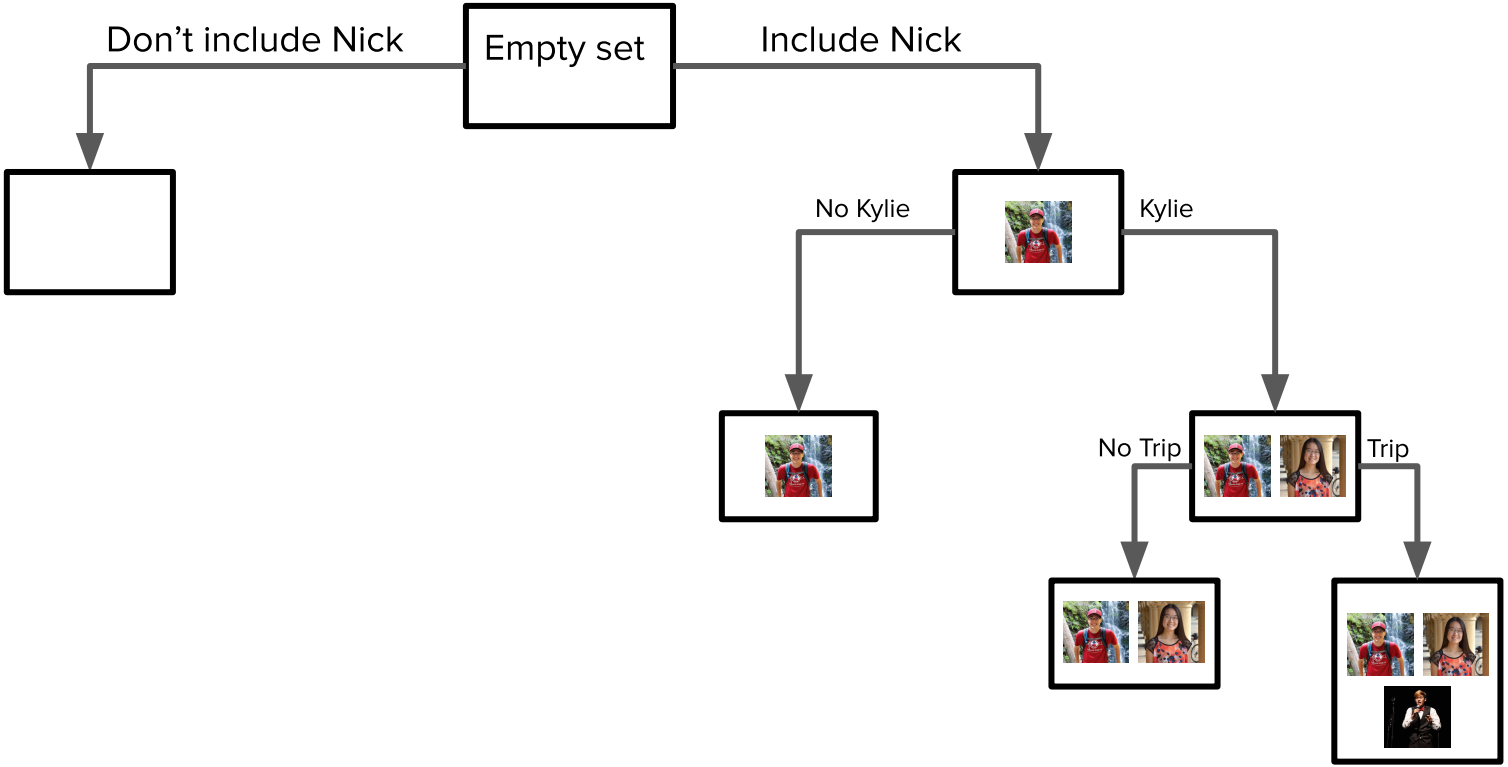
Trial Version Wondershare PDFelement

Decision tree



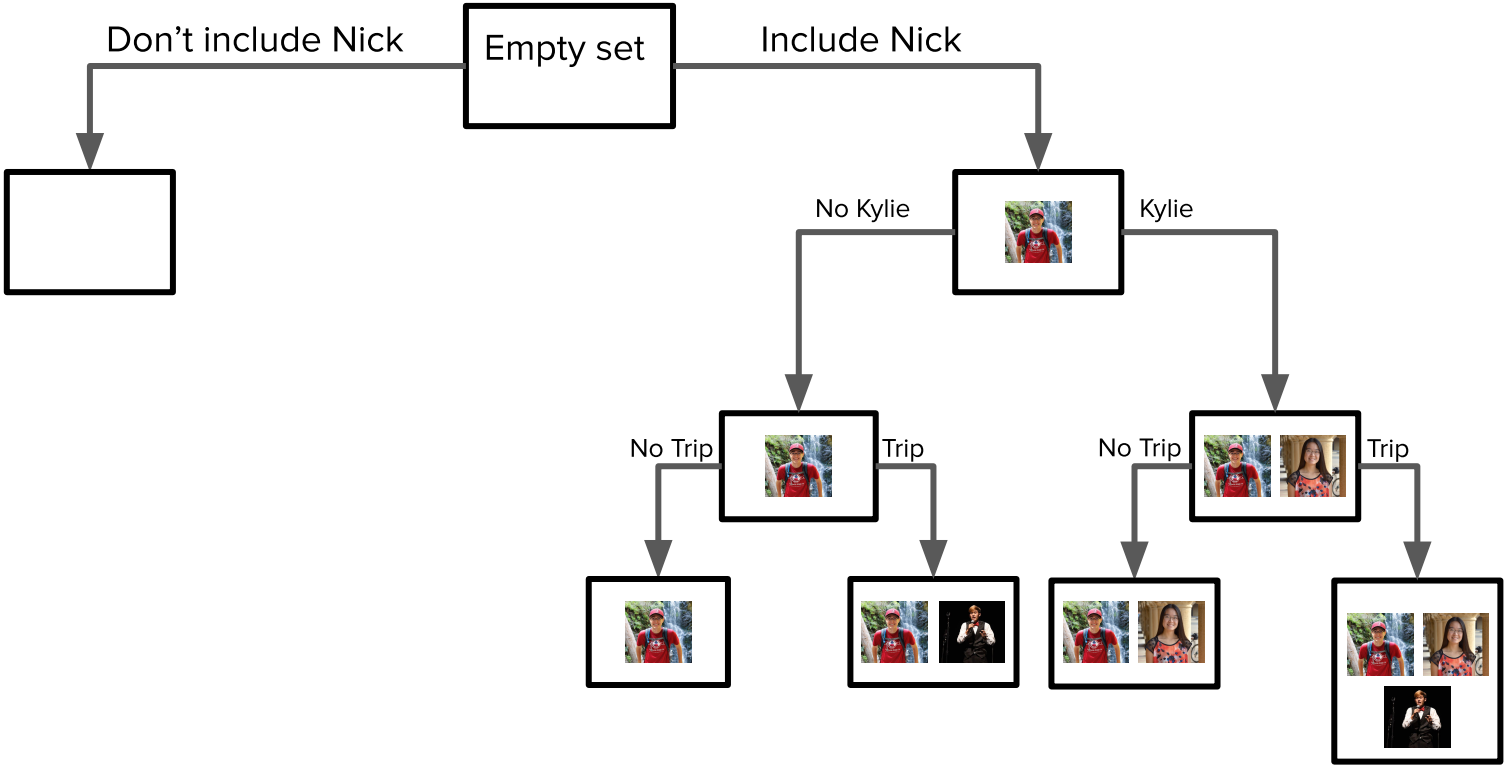
Trial Version Wondershare PDFelement

Decision tree



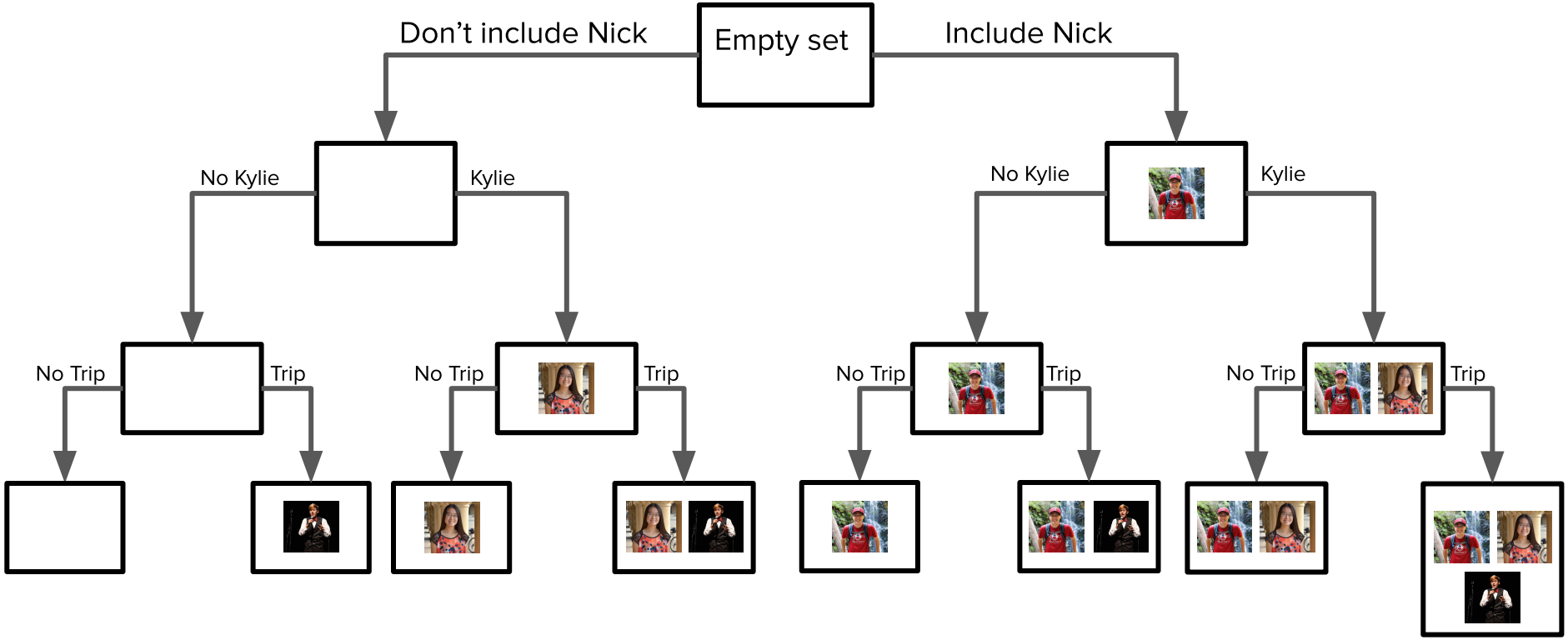
Trial Version Wondershare PDFelement

Decision tree



Trial Version Wondershare PDFelement

Decision tree



Trial Version Wondershare PDFelement

What defines our subsets decision tree?

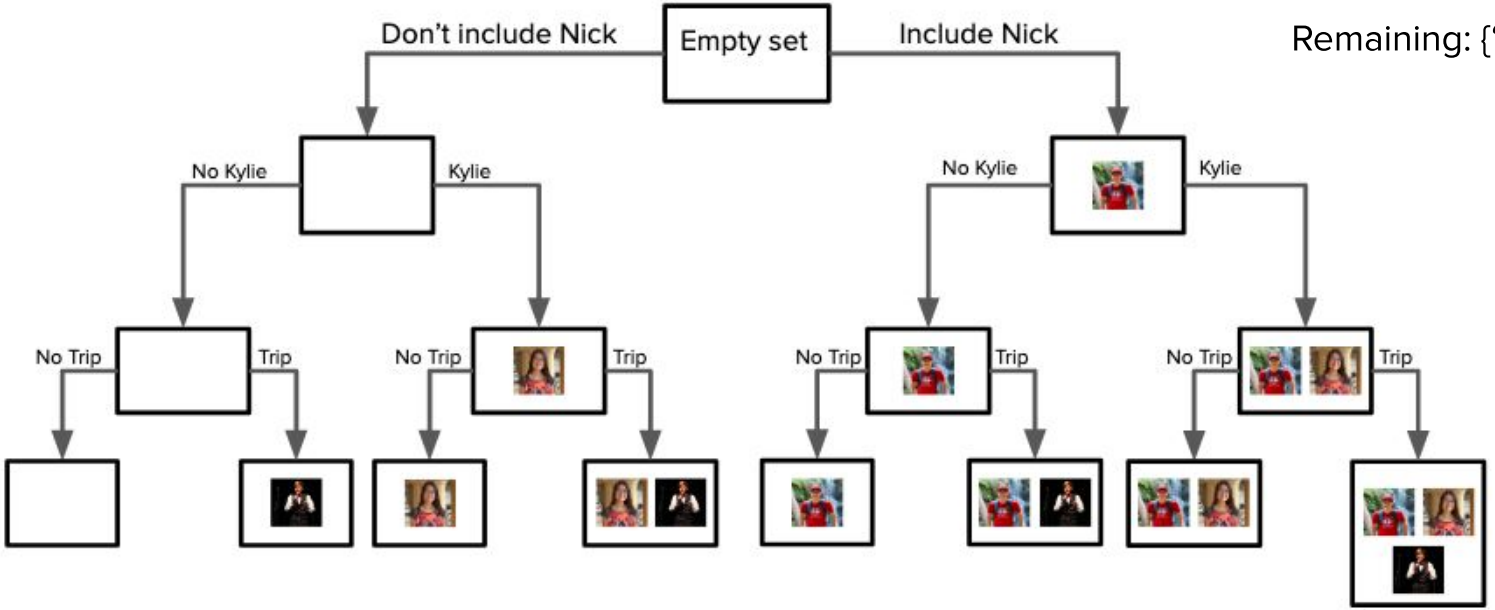
- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - **The remaining elements in the original set**

Trial Version Wondershare PDFelement

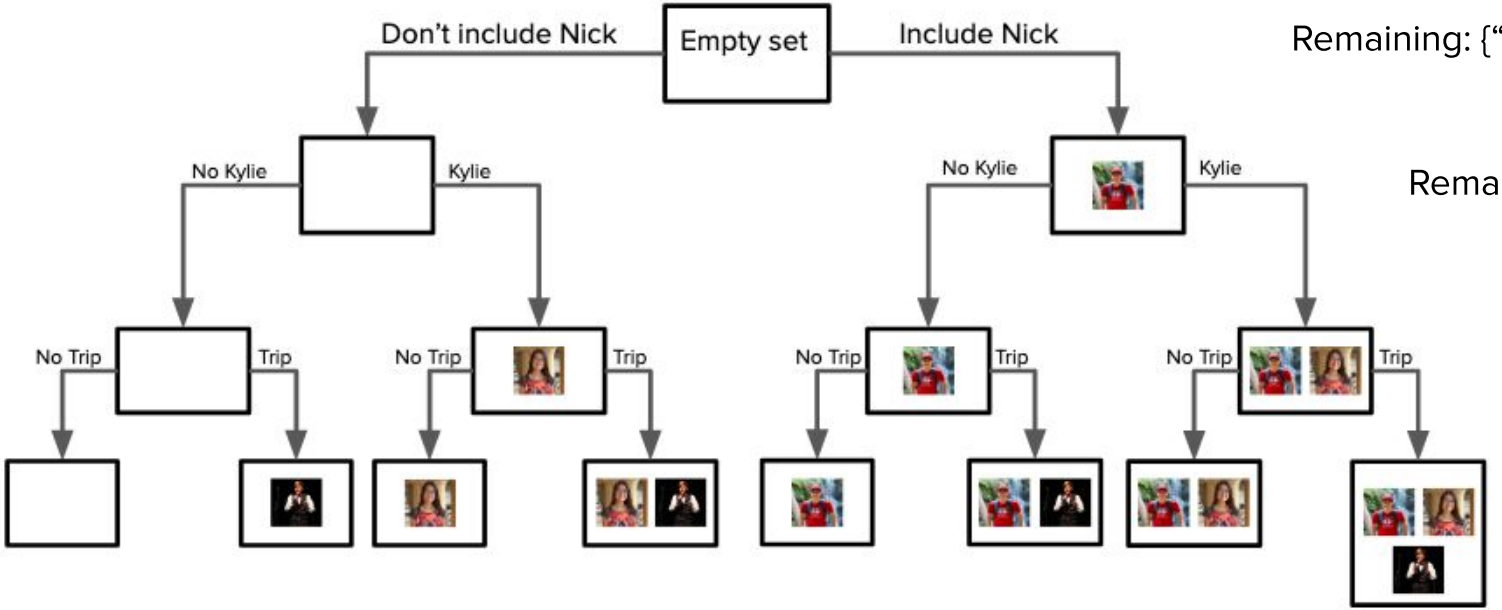
Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

Trial Version Wondershare PDFelement

Decision tree

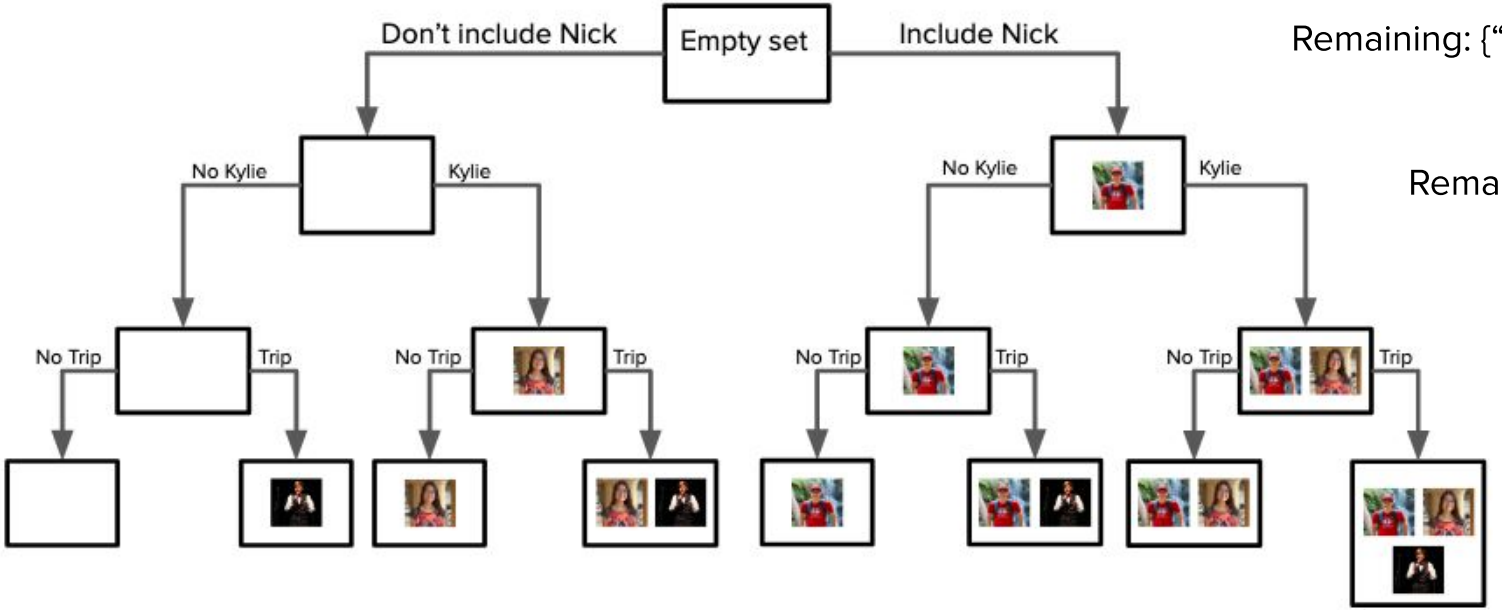


Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Trial Version Wondershare PDFelement

Decision tree



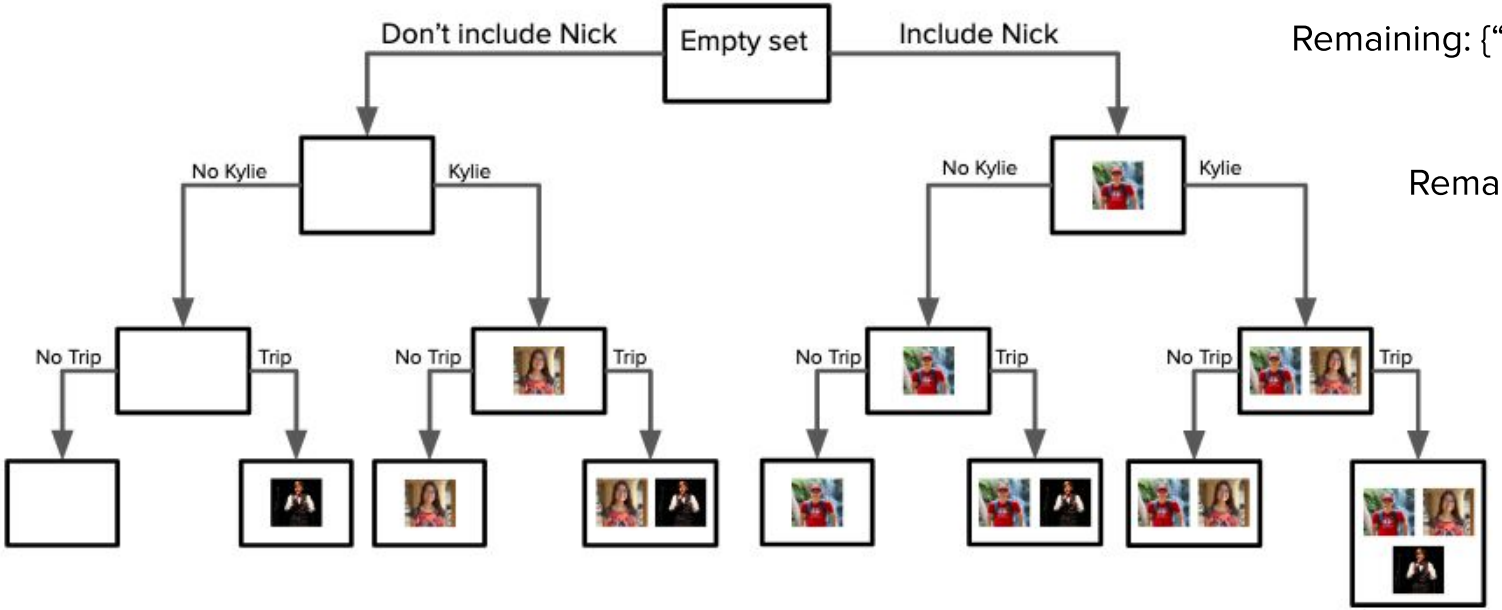
Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Trial Version Wondershare PDFelement

Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

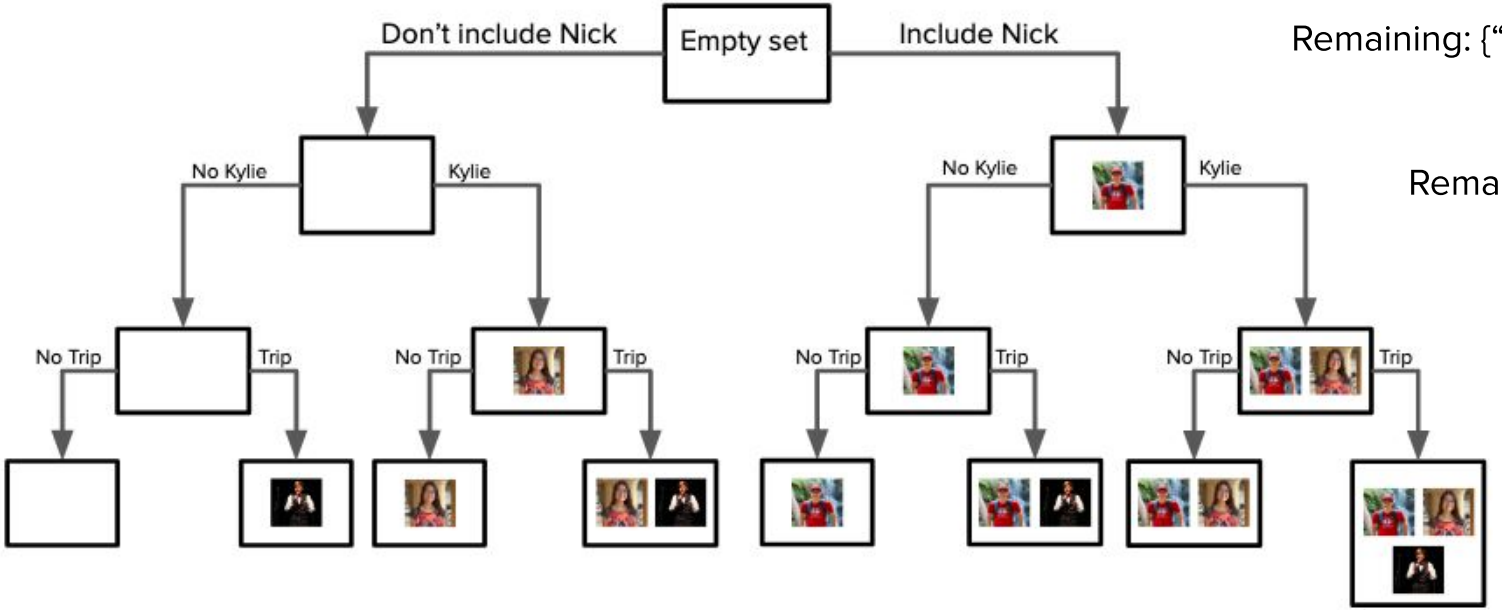
Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

Trial Version Wondershare PDFelement

Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

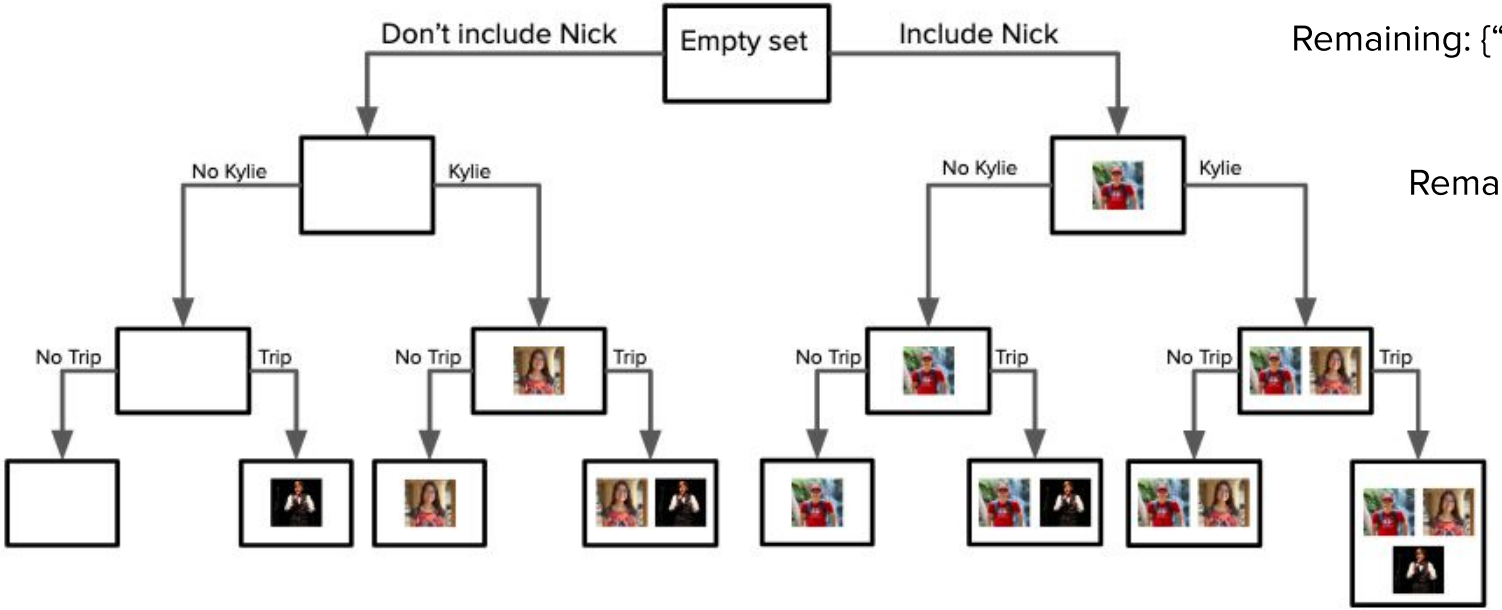
Remaining: {"Trip"}

Remaining: {}

Base case: No people remaining to choose from!

Trial Version Wondershare PDFelement

Decision tree



Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {}

Recursive case: Pick someone in the set. Choose to include or not include them.

Trial Version Wondershare PDFelement

Let’s code it!

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Choose

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Explore (part 1)

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Explore (part 2)

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Explicit Unchoose (i.e. undo)

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

Without this step, we could not explore the other side of the tree

```
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.

Trial Version Wondershare PDFelement

Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.

- The subset problem contains themes we’ve seen in backtracking recursion:
  - Building up solutions as we go down the decision tree
  - Using a helper function to abstract away implementation details

Trial Version Wondershare PDFelement

Summary

Trial Version Wondershare PDFelement

Backtracking recursion: **Exploring many possible solutions**

Overall paradigm: choose/explore/unchoose

Two ways of doing it

- **Choose explore undo**
  - Uses pass by reference; usually with large data structures
  - Explicit unchoose step by "undoing" prior modifications to structure
  - E.g. Generating subsets (one set passed around by reference to track subsets)
- **Copy edit explore**
  - Pass by value; usually when memory constraints aren't an issue
  - Implicit unchoose step by virtue of making edits to copy
  - E.g. Building up a string over time

Three use cases for backtracking

1. Generate/count all solutions (enumeration)
2. Find one solution (or prove existence)
3. Pick one best solution

General examples of things you can do:

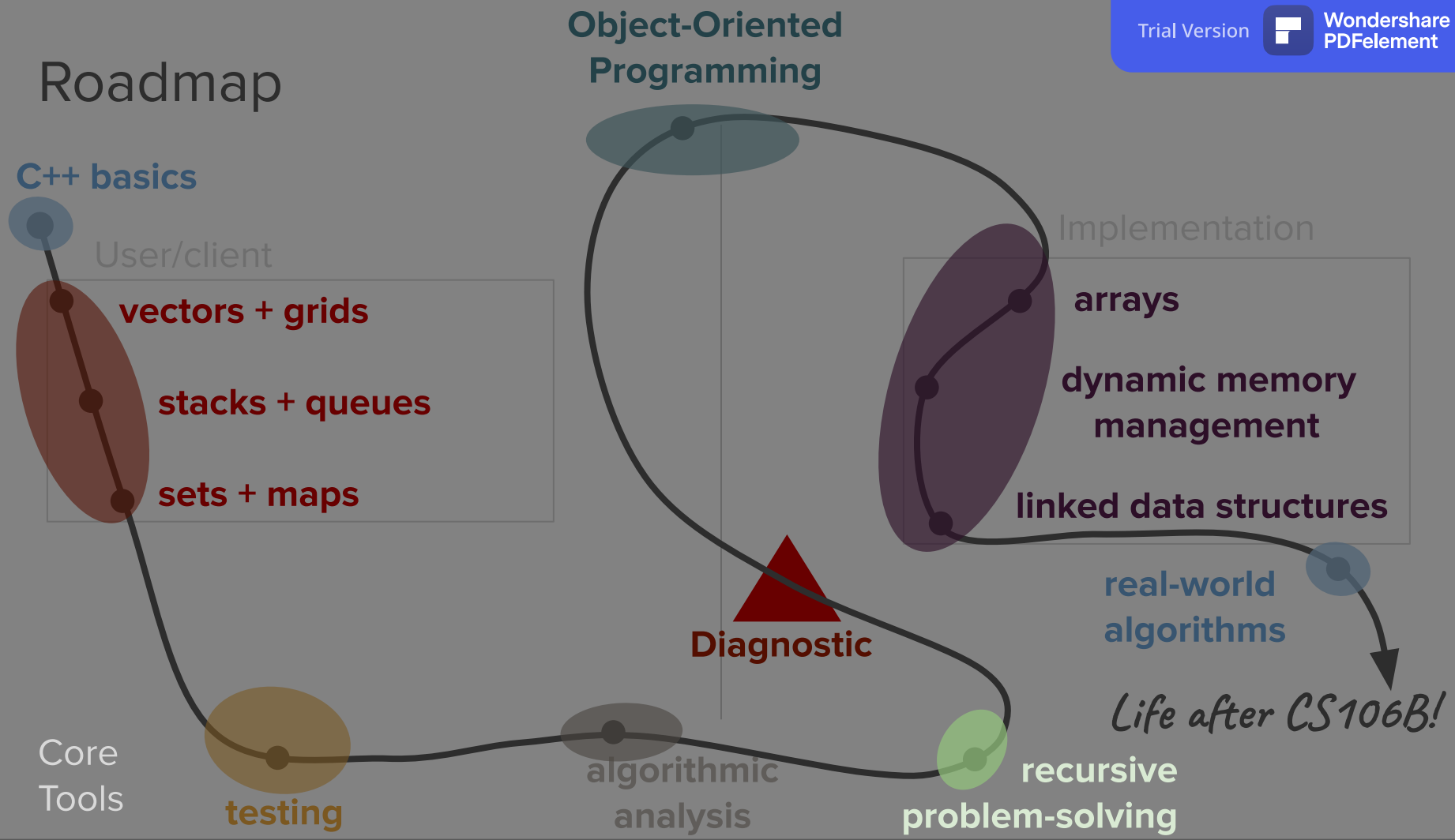
- Permutations
- Subsets
- Combinations
- etc.

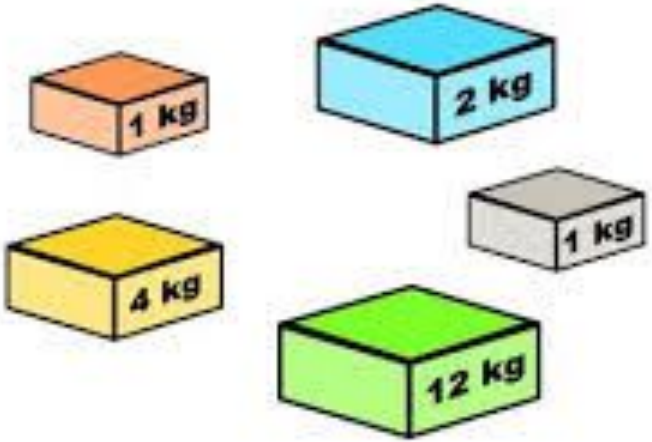
Trial Version Wondershare PDFelement

What’s next?

Trial Version Wondershare PDFelement

Roadmap



Trial Version Wondershare PDFelement

More Recursive Backtracking