Thinking Recursively

Part II

Outline for Today

- ***The Recursive Leap of Faith***
  - On trusting the contract.
- ***Enumerating Subsets***
  - A classic combinatorial problem.
- ***Decision Trees***
  - Generating all solutions to a problem.
- ***Wrapper Functions***
  - Hiding parameters and keeping things clean.

Some Quick Refreshers

Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

Answer at
https://pollev.com/cs106bwin23

Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

`Set<int> mySet`

Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

```
Set<int> mySet
```

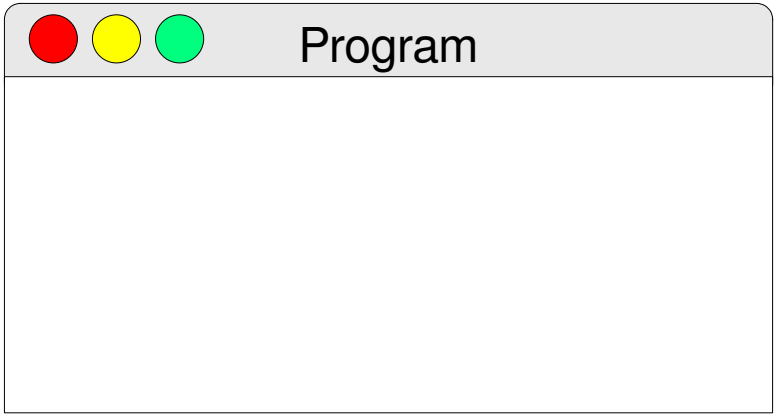
Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

```
Set<int> mySet
```



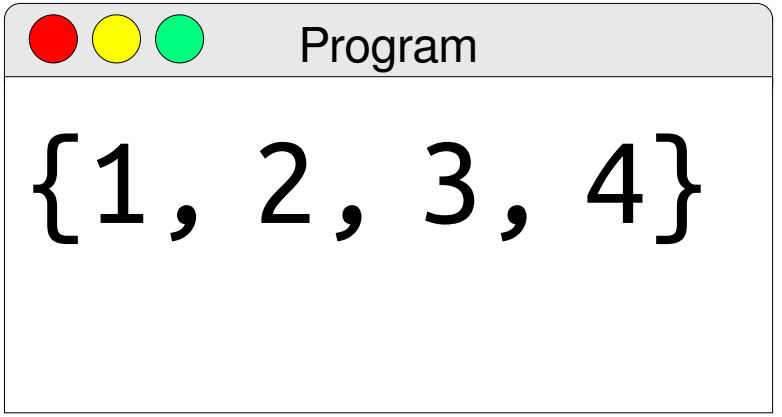
Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

```
Set<int> mySet
```



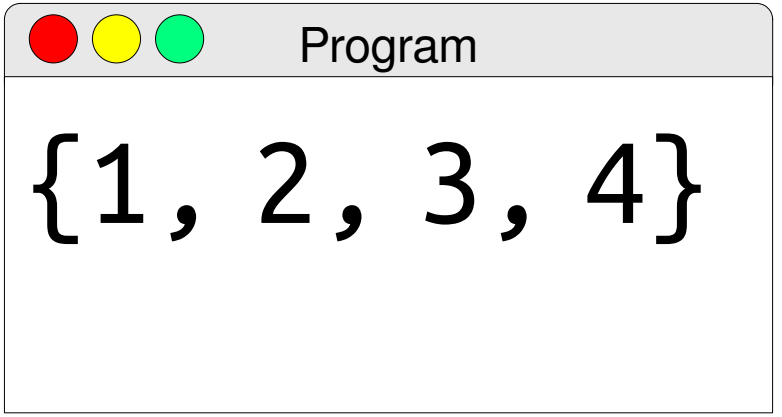
Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

Set<int> mySet



Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

`Set<int> mySet`

Program

```
{1, 2, 3, 4}
{1, 2}
```

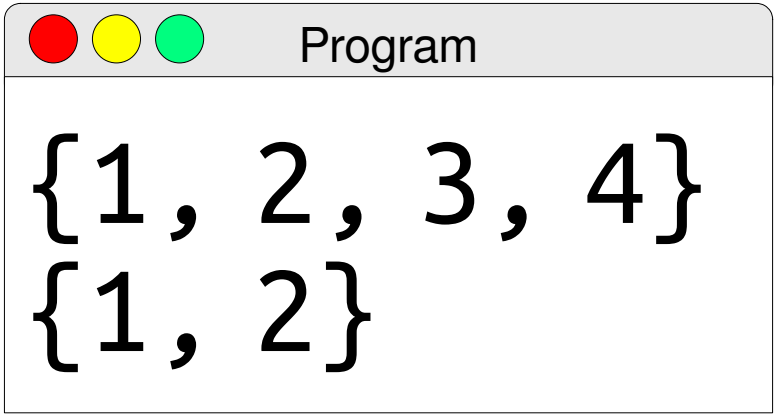
Set Refresher

- What's printed at Line *A* and Line *B*?

```
Set<int> mySet = {1, 2, 3};
cout << (mySet + 4) << endl; // Line A
cout << (mySet - 3) << endl; // Line B
```

```
{1, 2, 3}
```

Set<int> mySet



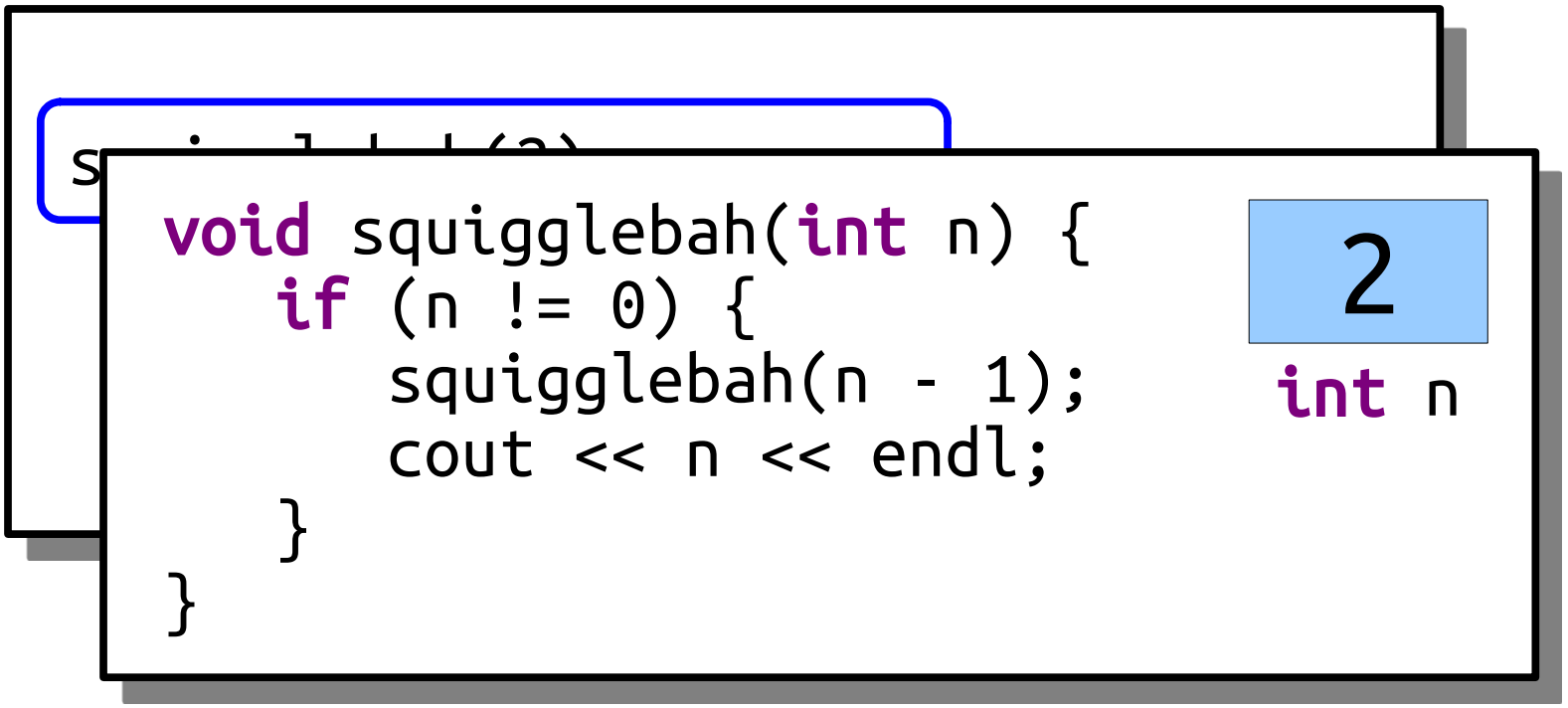
Recursion Refresher

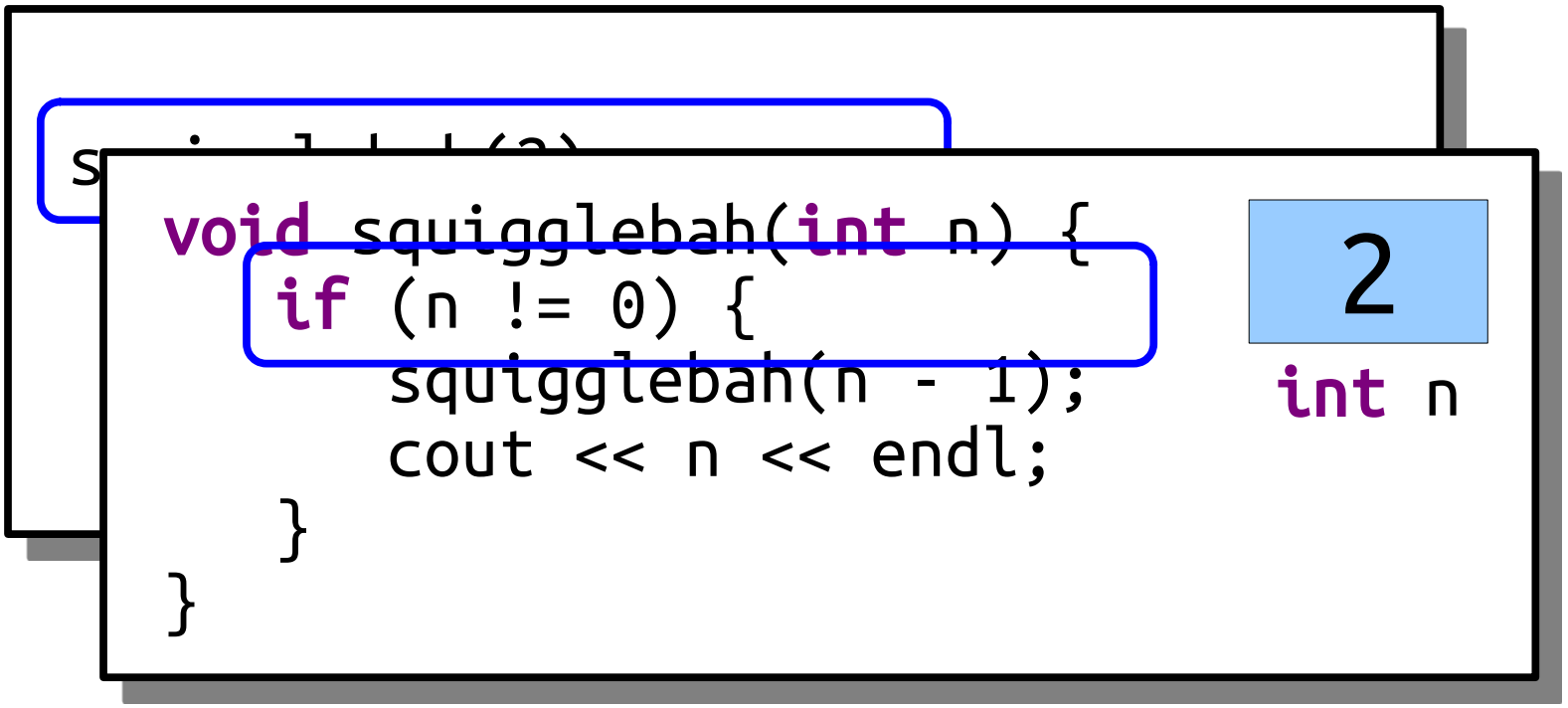
- What does this code print?

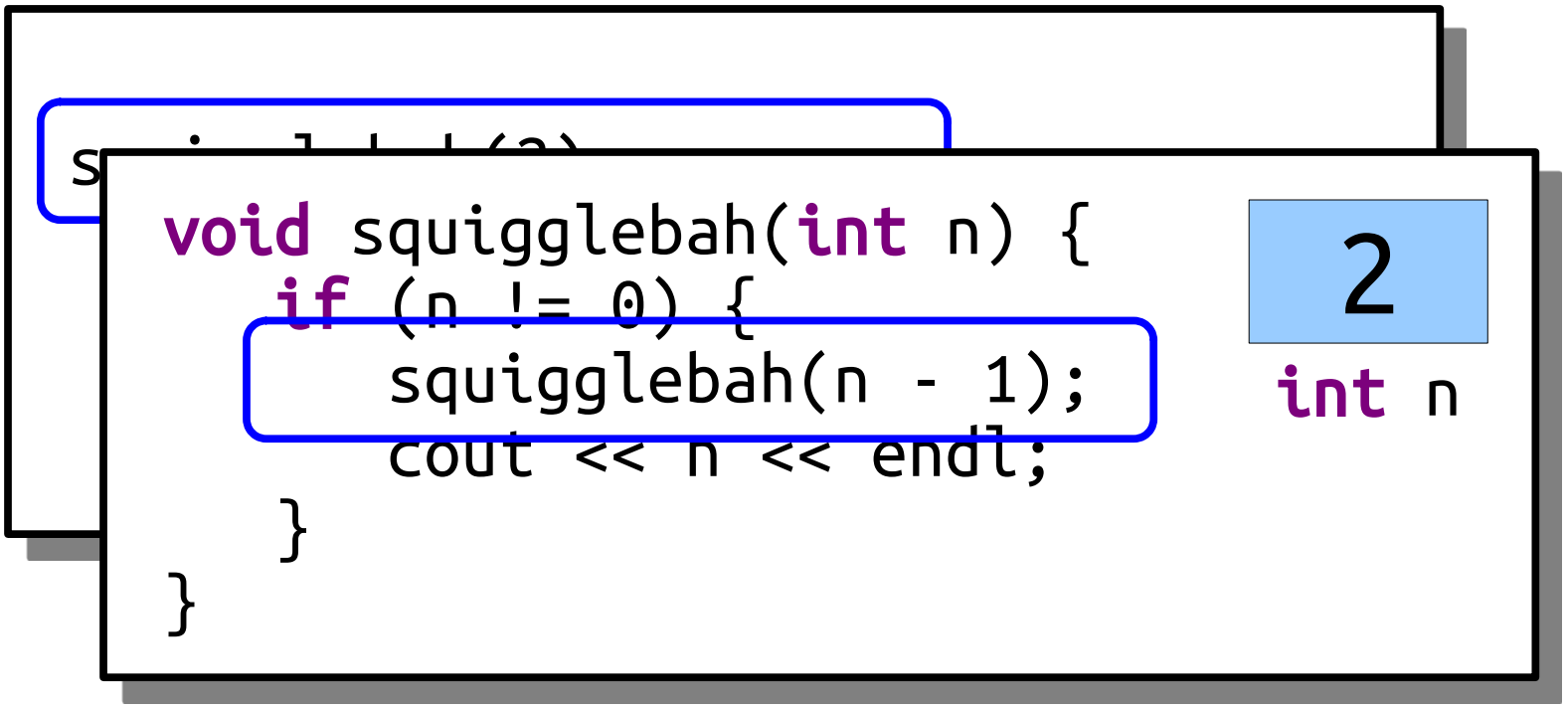
```
void squigglebah(int n) {
    if (n != 0) {
        squigglebah(n - 1);
        cout << n << endl;
    }
}

squigglebah(2);
```

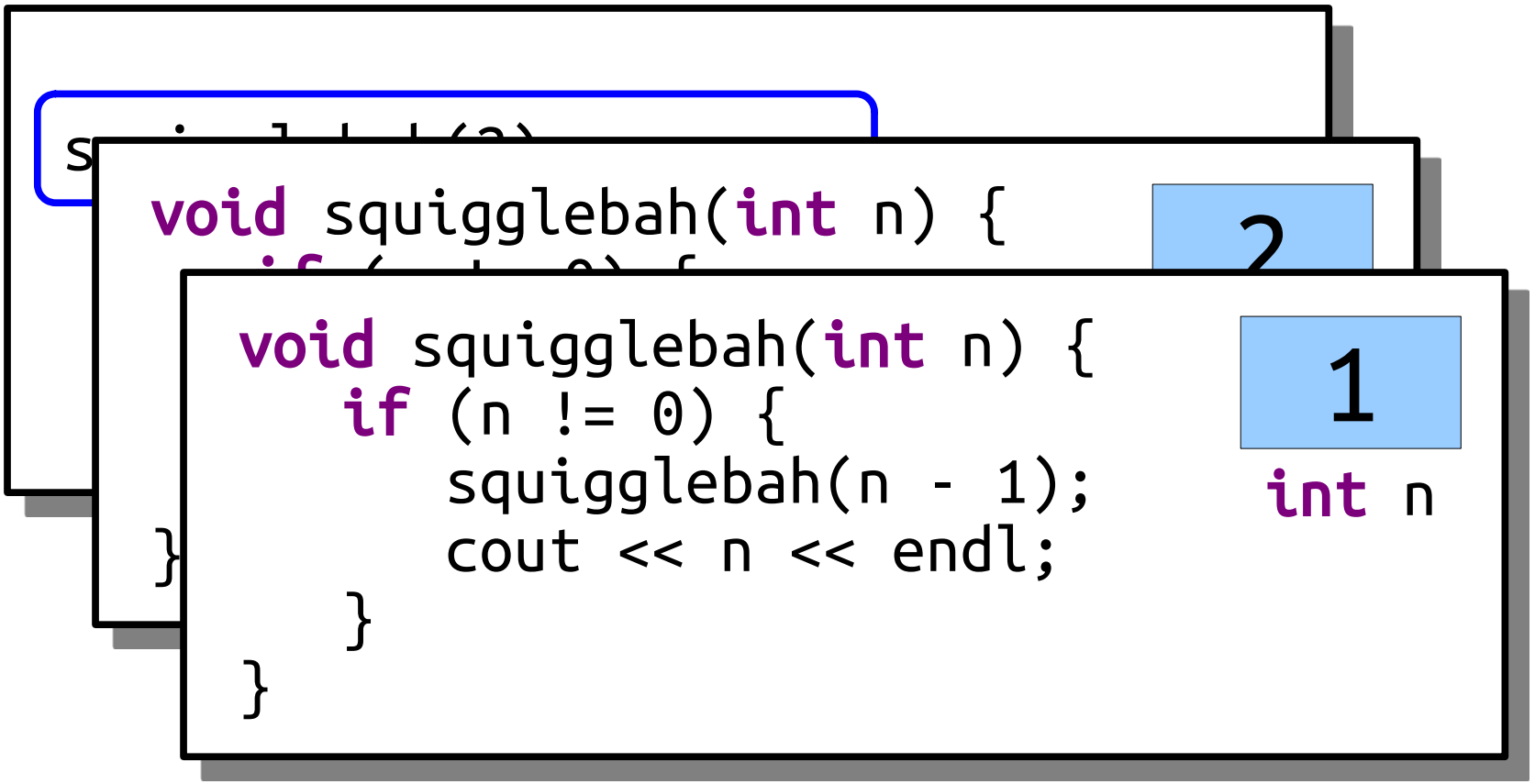
Answer at
https://pollev.com/cs106bwin23

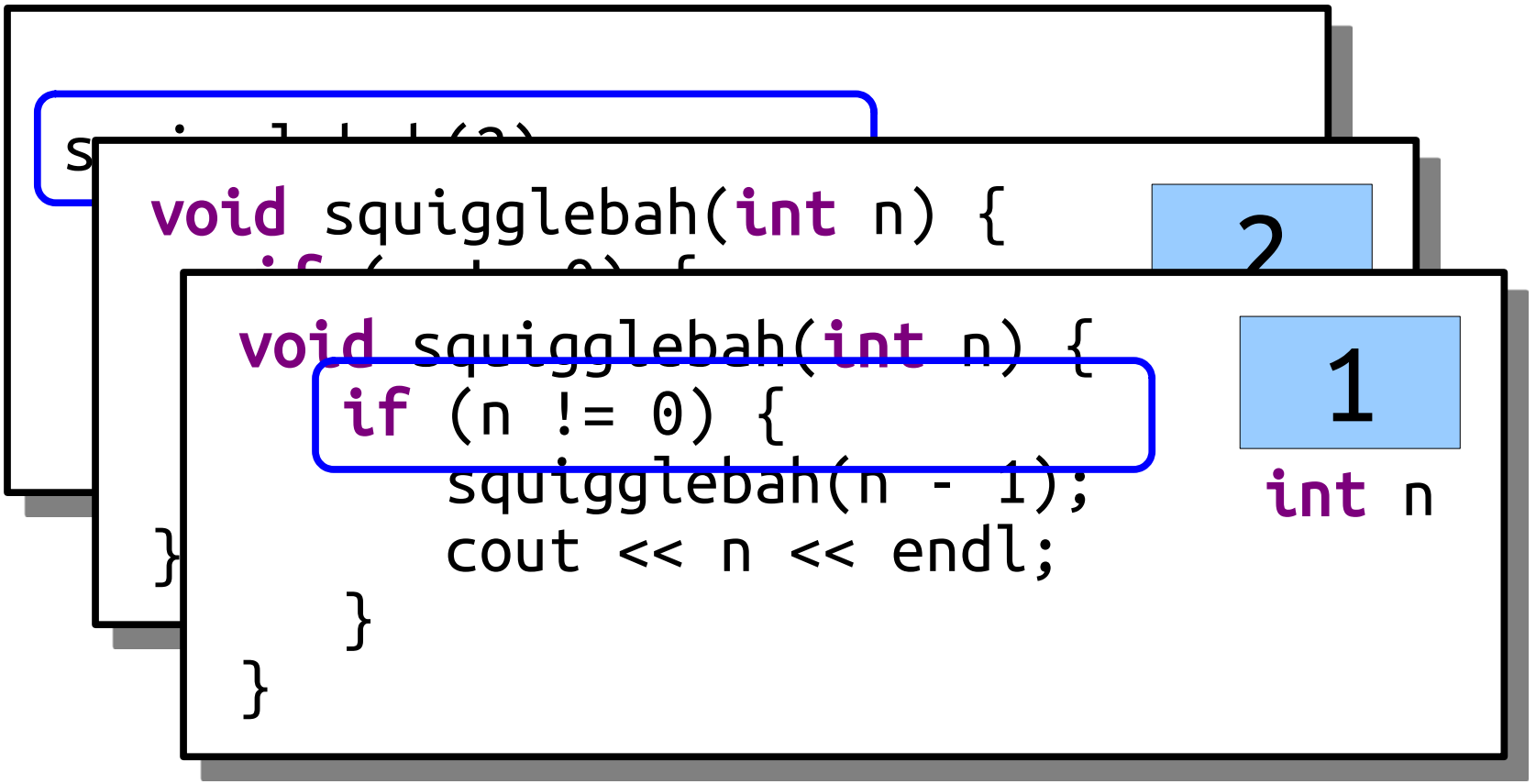
```
squigglebah(2);
```

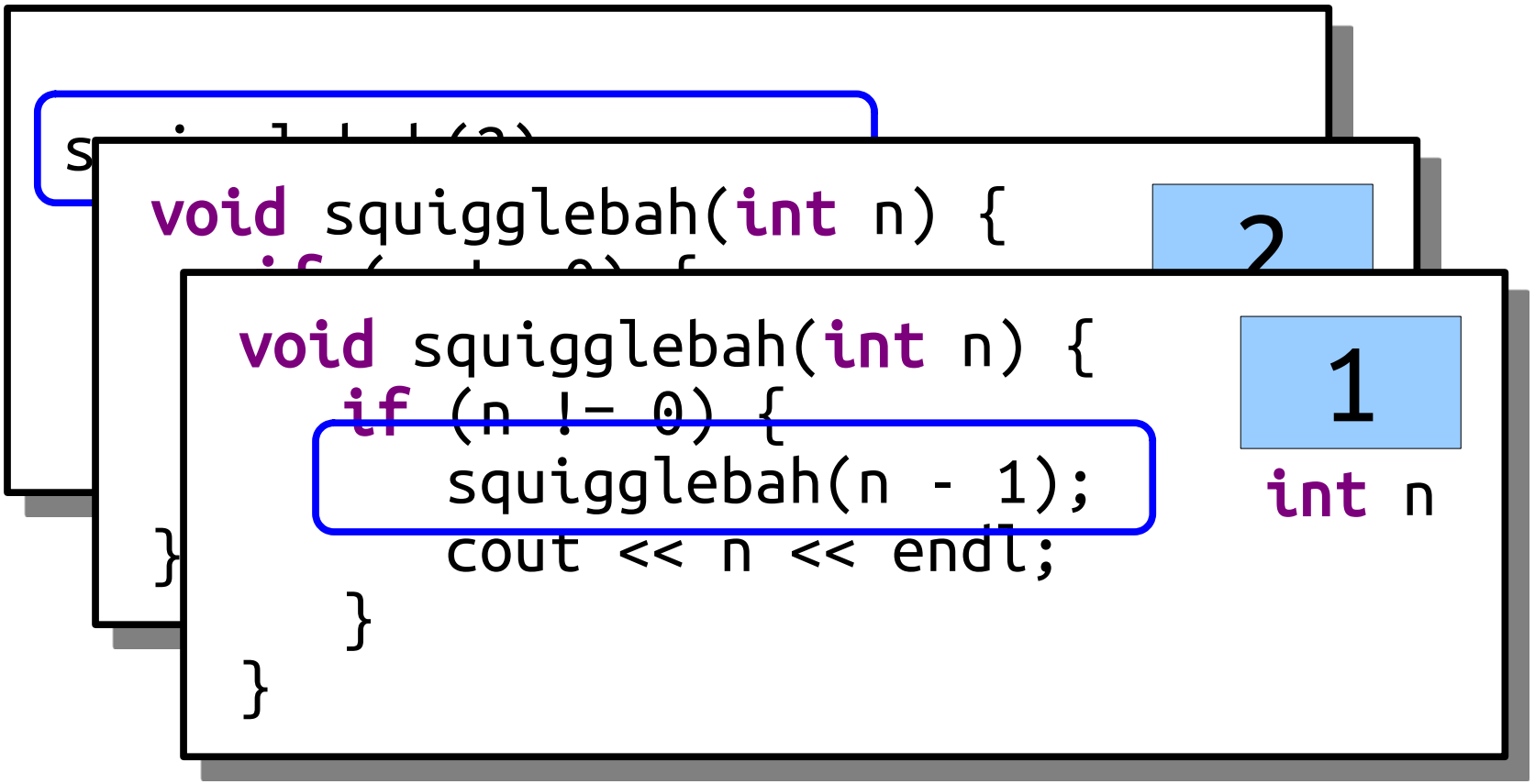

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n

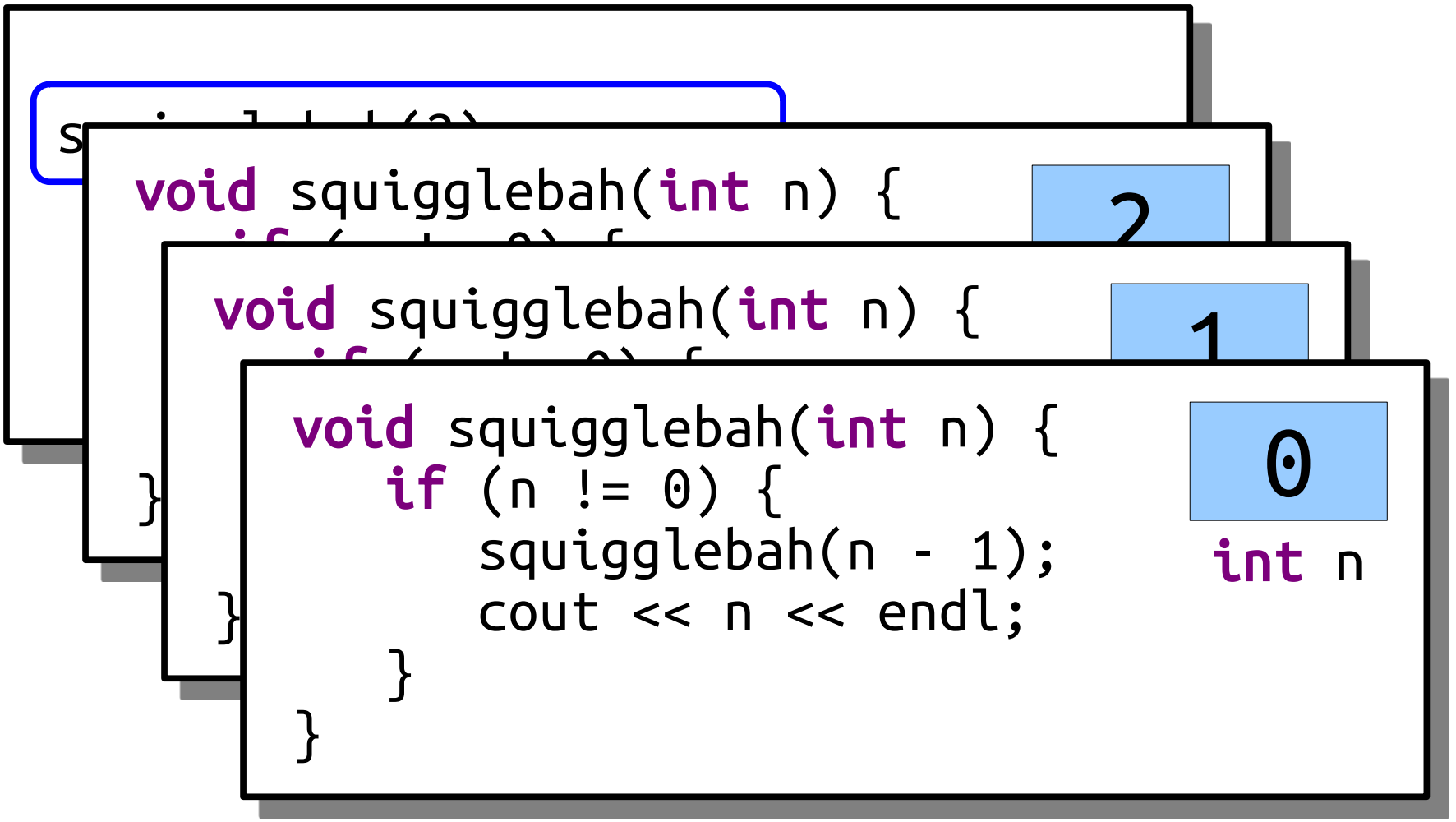

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n

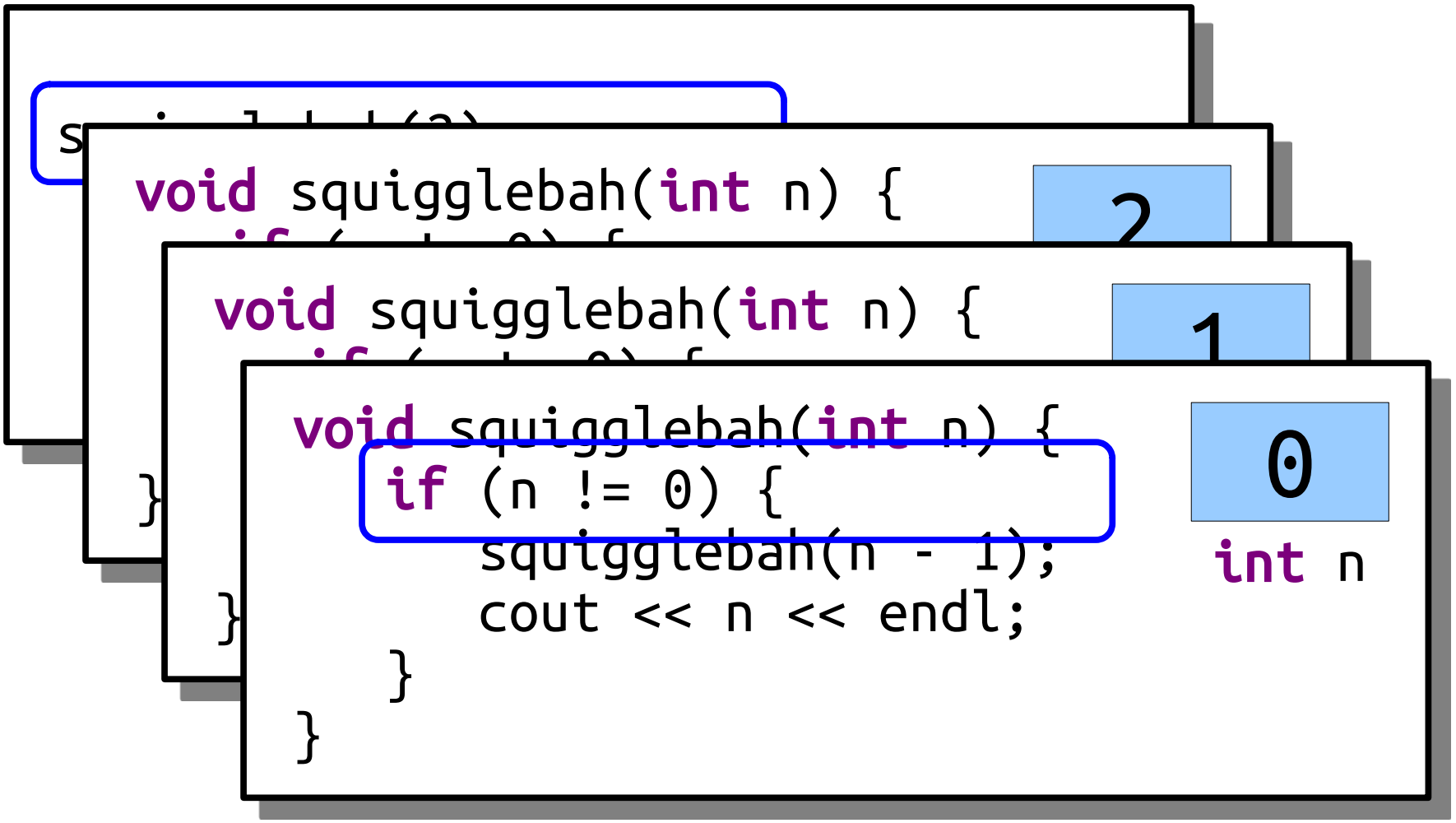

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n

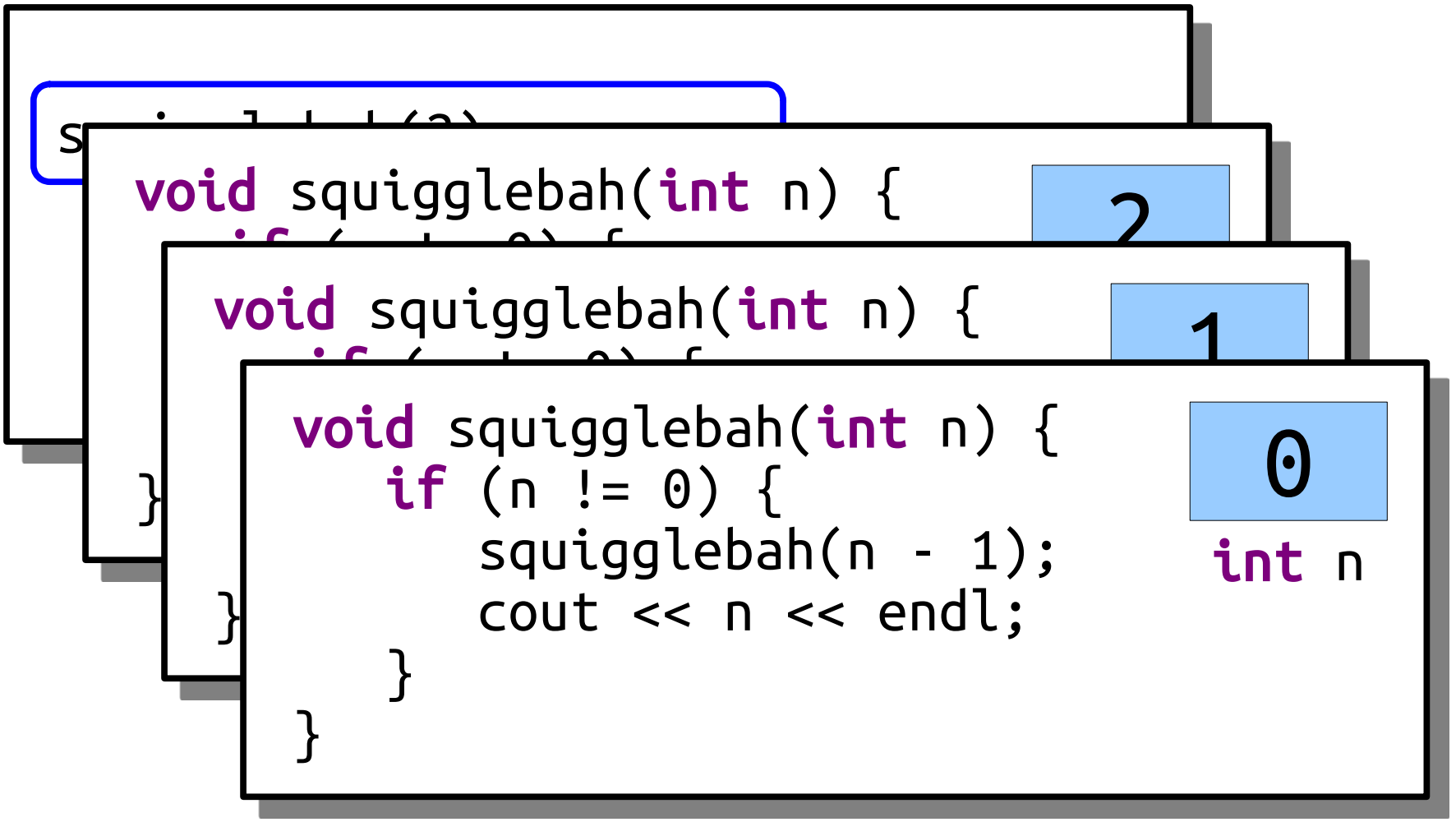

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

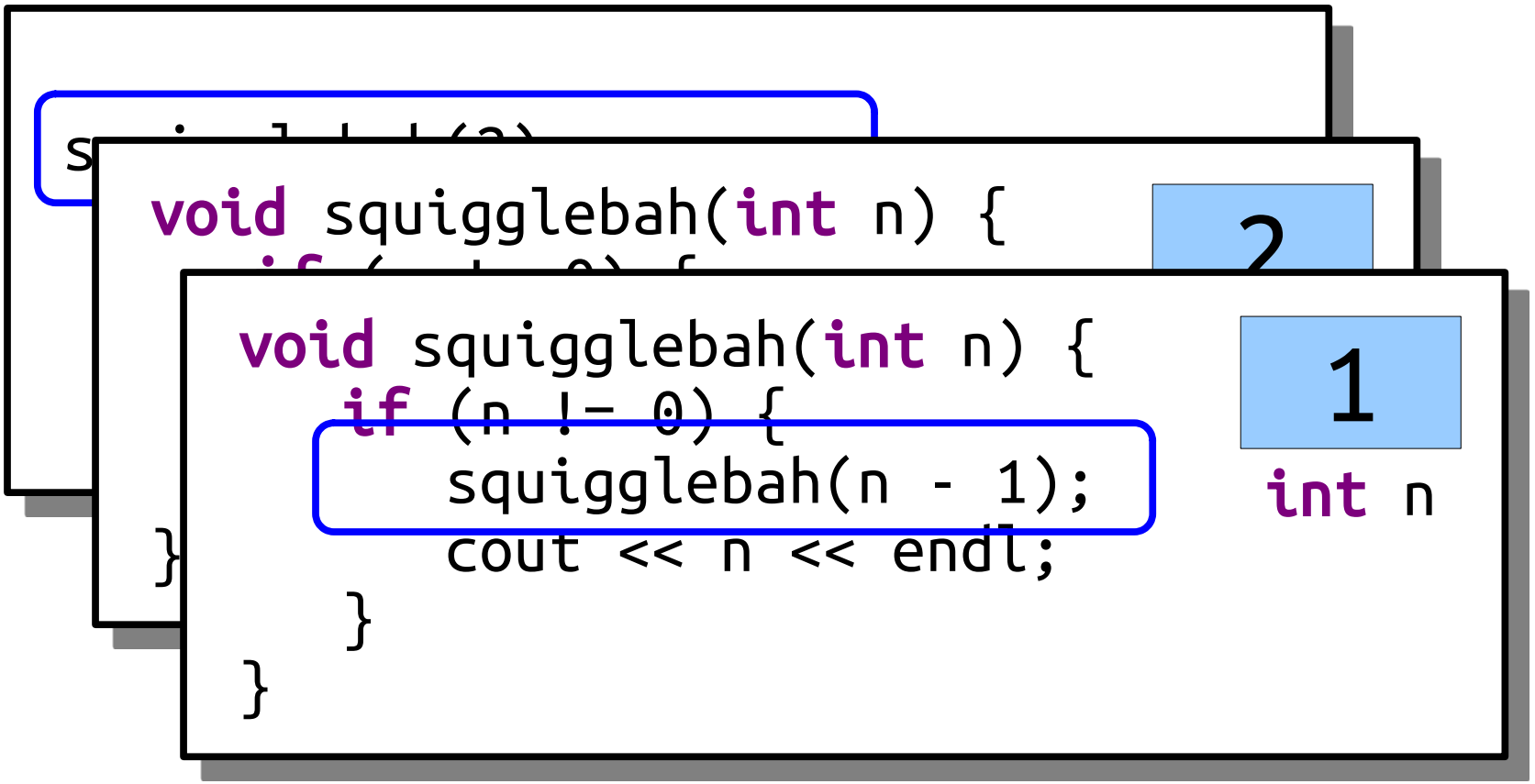

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

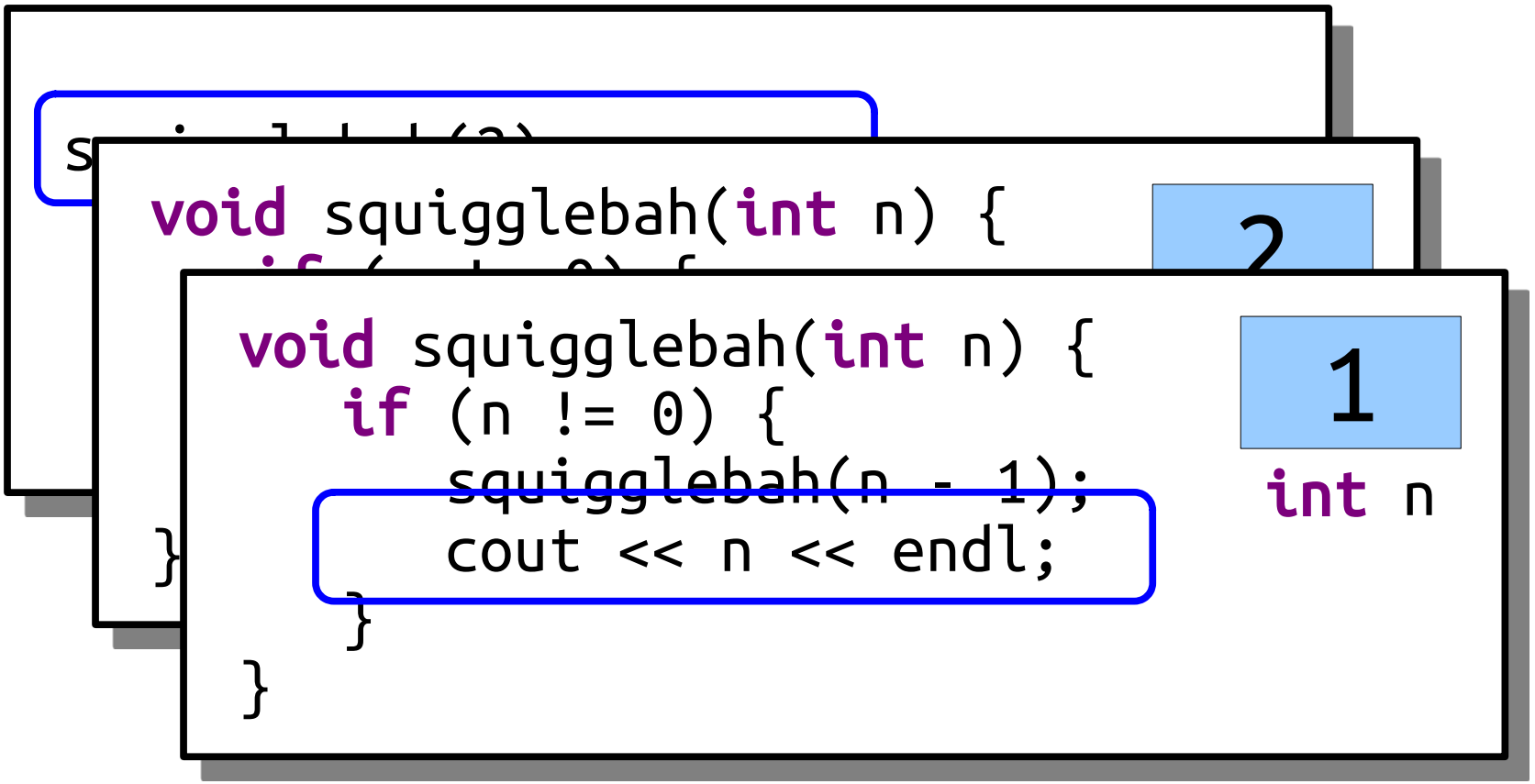

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

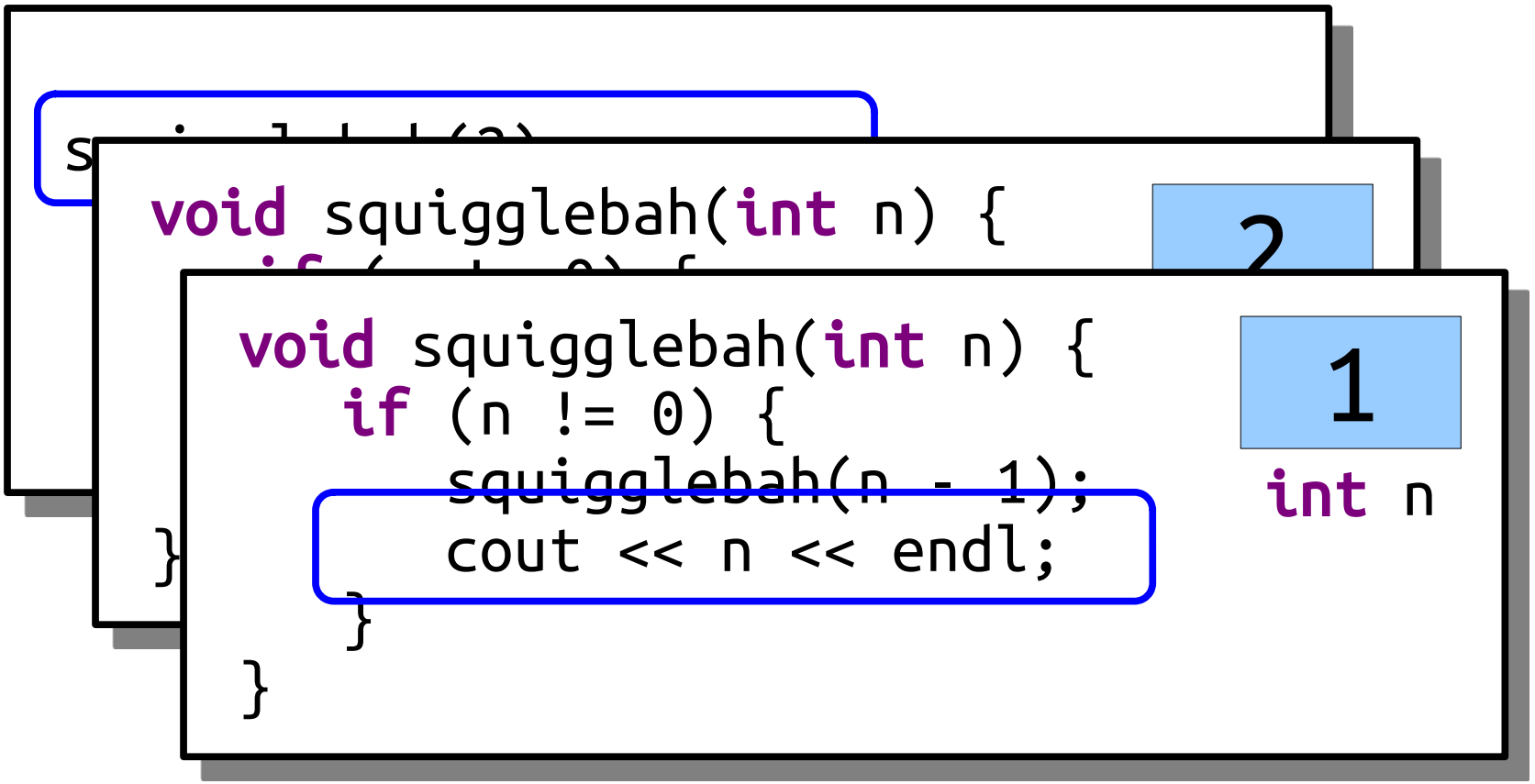

void squigglebah(int n) {
2
void squigglebah(int n) {
1
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
0
int n

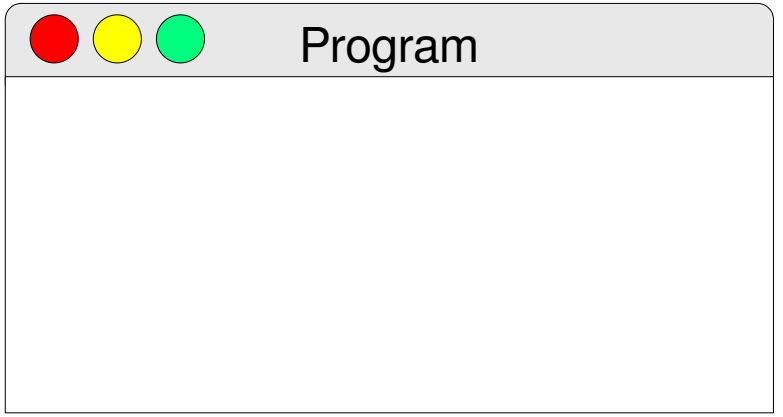

void squigglebah(int n) {
2
void squigglebah(int n) {
1
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
0
int n

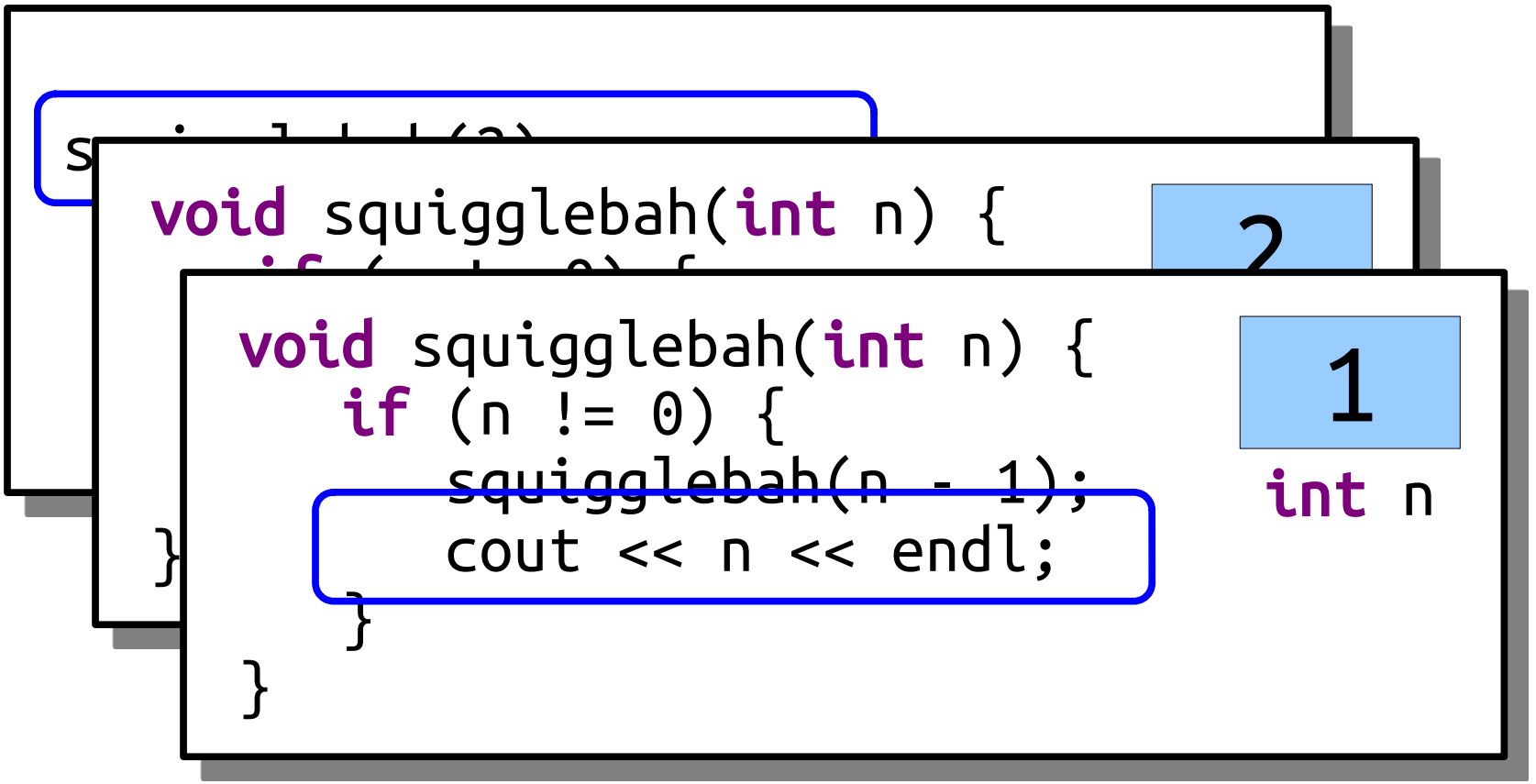

void squigglebah(int n) {
2
void squigglebah(int n) {
1
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
0
int n

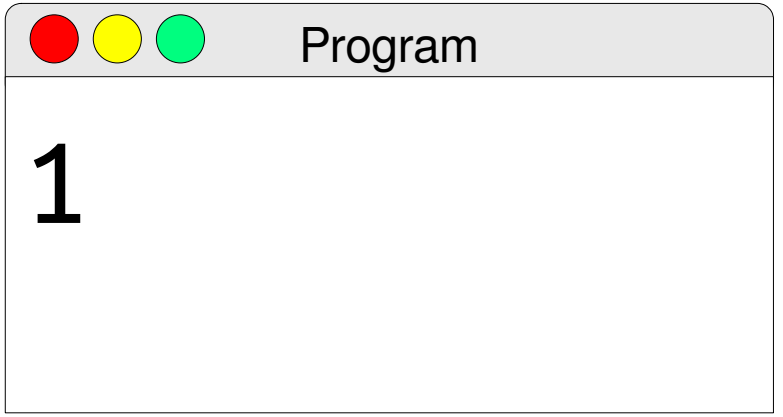

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

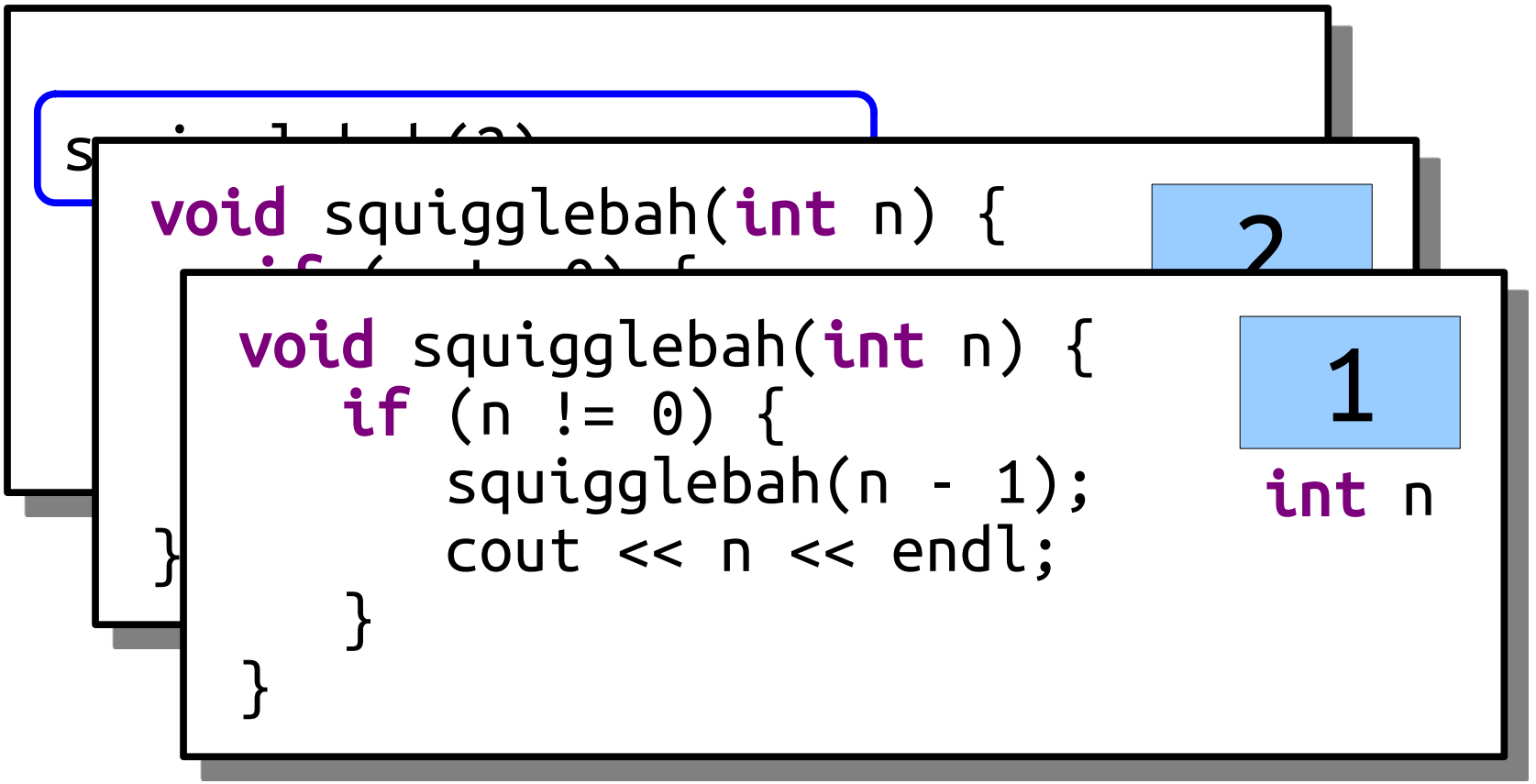

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

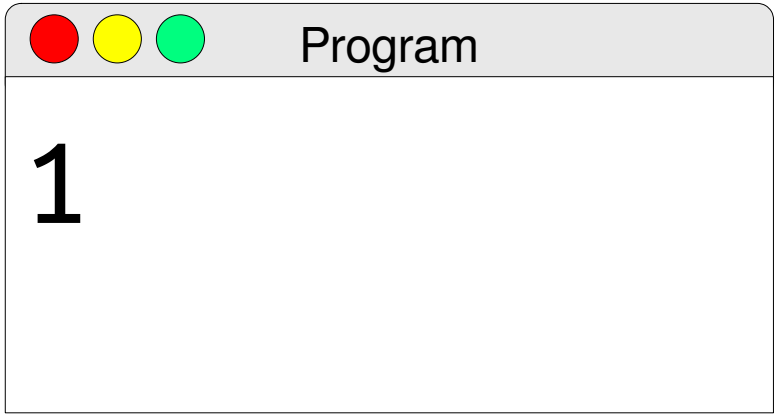

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

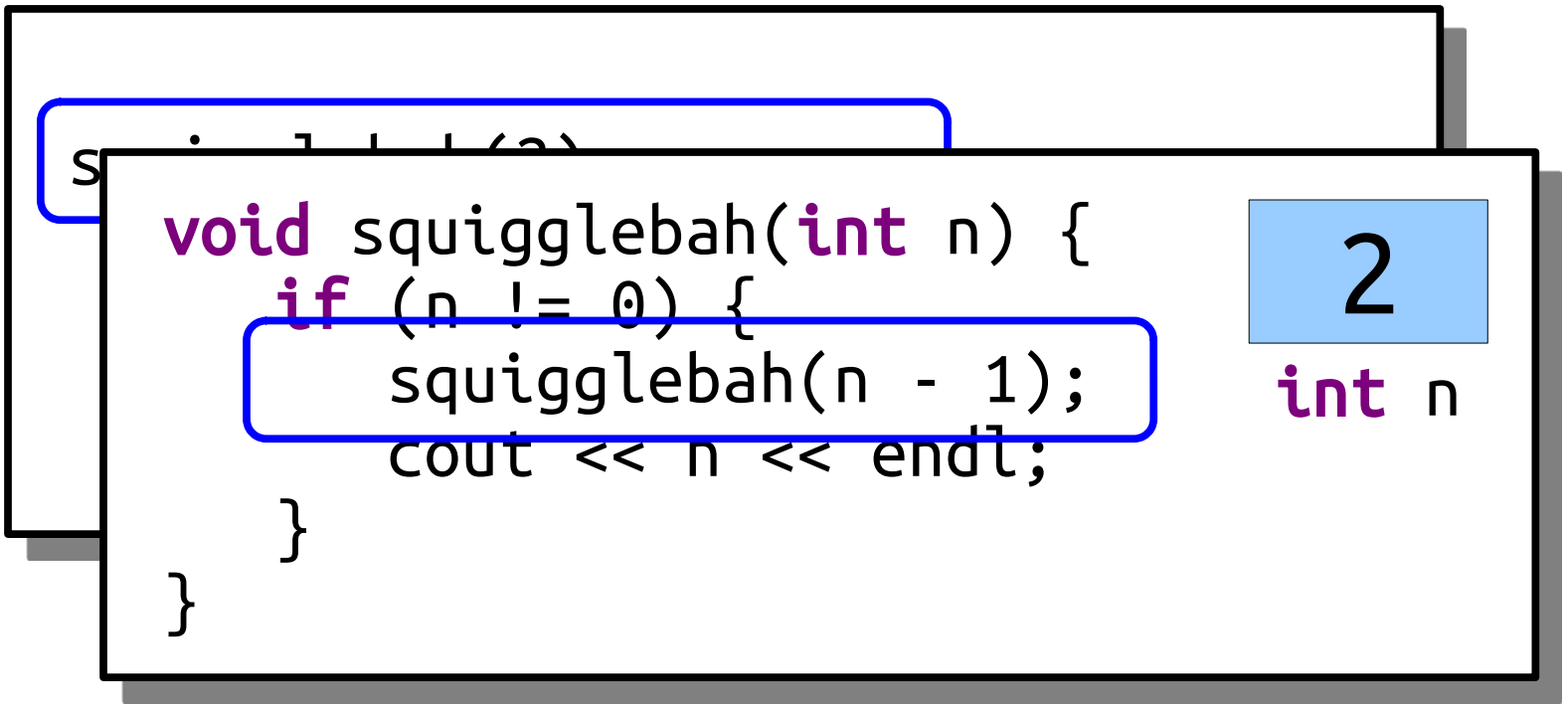

Program

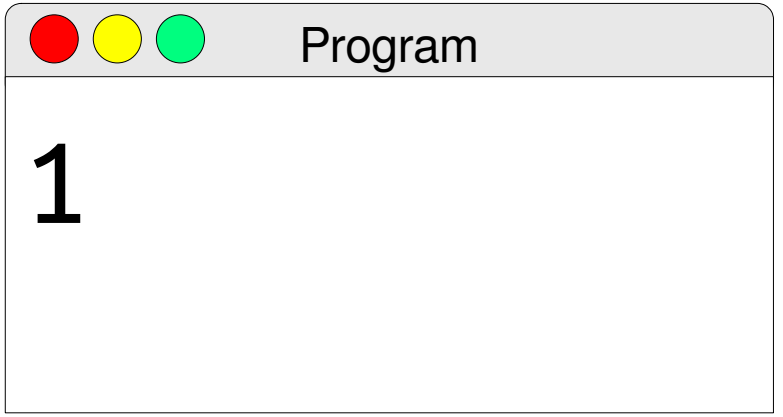

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

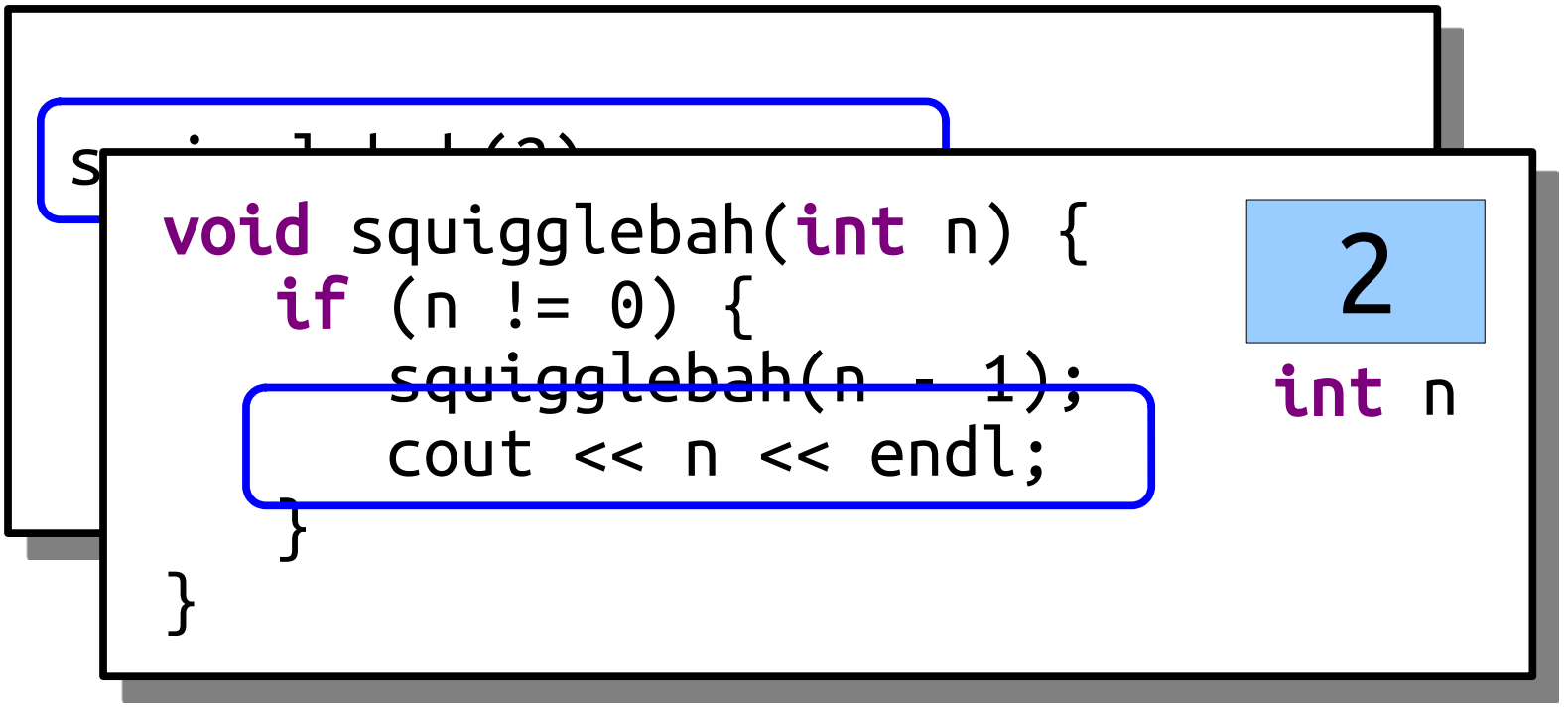

Program
1

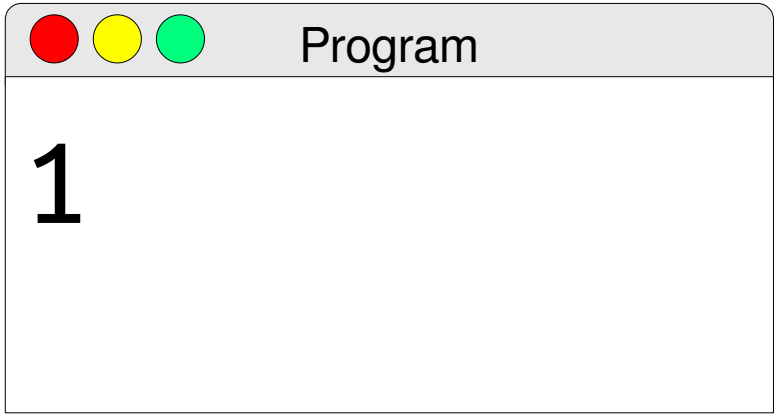

void squigglebah(int n) {
2
void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
1
int n

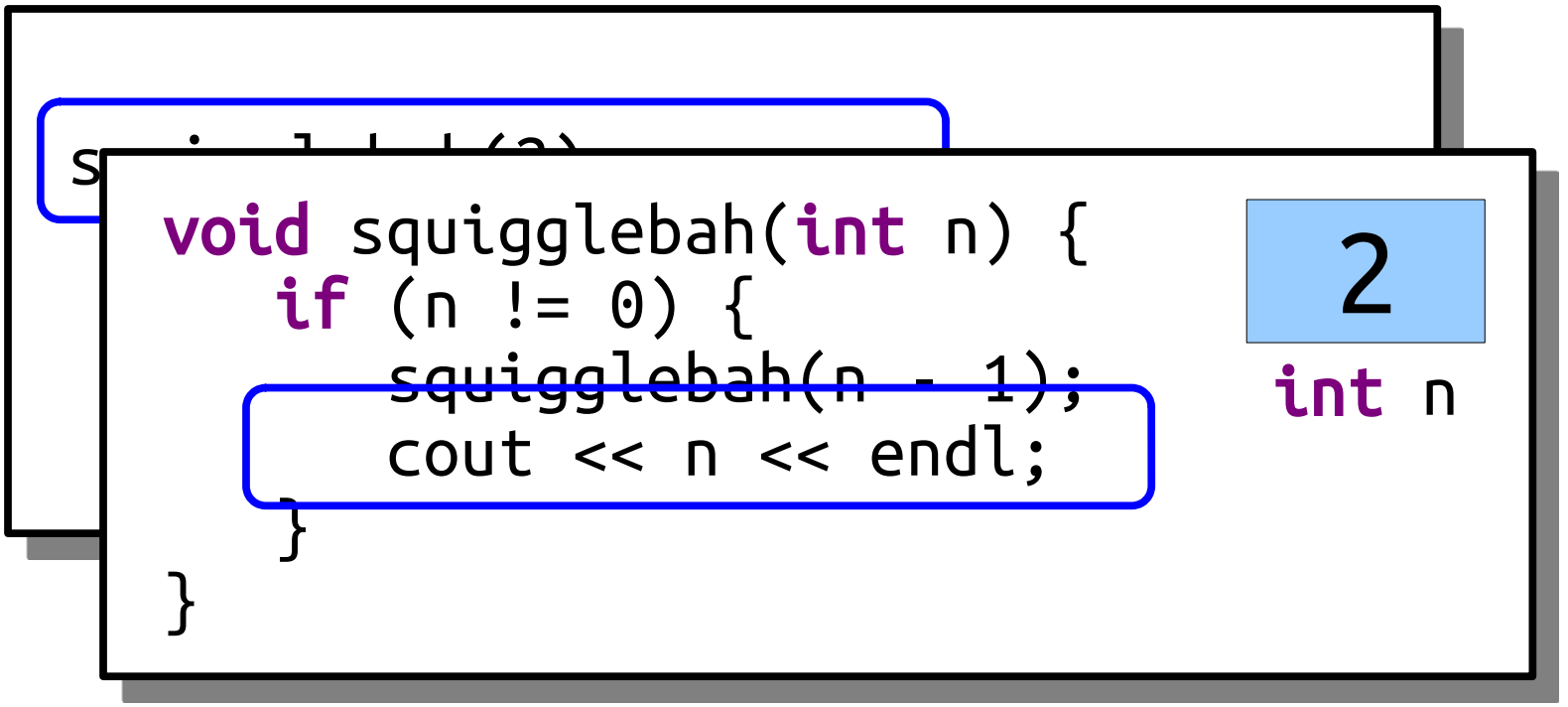

Program
1

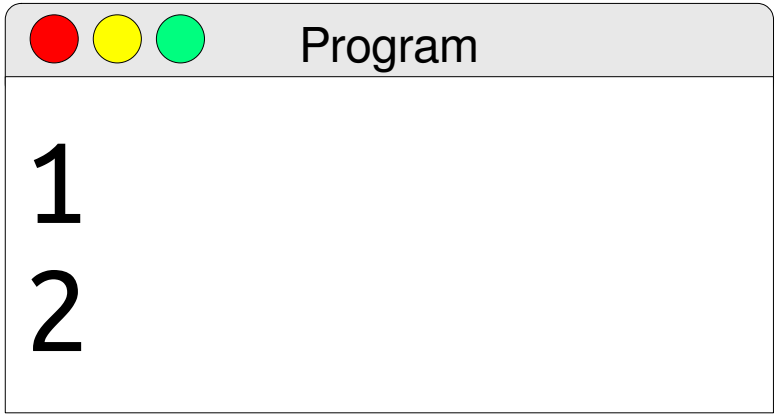

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n

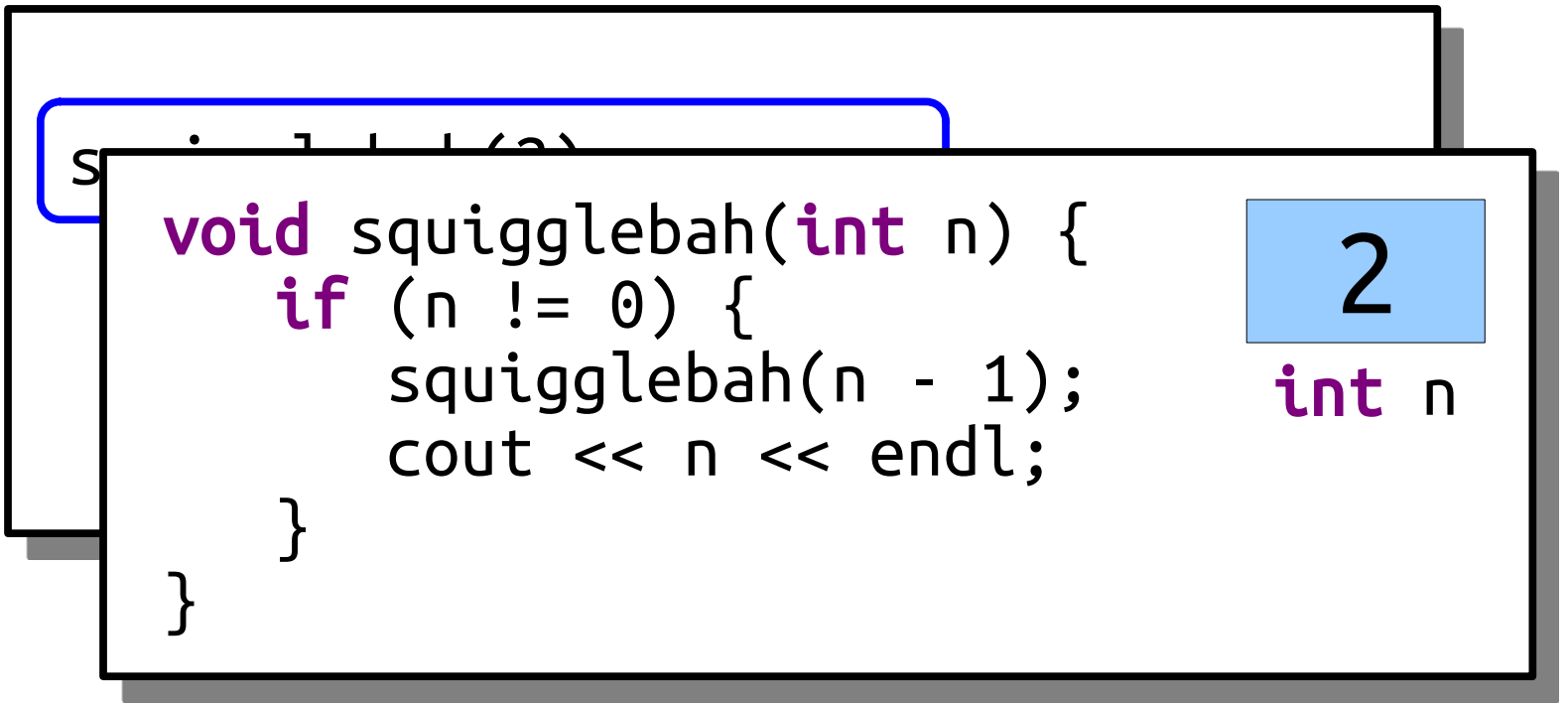

Program
1

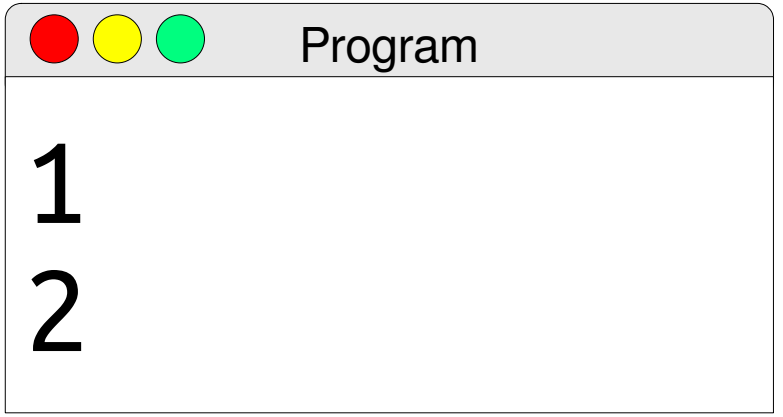

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n


Program
1

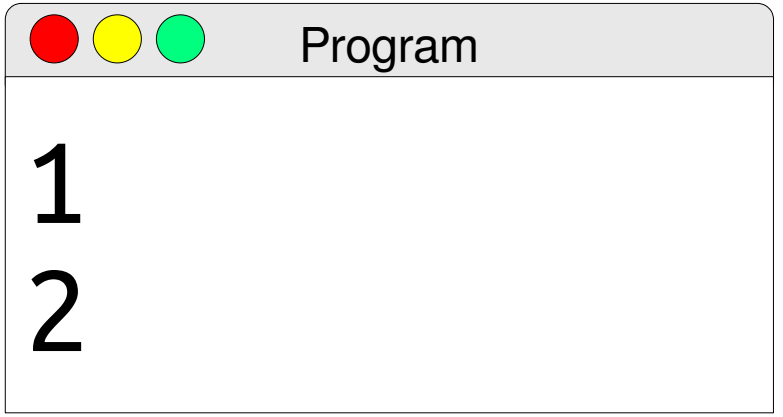

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n


Program
1
2

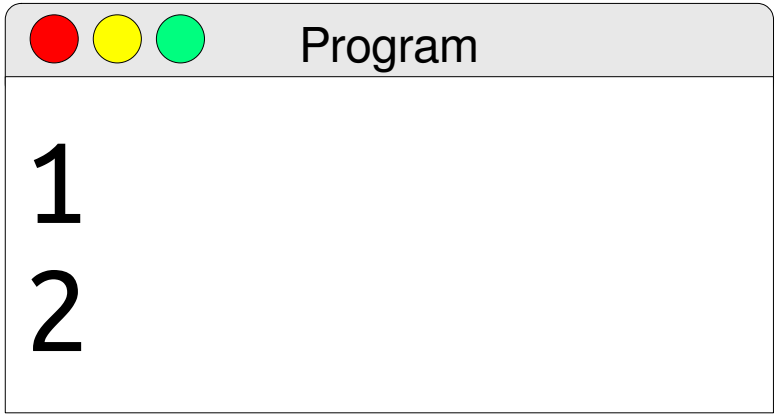

void squigglebah(int n) {
if (n != 0) {
squigglebah(n - 1);
cout << n << endl;
}
}
2
int n


Program
1
2

```
squigglebah(2);
```



```
squigglebah(2);
```

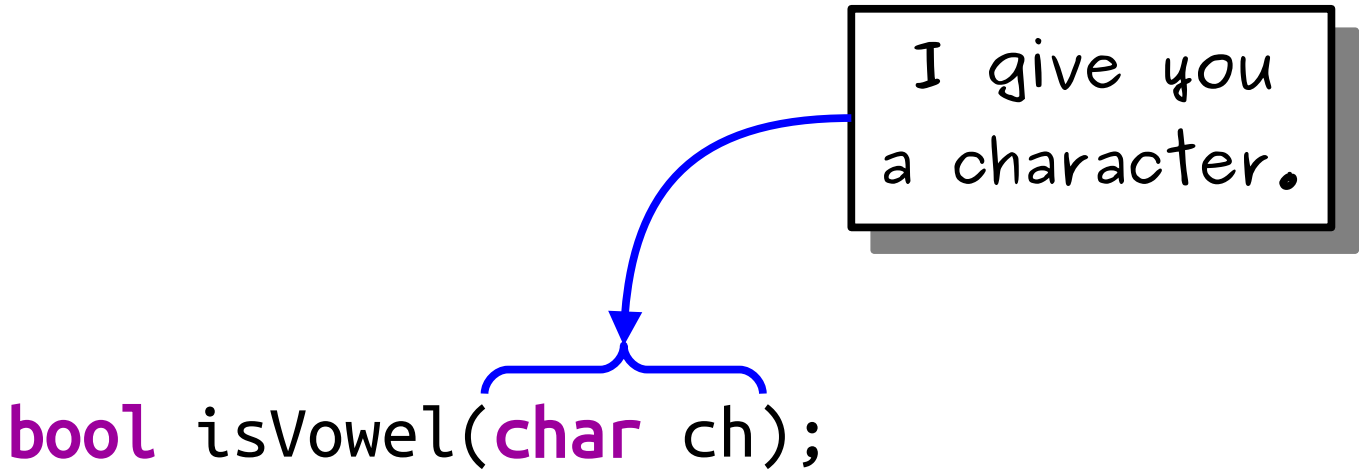


The Recursive Leap of Faith

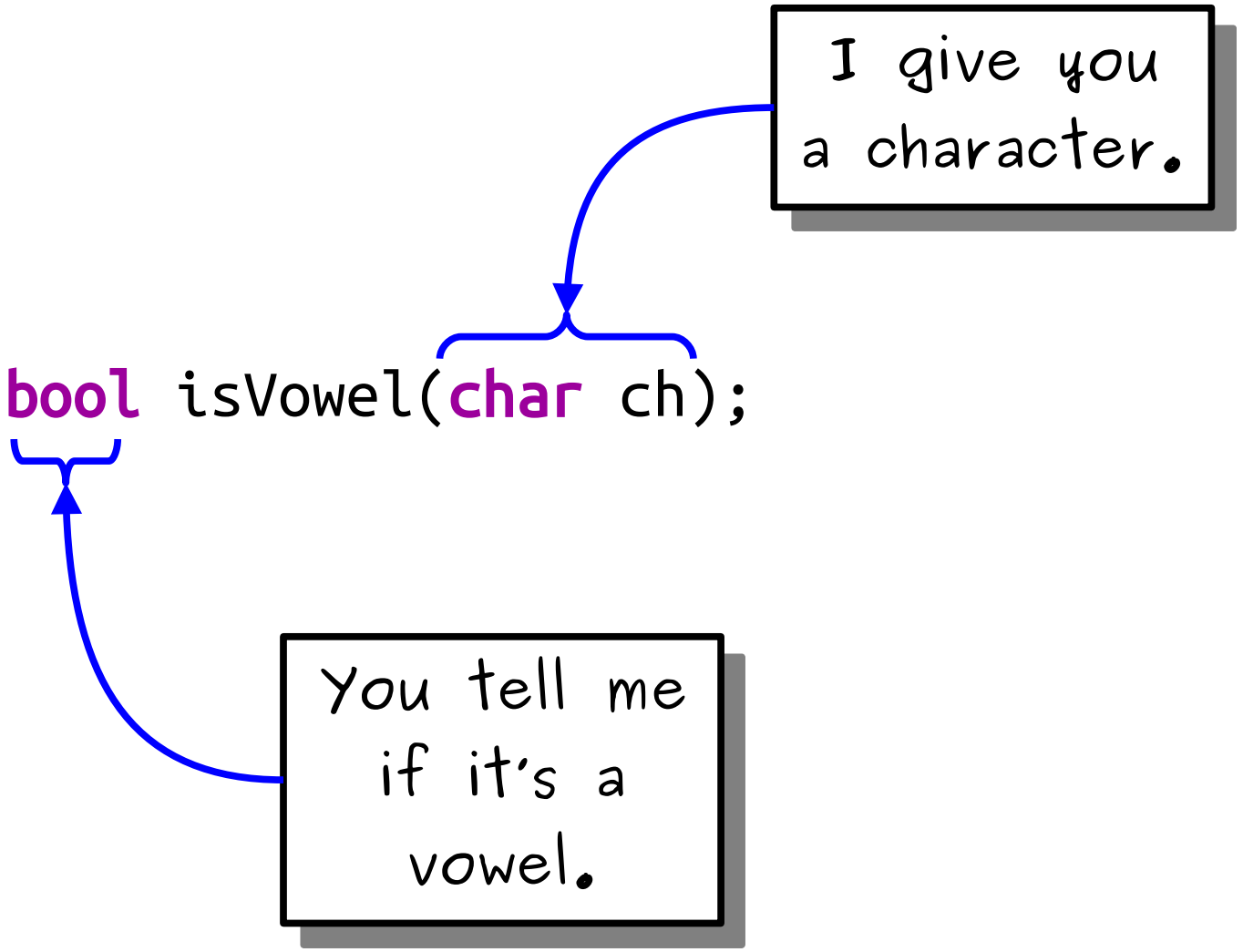
The Contract

```
bool isVowel(char ch);
```

The Contract



The Contract



The Contract

```
bool isVowel(char ch) {
    ch = toLowerCase(ch);
    return ch == 'a' ||
           ch == 'e' ||
           ch == 'i' ||
           ch == 'o' ||
           ch == 'u';
}
```

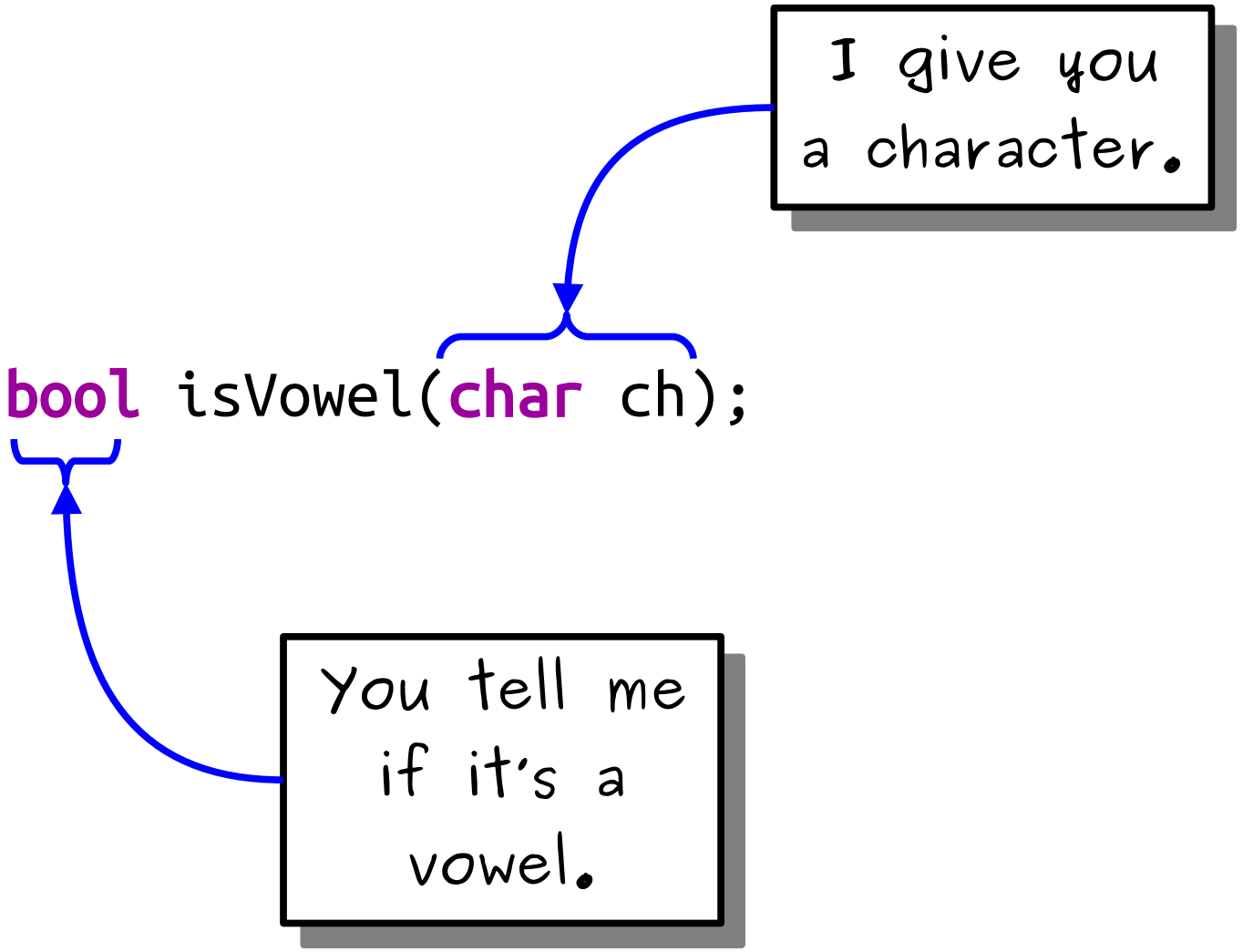
The Contract

```
bool isVowel(char ch) {
    switch(ch) {
        case 'A': case 'a':
        case 'E': case 'e':
        case 'I': case 'i':
        case 'O': case 'o':
        case 'U': case 'u':
            return true;
        default:
            return false;
    }
}
```

The Contract

```
bool isVowel(char ch) {
    ch = tolower(ch);
    return string("aeiou").find(ch) != string::npos;
}
```

The Contract

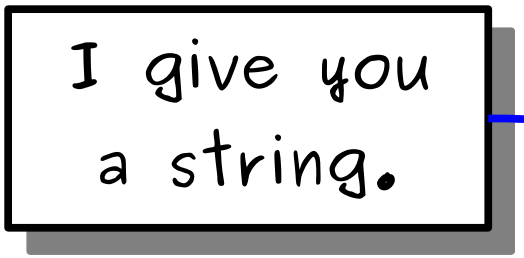


The Contract

The Contract

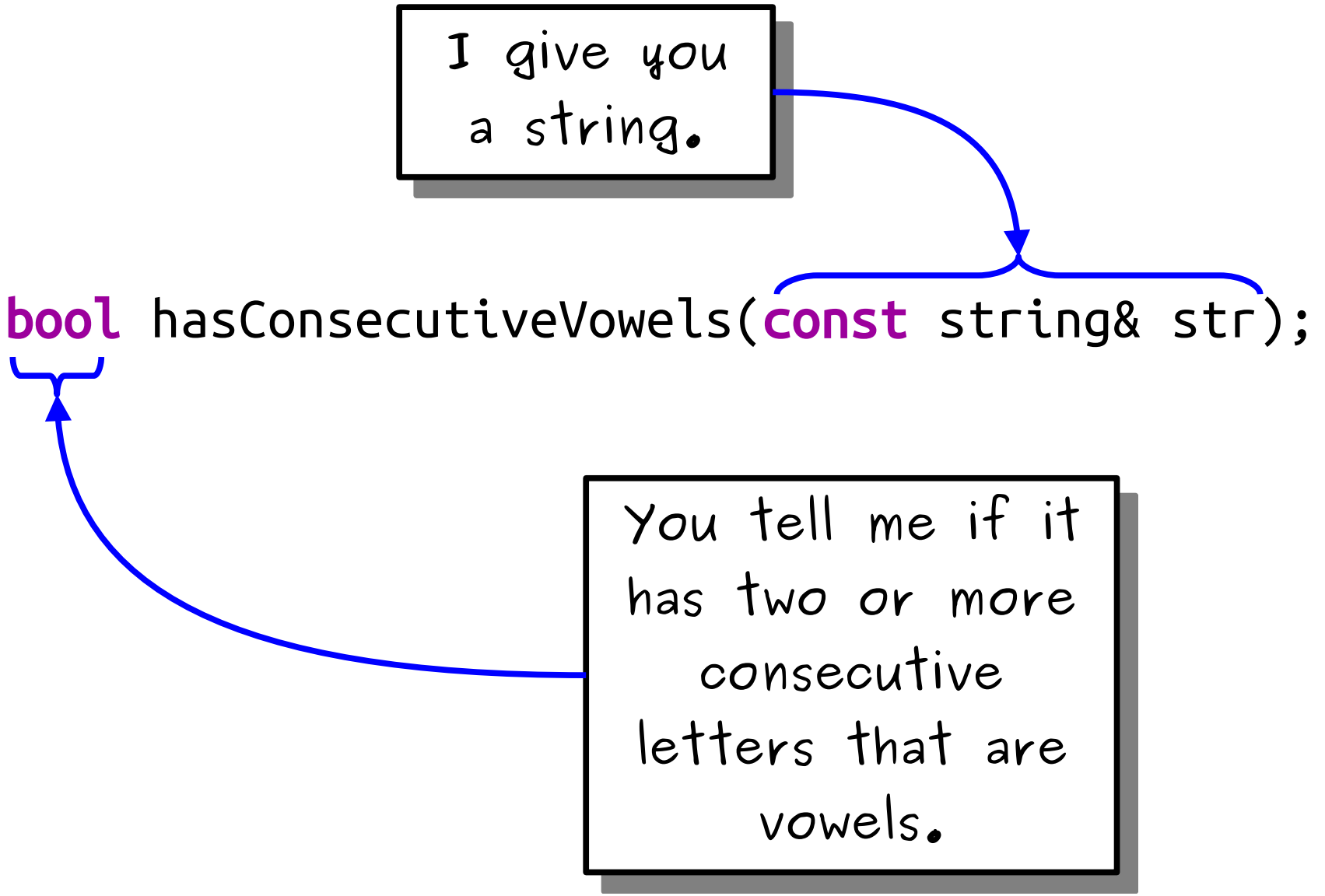
```
bool hasConsecutiveVowels(const string& str);
```

The Contract




```
bool hasConsecutiveVowels(const string& str);
```

The Contract



Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {




}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {



  }

}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (str[i - 1] is a vowel && str[i] is a vowel) {
       return true;
    }
  }

}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (str[i - 1] is a vowel && str[i] is a vowel) {
       return true;
    }
  }
  return false;
}
```

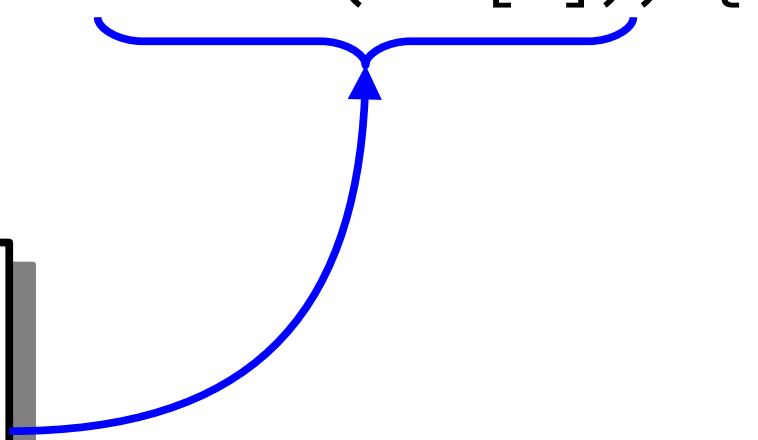
Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (isVowel(str[i - 1]) && isVowel(str[i])) {
       return true;
    }
  }
  return false;
}
```

Trusting the Contract

```
bool isVowel(char ch);

bool hasConsecutiveVowels(const string& str) {
  for (int i = 1; i < str.length(); i++) {
    if (isVowel(str[i - 1]) && isVowel(str[i])) {
       return true;
    }
  }
  return
}
```

It doesn't matter how isVowel is implemented. We just trust that it works.

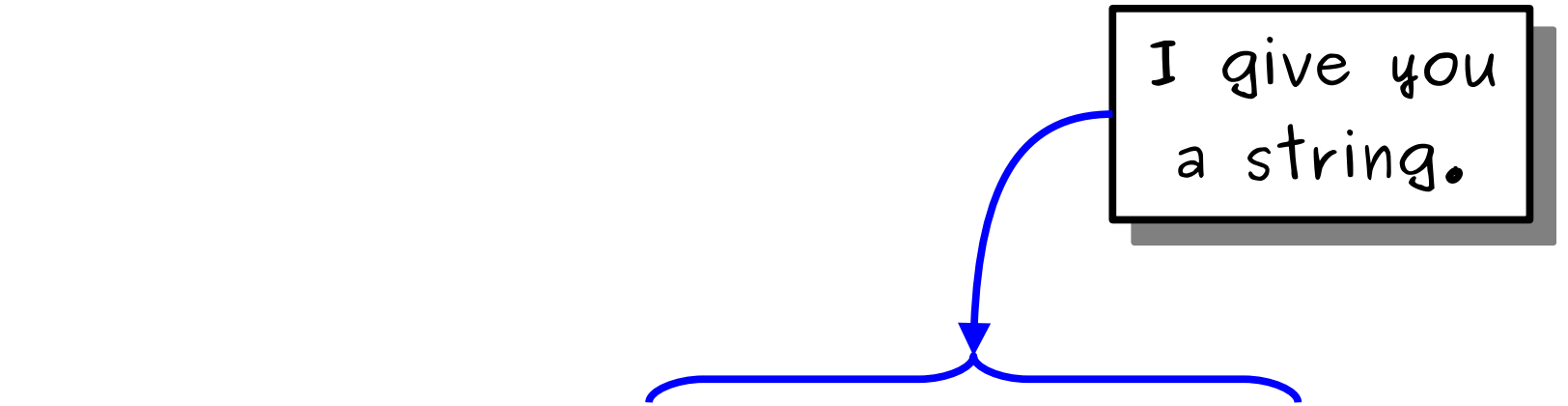


The Contract

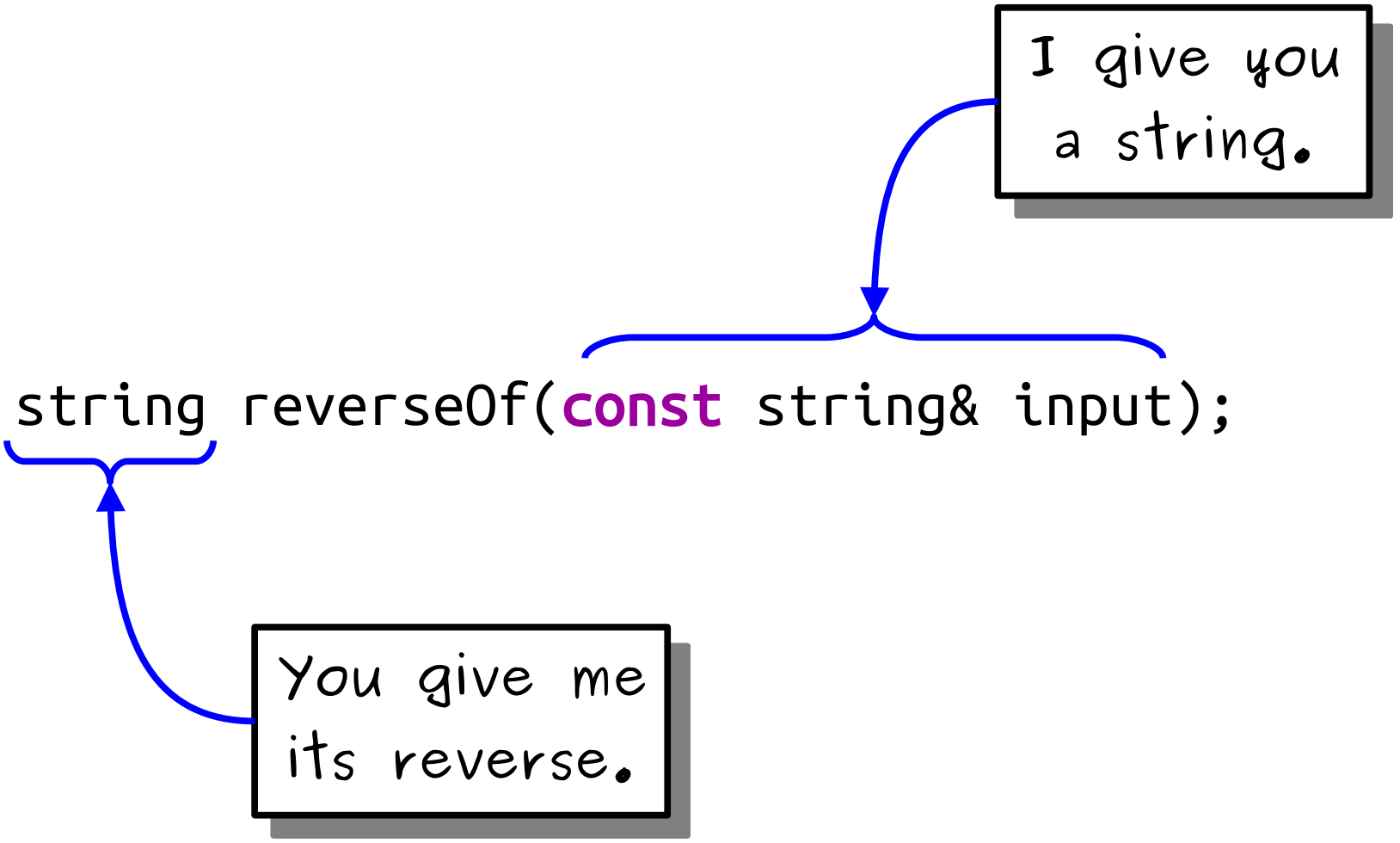
The Contract

```
string reverseOf(const string& input);
```

The Contract



The Contract



Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {



}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {

    } else {

    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {

    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return the reverse of input.substr(1) + input[0];
    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```

Trusting the Contract

```
string reverseOf(const string& input);

string reverseOf(const string& input) {
    if (input == "") {
        return "";
    } else {
        return reverseOf(input.substr(1)) + input[0];
    }
}
```

It doesn't matter how reverseOf reverses the string. It just matters that it does.

The Contract

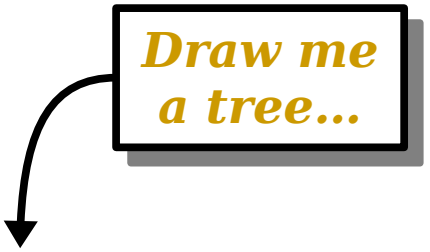
The Contract

```
void drawTree(double x, double y,
              double height,
              double angle,
              int order);
```

The Contract

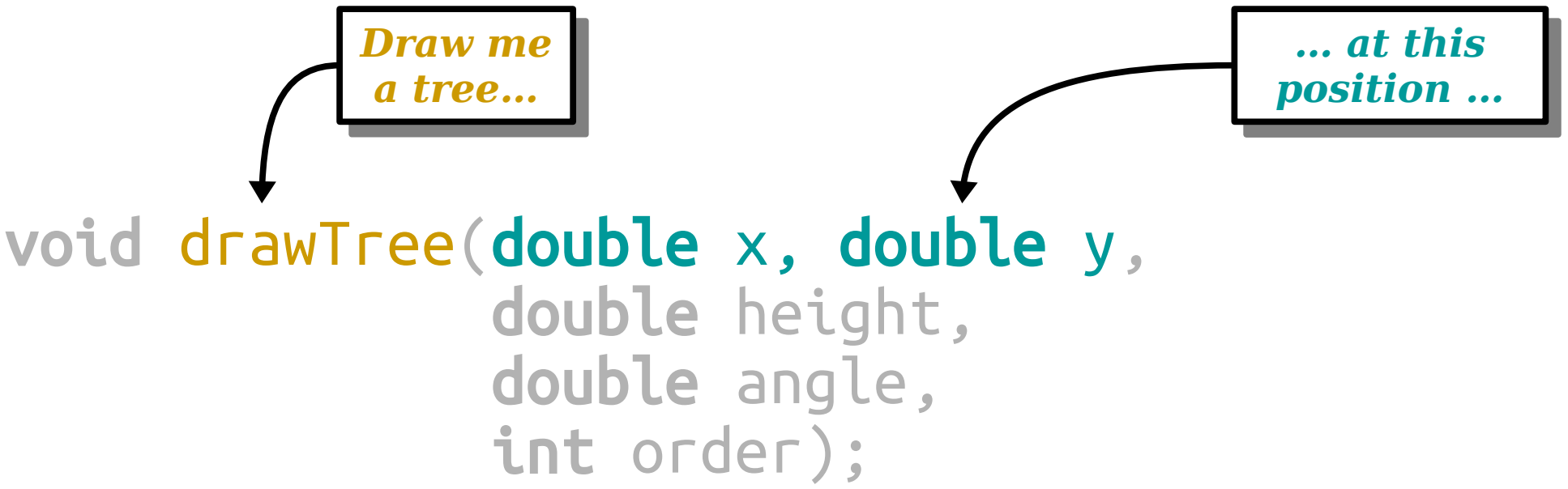
```
void drawTree(double x, double y,
              double height,
              double angle,
              int order);
```

The Contract

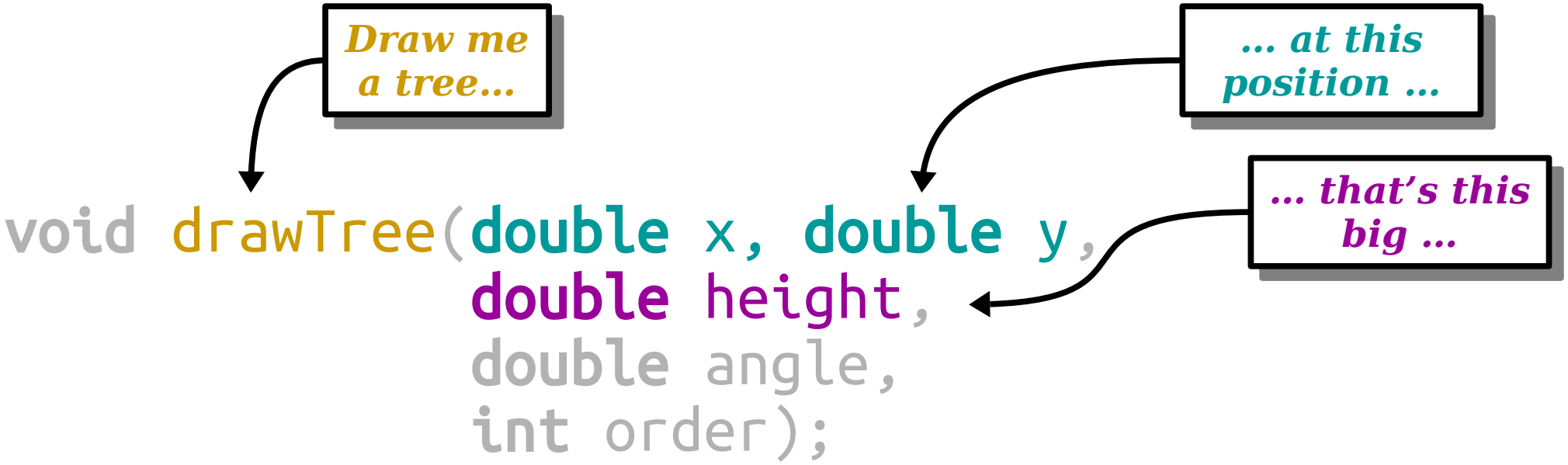



```
void drawTree(double x, double y,
              double height,
              double angle,
              int order);
```

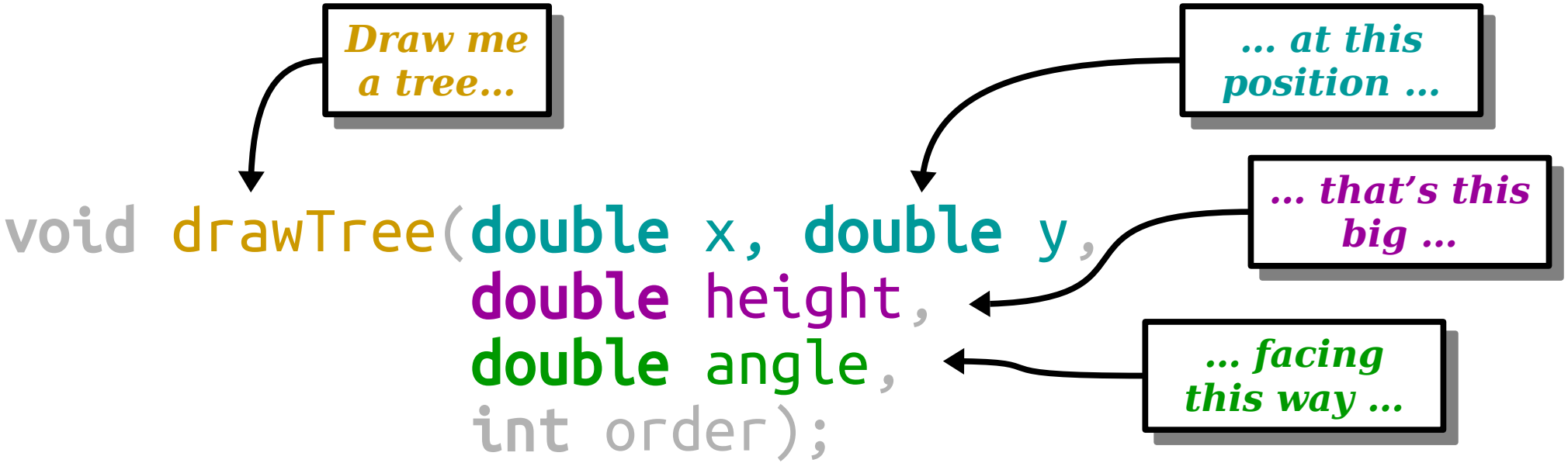
The Contract



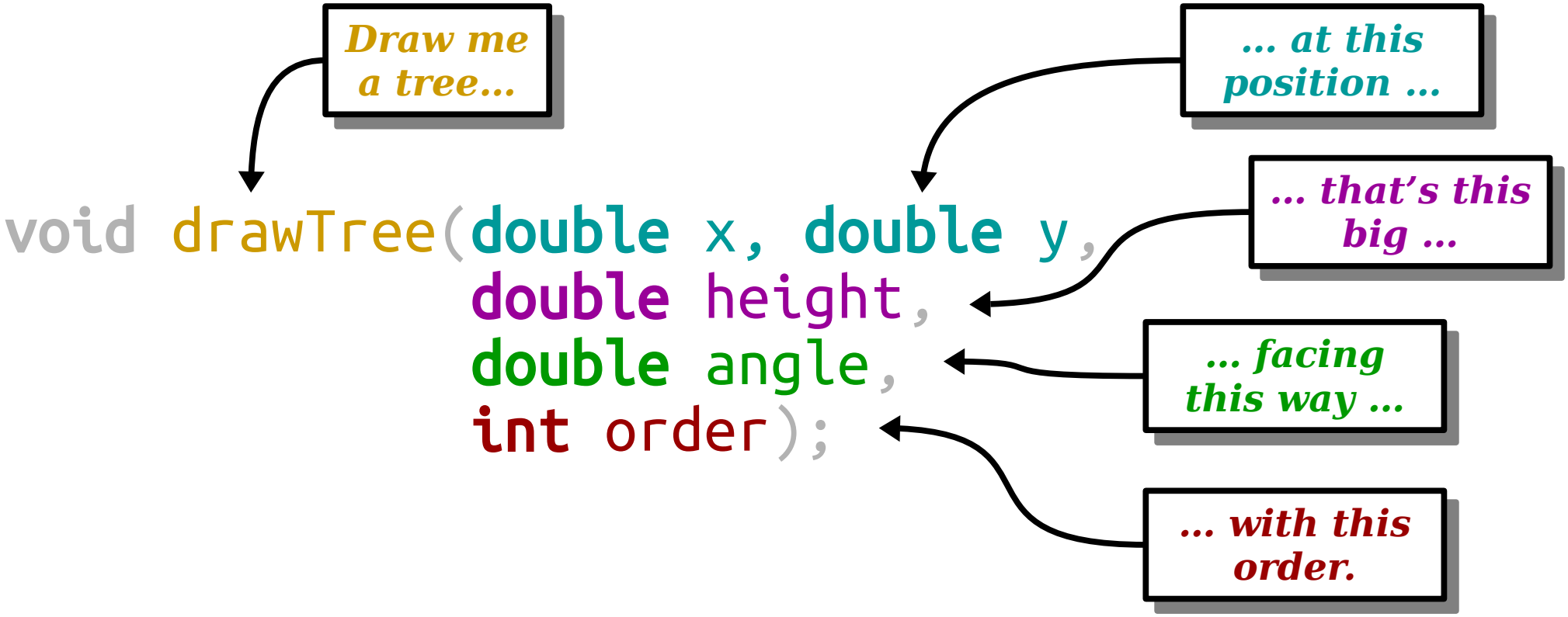
The Contract



The Contract



The Contract



Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, double angle,
              int order) {









}
```

Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;




}
```

Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);



}
```

Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    draw a tree angling to the left
    draw a tree angling to the right
}
```

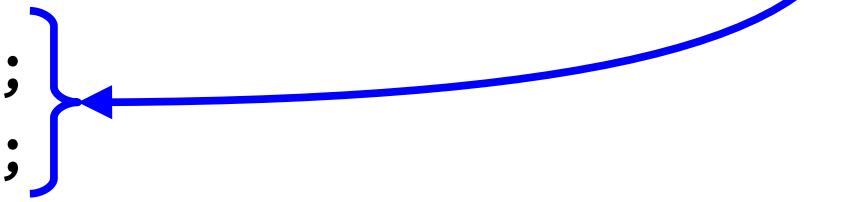
Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, double angle,
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    drawTree(/* … */);
    drawTree(/* … */);
}
```

Trusting the Contract

```
void drawTree(double x, double y,
              double height, double angle,
              int order);

void drawTree(double x, double y,
              double height, doubl
              int order) {
    if (order == 0) return;

    GPoint endpoint = drawPolarLine(/* … */);

    drawTree(/* … */);
    drawTree(/* … */);
}
```

It doesn't matter how drawTree draws a tree. It just matters that it does.



The Recursive Leap of Faith

- When writing a recursive function, it helps to take a ***recursive leap of faith***.
- Before writing the function, answer these questions:
  - What does the function take in?
  - What does it return?
- Then, as you're writing the function, trust that your recursive calls to the function just "work" without asking how.
- This can take some adjustment to get used to, but is a necessary skill for writing more complex recursive functions.

Time-Out for Announcements!

Recursive Drawing Contest

- We are holding a (purely optional, just for fun) Recursive Drawing contest!
- Visit ***http://recursivedrawing.com/***, draw whatever you'd like, and post it to the EdStem thread for the contest.
- We'll award recursion-themed prizes to a small number of entries.
- Deadline to submit is Monday at 1:00PM Pacific.

Assignment 2

- Assignment 2 is due this Friday at 1:00PM.
  - If you’re following our timetable, you’ll have finished Rosetta Stone at this point and be midway through Rising Tides.
- Have questions?
  - Stop by the LaIR!
  - Ask on EdStem!
  - Visit our office hours!

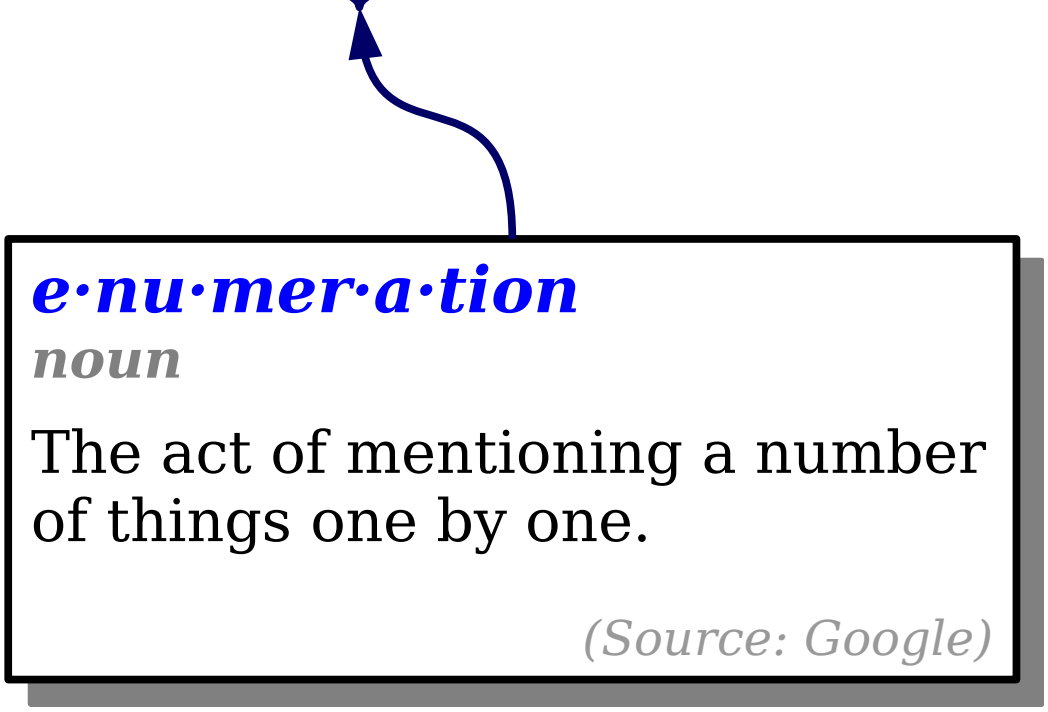
WICS FRESHMEN ICE CREAM RUN

January 28th @ 1 PM

RSVP: tinyurl.com/wics-salt-straw

Back to CS106B!

Recursive Enumeration



Listing Subsets

- A set S is a ***subset*** of a set T if every element of S is an element of T.
- There are two subsets of {2}:

  { } {2}
- There are four subsets of {2, 3}:

  { } {2} {3} {2, 3}
- How many subsets are there of {2, 3, 4}?

Answer at
https://pollev.com/cs106bwin23

Listing Subsets

- A set S is a ***subset*** of a set T if every element of S is an element of T.
- There are two subsets of {2}:

  { } {2}
- There are four subsets of {2, 3}:

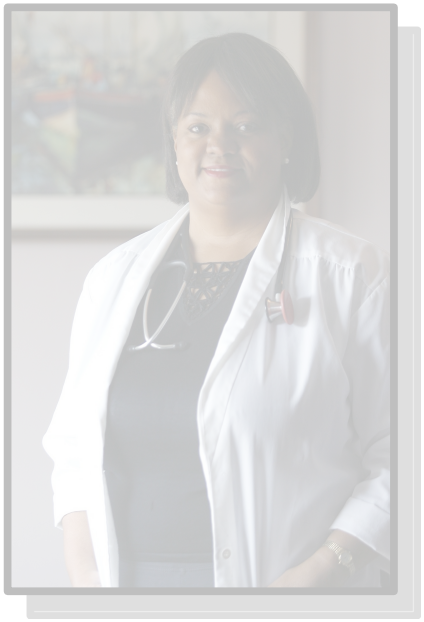
  { } {2} {3} {2, 3}
- How many subsets are there of {2, 3, 4}?

  { }
  {2} {3} {4}
  {2, 3} {2, 4} {3, 4}
  {2, 3, 4}
- The only subset of { } is { }.

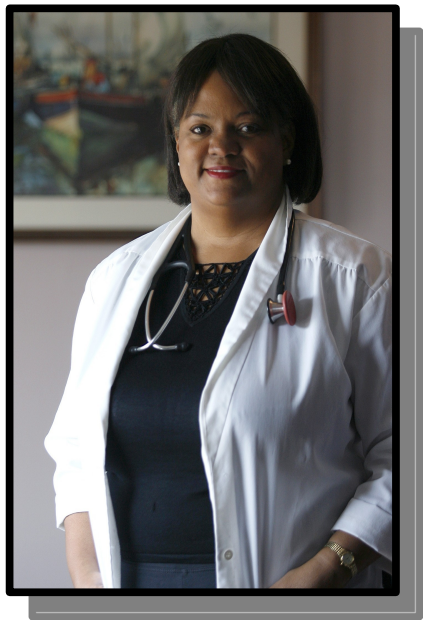
You need to send an emergency team of doctors to an area.

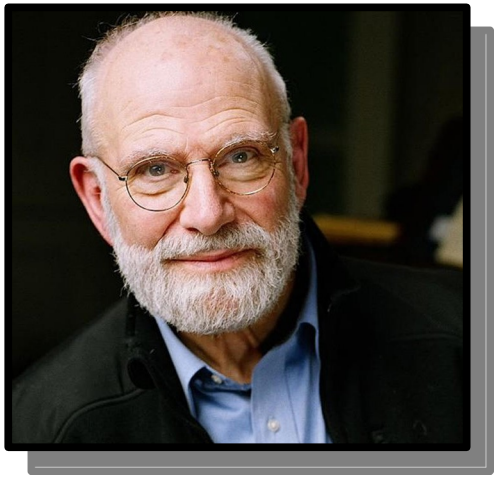
You know which doctors you have available to send.

List all the possible teams you can make from your list of all the doctors.




NYU
Medical
Center

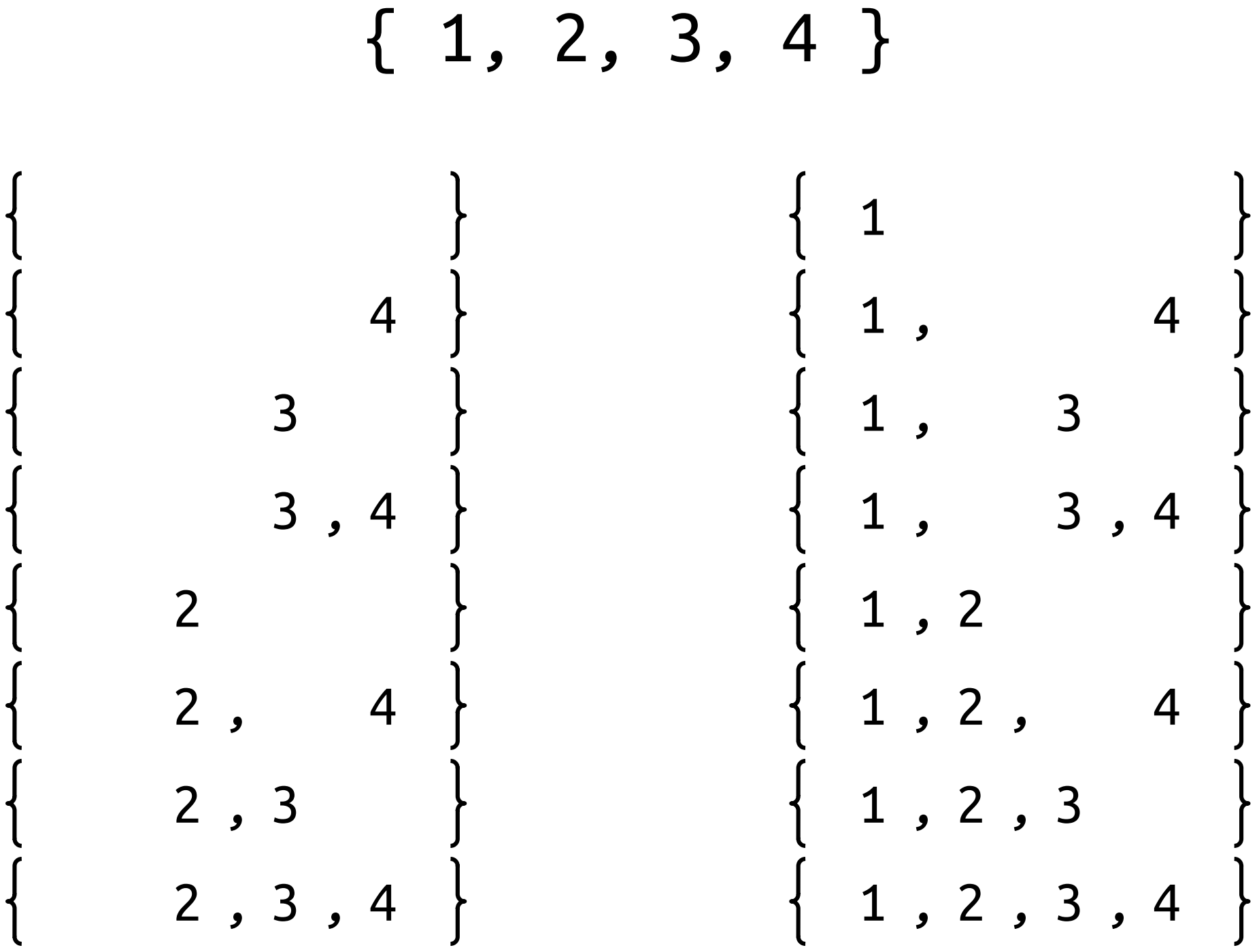


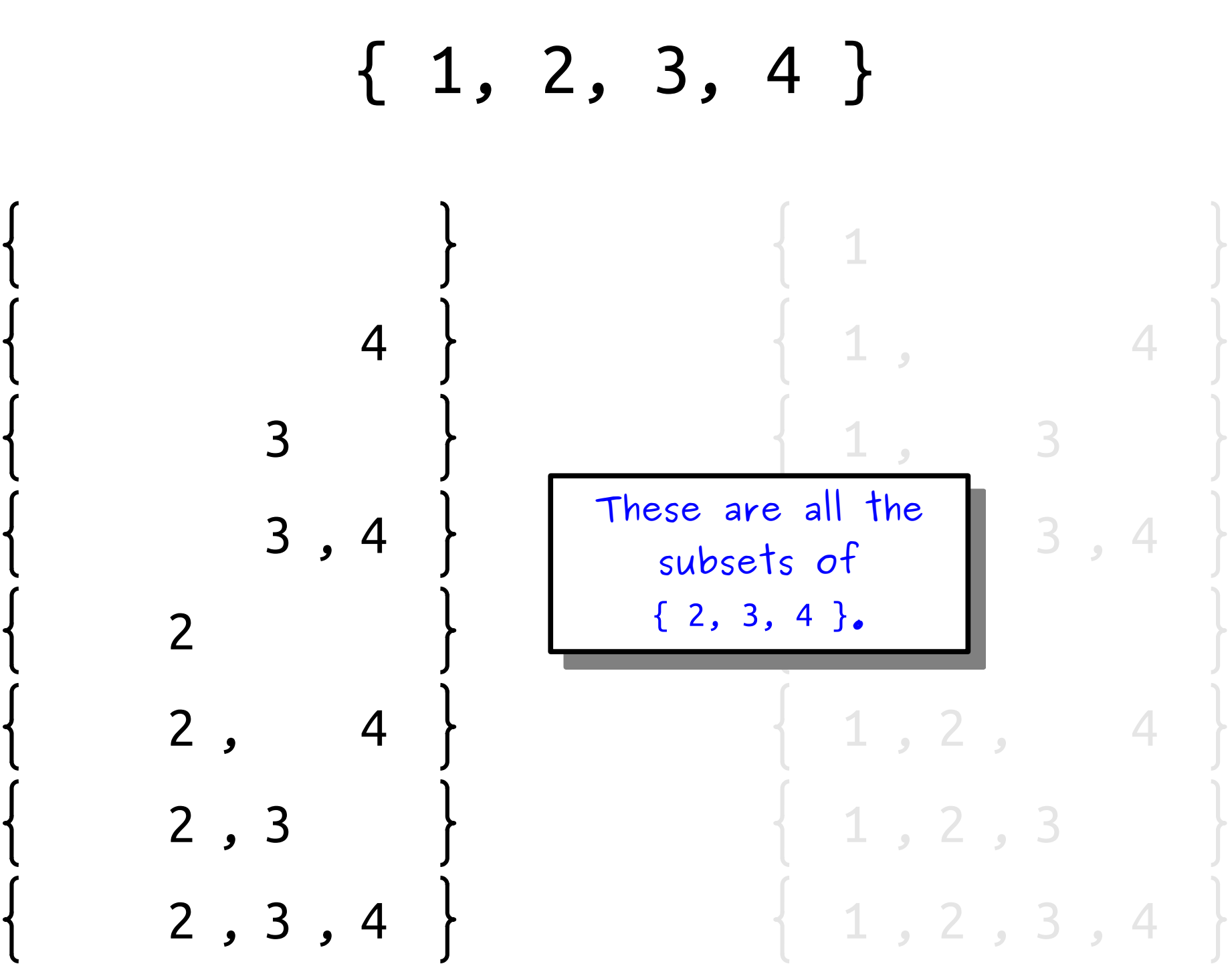



NYU
Medical
Center

1 2 3 4

{ 1, 2, 3, 4 }


{ 1, 2, 3, 4 }
{ }
{ 4 }
{ 3 }
{ 3, 4 }
{ 2 }
{ 2, 4 }
{ 2, 3 }
{ 2, 3, 4 }
{ 1 }
{ 1, 4 }
{ 1, 3 }
{ 1, 3, 4 }
{ 1, 2 }
{ 1, 2, 4 }
{ 1, 2, 3 }
{ 1, 2, 3, 4 }


{ 1, 2, 3, 4 }
{ }
{ 4 }
{ 3 }
{ 3, 4 }
{ 2 }
{ 2, 4 }
{ 2, 3 }
{ 2, 3, 4 }
{ 1 }
{ 1, 4 }
{ 1, 3 }
{ 3, 4 }
{ 1, 2, 4 }
{ 1, 2, 3 }
{ 1, 2, 3, 4 }
These are all the subsets of { 2, 3, 4 }.

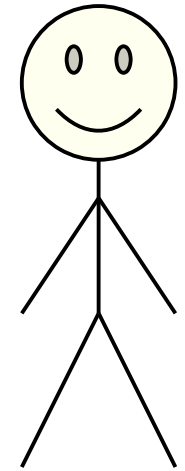
{ 1, 2, 3, 4 }

These are all the subsets of { 2, 3, 4 } with 1 inserted into them.

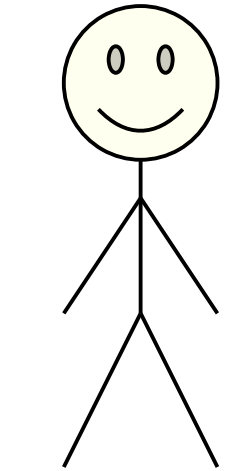
{ 1 }
{ 1 , 4 }
{ 1 , 3 }
{ 1 , 3 , 4 }
{ 1 , 2 }
{ 1 , 2 , 4 }
{ 1 , 2 , 3 }
{ 1 , 2 , 3 , 4 }

Andrews List Subsets

Andrews List Subsets

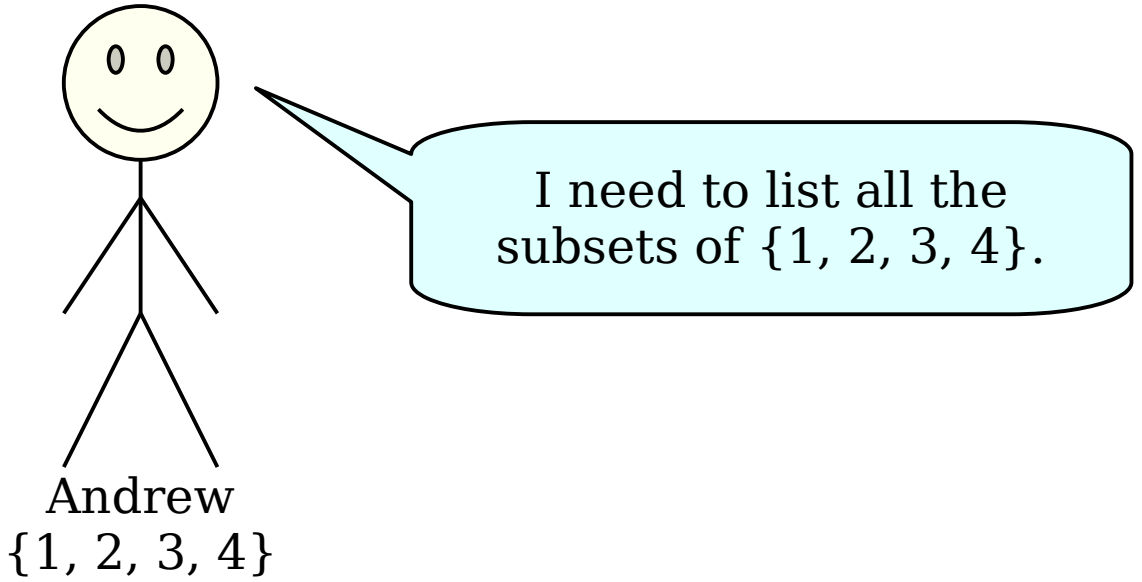


Andrews List Subsets

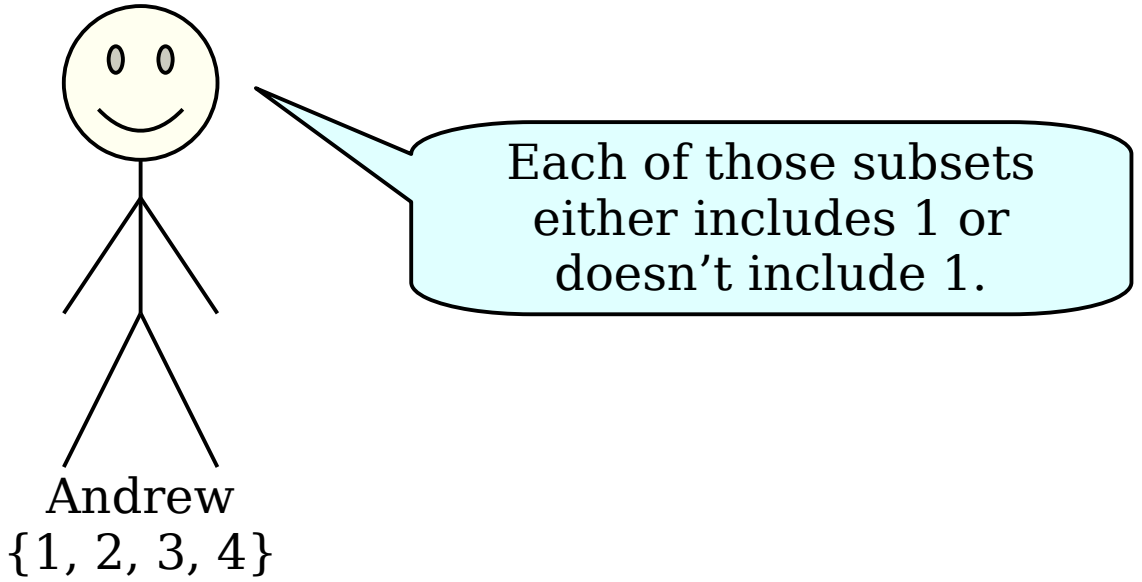


Andrew
{1, 2, 3, 4}

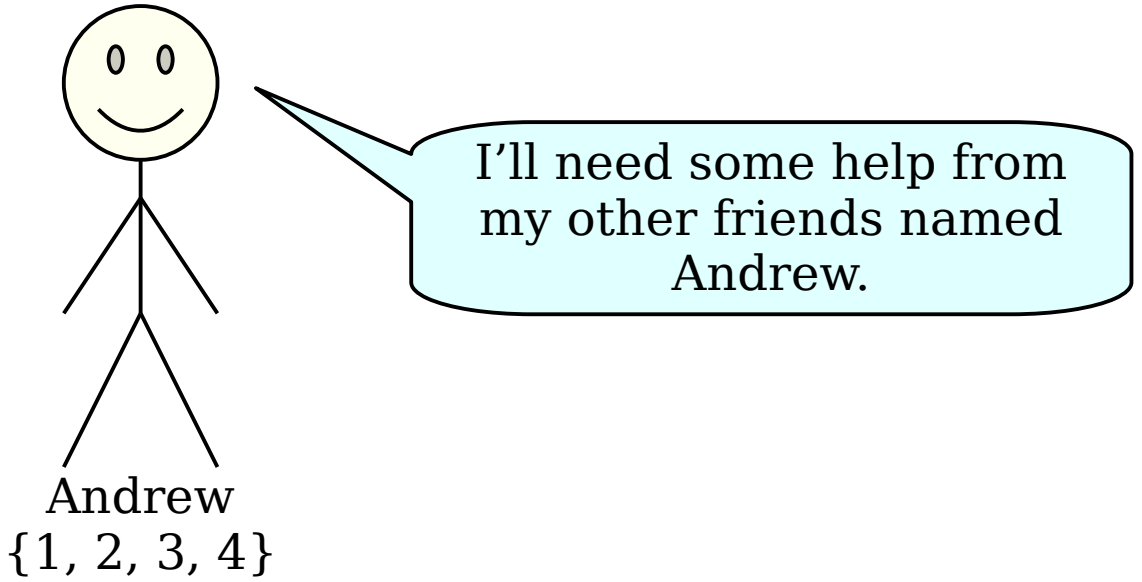
Andrews List Subsets



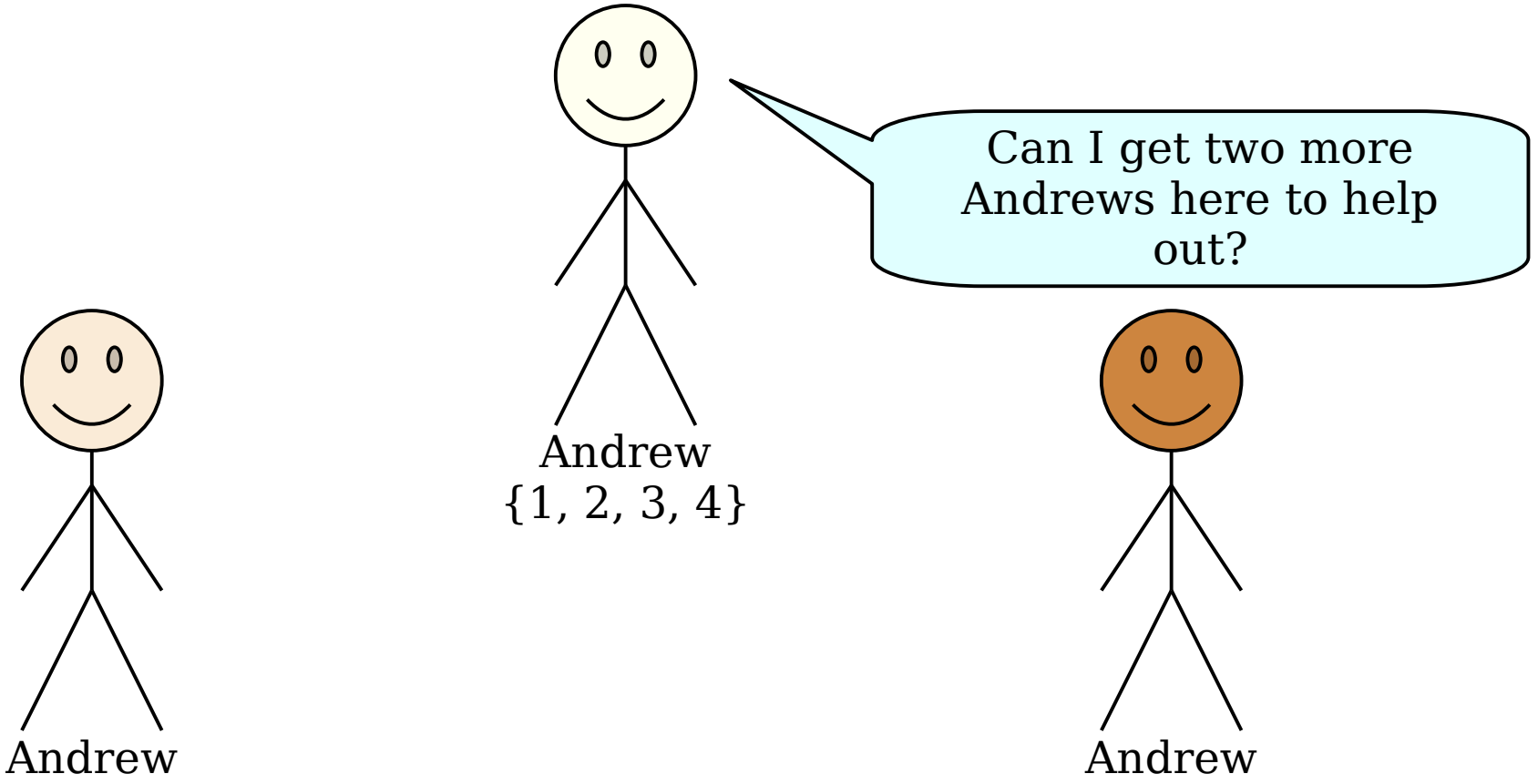
Andrews List Subsets



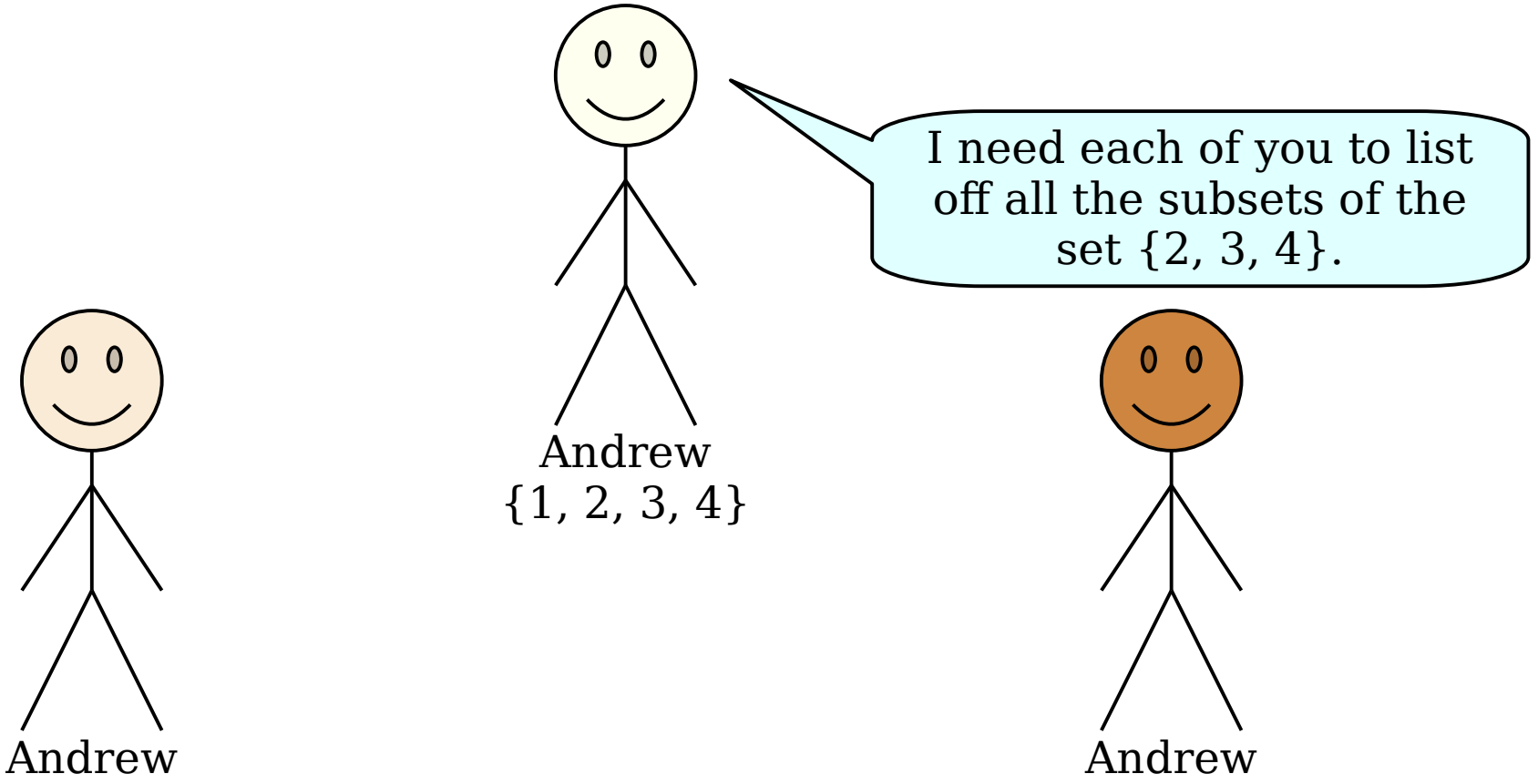
Andrews List Subsets



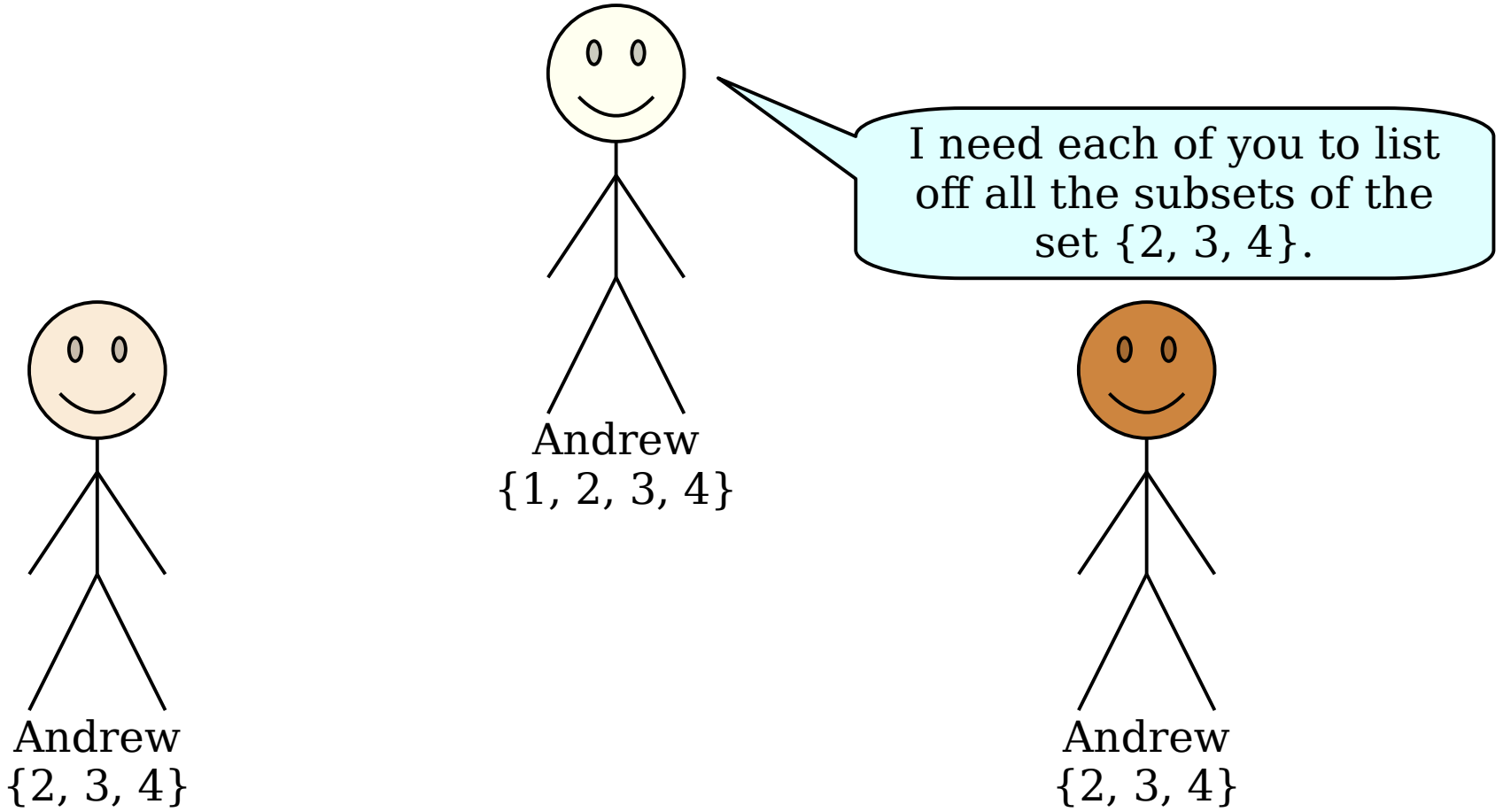
Andrews List Subsets



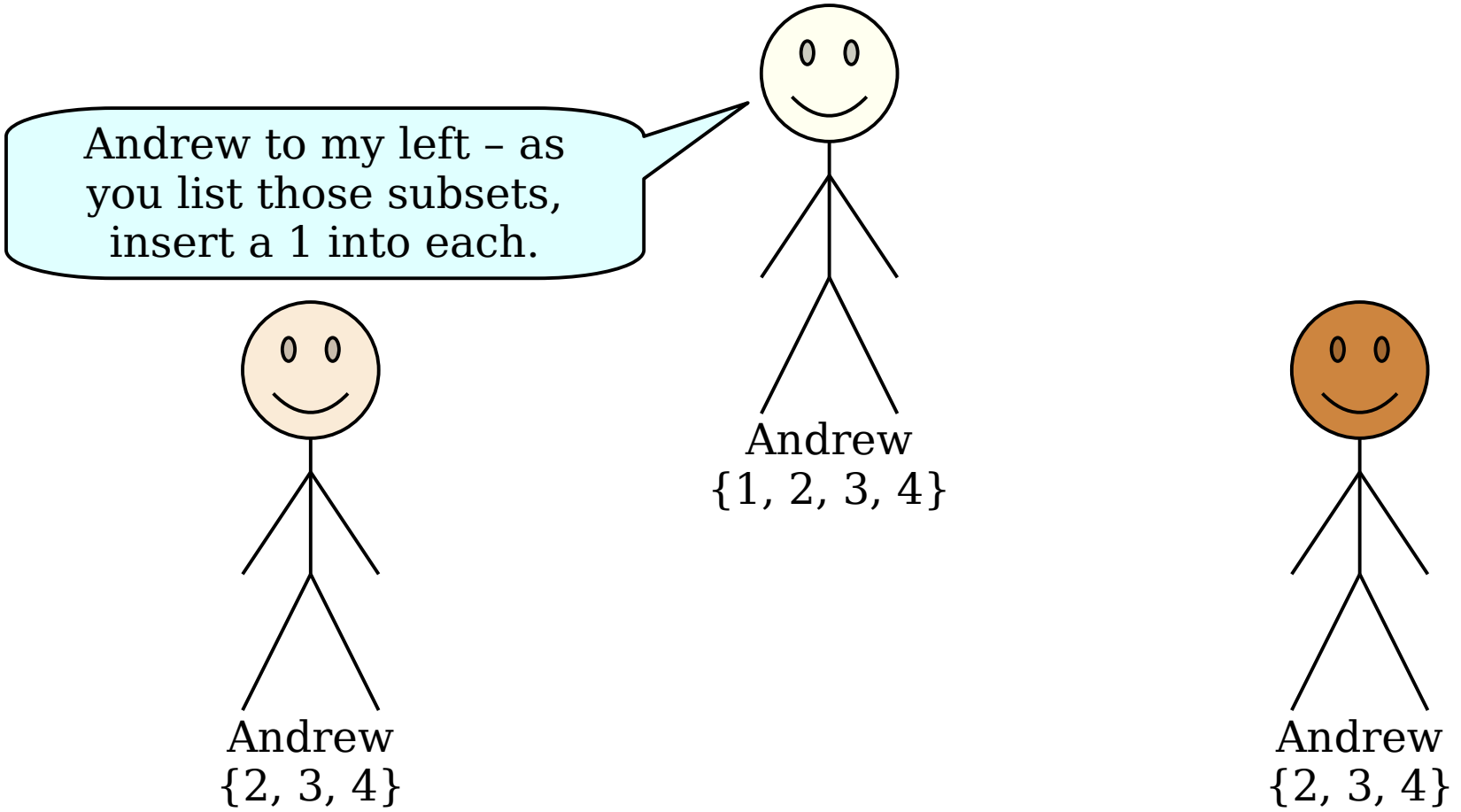
Andrews List Subsets



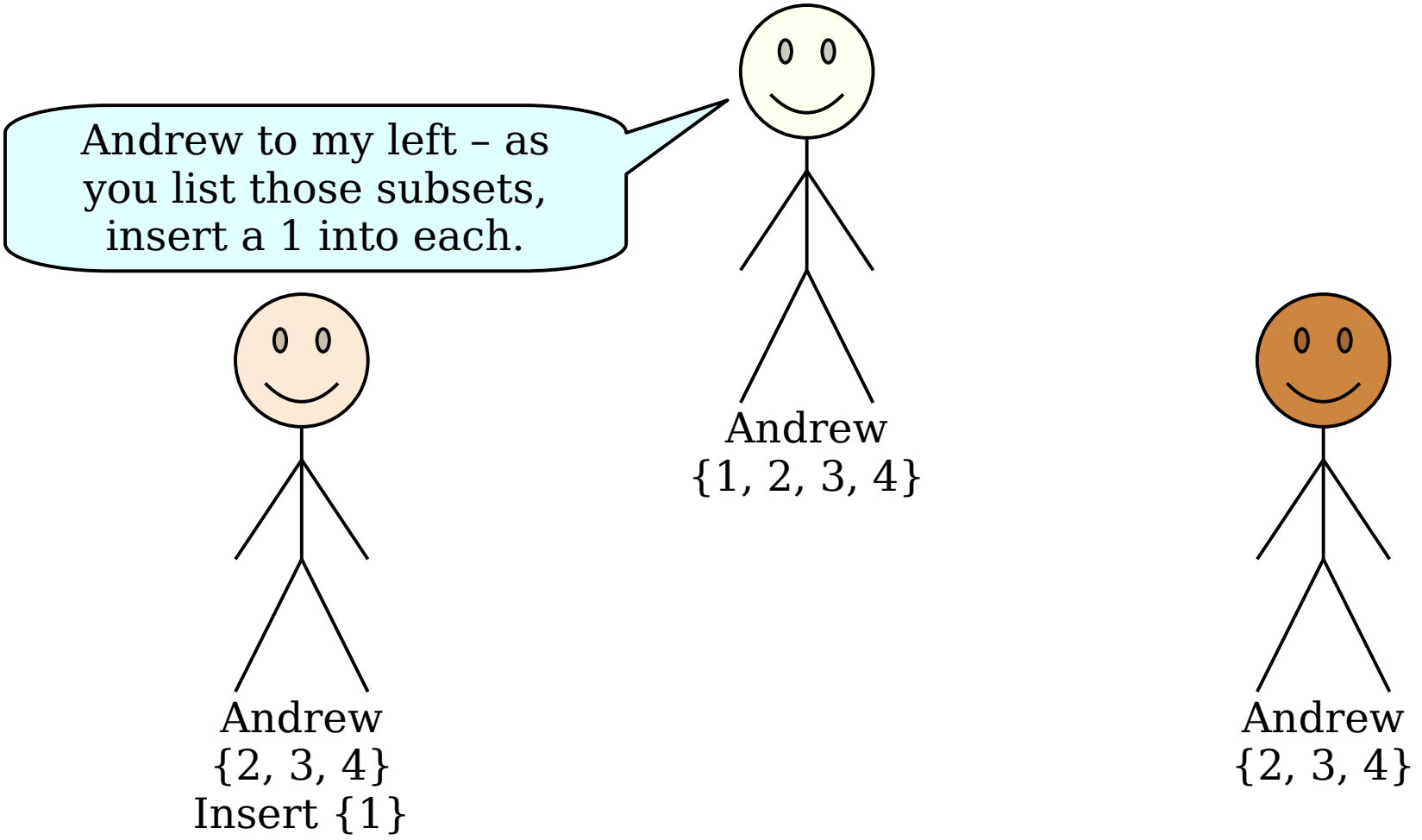
Andrews List Subsets



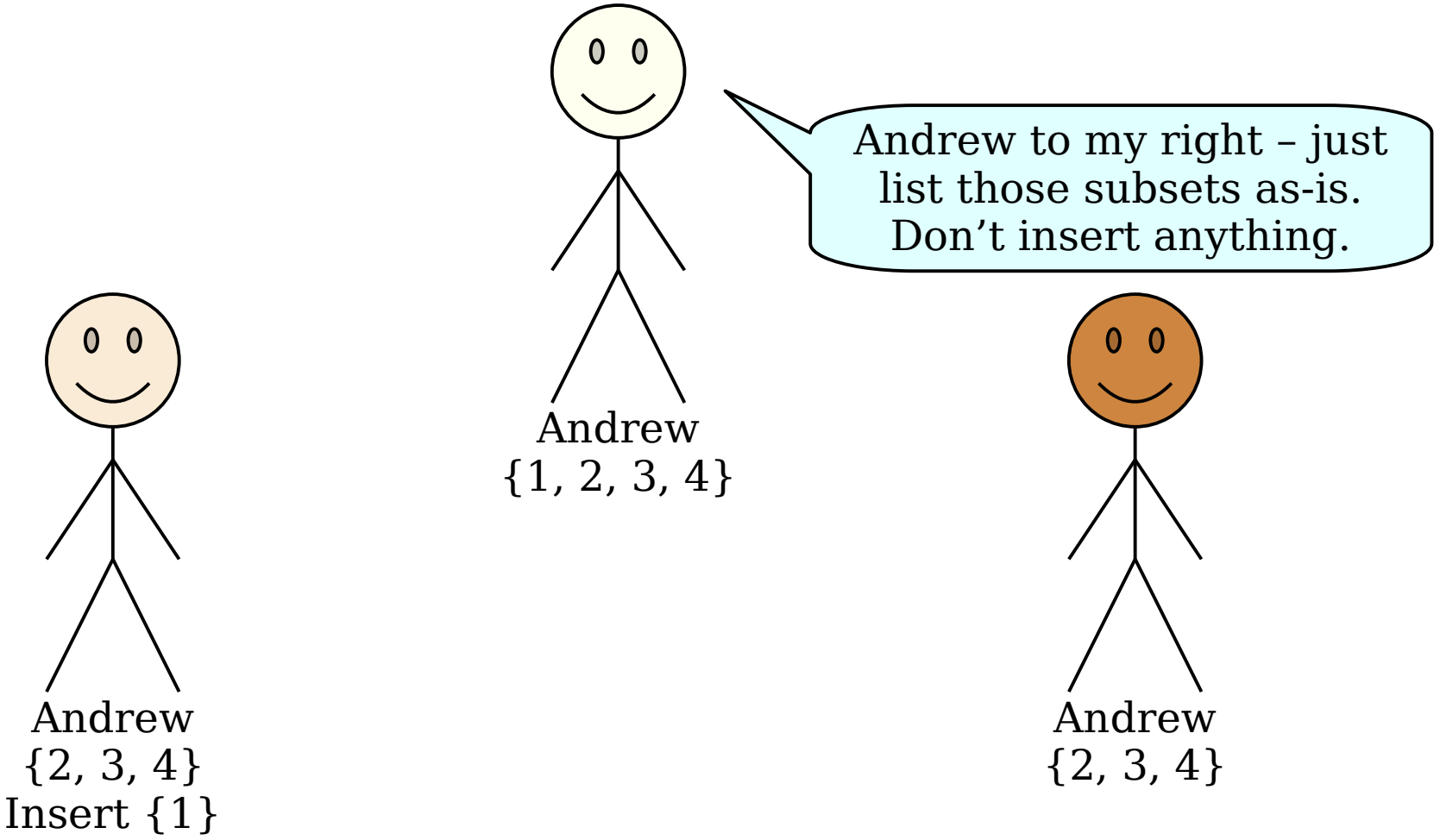
Andrews List Subsets



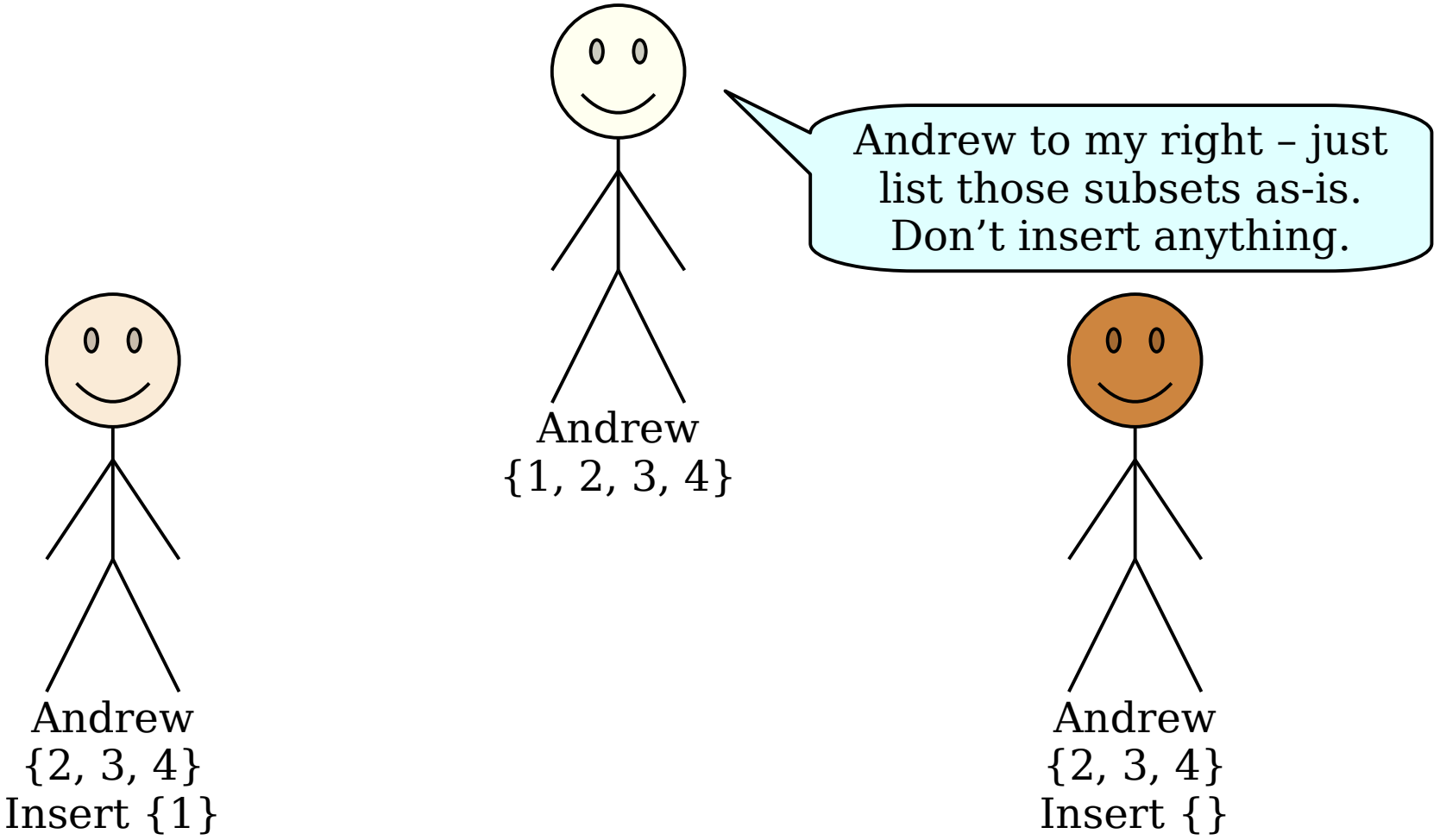
Andrews List Subsets



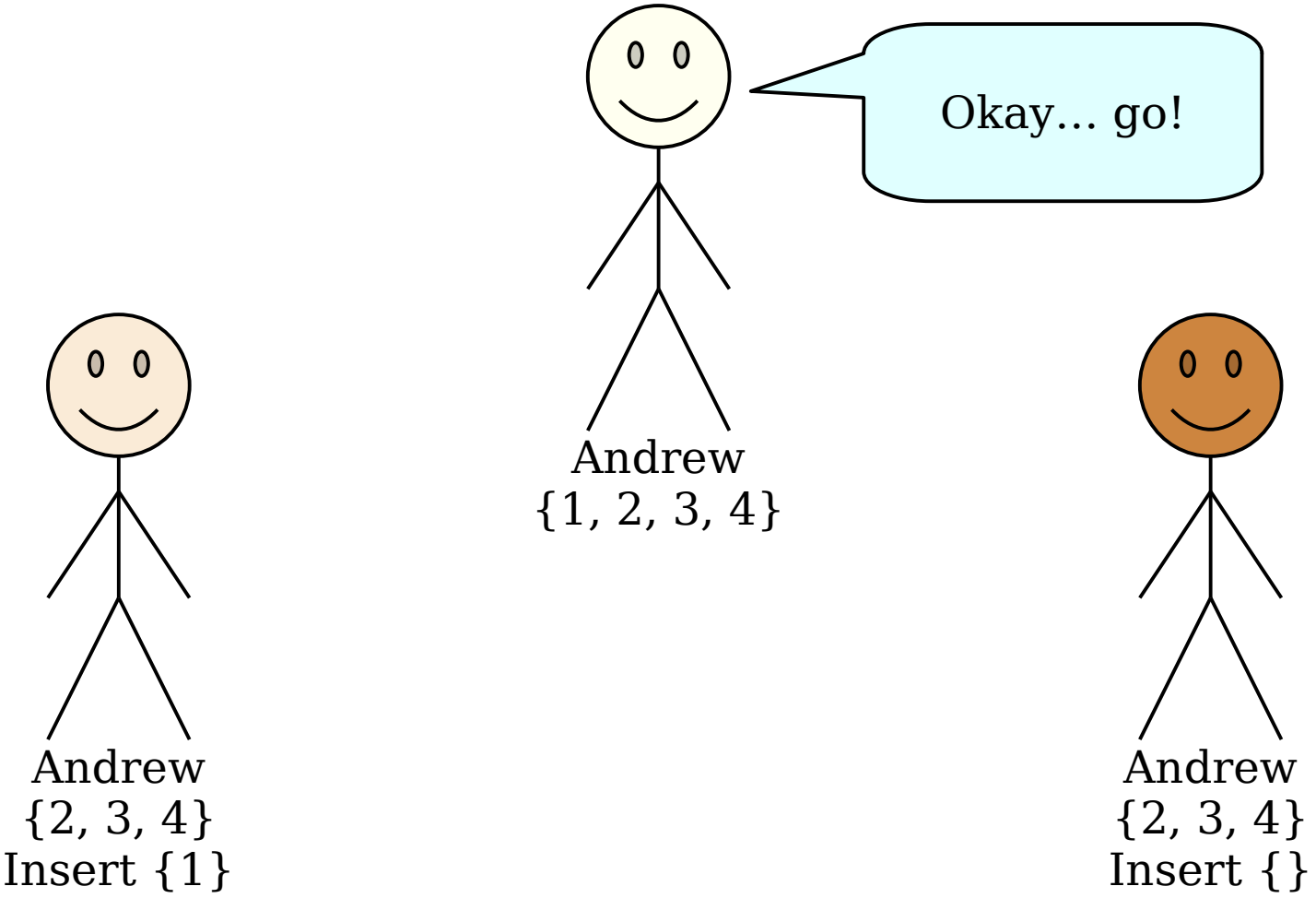
Andrews List Subsets



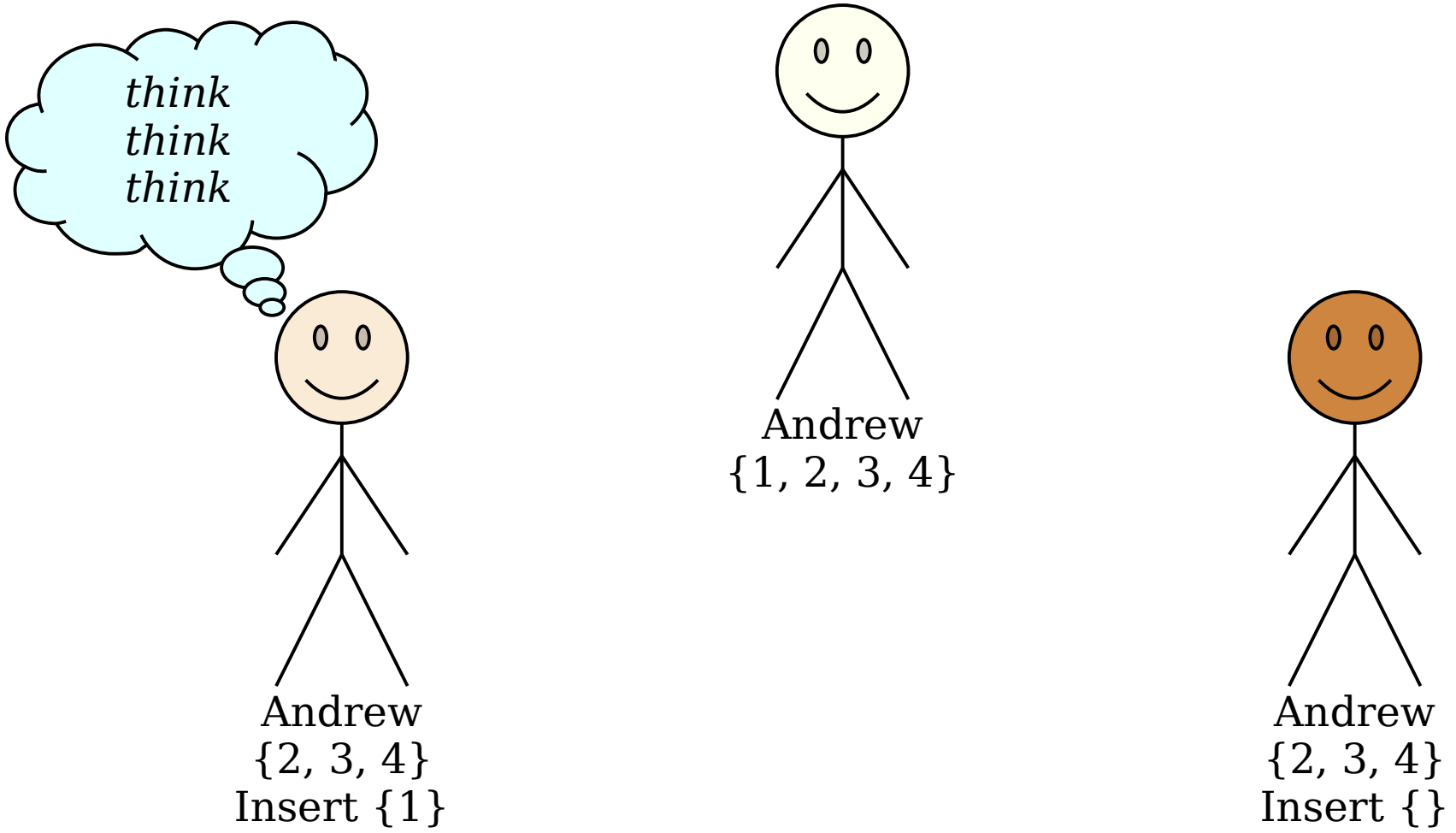
Andrews List Subsets



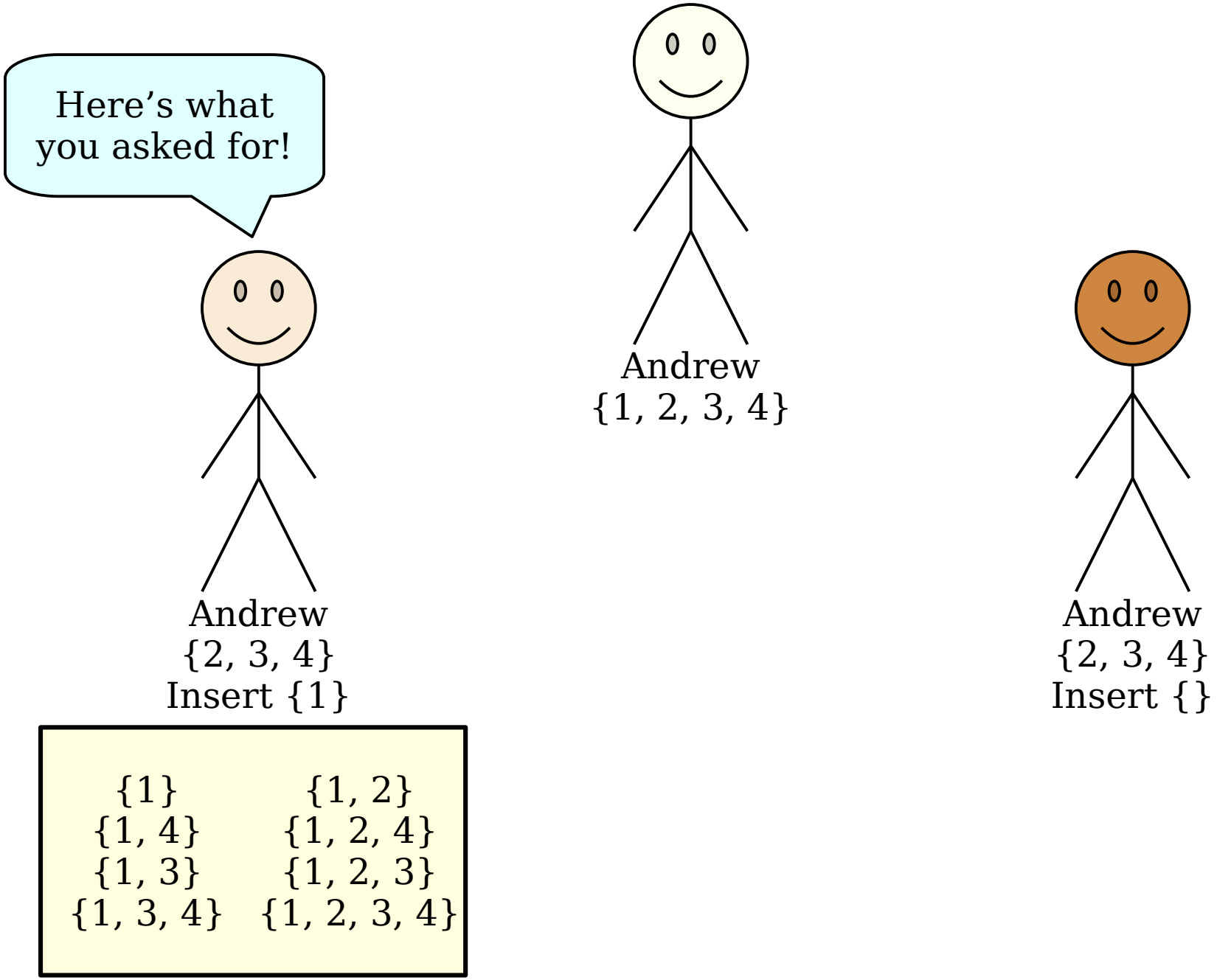
Andrews List Subsets



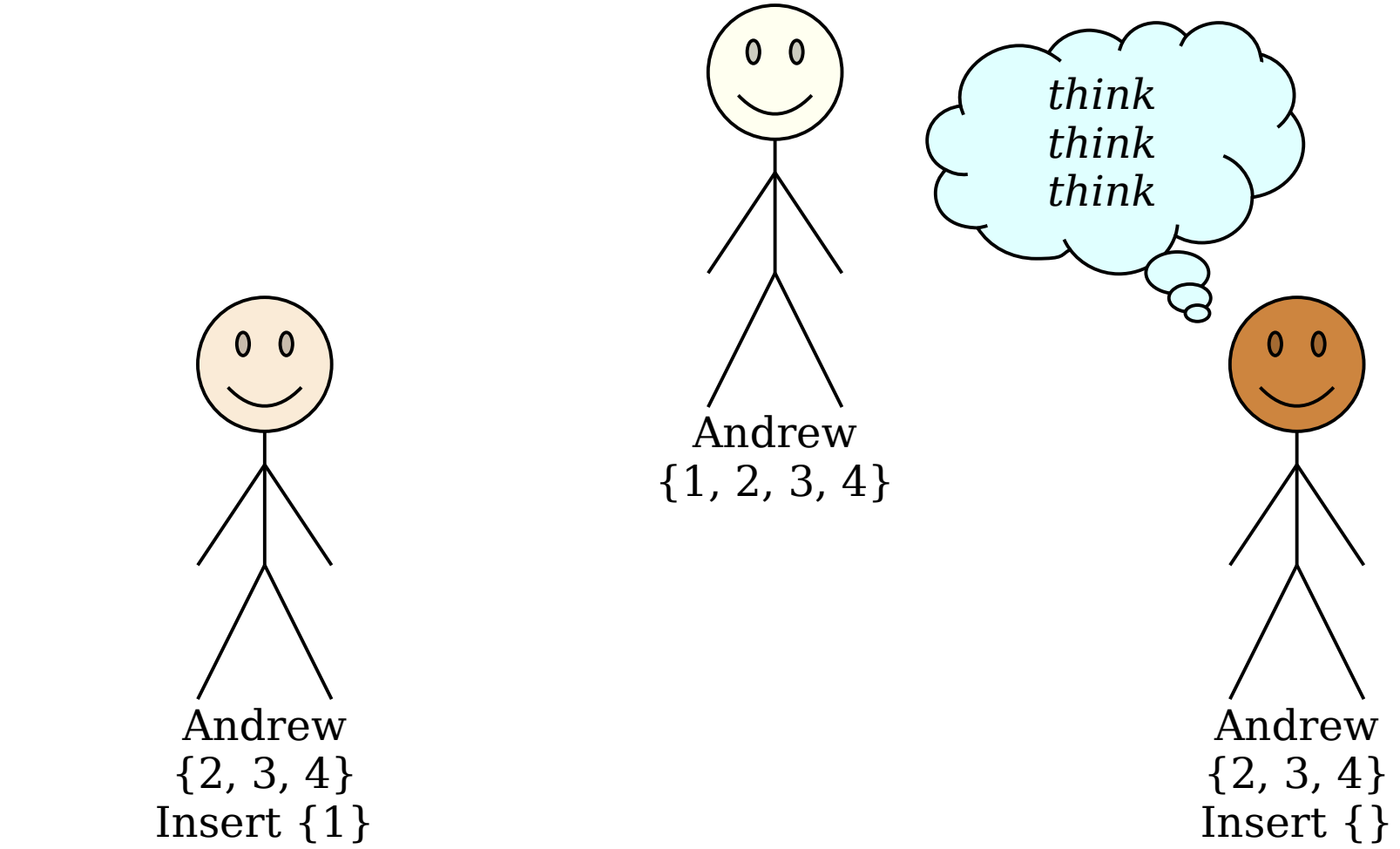
Andrews List Subsets



Andrews List Subsets

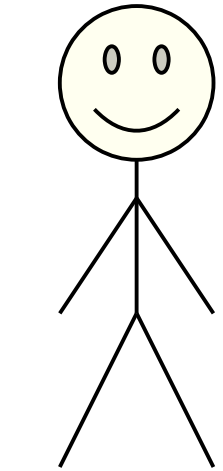


Andrews List Subsets

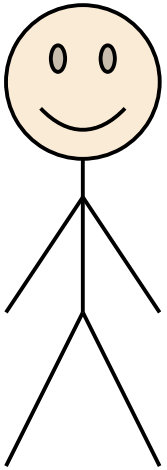


{1}	{1, 2}
{1, 4}	{1, 2, 4}
{1, 3}	{1, 2, 3}
{1, 3, 4}	{1, 2, 3, 4}

Andrews List Subsets

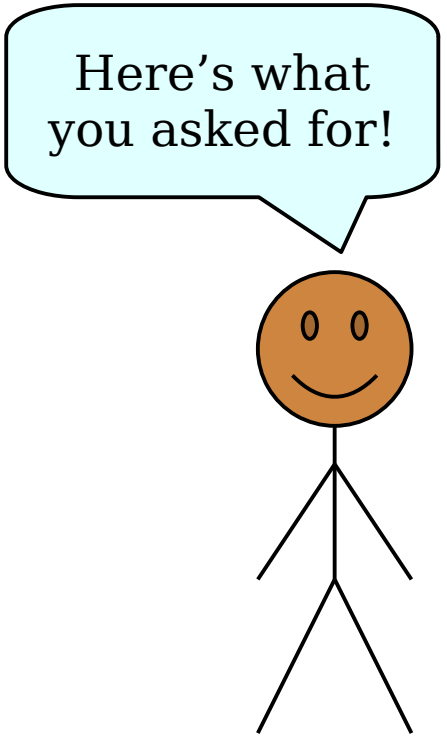


Andrew
{1, 2, 3, 4}



Andrew
{2, 3, 4}
Insert {1}

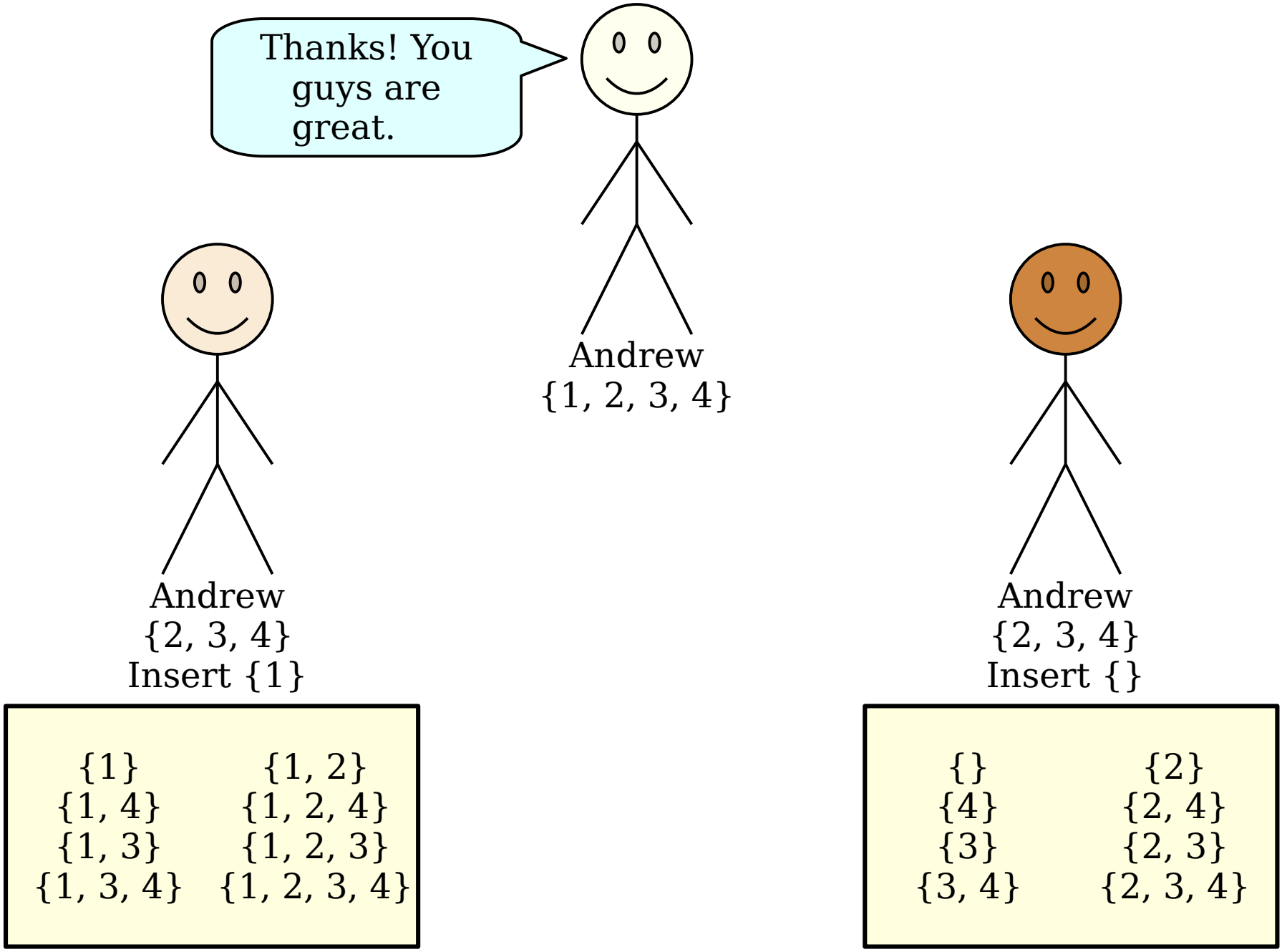
{1}	{1, 2}
{1, 4}	{1, 2, 4}
{1, 3}	{1, 2, 3}
{1, 3, 4}	{1, 2, 3, 4}



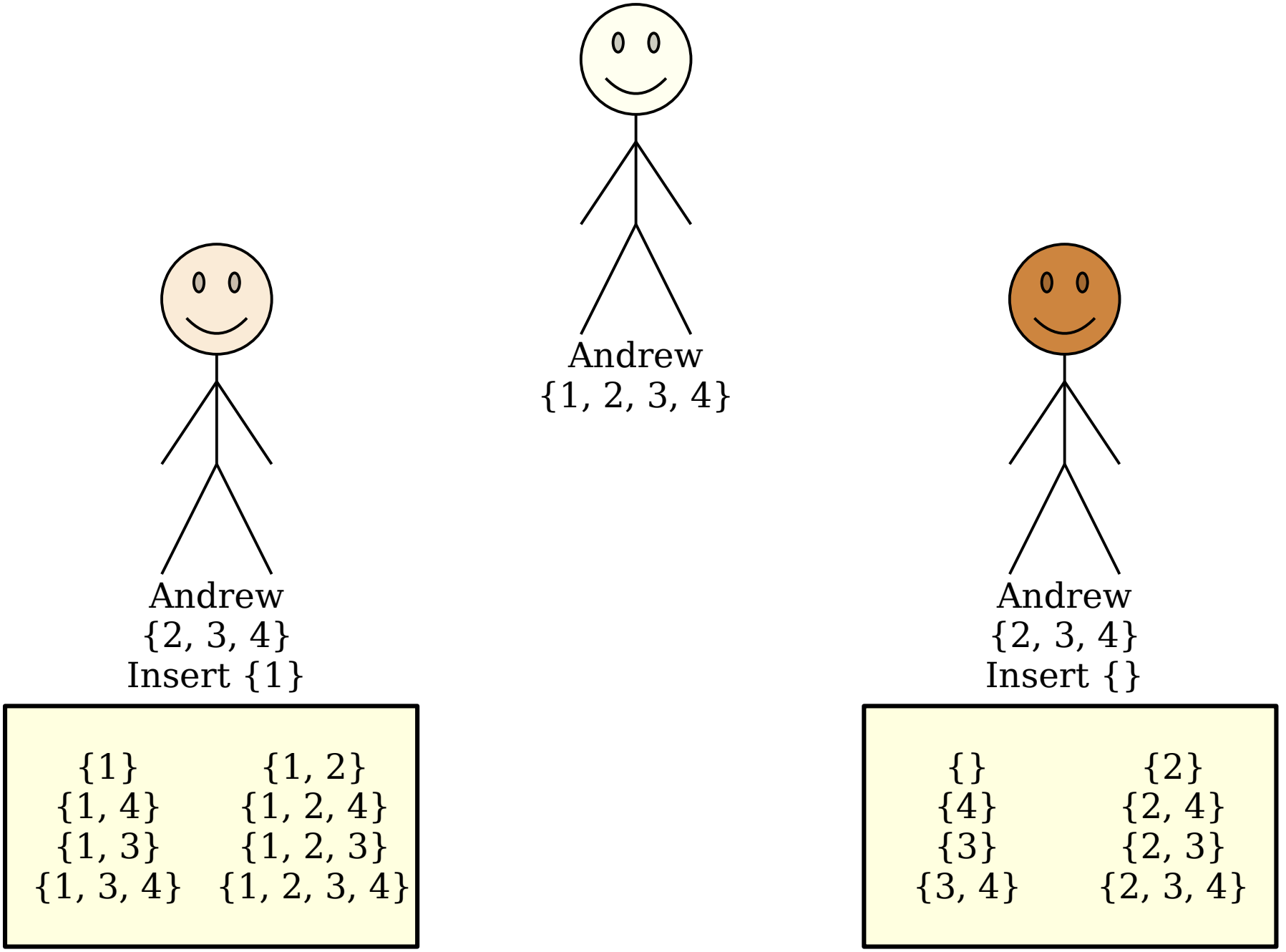
Andrew
{2, 3, 4}
Insert {}

{}	{2}
{4}	{2, 4}
{3}	{2, 3}
{3, 4}	{2, 3, 4}

Andrews List Subsets

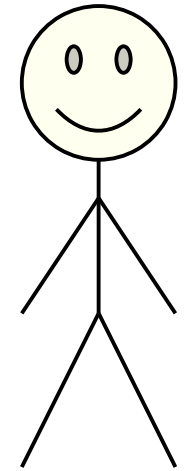


Andrews List Subsets



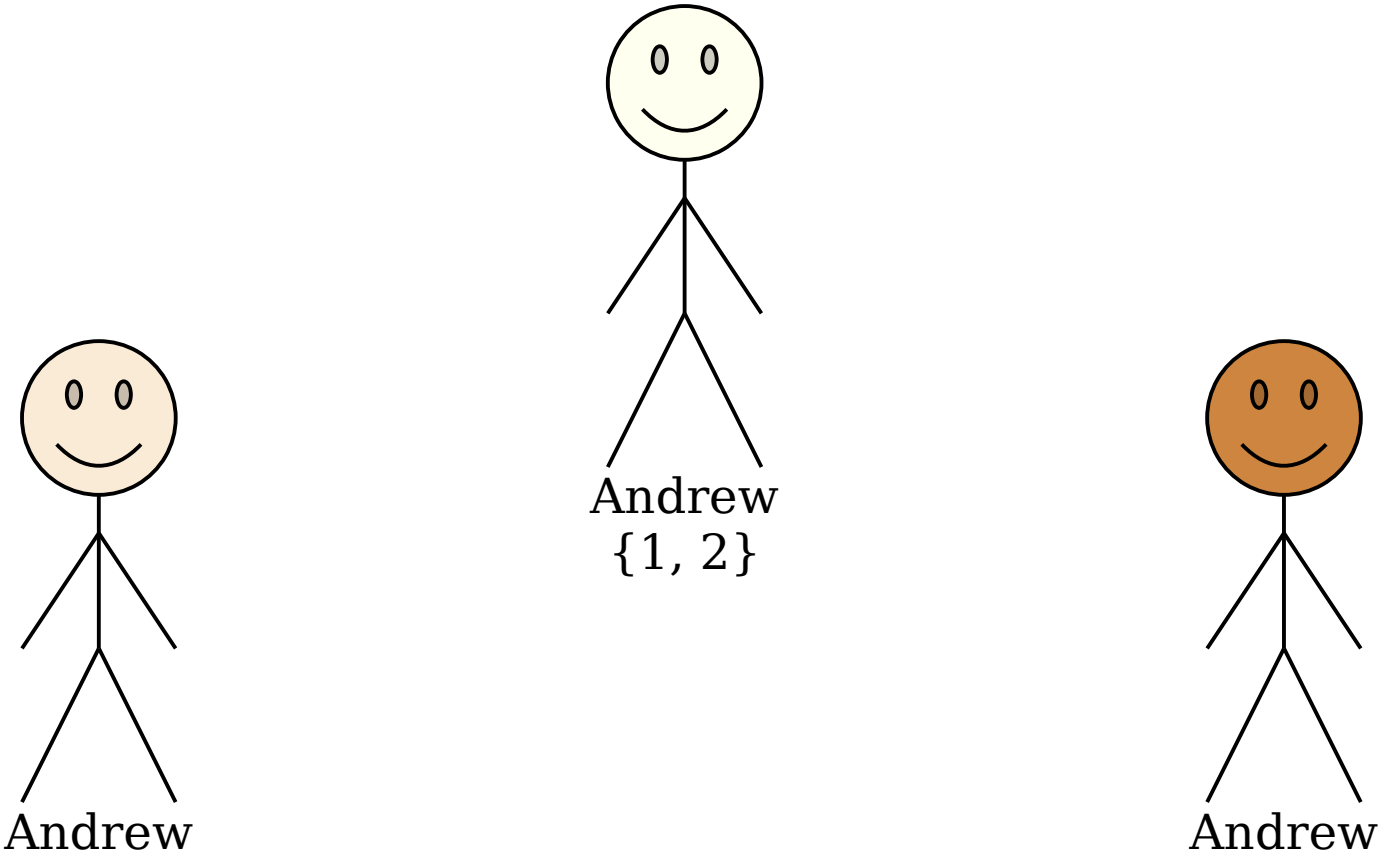
Thinking Recursively

Andrews List Subsets

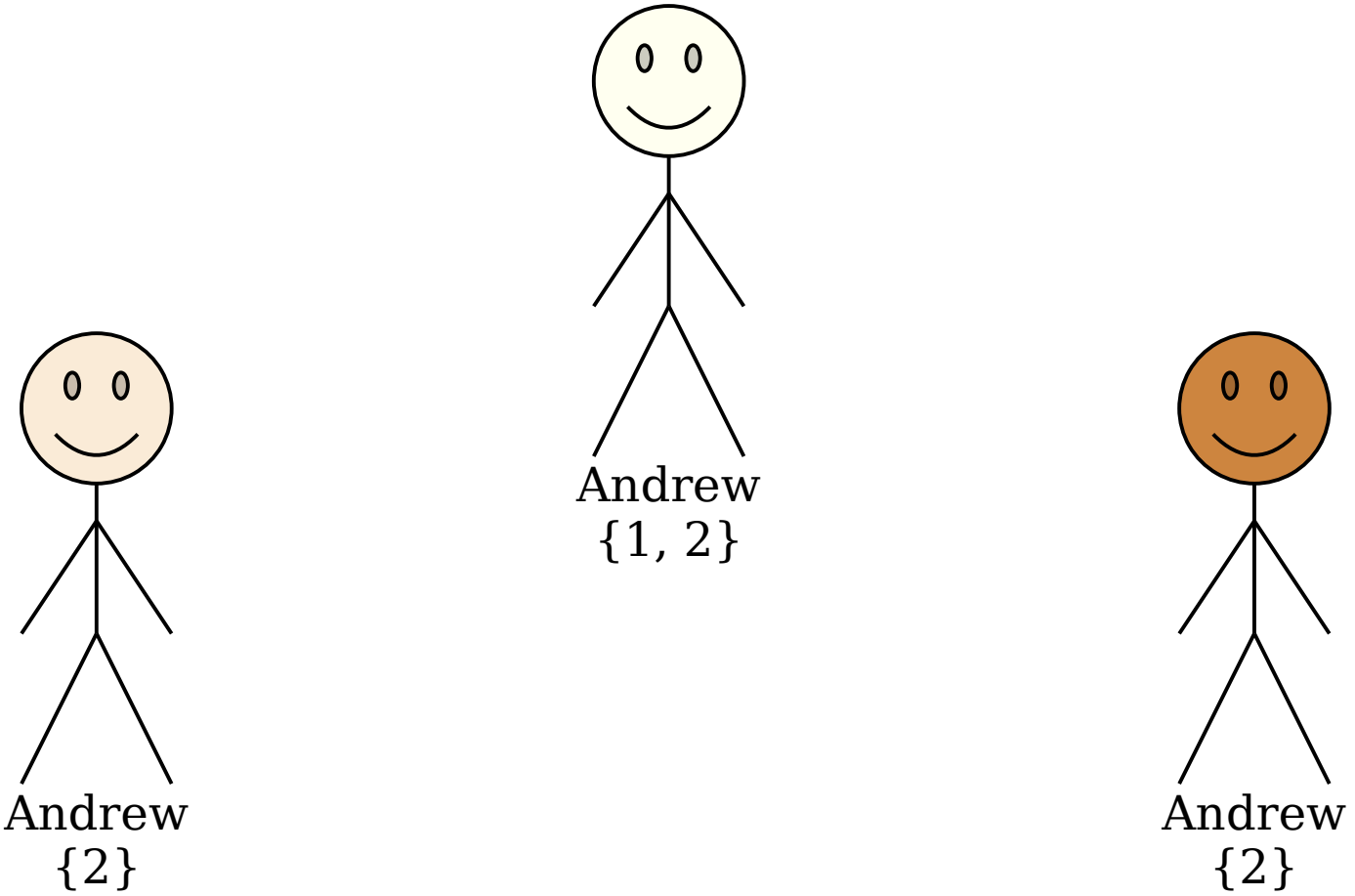


Andrew
{1, 2}

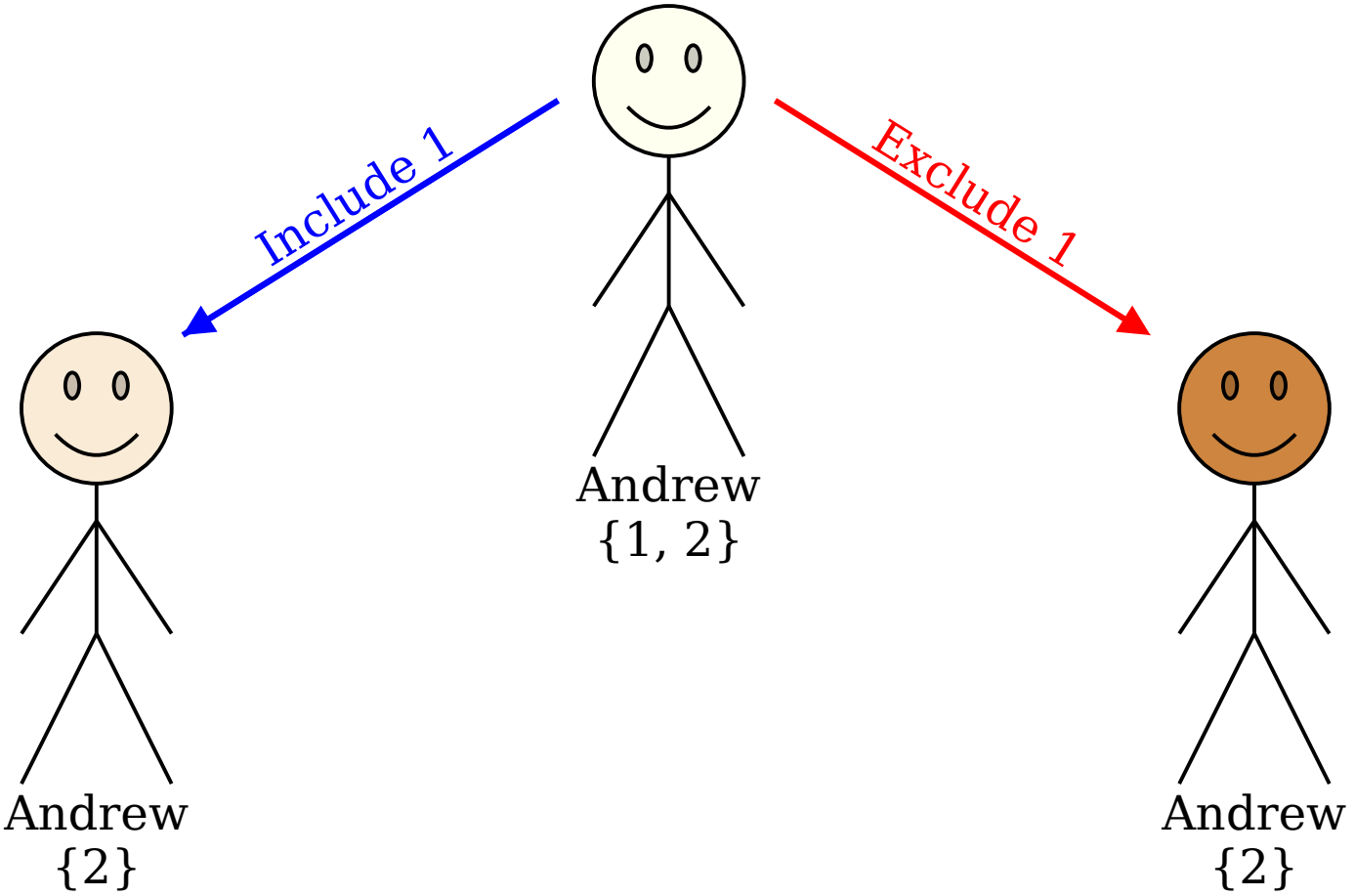
Andrews List Subsets



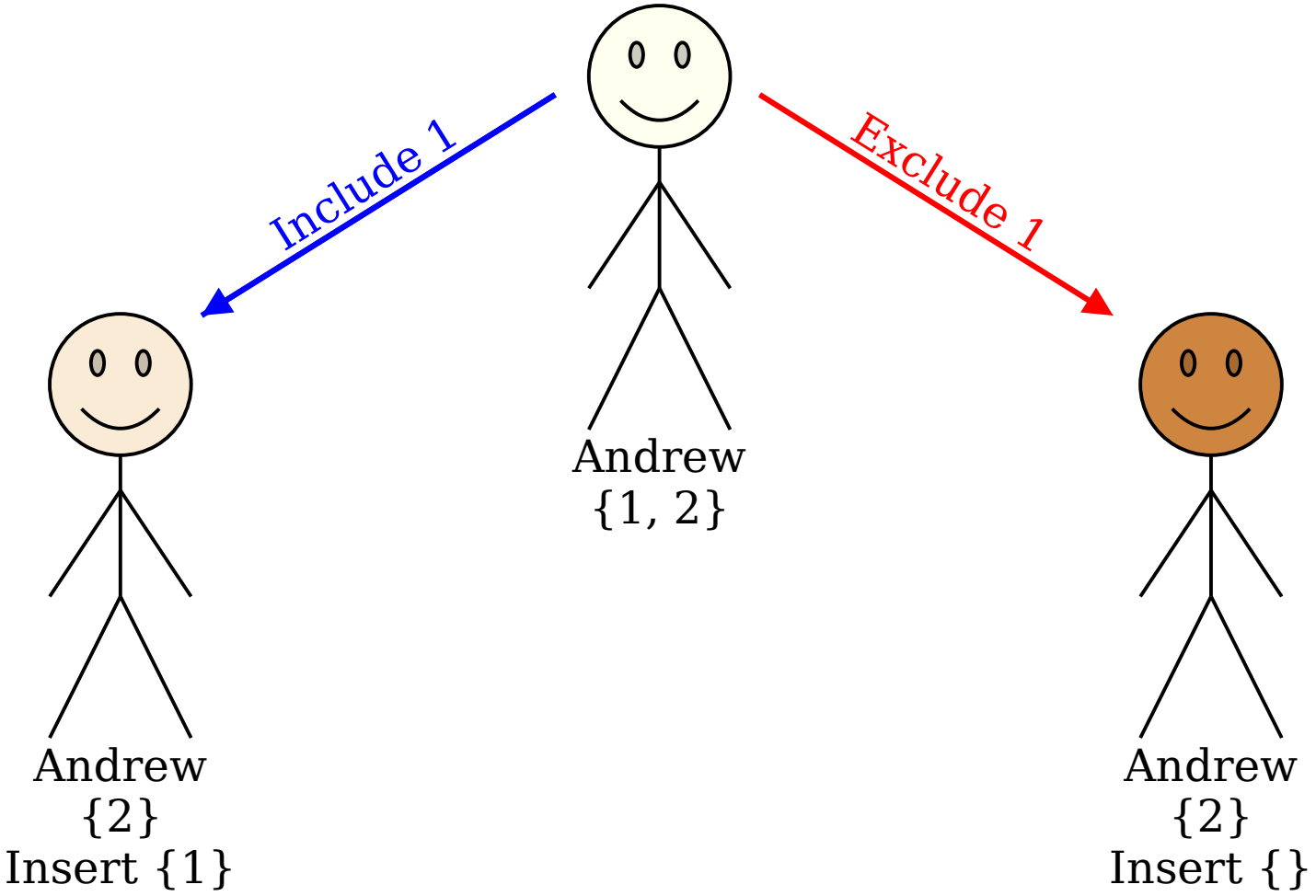
Andrews List Subsets



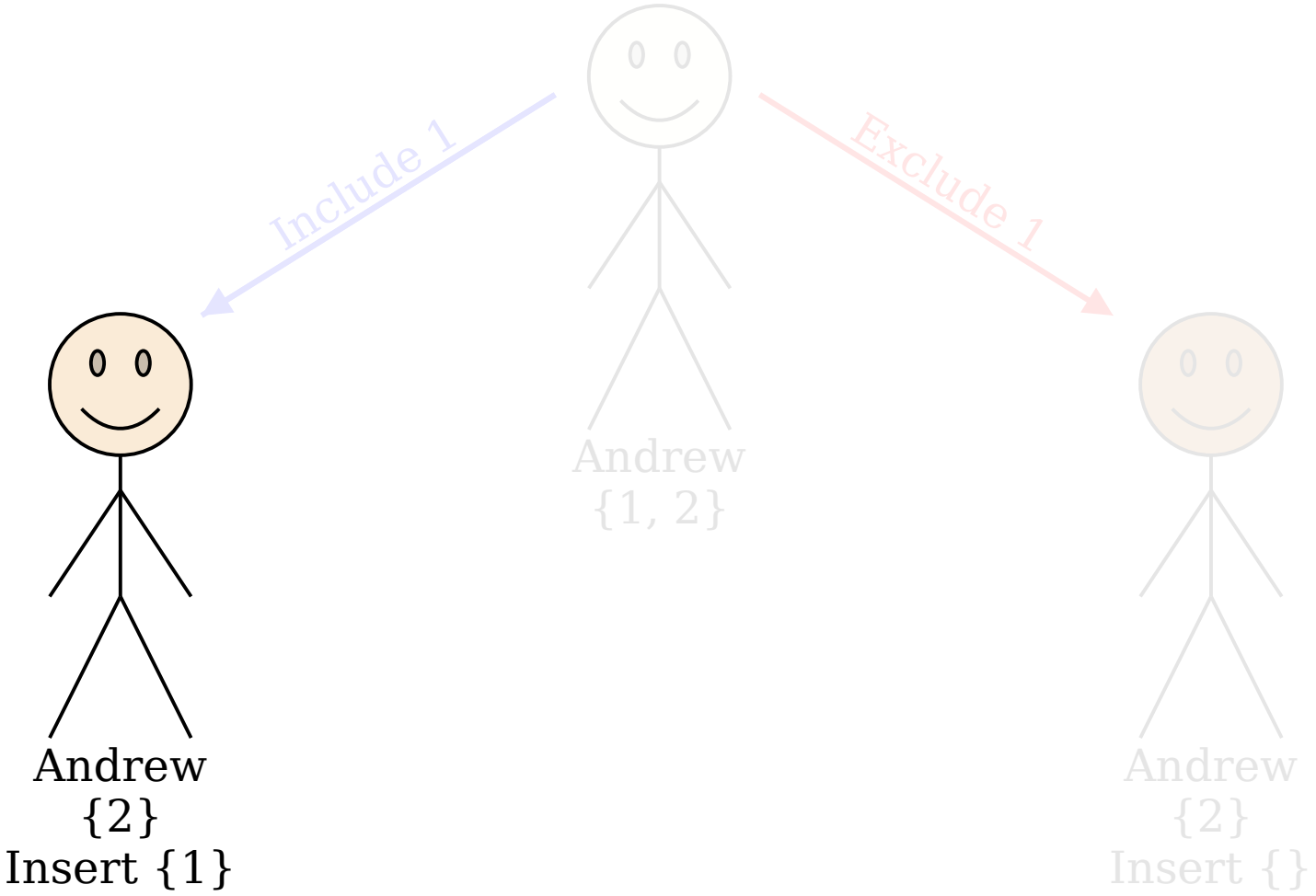
Andrews List Subsets



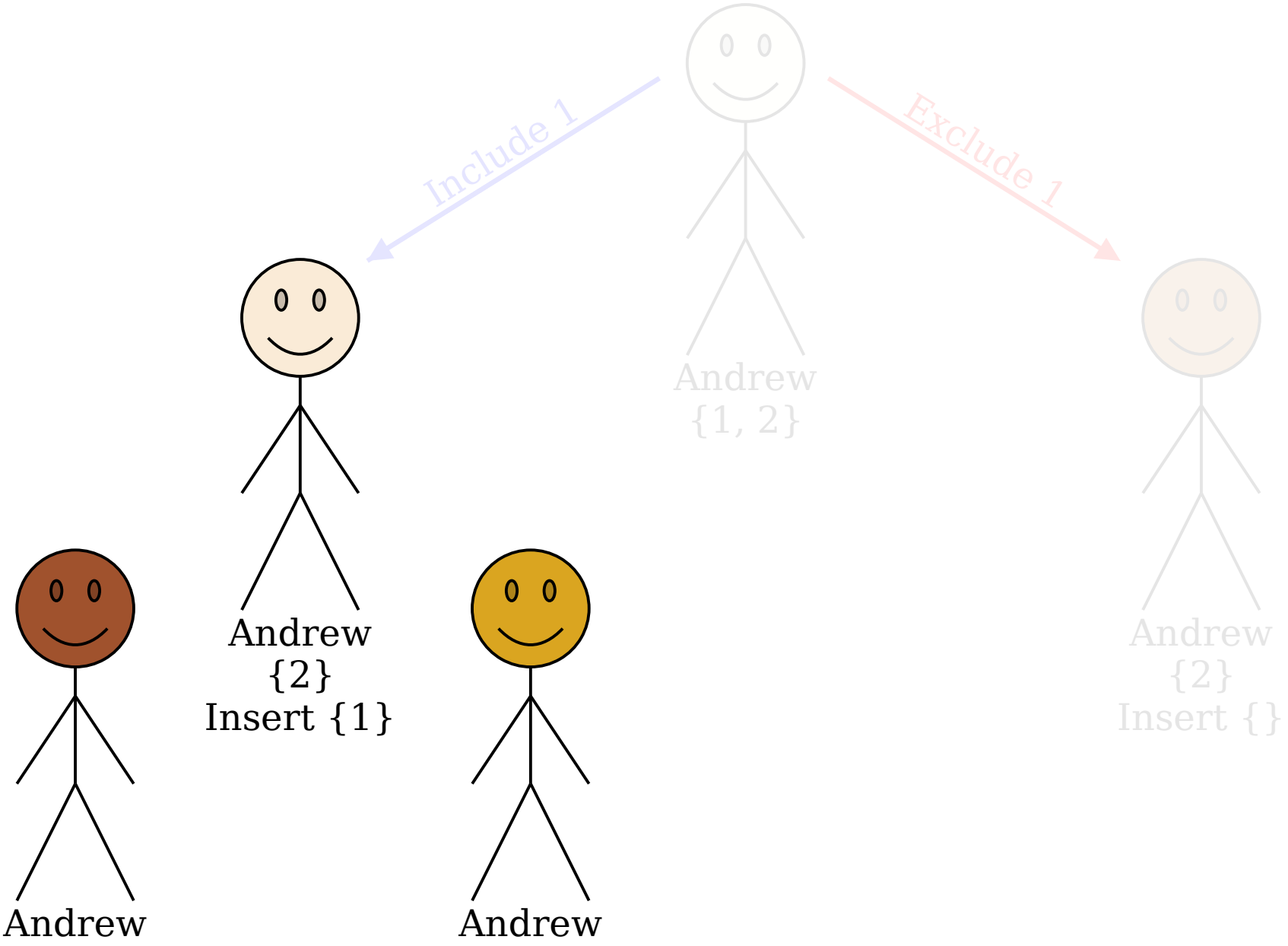
Andrews List Subsets



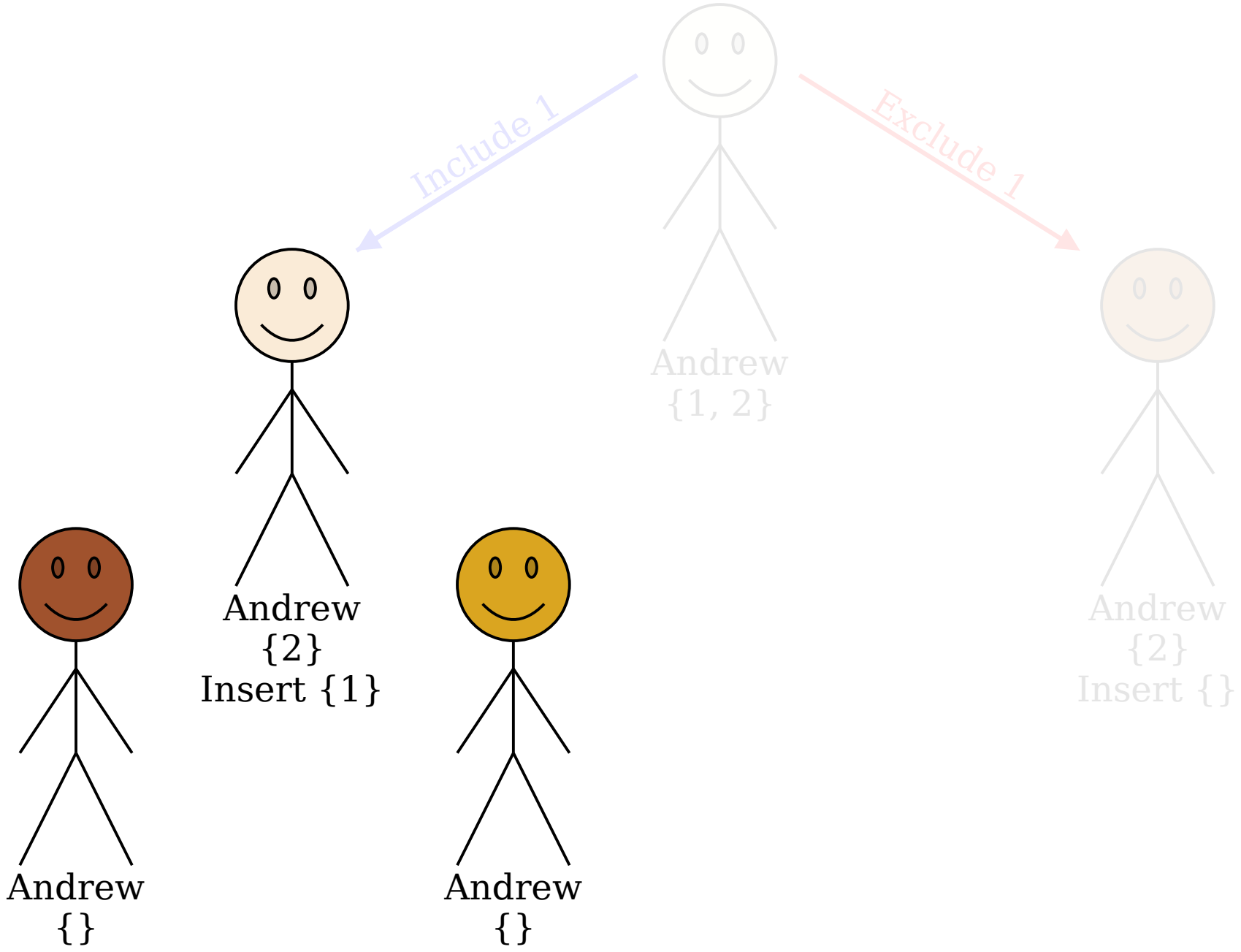
Andrews List Subsets



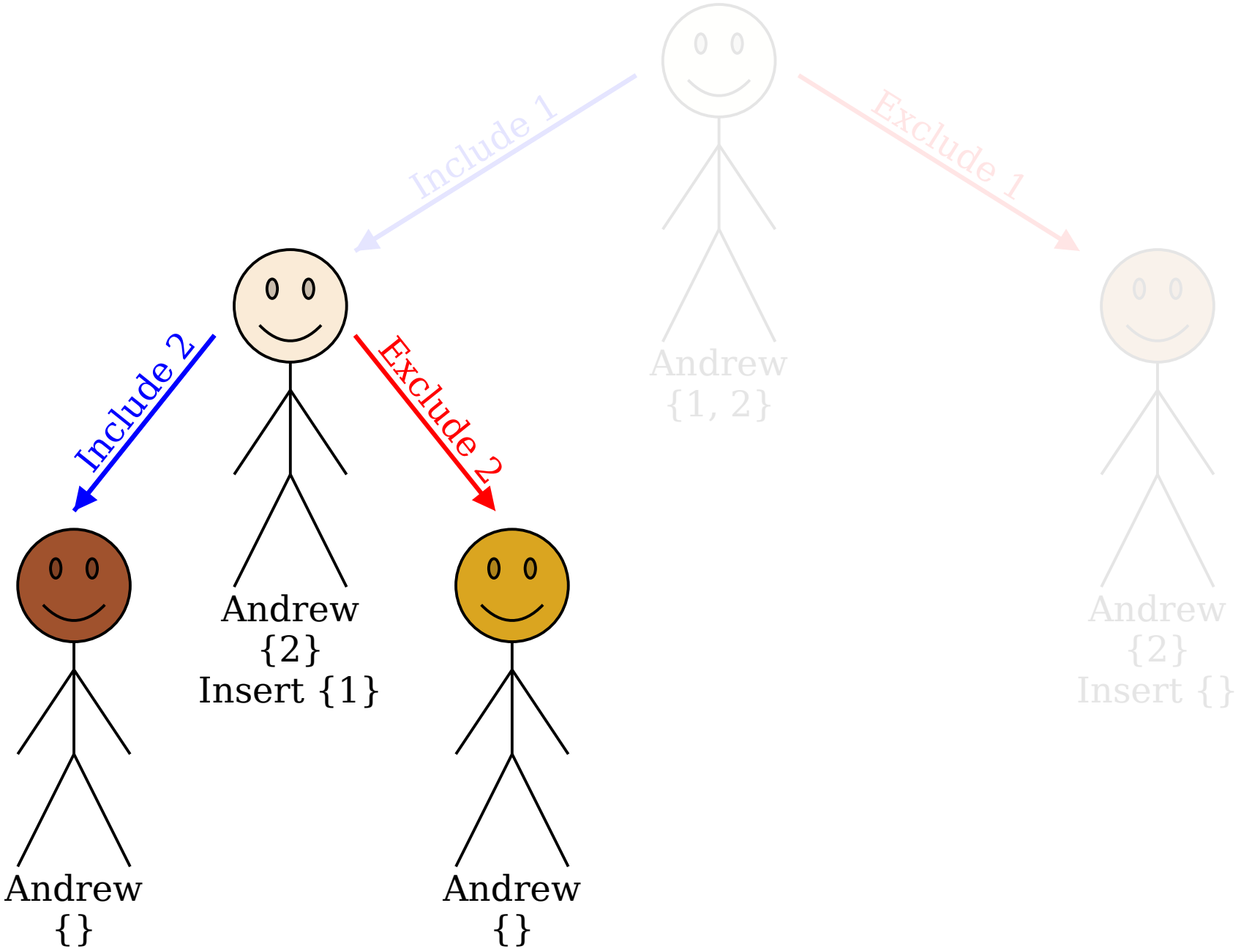
Andrews List Subsets



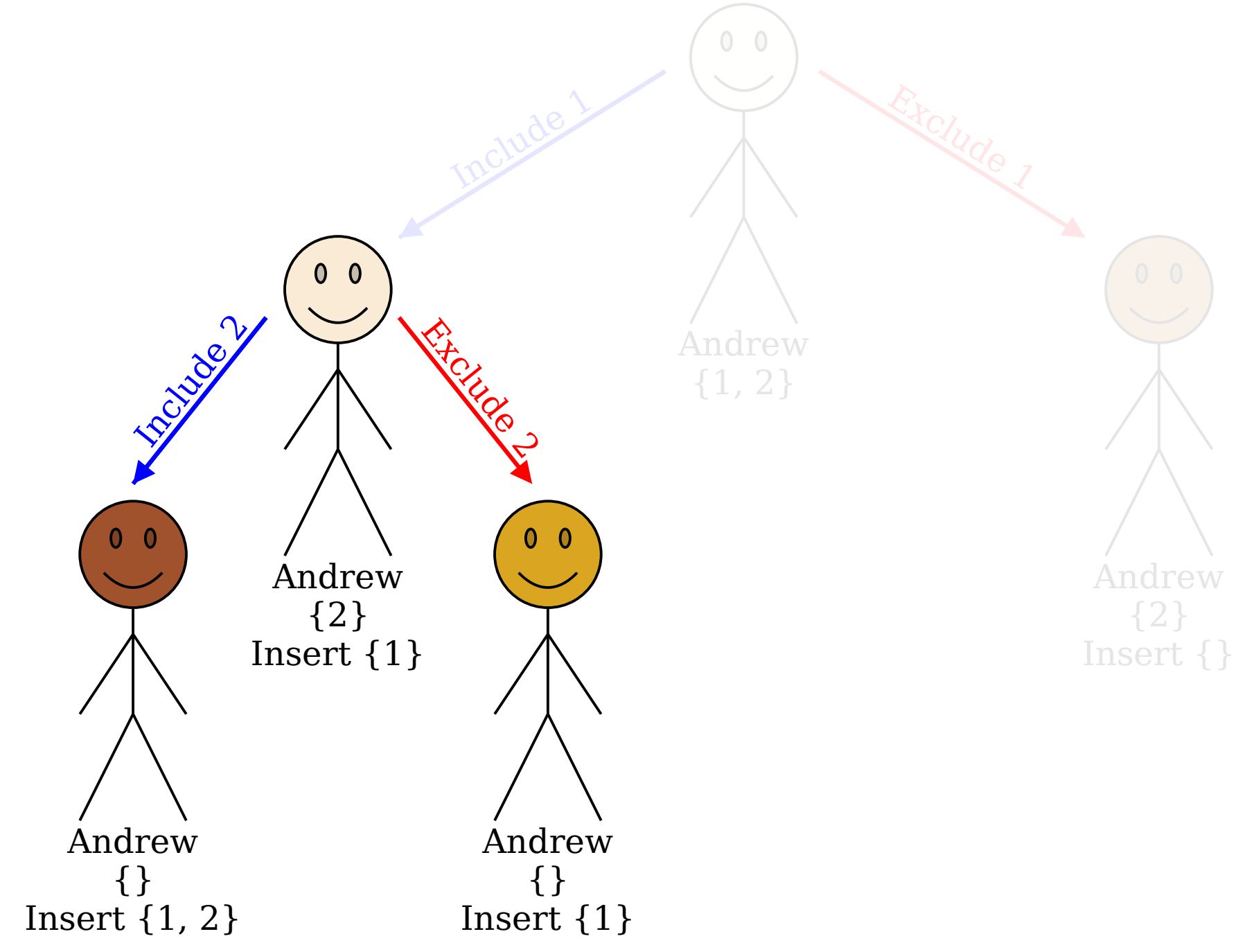
Andrews List Subsets



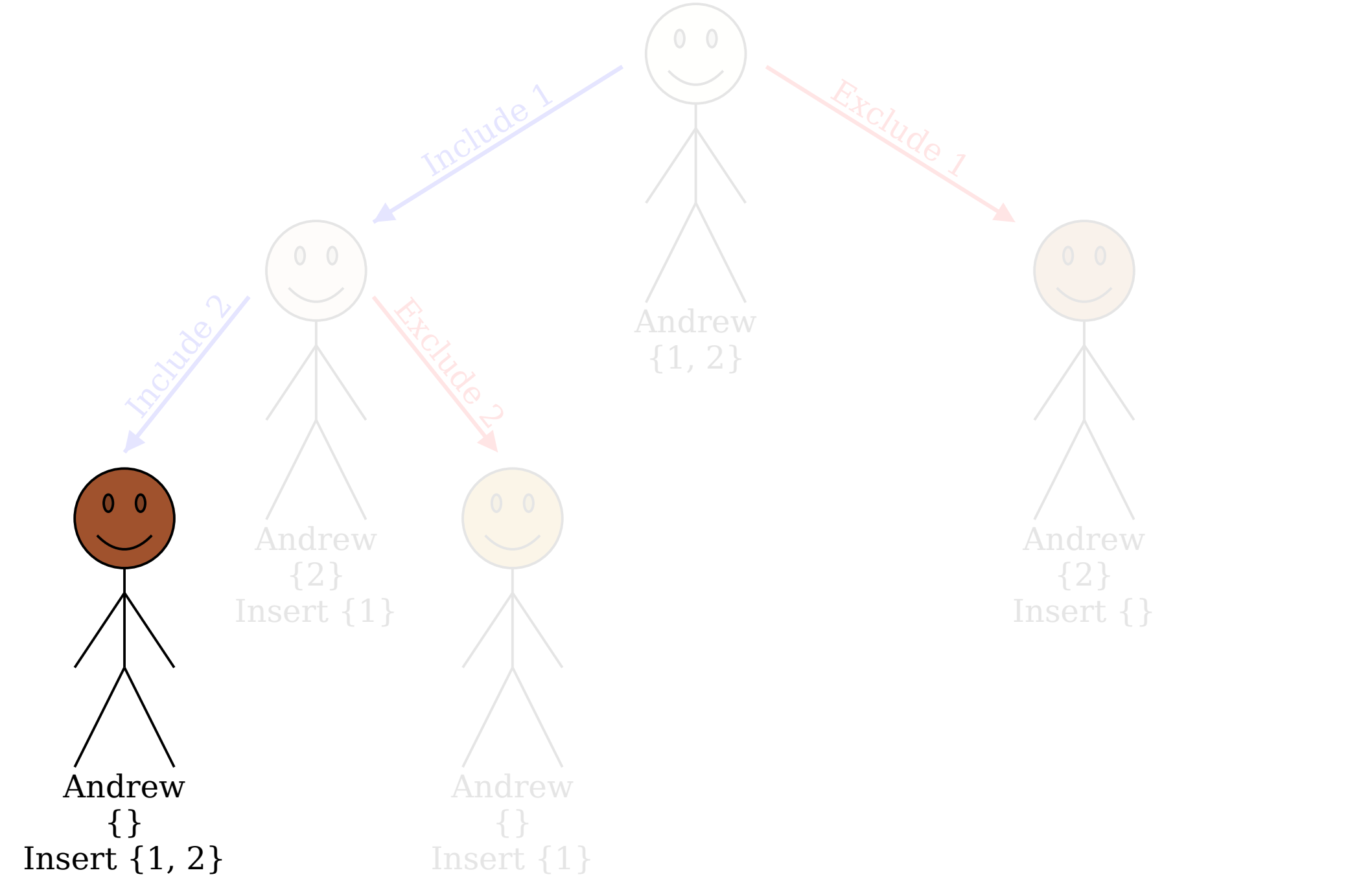
Andrews List Subsets



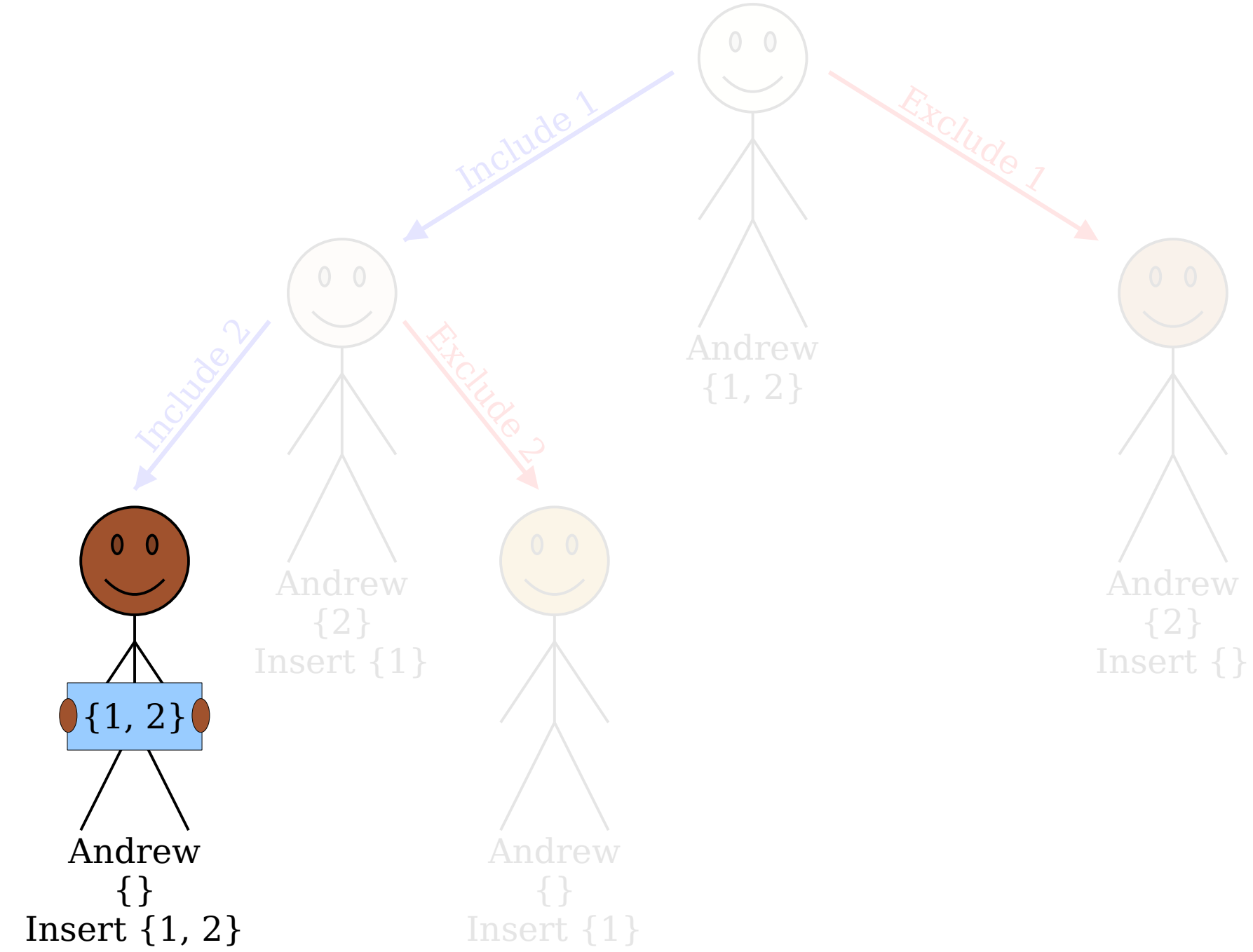
Andrews List Subsets



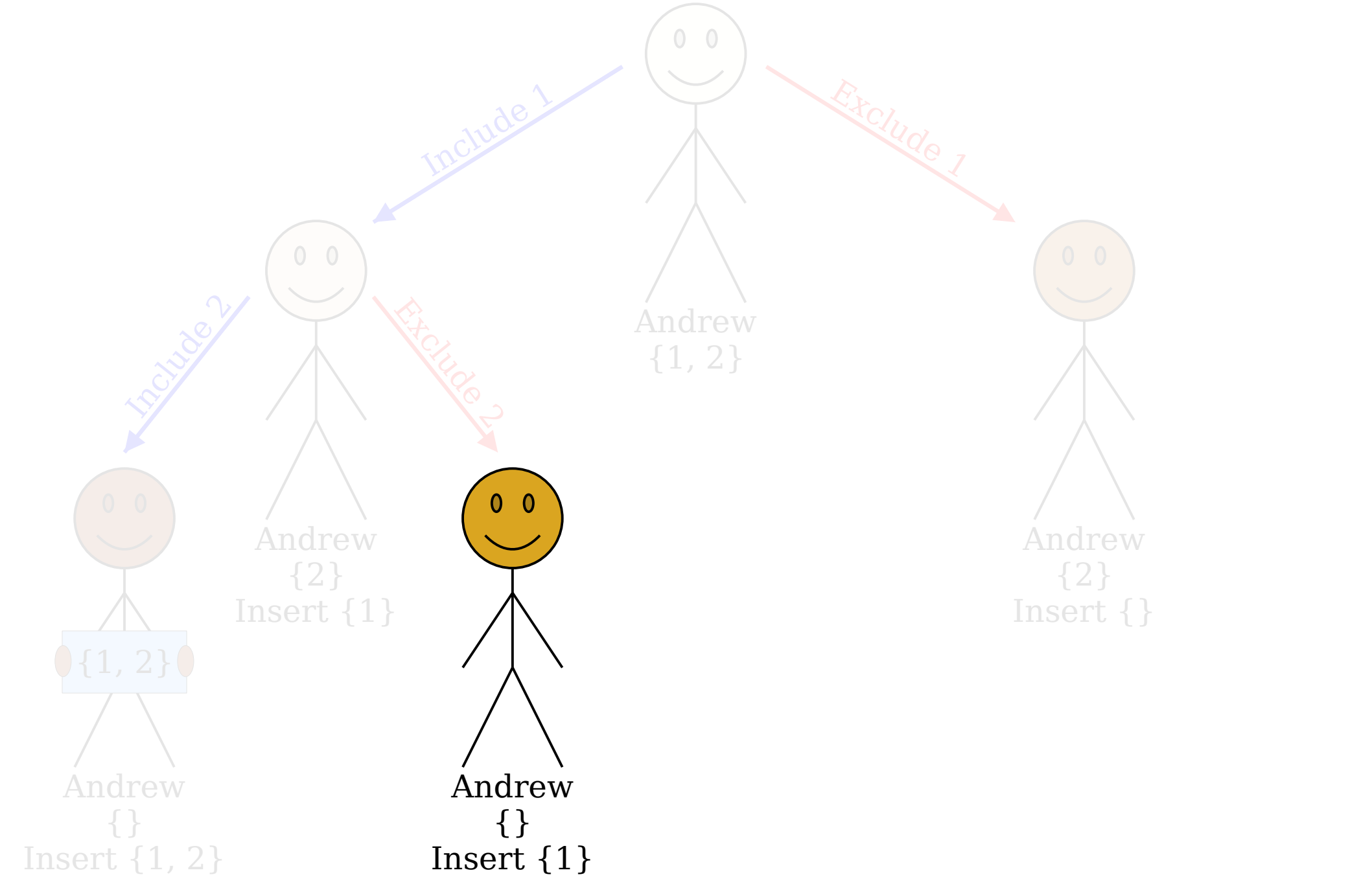
Andrews List Subsets



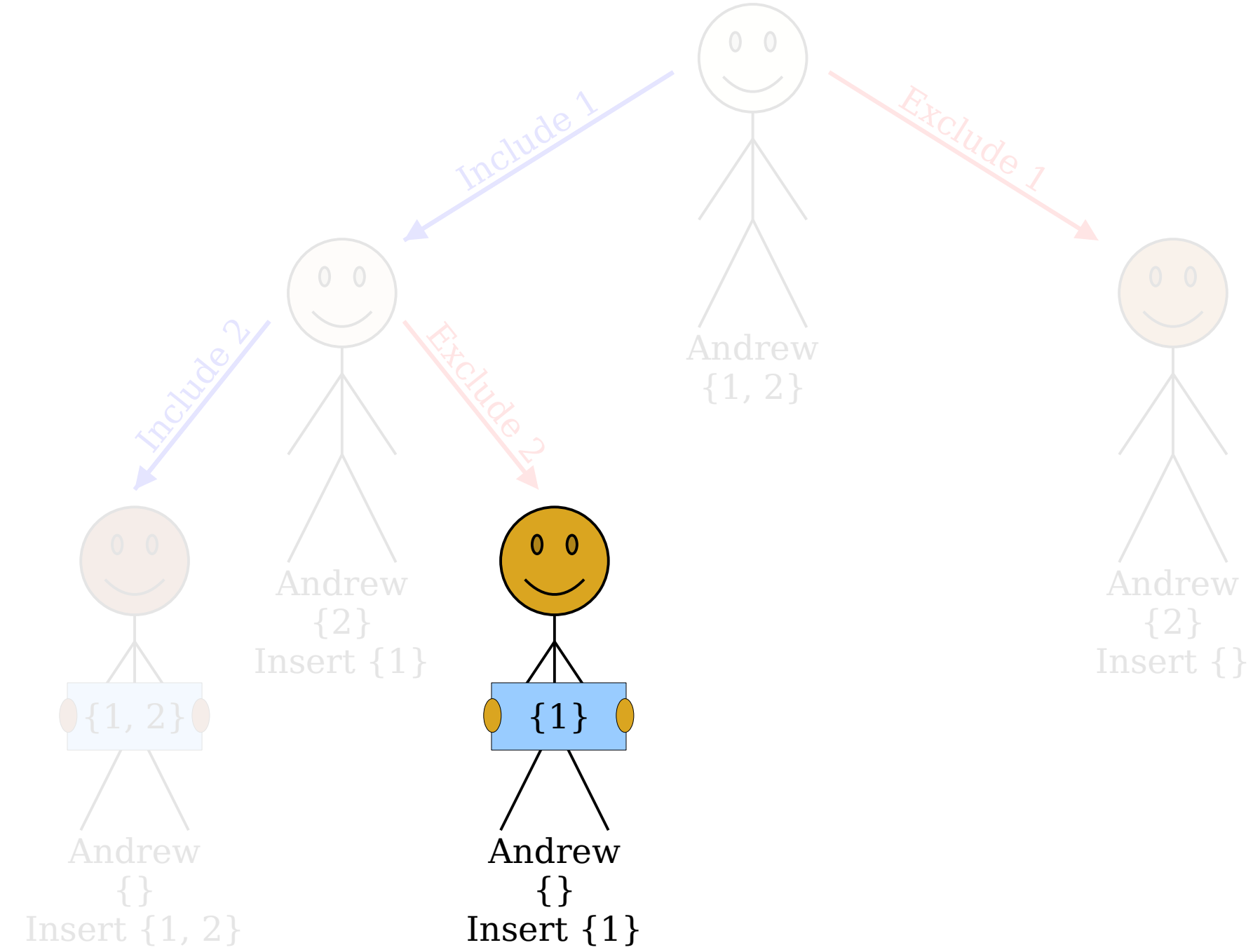
Andrews List Subsets



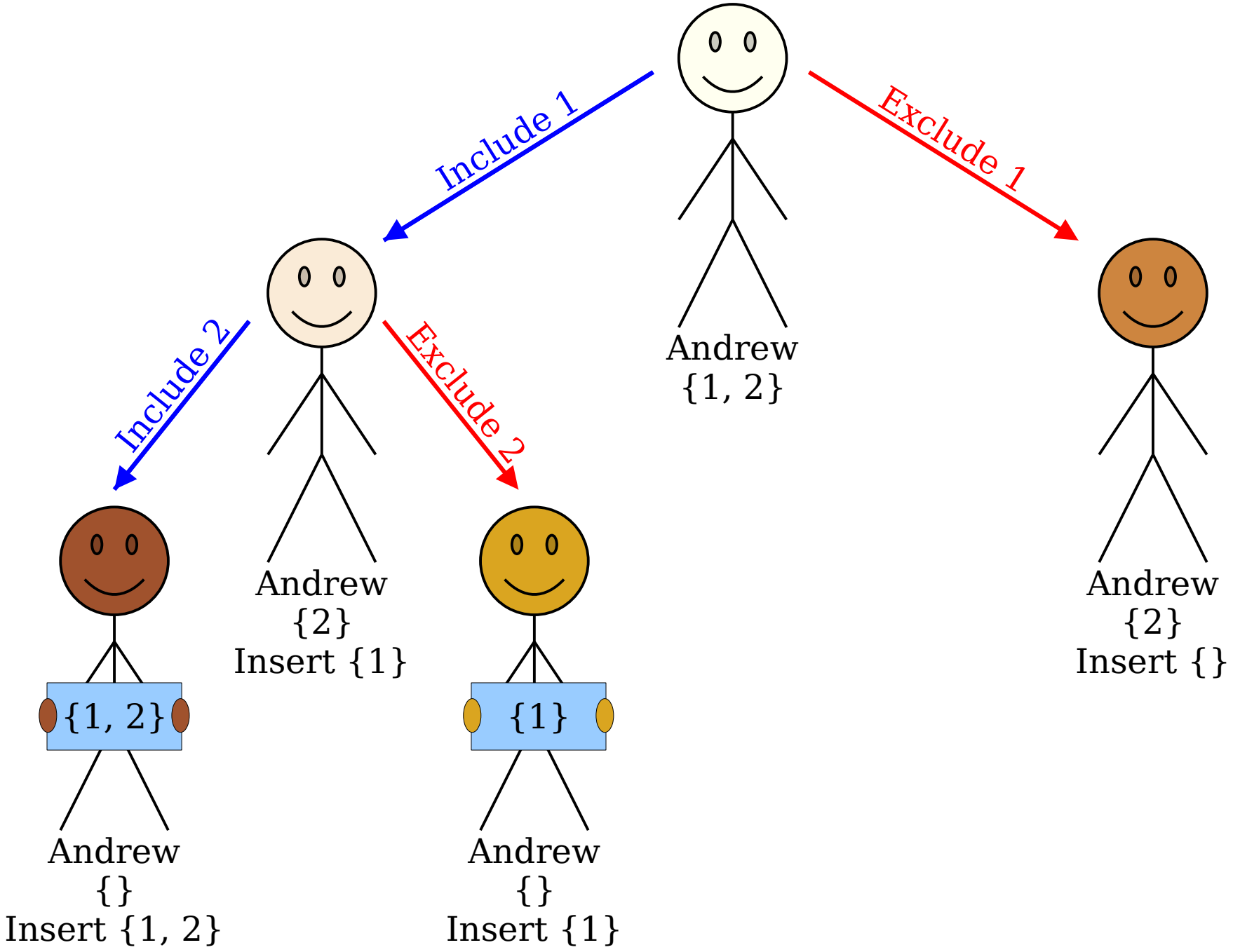
Andrews List Subsets



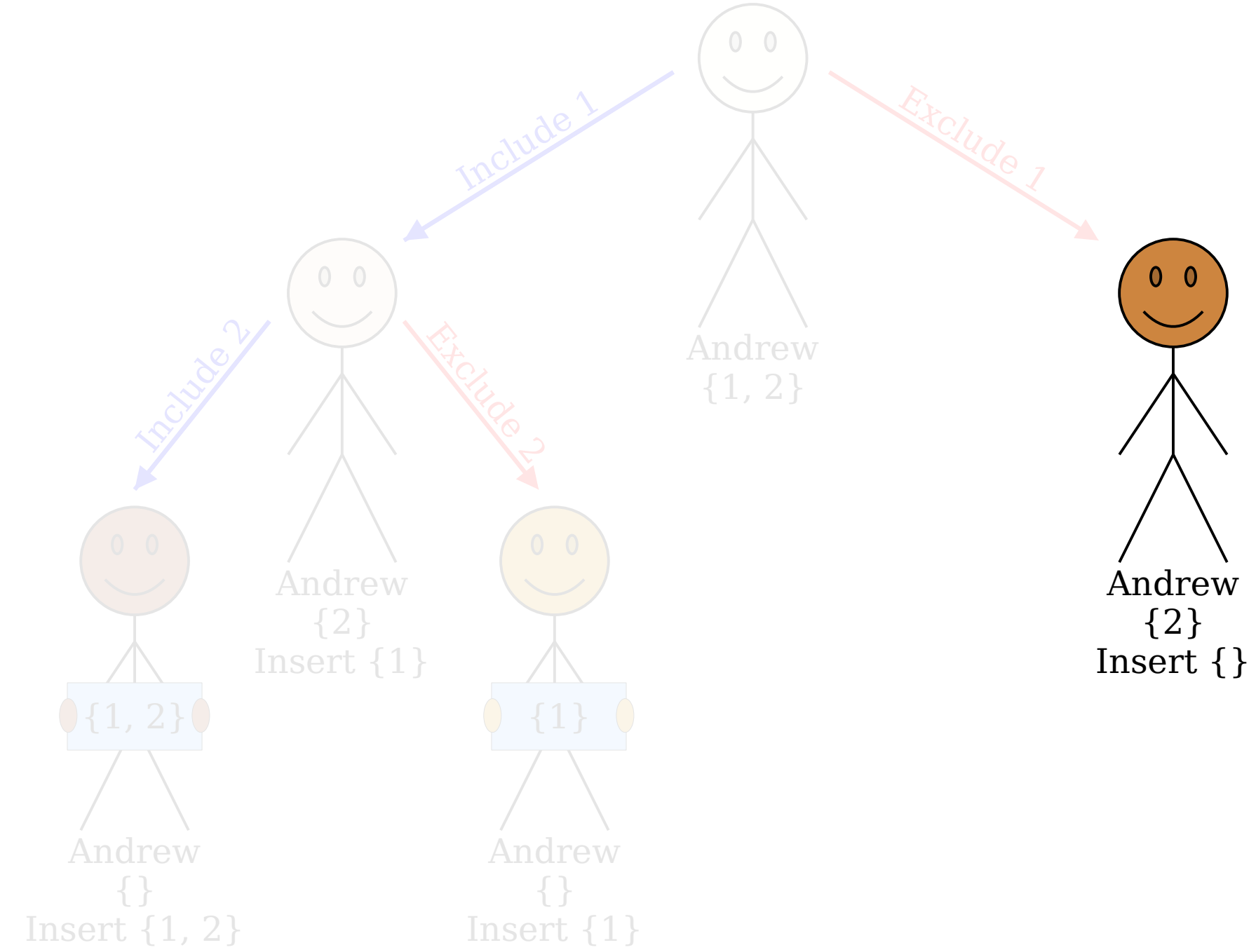
Andrews List Subsets



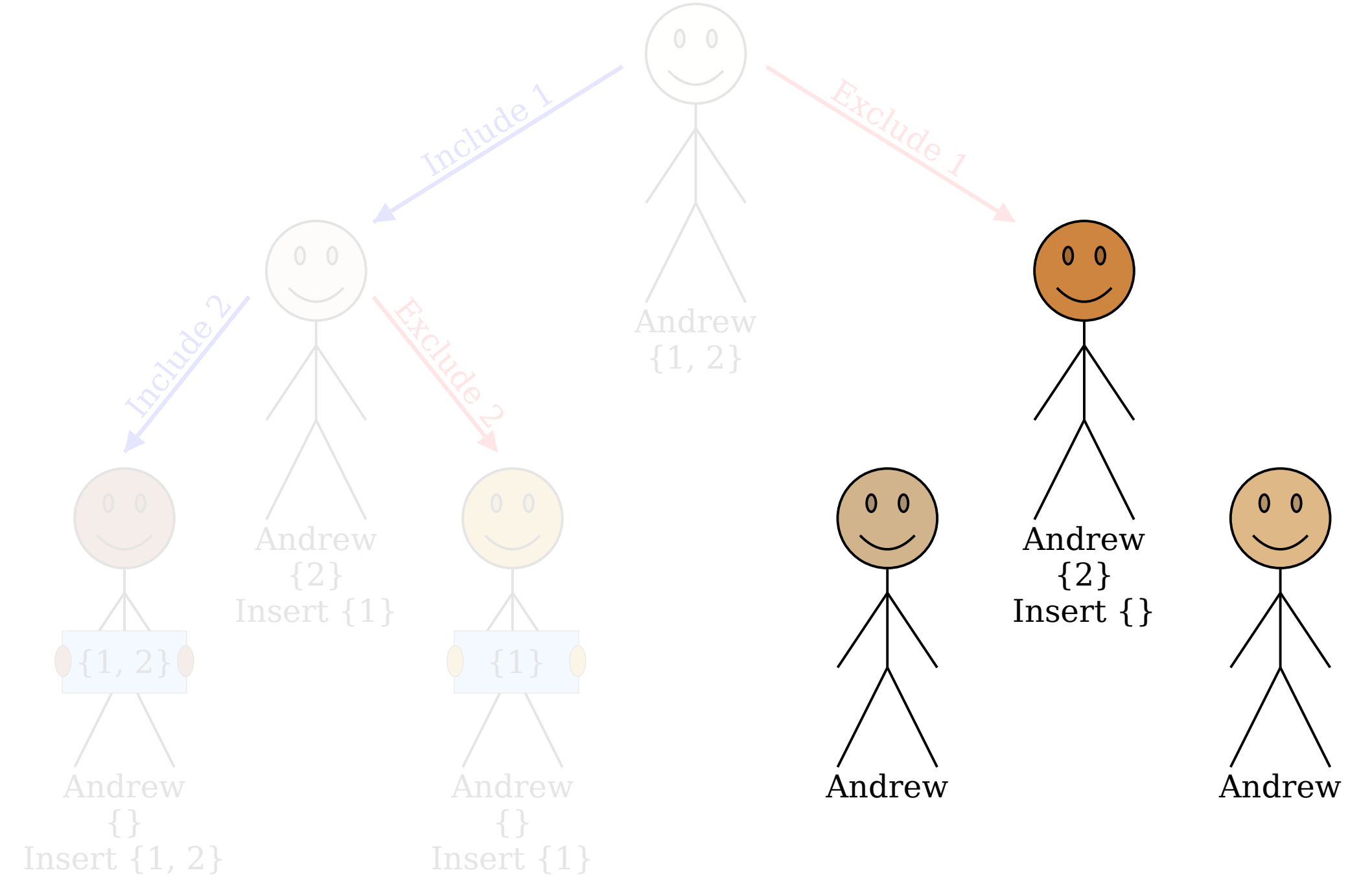
Andrews List Subsets



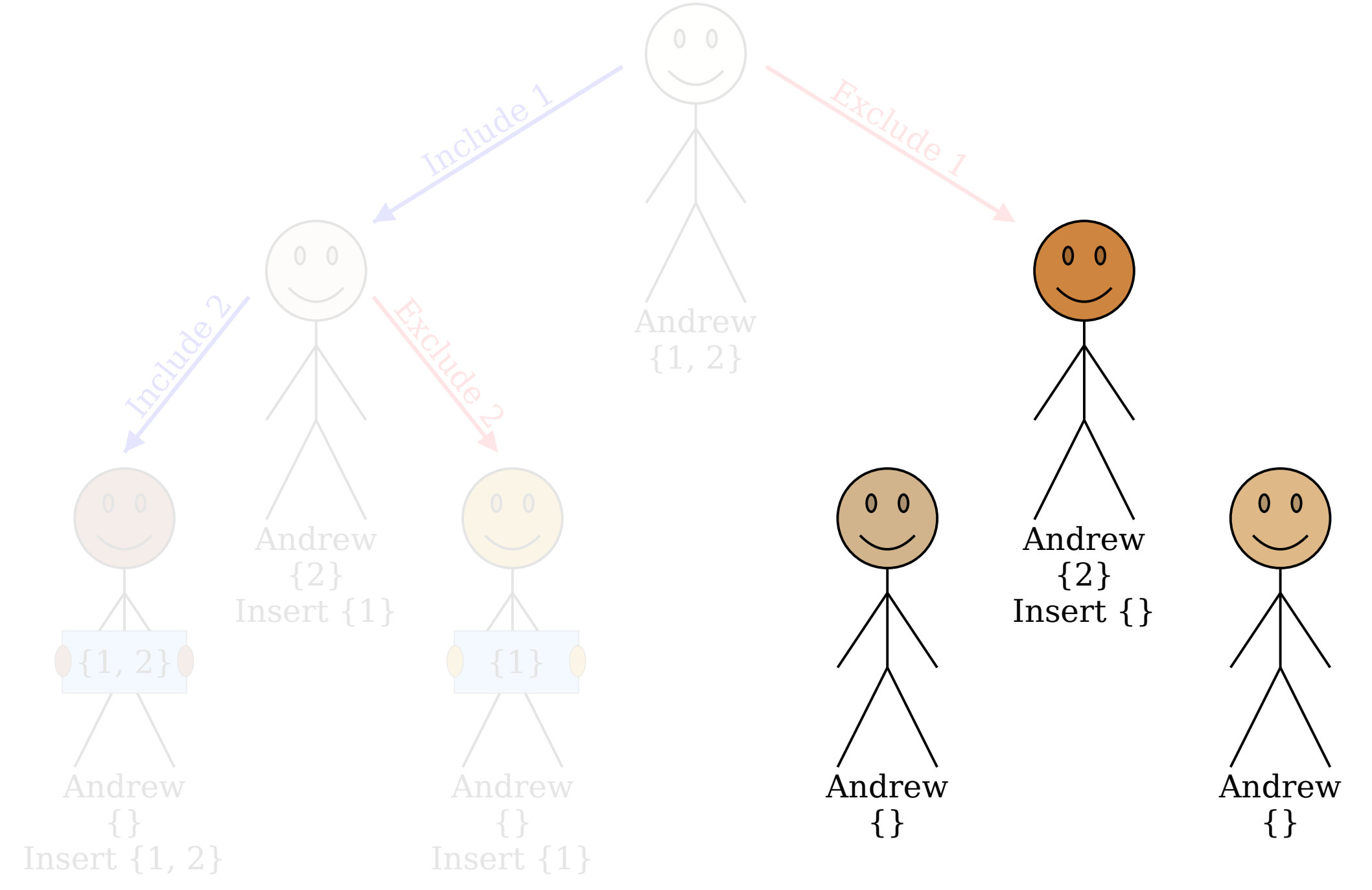
Andrews List Subsets



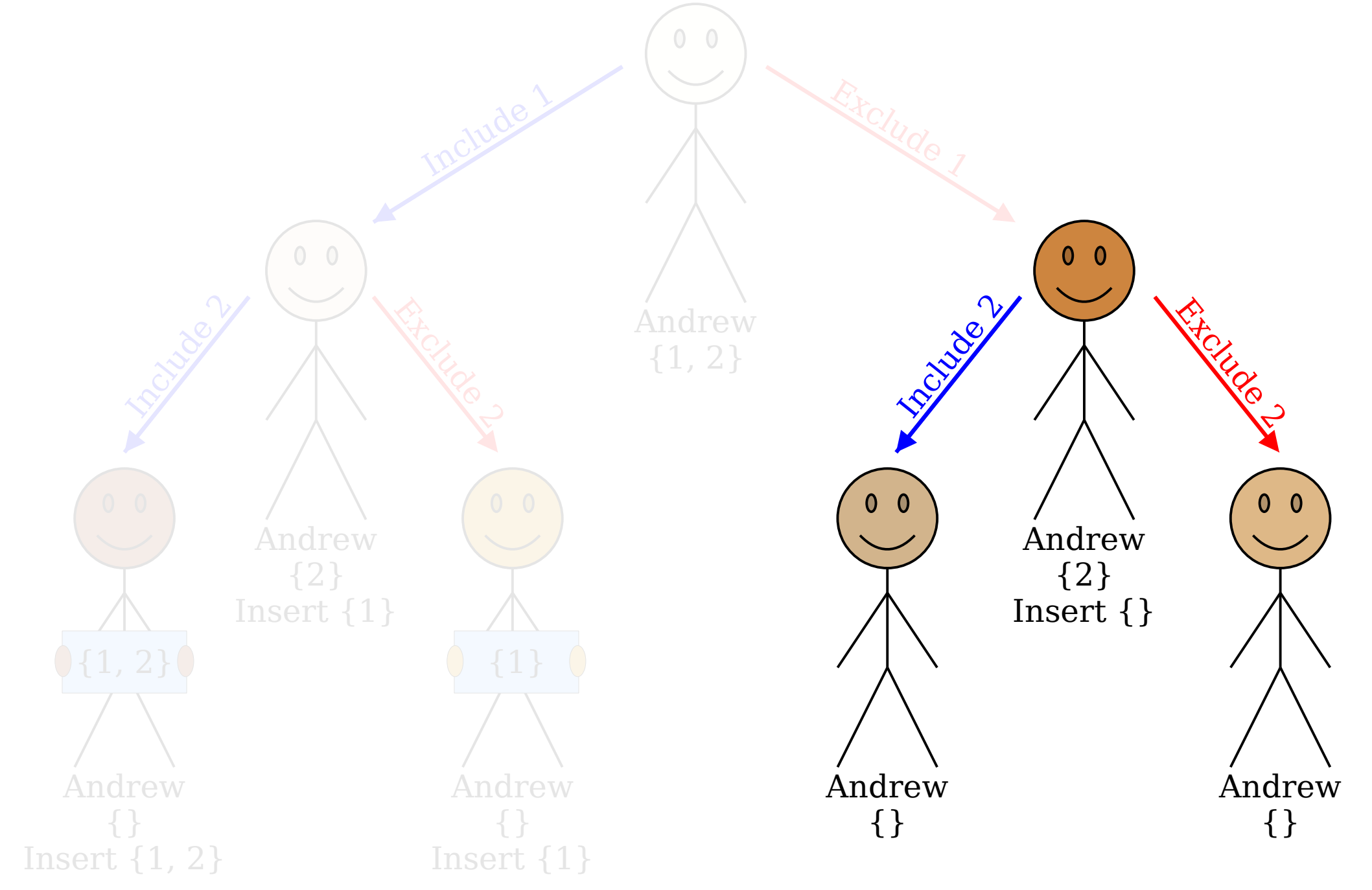
Andrews List Subsets



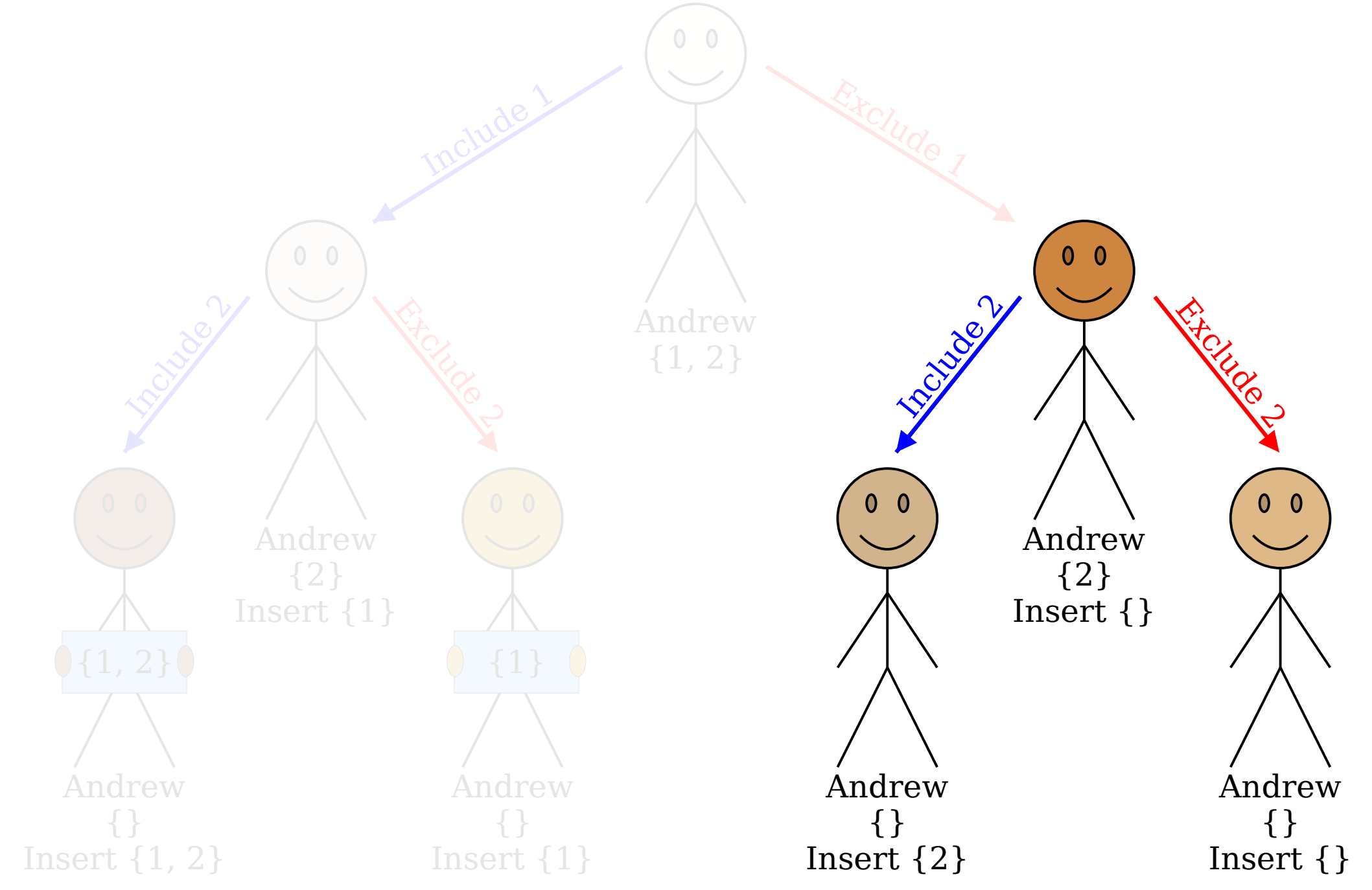
Andrews List Subsets



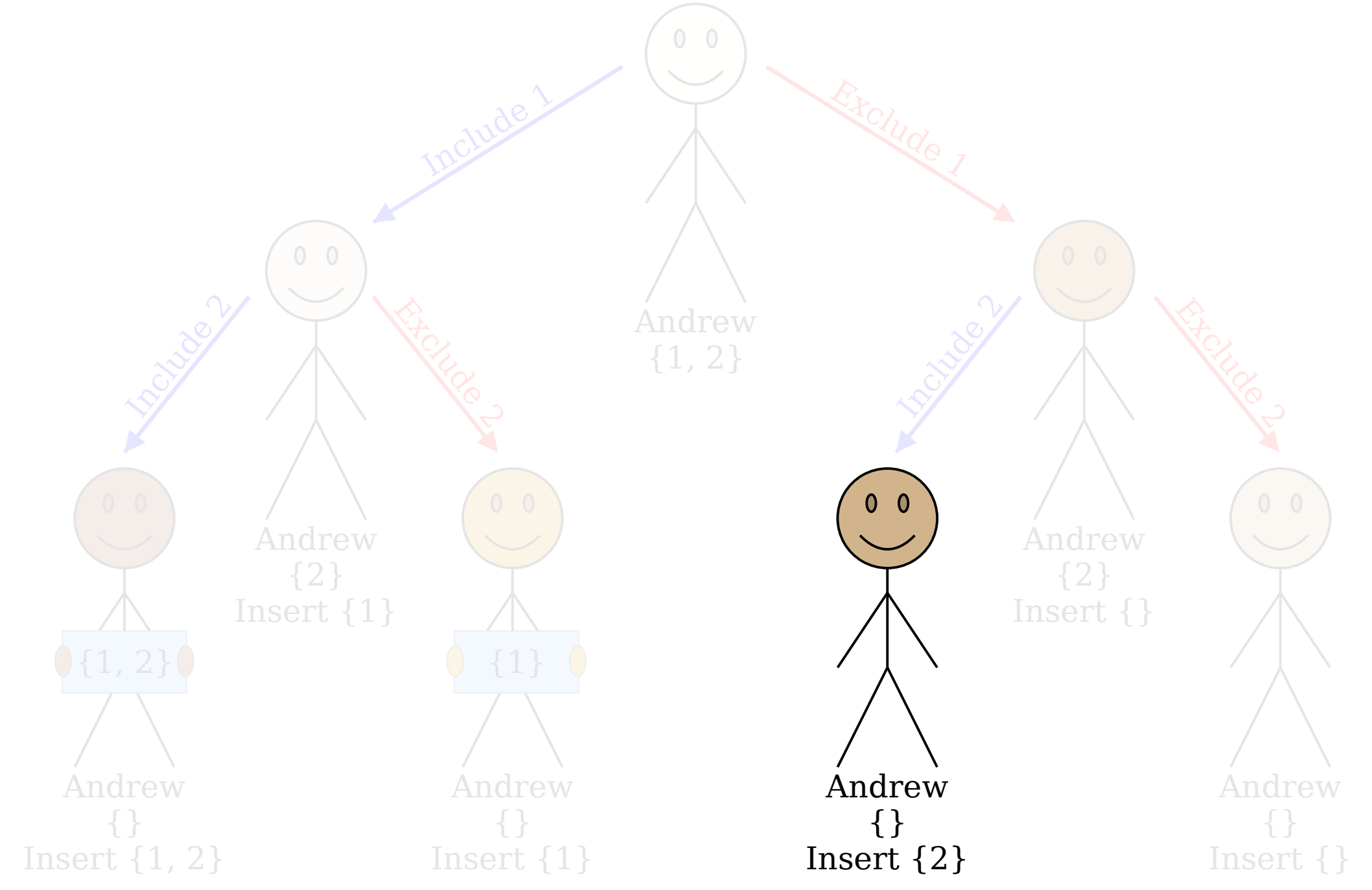
Andrews List Subsets



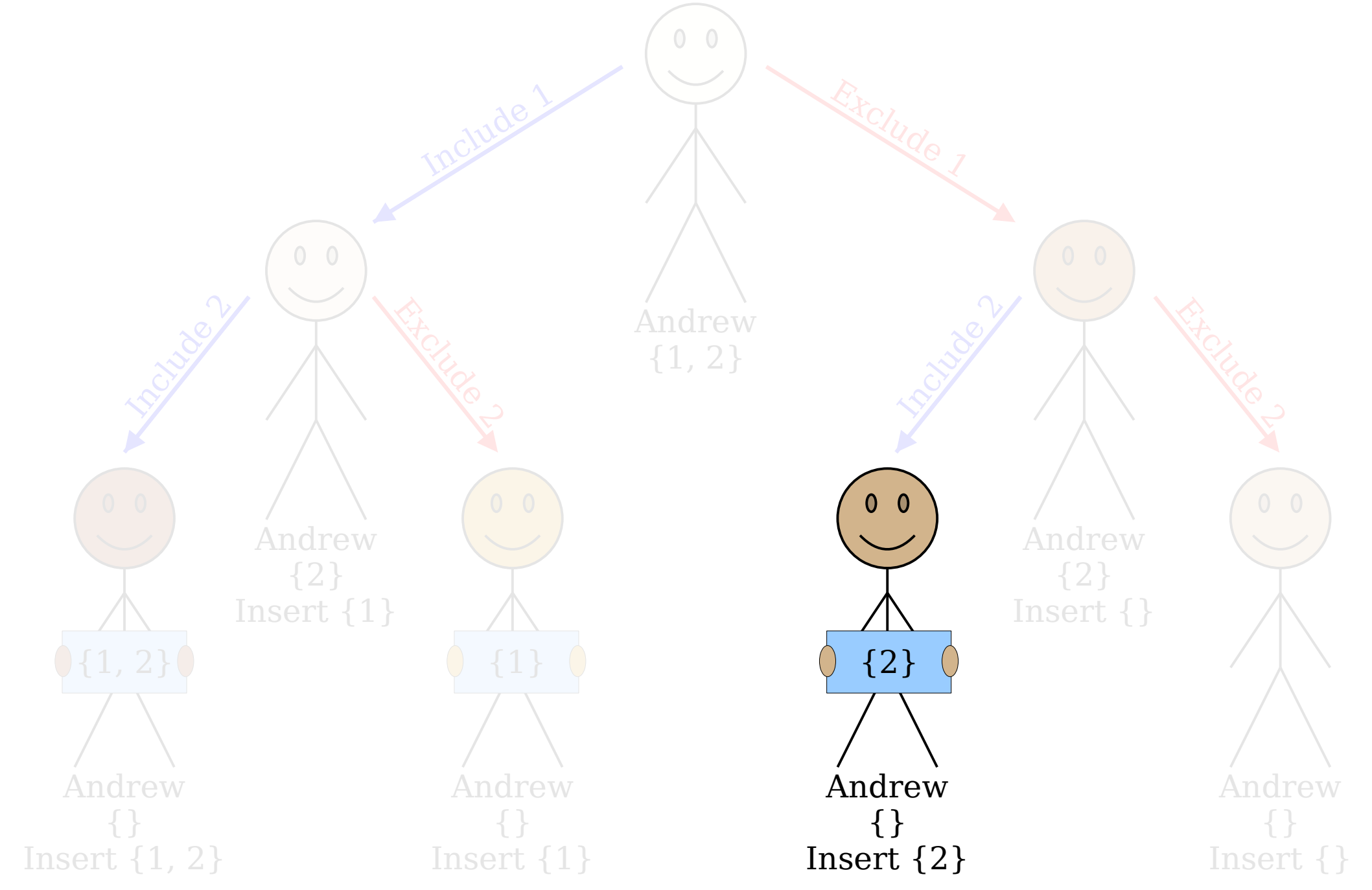
Andrews List Subsets



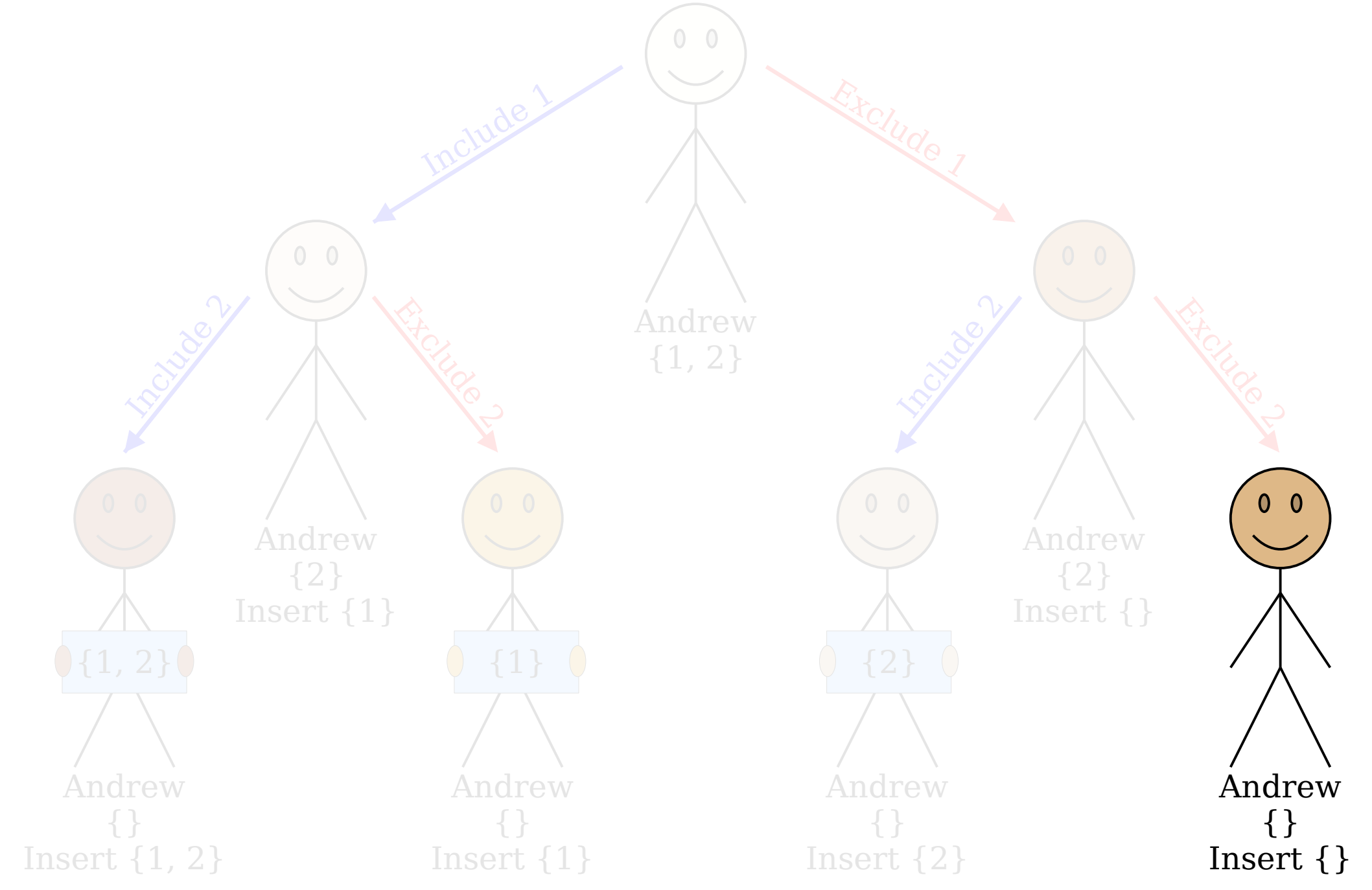
Andrews List Subsets



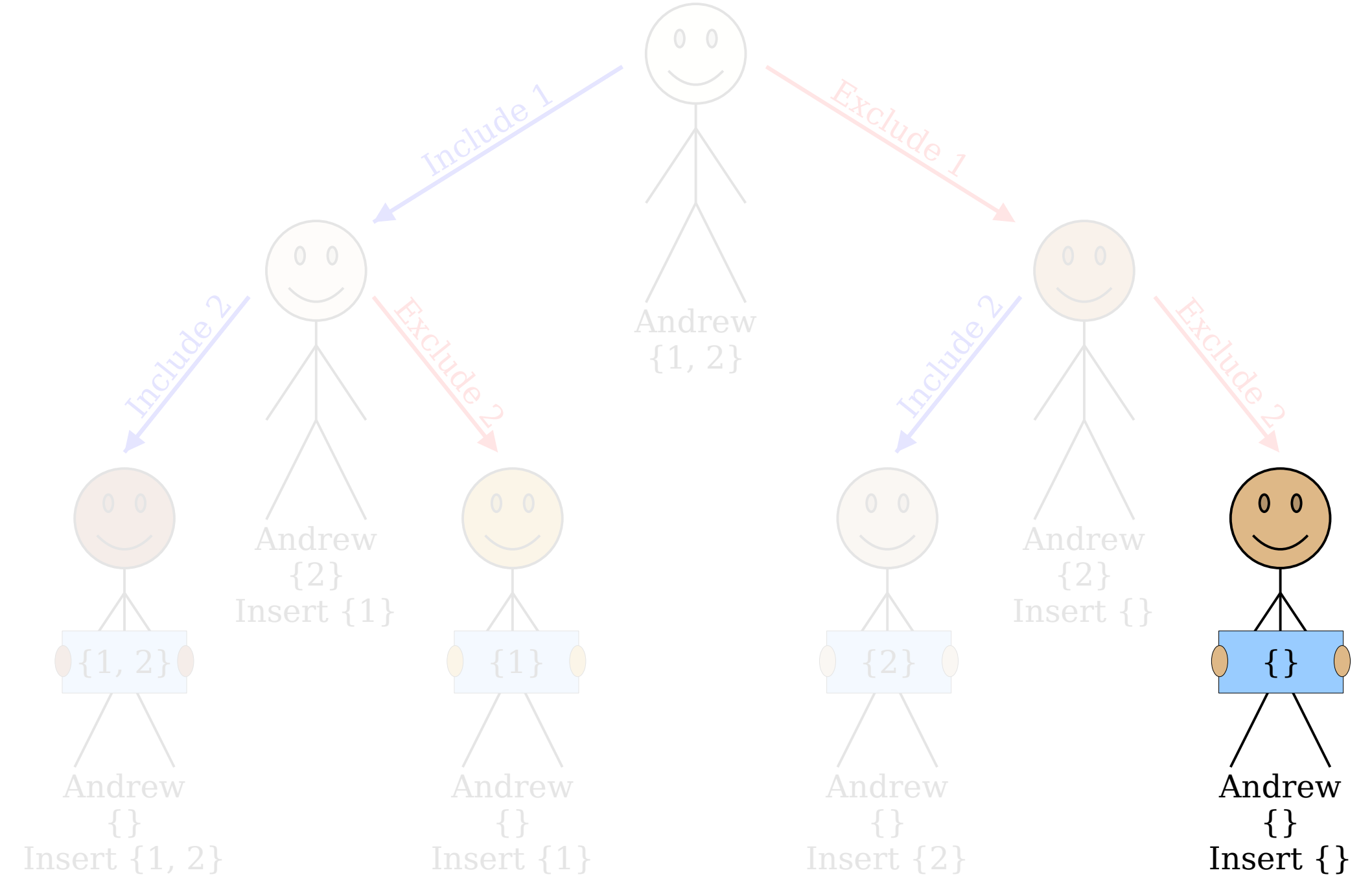
Andrews List Subsets



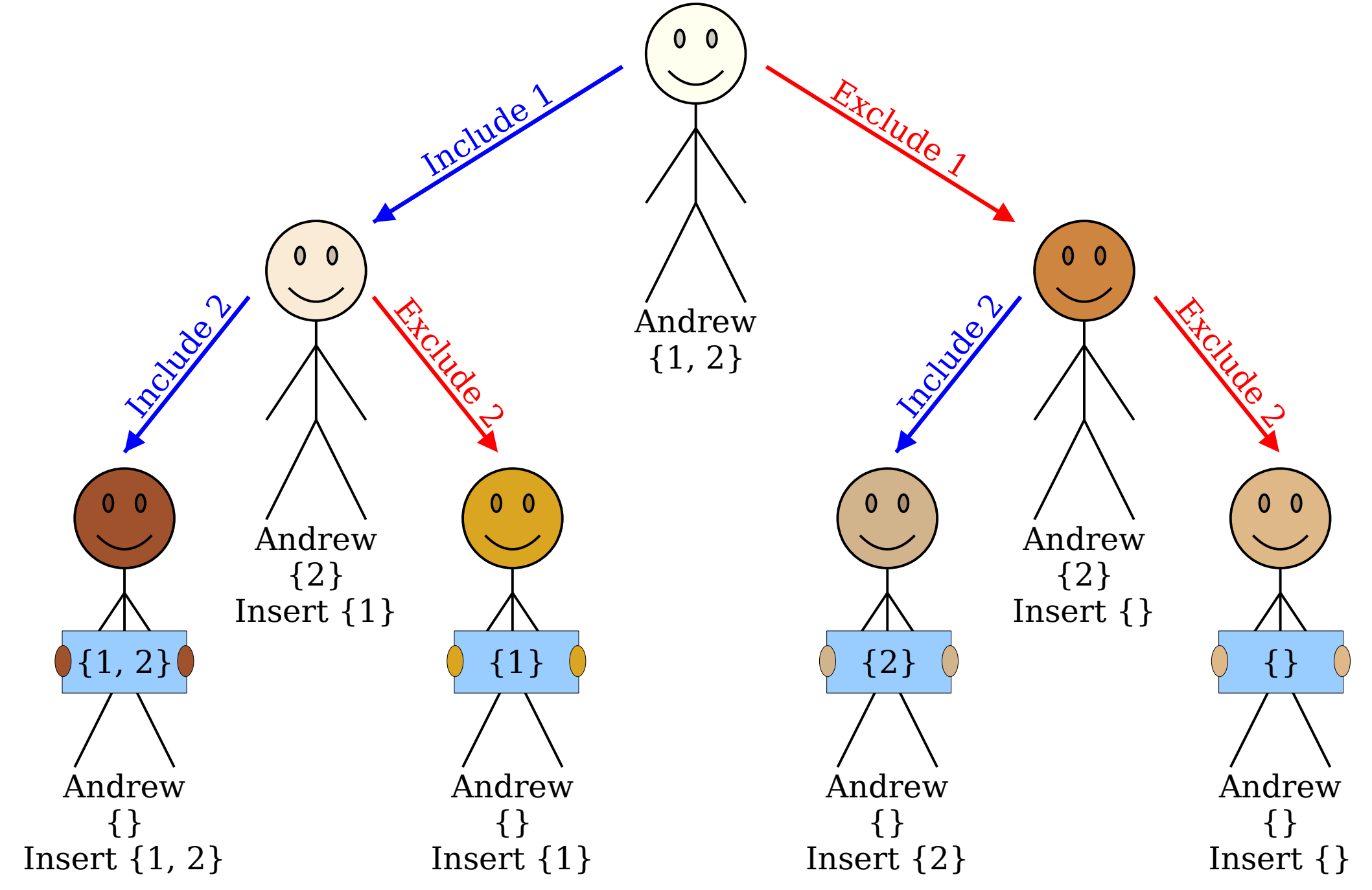
Andrews List Subsets



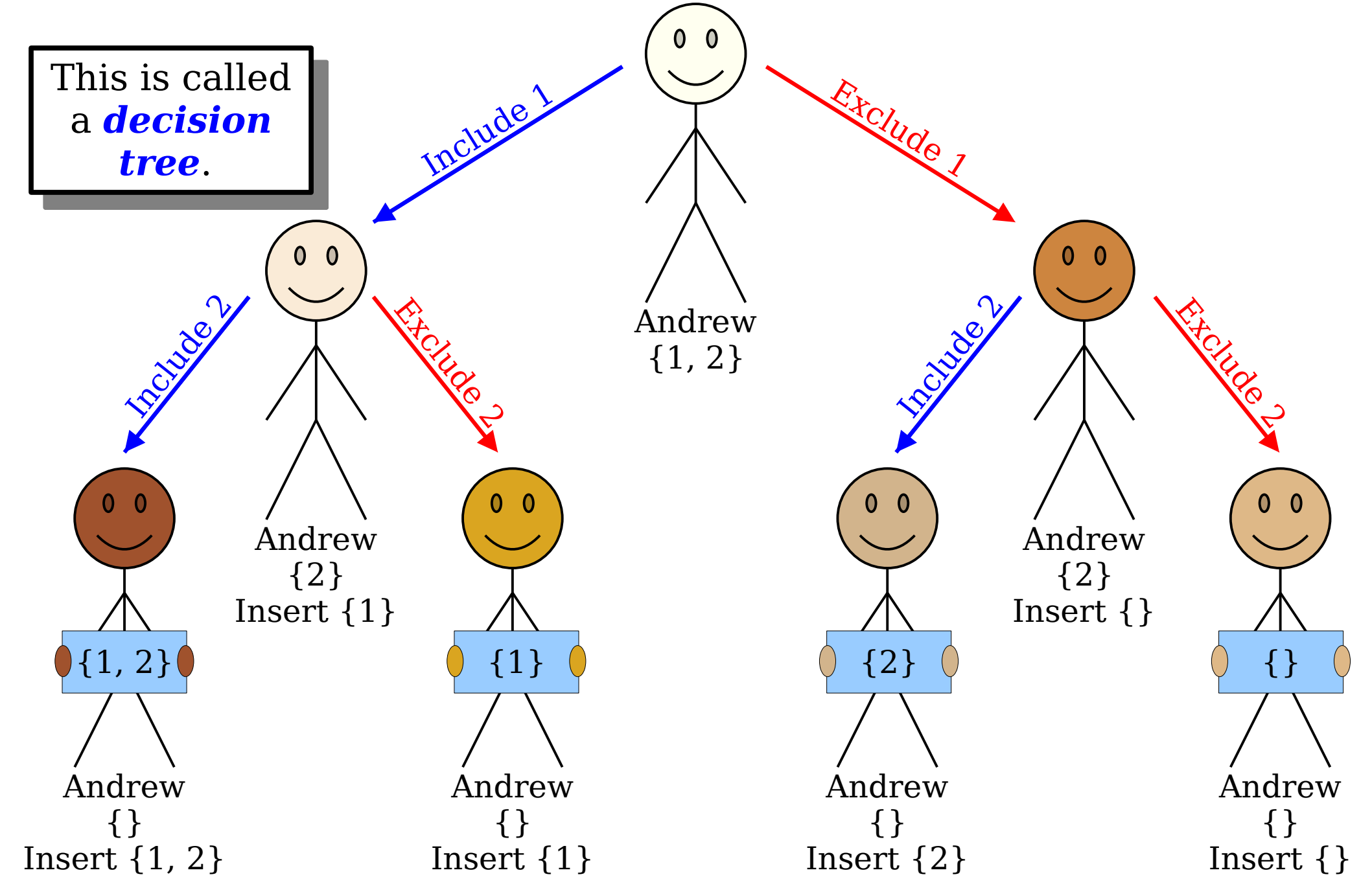
Andrews List Subsets



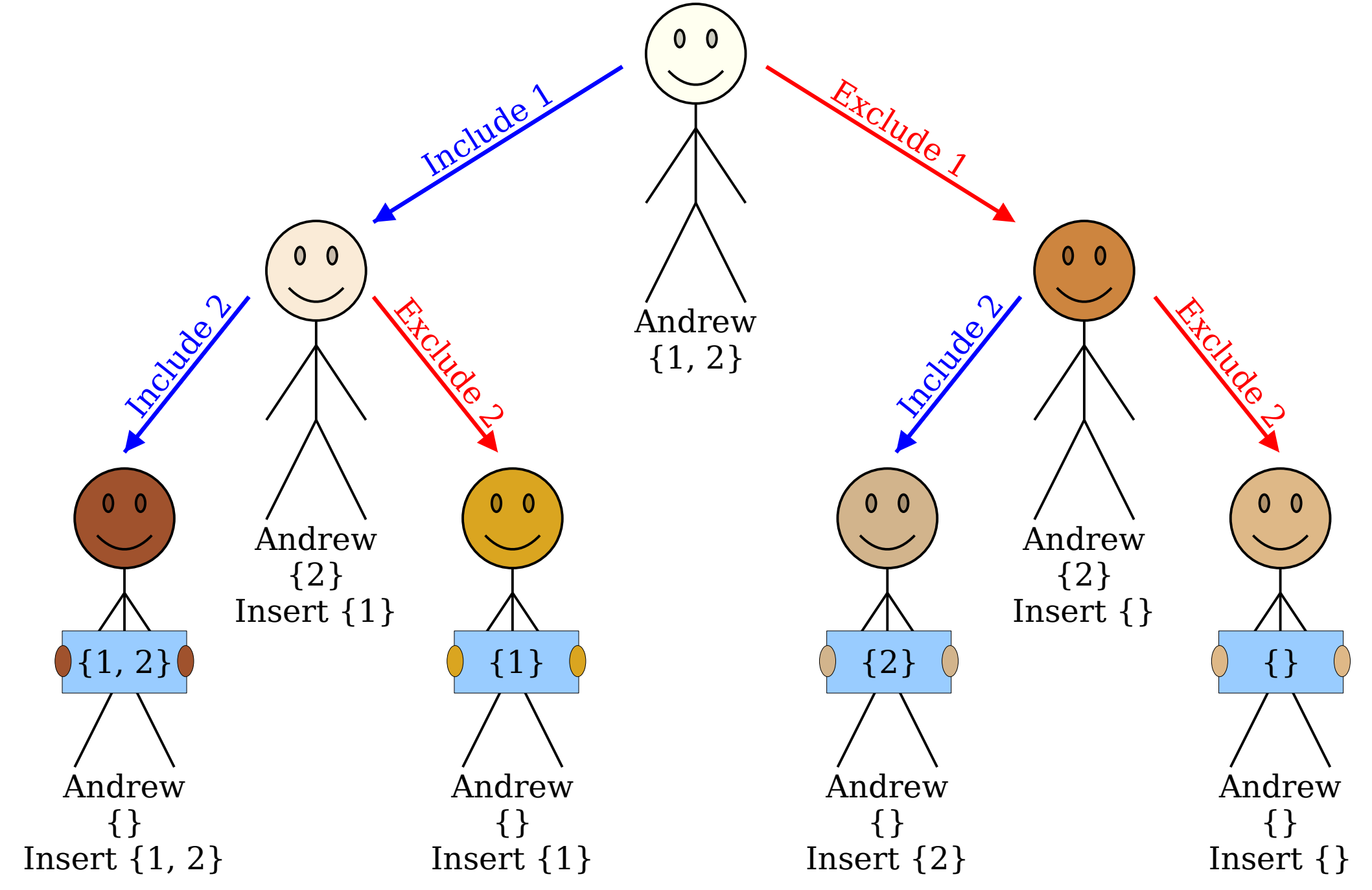
Andrews List Subsets



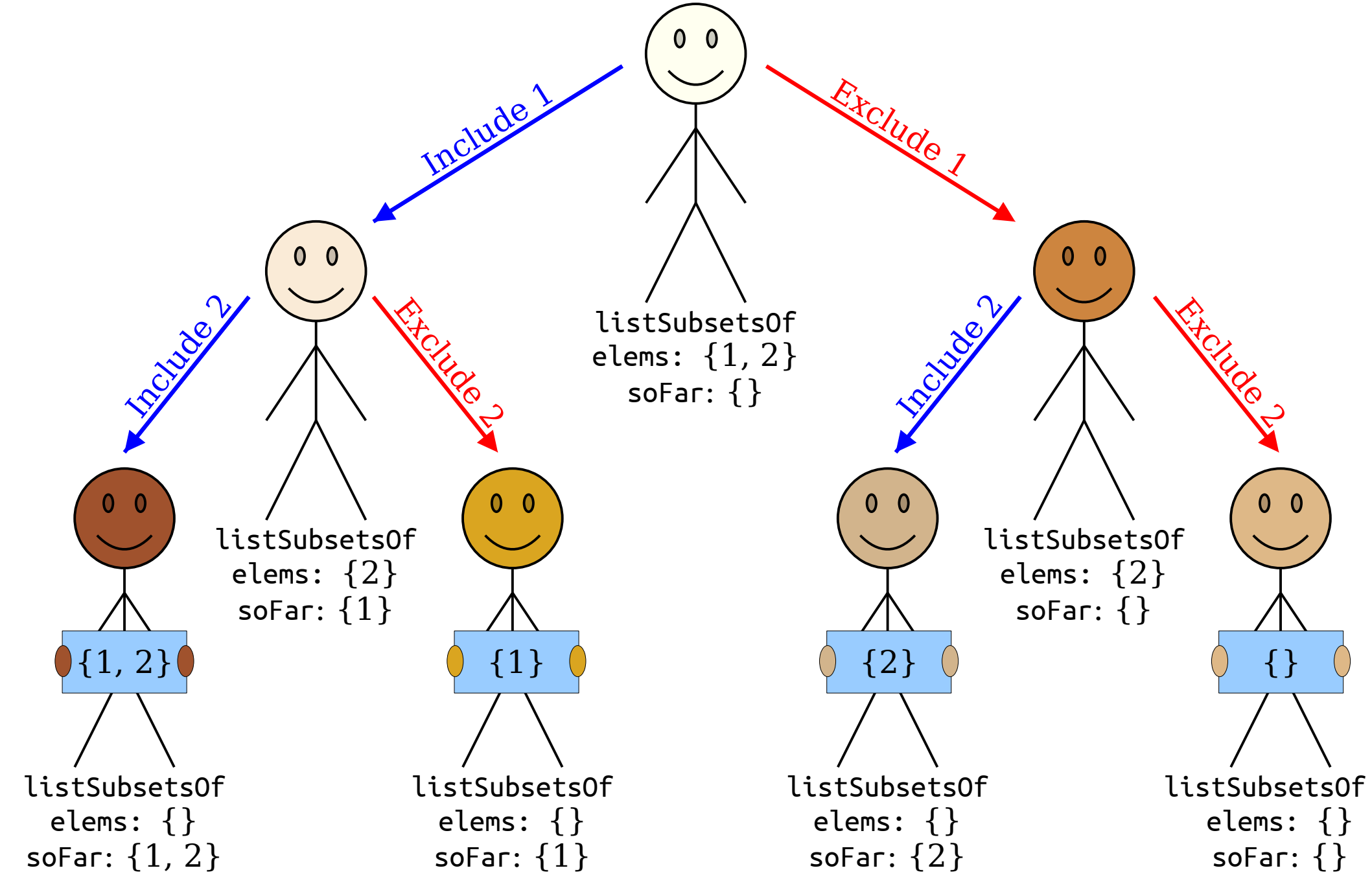
Andrews List Subsets



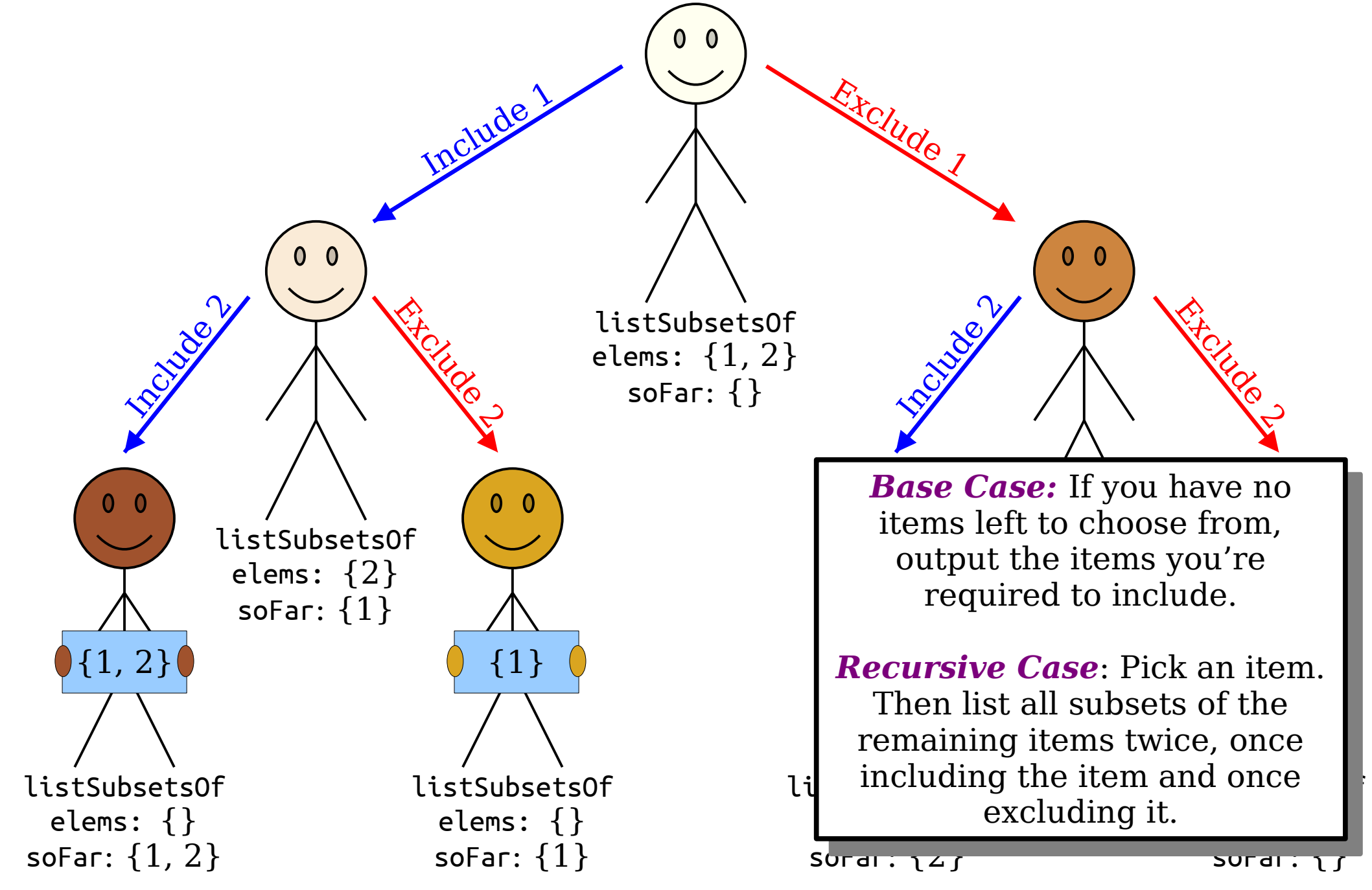
Andrews List Subsets



Andrews List Subsets



Andrews List Subsets

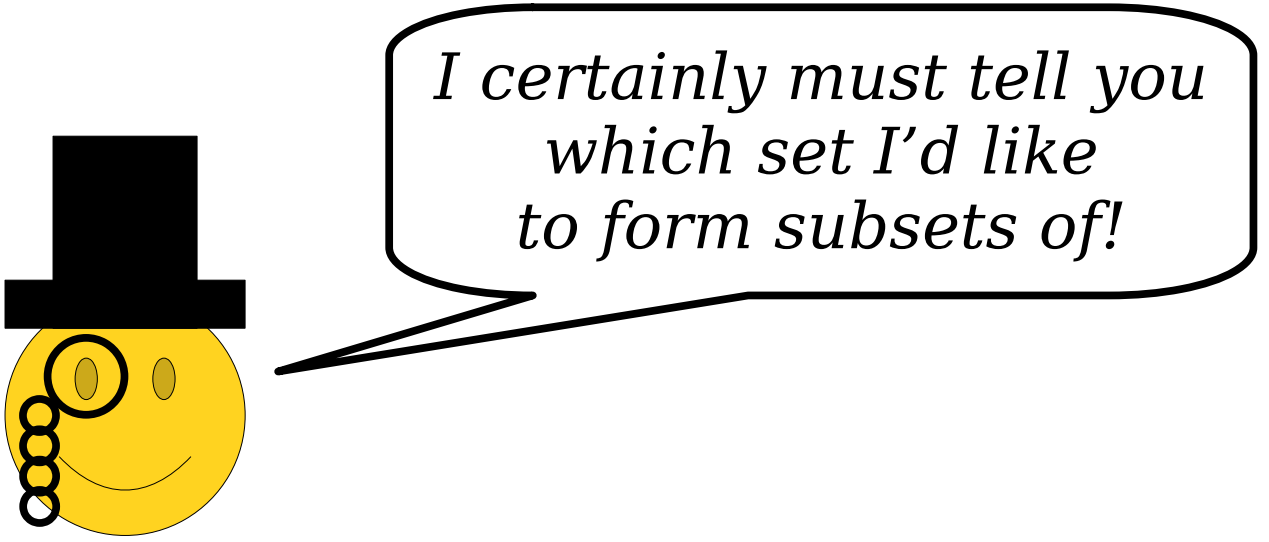


A Question of Parameters

```
listSubsetsOf({1, 2, 3}, {});
```

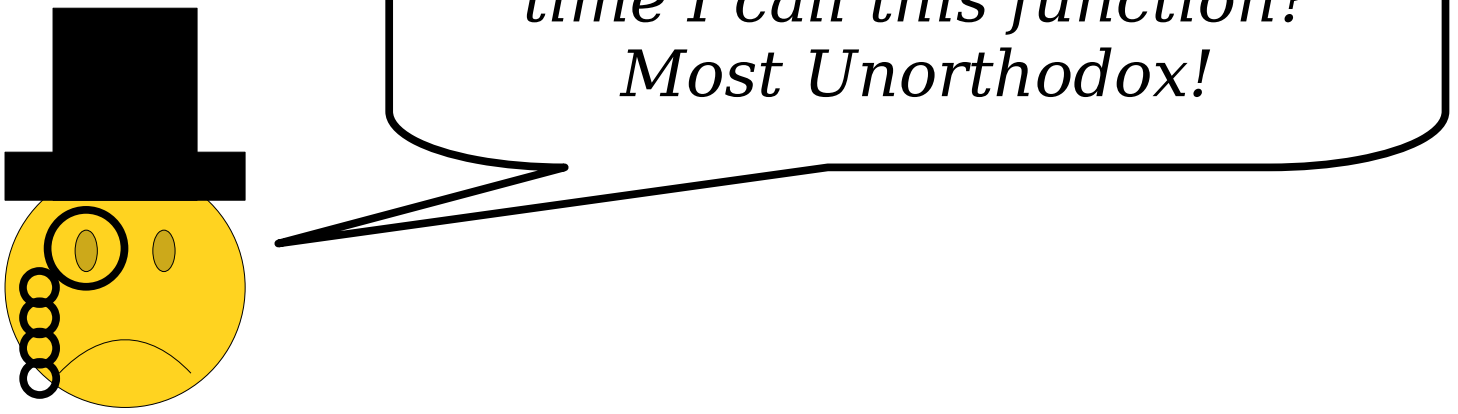
```
listSubsetsOf({1, 2, 3}, {});
```

```
listSubsetsOf({1, 2, 3}, {});
```



```
listSubsetsOf({1, 2, 3}, {});
```

```
listSubsetsOf({1, 2, 3}, {});
```

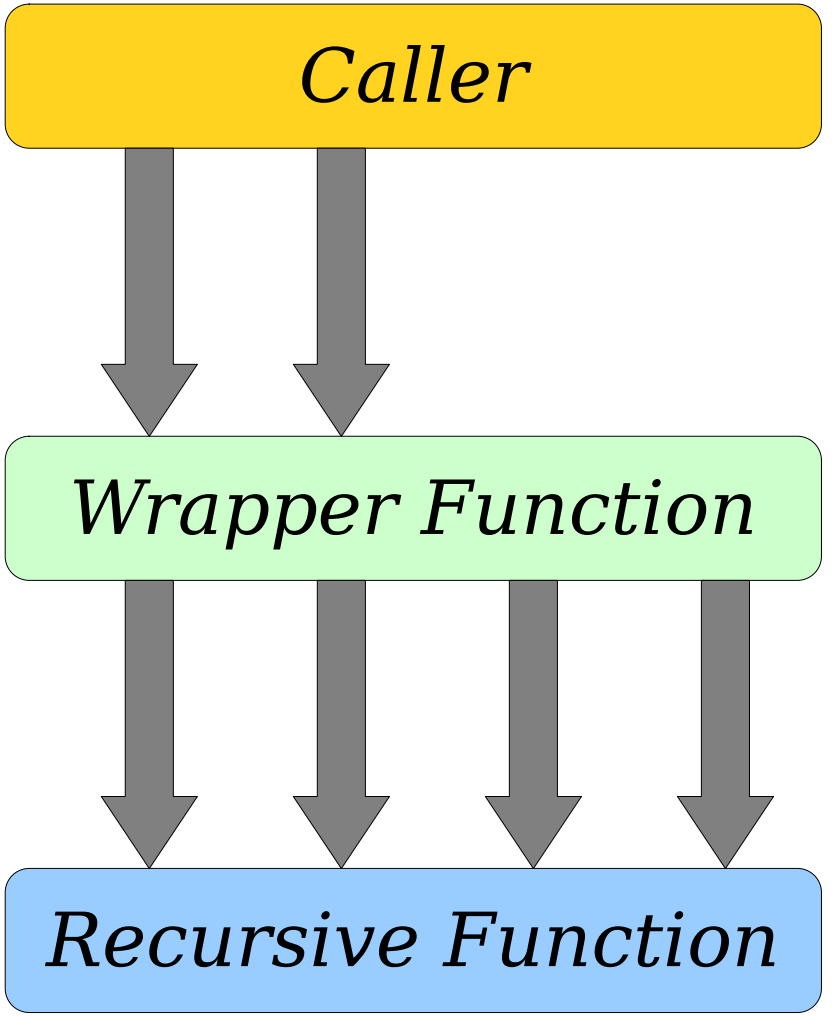


```
listSubsetsOf({1, 2, 3});
```

```
listSubsetsOf({1, 2, 3});
```

Wrapper Functions

- Some recursive functions need extra arguments as part of an implementation detail.
  - In our case, the set of things chosen so far is not something we want to expose.
- A ***wrapper function*** is a function that does some initial prep work, then fires off a recursive call with the right arguments.



Summary For Today

- Making the ***recursive leap of faith*** and trusting that your recursive calls will perform as expected helps simplify writing recursive code.
- A ***decision tree*** models all the ways you can make choices to arrive at a set of results.
- A ***wrapper function*** makes the interface of recursive calls cleaner and harder to misuse.

Your Action Items

- ***Read Chapter 8.***
  - There's a lot of great information there about recursive problem-solving, and it's a great resource.
- ***Finish Assignment 2***
  - If you're following our suggested timetable, at this point you'll have finished Rosetta Stone and will have started working on Rising Tides.
  - Come to LaIR or ask on EdStem if you have any questions!

Next Time

- ***Iteration + Recursion***
  - Combining two techniques together.
- ***Enumerating Permutations***
  - What order should we perform tasks in?

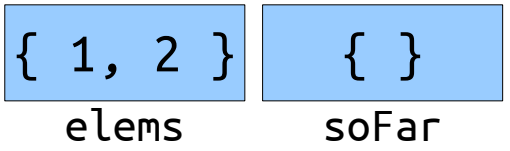
Appendix: Tracing the Recursion

```
int main() {

    listSubsetsOf({ 1, 2 }, { });
    return 0;

}
```

```
int main() {

    listSubsetsOf({ 1, 2 }, { });
    return 0;

}
```

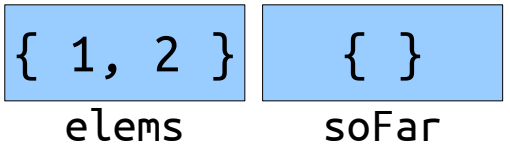
```
int main() {

}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



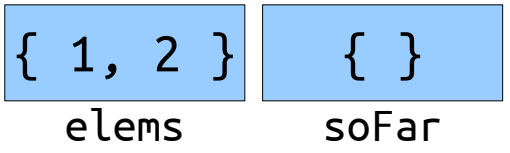
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



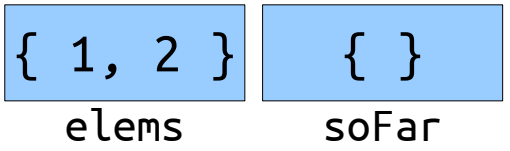
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



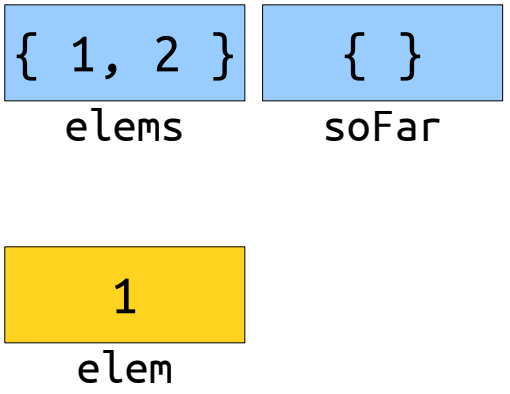
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



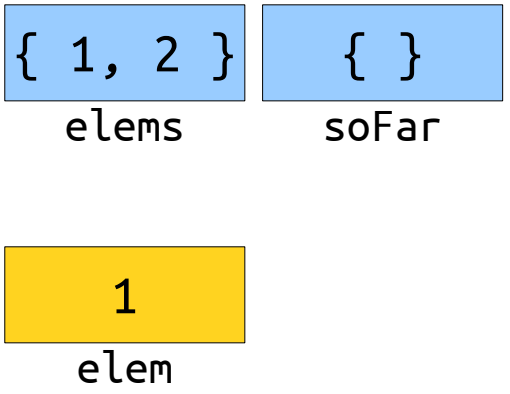
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



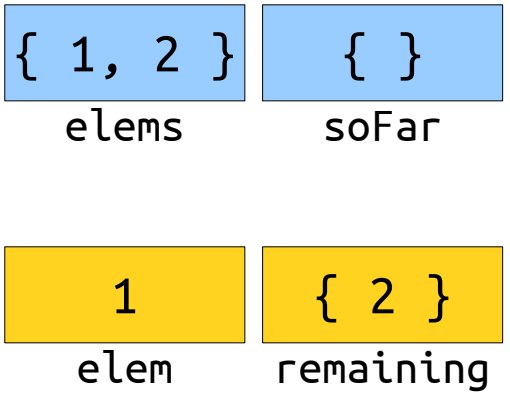
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



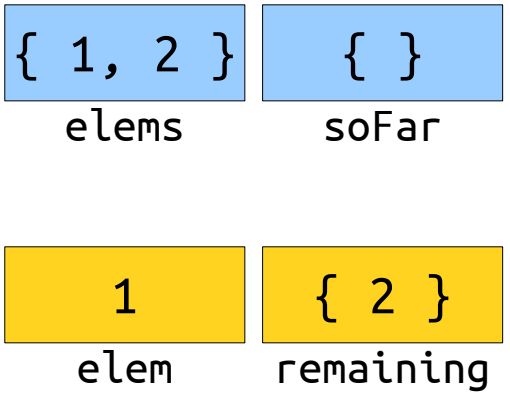
```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { 1 }

elems soFar

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 }

{ }

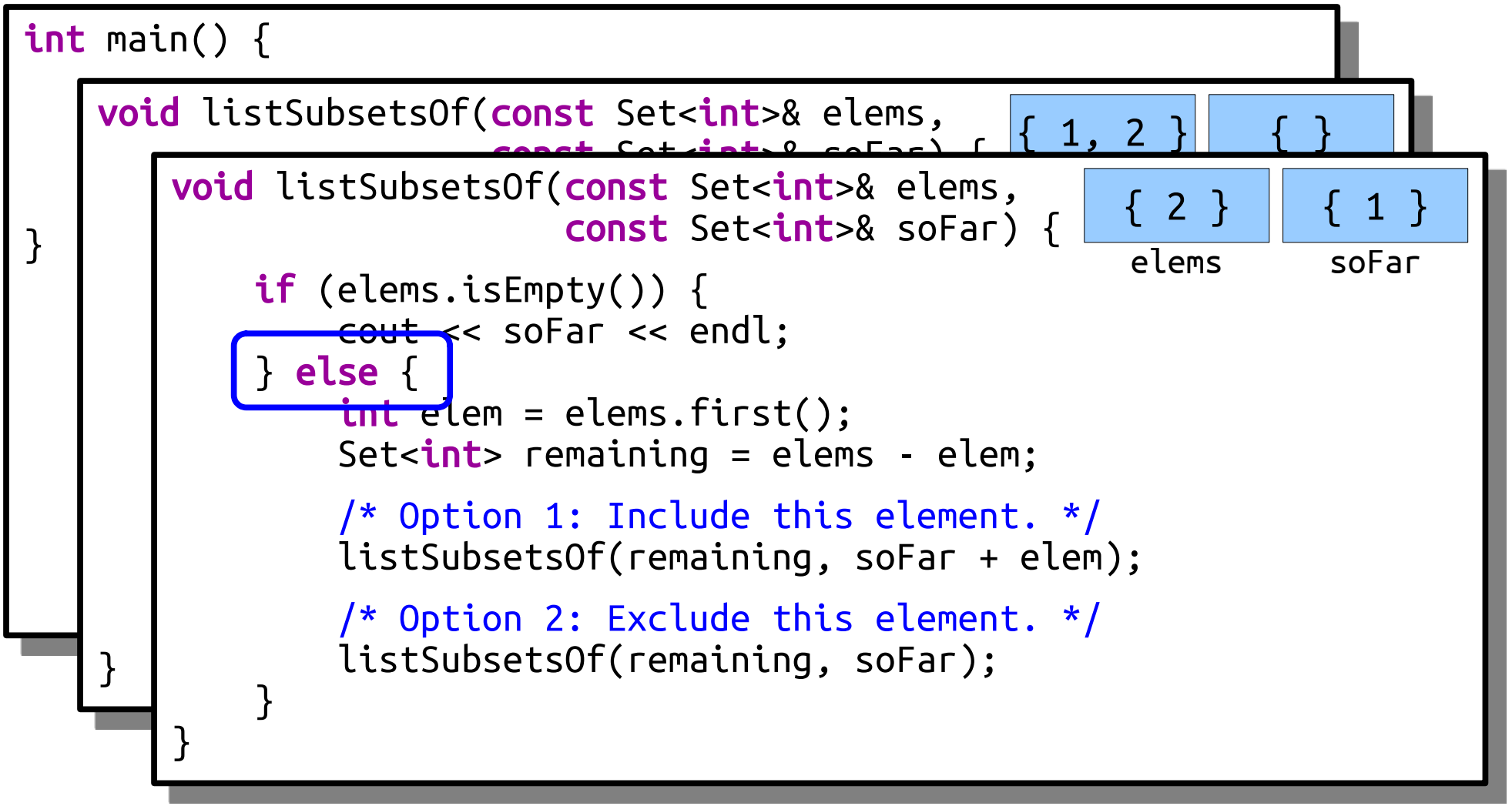
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

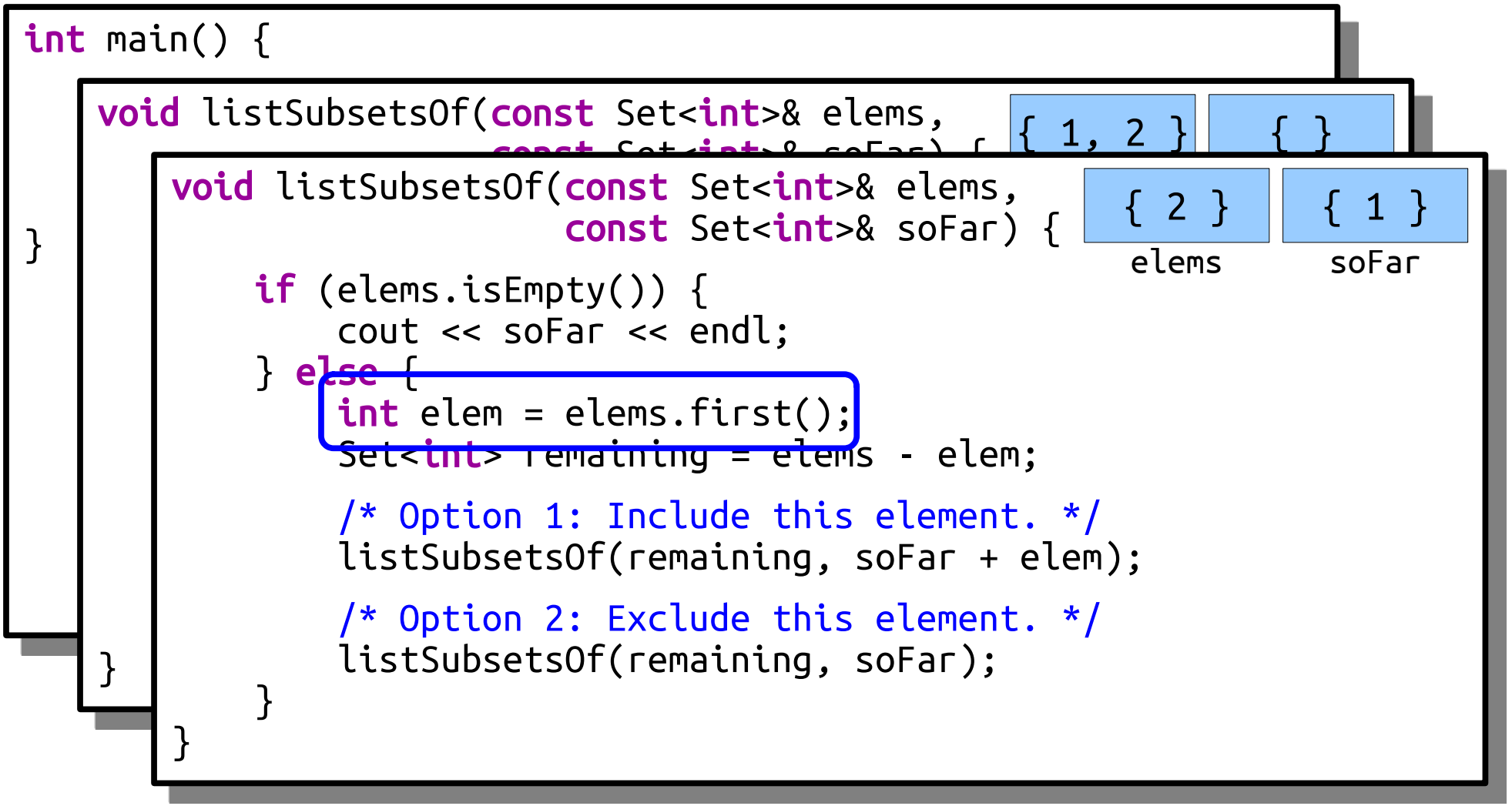
{ 2 }

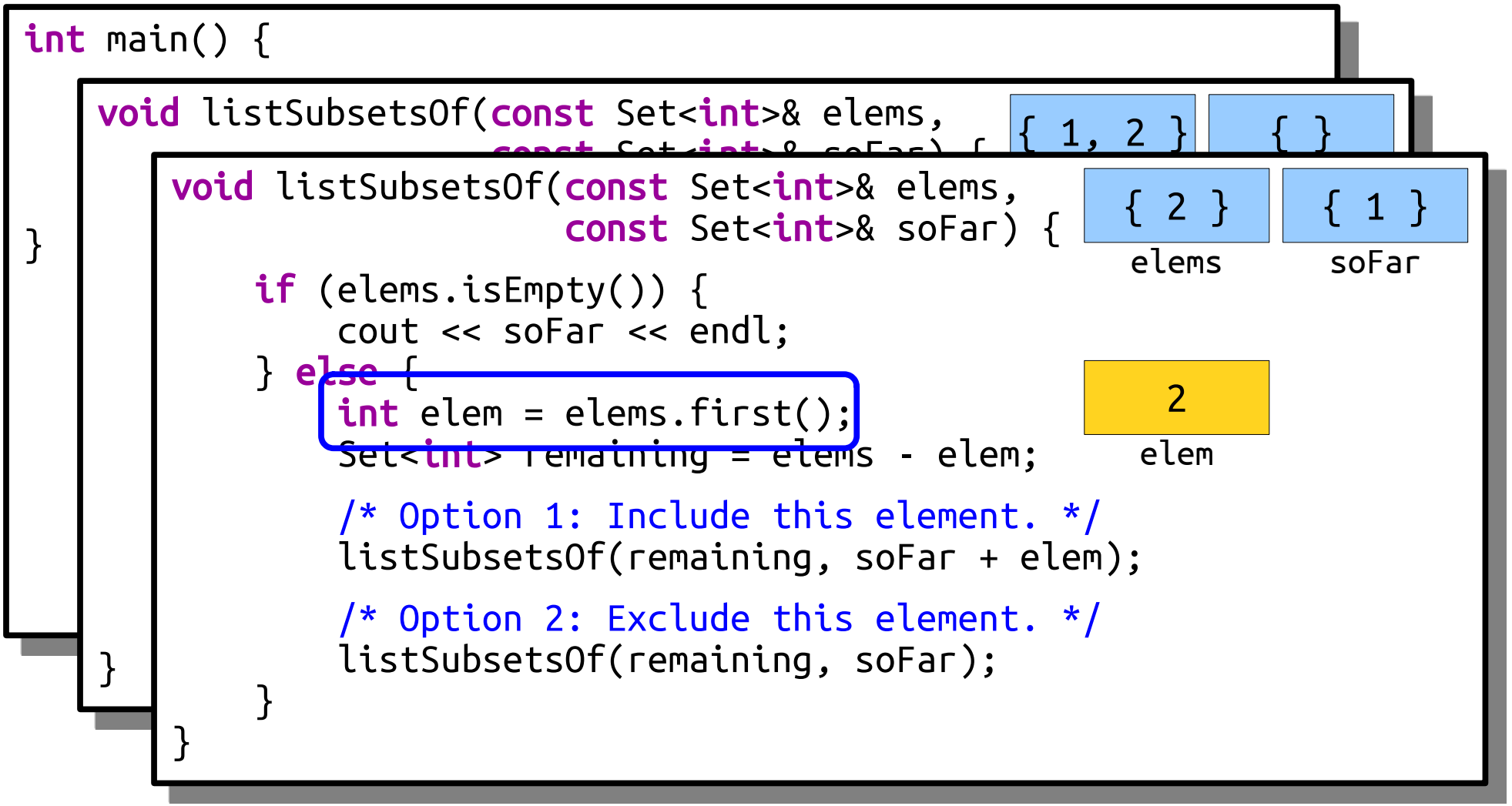
elems

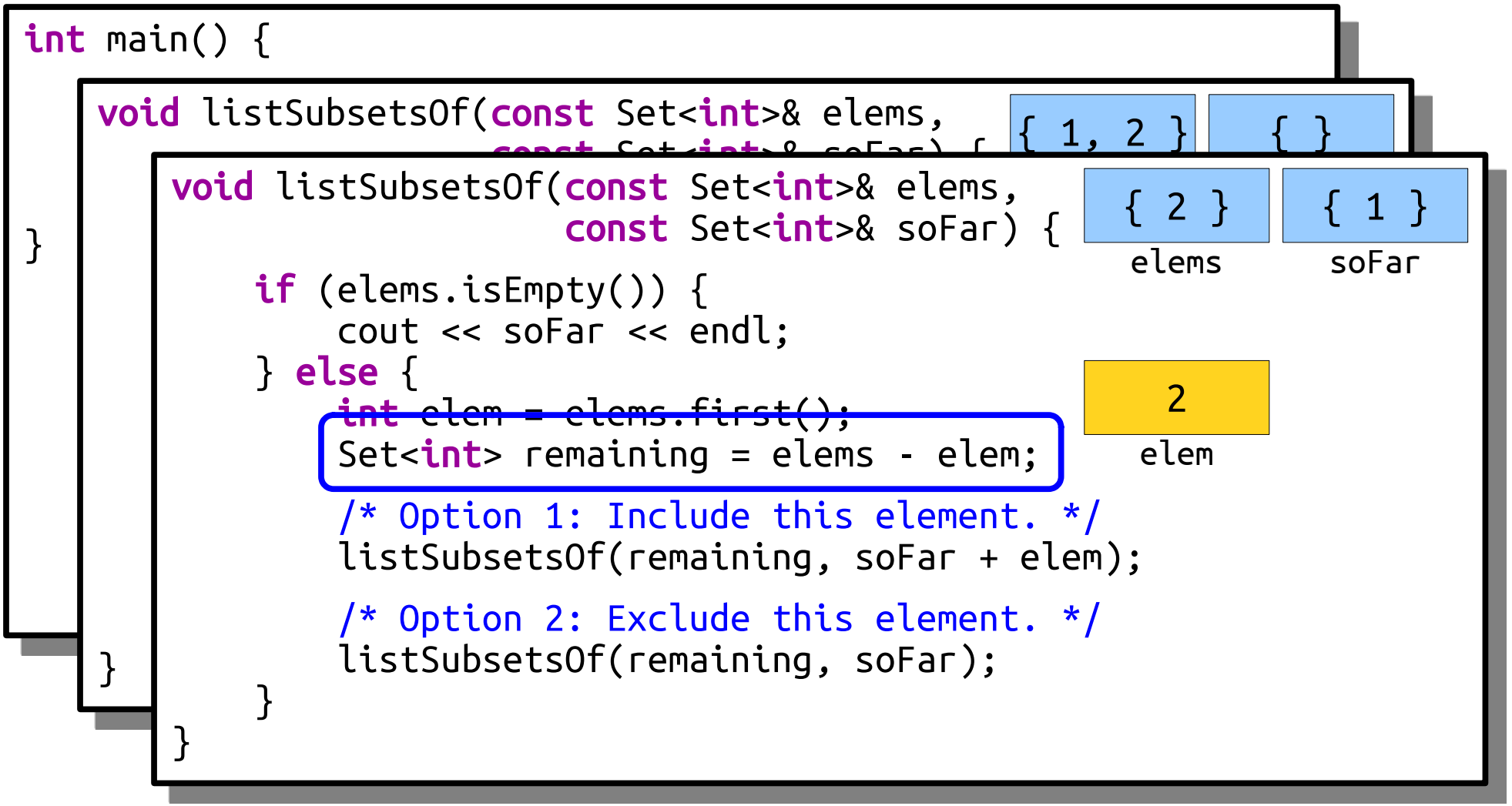
{ 1 }

soFar


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
{ 1, 2 }
{ }
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}
{ 2 }
{ 1 }
elems
soFar


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
{ 1, 2 }
{ }
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
{ 2 }
{ 1 }
elems
soFar
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
{ 1, 2 }
{ }
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}
{ 2 }
{ 1 }
elems
soFar
2
elem


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
{ 1, 2 }
{ }
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}
{ 2 }
{ 1 }
elems
soFar
2
elem

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

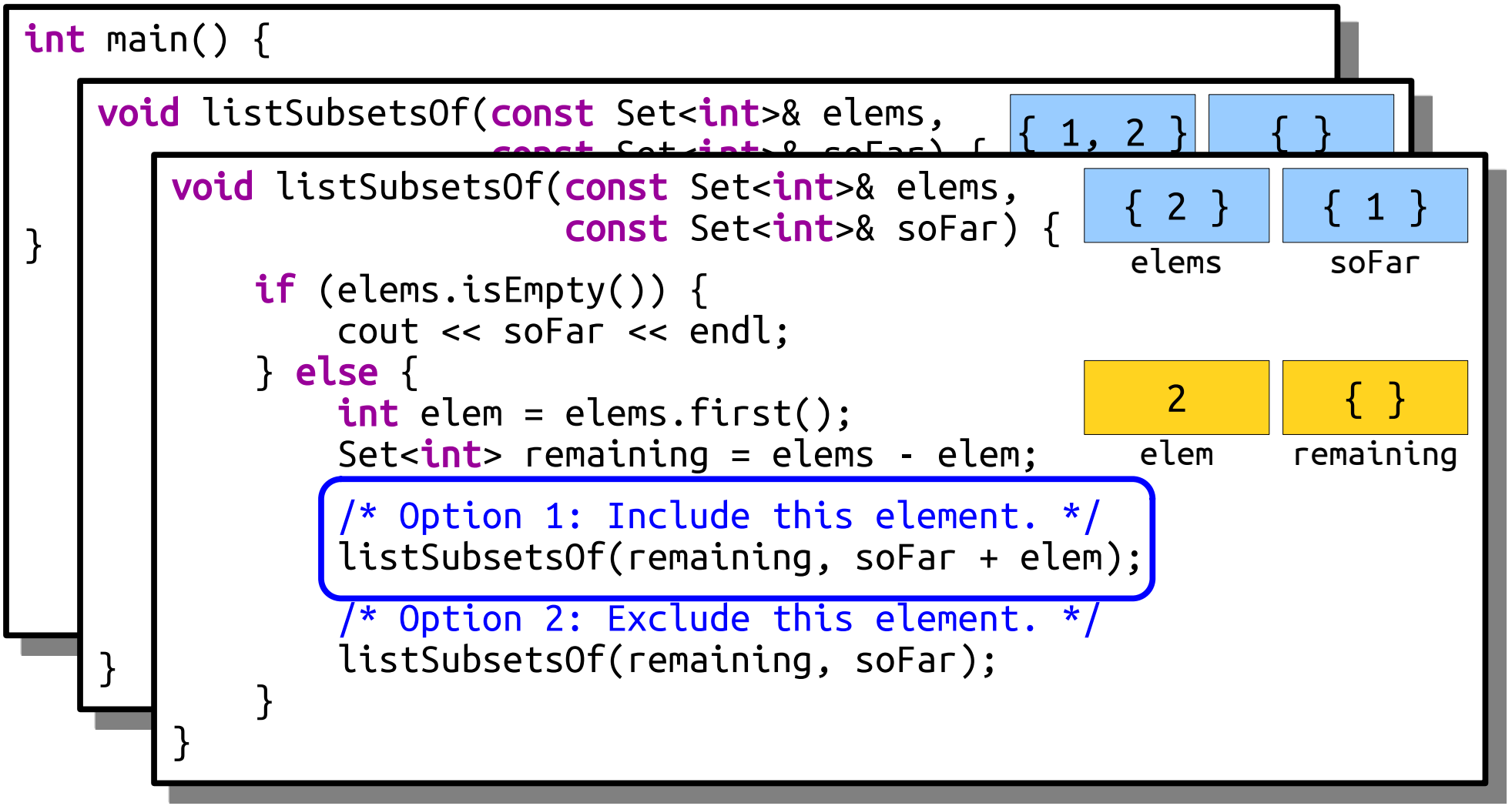
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

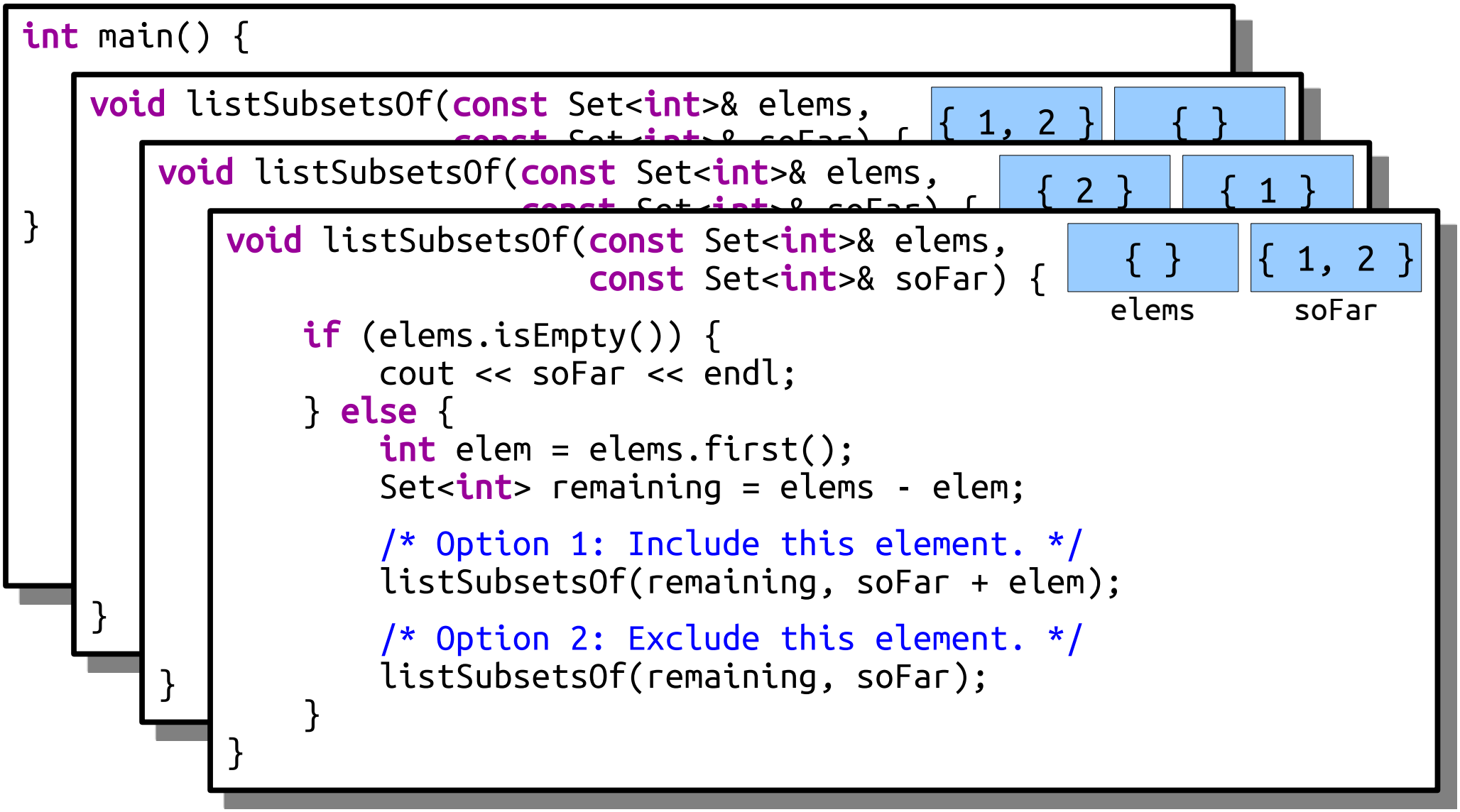
{ 2 } { 1 }

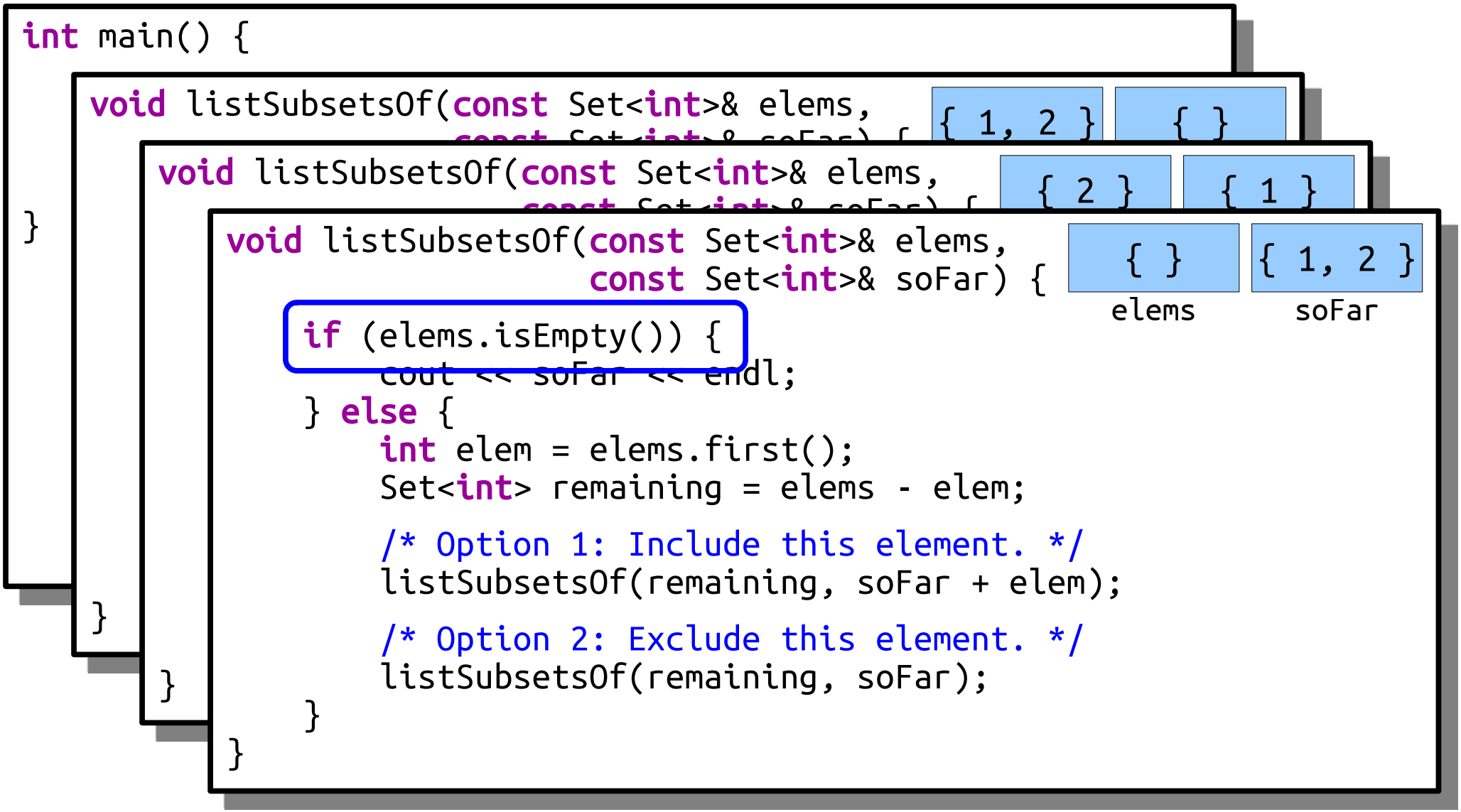
elems soFar

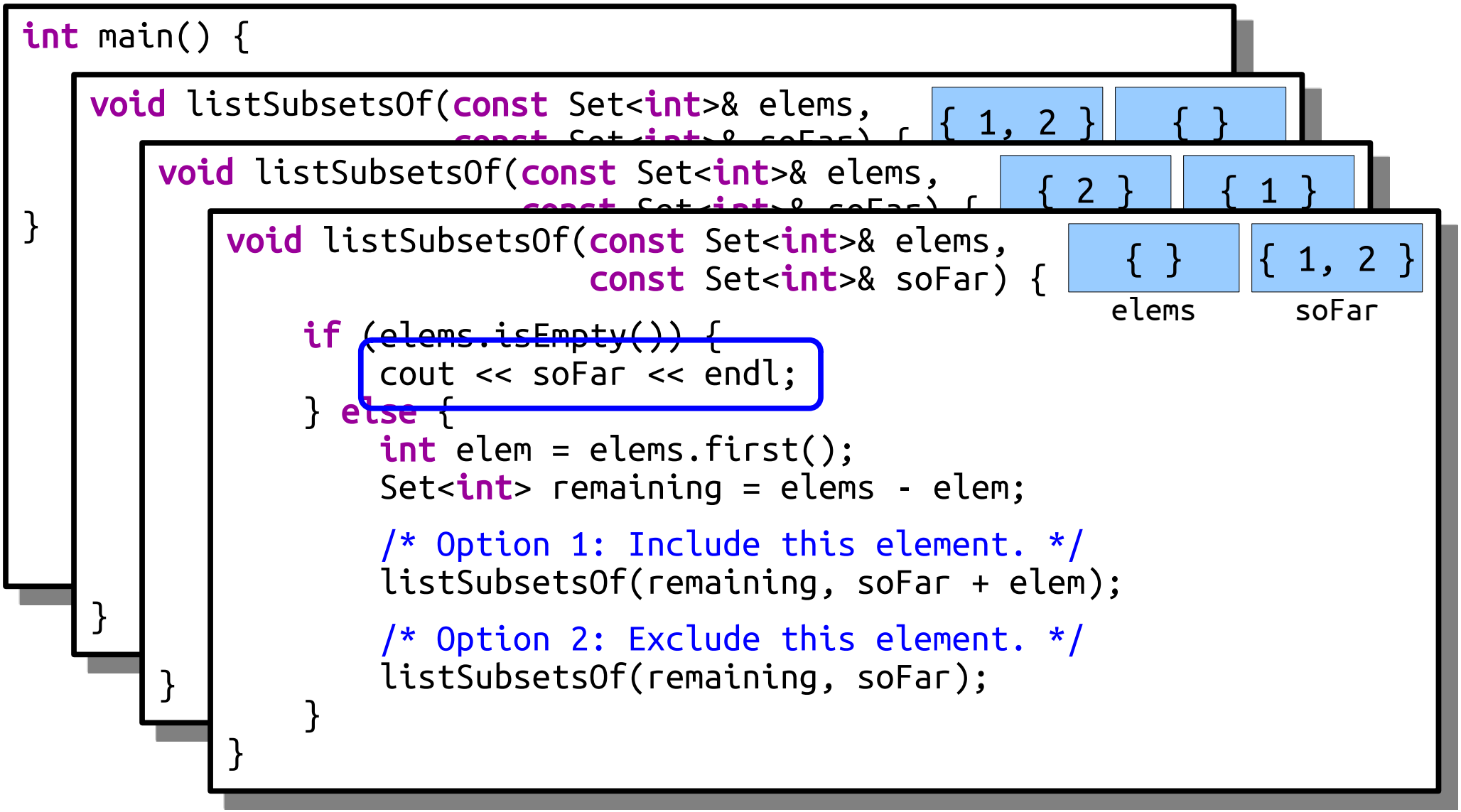
2 { }

elem remaining


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}
{ 1, 2 }
{ }
{ 2 }
{ 1 }
elems
soFar
2
{ }
elem
remaining


int main() {
}
void listSubsetsOf(const Set<int>& elems,
{ 1, 2 }
{ }
}
void listSubsetsOf(const Set<int>& elems,
{ 2 }
{ 1 }
}
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
{ }
{ 1, 2 }
elems
soFar
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}


int main() {
}
void listSubsetsOf(const Set<int>& elems,
}
void listSubsetsOf(const Set<int>& elems,
}
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}
{ 1, 2 }
{ }
{ 2 }
{ 1 }
{ }
{ 1, 2 }
elems
soFar


int main() {
}
void listSubsetsOf(const Set<int>& elems,
{ 1, 2 }
{ }
}
void listSubsetsOf(const Set<int>& elems,
{ 2 }
{ 1 }
}
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
{ }
{ 1, 2 }
elems
soFar
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

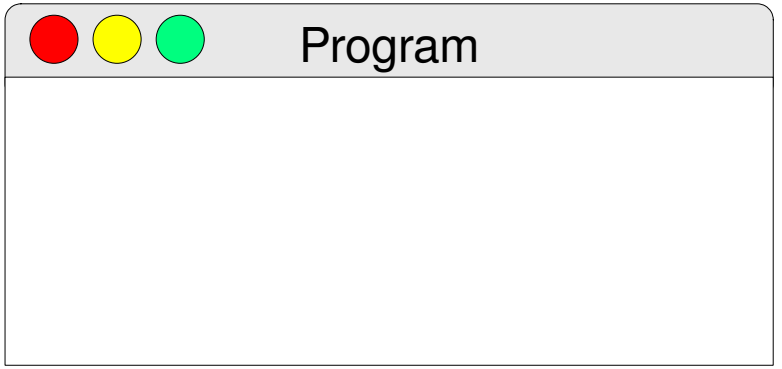
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

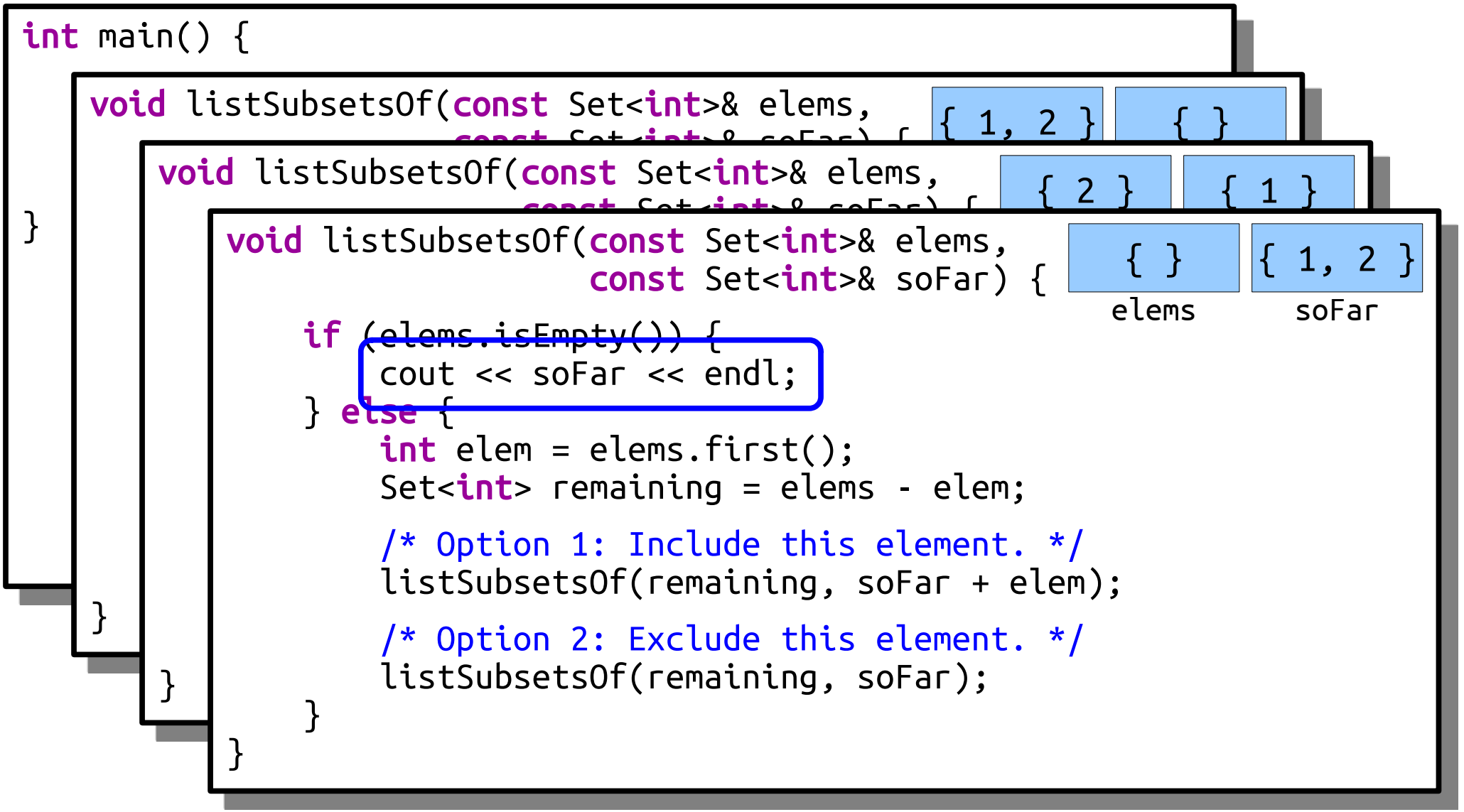
{ 2 } { 1 }

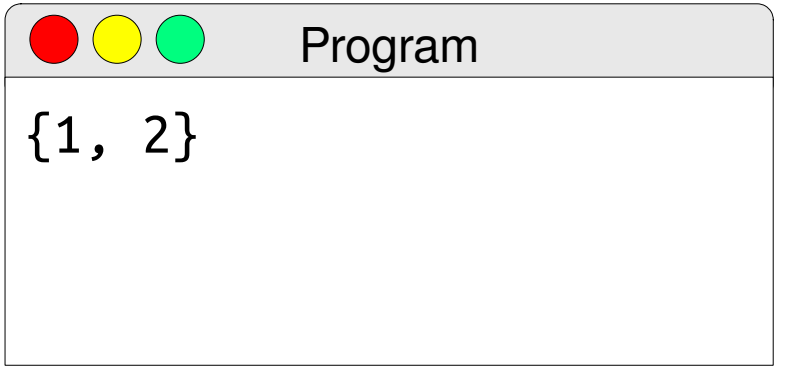
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ } { 1, 2 }

elems soFar




int main() {
}
void listSubsetsOf(const Set<int>& elems,
{ 1, 2 }
{ }
}
void listSubsetsOf(const Set<int>& elems,
{ 2 }
{ 1 }
}
void listSubsetsOf(const Set<int>& elems,
const Set<int>& soFar) {
{ }
{ 1, 2 }
elems
soFar
if (elems.isEmpty()) {
cout << soFar << endl;
} else {
int elem = elems.first();
Set<int> remaining = elems - elem;
/* Option 1: Include this element. */
listSubsetsOf(remaining, soFar + elem);
/* Option 2: Exclude this element. */
listSubsetsOf(remaining, soFar);
}
}


Program
{1, 2}

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 2 } { 1 }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ } { 1, 2 }

elems soFar

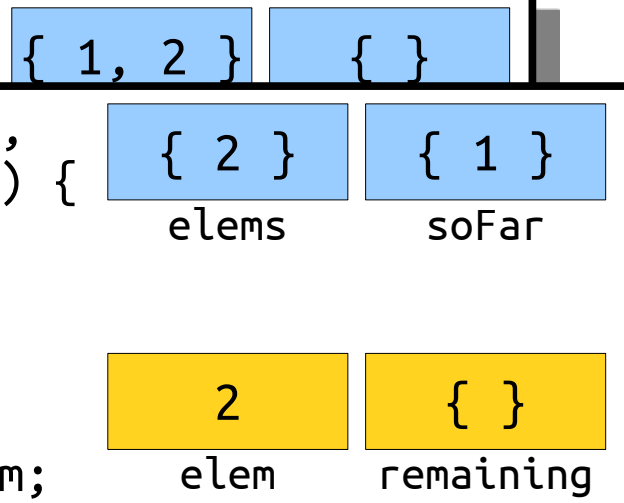
Program

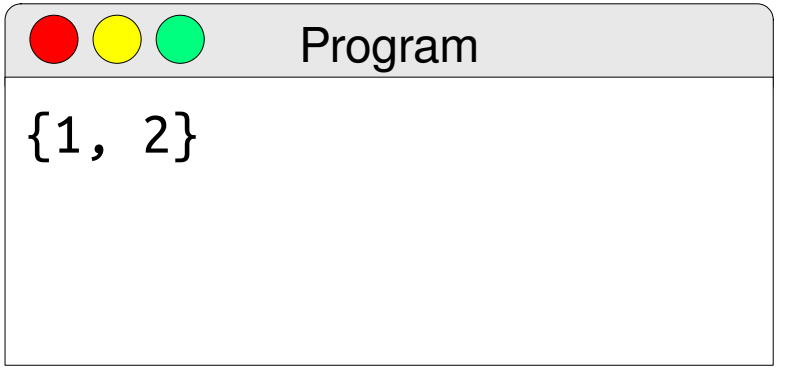
```
{1, 2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```





```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { 1 }

elems soFar

2 { }

elem remaining

Program

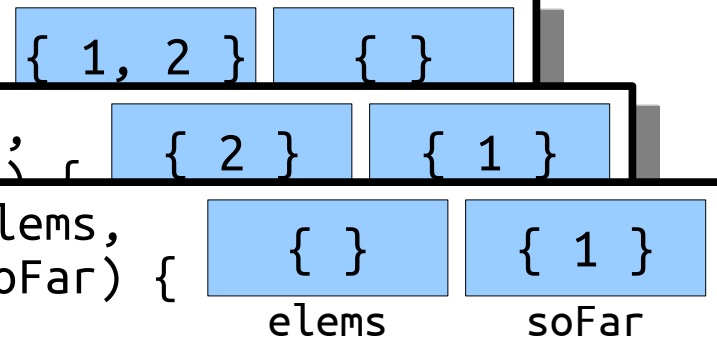
{1, 2}

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

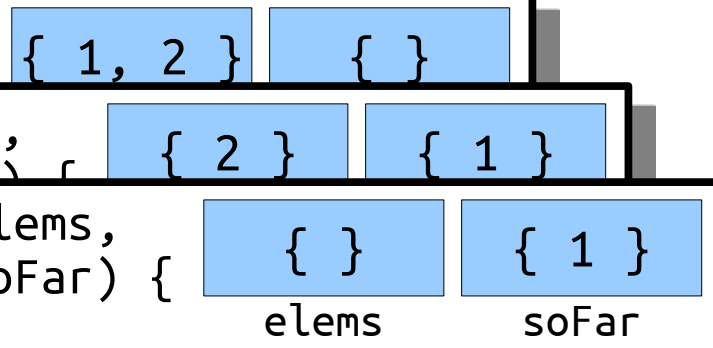
```
{1, 2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

```
{1, 2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 1, 2 }	{ }
{ 2 }	{ 1 }
{ }	{ 1 }
elems	soFar

Program

```
{1, 2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

elems	soFar
{ 1, 2 }	{ }
{ 2 }	{ 1 }
{ }	{ 1 }

Program

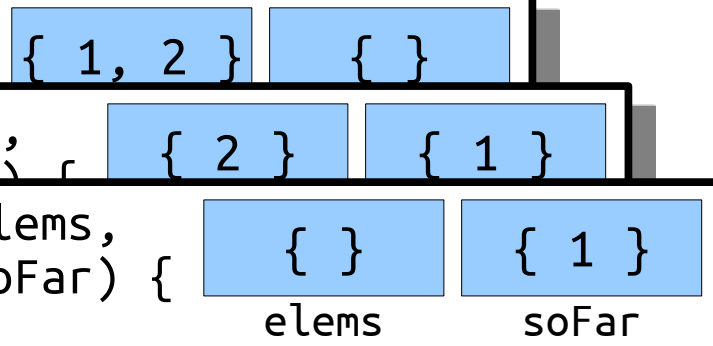
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { 1 }

elems soFar

2 { }

elem remaining

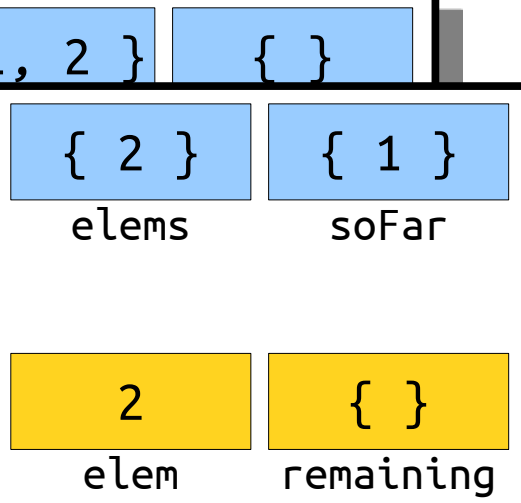
Program

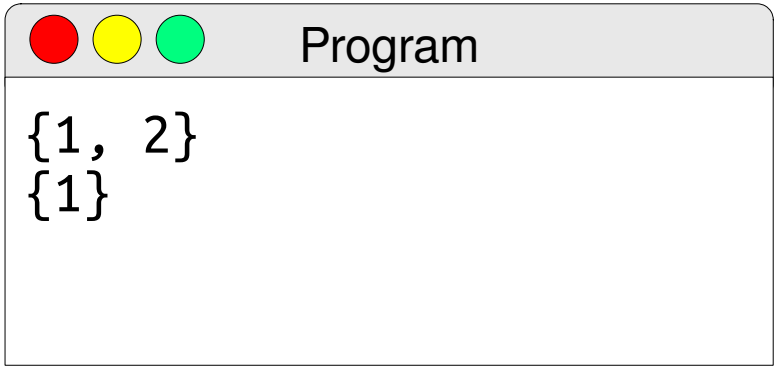
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

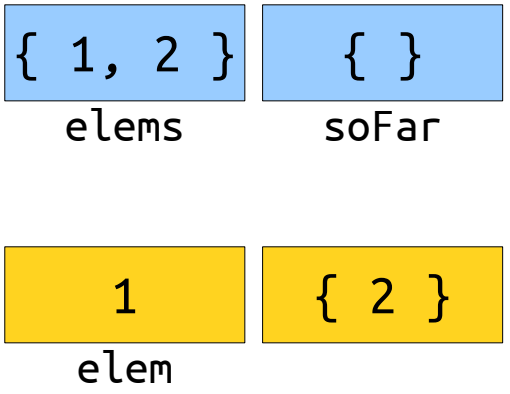
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

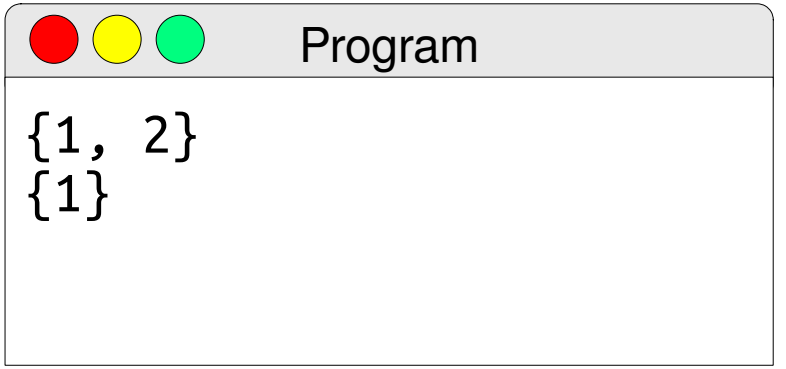




```
int main() {
}
```

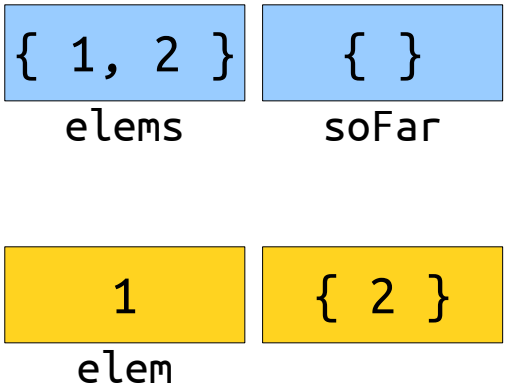
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

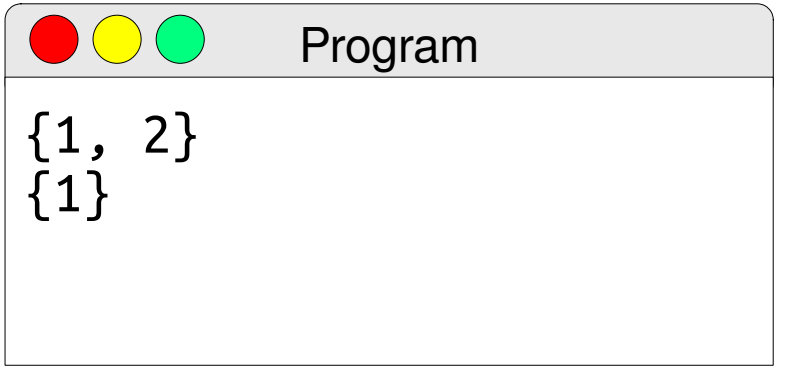




```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```





```
int main() {

}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {

}
```

{ 1, 2 }

{ }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 }

elems

{ }

soFar

Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 }

{ }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 }

elems

{ }

soFar

Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { }

elems soFar

Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 }

{ }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 }

elems

{ }

soFar

Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 }

{ }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 }

elems

{ }

soFar

2

elem

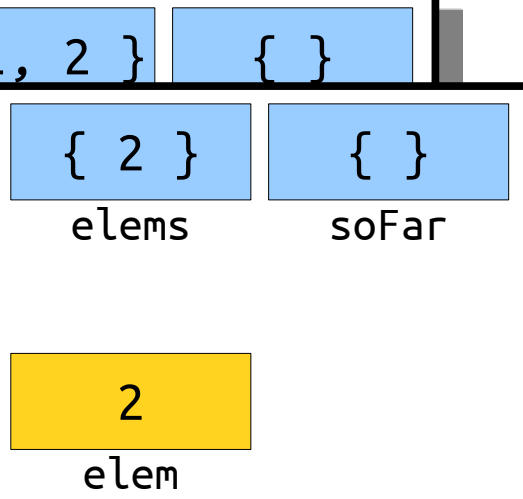
Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



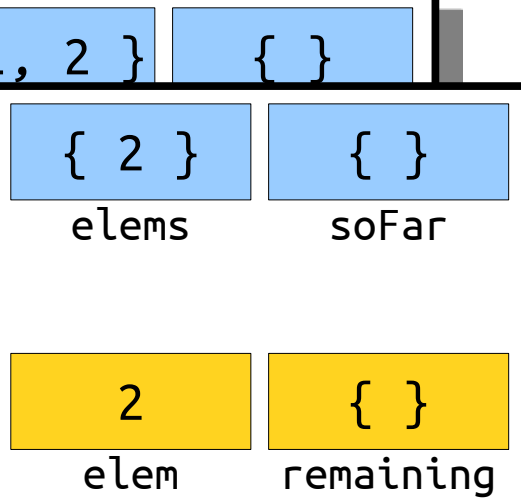
Program

```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



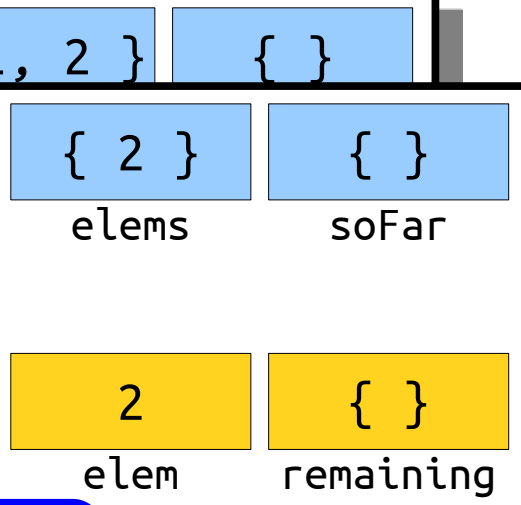
Program

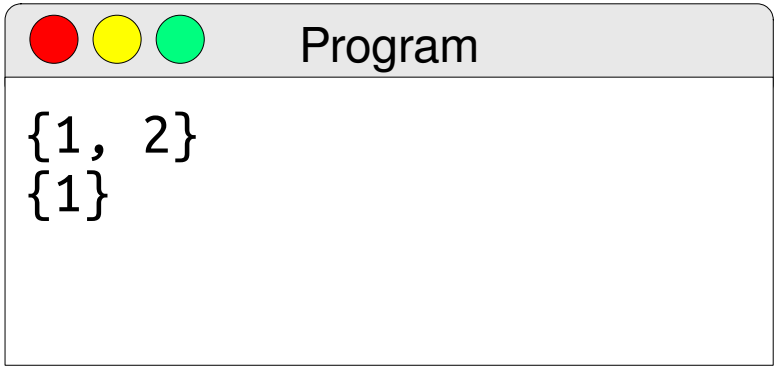
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



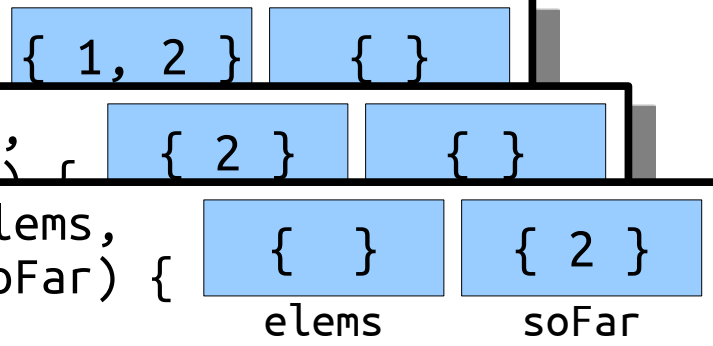


```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

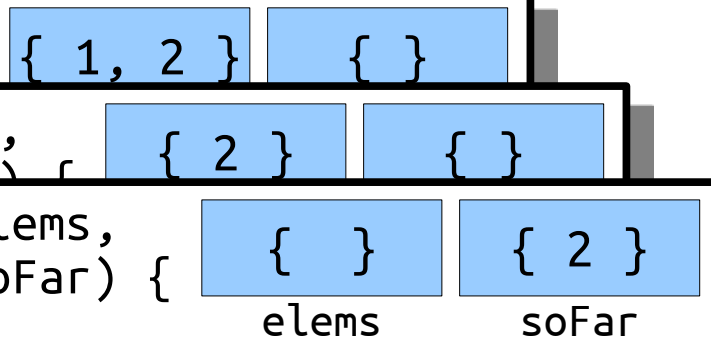
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

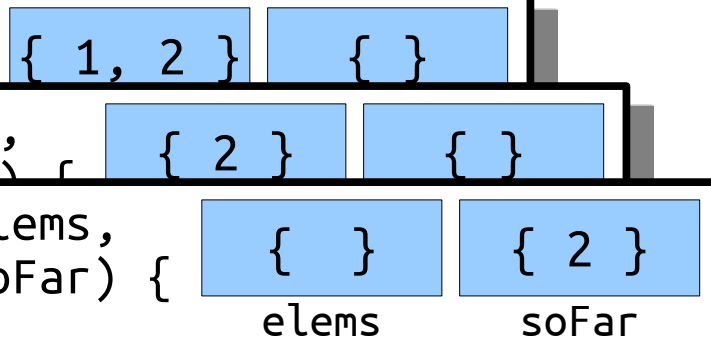
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

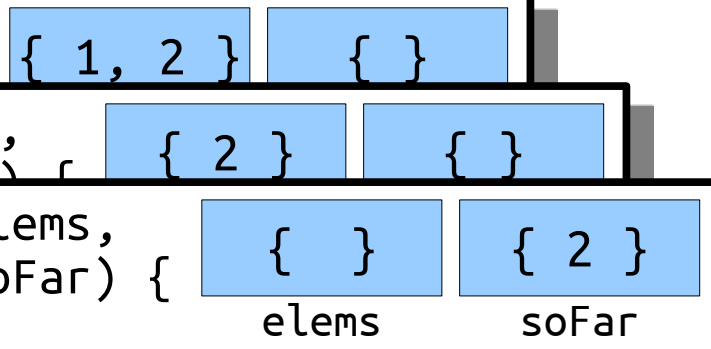
```
{1, 2}
{1}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

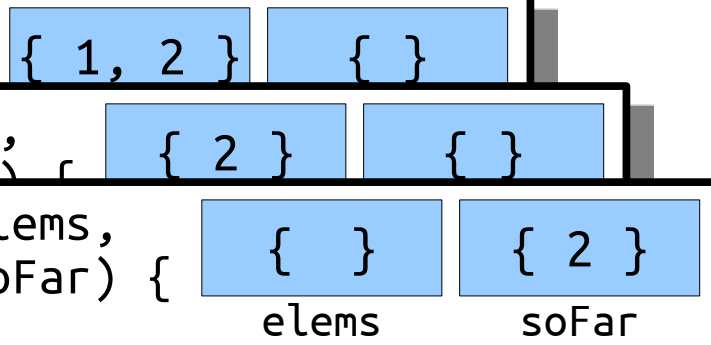
```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { }

elems soFar

2 { }

elem remaining

Program

```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { }

elems soFar

2 { }

elem remaining

Program

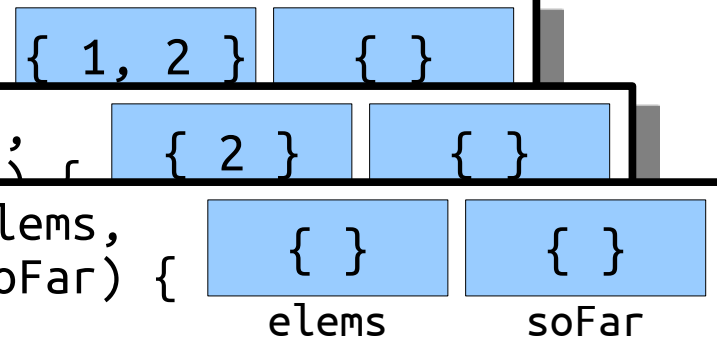
```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

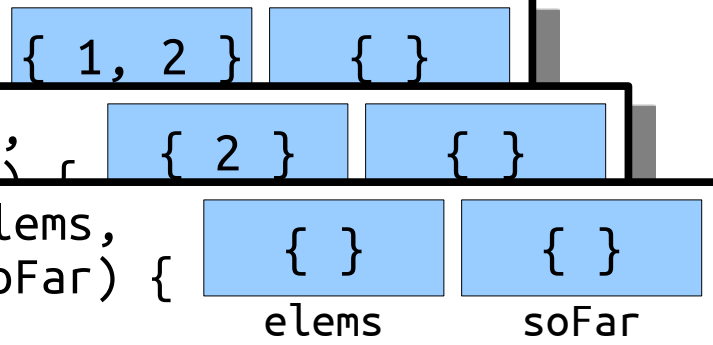
```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

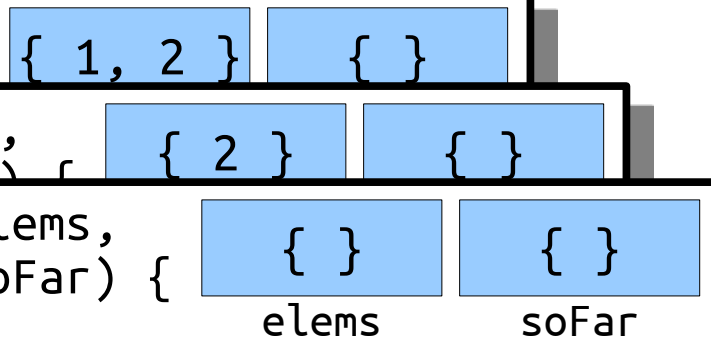
```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

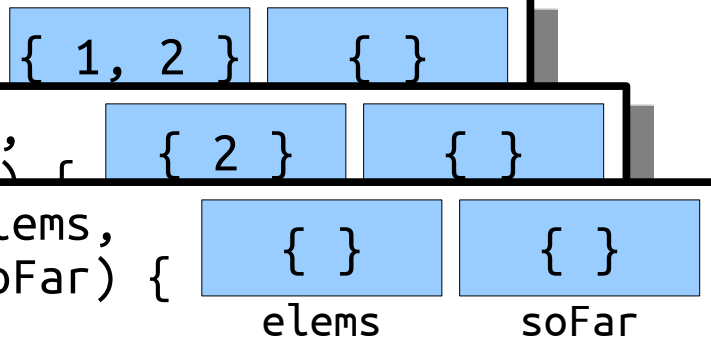
```
{1, 2}
{1}
{2}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

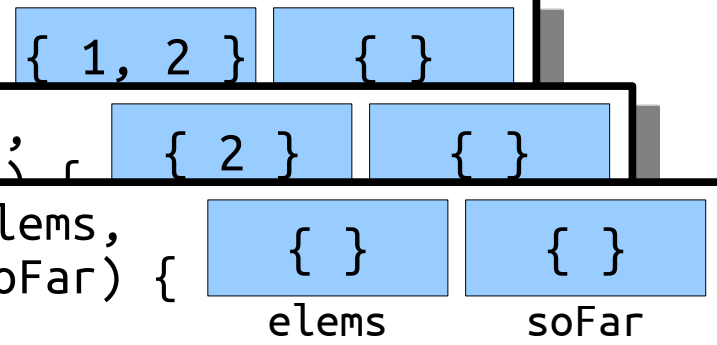
```
{1, 2}
{1}
{2}
{}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```



Program

```
{1, 2}
{1}
{2}
{}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

{ 1, 2 } { }

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

{ 2 } { }

elems soFar

2 { }

elem remaining

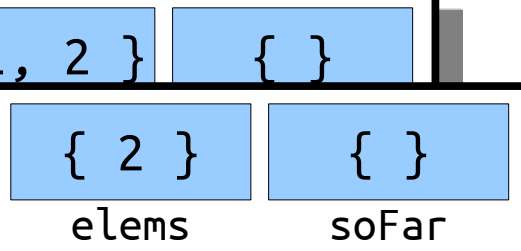
Program

```
{1, 2}
{1}
{2}
{}
```

```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

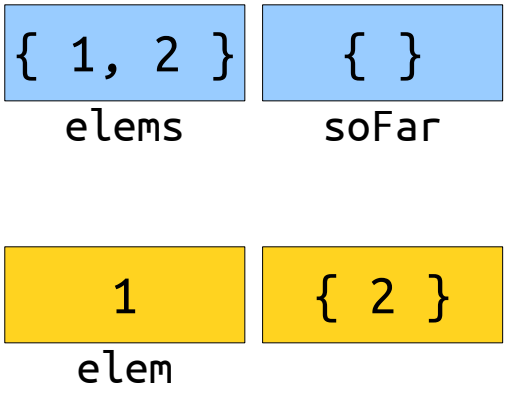


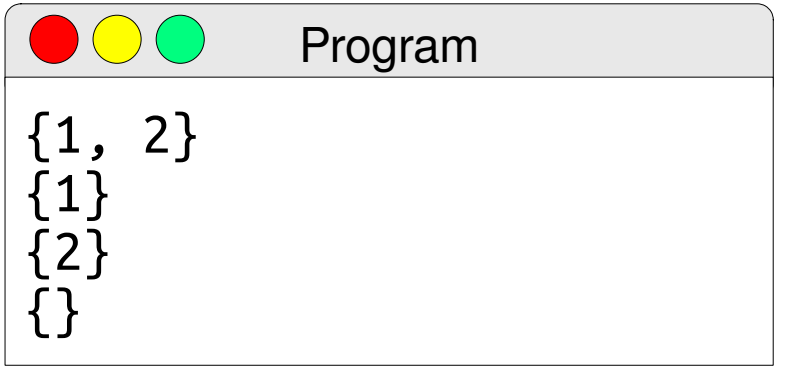
Program

```
{1, 2}
{1}
{2}
{}
```

```
int main() {
}
```

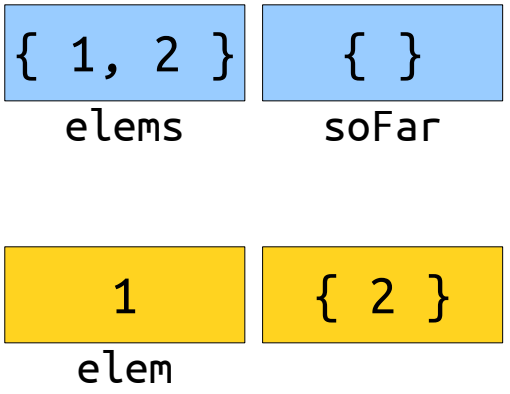
```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```

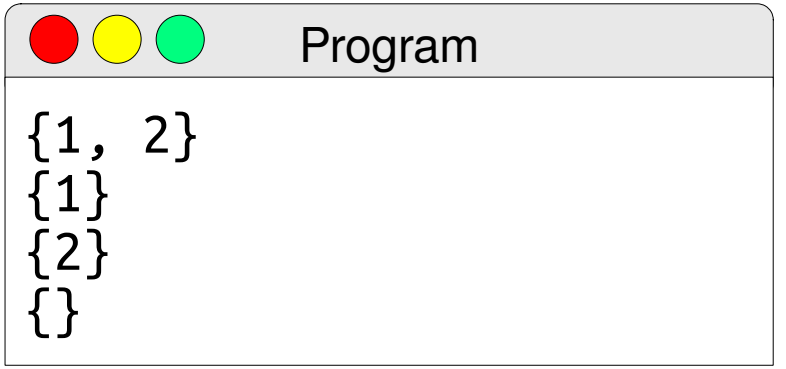




```
int main() {
}
```

```
void listSubsetsOf(const Set<int>& elems,
                   const Set<int>& soFar) {
    if (elems.isEmpty()) {
        cout << soFar << endl;
    } else {
        int elem = elems.first();
        Set<int> remaining = elems - elem;

        /* Option 1: Include this element. */
        listSubsetsOf(remaining, soFar + elem);

        /* Option 2: Exclude this element. */
        listSubsetsOf(remaining, soFar);
    }
}
```





```
int main() {

    listSubsetsOf({ 1, 2 }, { });
    return 0;

}
```

Program

```
{1, 2}
{1}
{2}
{}
```