The Big Picture

Please Evaluate this Course on Axess

Your comments really make a difference.

Final Exam Logistics

- Our final exam is next ***Monday, March 18th*** from ***8:30AM – 11:30AM***.
  - Locations TBA.
- This is a cumulative exam; all topics from all lectures and all assignments are fair game.
  - Concepts purely from the textbook will not be tested, though reading the textbook is not a bad idea.
  - Focus is on concepts you explored on the assignments.
- The exam is administered through BlueBook. As with the midterm, we'll give out a notes sheet. It will go up on the course website by tomorrow evening.
- The exam is closed-book and limited-note. You can bring one double-sided sheet of 8.5" × 11" paper with you to the exam, decorated however you'd like.

Practice Final Exam

- We've posted a BlueBook practice final exam to the course website under the "exams" section.
- This is the final exam we gave out in Winter 2017. We can't guarantee that the actual exam will match the format of this practice final, but it's probably the best indicator of the shape of things to come.
- ***Recommendation:*** Ping your section leader and set up a time to review your answers to the problems. Ask for honest and polite feedback.

Preparing for the Exam

- We'd like everyone to be as prepared as possible going into the final exam. Here are some suggestions of how to prepare:
  - ***Section:*** This week's section is designed as an open-ended review. Section Handout 9 contains a huge number of problems from all the topics we've covered this quarter.
  - ***Midterm:*** Review your midterm, ideally with your section leader. See where your strengths and weaknesses are, and focus on those. If you haven't yet, download the starter files and see if you can get your code into a fully working state.
  - ***Code Step By Step:*** Work through practice problems on any and all topics that seem most relevant for you. This is a great resource for practicing your coding and seeing strategies for solving problems.
  - ***Practice Final:*** If you can, take it under realistic conditions as soon as possible to identify where to focus your efforts.

CS2 Post-Test

- Cynthia Lee is working on a national effort to make a standardized CS2 post-test.
- The test she's designed is different in format from our final exam, but hits on many of the same topics.
- You're welcome to stick around after class to work through the exam if you'd like. We'll give more details when that rolls around.

The Big Picture

Today’s Format

- We’ll spend a few minutes recapping each of the major topics from the course.
- After each section, we’ll provide some questions that we recommend thinking through / working through as you’re studying for the final exam.
- To clarify: this is not us giving you a giant list of possible final exam questions to commit to memory. Rather, these questions are designed to help focus your efforts as you prep for the final.

Week 1: Functions and Recursion

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

int sumOfDigitsOf(int n) {
  if (n < 10) {
    return n;
  } else {
    return sumOfDigitsOf(n / 10) + (n % 10);
  }
}

int digitalRootOf(int n) {
  if (n < 10) {
    return n;
  } else {
    return digitalRootOf(sumOfDigitsOf(n));
  }
}
```

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

bool areComplementary(string one, string two) {
    if (one == "" && two == "") {
        return true;
    } else if (one == "" || one == "") {
        return false;
    } else if (!pairsWith(one[0], two[0])) {
        return false;
    } else {
        return areComplementary(one.substr(1), two.substr(1));
    }
}
```

Questions to Ponder

- Is it safe to change the recursive case of `sumOfDigitsOf` to read as follows?

```
return (n % 10) + sumOfDigitsOf(n / 10);
```

- Is it safe to change the recursive case of `reverseOf` to the following?

```
return input[0] + reverseOf(input.substr(1));
```

- The *space complexity* of a piece of code is the amount of memory used when executing that code. Using big-O notation, give the space complexity of the recursive `factorial` and `sumOfDigitsOf` functions.
- We covered these functions before discussing pass-by-reference. Which functions need to be changed? Why?
- What happens if you reorder the first two base cases in the `areComplementary` function?
- Many recursive functions that work on sequences can be rewritten to leave the sequence unmodified and instead pass an extra index parameter down the recursion chain indicating where the next item to process is. Which of these string recursions can be rewritten that way? What would that look like?

Week 2: Container Types

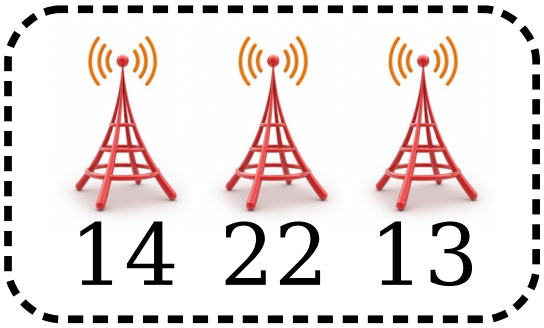
Buying Cell Towers

137 106 107 166 103 261 109

✗ ✓ ✗ ✓ ✗ ✗ ✓

Towers can't be built in two adjacent cities.

People Covered: 106 + 166 + 109 = **381**.



14 22 13

Include 14?

No

Yes

22 13

Include 22?

No

Yes

13

Include 13?

No

Yes

13

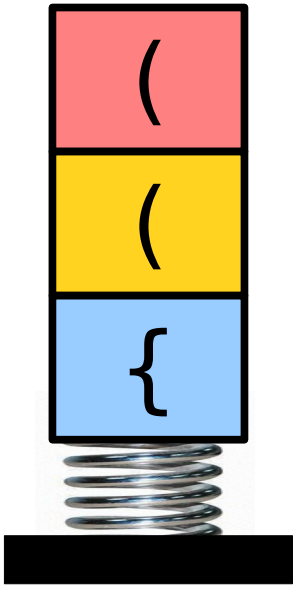
Include 13?

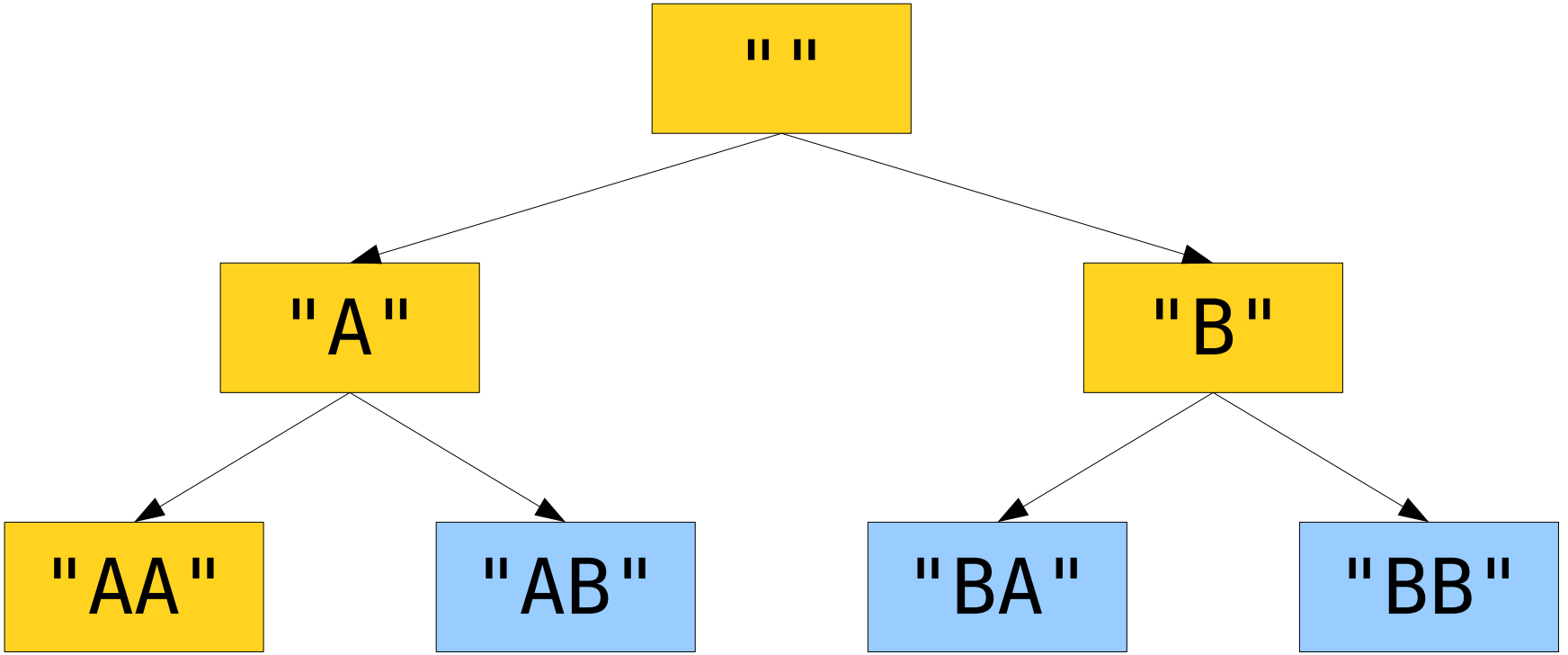
No

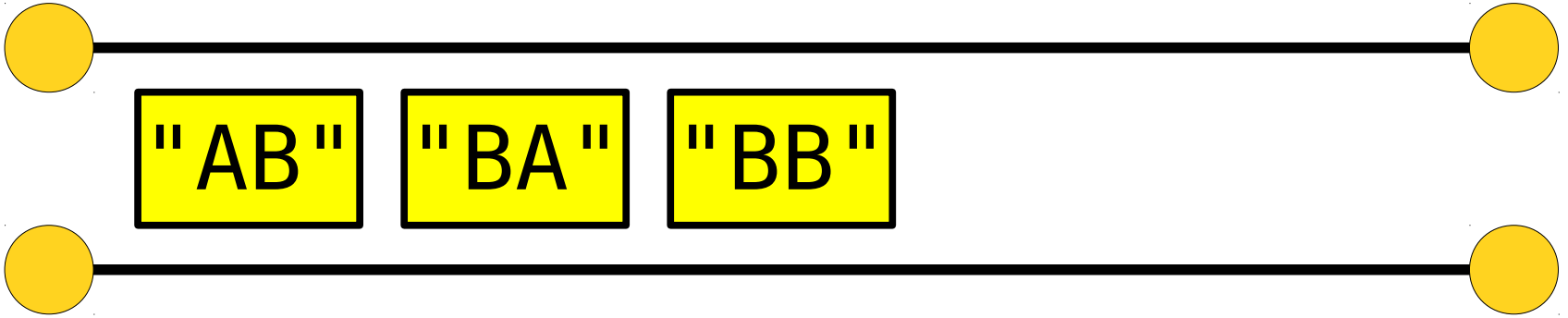
Yes

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```




" "
"A"
"B"
"AA"
"AB"
"BA"
"BB"


"AB"
"BA"
"BB"

Anagrams

- Two words are ***anagrams*** of one another if the letters in one can be rearranged into the other.
- What you told us: anagrams in Bambara:
  - lamaga (to stir) / galama (a ladle made out of a particular kind of gourd)
  - denso (womb/guts) / sedon (“d-day”, the day someone arrives or something starts) / soden (a room in a home)

Credit: Caroline Condon

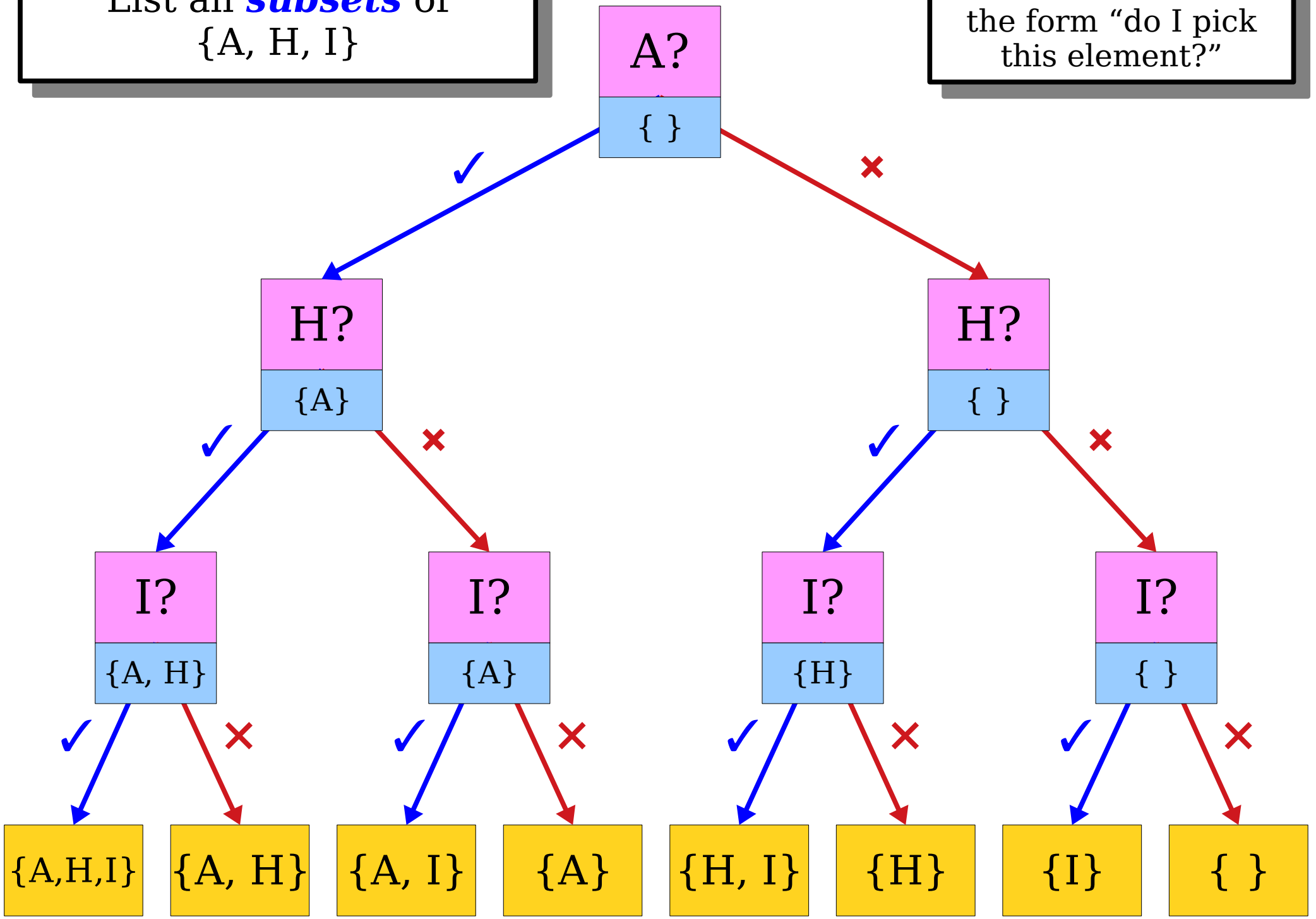
Questions to Ponder

- How does the cell towers problem fit into our framework of recursive exploration and decision trees?
- Why, intuitively, are nested parentheses modeled by a `Stack` rather than a `Queue`?
- How does the breadth-first search we used to print out all possible strings relate to the breadth-first search we saw in graph theory?
- How does the Crystals assignment relate to breadth-first search?
- In finding anagram clusters, we rearranged the characters in strings into sorted order. What sorting algorithm would you recommend for this case? Why?

Week 3: Enumeration / Optimization

List all ***subsets*** of {A, H, I}

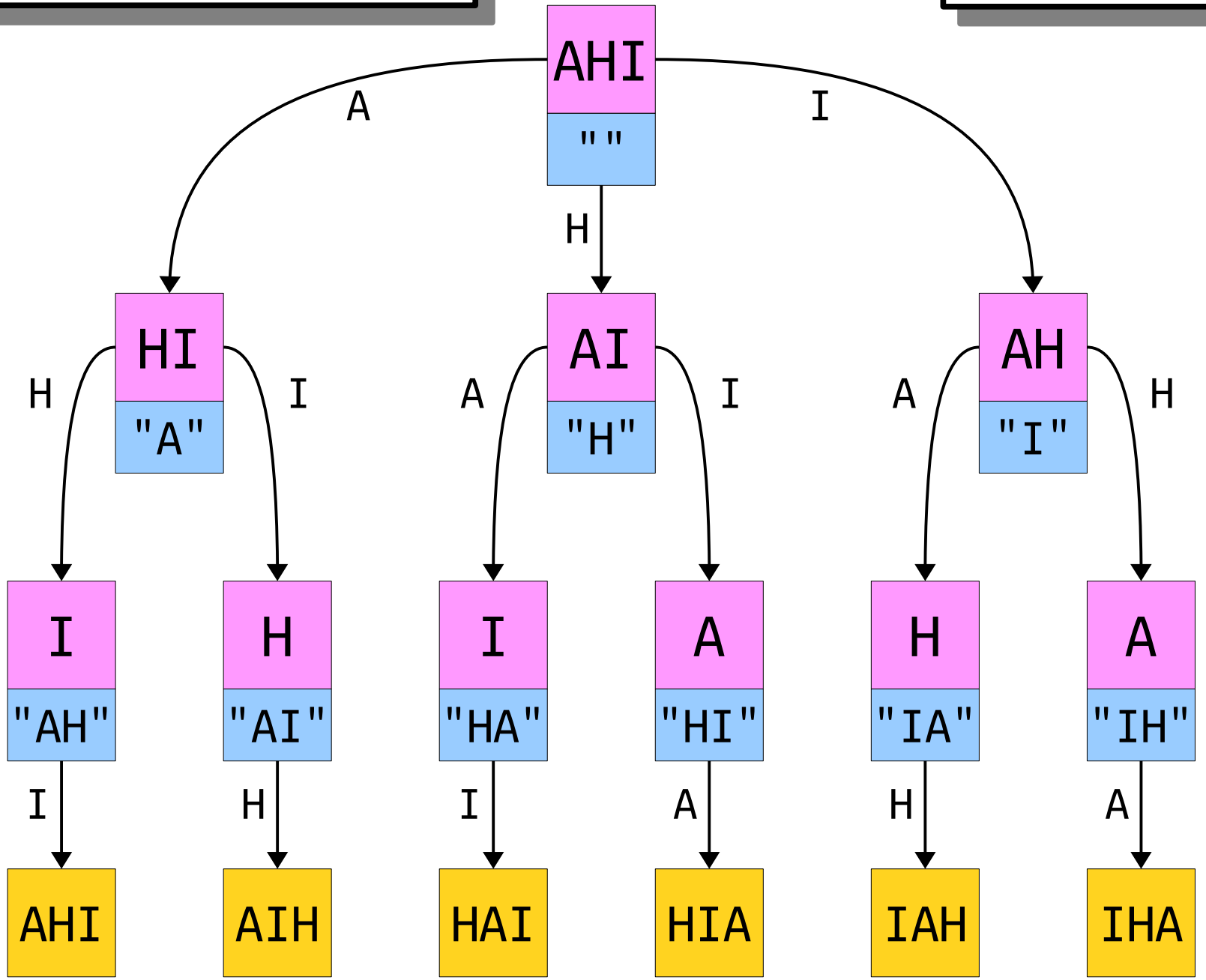
Each decision is of the form “do I pick this element?”



```
void listSubsetsRec(const Set<int>& remaining,
                    const Set<int>& used) {

  if (remaining.isEmpty()) {
    cout << used << endl;
  } else {
    int elem = remaining.first();

    /* Option 1: Include this element. */
    listSubsetsRec(remaining - elem, used + elem);

    /* Option 2: Exclude this element. */
    listSubsetsRec(remaining - elem, used);
  }
}
```

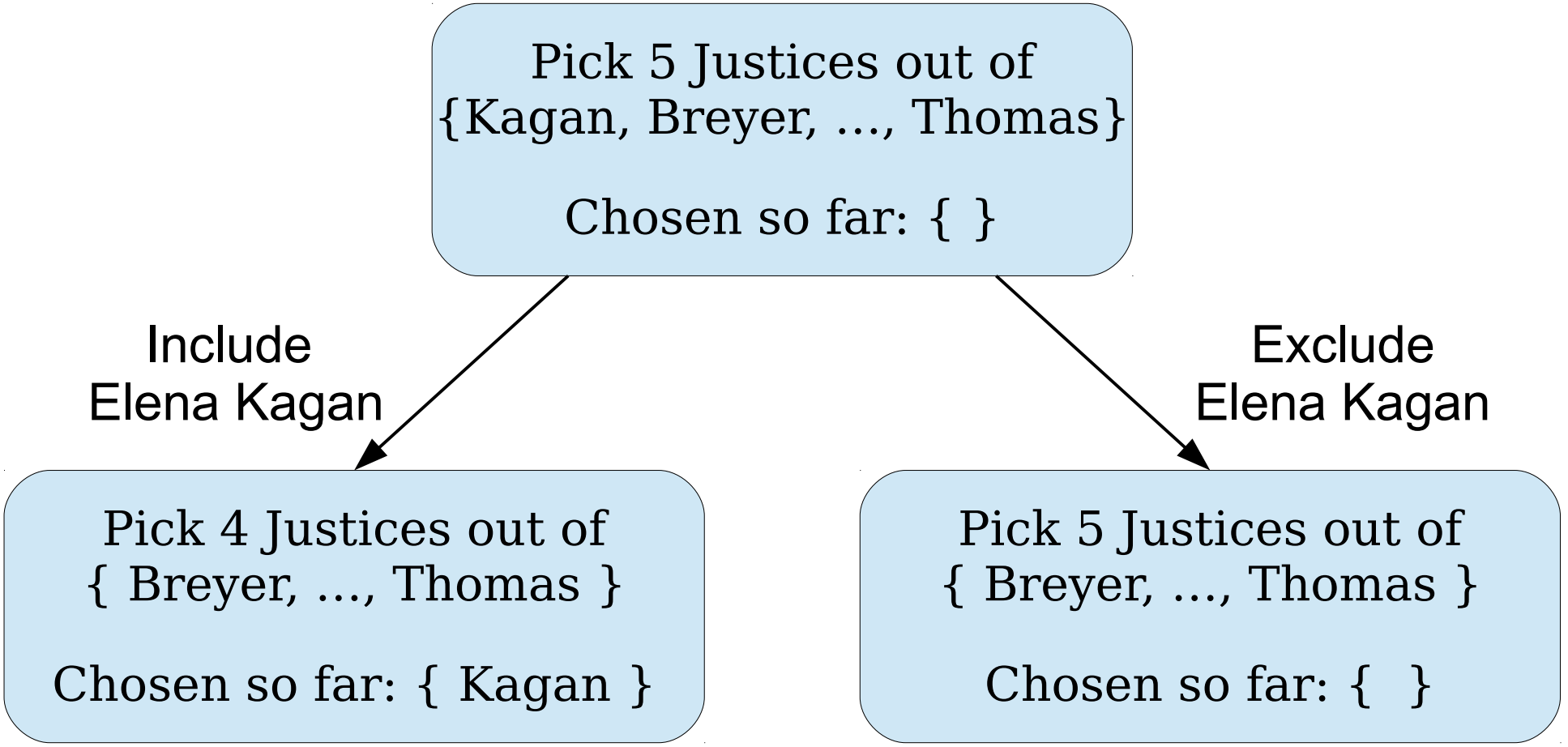
List all ***permutations*** of {A, H, I}

Each decision is of the form “what do I pick next?”



```
void listPermutationsRec(const string& remaining,
                         const string& used) {

  if (remaining == "") {
    cout << used << endl;
  } else {
    /* Decide what comes next. */
    for (int i = 0; i < remaining.size(); i++) {
      listPermutationsRec(remaining.substr(0, i) +
                          remaining.substr(i + 1),
                          used + remaining[i]);
  }
}
```

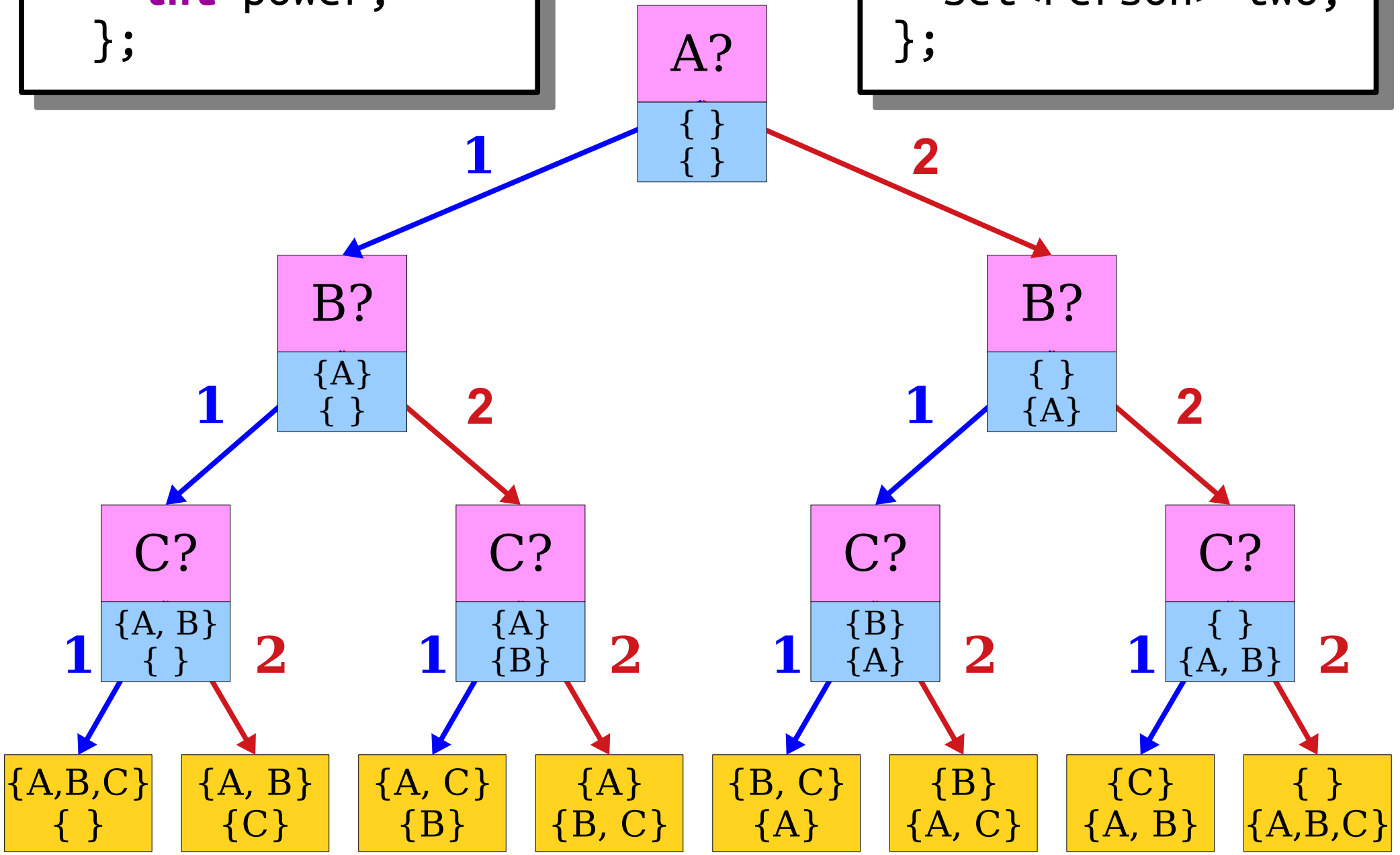
Judicial Decisions



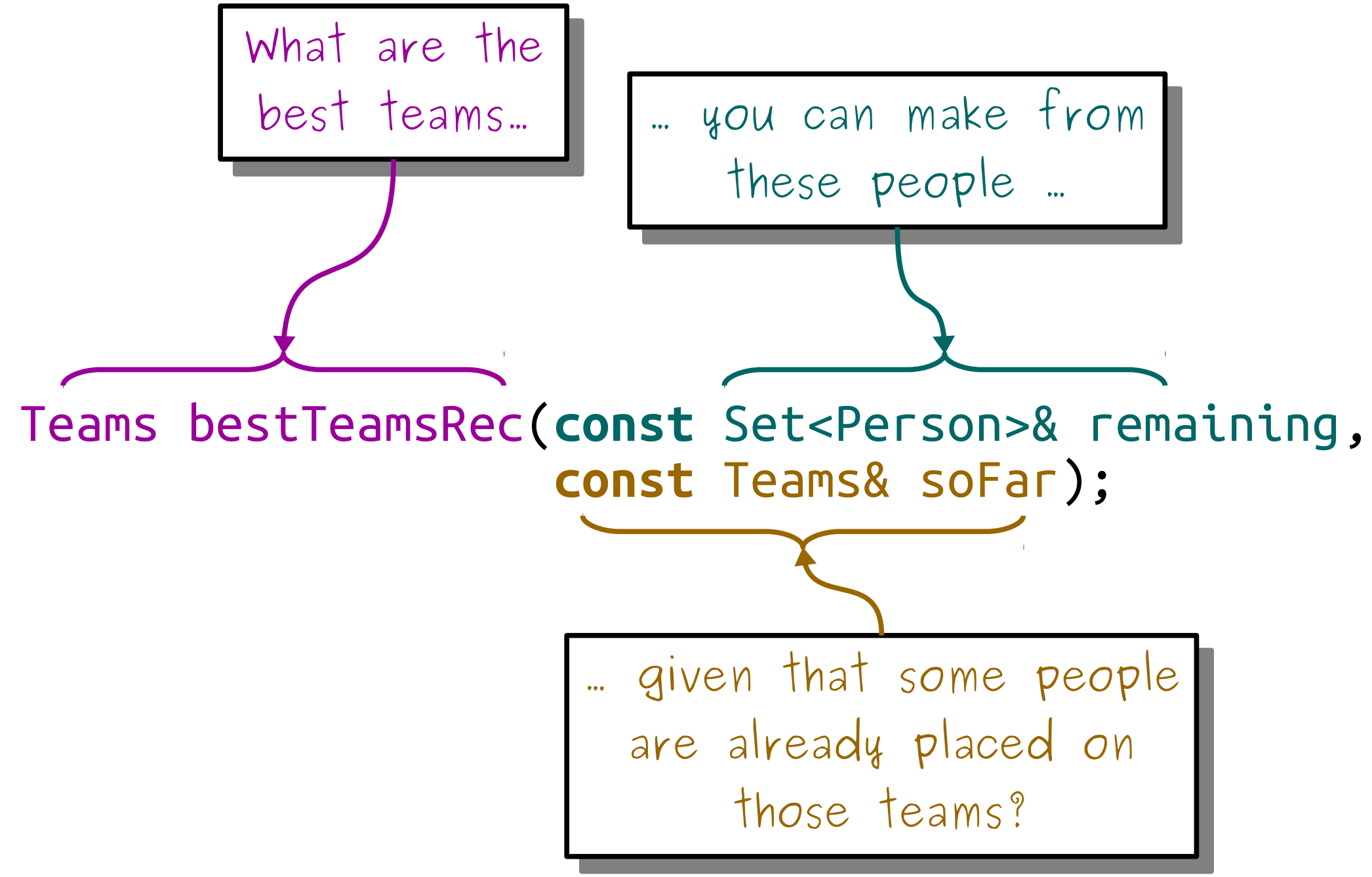
```
void listCombinationsRec(const Set<int>& remaining, int k,
                         const Set<int>& used) {
  if (k == 0) {
    cout << used << endl;
  } else if (remaining.isEmpty() || k > remaining.size()) {
    return; // Can't succeed.
  } else {
    int elem = remaining.first();

    /* Option 1: Exclude this element. */
    listCombinationsRec(remaining - elem, k, used);

    /* Option 2: Include this element. */
    listCombinationsRec(remaining - elems, k - 1, used + elem);
  }
}
```

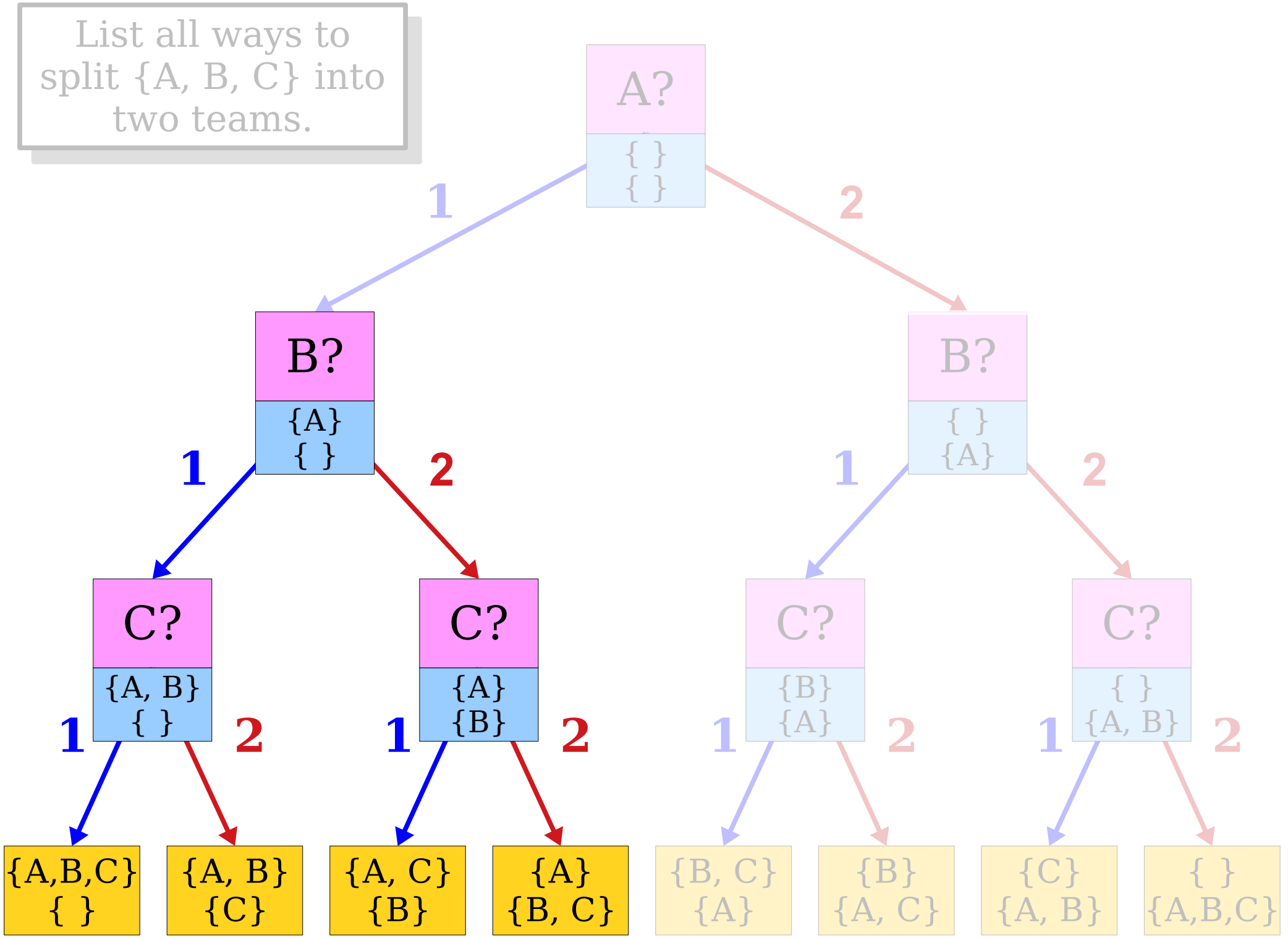
```
struct Person {
  string name;
  int power;
};
```

```
struct Teams {
  Set<Person> one;
  Set<Person> two;
};
```



```
Teams bestTeamsRec(const Set<Person>& remaining,
                   const Teams& soFar) {
  if (remaining.isEmpty()) {
    return soFar;
  } else {
    Person curr = remaining.first();

    /* Option 1: Put this person on Team 1. */
    Teams best1 = bestTeamsRec(remaining - curr,
                               { soFar.one + curr, soFar.two });

    /* Option 2: Put this person on Team 2. */
    Teams best2 = bestTeamsRec(remaining - curr,
                               { soFar.one, soFar.two + curr });

    if (imbalanceOf(best1) < imbalanceOf(best2)) {
      return best1;
    } else {
      return best2;
    }
  }
}
```


What are the best teams...
... you can make from these people ...
Teams bestTeamsRec(const Set<Person>& remaining,
const Teams& soFar);
... given that some people are already placed on those teams?


List all ways to split {A, B, C} into two teams.
A?
{ }
{ }
1
2
B?
{A}
{ }
B?
{ }
{A}
1
2
1
2
C?
{A, B}
{ }
C?
{A}
{B}
C?
{B}
{A}
C?
{ }
{A, B}
1
2
1
2
1
2
1
2
{A,B,C}
{ }
{A, B}
{C}
{A, C}
{B}
{A}
{B, C}
{B, C}
{A}
{B}
{A, C}
{C}
{A, B}
{ }
{A,B,C}

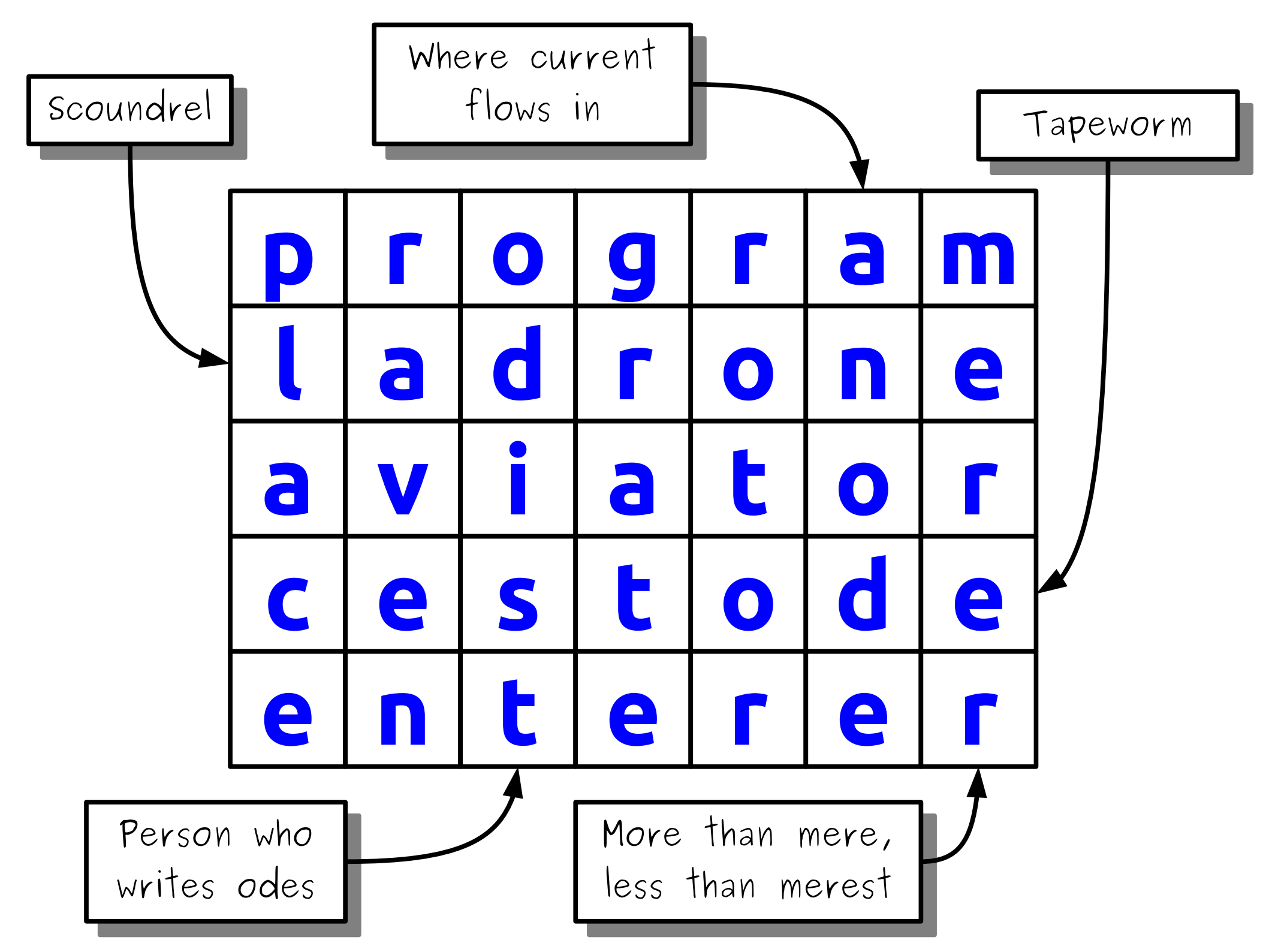
Questions to Ponder

- Why do these problems have the decision trees that they have? Why does each problem's decision tree not work for the other problems?
- Using the decision trees as a starting point, why is each recursive function structured the way it is? For example, why is there a for loop in the recursion for permutations but not for subsets, combinations, or tug-of-war?
- What happens if you swap the two base cases in the recursive combinations logic?
- Why, intuitively, does forcing the first person onto the first team cut the work down in tug-of-war by one half?
- We implemented our recursive subsets code by calling `Set::first` to pick out some element to remove. The set is backed by a BST. Which element does this mean we're picking out? Based on that, can you predict the order in which the subsets are listed?
- Are our recursive exploration functions closer to breadth-first search or depth-first search? Why?

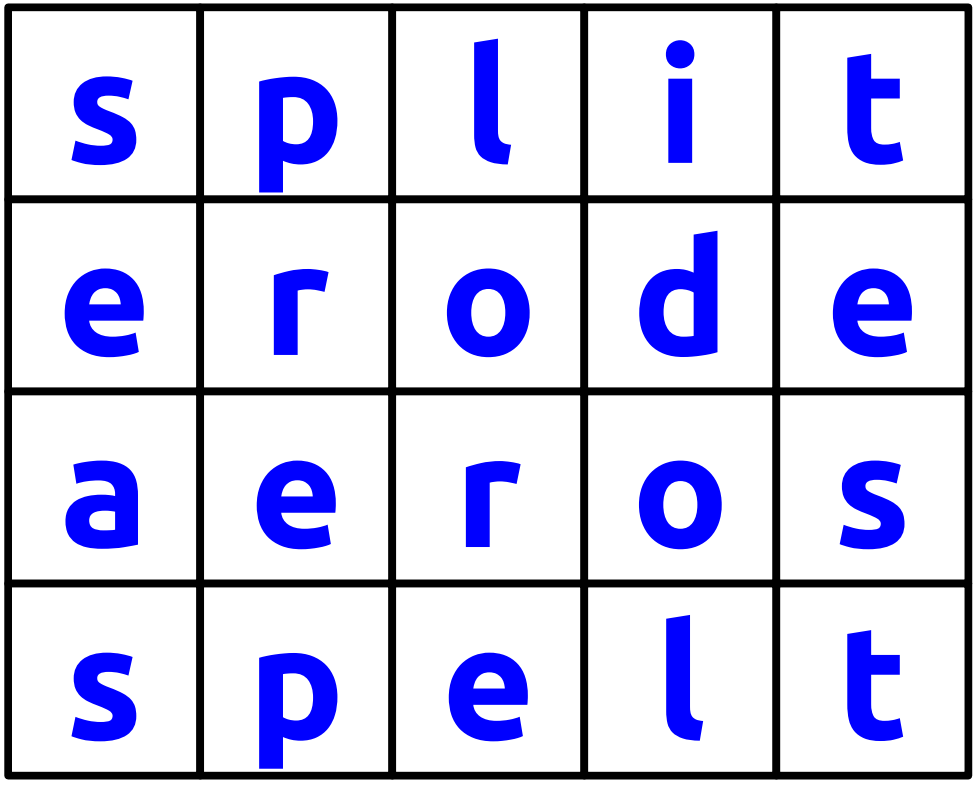
Week 4: Backtracking

“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?”

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }
    return false;
}
```


Scoundrel
Where current flows in
Tapeworm
p r o g r a m
l a d r o n e
a v i a t o r
c e s t o d e
e n t e r e r
Person who writes odes
More than mere, less than merest

What You Discovered



Credit: Jose Francisco

Questions to Ponder

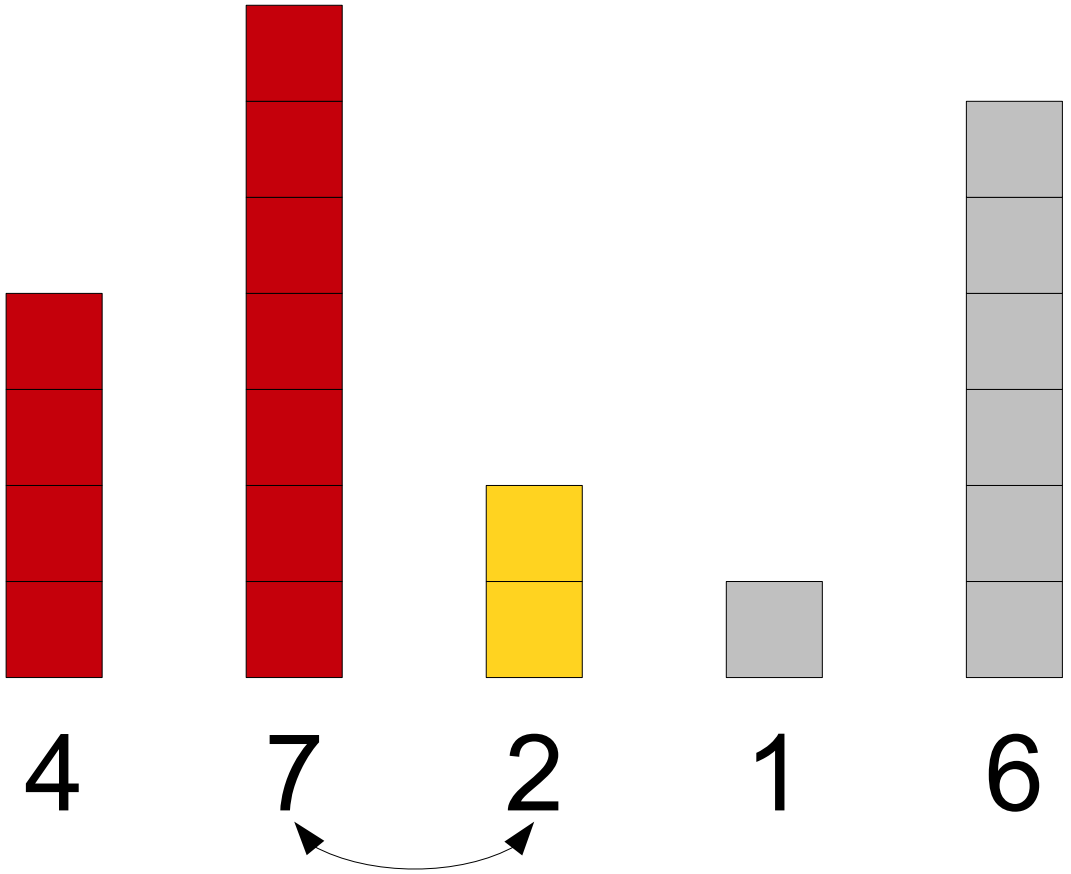
- Why is the return statement in recursive backtracking guarded by an if statement?
- There's an asymmetry in backtracking: if the subcall works, we return true, but if it fails we don't return false. Why not?
- We optimized our crossword search by using `Lexicon::containsPrefix`. How is the `Lexicon` implemented? How is `Lexicon::containsPrefix` implemented? How fast is it?
- Model the shrinkable words problem as a graph search problem. What would the nodes be? What would the edges be? Is the recursive function closer to breadth-first search or depth-first search?
- How would you find all possible ways to shrink a word down to a single-letter word?

Week 5: Big-O and Sorting

Big-O Notation

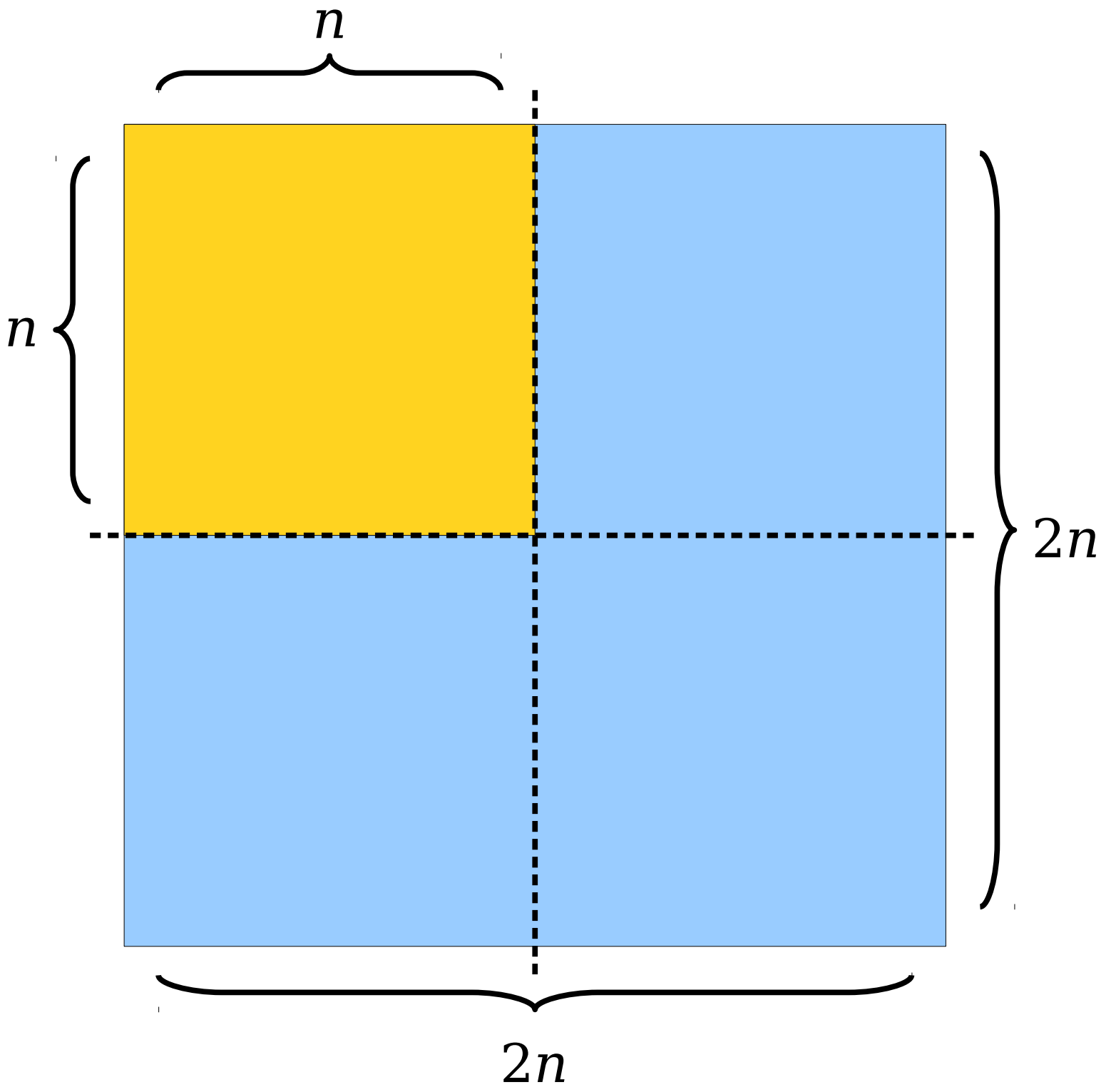
- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$

An Initial Idea: ***Insertion Sort***

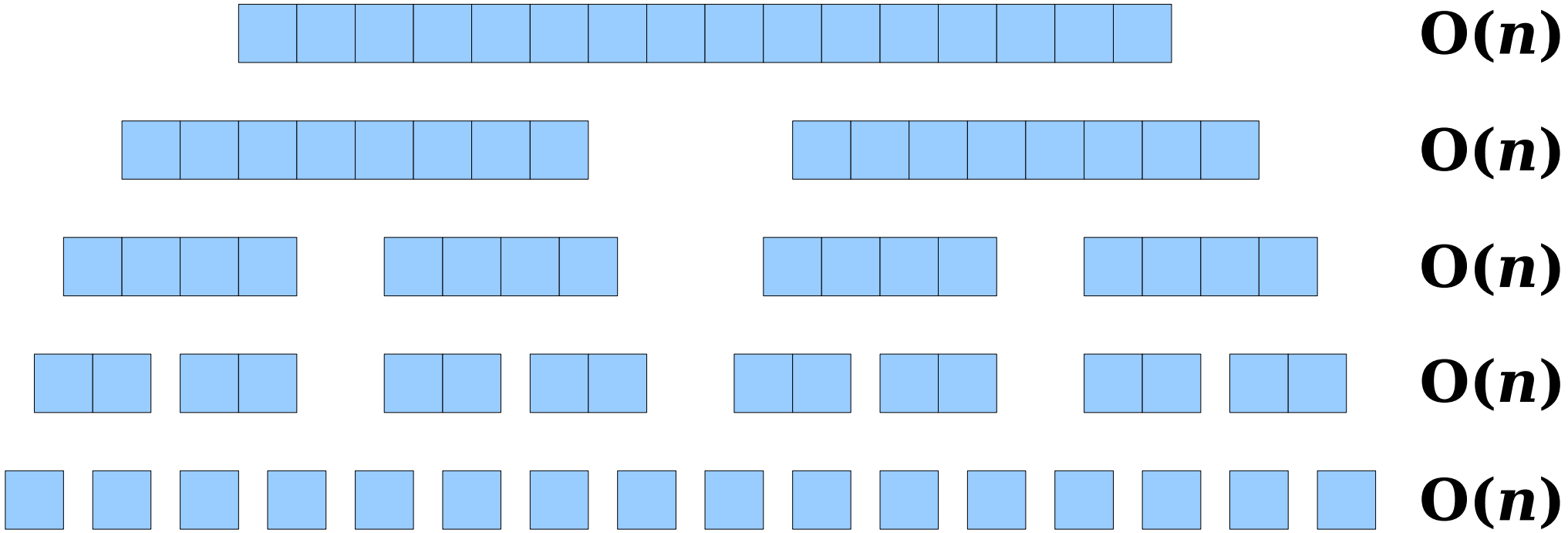


Rule: Swap each element to the left until it doesn't have a bigger element before it.

```
/**
 * Sorts the specified vector using insertion sort.
 *
 * @param v The vector to sort.
 */
void insertionSort(Vector<int>& v) {
  for (int i = 0; i < v.size(); i++) {
    /* Scan backwards until either (1) there is no
     * preceding element or the preceding element is
     * no bigger than us.
     */
    for (int j = i - 1; j >= 0; j--) {
      if (v[j] <= v[j + 1]) break;

      /* Swap this element back one step. */
      swap(v[j], v[j + 1]);
    }
  }
}
```


n
n
$2n$
$2n$

```
void mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are
    * already sorted.
    */
   if (v.size() <= 1) return;

   /* Split v into two subvectors. */
   Vector<int> left, right;
   for (int i = 0; i < v.size() / 2; i++) {
       left += v[i];
   }
   for (int i = v.size() / 2; i < v.size(); i++) {
       right += v[i];
   }

   /* Recursively sort these arrays. */
   mergesort(left);
   mergesort(right);

   /* Combine them together. */
   merge(left, right, v);
}
```



There are O(log n) levels in the recursion.

Each level does O(n) work.

Total work done: **O(n log n)**.

```
bool binarySearchRec(const Vector<int>& elems, int key,
                     int low, int high) {
    /* Base case: If we're out of elements, horror of horrors!
     * Our element does not exist.
     */
    if (low == high) return false;

    /* Probe the middle element. */
    int mid = low + (high - low) / 2;

    /* We might find what we're looking for! */
    if (key == elems[mid]) return true;

    /* Otherwise, discard half the elements and search
     * the appropriate section.
     */
    if (key < elems[mid]) {
        return binarySearchRec(elems, key, low, mid);
    } else {
        return binarySearchRec(elems, key, mid + 1, high);
    }
}

bool binarySearch(const Vector<int>& elems, int key) {
    return binarySearchRec(elems, key, 0, elems.size());
}
```

Questions to Ponder

- Express the volume of a sphere in big-O notation as a function of its radius r.
- If it takes k seconds to insertion sort 100,000 elements in random order, how long should it take to insertion sort 1,000,000 elements in random order?
  - Does your answer change if the elements are in ascending order?
  - Does your answer change if the elements are in descending order?
- Suppose you run insertion sort on an array of random values and pause it exactly halfway into its execution. What would you expect to see in the array at that point in time?
- How many recursive calls are made when using merge sort on an array of n elements? You can assume n is a power of two.
- Suppose you merge two sorted arrays where the smallest element of the second array is bigger than the largest element of the first array. What happens? Using that, modify mergesort so that it runs in time O(n) on already-sorted inputs.
- *Ternary search* is like binary search, except that instead of picking the middle element, you pick elements at the $^1/_3$ and $^2/_3$ points in the array and recursively explore just one third. Implement ternary search.
- What happens if you use mergesort, but split the array into thirds instead of halves? Does that change the big-O runtime? Code this variation up.

Week 6: Class Design

Classes

- `Vector`, `Stack`, `Queue`, `Map`, etc. are ***classes*** in C++.
- Classes contain
  - an ***interface*** specifying what operations can be performed on instances of the class, and
  - an ***implementation*** specifying how those operations are to be performed.

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() const {
   return elems.size();
}

bool RandomBag::isEmpty() const {
   return size() == 0;
}
```

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size() const;
   bool isEmpty() const;

private:
   Vector<int> elems;
};
```

```
void OurStack::grow() {
    allocatedLength *= 2;
    int* newElems = new int[allocatedLength];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
```

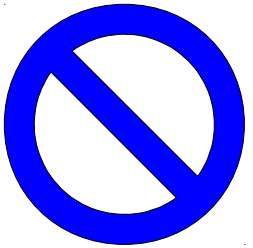
Questions to Ponder

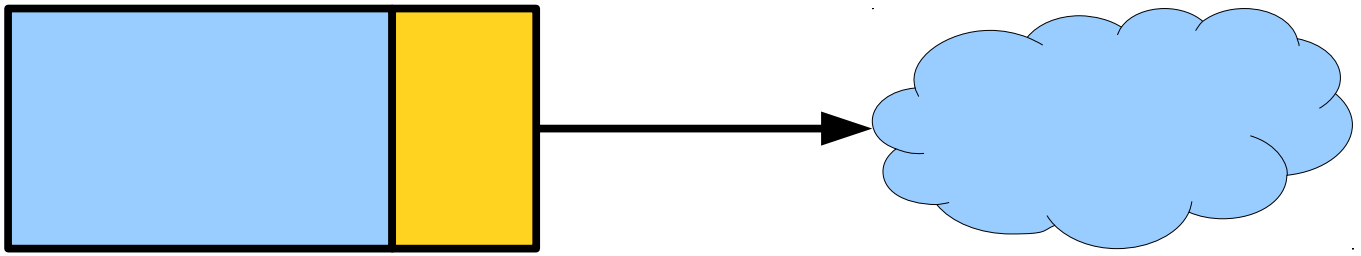
- What does it mean for a member function to be `const`?
- What's a destructor? When is it invoked?
- What happens if we swap the first two lines of `OurStack::grow()`?
- Why can we use the `size()` member function when implementing `OurStack::grow()`?
- Why can we use the `elems` variable after we `delete[]` it in `grow`?
- Why is it faster to double the size of the elements array than to increase its size by one or two when it grows?
- Why don't we just make that array significantly bigger each time, say, by squaring its size?
- How do you think `Vector` is implemented internally? Go and build it for yourself, supporting the `Vector::add`, `Vector::remove`, and `Vector::get` functions.
- Explain why the average cost of pushing an element onto a `Stack` is O(1), while the worst-case cost is still O(n).

Week 7: Linked Lists

A Linked List is Either…

…an empty list, represented by `nullptr`, or…





a single linked list cell that points…

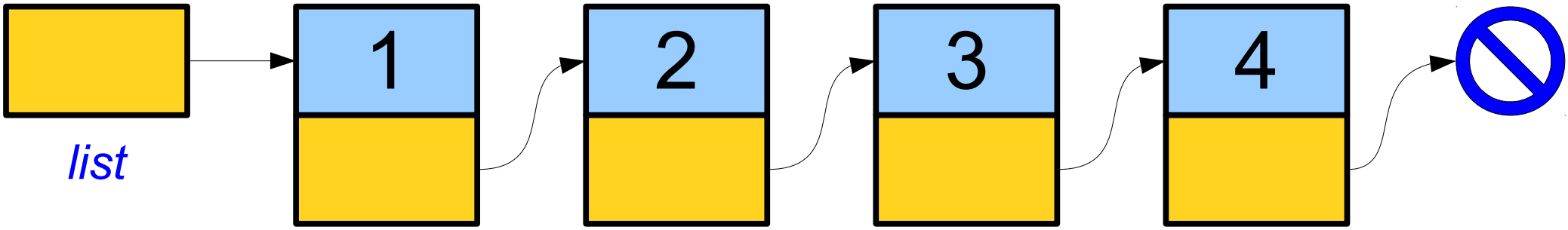
… at another linked list.

```
void printList(Cell* list) {
    if (list == nullptr) return;

    cout << list->value << endl;
    printList(list->next);
}
```

Linked Lists, Iteratively

- You can also navigate a linked list using a traditional `for` loop:

```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```

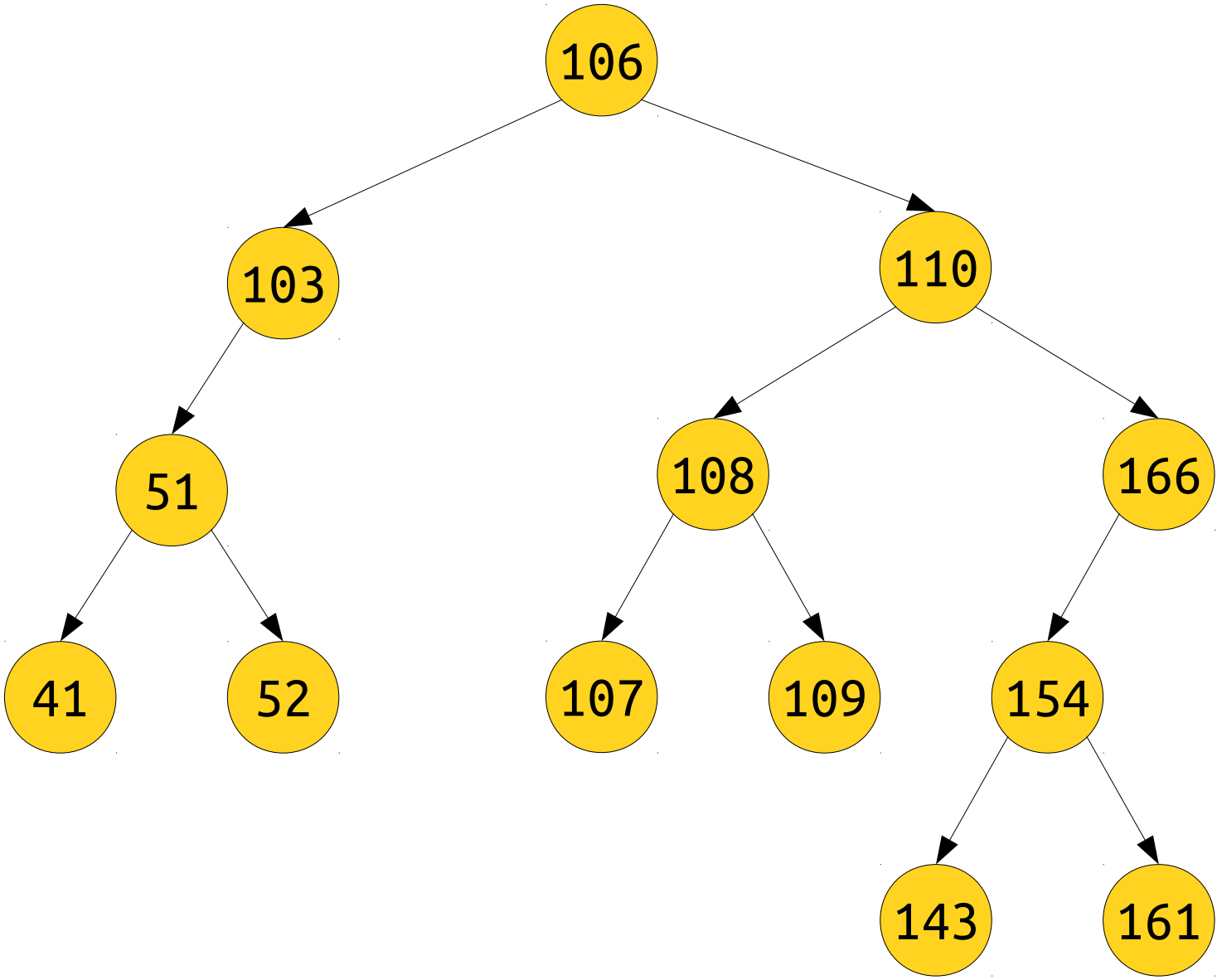


```
Cell* result = nullptr;
while (true) {
   string line = getLine("Next entry? ");
   if (line == "") break;

   Cell* cell = new Cell;
   cell->value = line;

   cell->next = result;
   result = cell;
}
return result;
```

Questions to Ponder

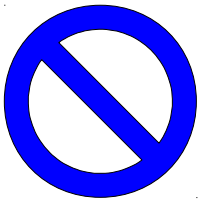
- What's one advantage of a linked list over a dynamic array?
- What's one advantage of a dynamic array over a linked list?
- What happens if you swap the last two lines of the recursive function to print a linked list? Why?
- Why can't you use the standard for loop to delete all the elements of a linked list?
- What's the space complexity of deleting a linked list recursively?
- When do you need to pass pointers into functions by reference?
- What's a tail pointer? When are they useful?
- What's a doubly-linked list? When would you use one?
- Implement the `Stack` using a linked list.
- Implement the `Queue` using a dynamic array, keeping the average cost of each operation at O(1).

Week 8: BSTs and Tries

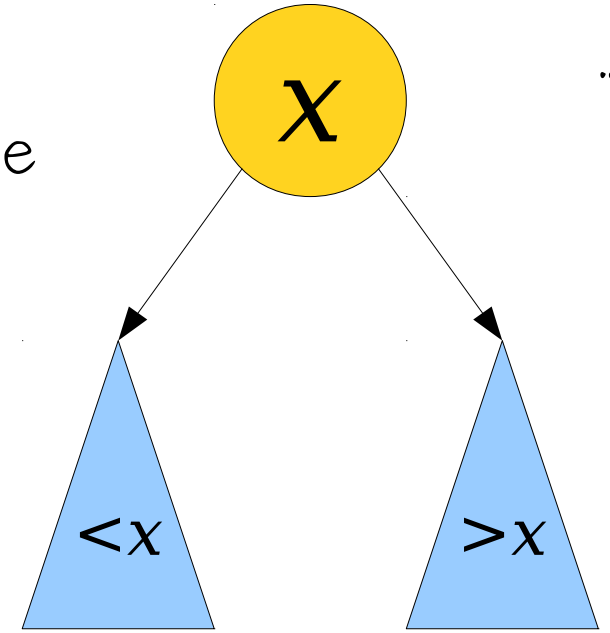

106
103
110
51
108
166
41
52
107
109
154
143
161

A Binary Search Tree Is Either…

an empty tree,
represented by
`nullptr`, or…



… a single node,
whose left subtree
is a BST of
smaller values …



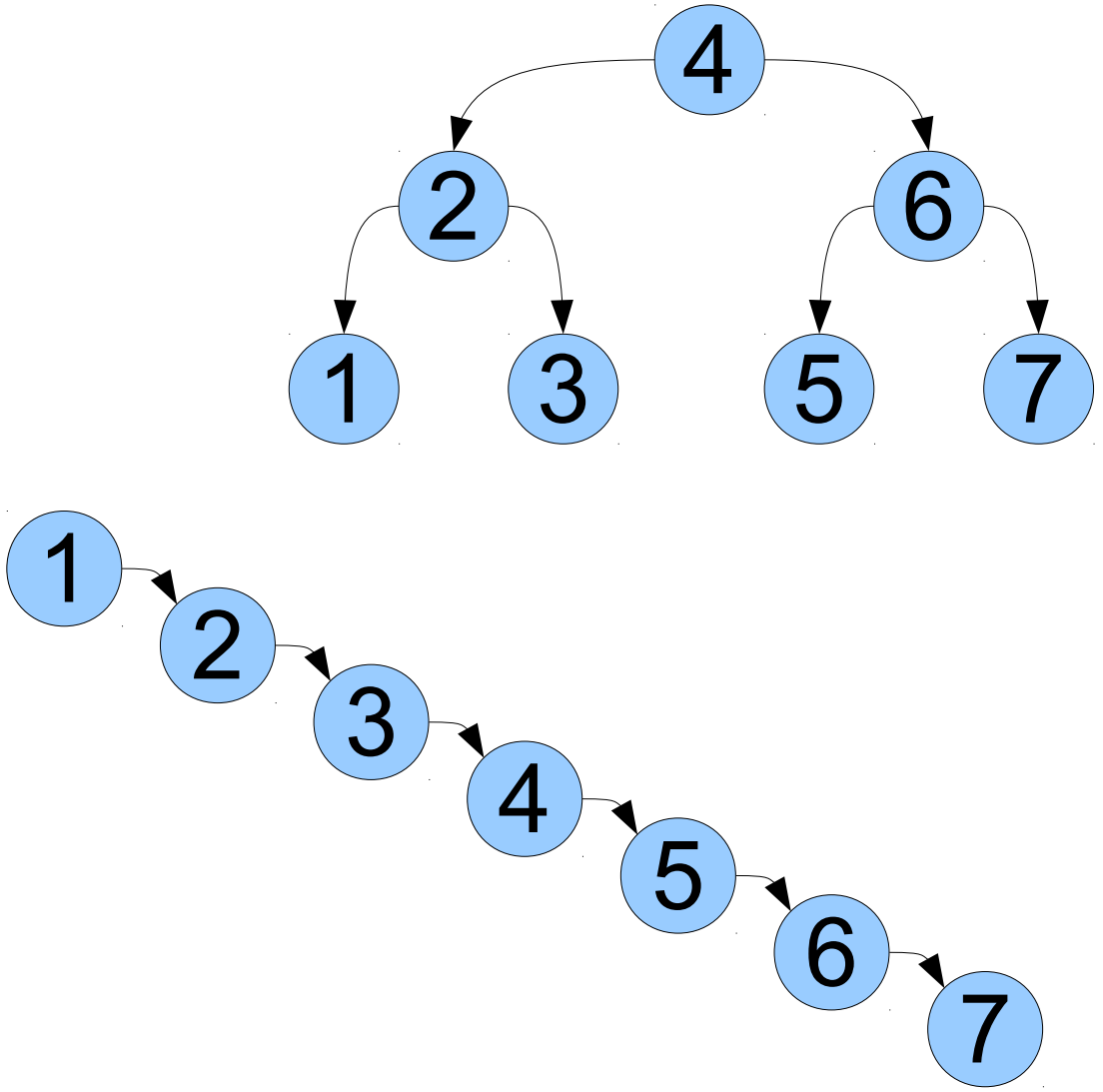
… and whose right
subtree is a BST
of larger values.

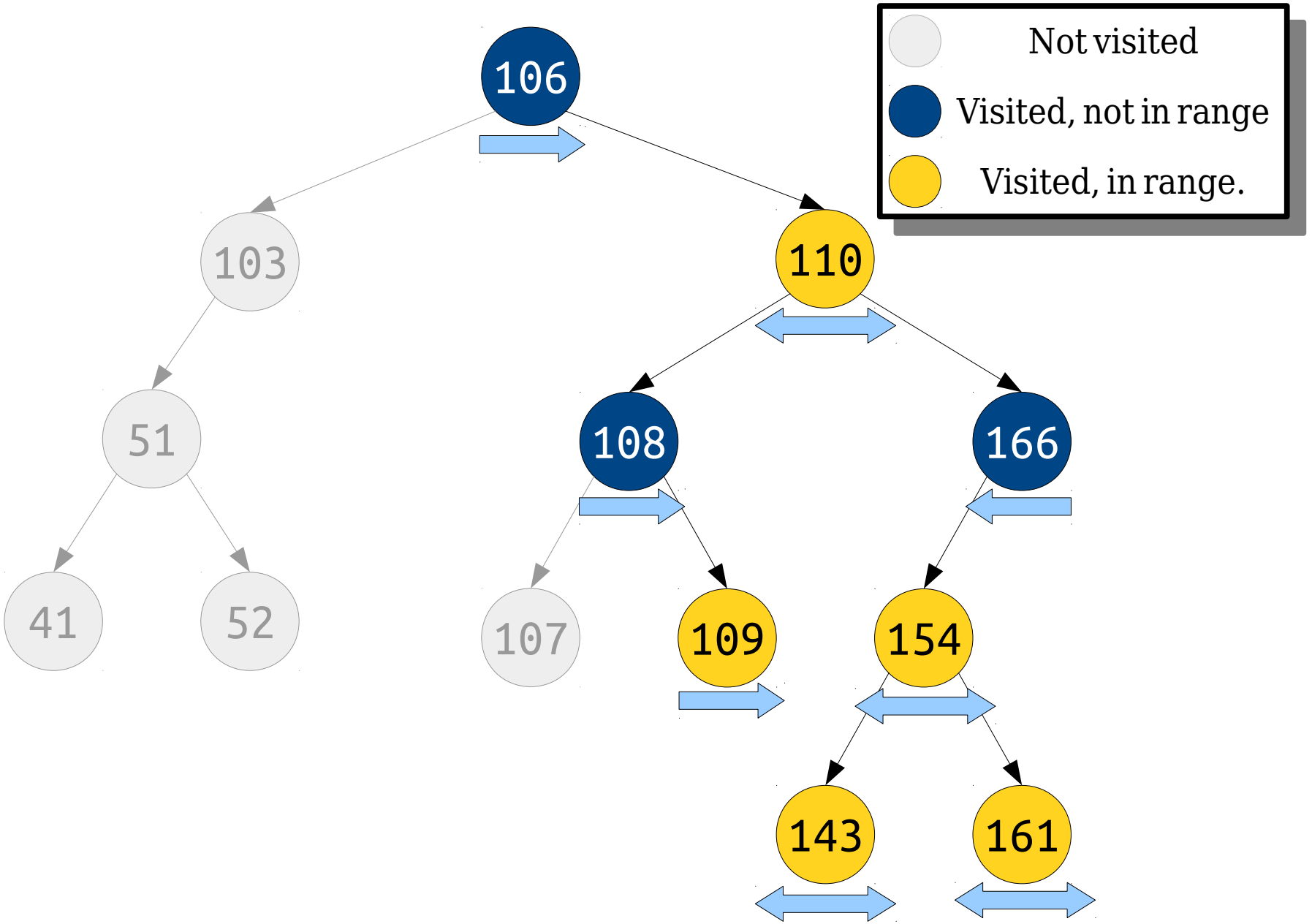
Tree Traversals

- In an ***inorder traversal***, we visit
  - the left subtree, then
  - the node itself, then
  - the right subtree.
- In a ***postorder traversal***, we visit
  - the left subtree, then
  - the right subtree, then
  - the node itself.

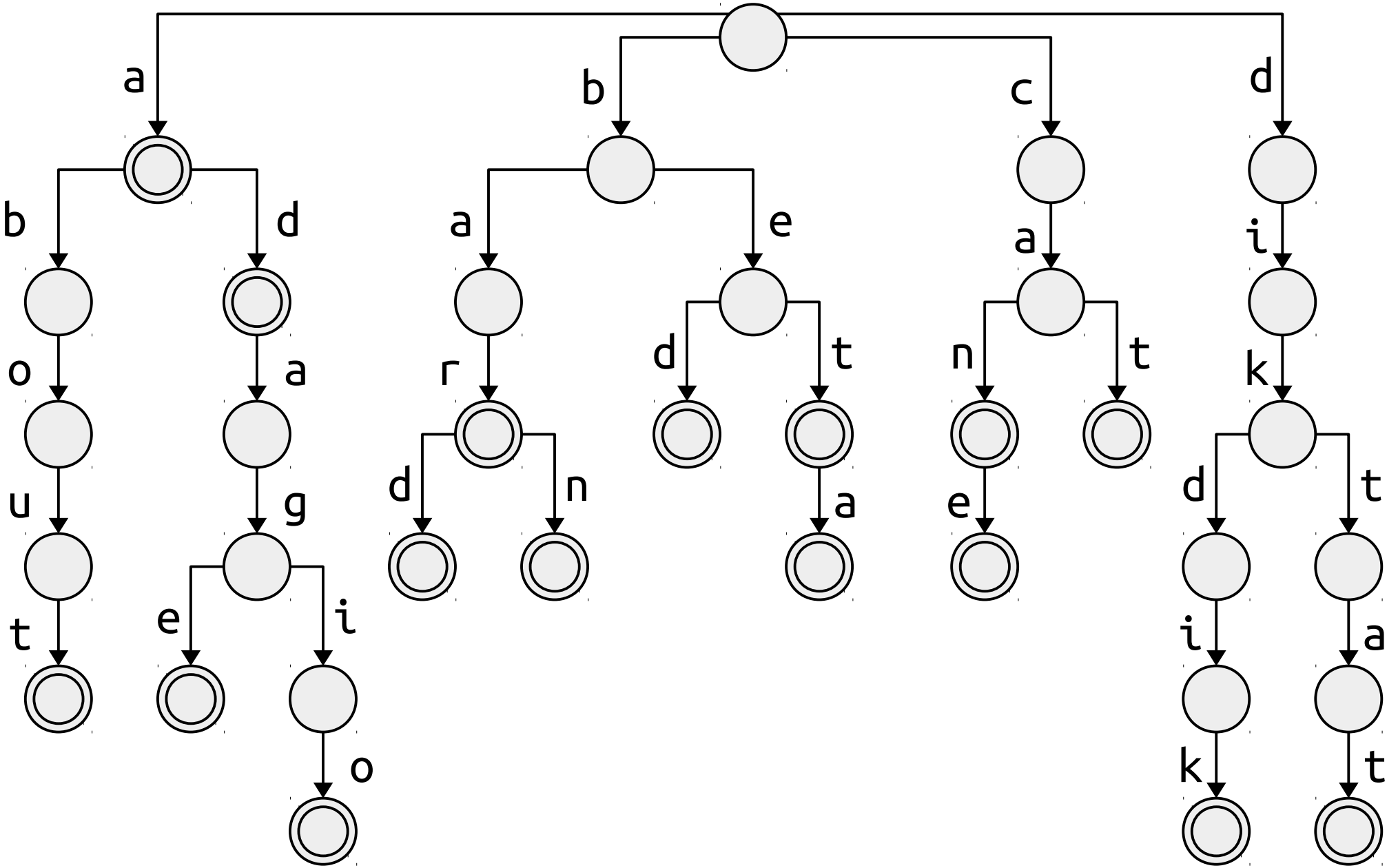
Efficiency Questions

- The time to add an element to a BST (or look up an element in a BST) depends on the height of the tree.
- The runtime is **O(*h*)**, where *h* is the height of the tree.



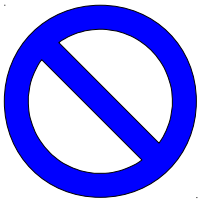


Find all elements in this tree in the range **[109, 163]**.

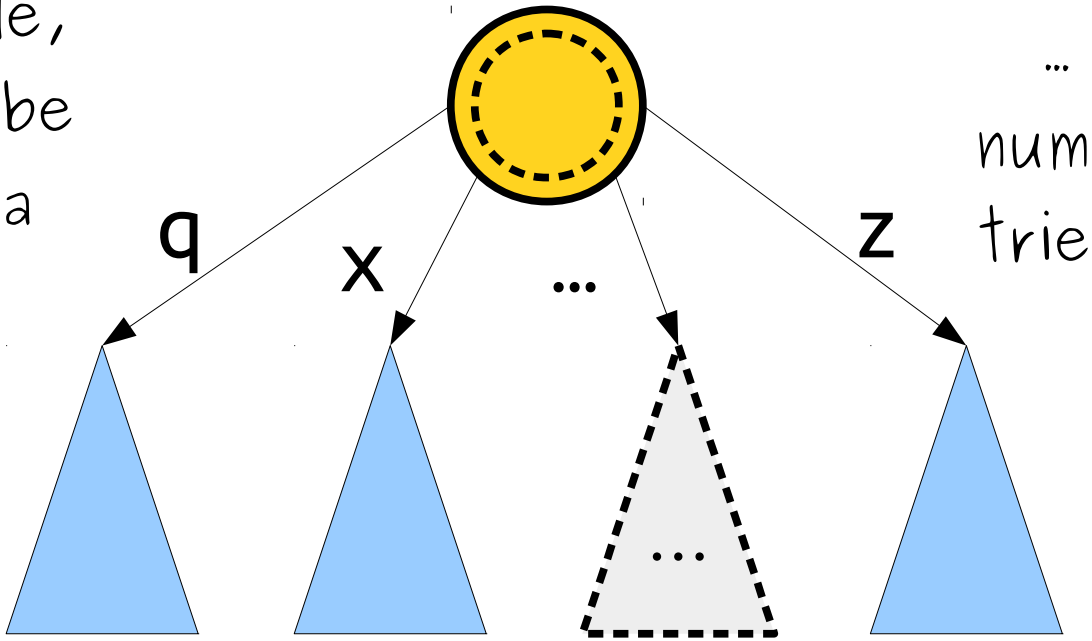

a
b
c
d
b
d
a
e
a
i
o
a
r
d
t
n
t
k
u
g
d
n
d
a
e
d
t
t
e
i
i
a
o
k
t

A Trie is Either…

an empty trie, represented by **`nullptr`**, or…



a single node, which might be marked as a word…

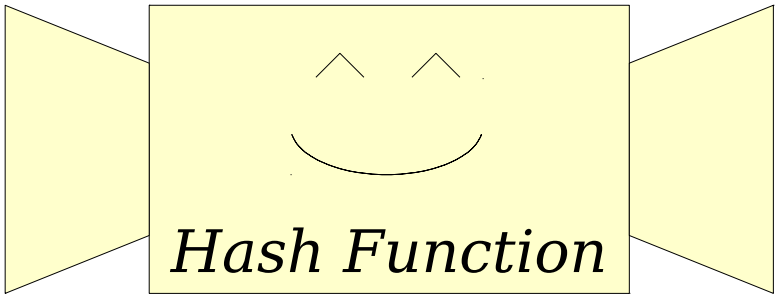


… with some number of child tries labeled by letters.

Questions to Ponder

- What is the largest possible height for a BST with n nodes? What's the smallest possible height for a BST with n nodes?
- Is it ever possible for a tree to be both a BST and a binary heap?
- Is it ever possible for a tree to be both a BST and a doubly-linked list (with "left" and "right" taking on the roles of "previous" and "next?"
- In what cases will you get the nodes in a BST back in sorted order if you use an inorder traversal?
- In what cases will you get back the nodes in a BST back in sorted order if you use a postorder traversal?
- In what cases will inserting a new node into a BST increase its height?
- Why is the runtime of inserting or looking up an element in a BST $O(h)$, where h is the height of a tree?
- Why *isn't* the runtime of a range search or inorder traversal $O(h)$?
- Give five ways to represent a node in a trie. Explain the pros and cons of each.
- Are the letters in a trie written on the nodes in the trie or on the edges?
- Give one advantage of a BST over a trie.
- Give one advantage of a trie over a BST.

Week 9: Hashing and Graphs

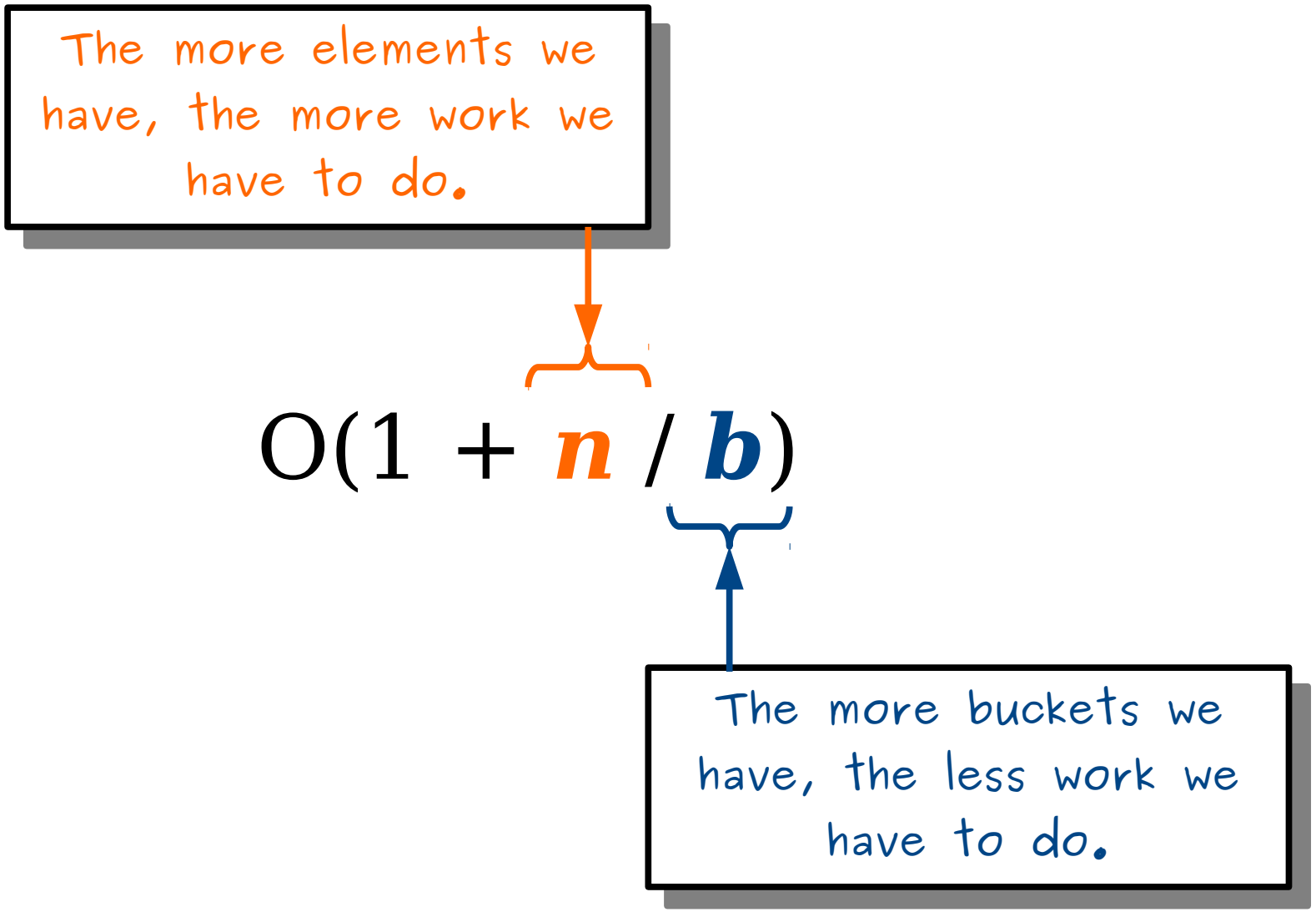


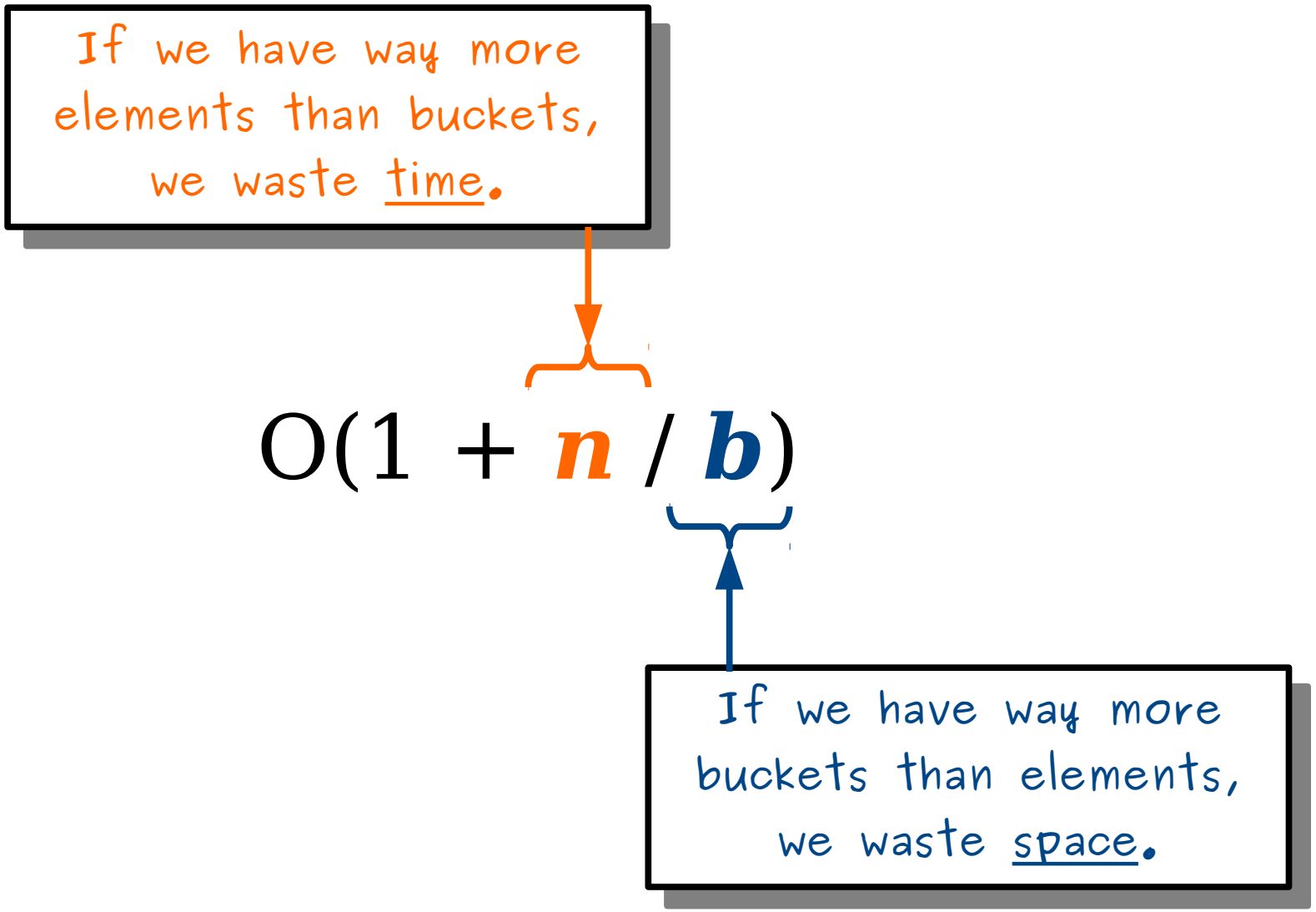
Equal inputs give equal outputs.

Unequal inputs (usually) give very different outputs.

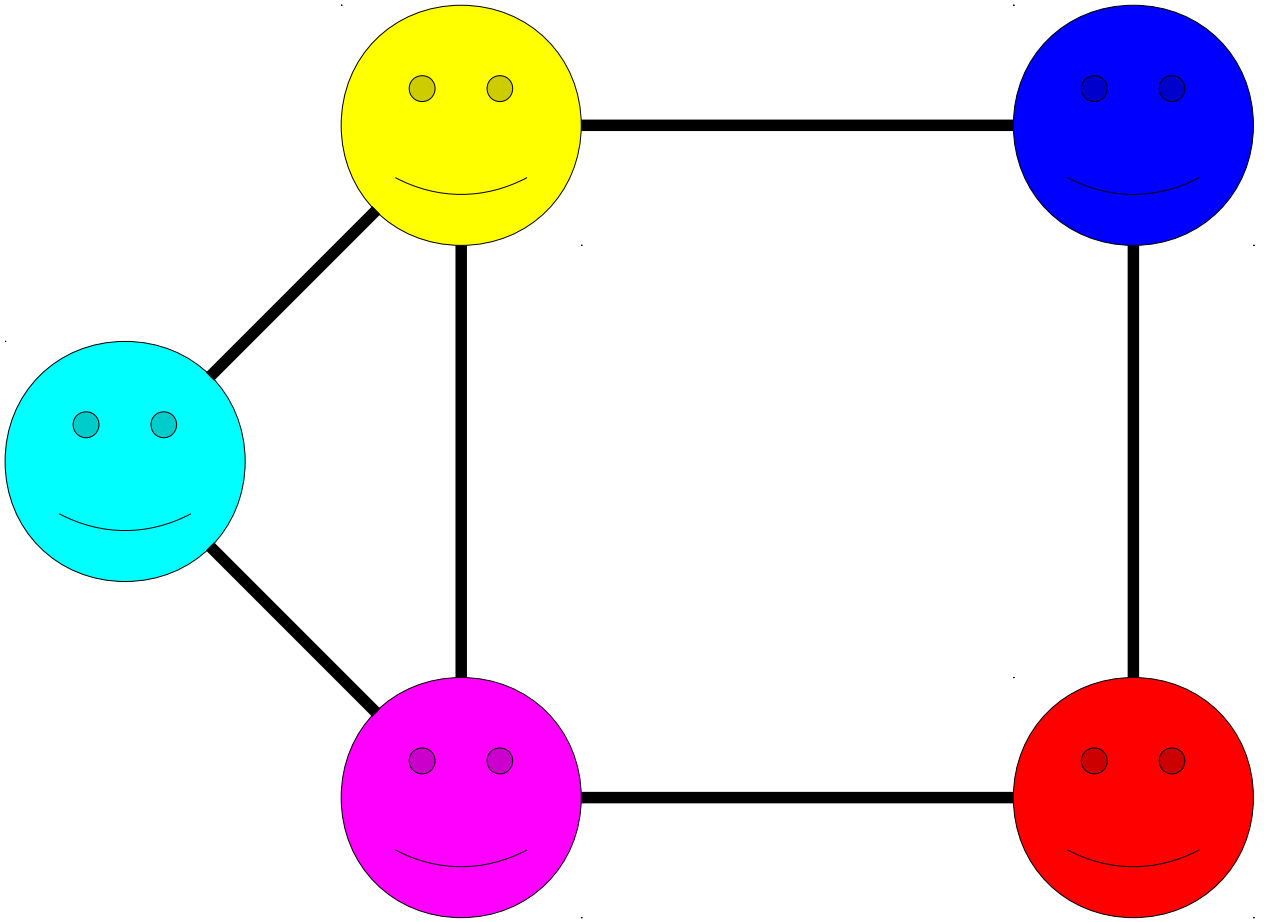
```
bool OurHashSet::contains(const string& value) const {
    int bucket = hashCode(value) % buckets.size();

    for (string elem: buckets[bucket]) {
        if (elem == value) return true;
    }

    return false;
}
```

```
void OurHashSet::add(const string& value) {
    int bucket = hashCode(value) % buckets.size();

    for (string elem: buckets[bucket]) {
        if (elem == value) return;
    }

    buckets[bucket] += value;
}
```


The more elements we have, the more work we have to do.
O(1 + n / b)
The more buckets we have, the less work we have to do.

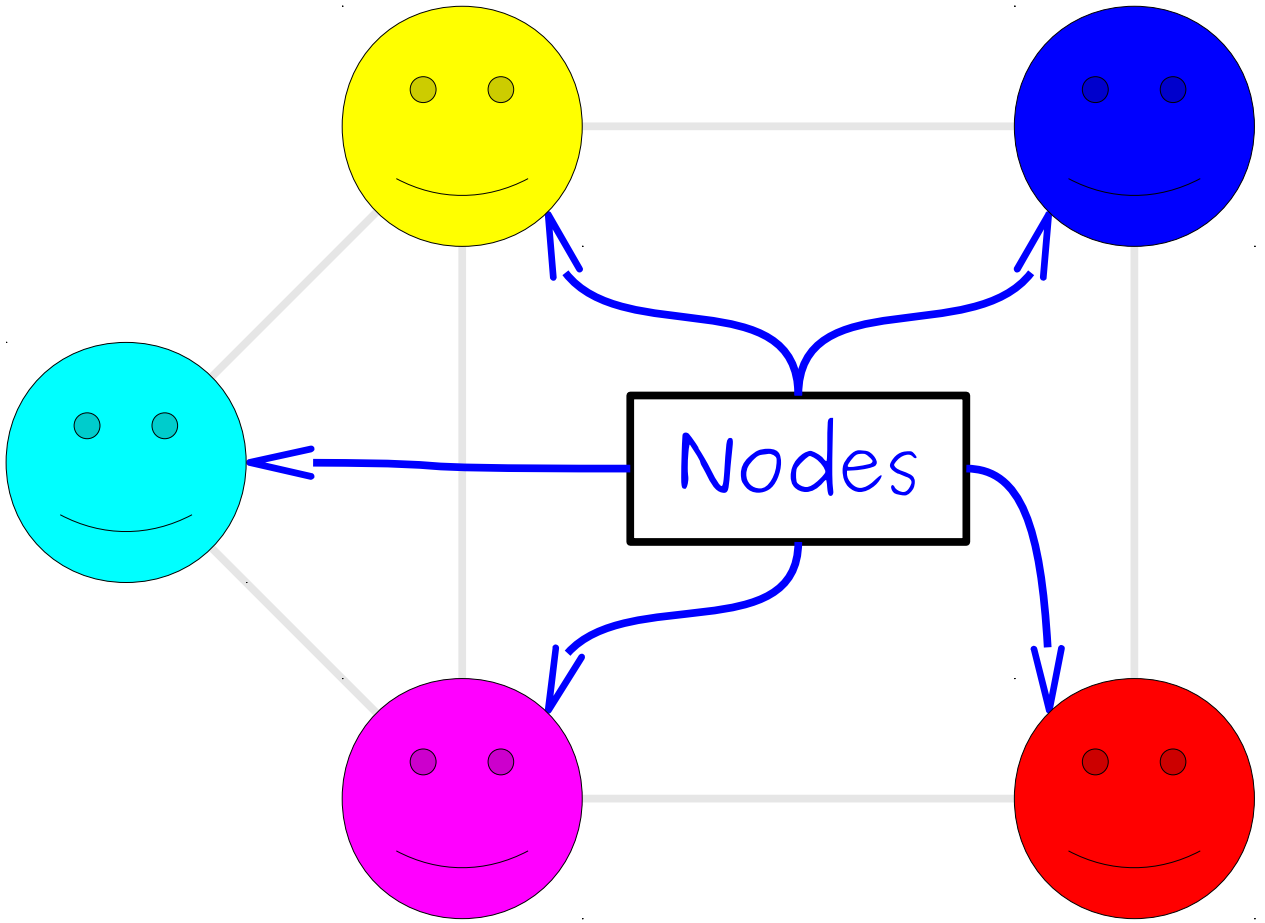

If we have way more elements than buckets, we waste time.
O(1 + n / b)
If we have way more buckets than elements, we waste space.

A ***graph*** is a mathematical structure for representing relationships.



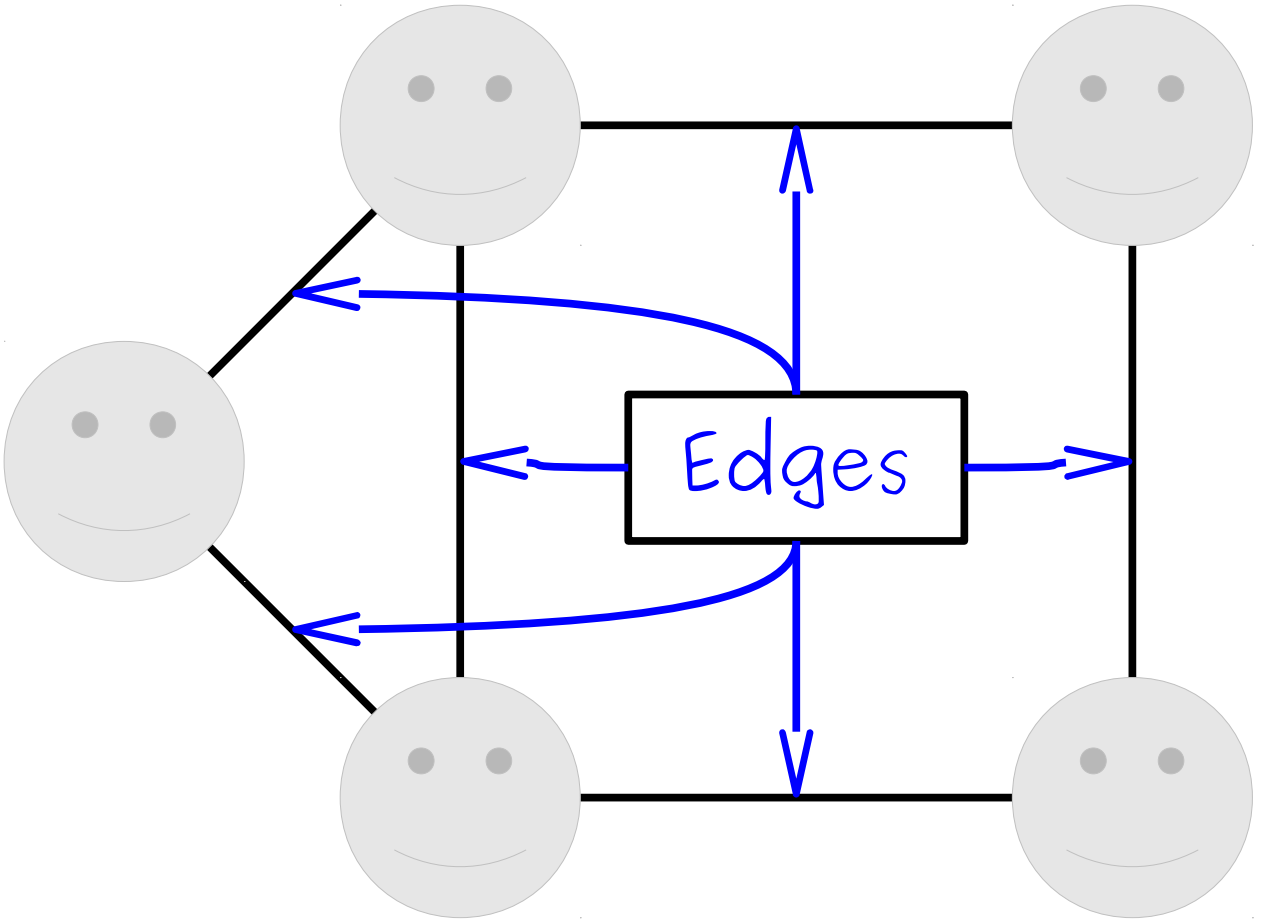
A graph consists of a set of ***nodes*** connected by ***edges***.

A ***graph*** is a mathematical structure for representing relationships.



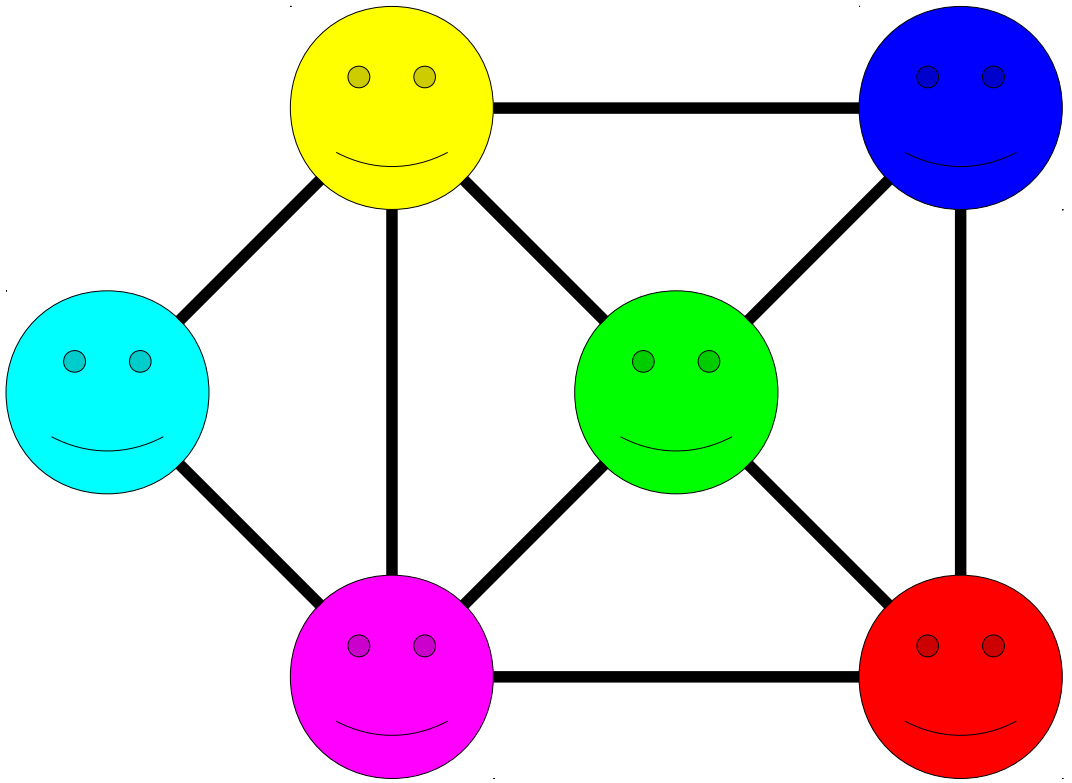
A graph consists of a set of ***nodes*** connected by ***edges***.

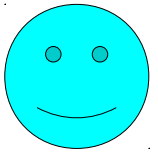
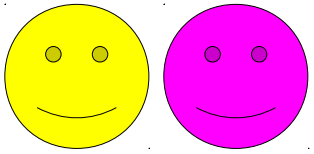
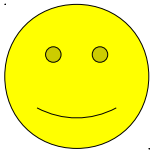
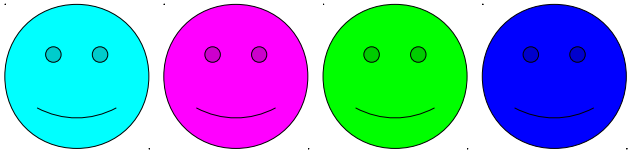
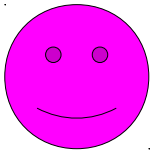
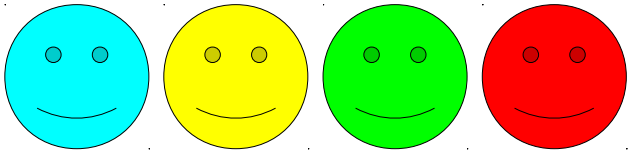
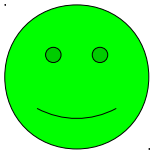
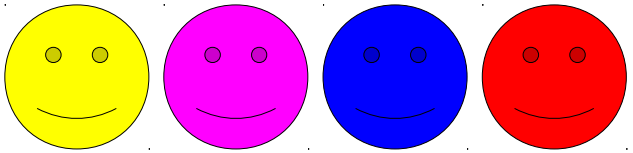
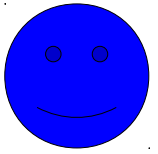
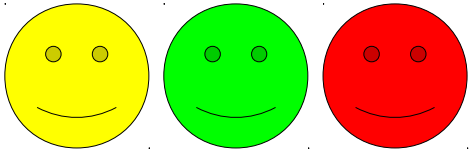
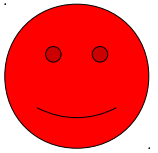
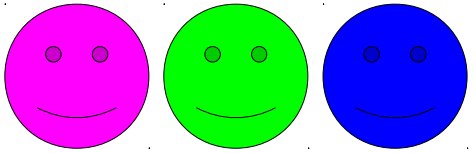
A ***graph*** is a mathematical structure for representing relationships.



A graph consists of a set of ***nodes*** connected by ***edges***.

We can represent a graph as a map from nodes to the list of nodes each node is connected to.

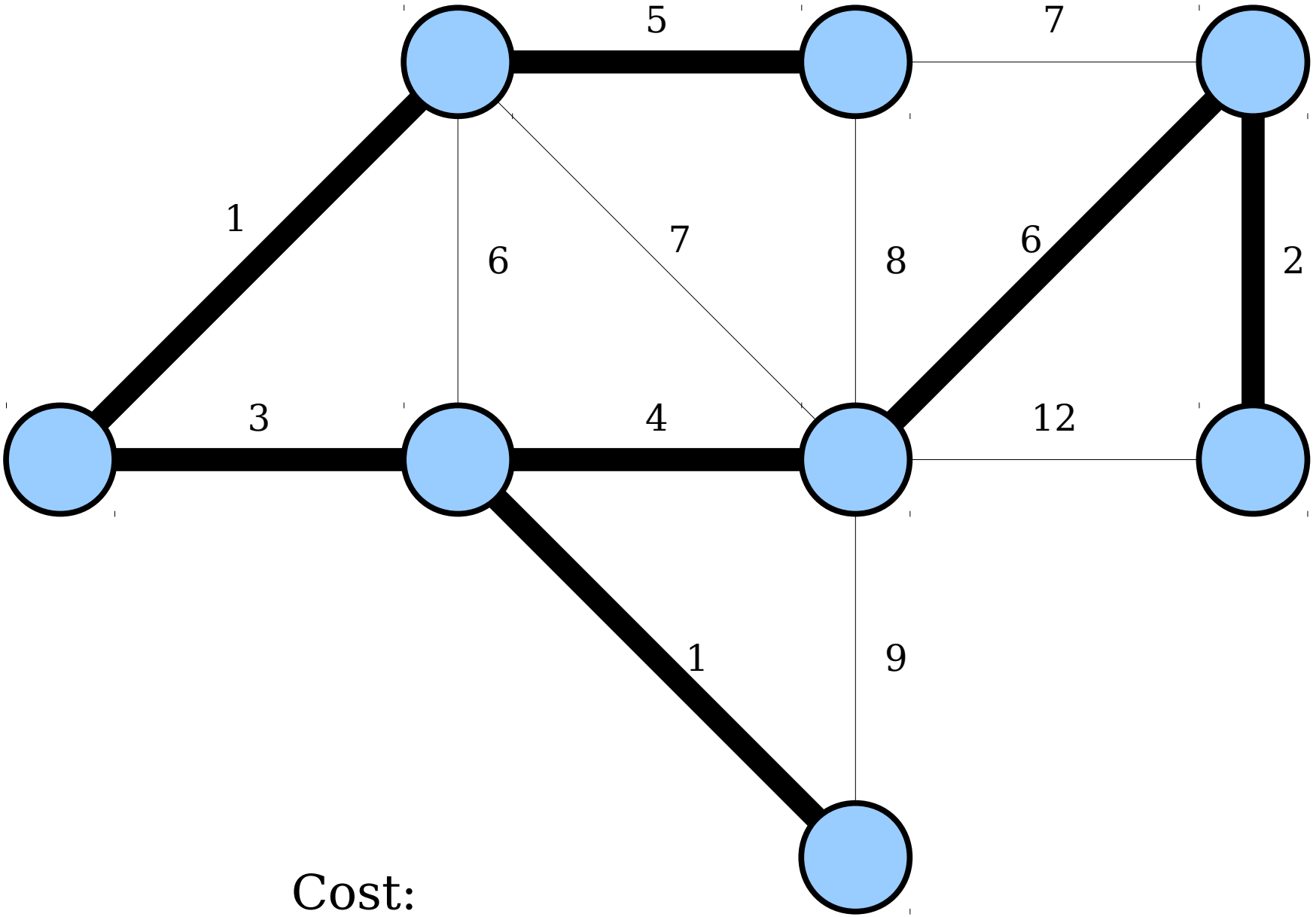


Node	Adjacent To
	
	
	
	
	
	

```
bfs-from(node v) {
    make a queue of nodes, initially seeded with v.

    while the queue isn't empty:
        dequeue a node curr.
        process the node curr.

        for each node adjacent to curr:
            if that node has never been enqueued:
                enqueue that node.
}
```

```
dfs-from(node v) {
    if this is first time we've called dfs-from(v):
        process node v
        for each node adjacent to v:
            call dfs-from on that node
}
```

```
dfs-topological-sort() {
    result = []
    for each node in the graph, in whatever order sparks joy:
        run a recursive DFS starting from that node.
        when you finish visiting a node, append it to result.
    return the reverse of result
```



Cost:

1 + 3 + 5 + 4 + 1 + 6 + 2 = **22**

A ***spanning tree*** in an undirected graph is a set of edges with no cycles that connects all nodes.

A ***minimum spanning tree*** (or ***MST***) is a spanning tree with the least total cost.

Kruskal's Algorithm:

Remove all edges from the graph.

Repeatedly find the cheapest edge that doesn't create a cycle and add it back.

The result is an MST of the overall graph.

Questions to Ponder

- Why is there the for loop in the code for inserting into a hash table?
- What's one advantage of an adjacency list over an adjacency matrix?
- If you run BFS or DFS from some starting node in a graph, what nodes will be visited when the search ends?
- Why do BFS and DFS run in time $O(m + n)$?
- Under what circumstances will BFS find the shortest path from the start node to each other node in the graph?
- Under what circumstances will DFS find the shortest path from the start node to each other node in the graph?
- Find a graph where no matter where you start searching from, DFS and BFS will always visit the nodes in a different order.
- Find a graph where no matter where you start searching from, DFS and BFS will always visit the nodes in the same order.
- Under what circumstances will Kruskal's algorithm find a tree that gives the shortest paths from some node to each other node?
- Both BFS and DFS can be used to create spanning trees by choosing the edges that first discover new nodes. How are those spanning trees similar? How are they different?
- You can also perform a topological sort by finding a node with no incoming edges, removing it, and appending it to the result until no nodes remain. Code this up. Can you make it run in time $O(m + n)$?

Week 10: Wrap-Up!

Questions to Ponder

- What's something you know now that, at the start of the quarter, you knew you didn't know?
- What's something you know now that, at the start of the quarter, you *didn't* know you didn't know?
- What's something you *don't* know now that, at the start of the quarter, you *didn't* know you didn't know?

Climbing a Mountain

Next Time

- ***What Can You Do Now?***
  - What you can do is magic. Don’t ever lose sight of that.
- ***Where to Go from Here***
  - What’s next in computer science?
- ***Final Thoughts***
  - Valedictions, send-offs, and commencements.