CS 2110
Lecture 10
Trees
Chapters 24-26
Administrivia

- A2 scores released
- A3 due today/tomorrow
- A4 out today
- Test2 next week
Hierarchies

Scene graphs
Hierarchies

Spatial refinement
Sequences are insufficient

• Need to represent parent-children relationships

• Abstraction: trees (branching relationships)
Rules for trees

• Composed of nodes
• One root node (no parent)
• Other nodes have exactly 1 parent and are reachable from root
• No loops allowed (can’t have an ancestor as a child)
Tree terminology

• Leaf
• Parent/child
• Ancestors/decedents
• Subtree
• Height/depth ("level")

Beware: 0-based vs. 1-based convention for height and depth
Restricted trees

General tree

• Each node may have any number of children; might not be ordered

Binary tree

• Each node has at most 2 children, distinguished as “left” and “right”
Example: decision tree
Trees are recursive

• A tree has a value, a left subtree, and a right subtree

• Extension of linked list Node – just add a second “next” pointer
Demo: Node class and recursive operations
Poll: How many nodes are in a tree rooted at this?

A. $\text{left.size()} + \text{right.size()}$
B. $\text{left.size()} + \text{right.size()} + 1$
C. $\text{left.size()} + \text{right.size()} + \text{data}$
D. $\max(\text{left.size()}, \text{right.size()})$
How many nodes are in a tree rooted at this?

<table>
<thead>
<tr>
<th>Option</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>left.size() + right.size()</td>
<td>0%</td>
</tr>
<tr>
<td>left.size() + right.size() + 1</td>
<td>0%</td>
</tr>
<tr>
<td>left.size() + right.size() + data</td>
<td>0%</td>
</tr>
<tr>
<td>max(left.size(), right.size())</td>
<td>0%</td>
</tr>
<tr>
<td>None of the above</td>
<td>0%</td>
</tr>
</tbody>
</table>
Poll: How high is a tree rooted at this?

A. left.height() + right.height()
B. left.height() + right.height() + 1
C. max(left.height(), right.height())
D. max(left.height(), right.height()) + 1
**How high is a tree rooted at this?**

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>left.height() + right.height()</td>
<td>0%</td>
</tr>
<tr>
<td>left.height() + right.height() + 1</td>
<td>0%</td>
</tr>
<tr>
<td>max(left.height(), right.height())</td>
<td>0%</td>
</tr>
<tr>
<td>max(left.height(), right.height()) + 1</td>
<td>0%</td>
</tr>
<tr>
<td>None of the above</td>
<td>0%</td>
</tr>
</tbody>
</table>
Complexity of size() and height()

• Time complexity
  • Must visit every node
  • Constant amount of work per node (including finding each child)
  • $O(\text{size})$

• Space complexity
  • Will accumulate stack frames until reaching a leaf
  • Stack will be popped between left and right subtrees
  • $O(\text{height})$
Binary search trees
Searching (contains())

- Binary search finds things quickly in by eliminating half of the array from consideration at each step

- Decision trees reach conclusions quickly by eliminating many possibilities from consideration at each step

- Can we store a collection in a tree to find things quickly?
  - Impose an invariant on subtrees to let us eliminate one of them each step
The BST invariant

1. All values in left subtree are *less* than this node’s value

2. All values in right subtree are *greater* than this node’s value
Poll: Which is not a BST?
Operations: contains()

• Base case
  • Current node’s data matches target
  • Next subtree to search is empty

• Recursive case
  • If target < current node’s data, recurse left
  • Otherwise, target > current node’s data, so recurse right
Operations: add()

• Same as contains(), but tweak base case action

• If next subtree to search is empty, replace it with a new node containing the new value
Complexity

• Time complexity
  • Worse-case base case: stop at leaf
  • Only visited leaf’s ancestors. Number of nodes along path from root is the leaf’s depth
  • Worst-case (maximum) depth is tree’s height
  • \( O(\text{height}) \)

• Space complexity
  • One recursive call per invocation until base case
  • \( O(\text{height}) \)
  • Can be written iteratively (tail recursive), which would use \( O(1) \) space
How does height relate to size (N)?

- Worst case: linked list
  - height == N

- Best case: each level has twice as many nodes as previous level
  - height ~ lg(N)

- Want **balance**: sibling subtrees with similar heights
  - Not guaranteed by our add algorithm – balance depends on insertion order
Operations: remove()?

• Moving a node’s children while preserving BST invariant is a little tricky

• Textbook spends 13 pages discussing this
A Set implementation using a BST

• State: root node
• Operations: add, contains, size, (remove), (iterator)
• Demo
  • Empty set: null root
Traversal: in-order

• To visit nodes in order,
  • Traverse left subtree
  • Visit self
  • Traverse right subtree
Expression trees

• $3 \times (1 + 4 \times 1) / 5$

• A binary tree (but not a BST)
  • (Assuming only binary operators)

• Leaves contain literals

• Parents contain operators
Traversal: pre-order, post-order

• In-order traversal needs parentheses to preserve order of operations

• Pre-order: visit self, then traverse left subtree, then right subtree

• Post-order: traverse left subtree, then right subtree, then visit self
Comparable
Tree traversals
Review: Binary search trees

• BST invariant
  1. All values in left subtree are \textit{less} than this node’s value
  2. All values in right subtree are \textit{greater} than this node’s value

• \textit{Find} algorithm
  1. If target matches value, return
  2. Else, if target is less than value, search left
  3. Else, search right
Comparing objects

• Can only use <, > operators on primitive types
  • What about Strings? Dates?
• The \texttt{Comparable<T>} interface
  • \texttt{int compareTo(T other)}
    • Negative if this < other
    • Positive if this > other
    • Zero if this and other compare as “equal”
      • Tip: stay consistent with \texttt{equals()}

Class declaration
\begin{verbatim}
class String implements Comparable<String>
\end{verbatim}

Client code
Instead of:
\begin{verbatim}
if (a < b)
\end{verbatim}
write:
\begin{verbatim}
if (a.compareTo(b) < 0)
\end{verbatim}
Implementing find, insert for trees of Comparables
Comparable type parameters

class BstNode<T extends Comparable<T>> {
    // ...
}

Requires that T must implement Comparable<T>
    (an instance of T can be compared with other instances of T)
See DSAJ for more flexible constraint
A Set implementation using a BST

• State: root node (null if set is empty)

• Operations: add, contains, size, (remove), (iterator)

• Time efficiency
  • Add: O(height)
  • Contains: O(height)

• Since height <= size, potentially more efficient than linked chains, arrays
Alternative: Comparator<T>

• What if there’s more than one way to order values of a type?
  • Should Countries be ordered by area? population? GDP?

• What if a client wants a different ordering than the implementer of the type provided?
  • E.g. ignore case when comparing strings

• Solution: delegate ordering to an independent object

• Required method: int compare(T a, T b)
Iteration: traversing in order

• To visit nodes in order,
  1. Traverse left subtree
  2. Visit self
  3. Traverse right subtree

• Can implement recursively in $O(\text{height})$ space
  • Need to know how to get back to self after traversing left

• Does not make sense for general trees (which children come before vs. after self?)
Alternative: **preorder** traversal

- **Algorithm**
  1. Visit self
  2. Traverse left subtree
  3. Traverse right subtree

- **General trees**
  1. Visit self
  2. Traverse child subtrees (in order)

- **Example:**
  M, K, B, F, D, H, X, Q, P, W, S
Alternative: postorder traversal

• Algorithm
  1. Traverse left subtree
  2. Traverse right subtree
  3. Visit self

• Poll
  A. K, X, M, B, D, H, F, P, W, Q, S
  C. K, B, F, D, H, X, Q, P, W, S, M
  D. S, W, P, Q, X, H, D, F, B, K, M
Application: evaluating expressions

\[
\begin{align*}
\text{3} \times \text{4} \times \text{5} \div \text{1} \\
= 3 	imes 4 \times 5 \div 1 \\
= 15 	imes 5 \\
= 75
\end{align*}
\]
Loop invariants
Searching
Loops are getting harder to write
/** Returns: 1 + ... + `n`. Requires: `n` is positive. */
int sumRange(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
/** Returns: the frequency of `item` in `items`. Requires: (Omitted: bag invariant on `items` and `size`). */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (items[i].equals(item)) {
            count++;
        }
    }
    return count;
}
/** ???. */
int mystery(int[] b, int v) {
    int k = -1;
    int t = b.length;
    while (k != t - 1) {
        int e = (k + t) / 2;
        if (b[e] <= v) {
            k = e;
        } else {
            t = e;
        }
    }
    return k;
}
// Pseudocode for an algorithm in week 14
frontier = new PriorityQueue();
root.dist = 0;
frontier.push(root);
while (frontier not empty) {
g = frontier.pop();
foreach (edge (g → v) in E) {
    if (v.dist == ∞) {
        v.dist = g.dist + d;
        frontier.push(v);
    } else {
        if (g.dist + d < v.dist) {
            v.dist = g.dist + d;
            frontier.increasePriority(v);
        }
    }
}
}
Recall: rewrite for as while loop loop

```java
for (init; guard; update) {
    body;
}

init;
while (guard) {
    body;
    update;
}
```
Exercise

Explain in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```
Exercise

**Exercise** in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. 
   Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```
Exercise

**Explain** in your own words why this loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

`count` is freq of `item` in first `size` elements of `items`
Exercise

**Exercise**

Explain in your own words why this loop works.

```c
/** Returns: the frequency of `item` in `items`. 
Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
  int count = 0;
  int i = 0;
  while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
  }
  return count;
}
```

`count` is freq of `item` in first 0 elements of `items`

<table>
<thead>
<tr>
<th>Items</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>...</th>
<th>$t_{size-1}$</th>
<th>$t_{size}$</th>
<th>/</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>size-2</td>
<td>size-1</td>
<td>size</td>
<td>...</td>
</tr>
</tbody>
</table>
Exercise

Explain in your own words why this loop works.

```java
/**
 * Returns: the frequency of `item` in `items`.
 * Requires: `items` has `size` items at indices 0..`size`-1.
 */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
```

That's so important, let's add it as a comment...
This comment, the loop invariant, is the core idea of why the loop works.

```java
/** Returns: the frequency of `item` in `items`. Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in first `i` elements of `items`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count; }
```
Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, ..., j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = j</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i &gt; j</td>
<td>empty</td>
<td>2..1 is empty</td>
</tr>
</tbody>
</table>
### Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, …, j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = j</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i &gt; j</td>
<td><em>empty</em></td>
<td>2..1 is empty</td>
</tr>
<tr>
<td>Closed range</td>
<td>[i..j]</td>
<td></td>
<td>Same as i..j</td>
<td>[2..4] is 2, 3, 4</td>
</tr>
<tr>
<td>Open range</td>
<td>(i..j)</td>
<td></td>
<td>Same as [(i+1)..(j-1)]</td>
<td>(2..4) is 3</td>
</tr>
<tr>
<td>Half-open range</td>
<td>[i..j)</td>
<td></td>
<td>Same as [i..(j-1)]</td>
<td>[2..4) is 2, 3</td>
</tr>
<tr>
<td></td>
<td>(i..j]</td>
<td></td>
<td>Same as [(i+1)..j]</td>
<td>([2..4] is 3, 4</td>
</tr>
</tbody>
</table>
## Some helpful, concise notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>When</th>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>i..j</td>
<td>i &lt; j</td>
<td>i, i+1, …, j-1, j</td>
<td>2..4 is 2, 3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = j</td>
<td>i</td>
<td>2..2 is 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i &gt; j</td>
<td><em>empty</em></td>
<td>2..1 is empty</td>
</tr>
<tr>
<td>Closed range</td>
<td>[i..j]</td>
<td></td>
<td>Same as i..j</td>
<td>[2..4] is 2, 3, 4</td>
</tr>
<tr>
<td>Open range</td>
<td>(i..j)</td>
<td></td>
<td>Same as [(i+1)..(j-1)]</td>
<td>(2..4) is 3</td>
</tr>
<tr>
<td>Half-open range</td>
<td>[i..j)</td>
<td></td>
<td>Same as [i..(j-1)]</td>
<td>[2..4) is 2, 3</td>
</tr>
<tr>
<td></td>
<td>(i..]</td>
<td></td>
<td>Same as [(i+1)..j]</td>
<td>(2..4] is 3, 4</td>
</tr>
<tr>
<td>Omitted first</td>
<td>[...j]</td>
<td>0-based</td>
<td>Same as [0..j]</td>
<td>[…2] is 0, 1, 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1-based</td>
<td>Same as [1..j]</td>
<td>[…2] is 1, 2</td>
</tr>
<tr>
<td>Omitted last</td>
<td>[i..]</td>
<td>Some last # implied</td>
<td>Same as [i..last]</td>
<td>Depends on context</td>
</tr>
</tbody>
</table>
# Some helpful, concise notation

<table>
<thead>
<tr>
<th>Context</th>
<th>Application</th>
</tr>
</thead>
</table>
| Arrays (0-based) | a[2..4] is a[2], a[3], a[4]  
                  | a[..3] is a[0], a[1], a[2], a[3]  
                  | a[1..] is a[1], ..., a[a.length – 1] |
| Strings (0-based) | “Hello”[2..4] is “llo”  
                    | “Hello”[..2] is “Hel” |
| Lists         | lst[3..4] are elements lst.get(3), list.get(4)  
                       | (Beware 0-based vs 1-based) |

⚠️ Ranges are not the same as the Python slice operator.
This comment, the **loop invariant**, is the core idea of why the loop works.

```java
/**
 * Returns: the frequency of `item` in `items`.
 * Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in `items[..i)`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count; }
```
What is a loop invariant?

• A **specification**. Just as functions need them, so do loops.

*Note: I am thanking the writer of this Ed post for asking an important question.*
What is a loop invariant?

• A **specification**. Just as functions need them, so do loops.
• Comment that describes what is always true **before** and **after** loop body, i.e., **invariant**
What is a loop invariant?

• A **specification**. Just as functions need them, so do loops.
• Comment that describes what is always true **before** and **after** loop body, i.e., **invariant**
• States the **relationship** between the important **variables** in the loop

```java
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
What is the purpose of a loop invariant?

• **Debugging:**
  • Assert loop invariant like you would a class invariant

• **Documentation:**
  • Understand meaning of loop after it has been written

• **Specification:**
  • Aid in development of a new loop
  • Understand whether existing loop is correct…
Loop checklist

- Does it start right?
- Does it maintain the invariant?
- Does it end right?
- (Does it make progress?)

**Goal:** answer these questions with aid of loop invariant to understand whether loop is correct, or what might be wrong with it.
Q: Does it start right?

• Loop invariant must be true just before `while`
• Aka establishment or initialization
• Accomplished by assigning to loop variables to `truthify` invariant

```java
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Q: Does it maintain the invariant?

• Body must **maintain** invariant:
  • Invariant must be true **after** body
  • In reasoning about that, we get to assume that invariant is true **before** body
  • We also get to assume guard holds before body executes

• Aka **preservation**

```java
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
  if (items[i].equals(item)) count++;
  i++;
}
```
Q: Does it end right?

• Loop invariant must be true just after **while**
• And **postcondition** must be established by loop invariant being true and guard becoming false

```
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Loop checklist

```java
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```

- Does it start right? ✓
- Does it maintain the invariant? ✓
- Does it end right? ✓
- (Does it make progress?) ❑
Answer, Part 2

/** Returns: the frequency of `item` in `items`.  
   Requires: `items` has `size` items at indices 0..`size`-1. */
int frequencyOf(T item, T[] items, int size) {
    int count = 0;
    int i = 0;
    // inv: `count` is freq of `item` in `items[..i)`
    while (i < size) {
        if (items[i].equals(item)) count++;
        i++;
    }
    return count;
}
Q: Does it make progress?

- Body must make **progress** toward **termination**
  - Every repetition of body must do something to get closer to making the guard false
  - Typically by strictly decreasing some quantity to zero, at which point guard becomes false
  - That quantity is called the **loop variant**

```java
int count = 0;
int i = 0;
// inv: 0 <= `i` <= `size`
// variant: `size` - `i`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```
Loop checklist

```c
int count = 0;
int i = 0;
// inv: `count` is freq of `item` in `items[..i)`
// inv: 0 <= `i` <= `size`
// variant: `size` - `i`
while (i < size) {
    if (items[i].equals(item)) count++;
    i++;
}
```

☑ Does it start right?
☑ Does it maintain the invariant?
☑ Does it end right?
☑ Does it make progress?
Flowchart

Loop precondition: code before loop

Loop initialization

Loop guard:
- True
- False

Loop invariant is true, guard is false, variant: ?
- Variant has decreased

Loop invariant is true, guard is true, variant is non-zero
- Code after loop

Loop

Postcondition
Loop invariants: are they worth the trouble?

“I really don’t understand why we are talking about loops in this fashion. It is totally unintuitive and cumbersome.”

“This ain’t it, chief.”
—Surely some anonymous CS 2110 student, Spring 2023.
Loop invariants: are they worth the trouble?

They are Cornell lore.

They are tools.
Use them when needed.

They are part of next-gen program verification tools in research at major companies.
Demo: Dafny
Java-like language developed at Microsoft
Search Algorithms
Linear Search

**Key idea:** search linearly through array from front to back to find item

```c
/** Returns: the smallest index i such that a[i] == v.  
   Requires: v is in a. */
int linear_search(int[] a, int v) {
    int i = 0;

    while (a[i] != v) i++;
    return i;
}
```
Exercise

**State** the loop invariant.

```c
/** Returns: the smallest index i such that a[i] == v. 
   Requires: v is in a. */
int linear_search(int[] a, int v) {
    int i = 0;
    // inv: TODO
    while (a[i] != v) i++;
    return i;
}
```
Discovering the loop invariant

/** Returns: the smallest index i such that a[i] == v. Requires: v is in a. */

Rule: We never draw indices directly above a line in diagram! Always to left or right.
Discovering the loop invariant

/** Returns: the smallest index $i$ such that $a[i] == v$. Requires: $v$ is in $a$. */

Discovering an invariant from the pre and post conditions requires creativity and practice.

**Theorem.** There is no algorithm that can do it for you.

**Corollary:** ChatGPT can’t replace human programmers yet!

Inv:
- $v$ not in $a[0..i)$
- $v$ in $a[i..]$
Linear Search: with invariant

/** Returns: the smallest index i such that a[i] == v.
   Requires: v is in a. */
int linear_search(int[] a, int v) {
    int i = 0;
    // inv: v not in a[0..i), and v in a[i..]
    while (a[i] != v) i++;
    return i;
}
Linear Search: loop checklist

- Does it start right?
- Does it maintain the invariant?
- Does it end right?
- Does it make progress?

```c
/** Returns: the smallest index i such that a[i] == v. Requires: v is in a. */
int linear_search(int[] a, int v) {
    int i = 0;
    // inv: v not in a[0..i), and v in a[i..]
    while (a[i] != v) i++;
    return i;
}
```
Binary Search

**Key idea:** maintain upper and lower bounds on where value could be.

```c
/** 
 * Returns: an index i such that a[i] == v. 
 * Requires: v is in a, and a is sorted in ascending order. */
int bin_search(int[] a, int v) {
    int l = 0;
    int r = a.length - 1;
    // inv: 0 <= l <= r < a.length, and v in a[l..r]
    while (l != r) {
        int m = (l + r) / 2;
        if (v <= a[m]) { r = m; }
        else { l = m + 1; }
    }
    return l;
}
```
Understanding the loop invariant

/** Returns: an index \( i \) such that \( a[i] = v \).
Requires: \( v \) is in \( a \), and \( a \) is sorted in ascending order. */

Post: \( a[0 \ldots i \ldots a.length] \leq v \leq v \leq a[0 \ldots i \ldots a.length] \)

Pre: \( a[0 \ldots a.length] \) is sorted

Inv: \( a[0 \ldots l \ldots r \ldots a.length] \leq v \leq v \leq a[0 \ldots l \ldots r \ldots a.length] \)

sorted

Nothing about the loop invariant requires halving the search space!

Efficiency is distinct from correctness.
Binary Search: loop checklist

- Does it start right?
- Does it maintain the invariant?
- Does it end right?
- Does it make progress?

```c
/** Returns: an index i such that a[i] == v.
 * Requires: v is in a, and a is sorted in ascending order.
 */
int bin_search(int[] a, int v) {
    int l = 0;
    int r = a.length - 1;
    // inv: 0 <= l <= r < a.length, and v in a[l..r]
    while (l != r) {
        int m = (l + r) / 2;
        if (v <= a[m]) { r = m; }
        else { l = m + 1; }
    }
    return l;
}
```
Loops (in)variants are your friends.