Data Structures
Exam II Review

Daniel Noland

October 23, 2014

1. *Q.* How does a binary search tree's shape depend on the order of the numbers inserted into it?

   A. A binary search tree's shape is *entirely* dependant on the order of insertions. If you insert into a binary tree in monotonically increasing or decreasing order, you will just get a (worse) linked list (think about it!). See figure 1 for an example of a randomly filled binary search tree, which is poorly "balanced."

2. *Q.* What parts are similar in the two processes of searching a binary tree and searching a sorted array by binary search? What parts are different? What depends on luck?

   A. Binary search of a sorted array is similar to searching a binary tree in that (for a balanced binary search tree) the problem size is shrinking by a factor of two at every step. Note that this is the best case for a binary search tree, as it is possible for the tree to be wildly out of balance. In the worst case, you end up with a linked list, which just gives linear seach time. If the tree is being populated with random data, it is largely luck dependant how well balanced the tree is. See figure 1 for an example.

3. *Q.* Given an arbitrary binary tree, print it out in preorder, inorder and postorder.

   A. If you insist. See algorithm 1 for preorder, algorithm 2 for inorder and algorithm 3 for postorder.

Algorithm 1 Source code of a simple preorder binary tree printer.

```
#include <iostream>

void print_preorder(binary_search_tree_node* node, std::ostream& out = std::cout) {
   if(node) {
      out << node->data << std::endl;
      print_preorder(node->left, out);
      print_preorder(node->right, out);
   }
}
```

4. *Q.* Given a bunch of numbers, in some order, insert them into a binary search tree.

   A. The rule is simple: if whatever number we are going to insert is less than the number of the node we are at, go left and repeat. Otherwise, go right and repeat. Stop when you find an empty node, because that is where your data should be inserted.

   Say you had the following list:

   62, 31, 22, 30, 87, 74, 15, 39, 70, 53, 48, 52, 83, 55, 92, 68

Algorithm 2 Source code of a simple inorder binary tree printer.

```cpp
#include <iostream>

void print_inorder(binary_search_tree_node* node, std::ostream& out = std::cout) {
   if(node) {
      print_inorder(node->left, out);
      out << node->data << std::endl;
      print_inorder(node->right, out);
   }
}
```

Algorithm 3 Source code of a simple postorder binary tree printer.

```cpp
#include <iostream>

void print_preorder(binary_search_tree_node* node, std::ostream& out = std::cout) {
   if(node) {
      print_preorder(node->left, out);
      print_preorder(node->right, out);
      out << node->data << std::endl;
   }
}
```

Those numbers would (assuming I did not make any mistakes) fit into a binary search tree as in figure 2.

5. (a) *Q.* Given the binary search code and a particular array of sorted numbers, tell me the first array slot the search code will check to find 3 in the array 1 3 5 6 8 9 11 14. What is the last array slot a search for the 3 will check?

   A. See figure 3 for details.

   (b) *Q.* Repeat the question but look for a number that's not in the array, like 10. What will be the last slot checked?

   A. Again, pictures. See figure 4.

6. *Q.* To get the 6 big_number comparison functions `==`, `!=`, `<`, `>`, `<=`, and `>=`, how many must you write and why, and what can you do for the other ones instead of writing them all from scratch?

   A. Ah, a logic puzzle! You just need one, but two is much more efficient. I would always pick `<` and `==` for template programming reasons. Check these answers, as I am *very tired* right now, and could easily make a logic error here.

   - $a = b \Leftrightarrow \neg(a < b) \wedge \neg(b < a)$, which is the same as $\neg\left((a < b) \vee (b < a)\right)$
   - $a \neq b \Leftrightarrow \neg(a = b)$, alternatively $(a < b) \vee (b < a)$
   - $a > b \Leftrightarrow b < a$
   - $a \leq b \Leftrightarrow \neg(b < a)$
   - $a \geq b \Leftrightarrow \neg(a < b)$

   So, basically, logic to the rescue. With that, each comparison operator can call some variant of `operator<` (possibly including `operator==` as an independent implementation), and you are done. Code reuse FTW. This is also why the STL uses `operator<` (or `std::less`) for all its sorting algorithms.

7. *Q.* What time penalty comes from using the `add_node` function when copying a list?

   A. It moves the operation from $O(n)$ to $O(n^2)$. The `add_node` function only has access to the `head_ptr` node, and so it is forced to "walk" to the end of the list again and again as we repeatedly call `add_node`. In contrast, an implementation written specifically for the purpose of copying a list (or even `std::copy`, provided you are smart enough to write your own iterator), would be much faster. See algorithm 4 for a simple pseudocode example.

Algorithm 4 Efficient List Copy

```
 1: procedure LIST_COPY(head_ptr)
 2:     if head_ptr = nullptr then
 3:         return nullptr
 4:     cursor ← head_ptr
 5:     target ← new node
 6:     new_list ← target
 7:     while cursor ≠ nullptr do
 8:         target ← new node
 9:         target->data ← cursor->data
10:         cursor ← cursor->next
11:         target ← target->next
12:     return new_list
```

8. *Q.* When is a binary search tree most efficient? Least efficient? Why?

   A. A binary search tree is *most* efficient when it is **perfectly balanced**. It is least efficient when its depth is equal to the number of nodes it contains (i.e., it has degenerated into a linked list). The point of a binary search tree is to provide for fast lookup and insert. It does this by putting its data in a "sorted" order. This order is such that, at every step of the insert and search algorithms, the problem size is cut in half, providing $O(\log n)$ execution time. This is best case for a binary search tree. In the worst case, the tree has degenerated into a line (linked list), and you get linear ($O(n)$) execution time.

9. *Q.* Given the code in `bintree.cpp`, can you make a function that multiplies every number in a binary tree by 7?

   A. Yeah, and I can do one better. Algorithm 5 will multiply any binary tree by any valid int. Make everything adjustable!

Algorithm 5 `C++` function which multiplies every node beneath the given node by a given factor (which defaults to 7).

```cpp
const int DEFAULT_FACTOR = 7;

void binary_tree_multiply(binary_tree_node* node, int factor = DEFAULT_FACTOR) {
   if(node) {
      node->data *= factor;
   }
   binary_tree_multiply(node->left, factor);
   binary_tree_multiply(node->right, factor);
}
```

10. *Q.* Given the code in `bintree.cpp`, can you make a function that reverese (mirror images) a binary search tree?

A. Yeah, but the output will be of a different type than the input (i.e., it will be a reversed binary search tree, not a normal one). For pseudocode, see algorithm 6. For a full `C++` implementaton, see algorithm 7.

Algorithm 6 Binary Search Tree Mirror

```
1: procedure MIRROR_TREE(src_node, dst_node)
2:     if src_node ≠ nullptr then
3:         if dst_node = nullptr then
4:             dst_node ← new node
5:         dst_node->data ← src_node->data
6:         MIRROR_TREE(src_node->left, dst_node->right)
7:         MIRROR_TREE(src_node->right, dst_node->left)
```

Algorithm 7 C++ implementation of Binary Search Tree Mirror

```cpp
void mirror_tree(binary_search_tree_node* src, binary_tree_node* dst) {
   if(src) {
      if(!dst) {
         dst = new binary_tree_node;
      }
      dst->data = src->data;
      mirror_tree(src->left, dst->right);
      mirror_tree(src->right, dst->left);
   }
}
```

11. *Q*. If you had a mirror imaged binary search tree, what would you need to do when inserting data into it?

    A. Follow the opposite rules as normal. If the default rule is to go left when `<`, and right otherwise, then you need to go left when *not* `<` (i.e., when `>=`) and right otherwise. See algorithm 8.

12. *Q*. Why is self assignment a problem for `operator=`?

    A. Self assignment is a problem for `operator=` because of resource management. Consider algorithm 9. It will fail if we call `a = a`, because we deallocate our dynamic resources *before* we do the assignment logic. But if we don't deallocate our resources before assignment, then we will leak those resources. This is actually easy to fix, just check for self assignment *before* resource release. See algorithm 10 for comparison.

13. *Q*. Why is self assignment *not* a problem for the copy constructor?

    A. Self assignment is *never* a problem for *any* kind of constructor. `C++` always calls the constructor when (naturally) a new object needs to be constructed. You can't construct an object from itself, so the whole idea is a complete non issue.

14. *Q*. What is the difference between an assignment operator and a copy constructor?

    A. The real question is, what is the same about them? From your compiler's perspective, `operator=` is pretty much just a normal function (that is actually not *quite* true, but don't worry about that for now). It just happens to be called in a somewhat unusual way. Really, basically the only thing which makes `operator=` special is that the compiler will write this function for you should you not write it. Sometimes it does a good job, othertimes not so much.

Algorithm 8 C++ implementation of reverse binary search tree insert

```cpp
void reverse_tree_insert(reverse_binary_search_tree_node* node, int data) {
   if(node) {
      if(!(data < node->data)) {
         if(node->left) {
            reverse_tree_insert(node->left, data);
         } else {
            node->left = new reverse_binary_search_tree_node;
            node->left->data = data;
            return;
         }
      } else {
         if(node->right) {
            reverse_tree_insert(node->right, data);
         } else {
            node->right = new reverse_binary_search_tree_node;
            node->right->data = data;
            return;
         }
      }
   }
   std::cerr << "Attempted to reverse empty tree!" << std::endl;
   exit(1);
}
```

Algorithm 9 Naive pseudocode for `operator=`

```
1: procedure OPERATOR=(x)
2:     //We are being assigned to, so clean up any resources we have allocated, asto not leak
3:     delete all child nodes
4:     close open files
5:     release control of network sockets / ports
6:     ...
7:     PRIVATE_ASSIGNMENT_LOGIC(x)
8:     return this object
```

Algorithm 10 Better pseudocode for `operator=`

```
1: procedure OPERATOR=(x)
2:     if This object is x then
3:         return this object
4:     else
5:         //We are being assigned to, so clean up any resources we have allocated, asto not leak
6:         delete all child nodes
7:         close open files
8:         release control of network sockets / ports
9:         ...
10:        PRIVATE_ASSIGNMENT_LOGIC(x)
11:        return this object
```

On the other hand, the constructor *is* a special function. It can *only* be called during object construction, and then only once per object. The *copy constructor*, is the constructor which is responsible for building an object from another object of the same type. If you don't write a copy constructor (and you don't explicitly tell `C++` *not* to generate one) then one will actually be written for you.

So what are the differences between these two methods?

- `operator=` acts on already constructed objects. Constructors are called only at birth.
- The copy constructor (and any other constructor) is only called once in the lifetime of your object (at birth from another object of the same type). `operator=` may be called any number of times, as your object gets reassigned.
- `operator=` can be defined to do just about whatever you want. It can return whatever you want, output whatever you want, delete whatever you want and so on. In fact, there is no rule that `operator=` even needs to actually do an assigment! Constructors, on the other hand, *have only one job.* They build an object. If you write a constructor which does not initialize your objects data members, they just won't get initialized.

15. *Q.* Use pointer arithmetic to write a function to reverse an array.

    A. Allrighty. The answer is in algorithm 11. Don't get confused by `std::swap` if you have not seen that before. It just swaps two variables. Saves me from writing tedious code.

Algorithm 11 C++ implementation of array reversal function

```
#include <utility>

void reverse_array(int* a, unsigned int length) {
   if(length) {
      int* left = a;
      int* right = a + length - 1;
      while ( left < right ) {
         std::swap(*left, *right);
         ++left;
         --right;
      }
   }
}
```

16. *Q.* Why can't we do binary search on a linked list?

    A. Because binary search only applys to containers which support random access. In a linked list, you only have access to the element next to the current one, so you can't hop forward (or backward) by more than one space at a time. Even if you wrote a "binary search" algorithm for a linked list, it would not have $O(\log n)$ execution time (in fact, it would be far worse than linear).

17. *Q.* Why is contains for a binary search tree faster than $O(n)$? Can binary tree contains be this fast?

    A. In the absolute worst case, a binary search tree is just a linked list, which has $O(n)$ lookup time. If the tree is better balanced than that (which is very likely in most cases), then you don't need to consider some of the nodes when you search, they just get eliminated from consideration at each iteration.

    Binary tree contains, on the other hand, has a harder time. In the most general case, you can't assume that the tree has any particular order (or even that the object you are searching on supports `operator<` which could even define an order in the first place). Thus, you woud need to search through each node in the tree and you would get (at best) $O(n)$ execution time.

18. *Q.* Suppose I am adding 2 big_numbers as follows.

```cpp
big_number alice(98);

big_number bobo(87);

alice += bobo;
```

In the code for operator+=,

```cpp
big_number& big_number::operator+=(const big_number& b)
```

which number, alice or bobo corresponds to b? Which number corresponds to *this?

A. Easy. *this is always on the left. So alice : *this as bobo : b.

19. *Q.* Tell me how a stack can be used to tell if a program has balance braces.

A. Sure. You just push onto the stack everytime your parser finds a '{', and pop from the stack everytime you encounter a '}'. I cooked up some quick code to demonstrate this. See algorithm 12.

20. *Q.* Trace out the tree_copy function for a particular binary tree. Which node is copied first? Last?

A. I plan to generate some nice graphics for this later (if I have time).

21. *Q.* Trace out the tree_clear function for a particular binary tree. Which node is copied first? Last?

A. I plan to generate some nice graphics for this later (if I have time).

22. *Q.* Be nauseatingly familiar with the copy command.

A. Yes mam! std::copy is one of the most generally useful standard library functions. I don't know what else to do for this but include a usage example. Copy just *iterates* over pointers and moves the data from one to another. See algorithm 13 for a usage example.

23. *Q.* Where is the smallest number in a binary search tree? How would you find it?

A. Go west until you find the fabled nullptr. Then you have arrived at the smallest value. But seriously, this can be a one liner (assuming the root node is not empty). See algorithm 14 for that answer. For a more easily understood example, see algorithm 15.

24. *Q.* When I compare 2 big_numbers, which digits should I compare first and why?

A. You can do this either way, but "little endian" is easier. That is, store the bytes from least to most significant. That way you don't need to worry about normalizing the lengths of your big_nums. If I have time I will make a nice graphic to explain this.

Algorithm 12 Program brace balance tester

```cpp
#include <string>
#include <stack>
#include <iostream>

const char OPEN = '{';
const char CLOSE = '}';
bool program_balanced(
      const std::string& program,
      char open = OPEN,
      char close = CLOSE
)
{
   // Make a stack to count our brace pairs.
   std::stack<char> x;
   for ( auto character : program ) {
      if(character == open) {
         // Stick something on the stack if you find an open brace.
         x.push(open);
      }
      if(character == close) {
         // If the stack is empty, we did not find a balanecd set
         // of braces.
         if (x.empty()) {
            return false;
         }
         // If the stack is not empty, pull an element off of it.
         x.pop();
      }
   }
   // If the stack is empty after all that, obviously the braces were
   // balanced.
   return x.empty();
}

int main(int argc, char *argv[]) {
   std::string myUnbalancedProgram = "{{ab{xyqzb{{}}}}";
   if (program_balanced(myUnbalancedProgram)) {
      std::cout << "Found a balanced program" << std::endl;
   } else {
      std::cout << "Found an unbalanced program" << std::endl;
   }
   std::string myBalancedProgram = "{ab{xyqzb{{}}}}";
   if (program_balanced(myBalancedProgram)) {
      std::cout << "Found a balanced program" << std::endl;
   } else {
      std::cout << "Found an unbalanced program" << std::endl;
   }
   return 0;
}
```

Algorithm 13 std::copy demo

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main(int argc, char *argv[]) {
   int a[5] = { 1, 2, 3, 4, 5 };
   std::vector<int> b = {{ 2, 4, 5, 3, 11 }};
   int c[5];

   int* a_start = a;
   int* a_end = a + 5;
   auto b_start = b.begin();
   auto b_end = b.end();
   int* c_start = c;
   int* c_end = c + 5;
   std::cout << "Copy a into c" << std::endl;
   std::copy(a_start, a_end, c_start);
   for(int* c_iter = c_start; c_iter != c_end; ++c_iter ) {
      std::cout << *c_iter << std::endl;
   }
   std::cout << "Copy vector b into array c" << std::endl;
   std::copy(b_start, b_end, c_start);
   for(int* c_iter = c_start; c_iter != c_end; ++c_iter ) {
      std::cout << *c_iter << std::endl;
   }
   std::cout << "Copy vector b into array c in reverse" << std::endl;
   auto b_reverse_start = b.rbegin();
   auto b_reverse_end = b.rend();
   std::copy(b_reverse_start, b_reverse_end, c_start);
   for(int* c_iter = c_start; c_iter != c_end; ++c_iter ) {
      std::cout << *c_iter << std::endl;
   }
   return 0;
}
```

Algorithm 14 Find the smallest value in a binary search tree (cute soluton)

```cpp
int tree_min(binary_search_tree_node* node) {
   return node->left ? tree_min(node->left) : node->data;
}
```

Algorithm 15 Find the smallest value in a binary search tree (less cute soluton)

```cpp
int tree_min(binary_search_tree_node* node) {
   if(node != nullptr) {
      if (node->left != nullptr) {
         return tree_min(node->left)
      }
      return node->data;
   } else {
      std::cerr << "Empty root node!" << std::endl;
      exit(1);
   }
}
```

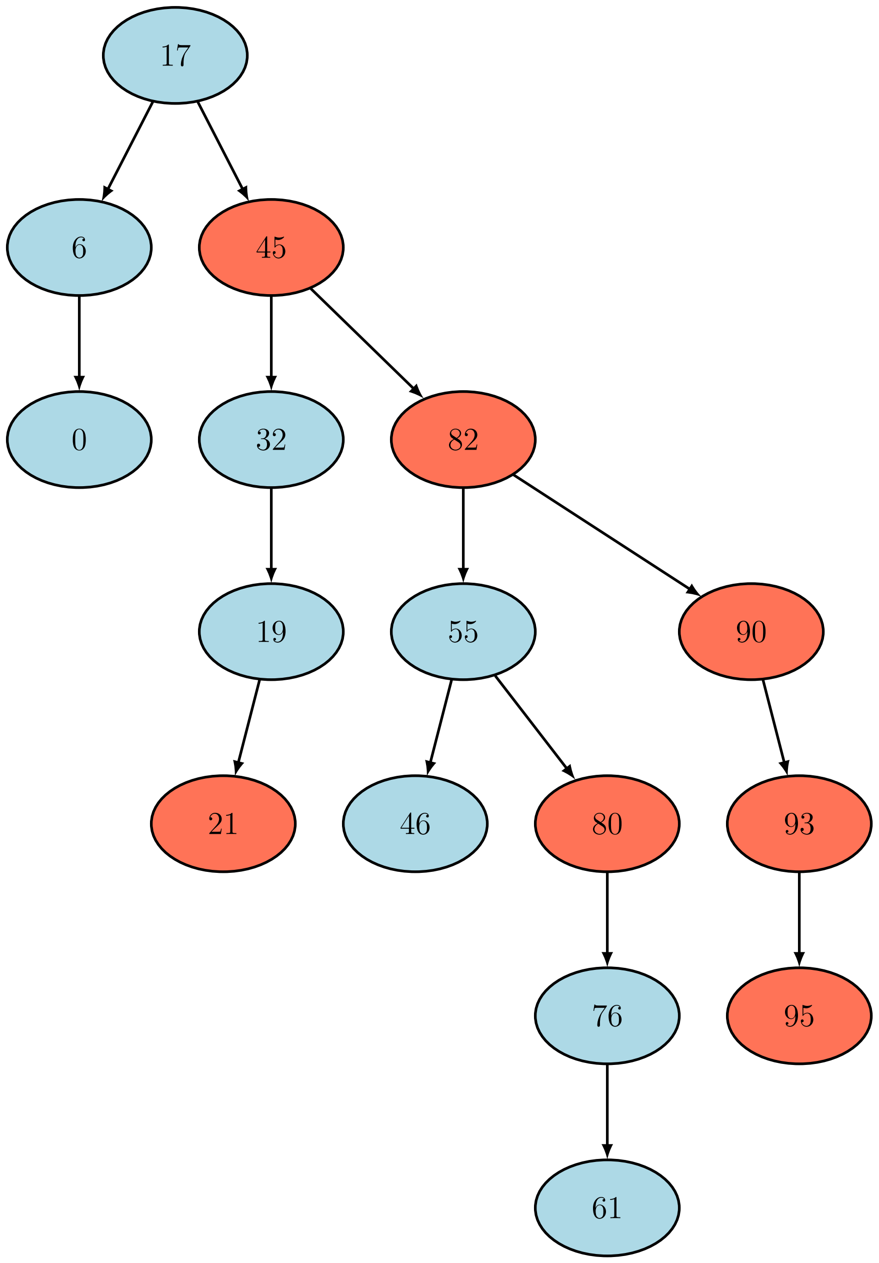

Figure 1: Randomly generated binary search tree. “Left” nodes shown in blue, “right” nodes shown in red. Note that this is an example of a poorly balanced binary search tree. Note how much more data is hanging off the right portion of the tree than the left side. This tree was built with random data, and so this is just “unlucky,” in that the link resistance for accessing nodes like 95 is quite high releative to the size of the tree

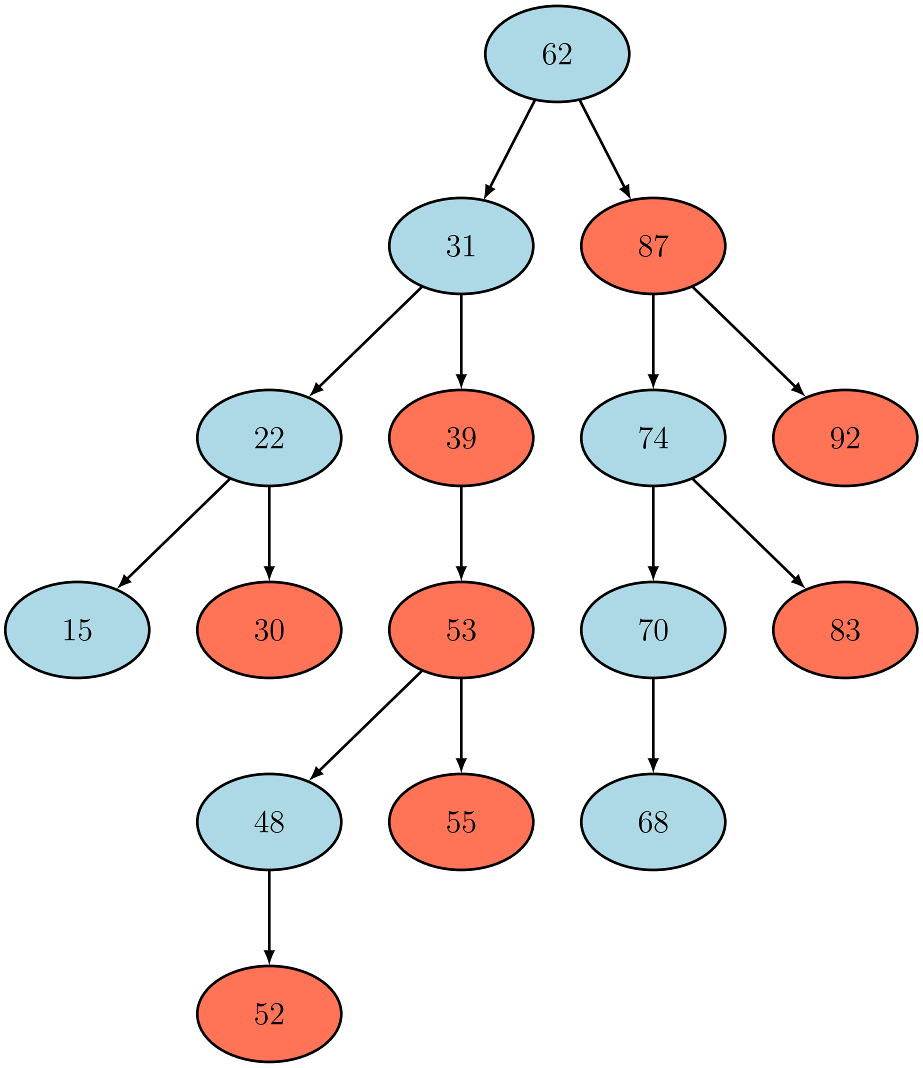

Figure 2: Again, blue (lower) nodes on the left, red (higher) nodes on the right.

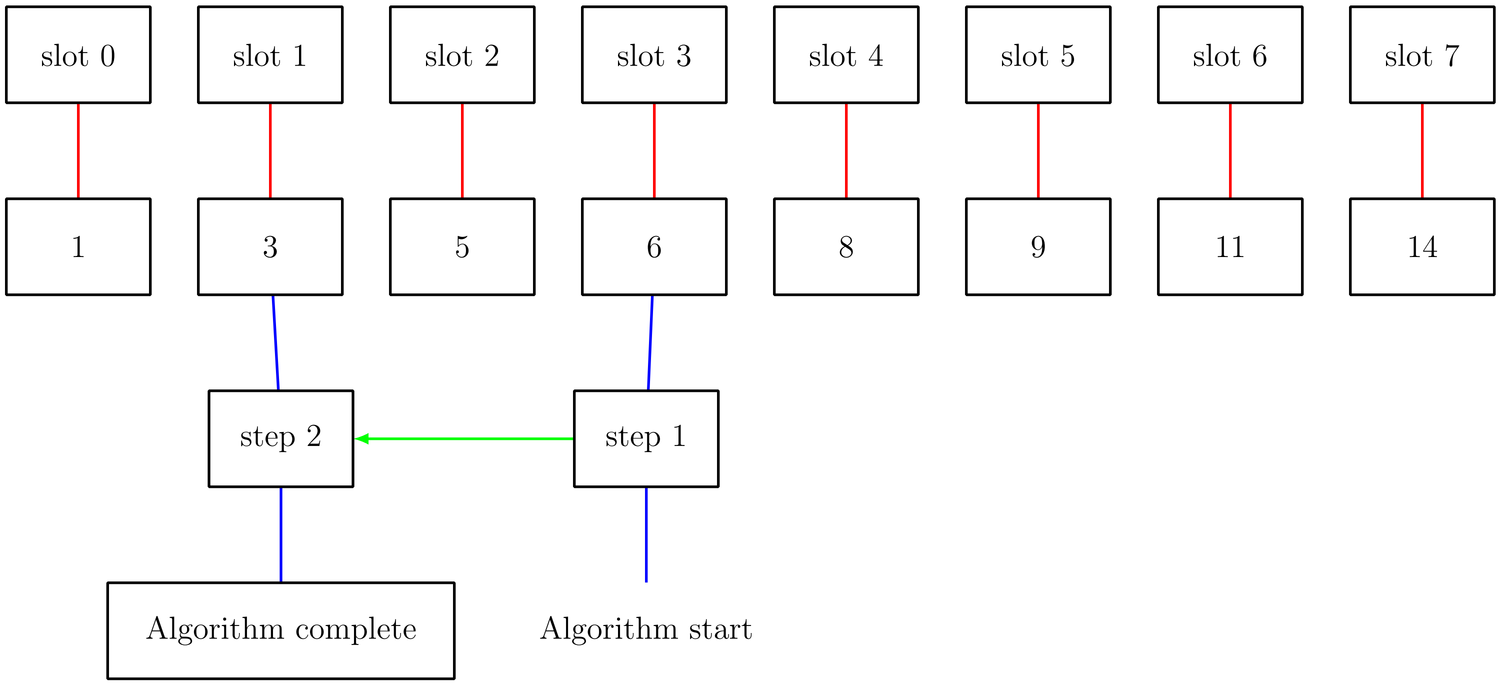

Figure 3: Binary search on a sorted array. Connections between locations in memory and memory contents are shown in red. Blue lines connect the location of the current memory pointer to the step number. Green arrows connect steps in the algorithm.

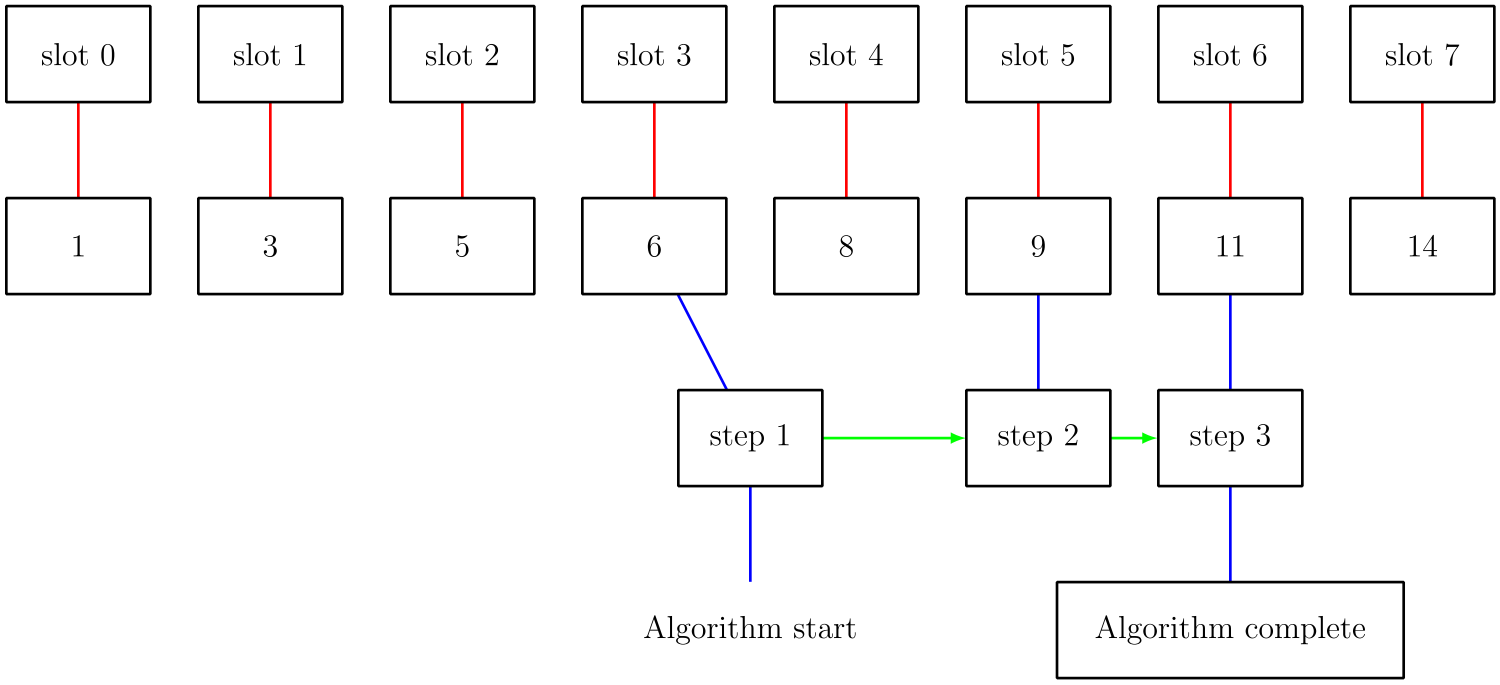

Figure 4: Binary search for a “missing” element. Colors and arrows are as in figure 3.