L7B: Binary Search and Symbolic Processing

SWS3012: Structure and Interpretation of Computer Programs

Martin Henz

July 11, 2023

Outline

- Binary Search and Binary Search Trees (2.3.3)
- Symbolic Processing (2.3.2)
- Preview: Programming with State

Outline

- Binary Search and Binary Search Trees (2.3.3)
- Symbolic Processing (2.3.2)
- Programming with State

Game: Guess the Secret Number

- I have a **secret number**, an integer in the range [**1, 100**]
- You make a **guess** and I tell you whether your guess is “**correct**”, “**too low**”, or “**too high**”
- Your goal is to **minimize** the **number of guesses**
- What is your **strategy**?
  - Guess the “**middle**” number
    - If “**too low**”, continue to find in the “**right**” half
    - If “**too high**”, continue to find in the “**left**” half
  - **Binary Search!!!**

Guess the Secret Number using Binary Search

[Show in Playground]

```
const N = 100;
...
function guess_secret_num(start, end) {
    if (start === end) {
        return start;
    } else {
        const guess = math_floor((start + end) / 2);
        const check = check_guess(guess);
        return check === "correct"
               ? guess
               : check === "too low"
               ? guess_secret_num(guess + 1, end)    // when "too low"
               : guess_secret_num(start, guess - 1); // when "too high"
    }
}
guess_secret_num(1, N);
```

Observations

- **Run time**
  - At each step (wrong guess), we **cut the search space in half**
  - If problem size is $N = 2^k$, we get to size 1 after k steps
  - Run time: $\boldsymbol{O(\log_2 N)}$
    - Equivalent to $\boldsymbol{O(\log N)}$

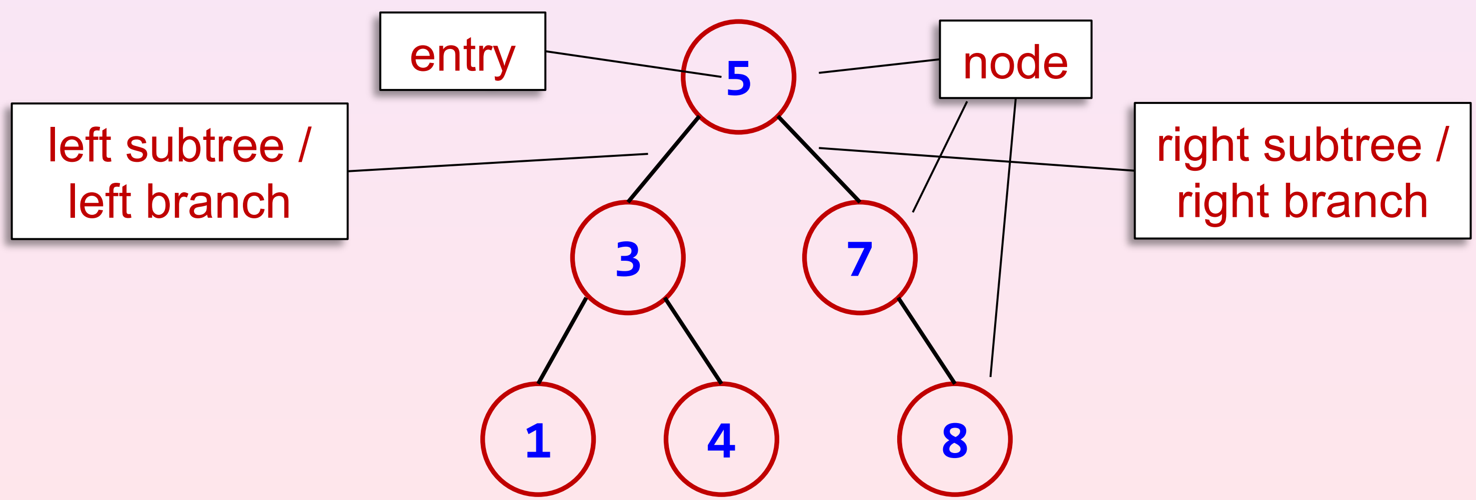
- **Why does this work?**
  - We need to assume that
    - **all numbers** to the “**left**” of the **secret number** are “**too low**”, and
    - **all numbers** to the “**right**” of the **secret number** are “**too high**”

Binary Search for Entries

- **The problem**
  - Check if an **entry** is included in a **collection of entries**
- **Property of entries**
  - A **total order** exists — two entries can be compared
    - They are “equal”, or one is either “smaller” or “larger” than the other
  - Examples: Numbers and Strings have such property
- **Efficiency**
  - Need to reach each “middle” entry in $O(1)$ time
  - May need a special data structure

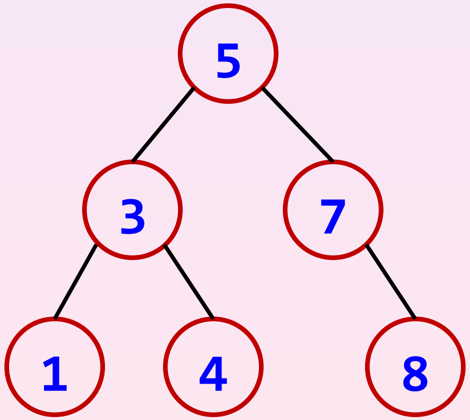
Binary Trees

- A ***binary tree*** is the **empty tree**, or it has
  - an ***entry*** (which is the data item)
  - a ***left branch*** or ***left subtree*** (which is a binary tree)
  - a ***right branch*** or ***right subtree*** (which is a binary tree)



Binary Search Trees (BST)

- A ***binary search tree (BST)*** is a **binary tree** where
  - all entries in the **left subtree** are **smaller than** the entry, and
  - all entries in the **right subtree** are **larger than** the entry



- A ***BST*** is an **abstraction** for **binary search**

Mission “Search and Rescue” (opens today)

- **Given:** A binary search tree (BST) of Strings
- **Problem 1:** Check if a given String occurs in BST
  - Use BST property to efficiently do so
- **Problem 2:** Add a new string to BST
  - Use BST property to efficiently do so
  - Preserve BST property

Outline

- Binary Search and Binary Search Trees (2.3.3)
- Symbolic Processing (2.3.2)
  - Symbolic Evaluation
  - Symbolic Differentiation
- Preview: Programming with State

Representing Functions: Directly

- Our first approach is to represent functions ***directly*** in Source using **function declarations** and **lambda expressions**

- **Example:**

```
function my_fun(x) {
    return x * x + 1;
}

function eval_numeric(f, x) {
    return f(x);
}

eval_numeric(my_fun, 7); // returns 50
```

Numerical Differentiation

```
// numerical differentiation; simplest method
function deriv_numeric(f) {
    const dx = 0.0001;
    return x => (f(x + dx) - f(x)) / dx;
}

// Example use:
const f = x => x * x + x + 4;
const f_prime = deriv_numeric(f);

f(3);        // returns 16
f_prime(3); // returns 7.000100000027487
```

Show in Playground

Symbolic Evaluation

- Now, we represent functions / expressions with **data structures**

- **Example expression:**

```
// my_exp represents x * x + x + 4
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", 4));
```

- **Symbolic evaluation:**

```
eval_symbolic(my_exp, "x", 3);
// should return 16
```

Definition of eval_symbolic

Show in Playground

```
function eval_symbolic(exp, name, val) {
  return is_number(exp)
         ? exp
         : is_variable(exp)
         ? (is_same_variable(exp, name) ? val : NaN)
         : is_sum(exp)
         ? eval_symbolic(addend(exp), name, val) +
           eval_symbolic(augend(exp), name, val)
         : is_product(exp)
         ? eval_symbolic(multiplier(exp), name, val) *
           eval_symbolic(multiplicand(exp), name, val)
         : error(exp, "unknown expression type");
}
```

Symbolic Differentiation

- **Symbolic representation:**

```
// my_exp represents x * x + x + 4
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", 4));
```

$$f(x) = x^2 + x + 4$$
$$f'(x) = 2x + 1$$

- **Symbolic differentiation:**

```
deriv_symbolic(my_exp, "x");
// should return make_sum(make_product("x", 2), 1)

eval_symbolic(deriv_symbolic(my_exp, "x"), "x", 3);
// should return 7
```

Symbolic Differentiation

- Rules of differentiation

$$\frac{dc}{dx} = 0 \text{ for } c \text{ a constant or a variable different from } x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

Definition of deriv_symbolic

```
function deriv_symbolic(exp, x) {
  return is_number(exp)
         ? 0
         : is_variable(exp)
         ? (is_same_variable(exp, x) ? 1 : 0)
         : is_sum(exp)
         ? make_sum(deriv_symbolic(addend(exp), x),
                    deriv_symbolic(augend(exp), x))
         : is_product(exp)
         ? make_sum(make_product(multiplier(exp),
                        deriv_symbolic(multiplicand(exp), x)),
                    make_product(
                        deriv_symbolic(multiplier(exp), x),
                        multiplicand(exp)))
          : error(exp, "unknown expression type");
}
```

[Show in Playground]

Implementation of Symbolic Representation

- **Constructors:**

```
function make_sum(a1, a2) {
    return list("+", a1, a2);
}

function make_product(m1, m2) {
    return list("*", m1, m2);
}
```

Implementation of Symbolic Representation

- **Accessors / Selectors:**

```
function addend(s) {
    return head(tail(s));
}
function augend(s) {
    return head(tail(tail(s)));
}
function multiplier(s) {
    return head(tail(s));
}
function multiplicand(s) {
    return head(tail(tail(s)));
}
```

Implementation of Symbolic Representation

- **Predicates:**

```
function is_variable(x) {
    return is_string(x);
}
function is_same_variable(v1, v2) {
    return is_variable(v1) && is_variable(v2) && v1 === v2;
}
function is_sum(x) {
    return is_pair(x) && head(x) === "+";
}
function is_product(x) {
    return is_pair(x) && head(x) === "*";
}
```

Revisiting Example

- **Symbolic representation:**

```
// my_exp represents x * x + x + 4
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", 4));
```

$$f(x) = x^2 + x + 4$$
$$f'(x) = 2x + 1$$

- **Symbolic differentiation:**

```
deriv_symbolic(my_exp, "x");
// should return make_sum(make_product("x", 2), 1)

// but instead returns a complicated expression
// equivalent to  x * 1 + 1 * x + 1 + 0
```

make_sum with Expression Simplification

```
function make_sum(a1, a2) {
    return number_equal(a1, 0)
           ? a2
           : number_equal(a2, 0)
           ? a1
           : is_number(a1) && is_number(a2)
           ? a1 + a2
           : list("+", a1, a2);
}
```

Show in Playground

make_product with Expression Simplification

```
function make_product(m1, m2) {
    return number_equal(m1, 0) ||
                number_equal(m2, 0))
              ? 0
              : number_equal(m1, 1)
              ? m2
              : number_equal(m2, 1)
              ? m1
              : is_number(m1) && is_number(m2)
              ? m1 * m2
              : list("*", m1, m2);
}
```

Show in Playground

Revisiting Example, Again

- **Symbolic representation:**

```
// my_exp represents x * x + x + 4
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", 4));
```

$$f(x) = x^2 + x + 4$$
$$f'(x) = 2x + 1$$

- **Symbolic differentiation:**

```
deriv_symbolic(my_exp, "x");
// should return make_sum(make_product("x", 2), 1)

// now returns an expression
// equivalent to  x + x + 1
```

Outline

- Binary Search and Binary Search Trees (2.3.3)
- Sorting
- Symbolic Processing (2.3.2)
- Preview: Programming with State

Functional Programming

- **Example:**
  - `factorial(5)` always gives 120
    - No matter how many times you call it, or when you call it

- Compare with a **bank account**:
  - Suppose it starts with $100
  - Function `withdraw` returns the balance if there is enough $, otherwise also displays error message

```
withdraw(40); → 60
withdraw(40); → 20
withdraw(40); → 20 "Insufficient funds"
withdraw(15); → 5
```

State

- Identical calls to `withdraw` produce different results
- Bank account has "**memory**"
  - It remembers something about the past
  - It has ***state***
- Functional programming does not allow our programs to have state
  - We need to use ***assignment***

Simple Bank Account — Using Assignment

```
function make_account(initial_balance) {
    let balance = initial_balance;

    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            display("Insufficient funds");
            return balance;
        }
    }
    return withdraw;
}

const W1 = make_account(100);
W1(40); → 60
W1(40); → 20
W1(40); → 20 "Insufficient funds"
```

[Show in Playground]

Simple Bank Account — Functional Approach

Show in Playground

```
function fn_make_account(initial_balance) {
    const balance = initial_balance;

    function withdraw(amount) {
        if (balance >= amount) {
            return fn_make_account(balance - amount);
        } else {
            display("Insufficient funds");
            return fn_make_account(balance);
        }
    }
    return withdraw;
}

const W1 = fn_make_account(100);
const W2 = W1(40); ➔ fn_make_account(60)
const W3 = W2(40); ➔ fn_make_account(20)
const W4 = W3(40); ➔ fn_make_account(20) "Insufficient funds"
```

Variable Declaration Statement

```
let name = expression;
```

- Declares a ***variable*** *`name`* in the current scope and initializes its value to the value of *`expression`*
- From now on, *`name`* will evaluate to the value of *`expression`*
- Note that from [Source §3]() onwards, **function parameters** are **variables**

Assignment Statement

```
name = expression;
```

- *`name`* is a **variable**; not evaluated
- *`expression`* is evaluated, then its value is ***assigned*** to the variable *`name`*
- From now on, *`name`* will evaluate to the value of *`expression`*

Example

```
let balance = 100;

balance; → 100

balance = balance - 20;

balance; → 80

balance = balance - 20;

balance; → 60
```

Multiple Accounts

```
const W1 = make_account(100);
const W2 = make_account(100);

W1(50); → 50
W2(70); → 30
W2(40); → 30 "Insufficient funds"
W1(40); → 10
```

- `W1` and `W2` are completely **independent**
  - Each has its own state variable `balance`
  - Withdrawals from one do not affect the other

Assignment: Pros

- Assignment allows us to create objects with ***state***
- State allows objects to behave differently over time

Assignment: Cons

- Harder to reason about programs
  - Harder to debug
  - Harder to verify correctness

- **Substitution model of evaluation breaks down!**
  - Not powerful enough to explain state
  - Need a more sophisticated model — ***Environment Model***

Substitution Model Breaks Down

- Consider

```
function make_simplified_withdraw(balance) {
    return amount => {
        balance = balance - amount;
        return balance;
    }
}
```

- Use **substitution model** to evaluate

```
(make_simplified_withdraw(25))(20);
```

Substitution Model Breaks Down

- Use substitution model to evaluate

```
(make_simplified_withdraw(25))(20);
```

⇩

```
(amount => { balance = 25 - amount; return 25; })(20);
```

⇩

```
balance = 25 - 20; return 25; // WRONG!
```

- It returns `25`, which is wrong!

Why Substitution Model Breaks Down?

- Substitution model considers a constant/variable as **just a name for a value**
  - Its value will not change
  - Therefore, one can be substituted for the other

- But **assignment** considers a variable as a **“container” holding a value**
  - The contents of the container may be **changed over time**
  - The container is maintained in a structure called an ***environment***

Summary

- **Binary search** and **binary search trees** (**BST**)
- **Symbolic processing**
  - Evaluation & differentiation
- **State**
  - **Increases expressive power**
  - Substitution model **breaks down**