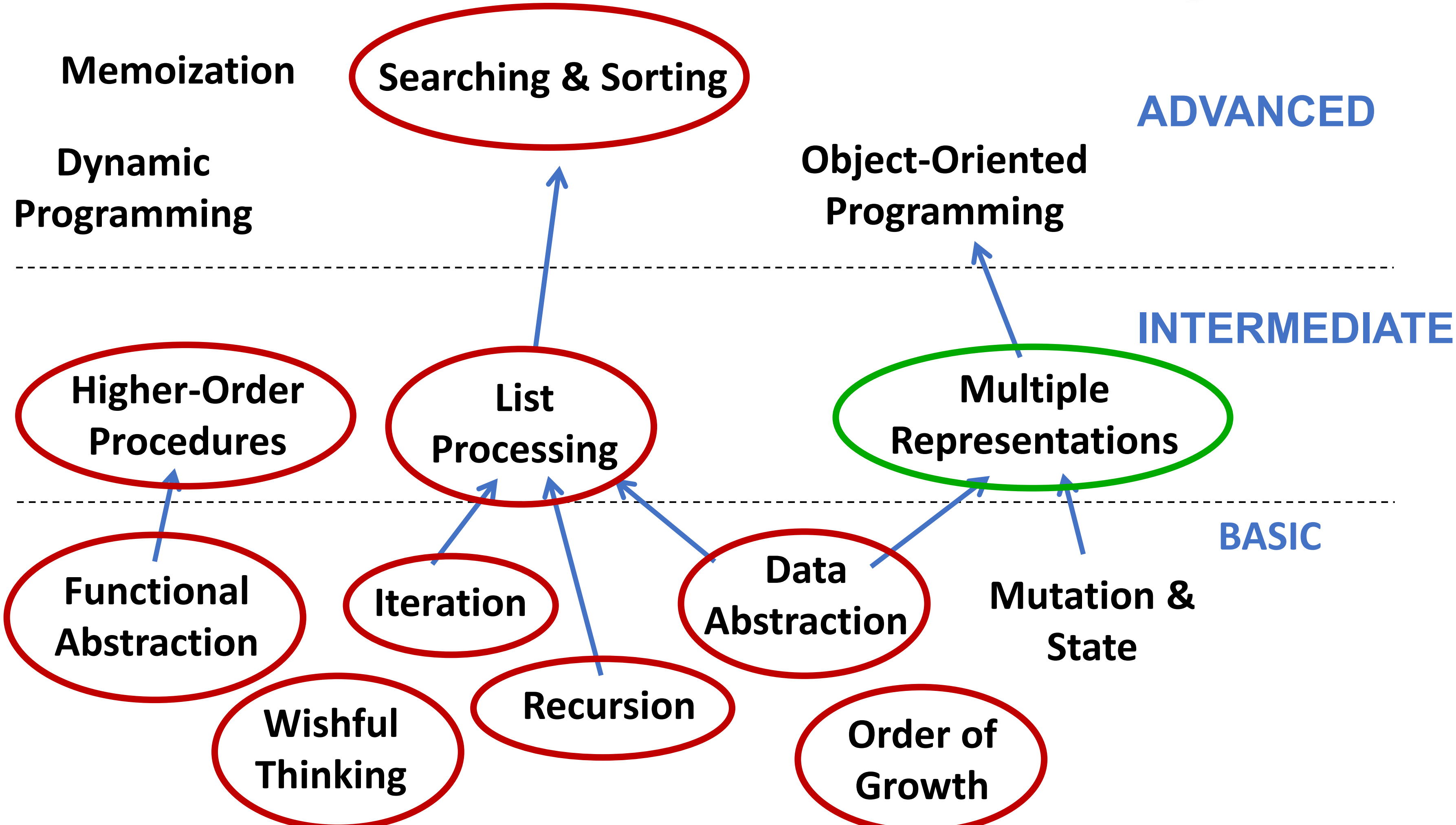
CS1010S Programming Methodology

Lecture 8

Implementing Data Structures

17 Oct 2018

CS1010S Road Map



Fundamental <u>concepts</u> of computer programming

Today's Agenda

- The Game of Nim
  - Simple data structures
- Designing Data Structures
- Multiple Representations

The Game of Nim

- Two players
- Game board consists of piles of coins
- Players take turns removing one or more coins from a single pile each time.
- Player who takes the last coin wins

Let’s Play!

Let’s start with a simple game of <u>two</u> piles

How to Write This Game?

1. Keep track of the game state
   - e.g. how many coins remains in each pile
2. Specify game rules
   - e.g. human/computer take turn to play
3. Figure out strategy (for computer to play)
4. Glue them all together

Implementing Game State

- What game state do we need to keep track of?
  - The number of coins in each of the two piles!

```
# n is the number of coins in the 1st pile
# m is the number of coins in the 2nd pile
def make_game_state(n, m):
    return (n, m) # game state
```

```
def size_of_pile(game_state, pile):
    return game_state[pile-1] # no of coins
```

Writing/Playing the Game

```
def remove_coins_from_pile(game_state, num_coins,
                           pile_number):

    pile1_coins = size_of_pile(game_state, 1)
    pile2_coins = size_of_pile(game_state, 2)

    if pile_number == 1:
        pile1_coins = pile1_coins - num_coins
    else:
        pile2_coins = pile2_coins - num_coins

    return make_game_state(pile1_coins, pile2_coins)
```

Writing/Playing the Game

- To start a new game, e.g.:

```
>>> play( make_game_state(5, 8), "human" )
```

```
def play(game_state, player): # game engine
    display_game_state(game_state)
    if is_game_over(game_state): # base case
        announce_winner(player)
    elif player == "human":
        play(human_move(game_state), "computer")
    else: # player == "computer":
        play(computer_move(game_state), "human")
```

```python
def display_game_state(game_state):
    print("pile 1:", size_of_pile(game_state, 1))
    print("pile 2:", size_of_pile(game_state, 2))
```

```python
def is_game_over(game_state):
    return size_of_pile(game_state, 1) == 0 and \
           size_of_pile(game_state, 2) == 0
```

```python
def announce_winner(player): # next player
    if player == "human":
        print("Human lose. Better luck next time")
    else:
        print("Human win. Congratulations")
```

```python
def human_move(game_state):
    p = input("Which pile will you remove from? ")
    n = input("How many coins to remove? ")
    print("Human removes", n, "coins from pile", p)
    return remove_coins_from_pile(game_state,
                                  int(n),
                                  int(p))
```

- Computer tries to remove 1 coin from pile 1 each time – is this a good strategy?

```
def computer_move(game_state):
    if size_of_pile(game_state, 1) > 0:
        p = 1
    else:
        p = 2
    print("Computer removes 1 coin from pile", p)
    return remove_coins_from_pile(game_state,
                                  1,
                                  p)
```

Game State: Another Implementation

- Previously we implemented game state as a tuple.

```
def make_game_state(n, m):
    return (n, m) # game state
```

- How about the following implementation, good?

```
def make_game_state(n, m):
    return n*10 + m
def size_of_pile(game_state, pile_number):
    if pile_number == 1:
        return game_state // 10
    else:
        return game_state % 10
```

Improving Nim

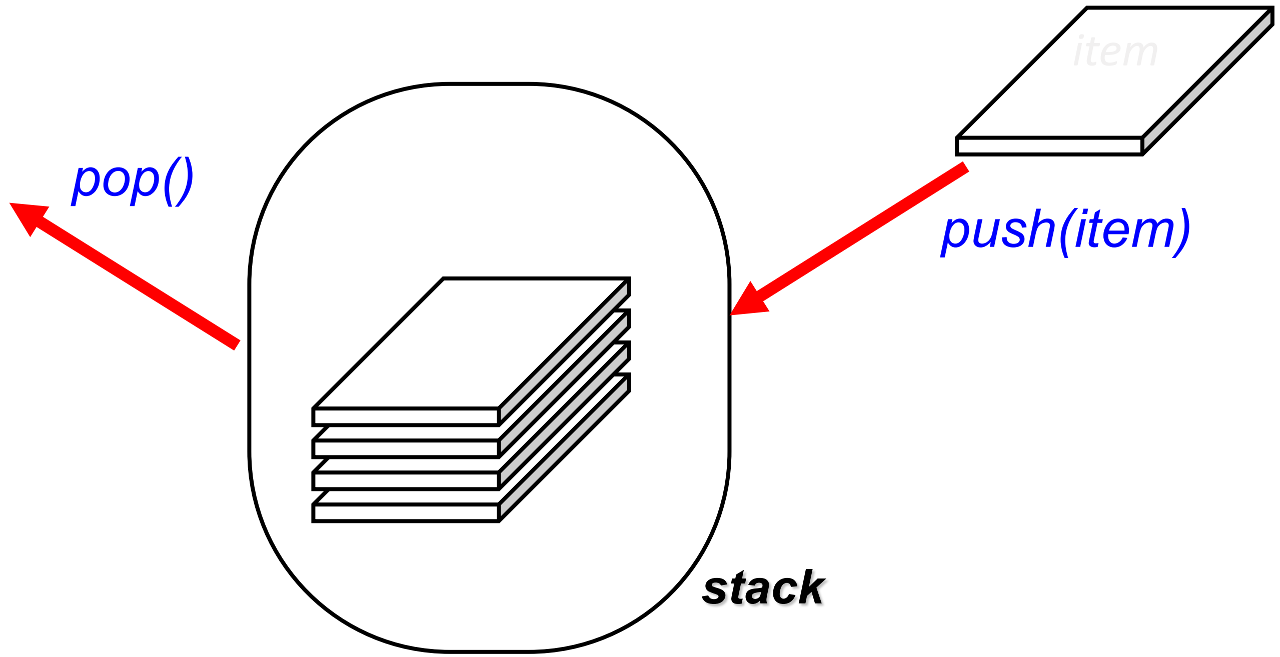
- Lets modify our Nim game by allowing "undo".
  - Only Human player can undo, not Computer.
  - Removes effect of the most recent move
    - i.e. undo most recent computer and human move
    - Human's turn again after undo
  - Human enters "0" to indicate undo

Key Insight

- We need a data structure to remember the history of game states.
- Before each human move, add the current game state to the history.
- When undoing,
  - Remove most recent game state from history
  - Make this the current game state

Data Structure: Stack

- A Stack is a collection of data that is accessed in a last-in-first-out (LIFO) manner.
  - Items are removed in the reverse order in which they were added.



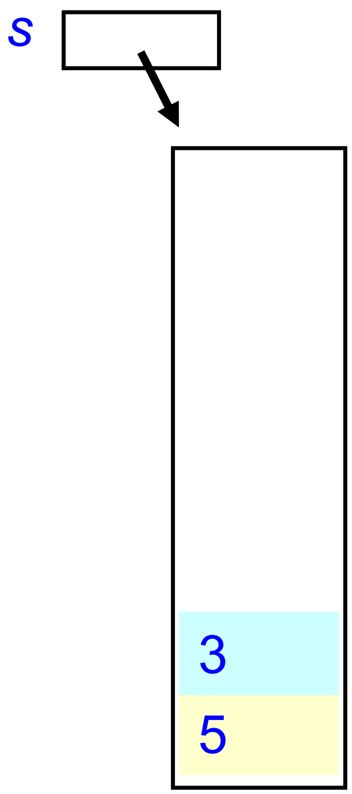
Common Stack Operations

Methods	Descriptions
`make_stack()`	returns a new, empty stack
`push(s, item)`	adds `item` to stack `s`
`pop(s)`	removes the most recently added item from stack `s`, and returns it
`is_empty(s)`	returns `True` if `s` is empty, `False` otherwise

Implement a stack as homework.

Example

```
>>> s = make_stack()
>>> pop(s)
None
>>> push(s, 5)
>>> push(s, 3)
>>> pop(s)
3
>>> pop(s)
5
>>> is_empty(s)
True
```



Changes to Nim

```
game_stack = make_stack() # empty stack to begin with

def human_move(game_state):
    p = input("Which pile will you remove from? ")
    if int(p) == 0:
        return handle_undo(game_state)

    n = input("How many coins to remove? ")
    print("Human removes", n, "coins from pile", p)
    push(game_stack, game_state)
    return remove_coins_from_pile(game_state,
                                  int(n),
                                  int(p))
```

Changes to Nim

```
def handle_undo(game_state):

    previous_state = pop(game_stack)

    if previous_state: # not None
        display_game_state(previous_state)
        return human_move(previous_state)
    else:
        print("No more previous move")
        return human_move(game_state)
```

Today's Agenda

- The Game of Nim
  - Simple data structures
- Designing Data Structures
- Multiple Representations

Data Structures: Design Principles

- When designing a data structure, need to spell out:
  - Specification (contract)
    - What does it do?
    - Allow users know how to use it
  - Implementation
    - How is it realized?
    - Users do not need to know this.
    - Choice of implementation

Data Structures: Specification

- Conceptual description of data structure
  - State assumptions, contracts, guarantees, etc.

  - Specify operations for users to use, such as:
    - Constructors
    - Selectors (Accessors)
    - Predicates
    - Printers

Example: Lists

- Specification:
  - A list is an indexed collection of objects.

  - Operations:
    - Constructors: `list(), [ ]`
    - Selector: `[ ]`
    - Predicates: `type, in, ==, is`
    - Printer: `print`

Another Example: Set

- A set is an unordered collection of objects (numbers, in our example) without duplicates.
  - `{3, 2, 1}` and `{1, 2, 3}` are the same
  - `{3, 3, 2, 1}` is an invalid set

- We will implement our own Set data structure.

Another Example: Set

- Set operations:
  - Constructors: `make_set(), adjoin_set(), union_set(), intersection_set()`
  - Selectors:
  - Predicates: `is_element_of_set(), is_empty_set()`
  - Printer `print_set()`

Set: Contract

- Some properties that hold:
  - For any set S and any object x, adjoining x to set S produces a set S that contains x.
  - The elements of $(S \cup T)$ are the elements that are in S or in T.
  - etc.

Set: Implementation

- Multiple representations:
  - Usually there are choices, e.g. lists, trees.
  - Different choices affect time/space complexity.
  - There may be certain constraints on the representation. They should explicitly stated.

- Implement constructors, selectors, predicates, printers, using your selected representation

Set Implementation #1

- Representation: unordered list
  - A set is represented by a list of objects.
  - Empty set is represented by empty list.
  - Must take care to avoid duplicates

```
# Constructor
def make_set():
    '''returns a new, empty set'''
    return []
```

Set Implementation #1

```
# Predicates
def is_empty_set(s):
    return not s
    # or: return len(s) == 0

def is_element_of_set(x, s):
    for e in s:
        if e == x:
            return True
    return False
```

Time complexity: $O(n)$, where n is the size of set

Set Implementation #1

```
# Constructors
def adjoin_set(x, s): # add x to s if x not in s
    if not is_element_of_set(x, s):
        s.append(x)
    return s

def intersection_set(s1, s2):
    pass
    # return a new set which is the intersection
    # of s1 and s2
    # write this and the rest functions yourself
```

Set Implementation #2

- Representation: ordered list
  - Empty set is represented by empty list.
  - Must take care to avoid duplicates
  - But now objects are sorted.
    - specs does not require sorting (so users are unaware of this)
    - only possible if the objects are comparable (e.g. numbers, strings)

Q: What’s the advantage of making objects sorted?

Set Implementation #2

```
# Constructor
def make_set():
    return [] # same as in implementation #1

# Predicate
def is_empty_set(s):
    return not s # as before

# Constructor
def adjoin_set(x, s):
    pass
    # write it yourself: add x to s
    # make sure s is still sorted after insertion
```

Set Implementation #2

```
# Predicate
def is_element_of_set(x, s):
    for e in s:
        if e == x:
            return True
        elif x < e: # remaining data are even bigger
            return False # early exit
    return False
```

Still $O(n)$ time complexity but actually slightly faster than implementation #1.

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

Set 1: {1 3 4 8}
Set 2: {1 4 5 6 8 9}

result: {}

Set Implementation #2

- Set intersection:
  - How to find intersection of two <u>sorted</u> sets?
  - Idea: similar to that of the merge() function in merge sort

```
          ⇩
Set 1: {1   3   4   8}
Set 2: {1   4   5   6   8   9}
          ⇧
```

result: {1}

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

```
Set 1: {1   3   4   8}
Set 2: {1   4   5   6   8   9}

result: {1}
```

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

```
Set 1:  {1   3   4   8}
Set 2:  {1   4   5   6   8   9}

result: {1  4}
```

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

Set 1: {1 3 4 8}
Set 2: {1 4 5 6 8 9}

result: {1 4}

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

Set 1: {1 3 4 8}

Set 2: {1 4 5 6 8 9}

result: {1 4}

Set Implementation #2

- Set intersection:
  - How to find intersection of two sorted sets?
  - Idea: similar to that of the merge() function in merge sort

```
Set 1:  {1   3   4   8}
Set 2:  {1   4   5   6   8   9}

result:  {1  4  8}
```

Set Implementation #2

```
def intersection_set(s1, s2): # return common elements
    result = []
    i, j = 0, 0
    while i < len(s1) and j < len(s2):
        if s1[i] == s2[j]:
            result.append(s1[i])
            i = i + 1
            j = j + 1
        elif s1[i] < s2[j]:
            i = i + 1
        else:
            j = j + 1
    return result
```

Time complexity: $O(n)$; much faster than that of unsorted sets.

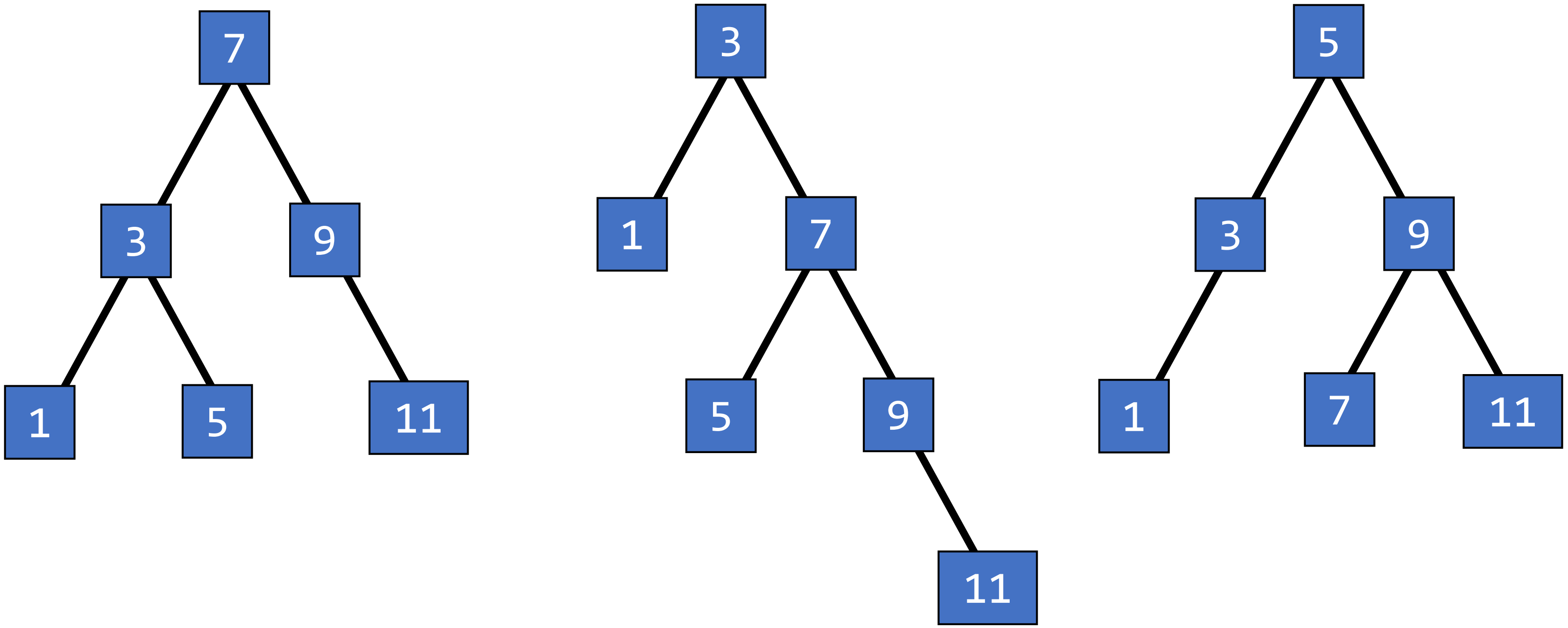
Set Implementation #3

- Representation: binary tree
  - Empty set is represented by empty tree.
  - Must take care to avoid duplicates
  - Objects are sorted.

Set Implementation #3

- Representation: binary tree
  - Each node stores 1 object.
  - Two branches
  - Left subtree contains objects smaller than this node.
  - Right subtree contains objects greater than this node.

Set Implementation #3



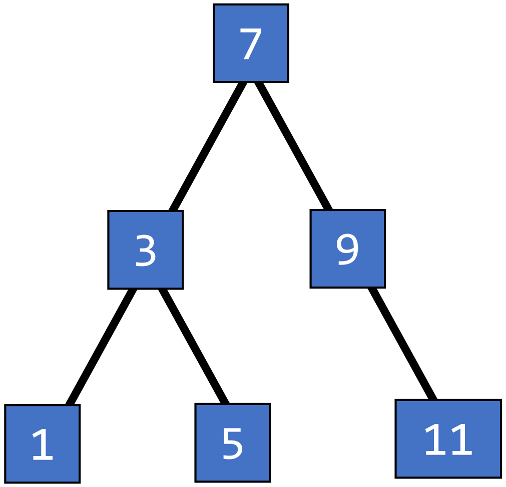
Three trees representing the set {1, 3, 5, 7, 9, 11}

Set Implementation #3

```
# Tree operators
def make_tree(entry, left, right):
    return (entry, left, right)

def entry(tree):
    return tree[0]

def left_branch(tree):
    return tree[1]

def right_branch(tree):
    return tree[2]
```

Set Implementation #3

- Search for 5 in the following tree representation of set {1, 3, 5, 7, 9, 11}?
  - How about search for 8?



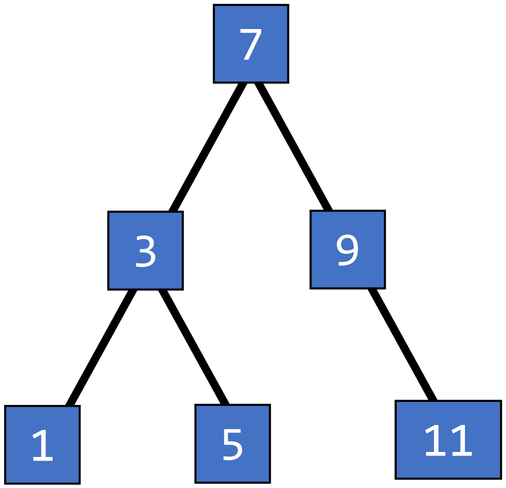
Set Implementation #3

```
# Predicate
def is_element_of_set(x, s):
    if is_empty_set(s):
        return False
    elif x == entry(s):
        return True
    elif x < entry(s):
        return is_element_of_set(x, left_branch(s))
    else: # x > entry(s)
        return is_element_of_set(x, right_branch(s))
```

Time complexity: $O(\log n)$

Set Implementation #3

- How to add 8 into the following tree representation of set {1, 3, 5, 7, 9, 11}?
  - How about 1 or 4?



Set Implementation #3

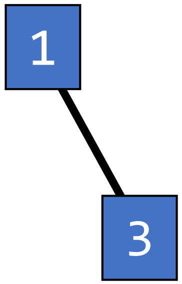
Time complexity: $O(\log n)$

```
def adjoin_set(x, s):
    if is_empty_set(s):
        return make_tree(x, [], [])
    elif x == entry(s): # duplicate
        return s
    elif x < entry(s):
        return make_tree(entry(s),
                         adjoin_set(x, left_branch(s)),
                         right_branch(s))
    else: # x > entry(s)
        return make_tree(entry(s),
                         left_branch(s),
                         adjoin_set(x, right_branch(s)))
```

Balancing Trees

- Insertion is $O(\log n)$ assuming that tree is balanced.
- But tree can become unbalanced after several operations.
  - Unbalanced trees break the $O(\log n)$ complexity.
- One solution: define a function to restore balance. Call it every so often.

Example: adding, in sequence, 5, 7, 9, 11 into the following tree representation of set {1, 3}



Today's Agenda

- The Game of Nim
  - Simple data structures

- Designing Data Structures

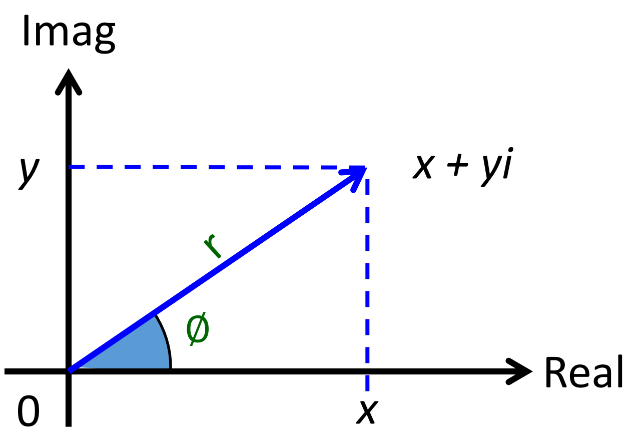
- Multiple Representations

Multiple Representations

- You have seen that for compound data, multiple representations are possible.
  - For example, set as
    - Unordered list, w/o duplicates
    - Ordered list, w/o duplicates
    - Binary tree, w/o duplicates

Complex-Arithmetic Package

- Complex number
  - rectangular form
    - comprises a real part x and an imaginary part y, and is written as $x + yi$.
  - polar form
    - written as $re^{-i\emptyset}$.
    - r: magnitude
    - $\emptyset$: angle



Abstraction barrier

Programs that use complex numbers

(use given functions)

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package

Rectangular representation	Polar representation

Rectangular Rep.

```
import math
def make_from_real_imag(x, y):
    return (x, y) # internal representation
def real_part(z): # selector: return real part
    return z[0]
def imag_part(z): # selector: return imaginary part
    return z[1]
def magnitude(z):
    return math.hypot(real_part(z), imag_part(z))
def angle(z):
    return math.atan( imag_part(z)/real_part(z) )
def make_from_mag_ang(r, a):
    return (r * math.cos(a), r * math.sin(a))
```

$(\sqrt{x^2 + y^2}$

Polar Rep.

```python
import math
def make_from_mag_ang(r, a):
    return (r, a) # internal representation
def magnitude(z): # selector
    return z[0]
def angle(z): # selector
    return z[1]
def real_part(z):
    return magnitude(z) * math.cos(angle(z))
def imag_part(z):
    return magnitude(z) * math.sin(angle(z))
def make_from_real_imag(x, y):
    return (math.hypot(x, y), math.atan(y/x))
```

Complex Number Operations

```
def add_complex(z1, z2):
    return make_from_real_imag(real_part(z1) + real_part(z2),
                               imag_part(z1) + imag_part(z2))

def mul_complex(z1, z2):
    return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                             angle(z1) + angle(z2))

def print_complex(z): # print x+yi
    print(str(real_part(z)) + '+' + str(imag_part(z)) + 'i')

# other functions skipped
```

User Code in Action

```
>>> from complex_rectanglar_rep import *
>>> a = make_from_real_imag(1, 2)
>>> b = make_from_real_imag(1, 1)
>>> print_complex(add_complex(a, b))
2+3i
```

Using the functions from rectangular rep.

```
>>> from complex_polar_rep import *
>>> a = make_from_real_imag(1, 2)
>>> b = make_from_real_imag(1, 1)
>>> print_complex(add_complex(a, b))
2.0000000000000004+3.0i
```

Using the functions from polar rep.

Observations

- Different representations might have slightly different accuracy because of internal representation.

- Power of data abstraction
  - the same code (e.g. `add_complex`) even though the representation is completely different

Multiple Representations

- Each representation has its pros/cons:
  - Typically, some operations are more efficient, while others are less efficient
  - “Best” representation may depend on how the compound data is used

Multiple Representations

- Typically in large software projects, multiple representations co-exist.
  - Because large projects have long lifetime, and project requirements change over time.
  - Because no single representation is suitable for every purpose.
  - Because programmers work independently and develop their own representations for the same thing.
  - etc.

Summary

- Data structure design principles.
  - Specification
  - Implementation
- Abstraction Barriers allow for multiple implementations
- Choice of implementation affects performance!