• Know how to prove the correctness of an algorithm
• How much time does it take as the size of the input grows
• Know how to analyze the efficiency of an algorithm
• Object oriented programming
• Heuristic algorithms

Learning Objectives:

Starts June 25th

Monday & Wednesday 4:00pm to 5:30pm

OVERVIEW

ROOTED TREES

LINKED LISTS

CLASSES, OBJECTS, AND POINTERS

DATA STRUCTURES INTRO

What is the Course About?
Course website: cslu.ohsu.edu/~heeman/cs532

Grading
- 25% Final
- 25% Midterm
- 50% Homework

Logistics

Background Needed

A course in discrete math is also useful.

- At a minimum, expected to have 1st year undergraduate CS major courses
- Anyone admitted to the program
- Intended to be taken earlier in your education
Critique

• Answer key will be emailed to you when you submit your answers.
  - You will have until Sunday at 11:59pm to submit a critique.
  - Explain what you did wrong, and why you made that mistake.
  - Explain where you did wrong, and why you made that mistake.
  - Worth up to half the marks that you lost.
  + Really good explanations might even get you more marks back.
  + You will need to show that you reviewed and understood the answer key and
  understood whether your answer was correct.

Homework

• Homework is due Friday at 11:59pm. If needed, you can email me with a request for an extension.
• Homework must be submitted through Sakei.
• Homework will be automatically graded.
• see sample homework for how to format this.
• We will test your code in several cases.
• Homework should be typeset.
• We will supply you with a code that will run your code on several examples.
• Turn in code as well.
• Can include pictures of hand-drawn solutions taken with cellphone.
• Single pdf with your answers on it.
• If you have problems, email me heemanp@ohsu.edu.

If you have problems, email me heemanp@ohsu.edu.
C versus Python

- Pointers are built into the language
- Does not have a built-in dictionary datatype
- Few people know C, so would spend a lot of time teaching this

Instead using Python
- Use classes/objects to get pointers
- Don’t use dictionaries or advanced packages

+ Not in textbook
+ Get a solid foundation in object-oriented programming
+ Get more in-depth experience with a programming language
+ Use classes/objects to get pointers
+ Document how you collaborated
+ You can do the homeworks with your colleagues

Academic Integrity

• You can do the homeworks with your colleagues
• Document how you collaborated
• Worked on entire question together, versus claimed what the
  question meant
• Nothing in, nothing out, wait at least an hour between
• After you have come up with a solution, redo it on your own

• You can do the homeworks with your colleagues
Dynamic Sets

- From: III Data Structures of textbook
- Set: collection of elements (like integers)
- Dynamic sets:
  - Can grow, shrink, or otherwise change over time
  - Can be manipulated by algorithms
  - Different algorithms might require different operations

Dynamic Sets

Review

- Rooked Trees
- Linked Lists
- Classes, Objects, and Pointers
- Data Structures Intro
- What is the Course About?
Chapter 10: Elementary Data Structures

**Arrays**
- Elements are numbered starting at 0 (or 1)
- Can access any element: A[0], A[5], ...
- Can read/write any element
- Elements are numbered starting at 0 (or 1)

**Stacks and Queues**
- For some tasks, you want to restrict how elements are accessed
- Stacks: Last-in, first-out
  - When it is read/popped, no longer on stack
  - Write an element to the stack, can read/pop most recent element added
  - Top keeps track of the most recently added element
  - Use an array of size n, \( S[0...n-1] \)

**Stack**
- Start pushing elements starting at index 0
- Assume a maximum size of stack \( n \)
- Use an array of size \( S[0...n-1] \)

**Chapter 10: Elementary Data Structures**

**Implementation for Stack**
- Last-in, first-out
- When it is read/popped, no longer on stack
- Write an element to the stack, can read/pop most recent element added
- Top keeps track of the most recently added element
Implementation for Queues

- Queues
  - Assume a maximum size of $n$

  - Use an array of size $S[0...n-1]$
  - Assume a maximum size of $n$

  - Need two variables:
    - $Q$ (most recently added)
    - $Q$ (next to be dequeued)

  - Dequeue: take off of queue
  - Enqueue: put on queue

- Use for Stack

```python
def Stack-Empty(S):
    if S_top == 0:
        return True
    else:
        return False

def Push(S,x):
    S_top = S_top + 1
    S[S_top] = x

def Pop(S):
    if Stack-Empty(S):
        error "underflow"
    S_top = S_top - 1
    return S[S_top+1]
```

Code for Stack
Code for Queues

- Code assumes queue array starts at 1
- What is the empty queue?
- What is maximum number of elements that can be stored?
- Why not have Q.tail point at last element, rather than next?
- What is maximum number of elements that can be stored?
- What is the empty queue?
Groups of Information

- Will need more complex data structures

Elements of a dynamic set
- Usually consists of a number of types of information
- Usually one type is viewed as the key, used for searching, sorting
- Like an street address, city, zip code, phone number
- Need data structure that can encapsulate pieces of information

Overview

What is the Course About?

- Classes, Objects, and Pointers
- Data Structures Intro
- Linked Lists
- Rooted Trees
How can we do this in Python?

Python

How can we do this in Python?

- Kind of ugly
  
  - Either and mother of each person
  - Can have an array for father and mother to which has the index for the
  - Each person has a unique index into arrays
  - Have an array for name, date of birth, gender

Also, need a way to refer to other pieces of data

Example: Family tree

Reference other set elements

- Not just their names, but a way that we can access all the data about them
- Want to list who their parents are
- For each person, want to store name, date of birth, gender
A Simple Class

```python
class Person:
    def __init__(self, name, gender, birthday):
        self.birthday = birthday
        self.gender = gender
        self.name = name

d = Person("donald", "male", "1946-6-14")
m = Person("melania", "female", "1970-4-26")
b = Person("baron", "male", "2006-3-20")

print(b.name)
```

Python with Classes

- **Class** specifies how the class works
  - Function `init` specifies how to create a new object
  - Called by using the name of the class
- **Object** has 3 fields in it: birthday, gender, name
- Should initialize all of the object fields
- Called by using the name of the class

---

Will use classes in constructing complex data types

- An **instance** of a group (or class) is called an object
- Associate functions on the group
- Include a `self` in the object that refers to another object
- Refer to an object with a constant or a variable
- Group data fields together as an object
- Classes allow you to
- Python has classes

---
Pointers

\[ d, m, b \] are not the actual objects, but pointers to them

```python
# Define the Person class
class Person:
    def __init__(self, name, gender, birthday):
        self.birthday = birthday
        self.gender = gender
        self.name = name
        self.father = None
        self.mother = None

d = Person("donald", "male", "1946-6-14")
m = Person("melania", "female", "1970-4-26")
b = Person("baron", "male", "2006-3-20")
b.father = d
b.mother = m
e = d
e.name = "donnie"
print(b.father.name)
```

Referencing variables inside of an object

```python
d = Person("donald", "male", "1946-6-14")
m = Person("melania", "female", "1970-4-26")
b = Person("baron", "male", "2006-3-20")
b.father = d
b.mother = m

# Accessing variables
self.name = None
self.father = None
self.mother = None
self.gender = "male"
self.birthday = "1946-6-14"

def __init__(self, name, gender, birthday):
    self.name = name
    self.gender = gender
    self.birthday = birthday

# Define the Person class
class Person:
```
Operations on Dynamic Sets

- For any dynamic set:
  - Search(S,k) returns pointer to an element or None
  - Insert(S,x)
  - Delete(S,x)
  - Minimum(S)
  - Maximum(S)
  - Predecessor(S,x)
  - Successor(S,x)

- For totally ordered sets:
  - Minimum(S)
  - Maximum(S)
  - Successor(S,x)
  - Predecessor(S,x)
Linked List (Chapter 10.2)

- Linked list is a data structure
  - Objects are arranged in a linear order
  - More flexible than an array as you can insert/delete objects faster

- Remainder of section: assume unsorted doubly linked

  - Circular: prev pointer of head element points to tail element
  - Sorted: min element is at head of list, next pointer points to next largest
  - Singly: elements point to next element
  - Doubly: also point to prev element
  - Circular: head element points to tail element
  - Sorted: ascending order
  - Singly or doubly linked

Linked List (Chapter 10.2)
Methods versus Functions

Methods

• Previous example had ListSearch as a function
  - Can work on any type of structure that has a head field, and the head points to something that has a next field, and that thing has a next etc.
  - Works because Python is not tightly typed
  - Can work on any type of structure that has a head field, and the head points to something that has a next field, and that thing has a next etc.

Alternative: make ListSearch a method

- Method is defined in a class and operates on an object
  - Method is defined in a class and operates on an object
  - Defined with the first argument named 'self', which is the object being operated on
  - Called by objectname followed by a dot followed by method name and any additional parameters
  - Called by objectname followed by a dot followed by method name and any additional parameters
  - Works because Python is not tightly typed
  - Defined with the first argument named 'self', which is the object being operated on

Searching a linked list

```python
class Element:
    def __init__(self, k):
        self.key = k
        self.next = None
        self.prev = None

class List:
    def __init__(self):
        self.head = None

def ListSearch(L, k):
    x = L.head
    while x is not None and x.key != k:
        x = x.next
    return x

l = List()
a = Element(1); l.head = a
b = Element(4); a.next = b; b.prev = a
c = Element(16); b.next = c; c.prev = b
print(ListSearch(l, 4).key)
```
Methods allow you to associate functions on class objects directly with the definition of the class.

Discussion

class List:
    def __init__(self):
        self.head = None
    def Search(self, k):
        x = self.head
        while x is not None and x.key != k:
            x = x.next
        return x

l = List()...
print l.Search(4).key
Deleting from a Linked List

- **List Method:**
- This has to be method for list, not for element
- It might have to update the head pointer
- Boundary conditions force additional checks

```python
def Delete(self, x):
    if x.prev is not None:
        x.prev.next = x.next
    else:
        self.head = x.next
    if x.next is not None:
        x.next.prev = x.prev
```

Inserting into a List

- **Inserting at end of list:** simpler, but need pointer to end of list
  ```python
  l = List()
a = Element(1); l.head = a
b = Element(4); a.next = b; b.prev = a
c = Element(16); b.next = c; c.prev = b
  ```
- **Inserting at start of list:** more complex but have pointer to start of list
  ```python
  def Insert(self, x):
      x.next = self.head
      if self.head is not None:
          self.head.prev = x
      self.head = x
      x.prev = None
  ```

- **List Method:** Insert at start
- **List Method:** Insert at end
Main point: logic is simpler rather than the slight gain in speed

```python
def Delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev

def Insert(self, x):
    x.next = self.sentinel.next
    self.sentinel.next.prev = x
    self.sentinel.next = x
```

Sentinels

- To get rid of boundary case, have a sentinel
- Sentinel is a dummy object always in the list
- Makes this into a circular list

Code with Sentinels

(a) (b) (c) (d)
Questions about Code

- What does search return if item is not in the list?
- How would you check for this?
- What is a better option?
- How would the code be changed?
- How would you check for not-found?
- What do we change code to store satellite data?

- Where should satellite data be stored?
- We do not want to make a new instance of List for each different type of satellite data.
- How do we change code to store satellite data?

class List:
    def __init__(self):
        self.sentinel = Element(None)
        self.sentinel.next = self.sentinel
        self.sentinel.prev = self.sentinel

    def Search(self, k):
        x = self.sentinel.next
        while x != self.sentinel and x.key != k:
            x = x.next
        return x

l = List();
l.Insert(Element(4)); l.Insert(Element(8)); l.Insert(Element(16))
print l.Search(8).key
print l.Search(5).key
Overview

What is the Course About?
- Linked Lists
- Data Structures Intro
- Classes, Objects, and Pointers
- Rooted Trees

Class Inheritance

```python
class Element:
    def __init__(self, k):
        self.key = k
        self.next = None
        self.prev = None

class MyEle(Element):
    def __init__(self, k, age, name):
        Element.__init__(self, k)
        self.age = age
        self.name = name

l = List(); l.Insert(MyEle(4, 35, "peter")); l.Insert(MyEle(8, 25, "paul"))
print l.Search(8).name
print l.Search(5).name

List uses Element class to define sentinel
- List doesn't care as long as entry has next, prev, and key fields
- Add entries using MyEle, which saves satellite data
```
Unbounded Branching

- Parent cannot reference every child, like in a binary tree
  - Define a class for nodes
    - If last sibling, pointer is None
    - Higher sibling points to next sibling
    - Left child: points to left-most child (like a single linked list)
      + If no children, pointer is None

Rooted Trees (Chapter 10.4)

- Define a class for trees
  - Has fields: parent, left child and right child pointers, and key
  - Has fields: parent, left child and right child pointers, and key

- Left child: points to left-most child (like a single linked list)
  + If no children, pointer is None

- Higher sibling points to next sibling
  + If last sibling, pointer is None

- Parent cannot reference every child, like in a binary tree
  - Define a class for nodes
    - If last sibling, pointer is None