Overview

⇒ What is the Course About?
• Data Structures Intro
• Classes, Objects, and Pointers
• Linked Lists

CS532: Design and Analysis of Algorithms

• Monday & Wednesday at 4:00pm to 5:30pm
• Starts June 25th
• Learning Objectives:
  - Improve your programming skills
    + Including complex data structures
    + Object oriented programming
  - Know how to analyze the efficiency of an algorithm
    + How much time does it take as the size of the input grows
  - Know how to prove the correctness of an algorithm
Background Needed

- Intended to be taken earlier in your education
- Anyone admitted to the program
  - At a minimum, expected to have 1st year undergraduate CS major courses
- A course in discrete math also useful

Logistics

- Grading
  50% homework
  25% midterm
  25% final
- Course website: cslu.ohsu.edu/~heeman/cs532
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
  - Worth up to half the marks that you lost
    + Really good explanations might even get more
  - Should show that you reviewed and understood answer key and understood whether you answer was correct
  - See sample homework for how to format this

Homework

• Homework is due Friday at 11:59pm
  - If needed, you can email me with a request for an extension
  - Extension for one homework will be automatically granted

• Homework must be submitted through Sakei
  - If you have problems, email it to me heemanp@ohsu.edu
  - Single pdf with your answers on it
    + Homework should be typeset
    + Can include pictures of hand-drawn solutions taken with cellphone
  - Turn in code as well
    + We will supply you with a code that will run your code on several examples
    + Make sure that your code interfaces with this correctly
    + We will test it other cases
C versus Python

• Could have taught this course in C
  - Pointers are built into the language
  - C does not have a built-in dictionary datatype
  - But few people know C, so would spend a lot of time teaching this

• Instead using Python
  - Don’t use dictionaries or advanced packages
  - Use classes/objects to get pointers
  - Plus: get more in-depth experience with a programming language
  - Plus: get a solid foundation in object-oriented programming
    - Note: not in textbook

Academic Integrity

• You can do the homeworks with your colleagues
  - Document how you collaborated
    - e.g., worked on entire question together, versus clarified what the question meant

• After you have come up with a solution, redo it on your own
  - Nothing in, nothing out, wait at least an hour between
Dynamic Sets

• From: III Data Structures of textbook
• Set: collection of elements (like integers)
• Dynamic sets
  - Can be manipulated by algorithms
  - Can grow, shrink, or otherwise change over time
• Different algorithms might require different operations
  - Many algorithms just need to insert, delete, and test membership
    + Dynamic set that supports this is called a dictionary
  - Some algorithms need to insert, and extract the smallest element
    + Min-priority queue
  - What operations are needed by an algorithm and how efficiently they need to be done determines what data structure type to use
Chapter 10: Elementary Data Structures

• Arrays
  - Elements are numbered starting at 0 (or 1)
    + Read, write any element

• Stacks and Queues
  - For some tasks, you want to restrict how elements are accessed

• Stack: last-in, first-out
  - Write an element to the stack, can read/pop most recent element added
  - When it is read/popped, no longer on stack

• Queue: first-in, first-out
  - Can write an element to queue, can read/dequeue the oldest element
  - When it is read/dequeued, no longer in queue

Implementation for Stack

• Stack
  - Assume a maximum size of stack $n$
  - Use an array of size $n$ S[0...n-1]
  - Start pushing elements starting at index 0
  - S.top keeps track of the most recently added element
Implementation for Queues

- Queues
  - Assume a maximum size of $n$
  - Use an array of size $n$ $S[0...n-1]$
  - Enqueue: put on queue
  - Dequeue: take off of queue
  - Need two variables:
    + $Q_{\text{head}}$ (next to be dequeued)
    + $Q_{\text{tail}}$ (most recently added)

---

Code 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]
```
Continued

• What is the empty queue?
• What is maximum number of elements that can be stored?
• Why not have $Q_{\text{tail}}$ point at last element, rather than next space?

```
def Enqueue(Q,x):
    Q[Q_tail] = x
    if Q_tail == Q_length:
        Q_tail = 1
    else:
        Q_tail = Q_tail + 1

def Dequeue(Q):  
x = Q[Q_head]  
if Q_head == Q_length:
    Q_head = 1
else:
    Q_head = Q_head + 1
return x
```

• Code assumes queue array starts at 1
Overview

• What is the Course About?
• Data Structures Intro
  ⇒ Classes, Objects, and Pointers
• Linked Lists

Groups of Information

• Will need more complex data structures
• Need data structure that can encapsulate pieces of information
  - like an street address, city, zipcode, phone number
• Elements of a dynamic set
  - Usually consist of a number of types of information
  - Usually one type is viewed as the key, used for searching, or sorting
  - Rest is viewed as satellite data
Reference other set elements

- Also, need a way to refer to other pieces of data
- Example: family tree
  - For each person, want to store name, date of birth, gender
  - Want to list who their parents are
    + Not just their names, but a way that we can access all the data about them

Python

- How can we do this in python?
  - Have an array for name, date of birth, gender
  - Each person has an unique index into arrays
  - Can have an array for father and mother to, which has the index for the father and mother of each person

- Kind of ugly
Python with Classes

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

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
```

- class specifies how the class works
- Function __init__ specifies how to create a new object
  - Called by using the name of the class
  - Should initialize all of the object fields
- Object has 3 fields in it: birthday, gender, name
Referencing variables inside of an object

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
print b.father.name

Pointers

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

• d, m, b are not the actual objects, but pointers to them
Overview

• What is the Course About?
• Data Structures Intro
• Classes, Objects, and Pointers
⇒ Linked Lists

Operations on Dynamic Sets

S is a set, k is a key, x is a pointer to an element

• For any dynamic set:
  - Search(S,k) returns pointer to an element or None
  - Insert(S,x)
  - Delete(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
- Singly or doubly linked
  - Singly: elements point to next element
  - Doubly: also point to prev element
- Sorted or not
  - Sorted: min element is at head of list, next pointer points to next largest element
- Circular or not:
  - Circular: prev pointer of head element points to tail element, and next pointer of tail points to head
- Remainder of section: assume unsorted doubly linked

Operation

![Diagram of linked lists with prev, key, and next pointers](image-url)
Methods versus Functions

- **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, and that thing has a next etc
  - Works because python is not tightly typed

- **Alternative: make ListSearch a method**
  - 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
    - Similar to how list objects have methods for append, insert, remove, sort, etc

```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
```
Discussion

• Methods allow you to associate functions on class objects directly with the definition of the class

• Makes it so that you can encapsulate code better so that only code inside of the class needs to know how the class works

• Forces you to think about what data you want, and what operations will work on it

Code

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
```
Sentinels

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

```
(a) L.nil
   ┌───┐ ┌───┐ ┌───┐ ┌───┐
   │ L │ │ 9 │ │ 16 │ │ 4 │
   └───┘ └───┘ └───┘ └───┘

(b) L.nil
   ┌───┐ ┌───┐ ┌───┐ ┌───┐
   │ L │ │ 9 │ │ 16 │ │ 4 │
   └───┘ └───┘ └───┘ └───┘
   ┌───┐ ┌───┐ ┌───┐ ┌───┐
   │ 25 │ │ 9 │ │ 16 │ │ 4 │
   └───┘ └───┘ └───┘ └───┘

(c) L.nil
   ┌───┐ ┌───┐ ┌───┐ ┌───┐
   │ L │ │ 9 │ │ 16 │ │ 4 │
   └───┘ └───┘ └───┘ └───┘
   ┌───┐ ┌───┐ ┌───┐ ┌───┐
   │ 25 │ │ 9 │ │ 16 │ │ 4 │
   └───┘ └───┘ └───┘ └───┘
```

Code with Sentinels

```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
    x.prev = self.sentinel

• Main point: logic is simpler rather than the slight gain in speed
```
Questions about Code

- Code for search is from textbook.
  - 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 now check for not-found?

- How 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 we want to store

```python
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
```
Unbounded Branching

- Parent cannot reference every child, like in a binary tree

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
- Define a class for nodes
- List doesn't care as long as entry has next, prev, and key fields