CS 304 Notes

Data Structures & Algorithms

Jaeden Bardati

Last modified April 5, 2022

1 Python and Object-Oriented Programming

Python is a *high-level programming language* which makes it useful when studying data structures and algorithms.

Python is an *interpreted* language, where *source code* (also referred to as *scripts*), in the form of files with the *.py* suffix, are run by an *interpreter*. It is common to use an integrated development environment (IDE) to aid in displaying and editing Python code. IDEs for Python include the built-in IDLE, PyCharm, Spyder, and others.

1.1 Objects in Python

Python is an *object oriented* language where **classes** are the basis of all data types. Examples of data types in Python include int, float, str.

An **assignment statement** assigns an **identifier** (or name) to an object. Identifiers are associated with a *memory address* and are similar to a pointer in languages such as C++ or Java. For example, the statement

```
1  temp = 98.6
```

associates the identifier "temp" to the float value 98.6.

Identifiers are **case-sensitive**. Namely, a variable named "temp" is different than one named "Temp".

Python is a **dynamically typed** language, unlike C++ or Java. That is to say, when making the association of an identifier in Python, the data type is not explicitly declared. In the code above, the data type is determined automatically by the interpreter to be a float.

It is possible to establish an **alias** by assigning a second identifier to an object as follows

```
temperature = temp
```

After an alias is made, either name can be used to refer to the object.

The process of creating a class is called **instantiation**. To do this, we call the **constructor** of a class. If we have defined the class called Animal then we would do this by

```
a = Animal()
```

Note that we can also pass parameters to the Animal constructor.

See Chapters 1 and 2 in the textbook for more.

2 Design and Analysis of Algorithms

2.1 Data structures vs. Abstract Data Types

There are many **abstract data types**: list, set, queue, stacks, dictionary, etc.

Data structures are the implementation of abstract data types. Examples include: arrays, trees, hash tables, etc.

Our goal is to find the best data structure to use. The best data structure is one that minimizes the **time complexity**.

For example, if we have a list $L = [1, 2, 3]$ and we want to insert an element. If we use an array abstract type, then it is easy to write to insert an element at the end of the list, but it is more complicated to insert it at the beginning of the list (must shift all other elements over 1).

Another example is the binary search algorithm (e.g. looking for a number in a phone book) has the time complexity of $\theta(log_2 n)$.

2.2 Insertion Sort

The insertion sort algorithm written in pseudocode in Algorithm 1.

We can check the code using a trace for A = [3, 2, 1]. The variables as they change are:

1. $j = 2$
2. key = 2
3. $i = 1$

4. $A = [3, 3, 1]$
5. $i = 0$
6. $A = [2, 3, 1]$
7. $j = 3$
8. key $= 1$
9. $i = 2$
10. $A = [2, 3, 3]$
11. $i = 1$
12. $A = [2, 2, 3]$
13. $i = 0$
14. $A = [1, 2, 3]$
15. $j = 4$

2.2.1 Algorithm efficiency

WE can go line by line through the program and associate an arbitrary time cost as well as a number of times that the line will run. We can represent this in Table 1.

The time that the algorithm takes is

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case: $t_j = 0$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) = (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8)$$

Algorithm 1 InsertionSort(A)

```
1: for (j = 2 to length(A)) do
2:     set key = A[j]
3:     // insert A[j] into the
4:     // sorted sequence A[1..j − 1]
5:     set i = j − 1
6:     while (i > 0 and A[i] >key) do
7:         set A[i + 1] = A[i]
8:         set i = i − 1
9:         set A[i + 1] = key
10: stop
```

Table 1: Insertion Sort Algorithm Cost

Line	Cost	Times
1	c_1	n
2	c_2	$n-1$
3	0	0
4	0	0
5	c_3	$n-1$
6	c_4	$\sum_{j=2}^{n} t_j$
7	c_5	$\sum_{j=2}^{n}(t_j-1)$
8	c_6	$\sum_{j=2}^{n}(t_j-1)$
9	c_7	$n-1$

where we let t_j be the number of times that the while loop is executed for a given j.

This can be written in the form $an+b$. Namely, the best case is a linear function with n.

Worse case: $t_j = j$ for $j = 2, 3, ...n$, therefore,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

And,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Thus,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

This can be written in the form $an^2 + bn + c$. Namely, the worst case is a quadratic function with n.

2.3 Asymptotic Analysis

We say insertion sort has a worst-case running time of $\Theta(n^2)$. All we care about is the order of n as n asymptotically increases without bound. For example,

$\frac{n^3}{1000} - 100n^2 - 100n + 3$ has a time complexity of $\Theta(n^2)$.

Definition: For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) \equiv \{f(n) : \exists \text{ positive constants } c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \, \forall \, n \geq n_0\}$$

For all $n \geq n_0$ f(n) is equal to g(n) within a constant factor. Note that we often abuse the notation and write that $f(n) = \Theta(g(n))$ rather than $f(n) \in \Theta(g(n))$ as one might expect.

For example, let us justify that $\frac{n^2}{2} - 3n = \Theta(n^2)$. We must determine positive constants c_1, c_2, n_0 s.t. $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2 \; \forall \; n \geq n_0$.

Dividing by n^2, $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$, where we choose $c_2 = \frac{1}{2}$, $c_1 = \frac{1}{14}$ for $n_0 \geq 7$.

Definition: For a given function $g(n)$, we denote $O(g(n))$ as the set of functions

$$O(g(b)) \equiv \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \, \forall \, n \geq n_0\}$$

We call this "big O" notation. See Figure 3.5 in the textbook for a visualization. It is important to note that if $f(n) = \Theta(g(n))$, then necessarily $f(n) = O(g(n))$. Also note that the $\Omega(g(n))$ notation indicates a minimum asymptotic trend.

3 Array-Based Sequences

3.1 Sequence Types in Python

The main sequence types in Python are the **list**, **tuple** and **string**. These sequence types can be **referential** or **compact**, and **mutable** or **immutable**. Referential are distinguished from compact arrays later on in these notes. Mutable arrays are array that can be changes in some way (appended to, values changed, values inserted, etc.), whereas immutable arrays cannot be changed. The following are the properties of the sequence types.

- lists are *referential* and *mutable*
- tuples are *referential* and *immutable*
- strings are *compact* and *immutable*

We can do a variety of operations on these sequence types. Some of them include "len(data)", which finds the length of the array data, or "data1 == data2", which checks if the arrays data1 and data2 are equal. See Tables 5.3 and 5.4

from the textbook for a comprehensive list.

The non-mutating behaviors are

- len(data): Finds length of array
- data[j]: Returns the jth index
- data.cout(value): Number of times value is in data
- data.index(value): The index of the value in data
- Value in data: Same as above
- data1 == data2: Checks if the arrays are equal
- d2 = data[j:k]: Creates a new array which is a slice from index j (inclusively) to k (exclusively)
- d2 = data1 + data2: Creates a new array which is the concatenation of the arrays
- d2 = c*data: Creates a new array which is c copies of data

and the mutating behaviors are

- data[j] = val: Change the value at index j to val
- data.append(val): Append to the end of the array
- data.insert(k, val): Insert val in an element at index k (increases size)
- data.pop(): Removes (and returns) the last element of the array
- data.pop(k): Removes (and returns) the kth element of the array
- del data[k]: Removes the kth element of the array
- data.remove(val): Searches the array for val and removes it
- data1.extend(data2): Appends the array data2 to the array data1
- data.reverse(): Reverses the order of the array
- data.sort(): Sorts the array in increasing order

An implementation of data.insert in our previous array class in Python is

```
1  def insert (self, k, value):
2    if self._n == self._capacity:
3      self._resize(2*self._capacity)
4    for j in range(self._n, k, -1):
5      self._A[j] = self._A[j-1]
6    self._A[k] = val
7    self._n += 1
```

3.2 Low-Level Arrays

3.2.1 What is an Array?

First, we must define RAM. Random-access memory (RAM) is memory that can be randomly-accessed. Randomly-accessed means that any random element can be accessed just as easily as any other. We can visualize this in a table as follows

S	A	M	P	L	E

An array is a group of related values sorted one after another in a contiguous portion of RAM. To access an array, we refer to the memory address that the array begins at and add the index. Namely,

```
A[i] = start + index
```

3.2.2 Note for Python Arrays

Python arrays use pointers to refer to an array in memory. Therefore, we need to explicitly copy an array to make sure it is copied. In the following example, an alias is made and so the sorted array [1, 2, 3] would be outputted.

```
x = [3, 2, 1]
y = x
y.sort()
print(x)
```

3.2.3 Referential Arrays

Referential Arrays are arrays that store memory addresses (object references). For example, an array of strings is a referential array because strings are stored as pointers to another location in memory (to a compact array).

Let us say that we have an array of primes:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

Then, we can create a sub-array with

```
temp = primes[3:6]
```

The array temp is a referential array to a part of the array primes (or rather, to the objects to which the elements reference). See Figure 5.5 in the textbook. If then we set

```
temp[2] = 15
```

Then the temp array at index 15 is now pointing to another location in memory with a new entry 15, but the rest of the array will still point to their original addresses. See Figure 5.6 in the textbook.

If we have an array,

```
counters = [0]*8
```

This will create an array of *pointers* to the same 0 object. See Figure 5.7 in the textbook. Adding 1 to one of the elements will create a new 1 object and make that element point to it (see Figure 5.8).

3.2.4 Compact Arrays

Compact arrays are when the elements are stored directly in the array (not an array of pointers). An example of this is a string, with is an array of characters (not pointers of characters). Compact arrays have advantages over referential arrays. Namely, they are good because they

1 Have less memory usage: No overhead devoted to storage of memory references
2 Have higher performance:
2a) direct access
2b) principle of locality (if we access a certain location in memory, we are likely to access the array again nearby)

To use a compact array, we can use the array module. To create one, we must pass a list of a certain type, along with the type code for that type. For a list of signed integers, we use the type code 'i'. For example, we could write

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

3.3 Dynamic Arrays and Amortization

Dynamic Arrays are arrays that can dynamically increase in time. To do this, dynamic arrays reserve a certain amount of space in memory for future appends until it runs out of reserved space, in which case it rewrites the new array along with more reserved space. Python's list class is a dynamic array. We can explore the relation with the code

```
for k in range(10):
  a = len(data)
  b = sys.getsizeof(data)
  print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
  data.append(None)
```

We can see that the size in memory increases exponentially. This is because the number of spots added is proportional to the size of the array at a time in Python lists. The amount of space that is dynamically added to the array at a time increases as the list grows (it will not always be 4).

Let us now implement a dynamic array ourselves

```
import ctypes

class DynamicArray:
  """A Dynamic array class akin to a simplified python list"""

  def __init__(self):
    """Create an empty array"""
    self._n = 0
    self._capacity = 1
    self._A = self._make_array(self._capacity)

  def __len__(self):
    """Return number of elements stored in array"""
    return self._n

  def __getitem__(self, k):
    """Return item at index k"""
    if not 0 <= k < self._n:
      raise IndexError("Invalid index")
    return self._A[k]

  def _resize(self, c)
    """Resize internal array to capacity c"""
    B = self._make_array(c)
    for k in range(self._n):
      B[k] = self._A[k]
    self._A = B
    self._capacity = c

  def _make_array(self, c):
    """Return a new array with capacity c"""
    return (C * ctypes.py_object)()

  def append(self, obj):
    """adds object to end of the array"""
    if self._n == self._capacity:
      self._resize(2*self._capacity)
      self._A[self._n] = obj
      self._n += 1

```

We can test it out using:

```
arr = DynamicArray()
arr.append(1)
```

We now would like to analyze the efficiency.

Reminder: $O(n)$ means *at most* n efficent and $\Omega(n)$ means *at least* n. Here we have $\Omega(n)$ efficiency.

Using amortization, we can show that an append is $O(1)$, and n appends are $O(n)$. Namely, a single append does not depend on the size of the array.

Proposition 5.1: Let S be a sequence implemented using a dynamic array with initial capacity 1. Using the strategy of doubling the array size when full, the total time to perform n operations is $O(n)$.

Justification: Assume 1 cyber-dollar pays for each append where we do not expand array. Assume growing array from size k to $2k$ requires k cyber-dollars. We charge each append 3 cyber-dollars. An overflow occurs when array S has 2^i elements for some integer $i \geq 0$ and the size of the array representing S is 2^i. Namely, doubling the size of the array will cost 2^i cyber-dollars. These cyber-dollars can be found in cells 2^{i-1} through $2^i - 1$.

Proposition 5.2: Performing a series of n append operations using a fixed increment with each resize takes $\Omega(n^2)$.

Justification: Let $c \geq 0$ represent a fixed increment in capacity used for each resize event. During the series of append operations, time will be spent initializing arrays of size c, $2c$, $3c$, ..., mc for $m = \frac{n}{2}$. So the overall time is proportional to $c + 2c + 3c + ... + mc = \sum_{i=1}^{m} ci = c\sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2}$. This is $\Omega(n^2)$.

3.4 Multidimensional Arrays

If we do

```
l = [[9]*10]*5
l[0][3] = 5
```

It will change the values of throughout the array. Instead, the correct method to instantiate a multidimensional array is

```
l = [[9]* 10 for j in range(5)]
l[0][3] = 5
```

This will work by ensuring that they do not all point to the same value.

3.5 Numpy Arrays

For big data, we can use the NumPy module. We can make an nd-array with

```
z = np.zeros((10, 10)) # will create a 10x10 array of 0s
```

There are many different methods that NumPy has which are highly optimized in precomplied C-libraries.

4 Stacks

The stack is one of the most important and most widely used data structures. Stacks are inserted and removed using the **last in - first out (LIFO)** principle.

Formally, a stack is an ADT (abstract data type) that supports two main methods:

- S.push(e): Add e to top of stack
- S.pop(): Remove and return top element

We can also make it support the following methods:

- S.top(): Returns a reference to the top element of the stack
- S.is_empty(): Returns true if empty, otherwise false
- len(S): Returns the length of the stack

The following table shows an example of these methods in use.

Operation	Return Value	Stack contents
S.push(5)	None	[5]
S.push(3)	None	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]

We will use the **adaptor pattern** to construct a stack from the list class using Python.

4.1 Stack Implementation in Python

```
class ArrayStack:
  """LIFO Stack implementation using Python list as underlying
  shape."""
  def __init__(self):
    """Create an empty Stack """
    self._data = []

  def __len__(self):
    """Returns length of stack"""
    return len(self._data)

  def is_empty(self):
    """Return True if stack is empty"""
    return len(self._data) == 0

  def push(self, e):
    """Adds e to the top of the stack"""
    self._data.append(e)

  def top(self):
    """Returns the top element"""
    if self.is_empty():
      raise EmptyException("Stack is empty")
    return self._data[-1]

  def pop(self):
    """Removes and returns the top element."""
    if self.is_empty():
      raise EmptyException("Stack is empty")
    return self._data.pop()

```

4.2 Complexity Analysis

The following table shows the time complexity of each of these methods.

Operation	Running Time
S.push(e)	$O(1)$*
S.pop()	$O(1)$*
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

* means it is amortized.

An example of using a stack in Python is the following.

```
a = [1, 2, 3], S = ArrayStack()

for i in range(len(a)):
  S.push(a[i])
# S = [1, 2, 3]

while not S.is_empty():
  a[i] = S.pop()
# a = [1, 2, 3]
```

4.3 Matching Parentheses using a Stack

In pseudocode,

1. Initialize empty stack
2. Read characters until end (call current character char)
3. If char == opening parentheses, push the char to stack
4. Else If char == closing parentheses and stack is empty, produce error
5. Otherwise, pop stack and if popped symbol does not match char, produce error
6. If at EOF (end of file) and stack is not empty, produce error

For example, '(())' would produce no errors. but '(()' would.

The Python code for this is

```
def is_match(expr):
  """Returns true if all delimiters match"""
  lefty = '([{'
  righty = ')]}'
  S = ArrayStack()
  for c in expr:
    if c in lefty:
      S.push(c)
    elif c in righty:
      if S.is_empty():
        return False
      if righty.index(c) != lefty.index(S.pop()):
        return False
  return S.is_empty()
```

4.4 Polish Notation

In polish notation, the expression

$$1 + 2 = 3$$

is written as

$$1\ 2\ +\ =\ 3$$

The infix expression

$$6 * (5 + (2 + 3) * 8 + 3) = 288$$

changes its value with a change in the placement of the brackets. However, its polish notation (postfix) equivalent

$$6\ 5\ 2\ 3\ +\ 8\ *\ +\ 3\ +\ *\ =\ 288$$

does not need that delineation.

To evaluate polish notation we

- Read expression from left to right
- If operand, push to stack
- If operator, pop two values off stack, perform operation and push back on stack

That is how to evaluate it, but how do we convert between the forms.

4.4.1 Algorithm for converting infix to postfix

The steps of the algorithm are

1. If operand, place in output
2. When we encounter an operator, place on stack, also stack left parentheses
3. If we encounter a right parentheses, pop the stack and write symbols until we encounter a left parentheses
4. If we see any other symbol (such as +, *, or '('), we pop entries from stack until we find an entry of lower priority
5. When popping is done, push operator to stack
6. If at end, pop entire stack and write it to the output

We can check this with the example

Infix: $a + b * c + (d * e + f) * g$
Postfix: $abc * +de * f + g * +$

5 Queues

Queues are **first-in first-out** (FIFO). For example, waiting in a grocery line is a queue.

The basic methods of the queue abstract data class are

- Q.enqueue(e): Adds e to the back of the queue.
- Q.dequeue(): Removes and returns the first element of queue.
- Q.first(): Returns the first element
- Q.is_empty()
- len(Q)

The following table shows an example of these methods in use.

Operation	Return Value	Q
Q.enqueue(5)	None	[5]
Q.dequeue(3)	None	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.is_empty()	False	[5]
Q.dequeue()	3	[ ]

We can write the queue in terms of the built-in Python list. However doing this as it stands give a dequeue time of $O(n)$, because we would need to swap all the elements down to fill the space.

To improve this, we could try setting the first element to None and then have a pointer that moves forward at each dequeue. But this would waste space with a lot of Nones at the beginning of the list. So, we must use circular arrays to avoid this increase in size each dequeue.
A **circular array** uses the modular arithmetic. In python, this is done using the modulus symbol %.

```
n = a // b     # integer division
m = a % b      # modulus
# m = a - b*n is the equivalent expression for m
```

We can get the location to insert the next enqueue using

```
index = (f + size(Q)) % len(Q)
```

where f is the pointer location, size(Q) is an arbituary size and len(Q) is the length of the Q.

We can then update the pointer with

```
f = (f + 1) % len(Q)
```

An implementation of a queue in Python is

```
class ArrayQueue:
  """FIFO Queue using Python List"""
  def __init__(self):
    self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0

  DEFAULT_CAPACITY = 10

  def __len__(self):
    return self._size

  def is_empty(self):
    return self._size == 0

  def first(self):
    if self.is_empty():
      raise Exception("List is empty")
    return self._data[self._front]

  def dequeue(self):
    if self.is_empty():
      raise Exception("List is empty")
    answer = self._data[self._front]

    # circular array implementation
    self._data[self._front] = None    # to help garbage collect
    self.front = (self._front + 1)% len(self._data)

    self._size -= 1

    if 0 < self._size < len(self._data)//4:
      # if 25 % of capacity is used, resize to make smaller
      self._resize(len(self._data) // 2)
    return answer

  def enqueue(self, e):
    if self._size == len(self._data):
      self._resize(2*len(self._data))
    avail = (self._front + self._size) % len(self._data)
    self._data[avail] = e
    self._size += 1

  def _resize(self, cap):
    old = self._data
    self._data = [None] * cap
    walk = self._front
    for k in range(self._size):
```

```
49            self._data[k] = old[walk]
50            walk = (walk + 1) % len(old)
51        self._front = 0
```

5.1 Complexity Analysis

Operation	Running Time
Q.enqueue(e)	$O(1)$*
Q.dequeue()	$O(1)$*
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

* means it is amortized.

6 Linked List

A linked list is an alternative to an array. There are singly and doubly linked lists.

6.1 Singly Linked List

A singly linked list is a collection of nodes that form a linear sequence.

Each node stores

- A reference to an object
- A reference to the next node

For example, a singly linked list of airports could have a node that points to a flight in Montreal and also points to another flight in Toronto, which also points to Vancouver.

The first node of a linked list is called the **head** and the last node of a linked list is called the **tail**.

Random access in a linked list is quite slow, however, increasing the size is easier, since it is not needed to shuffle all the elements over as you do in an array.

To add to the front of the list, we could use the psudeocode:

```
def add_first(L, e):
  newest = Node(e)
  newest.next = L.head
  L.head = newest
  L.size = L.size + 1
```

We can also add to the tail of the list in a similar way.

```
def add_last(L, e):
  newest = Node(e)
  newest.next = None
  L.tail.next = newest
  L.tail = newest
  L.size = L.size + 1
```

To remove the first node, we simply need to set the head to the next one and decrement the size. To remove the tail, we need to set the node before it to None, but that requires iterating through the linked list which is $O(n)$.

6.2 Stack Implementation using a Singly-Linked List

We can implement a stack using a linked list as the underlying structure.

```
class LinkedStack:
  """LIFO Stack implementation using a Linked List"""

  class _Node:
    __slots__ = '_element', '_next'   # optimization for the
  store of these two fields of the class node

    def __init__(self, element, next):
      self._element = element
      self._next = next

  def __init__(self):
    self._head = None
    self._size = 0

  def __len__(self):
    return self._size

  def is_empty(self):
    return self._size == 0

  def push(self, e):
    self._head = self._Node(e, self._head)
    self._size += 1

```

```
25      def top(self):
26        if self.is_empty():
27          raise Exception("Stack is empty")
28        self._head._element
29
30      def pop(self):
31        if self.is_empty():
32          raise Exception("Stack is empty")
33        answer = self._head._element
34        self._head = self._head._next
35        self._size -=1
36        return answer
```

6.2.1 Complexity Analysis

Operation	Running Time
S.push()	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Linked stacks are good in real-time applications since no amortization occurs (could happen at a critical time).

6.2.2 Queue Implementation using a Singly-Linked List

We can implement a queue using a linked list as the underlying structure if we add a reference to the tail. An enqueue should be done to the end of the list so that the dequeue is easily done in the front of the list. The other way around would result in a dequeue time of $O(n)$ instead of the much better $O(1)$.

This would be much better done using a doubly linked list.
Note: Technically a queue is an abstract data type which is implemented using a data structure (such as an array or a linked list).

6.3 Doubly Linked list

The idea of a doubly linked list is that each node carries two pointers: One for the node in front of it, and another for the node in behind it.

In a doubly-linked list, we always have a header node in the very front and a trailer in the very back. These are called sentinels, that allow us to avoid having special cases on the head and tail nodes.

An empty doubly linked list has the header set to point to the trailer and the trailer point to the header. This is updated if a new node is added.

```
class _DoublyLinkedBase:
  class _Node:
    # A lightweight, non-public class for storing doubly linked
  node
    def __init__(self, element, prev, next):
      self._element = element
      self._prev = prev
      self._next = next

  def __init__(self):
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self._trailer
    self._trailer._prev = self._header
    self._size = 0

  def ___len__(self):
    return self._size

  def is_empty(self):
    return self_size == 0

  def _insert_between(self, e, predecessor, successor):
    newest = self._Node(e, predecessor, successor)
    predeccessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest

  def _delete_node(self, node):
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element
    node._prev = node._next = node._element = None
    return element
```

6.3.1 Doubly-Ended Queue

From this data structure, we can implement a **doubly-ended queue**. This is a queue that allows adding and removing from the front and the back.

```
class Linked DoublyEndedQueue(_DoublyLinkedBase):
  # Double eneded queue based on a doubly linked list

  def first(self):
    if self.is_empty():
      raise Exception("Queue empty")
    return self._header._next._element

  def last(self):
    if self.is_empty():
      raise Exception("Queue empty")
    return self._trailer._prev._element

  def insert_first(self, e):
    self._insert_between(e, self._header, self._header._next)

  def insert_last(self, e):
    self._insert_between(e, self._trailer, self._trailer._prev)

  def delete_first(self):
    if self.is_empty():
      raise Exception("Queue empty")
    return self._delete_node(self._header._next)

  def delete_last(self):
    if self.is_empty():
      raise Exception("Queue empty")
    return self._delete_node(self._trailer._prev)

```

6.4 Positional List

A positional list is another ADT (abstract data type). A positional list is a sequential container of elements that allows positional access. It is easy to assign a position in an array (the indices), but it is harder for a linked list.

We could try to assign indices to the nodes in order, but this is not a good solution as such a reference to a node would make it hard to delete or find the node itself.

We could also try to use the entire node as a positional reference, but this is also not good to do. The problem is that it complicates the use of the list as you would need to specify the next and previous nodes. This is also dangerous to hand out as they could change the next and previous nodes as they like. Finally, this also has a lack of encapsulation (we do not have the option to switch the underlying data structure if we like).

Instead, we will use a subclass that wraps a node in our positional list. An implementation is

```
class PositionalList(DoublyLinkedBase):
  # An implementation of a positional list
  class Position:
    # Abstraction representing the location of single element
    def __init__(self, container, node):
      self._container = container
      self._node = node

    def element(self):
      return self._node._element

    def __eq__(self, other):
      return type(other) is type(self) and other._node is self._node

  def _make_position(self, node):
    # return position instance for give node.
    if node is self._header or self._trailer:
      return None
    return self.Position(self, node)

  def before(self, p):
    node = self._validate(p) # converts a position back to a node if a validate position
    return self._make_position(node._prev)

  def add_after(self, p, e):
    original = self._validate(p)
    self._insert_between(e, original, original._next)

```

This is only some of the more important methods. The full code is available in the textbook.

6.5 Advantages and Disadvantages

The advantages of arrays compared to linked lists (LLs) are

- $O(1)$ time access to element based on index (e.g. e = A[i])
- $A.append$ is $O(1)$* whereas LLs are Node(1). It is a constant factor fast than LLs for $O(1)$ operations
- Less memory than LLs

The advantages of linked lists are

- Worst case time bound (O(1) appends always, not O(1)*)
- O(1) insertions and deletions at arbitrary locations (if you have the node)

7 Sorting Algorithms

We would like to have sorted arrays so that we can easily access the array (via some algorithm like binary search) faster. Another advantage is that it's easier to compare two sorted arrays than unsorted arrays. It is also easier to merge sorted arrays.

7.1 Selection Sort

In this method, we search for the smallest element in the array N times.

```
def slection_sort(arr):
  for i in range(len(arr)):
    smallest = inf
    smallest_index = 0
    for j in range(i, len(arr)):
      if arr[j] < smallest:
        smallest = arr[j]
        smallest_index = j
  temp = arr[i]
  arr[i] = smallest_index
  arr[smallest_index] = temp
```

The complexity of this algorithm is $O(n^2)$. The number of iterations is $(n-1)+(n-2)+...+1 = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} \approx \frac{1}{2}n^2$ for large n.

7.2 Insertion Sort

```
def insertion_sort(arr):
  for i in range(1, len(arr)):
    elem = arr[i]
    j=i-1
    while j > -1 and arr[j]>elem
      arr[j+1] = arr[j]
      j-=1
    arr[j+1] = elem
```

The time complexity for this algorithm is $O(n^2)$ in general. However, in the case of an already sorted list, the time complexity is $O(n)$. Selection sort does do this. The maximum number of swaps are n^2 here, whereas in the selection sort algorithm has only n swaps.

7.3 Divide and Conquer

Insertion sort is an incremental approach where we handle elements one at a time. Divide and conquer, however, is a recursive approach that:

- Breaks the problem down into sub-problems that are similar to the original than the original problem but smaller in size
- Combines solutions to sub-problems to solve original problem

This strategy has three steps

- Divide
- Conquer sub-problem by solving them recursively
- Combine solutions to sub-problems into overall solutions

A **recursive function** is a function that calls itself. An example of this is with the factorial: $n! = n(n-1)(n-2)...1$. We can do this with a for loop:

```
fact = 1
for i in range(1, 5):
  fact = fact*i
```

Recursively, this is

```
def factorial(n):
  if n==1:
    return 1
  else:
    return factorial(n-1)*n
```

7.4 Merge Sort

Merge sort follows the divide and conquer approach. The steps are

- Divide array to be sorted into two sub-sequences of size $\frac{n}{2}$ each
- Sort sub-sequences recursively using merge sort
- Merge the two sorted sub-sequences to produce the final sorted array

INSERT STUFF
The Python code for this is

```
def merge_sort(A, p, r):
  if p < r:
    q = (r-p)//2
    merge_sort(A, p, q)
    merge_sort(A, q+1, r)
    merge(A, p, q, r)

```

```
8   def merge(A, p, q, r):
9     n1 = q-p + 1
10    n2 = r-q
11    L=[]
12    R=[]
13    for i in range(n1):
14      L.append(arr[p+i])
15    for i in range(n2):
16      R.append(arr[q+i+1])
17    L.append(inf)
18    R.append(inf)
19
20    i, j = 0, 0
21    for k in range(p, r+1):
22      if L[i] <= R[j]:
23        A[k] = L[i]
24        i+=1
25      else:
26        A[k] = R[j]
27        j+=1
```

7.4.1 Analysis of Merge Sort

The time complexity of this algorithm is $O(n \log n)$. The $\log n$ comes from the splitting of the arrays in half each time and the n comes from the merge operations. We will do this more concretely now.

Assume n is a power of 2. Each divide step yields sub-sequences of size exactly $\frac{n}{2}$. The divide step is $O(1)$.

In the conquer step, we recursively solve for two sub problems each of size $\frac{n}{2}$. This will contribute $2T(\frac{n}{2})$.

In the combine step, we merge n-element sub-array which is $O(n)$.

We now that $O(n)+O(1) = O(n)$. So, $T(n) = O(1) \text{if} n = 1; 2T(\frac{n}{2}) + O(n) \text{if} n > 1$. So, $T(n) = c \text{if} n = 1; 2T(\frac{n}{2}) + cn \text{if} n > 1$, where c represents the time required to solve problems of size 1.

Thus, it takes cn time for each merge sort call. We call merge sort $\log n$ times, so the final time complexity is $O(n \log n)$.

7.5 Quick Sort

Quick sort is a divide and conquer algorithm. In the worst case, it is $O(n^2)$ and is $O(n \log n)$ on average.

In the *divide step*, it partitions the array A[p..r] into two possible empty subarrays A[p..q-1], A[q+1..r] such that each element of A[p..q-1] is less than or equal to A[q], which is in turn less than or equal to A[q+1..r].

If we have the array [9, 6, 3, 1, 5], the partition step would result in [[1], 3, [3, 5, 6, 9]].

In the *conquer step*, the two subarrays are sorted by recursive calls to quick sort.

There is no combine step needed.

In code this is

```
1   def quicksort(A, p, r):
2     if p < r:
3       q = partition(A, p, r)
4       quicksort(A, p, q)
5       quicksort(A, q+1, r)
6
7   def partition(A, p, r):
8     x = A[r]      # pivot element
9     i = p-1
10    for j in range(p, r):
11      if A[j] <= x:
12        i += 1
13        temp = A[i]
14        A[i] = A[j]
15        A[j] = temp
16    temp = A[r]
17    A[r] = A[i+1]
18    A[i+1] = temp
19    return i
```

For example, consider the array [2, 8, 7, 1, 3, 5, 6, 4]. First, we start with p = 0 and r = 7.

x = A[7] = 4
i = 0 -1 = -1
j = 0
A[0] $\leq$ 4? Yes $\implies$ i = -1+1 = 0, swap $\implies$ the array stays the same.
j = 1
A[1] $\leq$ 4? No, loop
j=2
A[2] $\leq$ 4? No, loop
j=3
A[3] $\leq$ 4? Yes, i = 0+1 = 1, swap $\implies$ A = [2, 1, 7, 8, 3, 5, 6, 4]
j=4
A[4] $\leq$ 4? Yes, i = 1+1 = 2, swap $\implies$ A = [2, 1, 3, 8, 7, 5, 6, 4]
j=5
A[5] $\leq$ 5? No, loop

j=6
A[6] $\leq$ 6? No, loop
j=7
A[7] $\leq$ 4? Yes, i = 2+1 = 3, swap $\implies$ A = [2, 1, 3, 4, 7, 5, 6, 8]

Everything less than i is positioned left of the pivot and everything right of i is greater than the pivot.

The worst case for time complexity occurs when the array is already sorted. The number of iterations is $1+2+...+n = O(n^2)$. This is caused by bad partitioning.

Randomized quick sort solves this by adding at the beginning of the partition

```
1  i = random(p, r)
2  swap(A[r], A[i])
```

In the best case quick sort is has $T(n) = 2T(\frac{n}{2}) + O(n)$, which is done $\log n$ times, so the algorithm is $O(n \log n)$.

7.6 Lower bound of comparison sorting

We will show that $\Omega(n \log n)$.

We can show this by showing the decision trees for such algorithms. These are binary trees that tell you what to do when stepping through the algorithm.

Let $l = [a_1 = 6, a_2 = 8, a_3 = 5]$. Using insertion sort, the first comparison compares $6 > 8$, then we compare $5 \leq 8$, and finally $6 > 5$. We can construct a decision tree using these three comparisons.

There are $3! = 6$ permutations of leaves in the decision tree here.

The length of the longest path from the root to any reachable leaf represents the worst-case number of comparisons that the sorting algorithm perform.

Theorem: Any comparison-based sort requires $\Omega(n \log n)$ comparisons in worst case.

Proof: Consider a decision tree of height h with l reachable leaves corresponding to comparison sort of n elements because of $n!$ permutations appears as some leaf we have $n! < l$, since a binary tree of height h has no more than 2^h leaves, we have $n! \leq l \leq 2^n \implies h \geq \log(n!) \implies \Omega(n \log n)$.

7.7 Counting Sort

Counting sort is an $O(n)$, non-comparison sort.

Assume the input is $A[1..n]$ of integers.

For each element x, count the number of element less than x and then use that info to place x directly in output.

For example, if we ahve x and count 17 elements that are less than x, then x should go to index 18.

In python,

```
def counting_sort(A, B, k):
  # A is the input list, B is the output list
  # k is the max element of the list +1

  C = [0]*k
  for j in range(len(A)): # store counts in C
    C[A[j]] = C[A[j]] + 1
  for j in range(1, k):   # store cummulative count in C
    C[j] = C[j] + C[j-1]
  for j in range(len(A) - 1, -1, -1): # store in output
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
```

This algorithm is really $O(n + k)$. As a whole, this algorithm is only $O(n)$ if $k = O(n)$.

7.8 Radix sort

This allows us to use counting sort on large numbers. In psudeocode,

```
def radix_rot(A, d):
  for i in range(d):
    counting_sort(A) with digit i
```

The key to this is that counting sort is a **stable sort**. This means that elements that have the same value in the input, will continue to be the same order as in the output of the sort.

8 Trees

A tree is an ADT that stores elements hierarchically. **Parent** nodes have **children** nodes. The top node in the tree is called the **root** and the bottom-most nodes in the tree are the **leaves**.

A tree T is a set of nodes, each node storing elements such that the nodes have a parent-child relationship, which satisfies the following properties:

- If T is non-empty, then it has a root
- Each node V of T different from the root has a unique parent node W, and every node with a parent W is a child W

Two nodes that have the same parent are called **sibilings**. The leaves are also called **external** nodes (any node that do not have children). Any node with children nodes are **internal** nodes.

A tree could, for instance, represent a file-system of directories.

An **edge** of a tree is a pair of nodes such that one of them is the parent of the other. A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.

A tree is **ordered** if there is a meaningful order among the children of each node.

Let p be the position of a node in a tree T, then the **depth** of p is the number of ancestors of p, excluding itself.

The height of position p in tree T is defined recursively as

- If p is a leaf, then the height is 0.
- Otherwise, the height of p is one more than the maximum of the heights of the children of p.

We will now implement an abstract base class for a tree in Python.

```
1  class Tree:
2    """Abstract base class representing a tree structure."""
3
4    class Position:
5      """An abstraction representing the location of a single
     element in the tree."""
6
7      def element(self):
8        """Returns the element stored at this position."""
9        raise NotImplementedError
```

```
10
11        def __eq__(self, other):
12          """Returns true if other position represents the same
    location"""
13          raise NotImplementedError
14
15        def __ne__(self, other):
16          """Returns true if other position represents a different
    location"""
17          return not (self == other)
18
19      def root(self):
20        """ Gets the root of the tree."""
21        raise NotImplementedError
22
23      def parent(self, p):
24        """ Gets the parent of p."""
25        raise NotImplementedError
26
27      def num_children(self, p):
28        """ Gets the number of children of p."""
29        raise NotImplementedError
30
31      def children(self, p):
32        """ A generator for positions representing position p's
    children."""
33        raise NotImplementedError
34
35      def __len__(self):
36        """Gets the number of elements in the tree."""
37        raise NotImplementedError
38
39      def is_root(self, p):
40        """Returns true if p is the root."""
41        return self.root() == p
42
43      def is_leaf(self, p):
44        """Returns true if p is a leaf."""
45        return self.num_children(p) == 0
46
47      def is_empty(self):
48        return len(self) == 0
49
50      def depth(self, p):
51        """Returns the number of levels separating p from the root.
    """
52        if self.is_root(p):
53          return 0
54        else:
55          return 1 + self.depth(self.parent(p))
56
57      def _height1(self, p):
58        """Returns the height of the tree. This is bottom-up approach
    : O(n^2) worst-case."""
59        return max(self.depth(p) for p in self.positions() if self.
    is_leaf())
60
```

```
61      def _height2(self, p):
62        """Returns the height of the subtree rooted at position p.
        This is top-down approach: O(n) worst-case."""
63        if self.is_leaf(p):
64          return 0
65        else:
66          return 1 + max(self._height2(p) for c in self.children())
67
68      def height(self, p=None):
69        """Returns the height of the subtree rooted at position p."""
70        if p is None:
71          p = self.root()
72        return self._height2(p)
```

8.1 Binary Trees

A binary tree is an ordered tree with the properties

- Every node has at most 2 children
- Each node is labeled as either a left or a right child
- The left child proceeds the right child in the order of children

For example, decision trees are binary trees (yes or no).

We can also express an arithmetic expression as a binary tree. In that case, we would place each operation as an internal node and each number as a external node.

We can write a binary tree in Python.

```
1   class BinaryTree(Tree):
2     """Abstract base class representing a binary tree structure."""
3
4     def left(self, p):
5       """Returns a postion representing p's left child."""
6       raise NotImplementedError
7
8     def right(self, p):
9       """Returns a postion representing p's right child."""
10      raise NotImplementedError
11
12    def sibiling(self, p):
13      """Returns a postion representing p's sibiling."""
14      parent = self.parent(p):
15      if parent is None:
16        return None
17      else:
18        if p == self.left(parent)
19          return self.right(parent)
```

```
20          else:
21            return self.left(parent)
22
23      def children(self, p):
24        """ A generator for positions representing position p's
        children."""
25        if self.left(p) is not None:
26          yield self.left(p)
27        if self.right(p) is not None:
28          yield self.right(p)
29
```

Let T be a non-empty binary tree and let n, n_E, n_I and h be the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then, T has the properties:

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq n - 1$

T is a **proper** tree if T has no nodes with only one leaf. If T is a proper tree, then it has the properties:

- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq \frac{1}{2}(n - 1)$

A **linked binary tree** is a tree which contains 4 pointers. The pointers are to the left, right and parent nodes, as well as the element. The leaves all have their children pointers set to null, and the root has its parent set to null.

In python, it is

```
1   class LinkedBinaryTree(BinaryTree):
2     """ Linked representation of a binary tree structure."""
3
4     class _Node:
5       """ A lightweight nonpublic class reperesenting a node"""
6       __slots = "_element", "_parent", "_left", "_right"
7
8       def __init__(self, element, parent=None, left=None, right=
        None):
9         self._element = element
10        self._parent = parent
```

```
11          self._left = left
12          self._right = right
13
14      class Position(BinaryTree.Position):
15        """An abstraction reperesenting the location of a single
        element."""
16
17        def __init__(self, container, node):
18          self._container = container
19          self._node = node
20
21        def element(self):
22          """Returns the element stored at this position."""
23          return self._element
24
25        def __eq__(self, other):
26          """Return true if the other is a position representing the
        same location in the tree."""
27          return type(other) is type(self) and other._node is self.
        _node
28
29      def _validate(self, p):
30        """ Return associated node if position is valid."""
31        if not isinstance(p, self.Position):
32          raise TypeError("p must be proper position type.")
33        if p._container is not self:
34          raise ValueError("p does not belong to this container.")
35        if p._node._parent is p._node: # convention for deprecated
        nodes
36          raise ValueError("p is no longer valid.")
37
38        return p._node
39
40      def _make_position(self, node):
41        """Return the position instance for a given node or None if
        there is no Node."""
42        return self.Position(self, node) if node is not None else
        None
43
44      def __init__(self):
45        """Creates an initially empty binary tree"""
46        self._root = None
47        self._size = 0
48
49      def __len__(self):
50        return self._size
51
52      def root(self):
53        return self._make_position(self._root)
54
55      def parent(self, p):
56        """Returns the position of ps parent"""
57        node = self._validate(p)
58        return self._make_position(node._parent)
59
60      def left(self, p):
61        """Returns the position of p's left child."""
```

```
62        node = self._validate(p)
63        return self._make_position(node._left)
64
65      def right(self, p):
66        """Returns the position of p's right child."""
67        node = self._validate(p)
68        return self._make_position(node._right)
69
70      def num_children(self, p):
71        """Returns the number of children for a position p."""
72        node = self._validate(p)
73        count = 0
74        if node.left is not None:
75          count += 1
76        if node.right is not None:
77          count += 1
78        return count
79
80      def _add_root(self, e):
81        """ Replaces an element e at the root of an empty tree and
    returns a new position.
82          Raises a ValueError if tree is nonempty.
83        """
84        if self._root is not None: raise ValueError("A root already
    exists.")
85        self._size = 1
86        self._root = self._Node(e)
87        return self._make_position(self._root)
88
89      def _add_left(self, e):
90        """ Creates a new left child for position p storing element e
    ."""
91        node = self._validate(p)
92        if node.left is not None: raise ValueError("Left child
    already exists.")
93        self._size += 1
94        node._left = self._Node(e, parent=node)
95        return self._make_position(node._left)
96
97      def _add_right(self, e):
98        """ Creates a new right child for position p storing element
    e."""
99        node = self._validate(p)
100       if node.right is not None: raise ValueError("Right child
    already exists.")
101       self._size += 1
102       node._right = self._Node(e, parent=node)
103       return self._make_position(node._right)
104
105     def _replace(self, p, e):
106       """Replaces the lement at position p with e and returns the
    old element"""
107       node = self._validate(p)
108       old = node._element
109       node._element = e
110       return old
111
```

```
112     def _delete(self, p):
113       """Deletes the node at position p and replaces it with child
        if it has one. Returns the element that had been stored at p
        and raises a ValueError if p is invalid or p has two children.
        """
114       node = self._validate(p)
115       if self.num_children(p) == 2: raise ValueError("p has two
        children")
116       child = node._left if node._left else node._right
117       if child is not None:
118         child._parent = node._parent
119       if node is self._root:
120         self._root = child
121       else:
122         parent = node._parent
123         if node is parent._left:
124           parent._left = child
125         else:
126           parent._right = child
127       self._size -= 1
128       node._parent = node
129       return node._element
130
131     def _attach(self, p, t1, t2):
132       """Attaches trees t1 and t2 as left and right subtrees of
        external p."""
133       node = self._validate(p)
134       if  not self.is_leaf(p): raise ValueError("position must be a
         leaf")
135       if not type(self) is type(t1) is type(t2): raise TypeError("
        Tree types must match")
136       self._size += len(t1) + len(t2)
137       if not t1.is_empty():
138         t1._root._parent = node
139         node._left = t1._root
140         t1._root = None
141         t1._size = 0
142       if not t2.is_empty():
143         t2._root._parent = node
144         node._left = t2._root
145         t2._root = None
146         t2._size = 0
147
```

8.2 Tree Traversals

The three types of tree traversals we will look at are preorder, postorder, and inorder. They all take a tree and a starting position.

For the preorder case, the psuedocode is

```
1   def preorder(T, p):
2     visit p
3     for each child c of p:
4       preorder(T, c)
```

For the postorder case, the psuedocode is

```
def postorder(T, p):
  for each child c of p:
    postorder(T, c)
  visit p
```

For the inorder case, the psuedocode is

```
def inorder(T, p):
  if p has left child lc:
    inorder(T, lc)
  visit p
  if p has right child rc:
    inorder(T, rc)
```

The preorder, postorder, and inorder tree traversals correspond to prefix, postfix, and infix notations for algebraic expression trees.

There is also the breadth-first and depth-first traversals.

The breadth-first algorithm psuedocode is

```
def breadthfirst(T):
  initialize queue Q to contain T.root
  while Q not empty:
    p = Q.deque()
    visit p
    for each child c of p:
      Q.enqueue(c)
```

8.3 Binary Search Tree

Let x be a node in a binary search tree (BST), if y is a node in the left sub-tree of x, then y.key $\leq$ x.key. Also, if y is a node in the right sub-tree of x, then y.key $\geq$ x. This is the basic property that the binary search tree has in order to be a binary search tree.

If we want to print the sorted list representation of the tree, we can use the inorder sorting algorithm. This allows us to access the array in $O(n)$ time and to sort the array in $O(n \log n)$.

To search through a binary search tree for a node x with a desired value k, we will use the algorithm

```
def treesearch(x, k):
  if x == None or k == x.key:
    return x
  if k < x.key:
    return tree_search(x.left, k)
  else:
    return tree_search(x.right, k)
```

This is a recursive search. If we want to use iteration instead, we can use

```
def interative_treesearch(x, k):
  while x != None or k != x.key:
    if k < x.key:
      x = x.left
    else:
      x.right
```

To find the minimum of a tree, we can use

```
def treemin(x):
  while x.left != None:
    x = x.left
  return x
```

And the max is found with

```
def treemax(x):
  while x.right != None:
    x = x.right
  return x
```

The algorithm to find the successor of a node is

```
def treemax(x):
  if x.right != None:
    return treemin(x.right)
  y = x.parent
  while y != None and x == y.right:
    x = y
    y = y.parent
  return y
```

To insert a new element z in a tree T, we use

```
def treeinsert(T, z):
  y = None
  x = T.root
  while x != None:
    y = x
    if z.key < x.key:
      x = x.left
    else:
      x = x.right
  z.parent = y
  if y == NoneL
    T.root = z
  elif z.key < y.key:
    y.left = z
  else:
    y.right = z
```

The algorithm for deletion of an element z has a few cases:

1. If z has no children, we simply remove by modifying the parents child.
2. If z has only one child, we elevate that child to take z's position in the tree by replacing modifying z's parent to replace z by z's child
3. If z has two children, we find z's successor y, which must be in z's right sub-tree, and have y take zs position in the tree.

The algorithm is

```
def treedelete(T, z):
  if z.left == None:
    transplant(T, z, z.right)
  elif z.right == None:
    transplant(T, z, z.left)
  else:
    y = treemin(z.right)
    if y.parent != z:
      transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
    transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
```

where

```
def transplant(T, u, v):
  if u.parent == None:    # if u is root
    T.root = v
  elif u == u.parent.left: # if it is a left child
    u.parent.left = v
  else:                    # if it is a right child
    u.parent.right = v
  if v != None:
    v.parent = u.parent
```

Such a tree can be used to construct a set or dictionary.

9 Priority Queue

A priority queue is an ADT that tries to optimize finding the minimum value. This minimum value has the highest priority. The basic methods we use are

- P.add(k, v)
- P.min()
- P.remove_min()
- P.is_empty()
- len(P)

We will look at the Priority Queue with different underlying data structures.

The base code in Python is

```
class PriorityQueueBase:
  """Abstract base class fro priority queue"""
  class _Item:
    """lightweight composite to store priority queue items."""
    __slots__ = "_key", "_value"

    def __init__(self, k, v):
      self._key = k
      self._value

    def __lt__(self, other):
      return self._key < other._key

  def is_empty(self):
    return len(self) == 0

```

9.1 Using an Unsorted Array

The time complexity table for an unsorted array implementation is

Operation	Running Time
P.min()	$O(n)$
P.remove_min()	$O(n)$
P.add(k, v)	$O(1)*$
P.is_empty()	$O(1)$
len(P)	$O(1)$

9.2 Using a Sorted Array

The time complexity table for a sorted array implementation is

Operation	Running Time
P.min()	$O(1)$
P.remove_min()	$O(n)$
P.add(k, v)	$O(n)$
P.is_empty()	$O(1)$
len(P)	$O(1)$

9.3 Using a Binary Search Tree

The time complexity table for a binary search tree (BST) implementation is

Operation	Running Time
P.min()	$O(\log n)*$
P.remove_min()	$O(\log n)*$
P.add(k, v)	$O(\log n)*$
P.is_empty()	$O(1)$
len(P)	$O(1)$

* corresponds to the case where the BST is balanced.

9.4 Using a Heap

The time complexity table for a heap implementation is

Operation	Running Time
P.min()	$O(1)$
P.remove_min()	$O(\log n)$
P.add(k, v)	$O(\log n)$
P.is_empty()	$O(1)$
len(P)	$O(1)$

The heap is defined by the heap-order property: In a heap T, for every position p, the key stored is $\geq$ the key stored at p's parent.

The heap is a *complete binary tree*. This means that levels 0, 1, 2, ..., h - 1 of T have the maximum number of nodes possible and the remaining nodes at level h reside in the leftmost positions in that level. We can therefore guarantee that $h = \log n$.

To add an element to a heap, we do a "bubble up" operation, where we do swaps until it satisfies the property condition of order $O(\log n)$.

If we want to remove the minimum (top of the heap), then we remove the last element in the heap and replace the top element of the heap with it, and then "bubble down" to reach the property condition order. The maximum number of

swaps is $O(\log n)$.

To sort an array using a heap, we can easily create the heap in $O(n \log n)$ time. We can then remove the minimum n times to sort the array in $O(n \log n)$ time.

We can actually construct a heap in $O(n)$ time using a bottom up heap construction.

To do this, we start by creating a heap for half of the items. Then, we join the heaps with a next layer of heaps constructed from half of the remaining items. We then do bubble down operations as needed to meet the heap order condition. This is repeated until there are no more items to add.

Of course, it takes $O(n)$ time to do the adding of the items to the heap. We must prove the proposition that the bottom-up construction of a heap with n entries take $O(n)$ time. The key is that if we sum over all the nodes from each path that are traced by each of the bottom up operations, we obtain n. It is because the paths are disjoint. Thus, this construction is $O(n)$.

9.4.1 Implementation

```
1   class HeapPriorityQueue(PriorityQueueBase):
2     """A min-oriented priority queue implemented with a binary heap
      """
3
4     def _parent(self, j):
5       return (j-1) //2
6
7     def _left(self, j):
8       return 2*j + 1
9
10    def _right(self, j):
11      return 2*j + 2
12
13    def _has_left(self, j):
14      return self._left(j) < len(self._data)
15
16    def _has_right(self, j):
17      return self._right(j) < len(self._data)
18
19    def _swap(self, i, j):
20      """ swap the elements at indices i and j of the array"""
21      self._data[i], self._data[j] = self._data[j], self._data[i]
22
23    def _upheap(self, j)
24      """ bubble up operation (called when adding an item) """
25      parent = self._parent()
26      if j > 0 and self._data[j] < self._data[parent]:
27        self._swap(j, parent)
28        self._upheap(parent)
```

```
29
30      def _downheap(self, j):
31        """ bubble down operation (called when removing an item) """
32        if self._has_left(j):
33          left = self._left(j)
34          small_child = left
35          if self._has_right(j):
36            right = self._right(j)
37            if self._data[right] < self._data[left]:
38              small_child = right
39          if self._data[small_child] < self._data[j]:
40            self._swap(j, small_child)
41            self._downheap(small_child)
42
43      def __init__(self):
44        self._data = []
45
46      def __len__(self):
47        return len(self._data)
48
49      def add(self, key, value):
50        self._data.append(self._Item(key, value))
51        self._upheap(len(self._data) - 1)
52
53      def min(self):
54        """ Returns but does not remove (k, v) tuple with the minimum
         key"""
55        if self.is_empty():
56          raise Exception("Queue is empty")
57        item = self._data[0]
58        return (item._key, item._value)
59
60      def remove_min(self):
61        """Removes the min and returns the (k, v) typle with minimum
        key."""
62        if self.is_empty()
63          raise Exception("The heap is empty")
64
65        self._swap(0, len(self._data) -1)
66        item = self._data.pop()
67        self._downheap(0)
68        return (item._key, item._value)
69
```

10 Maps

In python, the built-in dictionary uses a map ADT. The data structure is a hash table.

The operations we can perform on the map are

- M[k] $\rightarrow$ returns the value v in the map

- M[k] = v → associate value v with key k in the map
- del M[k] → deletes the value at that key in the map
- len(M) → size of the map
- iter(M) → creates an iterative instance of the map

We could implement this in a BST and get $O(\log n)$ complexity for M[k] and M[k] = v operations. However, there is a way to do this faster. We can use a has table to do this. Together, this is called a hashmap.

10.1 Hash Table

A hash map is a data structure that uses a **hash function** which converts an arbitrary object (typically a string) into a integer using a **hash code** and then, uses a **compression function** to compress it down to the size of the map we want. This function is deterministic, since we do not want randomness in the function which would make it hard to repeat.

If we had the hash function was the modulus operator of the length of the map. We could, for instance, map any number from 0 to 9. However, the function is not unique for 10, 20, 30, etc. Instead, we can use the modulus of a prime length (e.g. 11).

Why prime? Any integer that shares a common factor with the table size will be hashed into an index that is a multiple of this factor.

Some c++ code for this is implementation of a hash function is

```
int hash(string key, int tablesize) {
  int hashval = 0;
  for (char ch : key)
    hashval += ch;
  return hashval% tablesize;
}
```

This is not a good has function, however. If we have a large tablesize (eg. 10,007) and keys that are 8 characters or less. The maximum integer of an integer is 127. Therefore, hashval has a max size of 1016. This wastes lots of space in the table. We need to find a way of expanding the hash function to use all of the table efficiently. We could try multiplying by a large value:

```
int hash(string key, int tablesize) {
  return (key[0] + 27*key[1] + 729*key[2]) % tablesize
}
```

However, there is yet another problem. This is because in the English language, it is not common to use "zxq" as a key word, but "cat" is much more likely. There will be many blanks in the table. Next, we could instead do

```
int hash(string key, int tablesize) {
  unsigned int hashval = 0;
  for (char ch: key)
    hashVal = 37*hashval + ch;
  return hashval/tablesize;
}
```

Once again, there is a problem. The problem comes from the fact that very similar words will have very different hashvals. Here, 37 is used simply because it is a good number with few collisions found by experiment.

For the hash function, we would like

- deterministic
- two keys close in value to hash to very different values
- a uniform distribution (evenly)
- minimizes collisions

However, there will always be collisions. There are a few ways to combat these remaining collisions.

Separate chaining is done by simply using a linked list as the elements of the table so that it can have multiple values at a particular hash table key. This is only good if it is possible to keep this linked list very short, so a lot of time is not spent in its traversal.

Linear probing starts with an empty table and a value in inserted into the table via the hash function. If we get a collision, we probe the rest of the table for an empty spot (immediately after the current element), and then inserts it there. The index used is $f(i) = i$ (hence, linear). The issue with this method is that it generates clustering of elements in table, which make it hard to add further elements.

A **load factor** λ tells us the fraction of data that is in the array to the amount of empty spots in the table. When inserting using linear probing, the expected number of probes is $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$. If the table is half full ($\lambda = 0.5$), there is an average of 2.5 probes, however, if is nine tenths full ($\lambda = 0.9$), then we have around 50 probes.

Quadratic probing uses a index of $f(i) = i^2$ instead (hence the quadratic). Using this method, on the second probe, we increase by an index of $2^2 = 4$.

This eliminates the primary clustering from linear probing. There is, however, secondary clustering, where we continue to jump over the same elements over and over. The disadvantage is that we might never fill the . If $\lambda = 1/2$ and the table size is prime, then quadratic is guaranteed to find an empty spot.

We can code a hash table in Python as follows.

```
from collections.abc import MutableMapping
from random import randrange

class MapBase(MutableMapping):
  """ Our abstract bas class that includes a nonpublic _Item
  class."""
  raise NotImplementedError


class HashMapBase(MapBase):
  """ Abstract base class for a map using a hadh table with
  multiply and divide (MAD) compression."""
  def __init__(self, cap=11, p=109345121):
    """ create empty hash table map"""
    self._table = cap * [None]
    self._n = 0
    self._prime = p
    self._scale = 1 + randrange(p - 1)
    self._shift = randrange(p)

  def _hash_function(self, k):
    return ((hash(k))*self._scale + self_shift) % self._prime %
  len(self._table)

  def __len__(self):
    return self._n

  def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)

  def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)
    if self._n > len(self._table) // 2:
      self._resize(2* len(self._table) - 1)

  def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)
    self._n -= 1

  def _resize(self, c):
    old = list(self.items())
    self._table = c * [None]
    self._n = c
    for (k, v) in old:
      self[k] = v

```

Separate chaining can be implemented with

```
class UnsortedTableMap(MapBase):
  raise NotImplementedError

class ChainHashMap(HashMapBase):
  """hash map implemented with separate chaining for collision
  resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError("Key Error")
    return bucket[k]

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap()
    oldsize = len(self._table[j])
    self._table[j][k] = v

  def __bucket_delitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError("Key Error")
    del bucket[k]

```

We can implement linear probing with

```
class ProbeHashMap(HashMapBase):
"""hash map implemented with linear probing for collision
  resolution."""

_AVAIL = object()

def _is_available(self, j):
  """ return true if index j is available in the table"""
  return self.table[j] is None or self._table[j] is ProbeHashMap.
  _AVAIL

def _find_slot(self, j, k):
  """
  Searches for the key k in bucket at index j
  Returns (success, index) tuple described as follows
  if match was found, sucess=True and index denotes its location
  if no match was found, sucess=False, and index denotes the
  first available slot
  """

  firstAvail = None
  while True: # probe until slot is available
    if self._is_available(j):
      if firstAvail is None:
        firstAvail = j
      if self._table[j] is None:
        return (False, firstAvail)
```

```
25        elif k == self._table[j]._key:
26          return (True, j)
27
28        j = (j+1) % len(self._table)
29
30    def _bucket_getitem(self, j, k):
31      found, s = self._find_slot(j, k)
32      if not found:
33        raise KeyError("Key Error")
34      return self._table[s]._value
35
36    def _bucket_setitem(self, j, k, v):
37      found, s = self._find_slot(j, k)
38      if not found:
39        self._table[s] = self._Item(k, v)
40        self._n += 1
41      else:
42        self._table[s]._value = s
43
44
45    def __bucket_delitem(self, j, k):
46      found, s = self._find_slot(j, k)
47      if not found:
48        raise KeyError("Key Error")
49      else:
50        self._table[s] = ProbeHashMap._AVAIL
51
```

10.2 Ordered Map

For the BST, it takes $O(\log n)$ time to insert into the tree. Whereas the hash table takes $O(1)$ time. Can we build an ordered map using $O(1)$ time insertions? The answer is no. It is because it would imply that would mean that it would take $O(n)$ time to sort a map, which was proven to not be true earlier.

10.2.1 Skip List

A skip list is a way to do a binary search on a linked list than looking for each element. We can traverse it in $O(\log n)$ as opposed to $O(n)$. A skip list consists of a series of lists $S_0, S_1, ..., S_h$ each storing a subset of items M, sorted by increasing key, plus sentinels $-\infty$ and $+\infty$.

The lowest level of the skip list has all the elements and the next levels have less and less levels. Each node has pointers to the lower and to the upper list. To generate a level, we flip a coin and if it is heads, we insert the node again, otherwise, we do not. The skip list has a pointer to the top, first element.

Let's say we want to find an item with key 44, we could use psuedocode

```
S.find(44)
item = S[44]
def SkipSearch(k):
  p = start
  while below(p) is not None:
    p = below(p)
    while k >= key(next(p)):
      p = next(p)
  return p
```

To insert into the skip list we can do

```
def SkipInsert(k, v):
  p = SkipSearch(k) # bottom level item with the largest key <= k
  q = None
  i = -1
  if (key(p) == k):
    #overwrite value with v
  else:
    #create new node after p and before p.next

  repeat:
    i = i+1
    if i >= h:
      h = h+1
      t = next(S)
      S = insertAfterAbove(None, S, (-infinity, None))
      insetAfterAbove(S, t, (+infinity, None))

    while above(p) is None:
      p = prev(p)

    p = above(p)
    q = insertAfterAbove(p, q, (k, v))
  until coinflip() == tails
  n = n+1
  return q
```

10.3 AVL tree

An **AVL** tree is a binary search tree (BST) with a balance condition. This guarantees that the tree's height is $O(\log n)$.

We could enforce this balance condition in a few different ways.
For instance, consider a tree where the left and the right child subtrees of the root has the same height. This is not the best and we could still get $O(n^2)$ for its height.

We could also consider the cases where every subtree has the same amount of elements in its left and right subtrees. This will have a height of $O(\log n)$, but it is hard to insert into. This is too strict of a condition.

The condition that we will use instead is the the *left and right subtrees can differ by at most 1.*

There are 4 methods of insertion that will destroy our condition. They are

1. An insertion into left subtree of left child of α
2. An insertion into right subtree of left child of α
3. An insertion into left subtree of right child of α
4. An insertion into right subtree of right child of α

Where α is the node which is the root of the subtree that fails the condition.

Case 1 and 4 can be solved with a single rotation. That is to say, we replace the subtree with α and then make α the child (left or right) of the new tree. The C++ code for case 4 is

```
1  void rotateWithRightChild(AvlNode *&k1) {
2    AvlNode *k2 = k1 -> right;
3    k1 -> right = k2 -> left;
4    k2 -> left = k2;
5    updateheight(k2);
6    updateheight(k1);
7    k1 = k2;
8  }
```

For case 1 it is

```
1  void rotateWithLeftChild(AvlNode *&k2) {
2    AvlNode *k1 = k2 -> left;
3    k2 -> left = k1 -> right;
4    k1 -> right = k2;
5    updateheight(k1);
6    updateheight(k2);
7    k2 = k1;
8  }
```

For case 2 and 3, we need to do a double rotation. This is two single rotations. The C++ code for case 3 is

```
1  void doubleWithRightChild(AvlNode *&k1) {
2    rotateWithLeftChild(k1 -> right);
3    rotateWithRightChild(k1);
4  }
```

The code to actually insert a node in an AVL tree is

```
void insert(const Comparable &x, AvlNode *&t) {
  if (t == nullptr){
    t = new AvlNode(x, nullptr, nullptr)
  }
  else if (x < t->element) {
    insert(x, t->left);
  }
  else if (t->element < x) {
    insert(x, t->right);
  }
  balance(t);
}
```

Where the we balance the tree with

```
void balance(AvlNode *&t) {
  if (t == nullptr){
    return;
  }
  if (height(t->left) - height(t -> right) > 1) {
    if (height(t->left->left) >= height(t->left->right)) {  //
  case 1
      rotateWithLeftChild(t);
    }
    else{
      doubleWithLeftChild(t);
    }
  }
  if (height(t->right) - height(t -> left) > 1) {
    if (height(t->right->left) >= height(t->right->right)) {  //
  case 1
      rotateWithRightChild(t);
    }
    else{
      doubleWithRightChild(t);
    }
  }
}
```

11 Graphs

A **graph** G consists of a set of *vertices* V and a set of *edges* E. Each edge is a pair of vertices (v, w), where $v, w \in V$.

Vertex w is *adjacent* to vertex v if and only if $(v, w) \in E$ (there is a edge between them).

A *path* in a graph is a sequence of vertices $w_1, w_2, ..., w_n$ such that $(w_i, w_{i+1}) \in E \ \forall i = 1, 2, ..., n$.

A *directed graph* has directed edges (ordered parings pf vertices). A DAG (directed .. ..) is a directed graph that has no cycles.

An adjacency matrix is matrix containing a true if the vertex pair corresponding to that row and column is connected, as follows.

	T	T	T			
			T	T		
					T	
					T	T
			T			T
					T	

Summing the number of trues along a row/column gives you the number of nodes going in or out of the node. A more compact way to store a graph is by storing a list of connected elements for each node in the graph.

11.1 Topological Sort

A topological sort produces ordering of vertices in a DAG such that if there is a path from v_i to v_j, then v_i appears after v_j in ordering.

The *indegree* of vertex v is the number of edges (u, v) (number of vertexes going into the node).

The psudeocode for this algorithm is

```
1  topsort () {
2    for (int counter=0; counter<NUM_VERTS;counter++}{
3      vertex v = findVertexOfIndegreeZero();
4      v.topNum = counter;
5      for (each vertex w adjacent to v){
6        w.indegree--;
7      }
8    }
9  }
```

This implementation is $O(V^2)$ where V is the number of vertices. A more efficient implementation is

```
1  topsort(){
2    Queue(vertex) q;
```

```
    int counter = 0;
    for each vertex v:
      if (v.indegree == 0)
        q.enque(v);
    while not q.isempty():
      vertex v = q.dequeue();
      v.topNum = ++counter;
      for each vertex w adjacent to v:
        if --w.indegree == 0:
          q.enqueue(w);
  }
```

This implementation is $O(|V| + |E|)$ (number of vertexes + number of edges).

11.2 Shortest Path Algorithm

11.2.1 Single source shortest path

The **single source shortest path** works as follows. Given the input G = (V, E) weighted graph (a graph with weights on the edges) and a distinguished vertex S, find the shortest weighted path from S to all other vertices in V.

The algorithm for an unweighted graph shortest path is

```
unweighted(vertex s){
  for each vertex v:
    v.dist = infinity
    v.known = false
  s.dist = 0
  for (in currdist=0; currdist<Num_Vertices; currdist++):
    for each vertex v :
      if (not v.known and v.dist = currdist):
        v.known = true;
        for each vertex w adjacent to v:
          if (w.dist == infinity):
            w.dist = currdist;
            w.path = v;
```

This implementation is $O(|V|^2)$. A better unweighted implementation is

```
unweighted(vertex s):
  Queue<vertex> q;  # contains the cnddidates to be known
  for each vertex v:
    v.dist = infinity;
  s.dist = 0;
  q.enqueue(s);
  while(!v,is_empty()):
    Vertex v = q.dequeue();
    for each vertex w adjacent to v:
      if w.dist = infinity:
        w.dist = v.dist +1;
        w.path = v;
        q.enqueue(w);
```

This is $O(|V| + |E|)$.