Data Organization: Trees and Graphs

Announcements

- The first lab exam is Monday during the lab session. Example for practice is on the web site. Full lab session
  - 4 Questions
  - Reference Sheet
  - Practice!
- Review 4:30 - 6:30 Tonight, GHC4405
- Lab tomorrow (Thursday)
- PS 6 Friday morning

Reviewing Data Structures

Arrays in Memory

- Example: data = [50, 42, 85, 71, 99]
  Assume we have a byte-addressable computer, integers are stored using 4 bytes (32 bits) and our array starts at address 100.

- If we want data[3], the computer takes the address of the start of the array (100 in our example) and adds **the index * the size** of an array element (4 bytes in our example) to find the element we want.

  Location of data[3] is 100 + 3*4 = 112

- **Do you see why it makes sense for the first index of an array to be 0?**

	Content
100:	50
104:	42
108:	85
112:	71
116:	99

Two-dimensional arrays

- Some data can be organized efficiently in a **table** (also called a **matrix** or **2-dimensional array**)
- Each cell is denoted with two subscripts, a row and column indicator

B[2][3] = 50

B	0	1	2	3	4
0	3	18	43	49	65
1	14	30	32	53	75
2	9	28	38	50	73
3	10	24	37	58	62
4	7	19	40	46	66

2D Lists in Python

```
data = [ [1, 2, 3, 4],
         [5, 6, 7, 8],
         [9, 10, 11, 12]
       ]
>>> data[0]
[1, 2, 3, 4]
>>> data[1][2]
7
>>> data[2][5]   index error
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Linked List Example

- To insert a new element, we only need to change a few pointers.
- Example: Insert 20 between 42 and 85

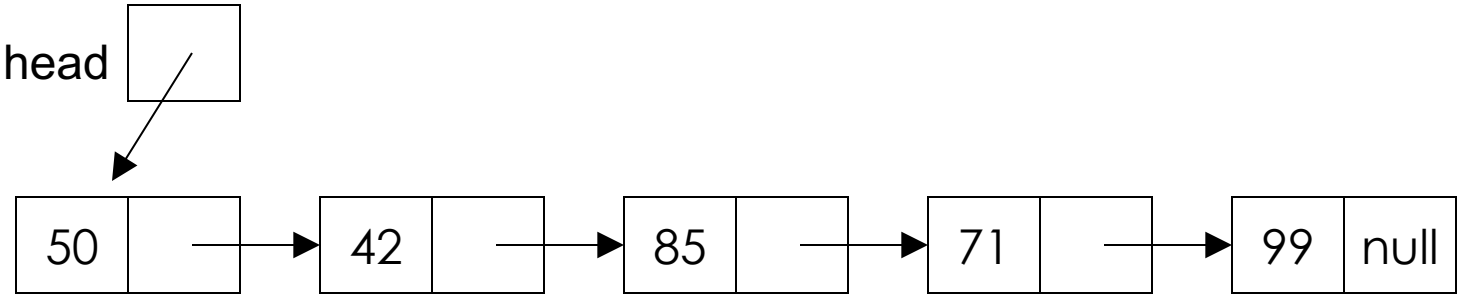
Starting Location of List (head)
124

Assume each integer and pointer requires 4 bytes.

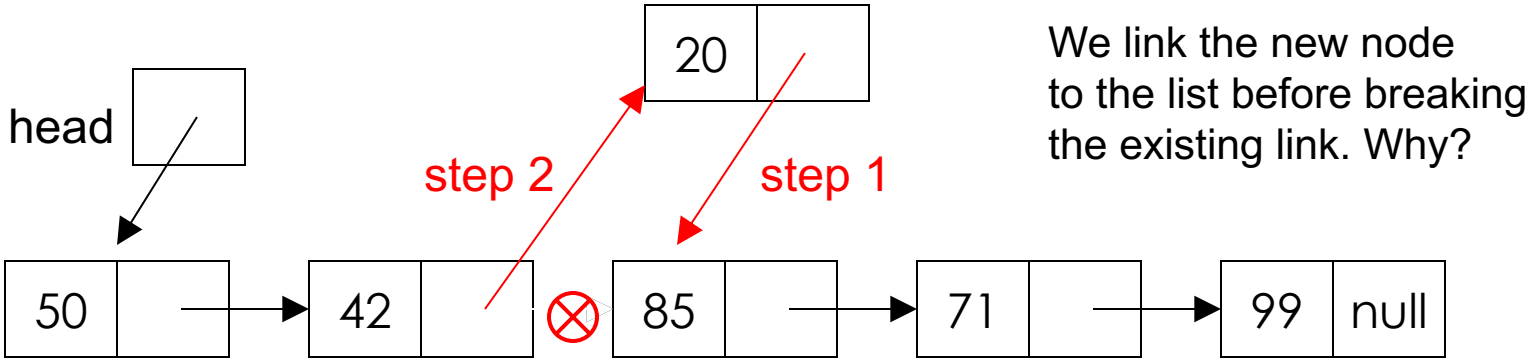
	data	next
100:	42	156
108:	99	0 (null)
116:		
124:	50	100
132:	71	108
140:		
148:	85	132
156:	20	148

Drawing Linked Lists Abstractly

- [50, 42, 85, 71, 99]



- Inserting 20 after 42:



We link the new node to the list before breaking the existing link. Why?

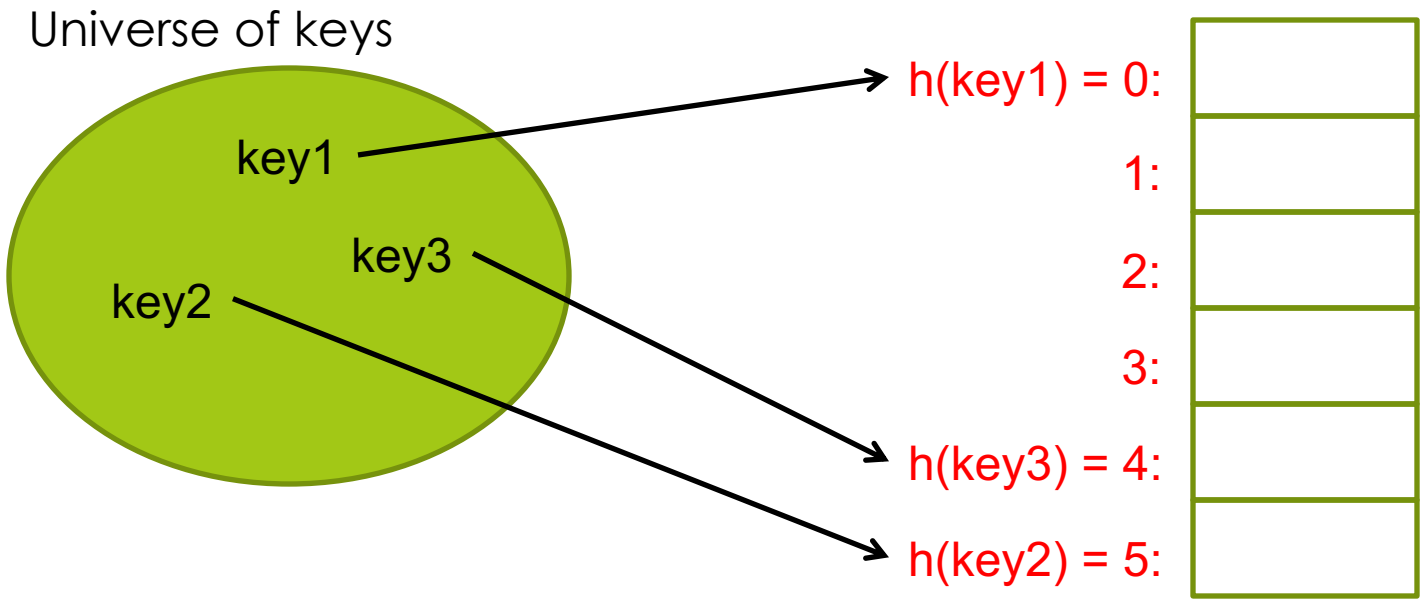
Recall Arrays and Linked Lists

	Advantages	Disadvantages
Arrays	Constant-time lookup (search) if you know the index	Requires a contiguous block of memory
Linked Lists	Flexible memory usage	Linear-time lookup (search)

How can we exploit the advantages of arrays and linked lists to improve search time in dynamic data sets?

Hashing

- A "hash function" `h(key)` that maps a key to an array index in 0..*k*-1.
- To search the array `table` for that key, look in `table[h(key)]`



A hash function `h` is used to map keys to hash-table (array) slots. Table is an array bounded in size. The size of the universe for keys may be larger than the array size. We call the table slots buckets.

Example: Hash function

- Suppose we have (key,value) pairs where the key is a string such as (name, phone number) pairs and we want to store these key value pairs in an array.
- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

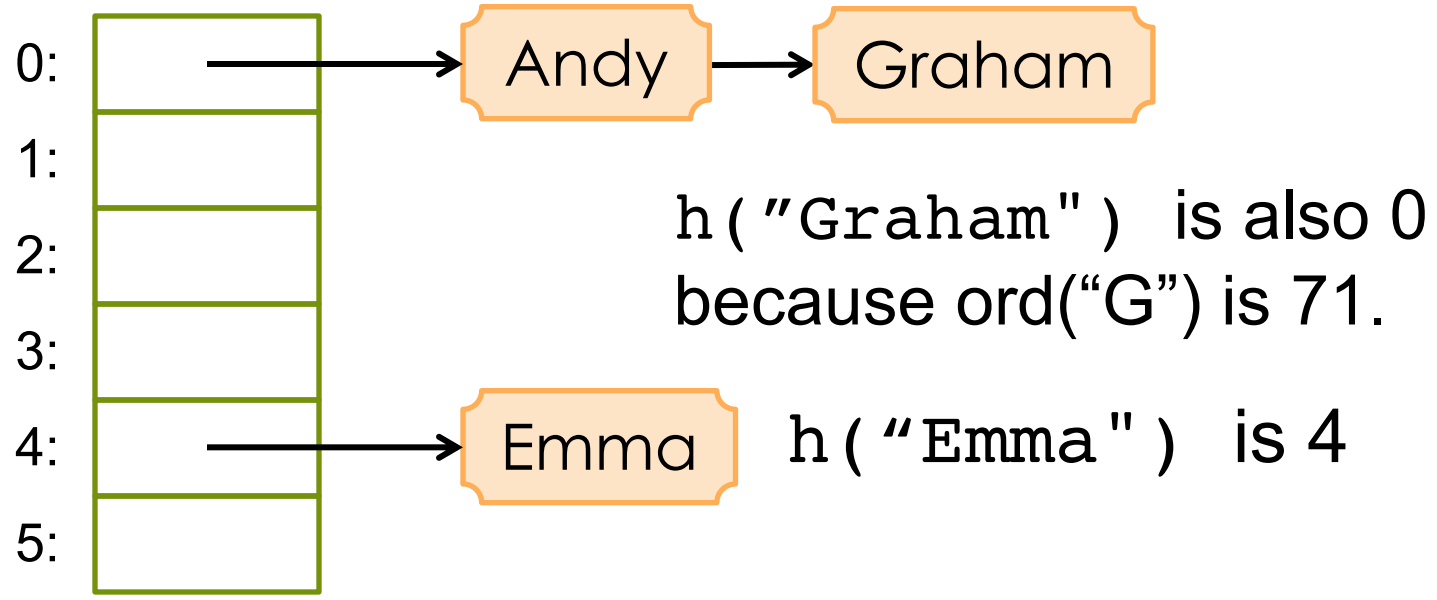
```
def h(str):
    return (ord(str[0]) - 65) % 6
```

Note ord('A') = 65

Unicode

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
0																				
20														!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~		€		‚	*f*	„	…	†	‡	ˆ	‰	Š	‹
140	Œ		Ž			‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯	°	±	2	3
180	´	*µ*	¶	·	¸	1	º	»	¼	½	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û

Add Element "Graham"



In order to add Graham's information to the table we had to form a link list for bucket 0.

Requirements for the Hash Function h(x)

- Must be fast: O(1)
- Must distribute items roughly uniformly throughout the array, so everything doesn't end up in the same bucket.

Summary of Search Techniques

Technique	Setup Cost	Search Cost
Linear search	0, since we're given the list	O(n)
Binary search	O(n log n) to sort the list	O(log n)
Hash table	O(n) to fill the buckets	O(1)

Associative Arrays

- Hashing is a method for implementing associative arrays. Some languages such as Python have associate arrays (**mapping** between keys and values) as a built-in data type.

- Examples:
  - Name in contacts list => Phone number
  - User name => Password
  - Product => Price

Dictionary Type in Python

This example maps car brands (*keys*) to prices (*values*).

```
>>> cars = {"Mercedes": 55000,
            "Bentley": 120000,
            "BMW":90000}
>>> cars["Mercedes"]
55000
```

Keys can be of any **immutable** data type.

Dictionaries are implemented using hashing.

Dictionary Type in Python

This example maps car brands (*keys*) to prices (*values*).

```
>>> cars = {"Mercedes": 55000,
            "Bentley": 120000,
            "BMW":90000}
>>> cars["Mercedes"]
55000
```

Keys can be of any **immutable** data type.

Dictionaries are implemented using hashing.

Iteration over a Dictionary

```
>>> for i in cars:
        print(i)

BMW
Mercedes
Bentley

>>> for i in cars.items():
        print(i)

("BMW", 90000)
("Mercedes", 55000)

("Bentley", 120000)
```

Think what the loop variables are bound to in each case.

Note also that there is no notion of ordering in dictionaries. There is no such thing as the first element, second element of a dictionary.

```
>>> for k,v in cars.items():
        print(k, ":", v )

BMW : 90000
Mercedes 55000
Bentley : 120000
```

Data relationship:

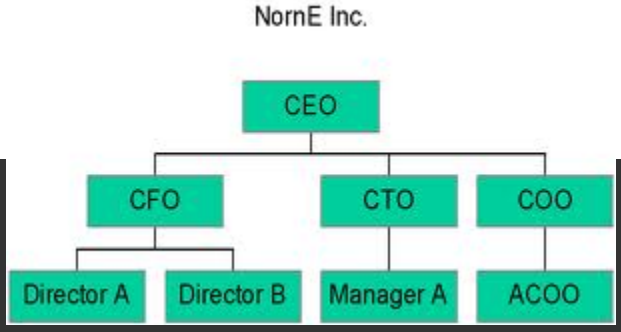
- Arrays
- Linked lists
- Hash tables

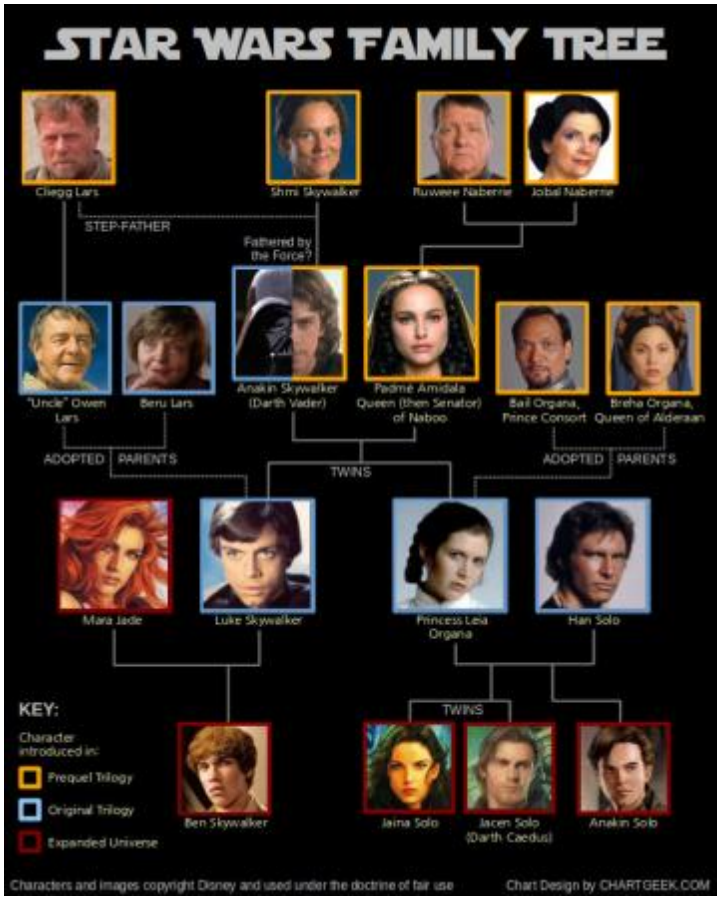
No hierarchy or relationship between data items, other than their order in the sequence in the case of arrays and linked lists

Today

- Data structures for hierarchical data

Hierarchical Data

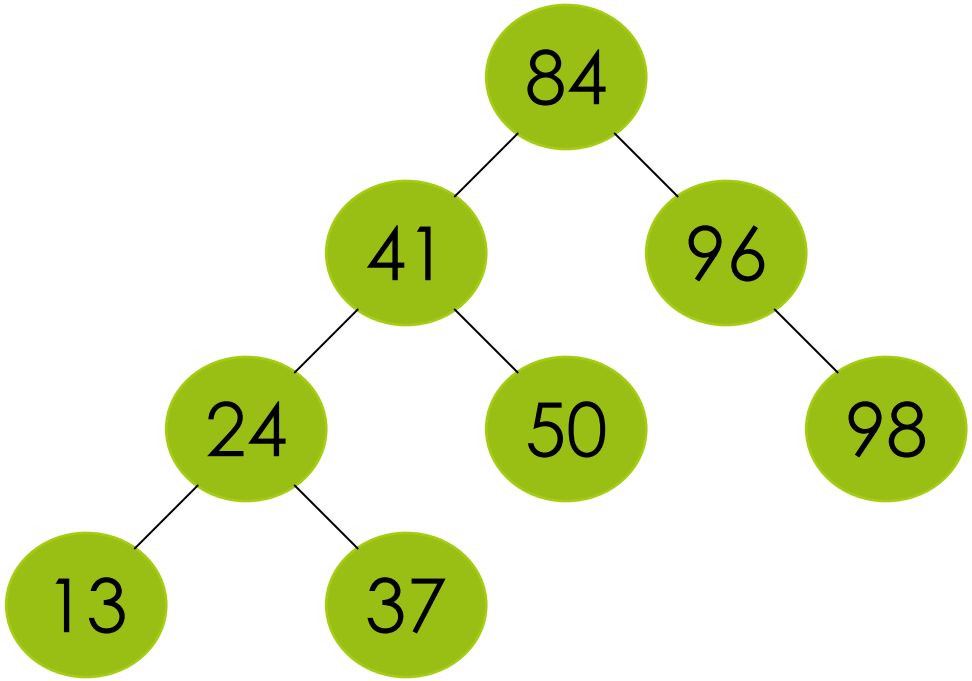




Trees

- A **tree** is a hierarchical data structure.
  - Every tree has a **node** called the **root**.
  - Each node can have 1 or more nodes as **children**.
  - A node that has no children is called a **leaf**.
- A common tree in computing is a **binary tree**.
  - A binary tree consists of nodes that have at most 2 children.
- Applications: data compression, file storage, game trees

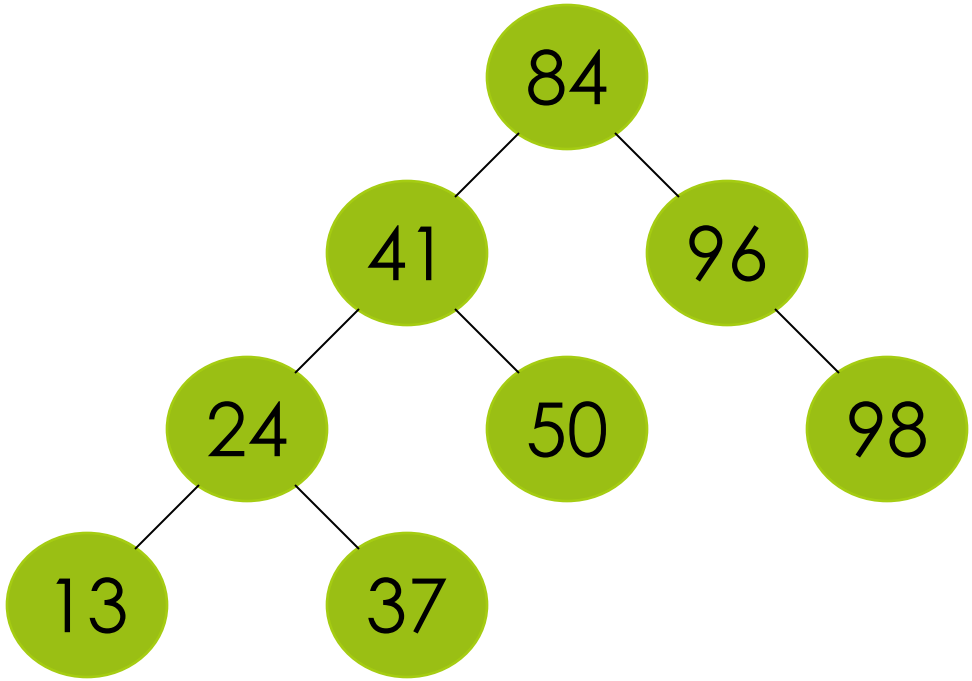
Binary Tree



In order to illustrate main ideas we label the tree nodes with the keys only.
In fact, every node would also store the rest of the data associated with that key. Assume that our tree contains integers keys.

Which one is the root?
Which ones are the leaves (external nodes)?
Which ones are internal nodes?
What is the height of this tree?

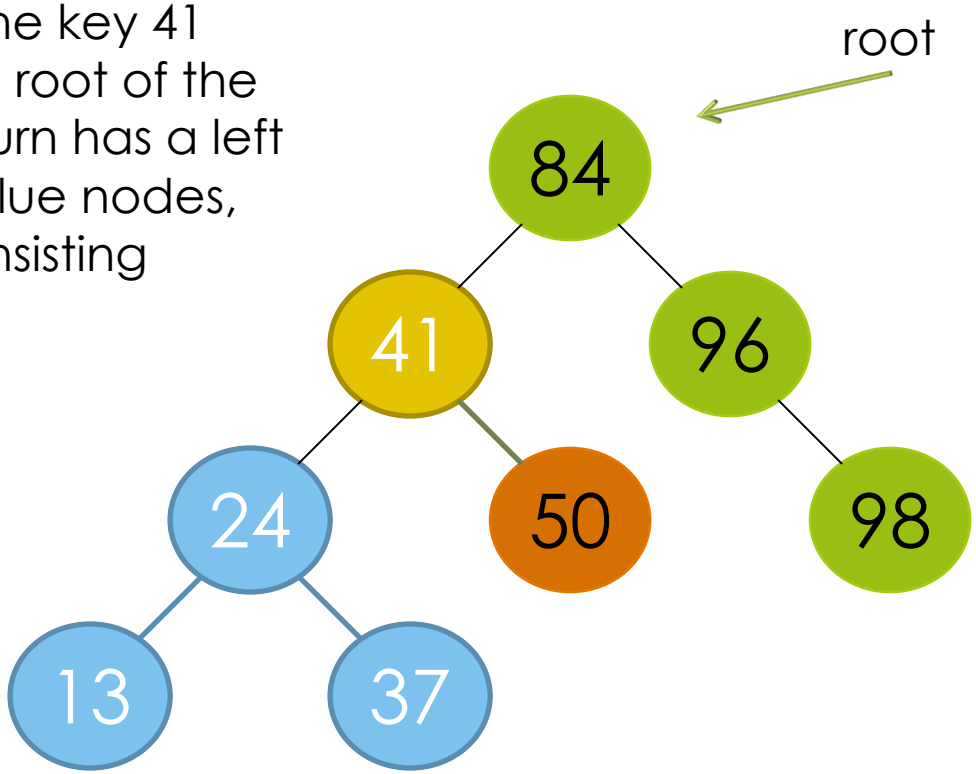
Binary Tree



The root contains the data value 84.
There are 4 leaves in this binary tree: nodes containing 13, 37, 50, 98.
There are 3 internal nodes in this binary tree: nodes containing 41, 96, 24
This binary tree has height 3 – considering root is at level 0,
the maximum level among all nodes is 3

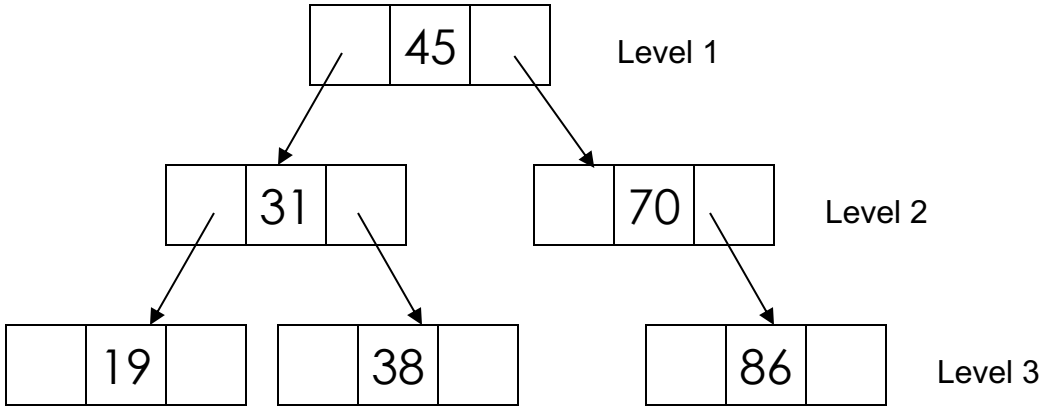
Binary Tree

Note the r**ecursive** structure:
The yellow node with the key 41 can be viewed as the root of the left subtree , which in turn has a left subtree consisting of blue nodes, and a right subtree consisting of orange nodes.



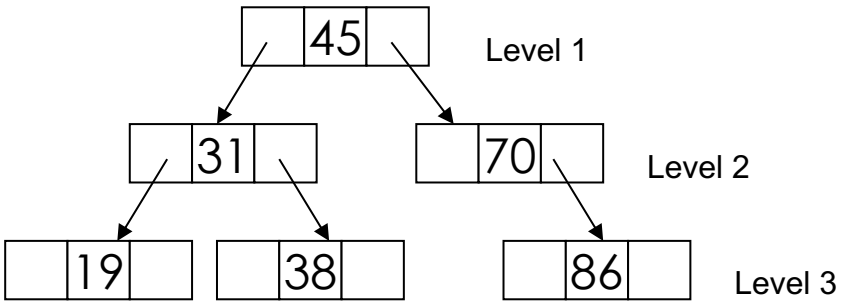
Binary Trees: Implementation

- One common implementation of binary trees uses nodes like a linked list does.
  - Instead of having a "next" pointer, each node has a "left" pointer and a "right" pointer.



Using Nested Lists

- Languages like Python do not let programmers manipulate pointers explicitly.
- We could use Python lists to implement binary trees. For example:



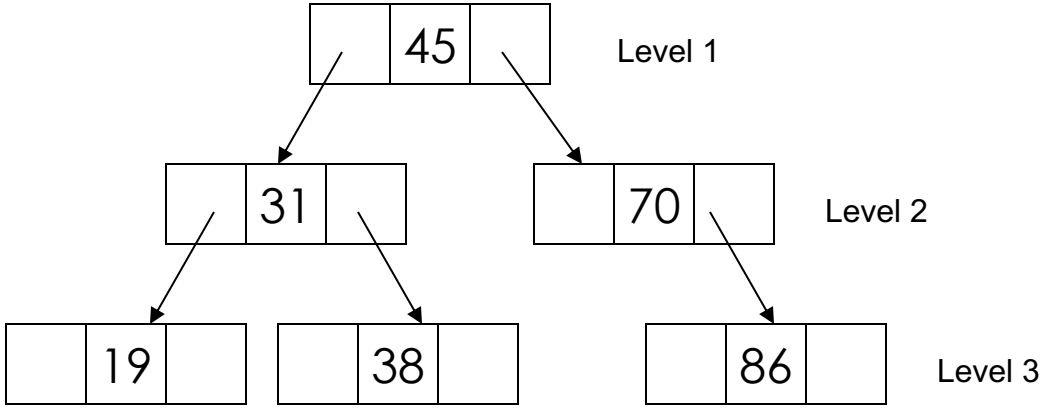

```
[45, left, right]

[45,[31,left,right],[70,left,right]]

[45, [31, [19,[],[]], [38,[],[]]],
      [70,[], [86, [], []]]
]
```

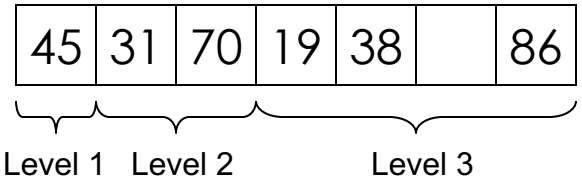
`[]` stands for an empty tree

Arrows point to subtrees

Using One Dimensional Lists

- We could also use a flat (one-dimensional list).





Dynamic Data Set Operations

- Insert
- Delete
- Search
- Find min/max
- ...

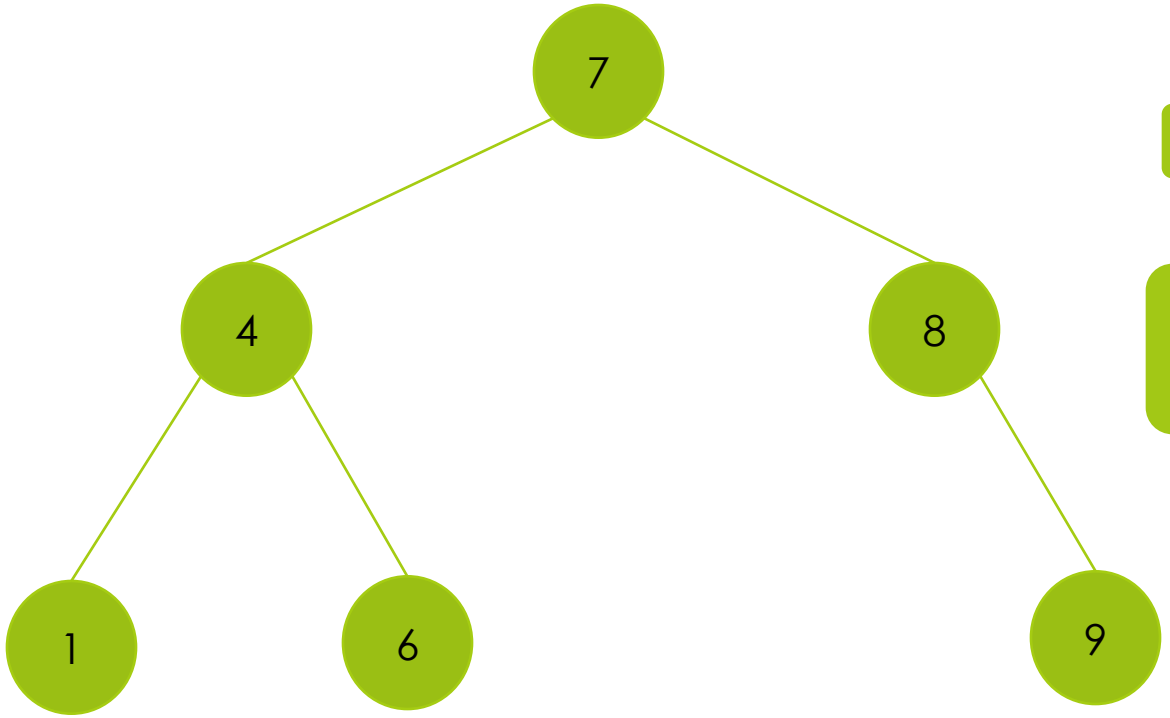
Choosing a specific data structure has consequences on which operations can be performed faster.

Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree that satisfies the binary **search tree ordering invariant** stated on the next slide

Example: Binary Search Tree

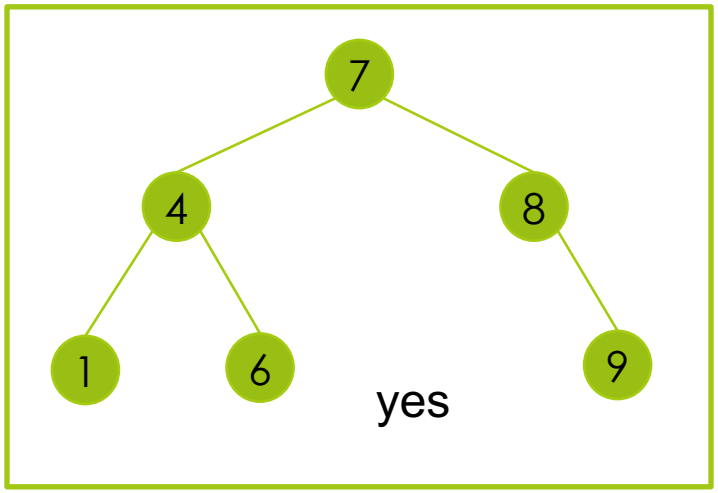
BST ordering invariant: At any node with key *k,* all keys of elements in the left subtree are strictly less than *k* and all keys of elements in the right subtree are strictly greater than *k* (assume that there are no duplicates in the tree)

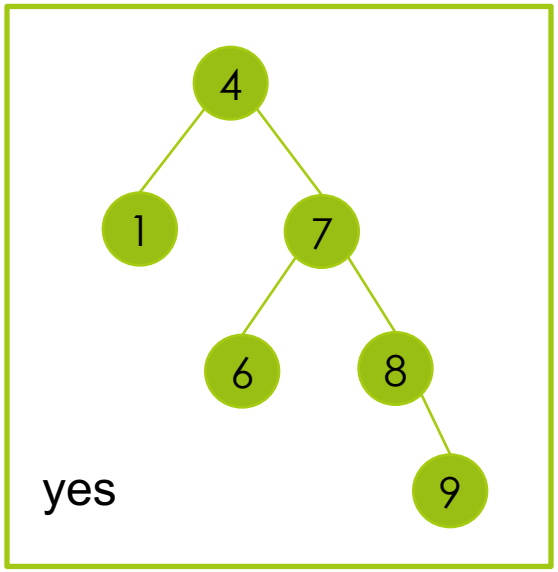


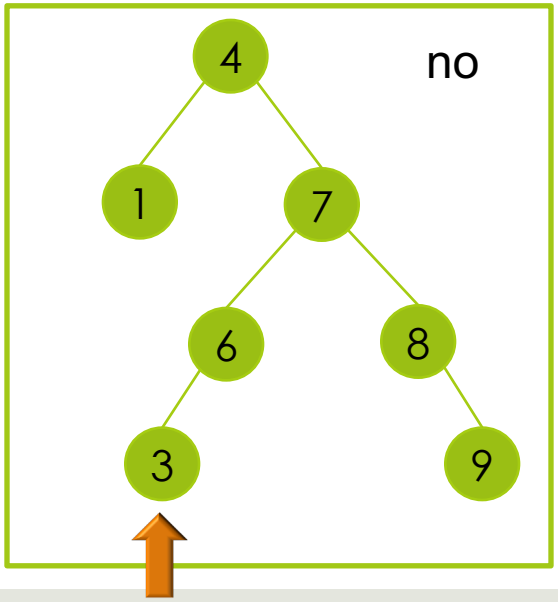
Binary tree

Satisfies the ordering invariant

Test: Is this a BST?





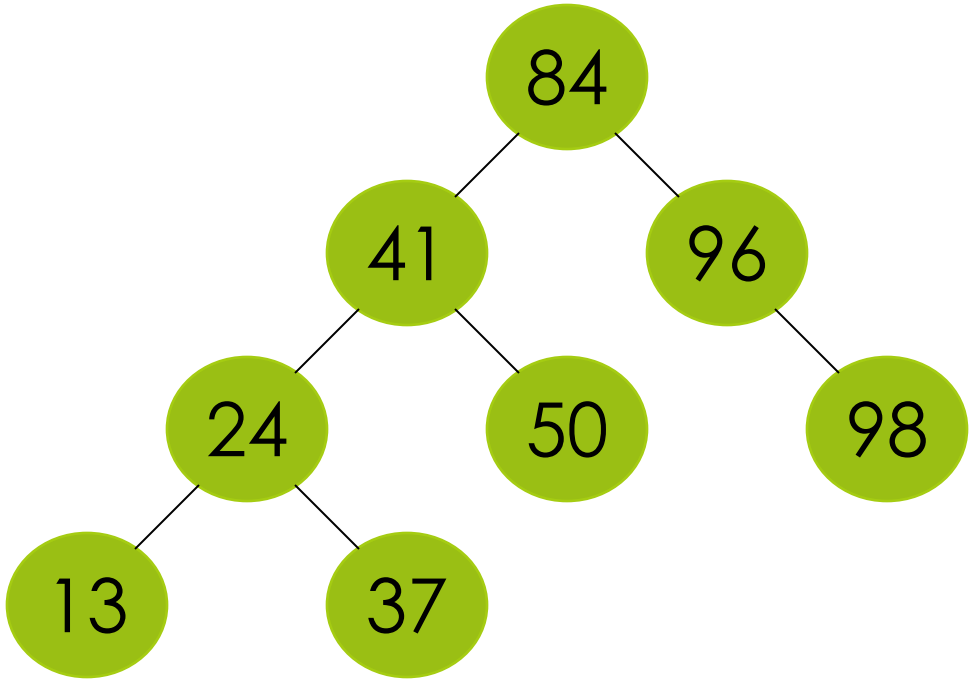


Inserting into a BST

- For each data value that you wish to insert into the binary search tree:
  - Start at the root and compare the new data value with the root.
  - If it is less, move down left. If it is greater, move down right.
  - Repeat on the child of the root until you end up in a position that has no node.
  - Insert a new node at this empty position.

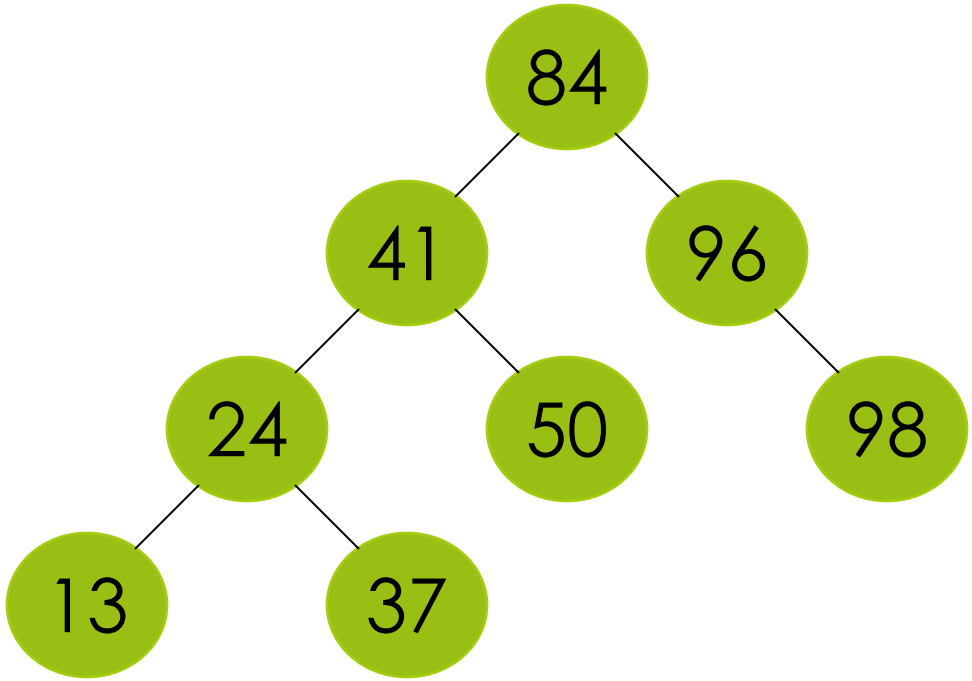
Example

- Insert: 84, 41, 96, 24, 37, 50, 13, 98



Using a BST

- How would you search for an element in a BST?

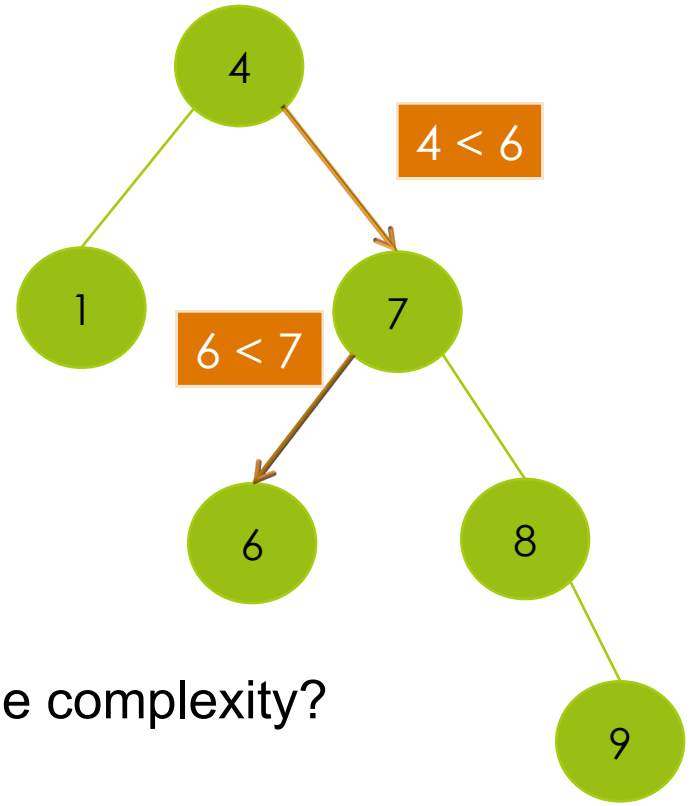


Searching a BST

- For the key that you wish to search
  - Start at the root and compare the key with the root. If equal, key found.
  - Otherwise
    - If it is less, move down left. If it is greater, move down right. Repeat search on the child of the root.
    - If there is no non-empty subtree to move to, then key not found.

Searching the tree

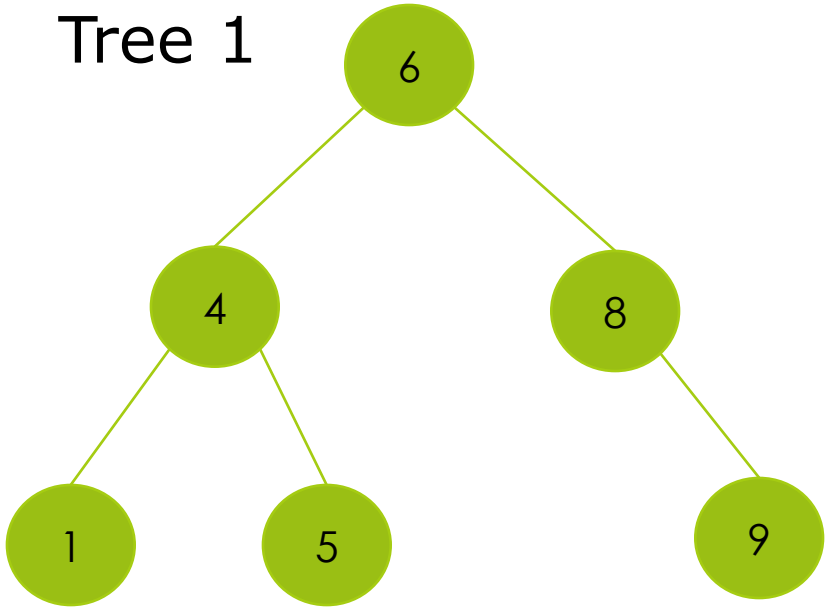
Example: searching for 6



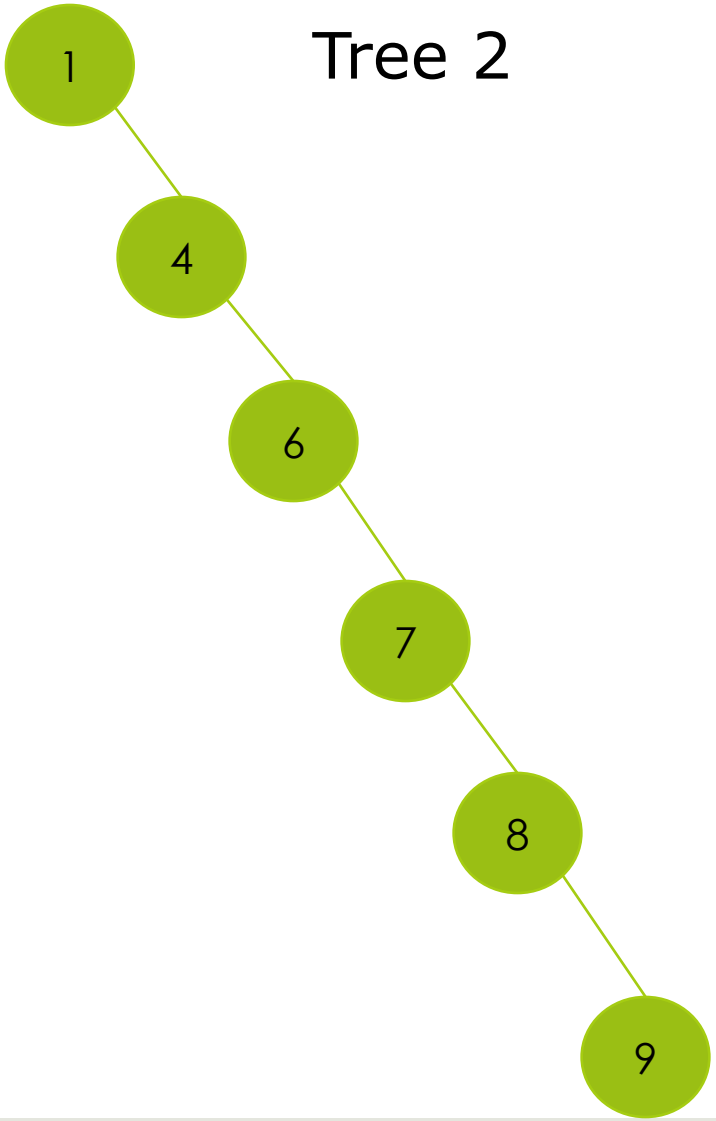
Can we form a conjecture about worst case complexity?

Time complexity of search

Tree 1



Tree 2



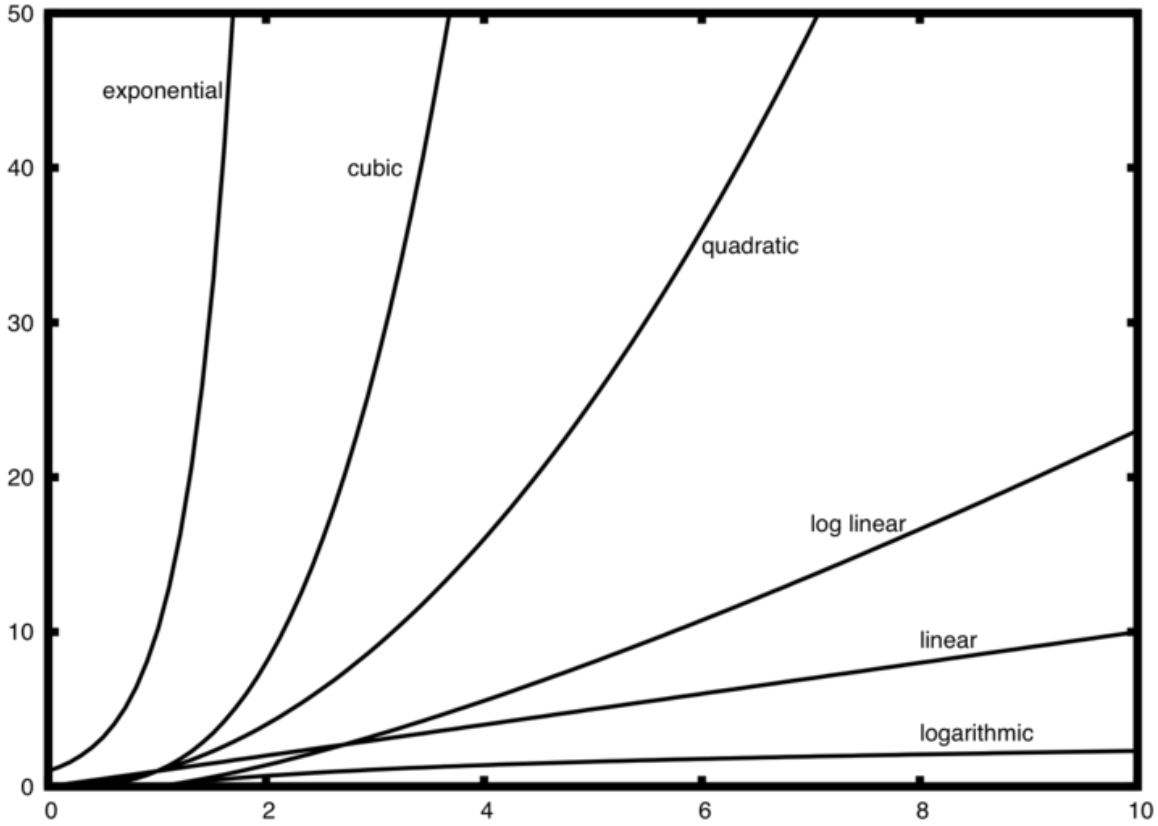
Number of nodes: *n*

Worst case: *O(height)*

Worst *height*: *n* ⇒ *O(n)*

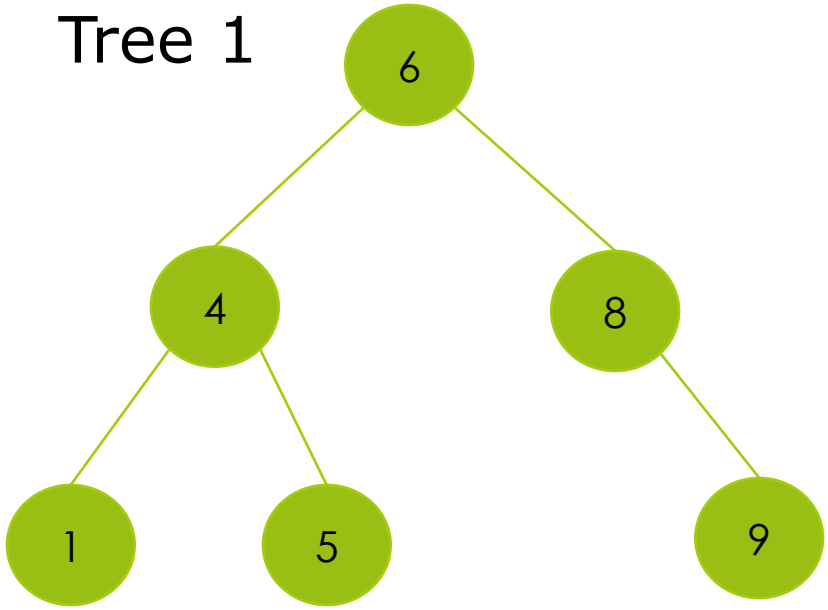
Big O

- O(1) constant
- O(log *n*) logarithmic
- O(*n*) linear
- O(*n* log *n*) log linear
- O(n^2) quadratic
- O(n^3) cubic
- O(2^n) exponential

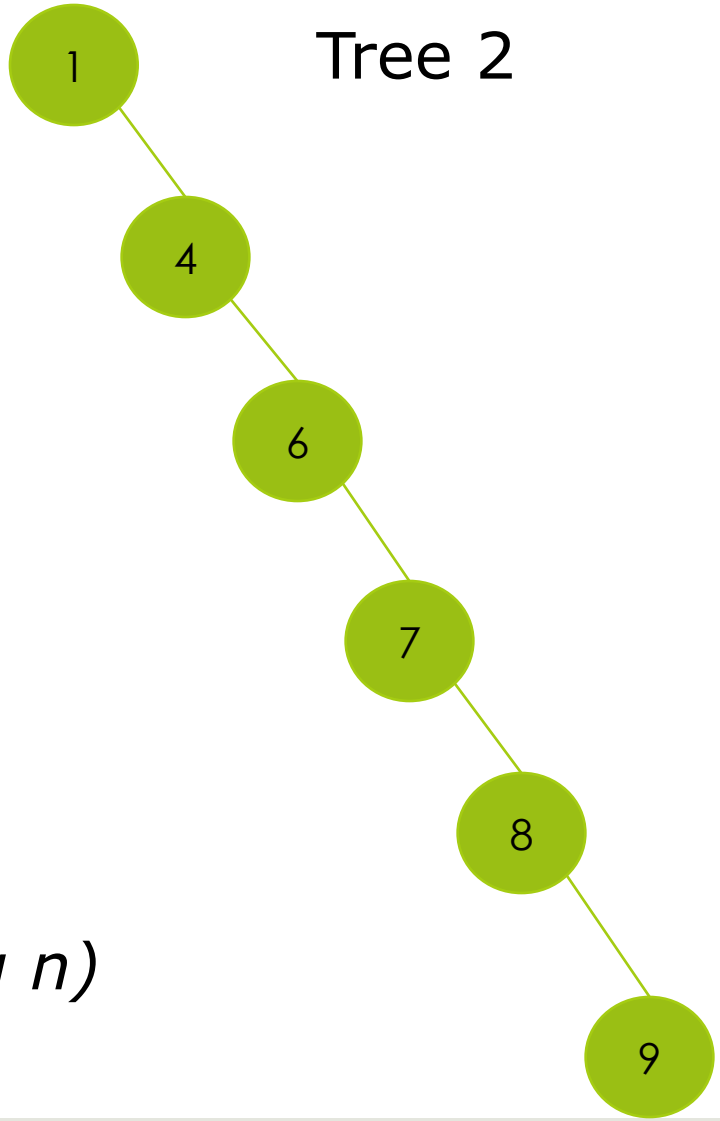


Time complexity of search

Tree 1



Tree 2

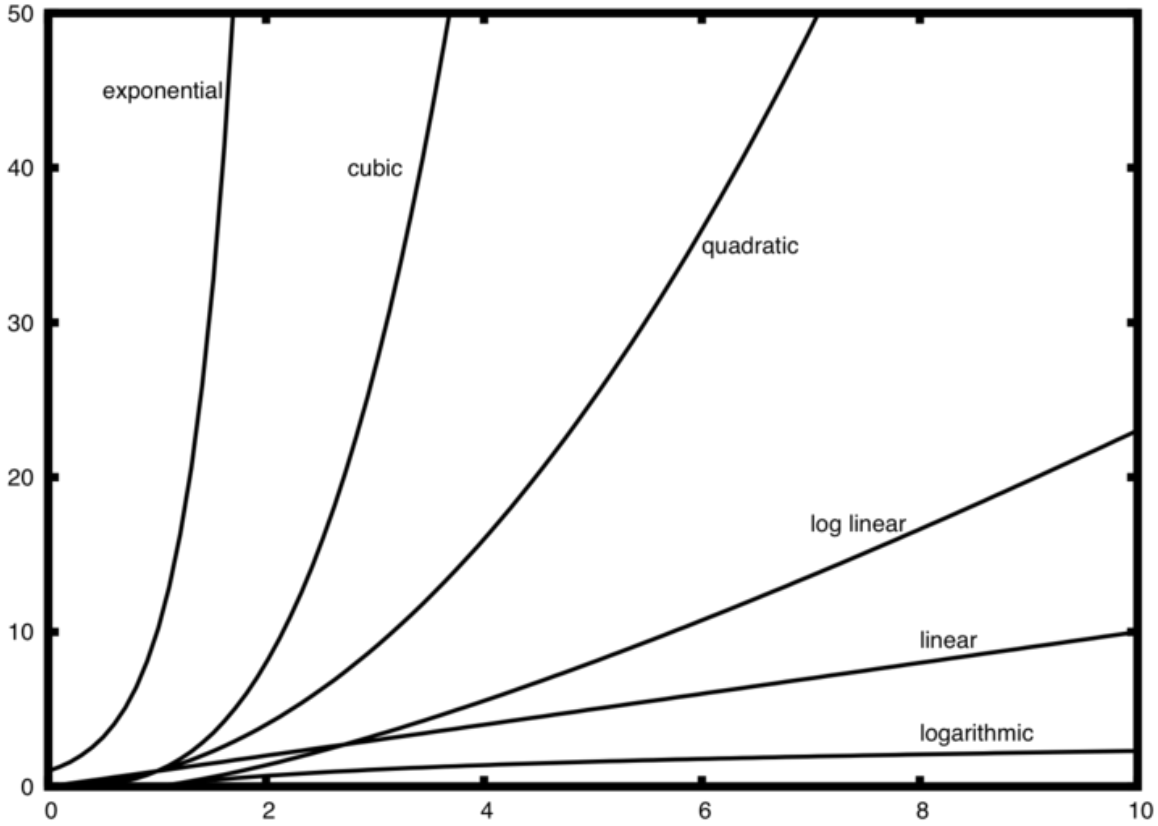


Number of nodes: *n*

What if we could always have balanced trees? ⇒ *O(log n)*

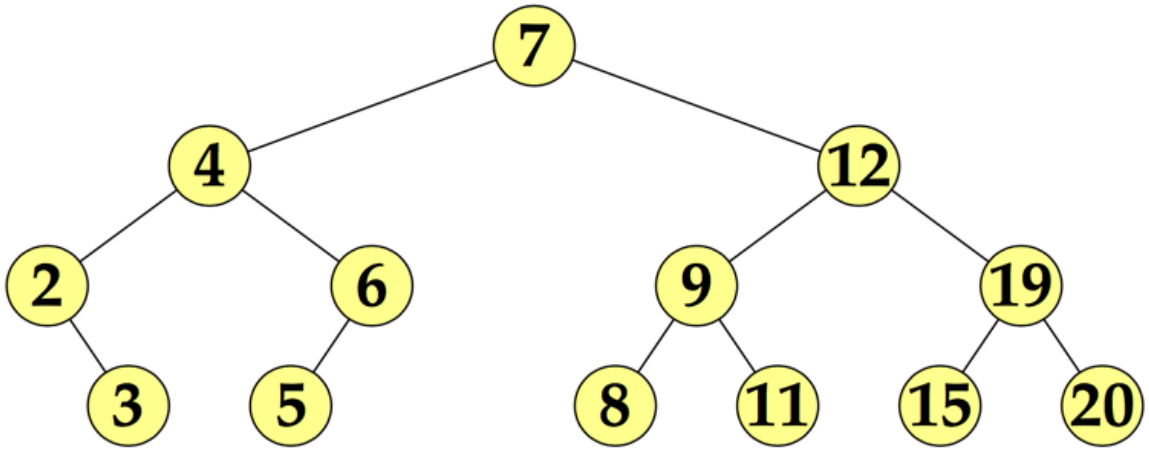
Big O

- O(1) constant
- O(log *n*) logarithmic
- O(*n*) linear
- O(*n* log *n*) log linear
- O(n^2) quadratic
- O(n^3) cubic
- O(2^n) exponential

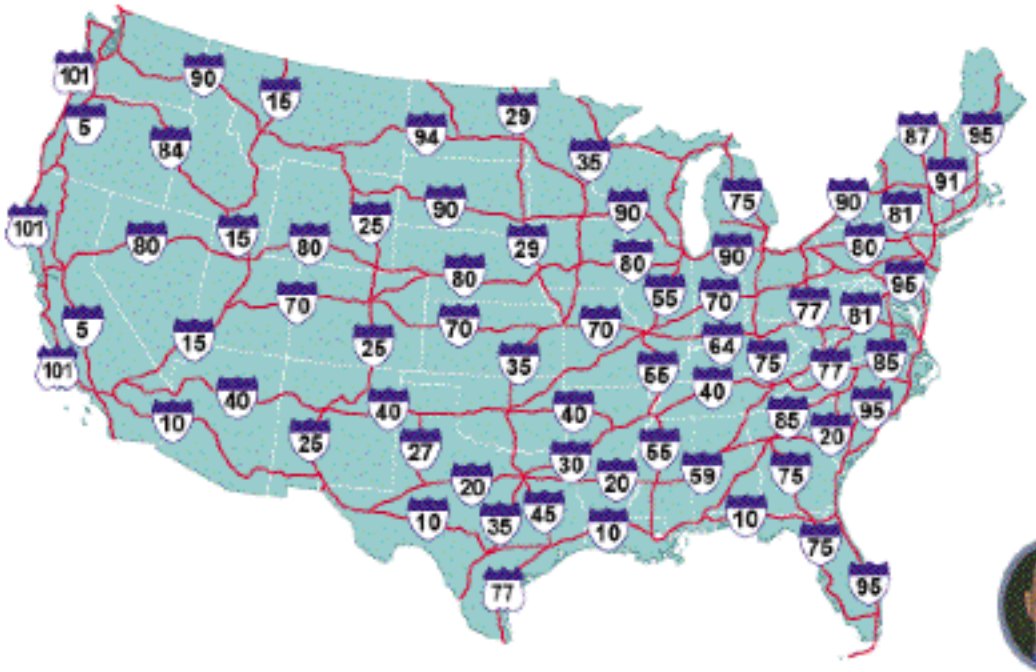


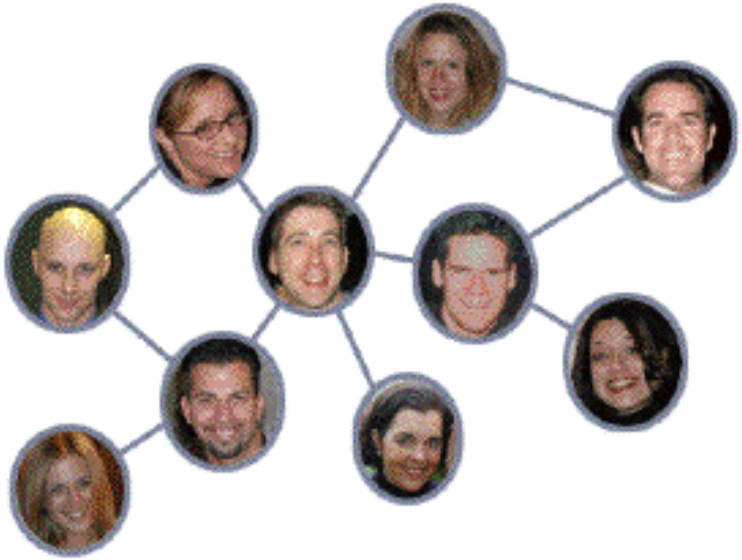
Exercises

- How you would find the minimum and maximum elements in a BST?
- What would be output if we walked the tree in left-node-right order?



Graphs

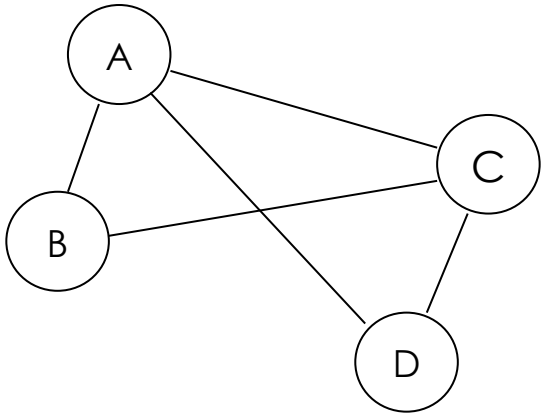

101
90
15
5
84
94
29
35
87
95
91
75
90
81
101
80
15
80
25
90
29
90
80
80
90
80
95
55
70
77
81
70
5
15
70
70
64
25
75
77
85
101
35
55
40
40
40
40
85
95
10
20
25
27
55
30
20
59
75
20
10
35
45
10
10
75
77
95

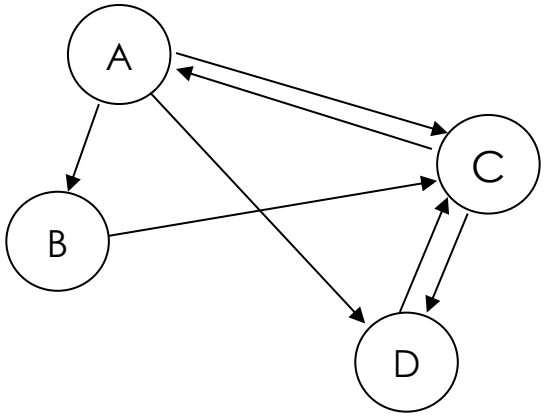


Graphs

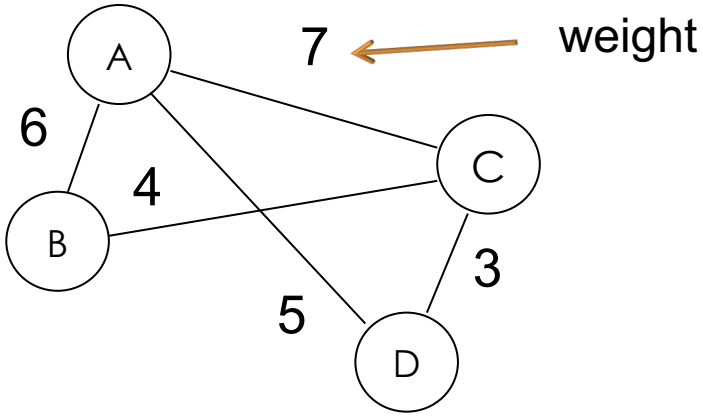
- A **graph** is a data structure that consists of a set of vertices and a set of edges connecting pairs of the vertices.
  - A graph doesn't have a root, per se.
  - A vertex can be connected to any number of other vertices using edges.
  - An edge may be bidirectional or directed (one-way).
  - An edge may have a weight on it that indicates a cost for traveling over that edge in the graph.
- Applications: computer networks, transportation systems, social relationships

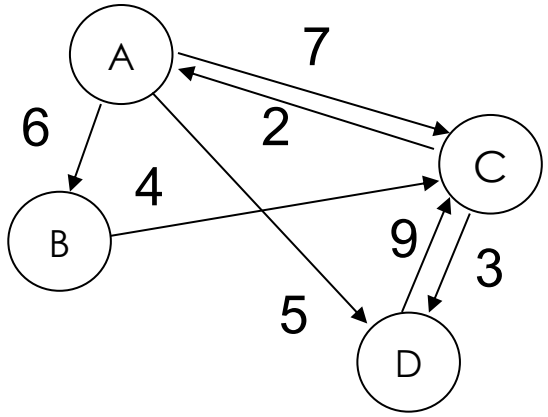
Undirected and Directed Graphs





Undirected and Directed Graphs

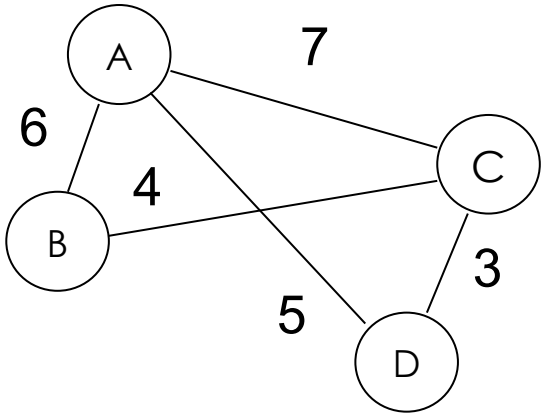




from \ to	A	B	C	D
A	0	6	7	5
B	6	0	4	∞
C	7	4	0	3
D	5	∞	3	0

from \ to	A	B	C	D
A	0	6	7	5
B	∞	0	4	∞
C	2	∞	0	3
D	∞	∞	9	0

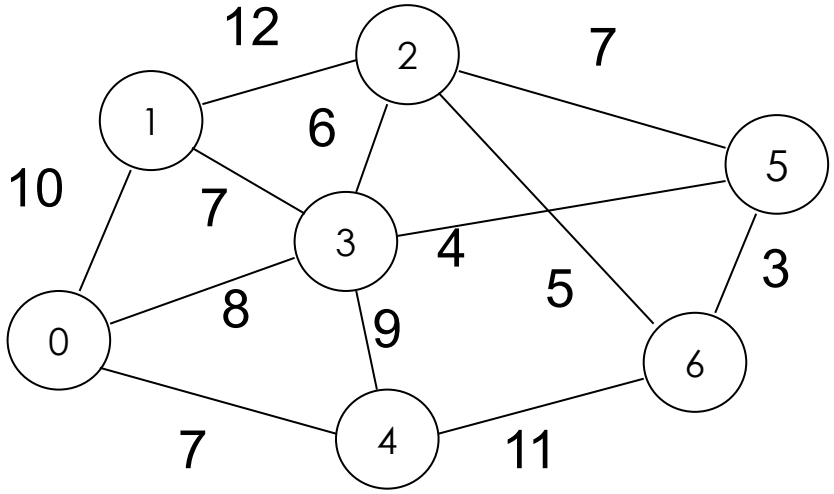
Graphs in Python



from \ to	A	B	C	D
A	0	6	7	5
B	6	0	4	∞
C	7	4	0	3
D	5	∞	3	0

```
graph =
[ [ 0, 6, 7, 5 ],
  [ 6, 0, 4, float('inf') ],
  [ 7, 4, 0, 3],
  [ 5, float('inf'), 3, 0] ]
```

An Undirected Weighted Graph



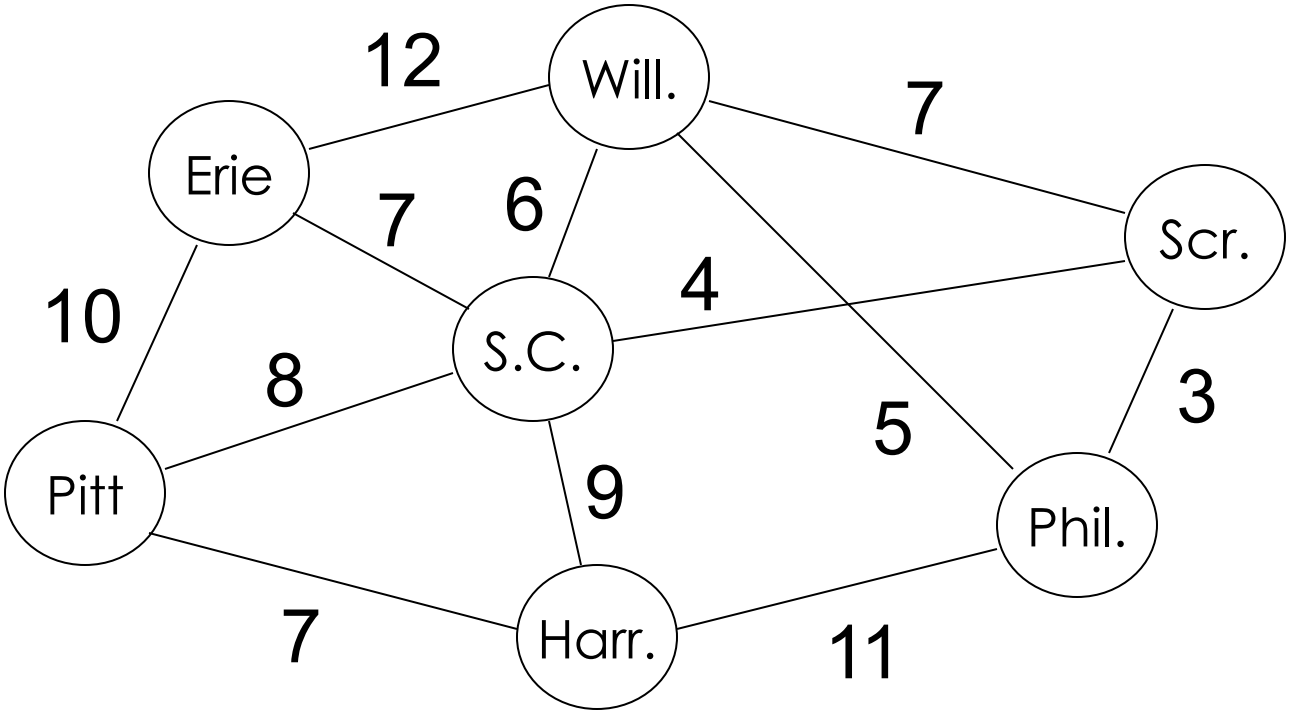
0	1	2	3	4	5	6
Pitt.	Erie	Will.	S.C.	Harr.	Scr.	Phil.

vertices

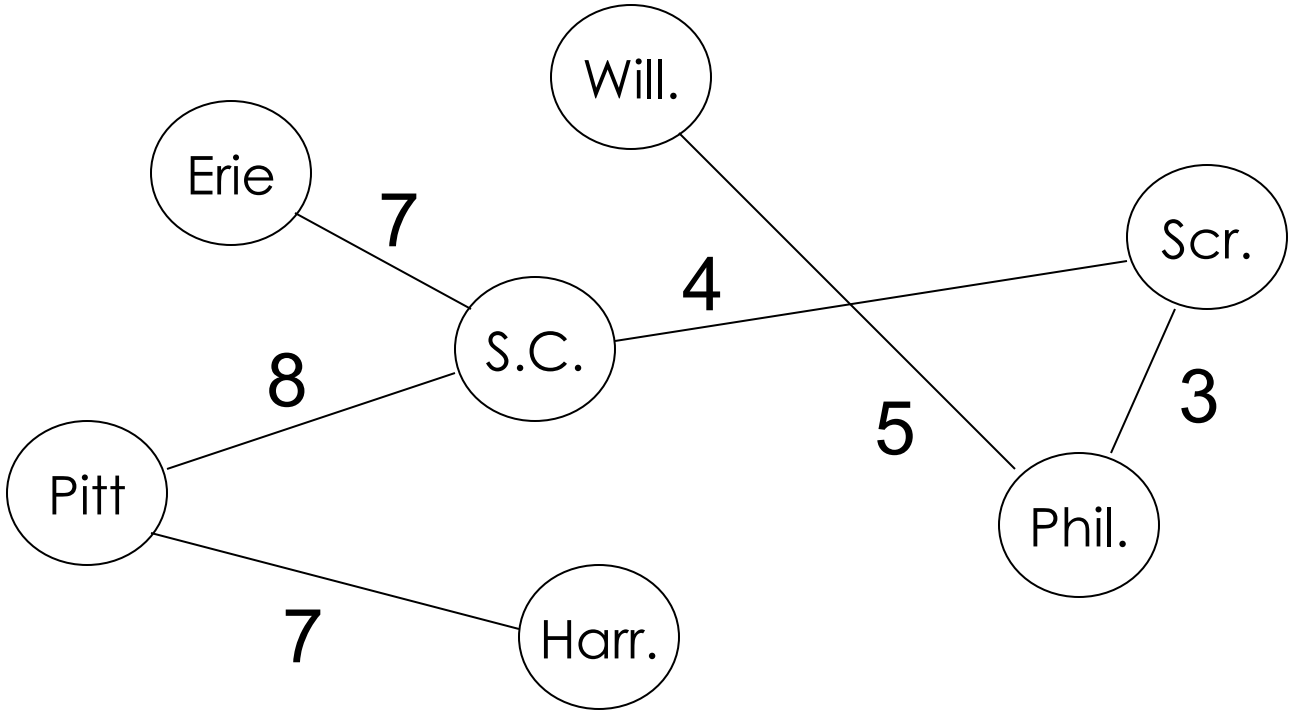
from \ to	0	1	2	3	4	5	6
0	0	10	∞	8	7	∞	∞
1	10	0	12	7	∞	∞	∞
2	∞	12	0	6	∞	7	5
3	8	7	6	0	9	4	∞
4	7	∞	∞	9	0	∞	11
5	∞	∞	7	4	∞	0	3
6	∞	∞	5	∞	11	3	0

edges

Original Graph

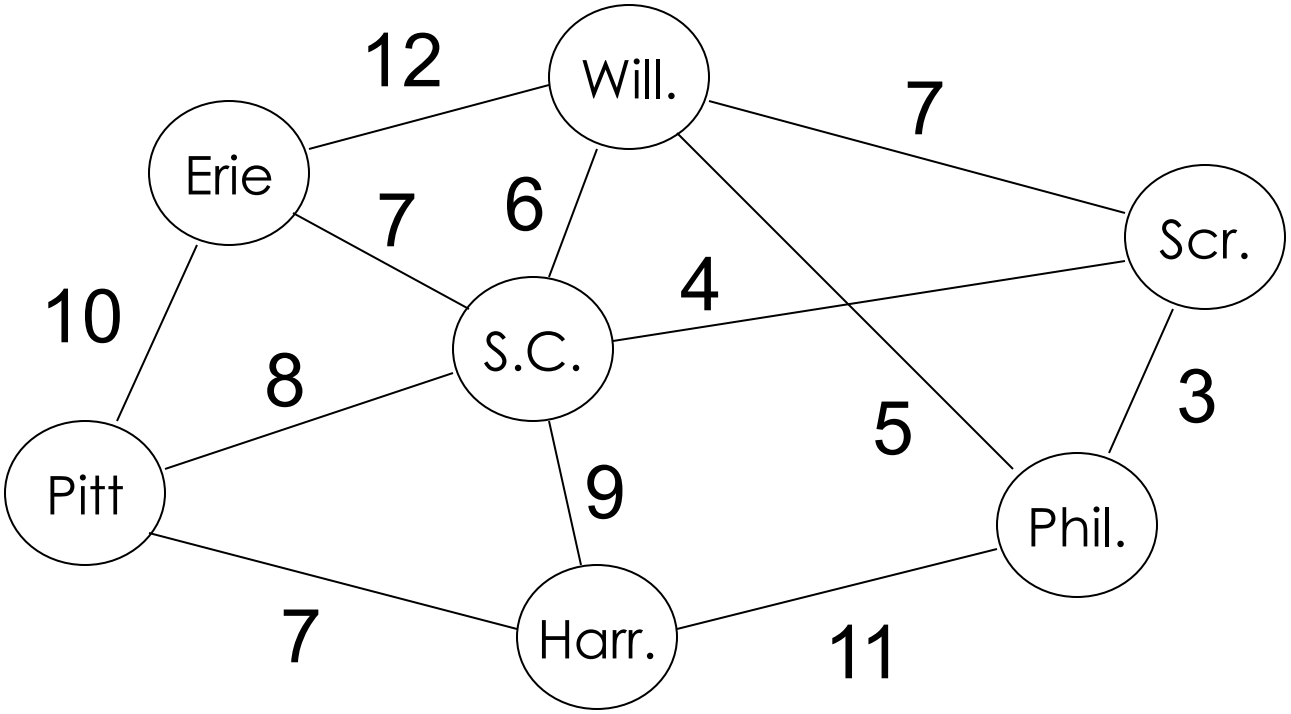


A Minimal Spanning Tree

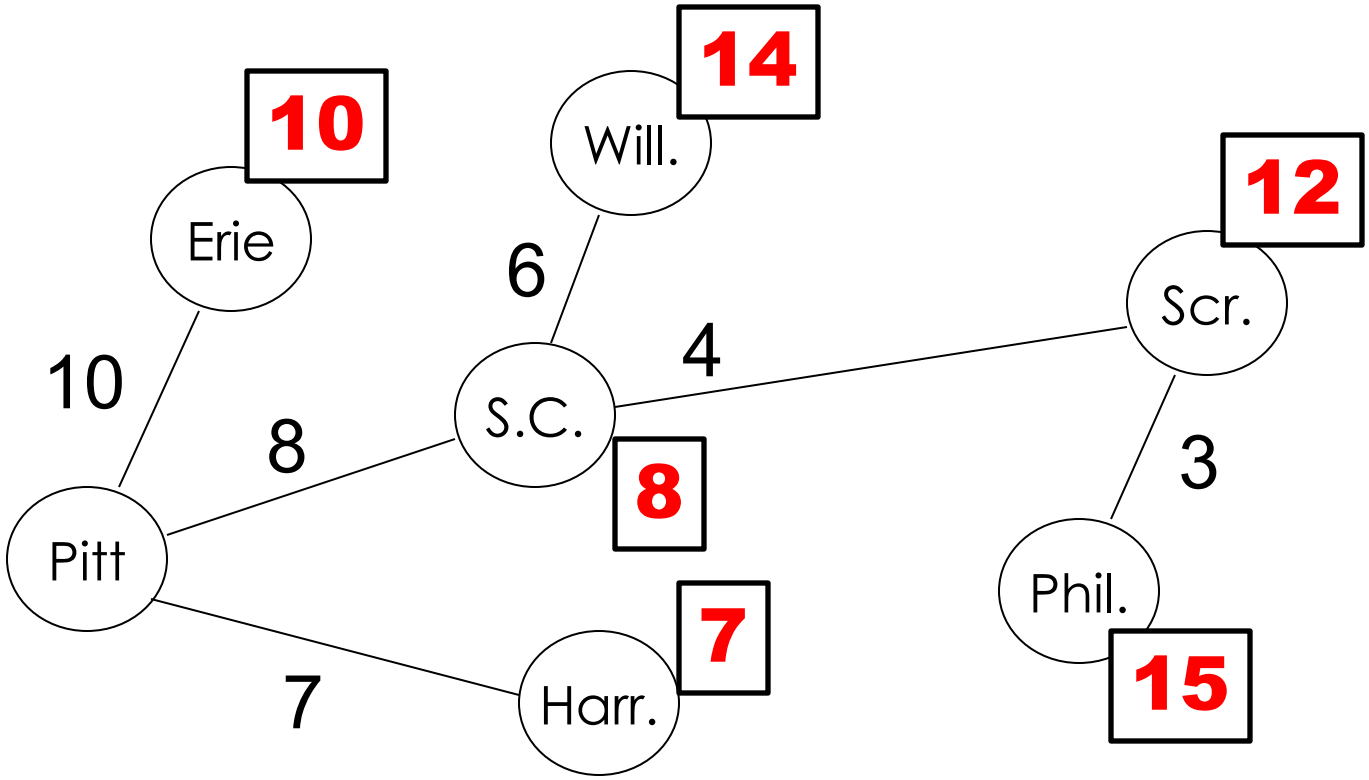


The minimum total cost to connect all vertices using edges from the original graph without using cycles. (minimum total cost = 34)

Original Graph



Shortest Paths from Pittsburgh



The total costs of the shortest path from Pittsburgh to every other location using only edges from the original graph.

Graph Algorithms

- There are algorithms to compute the minimal spanning tree of a graph and the shortest paths for a graph.
- There are algorithms for other graph operations:
  - If a graph represents a set of pipes and the number represent the maximum flow through each pipe, then we can determine the maximum amount of water that can flow through the pipes assuming one vertex is a "source" (water coming into the system) and one vertex is a "sink" (water leaving the system)
  - Many more graph algorithms... very useful to solve real life problems.

We did not focus on graph algorithms in this unit. We only covered how to represent them with lists.