

Core data structures

It's all about relationships

Terence Parr

MSDS program

University of San Francisco

Quick sidenote:

Yes, much of this initial lecture is familiar to you from previous coursework here or during undergraduate work. I'm mainly being thorough and complete, but I also hope to show you a new perspective and show you that data structures can often be viewed as versions of each other.

Data structures organize data

- Data structures group or encode relationships between data elements
- There's a difference between the *abstract data type* and the implementation (list vs array, dictionary vs hashtable, ...)
- Two methods to organize data:
 - physical adjacency or relative position in memory (RAM)
 - *pointers*
- Algorithms operate on data structures; e.g., sorting algorithm operates on a list
- Often algorithms are needed to construct structures too, but let's get familiar with what these data structures look like and then focus on algorithms that operate on them

Advice on choosing data structures

- Use the simplest data structure you can initially because you never know if that code will survive very long
- Waste processor, memory power before brainpower (if possible)
- There is a trade off between time and space
 - We can often make faster algorithm using more memory
 - It's like driving to the other side of town to save 10% on gas; what are you trying to optimize? time or \$\$\$
- Prep work or more sophisticated data structure can help
 - E.g., element lookup via: unordered list vs sorted list vs hash table

$O(n)$

$O(\log n)$

$O(1)$

Why you should know about DS/Alg

- Consider Enron emails
- Represent how?
- Depends on what?
- Depends on the info we want to extract
- Find all emails by Keith
- Find email path from Keith to Phillip or find path length
- Find all direct emailers to Keith

Date: Wed, 18 Oct 2000 03:00:00 -0700 (PDT)
From: phillip.allen@enron.com
To: leah.arsdall@enron.com

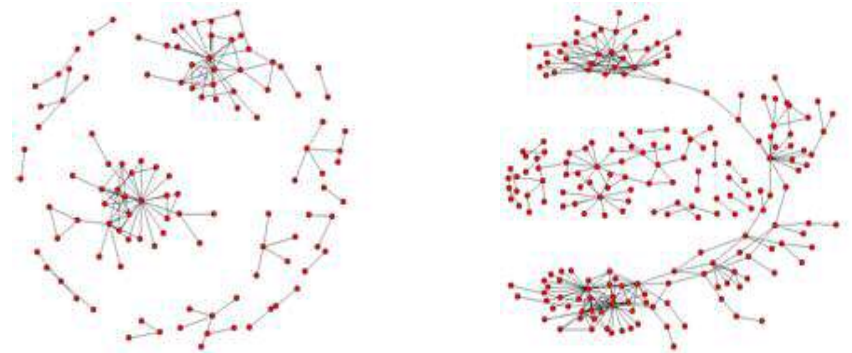
Subject: Date: Mon, 16 Oct 2000 06:42:00 -0700 (PDT)
Mime-Version: From: phillip.allen@enron.com
Content-Type: To: buck.buckner@honeywell.com
...

Subject: Date: Mon, 9 Oct 2000 07:00:00 -0700 (PDT)
gas price From: phillip.allen@enron.com
Mime-Version: To: keith.holst@enron.com
Content-Type: Subject: Consolidated positions: Issues & To Do list
... Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
...

What can we learn, what alg's do we need

- Fast string search to find emails
- Compute edit distance to find similar or misspelled email addrs
- Shortest path analysis to discover company relationships not on org chart
- k-cliques (subcommunities) became more common as crisis built at Enron

Date: Mon, 9 Oct 2000 07:00:00 -0700 (PDT)
From: phillip.allen@enron.com
To: keith.holst@enron.com
Subject: Consolidated positions: Issues & To Do list
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
...



See *Social Network Analysis and Organizational Disintegration: The Case of Enron Corporation*

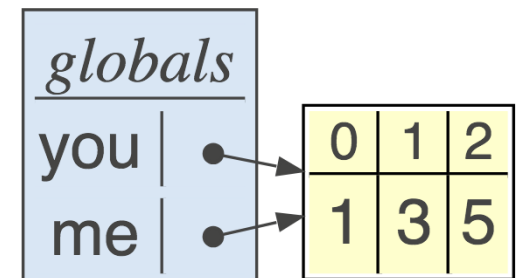
Elemental data in memory (RAM)

- (not disk formats, which we covered in data acquisition MSDS692)
- What's the *type*?: typically int, float, string
- Numbers can be of different sizes; e.g., np.float32, np.float64
- Data *values*: an int can represent a number, signed or unsigned, but can also represent a categorical item such as US state
- We can also use strings for categorical but it's much less efficient in space, and often time
- You can even encode multiple things within a single number, such as 5 and 1105 could be floor 11, room 05
- Data *properties*: e.g., can such values be ordered? Is there a notion of distance between values?

Pointer data type

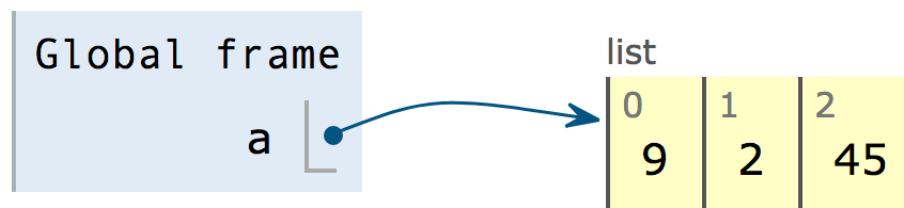
- A pointer `p` is implemented as an integer variable that holds a memory address, such as “`p = Point(3,4)`”
- Python knows variables are actually references to memory locations
- Pointers are also called *references*

```
you = [1,3,5]  
me  = you
```



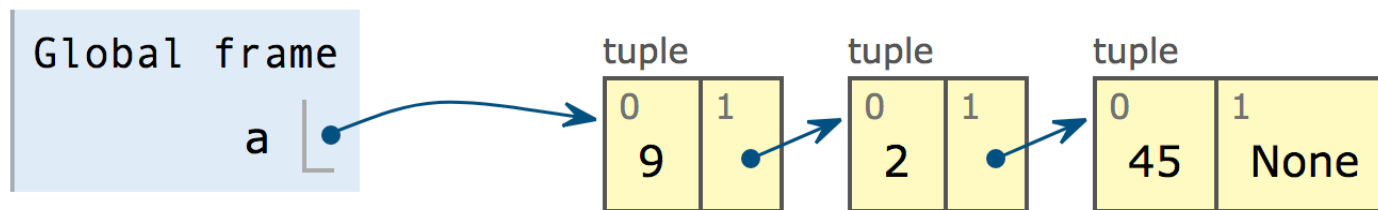
List abstract data type

- *Array* implementation is most common implementation of the *list* abstract data structure
- Lists are ordered but items aren't necessarily sortable
- Arrays use contiguous memory locations to associate items
- Code “a=[9,2,45]” yields a pointer to contiguous block of cells



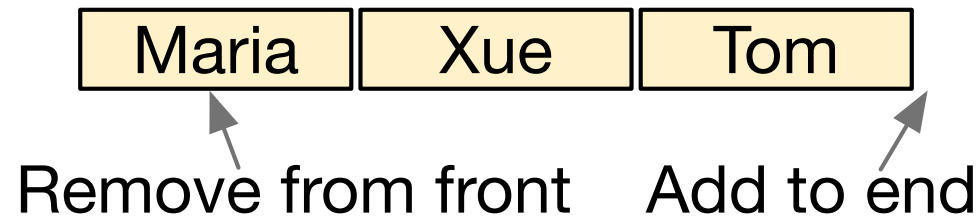
Non-contiguous lists: *linked lists*

- The other way to implement a list data type is with explicit pointers from one element to the next: “a = (9,(2,(45,None)))”



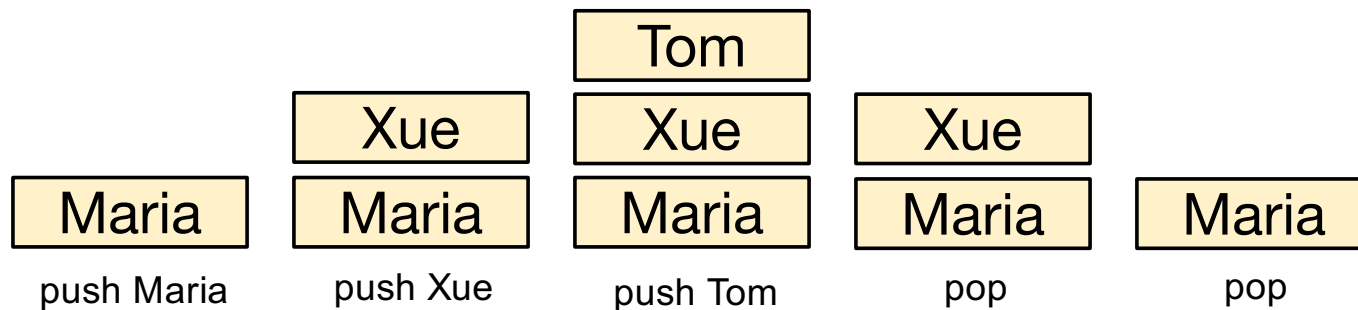
Queue: ordered list

- First In, First Out (**FIFO**); Key ops: ENQUEUE, DEQUEUE
- A list restricted to add to the end and delete from the front
- Most commonly an array implementation



Stacks: like stacks of plates

- Most commonly an array implementation
- First In Last Out (**FILO**); key ops: PUSH, POP
- Just a list restricted to add items to end and take from the end
- For us, used as “work list” for non-recursive tree walking
- Also reverses a sequence; push Maria,Xue,Tom;
then pop Tom,Xue,Maria

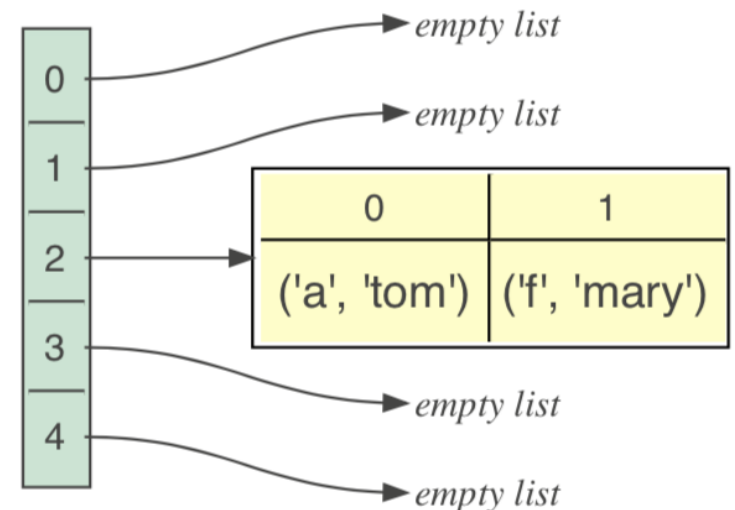


Set: unordered, unique collection

- Typical implementation is a hash table
- Operations are add, delete, contains, union, intersection, etc...
- “contains” operation takes constant time for hashtable implementation

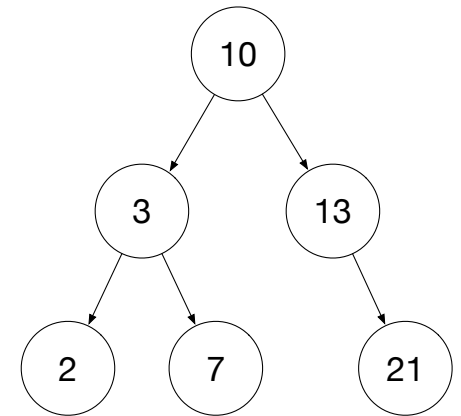
Dictionary abstract data structure

- Maps key to value; i.e., $d[\text{key}] = \text{value}$
- Look up values by key; i.e., $d[\text{key}]$
- Hashtable is implementation of choice
- Recall hashtable is array of buckets, each bucket is array of (key,value) pairs
- Hashcode is function of key then mod with $\text{len}(\text{htable})$ to get bucket index; add key/value pair to that bucket

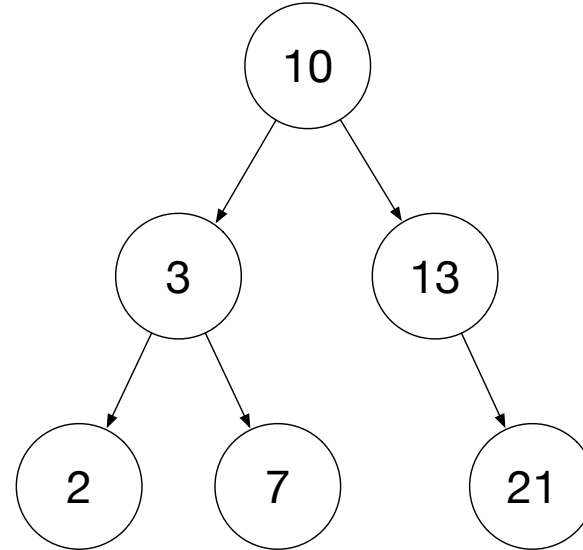
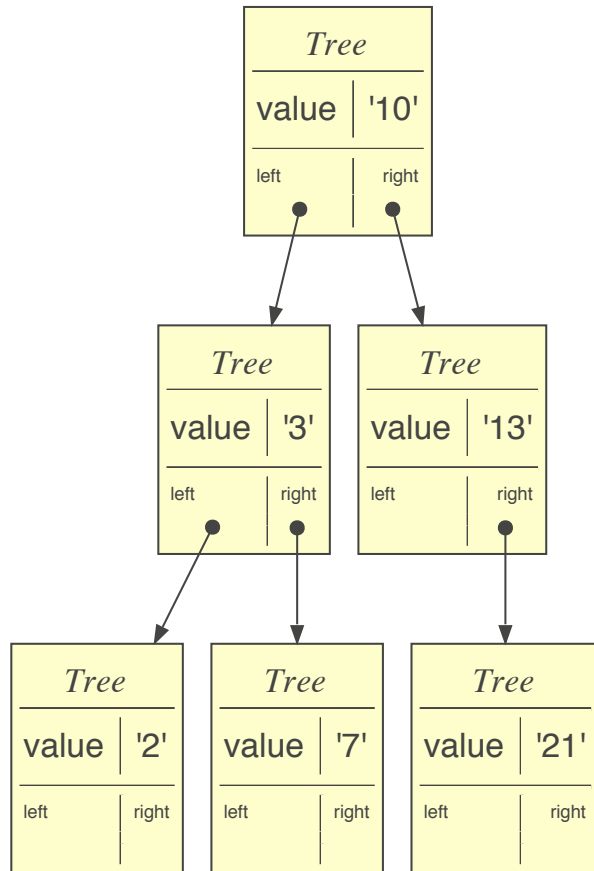


Binary tree abstract data structure

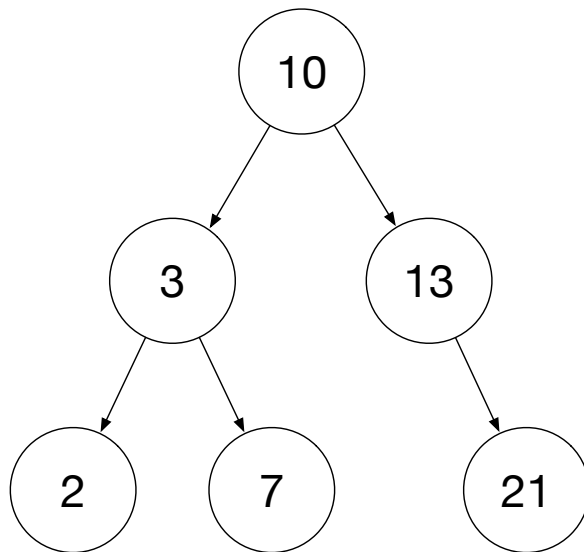
- A directed-graph with internal nodes and leaves
- No cycles and each node has at most one parent
- Each node has at most 2 child nodes
- For n nodes, there are $n-1$ edges
- A *full* binary tree: all internal nodes have 2 children
- Height of full tree with n internal nodes is about $\log_2(n)$
- Height defined as number of edges along path root->leaf
- Level 0 is root, level 1, ...
- Note: binary tree doesn't imply *binary search tree*



Binary tree implementation using pointers



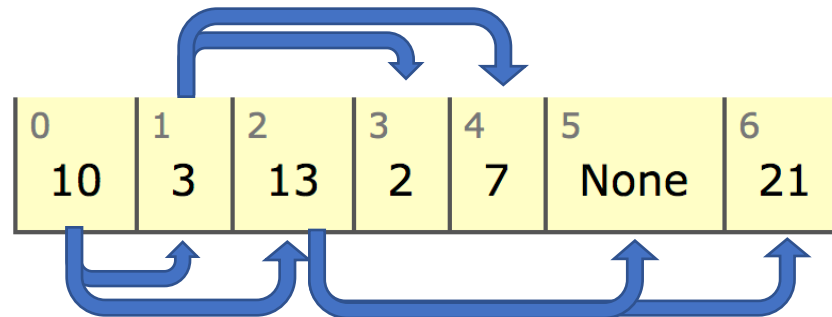
Concrete binary tree using contiguous array



We don't need child pointers!

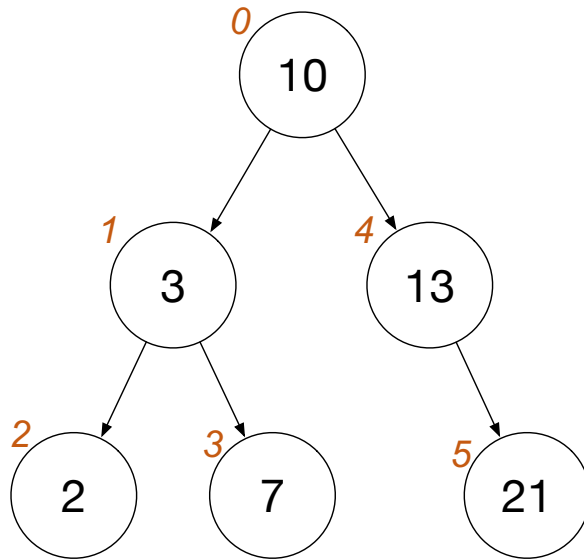
left child is $2i+1$

right child is $2i+2$



Indexes as pointers

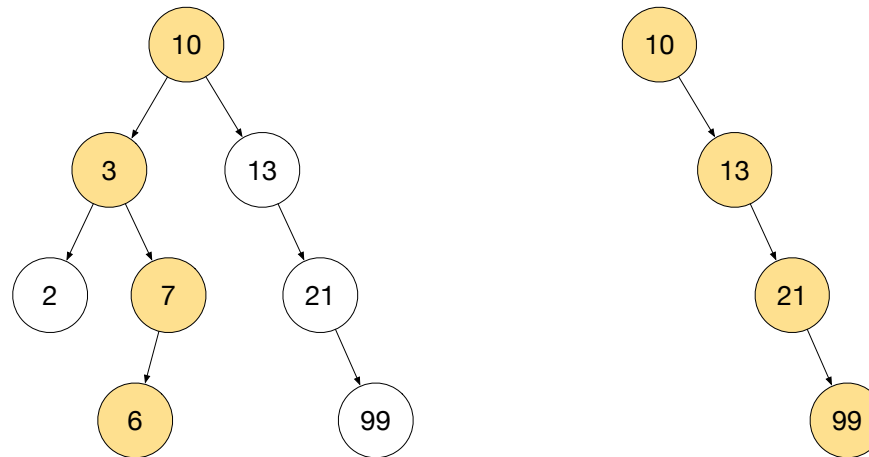
- sklearn doesn't use nodes with pointers
- Uses node **ID's** and parallel arrays like *left*, *right*, *value*



left[0] = 1	right[0] = 4	value[0] = 10
left[1] = 2	right[1] = 3	value[1] = 3
left[2] = -1	right[2] = -1	value[2] = 2
left[3] = -1	right[3] = -1	value[3] = 7
left[4] = -1	right[4] = 5	value[4] = 13
left[5] = -1	right[5] = -1	value[5] = 21

Binary search trees

- Nodes have values
- Elements in left subtree are all less than node's value, all elements in the right subtree are greater than the node's value



Constructing binary search trees

- Result of add() function is the modified tree

```
def add(p:TreeNode, value) -> TreeNode :  
    if p is None:          return TreeNode(value)  
    if value < p.value:     p.left = add(p.left, value)  
    elif value > p.value:   p.right = add(p.right, value)  
    return p # do nothing if equal (already there)
```

- Initial condition: p is None: `root = add(None, 9)`
- If node.value==value, return that node:

`root = add(root, 9)`

TreeNode	
value	'9'
left	right



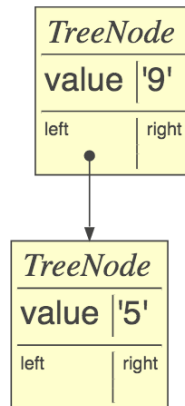
Constructing binary search trees cont'd

- If $\text{value} < \text{current node}$, add to the left

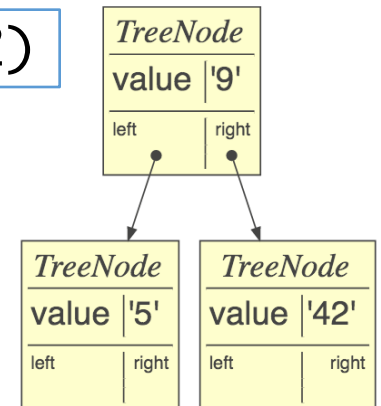
```
if value < p.value: p.left = add(p.left, value)
```

- If $\text{value} > \text{current node}$, add to the right

```
root = add(root, 5)
```



```
root = add(root, 42)
```



Consider similarity of search / build

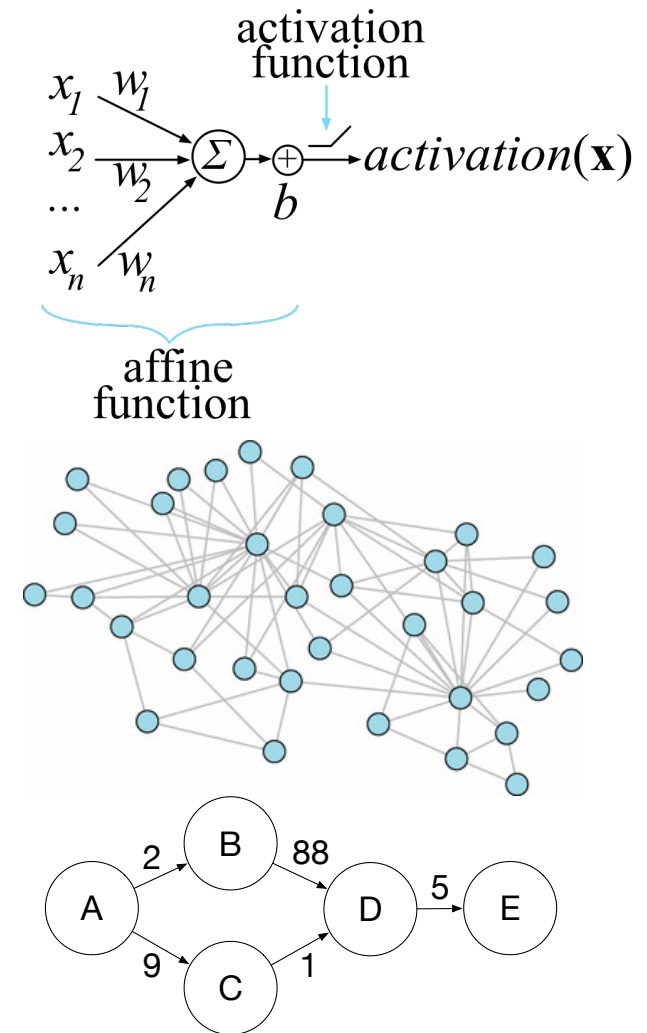
```
def search(p:TreeNode, x:object):  
    if p is None: return None  
    if x < p.value:  
        return search(p.left, x)  
    if x > p.value:  
        return search(p.right, x)  
    return p
```

```
def add(p:TreeNode, v):  
    if p is None: return TreeNode(v)  
    if v < p.value:  
        p.left = add(p.left, v)  
    elif v > p.value:  
        p.right = add(p.right, v)  
    return p
```



Graphs

- An arbitrary number of outgoing edges, not just 2 like binary trees
- Or, use an adjacency matrix
- Edges can have labels or not
- Edges can be directed or undirected
- Can be pointed at by any number of nodes
- Cycles are ok unless specified otherwise; e.g., directed acyclic graph (DAG) is a semi-common term



blue image from <https://skymind.ai/wiki/graph-analysis>

Basic node definitions (Tattoo these somewhere)

```
class LLNode:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next
```

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

```
class Node:
    def __init__(self, value):
        self.value = value
        self.edges = [] # outgoing edges
```

only edges differ

Summary

- Abstract data types:
List, Set, Queue, Stack, Dictionary, Binary tree, Graph
- Concrete implementations:
arrays, linked lists, node object with 1+ outgoing edge pointers
- The questions you must ask of the data dictates the data structure and algorithms you need
- Waste processor, memory power before brainpower
(start with simplest data structure that will work)