ECS 032B: Introduction to Data Structures

Course Material Summary

University of California at Davis

Last Edit Date: 03/13/2022

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. Although most of the content in this summary is originally written by the creator, there may be still some of the content that is adapted (derived) from the slides and codes from *Professor Kurt Eiselt*. We use those as references and quotes in this file. Please contact us to delete this file if you think your rights have been violated.
3. This work is licensed under a Creative Commons Attribution 4.0 International License.

ECS 032B Summary

Chapter 3: Analysis

1. Big-O (worst-case time complexity, upper bound)

Steps to find Big-O: 1) Find T(n) first. 2) Let **T(n) ≤ O(n).** 3) Simplify the equation. 4) Plug values of n (start from 1) and c into the simplified function. 5) T(n) ≤ O(n) satisfied, we conclude that it is true at n and c.

2. Big-Ω (lower bound)

Steps to find Big-Ω: 1) Find T(n) first. 2) Let **T(n) ≥ O(n).** 3) Simplify the equation. 4) Plug values of n (start from 1) and c into the simplified function. 5) T(n) ≤ O(n) satisfied, we conclude that it is true at n and c.

3. Big-θ (equal)

T(n) is θ(n) if and only if T(n) is O(n) and T(n) is Ω(n). (Cannot write T(n) = O(n) or T(n) = Ω(n) because describes a set of functions)

4. Function increase rate (from slowest to fastest)

1 (constant), $\log n$ (logarithmic), n (linear), n log n (log linear), n^2 (quadratic), n^3 (cubic), 2^n (exponential), n! (factorial). (We cannot say one is always faster than another one since the slow one might grow faster in the beginning.)

Chapter 4: Basic Data Structures

1. Abstract Data Structure (ADT)

An abstract data type or ADT is a combination of a collection of data items a set of operations on those items. (ex: python list operations)

The names of the parameters declared in a method header are called formal parameters, and the values passed to the method are called actual parameters.

2. Class and objects

Class blueprint, objects are built from blueprint. Objects derive from class.

3. Encapsulation: Easy for other people to understand, some people are lazy.

4. Stack (Last in first out, LIFO)

`Stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.

`push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.

`pop()` removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

`peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

`isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.

`size()` returns the number of items on the stack. It needs no parameters and returns an integer.

Stack Operation	Stack Contents	Return
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4, 'dog']	
s.peek()	[4, 'dog']	'dog'
s.push(True)	[4, 'dog', True]	
s.size()	[4, 'dog', True]	3
s.isEmpty()	[4, 'dog', True]	False
s.push(8.4)	[4, 'dog', True, 8.4]	
s.pop()	[4, 'dog', True]	8.4
s.pop()	[4, 'dog']	True
s.size()	[4, 'dog']	2

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

5. Queue (First in first out, FIFO)

`Queue()` creates a new queue that is empty. It needs no parameters and returns an empty queue.

`enqueue(item)` adds a new item to the rear of the queue. It needs the item and returns nothing.

`dequeue()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.

`isEmpty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

`size()` returns the number of items in the queue. It needs no parameters and returns an integer.

Queue Operation	Queue Contents	Return
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog', 4]	
q.enqueue(True)	[True, 'dog', 4]	
q.size()	[True, 'dog', 4]	3
q.isEmpty()	[True, 'dog', 4]	False
q.enqueue(8.4)	[8.4, True, 'dog', 4]	
q.dequeue()	[8.4, True, 'dog']	4
q.dequeue()	[8.4, True]	'dog'
q.size()	[8.4, True]	2

```
class Queue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

6. Deque

`Deque()` creates a new deque that is empty. It needs no parameters and returns an empty deque.

`addFront(item)` adds a new item to the front of the deque. It needs the item and returns nothing.

`addRear(item)` adds a new item to the rear of the deque. It needs the item and returns nothing.

`removeFront()` removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.

`removeRear()` removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.

`isEmpty()` tests to see whether the deque is empty. It needs no parameters and returns a boolean value.

`size()` returns the number of items in the deque. It needs no parameters and returns an integer.

Deque Operation	Deque Contents	Return
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear('dog')	['dog', 4]	
d.addFront('cat')	['dog',4, 'cat']	
d.addFront(True)	['dog',4,'cat',True]	
d.size()	['dog',4,'cat',True]	4
d.isEmpty()	['dog',4,'cat',True]	False
d.addRear(8.4)	[8.4,'dog',4,'cat',True]	
d.removeRear()	['dog',4,'cat',True]	8.4
d.removeFront()	['dog',4,'cat']	True

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0,item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

7. Unordered Link List

`List()` creates a new list that is empty. It needs no parameters and returns an empty list.

`add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.

`remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.

`search(item)` searches for the item in the list. It needs the item and returns a boolean value.

`isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.

`size()` returns the number of items in the list. It needs no parameters and returns an integer.

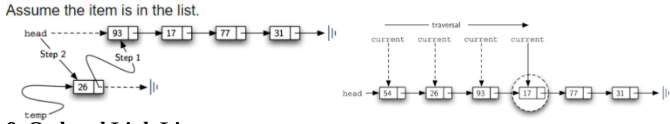
`append(item)` adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.

`index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.

`insert(pos,item)` adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.

`pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.

`pop(pos)` removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.



8. Ordered Link List

`add(item)` adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.

`remove(item)` removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.

`search(item)` searches for the item in the list. It needs the item and returns a boolean value.

`isEmpty()` tests to see whether the list is empty. It needs no parameters and returns a boolean value.

`size()` returns the number of items in the list. It needs no parameters and returns an integer.

`index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.

`pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.

`pop(pos)` removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

Chapter 5: Recursion

1. Definition: Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually, recursion involves a function calling itself.

2. Recursion space and time analysis example

	space complexity	time complexity
recursive factorial	O(n)	O(n)
recursive fibonacci	O(n)	$O(2^n)$

3. Tail Recursion

It often involves the introduction of an additional parameter used as a "variable" to hold the partially-computed result instead of storing postponed computations on the stack. Python cannot save space by using tail recursion.

Chapter 6: Searching and Sorting

1. Sequential (linear) Search

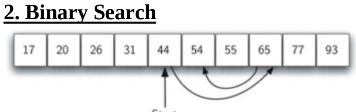
Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

```
def sequentialSearch(alist, item):
    pos = 0
    found = False
    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1
    return found
```

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1
    return found
```

2. Binary Search

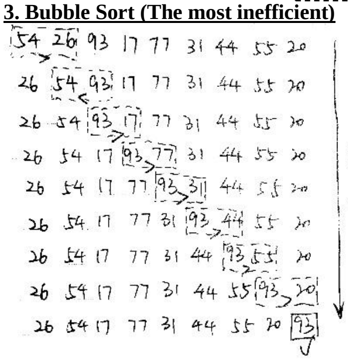


Time Complexity: O (log2 n)

Comparsions	Approximate Number of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

```
def binarySearch(alist, item):
  if len(alist) == 0:
    return False
  else:
    midpoint = len(alist)//2
    if alist[midpoint]==item:
     return True
    else:
     if item<alist[midpoint]:
      return binarySearch(alist[:midpoint],item)
     else:
      return binarySearch(alist[midpoint+1:],item)
```

3. Bubble Sort (The most inefficient)



Comparisons:

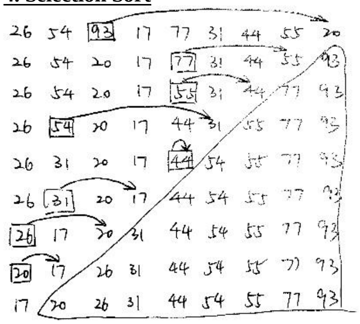
Pass	Comparisons
1	n−1
2	n−2
3	n−3
⋮	⋮
n−1	1

Time Complexity: $O(n^2)$

Worst case: Reverse ordered list

Best case: ordered list.

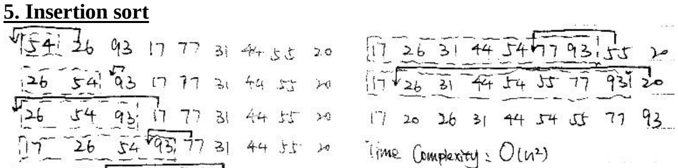
4. Selection Sort



93 is largest
77 is largest
55 is largest
54 is largest
44 is largest
31 is largest
26 is largest
20 is largest
Sorted!

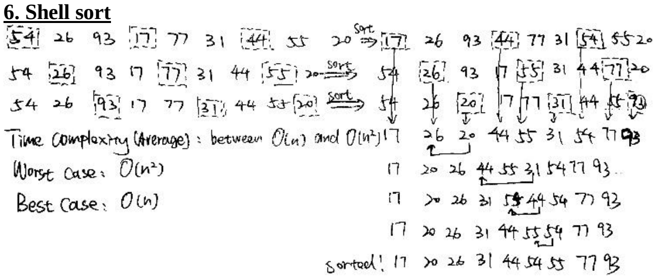
Decreasing example
From right (largest) to smallest (left).
Time complexity: $O(n^2)$
Worst case: Reversed swap.
8 7 6 5 4 3 2 1

5. Insertion sort



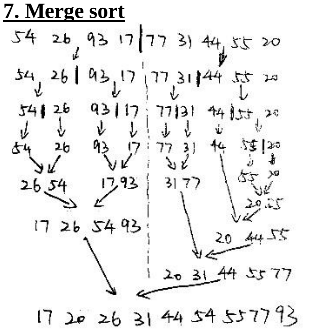
Time Complexity: $O(n^2)$
Worst case: Reversed sorted list
Best case: Sorted list.

6. Shell sort



Time Complexity (Average): between $O(n)$ and $O(n^2)$
Worst case: $O(n^2)$
Best Case: $O(n)$

7. Merge sort



Time complexity: $O(n\log n)$

8. Quick sort

Time Complexity: $O(n\log n)$
Worst: $O(n^2)$ sorted list.

9. Summary of sort

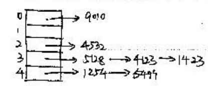
Method	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$O(n)$	$O(n)$ or $O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$ or $O(nlogn)$
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(n)$

10. Hashing

Definition: The hash map is applied to the key first, then the compression map is applied to the result: h(x) = h2(h1(x)).
Formula for hash function: h(k)=(ak+b)%N (a, b are integers, N is array size)

11. Chaining

ex: $h(x) = \text{floor}(x/10) \bmod 5$ (linked list)
Hash: 12540, 51288, 90100, 41233, 54491, 45329, 14236



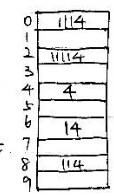
Advantage: 1) Unlimited size 2) Easy to program 3) Easy deletion
Disadvantages: 1) Overhead pointers can be large 2) As the table becomes full, more inefficient (table size N, worst case searching time can be > N).

12. Open Addressing (probe sequence)

1. Linear Probing

$h(k),\ h(k)+R,\ h(k)+2R,\ h(k)+3R,\ \cdots,\ h(k)+(N-1)R$ (all mod N)

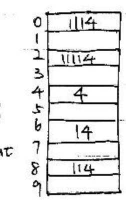
To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).
ex, $h(x) = x \bmod 10$, R=2 (not relatively prime to N)
Hash keys: 4, 14, 114, 1114, 11114
Whether R is relatively prime to n or not, there will still be collisions, and there will still be clustering, but the clusters are not consecutive locations.



2. Probing every Rth location

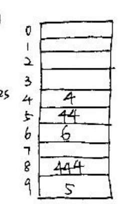
$h(k),\ h(k)+R,\ h(k)+2R,\ h(k)+3R,\ \cdots,\ h(k)+(N-1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).
ex, $h(x) = x \bmod 10$, R=2. (not relatively prime to N)
Hash keys: 4, 14, 114, 1114, 11114
Whether R is relatively prime to n or not, there will still be collisions, and there will still be clustering, but the clusters are not consecutive locations.



3. Quadratic probing

$h(k),\ h(k)+1^2,\ h(k)+2^2,\ \cdots,\ h(k)+(N-1)^2$ (all mod N)
This method avoid concecutive clustering.
Drawback: $1^2 = (N-1)^2 \bmod N$ so the probe sequence examines only about half the Table.
ex: $h(x) = x \bmod 10$
Hash keys: 4, 44, 444, 6, 5
can also use: $h(k),\ h(k)+1^2,\ h(k)-1^2,\ h(k)+2^2, \cdots$



4. Pseudo-random probing and Double hashing

④ Pseudo-random probing
$h(k),\ h(k)+r_1,\ h(k)+r_2, h(k)+r_3, \cdots, h(k)+r_{N-1}$ (all mod N)
Drawback: we must store pseudo-random numbers.

⑤ Double hashing
$h(k),\ h(k)+1h_2(k),\ h(k)+2h_2(k), \cdots,\ h(k)+(N-1)h_2(k)$ (all mod N) if $h_2(k)=0$ set $h_2(k)=1$
A common choice is $h_2(k) = q-(k \bmod q)$ where q is a prime < N
The hope is that if $h(x)=h(y)$, then $h_2(x) \neq h_2(y)$

5. Advantages and disadvantages
Advantages: No memory is wasted on pointers.
Disadvantages: 1) Clusters can run into one another 2) The hash table must be at least as the number of items hashed, and preferably larger 3) Deletion can be a problem. Have to mark empty space with a “tombstone”.
The performance of hashing depends on the quality of hash function, the collision resolution algorithm, and the available space in the hash table.

Chapter 7: Trees and Tree Algorithms

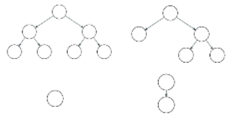
1. Trees

Definition: A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges with only one pathway from the root node to a given node. The tree with no nodes is called the null or empty tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

2. Binary trees

Definition: A binary tree is a tree that is either empty (or null) or each node has a maximum of two children, left subtree and a right subtree.

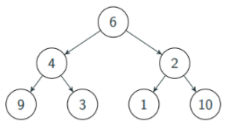
```
class BinaryTree:
  def __init__(self, rootObj):
  self.key = rootObj
  self.leftChild = None
  self.rightChild = None
```



3. Binary tree traversal

1) Preorder traversal: Visit the root; traverse the left subtree; traverse the right subtree.

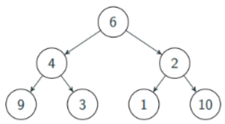
```
def preorder ( root ) :
  if root :
    print ( root . val )
    preorder ( root . left )
    preorder ( root . right )
```



Printed message:
6, 4, 9, 3, 2, 1, 10

2) Inorder traversal: Traverse the left subtree; visit the root; traverse the right subtree.

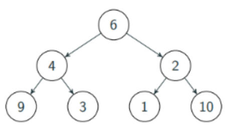
```
def inorder ( root ) :
  if root :
    inorder ( root . left )
    print ( root . val )
    inorder ( root . right )
```



Printed message:
9, 4, 3, 6, 1, 2, 10

3) Postorder traversal: Traverse the left subtree; traverse the right subtree; visit the root.

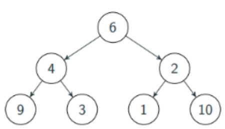
```
def postorder ( root ) :
  if root :
    postorder ( root . left )
    postorder ( root . right )
    print ( root . val )
```



Printed message:
9, 3, 4, 1, 10, 2, 6

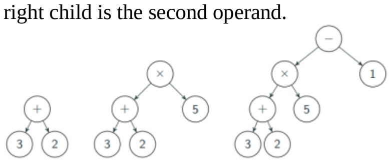
4) Level-order traversal:

```
def levelOrder ( root ):
  q = Queue ()
  q. enqueue ( root )
  while not q . isEmpty () :
    node = q. dequeue ()
    if node :
      print ( node . val )
      q. enqueue ( node . left )
      q. enqueue ( node . right )
```



Printed message:
6, 4, 2, 9, 3, 1, 10

5) Binary expression trees: Root is operator, left child if the first operand, right child is the second operand.



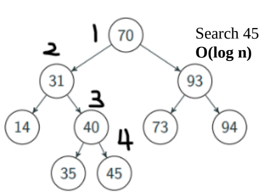
(3+2)*5-1
Postorder: 3 2 + 5 × 1−
Same as postfix notation

4. Binary search tree

Definition: A binary search tree is a tree in which every node is empty or the root of a binary tree in which all the values in the left subtree are less than the value at the root, and all the values in the right subtree are greater than the value at the root.

1) Search

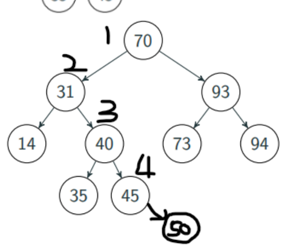
```
def search ( root , target ):
  if not root :
    return None
  if root . val == target :
    return root
  if root . val < target :
    return search ( root . right , target )
  return search ( root . left , target )
```



Search 45
O(log n)

2) Insert

```
def insert ( root , val ):
  if root is None :
    return Node ( val )
  else :
    if root . val == val :
      return root
    elif root . val < val :
      root . right = insert ( root . right , val )
    else :
      root . left = insert ( root . left , val )
    return root
```



3) Delete

Case 1: Node to be deleted is leaf. Simply remove from the tree.

```
      50                               50
   /     \         delete(20)       /   \
  30      70       --------->     30     70
 / \     / \                        \    / \
20  40  60  80                      40  60  80
```

Case 2: Node to be deleted has one child. 1) Change the value of the node to the child's value. 2) Delete the child.

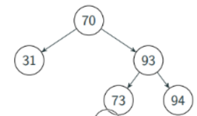
```
     50                             50
  /     \         delete(30)     /    \
30       70       --------->   40      70
  \     / \                           /  \
  40   60  80                        60   80
```

Case 3: Node to be deleted has two children. 1) Find the smallest one in the node's right subtree (inorder successor). 2) Change the value of the node to the value of the inorder successor. 3) Delete the inorder successor recursively.

```
     50                             60
  /     \         delete(50)     /    \
40       70       --------->   40      70
        /  \                             \
       60   80                            80
```

5. Full binary tree

Definition: A binary tree in which every node other than the leaves has to children. In other words, every node has exactly 0 or 2 children.

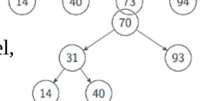


6. Perfect binary tree

Definition: A full binary tree in which all leaves are at the same level, and every parent has two children.

7. Complete binary tree

Definition: A binary tree in which every level, except possibly the last, is completed filled, and all nodes are as far left as possible.



8. Time complexity of search, insert, and delete in binary search tree

O(h), where h is the height of the tree.
- When the BST is almost complete, h = log n.
- When the BST is not complete, h = n.

9. Height

1. Height of a node N is the length of the longest path from N to a leaf node (a node with no child). The height of a leaf node is 0.
2. Height of a tree is the height of its root node. The height of the empty tree is -1. The height of a tree with only a single node is 0.

10. Balanced binary search tree

Balance factor of a node n: the difference between the height of the left subtree and the height of the right subtree.
Left heavy: A tree is left heavy when there is some node n in the tree such that balance(n)>0.
Right heavy: A tree is right heavy when there is some node n in the tree such that balance(n)<0
Perfectly in balance: The balance factor of every node in the tree is 0.

11. Basic ideas of AVL trees

1. The easiest way to keep a tree balanced is never to let it become unbalanced.
2. When a node is inserted (or deleted), the AVL algorithm checks the balance factor of each parent node up the insertion (or deletion) path.
3. If we encounter a node that is out of balance, we need to change the relative height of its left and right subtrees while preserving the BST property.
4. If the subtree is left heavy, then we rotate it to the right. If the subtree is right heavy, then we rotate it to the left.

12. AVL tree rotation rebalance

1. Right rotate the (sub)tree with root y: make y's left child the new root.
2. Left rotate the (sub)tree with root x: make x's right child the new root.

```
     y                                x
    / \     Right Rotation           /  \
   x   T3   - - - - - - - - >      T1    y
  / \       < - - - - - - - -           / \
T1   T2     Left Rotation             T2   T3
```

Insert 50
O(log n)

Case1: Left Left rebalance (y is right child of z and x is right child of y)

```
     z                                y
    /  \                            /   \
  T1    y     Left Rotate(z)       z      x
       /  \   - - - - - - - ->    / \    / \
     T2    x                     T1  T2 T3  T4
          / \
        T3   T4
```

Case 2: Left Left rebalance (y is left child of z and x is left child of y)

```
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Case 3: Left Right rebalance (y is left child of z and x is right child of y)

```
     z                               z                              x
    / \                            /   \                           / \
   y   T4  Left Rotate (y)        x     T4  Right Rotate(z)      y      z
  / \      - - - - - - - - ->    / \        - - - - - - - ->   / \    / \
T1   x                          y   T3                        T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

Case 4: Right left rebalance (y is right child of z and x is left child of y)

```
   z                              z                            x
  / \                            / \                          / \
T1   y   Right Rotate (y)      T1   x      Left Rotate(z)    z     y
    / \  - - - - - - - - ->        / \   - - - - - - - ->   / \   / \
   x   T4                        T2   y                   T1  T2 T3  T4
  / \                                / \
T2   T3                            T3   T4
```

<u>13. Time complexity in AVL tree</u>

1. Given a tree with n nodes, the maximum height of the tree is approximately 1.44logn.
2. Time complexity for searching is O(logn).
3. Time complexity for insertion and deletion is also O(logn).

<u>14. Heap</u>

1. Min heap is a complete binary tree in which if n1 is the parent node of n2, then the value of n1 is less than the value of n2.
2. Max heap is a complete binary tree in which is n1 is the parent node of n2, then the value of n1 is greater than the value of n2.

<u>15. Insertion in a min heap</u>

1. Insert the new item in the next position at the bottom of the heap.
2. While new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap

<u>16. Deletion from a min heap</u>

1. Replace the item at the root node with the last item in the heap (LIH)
2. While item LIH has children and item LIH is larger than at least one child
3. Swap item LIH with the smaller of its children, moving LIH down the heap

<u>17. Time complexity of heap</u>

1. Reheap up and Reheap down: O(h)
2. Number of nodes in a complete binary tree of height h: 2^h <= n <= 2^(h+1)-1
3. Reheap up and Reheap down: O(logn)

<u>18. Implementing a heap</u>

[-, 6, 14, 12, 28, 18, 17, 33, 41, 52, 47, 19, 22, -, -, -]

1. We can find the left child of k at index: 2k
2. We can find the right child of k at index 2k+1
3. We can find the parent of k at index: k/2 or k//2

<u>19. Priority queues</u>

1. Higher priority numbers on the top, lower priority numbers on the bottom. Removing the minimum value (highest priority item) from the priority queue requires Reheap down, which takes O(logn) time. Adding a new item to the priority queue requires Reheap up, which also takes O(logn) time.
2. Optimization on hepify: 1) No need to hepify the leaf nodes 2) From the last non-leaf node to the root, perform the reheap down from each of the nodes. Time complexity O(n).

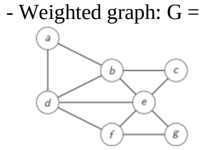
<u>20. Heapsort</u>

1. Heapify: build a heap using the elements to be sorted.
For each item in the sequence to be sorted, add the item to the next available position in the complete binary tree, restore the heap property (using Reheap up)
2. Sort: use the heap to sort the data.
While the heap is not empty, remove the first item from the heap by swapping it with the last item in the heap, reduce the size of the heap by one, restore the heap property.
3. Time complexity: O(nlogn)

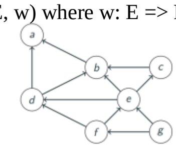
Chapter 8: Graphs

<u>1. Introduction, definition, and terminology</u>

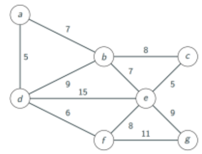
- A graph is a data structure consisting of a finite set of vertices/nodes and a finite set of edges that describer the relationships between the vertices.
- A tree is a specialized graph.
- A G = (V, E) consists of 1) a non-empty set of vertices V, and 2) a set of edges E ⊆ V * V such that e = (u, v) ∈ E means an edge from vertices u to v.
- Undirected graph: If (u, v) ∈ E, then (v, u) ∈ E.
- Directed graph: Edges have directions. (u, v) ∈ E does not mean (v, u) ∈ E.
- Weighted graph: G = (V, E, w) where w: E => R is a weight function.



$G = (V, E)$
$V = \{a, b, c, d, e, f, g\}$
$E = \{(a, b), (a, d), (b, c), (b, e), (b, d), (c, e), (d, e), (d, f), (e, f), (e, g), (f, g)\}$



$G = (V, E)$
$V = \{a, b, c, d, e, f, g\}$
$E = \{(b, a), (c, b), (d, a), (d, b), (e, b), (e, c), (e, d), (f, d), (f, e), (g, e), (g, f)\}$



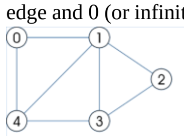
- A <u>vertex</u> v is said to be adjacent to another vertex u if the graph contains an edge (u, v). In other words, if (u, v) ∈ E, then v is adjacent to u.
- A <u>path</u> from vertex v1 to vertex vn is a sequence of vertices v1, v2, ..., vn such that (vi, vi+1) ∈ E for all 1 ⩽ i ⩽ n − 1.
- If there is a path from vertex u to vertex v, we say v is reachable from u.
- A cycle is a path in which the first and last vertices are the same.
- A loop is an edge that connects a vertex to itself.

<u>2. Density: |E|/|V|^2</u>

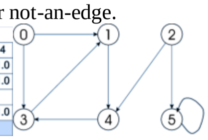
- Sparse graph: 1) Most of the vertices are not adjacent to each other 2) |E| is far less than |V|^2, approximately |V| (ex: Facebook connections)
- Dense graph: 1) Most of the vertices are adjacent to each other 2) |E| is far greater than |V|, approximately |V|^2 (ex: Flights of major cities)

<u>3. Adjacency Matrix and Adjacency List</u>

<u>Adjacency Matrix:</u> A |V| × |V| matrix with 1 (or the weight) (or true) for an edge and 0 (or infinity) (or false) for not-an-edge.

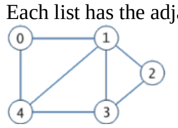


	0	1	2	3	4
0		1.0			1.0
1	1.0		1.0	1.0	1.0
2		1.0		1.0	
3		1.0	1.0		1.0
4	1.0	1.0		1.0	

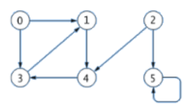


	0	1	2	3	4	5
0		1.0		1.0		
1					1.0	
2					1.0	1.0
3		1.0				
4				1.0		
5						1.0

<u>Adjacency List:</u> A vector (array) of lists, one list for each vertex.
Each list has the adjacent vertices.



0	1, 4
1	0, 4, 3, 2
2	1, 3
3	1, 2, 4
4	0, 1, 3

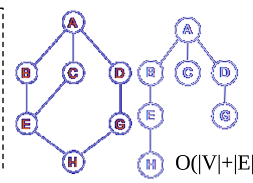


0	1, 3
1	4
2	4, 5
3	1
4	3
5	5

In general, if the graph is dense, the adjacency matrix is better, and if the graph is sparse, the adjacency list is better. Usually, our graphs are sparse, but not always.

<u>4. Breadth-first Search</u>

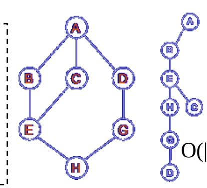
```
While the queue is not empty
  Take a vertex, u, out of the queue (Begin visiting u)
  For all vertices v, adjacent to u,
    If v has not been identified or visited
      Mark it identified (color it green)
      Place it into the queue
  We are now done visiting u (color it orange)
```



O(|V|+|E|)

5. Depth-first Search

```
While the stack is not empty
  Pop a vertex, u, from the stack (Begin visiting u)
  Mark it identified (color it green)
  For all vertices v, adjacent to u,
    If v has not been identified or visited
      Push it on the stack
  We are now done visiting u (color it orange)
```



O(|V|+|E|)