Data Structures + Algorithms = Programs

↓ ways to store data

↓ ways/functions to use data structures to write programs

↓ instructions for computer

Find the best Data structures & Algorithms for a specific task to write great programs

---

Good code is:

1) Readable - Clean code that is maintainable

2) Scalable < Speed (CPU) / Memory (RAM)

Some solutions have a tradeoff between speed & memory
- Sacrificing speed for memory & vice-versa

When a program executes it has 2 ways it uses memory:

1) Heap, where we store variables

2) Stack, where we keep track of function calls

Examples of Good Design:

Instead of keeping data in an array where we need to lookup properties we could instead propagate data into a hashmap to optimize an O(n) time → O(1) time

It's important to keep Big O in mind when writing code

Each D.S. is optimal to its specific use
- each has its own advantage / drawbacks
- each are just different variations of how to store data

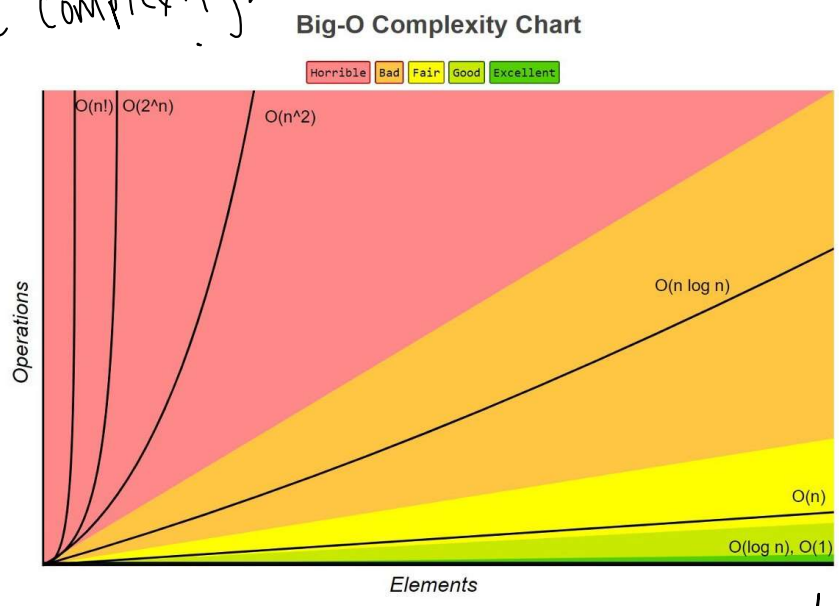
2. Big O Notation

Saturday, January 29, 2022 4:54 AM

*Big O is important for making scalabe code

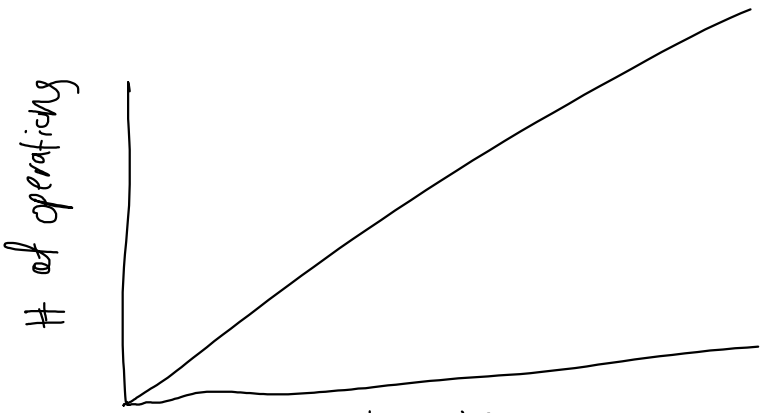
(consists of:
- Time complexity (Runtime efficiency)
- Space complexity

Time complexity:



- O(n) is linear time, b/w # of operations and and elements

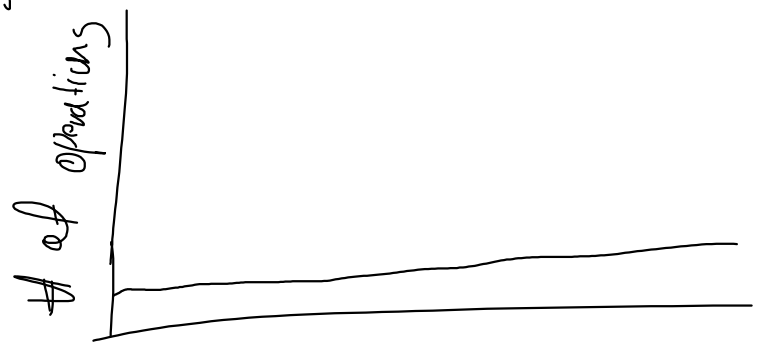
e.g.: for loops & while loops



Elements

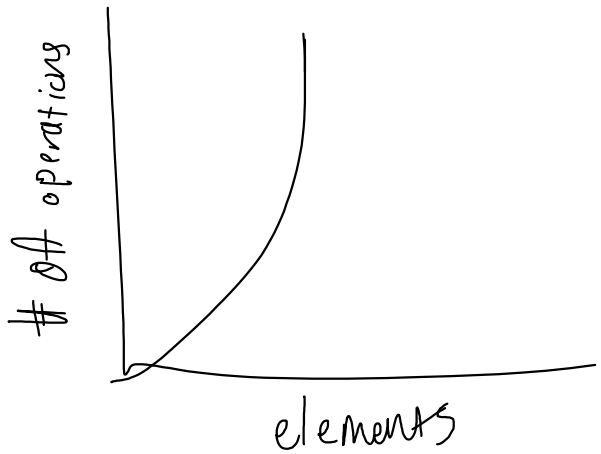
- $O(1)$ is constant time, the operation only does one thing regardless of # of elements

e.g.:



- $O(n^2)$ is quadratic time, every element in a collection needs to be compared to every other element

e.g.: nested for loops



- $O(\log N)$ logarithmic time, used alot for searching algorithms, second best after $O(1)$ time
  e.g.: Binary Search tree, Skip list, AVL tree, etc

- $O(n \log(n))$ log linear time, used alot for sorting algorithms
  e.g. Quick sort, Merge sort, Heap Srt, etc

- $O(2^n)$ Exponential Time, Recursive algorithms to solve problems of size n

- $O(n!)$ Factorial Time, a loop loop is added for each element
  - almost never used
    - in rare cases where you need to calculate all permutations of an array

---

Space complexity:
- only count additional space created after executing your function
  * don't include space taken by inputs

e.g.

1) If we create an array that is dependent on an input's size, n, then the complexity is O(n)

2) If we create an array that is always size = 2, then complexity is O(1), even if our inputs size could be a million.

2.1 Big O Rule Book

Saturday, January 29, 2022 6:29 AM

#1: Always worst case

#2: Remove constants

#3: Different terms for inputs

- If you have 2 separate collections (Inputs):
  - O(n+m) for loops in order
    e.g.: 2 for loops w/ 2 different input arrays
  - O(n * m) for nested loops, Not $O(n^2)$

#4: Drop non dominants terms

$O(n^2 + n)$ becomes $O(n^2)$ because we are only concerned with worst time

---

Things that causes time complexity:

1) operations (+, -, *, %)

2) comparisons (<, >, ==)

3) Loops (for, while)

3) Loops (for, while)

4) Outside function calls (add(a,b))

Things that causes Space complexity:

1) Variables
2) Data Structures
3) Function calls
4) Allocations

Off -How To Approach Problems

Tuesday, February 1, 2022 12:08 AM

Skills interviewers look for:

1) Analytic skills - How do you think through problems
2) Coding skills - is your code clean, organized, readable
3) Technical skills - Do you know the fundamentals
4) Communication skills - is your personality good?

Step by Step:

* Ask high quality clarifying question
1) write down key points (i.e. sorted array)
2) check inputs/outputs
3) code best solution, comment worse alternatives and their complexity
4) Pseudo code first, add comments to code
5) Modularize your code (break it up into beautiful small pieces). break func into separate named methods if applicable, Modularize AFTER or mention it only b/c Sometimes can't for optimal solution
6) think about error checks & if code can be broken
   - check for false inputs
   - Show interviewer some tests to see if function fails (you don't have to actually write them out)
7) Don't use bad names like i and j. write readable code
8) check for no params, 0, undefined, async code

a) talk about possible improvements or things you could google to improve the code

b) If asked: How would you handle if input is too large to fit into memory

Ans: Divide & conquer - perform distributed processing of data and only send certain chunks of input from disk to memory, write output back to disk and combine them later

3. Guidelines

Tuesday, February 1, 2022 2:57 AM

Checklist:

1) Good use of data structures
2) Avoid Code Re-use | Do not repeat yourself (DRY)
3) Less than $O(n^2)$. Avoid nested loops
   - 2 separate loops is better
4) Low space-complexity
   - Recursion can cause stack overflow, copying large arrays may exceed memory.

Guidelines (not always applicable):

1) Hashmaps is often used to improve time complexity
2) Binary Search Tree (BST) is great for sorted arrays w/ $O(\log N)$ time
   - Divide & conquer - divide a data set into smaller chunks & repeating a process w/ a subset of data
3) Try sorting inputs
4) Hashmaps and precomputed data (i.e. sorted arrays) can optimize time well
5) Time & space tradeoff. Sometimes storing extra state in memory can improve runtime
   - Creating Hashmaps uses more space, but can greatly optimize time

4. Overview

Friday, February 4, 2022 1:39 AM

Data Structure is a collection of values

Algorithms are the steps/processes to manipulate these collection of values

---

What is a data structure

- collection of values
  - values can have relationships among them & functions applied to them
- each is specialized for its one purpose

* Metaphorical to different type of storage containers
  - a drawer, refrigerator, backpack
  - each has its own specific uses

//Blockchain is also a data structure

Programming models to real-life scenarios
- organization is akin to data structures

To master data structures:

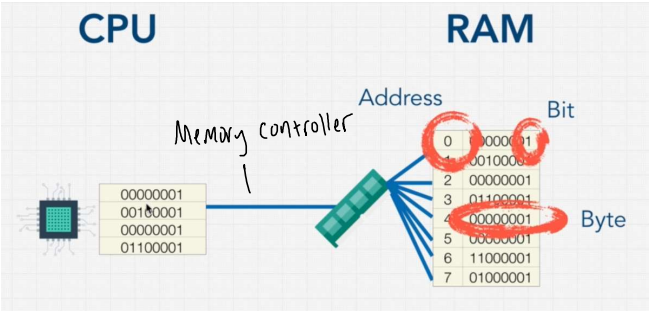
1) How to build one
2) How to use it ← more important, since D.S. is pre-built

Why D.S. is important:

CPU
|
Computations

RAM
|
Temporary storage (fast)
[variables, int a = 1]

Storage
|
Permanent storage (slow)



optimize bits
int vs double
32 vs 64

age | money
using double to store an age is wasteful

The closer the memory, the faster the computer uses it.
- kind of like the CPU cache

- D.S. is an arrangement of data, you define the way you interact with this data and how it is arranged in RAM
  . Some D.S. in RAM are organized next to each other or far away

* Goal is to: minimize the operation needed for the CPU to manipulate the information (insert, delete, ...)

4.1 Operations on Data Structures

Friday, February 4, 2022 2:38 AM

Various operations that can be stored in D.S.:

1) Insertion
2) Deletion
3) Traversal - access each data item at least once, so it can be processed
4) Searching - find out location of data item if it exists in a collection
5) Sorting
6) Access (lookup) - how we access data we have in computer

5. Arrays

Friday, February 4, 2022 2:49 AM

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

- organizes items sequentially in memory
  index 0 -> n
- Good for iterating if you know the indices
* optimal for access, indexing (access element by index number), O(1)
  • insert/delete/search is O(n)

// Array's push is O(1)

- Treat strings as array of characters

2 Types

1) Static - fixed size

2) Dynamic - reserved size for additional elements and can grow in size

String reversal practice

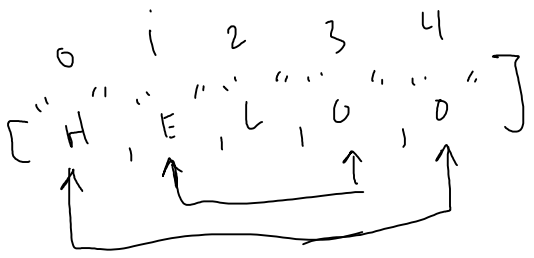
Input = ["H", "E", "L", "O", "O"]

output = ["O", "O", "L", "E", "H"]

Approach:

2 pointer approach

needs: temp value for storage, divide loop length by 2




```
class Solution {
  public void reverseString(char[] s) {
    for (int i = 0; i < s.length / 2 ; i++) {
      char temp = 'a'; // Saves placeholder value for swapping
        temp = s[i];
```

```
        s[i] = s[s.length - (1 + i)];
        s[s.length - (1 + i)] = temp;
    }
  }
}
```

Time: O(n/2) -> O(n), because we have 1 for loop that is dependent on size of input

Space: O(1), because we create a constant space char

Merge Sorted Arrays Practice

Input:

- 2 integer arrays, nums1 and nums2, sorted in non-decreasing order
- 2 integers, m and n, representing number of elements in nums1 and nums2 respectively

Output: Merge nums1 and nums2 into a single array sorted in non-decreasing order

Assume: final sorted array should be stored inside nums1. nums1 has a length of m+n, where m are the elements that should be merged, and the last n elements are arbitrary and set to 0. nums2 is length of n

Example 1:

Input: nums1 = [1,2,3,0,0,0] m = 3

nums2 = [2,5,6], n = 3

output: nums1 = [1,2,2,3,5,6]

Approach 1:

create temp array with size m, do sort & insert of nums1[i] and nums2[j] into temp array then copy into nums1

Time: O(n), b/c 1 for loop

Size: O(n), b/c we're creating an array dependent on input size

---

Approach 2: fill values starting the back of nums1 to avoid shifting elements

3-pointer approach, each pointer at the end of both arrays,

Compare values at pointers and insert the larger value, decrement the pointer that was inserted

Edge case: pointers < 0, to avoid out of bounds error if any of the arrays are empty or when pointer goes to end of array

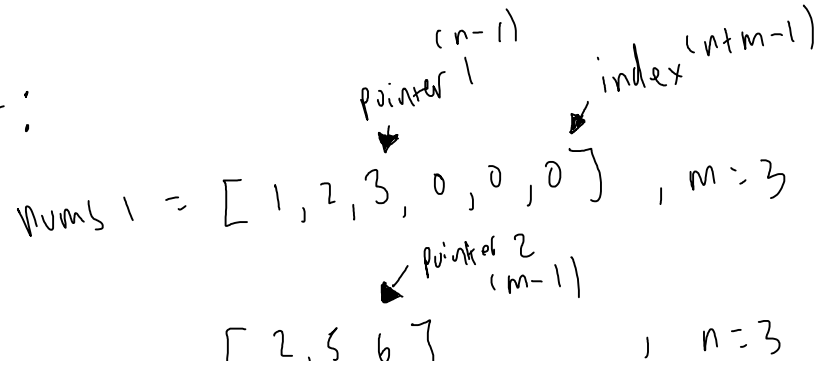
e.g. input: nums1 = [1], m=1 | nums2 = [], n=0

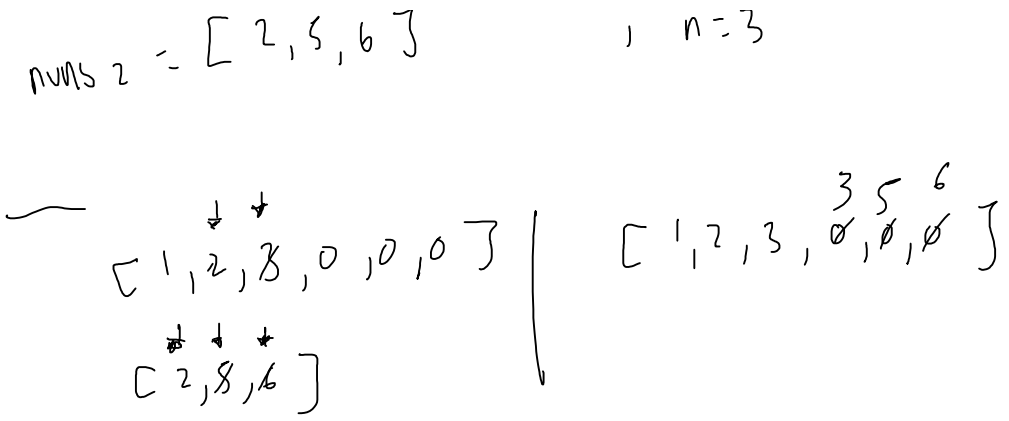
output: 0

input: nums1[0], m=0 | nums2 = [1], n=1

output: [1]

Input:





```
class Solution {
   public void merge(int[] nums1, int m, int[] nums2, int n) {
      // 3 Pointer Approach
      int pointer1 = m-1;
      int pointer2 = n-1;
      int index = m + n -1;

      while (index >= 0) {
         // Edge case check if one of the arrays are empty or when
pointer goes to the end of an array
         if (pointer1 < 0 ) {
            nums1[index] = nums2[pointer2--];
            }
         else if (pointer2 < 0){
            nums1[index] = nums1[pointer1--];
         }

         // Comparison because we are not the end of either arrays
         else{
            if (nums1[pointer1] > nums2[pointer2]){
               nums1[index] = nums1[pointer1--];
            }
            else{
               nums1[index] = nums2[pointer2--];
            }
         }
         // Decrement after checks
         index--;
      }
   }

   // O(n+m) time, because it is dependent if n>m or n<m, O(max(n,m))

   // O(1) space, because space created is constant
```

6. Hashmaps

Friday, May 6, 2022 3:05 AM

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
^ stores only keys, not values

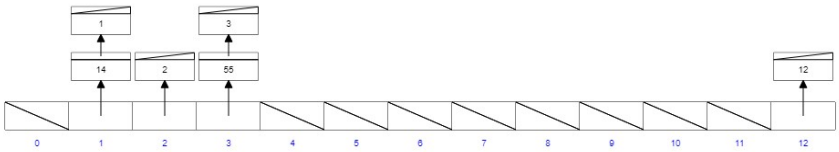
- Key-Value Pair
  key is used as index of where to find the value in memory
  - uses a hash function to generate to memory e.g. MD5

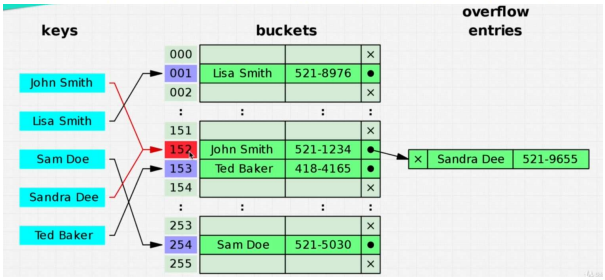
- useful in databases & caches

* optimal for insert, delete, search, O(1) on average

https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

* Main problem is hash collision





- Basically multiple things will be stored at the same address memory space (152 John & Sandra)

- Hash collision will slow down reading and writing with O(n/k) -> O(n)
  k is size of your hash table

- Can use linked list to deal with collision

	Arrays	Hashmaps
search	O(n)	O(1)
look up	O(1)	O(1)
insert	O(N)	O(1)

insert O(1)

delete O(n)

First Unique character in a string Practice

Input: string, s,
find the first non-repeating character

Output: return its index or -1 if it doesn't exist

e.g. s = "leetcode",
output: 0, because L is the first non-repeating element

Approach:

1. Populate string into a hashmap, for loop with a repeats value
2. Scan for first unique character, for loop + conditional
repeats 0 = false
1 = repeats from map.containsKey

map

key	value
l	1
e	2
t	1

```
class Solution {
    public int firstUniqChar(String s) {
        // Key, Value
        HashMap<Character, Integer> map = new HashMap<Character,
Integer>();

        // 0 for unique, 1 for repeating
        int repeats = 0;

        // Populate string into hashmap
        for (int i = 0; i < s.length(); i++) {
            if(map.containsKey(s.charAt(i))) {
                map.put(s.charAt(i), repeats+1);
            }
            else {
                map.put(s.charAt(i), repeats);
            }
```

O(n) time with for loops

O(1) space with 26 letters in

```
            map.put(s.charAt(i), repeats + 1);
        }
        else {
            map.put(s.charAt(i), repeats);
        }

    }

    // Scan for first unique character in the string
    for (int i = 0; i < s.length(); i++) {
      if (map.get(s.charAt(i)) == 0){
        return i;
      }
    }
    return -1;
  }
}
```

O(1) space with 26 letters in alphabet

Cleaner solution with map.getOrDefault(key, defaultValue)

```
class Solution {
  public int firstUniqChar(String s) {
    // Key, Value
    HashMap<Character, Integer> map = new HashMap<Character,
Integer>();

    // Populate string into hashmap
    for (int i = 0; i < s.length(); i++) {
      map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 1) - 1);
    }

    // Scan for first unique character in the string
    for (int i = 0; i < s.length(); i++) {
      if (map.get(s.charAt(i)) == 0){
        return i;
      }
    }
    return -1;
  }
}
```

Summary

Pros

Fast lookups

Fast insert

Flexible keys

Cons

unordered

slow key iteration

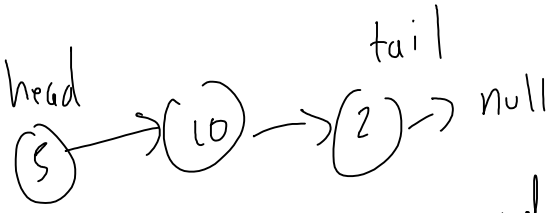
7. Linked Lists

Thursday, May 12, 2022 7:53 AM

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

https://visualgo.net/en/list?slide=1

Nodes
- consists of 2 things:
  - value of data
  - pointer to next node



head is start, tail is end

Why linked list:
- Don't have to shift every element when inserting elements
- while loop to traverse Linked List
  - because linked list doesn't have a fixed size, you just have to hit null
- Delete nodes is easier vs array
- L.L. has sequential order, which can be sorted unlike hashmaps

* prepend O(1) - beginning of L.L.
append O(1) - end of L.L.
lookup O(n) - traversal
insert O(n) - only O(n) at worst case
delete O(n)

What is a pointer

- reference to a place in memory / object / node

e.g.

```
int number = 5;
int pointer = number;
```

Java has automatic garbage collection, so you don't have to delete value + pointer if unused

Creating a Linked List

- 3 classes

Node
- int data
- Nuode next

LinkedList
- append
- prepend
- insert
- delete

Main

head -> null , head = null

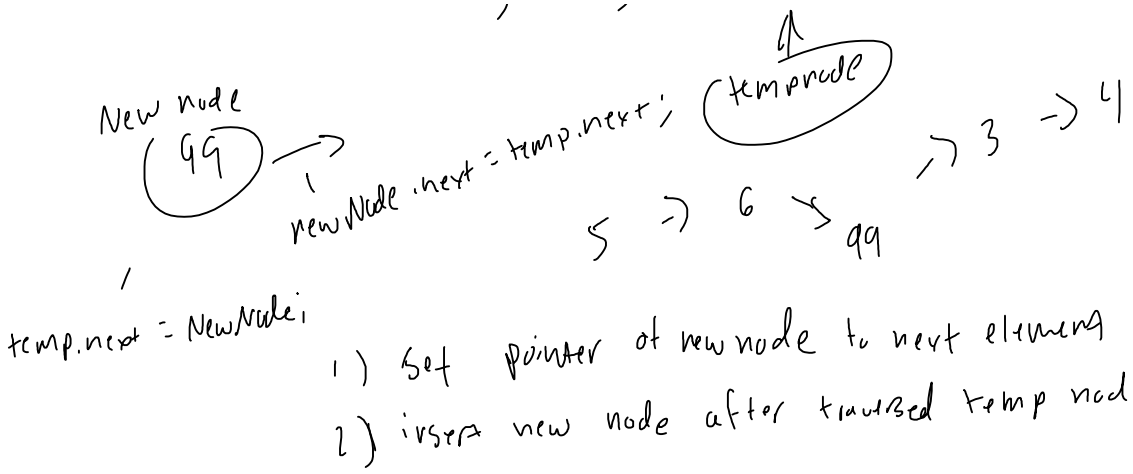
head -> 5 -> 6 -> null

temp node = head (for traverse list) ↗ temp.next != null

insert At(1) 0 1 fail

head -> 5 -> 6 -> 3 -> 4 -> null

↑ tempnode



Singly Linked List

```
public class Node {

   int data;
   Node next; // Refers to the next node

   // Constructor
   Node(int d) {
      data = d;
      next = null;
   }

}
```

```
class Main {
   public static void main(String[] args) {
      LinkedList newList = new LinkedList();

      newList.append(3);
      newList.append(4);
      newList.prepend(2);
      newList.prepend(1);
      newList.insert(3,61);
      newList.show();

      System.out.println("----------");
      newList.delete(2);


      newList.show();

   }
}
```

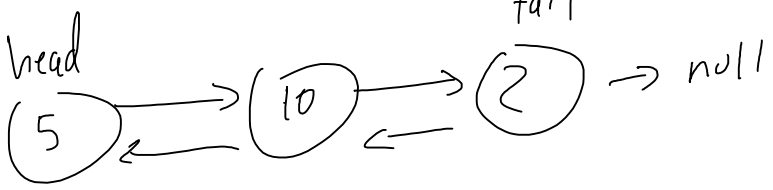
```
public class LinkedList {
   Node head; // refers to first node
   int length = 1;

   // Insert node with input data at end of Linked List
   public Node append(int data) {

      // Create new node with given data
      Node newNode = new Node(data);

      // Check if first node or not, if empty then new node is head
      if (head == null)
      {
         head = newNode;
      }
      else {
         // Traverse to last node then append the new node to the end
         // Temp node refers to head node, then checks the next
         // nodes until it hits last
         Node temp = head;
         while (temp.next != null) {
            temp = temp.next;
         }
         temp.next = newNode;
```

```
        }
        length++;
        return head;
    }

    // Insert node with input data at start of Linked List
    public Node prepend(int data) {

        Node newNode = new Node(data);


        // Let the new Node point to the head
        newNode.next = head;

        // Make new node the head
        head = newNode;

        length++;
        return head;

    }

    // Insert node at position

    public Node insert(int index, int data){

        Node newNode = new Node(data);

        // Edge case checks
        if (index <= 0) {
            prepend(data);
        }
        else if (index >= length) {
            append(data);
        }
        else {
            Node temp = head;
            // Traverse to index where node is to be inserted
            for (int i = 0; i < index - 1; i++) {
                temp = temp.next;
            }

            // new node now points to the next node that temp node
traversed
            newNode.next = temp.next;
            // insert new node after traversed temp node
            temp.next = newNode;


        }

        length++;
        return head;
    }

    public Node delete(int index){
        if (index < 0 ){
            index = 0;
        }
        if (index == 0) {
            head = head.next;
        }


        Node temp = head;
        for (int i = 0 ; i < index -1; i++) {
            temp = temp.next;
```

```
    }
    /* for garbage collection
    Node delete = null;
    delete = temp.next;
    temp.next = delete.next;

     */
    temp.next = temp.next.next;
    length--;
    return head;
  }


  // Prints whole list
  public void show(){
    // Temp node to traverse list
    Node temp = head;

    while (temp.next != null)
    {
      System.out.println(temp.data);
      temp = temp.next;
    }
    // Prints last element
    System.out.println(temp.data);
    System.out.println("length: " + length);
  }


}
```

Doubly Linked List

- Links to node before it with a prev pointer



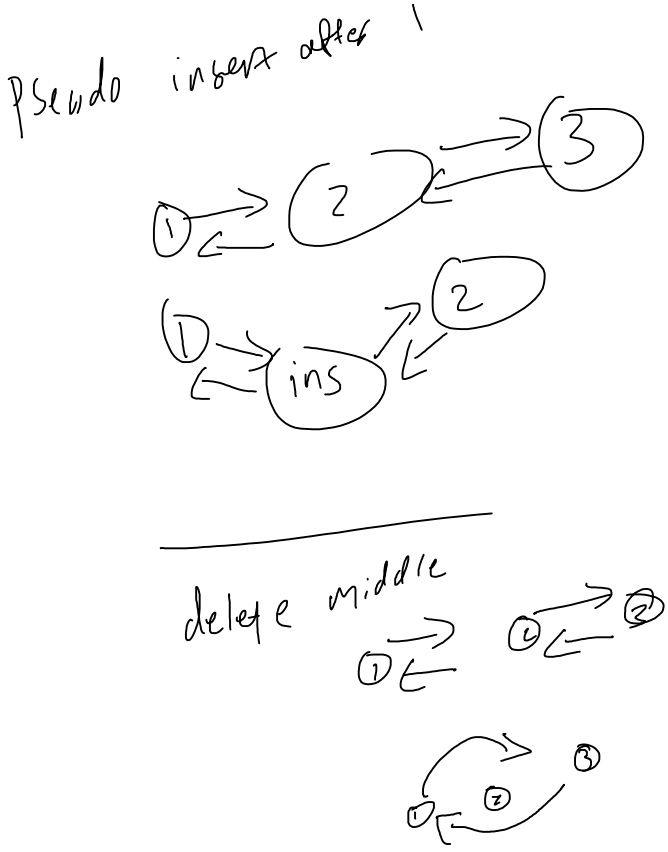
~~*~~ prepend O(1)

append O(1)

lookup O(n/2) → O(n) , traversal

insert O(n)

delete O(n)




```
public class LinkedList {
   Node head; // refers to first node
   int length = 1;

   // Insert node with input data at end of Linked List
   public Node append(int data) {

      // Create new node with given data
      Node newNode = new Node(data);

      // Check if first node or not, if empty then new node is head
      if (head == null)
      {
         head = newNode;
         newNode.prev = null;
         return head;

      }
      else {
         // Traverse to last node then append the new node to the end
         // Temp node refers to head node, then checks the next
         // nodes until it hits last
         Node temp = head;
         while (temp.next != null) {
            temp = temp.next;
         }
         temp.next = newNode; // append newNode to end of list
         newNode.prev = temp; // newNode prev pointer set to node
before
      }
      length++;
      return head;
   }
```

```
public class Node {

   int data;
   Node next; // Refers to the next node
   Node prev; // Refers to prev node

   // Constructor
   Node(int d) {
      data = d;

   }

}
```

```
class Main {
   public static void main(String[] args) {
      LinkedList newList = new LinkedList();

      newList.append(3);
      newList.append(4);
     newList.prepend(2);
      newList.prepend(1);
      newList.insert(2,61);
     newList.show();

      System.out.println("----------");
      newList.delete(2);
```

```
    }
    length++;
    return head;
  }

  // Insert node with input data at start of Linked List
  public Node prepend(int data) {

    Node newNode = new Node(data);


    // Let the new Node point to the head and new node point to null
    newNode.next = head;
    newNode.prev = null;

    // current head prev pointer points to new head
    if (head != null) {
      head.prev = newNode;
    }

    // make newNode the head
    head = newNode;

    length++;
    return head;

  }

  // Insert node at position

  public Node insert(int index, int data){

    Node newNode = new Node(data);

    // Edge case checks
    if (index <= 0) {
      prepend(data);
    }
    else if (index >= length) {
      append(data);
    }
    else {
      Node temp = head;
      // Traverse to index where node is to be inserted
      for (int i = 0; i < index - 1; i++) {
        temp = temp.next;
      }

      // new node now points to the next node that temp node
traversed
      newNode.next = temp.next;
      // insert new node after traversed temp node
      temp.next = newNode;

      // newNode prev pointer points to traversed temp
      newNode.prev = temp;

      // prev pointer of node after inserted new node points to this
new node
      newNode.next.prev = newNode;


    }

    length++;
    return head;
  }
```

```
      newList.show();

    }
  }
```

```
Node@4554617c, data: 1, next:Node@74a14482, prev:null
Node@74a14482, data: 2, next:Node@1540e19d, prev:Node@
4554617c
Node@1540e19d, data: 61, next:Node@677327b6,
prev:Node@74a14482
Node@677327b6, data: 3, next:Node@14ae5a5, prev:Node@
1540e19d
Node@14ae5a5, data: 4, next:null, prev:Node@677327b6
length: 5
----------
Node@4554617c, data: 1, next:Node@74a14482, prev:null
Node@74a14482, data: 2, next:Node@677327b6, prev:Node@
4554617c
Node@677327b6, data: 3, next:Node@14ae5a5, prev:Node@
74a14482
Node@14ae5a5, data: 4, next:null, prev:Node@677327b6
length: 4

Process finished with exit code 0
```

```
  public Node delete(int index){
    if (index < 0 ){
      index = 0;
    }
    if (index == 0) {
      head = head.next;
    }


    Node temp = head;
    for (int i = 0 ; i < index -1; i++) {
      temp = temp.next;
    }
    /* for garbage collection
    Node delete = null;
    delete = temp.next;
    temp.next = delete.next;

     */

    // Deletes node by having current pointer point to node after node
you want to delete
    temp.next = temp.next.next;
    // Prev pointer of the node after your deleted node points to the
node before deleted
    temp.next.prev = temp;
    length--;
    return head;
  }



  // Prints whole list
  public void show(){
    // Temp node to traverse list
    Node temp = head;

    while (temp.next != null)
    {
      System.out.println( temp + ", data: " + temp.data + ", next:" +
temp.next + ", prev:" + temp.prev);
      temp = temp.next;
    }
    // Prints last element
    System.out.println( temp + ", data: " + temp.data + ", next:" +
temp.next + ", prev:" + temp.prev);
    System.out.println("length: " + length);
  }


}
```

Singly

- less memory
- slightly faster

Don't have to

VS

Doubly Linked List

- traversal from front & back
- deleting a prev node is faster
- requires more memory & storage

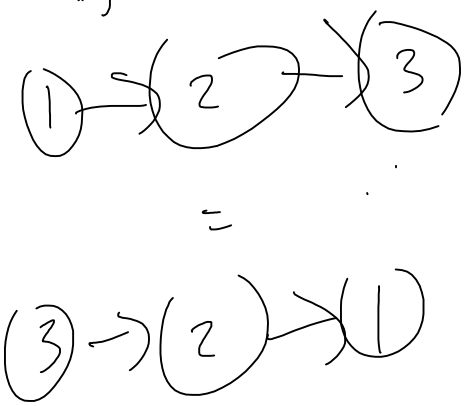
- Slightly faster (b/c we don't have to update pointers)
- good for fast insertion & deletion and you don't need much searching
- requires more memory & storage
- good for searching - forwards & back

Reverse a linked list Practice

input: head of a singly linked list

output: return the reversed list

e.g.

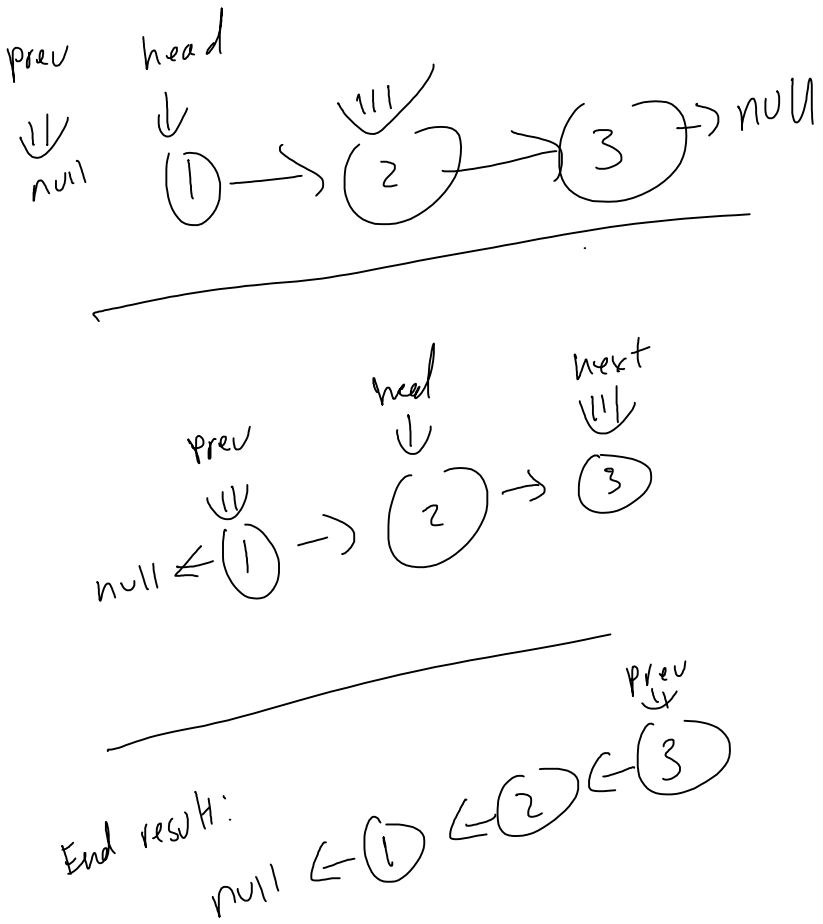


Approach:

Edge case: head == null
return head

... approach ... prev & head & next pointer

3 pointer ...



Node prev = null;

(head != null) {

① Node nextNode = head.next

② head -> prev

③ prev = head

④ head = next

}

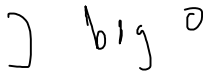
return prev

```
class Solution {
  public ListNode reverseList(ListNode head) {
    // Edge case check
    if (head == null) {
      return head;
    }

    // 3 Pointer Approach
    // Initialize outside because we return prev and it points the
reverse
    ListNode prev = null;
    while (head != null) {
      ListNode nextNode = head.next; // Initialize inside because it's a
temp placeholder
      head.next = prev;
      prev = head;
      head = nextNode;
    }
    return prev;
  }
  // O(n) time, because while loop depends on N size of the
LinkedList.size()
```

- reverse, current head -> prev
- advance nodes

big O

```
        return prev;
    }
    // O(n) time, because while loop depends on N size of the
LinkedList.size()
    // O(1) space, because space created was constant and not
dependent on anything
}
```



8. Stacks and Queues

https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Sunday, May 15, 2022 5:52 AM

https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

- Linear data structure
  - traverse with only 1 element can be reached
    - sequentially one by one
  - push, peak, pop
- higher level vs lower

Stacks (LIFO)

Last in First at

- Browser history (open last closed browser)
- ctrl + z, undo in word

analogy to a stack of plates
- first plate gets removed

Lookup	O(n)
pop	O(1)
push	O(1)
peek	O(1)

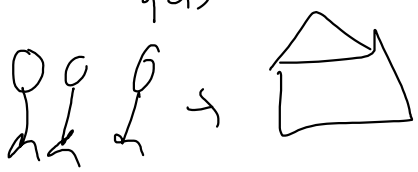
Queues (FIFO)

First in First out

- waiting area to buy concert tickets
- reservation

- restaurant check in, first to make ...
- printer app

analogy to a line at a restaurant, first person gets served



* lookup O(1)

enqueue O(1) push

dequeue O(1) pop

Peek O(1)

Stacks vs queues

building w/

arrays	arrays
• fast b/c memory index is close to each other	• bad b/c shifting of indexes
Linked List	Linked List
• more dynamic size, needs memory for pointer	• same as stacks

---

Stack Implementation w/ Linked List (LIFO)

Node

data

next

Stackey

push

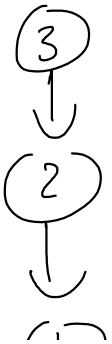
peek

pop

main

{

Test



next peek pop

```
public class Stacky {
   Node top;
   Node bottom;
   int length = 0;

   public Node peek() {
      if (top == null) {
         System.out.println("Stack is Empty");
         return null;
      }
      else {
         System.out.println("Peek " + top.data);
         return top;
      }
   }

   public Node push(int data){
      Node addNode = new Node(data);
      if (this.length == 0) {
         top = addNode;
         bottom = addNode;
      } else {
         // New node will point to previous top, add the new node as the
top
         // Because Stack is LIFO
         addNode.next = top;
         top = addNode;
      }
      length++;
      System.out.println("Push: " + top.data);
      return top;
   }

   public Node pop(){
      if (top == null) {
         System.out.println("Stack is empty");
         return null;
      }
      Node delNode = top;
      top = top.next;
      System.out.println("Popped: " + delNode.data);
      length--;
      return delNode;
   }
}
```

```
public class Node {
   int data;
   Node next;

   Node(int d) {
      data = d;

   }
}
```

```
public class Main {

   public static void main(String[] args) {
      Stacky stack = new Stacky();

      stack.push(5);
      stack.push(10);
      stack.push(20);
      stack.pop();
      stack.peek();
      stack.pop();
      stack.pop();
      System.out.println("length: " + stack.length);
   }
}
```

addNode = top

addNode.next

```
Push: 5
Push: 10
Push: 20
Popped: 20
Peek 10
Popped: 10
Popped: 5
length: 0
```

Queue Implementation Linked List (FIFO)

Node
next

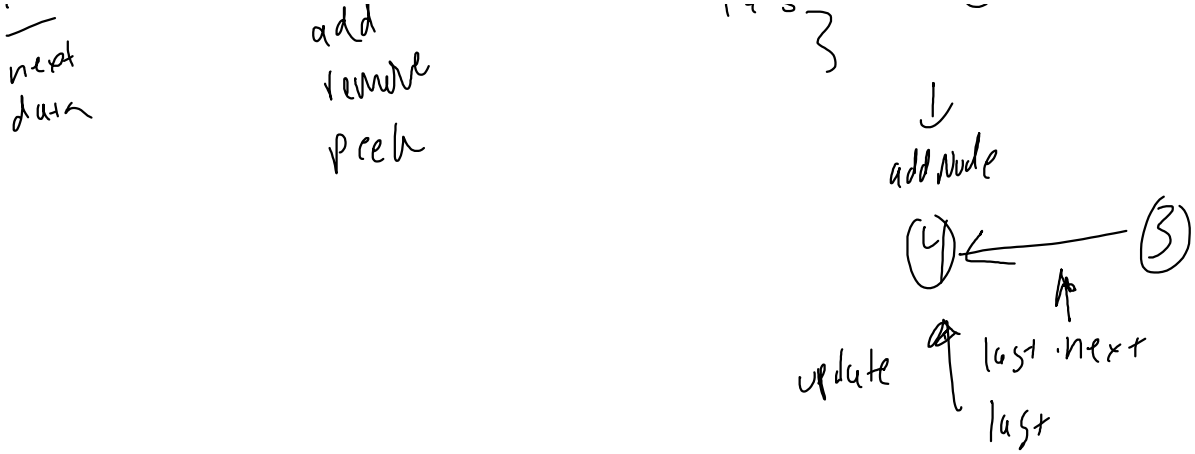
Queue
add
remove

main
Test

Last

Fist

3 ← 2 ← 1




```
public class testQueue {
   Node first;
   Node last;
   int length = 0;

   public Node peek() {
      System.out.println("Peek: " + first.data);
      return first;
   }

   public Node add(int d){
      Node addNode = new Node(d);

      // if queue is empty, this added node should be the first and last
node
      if (first == null) {
         first = addNode;
      }
      // If this isn't the first node ever added, then previous node before
should point to this new node
      if (last != null) {
         last.next = addNode;
      }

      // FIFO, the new node should be in the back of the line
      last = addNode;

      System.out.println("Added node is: " + addNode.data);
      length++;
      return first;
   }

   public Node remove(){
      if (first == null) {
         System.out.println("Queue is empty, nothing to remove");
         return null;
      }

      // Temp placeholder to show removed node
      Node removedNode = first;
```

```
public class Node {
   int data;
   Node next;

   Node(int d) {
      data = d;
   }
}
```

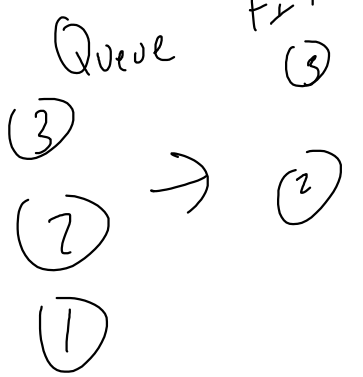
```
 public class Main {

    public static void main(String[] args) {
      testQueue queue = new testQueue();
      queue.add(2);
      queue.add(5);
      queue.add(10);
      queue.remove();
           queue.remove();
           queue.peek();
           queue.remove();
      System.out.println("Length: " +
      queue.length);
    }
 }
```

```
Added node is: 2
Added node is: 5
Added node is: 10
Removed: 2
Removed: 5
Peek: 10
Removed: 10
Length: 0
```

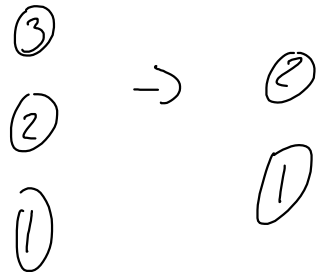
```
    // FIFO, the new first is set next to removedFirst
    first = first.next;

    // If removed node was the last in queue, then set last to null too
    if (first == null) {
      last = null;
    }
    System.out.println("Removed: " + removedNode.data);
    length--;
    return removedNode;
  }
}
```

Implement Queue using Stack Practice

Queue FIFO



Stack LIFO (2 stacks)



Approach:
use one to keep inputs and the other for outputs

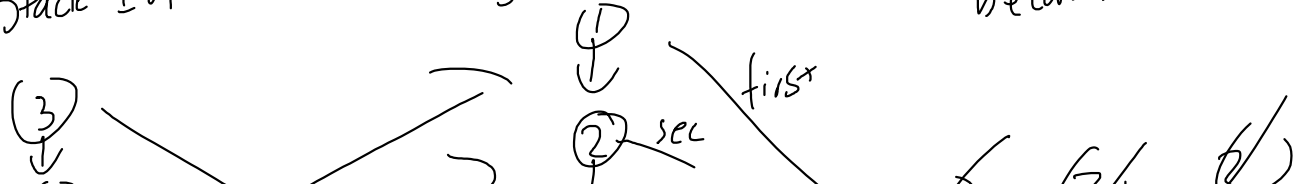
input stack - to push

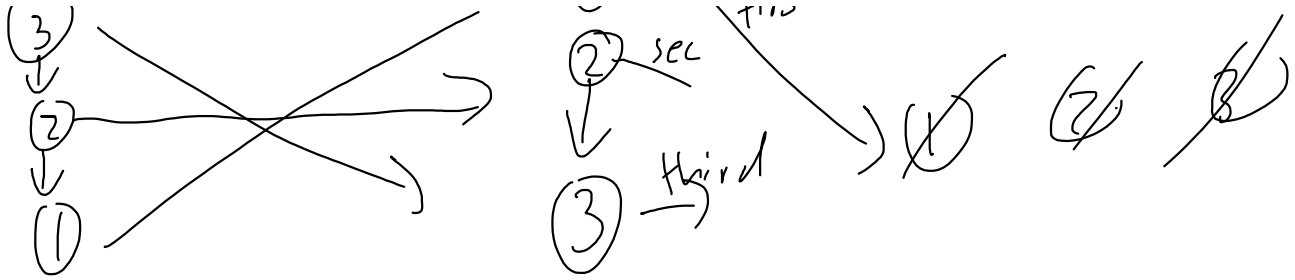
output stack - pop & peek

empty input stack to output stack when pop is called
then pop output stack for FIFO

e.g.

Stack In — pop LIFO — Stack Out — pop LIFO — becomes FIFO





*only refill output stack when it's empty again so you can pop any remaining elements in output stack in FIFO order

```
class MyQueue {

    public MyQueue() {

    }

    // 2 LIFO ----> FIFO
    // One stack handles push, the other handles pop/peek
    // When pop is needed, empty input stack to output stack, and let it
pop out in FIFO order
    Stack<Integer> stackIn = new Stack<Integer>();
    Stack<Integer> stackOut = new Stack<Integer>();
    // Temp declaration in case nothing was ever popped
    int trackTop;

    public void push(int x) {
        if (stackIn.isEmpty()) {
            trackTop = x;
        }
        stackIn.push(x);
    }

    public int pop() {
        // Empty input stack to output stack, it becomes FIFO order
        if (stackOut.isEmpty()) {
        while (!stackIn.isEmpty()){
            stackOut.push(stackIn.pop());
        }
        }

         // Pop the FIFO'd output stack
        return stackOut.pop();
    }

    public int peek() {
        // No values were popped yet, so just return the tracked top
        if (stackOut.isEmpty()) {
            return trackTop;
        }
        return stackOut.peek();
```

O(1) time b/c O(1) for push and O(1) for pop amortized over a lot of

```
            if (stackOutIsEmpty()) {
                return trackTop;
            }
            return stackOut.peek();
        }

        public boolean empty() {
            return stackIn.isEmpty() && stackOut.isEmpty();
        }
    }
```

and O(1) for pop amort
(on avg over a lot of operations)

O(1) Space b/c space was constant

Better Solution Found:

```
class MyQueue {

    Stack<Integer> input = new Stack();
    Stack<Integer> output = new Stack();

    public void push(int x) {
        input.push(x);
    }

    public void pop() {
        peek();
        output.pop();
    }

    public int peek() {
        if (output.empty())
            while (!input.empty())
                output.push(input.pop());
        return output.peek();
    }

    public boolean empty() {
        return input.empty() && output.empty();
    }
}
```

Summary:

Pros

Fast insert & remove

Cons

slow lookup

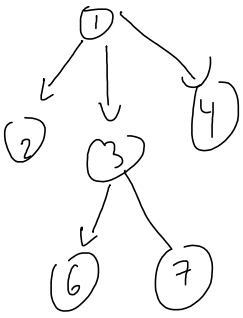
Fun peek
ordered

9. Trees

https://visualgo.net/en/bst

Monday, May 16, 2022 7:01 AM

Hierarchical Data Structure



root is Topmost node
- 1

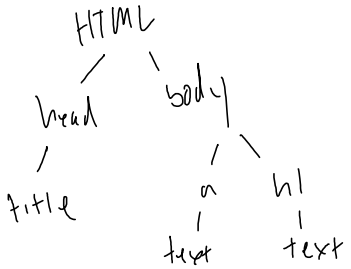
Parents are Elements directly above elements
- 1, 3

Children are Elements directly below elements
- 2 3, 4 and 6, 7

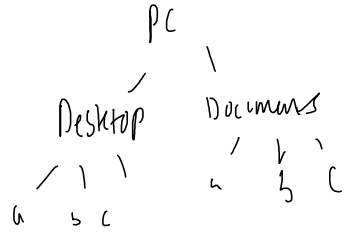
leaves are Elements without children
- 2, 4, 6, 7

Siblings are elements that share the same parent
- 2, 3, 4 and 6, 7

Examples: HTML DOM



PC File directory



Linked Lists are linear trees

Binary Trees

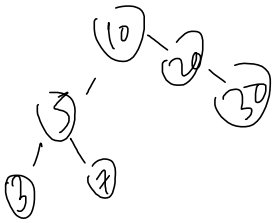
1) Each node can only have 0, 1, or 2 child node
2) Each child has only one parent

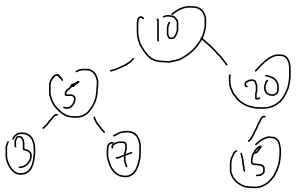
Types:

incomplete binary tree vs

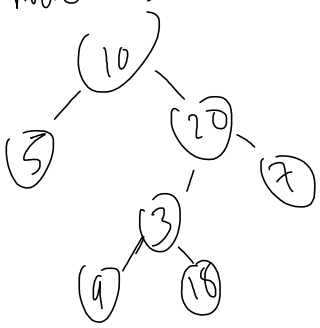
Full Binary Tree

- every level is fully filled, except for perhaps the last level. the last level is filled left to right.

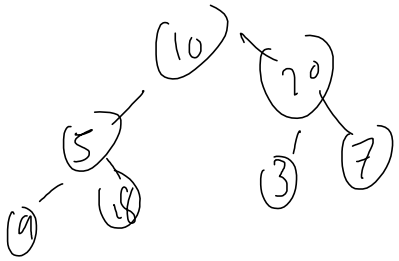




Full Binary Tree
- every node has either 0 or 2 children. No node has 1 child



Perfect Binary Tree ($2^k - 1$ nodes, half of nodes at bottom)
- Both full and complete. all leaf nodes are at the same level and this level has the max amount of nodes

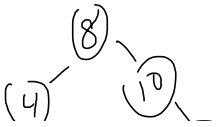


* Binary Search Tree

https://visualgo.net/en/bst

	Best	Worst
. lookup	O(log N)	O(n)
. insert	O(log N)	O(n)
. Delete	O(log N)	O(n)

- BST is a binary tree in which every node is: all left descendants <= n < all right descendants
  - This applies to all descendants and not just immediate children



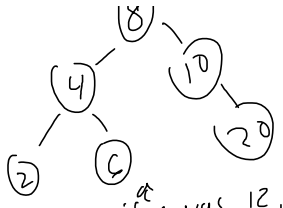
children

level 0 : $2^0 = 1$

1 : $2^1 = 2$

2 : $2^2 = 4$

3 : $2^3 = 8$



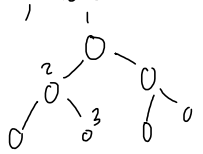
if 6 was 12, then invalid b/c 12 is left of 8

# of total nodes = $2^h - 1$, where h = height

log nodes = steps

log 100 = 2 because $10^2 = 100$

log N, based on height, the max # of decisions is log N

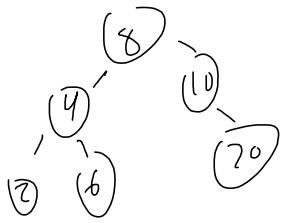


max of 3 steps instead of 8

b/c of divide & conquer

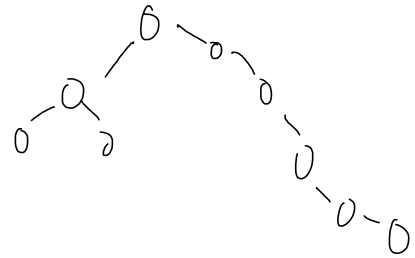
- analogous to looking through a phonebook
  - you go by names, sections, not entire book

- good for searching, eg. finding 6



1. 6 < 8, so go left
2. 6 > 4, so go right

- bad for insert & delete b/c if there are a lot of child nodes then you have to shift a lot of nodes
  - or trees can be really unbalanced, which is O(n)



BST

Pros

- n log than O(n)

Cons

- no O(1) operations
- will need to iterate

Pros

- Better than O(n)
- Ordered (sorted data)
- Flexible size

Cons

- No O(1) operations
  - b/c you need to iterate

BST Implementation

Node

- data
- left pointer
- right pointer

BST

- lookup
- insert
- delete

Main

- Test

Insert(value) & lookup(value) pseudocode

- Declare temp node, currentNode, for traversal
- while loop or recursive

```
if (value < currNode.data) {
    if (currNode.left == null) {      // insert if empty
        currNode.left = newNode;
    }
    currNode = currNode.left          // keep traversing
}
```

Delete

- temp node for traversal
- temp node to keep reference of parent node that you want to delete
- while / recursive

...to it's parent

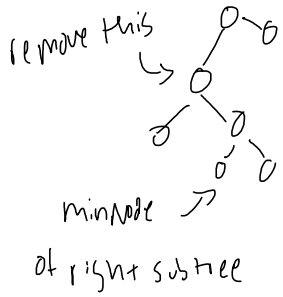
1) traverse while tracking currNode & ...

2) Match

3 cases:

1 child [1) no left child: overwrite w/ right child
2) no right child: overwrite w/ left child

3) 2 child: find min node in right subtree,
append currNode.left (node to be deleted) to minNode.left
so it saves deleted node's left subtree




```
public class binarySearchTree {
   Node root = null;

   // For print2D function
   static final int COUNT = 10;


   public Node insert(int value){
      // Initialize new node to be inserted
      Node newNode = new Node(value);

      // Set new node as root if tree is empty
      if ( root == null) {
         root = newNode;
         return newNode;
      }

      // set temp currentNode for traversal
      Node currentNode = root;

      while(true) {
         // left
         if (value < currentNode.data) {
            // Insert if left of traversed node is empty
            if (currentNode.left == null) {
               currentNode.left = newNode;
               return newNode;
            }
            // if not empty, keep traversing
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data) {
            // Insert if right of traversed node is empty
            if (currentNode.right == null) {
               currentNode.right = newNode;
               return newNode;
            }
```

```
   public class Node {
      int data;
      Node left;
      Node right;

      Node(int d) {
         data = d;
      }
   }



public class Main {

   public static void main(String[] args) {
      binarySearchTree tree = new binarySearchTree();
      tree.insert(8);
      tree.insert(4);
      tree.insert(2);
      tree.insert(6);
      tree.insert(10);
      tree.insert(20);
      System.out.println(tree.lookup(4));
      System.out.println(tree.lookup(9));

      tree.print2D();
      tree.remove(4);
      System.out.println("------------------------------------");
      tree.print2D();
   }
}
```

```
Key 4 exists:
true
Key 9 exists:
false
```

```
            if (currentNode.right == null) {
               currentNode.right = newNode;
               return newNode;
            }
            // If not empty, keep traversing
            currentNode = currentNode.right;
         }
      }


   }

   // Check if the node/value ur existing for exists

   public boolean lookup(int value) {
      System.out.println("Key " + value + " exists: ");
      if (root == null) {
         return false;

      }
      Node currentNode = root;
      // set temp currentNode for traversal, if currentNode finishes
traversing and there's nothing left then exit
      while (currentNode != null) {
         // left
         if (value < currentNode.data ) {
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data ) {
            currentNode = currentNode.right;
         } else if (currentNode.data == value) {
            return true;
         }
      }
      return false;
   }

   public Node remove(int value){
      if (root == null) {
         System.out.println("Tree is empty, nothing to delete");
         return null;
      }

      Node currentNode = root; // For traversal to node you want to
delete
      Node parentNode = null; // Reference to parent node of node you
want to delete

      // traverse
      while (currentNode != null && currentNode.data != value) {
         // save parent ref
         parentNode = currentNode;
         // left
         if (value < currentNode.data) {
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data) {
            currentNode = currentNode.right;
         }
      }
      // Match, value == currentNode.data
      // Node to be deleted is the root node
      if (parentNode == null) {
         return removeThisNode(currentNode);
      }
      // Node to be deleted is left of parent
      if (parentNode.left == currentNode) {
         parentNode.left = removeThisNode(currentNode);
      }
      // Node to be deleted is right of parent
      else if (parentNode.right == currentNode) {
         parentNode.right = removeThisNode(currentNode);
      }
          return root;
   }

   private Node removeThisNode(Node curr) {

      // 1 child case
```

```
Key 4 exists:
true
Key 9 exists:
false

                20

        10

8

                6

        4

                2
-----------------------------------

                20

        10

8

        6

                2
```

irs, del, lookup

time: O(log N) for balanced trees,
basically the tree's height
O(N) worst case

space: O(1) because we're iterative
not recursive

```
        // No left child, return right child, so it can be overwritten
        if (curr.left == null) {
            return curr.right;
        }
        // No right child, return left child, so it can be overwritten
        if (curr.right == null) {
            return curr.left;
        }

        // 2 child case

        // Find min node in right child subtree
        // Append curr.left to minNode.left, so it saves deleted node's left
subtree
        // Return right subtree, so it can be overwritten

        Node minNode = findMin(curr.right);
        minNode.left = curr.left;
        return curr.right;
    }


    private Node findMin(Node curr) {
        while (curr.left != null) {
            curr = curr.left;
        }
        return curr;
    }

    // GFG helper function to print tree
    public static void print2DUtil(Node root, int space)
    {
        // Base case
        if (root == null)
            return;

        // Increase distance between levels
        space += COUNT;

        // Process right child first
        print2DUtil(root.right, space);

        // Print current node after space
        // count
        System.out.print("\n");
        for (int i = COUNT; i < space; i++)
            System.out.print(" ");
        System.out.print(root.data + "\n");

        // Process left child
        print2DUtil(root.left, space);
    }

    // Wrapper over print2DUtil()
    public void print2D()
    {
        // Pass initial space count as 0
        Node printNode = root;
        print2DUtil(printNode, 0);
    }

}
```

https://visualgo.net/en/bst?slide=1

Balancing Trees

AVL Tree | Red-Black Tree

AVL Tab

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

balances itself

. does switch and/or rotation to auto balance

https://visualgo.net/en/heap?slide=1

Binary Heap Tree

lookup: O(n)

insert: O(log n)

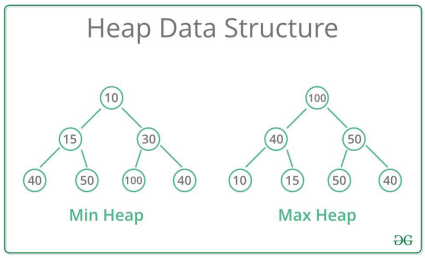
delete: O(log n)

max heap: root node is highest

min heap: root node is lowest

left & right can be value, as long as it's less than the top value

- good for comparative operations
  - I want all values under 31 → [



- good for priority queues, data storage, sorting algorithms

insertion is left to right (w/ node switch)
- always complete
- never unbalanced
- very efficient b/c balanced

Memory heap != heap data structure (free storage)

Priority Queues (why heaps are important)

- each element has a priority
- elements with higher priority are served first
  - analogous to airplanes (pilot boards first stewardess than passengers)
  - nodes switch places to correct priority order (even if inserted in the wrong order)

heaps

pros: priority

cons: slow lookup

pros: priority slow lookup

Flexible size

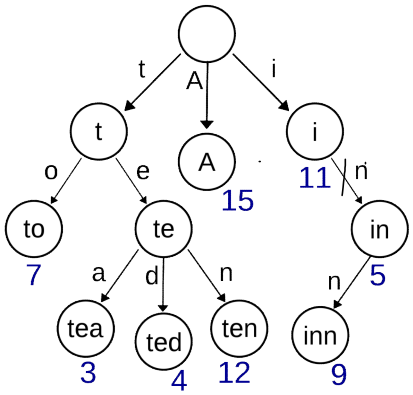
Fast insert

good for find max or find min , O(1)

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Trie Tree

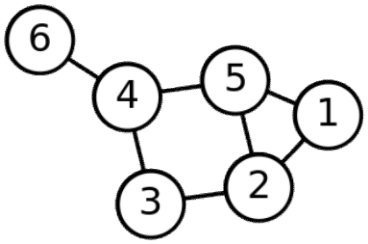
- aka prefix tree
- good for searching words for a dictionary / providing auto suggesting / IP routing
- specialized tree for searching
  - most often with text
- finds if a word / part of word exists in a body of text
- usually has an empty root node as the start



- Big O O(length of word)

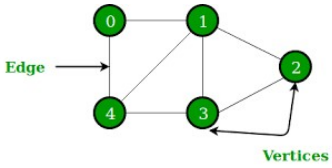
10. Graphs

Monday, May 23, 2022 10:21 PM

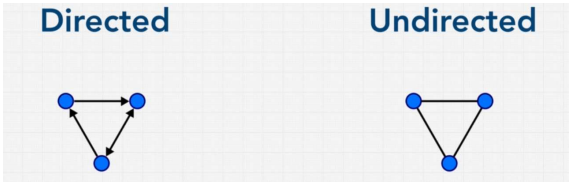


- nodes / vertices, interchangeable
- nodes are connected with edge

graphs are:
- set of values related in a pair-wise fashion



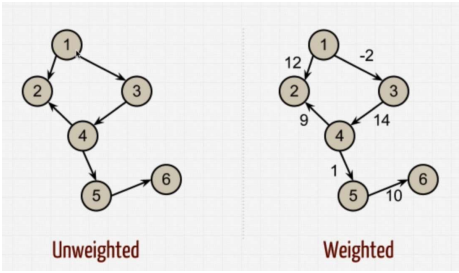
Types of graphs



Directed vs undirected

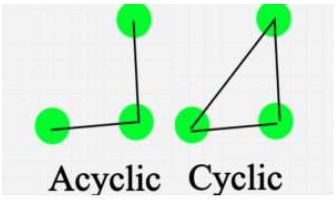
- movement is not bi-directional
- similar to one way street
- similar to twitter

- can go back & forth between nodes
- similar to highways
- similar to facebook, friends



unweighted vs weighted

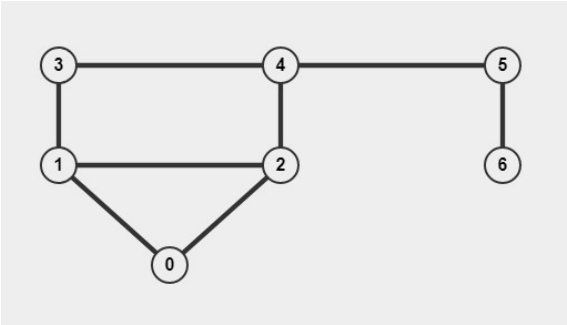
- google maps calculate optimal paths from A to b



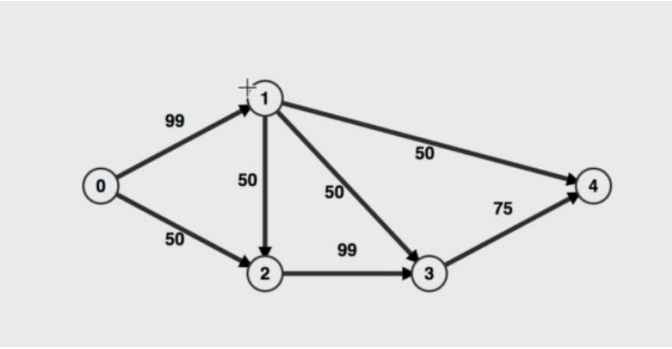
Acyclic vs Cyclic

- can circle back
  - like google maps
- only needs at least 1 cycle

e.g.



is undirected, unweighted, cyclic



is directed, weighted, acyclic

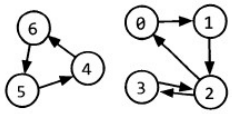
Graph Implementation (can be w/ array, hashmap, arraylist, LinkedList, etc)

3 ways to implement:

1) Edge List (u, v), shows connection from node u to v
   - if undirected, you need (u, v) and (v, u)

2) Adjacency List, hybrid of edge list & adjacency matrix (most common)
   - index of array is value of graph's node
   - value of index is the node's neighbor
   - if undirected, store twice

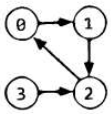
   represented w/ an array (or hashmap) of lists (arr, arrList, linked list, etc)



0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5

graph = [[1], [2], [0, 3], [2], [6], [4], [5]]

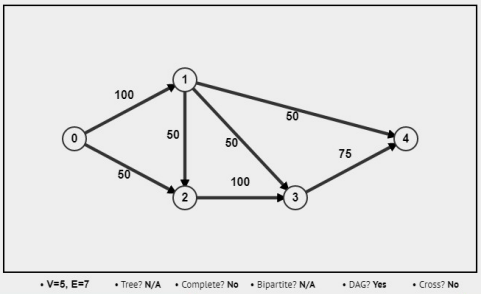
3) Adjacency Matrix, N x N boolean (where N is number of nodes)
   - 0s and 1s indicating if node x has a connection to node y
     (0 = false, 1 = true)
   - 0/1 can be replaced with weight



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

- 0 is connected to 1
- 1 to 2
- 2 to 0
- 3 to 2

graph = [
[0, 1, 0, 0],
[0, 0, 1, 0],
[1, 0, 0, 0],
[0, 0, 1, 0]
]



• V=5, E=7 • Tree? N/A • Complete? No • Bipartite? N/A • DAG? Yes • Cross? No

Adjacency Matrix

	0	1	2	3	4
0	0	100	50	0	0
1	0	0	50	50	50
2	0	0	0	100	0
3	0	0	0	0	75
4	0	0	0	0	0

Adjacency List

0:	(1, 100)	(2, 50)	
1:	(2, 50)	(3, 50)	(4, 50)
2:	(3, 100)		
3:	(4, 75)		
4:			

Edge List

0:	0	1	100
1:	0	2	50
2:	1	2	50
3:	1	3	50
4:	1	4	50
5:	2	3	100
6:	3	4	75

- can also hold weight in list
- lists can be objects to hold key, value

e.g. graph = [
0: [0, 1, 100],
1: [0, 2, 50]
...etc
]

Implementing Graph w/ Adjacency List

Graph

- addVertex(node)
- addEdge(node1, node2)

Main

C Test]

* Use ArrayList<LinkedList<Integer>>, because it is easier to add/remove new vertices & edges over array of arrays (b/c you have to shift elements)

* Time: addVertex O(1)
addEdge O(1)

Space: O(V+E)

```
import java.util.*;

public class Graph {
   // Adjacency List
   ArrayList<LinkedList<Integer>> adjacencyList = new
ArrayList<LinkedList<Integer>>();


   public void addVertex(int value) {
      LinkedList<Integer> newVertex = new LinkedList<Integer>();
      newVertex.add(value);

      // Move the LinkedList to ArrayList
      adjacencyList.add(newVertex);

   }

   public void addEdge(int source, int destination) {
     adjacencyList.get(source).add(destination);
 }

   public void printGraph() {
      for (LinkedList<Integer> inner : adjacencyList) {
         System.out.print(inner.getFirst() + " -> ");
         for (int i = 1; i < inner.size(); i++) {
            System.out.print(inner.get(i) + ", ");
         }
         System.out.println();
      }
   }
}
```

```
public class Main {

   public static void main(String[] args) {

      Graph graph = new Graph();

      graph.addVertex(0);
      graph.addVertex(1);
      graph.addVertex(2);
      graph.addVertex(3);
      graph.addVertex(4);
      graph.addEdge(0, 1);
      graph.addEdge(1, 0);
      graph.addEdge(1, 4);
      graph.addEdge(3, 1);
      graph.addEdge(4, 1);
      graph.addEdge(4, 2);
      graph.addEdge(4, 0);

        graph.printGraph();
   }
}
```

```
0 -> 1,
1 -> 0, 4,
2 ->
3 -> 1,
4 -> 1, 2, 0,
```

Summary

Pros

Good for relationships

(good for finding shortest paths)

Cons

Scaling is hard

11. Recursion

Thursday, May 26, 2022 4:24 AM

- Good for tree & graph manipulation, sort, search, D.P.
  - makes it a lot simpler

* iterative > recursive, space-wise because we allocate memory for recursive stack
  - stack overflow

Simply:
- defines in term of itself
  - function refers to itself inside the function

- Some examples:
  - DOM Traversal, file tree manipulation

* needs base case (exit condition) or call stack overflow will happen

Anatomy
1) Identify base case
2) Identify recursive case
3) 2 return statements, 1 for base & 1 recursive case
   - return recursive case, so result will go up the stack recurs i.e return should call itself while getting closer to base case

Practice factorial
- factorial(5) < factorial(4) < factorial(3) < factorial(2) < factorial(1)

- factorial ( ) factorial(4) factorial(3) - factorial(2) - factorial(1)

5 * 4 * 3 * 2 * 1

```
fac(5);

public static int fac(int d){
    if (d == 1){
      return 1;
    }
    return d * fac(d-1);
  }
```

5 * 24 = 120

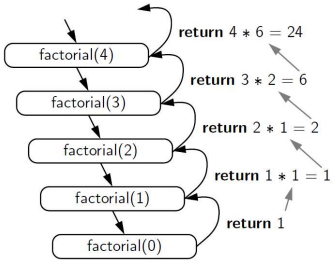
5 . fac(4) return 4 * 6 = 24

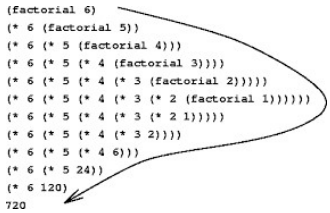
↳ 4 . fac(3) return 3 * 2 = 6

↳ 3 . fac(2) return 2 * 1

↳ 2 . fac(1) return 1

↳ 1





down then up

Practice Fibonacci

Input: int n, where n = index value of fib sequence

output: f(n)

where f(n) = f(n-1) + f(n-2), for n > 1

sum of previous 2 numbers

e.g. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

n = 2 output: 1, F(2) = F(1) + F(0) = 1 + 0 = 1

n = 3 output = 2, F(3) = F2 + F(1) = 1 + 1 = 2

n = 4 F(4) = F(3) + F(2) = 2 + 1 = 3

fib(4)

↳ fib(3) + fib(2)

↳ fib(2) + fib(1) + fib(1) + fib(0)

n = 4

output = F(4) = F(3) + F(2) = 2 + 1 = 3

class Solution {

Space : O(n), public int fib(int n) {

```
  if (n == 0 ){
    return 0;
  }
  if (n < 3){
    return 1;
  }

   return fib(n-1) + fib(n-2);
  }
}
```

```
class Solution {
  public int fib(int n) {
  if(n <= 1)
        return n;

   return fib(n-1) + fib(n-2);
  }
}
```

or n C 2

or

Time : $O(2^n)$, exponential time bc tree expands exponentially

Space : O(n), for recursive stack

iterative is O(n) time,

Divide & conquer w/ Recursion

1) Divide into a number of subproblems that are smaller instances of the same problems
2) Each instance of the subproblem is identical in nature
3) The solution of each subproblem can be combined to solve the problem at hand

- Merge sort, Quick sort, Tree Traversal, Graph Traversal

iterative (in main) → iterative (in function) → identify minimal problem (base case) → convert remaining logic to recursion

reverse string recursive

```
void reversed(char[] s, int first, int last){
   if (first > last) {
      return;
   }
   char temp = s[first];
   s[first] = s[last];
   s[last] = temp;
   reversed(s, ++first, --last);
}
```

12. Sorting

Wednesday, June 1, 2022 5:30 PM

https://www.toptal.com/developers/sorting-algorithms

- Matters for big data sets
- Focusing on Bubble, Selection, insertion, Merge, Quick, Heap, radix, counting (bad; non-compare)
- Arrays.sort uses Timsort for object arrays (stable) (good)

http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/46c727d6ecc2/src/share/classes/java/util/DualPivotQuicksort.java

Quicksort for primitive arrays
insertion for small arrays
Merge for mostly sorted arrays
dual pivot Quicksort for everything else

Tradeoffs (when to use which)

- Bubble sort
  - never use, $O(n^2)$ average
  - double for loop, comparing $O(i)$ and $O(i+1)$ n^2 times

```
void bubbleSort(int arr[])
  {
     int n = arr.length;
     for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
           if (arr[j] > arr[j + 1]) {
              // swap arr[j+1] and arr[j]
              int temp = arr[j];
              arr[j] = arr[j + 1];
              arr[j + 1] = temp;
           }
  }
```

https://www.geeksforgeeks.org/bubble-sort/

- Selection Sort
  - slightly better than Bubble
    - $O(n^2)$ average, but selection has less swaps
  - Scans for smallest item, then swap the index

```
void sort(int arr[])
  {
     int n = arr.length;

     // One by one move boundary of unsorted subarray
```

https://www.geeksforgeeks.org/selection-sort/

```
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Good sorts

Tradeoffs (when to use which)

https://www.geeksforgeeks.org/insertion-sort/

- Insertion sort
  - Good for small data sets and partially sorted data
  - O(n) best case, $O(n^2)$ avg/worse
  - split array into sorted and unsorted part, values from unsorted are picked and place in correct order in sorted part

GFG code

first pass:
- index 0 is unsorted section
  1 is sorted section

```
void sort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];   <- first unsorted element
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];   - swap
            j = j - 1;   - go back 1 index
        }
        arr[j + 1] = key;
    }
```

- repeat
  1) select first unsorted element
  2) swap other element to right to create correct pos & shift unsorted element
  3) advance pointer to right one.

$O(n^2)$

```
for ( ) {
    while ( ) {
    }
}
```

- shifts all element to the right to create pos for unsorted element
- inserts unsorted element at correct pos

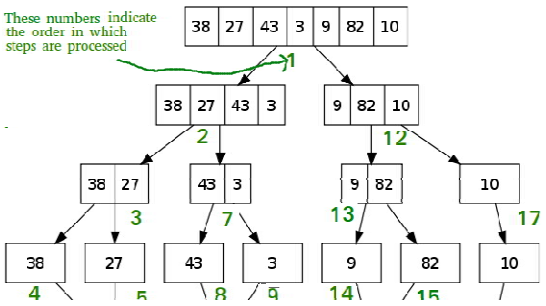
- Divide and conquer (Merge sort and Quick sort)
  - Merge: $O(n \log n)$ time, $O(n)$ space, stable sorting
  - Quick: $O(n \log n)$ time | $O(n^2)$ worse time, $O(\log n)$ space, not stable

https://www.geeksforgeeks.org/merge-sort/

Merge sort
- Divides array into 2 halves, calls itself for two halves (until it is 1 item), then merge sorted halves
- stable, if there are equivalent elements, the original order is preserved
- compares local list to each other instead of every element
- $O(n)$ space b/c we need to create divided arrays

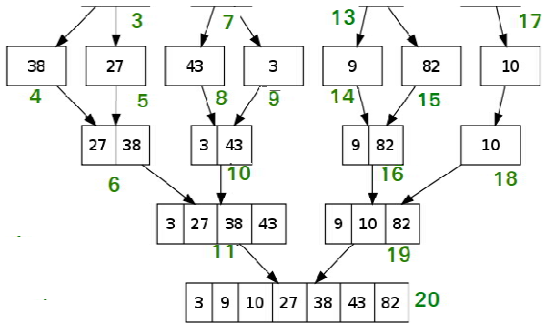
```
// Merges two subarrays of arr[].
  // First subarray is arr[l..m]
  // Second subarray is arr[m+1..r]
  void merge(int arr[], int l, int m, int r)
  {
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
      L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
      R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarray array
    int k = l;
    while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
      }
```

compare elements of 2 subarrays and merges



```
        arr[k] = L[i];
        i++;
      }
      else {
        arr[k] = R[j];
        j++;
      }
      k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

compare elements of 2 subarrays and merges them



sort() recursively halves array

merge() combines array

Main {

int arr[] = {...};

MergeSort ob = new MergeSort();

ob.sort(arr, 0, arr.length - 1);

}

recursively splits until left of arr = right of arr (only 1 element)

arr.length == 1

https://www.geeksforgeeks.org/quick-sort/

Quick sort

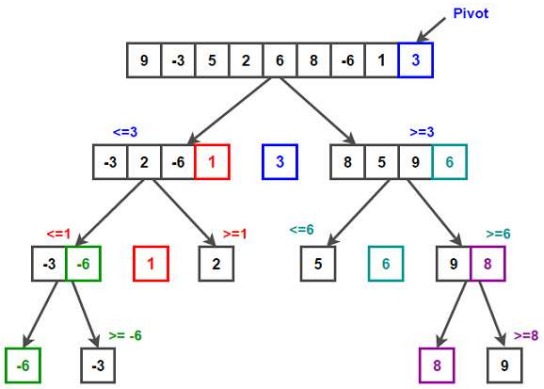
- picks an element as pivot (many ways) and partitions the array around the pivot

target of partition: given an element x as a pivot, place x in its correct position by putting all smaller elements before x and all larger elements after x

- O(log(n)) space

- O(n log n) time average, $O(n^2)$ worse if bad pivot

```
// A utility function to swap two elements
static void swap(int[] arr, int i, int j)
{
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}

/* This function takes last element as pivot, places
  the pivot element at its correct position in sorted
  array, and places all smaller (smaller than pivot)
  to left of pivot and all greater elements to right
  of pivot */
static int partition(int[] arr, int low, int high)
{

   // pivot
   int pivot = arr[high];

   // Index of smaller element and
   // indicates the right position
   // of pivot found so far
   int i = (low - 1);

   for(int j = low; j <= high - 1; j++)
   {

      // If current element is smaller
      // than the pivot
      if (arr[j] < pivot)
      {

         // Increment index of
         // smaller element
         i++;
         swap(arr, i, j);
      }
   }
   swap(arr, i + 1, high);
   return (i + 1);
}

/* The main function that implements QuickSort
        arr[] --> Array to be sorted,
        low --> Starting index,
        high --> Ending index
 */
static void quickSort(int[] arr, int low, int high)
{
   if (low < high)
   {

      // pi is partitioning index, arr[p]
      // is now at right place
      int pi = partition(arr, low, high);

      // Separately sort elements before
      // partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
```



```
    }
}
```

Merge vs Quick

- if worried about worse case: use merge
- if worried about memory: use Quick
  - unless the sorting is external, then use merge
- merge sort is stable

Radix sort and counting sort (non-comparison sort)

- only works with integers in a restricted range
- uses assumptions about how numbers are stored in memory

https://opendatastructures.org/newhtml/ods/latex/sorting.html#tex2htm-121

13. DFS, BFS, Dijkstra, Bellman-Ford

Thursday, June 2, 2022 9:04 PM

Quick Review of basic search

1) Linear search

arr = [3, 9, . . . , 15]

O(1) → O(n)

find x, x = 15

, iterate, if x, return T, else F

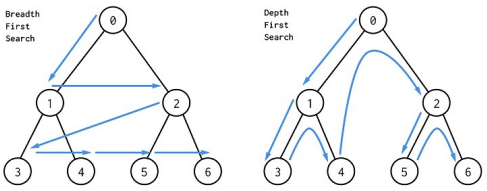
2) Binary search arr = [1, 3, . . . 15] log(n)

b/c it's a tree

- has to be sorted
- start at middle, find x
  - if x, true
  - if >x, go to middle of right
  - if <x, go to middle of left
  - repeat

Graph and Tree Traversal

- BFS and DFS
- O(n) for tree



Breadth First Search (BFS)

1) start at root

2) move left to right on first level, then left to right on second level, to n level until node match or tree ends

- ocd on close targets, finding shortest path, more memory needed
- uses queue to keep track of children you want to traverse to before jumping and a optional (for graphs) visited{} bool to keep track of visited nodes

Depth First Search (DFS)

1) start at root

2) follow one branch down to leaf, if not found then it continues at the nearest ancestor with unexplored children

- good on far targets, finding if path exists, less memory needed
- can get slow if tree is deep
- uses stack recursion

BFS Implementation

needs:

1) Declare Queue, visited[], currentNode(temp traversal)

2) push root

3) while (queue.size != 0)

{

currentNode = queue.poll() // removes or returns null if empty

add curr to visited

if (left child exists)

add to queue

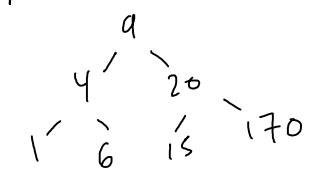
if (right child exists)

add to queue

```
public void breadthFirstSearch(){

    // Temp Node for traversal
    Node currentNode = root;

    // Optional, but mandatory for Graphs
    ArrayList<Integer> visited = new ArrayList<Integer>();


    Queue<Node> queue = new LinkedList<Node>();
    queue.add(currentNode);

    while (queue.size() != 0) {
      currentNode = queue.poll();
      visited.add(currentNode.data);

      if (currentNode.left != null) {
        queue.add(currentNode.left);
      }
      if (currentNode.right != null) {
```

```
            queue.add(currentNode.right);
          }

        }

        System.out.println("bfs: " + visited);
      }
```

DFS, 3 Types



Inorder (Left, root, right): [1, 4, 6, 9, 15, 20, 170]
- In a BST, it gives nodes in order

preorder (Root, Left, right): [9, 4, 1, 6, 20, 15, 170]
- Good for copying a tree

post order (Left, Right, root): [1, 6, 4, 15, 170, 20, 9]
- Good for deleting a tree

DFS Implemantation

Just move your operation around

tree height = deepest recursive function

memory = O(height)

```
public static void dfsInOrder(Node root) {
    if (root.left != null ){
      dfsInOrder(root.left);
    }
    System.out.print(root.data + " "); // inOrder
    if (root.right != null ){
      dfsInOrder(root.right);
    }
  }
```

```
public static void dfsPreOrder(Node root) {
    System.out.print(root.data + " "); // preOrder
    if (root.left != null ){
      dfsPreOrder(root.left);
    }
    if (root.right != null ){
      dfsPreOrder(root.right);
    }
  }
```

```
public static void dfsPostOrder(Node root) {
    if (root.left != null ){
      dfsPostOrder(root.left);
    }
    if (root.right != null ){
      dfsPostOrder(root.right);
    }
    System.out.print(root.data + " "); // postOrder
  }
```

Graph Traversal

https://visualgo.net/en/dfsbfs

- needs boolean visited[] to mark all vertices as not visited
  - default false
  - For BFS and DFS
  - O(V+E) for adjacency list

BFS : shortest path, closer nodes / cons: more memory

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

```
// prints BFS traversal from a given source s
  void BFS(int s)
  {
    // Mark all the vertices as not visited(By default
    // set as false)
    boolean visited[] = new boolean[V];

    // Create a queue for BFS
    LinkedList<Integer> queue = new LinkedList<Integer>();

    // Mark the current node as visited and enqueue it
    visited[s]=true;
    queue.add(s);

    while (queue.size() != 0)
    {
      // Dequeue a vertex from queue and print it
      s = queue.poll();
      System.out.print(s+" ");

      // Get all adjacent vertices of the dequeued vertex s
      // If a adjacent has not been visited, then mark it
      // visited and enqueue it
      Iterator<Integer> i = adj[s].listIterator();
      while (i.hasNext())
      {
        int n = i.next();
        if (!visited[n])
        {
          visited[n] = true;
          queue.add(n);
        }
      }
    }
  }
```

cons

Pros

- ... more backtracking after dead end | can get slow if

Pros

DFS: Solving a maze, backtracking after dead end
less memory
Does path Exist?

| can get slow if deep graph

```
void DFSUtil(int v, boolean visited[])
 {
    // Mark the current node as visited and print it
    visited[v] = true;
    System.out.print(v + " ");

    // Recur for all the vertices adjacent to this
    // vertex
    Iterator<Integer> i = adj[v].listIterator();
    while (i.hasNext()) {
       int n = i.next();
       if (!visited[n])
          DFSUtil(n, visited);
    }
 }

 // The function to do DFS traversal. It uses recursive
 // DFSUtil()
 void DFS()
 {
    // Mark all the vertices as not visited(set as
    // false by default in java)
    boolean visited[] = new boolean[V];

    // Call the recursive helper function to print DFS
    // traversal starting from all vertices one by one
    for (int i = 0; i < V; ++i)
       if (visited[i] == false)
          DFSUtil(i, visited);
 }
```

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

for adjacency matrix

this are factors in
unconnected graphs
BFS can also do the same

Weighted Graphs (Shortest Path for Weighted graph)

https://www.geeksforgeeks.org/what-are-the-differences-between-bellman-fords-and-dijkstras-algorithms/

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Dijkstra
- Greedy algorithm
- Can only handle positive weights
- $O(V + E \cdot \log(V))$, more efficient than Bellman

https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Bellman Ford
- Dynamic Programming
- handles negative and positive weight
- than Dijkstra

- handles negative and positive
- O(V·E), longer than Dijkstra

14. Dynamic Programming

Wednesday, June 8, 2022 8:33 PM

- Optimization technique using caching
  - Solve Problem by dividing into subproblems, solving each subproblem once and storing their solution in case the same subproblem occurs
  - Divide & conquer + Memoization

Memoization ~~=~~ caching

- storing values to use later on
- caching is just a way to speed up programs by holding data in an easily accessible box
- analogous to bringing a backpack to school, so you don't have to go home to retrieve items

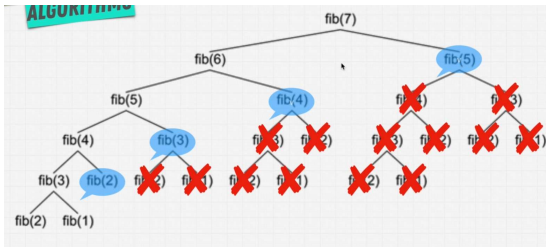
Memoization

```
if ( input in cache )
    return cache[n]
else
    calculation
    cache[n] = ...
    return cache[n]
```

simply remembering a solution to a subproblem and if subproblem return remembered solution

D.P.

1) Can be divided into subproblems
2) Recursive solution
3) Are there repetitive subproblems?
4) Memoize subproblems



original

fib w/ DP.

- original case is $O(2^n)$

```
public int fib(int n) {

if (n < 2){
   return n;
}

 return fib(n-1) + fib(n-2);
}
```

- DP is $O(n)$

Top-Down

```
class Solution {
   public int fib(int n) {
      return fibHelper(n, new int[n+1]);
   }

   private int fibHelper(int n, int[] memo) {
      if (n < 2 ) {
         return n;
      }

      // Check to see if value is in cache, since array indexes default value to 0
      if (memo[n] != 0) {
         return memo[n];
      }
      else {
         memo[n] = fibHelper(n-1, memo) + fibHelper(n-2, memo);
         return memo[n];
      }
   }
}
}
```

w/ outside memo declaration

not mine ->

```
class Solution {
   //dp memoization
   Map<Integer, Integer> cache = new HashMap<>();
   public int fib(int n) {
      if(n <= 0) return 0;
      if(n == 1) return 1;
      if(cache.containsKey(n)){ return cache.get(n);}
      else{
        int fibn =  fib(n - 1) + fib(n-2);
        cache.put(n, fibn);
        return fibn;
      }
   }
}
```

Bottom - up (iterative)

```
class Solution {
```

Basically:

Start from simpler solution and work your way up higher and higher towards the complex solutions

```
class Solution {
  public int fib(int n) {
    if (n < 2) {
      return n;
    }

    int[] memo = new int[n+1];
    memo[0] = 0;
    memo[1] = 1;
    for (int i = 2; i <= n; i ++){
      memo[i] = memo[i-1] + memo[i-2];
    }

    return memo[n];

  }
}
```