Table of Contents - Sections Pages

Saturday, December 15, 2018 6:58 PM

        - Recursive Traversals
        - The Search and Insert Problem
        - Question***
        - min
        - max
        - Successors and Predecessors
        - inOrderSuccessor
        - inOrderPredecessor
        - Delete Root, replace with inOrderPredecessor***
        - preOrderSuccessor***
        - Delete Any Node
        - Delete Smallest***
        - Delete Largest***
        - InOrder Traversal (Non recursive)
        - Level Order Traversal (Non recursive)
        - isBST***
        - Weight, numLeaves***
        - Moment, numNodes***
        - Height***
        - isDegenerate***
        - getLevel***
        - Rank***
        - reverseInOrder***
        - Level Count
        - Question***
        - Level Sum
        - Question***
        - External Nodes Question***
    - Heaps
        - isMaxHeap***
        - isZTree***
        - Sift Up Question***
        - Sift Down Question***
        - Priority Queue***
    - AVL Trees***
        - isAVLTree***
- Sorting
    - The Sound of Sorting…
    - Heap sort
    - Partitioning
        - Partition Question***
    - Kth Smallest
        - Kth Smallest Question***
    - Quick Sort Recursive Algorithm
    - Quick sort iterative algorithm
    - Merge Sort (OMITTED)
    - Shell Sort
- Dev
    - https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
    - Planning

The mod function

Monday, December 3, 2018 2:35 PM

See Programming 2 pages as a review:

1. Review of mod operator
2. Random number generation

5 % 2 = 2 R 1 10 % 15 = 0 R 10

✓ % gives the remainder from division

RULE x % n ∈ 0 → n − 1

Very useful Concept
If you MOD by 'n', the Output MUST be between 0 & n−1
1 & n ADD "1"

TABLE OF MODS.

Row % Col	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	0	0	0	0	0	0	0	0	0	✓ 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
→ 6	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9	0	1	0	1	4	3	2	1	0	✓ 9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
10	0	0	1	2	0	4	3	2	1	✓ 0	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
11	0	1	2	3	1	5	4	3	2	1	0	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
12	0	0	0	0	2	0	5	4	3	2	1	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
13	0	1	1	1	3	1	6	5	4	3	2	1	0	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
14	0	0	2	2	4	2	0	6	5	4	3	2	1	0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
15	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
16	0	0	1	0	1	4	2	0	7	6	5	4	3	2	1	0	16	16	16	16	16	16	16	16	16	16	16	16	16	16
17	0	1	2	1	2	5	3	1	8	7	6	5	4	3	2	1	0	17	17	17	17	17	17	17	17	17	17	17	17	17
18	0	0	0	2	3	0	4	2	0	8	7	6	5	4	3	2	1	0	18	18	18	18	18	18	18	18	18	18	18	18
19	0	1	1	3	4	1	5	3	1	9	8	7	6	5	4	3	2	1	0	19	19	19	19	19	19	19	19	19	19	19
20	0	0	2	0	0	2	6	4	2	0	9	8	7	6	5	4	3	2	1	0	20	20	20	20	20	20	20	20	20	20
21	0	1	0	1	1	3	0	5	3	1	10	9	8	7	6	5	4	3	2	1	0	21	21	21	21	21	21	21	21	21
22	0	0	1	2	2	4	1	6	4	2	0	10	9	8	7	6	5	4	3	2	1	0	22	22	22	22	22	22	22	22
23	0	1	2	3	3	5	2	7	5	3	1	11	10	9	8	7	6	5	4	3	2	1	0	23	23	23	23	23	23	23
24	0	0	0	0	4	0	3	0	6	4	2	0	11	10	9	8	7	6	5	4	3	2	1	0	24	24	24	24	24	24
25	0	1	1	1	0	1	4	1	7	5	3	1	12	11	10	9	8	7	6	5	4	3	2	1	0	25	25	25	25	25
26	0	0	2	2	1	2	5	2	8	6	4	2	0	12	11	10	9	8	7	6	5	4	3	2	1	0	26	26	26	26
27	0	1	0	3	2	3	6	3	0	7	5	3	1	13	12	11	10	9	8	7	6	5	4	3	2	1	0	27	27	27
28	0	0	1	0	3	4	0	4	1	8	6	4	2	0	13	12	11	10	9	8	7	6	5	4	3	2	1	0	28	28
29	0	1	2	1	4	5	1	5	2	9	7	5	3	1	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	29
30	0	0	0	2	0	0	2	6	3	0	8	6	4	2	0	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The search and insert problem (Linear Probing)

Sunday, December 2, 2018 6:38 PM

Kalicharan pg. 261 +8

HASHING TRIES TO SOLVE: THE SEARCH & INSERT PROBLEM

" Given a list of items (may be initially empty)
search for a given item in the list.
If the item does not exist:
Insert it "

ADVANTAGES OF HASHING: Extremely fast searching & inserting.

GOAL OF HASHING: To map values to locations by a hash function (H) ← output of hash function is the possible location.

PROBLEM OF HASHING: COLLISIONS

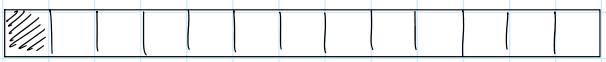
INTRODUCTION LINEAR PROBING

EXAMPLE: Array of size n = 12 (Ignore location zero!) Kalicharan Pg 262 +8

Hash function H(key) = key % n + 1 } LOCATION (INDEX)

Keys : 52 33 84 43 16 59 31 23 61.

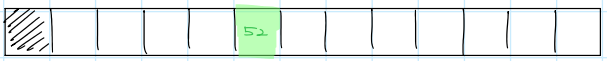
INIT



(1) H(52) = 52 % 12 + 1 = 5 :

Index 5 is free

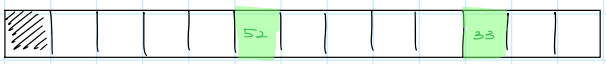
=> Put 52 in index 5.



(2) H(33) = 33 % 12 + 1 = 10

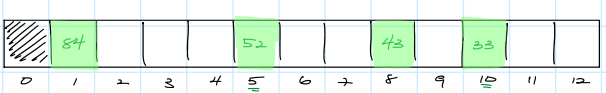
Index 10 is free

=> Put 52 in index 10



...

Eventually we get:



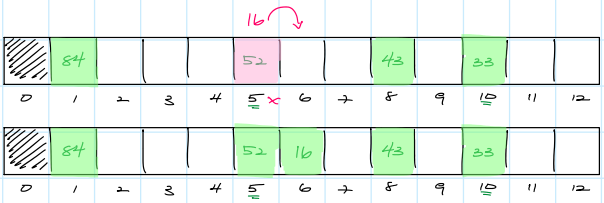
(3)

(3) But $H(16) = 16 \bmod 12 + 1 = 5$

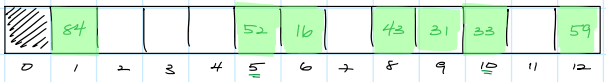
COLLISION!

RESOLUTION: Use Next FREE INDEX! ie 6.

(LINEAR PROBING)



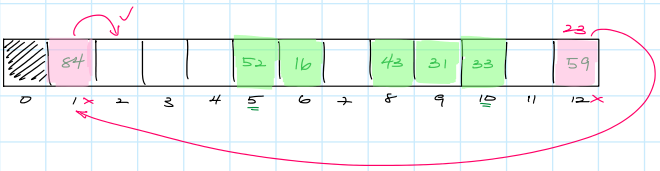
... SKIPPING AHEAD



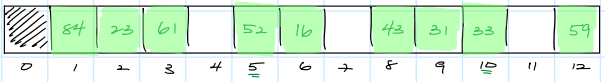
(4) $H(23) = 12$ ⇒ COLLISION

NOTE THE ARRAY IS CIRCULAR!

Next free spot: 2



... EVENTUALLY



* 2 Possible states for a location: Occupied
Empty (0)

ALGORITHM 1.

No Deletion
SEARCH & INSERT

DE MORGAN'S LAW

loc = H(key)
while (num[loc] is occupied & NOT key) (STOP IF EMPTY; STOP IF KEY)
    Go to the next location. loc = loc mod n + 1
if (num[loc] is empty)
    Put key in loc
else
    We have found the location of key.

mod n + 1

HUGE: Infinite loop if full!

HUGE: Infinite loop if full!
Keep extra locations free so the loop will exit

Hashing Strings

Sunday, December 2, 2018 7:43 PM

The ASCII table [Characters Mapped to Integers] $0 \rightarrow 127$ 2^7

C++. Show DEMO

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

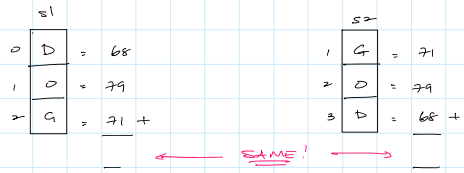
asciicharstable.com

PROBLEM : Hashing the integer value directly causes different words with same letters to be hashed to the same location

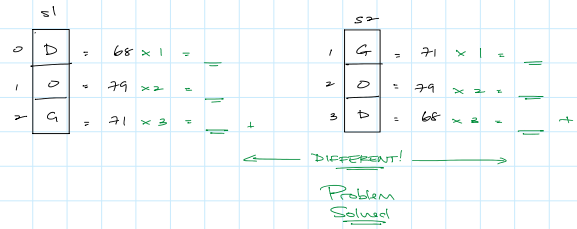
SOLUTION : Weights. (Arbitrary)

Example GOD = DOG

(1) Without weights:



(2) With Weights Start w at 1 & Add 1



ALGORITHM

```
value = 0
weight = 1
while (have not reached end of string):
        value += word[i] * weight
        weight += 1
```

Hashing Strings***

Monday, December 3, 2018 2:28 PM

comp2611_1_18
COMP2000_1_17
COMP2000_1_16

2. (a) In a hashing application, the key consists of a string of letters. Write a hash function which, given a key and an integer `max`, returns a hash location between 1 and `max`, inclusive. Your function must use *all* of the key and should not deliberately return the same value for keys consisting of the same letters. [3]

Deleting from hash table

Sunday, December 2, 2018 8:02 PM

PROBLEM: if you set the location to the EMPTY (0) state then: after deletion, the hash table cannot locate some values.

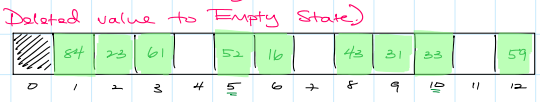
SOLUTION: Introduce a DELETED (-1) State.

Keep track of last deleted cell (if encountered) & insert there.

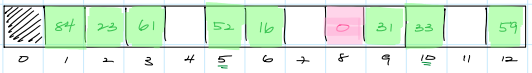
Do not stop your search at a deleted cell (ONLY STOP AT EMPTY OR FOUND)

New (3) possible states : Occupied
: Empty (0)
: deleted (-1)

BAD EXAMPLE (Setting A Deleted value to 0 Empty State)

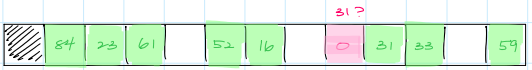


Attempt to delete 43
by setting it to empty (0)



Now Attempt to locate 31.
h(31) = 8

Location 8 is empty so we
conclude that 31 DOES NOT EXIST ← ERROR



SOLUTION: Set value to (-1) (Deleted state) & Adopt a new Search & Insert Algorithm.

This time we set the deleted
(8) location to -1.
Now 31 hashes to (8)

HUGE: Write the Algorithm
so that we do not
stop searching @ "-1"

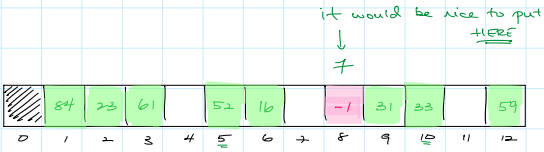
→ We Continue down the
chain as Normal
& we find 31 in location (9)

BUT What if we wanted to add 7 to the table?

7 Hashes to location (8)

We SHOULD put 7 in (8)

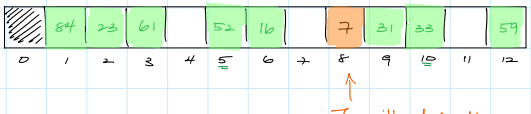
BUT the new Algorithm will not stop @ Deleted State '-1'



Algorithm will keep checking until 7 is found or until we Reach "Empty State" in location (11)

BUT - We should not put 7 in (11) because it would WASTE the deleted space in (8)

So we write the algorithm to take NOTE of any deleted State spaces along the way and we put key THERE



ALGORITHM 2 Search & Insert with DELETE State

```
loc = H(key)
deletedloc = 0          # Because index 0 is UNUSED ↳ Implied No deleted location found!

while (location is occupied & Not key):   # Not Empty   HUGE: Do Not Stop searching @ '-1' Deleted State
    if (deletedLoc = 0 & num[loc] is deleted):
        deletedLoc = loc                  # Take Note if a deleted location was encountered.
    loc = loc mon + 1                     - go to next location.

if (num[loc] is empty):
    if (deletedLoc != 0):                 - if a deletedLoc was encountered.
        loc = deletedLoc                  - Put key in the deletedLoc
    num[loc] = key

else:
    key found.                            - Do nothing.
```

Search and Insert (with delete)***

Monday, December 3, 2018 3:46 PM

The key is passed through a secondary hash function and the output is used as an increment

h2 } Discussed later

COMP2000_1_17

Integers are inserted in an integer hash table `list[1]` to `list[n]` using "open addressing with double hashing". Assume that the function `h1` produces the initial hash location and the function `h2` produces the increment. An available location has the value `Empty` and a deleted location has the value `Deleted`.

Write a function to search for a given value `key`. If found, the function returns the location containing `key`. If not found, the function inserts `key` in an `Empty` location. However, if a deleted location was encountered in the search for `key`, `key` must be inserted in the *first* deleted location encountered. The function returns the location in which `key` was inserted. You may assume that `list` has room for a new integer. [5]

Same Algorithm Discussed Previously !

Primary and Secondary Clustering

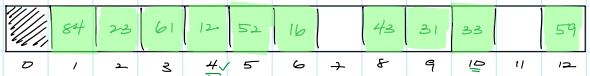
Sunday, December 2, 2018 8:24 PM

Already Explained.

PROBLEM : CLUSTERING

EXAMPLE

12 will hash to 4



We form a LONG CHAIN (Cluster) that has to be searched sequentially

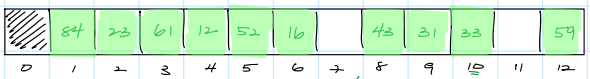
if something hashes to 7 the chain grows and the problem gets worse!

2 TYPES OF CLUSTERING

(1) Primary - Keys hashing to different locations trace the same sequence when searching/inserting.

Ex if something hashes to 1
SEQUENCE : 1, 2, 3, 4, 5, 6, 7

if something hashes to 3
SEQUENCE: 3, 4, 5, 6, 7



LINEAR PROBING HAS BOTH

(2) SECONDARY Keys hashing to SAME locations trace the same sequence when searching/inserting.

THEOREM: if the table size is M and $loc = loc_{prev} + (k) \bmod M$ & k are relatively prime
=> All locations will be generated.

If k VARIES with key => LINEAR PROBING W/ DOUBLE HASHING => BEST Hashing Method.

Quadratic Probing

Sunday, December 2, 2018 9:01 PM

DEFINITION

1. $h_1(key)$ – Initial Hash.
2. $h_2(key) = ai + bi^2$ where a,b are constants
   i is the current collision count.
   (increments until empty or found.

ADVANTAGES: Eliminates Primary Clustering.

Nb – if the table size (n) is prime ⇒ the method can reach ½ the locations in the table.
– if $n = 2^m$, only a small fraction of the table is reached.

ALGORITHM 3

Search & Insert
No Delete State
Quadratic Probing

```
loc = H(key)
i = 0
while (num[loc] is occupied and Not key):
        i++
        loc += a*i + b*i*i   - Quadratic Probing.
        while (loc > n):     - DO NOT USE loc = loc%n
                loc -= n       if loc = n then we get ZERO

if (num[loc] is Empty):
        num[loc] = key

else
        key Found.
```

Linear Quadratic Question***

Sunday, December 9, 2018 2:17 PM

18-19 S1 COMP2611 CW3

Integers are inserted into a hash table `H[1..11]` using the primary hash function `h(k) = 1 + k mod 11`.

Show the state of the array after inserting the keys 10, 22, 31, 4, 15, 28, 17, 88 and 58 using:

i. Linear probing [2 marks]

ii. Quadratic probing with probe function `I + i²` where `i` represents the number of times the current key has collided. [4 marks]

Total marks 6

$ai + bi^2$

i = # of collisions

Increment $i + i^2$

0	1	2	3	4
0	2	6	12	

i

1	22	~~88~~
2	88	
3		
4	58	
5	4	~~15~~
6	15	
7	28	~~17~~
8	17	
9		
10	31	
11	10	

$h(10) = 1 + 10 \bmod 11 = 1 + 10 = 11$

$h(22) = 1 + 22 \bmod 11 = 1 + 0 = 1$

$h(31) = 1 + 31 \bmod 11 = 1 + 9 = 10$

$h(4) = 1 + 4 \bmod 11 = \quad = 5$

$h(15) = 1 + 15 \bmod 11 = 1 + 4 = 5$

$h(28) = 1 + 28 \bmod 11 = 1 + 6 = 7$

$h(17) = 1 + 17 \bmod 11 = 1 + 6 = 7$

$h(88) = 1 + 88 \bmod 11 = 1 + 0 = 1$

$h(58) = 1 + 58 \bmod 11 = 1 + 3 = 4$

ii

1	22	~~88~~
2		
3	88	
4	17	~~58~~
5	4	~~15~~
6	58	
7	15	~~28~~ ~~17~~
8		
9	28	~~17~~
10	31	
11	10	

$h(10) = 11$

$h(22) = 1$

$h(31) = 10$

$h(4) = 5$

$h(15) = 5$

$h(28) = 7$

$h(17) = 7$

$h(88) = 1$

$h(58) = 4$

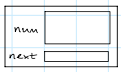
Chaining

Sunday, December 2, 2018 9:18 PM

All items which hash to the same location are held on a (1) linked list or (2) An array with array indices subscripts as links

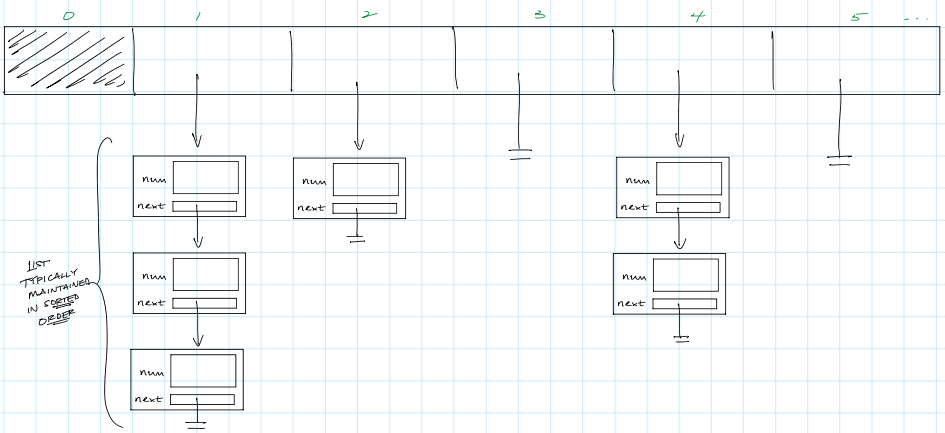
METHOD #1 LINKED LIST.

LINKED LIST NODE STRUCT




```
struct Node {
    int num;
    Node *next;
};
```

VISUALIZING HASH TABLE: Node *hash[10];



METHOD #2 ARRAY

Kali pg 275-8

IDEA: Use single array and subscripts as links (Chaining IDEA)

The First Part of the table hash[1...n] is the actual Hash table

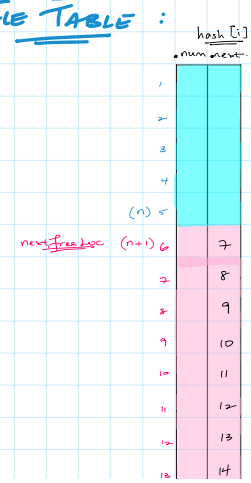
The Rest of the table hash[n+1...N] is used for Chaining (Overflow table)

Max Array Size

Visualize:


```
struct Node {
    int num
    int next
};
```

VISUALIZE THE TABLE:



SEARCH & INSERT

- key hashes to location k
  - if hash[k] is empty.
    - hash[k].num = key // Put key in k
    - hash[k].next = -1 // Nothing else in chain so set next to -1 (Null)
  - if hash[k] is Not Empty. Put key in freeLoc
    - //Then search the chain list for the next free location and link key to the chain (IN ORDER)
    - ie hash[freeLoc].next = hash[k].next.
      hash[k].next = freeLoc.

OR Link @ the End of the list

hash[lastLocation].next = freeLocation.
hash[freeLoc].next = -1. //this makes the newly added key the last location in the Chain

DELETE

IDEA Maintain a free list (link all free items) of all available locations in the chain table.
Have a variable called "nextFreeLoc" be the location of the next free spot in the chain.
(init to the first loc in chain table [n+1].)

We will put key here → k = nextFreeLoc ← key goes in the nextFree spot.
nextFreeLoc = hash[nextFreeLoc].next ← keep track of the next free spot.

2 DELETE CASES:

CASE 1 the item to be deleted is in the hashTable [1..n]
if (hash[k].next == -1) (No items in Chain [n+1..N])
set hash[k].num to Empty
else (items in the Chain)
//Move item from chain to hash.
j = hash[k].next;
hash[k] = hash[j];
add j to the free list.
↳ hash[j].next = nextFreeLoc
nextFreeLoc = j

CASE 2 item in the chain table. [n+1..N]
delete from chain and relink
keep track of prev → hash[prev].next = hash[curr].next
add curr to free list

Search & Insert Final Algorithm.

Chaining Linked List ***

Monday, December 3, 2018 2:31 PM

(b) In a certain application, keys which hash to the same location are held on a linked list. The hash table location contains a pointer to the first item on the list and a new key is placed at the end of the list. Each item in the linked list consists of an integer key, an integer count and a pointer to the next element in the list. Storage for a linked list item is allocated as needed. Assume that the hash table is of size n and the call H(key) returns a location from 1 to n, inclusive.

initial Hash

(i) Write programming code, including all relevant declarations, to initialize the hash table. [2]

FIX APPLY H(key)

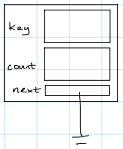
(ii) Write a function which, given the key k, searches for it. If not found, add k in its appropriate position and set count to 0. If found, add 1 to count; if count reaches 10, delete the node from its current position, place it at the head of its list and set count to 0. [7]

END OF LIST

STILL NEED TO IMPLEMENT

comp2611_1_18

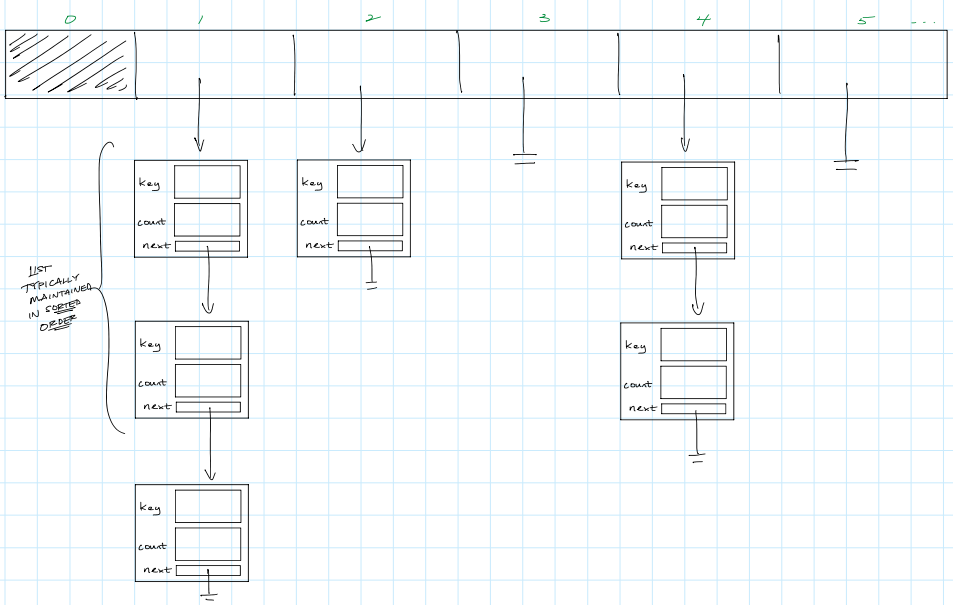
```
struct Node{
    int key, count;
    Node * next;
};
```



(i)

```
Node *hash[n];
// iterate & set to NULL
```

VISUALIZING HASH TABLE:



(ii)

```
82  ////////////////////////////////////////////////////////////////////////////////////////////////////
83  void searchAndInsert(Node *hashTable[], int key){
84
85      int loc = H(key);                           // initial hash
86
87      if(hashTable[loc] == NULL){                 // if the location is totally free
88          hashTable[loc] = makeNode(key);             // put key in that location
89          return;
90      }
91
92      Node *curr = hashTable[loc];
93      Node *parent = curr;
94      while(curr != NULL && curr->key != key){    // else location is not free so search linked list for key
95          parent = curr;
96          curr=curr->next;
97      }
98
99      if(curr == NULL){                           // if key not found then insert key at end of linked list
100         parent->next = makeNode(key);
101     }
102
103     else{                                       // else key was found
104         curr->count++;                              // so increment the count
105         if(curr->count == 10){                      // if the count reaches 10
106             curr->count = 0;                            // reset count to zero and move key to head of list
107             if(parent == curr){                         // if there is only one item in the list then
108                 return;                                     // do nothing because key is already at the head
109             }
110
111             parent->next = curr->next;              // else key is not at the head so delete key from wherever it is
```

```
109                 }
110
111                 parent->next = curr->next;                  // else key is not at the head so delete key from wherever it is
112                 curr->next = hashTable[loc];                // and move it to the head
113                 hashTable[loc] = curr;
114             }
115         }
116     }
117     ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Chaining Array***

Sunday, December 9, 2018 3:00 PM

comp2000_3_14
COMP2000_1_16

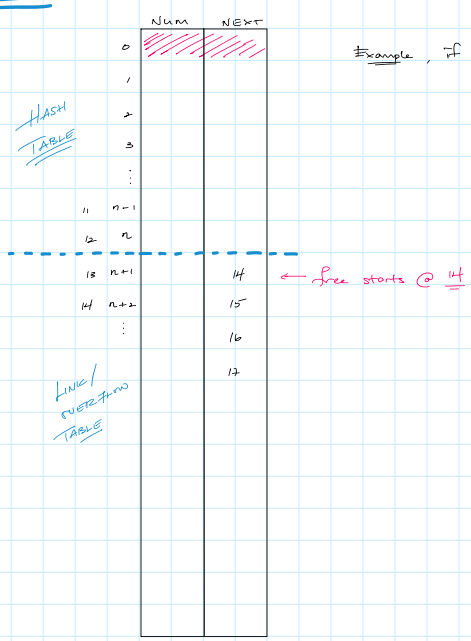
(b) A hash table of size **n** contains two fields - an integer data field and an integer link field - (called *data* and *next*, say). The *next* field is used to link data items in ascending order. A value of -1 indicates the end of the list. The variable **top** (initially set to -1) indicates the location of the smallest data item.

Integers are inserted in the hash table using "open addressing with double hashing". Assume that the function **h1** produces the initial hash location and **h2** produces the increment. An available location has the value **EMPTY** and no item is ever deleted from the table.

Write programming code to search for a given value **key**. If found, do nothing. If not found, insert **key** in the table and *link it in its ordered position*. You may assume that the table contains room for a new integer. [10]

```
k = H(key)    //H is the hash function
if (hash[k].num == Empty) { — Insert into hash table
  hash[k].num = key
  hash[k].next = -1
}
else { — Hash table loc occupied.
  curr = k
  prev = -1                                  < for ascending
  while (curr != -1 && hash[curr].num != key) {
    prev = curr
    curr = hash[curr].next
  }
  if (curr != -1) key is in the list at location curr  // Do Nothing
  else { //key is not present — Insert into next free loc in hash table
    hash[free].num = key     //assume free list is not empty
    hash[free].next = -1 — hash[curr].next
    hash[prev].next = free
    free = hash[free].next
  } //end else
} //end else
```

VISUALISE



Example, if 5 & 10 both hash to 2

← free starts @ 14

Linear Probing With Double Hashing

Sunday, December 2, 2018 9:40 PM

loc = loc + (k) → the Increment varies with key. (Key is ~~ALSO~~ HASHED).

DEFN – 2 hash functions

(1) h1 (key) – Initial Hash. (loc)

(2) h2 (key) – Gives the INCREMENT (k). if a collision occurs

ADV: Eliminates BOTH Primary & Secondary Clustering

THM

Recall: choose n as a prime #. => Many Locations will be probed

Ex. loc = key % n + 1

k = key % (n-2) + 1.

ALGORITHM H

Search & Insert

Linear Probing w/ Double Hashing

No Delete

```
loc = h1(key)
k = h2(key)
while (arr[loc] is occupied and Not key):
    loc = loc + k
    if (loc > n):
        loc = loc - n

if (arr[loc] is Empty)
```

```
if (num[loc] is Empty)
        num[loc] = key

else :
        key found.
```

Open Addressing (Linear Probing) with double hashing

***

Monday, December 3, 2018 3:48 PM

COMP2000_1_17

Integers are inserted into a hash table `H[1..13]` using "open addressing with double hashing". The primary hash function is $h_1(k) = 1 + k \% 13$. The secondary hash function is $h_2(k) = 1 + k \% 9$. Show the state of the array after inserting, in order, the keys 31, 22, 17, 25, 28, 30, 34, 15, 39, 42. If a calculated location is greater than 13, subtract 13 to get a valid location. For example, 18 converts to 5. [4]

1	
2	
3	28
4	
5	17
6	31
7	
8	
9	30
10	22
11	
12	
13	25

KEY	H1	H2
31	6	
22	10	
17	5	
25	13	
28	3	
30	5	4
34		
15		
39		
42		

All methods Question***

Sunday, December 9, 2018 3:32 PM

COMP2000_2_17

Question 3

You are given the hash function $f(k) = k \% N$, where $N = 11$. Show the state of the hash table of size N after inserting the nonnegative integers 77, 24, 35, 59, 84, 68, 42, and 28 into the hash table using the following hashing methods.

a. The linear probing method with the rehash function is $f_t(k) = (f(k) + t) \% N$, where the collision number $t = 1, 2, ...$ [3 marks]

b. The quadratic probing method with the rehash function is $f_t(k) = (f(k) + t^2) \% N$, where the collision number $t = 1, 2, ...$ [4 marks]

c. The double hashing method with the rehash function is $f_t(k) = (f_{t-1}(k) + h(k)) \% N$, where the collision number $t = 1, 2, ...$, the second hash function is $h(k) = q - (k \% q)$, the constant $q = 5$, $f_0(k) = f(k)$. [4 marks]

d. The coalesced chaining method [4 marks]

[Total mark: 15]

COMP2000_2_16

Question 3 **[15 marks]**

You are given the hash function $h(k) = k \% M$, where $M = 11$. Show the state of the hash table of size M after inserting the integers 66, 35, 24, 48, 73, 57, 31, and 94 into the hash table using the following hashing methods.

a. The linear probing method with the rehash function is $h_t(k) = (h(k) + t) \% M$, where the collision number $t = 1, 2, ...$ [3 marks]

b. The quadratic probing method with the rehash function is $h_t(k) = (h(k) + t^2) \% M$, where the collision number $t = 1, 2, ...$ [4 marks]

c. The double hashing method with the rehash function is $h_t(k) = (h_{t-1}(k) + g(k)) \% M$, where the collision number $t = 1, 2, ...$, the second hash function is $g(k) = c - (k \% c)$, the constant $c = 7$, $h_0(k) = h(k)$. [4 marks]

d. The coalesced chaining method [4 marks]

Triangular Matrices

Sunday, December 2, 2018 6:38 PM

Must be Square

Lower Triangular ($i \geq j$)

Strictly => Diagonal Excluded

Not Strictly => Diagonal Included

i \ j	1	2	3	4	5
1	A_{11}^{1}				
2	A_{21}^{2}	A_{22}^{3}			
3	A_{31}^{4}	A_{32}^{5}	A_{33}^{6}		
4	A_{41}^{7}	A_{42}^{8}	A_{43}^{9}	A_{44}^{10}	
5	A_{51}^{11}	A_{52}^{12}	A_{53}^{13}	A_{54}^{14}	A_{55}^{15}

# of Elements = $\sum_{r=1}^{n} r = \frac{1}{2}n(n+1)$ (with Diagonal)

without Diagonal, $= \frac{1}{2}n(n+1) - n$

Mapped using Row-Major Order

1D Array B:

A_{11}	A_{21}	A_{22}	A_{31}	A_{32}	A_{33}	A_{41}	A_{42}	A_{43}	A_{44}	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A_{ij} : $i-1$ rows before row i with sum $1+2+\dots+(i-1) = \frac{1}{2}i(i-1)$

In row i, we are in column j.

$$\Rightarrow \quad A_{ij} = B_{\frac{1}{2}i(i-1)+j}$$

Upper Triangular. ($i \leq j$)

Strictly => Diagonal Excluded

Not Strictly => Diagonal Included

i \ j	1	2	3	4	5
1	A_{11}^{1}	A_{12}^{2}	A_{13}^{4}	A_{14}^{7}	A_{15}^{11}
2		A_{22}^{3}	A_{23}^{5}	A_{24}^{8}	A_{25}^{12}
3			A_{33}^{6}	A_{34}^{9}	A_{35}^{13}
4				A_{44}^{10}	A_{45}^{14}
5					A_{55}^{15}

$\frac{1}{2}j(j-1)+i$ — Flip $i \leftrightarrow j$

Mapped using Column Major Order

A_{11}	A_{12}	A_{22}	A_{13}	A_{23}	A_{33}	A_{14}	A_{24}	A_{34}	A_{44}	A_{15}	A_{25}	A_{35}	A_{45}	A_{55}
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A_{ij} : We have $j-1$ columns before j with sum $1+2+3+\dots+j-1 = \frac{1}{2}j(j-1)$

In column j, we are in row i.

$\Rightarrow$ $\boxed{A_{ij} = B_{\frac{1}{2}j(j-1)+i}}$

Symmetric and Skewed Symmetric

Monday, December 3, 2018 2:21 PM

SYMMETRIC

Row Major Order

lower triangle.

```
if (i >= j):
    B[ i*(i-1)/2 + j ]
```

same as lower triangular

strictly upper — else:

```
    B[ j*(j-1)/2 + i ]
```

swap i & j

$A_{ij} = A_{ji}$

A_{11} A_{12} A_{22} A_{13} A_{23} A_{33} A_{14} A_{24} A_{34} A_{44} A_{15} A_{25} A_{35} A_{45} A_{55}

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

SKEWED SYMMETRIC

Diagonal is zero

Same value

Opposite sign

```
if (i==j):
    return 0.
```

$n = i-2$

strictly lower — if (i > j)

```
    B[ (i-1)*(i-2)/2 + j ]
```

strictly upper — else:

```
    B[ (j-1)*(j-2)/2 + i ]
```

A_{21} A_{31} A_{32} A_{41} A_{42} A_{43} A_{51} A_{52} A_{53} A_{54}

1 2 3 4 5 6 7 8 9 10

Football***

Monday, December 3, 2018 1:00 PM

comp2611_1_18
COMP2000_1_17
18-19 S1 COMP2611 CW3
COMP2000_1_16

(c) An n × n matrix A is used to store the points obtained in football matches among n teams. A team gets 3 points for a win, 1 point for a tie and 0 points for a loss. A[i,j] is set to 3 if team i beats team j; it is set to 1 if the match is tied and it is set to 0 if team i loses to team j.

DISGUISED SKEWED SYMMETRIC

In order to conserve storage, the values in the (strictly) lower triangle of A are stored in an array B[1..m] in row order.

(i) What is the value of m in terms of n? $m = \frac{1}{2}n(n+1) - n$ [1]

(ii) Write a function score(i,j) which, by accessing B, returns the value of A[i,j]. If i or j is invalid, the function returns -1. [4]

(iii) Using the function in (ii), write another function which, given t, returns the total number of points earned by team t. [3]

sum the row
lower triangular
for (j = 1; j <= n; j++)
if (Not invalid):
sum += score(B,t,j)

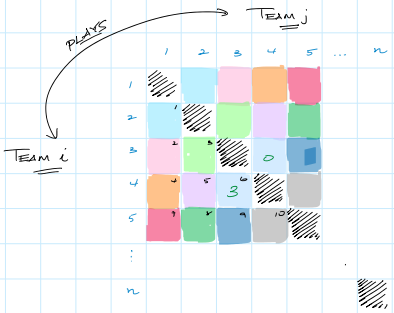
Total marks Q2: 20

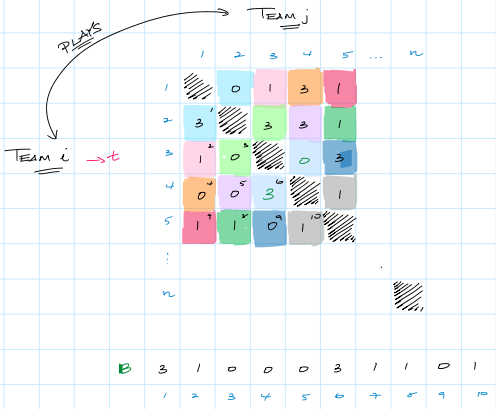
i<1 or i>n or j<1 or j>n or i=j
1 2 3 4 5

if INVALID:
return -1
else:
i>j
if strictly lower triangular.
return value = B [1+(i-1)(i-2)/2+j]
strictly upper triangular
else:
i<j
value = B [1+(j-1)(j-2)/2+i]
if value is 1 => A tie => return 1
return opposite of value (if 3 => return 0)
(if 0 => return 3)

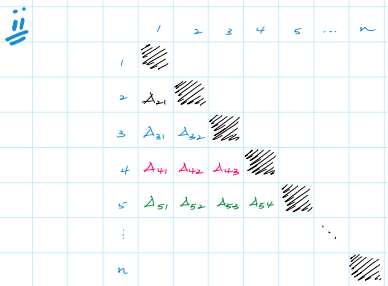
VISUALIZE !!

3 = WIN
1 = TIE
0 = LOSE





B 3 1 0 0 0 3 1 1 0 1
1 2 3 4 5 6 7 8 9 10



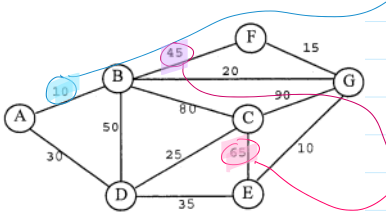
MAJOR ROW ORDER →

Adjacency Matrix***

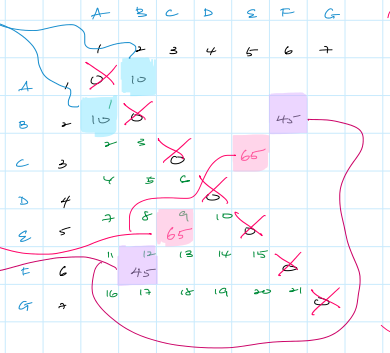
Sunday, December 9, 2018 3:07 PM

comp2000_3_14

3. (a) Given the following undirected graph:



Follow the LINES!



Recognise this can be Considered a Symmetric Matrix without Diagonal

(b) Let **A** be the adjacency matrix representation of the graph in 3 (a), assuming that each letter node is represented by its position in the alphabet. **A[j, j] = 0** for all **j**. If there is no edge from vertex **i** to vertex **j**, let **A[i, j] = 999**.

(i) Explain how **A** can be stored in a one-dimensional array **B[1..m]** conserving storage as much as possible. Assuming that **A** is an **n x n** matrix, what is the value of **m** in terms of **n**? [3]

Store the strictly lower triangle in Row Major Order.

(ii) Write a function which, given a node **j**, accesses **B** and returns the sum of the weights of edges leaving **j**. It may be helpful to write another function which returns the weight of an edge from node **i** to node **j**. [4]

```
weight :
    if i=j ?
        return 0.
    else if i>j ?
        return B[½(i-1)(i-2)+j]
    else
        Return B[½(j-1)(j-2)+i]
```

```
sum(B, j, n)
    s = 0
    for(int i=1; i<=n; i++)
        s += weight(B, j, i)
```

Terminology

Sunday, December 9, 2018 7:27 PM

DEFINITION

A Graph 'G' is
a pair (V, E)
where V is a finite set of vertices — $|V|$ = # of Vertices
AND E is a binary (mapping) relation on V (edges) — $|E|$ = # of Edges

Directed vs Undirected

$(u, v) \in E \Rightarrow v$ is Adjacent to u.
$\Rightarrow (u, v)$ is incident (leaves) from u
$\Rightarrow u$ & v are neighbours

In Degree – # of edges entering a vertex.
Out Degree = ___
Degree = In + Out = # of Neighbours of a vertex

Path – $\exists$ a path of length n
from u to v
if $\exists$ a sequences of vertices $u = v_0, \ldots, v_n = v$
s.t $(v_{i-1}, v_i) \in E$ $\forall$ $i = 1, \ldots, n$

Reachable if $\exists$ a path.

Simple Path - All vertices are distinct

Cycle –

Acyclic

Connected

Strongly Connected

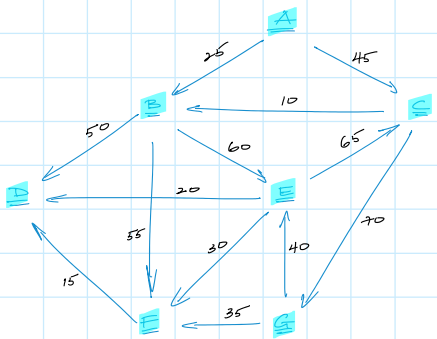
Free Tree

Weighted Graph

Adjacency Matrix Representation

Sunday, December 9, 2018 7:29 PM

Kali text pg. 205 +5



ZEROS ON DIAGONAL

TO

FROM	1	2	3	4	5	6	7
	A	B	C	D	E	F	G
1 A	0	25	45				
2 B		0		50	60	55	
3 C		10	0				70
4 D				0			
5 E			65	20	0	30	
6 F				15		0	
7 G					40	35	0

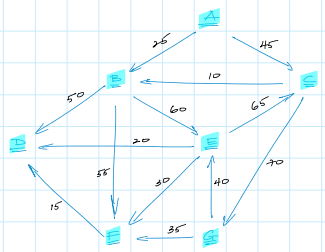
EMPTY SPACES => (INF) No Edge

CODE : int A[7][7]

RECALL : Matrix → 1D Array Skew Symmetric Question.

Adjacency List Representation

Sunday, December 9, 2018 7:31 PM



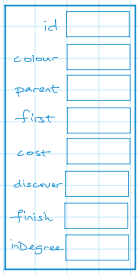
Typically uses An Array (but can also use BST & HashTable)

1 A → B, 25 → C, 45 ⏚
2 B → D, 50 → E, 60 → F, 55 ⏚
3 C → B, 10 → G, 70 ⏚
4 D ⏚
5 E → C, 65 → D, 20 → F, 20 ⏚
6 F → D, 15 ⏚
7 G → E, 40 → F, 35 ⏚

= NODES = EDGES

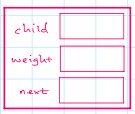
NOTE: Edge List Maintained in Alphabetical Order

Node Structure




```
struct Node{
    char id;      //the name of the node; a single character e.g. A, F, T
    int colour;
    int parent;   //an array subscript indicating the location of a parent node
    GEdgePtr first; //pointer to the first edge from the node; null, if none
};
```

Edge Structure

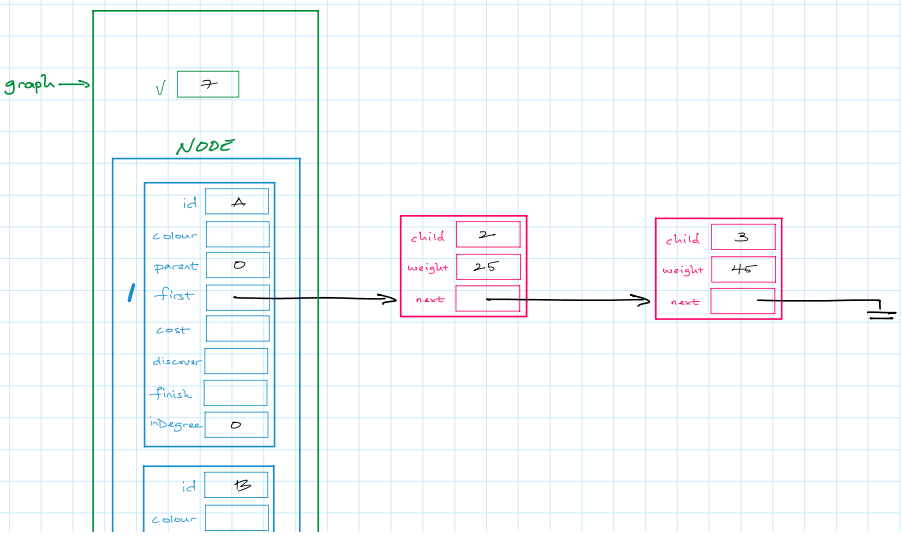


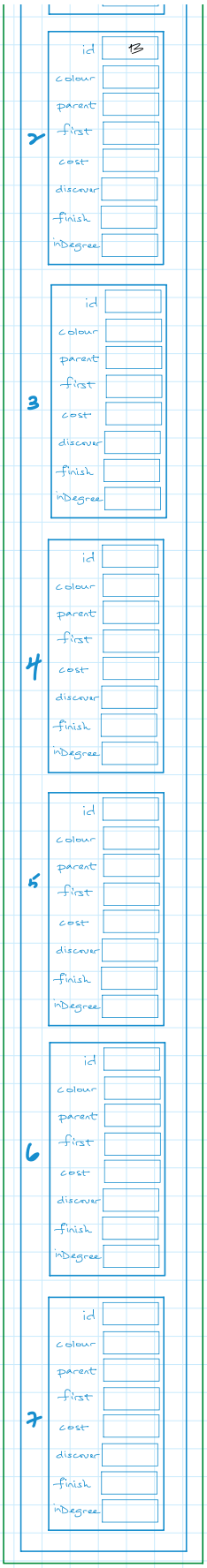
Each edge node has the following fields:

```
struct Edge{
    int child;     //a subscript in G; G[child] is the child node
    int weight;
    GEdgePtr next; //pointer to the next edge; null, if none
};
```

Graph Structure

Visualizing a C++ Implementation.




2
id
13
colour
parent
first
cost
discover
finish
inDegree
3
id
colour
parent
first
cost
discover
finish
inDegree
4
id
colour
parent
first
cost
discover
finish
inDegree
5
id
colour
parent
first
cost
discover
finish
inDegree
6
id
colour
parent
first
cost
discover
finish
inDegree
7
id
colour
parent
first
cost
discover
finish
inDegree

Question from revoked bonus

Tuesday, December 11, 2018 12:45 PM

int adjacentTo (Graph * graph, string v, string adj[]):

Copies all the vertices adjacent to vertex *v* in the graph into the *adj* array passed as a parameter. It returns the amount of vertices adjacent to *v* (incidentally, this value is the same as the out-degree of *v*).

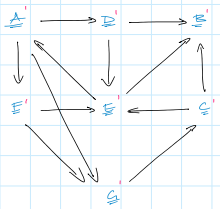
```
30  int adjacentTo (Graph * graph, string v, string adj[]){
31
32    // find v
33    int loc = 0;
34    for(int i = 1; i <= graph->numVertices; i++){
35      if(graph->vertices[i].id == v){
36        loc = i;
37        break;
38      }
39    }
40
41    // v is not found
42    if(loc == 0){
43      return -1;
44    }
45
46    // v is found in loc
47    int j = 0;
48    Edge *curr = graph->vertices[loc].firstEdge;  // first node in linked list
49    while(curr != NULL){
50      j++;
51      adj[j] = graph->vertices[curr->dest].id;
52      curr = curr->next;
53    }
54    return j;
55  }
```

DFT (easier - recursive algorithm)

Tuesday, December 11, 2018 2:15 AM

IDEA: Go forward (deeper) into the Graph In (Numerical) Alphabetical Order while possible
If not possible then backtrack and try again

Ex



A D B E F G C

ALGORITHM

3 states of a vertex u:
(i) unvisited (w)
(ii) visiting (G)
(iii) visited (B)

No Parent as -1 (Not in Array)

For 1 Vertex: DFT(G, v) — starting vertex

visit v and mark as visiting
for each edge from v to w:
  if w is unvisited
    set parent of w to v
    DFT(G, w)
mark v as visited.

For Entire Graph: DFTGraph(G, s) — Starting from vertex s.

mark all (Nodes) vertices as unvisited AND parents as (0) None
DFT(G, s)
for each vertex v:
  if v is unvisited:
    DFT(v)

```
152  ////////////////////////////////////////////////////////////////////////////////
153  // DEPTH FIRST TRAVERSAL
154  ////////////////////////////////////////////////////////////////////////////////
155  void DFTNode(Graph *graph, int nodeIndex){
156      printNode(graph->node[nodeIndex]);                  // visit the node
157      graph->node[nodeIndex].colour = VISITING;           // mark node as VISITING
158
159      Edge *curr = graph->node[nodeIndex].first;
160      while(curr != NULL){                                // for all nodes leaving the current node
161          if(graph->node[curr->child].colour == UNVISITED){     // if the node is unvisited
162              DFTNode(graph, curr->child);                      // then visit it
163          }
164          curr = curr->next;
165      }
166
167      graph->node[nodeIndex].colour = VISITED;            // mark current node as visited
168  }
169
170
171  void DFTGraph(Graph *graph, int startIndex){
172      for(int i = 1; i <= graph->V; i++){                 // init
173          graph->node[i].colour = UNVISITED;
174          graph->node[i].parent = NO_PARENT;
175      }
176
177      DFTNode(graph, startIndex);                         // visit the node
178
179      for(int i = 1; i <= graph->V; i++){                 // visiting any left over unvisited nodes
180          if(graph->node[i].colour == UNVISITED){
181              DFTNode(graph, i);
182          }
183      }
184  }
```

Classifying Edges based on DFT

Sunday, December 9, 2018 7:31 PM

Tree
Back
Forward
Cross

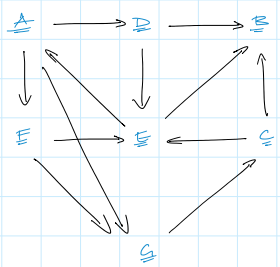
Discovery and Finish Times

BFT (width, level order)

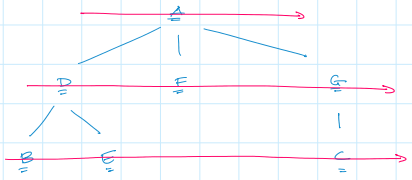
Tuesday, December 11, 2018 2:15 AM

IDEA : Traverse the graph in layers (levels)
Address all Edges coming from a Node, before moving on

EX



TRICK:
REDRAW Graph as a TREE in Alphabetical level ORDER.
Then perform a level order traversal



A D F G B E C

ALGORITHM → uses a Queue

set all vertices to unvisited and parents to None.
enqueue the first vertex
set first vertex to visiting.
while queue is Not Empty:
    p = dequeue
    visit p
    for each edge (p, x):
        if x is unvisited:
            set x to visiting
            set parent to p
            enqueue x
    set p to visited.

Topological Sort

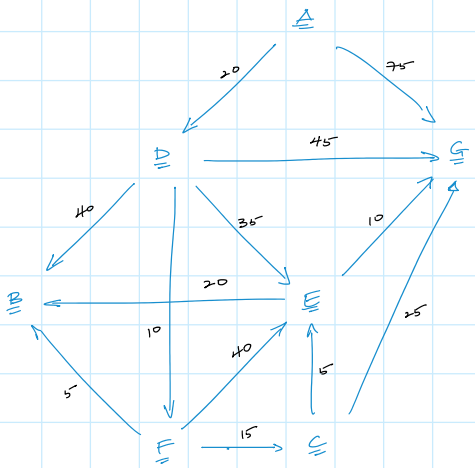
Sunday, December 9, 2018 7:31 PM

Dijkstra's MCP (non negative weights)

Sunday, December 9, 2018 7:31 PM

IDEA Finding the Minimal Cost Path from a Source vertex to ANY other reachable vertex.

EX



STAGE (0) — Source

VERTEX	A	B	C	D	E	F	G
PARENT	/	/	/	/	/	/	/
COST	0	∞	∞	∞	∞	∞	∞
MIN PRIORITY QUEUE	~~A 0~~	B_∞	C_∞	D_∞	E_∞	F_∞	G_∞

DEQUEUE A & CONSIDER EDGES FROM A.

STAGE (1) — Source — **A**

VERTEX	A	B	C	D	E	F	G
PARENT	/	/	/	A	/	/	A
COST	0	∞	∞	~~∞~~ 20	∞	∞	~~∞~~ 75
MIN PRIORITY QUEUE	D_{20}	G_{75}	B_∞	C_∞	E_∞	F_∞	

DEQUEUE D & CONSIDER EDGES

STAGE (2) — Source — **D**

VERTEX	A	B	C	D	E	F	G
PARENT	/	D	/	A	D	D	~~D~~
COST	0	~~∞~~ 60	∞	20	~~∞~~ 55	~~∞~~ 30	~~75~~ 65
MIN PRIORITY QUEUE	~~F_{30}~~	E_{55}	B_{60}	G_{65}	C_∞		

DEQUEUE F & CONSIDER EDGES

STAGE (3) — Source — **F**

VERTEX	A	B	C	D	E	F	G
PARENT	/	~~D~~ F	/	A	D	D	D
COST	0	~~60~~ 35	~~∞~~ 45	20	55	30	65
MIN PRIORITY QUEUE	~~B_{35}~~	~~C_{45}~~	E_{55}	G_{65}			

DEQUEUE(B) BUT B HAS NO EDGES LEAVING IT

SO DEQUEUE (C) TOO & EXAMINE EDGES

C

STAGE (4)	Source						
VERTEX	A	B	C	D	E	F	G
PARENT	/	F	F	A	~~D~~ C	D	D
COST	0	35	45	20	~~55~~ 50	30	65

MIN PRIORITY QUEUE: ~~E 55~~ G 65

DEQUEUE & EXAMINE!

E

STAGE (4)	Source						
VERTEX	A	B	C	D	E	F	G
PARENT	/	F	F	A	C	D	~~D~~ E
COST	0	35	45	20	50	30	~~65~~ 60

MIN PRIORITY QUEUE: G 60

DEQUEUE G HAS NO EDGES LEAVING IT SO WE ARE DONE!

FINAL

STAGE (4)	Source						
VERTEX	A	B	C	D	E	F	G
PARENT	/	F	F	A	C	D	E
COST	0	35	45	20	50	30	60

MIN PRIORITY QUEUE

ALGORITHM (Omitted) Pg 236

Bellman Ford's MCP (Negative Weights)

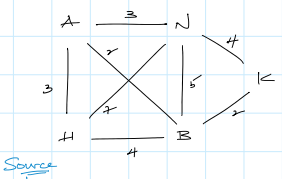
Monday, December 10, 2018 10:29 PM

Prim's MST

Sunday, December 9, 2018 7:32 PM

IDEA Always choose the smallest edge that connects a Visited & Unvisited Node

Ex

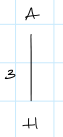


Source

VERTEX	A	B	H	K	N
PARENT	H	H	/	/	H
COST	3	4	0	∞	7
☑	A_3	B_4	N_7		

⟹

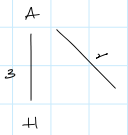
Start with H (Arbitrary), Choose the smallest edge from H.



VERTEX	A	B	H	K	N
PARENT	H	A	/	/	A
COST	3	2	0	∞	3
☑	B_2	N_3			

⇓

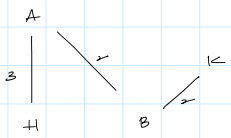
Choose the smallest edge from A & H



VERTEX	A	B	H	K	N
PARENT	H	A	/	B	A
COST	3	2	0	2	3
☑	K_2	N_3			

⇓

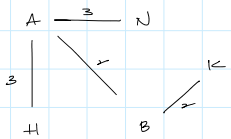
Choose smallest from A, H, B.



VERTEX	A	B	H	K	N
PARENT	H	A	/	B	A
COST	3	2	0	2	3
☑	N_3				

⇓

Only N Remains.

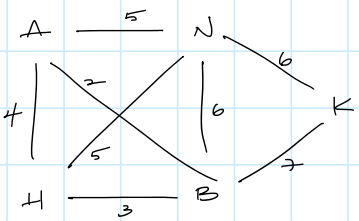


Kruskal's MST

Sunday, December 9, 2018 7:32 PM

IDEA: Fill in Edges in Ascending Order By Weight

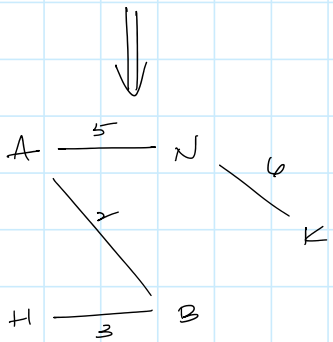
Ex



Sort Edges:

2	A	B
3	H	B
4	A	H
5	H	N
5	A	N
6	N	K
6	B	N
7	B	K

Order Does Not Matter

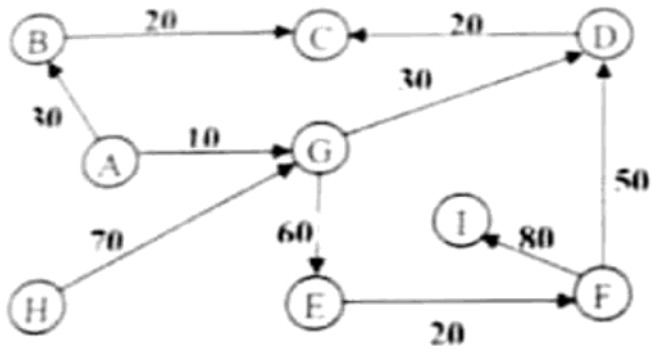


Question: ADJ, DFT, BFT, Kruskal***

Sunday, December 2, 2018 6:38 PM

18-19 S1 COMP2611 CW3

Given the following graph.



Draw the adjacency list representation of the graph. List the nodes in alphabetical order. [2 marks]

Give the depth-first and breadth-first traversals of the graph starting at **G**. Edges leaving a node are processed in alphabetical order. [4 marks]

Assuming that all the edges in the above graph are undirected.

i. Use Kruskal's algorithm to derive a minimum spanning tree (MST). Show all appropriate structures at each stage of the derivation. [5 marks]

ii. Draw a second distinct MST that the algorithm can also produce. [2 marks]

iii. What is the total cost for any MST derived? [1 mark]

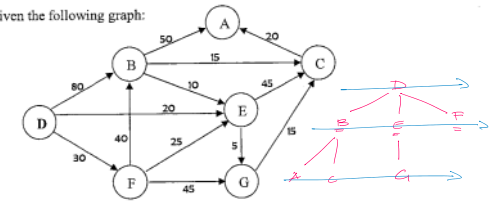
Total marks 14

Question: ADJ, DFT, BFT, Dijkstra's***

Sunday, December 9, 2018 2:38 PM

comp2611_1_18

3. (a) Given the following graph:

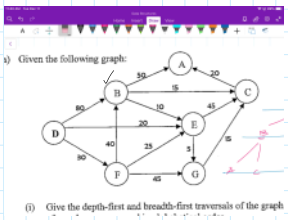


DFT D B A C E G F

BFT D B E F A C G

(i) Give the depth-first and breadth-first traversals of the graph starting at **D**. Edges of a node are processed in alphabetical order. [2]

(ii) Derive the minimal-cost paths from node D to every other node using Dijkstra's algorithm. For each node, give the cost and the path to get to the node. At each stage of the derivation, show the minimal cost fields, the parent fields and the priority queue. In your table heading, list the nodes in *alphabetical order*. [8]



VERTEX	A	B	C	D	E	F	G
PARENT	C	F	G	/	B	D	E
cost	60	70	40	0	20	30	25
Queue							

(b) A directed graph `G(V, E)` is represented using *adjacency lists*. The graph has *n* nodes stored in an array `G[1..n]`. Each node, `G[i]`, has the following fields:

```
char id;      //the name of the node; a single character e.g. A, F, T
int colour;
int parent;   //an array subscript indicating the location of a parent node
GEdgePtr first; //pointer to the first edge from the node; null, if none
```

Each edge node has the following fields:

```
int child;     //a subscript in G; G[child] is the child node
int weight;
GEdgePtr next; //pointer to the next edge; null, if none
```

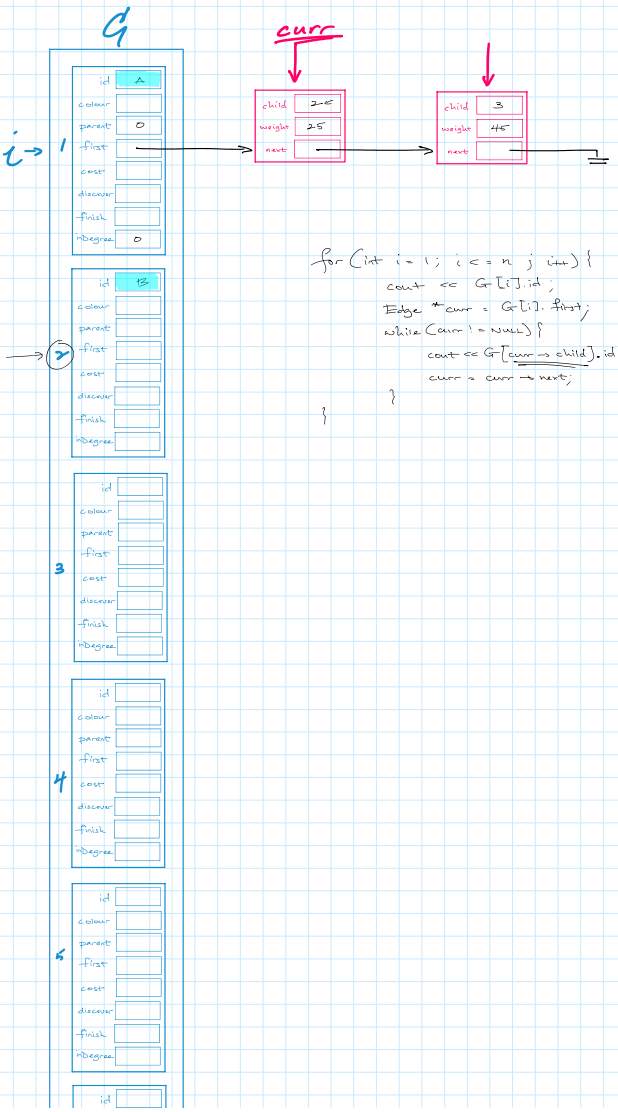
(i) Write a function which, given G and n, outputs the graph, listing the nodes in the order in which they appear in G. Each node is followed by the name and weight of the edges leaving it. [3]

```
for (int i = 1; i <= n; i++) {
    cout << G[i].id;
    Edge *curr = G[i].first;
    while (curr != NULL) {
        cout << G[curr->child].id << "\t" << curr->weight;
        curr = curr->next;
    }
}
```

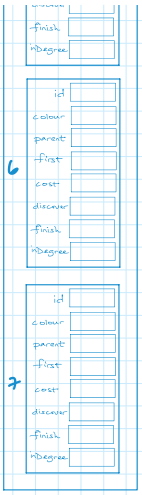
COMP2000_1_17

Visualizing the C++ Implementation

Note the green graph pointer to the vertex array has been removed!




```
for (int i = 1; i <= n; i++) {
    cout << G[i].id;
    Edge *curr = G[i].first;
    while (curr != NULL) {
        cout << G[curr->child].id << "\t" << curr->weight;
        curr = curr->next;
    }
}
```



(ii) Write a function which, given G, n and a node name S, performs a depth-first traversal of G starting at S. Output the name of a node as it is visited and set the parent fields so that a depth-first path can be determined. Assume that all nodes are reachable from S. [7]

Click here for solution
DFT (easier - recursive algorithm)

(iii) Assuming that a depth-first traversal has already been done as in (ii), write a function which, given G, n and a node *name* D, prints the names of the nodes, in the order visited, on the depth-first path from the source node to D. [4]

COMP2000_1_17 ONLY

Need to comment this

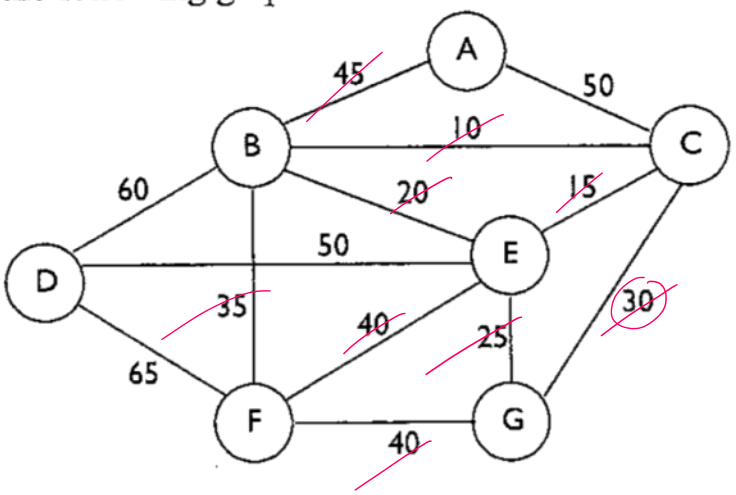
```
187  void followPath(Graph *graph, int index){
188      if(index != 0){
189          followPath(graph, graph->node[index].parent);
190          printNode(graph->node[index]);
191      }
192  }
193
194
195  void printPath(Graph *graph, string destination){
196      int destinationIndex = getIndex(graph, destination);
197      cout << "Path to " << destination << endl;
198      followPath(graph, destinationIndex);
199  }
```

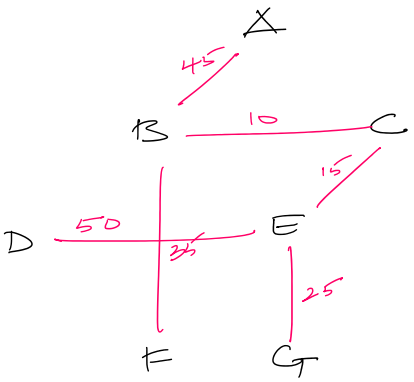
Question: Kruskal***

Sunday, December 9, 2018 2:49 PM

COMP2000_1_17

(b) Given the following graph:





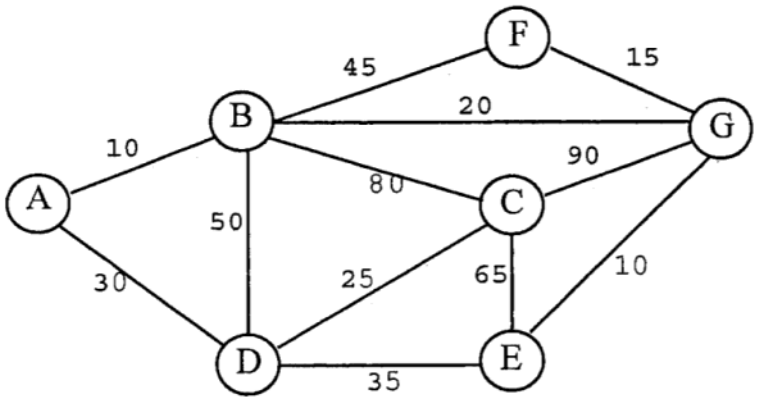
Draw the minimum spanning tree obtained using Kruskal's algorithm. Show the steps in your derivation. [3]

Question: DFT, BFT, Dijkstra's, Kruskal's***

Sunday, December 9, 2018 3:04 PM

comp2000_3_14

3. (a) Given the following undirected graph:



(i) Give the depth-first and breadth-first traversals of the graph starting at C. Edges of a vertex are processed in alphabetical order. [2]

(ii) Derive the minimal-cost paths from vertex C to every other vertex using Dijkstra's algorithm. For each vertex , give the cost and the path to get to the vertex . At each stage of the derivation, show the minimal cost fields, the parent fields and the priority queue. [8]

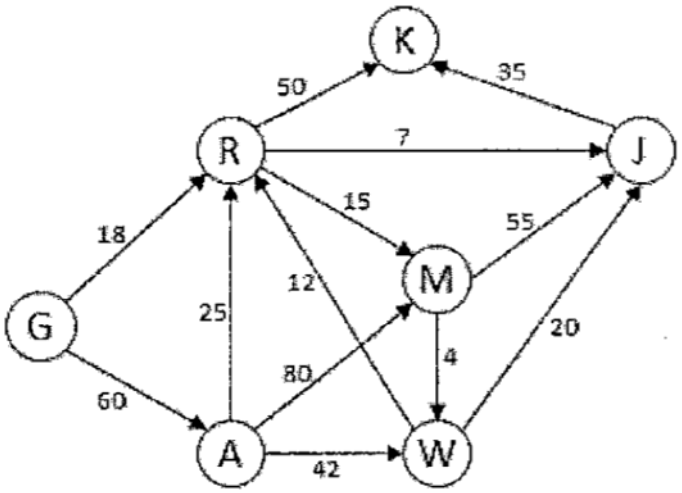
(iii) Draw the minimum spanning tree obtained using Kruskal's algorithm. At each step, identify the edge selected for adding to the tree. [3]

Question: ADJ, DFT, D/F Times, Topological Sort, Kruskal's

Sunday, December 9, 2018 3:44 PM

COMP2000_1_16

3. (b) Given the following graph:



(i) Draw the adjacency list representation of the graph. List nodes and edges in alphabetical order. [2]

(ii) Give the depth-first and breadth-first traversals of the graph starting at A. Edges leaving a node are processed in alphabetical order. [2]

(iii) Make a copy of the graph *without* the edge weights. Assume that a depth-first traversal is performed starting at A and that *edges of a node are processed in alphabetical order*. Indicate the discovery and finish times for each node and label each edge with T (tree edge), B (back edge), F (forward edge) or C (cross edge), according to its type. [4]

(iv) State, with a reason, whether or not it is possible to topologically sort the nodes of the graph. [1]

(v) Assuming that the graph in (a) is undirected, draw the minimal spanning tree obtained by using Kruskal's algorithm. Show the steps in your derivation. [3]

Topological Sort***

Sunday, December 9, 2018 3:08 PM

comp2000_3_14

(c) Write an algorithm to perform a *topological sort* on a directed acyclic graph using a modified depth-first traversal. [5]

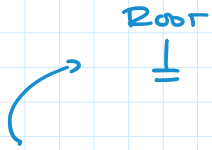
Binary Trees

Saturday, December 8, 2018 9:57 PM

Kali Book Pg. 114 +8

DEFINITION

A binary tree can (I) be empty

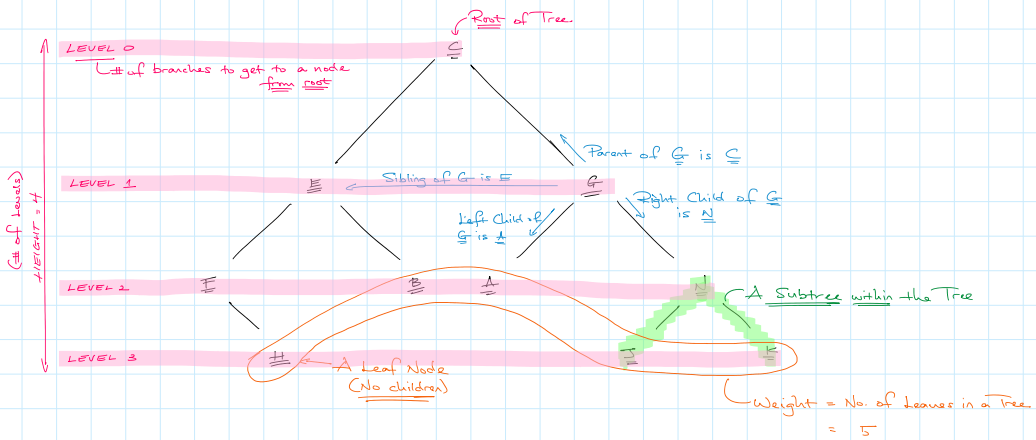


OR (II) Consist of a root and 2 subtrees (L) & (R), each itself being a binary tree

The recursive nature of the definition allows us to use recursive functions. (ALOT of Recursive functions)

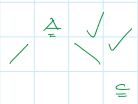
IMPORTANT TERMS

Moment = No. of Nodes in tree = 10

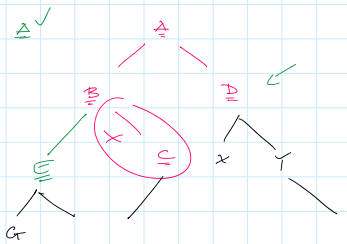


TYPES

✓ FULL - Every node has either 0 or 2 children. Ex



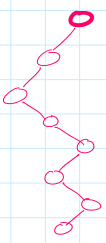
✓ ALMOST COMPLETE - Every level except possibly the last is completely filled. (HEAPS)
AND all Nodes at the last level are to the far left.



✓ COMPLETE/PERFECT - All interior nodes are filled (2 Children)
AND All leaves have the same depth (level)

✓ BALANCED - L & R Subtrees differ in height by no more than 1

✓ DEGENERATE - Each parent has exactly 1 child ⇒ linked list



Properties of binary trees [edit]

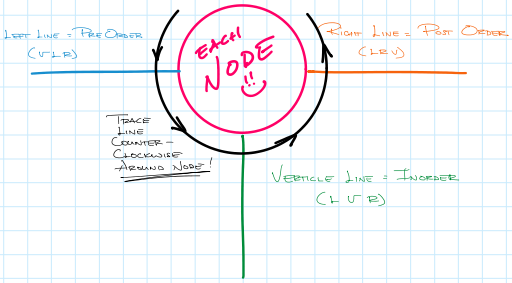
- The number of nodes n in a full binary tree, is at least $n = 2h + 1$ and at most $n = 2^{h+1} - 1$, where h is the height of the tree. A tree consisting of only a root node has a height of 0.
- The number of leaf nodes l in a perfect binary tree, is $l = (n + 1)/2$ because the number of non-leaf (a.k.a. internal) nodes $n - l = \sum_{k=0}^{\log_2(l)-1} 2^k = 2^{\log_2(l)} - 1 = l - 1$.
- This means that a perfect binary tree with l leaves has $n = 2l - 1$ nodes.
- In a **balanced** full binary tree, $h = \lceil \log_2(l) \rceil + 1 = \lceil \log_2((n + 1)/2) \rceil + 1 = \lceil \log_2(n + 1) \rceil$ (see ceiling function).[*citation needed*]
- In a **perfect** full binary tree, $l = 2^h$ thus $n = 2^{h+1} - 1$.
- The maximum possible number of null links (i.e., absent children of the nodes) in a **complete** binary tree of *n* nodes is (*n*+1), where only 1 node exists in bottom-most level to the far left.
- The number of internal nodes in a **complete** binary tree of *n* nodes is $\lfloor n/2 \rfloor$.
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.[24]

Traversals (Pre, In, Post)***

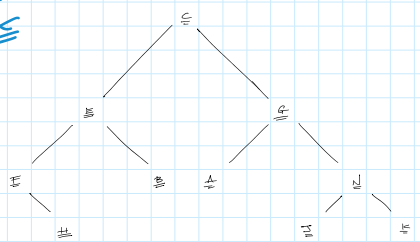
Sunday, December 9, 2018 2:22 PM

LINE METHOD HACK (SHORTCUT)

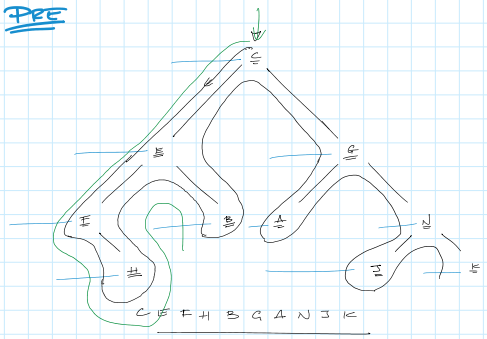
L = Go Left
V = Visit Node (Print)
R = Go Right



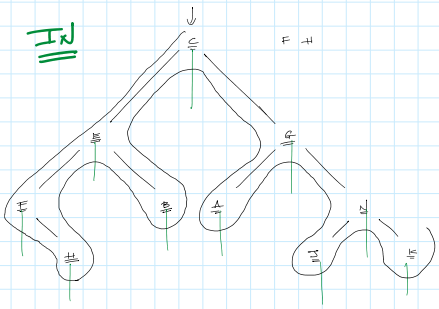
EX



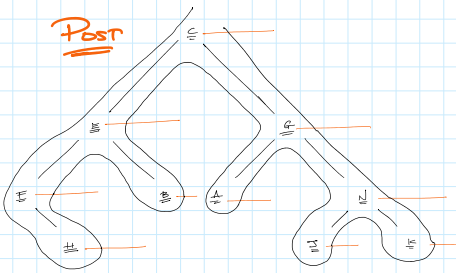
PRE



IN



POST



1. Consider the binary search tree (BST) in Figure 1 below:

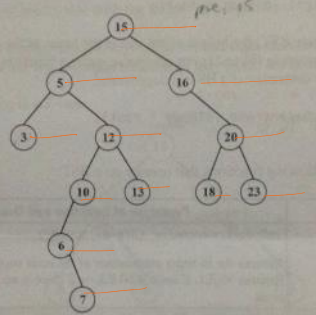


Figure 1

(a) Give the pre-order, in-order, post-order, and level-order traversal of the BST. [2]

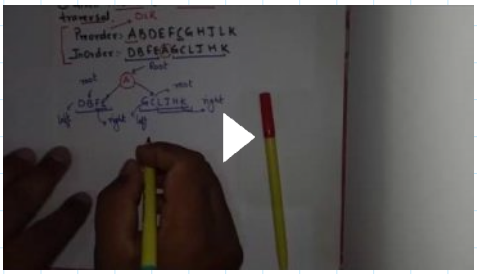
18-19 S1 COMP2611 CW1

Build from Pre and In order Traversals***

Sunday, December 9, 2018 2:22 PM

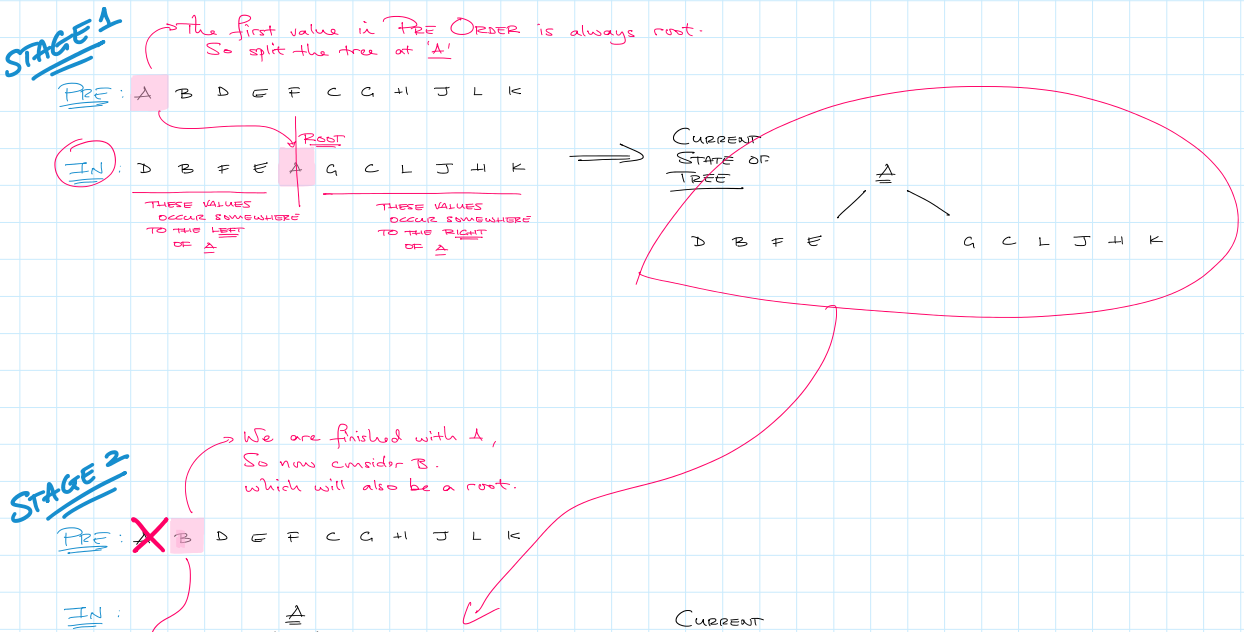
Examples

Construction of Binary Tree from PreOrder and InOrder Traversal(Hindi, English) with Example



Approach

Use Pre Order
(Moving from Left to Right)
to determine which Nodes are roots,
then split the tree

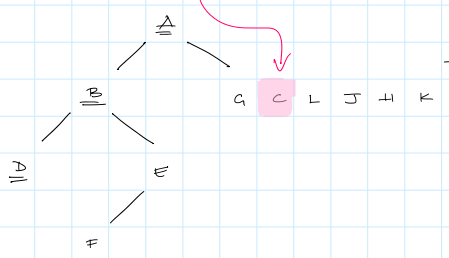


Hope you guys get the point...
cus i'm getting tired of writing out all the steps
so I'll move faster now!!

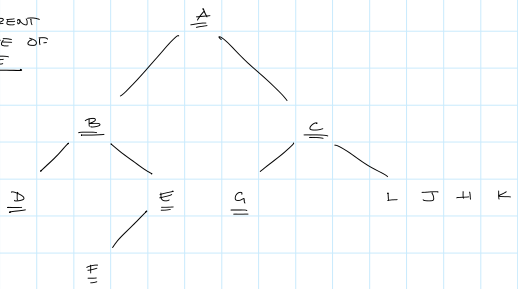
STAGE 4

PRE : ~~A B D E F~~ C G I J L K

IN : ~~D B E F A~~ G C L J I K



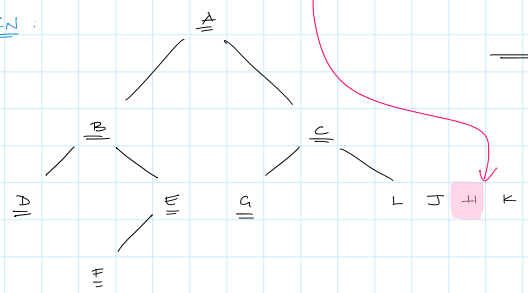
CURRENT STATE OF TREE



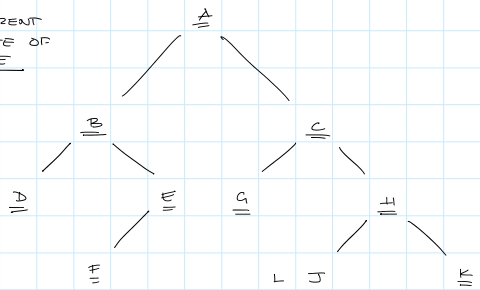
STAGE 5

PRE : ~~A B D E F C G~~ I J L K

IN : ~~D B E F A G C~~ L J I K



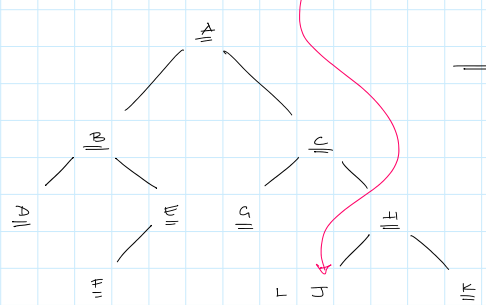
CURRENT STATE OF TREE



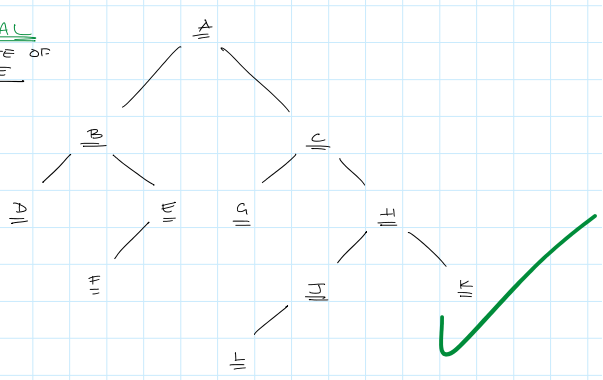
STAGE 6

PRE : ~~A B D E F C G I~~ J L K

IN : ~~D B E F A G C I~~ L J K



FINAL STATE OF TREE

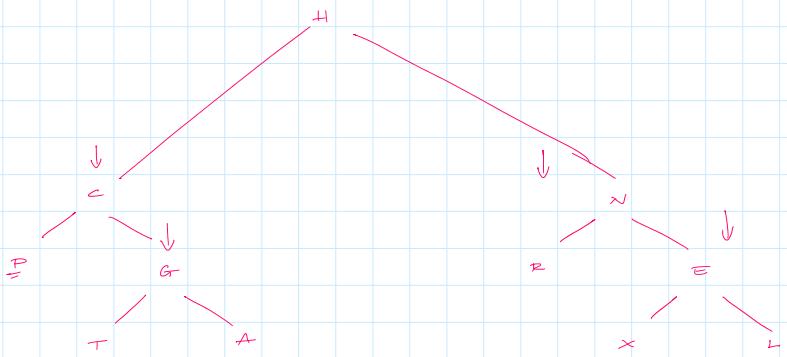


comp2611_1_18

1. (a) The following are the *pre-order* and *in-order* traversals of the nodes of a binary tree:

Pre-order: H C P G T A N R E X L
In-order: P C T G A H R N X E L

Draw the tree. [4]



COMP2000_1_17

1. (a) The following are the *pre-order* and *in-order* traversals of the nodes of a binary tree:

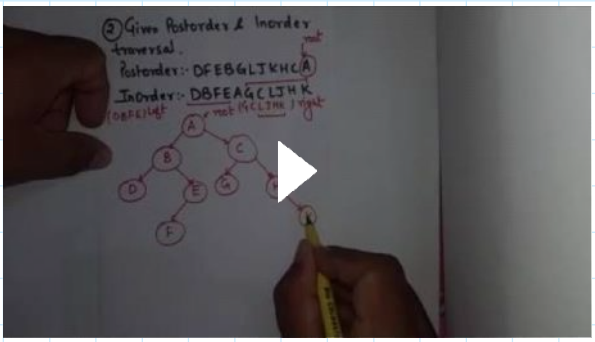
Pre-order: X K P H T A N R E S D
In-order: P K T H A X R N S E D

Draw the tree. [4]

Build from In and Post Order Traversals***

Sunday, December 9, 2018 3:27 PM

Construction of Binary Tree from PostOrder and InOrder Traversal(Hindi, English) with Example



Approach! (Similar to Preorder)

Use Post Order

(Moving from Right to Left)

to determine which Nodes are roots, then split the tree

COMP2000_2_17

c. You are given the following inorder and postorder traversals of a binary tree.

Inorder:	L	X	N	P	M	Y	R	B	S
Postorder:	L	N	P	X	R	S	B	Y	M

Draw the binary tree. [3 marks]

M

L X N P Y R B S

COMP2000_2_16

b. You are given the following postorder and inorder traversals of a binary tree.

Postorder	F	E	C	H	G	D	B	A
Inorder	F	C	E	A	B	H	D	G

Draw the binary tree. [4 marks]

Build from Array***

Sunday, December 9, 2018 3:38 PM

Binary Tree as an Array

$l = 2i$

$r = 2i + 1$

$p = i/2$

$2i = 10$ $\quad 5/2 = 2$

COMP2000_1_16

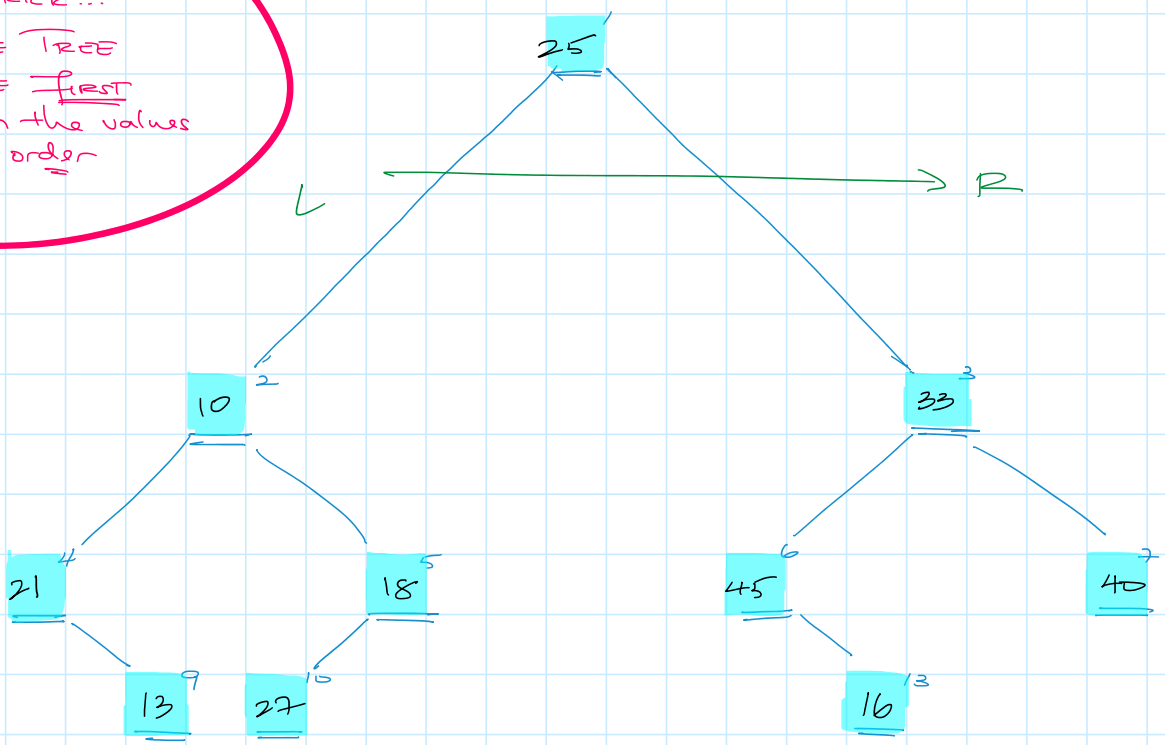
$i = 5$

(c) The following array represents a binary tree with the root in location 1. "@" represents the null pointer.

	1	2	3	4	5	6	7	8	9	10	11	12	13
	25	10	33	21	18	45	40	@	13	27	@	@	16
	i	2i	2i+1	i	i	i	2i+1	2i	2i+1	2i	2i+1	2i	2i+1

Draw the tree and give the in-order and post-order traversals of the tree. [3]

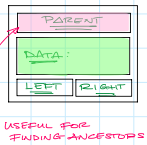
Clever Trick...
Draw the tree structure first
Then fill in the values in level order



C++ Pointer Implementation Visualized

Sunday, December 9, 2018 4:31 PM

To Store Integers

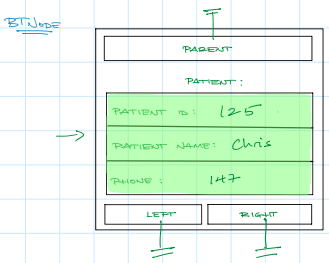


USEFUL FOR FINDING ANCESTORS

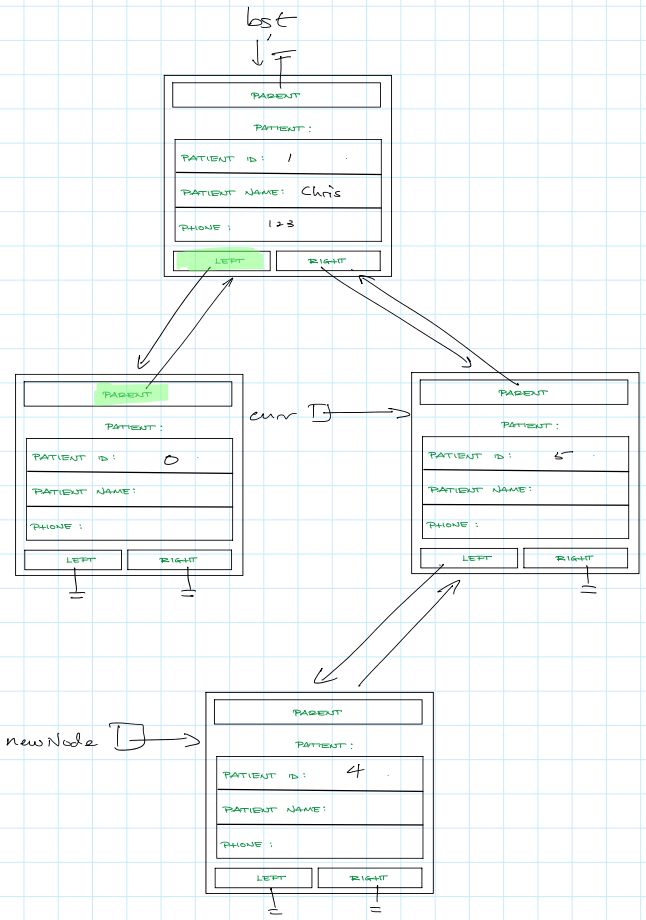
```
Struct Node {
    int data;
    Node *parent, *left, *right;
};
```

```
struct BTNode {
        string data;
        BTNode * left;
        BTNode * right;
        BTNode * parent;
};
```

To Store Patient Information




```
Struct Patient {
    string patientId, patientName, phone;
};

Struct Node {
    Patient data;
    Node *parent, *left, *right;
};
```



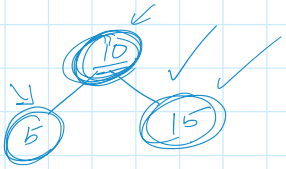
Binary Search Trees

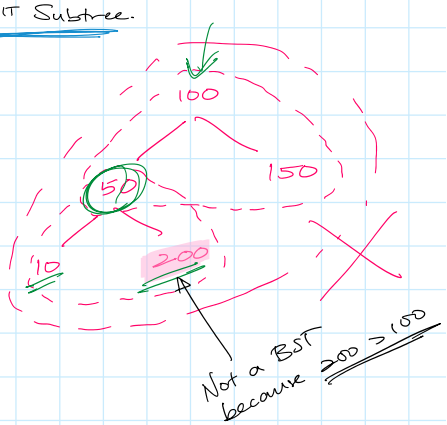
Saturday, December 8, 2018 9:57 PM

Bst checks
Null
End of tree
Leaf node
No left
No right

DEFINITION

A binary tree where
at any given node
the value in the Node is Bigger than all the values in the LEFT Subtree.
& Smaller than all the values in the RIGHT Subtree.



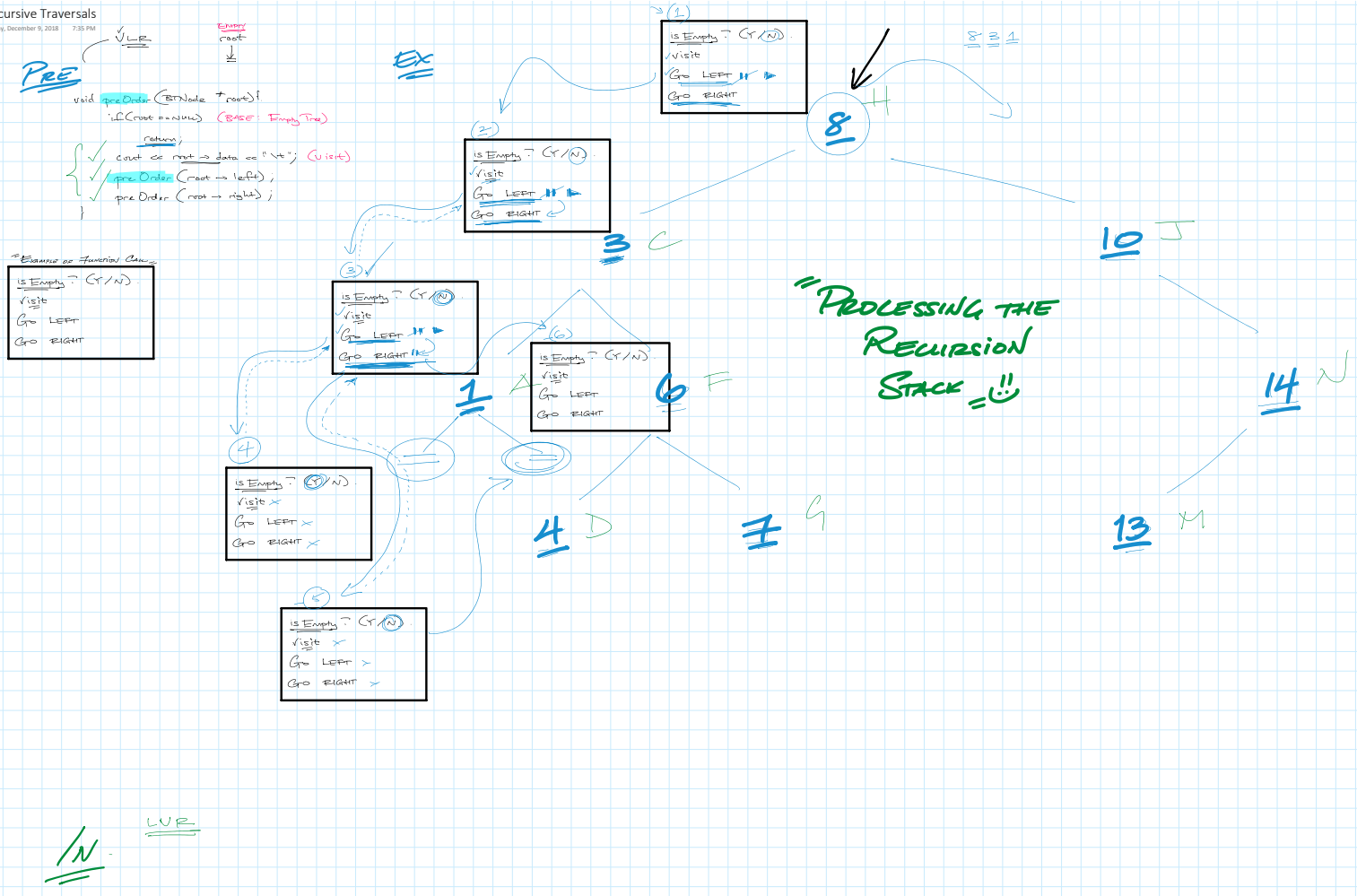


HUGE

Consider CASES when designing algorithms

Recursive Traversals

Sunday, December 9, 2018 7:35 PM



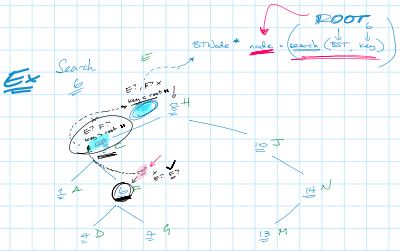
N LVR

Visualise the Cases!

Searching.

Algorithm

Start at root
if tree is empty or key is found in root. CASE 1 CASE 2
we are done.
if key < root CASE 3
check left subtree.
if key > root CASE 4
check right subtree.

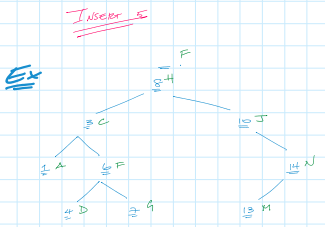


Picture Subtrees!

Inserting

New keys are always inserted as leaves.
This means we need to reach the appropriate leaf & insert there.

if we have reached an empty tree CASE 1
Insert here as leaf
if key is in left subtree CASE 2
recur down to left subtree until we reach a leaf.
set parent pointer
if key is in left subtree CASE 3
recur down to left subtree until we reach a leaf.
set parent pointer
return root (subtree)




```
49  // returns NULL, LEAF POINTERS, and SUBTREE POINTERS
50  BTNode * search(BTNode * root, string key) {
51
52      if(root == NULL || root->data == key)       // (BASE CASES) CASE 1: the tree at root is empty, CASE 2: the root node contains key
53          return root;
54
55      if(key < root->data)                        // CASE 3: the key may be in the left subtree
56          return search(root->left, key);
57
58      else                                        // CASE 4: the key may be in the right subtree
59          return search(root->right, key);
60
61  }
```

```
141 // returns NULL, LEAF POINTERS, and SUBTREE POINTERS
142 BTNode * insert(BTNode * root, string key) {
143     if(root == NULL)                                    // CASE 1: we have reached an empty tree
144         return createBTNode(key);                               // so insert here as a leaf
145
146     if(key < root->data){                               // CASE 2: key should be in the left subtree
147         BTNode *leftChildSubtree = insert(root->left, key);     // recur down the left subtree and catch whatever it returns (either the newNode or a subtree at root)
148         root->left = leftChildSubtree;                          // set the left subtree at root
149         leftChildSubtree->parent = root;                        // set parent of leftChildSubtree
150     }
151     if(key > root->data){                               // CASE 3: key should be in the right subtree
152         BTNode *rightChildSubtree = insert(root->right, key);   // recur down the right subtree and catch whatever it returns (either the newNode or a subtree at root)
153         root->right = rightChildSubtree;                        // set the right subtree at root
154         rightChildSubtree->parent = root;                       // set parent of rightChildSubtree
155     }
156
157     return root;                                        // return the subtree at root
158 }
```

Question***

Sunday, December 9, 2018 2:24 PM

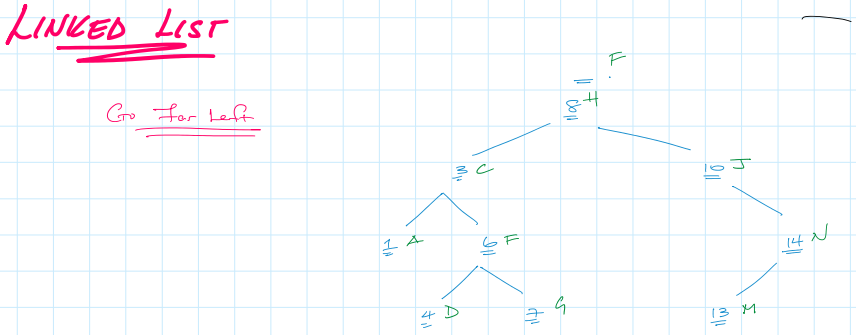
comp2611_1_18
COMP2000_1_17
comp2000_3_14

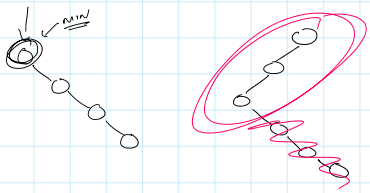
(b) Each node of a *binary search tree* has fields `left`, `right`, `key` (an integer) and `parent`, with the usual meanings.

(i) Write a function which, given a pointer to the root of the tree and an integer `n`, searches for `n`. If found, return a pointer to the node. If not found, add `n` to the tree, ensuring that *all fields* are set correctly; return a pointer to the new node. You may assume that the function call `newNode(n)` creates a node, stores `n` in it and returns a pointer to the node. [5]

min

Tuesday, December 11, 2018 8:54 AM






```
1   // Think about this like a linked list
2   BTNode * min(BTNode * root) {
3       BTNode* current = root;
4
5       if(current == NULL)                 // empty
6           return current;
7
8       while (current->left != NULL) {     // go all the way to the left
9           current = current->left;
10      }
11
12      return current;
13  }
```

max

Tuesday, December 11, 2018 8:54 AM

Linked List

Go far Right

```
16  // Think about this like a linked list
17  BTNode * max(BTNode * root) {
18      BTNode* current = root;
19
20      if(current == NULL)                 // empty
21          return current;
22
23      while (current->right != NULL) {    // go all the way to the right
24          current = current->right;
25      }
26
27      return current;
28  }
```

Successors and Predecessors

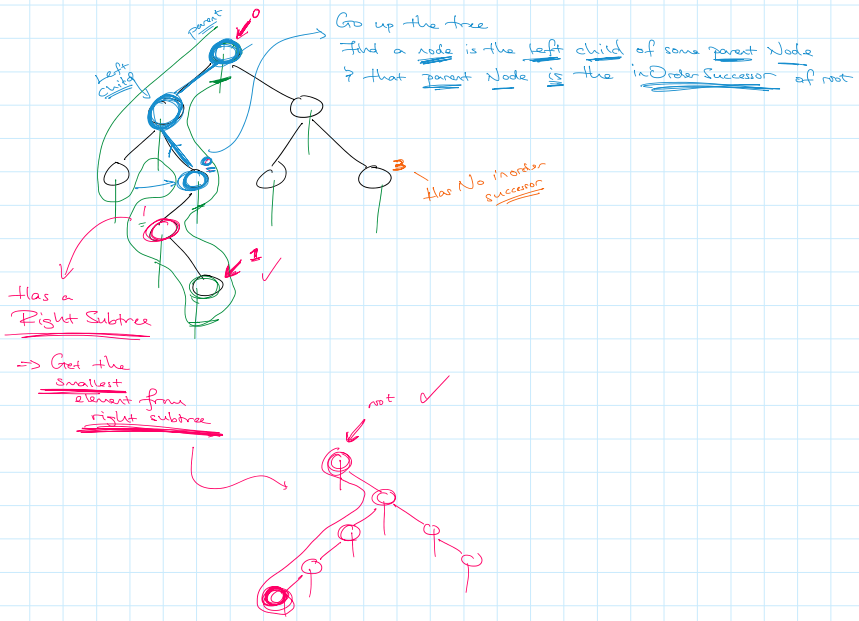
Sunday, December 9, 2018 2:44 PM

inOrderSuccessor

Sunday, December 9, 2018 3:26 PM

https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/

If has Right Subtree then.
Minimum value in Right Subtree.
Else
Get first Node which is the left child of its parent.
This parent is InOrderSuccessor




```
124  struct BTNode * inOrderSuccessor_G4G(struct BTNode *root, struct node *n)
125  {
126    // step 1 of the above algorithm
127    if( n->right != NULL )                  // if we have a right subtree
128      return min(n->right);                     // the inOrderSuccessor is the smallest element in that right subtree
129
130    // step 2 of the above algorithm
131    struct BTNode *p = n->parent;
132    while(p != NULL && n == p->right)       // while n has a parent AND n is a right child of that parent
133    {
134       n = p;
135       p = p->parent;                           // keep going up the tree
136    }
137    return p;                               // stop when the node n is a left child of parent
138  }
```

inOrderPredecessor

Sunday, December 9, 2018 7:53 PM

If has Left Subtree.

Maximum Value in left Subtree

Else

Get first Node that is Right Child of its parent

this parent is InOrderPredecessor

} Opposite of In order Successor

Delete Root, replace with inOrderPredecessor***

Sunday, December 9, 2018 3:02 PM

comp2000_3_14

(c) Given a pointer to the root of a *binary search tree* of integers, delete the root and return a pointer to the root of the new tree. If the root has non-empty left and right subtrees, replace it by its *inorder predecessor*. Assume that each node of the tree contains 3 fields – **data**, **left** and **right**. [6]

preOrderSuccessor***

Sunday, December 9, 2018 2:44 PM

COMP2000_1_17

(ii) Write a function which, given a pointer to any node of the tree, returns a pointer to the *pre-order successor* of the node. Return `NULL` if there is no successor. [3]

1. If left child of given node exists, then the left child is preorder successor.
2. If left child does not exist and given node is left child of its parent, then its sibling is its preorder successor.
3. If none of above conditions are satisfied (left child does not exist and given node is not left child of its parent), then we move up using parent pointers until one of the following happens.
    - We reach root. In this case, preorder successor does not exist.
    - Current node (one of the ancestors of given node) is left child of its parent, in this case preorder successor is sibling of current node.

From <https://www.geeksforgeeks.org/preorder-successor-node-binary-tree/>

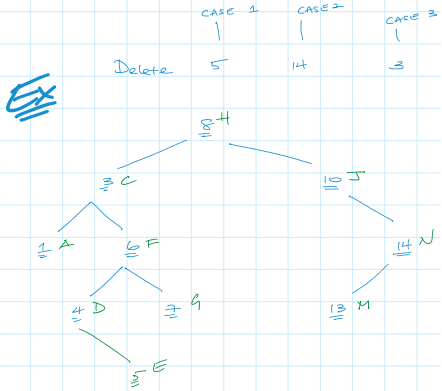
Delete Any Node

Sunday, December 9, 2018 2:24 PM

Kali Page 152+8
Cases!!!

3 Possible Cases

(0) Node does not exist.
(i) Node to be deleted is a leaf
=> Just Delete
(ii) Node has 1 child (subtree)
=> Replace with child
(iii) Node has 2 children (subtrees)
=> Copy the inorder successor node & delete inorder successor.
(Replace with. Or inorder predecessor)
Smallest value in right subtree



1. Consider the binary search tree (BST) in Figure 1 below:

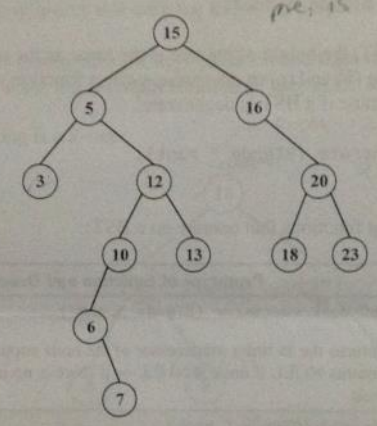


Figure 1

(b) Draw separate diagrams to show the BST in Figure 1 *after* **each** of the following nodes has been deleted, in the order given:

- The node with key 13.
- The node with key 16.
- The node with key 5. [4]

18-19 S1 COMP2611 CW1

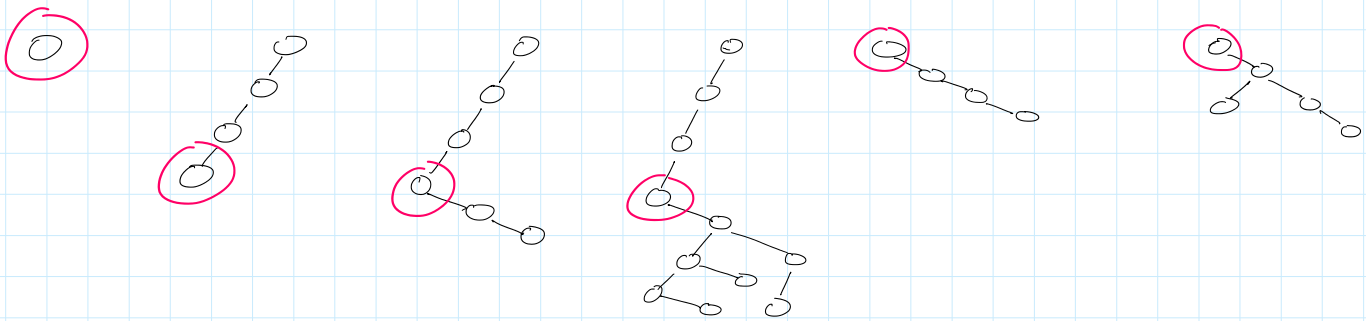
Delete Smallest***

Sunday, December 9, 2018 2:45 PM

COMP2000_1_17
COMP2000_1_16

(c) Write a function `deleteSmallest` which, given a pointer to the root of a *binary search tree*, deletes the node containing the *smallest* value and returns a pointer to the root of the (new) tree. Your function must work for all cases. [4]

Go far left (Get Smallest Element)
Replace with Right Child



Delete Largest***

Sunday, December 9, 2018 2:25 PM

comp2611_1_18

(c) Write a function `deleteLargest` which, given a pointer to the root of a *binary search tree*, deletes the node containing the *largest* value and returns a pointer to the root of the (new) tree. Your function must work for all cases. [3]

CASES Root

0 1 Child

InOrder Traversal (Non recursive)

Wednesday, December 12, 2018 8:59 AM

Uses a Stack

Non-Recursive

void	**inOrderIterative**(BTNode * root) Outputs the in-order traversal of the binary tree. The code must be non-recursive.

COMP2000_1_16

(d) An almost complete binary tree of n nodes is stored in an an integer array `num[1..n]`. Write a structured, non-recursive algorithm to print the numbers in *inorder*. You may assume the existence of procedures push and pop for manipulating a stack. [4]

https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
https://www.geeksforgeeks.org/iterative-preorder-traversal/

INORDER

Start at curr root
Repeat until current is NULL AND stack is empty.
  push all left nodes of curr into stack. — Because we always go for left first.
  pop into curr
  visit curr — then visit
  go right from curr — then go right

↑ Go for left AGAIN!

Level Order Traversal (Non recursive)

Sunday, December 9, 2018 3:12 PM

```
//Uses a Queue//

Add the root to Queue.
While Queue is not Empty.
    Dequeue Queue into root
    visit root.
    if root has left child
        Queue left child.
    if root has right child
        Queue right child.
```

isBST***

Sunday, December 9, 2018 4:42 PM

3. Assume that a node in a binary tree is defined as follows:

```
struct BTNode {
       int data;
       BTNode * left;
       BTNode * right;
       BTNode * parent;
};
```

(a) Write a recursive function, *isBST*, which accepts *bt*, a pointer to a *BTNode* as a parameter, and determines if the binary tree rooted at *bt* is a binary search tree. Assume that there are no duplicate keys. [4 marks]

Check that Max Value in left Subtree is Less than Node
AND Min Value in Right Subtree is More than Node
Recur down tree

```
36  bool isBST(BTNode *root){
37      if(root == NULL)                                    // if the tree is empty
38          return true;
39      if(root->left != NULL && max(root->left)->data > root->data)    // max value of left subtree must be less than root value, if not then not a BST
40          return false;
41      if(root->right != NULL && min(root->right)->data < root->data)  // min value of right subtree must be more than root value, if not then not a BST
42          return false;
43      if(!isBST(root->left) || !isBST(root->right))       // recur down the tree
44          return false;
45      return true;                                        // is BST
46  }
```

Weight, numLeaves***

Sunday, December 9, 2018 3:13 PM

int	**weight**(BTNode * root) Returns the weight of the binary tree. The weight of a binary tree is the amount of leaves in the tree.

numLeaves :

if root is NULL (No leaves)

return 0

if (No left or right subtree) (Leaf)

return 1

return numLeaves (root -> left) + numLeaves (root -> right) (recur down tree)

Moment, numNodes***

Wednesday, December 12, 2018 2:27 AM

int	**moment**(BTNode * root) Returns the moment of the binary tree. The moment of a binary tree is the amount of nodes in the binary tree.

```
numNodes (root):
    if (root is Null)        (Empty)
        return 0
    return numNodes (root -> left) + 1 + numNodes (root -> right);
```

Node we are ON

(Recur down the tree)

number of Nodes in Left Subtree

number of Nodes in Right Subtree

(b) Write a *non-recursive* function with the following prototype to find the number of nodes in a binary tree:

```
int numNodes (BTNode *root);
```

You may use a *Stack* or *Queue*, with the usual definitions. [5]

// USES A QUEUE, PERFORM LEVEL ORDER TRAVERSAL

```
count = 0
Add the root to Queue.
While Queue is not Empty.
    Dequeue Queue into root
    count ++
    if root has left child
        Queue left child.
    if root has right child
        Queue right child.
```

Height***

Sunday, December 9, 2018 3:14 PM

(c) Write a *recursive* function with the following prototype to find the height of a binary tree: [2]

```
int height (BTNode * root);
```

18-19 S1 COMP2611 CW1

See c++ file

isDegenerate***

Sunday, December 9, 2018 7:48 PM

(d) In a degenerate BST, the height of the tree is the same as the number of nodes. Using the functions written in (b) and (c) or otherwise, write a function with the following prototype to determine if a BST is degenerate:

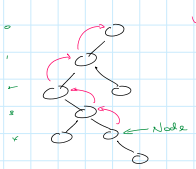
```
bool isDegenerate (BTNode * root);
```

[2]

Just a linked list
Height = moment

getLevel***

Sunday, December 9, 2018 7:59 PM



Visualisation

18-19 S1 COMP2611 CW2

(b) You are given the address of a random node in a binary tree, *node*. Write code to find out the *level* of the node in the tree. Assume the root is at level 0. [3 marks]

Total Marks: 25

https://www.geeksforgeeks.org/get-level-of-a-node-in-a-binary-tree/

```
void getLevel (BTNode *node) {
    int count = 0;
    while (node -> parent != NULL) {   — while we have not reached the root
        count++;                       — Increment level Count
        node = node -> parent;         — Go up the tree.
    }
    return count;
}
```

Rank***

Wednesday, December 12, 2018 6:29 AM

int	**rank**(BTNode * root, **int** key) Returns the number of keys in the BST which are strictly less than *key*.

count = 0

Start @ the root key

Repeat until we reach the root of the tree

get IN ORDER PREDECESSOR

count++

countLessThenKey:

```
curr = search(key)
count = 0
while(curr != NULL):
    count++;
    curr = InOrderPredecessor(curr)
```

reverseInOrder***

Wednesday, December 12, 2018 6:30 AM

void	reverseInOrder (BTNode * root) Outputs the in-order traversal of the BST *in reverse order*.

Start @ the max value in BST
Perform Inorder Predecessor until NULL

```
reverseInOrder (root):
    node = max (root)
    while (node != NULL)
        visit node
        node = InOrderPredecessor (node)
```

Level Count

Sunday, December 9, 2018 3:03 PM

Question***

Sunday, December 9, 2018 3:03 PM

comp2000_3_14

(d) Write a function which, given the root of binary tree and an integer **k**, returns the *number of nodes* at level **k**. Assume the root is at level 0 and an appropriate queue is available with the usual operations. Pseudocode may be used in the body of the function. Hint: store a node and its level in each queue item. [6]

Level Sum

Sunday, December 9, 2018 2:56 PM

Question***

Sunday, December 9, 2018 2:56 PM

comp2000_3_14

(d) Write a function which, given a pointer to the root of a binary tree, returns the sum of the levels of the nodes in the tree. Assume the root is at level 0. [3]

External Nodes Question***

Sunday, December 9, 2018 2:55 PM

comp2000_3_14

(c) In a binary tree of n nodes, an 'external' node is attached to each null pointer. If I is the sum of the levels of the (internal) nodes of the tree and E is the sum of the levels of the external nodes, prove that $E - I = 2n$. [3]

Heaps

Sunday, December 2, 2018 3:04 PM

DEFINITION

Think HEAP (Hinds)

An almost complete BT s.t

(1) the value at the root is greater than or equal to the values @ the L and R children (max heap)

AND (2) L and R subtrees are also heaps.

The largest value is always at the root

Implemented using arrays

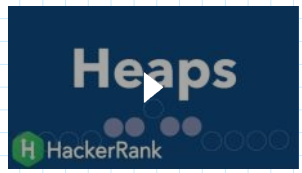
LEFT CHILD : $2i$

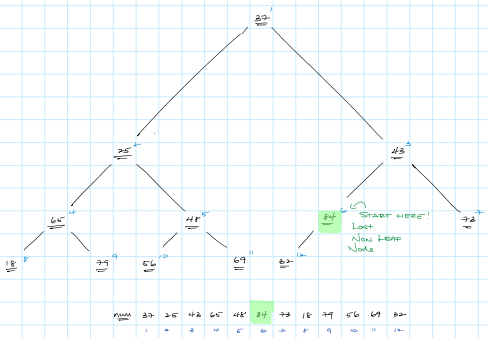
RIGHT CHILD : $2i + 1$

PARENT : $i/2$

SIFT DOWN – Making a heap from a BT Array.

Data Structures: Heaps





Observe all leaves are heaps.

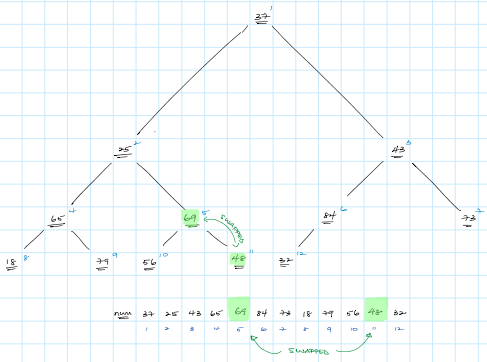
Start at the last non leaf node (6) & convert the tree rooted there to a max heap

It is already a max heap, so do nothing.

Move backwards to the next non-leaf node. (5) converting it to a max heap.

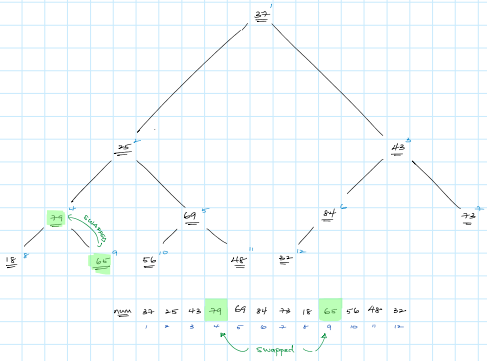
Swap with the larger of its children

Being a Non-Leaf Node
- it will always have a left child.
- may not have a right child.



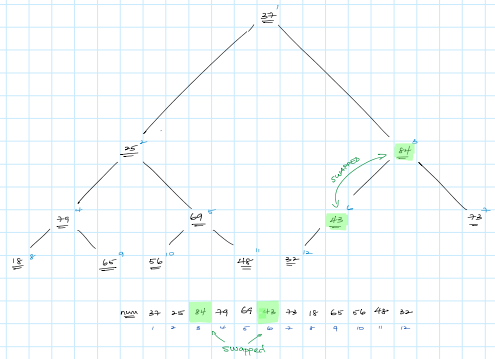
Move backwards to the next non-leaf node. (4) converting it to a max heap.

Swap with the larger of its children



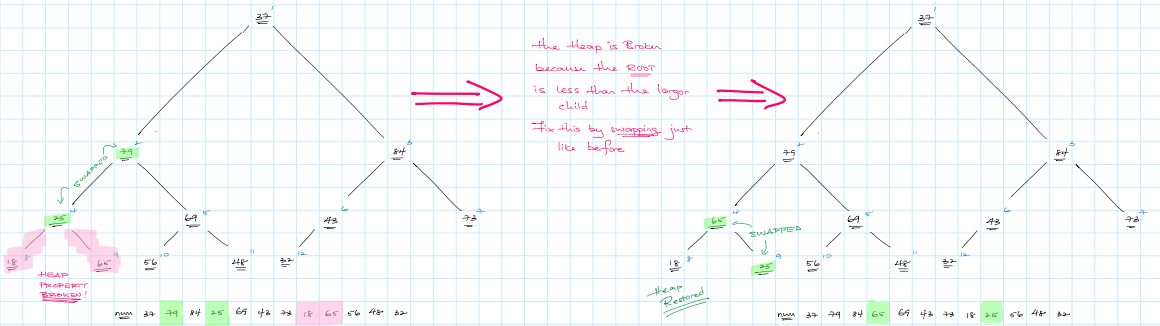
Move backwards to the next non-leaf node: (3) converting it to a max heap.

Swap with the larger of its children



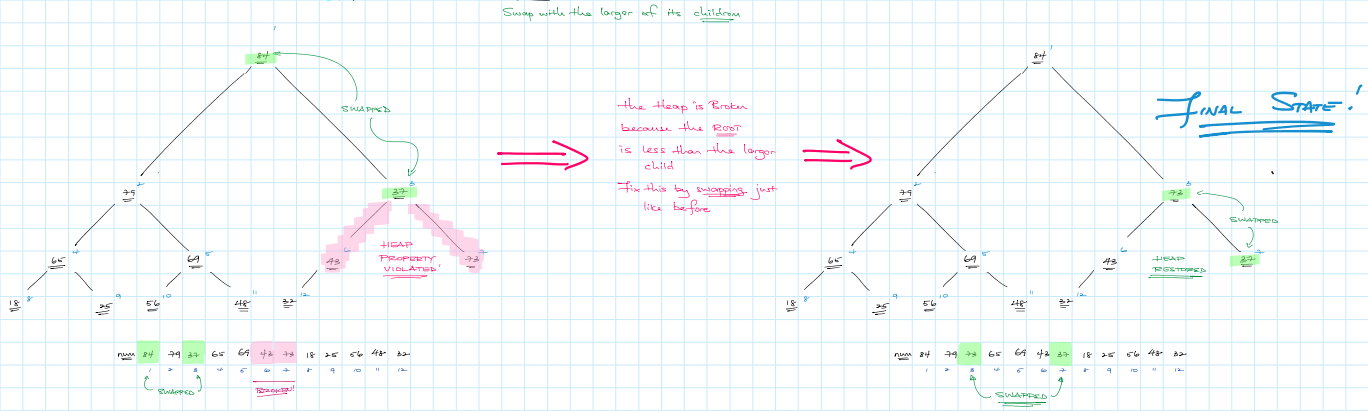
Move backwards to the next non-leaf node: (2) converting it to a max heap.

Swap with the larger of its children



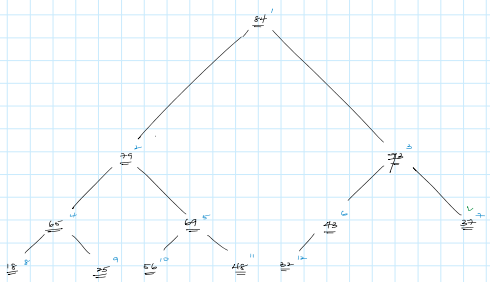
Move backwards to the next non-leaf node: (1) converting it to a max heap.

Swap with the larger of its children



```java
// puts the new item in location i and sifts down
void siftDown_v2 (int newItem, int heap[], int heapSize, int i) {

    heap[i] = newItem;                                          // put newItem in heap
    while(hasLeftChild(i, heapSize)){                           // while i has children (only need to check for left because heaps store left to right)

        int largerIdx = leftChild(i);                               // determine the larger of two children index
        if(hasRightChild(i, heapSize) && heap[rightChild(i)] > heap[leftChild(i)]){
            largerIdx = rightChild(i);
        }

        if(heap[i] > heap[largerIdx]){                              // if newItem is larger than the larger child, do nothing
            return;
        }

        swap(A, largerIdx, i);                                      // else swap with the larger child and move down the tree
        i = largerIdx;
    }
}
```

Sift UP



```java
// sift up the value in A[i]
void siftUp(int A[], int n, int i){
    while(hasParent(i, n)){                     // while we have a parent
        int parentIdx = parent(i);
        if(A[i] < A[parentIdx])                 // if i is less than parent, then it is a heap, so STOP
            return;
        swap(A, i, parentIdx);                  // else swap parent and i
        i = parentIdx;                          // move up the tree
    }
}
```

isMaxHeap***

Sunday, December 9, 2018 7:53 PM

Array Representation

18-19 S1 COMP2611 CW2

(a) Write the function, *isMaxHeap*, given in Table 1. You can use any of the functions given in Table 1. [6 marks]

```
bool isMaxHeap(int A[], int i, int n){
    if(!hasLeftChild(i))                                    // BASE CASE: if it is a leaf (no children AKA no left child)
        return true;                                            // all leaves are heaps

    // else, we have at least a left child
    if(A[leftChild(i)] > A[i] || (hasRightChild(i, n) && A[rightChild(i)] > A[i]))      // if either of the children are bigger than the parent
        return false;                                           // not a heap

    return isMaxHeap(A, leftChild(i), n) && isMaxHeap(A, rightChild(i), n);             // else recur down the heap and check maxheaps at each child
}
```

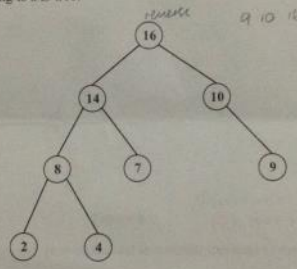
isZTree***

Saturday, December 15, 2018 4:35 PM

4. A *Z-tree* is a certain type of binary tree with the following properties:

- The key of the root is greater than the keys of its left and right children (if they exist)
- The subtrees rooted at the left and right children (if they exist) are themselves Z-trees.

For example, the following is a Z-tree:



(a) Write a recursive function with the following prototype, to determine if a binary tree is a Z-tree. Assume that the keys are positive integers.

```
bool isZTree (BSTNode * root);
```

[5]

NOTE:

this is a Heap (Max Heap)

```
isZTree (root)
    if (isLeafNode (root))
        true.

    If (hasLeftChild (root) AND LeftChild > root
        OR hasRightChild AND RightChild > root)
        FALSE

    else  recur down tree checking if L & R child subtrees are trees.
        isZTree (root -> left) AND isZTree (root -> right)
```

Sift Up Question***

Sunday, December 9, 2018 2:32 PM

comp2611_1_18
comp2000_3_14
COMP2000_1_16

(d) The integer elements of an almost complete binary tree are stored in an array `A[1..n]`, with the root in location 1.

(i) *Without sorting*, write a function to rearrange the elements of A so that the *largest* integer is in location 1, and the largest integer of each subtree is also in the root of that subtree. The elements are to be processed in the order `A[2]`, `A[3]`, and so on, up to `A[n]`. [5]

→ Indicates Sift Up

(iii) Given an array like A, with `A[1]` to `A[n]` containing a *min*-heap, write a self-contained function to sort the array in *descending* order. [6]

COMP2000_1_16

(ii) If the array A contains the following values initially (n = 8):

53 36 25 47 61 32 75 59

Show the contents of `A[1]` to `A[i]` (i = 2 to 8) after each element `A[i]` is processed. [3]

(ii) If the array **A** contains the following values initially (**n** = 8):

comp2000_3_14

42 25 14 36 50 21 12 31

show the contents of **A[1]** to **A[j]** after each element **A[j]** is processed (j = 2..8).[3]

(ii) Suppose the array A contains the following values initially (remember `A[0]` is the number of elements in the heap):

8	29	18	14	23	20	12	31	16
0	1	2	3	4	5	6	7	8

Show the contents of `A[1]` to `A[j]` after element `A[j]` (`j = 2..8`) is processed. [3]

Sift Down Question***

Sunday, December 9, 2018 2:47 PM

COMP2000_1_17

(d) An *integer min*-heap is stored in an array (A, say) such that the size of the heap (n, say) is stored in `A[0]` and `A[1]` to `A[n]` contain the elements of the heap with the *smallest* value in `A[1]`.

Write a function `deleteMin` which, given an array like A, deletes the smallest element and reorganizes the array so that it remains a heap. [4]

Priority Queue***

Friday, December 7, 2018 12:52 AM

18-19 S1 COMP2611 CW2

(f) A process on a computer has a process ID, a process name, and a priority as shown in the structure below.

```
struct Process {
   int ID;
   string name;
   int priority;
};
```

The processes to be executed on a computer are kept in a max priority queue. The next process to be executed is the one with the highest priority. The max priority queue is implemented using a max-heap.

Write a function *changePriority*, which given the max priority queue, the queue size, the location *i* of a process in the queue, and the new priority, changes the priority of process *i*.

NB: The new priority can be smaller or bigger than the current priority. [5 marks]

```
if newPriority > ParentPriority :
    siftUp(NewPriority)
else
    siftDown(NewPriority)
```

— they want you to write this (See page on Sift UP)

— this was given

AVL Trees***

Sunday, December 9, 2018 2:17 PM

18-19 S1 COMP2611 CW3

The use of non-pivate[illegible] allowed.

1. Assume that a node in a binary tree is defined as follows.

```
struct BTNode {
        int data;
        BTNode * left;
        BTNode * right;
};
```

An *AVL tree* is a binary search tree that is **height-balanced**, i.e., for each node x, the heights of the left and right subtrees of x differ by at most 1.

a. Write a function, *isAVLTree*, with the following prototype, that accepts a binary search tree rooted at *bst* as a parameter and returns *true* if *bst* is an AVL tree, and *false* otherwise:

```
bool isAVLTree (BTNode * bst);
```

NB: Assume that a function, *height*, with the following prototype, is available and returns the height of the binary tree rooted at *bt*:

```
int height (BTNode * bt);
```

[5 marks]

b. The following keys are inserted into a binary search tree, in the order given:

12 8 18 5 4 11 16

i. Determine if the resulting binary search tree is an AVL tree. Show your working. [2 marks]

ii. Give the *pre-order traversal* of the resulting binary search tree. [1 mark]

c. The following keys are inserted into a binary search tree in the order given:

12 8 11 5 7 4 18 16 2

Determine if the resulting binary search tree is an AVL tree. Show your working.

[2 marks]

isAVLTree***

Saturday, December 15, 2018 6:39 PM

An *AVL tree* is a binary search tree that is **height-balanced**, i.e., for each node x, the heights of the left and right subtrees of x differ by at most 1.

a. Write a function, *isAVLTree*, with the following prototype, that accepts a binary search tree rooted at *bst* as a parameter and returns *true* if *bst* is an AVL tree, and *false* otherwise:

```
bool isAVLTree (BTNode * bst);
```

NB: Assume that a function, *height*, with the following prototype, is available and returns the height of the binary tree rooted at *bt*:

```
int height (BTNode * bt);
```

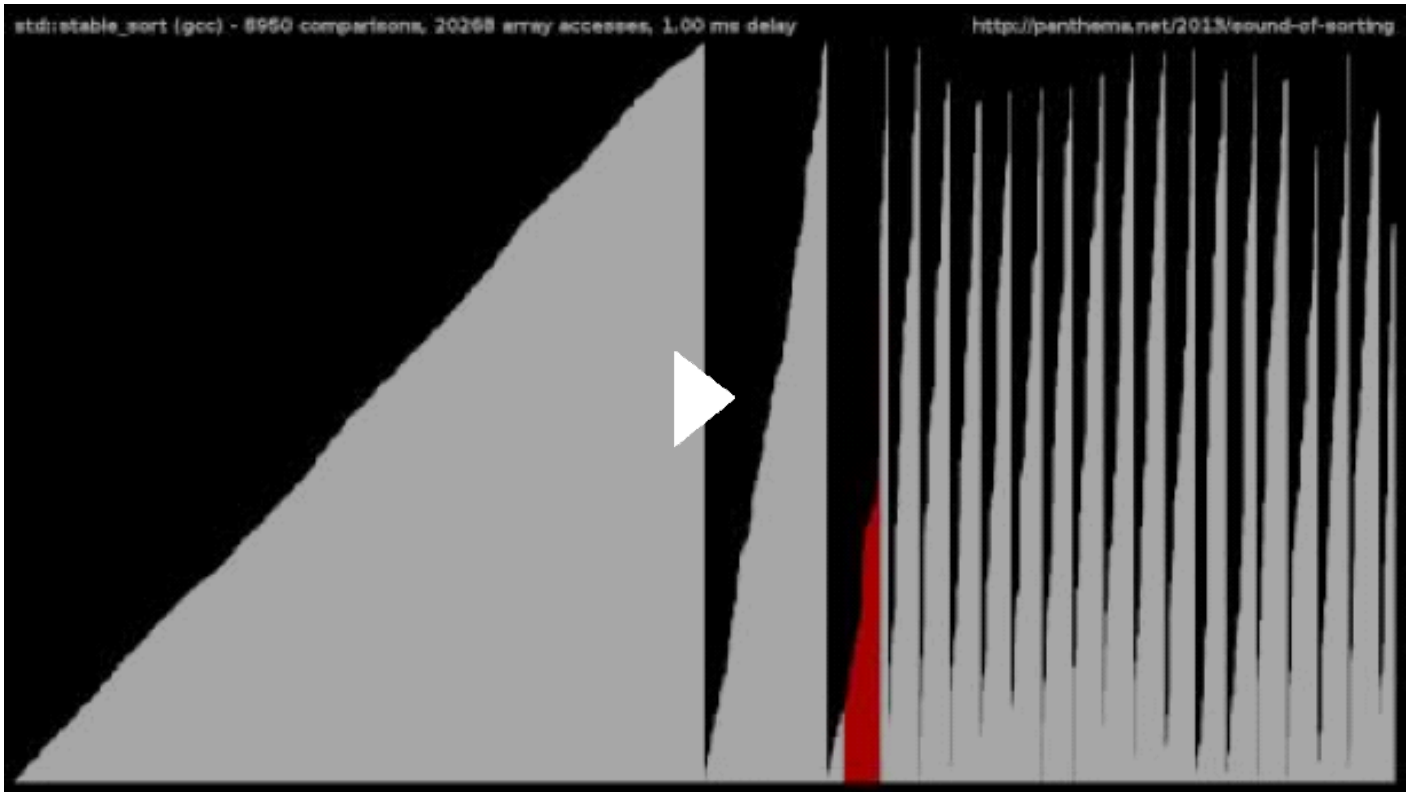
[5 marks]

```
bool isAVL (BTNode *root){
    if (root == NULL)          // BASE CASE
        return true;
    if ( abs (height (root -> left) - height (root -> right)) > 1)   // if the height of left & Right Subtrees differ by More than 1
        return false;
    return isAVL (root -> left) && isAVL (root -> right);   // Recur down the tree & check AVL on left & Right Subtrees
```

The Sound of Sorting…

Friday, December 7, 2018 3:45 PM

15 Sorting Algorithms in 6 Minutes



Heap sort

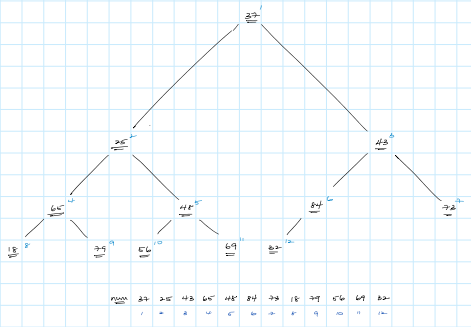
Sunday, December 2, 2018 6:38 PM

Almost complete BT at the

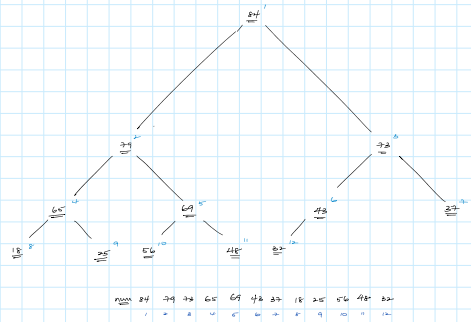
Uses a heap represented as an Array

Process

Start off with the (original) non-heapified array



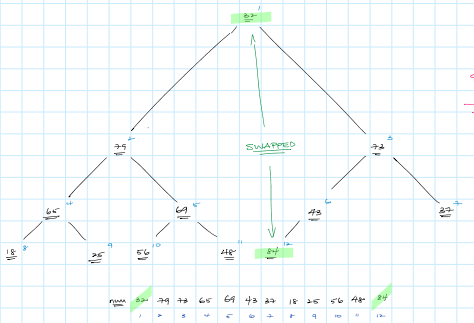
Use Sift Down to build (ascending) max heap or (descending) min heap

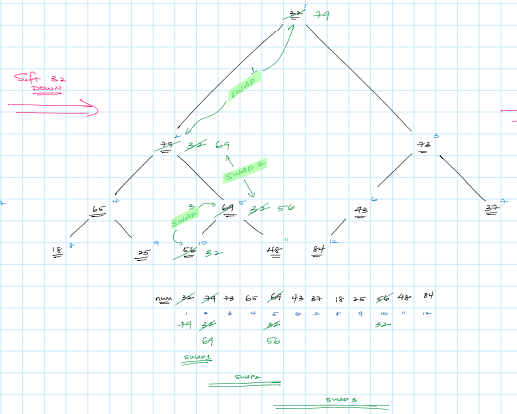


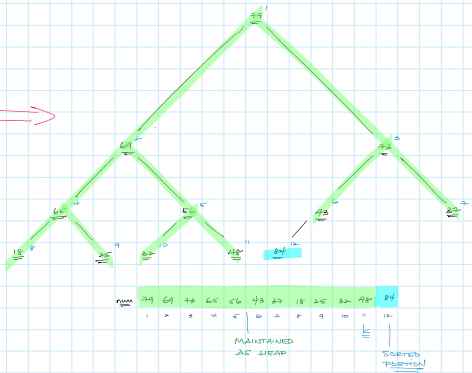
Iteration 1

Swap the (1) first and (k) last items and sift down from (1) to (k-1)

k is current size of heap. (initially k=n)



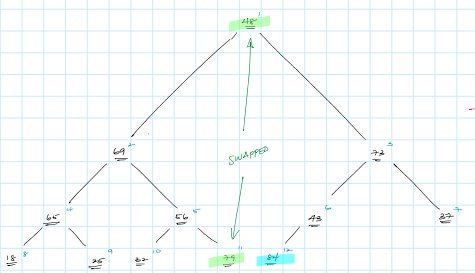


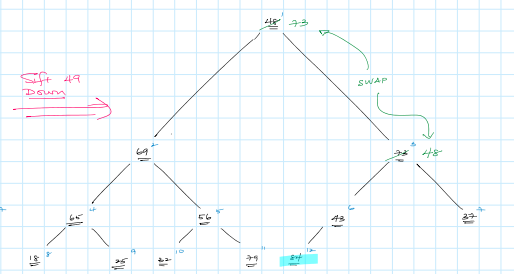


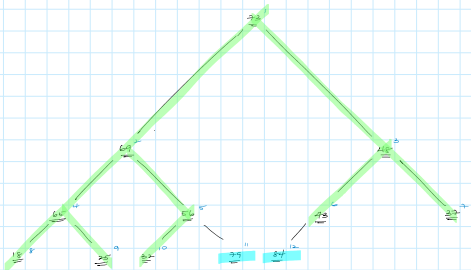
Iteration 2

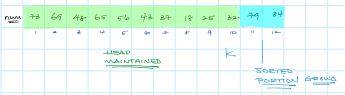
Swap the (1) first and (k) last items and sift down from (1) to (k-1)

k is current size of heap. (initially k=n)









Continue Until k = 2

I'm tired so I won't do the rest
Plus I think you got the point.

```
8  void heapSort(int A[], int n){
9      for(int i = n; i >= 2; i--){
10         swap(A, 1, i);
11         siftDown(A[1], A, i-1, 1);
12     }
13 }
14
```

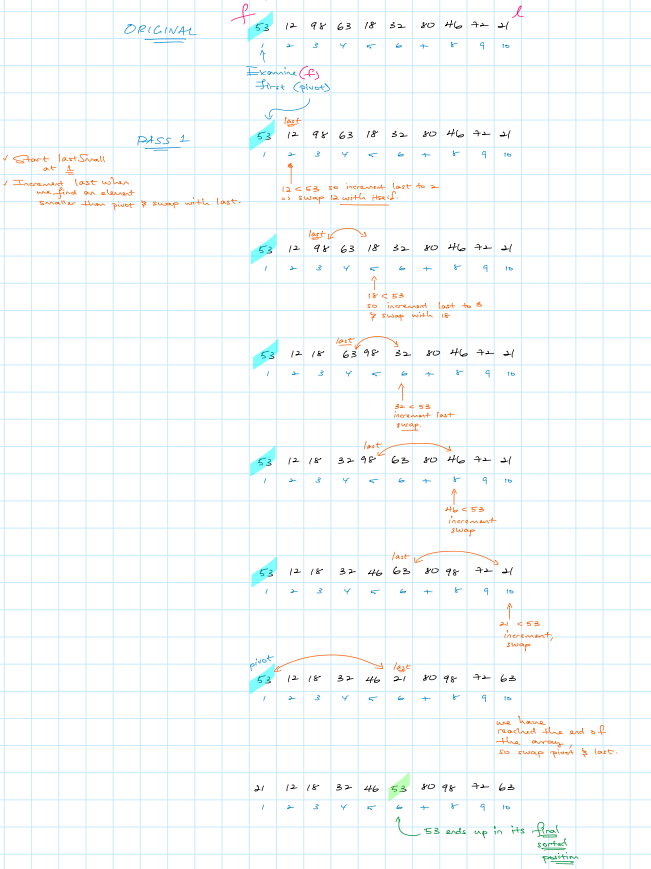
Partitioning

Sunday, December 9, 2018 2:52 PM

GOAL: At each stage, place the item (pivot) being examined in such a location that all the values to the left are smaller & to the right are bigger.

VERSION #1

Pivot ends up in final sorted position

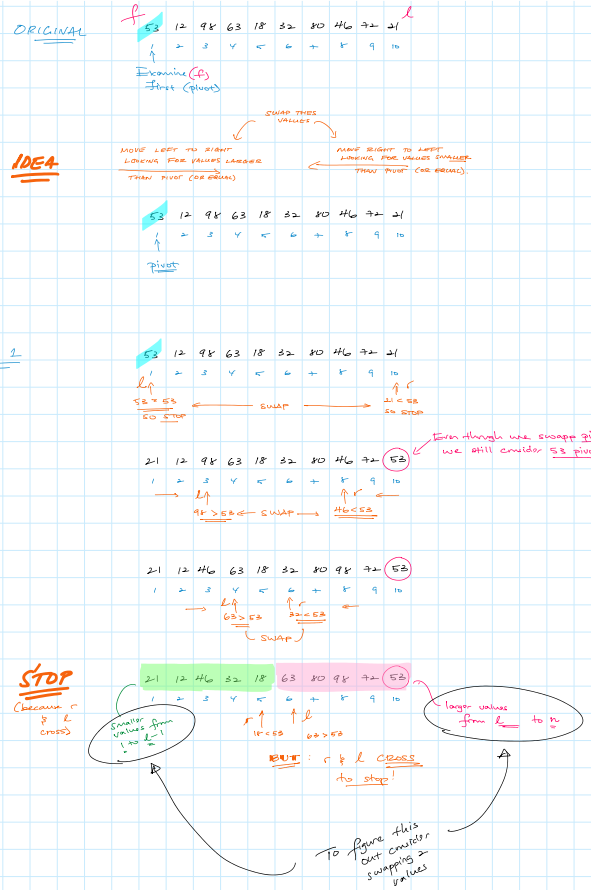


ALGORITHM

```
8   // puts pivot in its place
9   int partition_v1(int A[], int startIndex, int endIndex){
10      int pivotLocation = startIndex;                 // consider the startIndex of the array as the pivotLocation
11      int lastSmallerIndex = pivotLocation;           // consider the pivotLocation the initial lastSmallerIndex
12
13      for(int i = pivotLocation+1; i <= endIndex; i++)    // loop from pivotLocation+1 to endIndex
14          if(A[i] < A[pivotLocation])                         // if we find a value that is smaller than the pivot value
15              swap(A, ++lastSmallerIndex, i);                     // FIRST increment lastSmallerIndex and, THEN swap with the smaller value we found
16
17      swap(A, pivotLocation, lastSmallerIndex);       // swap pivot and lastSmallerIndex
18      return lastSmallerIndex;                        // return the division point (dp)
19  }
```

VERSION #2

Push smaller values to the right & bigger values to the left.
Pivot does not end up in final sorted position.



ALGORITHM

```
22  // does not put pivot in its place
23  int partition_v2(int A[], int left, int right){
24      int pivotValue = A[left];                   // make left the pivot value
25      while(left <= right){                       // while left and right have not crossed
26          while(A[left] < pivotValue) left++;         // go left until we find a value greater than or equal to pivot
27          while(A[right] > pivotValue) right--;       // go right until we find a value less than or equal to pivot
28          if(left <= right) swap(A, left++, right--);     // swap left and right values THEN increment left and decrement right
29      }
30      return left;                                // this is our division point: all values from [1..left-1] are smaller than [l]
31  }
```

Partition Question***

Sunday, December 9, 2018 2:59 PM

comp2000_3_14

(e) A function is given an integer array **A** and two subscripts **m** and **n**. *Without sorting*, the function must rearrange the elements **A[m]** to **A[n]** by placing element **A[m]** in location **d** such that all elements to the left of **d** are less than or equal to **A[d]** and all elements to the right of **d** are greater than **A[d]**. Write the function to rearrange the elements and return **d**. [4]

Kth Smallest

Sunday, December 9, 2018 2:51 PM

Uses Partition #1

At the end of each call to Partition_v1,
the pivot always ends up in its final sorted place.
if it ended up in the i^{th} position, it would be the i^{th} smallest number.
Keep partitioning until $i = k$

Algorithm

Given $A[1, .., n]$, Returns the k^{th} smallest element.

```
kth smallest (A, n, k):
    dp = partition (A, 1, n)
    left = 1, right = n.
    while (dp != k):
        if (k < dp): right = dp - 1;
        if (k > dp): left = dp + 1;
        dp = parti
```

Kth Smallest Question***

Sunday, December 9, 2018 2:51 PM

COMP2000_1_17

(c) A function `partition` has been written for you. When given an integer array `A` and two subscripts `m` and `n`, it rearranges the elements `A[m]` to `A[n]` and returns a subscript `d` such that all elements to the left of `d` are less than `A[d]` and all elements to the right of `d` are greater than `A[d]`.

Using `partition`, write an *efficient* function which, given an integer array `B`, and two integers *n* and *k*, returns the *k*th smallest integer from `B[1]` to `B[n]`. [4]

Quick Sort Recursive Algorithm

Sunday, December 9, 2018 3:10 PM

```
21  void quickSort_partition_v1(int A[], int left, int right){
22      if(left >= right) return;                          // BASE CASE: 0 or 1 element to sort
23      int divisionPoint = partition_v1(A, left, right);  // ELSE: partition and find division point
24      quickSort_partition_v1(A, left, divisionPoint-1);  // Recur to the left of division point
25      quickSort_partition_v1(A, divisionPoint+1, right); // Recur to the right of division point
26  }
27
28
29  void quickSort_partition_v2(int A[], int left, int right){
30      if(left >= right) return;                          // BASE CASE: 0 or 1 element to sort
31      int divisionPoint = partition_v2(A, left, right);  // ELSE: partition and find division point
32      quickSort_partition_v2(A, left, divisionPoint-1);  // Recur to the left of division point
33      quickSort_partition_v2(A, divisionPoint, right);   // Recur to the right of division point
34  }
```

Quick sort iterative algorithm

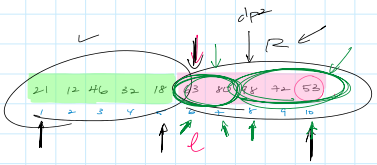
Thursday, December 13, 2018 4:03 AM

USE A STACK (keeps track of unsorted (still to be processed) sublists)

Stack the (sublists) pieces of the list that is still unsorted

Add the shorter sublist to the stack. first

StackNode
- int left — start of unsorted sublist
- int right — end of unsorted sublist.



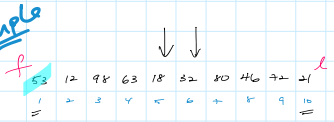
ALGORITHM

```
quick (A, left, right):
    push entire unsorted list onto stack.   (A[left] to A[right])
    while (stack is Not Empty)
        sublist = pop(Stack).
        if sublist is more than 1 element        (stack element.left < stack element.right)
            divisionPoint = partition2 (A, sublist.left, sublist.right).   ← the partitioning will rearrange the elements in an Array.
            if left sublist is smaller
                push left sublist then right
            if right sublist is smaller.
                push right then left sublist.
```

LEFT — (left → dp−1)
RIGHT — (dp → right)

ADD MORE DIAGRAMS.

Example



PROCESSING

Merge Sort (OMITTED)

Sunday, December 9, 2018 2:51 PM

```
void mergeSort(int A[], int lo, int hi) {
  void merge(int[], int, int, int);
  if (lo < hi) { //list contains at least 2 elements
    int mid = (lo + hi) / 2; //get the mid-point subscript
    mergeSort(A, lo, mid); //sort first half
    mergeSort(A, mid + 1, hi); //sort second half
    merge(A, lo, mid, hi); //merge sorted halves
  }
} //end mergeSort
```

where **merge(A, lo, mid, hi)** merges the sorted pieces **A[lo..mid]** and **A[mid+1..hi]** into one sorted piece **A[lo..hi]**. Here is **merge**:

```
void merge(int A[], int lo, int mid, int hi) {
//A[lo..mid] and A[mid+1..hi] are sorted;
//merge the pieces so that A[lo..hi] are sorted
  static int T[MaxNum];
  int i = lo;
  int j = mid + 1;
  int k = lo;
  while (i <= mid || j <= hi) {
    if (i > mid) T[k++] = A[j++];        //A[lo..mid] completely processed
    else if (j > hi) T[k++] = A[i++];    // A[mid+1..hi] completely processed
```

199


```
    else if (A[i] < A[j]) T[k++] = A[i++]; //neither part completed
    else T[k++] = A[j++];
  }
  for (j = lo; j <= hi; j++) A[j] = T[j]; //copy merged elements from T to A
} //end merge
```

Shell Sort

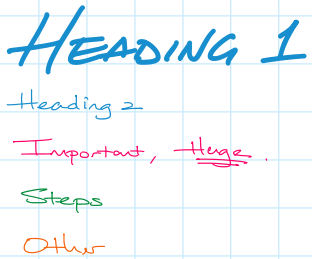
Sunday, December 9, 2018 3:09 PM

Saturday, October 13, 2018 2:28 PM

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

Planning

Saturday, October 13, 2018 2:27 PM



- [x] To go through
- [x] Courseworks
- [x] AH emailed material

- [ ] Finals Teardown
- [x] comp2611_1_18
- [x] COMP2000_1_17
- [x] comp2000_3_14
- [x] COMP2000_2_17
- [x] COMP2000_2_16
- [x] COMP2000_1_16

Alternate concept
Just leave space In the notes to work things out

Day	Original Plan	Reality	Altered Plan
1	Hashing (2) and Matrices (2)	Nearly Perfect According to plan (Medium to Fast Paced)	
2	Graphs, BSTs	Did not start binary trees	
3	BSTs	Did not complete non recursive methods, did not do heaps	
4	Sorting	Completed everything I needed to…Merge Sort omitted	Heaps

Conclusion: SUCCESS

Move start time 1/2 hr back