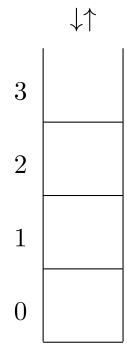
Stacks and Queues

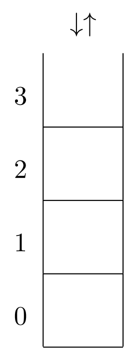
1. Assume a stack is implemented with a fixed size **array** as shown below. Also assume that the `top` pointer is initialized to `-1`. Show the state of the stack after the following operations. Be sure to indicate the final location of the `top` pointer.

   a) **push(3), push(2), push(1)**



   Continuing from where you left off above, do the remaining operations in the stack below (the **BOLD** operations are the remaining operations). Don't forget to label the location of the `top` pointer.

   b) push(3), push(2), push(1), **pop(), push(4), pop(), pop(), push(5)**



2. The stack described in question #1 initializes the `top` pointer to `-1`. Write pseudocode for an `isEmpty` method that utilizes the `top` pointer to determine if the stack is empty. Your method should return `true` if the stack is empty, `false` otherwise.

3. Assume a queue is implemented with a fixed size **array** as shown below. Also assume that both the `head` and `tail` pointers are initialized to 0. The first enqueue(3) operation writes the value 3 to index 0 of the backing array. Provide the final configuration of the backing array after the sequence of operations has completed. Also, indicate the position of both the `head` and `tail` pointers.

   a) **enqueue(3), enqueue(2), enqueue(1)**

0	1	2	3

   Continuing from where you left off above, do the remaining operations in the queue below (the **BOLD** operations are the remaining operations). Don't forget to label the location of the `head` and `tail` pointers.

   b) enqueue(3), enqueue(2), enqueue(1), **dequeue(), enqueue(4), dequeue(), enqueue(5), dequeue()**

0	1	2	3

4. The queue described in question #3 initializes both the `head` and `tail` pointers to 0. Write pseudocode for an `isEmpty` method that utilizes the `head` and `tail` pointers to determine if the queue is empty. Your method should return `true` if the queue is empty, `false` otherwise.

5. List one advantage and one disadvantage of using a *preallocated*, i.e. fixed size, backing array to implement stacks and queues.

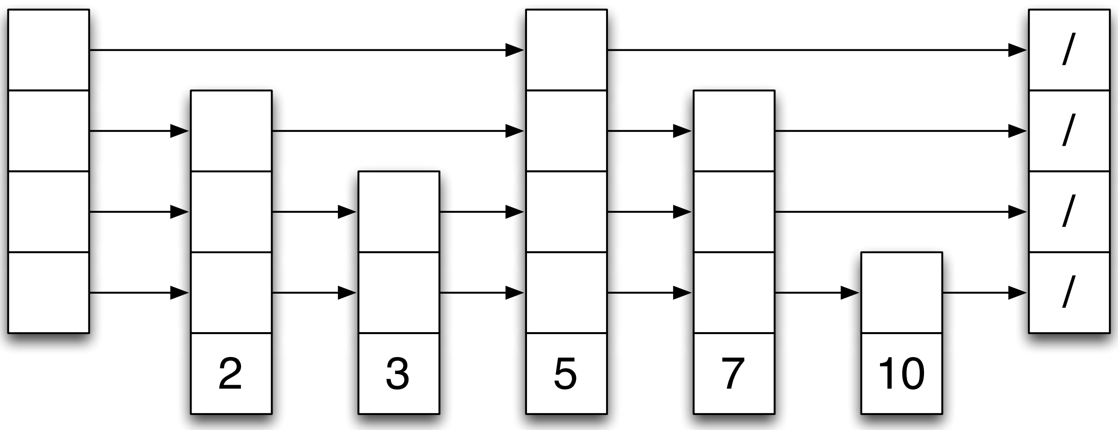
Skip List - Find

For the following skip list, mark the pointers (i.e. the ARROWS) that are *touched* for the operation.

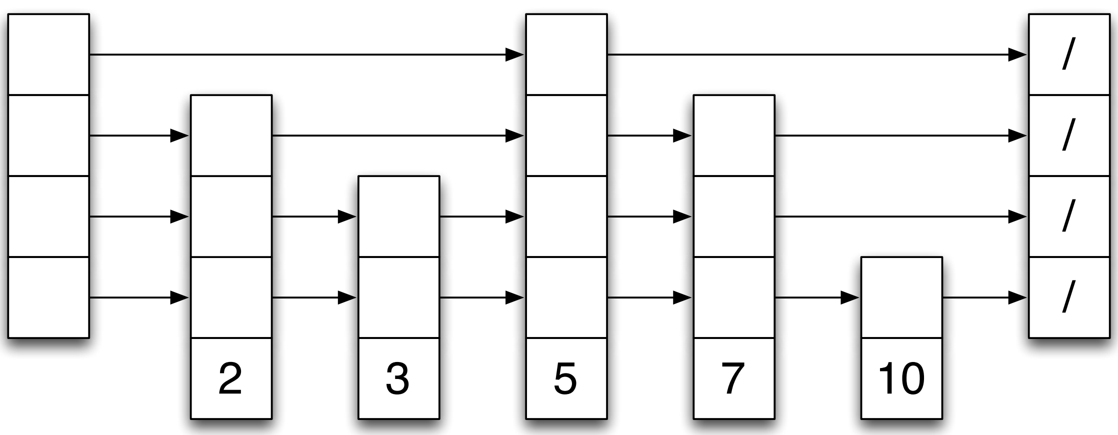
- If a pointer is checked but not followed, mark the pointer by writing an **'C'** directly on the arrow.
- If a pointer is checked and followed, mark the pointer by writing an **'F'** directly on the arrow.

Additionally, draw a vertical arrow to indicate where the level is decremented.

1. Find(3)



2. Find(8)



3. What is the primary advantage of find for a skip list compared to a standard linked list.

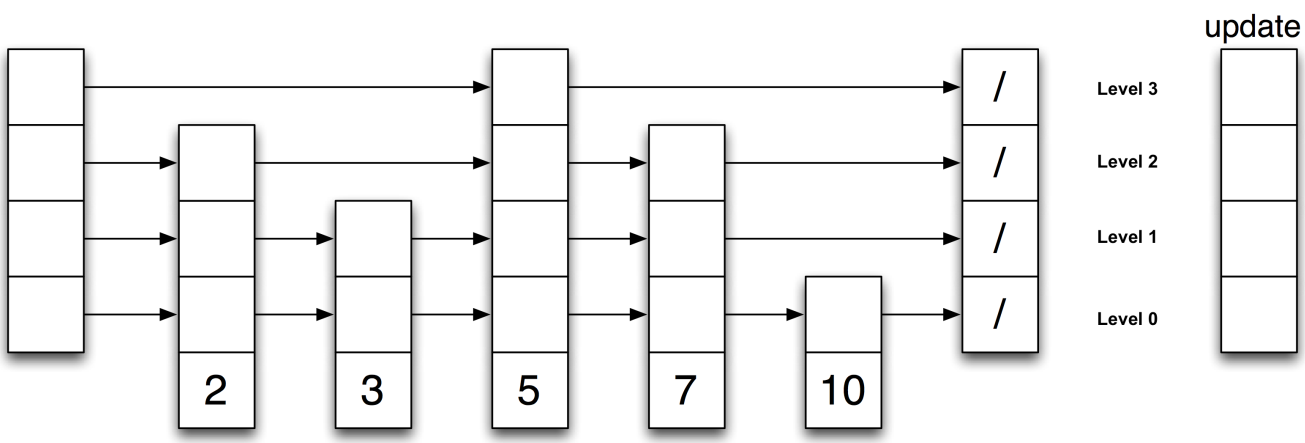
Skip List - Insert/Remove

For the following skip list, mark the pointers (i.e. the ARROWS) that are *touched* for the operation.

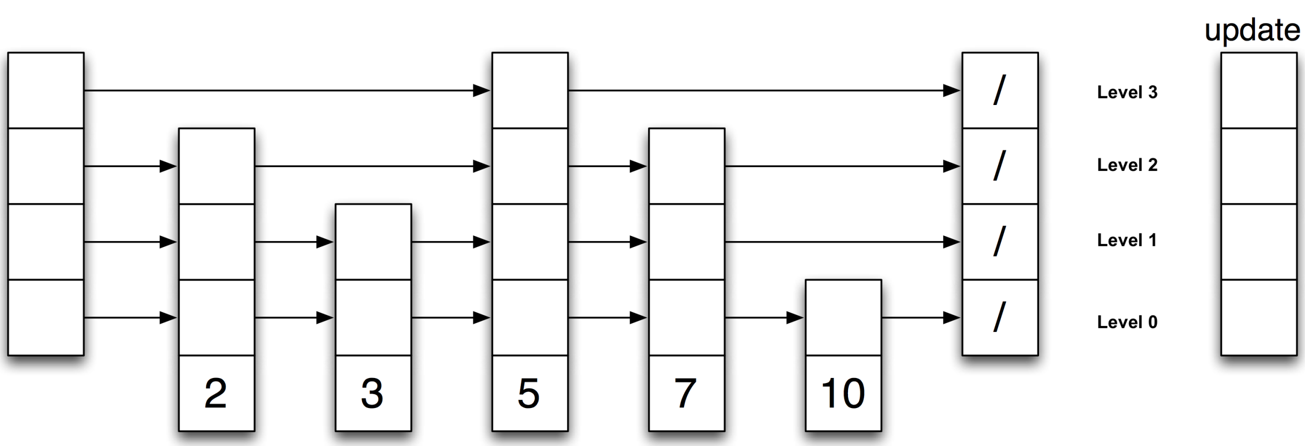
- If a pointer is checked but not followed, mark the pointer by writing an **'C'** directly on the arrow.
- If a pointer is checked and followed, mark the pointer by writing an **'F'** directly on the arrow.

Additionally, draw a vertical arrow to indicate where the level is decremented.

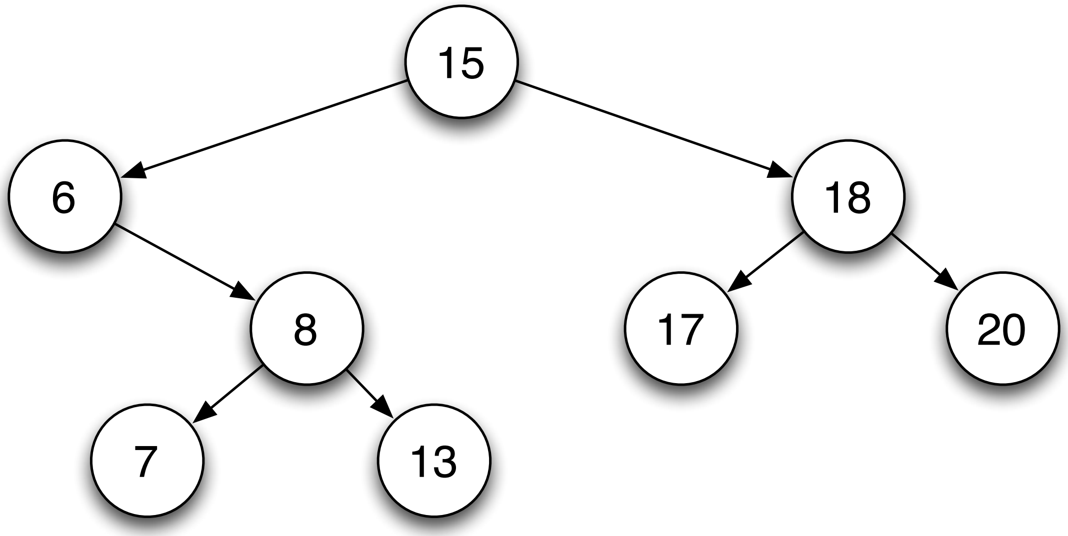
1. Insert(4) at level 3



2. Remove(7)

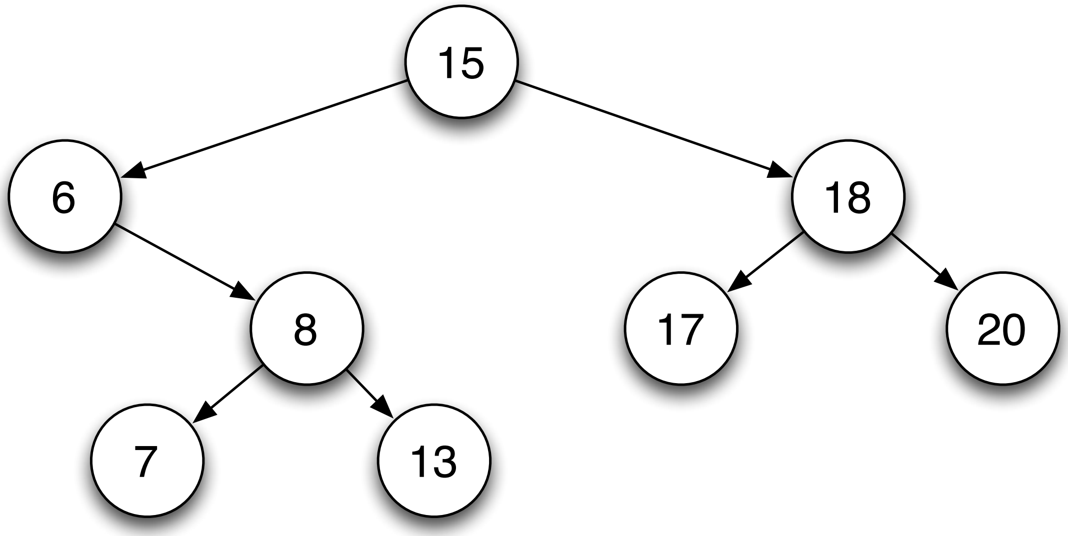


BST - Remove



1. Draw the BST that results from calling **`remove(13)`** on the tree above.
2. Draw the BST that results from calling **`remove(6)`** on the <u>original</u> tree above.
3. Draw the BST that results from calling **`remove(15)`** on the <u>original</u> tree above.

AVL Trees - Insert



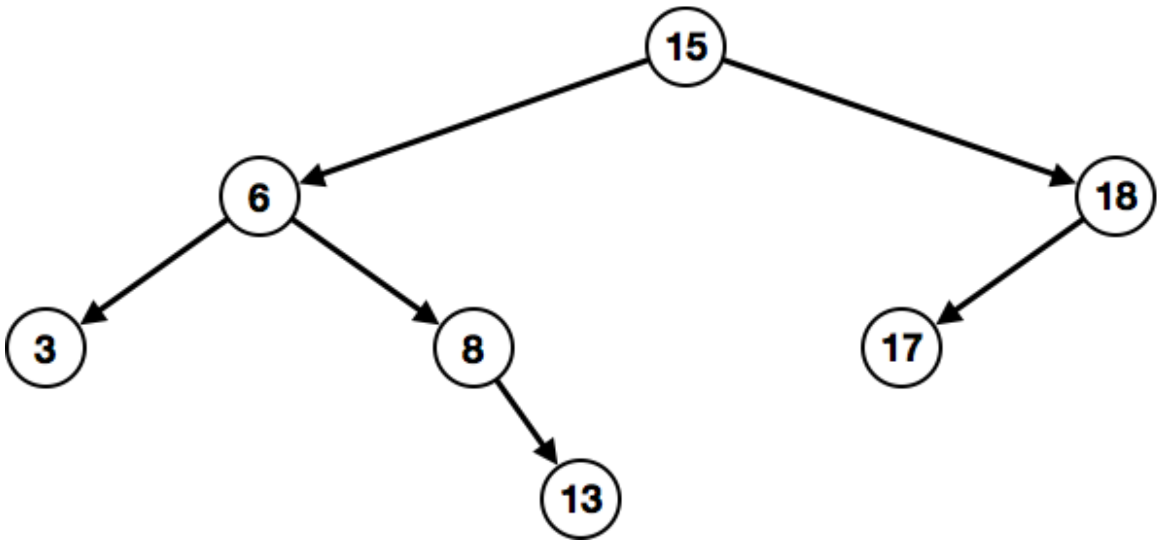
1. Label the height of each node. Is this a valid AVL tree?

2. Add a left child of node 6 with value 3. Now is this a valid AVL tree?

3. Perform the following operations on the AVL tree from part 2:

    - Insert(19)
    - Insert(10)

AVL Trees – Remove

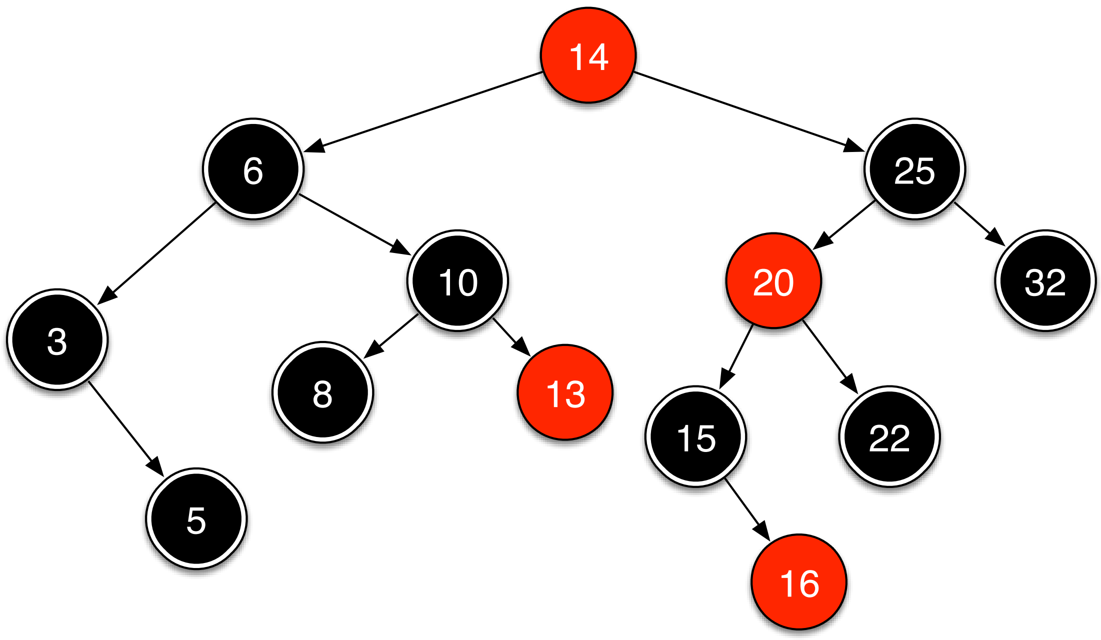


1. Label the height of each node. Is this a valid AVL tree?

2. Perform the following operations on the AVL tree above:
   remove(3)

3. Perform the following operations on the **<u>ORIGINAL</u>** AVL tree above (use the successor node, not the predecessor node):
   remove(15)

Red-Black Trees – Insert #1



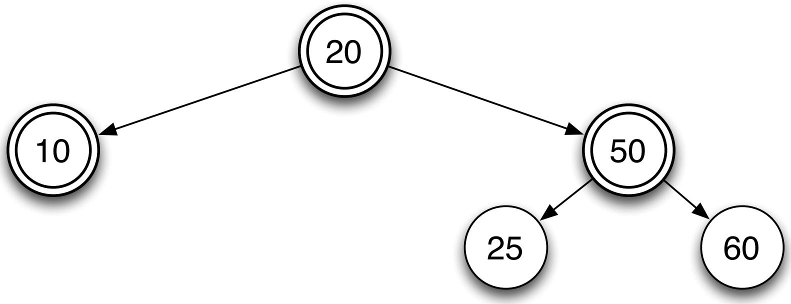
1. Is this a valid Red-Black tree? Why or why not?

2. Adjust node colors to make the tree a valid Red-Black Tree. (Hint: Consider nodes 14, 8, and 5)

3. Perform the following operations on the Red-Black tree from part 2 – indicate black nodes with double lines and red nodes with single lines:

   - Insert(23)
   - Insert(12)

Red-Black Trees – Insert #2

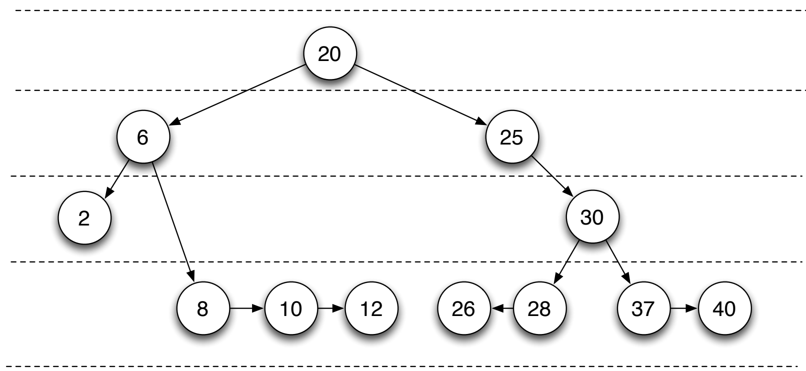


Perform the following operations on the Red-Black tree (where double-line indicates black nodes):

- Insert(40)
- Insert(30)
- Insert(35)

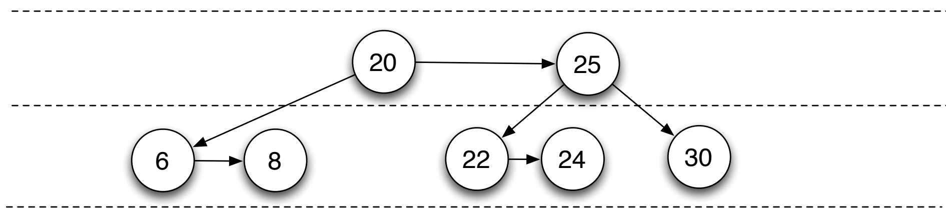
AA Trees - Validation / Insertion

For the following AA tree



List which nodes are invalid and briefly specify the reason.

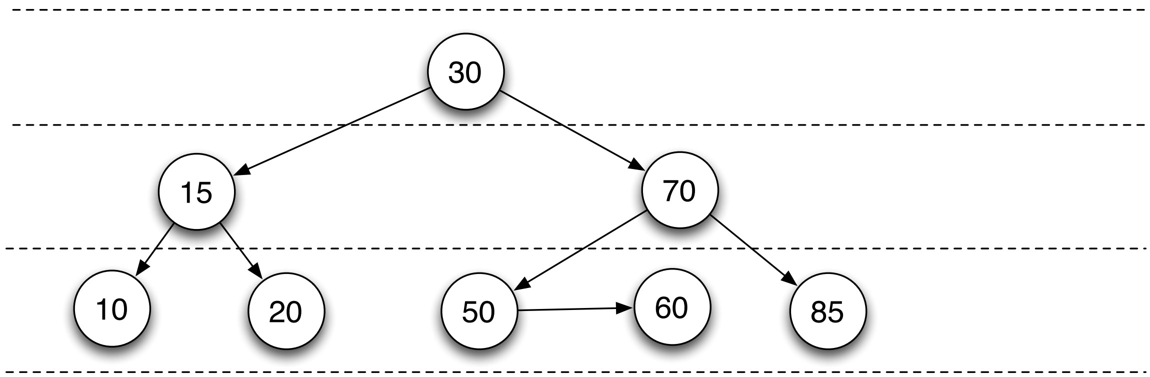
For the following AA tree



a. Indicate in the figure above where a node with value of 13 would *initially* be inserted, i.e. before any rebalancing operations are performed.

b. Show the rebalanced AA tree after insertion of 13. List the sequence of intermediate steps performed along with which node the step operates on - even if they perform no modifications - but only redraw the tree when adjustments occur (you may use triangles to represent unaltered subtrees).

AA Trees – Insert / Delete



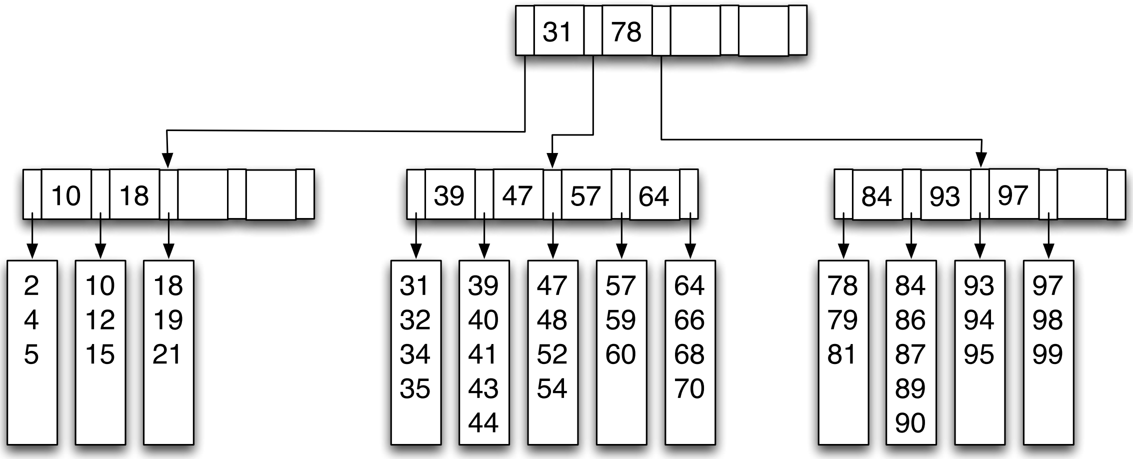
The above AA tree was created using the following sequence of inserts (just in case you want to recreate it):

10, 85, 15, 70, 20, 60, 30, 50

Perform the following operations on the AA tree:

- Insert(13)
- Insert(65)
- Delete(50)
- Delete(30)

B-Trees – Insert/Delete

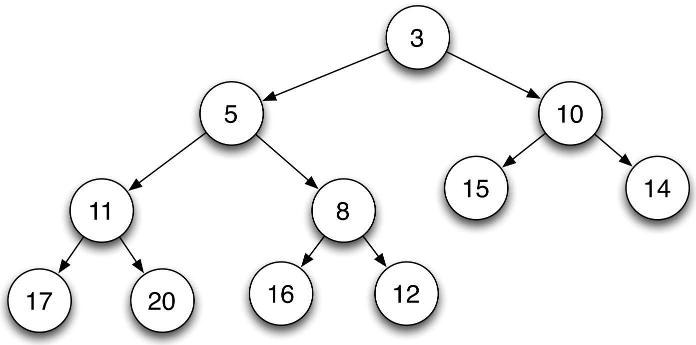


Create the above B-tree by a sequence of inserts (JUST KIDDING!)

Perform the following operations on the B-tree:

- Insert(82)
- Insert(91)
- Delete(60)
- Delete(15)

Binary Heaps - Insert / removeMin



1. Show the final heap as a tree after the operations **insert(19)**, **insert(4)**.

2. Show the final heap as a tree after the operations: **removeMin()**, **removeMin()**, **removeMin()**.

Hash Tables - Probing & Chaining

Insert the following values *in order* into each hash table using the specified method for resolving collisions. For all methods, use the hash function $h(k) = k \mod m$ (note $m = 10$ so consider whether or not this is a good hashing function).

$$15, 55, 91, 27, 89, 46, 77, 35$$

Linear Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

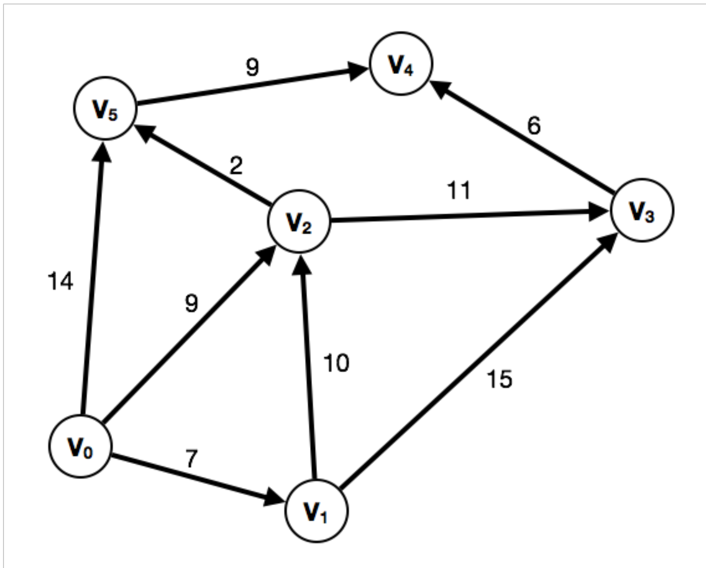
Chaining

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

What is the *load factor* of this hash table?

Graphs - Adjacency Matrix & Adjacency List

For the graph below, fill in the adjacency matrix.



	V_0	V_1	V_2	V_3	V_4	V_5
V_0						
V_1						
V_2						
V_3						
V_4						
V_5						

In the space below, draw an adjacency list that represents the graph above.