Name:____________________________________ UW NetID:____________

CSE332 Summer 2012 Midterm Exam, July 18, 2012 SOLUTIONS!

Please do not turn the page until the bell rings.

Rules:

– The exam is closed-book and limited-note. You are permitted a single, handwritten 3x5 index card of notes. You must turn in this card with your exam.

– Calculators are also permitted but not necessary.

– Please stop promptly at 12:20.

– You can rip apart the pages, but please staple them back together before you leave.

– Blank paper for extra room are available upon request.

– This exam contains 10 questions (many with multiple parts). There are **110 points** total, but the exam is worth **100 points**, meaning that you may earn some extra points.

Advice:

– The questions are not necessarily in order of difficulty. Read through the entire exam first and then **skip around** as you see fit. Make sure you get to all the problems.

– Read questions carefully. Understand a question before you start writing.

– Write down thoughts and intermediate steps to earn partial credit. **Circle your final answer**.

– If you have questions, ask.

– Relax. You are here to learn.

EXAM SCORE	/ 100

1. (14 pts) Algorithmic Analysis

The following questions all refer to big-O, big-Θ, and big-Ω notation and algorithmic analysis. For parts (a) and (b), you will need to provide [dis]proof of the bounds. For parts (c)-(l), you only need to write down the asked-for bound in the underlined area. Each bound should be as tight and simple as possible

(a) Prove (give a c and n_0) or disprove that $5n^2 - 20n + 3$ is $O(n^3)$

True. For $c = 1$ and $n_0 > 5$, the following is true for $n > n_0$: $5n^2 - 20n + 3 < 5n^2 < n^3$ as $n > 5$

(b) Prove (give a c and n_0) or disprove that 2^{3n} is $O(2^n)$

False. $2^{3n} = 2^n 2^n 2^n$ which will clearly always be bigger than 2^n for any $n > 0$.

(c) What is the tightest bound you can give for $f(n) = 2n^3 - 3n \log n$? (c) $\Theta(n^3)$

(d) What is the tightest bound that you can give for the summation $\sum_{i=0}^{n} i^k$? (d) $\Theta(n^{k+1})$

(e) What is the worst-case performance for a building a binary min-heap of n items **without** using a call to BuildHeap? (e) $O(n \log n)$

(f) What is the big-O bound for performing find in a perfect BST? (f) $O(\log n)$

(g) What is the tightest bound you can give for deletion in an unsorted array (ignoring the cost of finding the element to delete)? (g) $O(1)$

(h) What is the average time for insertion in a B tree with $M = 32$ and $L = 8$? (h) $O(\log_{32} n)$

(i) What is the worst-case time for insertion into a 5-heap? (i) $O(\log_5 n)$

(j) What is the worst-case time for a **single** call to enqueue in an array-based queue? (j) $O(n)$

(k) What is the **amortized** time for enqueue in an array-based queue? (k) $O(1)$

(l) What is the run-time for the following code? (l) $O(n \log n)$

```
for(i=1; i<n; i++) {
   x = n;
   while (x > 0) {
      sum++;
      x = x / 2;
   }
}
```

2. (8 pts) Recurrence Relations

For this problem you will be working with the following recurrence relation:

$$T(n) = \begin{cases} 4 & n = 1 \\ T(n) = 4 + 4T(\lfloor \frac{n}{2} \rfloor) & n > 1 \end{cases}$$

(a) Show the values for $T(n)$ for all integers in the range $1 \leq n \leq 8$

(b) Provide the closed form for $T(n)$. Show your work; do not just show the final equation. You may assume n is sufficiently large such that the floor function does not lead to rounding issues.

Solution:

(a) Values of $T(n)$ for $1 \leq n \leq 8$

$T(1) = 4$
$T(2) = 4 + 4 \cdot T(1) = 4 + 4 \cdot 4 = 20$
$T(3) = 4 + 4 \cdot T(1) = 4 + 4 \cdot 4 = 20$
$T(4) = 4 + 4 \cdot T(2) = 4 + 4 \cdot 20 = 84$
$T(5) = 4 + 4 \cdot T(2) = 4 + 4 \cdot 20 = 84$
$T(6) = 4 + 4 \cdot T(3) = 4 + 4 \cdot 20 = 84$
$T(7) = 4 + 4 \cdot T(3) = 4 + 4 \cdot 20 = 84$
$T(8) = 4 + 4 \cdot T(4) = 4 + 4 \cdot 84 = 340$

(b) We will solve this through the common repeated substitution method:

$$T(\frac{n}{2}) = 4 + 4T(\lfloor \frac{n}{4} \rfloor)$$

implies

$$\begin{aligned} T(n) &= 4 + 4 \cdot (4 + 4T(\lfloor \frac{n}{4} \rfloor)) \\ &= 4 + 4^2 + 4^2 \cdot T(\lfloor \frac{n}{4} \rfloor) \end{aligned}$$

Repeating this substitution gives:

$$T(n) = 4 + 4^2 + 4^3 + \cdots + 4^k + 4^k \cdot T(\frac{n}{2^k})$$

Letting $k = \lfloor \log_2 n \rfloor$, then

$$\begin{aligned} T(n) &= 4 + 4^2 + 4^3 + \cdots + 4^k + 4^k \cdot T(1) \\ &= 4 + 4^2 + 4^3 + \cdots + 4^k + 4^k \cdot 4 \\ &= 4 \cdot (1 + 4 + 4^2 + \cdots + 4^k) \\ &= 4 \cdot \frac{4^{k+1} - 1}{3} \\ &= \frac{4}{3} \cdot (4 \cdot 4^k - 1) \\ &= \frac{4}{3} \cdot (4 \cdot 4^{\lfloor \log_2 n \rfloor} - 1) \\ &= \frac{16 \cdot 4^{\lfloor \log_2 n \rfloor} - 4}{3} \end{aligned}$$

3. (10 pts) Move-to-Front Structure

The splay tree's idea of moving recently touched elements to positions that are more easily fetched can be applied to other data structure. In particular, consider a linear data structure (i.e., not a tree) called *MoveToFront*. MoveToFront provides the following functionality:

- *insert(x)*: The item *x* is inserted at the front of MoveToFront.
- *find(x)*: Returns *true* or *false* depending on if *x* is currently in MoveToFront. After a call to find, *x* is move to the front so that it will be accessed first on the next find.
- *remove(x)*: Removes element *x* from the structure. Other than this removal, the ordering of the other elements remains unchanged.

For this question, you need to do the following:

(a) Provide a description of how you would implement MoveToFront. Provide pseudocode for both *insert* and *find*. You do NOT need to implement *remove*. Your pseudocode should be clear and unambiguous but does not need to be exact Java code.

(b) Provide a brief justification for your design decisions.

(c) Describe the best AND worst case performance for both insert AND find.

You may make any assumptions you feel are necessary as long as you state them in your writeup.

Solution 1: Singly-Linked List

(a) We will use a singly-linked list for implementing MoveToFront.

```
void insert(x) {
    Node n = new Node(x)
    if(this.isEmpty())
        this.head = n
    else
        n.next = this.head
        this.head = n
}
```

```
boolean find(x) {
    if(n.key == x)
        return true
    Node n = this.head, prev = null
    while(n != null  &&  n.key != x)
        prev = n
        n = n.next
    if(n == null)
        return false
    prev.next = n.next
    n.next = head
    head = n
    return true
}
```

(b) Using a linked list was chosen because inserting at the head automatically shifts everything to the left. Moreover, swapping a node to the front just requires rearranging a few links. Thus, the second item in a list will always be the one previously inserted or found before the last.

(c) Insert is Θ(1) for best and worst time as it is just an insert. Find is at best O(1) if the item is at or near the front of the list but is at worst O(*n*) if the find has to iterate through the entire list.

Solution 2: Array

(a) We will use an unsorted array for implementing MoveToFront. To do insert, we will move the current front of the array to the end and then insert the new item there. For find, we will just swap the positions.

```
void insert(x) {
    if(this.size == arr.length)
        ResizeArray()
    arr[this.size] = arr[0]
    arr[0] = x
    this.size += 1
}
```

```
boolean find(x) {
    for(i=0; $i < arr.length$; i++)
        if(arr[i] == x)
            swap arr[0] and arr[i]
            return true
    return false
}
```

(b) The array implementation was chosen as it is conservative in space due to no need for pointers. By swapping the first position and the end makes insert constant time. This does break recent locality by moving recent things to the end. It's not exactly like a splay tree. However, if we shifted the array insert would become O(n).

(c) Insert is O(1) for best and worst time when amortized as it is just a simple swap. Resizing will push the worst time for insert to O(n) rarely. Find is at best O(1) if the item is at or near the front of the array but is at worst O(n) if the find has to iterate over the unsorted array.

Solution 3: Circular Array

(a) We will use an unsorted circular array for implementing MoveToFront.

```
void insert(x) {
    if(this.size == $arr.length - 1$)
        ResizeArray()
    this.front = (this.front - 1) % arr.length
    arr[this.front] = x
    this.size += 1
}
```

```
boolean find(x) {
    for(i=front; i != this.back; $i = (i+1)\%arr.length$)
        if(arr[i] == x)
            this.front = (this.front - 1) % arr.length
            move arr[i] to arr[front]
            shift contents of array to fill hole at arr[i]
            return true
    return false
}
```

(b) The array implementation was chosen as it is conservative in space. By making the array circular, we can insert at the front and maintain recency without having to shift everything. Since find is at worst a linear time operation, having to shift the content is not too bad (especially since we can always shift to *front* or *back* and thereby only have to shift at most half of the array. The circular code is more complex, however. In particular, for the find code to be simpler, we need to make sure that the array always as at least one empty index. Otherwise, the $front - 1$ step in find will overwrite a value. Thus, we resize when the current array has $arr.length - 1$ items.

(c) Insert is O(1) in general unless there is a resize. In those rare circumstances, the performance becomes O(n). Find is at best O(1) if the item is at or near the front of the array even if we have to shift a few items. In general, though, it will be O(n) because it has to both iterate over the entire array and may have to shift $\frac{n}{2}$ of the array elements.

4. (10 pts) Min-Heaps

For this question, you will be working with binary min-heaps.

(a) What are the two properties required by a heap?

(b) Draw the heap produced by *BuildHeap()* given the following sequence of keys:

15, 29, 3, 8, 11, 35, 25, 57, 7, 1, 44, 16

(c) Using your answer from (b), draw the heap after a call to *deleteMin()*

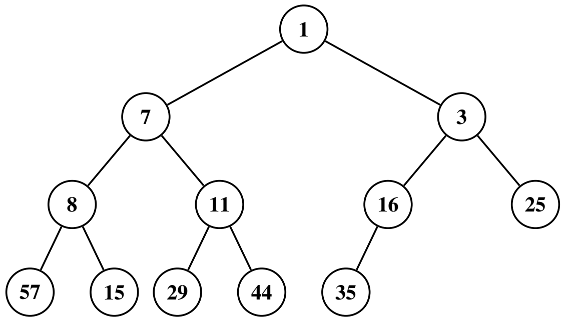
(d) Using your answer from (c), draw the heap after a call to *insert(6)*

(e) Using your answer from (d), show the array representation for the heap.

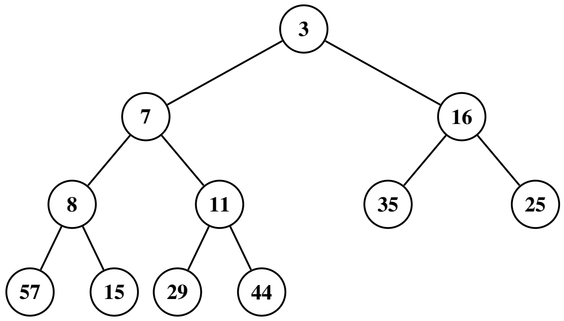
Solution

(a) The heap must be a complete tree and it must meet the property that for every node in the heap, its key is less or equal to the key's of its children (for a min-heap that is).

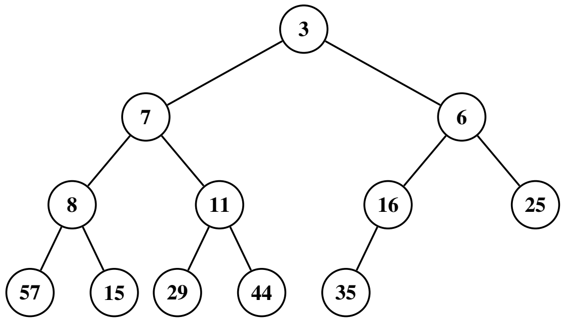
(b)



(c)



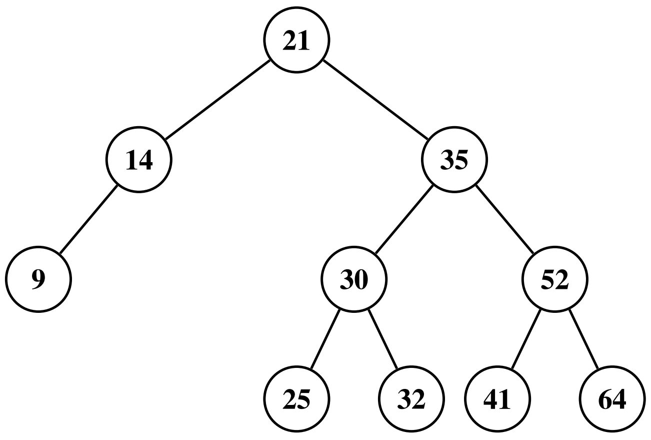
(d)



(e)

0	1	2	3	4	5	6	7	8	9	10	11	12
	3	7	6	8	11	16	25	57	15	29	44	35

5. (14 pts) AVL Trees



(a) What is the balance for the following nodes in the AVL tree: **14**, **21**, **35**, and **41**?

(b) Show the AVL tree after inserting **3** into the provided tree.

(c) Show the AVL tree after inserting **36** into your answer from (b).

(d) Show the AVL tree after inserting **70** into your answer from (c).

(e) Show the AVL tree after deleting **52** from your answer from (d). Your deletion should use the immediate predecessor for replacement.

(f) Show the AVL tree after deleting **36** from your answer from (e). Your deletion should use the immediate predecessor for replacement.

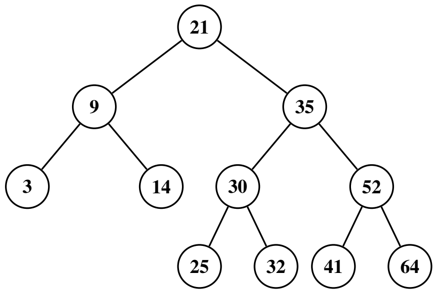
Solution

(a) Balances are as follows using the formula: height(*left child*) − height(*right child*)

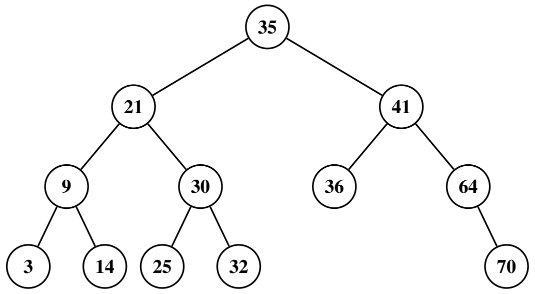
balance(14) = 0 − (-1) = 1 balance(35) = 2 − 2 = 0

balance(21) = 1 − 2 = -1 balance(41) = (-1) − (-1) = 0

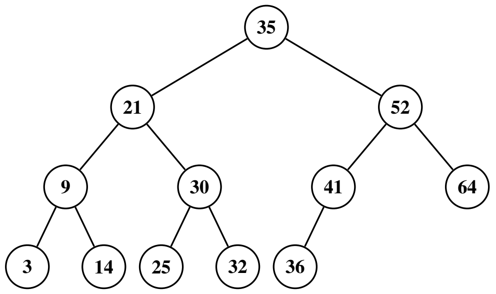
(b)



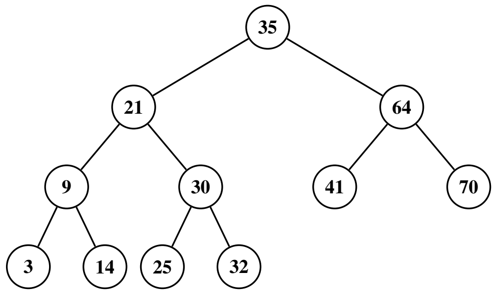
(e)



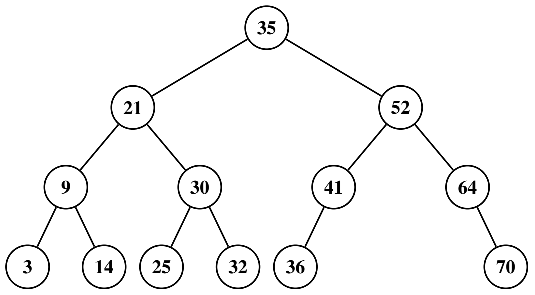
(c)



(f)



(d)



6. (10 pts) AVL Confirmation

Give pseudocode for a O(n) algorithm that veries that an AVL tree is correctly maintained. Assume every node has fields key, data, height, left, and right and that keys can be compared with $<$, $==$, and $>$. The algorithm should verify all of the following:

– The tree is a binary search tree

– The height information of every node is correct

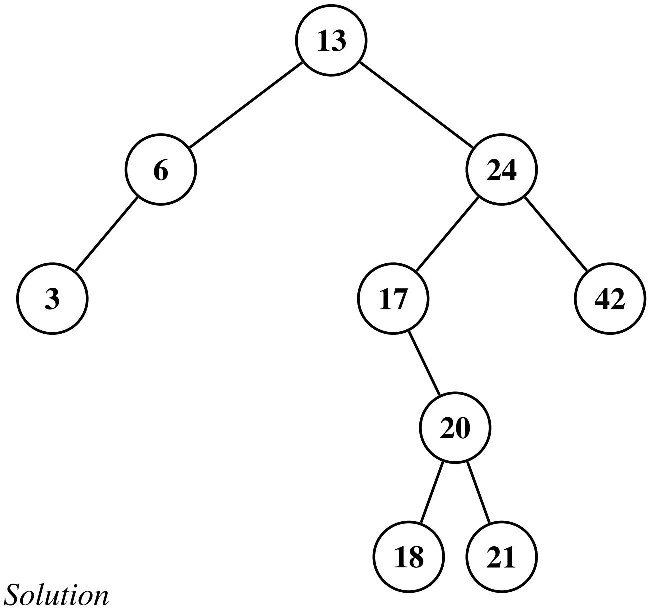
– Every node is balanced

Solution

We will write a recursive function to do this.

```
boolean AVLConfirm(Node n) {
    if(n == null)
        return true
    if(n.left == null  &&  n.right == null        // n is a leaf
        return (n.height == 0)
    // Check left subchild recursively, check BST property, grab its height
    int leftHeight = -1
    if(n.left != null)
        if(!AVLConfirm(n.left))
            return false
        if(n.key < n.left.key)
            return false
        leftHeight = n.left.height
    // Check right subchild recursively, check BST property, grab its height
    int rightHeight = -1
    if(n.right != null)
        if(!AVLConfirm(n.right))
            return false
        if(n.key > n.right.key)
            return false
        rightHeight = n.right.height
    // check height and balance
    if(max(leftHeight, rightHeight) + 1 != n.height)
            return false
    return (abs(leftHeight − rightHeight) ≤ 1)
}
```

7. (6 pts) Splay Trees



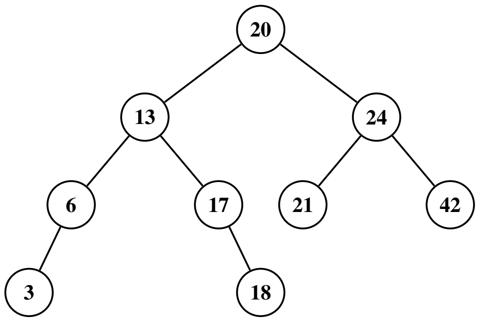
(a) Show the splay tree after a call to **find(20)**.

(b) Using your answer from (a), show the splay tree after a call to insert **insert(8)**. You should use the find/split approach for inserting.

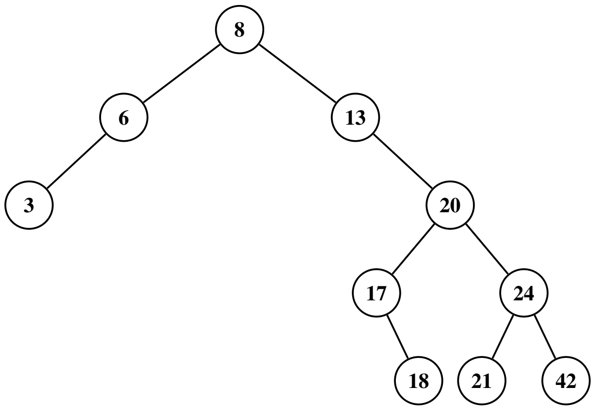
(c) Using your answer (c), show the splay tree after a call to **delete(17)**. Your should use the find/join approach for deletion and use the max from the left subtree for replacement.

Solution

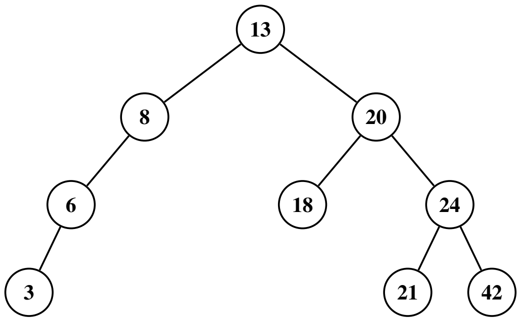
(a)



(b)



(c)



8. (10 pts) Splay Tree Range Restrict

A common enhancement to the Dictionary ADT is **RangeRestrict(x,y)**. This method constrains the dictionary to only contain keys between *x* and *y* inclusive. In other words, it removes all nodes whose keys are $< x$ or $> y$.

For this question, you are to provide pseudocode for how you would implement **RangeRestrict(x,y)** in a splay tree. You may assume that the splay tree class is a subclass of a general binary search tree class. The BST class has implementations of *insert*, *find*, *remove*, *findMax*, *findMin*, *findPredecessorOf*, and *findSuccessorOf*. The Splay class implements its own versions of *insert*, *find*, and *remove*. Thus, you may use any of the standard splay tree operations or their BST equivalents (i.e., be sure to state if you are using BSTfind or SplayFind). Your solution should be efficient and correct.

It is up to you to decide how to handle situations where $y \geq x$.

If you make any further assumptions, be sure to state them clearly.

Solution

The primary idea here is that if you make a call to *find*(x), the splay tree moves x to the root. At this point, you can simply cut off the left subtree and thereby complete half of the range restrict. The same idea applies if you do a call to *find*(y) and then cut off the right subtreee.

The *catch* however is that neither x nor y may be in the tree so that means you have account for those situations. On failure, *find* in a splay tree will bring the last node search to the root. However, the root may be less than or greater than the value you searched for. What we need to do is to make sure that we cut off the correct branches at the correct point.

Here are two approaches

Approach 1: Predecessor/Successor

```
void RangeRestrict(x, y) {
   if y > x
      throw and exception

   Node n = root
   Search down tree from n to find key x
   if(x is not found)        // n is currently the last node searched
      while(n.key < x)
         n = this.findSuccessorOf(n)
   // n now either contains x or has the smallest key in the tree that is gerater than x
   splay n to the root
   set n.left to null

   Search down tree from n/root to find key y
   if(y is not found)        // n is currently the last node searched
      while(n.key > y)
         n = this.findPredecessorOf(n)
   // n now either contains y or has the largest key less than y
   splay n to the root
   set n.right to null
}
```

The above code is missing checks for if *n* is *null* but delivers the basic idea. We could also make the code simpler by doing a check for when $x == y$ which results in either a single node or an empty tree depending on if *x* is in the tree (see next answer below).

Approach 2: Elegance

The above is a lot of work and may require multiple calls to the predecessor and successor methods. An easier approach for handling when either *x* or *y* is not in the tree is to insert them, do the splaying trick, delete the appropriate subtree, and then delete whatever nodes we added in the first place. Presto!

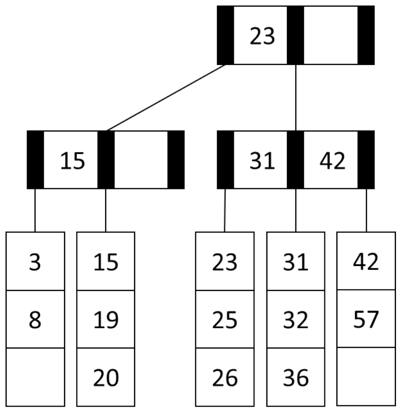
```
void RangeRestrict(x, y) {
  if $y > x$
    throw and exception
  if($x == y$)
    SplayFind(x)
    if(this.root.key != x)
      this.root = null
    else
      set this.root.left to null
      set this.root.right to null
    return
  SplayFind(x)
  if(this.root.key != x)
    SplayInsert(x)
  set this.root.left to null
  if we inserted x earlier
    SplayDelete(x)
  SplayFind(y)
  if(this.root.key != y)
    SplayInsert(y)
  set this.root.right to null
  if we inserted y earlier
    SplayDelete(y)
}
```

9. (10 pts) B Trees

(a) Using the B tree ($M = 3$ and $L = 3$) to the right, show the resulting tree after inserting **43**.

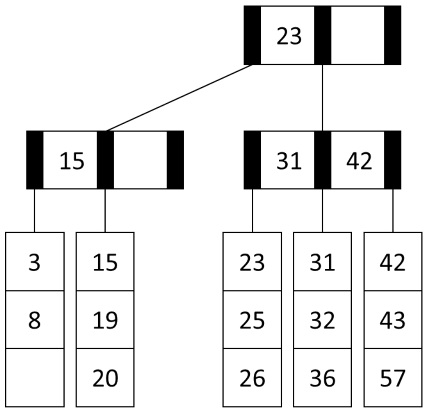
(b) Using your answer from (a), show the resulting tree after inserting **17**.

(c) Using your answer from (b), show the resulting tree after inserting **39**.

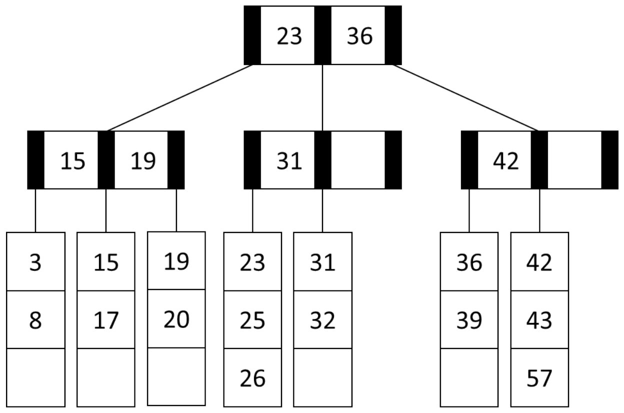


Solution:

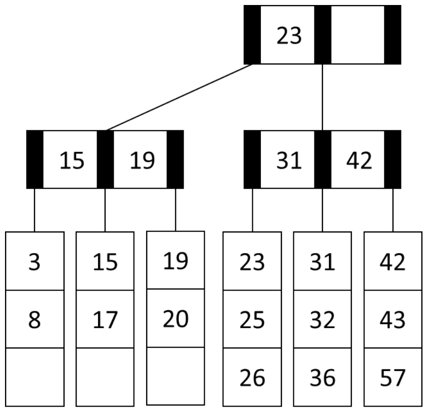
(a)



(c)

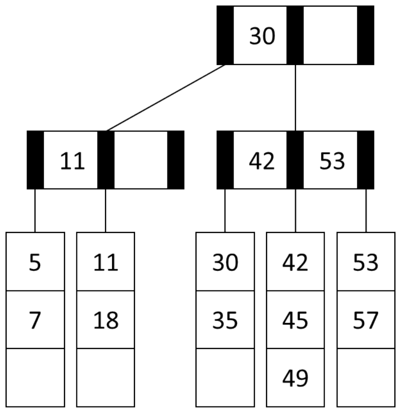


(b)

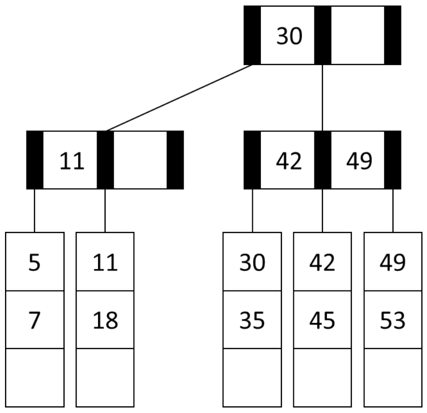


(d) Using the B tree ($M = 3$ and $L = 3$) to the right, show the resulting tree after deleting **57**.

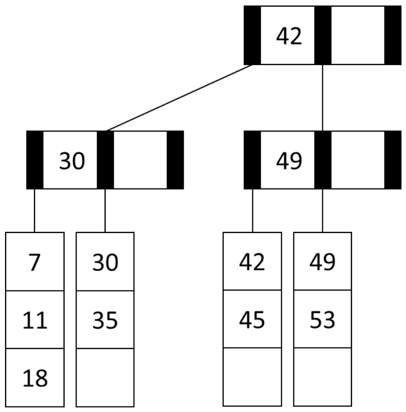
(e) Using your answer from (d), show the resulting tree after deleting **5**.



(d)



(e)



10. (18 pts) Hash Tables

For each of the following versions of hash tables, insert the following elements in this order:

34, 16, 45, 53, 6, 29, 37, 78, and 1

For each table, $TableSize = 11$, and you should use the primary hash function $h(k) = k\%11$. If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

For double hashing, the secondary hash is $g(k) = 1 + (k/4)\%T$, where T is the table size.

For separate chaining, insert at the front of the list.

Linear Probing

0	
1	34
2	45
3	78
4	37
5	16
6	6
7	29
8	1
9	53
10	

Quadratic Probing

0	
1	34
2	45
3	
4	37
5	16
6	6
7	29
8	
9	53
10	78

1 cannot be inserted

Double Hashing

0	
1	34
2	45
3	1
4	37
5	16
6	6
7	29
8	
9	53
10	78

Separate Chaining

0	/				
1		1	78	45	34
2	/				
3	/				
4		37			
5		16			
6		6			
7		29			
8	/				
9		53			
10	/				

What is the load factor of the hash table using *linear probing*?

$\frac{9}{11} = 0.818181...$

What is the load factor of the hash table using *separate chaining*?

$\frac{9}{11} = 0.818181...$