CS 240 Tutorials 6-11

6.1 We can sort the strings by inserting each string into a multiway trie, then performing a preorder traversal of the trie to obtain the words in lexicographical ordering. We perform a preorder traversal by visiting each node. At each node, we visit its children in lexicographical order. When we encounter a leaf, we add its word to our output array. Inserting each word x takes O(|x|) time, so the insertion step takes O(l) time. Traversing the tree visits every node once, so takes O(l) time.

```
preorderTraverse(v: trieNode) {
    if v is null return []
    else if v is a leaf {
        return v
    } else {
        output = []
        // Assume children stored as array of ptrs of length alphabet
        for child in v.children {
            output += preorderTraverse(child)
        }
        return output
    }
}

sortWords(words) {
    trie = Trie()
    for word in words {
        trie.insert(word)
    }
    sortedArray = preorderTraverse(trie)
}
```

6.2 We start interpolation search with l = 0 and r = n-1

If k is in A, we calculate our first m as

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor = 0 + \left\lfloor \frac{aj + b - b}{a(n-1) + b - b} (n - 1 - 0) \right\rfloor = \left\lfloor \frac{aj(n-1)}{a(n-1)} \right\rfloor = \lfloor j \rfloor$$

$$= j$$

Hence if k is in A, we find j on the first iteration, which is O(1) time.

If k is not in A, we have 3 cases:

k < A[0]. We enter the if statement k < A[l] on our first iteration, which returns not found. This is in O(1) time.

k > A[n-1]. We enter the if statement k > A[r] on our first iteration, which returns not found. This is in O(1) time.

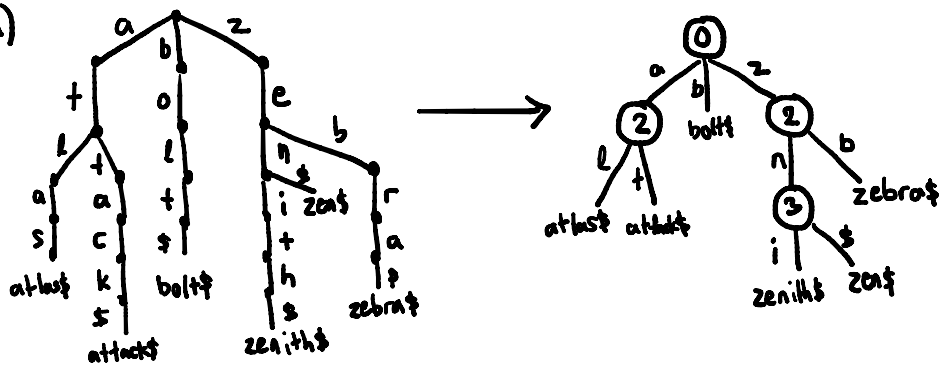
k is between consecutive elements of A. This means we have some index 0 <= j <= n-2 where A[j] < k < A[j+1]. This means aj+b < k < a(j+1) + b, so we have k = aj+b+c, where 0 < c < a. The algorithm calculates the index

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r - l) \right\rfloor = 0 + \left\lfloor \frac{aj + b + c - b}{a(n-1) + b - b} (n - 1 - 0) \right\rfloor = \left\lfloor j + \frac{c}{a} \right\rfloor = j$$

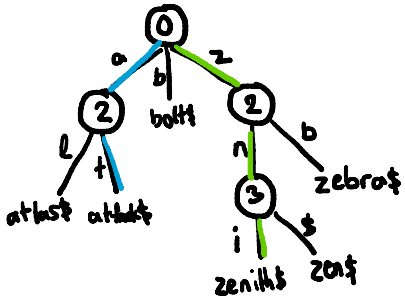
Hence we find the index j where A[j] < k < A[j+1] in the first iteration. Since A[j] < k, we search again in the right subarray, by setting l = m+1 = j+1. In this iteration, we enter the if statement k < A[l] = A[j+1], which returns not found. All operations were in O(1) time, so this is in O(1) time.

In all cases, the algorithm terminates in O(1) time.

7.1 a)



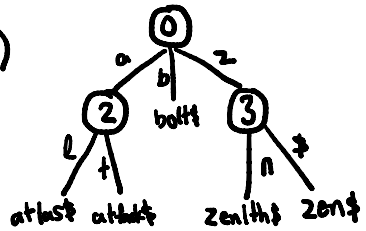
b)



at : not found

zenith : found

c)



7.2 a)

0	14 [deleted]	2
1	13	found
2	7	
3	10	
4	17	
5		
6	20	1

b)

0	14 [deleted]
1	
2	7
3	10
4	13
5	17
6	20

3
2
found
1

c)

0	7	0	
1		1	10
2		2	
3	17	3	
4		4	13
5		5	14 [deleted]
6	20	6	

found

7.3 We can use hashing with separate chaining, where each chain is implemented as an AVL tree. We maintain alpha as between 0.5 and 1.5, and set the minimum table size as 100. These numbers can be chosen arbitrarily.

It takes O(1) time to compute the hash function of a key. In the worst case, all elements are in a single AVL tree corresponding to a single index of the hashtable. The worst-case runtime for search, insert, and delete in an AVL tree is O(log n). Rehashing takes O(nlog(n)) time in the worst-case, but is performed at most every n operations, so takes O(log n) amortized time. Hence the worst-case runtime for search, insert, and delete is O(log n).

Since the keys are uniformly distributed, the expected number of keys that map to a particular index is n/M = alpha. Since we maintain alpha <= 1.5, which is in O(1), the expected number of keys in each index is O(1). Performing an AVL tree operation on a tree with expected size O(1) takes O(1) time, so the expected runtime is O(1). Rehashing is O(n) expected runtime, but with the same logic as above, is O(1) amortized.

We use O(n + M) space. Since alpha >= 0.5, we have M <= 2n, which is in O(n), so the space complexity is O(n).

```
Dict::search(target) {
        index = Dict.hash(target)
        return Dict[index].AVLsearch(target)
}
Dict::insert(key, value) {
        newAlpha = (Dict.numKeys + 1)/Dict.tableSize
        if (newAlpha > 1.5) { // Rehash
                Dict.rehash(2*Dict.tableSize)
        }
        index = Dict.hash(key)
        Dict[index].AVLinsert(key, value)
        Dict.keycount += 1
}
Dict::delete(key) {
        index = Dict.hash(target)
        result = Dict[index].AVLdelete(target)
        Dict.keycount -= 1
        newAlpha = Dict.numKeys/Dict.tableSize
```

```
        if (newAlpha < 0.5 && Dict.tableSize > 100) { // Rehash
            Dict.rehash(min(floor(Dict.tableSize/2), 100)
        }
}
```

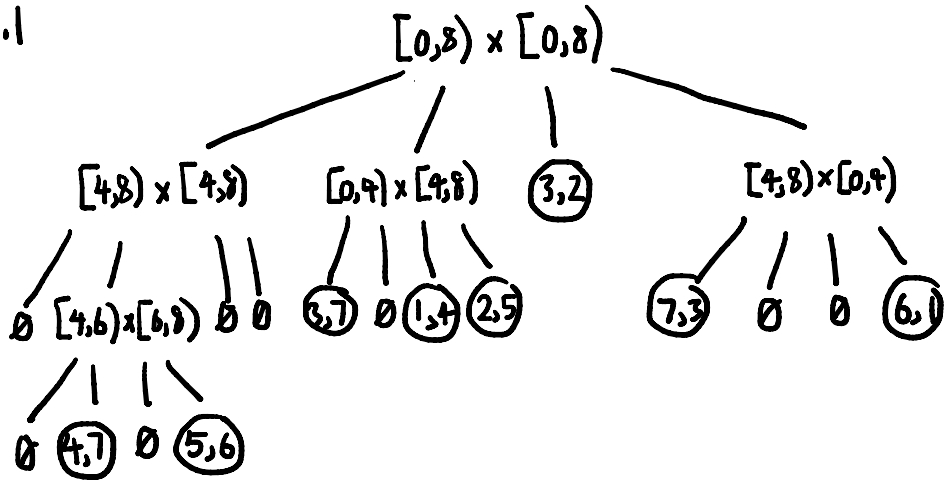
7.4

2c: This is not possible. Assume for sake of contradiction that it is possible. Then given any array of numbers, we can use heapify() to build a max-heap of the array in O(n) time, and build a BST of the array in O(n) time. We can obtain a sorted order of this array from the BST in O(n) time using an inorder traversal in O(n) time. This is a contradiction, since no comparison-based algorithm can sort an array in less than Omega(nlog(n)) time. Hence it is not possible to build a BST with the same key-value pairs in O(n) time.

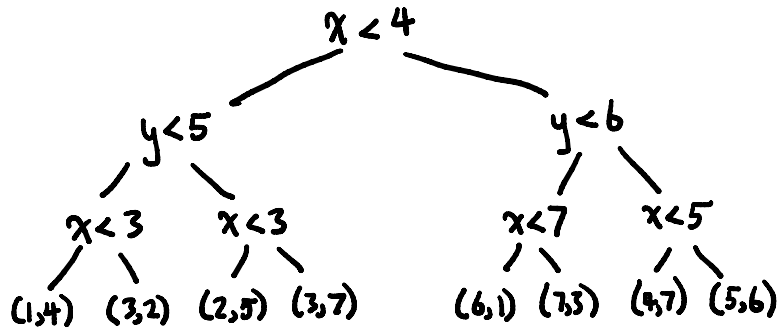
2e: The smallest keys in a max-heap will always be at the leaves of the heap. However, it cannot be determined without searching which leaves have the smallest key. Hence all the leaves must be searched, which takes Omega(n) time.

4a: At depth k, there are O(n) calls to bucket sort. Bucket sort has time complexity O(n+R) for a bucket of size n. The total running time is hence in O(nR).

8.1



8.2



8.3

The probe starts with h1(k), then steps by h2(k) each time along the table. If the GCD of h2(k) and m is d, then the algorithm must step through m/d elements before returning to the starting position h1(k).

More formally, let c = h1(k) mod m. We have that h(k, i) = (h1(k) + i*h2(k)) mod m = c + (i*h2(k) mod m). For each value of i, C is constant. Hence it varies with respect to i*h2(k), as the value of i is incremented. Hence we have

$$h(k, i+\tfrac{m}{d}) = \left[C + (i+\tfrac{m}{d})h_2(k)\right] \bmod m = \left[C + ih_2(k)\right] \bmod m + \tfrac{m}{d}h_2(k) \bmod m$$

$$= \left[C + ih_2(k)\right] \bmod m = h(k, i)$$

Hence the hash function is m/d-periodic, so the probe sequence examines m/d values before returning to h1(k), which is 1/dth of the hash table.

8.4 The probe starts with h(k), then steps by i^2 each time along the table, where i increments by 1 each time. Hence the probe varies by i^2 each time, and h(k) stays constant. We have that

$$i^2 \bmod M = (-i)^2 \bmod M$$

$$(-i) \bmod M = (M-i) \bmod M$$

$$i^2 \bmod M = (M-i)^2 \bmod M$$

Hence we have that

$$(h(k) + 0^2) \bmod M = (h(k) + M^2) \bmod M$$

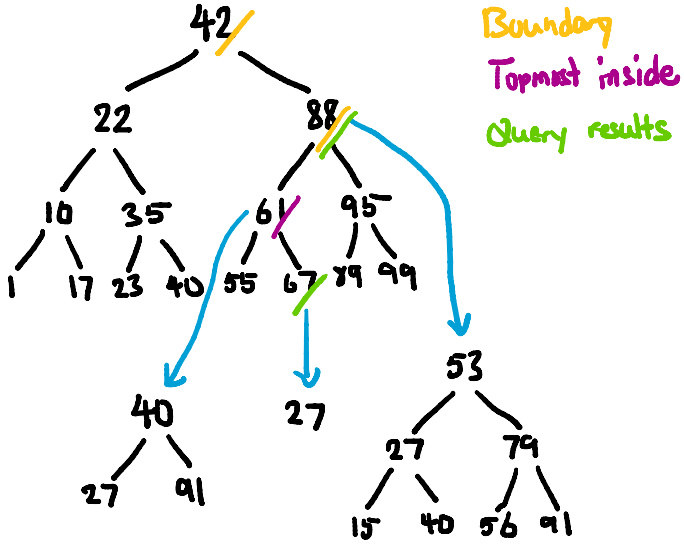
$$(h(k) + 1^2) \bmod M = (h(k) + (M-1)^2) \bmod M$$

$$\vdots$$

$$(h(k) + \lfloor \tfrac{M}{2} \rfloor^2) \bmod M = (h(k) + (M - \lfloor \tfrac{M}{2} \rfloor)^2) \bmod M$$

There are only ceil((M+1)/2) distinct values above, and hence the probe sequence visits a maximum of ceil((M+1)/2) possible locations.

9.1 a)



b)

c)

9.2 a) We use a 3D range tree, with dimensions y, x-coordinate of left point, and x-coordinate of y-point. A line segment is entirely contained in the query rectangle if both endpoints are inside the query range. Hence to do a range-search, we use the query [c,d] for y, [a,b] for x-left, and [a,b] for x-right. Since the runtime of range-search in a d-dimensional tree is O(log^d(n) + s), for

a 3D range tree it is O(log^3(n) + s).

If a line segment is reported, it must have both endpoints in the query rectangle; y is in [c,d], and both x-left and x-right are in [a,b]. If a line segment is not reported, it must violate one of the ranges in the query, which means at least one of the endpoints is not in the query rectangle.

b) We use a 3D range tree, with dimensions y, x-coordinate of left point, and x-coordinate of y-point. A line segment must have its y-coordinate in the y-range to intersect the query rectangle. We now consider the x-range. There are only two cases where the line segment does not intersect: either both endpoints are to the left of the range, or both endpoints are to the right. Hence we have that a line does not intersect if

x-right < a or x-left > b.

Hence we have that a line does intersect if

!(x-right < a or x-left > b) = x-right >= a and x-left <= b

Hence we do a range-search with the query [c,d] for y, [-inf, b] for x-left, and [a, inf] for x-right.

Since the runtime of range-search in a d-dimensional tree is O(log^d(n) + s), for a 3D range tree it is O(log^3(n) + s).

10.1 a)

0	1	0	1	2	3	4	5

b) Not found

A	B	A	C	A	B	A	C	D	A	B	A	C	A
A	B	A	C	A	B	A	C	x					
				[a]	[b]	[a]	[c]	x					
								x					
									A	B	A	C	A

10.2 For each internal node whose depth is greater than or equal to l, we count the number of leaves in its subtrie. We keep track of the node with the highest number of leaves, and return its length l prefix.

If there exists a substring of length l that appears more than once, there are at least 2 suffixes that both contain that same prefix. All of these suffixes would be under a subtrie such that the depth of the root is at least l. If there does not exist any substring of length l that appears more than once, the mostLeaves function would return {NULL, -1}, since there are no internal nodes of depth >= l, so the main function just returns the first substring of length l.

Each node of depth >= l is visited by countLeaves once, since it is only called in mostLeaves for nodes of depth >= l, and countLeaves visits each node in the subtrie exactly once. Each node of depth <= l is visited by mostLeaves once, since mostLeaves does an in-order traversal of each node of depth <= l. The main function calls mostLeaves once on the root. Hence every node is visited once, and the algorithm runs in O(n) time.

Assume each node contains its depth and a reference to its leaf with the longest suffix. Assume every leaf contains start, the start index of that suffix in s.

```
// Returns number of leaves in a node's subtrie
countLeaves(T: trie) {
    if T is empty return 0
    if T is a leaf return 1
    sum = 0
    for child in T.children {
        sum += countLeaves(child)
```

```
        }
        return sum
}

// Returns the internal node with depth >= l in the trie with the most number of leaves
// as well as its number of leaves
mostLeaves(T: trie, l: int) {
        if T is empty or a leaf return {NULL, -1}
        if T.depth >= l return {T, countLeaves(T)}
        curMax = -1
        curMaxNode = NULL
        for child in T.children {
                node, leafCount = mostLeaves(child, l)
                if (leafCount > curMax) {
                        curMax = leafCount
                        curMaxNode = node
                }
        }
        return curMaxNode, curMax
}

// Algorithm
solution(s: string, l: int, T: trie) {
        n = s.length
        node, leafCount = mostLeaves(T, l)
        if (leafCount == -1) {
                // No substring of length l appears more than once
                return s[0…l-1]
        }
        i = node.longestLeaf.start
        return s[i…i+l]
}
```

10.3

0	ALOMOMOLA$
1	LOMOMOLA$
2	OMOMOLA$
3	MOMOLA$
4	OMOLA$
5	MOLA$
6	OLA$
7	LA$
8	A$
9	$

9	$
8	A$
0	ALOMOMOLA$
7	LA$
1	LOMOMOLA$
5	MOLA$
3	MOMOLA$
6	OLA$
4	OMOLA$
9	$

A = [9, 8, 0, 7, 1, 5, 3, 6, 4, 9]

We perform binary search on A.

L = 0, R = 9, mid = 4, A[4] = 1.
We have that A[1…3] = LOM. Since MOM > LOM, we search in the right subarray.

L = 5, R = 9, mid = 7, A[7] = 6.
We have that A[6…9] = OLA. Since MOM < OLA, we search in the left subarray.

L = 5, R = 6, mid = 5, A[5] = 5.

We have that A[5…8] = MOL. Since MOM > MOL, we search in the right subarray.

L = 6, R = 6, m = 6, A[6] = 3.
We have that A[3…6] = MOM. We found the pattern.

11.1 a) 66-65-78-129-65-95-128-78-68-131

128: BA
129: AN
130: NA
131: ANA
132: A_
133: _B
134: BAN
135: ND
136: DA

b) GIVE_ME_REGIGIGAS

128: GI
129: IV
130: VE
131: E_
132: _M
133: ME
134: E_R
135: RE
136: EG
137: GIG
138: AS

11.2 a) 66-65-78-95-131-136-83-135-68-95-129-138

128: BA
129: BAN
130: AN
131: AN_
132: N_
133: N_A
134: _A
135: _AN
136: ANA
137: ANAN
138: ANAS
139: ANAS_
140: S_
141: S_A
142: _AND
143: _AND_
144: D_
145: D_B
146: _B
147: _BA
148: BANA
149: BANAN

b) We keep a global counter for the first free code-number: idx. We keep s_prev (the previous decoded value) and s_prev2 (the previous previous decoded value). For each encoded char, if the code char is in the dictionary, than we add its dictionary lookup to the table. If it is not in the table, there are two cases. If it is 1 greater than idx, we know that it must be s_prev concatenated with s_prev[0…1]. If it is equal to idx, we know that it must be s_prev concatenated with s[0].

```
LZW2::decode(C, S) {
    idx = 128
    code = C.pop()
    s = LZW2::dictionaryLookup(D, code)
    S.append(s)
    while C is not empty {
        s_prev2 = s_prev
        s_prev = s
        code = C.pop()
        if (code < idx) {
            s = LZW2::dictionaryLookup(D, code)
        } else if (code == idx) {
            s = s_prev + s[0]
        } else if (code = idx+1) {
            s = s_prev + s[0…1]
```

```
            } else {
                return "invalid encoding"
            }
            S.append(s)
            insert s_prev2 + s[0] into D with codeword idx
            idx++
            insert s_prev + s[0] into D with codeword idx
            idx++
        }
    }
```

c) ROTO_MEMENTO_TO_TOROMENT

128: RO
129: ROT
130: OT
131: OTO
132: TO
133: TO_
134: O-
135: O_M
136: -M
137: _ME
138: ME
139: MEH
140: EM
141: EME
142: MEN
143: MENT
144: NT
145: NTO
146: TO_T
147: TO_TO
148: TO_TOR
149: TO_TORO
150: ROM
151: ROME

11.3 a) IPSSM$PISSII

MISSISSIPPI$	$MISSISSIPP**I**
ISSISSIPPI$M	I$MISSISSIP**P**
SSISSIPPI$MI	IPPI$MISSIS**S**
SISSIPPI$MIS	ISSIPPI$MIS**S**
ISSIPPI$MISS	ISSISSIPPI$**M**
SSIPPI$MISSI	MISSISSIPPI**$**
SIPPI$MISSIS	PI$MISSISSI**P**
IPPI$MISSISS	PPI$MISSISS**I**
PPI$MISSISSI	SIPPI$MISSI**S**
PI$MISSISSIP	SISSIPPI$MI**S**
I$MISSISSIPP	SSIPPI$MISS**I**
$MISSISSIPPI	SSISSIPPI$M**I**

b) AIMOEOOPN$TOA

$, 9	A, 0	A, 12	E, 4	I, 1	M, 2	N, 8	O, 3	O, 5	O, 6	O, 11	P, 7	T, 10

(O, 6) (N, 8) (O, 5) (M, 2) (A, 12) (T, 10) (O, 11) (P, 7) (O, 3) (E, 4) (I, 1) (A, 0) ($, 9)

ONOMATOPOEIA$