Melissa Auclaire

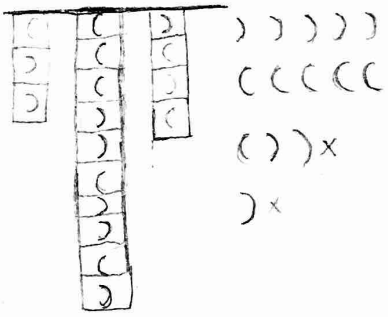
Algorithm Design Manual
Skiena - Chapter 3
Exercises

3-1 [3] A common problem for compilers and text editors is determining whether the parentheses in a string are balanced and properly nested. For example, the string ((())())() contains properly nested pairs of parentheses, which the strings)()(and ()) do not. Give an algorithm that returns true if a string contains properly nested and balanced parentheses, and false if otherwise. For full credit, identify the position of the first offending parenthesis if the string is not properly nested and balanced.

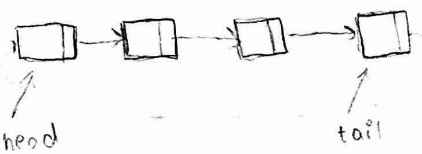
```
int lbrace = 0, int rbrace = 0
for int i=0 to string.length - 1
    if S[i] == '('
        lbrace ++
    Else if S[i] == ')'
        rbrace ++
if (lbrace == rbrace)
    return true
else
    return false
(// this solution is incorrect ***)
```

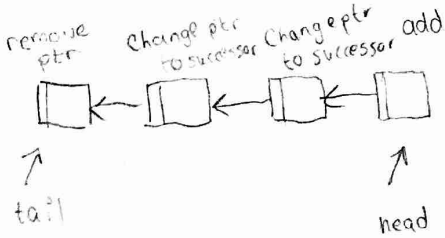
```
stack S , int lparen = 0 int rparen = 0
for int i = 0 to string str.length - 1
    S.push( S[i] )

for int i = 0 to string str.length - 1
    if (S.pop() == ')')
        rparen++;
    Else if (S.pop() == '(')
        lparen++ ;
    if (rparen > lparen)
        print "Parenthesis error at index" + i
        return false
if (lparen == rparen)
    return true
else
    print "Parenthesis error at index" + (lparen - rparen)
    return false
```

(((( )) ( )) ( ) ,) ( ) (, ( ))



3-2 [3] Write a program to reverse the direction of a given singly-linked list. In other words, after the reversal all pointers should now point backwards. Your algorithm should take linear time.





steps

1. Swap head and tail pointers
2. Add ptr to successor on last node
3. Reassign all nodes n-1 to 1's ptrs to successor
4. Remove ptr from the tail node

$O(n^2)$ Algorithm

$O(n)$ Algorithm

(Program will be written)

Melissa Auclaire

Algorithm Design Manual
Skiena - Chapter 3
Exercises

3-3 [5] We have seen how dynamic arrays enable arrays to grow while still achieving constant time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink on demand.

(a) Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.

(b) Then, give a better underflow strategy than that suggested above, one that achieves constant amortized cost per deletion.

a) Underflow Strategy

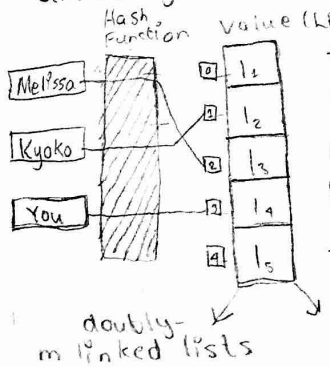
- On deletion, iterate through elements and increment a counter
- If the counter is less than $\frac{n}{2}$ full after the iteration, it must be resized
- Create a new array of $\frac{n}{2}+1$ elements if n is odd, or $\frac{n}{2}$ elements if n is even
- Store old array elements sequentially in the new array, or at $\frac{i}{2}$ with sequential probing for colliding insertions
- Delete the old array. If necessary, deallocate the memory block.

Insertions and deletions which cause the array to be slightly more or slightly less than $\frac{n}{2}$ consistently, or small arrays will cause the array to be resized often, leading to increased amortized cost.

b) Insert elements linearly into the new array using a hashing data structure. Compare the hashes of each element. This will achieve constant average time amortized cost.

Trees and other Dictionary Structures

3-4 [3] Design a dictionary data structure in which search, insertion, and deletion can all be processed in $O(1)$ time in the worst case. You may assume the set elements are integers drawn from a finite set $1, 2, \ldots, n$, and initialization can take $O(n)$ time.



or lookup table (Perfect Hash Function)

$\Theta(1)$ for search, insertion, sort

$O(n)$ for search initialization

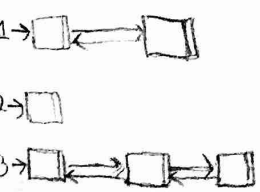
or

Perfect Hash Function

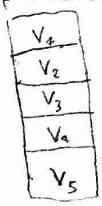
$\Theta(1)$ in all cases

Lookup table ex.

doubly-linked lists



Array Initialization for lookup table is $O(n)$ time



Melissa Auclaire

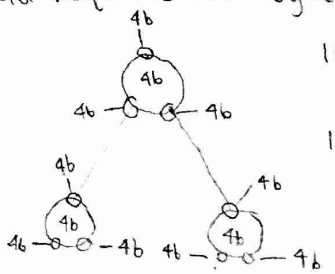
Algorithm Design Manual
Skiena - Chapter 3
Exercises

3-5 [3] Find the overhead fraction (the ratio of data space over total space) for each of the following binary tree representations on n nodes.

(a) All nodes store data, two child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

(b) Only leaf nodes store data; internal nodes store two child pointers. The data field requires four bytes and each pointer requires two bytes.

a)



16 bytes per node
n nodes
16n bytes in tree

$\frac{12}{16}$ bytes are overhead

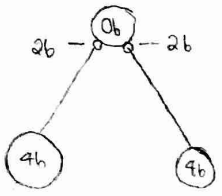
$\frac{12}{16} = \frac{3}{4}$ n bytes are overhead in tree

$\frac{1}{4}$ is the ratio of data to total tree space

$n = E + 1$

$n = n_1 + n_2 + \dots + n_h + 1$

b)



Internal nodes are 4b overhead
External nodes are 4b data
n internal nodes [overhead]
m external (leaf) nodes [data],
Tree contains 4n + 4m bytes

In a complete Binary tree $m = n - 1 \Rightarrow$

$$\frac{4n}{4n + 4(n-1)} = \frac{n}{2n-1}$$

is the ratio of data to total tree space (complete binary tree)

$$\frac{4n}{4n + 4m} = \frac{n}{n+m}$$

is the ratio of data to total tree space

3-6 [5] Describe how to modify any balanced tree data structure such that search, insert, delete, minimum, and maximum still take O(log n) time each, but successor and predecessor now take O(1) time each. Which operations have to be modified to support this?

Traversal methods: Insert, delete, search

Add a variable called current node which copies the current node on traversal. All traversal methods could be modified. The max and min can also be stored in this node, or modified. You could also link the nodes together by providing pointers from parent to child and child to parent so that they also satisfy the condition of being a doubly-linked list which has constant time access for both predecessor and successor.

3-7 [5] Suppose you have access to a balanced dictionary data structure, which supports each of the operations search, insert, delete, minimum, maximum, successor and predecessor in O(log n) time. Explain how to modify the insert and delete operations so they still take O(log n) but now minimum and maximum take O(1) time. (Hint: Think in terms of using the abstract dictionary operations, instead of mucking about with pointers and the like.)

You could store the min max in a node object as in the last problem.
To do this you would need O(1) extra space complexity and another data structure such as a lookup table, node, or 2 item array. The abstract dictionary operations could update these values in O(log n) time, paying the cost beforehand so that the access time remains O(1).

Algorithm Design Manual
Skiena - Chapter 3
Exercises

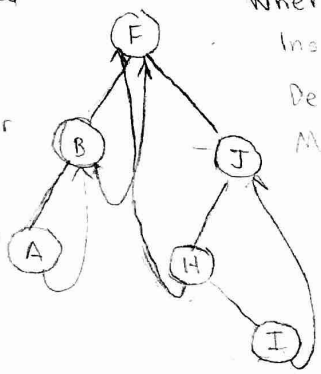
Melissa Auclaire

3-8 [6] Design a data structure to support the following operations:

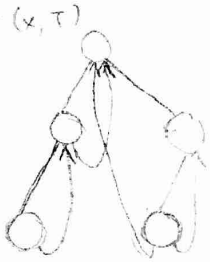
- insert(x,T) - Insert item x into the set T.
- delete(k,T) - Delete the k^{th} smallest element from T.
- member(x,T) - Return true iff $x \in T$.

All operations take O(logn) time on an n-element set.

Threaded Binary Tree allows for log(n) complete traversal



When $T = \{1, 2, \ldots, n\}$
Insert(x, T) → O(n)
Delete(k, T)
Member(x, T)

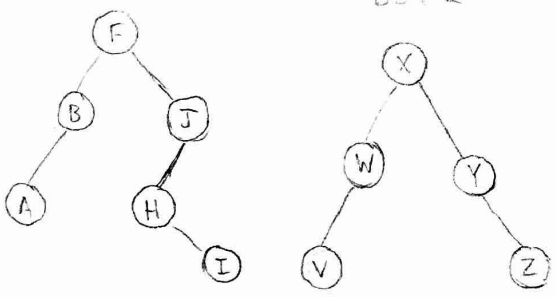


A threaded binary tree satisfies the time complexity requirements for all of these operations.

(A recursive BST traversal algorithm also satisfies O(logn) time complexity.)

3-9 [8] A concatenate operation takes two sets S_1 and S_2, where every key in S_1 is smaller than any key in S_2, and merges the two together. Give an algorithm to concatenate two binary search trees into one binary search tree. The worst-case running time should be O(h), where h is the maximal height of the two trees.

BST 1 < BST 2



Algorithm:

Retrieve max of BST 1 or min of BST 2
Delete the pointer to the parent node
Add pointer from root node of BST 1 and BST 2 to the max/min node.

3-10 [5] In the bin-packing problem, we are given n metal objects, each weighing between zero and one kilogram. Our goal is to find the smallest number of bins that will hold n objects, with each bin holding one kilogram at most.

- The best-fit heuristic for bin packing is as follows. Consider the objects in the order in which they are given. For each object place it into the partially filled bin with the smallest amount of extra room after the object is inserted. If no such bin exists, start a new bin. Design an algorithm that implements the best-fit heuristic (taking as input the n weights $w_1, w_2, \ldots, w_n$ and outputting the number of bins used) in O(nlogn) time.
- Repeat the above using the *worst-fit heuristic*, where we put the object in the partially filled bin with the largest amount of extra room after the object is inserted.

Algorithm Design Manual
Skiena - Chapter 3
Exercises
3-10[5] contd.

Melissa Auclaire

Best-fit algorithm -

```
ArrayList<Bin> bins = new ArrayList<Bin>();
ArrayList<Bin> insertable = new ArrayList<Bin>();
current ObjectWeight = objects.next().weight
for b in bins
        if(currentobjectWeight + b.value < 1000)
                insertable.add(b)
                if ( 1000 - (currentObjectWeight + b.value) < minValue )
                        minValue = 1000 - (currentObjectWeight + b.value )
                        index = b.index
bins.getItem(index).addItemTo(currentObjectWeight)
```

(Program will be written)

Worst-fit algorithm

Similar to the above, with sign changed and the difference measured against max value

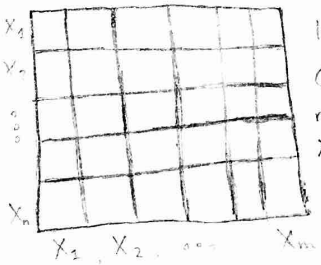
(Program will be written)

3-11 [5] Suppose that we are given a sequence of n values $x_1, x_2, \ldots, x_n$ and seek to quickly answer repeated queries of the form: given i and j, find the smallest value in $x_i, \ldots, x_j$.

(a) Design a data structure that uses $O(n^2)$ space and answers queries in $O(1)$ time.

(b) Design a data structure that uses $O(n)$ space and answers queries in $O(\log n)$ time. For partial credit, your data structure can use $O(n \log n)$ space and have $O(\log n)$ query time.

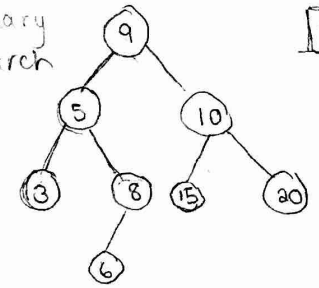
a) Build an $n \times n$ lookup table that contains all values for any given pair (i, j) $O(n^2)$ Space Complexity, $O(1)$ time complexity



Input (i, j)
Output:
min of
$x_i, x_2, \ldots, x_j$

b) Build a Binary Search Tree and store it as an array. In this case, space complexity is $O(n)$ and the time complexity is $O(\log n)$

Binary Search



9	5	10	3	8	15	20	0	0	6	0	0	...
1	2	3	4	5	6	7	8	9	10	11	12	...

Input (i, j)
Start traversal from node i until node j

Output
min of
$x_i, x_2, \ldots, x_j$

Algorithm Design Manual
Skiena - Chapter 3

Melissa Auclaire

3-12[5] Suppose you are given an input set S of n numbers, and a black box that if given any sequence of real numbers and an integer k instantly and correctly answers whether there is a subset of input sequence whose sum is exactly k. Show how to use the blackbox O(n) times to find a subset of S that adds up to k.

$S = \{x_1, x_2, x_3, \ldots x_n\}$

Black Box O(1) time
if given $\{1, 2, 3, \ldots, n \mid n \in \mathbb{R}\}$, k
output => (true or false), depending on if there is a subset of S that adds up to k

```
if (Blackbox(S, k))
    for xi in S
        S2 = S1 - xi //remove xi from S1
        if (Blackbox(S2, k))
            S1 = S1 - xi
    return S1
else
    print "There is no such subset in S"
    return {}
```

3-13 [5] Let A[1..n] be an array of real numbers. Design an algorithm to perform any sequence of the following operations:

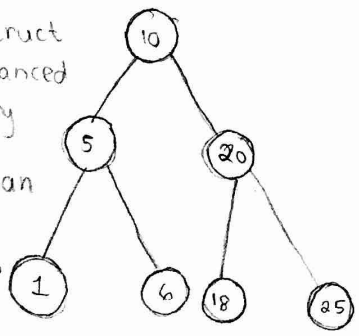
- Add(i, y) - Add the value y to the ith number.
- Partial-sum(i) - Return the sum of the first i numbers $\sum_{j=1}^{i} A[j]$.

There are no insertions or deletions; the only change is to the values of the numbers. Each operation should take O(log n) steps. You may use one additional array of size n as a workspace.

Add

```
a[i] = a[i] + y   O(1)
```

Construct a Balanced Binary Tree within an array of n elements



Partial Sum

Add the elements of each subtree to the left while descending the tree until node i is reached

```
public void sum(ith node)
    if (ith node == null)
        return 0
    return node.value + sum(node.leftChild)
```

Melissa Auclaire

Algorithm Design Manual
Skiena - Chapter 3
Exercises

3-14 [8] Extend the data structure of the previous problem to support insertions and deletions. Each element now has both a key and a value. An element is accessed by its key. The addition operation is applied to the values, but the elements are accessed by its key. The Partial_sum operation is different.

- Add(k,y) - Add the value y to the item with key k.
- Insert(k,y) - Insert a new item with key k and value y.
- Delete(k) - Delete the item with key k.
- Partial-Sum(k) - Return the sum of all the elements currently in the set whose key is less than y, i.e. $\sum_{x_i < y} x_i$.

> Implement the data structure as a self balancing AVL tree, while adding a hashtable at O(n) space complexity filled with the element key and value pairs.

> Add: Descend to the element with key k O(logn) or use hash table for lookup O(1) average case.

> Insert: Insert the node using the recursive BST insertion algorithm. Perform rotations if necessary upon insertion to maintain the AVL tree property.

> Delete: Update the node's ancestors, and rotate the binary tree to restore the search property if node k has 2 children.

Partial Sum: Access the element by descending to node k or use hash table lookup. Use the algorithm in 3-13 to add the subtree of node k.

3-15 [8] Design a data structure that allows one to search, insert, and delete an integer X in O(1) time (i.e. constant time, independent of the total number of integers stored). Assume that $1 \leq X \leq n$ and that there are $m+n$ units of space available, where m is the maximum number of integers that can be in the table at any one time. (Hint: use two arrays A[1..n] and B[1..m].) You are not allowed to initialize either A or B, as that would take O(m) or O(n) operations. This means the arrays are full of random garbage to begin with, so you must be very careful.

Init
K = 0

Insert X:
K = K+1;
A[X] = K;
B[K] = X;

Search X:
return (A[X] < K) and (B[A[X]] == X)

Delete X:
A[B[K]] = A[X]
B[A[X]] = B[K]
K = K-1

Note: This data structure does not support inserting the same X more than once

Melissa Auclaire

Algorithm Design Manual
Skiena - Chapter 3
Exercises

Implementation Projects

3-16 [5] Implement versions of several different dictionary data structures, such as linked lists, binary trees, balanced binary search trees, and hash tables. Conduct experiments to assess the relative performance of these data structures in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. This application can be efficiently implemented by maintaining a dictionary of all distinct words that have appeared thus far in the text and inserting/reporting each word that is not found. Write a brief report with your conclusions.

(Will be posted to Github)

3-17 [5] A Ceaser shift (see section 18.6 (page 641)) is a very simple class of ciphers for secret messages. Unfortunately, they can be broken using the statistical properties of English. Develop a program capable of decrypting ceaser shifts of sufficiently long texts.

(Will be posted to Github)

Interview Problems

3-18 [3] What method would you use to look up a word in a dictionary?

You can look up all the words in a dictionary of n words in a binary tree of n nodes in $O(n)$ space complexity and $O(\log n)$ time complexity. To do this, implement a binary search tree with a comperator with the words stored in each node.

3-19 [3] Imagine you have a closet full of shirts. What can you do to organize your shirts for easy retrieval?

You can organize them by type, color, or when they were purchased. It is also possible to sub categorize them with attributes that have a hierarchy of precedence (e.g. By color, then by type, then purchase date) so that all red shirts will be organized by type, and every type will be organized by the date of purchase etc., for easier access and retrieval.

3-20 [1] Write a function to find the middle node of a singly-linked list.

Assuming the middle element is 3 when even

$O(n)$ Algorithm:

- Traverse the linked list and store the number of nodes m
- Traverse the linked list again and stop at the node index $\frac{m}{2}$.

(This step may be unnecessary if just the index is needed.

Algorithm Design Manual
Skiena · Chapter 3
Exercises

3-20[4] contd.

```
// Java Solution                          LinkedList
int m = 0;                                <Integer>
public Integer linkedListHalfwayPoint (dLinkedList) {
    LinkedList<Integer> list = aLinkedList;
    Iterator itr = list.listIterator();
    while (itr.next() != null) {
        m++;
    }
    itr = list.listIterator();
    while (itr.hasNext()) {
        if (itr.next() == m/2)
            return itr.next().gethash();
    }
    return -1;
}
```

3-21 [4] Write a function to compare whether two binary trees are identical. Identical trees have the same key value at each position and the same structure.

```
int[] binaryTreeA = new int[n]; //n nodes
int[] binaryTreeB = new int[m] //m nodes

initializeA(binaryTreeA);
initializeB(binaryTreeB);

public boolean compareTrees(binaryTreeA,
binaryTreeB) {
if (binaryTreeA.getClass().equals(binaryTreeB
.class) || if(binaryTreeA.length != binaryTreeB
.length) return false;

    for(int i=0; i< binaryTreeA.length; i++) {
        if(binaryTreeA.length != binaryTreeB.length)
            return false;

    return true;
```

Note: (This code works for binary trees implemented as arrays.)

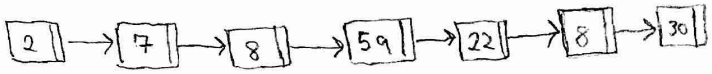
```
//Code for Linked Lists (C++)

typedef struct list {
    item_type item;   /* data items */
    struct list *next; /* point to successor */
} list;

        //Searching a list
list *search_list(list *l, item_type x) {
    if (l == NULL) return(NULL);
    if (l->item == x)
        return(l);
    else
        return(search_list(l->next,x));
}
        //insertion into a list
void insert_list(list **l, item_type x) {
    list *p;
    p = malloc(sizeof(list));
    p->item = x;
    p->next = *l;
    *l = p;
}

        //find predecessor in list
list *predecessor_list(list *l, item_type x){
    if ((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought
        on null list.\n");
        return(NULL);
    }
    if ((l->next)->item == x)
        return l;
    else
        return(predecessor_list(l->next, x));
}

        //delete node from list
delete_list(list **l, item_type x) {
    list *p;
    list *pred;
    list *search_list(), *predecessor_list();

    p= search_list(*l, x);
    if (p != NULL) {
        pred = predecessor_list(*l, x);
        if(pred == NULL) /* splice out
            *l = p->next;  Out lis*/
        else
            pred->next = p->next;
        free(p);
    }
}
```

Algorithm Design Manual
Skiena - Chapter 3
Exercises

Melissa Auclaire

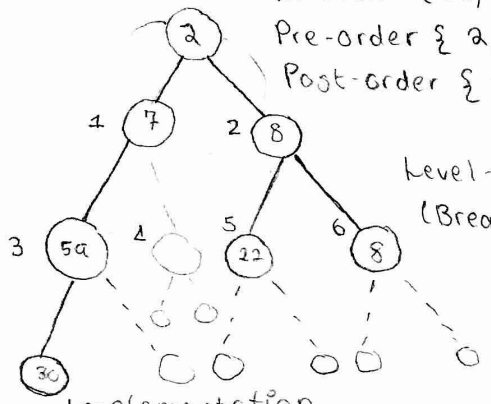
3-22 [4] Write a program to convert a binary tree into a linked list.

Linked List



In order to convert the binary tree to a linked list, we would perform a level order traversal of the Binary Tree, and set pointers to the next node of traversed nodes.

Binary Tree



Traversals:
In-order { 30, 59, 7, 22, 8, 8, 2 }
Pre-order { 2, 7, 59, 30, 8, 22, 8 }
Post-order { 30, 59, 7, 22, 8, 8, 2 }
Level-order { 2, 7, 8, 59, 22, 8, 30 }
(Breadth-first)

Binary Tree

```
class Node {
    int data;
    Node left;
    Node right;
public void addLR(){ }
    if(root.left != null)
        list.add(root.left);
    if(root.right != null)
        list.add(root.right);
}
public Node(int data) {
    this.data = data;
    this.left = null;
    this.right = null;
}
```

```
// Java Implementation
LinkedList<Node> list = new LinkedList<Node>();
public void dfs(Node root){
    if(root == null)
        return;
    list.add(root);
```

BFS Process: (Iterative)

```
addLR();
root = root.left;
addLR();
root = parent.right;
addLR();
root = parent.left.left;
addLR();
root = parent.right;
addLR();
root = parent.parent.right.left;
addLR();
root = parent.right;
addLR();
```

```
public void levelOrderQueue(Node root) {
    Queue<Node> q = new LinkedList<Node>();
    if(root == null)
        return;
    q.add(root);
    while(!q.isEmpty()) {
        Node n = (Node) q.remove();
        list.add(n.data);
        if(n.left != null)
            q.add(n.left);
        if(n.right != null)
            q.add(n.right);
}
```

3-23 [4] Implement an algorithm to reverse a linked list. Now do it with recursion.

See problem 3-2 for conceptual background
(Program will be written)

```
public void reverseList() {
    head.reverse(null);
    Node temp = head;
    head = tail;
    tail = temp;
}
```

```
public void reverse(Node prev) {
    if(next != null)
        next.reverse(this);
    next = previousNode;
```

Algorithm Design Manual
Skiena - Chapter 3
Exercises

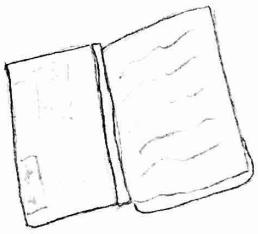
Melissa Auclaire

3-24 [5] What is the best data structure for maintaining URLs that have been visited by a web crawler? Give an algorithm to test whether a given URL has already been visited, optimizing both space and time.

The Bloom Filter provides a method of optimizing for space and time while substituting a heuristic for a complete solution. It can exclude completely all negatives (though it may also contain some false positives.)

3-25 [4] You are given a search string and a magazine. You seek to generate all the characters in search string by cutting them out from the magazine. Give an algorithm to efficiently determine whether the magazine contains all the letters in the search string.

Note: This is a pattern matching question which ask if 1 string is a subsequence of another



```
// initial string
String s1 = "xvfractiLZz";
// magazine
String s2 = "Four score and seven years ago our fathers brought
forth on this continent, a new nation, conceived in Liberty and
dedicated to the proposition that all men are created equal.";
```

Hash table complexities

	Average	Worst
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Algorithm: O(nm)

Set a character to the current character in s1 O(1)
Store magazine contents as a string s2 O(n)
while iterating through s(1) O(n) — O(n²)
    Check if character is in String s2 (O(n))
        if it is remove it
        if it isn't return false
// we reach the
return true

O(nm) worst
O(n+m) Average

Algorithm: Traverse both strings $O(n+m)_{time}$ / O(1) space
Traverse s1 and s2
Start pointers at the beginning of s1 and s2
while s2 still has characters.
    increment the pointer on s2
    Check if pointer on s1 == s2
        if so check if s1 has next
            if not return true
            if so increment s1 pointer

    If not
        check if s2 has more characters
            if not return false
            if so increment s2 pointer

return false

Algorithm:
1. Store every character in s2 in a hash table O(m) space & time
For every character in m, lookup the character, if present delete O(n)

Algorithm Design Manual
Skiena - Chapter 3
Exercises

Melissa Auclaire

3-26 [4] Reverse the words in a sentance - i.e., "My name is Chris" becomes "Chris is name My." Optimize for time and space.

Algorithm O(n+m+s) space O(2s) time
Swap first word with last
Swap 2nd word with next to last
...
middle is reached

s is the full string
n is a word
m is a second word

m	y		n	a	m	e		i	s		C	h	r	i	s
s	i	r	h	C		s	i		e	m	a	n		y	m
C	h	r	i	s		i	s		n	a	m	e		M	y

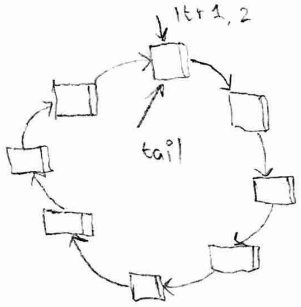
Normal
Reversed
Reverse words

Algorithm O(s) time O(1) space

O(n/2)

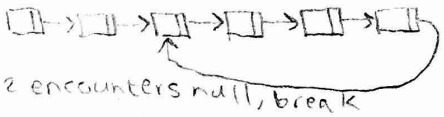
Reverse strings by reversing last character and first, next to last with second etc. O(n)
Iterate through s and reverse each word using the same method O(n/2) => O(n)

3-27 [5] Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.



Algorithm (O(λ+μ)) time
Make 2 pointers on linked list
Increment p1 pointer by 1
Increment p2 pointer by 2 -> if p2 encounters null, break
Test if the pointers are on the same node
Stop p2 and move p1 to the beginning of the list
Start incrementing both pointers by 1.
The place where p1 == p2 is the start of the loop

Use Floyd's cycle algorithm
(Tortoise and Hare Algorithm)



3-28 [5] You have an unordered array X of n integers. Find the array M containing n elements where Mi is the product of all integers in X except for Xi. You may not use division. You can use extra memory. (Hint: There are solutions faster than O(n²)).

$\{x_1, x_2, x_3, ..., x_i\} = X$

$\{m_1, m_2, m_3, ... M_i\} = M$

$M_i = \{x_2 \cdot x_2 \cdot x_3 \cdot ... \cdot x_i\}$

// Not the right question?

Algorithm - O(2m)
for all elements in X n := 0
    insert x_n into m_n

for all elements in M, n := 0
    multiply m_n with m
    store above multiplication in m

Set Mi to m

Algorithm: Brute force O(n²)
for every element in X_n
    for every element in m to m_n
        $X_n = m_1 \cdot m_2 \cdot ... \cdot m_n$

Faster!
Algorithm: Initialize another array of length n O(n) time/O(n) space
for each X_n in X
    m *= X_n
    m_n = m

Algorithm Design Manual
Skiena - Chapter 3
Exercises

Melissa Auclaire

3-28 [5] contd

Calculate the cumulative production P and Q:

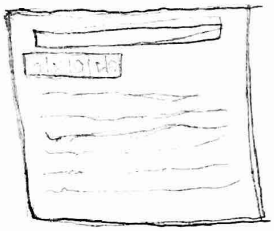
$P_0 = 1, P_k = X_k P_{k-1} = \prod_{i=1}^{k} X_i$

$Q_n = 1, Q_k = X_k Q_{k+1} = \prod_{i=k}^{n} X_i$ (Program will be written)

Calculate M

$M_k = P_{k-1} Q_{k+1}, k \in [1, n]$

3-29 [6] Given an algorithm for finding an ordered word pair (e.g. "New York") occuring with the greatest frequency in a given webpage. Which data structures would you use? Optimize both time and space.



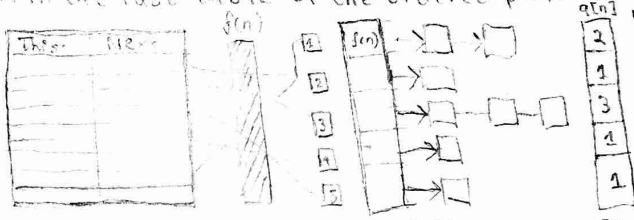
Algorithm
Count the 10 most frequently occuring words

Algorithm - Brute Force $O(3n) \Rightarrow O(n)$ / $O(2n) \Rightarrow O(n)$

If there is no word (last case) store the word and then null

1. Store all webage text in a string $O(n)$ / $O(n)$
2. For each word store the word and the next word in a hashtable $O(n)$ / $O(2n) \Rightarrow O(n)$
3. Iterate through the hash table and check for duplicate hashes $O(n)$ / $O(1)$
4. Create an array and store the number of occurences
5. Find the hash code with the highest value (call max on the hash table)

The hashcode key in this table is the hash in the last table of the ordered pair occuring with the greatest frequency



$O(n^2)$ space

$O(n)$ space

An array can also be used to store the extra value instead of another hashtable

Alternative solution:
Use a trie! (Program will be implemented)