Melissa Auclaire

Algorithm Design Manual
Skiena - Chapter 3
Exercises

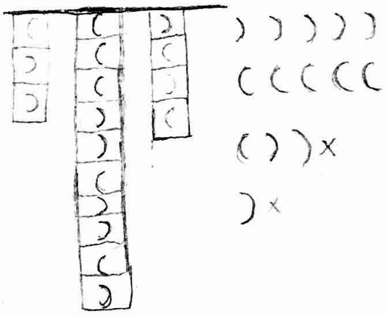
3-1 [3] A common problem for compilers and text editors is determining whether the parentheses in a string are balanced and properly nested. For example, the string ((())())() contains properly nested pairs of parentheses, which the strings)()(and ()) do not. Give an algorithm that returns true if a 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 s.length - 1
    if s[i] == '('
        lbrace ++
    else if s[i] == ')'
        rbrace ++
if (lbrace == rbrace)
    return true
else
    return false
(// this solution is incorrect ***)
```

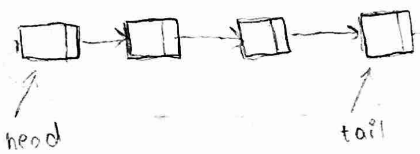
```
stack s , int lparen = 0 int rparen = 0
for int i = 0 to string str.length - 1
    s.push(s[i])

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

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



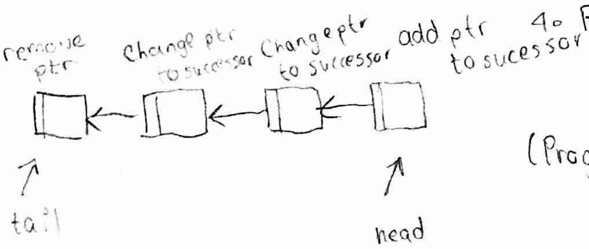
3-2 [3] Write a program to reverse the direction of a given singly-linked list. In other words, after the reversal all pointers should now point backwards. Your algorithm should take linear time.



Steps

O(n²) Algorithm

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



O(n) Algorithm

(Program will be written)

Algorithm Design Manual
Skiena - Chapter 3
Exercises

Melissa Auclaire

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.

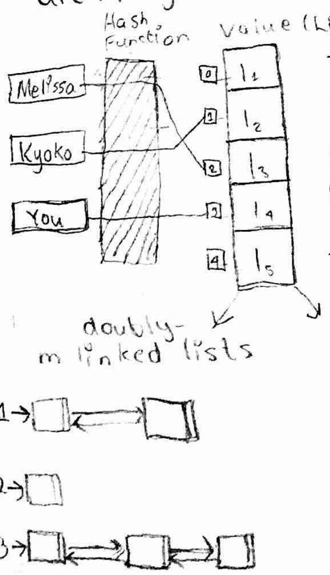
Insertions and deletions which cause the array to be slightly more or slightly less than $\frac{n}{2}$ consistently, or small arrays will cause the array to be resized often, leading to increased amortized cost.

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, ..., n, and initialization can take O(n) time.

Value (Linked Lists) or lookup table (Perfect Hash Function)



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

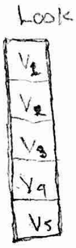
$O(n)$ for search initialization

or

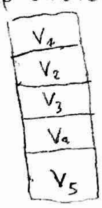
Perfect Hash Function

$\Theta(1)$ in all cases

Lookup table ex.



Array Initialization for lookup table is O(n) time



Algorithm Design Manual
Skiena - Chapter 3
Exercises

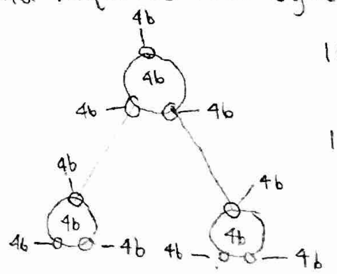
Melissa Auclaire

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

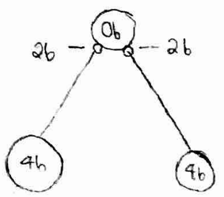
$\frac{12}{16} = \frac{3}{4}n$ bytes are overhead in tree

$\frac{1}{4}$ 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
$m = n - 1 \Rightarrow \dfrac{4n}{4n + 4(n-1)} = \dfrac{n}{2n-1}$ is the ratio of data to total tree space (complete binary tree)

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

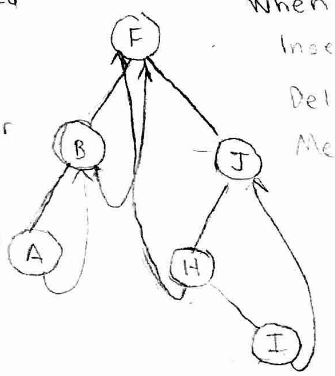
Melissa Auclaire

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 $O(\log n)$ time on an n-element set.

Threaded Binary Tree allows for $\log(n)$ complete traversal

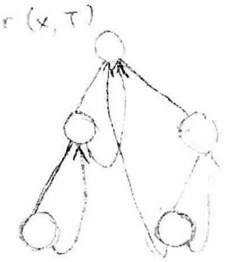


When $T = \{1, 2, \ldots, n\}$

Insert(x, T) → O(n)

Delete(k, T)

Member(x, T)

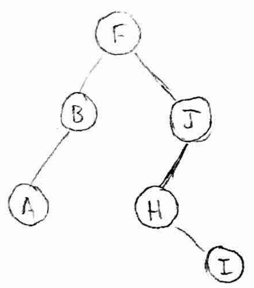


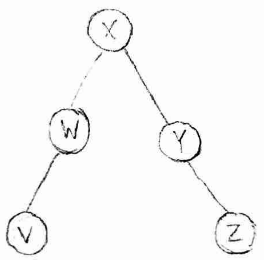
A threaded binary tree satisfies the time complexity requirements for all of these operations.

(A recursive BST traversal algorithm also satisfies $O(\log n)$ time complexity.)

3-9 [8] A concantenate 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 them together. Give an algorithm to concanterate 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 $O(n \log n)$ 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.