Fall 2018, CMPSC 465
Homework #4 solutions

1

Consider the following *hybrid* linked list-array data representation for storing integers in $[1, C]$. We use a linked list of size B objects or nodes to store the integers. Each list node points to an (initially-empty) dynamic array. The ith list node holds keys lying in an interval of width C/B. The dynamic array is resized by doubling the size with new insertions, i.e., the array size would grow as $2, 4, 8, 16, \ldots$. The arrays hold integer keys (and their counts in the adjacent cell) in unsorted order.

- Write pseudocode for testing membership of key k. If the key is present, the algorithm must return the count. If not, it must return 0. What are the best-case and worst-case times for membership queries (in terms of n, C, B), given that there are n elements already stored in the structure?

  Solution:
  In the hybrid structure, each list node has two fields, A and *next*. For any node x, $x.A$ points to the dynamic array, and $x.next$ points to the next node in the the list. Let us assume two variables associated with the array A, *size* and *count*. $A.size$ has the current size of the array and $A.count$ has the current number of values in the array. Further, assume that B divides C, i.e., C/B is an integer.

  For testing membership, we first need to locate the appropriate list node the key may belong to. Assuming list nodes are numbered from 1 to B, the array A of node i would hold keys in the interval $[(C/B)i + 1, (C/B)(i + 1)]$. So, given a key k, we first need to traverse down to the list node i that satisfies the condition $(C/B) * i + 1 \leq k \leq (C/B) * (i + 1)$. This list traversal requires i operations. The best case is $i = 1$ (first node) and the worst case is $i = B$ (last node). Once we reach the appropriate list node, because A is unsorted, we need to inspect all values in the array to see if the key lies in it. This requires $A.count$ operations. $A.count$ can range from 0 to C/B, and so the worst-case operation count is C/B. Also, for the case when $C/B >> n$, all n elements may be in the bucket A of the node i we are considering, and so a tighter upper bound on the worst-case time would be n in this case. Thus, the best case time is $\Theta(1)$ and the worst case is $\Theta(B + \frac{C}{B})$, or $\Theta(B + n)$ if $C/B >> N$.

  The key search pseudocode is given below:

```
HybDataStruct-Search(L, k)
x = L
i = 1
while k < ((C/B) * i + 1)
    x = L.next
    i = i + 1
c = x.A.count
for i = 1 to 2c by 2
    if x.A[i] == k
        return x.A[i + 1]
return 0
```

- Write pseudocode for inserting a key k that is known to be already present in the structure. What are the best-case and worst-case times for this operation, given that there are n elements already stored in the structure?

  Solution: If the key is already known to be present, its count would be non-zero. We just need to locate the key and increment its count. The best and worse-case times are the same as the previous case, as the algorithm is nearly identical. The count update pseudocode is given below:

```
HybDataStruct-UpdateCount(L, k)
x = L
i = 1
while k < ((C/B) * i + 1)
    x = L.next
    i = i + 1
c = x.A.count
for i = 1 to 2c by 2
    if x.A[i] == k
        x.A[i + 1] = x.A[i + 1] + 1
        return
```

- Write pseudocode for inserting a new key k into the structure. What are the best-case and worst-case times for this operation, given that there are n elements already stored in the structure?

  Solution: For inserting a new key, we need to first locate the appropriate node, then check the size of array A that the node points to. If the array needs to be expanded, we need to initialize a new array of double the size, copy old values into new array, and then finally add new key in there. The best-case and worst-case bounds are again the same as previous.

```
HybDataStruct-Insert(L, k)
x = L
i = 1
while k < ((C/B) * i + 1)
    x = L.next
    i = i + 1
c = x.A.count
s = x.A.size
if s == 0
    Let B[1..2] be a new array
    B.size = 2
    B.count = 1
    B[1] = k
    B[2] = 1
    x.A == B
elseif 2c == s
    Let B[1..2s] be a new array
    B.size = 2s
    B.count = c + 1
    Copy 2c elements of x.A to B, free x.A
    B[2c + 1] = k
    B[2c + 2] = 1
    x.A == B
else x.A.[2c + 1] = k
    x.A[2c + 2] = 1
    x.A.count = c + 1
```

2

- Consider a sequence of +1's and -1's with the property that the sum of any prefix of the sequence is never negative. For example, the sequence $+1, -1, +1, -1$ satisfies this property, but $+1, -1, -1, +1$ does not, since the prefix $+1 - 1 - 1 < 0$. Describe any relationship between such a sequence and a Stack push(x) and pop() operations.

  Solution:
  Associate a push operation with a +1 in the sequence and pop() operation with a -1. The sequence is given to be such that the sum of any prefix of the sequence is never negative. If we perform push() and pop operations in the same sequence, then the property the stack would satisfy is that a pop operation will never be called on an empty stack, or that a pop operation will never return NIL.

- A *matched string* is a sequence of {, }, (,), [, and] characters that are properly matched. For example, "{{()[]}}" is a matched string, but this "{{()]}" is not, since the second { is matched with a]. Show how to use a stack so that, given a string of length n, you can determine if it is a matched string in O(n) time.

Solution:
The input is a sequence of left and right brace/parenthesis/bracket characters in some order. Here's how we can use a stack to tell if the string is *matched*: when we see a left {, (, or [character in the sequence, we *push* this character to the stack. When we see a right },), or] character, we perform a *pop*. If the character returned in the pop operation does not match the type of the current symbol (e.g., if the current symbol is), but the result of pop is [), then we can say that the string is not *matched*. We continue performing push and pop operations in this manner, until the end of the string is reached. Since there are n characters and assuming push/pop operations takes constant time with our implementation of the Stack ADT, the overall running time is $O(n)$.

3

Describe a simple array-based implementation of the List ADT. What are the asymptotic running time costs of various List ADT operations? For simplicity, ignore array resizing costs.

Solution:
The List ADT is used to store a linear sequence $x_0, x_1, \ldots, x_{n-1}$ and supports five operations: `size()`, `get(i)`, `set(i, y)`, `add(i, p)`, and `remove(i)`. Let us use an array X to store the sequence. Further, let's assume that we store the current sequence size in a separate variable s.

For `size()`, we simply return the value of s. This is constant time.

For `get(i)`, we return $X[i]$. This is also constant time.

For `set(i, y)`, we do $X[i] = y$. This is also constant time.

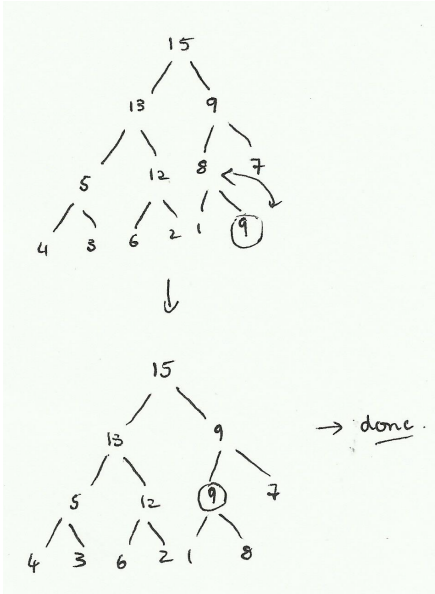
For `add(i, p)`, we need to expand the array size by 1 and then displace all the elements in the sequence $X[i]$, $X[i+1]$, ..., to the right by 1. We then set $X[i] = p$. The worst case for diplacement would be when we are adding at position 0, and the array already has n elements. This requires n right shifts, and so the worst-case cost of this step is $\Theta(n)$.

For `remove(i)`, we need to return $X[i]$ and decrement s (which are constant time operations), but then displace all the elements in the sequence $X[i+1]$, $X[i+2]$, ..., to the left by 1. The worst case for diplacement would be when we are removing the value at position 0, and the array already has n elements. This requires $\Theta(n)$ left shifts, and so the worst-case cost of this step is $\Theta(n)$.

4

- Illustrate the operation of MAX-HEAP-INSERT(A, 9) on the the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 3, 6, 2, 1 \rangle$.

Solution:



- Give pseudocode for the routine MIN-HEAP-DELETE(A, k). Assume that key k is at location l of the heap and that $1 \leq l \leq n$.

  Solution:

  MIN-HEAP-DELETE (A, k)

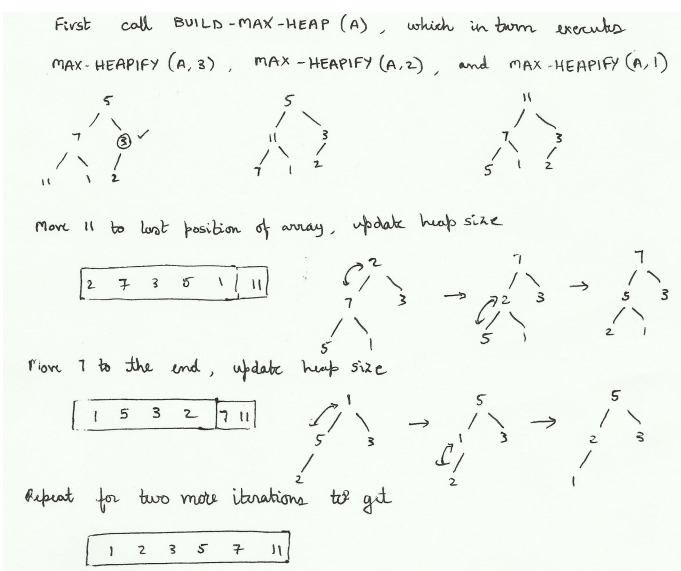
  Exchange A[k] with A[A.heap-size]

  A.heap-size = A.heap-size - 1

  MIN-HEAPIFY (A, k)

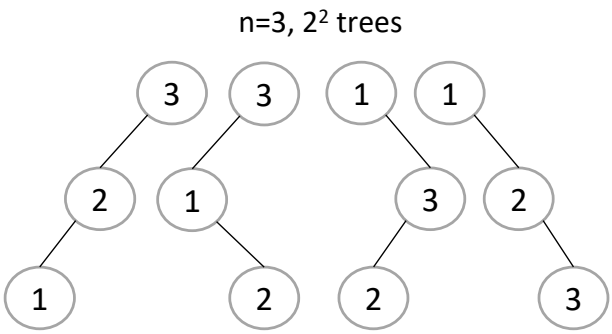
- Show how heapsort would work for the input array $A = \langle 5, 7, 3, 11, 1, 2 \rangle$.

  Solution:



5

- Describe how to add the elements $\{1, \ldots, n\}$ to an initially empty binary search tree in such a way that the resulting tree has height $n-1$. How many ways are there to do this?
  Solution:
  Let us assume the size of binary tree n refers to the number of internal nodes, or the nodes that store a data value. A binary tree with n nodes will have height $n-1$ if and only if all nodes except one node has one leaf (dummy) child and one non-leaf child, and exactly one node has two leaf children. we want all the internal nodes to form a chain. Also, the data values must satisfy the BST property. Starting from the root, each time, we have make of two choices, either branch right or branch left. Since we do this for $n-1$ nodes, the total number of possibilities is 2^{n-1}. Also, because the elements are distinct, notice that the tree structure uniquely determines the values to be stored at each node. Hence, the number of ways is 2^{n-1}. We show below the BSTs when $n = 3$. The (dummy) leaf nodes are not shown.



- Prove that, if a binary tree, T, has at least one leaf, then either (a) T's root has at most one child or (b) T has more than one leaf.
  Solution:
  The phrase "T has at least one leaf" can be broken down to two cases: (i) T has exactly one leaf, and (ii) T has more than one leaf. Case (ii) is exactly (b), so there is nothing to prove. So we need to show that if T has exactly one leaf node, then the root has at most one child. If T has exactly one leaf node (say node l), then there must be a unique path from the root r to this leaf node (using the result from exam 2, that in any tree, there is a unique path between every pair of nodes). Let this path be $r, u, \ldots, l$. Now u must be r's only child, because if r has another child v, the unique path result does not hold true.

6

Consider inserting the keys 10, 12, 34, 4, 15, 28, 17, 88, 59 into a hash table of length $N = 11$ using open addressing with the hash function $h(k) = k \bmod N$. Illustrate the result of inserting the keys using linear probing.

Solution:
The hash table of size 11 will have 88, 12, 34, X, 4, 15, 28, 17, 59, X, 10. The X symbol indicates an empty cell.

7

- Consider a BST T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.
  Solution:
  If y is a successor of x, then the value at y must be greater than the value at x. y should occur to the right of x. x could be the left child of a node or the right child of a node. If it is a left child, then all nodes from the root to the parent are ancestors. y has to be the parent because all the nodes above the parent will have a value greater than y. If x is a right child, then the successor must be an ancestor of the parent of x. (This is better illustrated with a figure, which I will add soon.)
- Give pseudocode for the TREE-PREDECESSOR procedure.
  Solution:
  Very similar to TREE-SUCCESSOR, see slides.