# cs758_assignment3

Tucker DiNapoli

September 22, 2014

## Contents

11.1-4: written in C like psuedo code void insert (dictionary dict, element x){ dict.arr[x.key] = x; $\text{mark}_{\text{used}}$(dict, x.key); } void delete (dictionary dict, element x){ dict.arr[x.key] = NIL; $\text{mark}_{\text{unused}}$(dict, x.key); } element search (dictionary dict, key k){ if ($\text{check}_{\text{used}}$(dict, k)){ dict.arr[k]; } else { NIL; } }

void $\text{mark}_{\text{used}}$(dictionary dict, key k){ int x = dict.aux[k/sizeof(int)]; x |= 1 « k % sizeof(int); } void $\text{mark}_{\text{unused}}$(dictionary dict, key k){ int x = dict.aux[k/sizeof(int)]; x &= ~(1 « k % sizeof(int)); } bool $\text{check}_{\text{used}}$(dictionary dict, key k){ int x = dict.aux[k/sizeof(int)]; return x & (1 « k % sizeof(int)); }

void $\text{mark}_{\text{used}}$(dictionary dict, key k){ push(dict.aux, k); } void $\text{mark}_{\text{used}}$(dictionary dict, key k){ for(i=0;i<dict.aux.len;i++){ if(dict.aux[i] `= k){ dict.aux[i]` = dict.aux[dict.aux.len–]; return; } } } bool $\text{check}_{\text{used}}$(dictionary dict, key k){ for(i=0;i<dict.aux.len;i++){ if(dict.aux[i] == k){ return 1; } } return 0; } void push($\text{aux}_{\text{array}}$ arr, key k){ if(arr.size `= arr.len){ arr.size *` 2 realloc(arr, arr.size); } arr[arr.len++] = k }

11.2-6 (optional):

11.3-1: search (list l, key k) hv = h(k) while(l.next) if(hv `= l.data.hv)` `if(k =` l.data.key) return l.data end end l = l.next end return nil end i.e store the hash value with the key and compare the hash values first, only comparing the keys if the hash values match. Personally this is how I search the buckets of a hash table using chaning.

12.1-2: The difference between a binary search tree and a min heap is that a binary search tree has an ordering between each element, whereas a min heap has an ordering only between a parent node and it's children. In a binary tree for a node n and it's children l and r the relation l <= n <= r holds, whereas in a heap only n <= l and n <= r hold, there is no ordering beteween the children.

It is fairly easy to show that it takes O(n lg n) time to print out the keys in a min heap of size n. Given that it takes O(n) time to print out n keys from a sorted array and it takes time O(lg n) to remove the minimum element of a heap (since each time the minimum element is removed the heap propertity must be reestablished, which takes time O(lg n)) thus to print out the elements from a heap of n elements in order will take O(n lg n)

12.2-4: I've spent a while thinking about this, and I must misunderstand the question because as far as I can tell this property is ensured by the ordering of a binary tree. For a node b with a left child a and right child, a (and it's children) <= b <= c (and it's children) by defination.

I can't think of a way for this not to be true. 12.2-5(optional): 12.3-3: we need to call tree-insert once for each element, ignoring time spent by tree-insert this takes time O(N). Thus the best and worst behavior are going to be the best and worst behavior of tree-insert * O(N).

The time complexity of tree-insert is O(h) where h is the height of the tree. In the best case the height is lg(n) giving us a best case running time of O(n lg n). In the worst case the height is n, giving a worst case running time of $O(n^2)$.