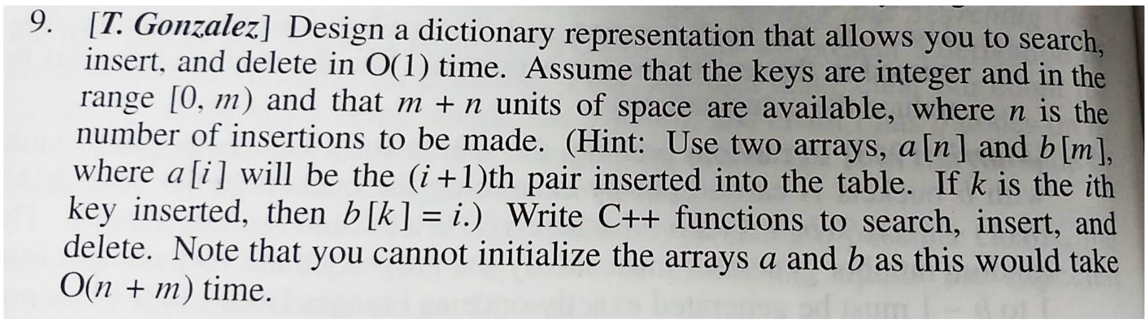
Homework 3

Data Structures and Object-oriented Programming

Instructor: Feng-Jian Wang
Due: 2021/05/31

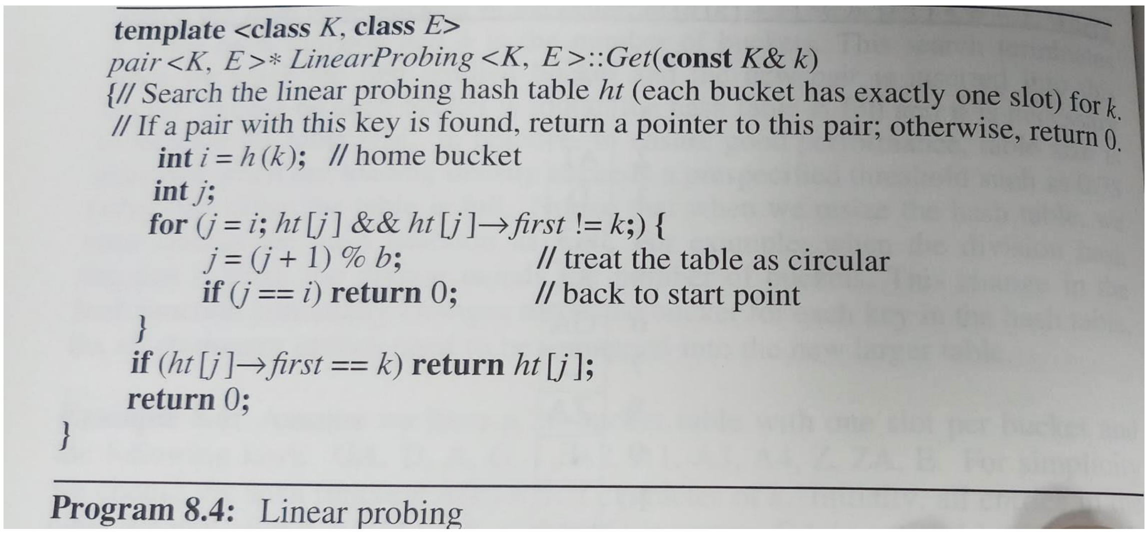
Note: Problem 10 is a bonus problem. It is not mandatory but if you solve it, you will get bonus marks.

1. Prove that a graph *G* is bipartite iff it contains no cycles of odd length.
2. Write a nonrecursive version of *QuickSort* incorporating the median-of-three rule to determine the pivot key.
3. Prove that *MergeSort* is stable.
4. Write an iterative Natural Merge Sort function using arrays as in function *MergeSort*. How much time does this function take on an initially sorted list? Note that *MergeSort* takes O(*n log n)* on such an input list. What is the worst-case computing time of the new function? How much additional space is needed?
5. Heap Sort is an unstable. Give an example of an input list in which the order of records with equal keys is not preserved.
6. [***T. Gonzalez***] Let $s = \{s_1, s_2, s_3, \ldots, s_n\}$ and $t = \{t_1, t_2, t_3, \ldots, t_r\}$ be two sets. Assume $1 \le s_i \le m$, $1 \le i \le n$, and $1 \le t_i \le m$, $1 \le I \le r$. Using the idea of Exercise 9, write a function to determine if $s \subseteq t$. Your function should work in O(r + n) time. Since $s \equiv t$ iff $s \subseteq t$ and $t \subseteq s$, one can determine in linear time whether two sets are the same. How much space is needed by your function?



9. [*T. Gonzalez*] Design a dictionary representation that allows you to search, insert, and delete in O(1) time. Assume that the keys are integer and in the range $[0, m)$ and that $m + n$ units of space are available, where n is the number of insertions to be made. (Hint: Use two arrays, $a[n]$ and $b[m]$, where $a[i]$ will be the $(i+1)$th pair inserted into the table. If k is the ith key inserted, then $b[k] = i$.) Write C++ functions to search, insert, and delete. Note that you cannot initialize the arrays a and b as this would take $O(n + m)$ time.

7. Write a C++ function to delete the pair with key *k* from a hash table that uses linear probing. Show that simply setting the slot previously occupied by the deleted pair to empty does not solve the problem. How must *Get* (Program 8.4) be modified so that a correct search is made in the situation when deletions are permitted? Where can a new key be inserted?



```
template <class K, class E>
pair<K, E>* LinearProbing<K, E>::Get(const K& k)
{// Search the linear probing hash table ht (each bucket has exactly one slot) for k.
// If a pair with this key is found, return a pointer to this pair; otherwise, return 0.
    int i = h(k);  // home bucket
    int j;
    for (j = i; ht[j] && ht[j]→first != k;) {
        j = (j + 1) % b;            // treat the table as circular
        if (j == i) return 0;       // back to start point
    }
    if (ht[j]→first == k) return ht[j];
    return 0;
}
```

Program 8.4: Linear probing

8. Let *t* be an arbitrary binary tree represented using the node structure for a leftist tree.
    a) Write a function to initialize the *shortest* data member of each node in *t*.
    b) Write a function to convert *t* into a leftist tree.
    c) What is the complexity of each of these two functions?
9. Let *S* be an initially empty stack. We wish to perform two operations on S: *Add*(x) and *DeleteUntil* (x). These are defined as follows:
    (a) *Add* (x): Add the element x to the top of the stack S. This operation takes O(1) time per invocation.
    (b) *DeleteUntil* (x): Delete elements from the top of the stack up to and including the first x encountered. If p elements are deleted, the time taken is O(*p*).

    Consider any sequence of *n* stack operations (*Adds* and *DeleteUntils*). Show how to amortize the cost of the *Add* and *DeleteUntil* operations so that the amortized cost of each is O(1). From this, conclude that the time needed to perform any such sequence of operations is O(n).
10. Assume that each node in an AVL tree has the data member *lsize*. For any node, *a*, *a→lsize* is the number of nodes in its left subtree plus one. Write a C++ function Avl::*Find(k)* to locate the *k*th smallest key in the tree. Show that this can be done in O(log *n*) time if there are *n* nodes in the tree.