BITS Pilani, Pilani Campus
2nd Semester 2016-17 - Mid-Semester Test
CS F211 Data Structures & Algorithms

==

Time: 90 minutes (Open Book) Marks: 50

==

IMPORTANT NOTE:

- ***Write algorithms in C-like pseudocode. Do not write prose.***
- ***Answers will be marked for correctness as well as for clarity / precision and efficiency.***

End of NOTE.

Q1. [Expected Time: 24 minutes Marks: 3+3+2+2=10M]

(a) Use the partitioning procedure (from Quick Sort) to design an algorithm to sort a list of *N* values each with a key *S*, *C*, *H*, or *D* using at most ***2*N*** comparisons. Assume *part(Ls,lo,hi,piv)* is available and returns *p* such that *Ls[j]<=piv* if *lo<=j<p*; *Ls[p]=piv*, and *Ls[j]>piv* if *p<j<=hi* . Also assume S < C < H < D.

(b) What is the worst-case number of *swap* operations required in an implementation of *partition* that scans the list from both ends? Let the size of the list be *N*. [Note: The answer should be an accurate function i.e. do not use order notation. End of Note.] When does the worst case occur?

(c) Why does Insertion Sort run faster than Quick Sort on "small" lists?

(d) Which of the following implementations of *insertInOrder* will perform better if the list Ls is stored in a magnetic tape (or other sequential access data storage)?

```
insertInOrder1(x, Ls, size)
{
        for (j=0; j<size && Ls[j]<x; j++) /*do nothing */ ;
        if (j==size) { Ls[j]=x; return; }
        do { t=Ls[j]; Ls[j]=x; x=t; j++; } while (j<size);
}

insertInOrder2(x, Ls, size)
{
        for (j=size; j>0 && Ls[j]>x ; j--) Ls[j]=Ls[j-1];
        Ls[j]=x;
}
```

Q2. [Expected Time: 5 minutes Marks: 1+3=4M]

Consider the following sorting algorithm:

```
maxsort(Ls, N)
{
  if (N<=1)  return;
  mI = findMax(Ls, N); // findMax returns the index of the maximum value in Ls[0]..Ls[N-1]
  swapArrElem(Ls,N-1,mI); // swapArrElem(A,x,y) swaps the values at A[x] and A[y]
  maxsort(Ls,N-1);
}
```

(a) What is the space complexity of *maxsort* (as given)? [*finMax* uses ***O(1)*** space, worst case] .

(b) Rewrite *maxsort* as an ***O(1)*** – worst-case – space algorithm.

Q3. [Expected Time: 15 minutes Marks: 6M]

Design an ADT *MaxStack* with the following operations – each of ***O(1)*** time complexity – on a LIFO list:

- *push* (adds the given element to the top of the list),
- *pop* (deletes the top element and returns it),
- *create* (creates a new list), and
- *max* (returns the maximum value in the list).

Q4. [Expected Time: 36 minutes **Marks: 4+4+6+4+6=24M]**

(a) Consider the following hash function for string *x* of length *k*:

$$h(x,a,m) = (x_0 * a^{k-1} + x_1 * a^{k-2} + \ldots + x_{k-2} * a + x_{k-1}) \bmod m$$

a is a constant in the range *1..m-1*. Write a $\theta(k*b^2)$ time algorithm for *h*, where ***b=log(m)***.

(b) Suppose the inputs are hyperlinks (i.e. URLs for web pages). Change the hash function in (a) and your algorithm so as to reduce the time complexity of hashing to $\theta(\log(k)*b^2)$.

[**Hint**: You may sample the characters. **End of Hint.]**

(c) Design a hashtable *T* (with operations *create*, *add*, and *find*) such that:

- the hash function, *h1*, is the one you designed in answer to (b)
- collision is resolved by storing the values in a nested hashtable using an additional hash function *h2*: i.e. *T[j]* is a hashtable for each *j* in *0..m-1* and
- each hashtable *T[j]* is separately chained i.e. collision due to *h2* is resolved by chaining elements in a linked list.

[**Note:** Write algorithms for the three operations. Assume duplicates don't happen. (i.e. input strings are unique.). **End of Note.**]

(d) Design a suitable hash function *h2* for your hashtable design in (c) considering the input strings (i.e. URLs), the size of the top level table *m*, and the hash function *h1*. Choose an appropriate modulus (i.e. size of the nested tables) and justify your choices (of hash function and modulus).

(e) Analyse the worst-case and expected-case time complexity measures of *add* and *find* operations in your answer to (c). Account for the hashing cost as well in your analyses.

Q5. [Expected Time: 10 minutes **Marks: 6M]**

Consider a list of (first degree) student records at BITS (from creation of the University to today). The list is to be sorted by ID i.e. to be *multi-key sorted* by the following fields in that order:
<**AdmissionYear**> <**Campus**> <**Degree**> <**3DigitCode**>.

- The University was created in 1964. Each year at most 1000 first degree students are admitted in a campus. Number of students within a degree may vary.
- Campus codes are P, D, G, and H. (P < D < G < H)
- Degree codes are A1,A2,A3,A4,A5,A7,A8,B1,B2,B3,B4,B5, and D1 in increasing order.

Design your own sorting algorithm using any or all of bucket, radix, and insertion sorting ideas:

- You may assume procedures *bucket(Ls, lo, hi, bucketList, numBucks, getKey)*, and *insertSort(Ls, lo, hi, getKey)* are available.
  - Mention any assumptions you make about these functions and any additional helper functions you use.
- Note that *getKey* is a function – that is passed as an argument to these sorting procedures - to get the key of a given element:
  - Mention all the *getKey* functions used (and what they return) for clarity.
- . Assume bucketList is an array of pairs of the form <bucket, count> i.e. for instance, bucketList[2].count gives the number of elements added to bucketList[2].bucket.

For instance, *bucketSort(Ls,lo,hi, bucketList, numBs, getCampus)* will bucket elements *Ls[lo]..Ls[hi]* into *numBs* buckets in *bucketList* using *getCampus(e)* to get the key of an element *e*
The sorted output should be back in the original space (i.e. Ls).

===================================END=====================================