CIS 2101 (DSA) Prefinal Review – Answer Key

In open hashing, the worst-case time complexity for searching an element within a dictionary of N elements in M memory spaces is: **O(N)**

This occurs if all elements reside with a single linked list (chain) – only one space in the hash table used.

In closed hashing where collisions are resolved using linear probing, the worst-case time complexity for searching an element within a dictionary of N elements inside M memory spaces is: **O(N)**

This occurs if the hash table is full (out of EMPTY spaces), especially in the later parts when elements are already being inserted at least once for every space in the hash table, regardless of the number of elements present in it.

Which of the following is/are TRUE? **A, C, D**

A. The value returned by all hash functions must be one that refers to a valid index in an array containing the dictionary's elements.
B. The hash function returns the location of the element.
C. Open hashing allows the set to be stores in potentially unlimited space.
D. Closed hashing uses a fixed space for storing and thus limits the size of the sets.

The hash function may not always return the location of the element but rather the starting point in searching for the element.

It is the condition occurring in closed hashing scenarios where an element cannot be stored in a position returned by the hash function because of a non-synonym element occupying the space. **Displacement**

Collision = inability for the element to be stored because of a synonym element occupying the space.

The relationship between the packing density and collision frequency in a closed hashing environment is:

Directly Proportional **Inversely Proportional** Can't be determined

Remember the apple story from Miss? If there are few apples, the more likely the people will fight for them compared to when there are many extra apples in it.

A hash function is designed to group names (case-insensitive) according to the first two letters of the name. How many memory locations must be allocated to be able to distinctively accommodate all the groups? **676**

26 * 26 = 676 (1 group for each combination of two letters)

In how many distinct ways can a searching operation in a closed hashing environment be terminated?

3

The scenarios are: (1) When element is found, (2) When an empty space is reached, and (3) When all spaces in the hash table are accessed (worst case)

If a heap is implemented to form a full binary tree with height 9 when the heap is fully populated, what must be the size of the array? **1023**

We can apply sum of geometric sequence where $a_1 = 1$, n = 9, r = 2, that will give 1 + 2 + 4 + … (9 terms total) = 1023

What is the average time complexity of the sorting x elements in an array using the heapsort algorithm? **O(x log x)**

For construction of the heap its O(x) for insertion or O(x/2) which is still O(x) when written as a notation if we start from lowest level parent, and for each iteration in insertion heapify takes up to O(log x). Then for deletemin/deletemax that's O(x) elements to be deleted and in each deletion its O(log x) to heapify again. Overall it takes [2(x * log x)] operations which can be denoted as O(x log x).

What is the average time complexity for finding the largest element in a max heap consisting of x elements? **O(1)**

In a max heap, the root always contains the largest element.

Consider a heap with 250 active elements and the root at index 0. If a heapify procedure is to be implemented, at what index should the process start with? **124**

Formula: Integer result (a.k.a. floor) of (lastNdx-1)/2 ➔ (249-1)/2 = 124

The following operation/s in priority queue share the same running time when implemented via array and singly linked list (with pointers to first and last elements) where in both cases the elements are sorted in ascending order:

insert() deleteMin() deleteMax() Both deleteMin() and deleteMax() **None of the choices**

The time for insertsorted() is inversely proportional for array and linked lists – example, if element is closer to the minimum, several elements will be shifted in array while in linked list we only have to traverse a few elements. For deleteMin() in array its O(n) and O(1) in linked list. For deleteMax its O(1) in array but O(n) in linked list since traversal is required to obtain the second largest element in the linked list.

A department store has a database management system whose data is stored via closed hashing whose collision resolution is through linear probing. Suppose that in all the hash tables, the packing density to be maintained is 80% by rule of thumb, fill in the blanks complete the table to determine the optimal number of data records to be inserted and the corresponding table size.

Category	Optimal # of data records	Table Size
Employees (Employee ID)	2048	**(1)**
Items (Item ID)	12120	**(2)**
Invoices (Invoice ID)	**(3)**	138000

(1) 2560 (2) 15150 (3) 110400

The following shows four hash functions where i is an integer key.

Which one of the following hash functions on integers work most efficiently over a hash table with 10 elements (indices 0 through 9)? Which among them, on the other hand, is the least efficient to use? Explain your answer accurately.

$(A)\ \ h(i) = i^2\ mod\ 10$	$(B)\ h(i) = i^3\ mod\ 10$	$(C)\ h(i) = \ (11 * i^2)\ mod\ 10$	$(D)\ h(i) = \ (12 * i)\ mod\ 10$

Consider the possible keys to be returned from each of the functions above. Our target is to ensure that all the spaces can be targeted to minimize collisions which can be achieved when all the spaces are being targeted and when there are no unused spaces.

Hash function:	Digits that were utilized: (Possible values to be return)
$i^2\ mod\ 10$	0, 1, 4, 5, 6, 9
$i^3\ mod\ 10$	0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (most efficient – all 10 of 10 spaces targeted)
$(11 * i^2)\ mod\ 10$	0, 1, 4, 5, 6, 9
$(12 * i)\ mod\ 10$	0, 2, 4, 6, 8 (least efficient – only 5 of 10 spaces targeted)

Therefore, function (B) is the best to be implemented while function (D) is the worst.

Question taken from: https://www.geeksforgeeks.org/practice-problems-on-hashing/ - question 6

Write down all the function calls appropriately to accomplish the following in order:

- Open a binary file named “prefinalexam.bin” using the internal/logical file int_file;
- Store the contents of the 5th – 9th question from the opened file to the last 5 elements of the *prefinals* array one record at a time;
- Append to the file the first three questions found in the *prefinals* array one record at a time. No content should be overwritten.
- Store in the *totalQuestions* variable the total number of item records found in the file.

Data structure definition and variable declarations:

```
struct question{
    char question[256];
    char options[5][62];  /*[option_index][option_text]*/
    int correctAnswerNdx; /*stores index of the correct
                            answer -> 0 to 4*/
    int points;
}prefinals[10];

int totalQuestions;
```

```
FILE* int_file;
int_file = fopen(“prefinalexam.bin”, “rb+”);                /* or ab+ (wb+ will not work) */
fseek(int_file, 4*sizeof(struct question), SEEK_SET);       /* 5th question is at “index 4” of file */
fread(&prefinals[5], sizeof(struct question), 5, int_file); /* reading one record at a time */
  /* or prefinals+5 */                                      /* (recall fread returns actual number of
                                                              items successfully read (? of 5) */

fseek(int_file, 0, SEEK_END);
fwrite(prefinals, sizeof(struct question), 3, int_file);
totalQuestions = ftell(int_file) / sizeof(struct question); /* output: 13 */
```

A dictionary is implemented using an array with 10 spaces (indices 0-9). The location of the items to be inserted will be determined by a hash function. Construct an appropriate hash function for each of the following sets of items in such a way **no collisions** will be occurred. Note that the hash values must always refer to a valid array index.

1.	{112, 405, 298, 157, 210, 934, 136, 509} => hash1(elem)
2.	{1111, 2211, 3311, 4411, 5511, 6611, 7711, 8811} => hash2(elem)
3.	{20100128, 19200128, 18100128, 20200128, 18200128} => hash3(elem)
4.	{11.5, 12.0, 12.5, 13.0, 13.5, 14.0} => hash4(elem)
5.	{0.18 ,0.33, 1.28, 1.46, 2.66, 2.70, 3.09} => hash5(elem)
6.	{“a”, “ba”, “cry”, “doge”, “etlog”, ”forest”} => hash6(elem)
7.	{“www.exam.net”, “www.facebook.com”, “www.google.com”, “www.cisco.com”, “www.instagram.com”} => hash7(elem)
8.	{“17379”, “263681”, “1635”, “420”, “01735”, “09126549712”, “77777777”} => hash8(elem)
9.	{101001100, 11001, 1, 1011011, 11000000, 1110111} => hash9(elem)

```
int hash1(int elem){
   return elem % 10;
}

int hash2(int elem){
   return elem / 100 % 10;
}

int hash3(int elem){
   elem /= 100000;
   return (elem / 10 – 18) * 2 + (elem – 1);
}
// 180 -> 0, 181 -> 1, 190 -> 2, 191 -> 3, …

int hash4(float elem){
   return (int)(elem – 11) * 2;
}

int hash5(float elem){
   return (int)(elem * 10) % 10;
}

int hash6(char elem[]){
   return (elem[0] – ‘a’) % 10;
}
```

```
int hash7(char elem[]){
   return (elem[4] – ‘a’) % 10;
}

int hash8(char elem[]){
   int temp = 0, x;
   for(x=0; elem[x]!=‘\0’; x++){
      elem[x] == ‘0’ ? temp++ : temp--;
   }
   return temp;
}
//number of nonzero digits minus number of zero
digits

int hash9(int elem){
   int temp = 0;
   while (elem > 0){
      if(elem % 10 == 1){
         temp++;
      }
      elem /= 10;
   }
   return temp;
}
//number of occurrences of 1’s in given number.
```

[15] Consider an array with the following values: **7, 24, 18, 52, 36, 54, 11, 23**.

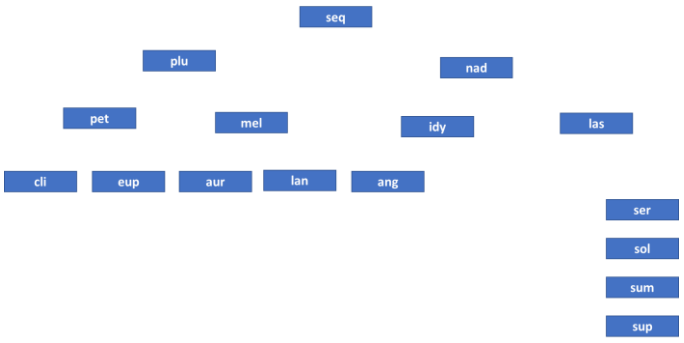
Insert the given values in order into a hash table with 9 memory locations where collisions are resolved using chaining/open hashing. Use the hash function ***hash(k) = k % 9.***	
1.	At what index is the longest chain located on? **0**
2.	List all values between 100 and 110 inclusive that will give the worst time possible for it to be inserted into the hash table. **108**
3.	List all values between 100 and 110 inclusive that will give the best time possible for it to be inserted into the hash table. **100, 102, 103, 107**
Insert the given values in order into a hash table with 9 memory locations where collisions are resolved using open addressing via linear probing. Use the hash function ***hash(k, i) = (k + 9 - i) % 9*** where ***i*** is the probe number starting from 0 which increases by 1 up to 8 for every collision/failed insertion attempt for the given key.	
4.	At what index is the remaining empty slot located? **1**
5.	List all values between 90 and 100 inclusive that will give the worst time possible for it to be inserted into the hash table. **90 and 99 (both required the highest number of probes)**
6.	List down the values in the hash table that are not placed in the immediate location returned by the hash function. **52, 36, 54, 23**
7.	Elements 7, 52, and 54 are to be removed from the hash table. How many probes/array accesses are required to complete the process for searching the location of key 54, applying the same hash function as above? **9 (Searching does not stop at spaces marked DELETED)**

A max heap is implemented whose root is located at index 0.	
1.	The right child of index 26 is at index ___. **54**
2.	The left child of index 37 is at index ___. **75**
3.	The parent of index 74 is at index ___. **36**
4.	At most how many swaps within the heap will occur if a new element is inserted at index 42 and the rest of the elements before that are occupied and form a heap? **5 (42-20-9-4-1-0)**
5.	Within indices 41-60 of the heap, the element at index 5 must be always larger than or equal to the elements in what indices? **47-54 (5 must be greater than 11-12, which must be greater than 23-26, which must be greater than 47-54)**
A min heap is implemented whose root is located at index 1.	
6.	The right child of index 9 is at index ___. **19**
7.	The left child of index 12 is at index ___. **24**
8.	The ancestors of index 30 excluding itself are indices ____. **15, 7, 3, 1**
9.	Which of the following is/are FALSE? **None** A. The element at index 10 must be less than or equal to that at index 80. **(80-40-20-10)** B. The element at index 33 must be greater than that at index 8. **(33-16-8)** C. The element at index 10 must be less than or equal to that at index 87. **(87-43-21-10)** D. The element at index 1 must be less than or equal to that at index 2^n for all positive integers n. **(TRUE – 1-2-4-8-16-…)**

Consider the following elements in the set: {serendipity, petrichor, supine, solitude, aurora, idyllic, clinomania, pluviophile, euphoria, sequoia, sumptuous, angst, nadir, lassitude, languor, mellifluous}.

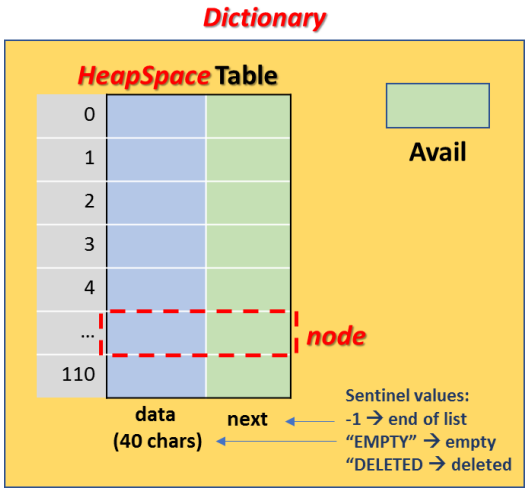
Insert them in order into an initially empty Partially Ordered Tree (max heap) with the rule **apple < zoo**.

Questions:	
1.	What is the root node? **supine (largest element)**
2.	What is the right child of idyllic? **None**
3.	What is the left child of sumptuous? **pluviophile**
4.	On the n^{th} deletion, the node lassitude is to be removed. What is n? **10 (10th largest)**
5.	At what level is the node clinomania located after all elements are inserted assuming the level of root is 1? **4**
6.	At what index will the node idyllic be after all elements are inserted assuming the root is at index 0? **12**
7.	When the element nadir is inserted, how many swaps are performed for it to be in its proper position? **1**
8.	Identify the leaves of the POT. **mel, eup, aur, seq, ang, idy, cli, lan**
9.	Using the heapsort algorithm to sort the elements, list down the order of the elements after four (4) deleteMax() operations. **(see figure)**



(9)

A modified hash table has the following properties: The first 99 nodes contain the prime data area while the remaining elements are reserved for the insertion of synonyms as a resolution to collisions. Each node contains the data and the link field to an index in the synonym area if it exists and -1 If otherwise. Refer to the illustration below.



1. Define the given data structure accurately. Use the datatypes (red) and the variables (black). **[5]**
2. Write the code for deleteElem() which deletes a given element if it exists in the given hash table. The function will invoke function *hash()* which will return an integer corresponding to the index of the beginning search location of a given element within the table. Ensure that all fields in the structure are appropriately adjusted thus avoiding any memory leak. **[10]**

```
#define SIZE 111

typedef struct {
  char data[40];
  int next;
} HeapSpace;

typedef struct {
  HeapSpace Table[SIZE];
  int Avail;
} Dictionary;



void deleteElem(Dictionary *D, char elem[]){
  int ndx = hash(elem);
  int *trav, temp;

  for(trav = &ndx;
     *trav != -1 && strcmp(D->Table[*trav].data, elem) != 0 && strcmp(D->Table[*trav].data, "EMPTY") != 0;
      trav = &D->Table[*trav].next){}

  /*check if item is found*/
  if(*trav != -1 && strcmp(D->Table[*trav].data, elem) != 0){
     strcpy(D->Table[*trav].data, "DELETED");

     /* if element deleted is in synonym area, do cursor-based manipulation
     if(*trav >= 99){
       temp = *trav;
       *trav = D->Table[temp].next;
       D->Table[temp].next = D->Avail;
       D->Avail = temp;
     }
  }
}
```

[15] Write an efficient function that updates the isMaxHeap field accordingly whether the given array represents a max heap or not. Use the definition below and refer to the indicated comments:

```
#define SIZE 51
typedef enum {TRUE, FALSE} Boolean;
typedef struct {
    int elem[SIZE]; /*index 0 stores the number of elements in the array, root is at index 1*/
    Boolean isMaxHeap;
} ElemList;
```

```
void checkMaxHeap(ElemList *EL){
  Boolean isHeap;
  int x, lastNdx, LC, RC, BC, LLP;   /* x – parent, LC and RC – left and right children,
                                        BC – bigger children, LLP – lowest level parent*/
  isHeap = TRUE;
  lastNdx = EL->elem[0];
  LLP = lastNdx/2;

  for(x=1; x<=LLP && isHeap==TRUE; x++){
     LC = x * 2;
     RC = x * 2 + 1;

     /* searching for bigger child */
     if(RC > lastNdx){
        BC = LC;
     } else {
        BC = EL->elem[LC] > EL->elem[RC] ? LC : RC;
     }

     /* compare current parent with larger child */
     if(EL->elem[x] < EL->elem[BC]){
        isHeap = FALSE;
     }
  }

  /* updating the result */
  EL->isMaxHeap = isHeap;

}
```

Write a function to solve the following problem with a running time as efficient as possible:

Jesse loves cookies and wants the sweetness of some cookies to be greater than value k. To do this, two cookies with the least sweetness are repeatedly mixed. This creates a special combined cookie with:

sweetness $= (1\times$ Least sweet cookie $+ 2\times$ 2nd least sweet cookie).

This occurs until all the cookies have a sweetness $\geq k$.

Given the **number of cookies, their sweetness represented through an array, and the threshold value k to be met** , determine the minimum number of operations required. If it is not possible, return -1.

Example

$k = 9$

$A = [2, 7, 3, 6, 4, 6]$

The smallest values are $2, 3$.

Remove them then return $2 + 2 \times 3 = 8$ to the array. Now $A = [8, 7, 6, 4, 6]$.

Remove $4, 6$ and return $4 + 6 \times 2 = 16$ to the array. Now $A = [16, 8, 7, 6]$.

Remove $6, 7$, return $6 + 2 \times 7 = 20$ and $A = [20, 16, 8, 7]$.

Finally, remove $8, 7$ and return $7 + 2 \times 8 = 23$ to A. Now $A = [23, 20, 16]$.

All values are $\geq k = 9$ so the process stops after 4 iterations. Return 4.

Logic:

Make the array a min heap (do not sort)

Repeat until either array size reaches 1 or the minimum element is >= threshold:

DeleteMin() twice, and add the two values deleted.

Insert the newly added value back to the heap. Perform necessary adjustments

Increment the number of iterations n.

Decrease array size by 1.

Return (minimum element >= threshold) ? n : -1

n can be also (original array size – new array size)