CS29002 Algorithms Laboratory

Assignment No: 5

Last date of submission: 24–August–2016

Let us consider hash-based storage of multi-dimensional data. Specifically, each item we want to store in our hash table T is a pair (x,y) of two strings. We call each component (x or y) an attribute of the data item. Viewed as an abstract data type (ADT), we want the following operations on the *hashtable* ADT.

Table 1: ADT functions for attribute-based hash tables *hashtable*

inittable(S)	Return an empty hash table with S slots.
searchpair$(T,(x,y))$	Check and print whether (x,y) is stored in T. No return value.
searchattr(T,z,a)	Attribute-based search. No return value. If $a=1$, print all the strings y such that (z,y) is stored in T. If $a=2$, print all the strings x such that (x,z) is stored in T.
insertpair$(T,(x,y))$	Insert the pair (x,y) in T, and return the updated table.
deletepair$(T,(x,y))$	Delete the pair (x,y) from T, and return the updated table.

Assume that each attribute is of length $\leqslant 20$. Use the following data type to store an attribute pair (x,y).

```
typedef struct {
   char x[21], y[21];   /* Pair of NULL-terminated strings */
} pair;
```

Pass such a pair as the second argument of *searchpair*, *insertpair*, and *deletepair*. The second argument of *searchattr* is a single string.

Write two different programs *openaddr.c*(pp) and *chaining.c*(pp) for two implementations of the *hashtable* ADT. The first program is based on open addressing with linear probing, and the second on chaining. The prototypes of the ADT functions must exactly match those given in Table 1. You may, however, use any number of helping functions with prototypes suited to your personal convenience.

Program 1: *openaddr.c*

We need to hash strings to array indices. Let S denote the size of the arrays we use in our hash tables. The hash $H_S(z)$ of a string $z = a_0a_1a_2\ldots a_{l-1}$ of length l is an integer in the range $0,1,2,\ldots,S-1$. We assume that z is a string of standard 7-bit ASCII characters. We treat z as an integer in base $2^7 = 128$, where each character a_i is identified with its ASCII value. The remainder of this number upon division by S is the hash $H_S(z)$ of z. The hash of the empty string (the string of length zero) is zero. For $l \geqslant 1$, we have

$$H_S(a_0a_1a_2\ldots a_{l-1}) = 128 \times H_S(a_0a_1a_2\ldots a_{l-2}) + ASCII(a_{l-1}) \pmod S.$$

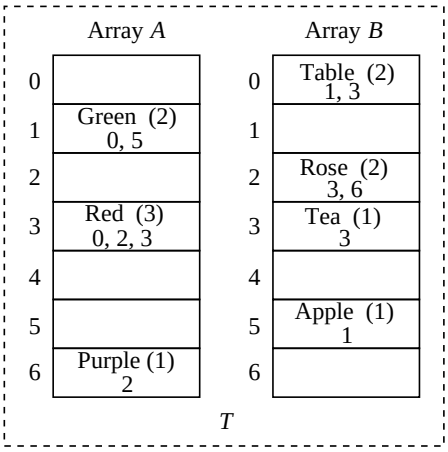
The number corresponding to z may be too large to fit 32-bit integers, so this number is not explicitly computed. Instead, compute the hash of z iteratively with all intermediate results reduced modulo S. Pass both z and S as parameters to the hash function.

The hash table T consists of two arrays A and B. A stores the first attribute x, and B stores the second attribute y of a data item (x,y). Independent storage of the two attributes introduces a problem: the association between the two attributes of a data item is lost. This means that if (x_1,y_1) and (x_2,y_2) are two items inserted in T, the table would confirm the presence of the items (x_1,y_2) and (x_2,y_1) which are not inserted in T. In order to overcome this problem, we maintain an association array against each attribute. Let $(x,y_1),(x,y_2),\ldots,(x,y_k)$ be all the items inserted in T with first attribute x. Suppose that x is inserted at index (slot) i in A, and $y_1,y_2,\ldots,y_k$ at indices $j_1,j_2,\ldots,j_k$ in B. The index $A[i]$ stores the indices $j_1,j_2,\ldots,j_k$ along with x. Likewise, if $(x_1,y),(x_2,y),\ldots,(x_l,y)$ are inserted in T for a fixed y with $x_1,x_2,\ldots,x_l$ inserted at indices $i_1,i_2,\ldots,i_l$ in A and y inserted at index j in B, then $B[j]$ stores y along with the indices $i_1,i_2,\ldots,i_l$. For simplicity, we assume that an attribute can be associated with at most ten attributes of the other type. Therefore the following data type can be used for each slot in the arrays.

```
typedef struct {
   char att[21];   /* The attribute name: a NULL-terminated string */
   int nassoc;     /* Number of attribute associations in the other table */
   int assoc[10];  /* Array of indices of the associated attributes */
} htnode;
```

An attribute-based hash table storing the six pairs (Green, Table), (Green, Apple), (Red, Table), (Red, Rose), (Red, Tea), and (Purple, Rose) is shown in Figure 1. The **nassoc** field appears within parentheses. Suppose that the strings Rose and Tea both hash to the same index 2. Let us use linear probing, that is, the string Tea is relocated to index 3. The association indices are the actual indices where the strings reside, not the default hash values.

Figure 1: Attribute-based hash table with open addressing and linear probing



Let S denote the size of the two arrays A and B of **htnode** structures. Assume that S is large enough to store all the data items that the user would insert. Therefore you do not need to check whether the table is full, or to expand the table. S is however needed for hash and index computations. You may use the following data type for the *hashtable* ADT.

```
typedef struct {
   int S;          /* Size of A and B */
   htnode *A, *B;  /* Attribute arrays */
} hashtable;
```

Searching, insertion and deletion algorithms are straightforward. Use linear probing in both A and B to resolve collisions. Appropriately maintain the association lists in both the arrays. There is no necessity to keep these lists sorted. Make linear searches in the lists for searching, insertion and deletion. An attribute-based search prints all the strings stored in the other array at the indices found in the association list.

Use a special string (like EMPTY) to indicate an empty slot in the arrays. Use another special string (like DELETED) to designate a slot made empty after deletion. Assume that the strings EMPTY and DELETED are not valid attributes in any data item. Also note that you delete a string from an array only when its association count drops to zero. For example, deletion of only (Red, Rose) and (Red, Table) in the example of Figure 1 still leaves the pair (Red, Tea) in the table, and so Red should not be deleted yet from A.

The *main*() function starts by reading S from the user and initializing T to an empty hash table. It then enters a loop, asking the user what to do. The user first enters an integer with the interpretation that 0 means break the loop, 1 means search on the first attribute, 2 means search on the second attribute, 3 means a search for a pair (x, y), 4 means insertion of a pair (x, y) in T, and 5 means deletion of a pair (x, y) from T. Depending upon the choice of the user, one or two strings (x and/or y) is/are additionally entered by the user. The ADT calls are made to print the results and other diagnostic messages (see the sample output).

The only functions called by your *main*() function must be the ADT functions. Your helping functions may be called by the ADT functions, but not directly by the *main*() function.

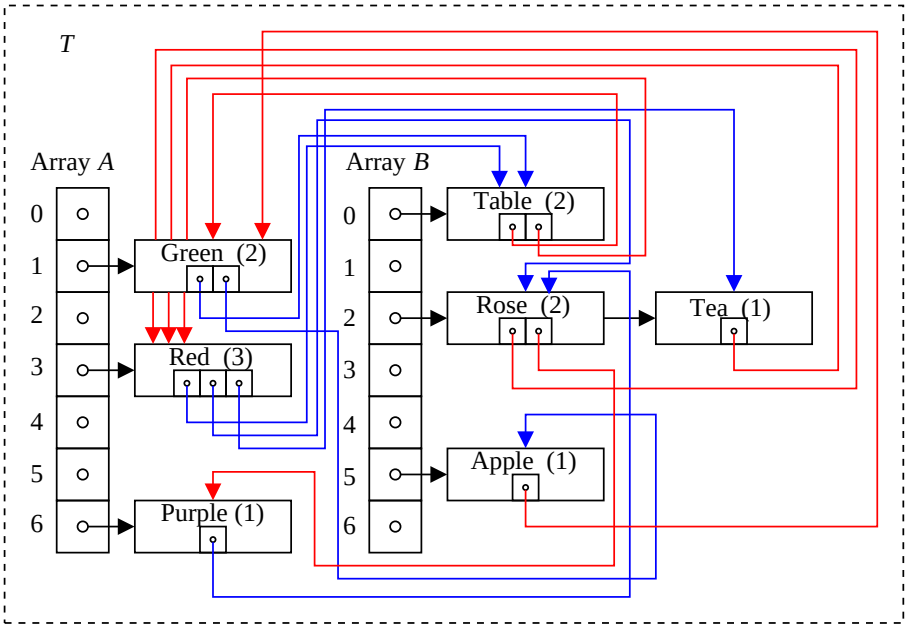
Sample output

```
S = 11
+++ Enter choice: 4 Green Apple
    hash = (3,10), insertion at (3,10)
+++ Enter choice: 4 Yellow Pen
    hash = (7,7), insertion at (7,7)
+++ Enter choice: 4 Red Shirt
    hash = (7,4), insertion at (8,4)
+++ Enter choice: 4 Brown Cake
    hash = (10,6), insertion at (10,6)
+++ Enter choice: 4 Red Table
    hash = (7,0), insertion at (8,0)
+++ Enter choice: 4 Blue Sky
    hash = (8,9), insertion at (9,9)
+++ Enter choice: 4 Black Dog
    hash = (5,10), insertion at (5,1)
+++ Enter choice: 4 Green Tea
    hash = (3,3), insertion at (3,3)
+++ Enter choice: 4 Black Sea
    hash = (5,9), insertion at (5,2)
+++ Enter choice: 4 Blue Sea
    hash = (8,9), insertion at (9,2)
+++ Enter choice: 4 Yellow Shirt
    hash = (7,4), insertion at (7,4)
+++ Enter choice: 4 Brown Table
    hash = (10,0), insertion at (10,0)
+++ Enter choice: 4 Black Shirt
    hash = (5,4), insertion at (5,4)
+++ Enter choice: 4 Yellow Table
    hash = (7,0), insertion at (7,0)
+++ Enter choice: 4 Purple Sky
    hash = (2,9), insertion at (2,9)
+++ Enter choice: 4 Gray Shirt
    hash = (5,4), insertion at (6,4)
+++ Enter choice: 4 Red Pen
    hash = (7,7), insertion at (8,7)
+++ Enter choice: 1 Yellow
    (Yellow,-) is associated with: Pen Shirt Table
+++ Enter choice: 1 White
    (White,-) is associated with:
+++ Enter choice: 1 Shirt
    (Shirt,-) is associated with:
+++ Enter choice: 2 Shirt
    (-,Shirt) is associated with: Red Yellow Black Gray
+++ Enter choice: 2 Jeans
    (-,Jeans) is associated with:
+++ Enter choice: 2 Red
    (-,Red) is associated with:
+++ Enter choice: 3 Black Dog
    (Black,Dog) found at (5,1)
+++ Enter choice: 3 Black Cat
    (Black,Cat) not found
+++ Enter choice: 2 Sky
    (-,Sky) is associated with: Blue Purple
+++ Enter choice: 5 Purple Sky
    hash = (2,9), deletion at (2,9)
+++ Enter choice: 2 Sky
    (-,Sky) is associated with: Blue
+++ Enter choice: 5 Blue Sky
    hash = (8,9), deletion at (9,9)
+++ Enter choice: 2 Sky
    (-,Sky) is associated with:
+++ Enter choice: 3 Green Apple
    (Green,Apple) found at (3,10)
+++ Enter choice: 5 Green Apple
    hash = (3,10), deletion at (3,10)
+++ Enter choice: 3 Green Apple
    (Green,Apple) not found
+++ Enter choice: 5 Black Sea
    hash = (5,9), deletion at (5,2)
+++ Enter choice: 2 Sea
    (-,Sea) is associated with: Blue
+++ Enter choice: 0
```

Program 2: *chaining.c*

In the chaining mode, multiple attributes hashing to the same index are inserted at the same slot. Therefore maintaining the array indices of associated attributes is not guaranteed to resolve all ambiguities. We instead use node pointers to point to associated nodes in the other array. Figure 2 illustrates this idea.

Figure 2: Attribute-based hash table with chaining



In the chaining mode, you maintain a linked list of nodes inserted at each slot (index). So the data type **htnode** should have a self-referencing pointer. Moreover, you need to replace the association array of indices by an array of node pointers. Finally, the hash table T consists of two arrays A and B of pointers to modified **htnode** structures, each of size S.

```
typedef struct _htnode {
   char att[21];               /* The attribute name: a NULL-terminated string */
   int nassoc;                 /* Number of attribute associations */
   struct _htnode *assoc[10]; /* Array of pointers to associated attributes */
   struct _htnode *next;       /* Next pointer for maintaining the linked list */
} htnode;

typedef struct {
   int S;                      /* Size of A and B */
   htnode **A, **B;            /* Arrays of pointers to head lists of nodes */
} hashtable;
```

A pair (x,y) with $i = H_S(x)$ and $j = H_S(y)$ resides in the table if and only if all of the following four conditions hold: (1) x is stored in the linked list headed by $A[i]$, (2) y is stored in the linked list headed by $B[j]$, (3) a pointer to the node storing y appears in the association list of x, and (4) a pointer to the node storing x appears in the association list of y. You do not need to keep the linked lists and the association lists sorted. Make linear searches in them. Delete a node only when its association count drops to zero. The special strings EMPTY and DELETED are not needed in this implementation.

In order to hash strings, this program *chaining.c* should use the same function $H_S(z)$ introduced in connection with *openaddr.c*. The *main*() function written for *openaddr.c* must work for *chaining.c* without any changes. Note once again that your *main*() function is allowed to make calls of only the ADT functions, but not of any helping function that you may implement.

Sample output

```
S = 11
```

```
+++ Enter choice: 4 Green Apple
    Insertion at (3,10)
+++ Enter choice: 4 Yellow Pen
    Insertion at (7,7)
+++ Enter choice: 4 Red Shirt
    Insertion at (7,4)
+++ Enter choice: 4 Brown Cake
    Insertion at (10,6)
+++ Enter choice: 4 Red Table
    Insertion at (7,0)
+++ Enter choice: 4 Blue Sky
    Insertion at (8,9)
+++ Enter choice: 4 Black Dog
    Insertion at (5,10)
+++ Enter choice: 4 Green Tea
    Insertion at (3,3)
+++ Enter choice: 4 Black Sea
    Insertion at (5,9)
+++ Enter choice: 4 Blue Sea
    Insertion at (8,9)
+++ Enter choice: 4 Yellow Shirt
    Insertion at (7,4)
+++ Enter choice: 4 Brown Table
    Insertion at (10,0)
+++ Enter choice: 4 Black Shirt
    Insertion at (5,4)
+++ Enter choice: 4 Yellow Table
    Insertion at (7,0)
+++ Enter choice: 4 Purple Sky
    Insertion at (2,9)
+++ Enter choice: 4 Gray Shirt
    Insertion at (5,4)
+++ Enter choice: 4 Red Pen
    Insertion at (7,7)
+++ Enter choice: 1 Yellow
    (Yellow,-) is associated with: Pen Shirt Table
+++ Enter choice: 1 White
    (White,-) is associated with:
+++ Enter choice: 1 Shirt
    (Shirt,-) is associated with:
+++ Enter choice: 2 Shirt
    (-,Shirt) is associated with: Red Yellow Black Gray
+++ Enter choice: 2 Jeans
    (-,Jeans) is associated with:
+++ Enter choice: 2 Red
    (-,Red) is associated with:
+++ Enter choice: 3 Black Dog
    (Black,Dog) found at (5,10)
+++ Enter choice: 3 Black Cat
    (Black,Cat) not found
+++ Enter choice: 2 Sky
    (-,Sky) is associated with: Blue Purple
+++ Enter choice: 5 Purple Sky
    Deletion at (2,9)
+++ Enter choice: 2 Sky
    (-,Sky) is associated with: Blue
+++ Enter choice: 5 Blue Sky
    Deletion at (8,9)
+++ Enter choice: 2 Sky
    (-,Sky) is associated with:
+++ Enter choice: 3 Green Apple
    (Green,Apple) found at (3,10)
+++ Enter choice: 5 Green Apple
    Deletion at (3,10)
+++ Enter choice: 3 Green Apple
    (Green,Apple) not found
+++ Enter choice: 5 Black Sea
    Deletion at (5,9)
+++ Enter choice: 2 Sea
    (-,Sea) is associated with: Blue
+++ Enter choice: 0
```

Submit two C/C++ source files *openaddr.c*(*pp*) and *chaining.c*(*pp*).
Do not use global/static variables. Do not make C++ STL calls.