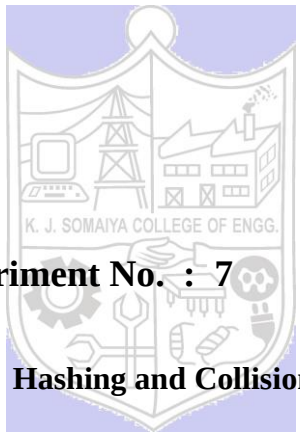


Experiment No. : 7

Title: Hashing and Collision Resolution

Batch: B1 Roll No.: 1914078 Experiment No.: 7

Aim: Implementing a program for Hashing with following functions. To resolve the collision implement Linear Probing and Quadratic Probing.

1. insert()
2. delete()
3. search()
4. display()

Resources needed: C/C++/JAVA/Python editor and compiler.

Theory

Hashing

Hashing is the process of mapping large amount of data item to a smaller table with the help of a **hashing function**. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e.collection size).
Due to this hashing process, the result is a **Hash data structure** that can store or retrieve data items in an average time disregard to the collection size.

Hash Table is the result of storing the hash data structure in a smaller table which incorporates the hash function within itself. The ***Hash Function*** primarily is responsible to map between the original data item and the smaller table itself. Here the mapping takes place with the help of an output integer in a consistent range produced when a given data item (any data type) is provided for storage and this output integer range determines the location in the smaller table for the data item. In terms of implementation, the hash table is constructed with the help of an array and the indices of this array are associated to the output integer range.

Hash Table Example :

Here, we construct a hash table for storing and retrieving data related to the citizens of a county and the social-security number of citizens are used as the indices of the array implementation (i.e. key). Let's assume that the table size is 12, therefore the hash function would be Value modulus of 12.

Hence, the Hash Function would equate to:

(sum of numeric values of the characters in the data item) %12

Let us consider the following social-security numbers and produce a hashcode:

120388113D => 1+2+0+3+8+8+1+1+3+13=40 Hence, (40)%12 => Hashcode=4

310181312E => 3+1+0+1+8+1+3+1+2+14=34 Hence, (34)%12 => Hashcode=10
041176438A => 0+4+1+1+7+6+4+3+8+10=44 Hence, (44)%12 => Hashcode=8

Therefore, the Hashtable content would be as follows:

```
-----------------------------------------------------
00: empty
01: empty
02: empty
03: empty
04: occupied Name: REMUS SSN:120388113D
05: empty
06: empty
07: empty
08: occupied Name: JAMES SSN:041176438A
09: empty
10: occupied Name: PETER SSN:310181312E
11: empty
-----------------------------------------------------
```


Collision :

A situation when the resultant hashes for two or more data elements in the data set *U*, maps to the same location in the hash table, is called a hash collision. In such a situation two or more data elements would qualify to be stored/mapped to the same location in the hash table.
E.g. For TableSize = 17, if *Value mod 17* is the **HashFunction** used for mapping then the keys 18 and 35 hash to the same value 18 mod 17 = 1 and 35 mod 17 = 1 i.e. Collision occurs.
There are several methods for dealing with this:

- **Separate chaining**
- **Open addressing**
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Separate Chaining (Open hashing):

The idea is to keep a list of all elements that hash to the same value.

- The array elements are pointers to the first nodes of the lists. – A new item is inserted to the front of the list.

Advantages:

- Better space utilization for large items.
- Simple collision handling: searching linked list.
- Overflow: we can store more items than the hash table size. – Deletion is quick and easy: deletion from the linked list.

Example for implementing separate chaining.

Consider the values to be inserted are

0, 1, 4, 9, 16, 25, 36, 49, 64, 81

AND **Hash Function** is

```
hash(value) = value % 10
```

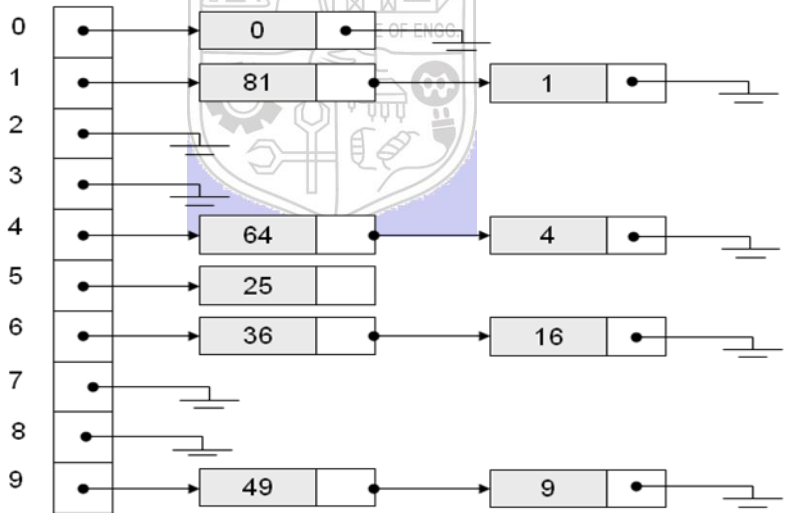


Figure 7.1 Example of separate chaining (Open Hashing)

Open addressing (Closed hashing):

Separate chaining has the disadvantage of using linked lists.

- Requires the implementation of a second data structure.

In an open addressing hashing system, all the data go inside the table.

- Thus, a bigger table is needed.

– If a collision occurs, alternative cells are tried until an empty cell is found.

There are three common collision resolution strategies:

– Linear Probing
– Quadratic probing
– Double hashing

Linear Probing:

In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.

– i.e. *f* is a linear function of *i*, typically *f(i)= i*.

Example:

– Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
– Table size is 10.
– Hash function is hash(x) = x mod 10.
  - *f(i) = i*;

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 7.2 Example of Linear Probing

Quadratic Probing:

Quadratic Probing eliminates primary clustering problem of linear probing.

Collision function is quadratic.

– The popular choice is $f(i) = i^2$.

If the hash function evaluates to h and a search in cell h is inconclusive, we try cells h + 1^2, h+2^2, … h + i^2.

– i.e. It examines cells 1,4,9 and so on away from the original probe.

Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

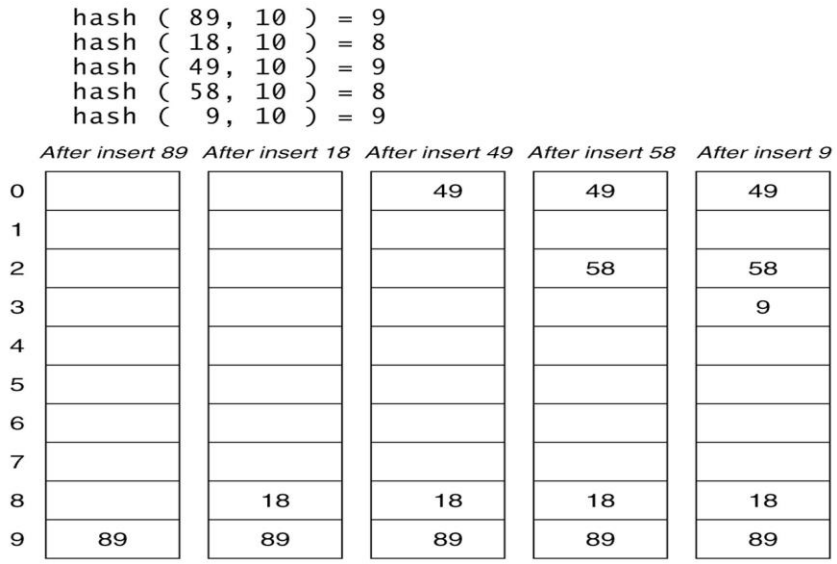


Figure 7.3 Example of Quadratic Probing

In this example of Figure 7.2 the table is poorly chosen as table size is not a prime number

Double hashing:
A second hash function is used to drive the collision resolution.

– $f(i) = i * hash_2(x)$

We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.

Algorithm :
We first create a structure 'hash' in which we create an array of size 100 and create an integer variable 'ct' which keeps a count of the elements in the Hash Table. struct hash

```
{
   int arr[100];
int ct;
};
```

1. **createHashTable(int N) –** creates a new hash table of size N and initializes it.
   - SET struct hashTable t;
   - FOR (int i=0;i<size;i++)

     t.arr[i]=-1; #Initialising hash table

   [END OF FOR LOOP]
   - SET t.ct=0;

- RETURN t;

2. **insert(typedef x)** – this void function inserts the element x on the hash table using appropriate collision resolution method. IF t->ct!=size IF type==1 int set=0;

```
int key=val%size
DO
  IF t->arr[key] == -1
t->arr[key] = val;
    PRINT "%d inserted at position-%d\n", val,key
    set=1
  [END OF IF]
  ELSE
    PRINT "Collision Detected."
    SET key=(key+1)%size
    PRINT “Trying insert at next index.”
  [END OF ELSE]
 }WHILE(set==0);
[END OF DO-WHILE LOOP]
ELSE
  int set=0
  int key=val%size
  FOR int
i=0;set==0;i++ IF
i==size break;
  IF t->arr[key] == -1
    SET t->arr[key] = val
```

PRINT %d inserted at position-%d”

set=1

[END OF IF]

ELSE

SET key=val%sizE

PRINT “Collision Detected”

SET key=(key+(i*i))%size

PRINT "Trying insert at next index %d”

[END OF ELSE]

[END OF FOR LOOP]

SET t->ct++;

[END OF ELSE] ELSE printf("The Hash Table is full cannot insert anymore elements.\n");

[END OF IF-ELSE]

---

3. **typedef delete(typedef x)** – this function searches for the element x on the hash table and removes it and returns it if found; otherwise returns -1.

INT key=search(t,val,type,size); #First we search the element in the hash tabkle.

IF key!=-1

SET t->arr[key]=-1;

SET t->ct--;

[END OF IF[

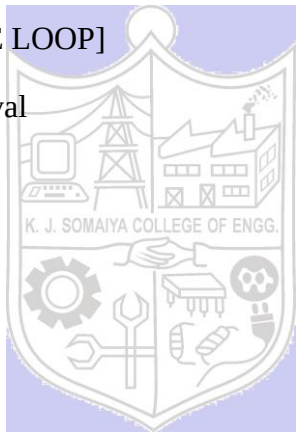
RTEURN key;

---

4. **search(typedef x)** – this boolean function searches for an element and returns 1 if found; otherwise returns 0.

IF isempty(t)==0

```
{       int
cnt=0;
    IF type==1
     {
       int key=val%size;
       WHILE t->arr[key]!=val
         SET key=(key+1)%size;
         SET cnt++;
IF cnt==size
break;
       [END OF WHILE LOOP]
        IF t->arr[key]==val
          return key;
       ELSE
          RTEURN  -1;
    [END OF IF]
    ELSE
    {
       SET int i=0;
       SET int key=val%size;
       WHILE t->arr[key]!=vaL
         SET key=val%size;
         SET key=(key+(i*i))%size
         SET  i++
         SET cnt++
IF cnt==size-1
```



```
            break;
        [END OF WHILE LOOP]
        IF t->arr[key]==val
            return
key         ELSE
return -1
    [END OF IF]
  [END OF IF]
  ELSE
    PRINT "The Hash Table is empty."
    RETURN -1
  [END OF ELSE]
```

5. **display()** – this void function displays all the elements present on the hash table if not empty; otherwise shows empty exception.

```
INT i;
FOR i = 0;i<size; i++
    IF t->arr[i]!=-1
        PRINT arr[%d] = %d\n",i,t->arr[i])
    ELSE
        PRINT "arr[%d] = NULL”
  [END OF FOR LOOP]
```

6. **isempty()** – this is a Boolean functions returns 1 if hashtable is empty; otherwise returns 0;

```
IF t->ct==0
    RETURN 1;
```

ELSE

RETURN 0;

---

7. **size()** – this function returns number of elements present on the hashtable. IF isempty(t)==0

RETURN t->ct; ELSE

printf("The Hash Table is empty.\n");

RETURN 0;

[END OF ELSE]

Activity : Implements the resolution methods as mentioned in the aim and show the number of collisions in each case.

Program:

```
#include <stdio.h>

struct hashTable
{
    int arr[100];
    int ct;
};

struct hashTable createHashTable(int size)
{
    struct hashTable t;
    for (int i = 0; i < size; i++)
    {
        t.arr[i] = -1;
    }
    t.ct = 0;
    return t;
}

void insert(struct hashTable *t, int val, int type, int size)
{
    if (t->ct != size)
    {
        if (type == 1)
```

```
        {
            int count = 0;
            int key = val % size;
            do
            {
                if (t->arr[key] == -1)
                {
                    t->arr[key] = val;
                    printf("%d inserted at the position - %d\n", val, key);
                    count = 1;
                }
                else
                {
                    printf("Collision Detected!");
                    key = (key + 1) % size;
                    printf("Will Try to insert at next index %d\n", key);
                }
            } while (count == 0);
        }
        else
        {
            int count = 0;
            int key = val % size;
            for (int i = 0; count == 0; i++)
            {
                if (i == size)
                    break;
                if (t->arr[key] == -1)
                {
                    t->arr[key] = val;
                    printf("%d inserted at the position - %d\n", val, key);
                    count = 1;
                }
                else
                {
                    key = val % size;
                    printf("Collision Detected!");
                    key = (key + (i * i)) % size;
                    printf("Trying insert at next index %d\n", key);
                }
            }
        }
        t->ct++;
    }
    else
        printf("The Hash Table is full\n");
}
```

```
int isempty(struct hashTable *t)
{
    if (t->ct == 0)
        return 1;
    else
        return 0;
}

int search(struct hashTable *t, int val, int type, int size)
{
    if (isempty(t) == 0)
    {
        int counter = 0;
        if (type == 1)
        {
            int key = val % size;
            while (t->arr[key] != val)
            {
                key = (key + 1) % size;
                counter++;
                if (counter == size)
                    break;
            }
            if (t->arr[key] == val)
                return key;
            else
                return -1;
        }
        else
        {
            int i = 0;
            int key = val % size;
            while (t->arr[key] != val)
            {
                key = val % size;
                key = (key + (i * i)) % size;
                i++;
                counter++;
                if (counter == size - 1)
                    break;
            }
            if (t->arr[key] == val)
                return key;
            else
                return -1;
        }
```

```
    }
    else
    {
        printf("The Hash Table is empty.\n");
        return -1;
    }
}

int deleteElement(struct hashTable *t, int val, int type, int size)
{
    int key = search(t, val, type, size);
    if (key != -1)
    {
        t->arr[key] = -1;
        t->ct--;
    }
    return key;
}

void display(struct hashTable *t, int size)
{
    for (int i = 0; i < size; i++)
    {
        if (t->arr[i] != -1)
            printf("arr[%d] = %d\n", i, t->arr[i]);
        else
            printf("arr[%d] = NULL\n", i);
    }
}

int size(struct hashTable *t)
{
    if (isempty(t) == 0)
        return t->ct;
    else
    {
        printf("The Hash Table is empty.\n");
        return 0;
    }
}

int main()
{
    int choice, key, value, n, c = 1, i, s, sz, type = 1;
    printf("--------- Hash Table -----------\n");
    printf("Enter size of Hash Table:");
    scanf("%d", &s);
```

```
    struct hashTable t;
    t = createHashTable(s);
    printf("\nChoose the type of probing:\n1.Linear\n2.Quadratic\nEnter your choi
ce:");
label:
    scanf("%d", &type);
    if (type > 2)
    {
        printf("Choose from given options only.\nEnter you choice:");
        goto label;
    }
    do
    {
        printf("\n\tMENU \n1.Insert element \n2.Delete element \n3.Search element
 \n4.Size of Table \n5.Display \n6.Exit\nEnter your choice:");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Inserting an element:-\n");
            printf("Enter key:");
            scanf("%d", &key);
            insert(&t, key, type, s);
            break;
        case 2:
            printf("Deleting element:-\nEnter the value to delete:");
            scanf("%d", &value);
            key = deleteElement(&t, value, type, s);
            if (key != -1)
                printf("%d found at position: %d\nElement deleted from table\n",
value, key);
            else
                printf("Entered value does not exist\n");
            break;
        case 3:
            printf("Enter value to be searched:");
            scanf("%d", &value);
            key = search(&t, value, type, s);
            if (key != -1)
                printf("Value %d found at position:%d\n", value, key);
            else
                printf("Entered value does not exist\n");
            break;
        case 4:
            sz = size(&t);
            if (sz != 0)
                printf("Size of Hash Table is: %d\n", sz);
```

```
            break;
        case 5:
            printf("Hash Table is:\n");
            display(&t, s);
            break;
        case 6:
            c = 0;
            break;
        default:
            printf("Choose from the given options only.\n");
        }
    } while (c == 1);
}
```


Output:

Linear Probing

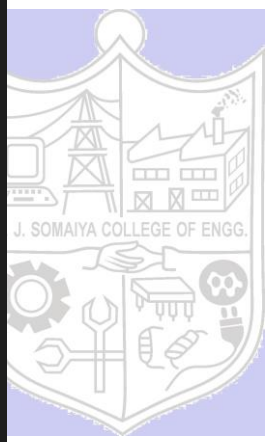
```
--------- Hash Table -----------
Enter size of Hash Table:10

Choose the type of probing:
1.Linear
2.Quadratic
Enter your choice:1

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
```

```
Inserting an element:-
Enter key:16
16 inserted at the position - 6

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:68
68 inserted at the position - 8
```



```
        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:53
53 inserted at the position - 3

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:98
Collision Detected      Will Try to insert at next index 9
98 inserted at the position - 9

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:5
```

```
Hash Table is:
arr[0] = NULL
arr[1] = NULL
arr[2] = NULL
arr[3] = 53
arr[4] = NULL
arr[5] = NULL
arr[6] = 16
arr[7] = NULL
arr[8] = 68
arr[9] = 98

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:4
Size of Hash Table is: 4

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:3
Enter value to be searched:96
Entered value does not exist
```

```
        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:2
Deleting element:-
Enter the value to delete:53
53 found at position: 3
Element deleted from table

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:5
Hash Table is:
arr[0] = NULL
arr[1] = NULL
arr[2] = NULL
arr[3] = NULL
arr[4] = NULL
arr[5] = NULL
arr[6] = 16
arr[7] = NULL
arr[8] = 68
arr[9] = 98
```



```
        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:6
```

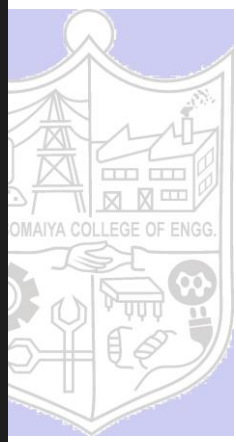
Quadratic Probing:

```
--------- Hash Table -----------
Enter size of Hash Table:10

Choose the type of probing:
1.Linear
2.Quadratic
Enter your choice:2

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:73
73 inserted at the position - 3

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:65
65 inserted at the position - 5
```



```
        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:95
Collision Detected!Trying insert at next index 5
Collision Detected!Trying insert at next index 6
95 inserted at the position - 6

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:1
Inserting an element:-
Enter key:8
8 inserted at the position - 8

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
```

```
Enter your choice:1
Inserting an element:-
Enter key:8
8 inserted at the position - 8

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:5
Hash Table is:
arr[0] = NULL
arr[1] = NULL
arr[2] = NULL
arr[3] = 73
arr[4] = NULL
arr[5] = 65
arr[6] = 95
arr[7] = NULL
arr[8] = 8
arr[9] = NULL

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:4
```

```
Size of Hash Table is: 4

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:3
Enter value to be searched:8
Value 8 found at position:8

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:2
Deleting element:-
Enter the value to delete:53
Entered value does not exist

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
```

```
Enter your choice:2
Deleting element:-
Enter the value to delete:73
73 found at position: 3
Element deleted from table

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:5
Hash Table is:
arr[0] = NULL
arr[1] = NULL
arr[2] = NULL
arr[3] = NULL
arr[4] = NULL
arr[5] = 65
arr[6] = 95
arr[7] = NULL
arr[8] = 8
arr[9] = NULL
```



```
Enter your choice:2
Deleting element:-
Enter the value to delete:73
73 found at position: 3
Element deleted from table

        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:5
Hash Table is:
arr[0] = NULL
arr[1] = NULL
arr[2] = NULL
arr[3] = NULL
arr[4] = NULL
arr[5] = 65
arr[6] = 95
arr[7] = NULL
arr[8] = 8
arr[9] = NULL
```

```
        MENU
1.Insert element
2.Delete element
3.Search element
4.Size of Table
5.Display
6.Exit
Enter your choice:6
```

Outcome:

CO4: Demonstrate different sorting and searching methods.

Conclusion:

Through this experiment we learnt the concept of Hashing and implemented various functions. We also saw how collision occurs and learnt how to resolve these collisions using Linear and Quadratic probing and implemented the same using a C program.

References:

Books/ Journals/ Websites:

- Y. Langsam, M. Augenstin and A. Tannenbaum, “Data Structures using C”, Pearson Education Asia, 1st Edition, 2002.
- Richard F. Gilberg & Behrouz A. Forouzan Data Structures A Pseudocode Approach with C CENGAGE Learning Second Edition.