**LAB SESSION 11: Hashing and Searching**

**AIM**: To implement hashing and searching methods.

**PROBLEM DEFINITION:**

1. Implement linear search and binary search using an employee structure (structure should have id, name and address).
2. Implement hashing using following collision resolution techniques

* Linear Probing
* Quadratic probing
* Double hashing

1. Implement hashing using separate chaining.

**THEORY**:

**Linear search:**

In Linear Search Algorithm,

* Every element is considered as a potential match for the key and checked for the same.
* If any element is found equal to the key, the search is successful and the index of that element is returned.
* If no element is found equal to the key, the search yields “No match found

**Binary search:**

In this algorithm,

* Divide the search space into two halves by [finding the middle index “mid”](https://www.geeksforgeeks.org/problem-binary-search-implementations/).
* Compare the middle element of the search space with the key.
* If the key is found at middle element, the process is terminated.
* If the key is not found at middle element, choose which half will be used as the next search space.
  + If the key is smaller than the middle element, then the left side is used for next search.
  + If the key is larger than the middle element, then the right side is used for next search.
* This process is continued until the key is found or the total search space is exhausted

**Linear probing:**

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.The main disadvantage of linear probing is primary clustering. Clustering increases the number of probes to search or insert a key and hence the search and insertion times of the records also increase.

## ****Quadratic Probing:****

Quadratic probing is an [open-addressing](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/) scheme where we look for the i2‘th slot in the i’th iteration if the given hash value x collides in the hash table.

## **How Quadratic Probing is done?**

Let hash(x) be the slot index computed using the hash function.

* If the slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S.
* If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S.
* If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S.
* This process is repeated for all the values of i until an empty slot is found

**Double hashing:**

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

However, double hashing has a few drawbacks. First, it requires the use of two hash functions, which can increase the computational complexity of the insertion and search operations. Second, it requires a good choice of hash functions to achieve good performance. If the hash functions are not well-designed, the collision rate may still be high.

**Advantages of Double hashing**

* The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
* This technique does not yield any clusters.
* It is one of the effective methods for resolving collisions.

Double hashing can be done using:   
**(hash1(key) + i \* hash2(key)) % TABLE\_SIZE**   
Here hash1() and hash2() are hash functions and TABLE\_SIZE is size of hash table.

## ****Separate Chaining:****

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

The **linked list**data structure is used to implement this technique.When multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven’t found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

**ALGORITHMS**:

1. Linear Search

1. Start from the first element of the collection.

2. Repeat steps 3-5 until the end of the collection is reached.

3. If the current element is equal to the target value, return the index of the current element.

4. Move to the next element in the collection.

5. If the end of the collection is reached and the target value has not been found, return a special value (e.g., -1) to indicate that the element is not present.

2. Binary Search

1. Set low to 0 and high to n-1, where n is the number of elements in the sorted collection.

2. Repeat steps 3-5 while low is less than or equal to high.

3. Calculate mid as the floor of (low + high) / 2.

4. If the element at index mid is equal to the target value, return mid.

5. If the element at index mid is less than the target value, update low to mid + 1.

If the element at index mid is greater than the target value, update high to mid - 1.

6. If low is greater than high, the target value is not present in the collection. Return a special value (e.g., -1) to indicate that the element is not found

3. Linear Probing

1. Hash the key to get the initial position (index) in the hash table.

2. If the position is empty, insert the key at that position.

3. If the position is occupied by another key, perform linear probing to find the next available slot.

- Start probing by incrementing the position by 1.

- If the new position is empty, insert the key at that position.

- If the new position is occupied, continue probing by incrementing the position until an empty slot is found.

4. Repeat steps 2-3 until the key is successfully inserted or the hash table is full.

4. Quadratic probing

1. Hash the key to get the initial position (index) in the hash table.

2. If the position is empty, insert the key at that position.

3. If the position is occupied by another key, perform quadratic probing to find the next available slot.

- Start probing by incrementing the position by i^2, where i is the iteration number.

- If the new position is empty, insert the key at that position.

- If the new position is occupied, continue probing using the next quadratic increment until an empty slot is found.

4. Repeat steps 2-3 until the key is successfully inserted or the hash table is full.

5. Double hashing

1. Hash the key to get the initial position (index) in the hash table.

2. If the position is empty, insert the key at that position.

3. If the position is occupied by another key, use a second hash function to calculate an increment value.

- Calculate the increment using a second hash function: increment = (hash2(key) % (table\_size - 1)) + 1.

- Start probing by incrementing the position by the calculated increment.

- If the new position is empty, insert the key at that position.

- If the new position is occupied, continue probing using the next increment until an empty slot is found.

4. Repeat steps 2-3 until the key is successfully inserted or the hash table is full.

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

struct employee{

    int empid;

    char empname[20];

    char empaddr[50];

};

int linear\_search(struct employee arr[], int n, int item){

    int i = 0;

    while(i < n && item != arr[i].empid)

        i++;

    if (i < n)

        return i;

    else return -1;

}

int binary\_search(struct employee arr[], int n, int item){

    int low = 0, high = n - 1, mid;

    while(low <= high){

        mid = (low + high) / 2;

        if(arr[mid].empid == item)

            return mid;

        else if(arr[mid].empid > item)

            low = mid + 1;

        else high = mid - 1;

    }

    return -1;

}

int main(){

    struct employee arr[MAX];

    int n, s, item;

    printf("Enter number of employees: ");

    scanf("%d", &n);

    printf("Enter the employee details:\n");

    for(int i = 0; i < n; i++){

        printf("Employee %d:\n", i+1);

        printf("ID: ");

        scanf("%d", &arr[i].empid);

        printf("Name: ");

        scanf("%s", arr[i].empname);

        printf("Address: ");

        scanf("%s", arr[i].empaddr);

        printf("-----------------------\n");

    }

    do{

        printf("1. Linear search\n");

        printf("2. Binary search\n");

        printf("3. Exit\n");

        printf("Enter your option: ");

        scanf("%d", &s);

        switch(s){

            case 1: printf("Enter the ID: ");

                    scanf("%d", &item);

                    item = linear\_search(arr, n, item);

                    if(item == -1)

                        printf("Item not found\n");

                    else printf("Item found at %dth location\n", item);

                    break;

            case 2: printf("Enter the ID: ");

                    scanf("%d", &item);

                    item = binary\_search(arr, n, item);

                    if(item == -1)

                        printf("Item not found\n");

                    else printf("Item found at %dth location\n", item);

                    break;

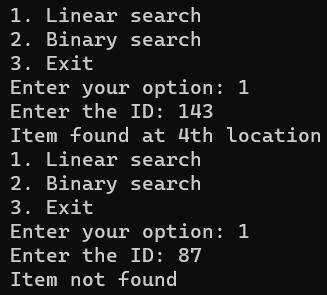
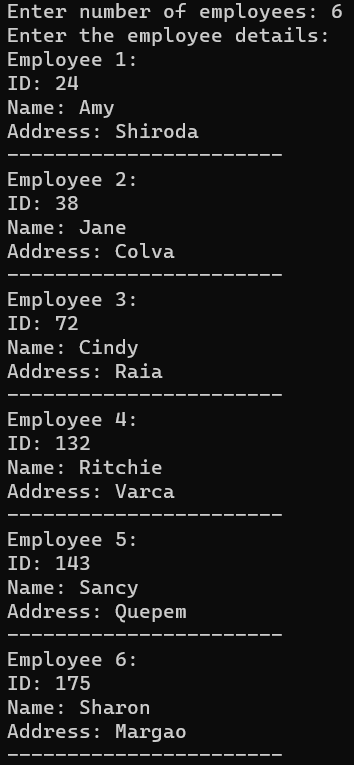
            case 3: break;

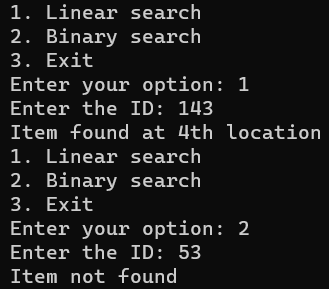
            case 4: printf("Invalid input\n");

        }

    }while(s != 3);

}





#include<stdio.h>

#include<stdlib.h>

#define MAX 10

enum type\_of\_record {EMPTY, DELETED, OCCUPIED};

int probing\_method;

struct employee{

    int empid;

    char empname[20];

    char empaddr[50];

};

struct Record{

    struct employee info;

    enum type\_of\_record status;

}table[MAX];

int hash(int key){

    return key % MAX;

}

int double\_hash(int key){

    return (7 + (key % (MAX - 1)));

}

int collision\_resolution(int h, int i, int key){

    switch(probing\_method){

        case 1: return ((h + i) % MAX);//linear probing

        case 2: return ((h + i\*i) % MAX);//quadratic probing

        case 3: return ((h + i\*double\_hash(key)) % MAX);//double

    }

}

int search(int key){

    int i, h, location;

    h = hash(key);

    location = h;

    for(i = 1; i < MAX; i++){

        if(table[location].status == EMPTY)

            return -1;

        if(table[location].info.empid == key)

            return location;

        location = collision\_resolution(h, i, key);

    }

    return -1;

}

void insert(struct employee emprec){

    int i, h, location;

    h = hash(emprec.empid);

    location = h;

    for (i = 1; i < MAX; i++){

        if(table[location].status == EMPTY || table[location].status == DELETED){

            table[location].info = emprec;

            table[location].status = OCCUPIED;

            return;

        }

        location = collision\_resolution(h, i, emprec.empid);

    }

    printf("Table overflow\n");

}

void delete(int key){

    int location = search(key);

    if(location == -1)

        printf("Not found\n");

    else{

        table[location].status = DELETED;

        printf("Deleted\n");

    }

}

struct employee get\_details(int i){

    struct employee temp;

    printf("Employee %d\n:", i+1);

    printf("ID: ");

    scanf("%d", &temp.empid);

    printf("Name: ");

    scanf("%s", temp.empname);

    printf("Address: ");

    scanf("%s", temp.empaddr);

    return temp;

}

void display(){

    int i;

    for (i = 0; i < MAX; i++){

        printf("[%d] : ", i);

        if (table[i].status == OCCUPIED){

            printf("Occupied : %d %s", table[i].info.empid, table[i].info.empname);

            printf(" %s\n", table[i].info.empaddr);

        }

        else if (table[i].status == DELETED)

            printf("Deleted\n");

        else

            printf("Empty\n");

    }

}

int main(){

    struct employee emp;

    int n, s, item;

    printf("1.Linear 2.Quadratic 3.Double hash\n");

    printf("Enter collision resolution technique: ");

    scanf("%d", &probing\_method);

    printf("Enter number of employees: ");

    scanf("%d", &n);

    printf("Enter the employee details:\n");

    for(int i = 0; i < n; i++){

        printf("Employee %d:\n", i+1);

        printf("ID: ");

        scanf("%d", &emp.empid);

        printf("Name: ");

        scanf("%s", emp.empname);

        printf("Address: ");

        scanf("%s", emp.empaddr);

        insert(emp);

        printf("-----------------------\n");

    }

    do{

        printf("1. Insert a record\n");

        printf("2. Search a record\n");

        printf("3. Delete a record\n");

        printf("4. Display table\n");

        printf("5. Exit\n");

        printf("Enter your option:");

        scanf("%d", &s);

        switch(s){

            case 1: insert(get\_details(n++));

                    break;

            case 2: printf("Enter ID: ");

                    scanf("%d", &item);

                    item = search(item);

                    if(item == -1)

                        printf("Item not found\n");

                    else printf("Item found at %dth location\n", item);

                    break;

            case 3: printf("Enter ID: ");

                    scanf("%d", &item);

                    delete(item);

                    break;

            case 4: display();

                    break;

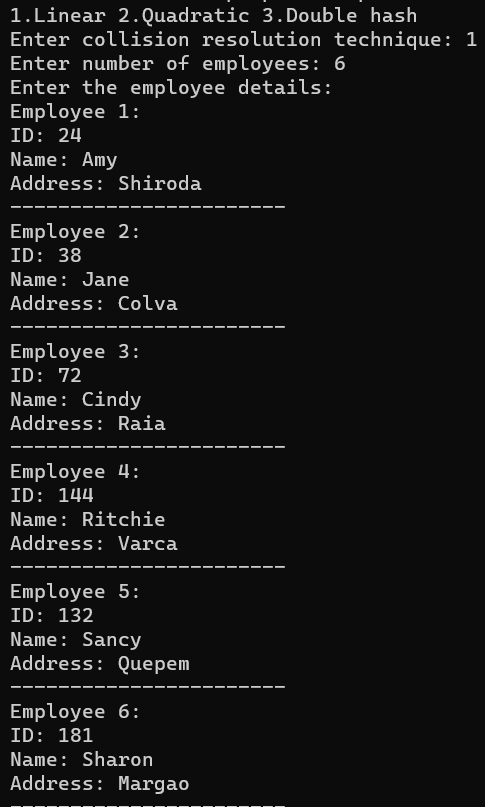
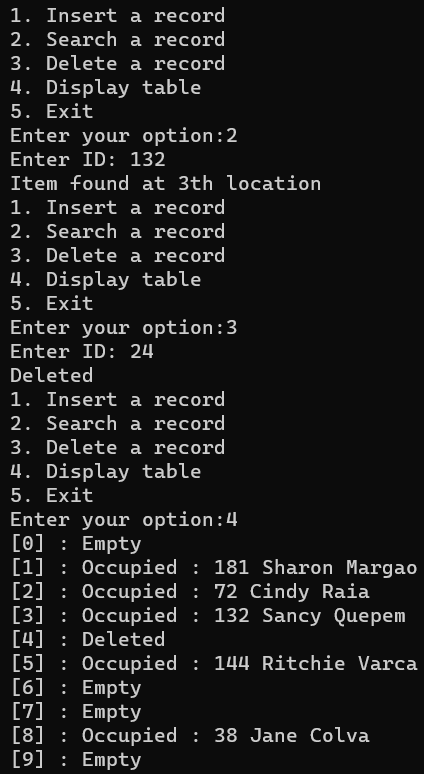
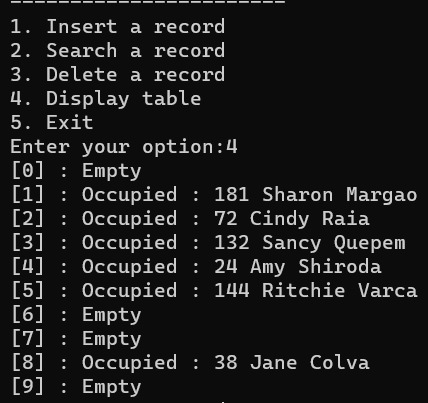
            case 5: break;

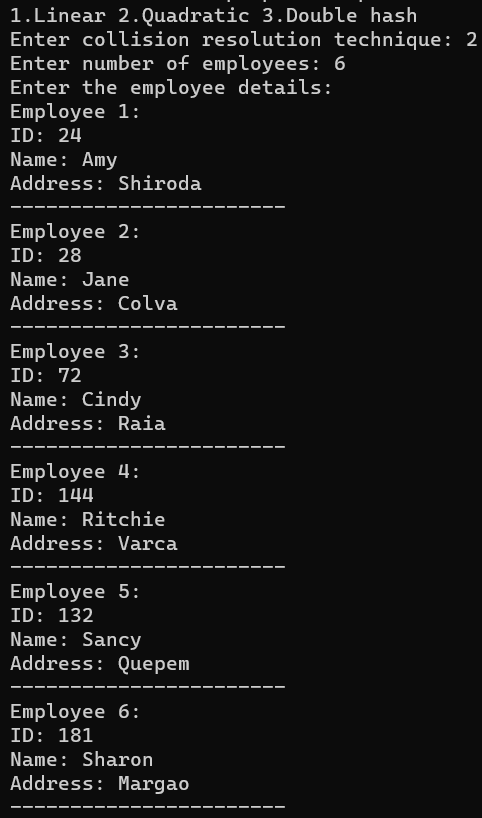
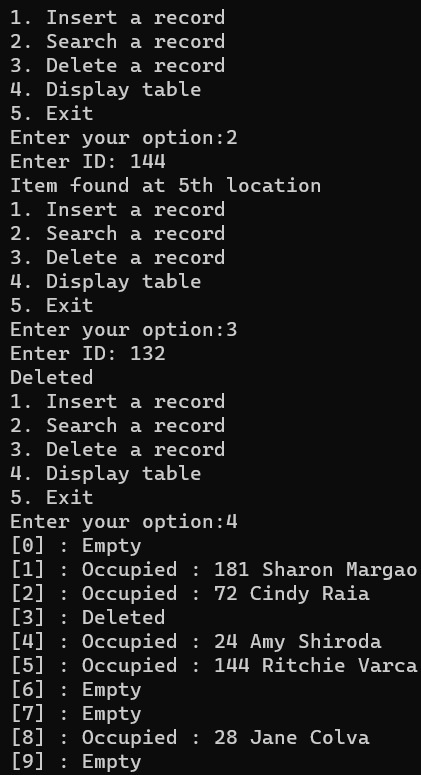
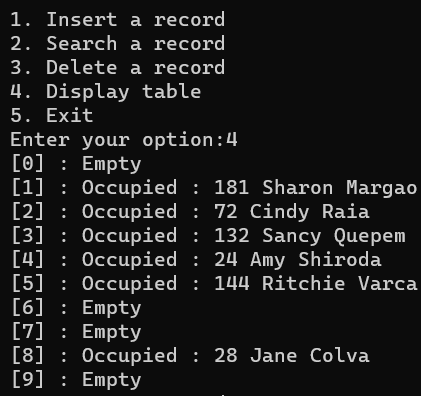
            default: printf("Invalid input\n");

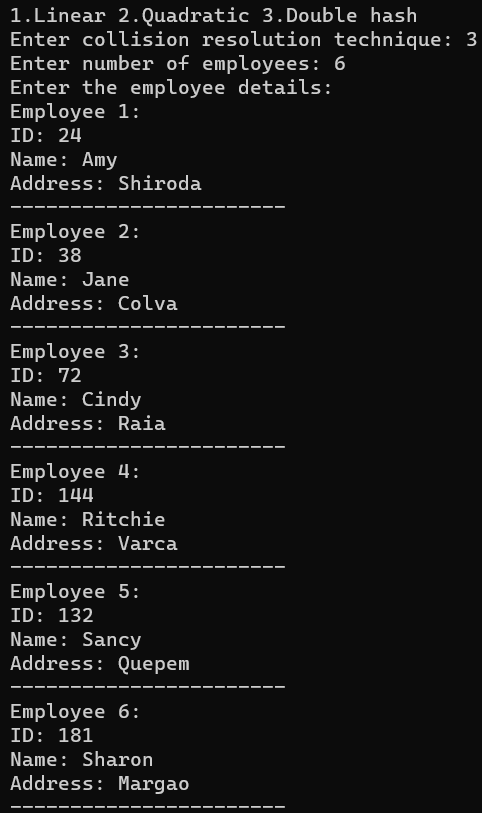
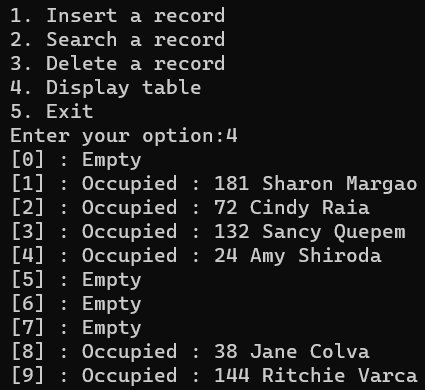
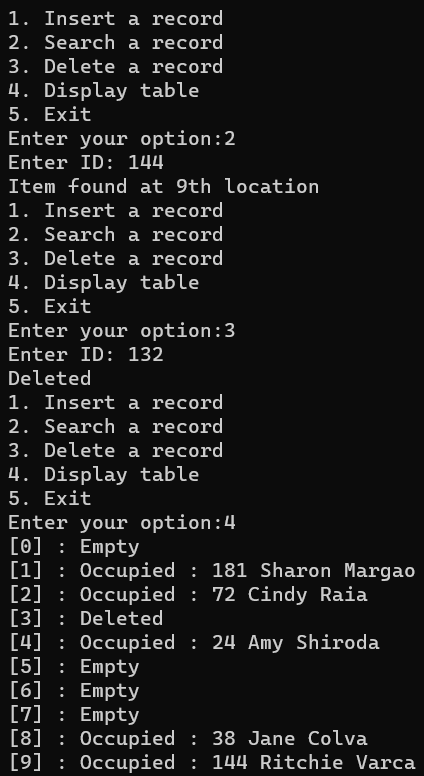
        }

    }while(s != 5);

}







Separate chaining

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

struct employee{

    int empid;

    char name[20];

    int age;

};

struct Record{

    struct employee info;

    struct Record \*link;

}\*table[MAX];

int hash(int key){

    return (key % MAX);

}

int search(int key){

    int h;

    struct Record \*ptr;

    h = hash(key);

    ptr = table[h];

    while (ptr != NULL){

        if (ptr->info.empid == key)

            return h;

        ptr = ptr->link;

    }

    return -1;

}

void insert(struct employee emprec){

    int h, key;

    struct Record \*tmp;

    key = emprec.empid;

    if (search(key) != -1){

        printf("Duplicate key\n");

        return;

    }

    h = hash(key);

    tmp = (struct Record \*)malloc(sizeof(struct Record));

    tmp->info = emprec;

    tmp->link = table[h];

    table[h] = tmp;

}

void delete(int key){

    int h;

    struct Record \*tmp,\*ptr;

    h = hash(key);

    if (table[h] == NULL){

        printf("Key %d not found\n", key);

        return;

    }

    if (table[h]->info.empid == key){

        tmp = table[h];

        table[h] = table[h]->link;

        free(tmp);

        return;

    }

    ptr = table[h];

    while (ptr->link != NULL){

        if (ptr->link->info.empid == key){

            tmp = ptr->link;

            ptr->link = tmp->link;

            free(tmp);

            return;

        }

        ptr = ptr->link;

    }

    printf("Key %d not found\n", key);

}

void display(struct Record \*table[]){

    int i;

    struct Record \*ptr;

    for (i = 0; i < MAX; i++){

        printf("\n[%d] ", i);

        if (table[i] != NULL){

            ptr = table[i];

            while (ptr != NULL){

                printf("%d %s %d\t", ptr->info.empid, ptr->info.name, ptr->info.age);

                if (ptr->link != NULL)

                    printf("->\t");

                ptr = ptr->link;

            }

        }

    }

    printf("\n");

}

int main(){

    struct employee emp, emprec;

    int i, key, choice, n;

    for (i = 0; i <= MAX-1; i++)

        table[i] = NULL;

    printf("Enter number of employees: ");

    scanf("%d", &n);

    printf("Enter the employee details:\n");

    for(int i = 0; i < n; i++){

        printf("Employee %d:\n", i+1);

        printf("ID: ");

        scanf("%d", &emp.empid);

        printf("Name: ");

        scanf("%s", emp.name);

        printf("Age: ");

        scanf("%d", &emp.age);

        insert(emp);

        printf("-----------------------\n");

    }

    do{

        printf("1.Insert a record\n");

        printf("2.Search a record\n");

        printf("3.Delete a record\n");

        printf("4.Display table\n");

        printf("5.Exit\n");

        printf("Enter your choice\n");

        scanf("%d", &choice);

        switch (choice){

        case 1:

            printf("Enter the record\n");

            printf("Enter empid, name, age : ");

            scanf("%d%s%d", &emprec.empid, emprec.name, &emprec.age);

            insert(emprec);

            break;

        case 2:

            printf("Enter a key to be searched : ");

            scanf("%d", &key);

            i = search(key);

            if (i == -1)

                printf("Key not found\n");

            else

                printf("Key found in chain %d\n", i);

            break;

        case 3:

            printf("Enter a key to be deleted\n");

            scanf("%d", &key);

            delete(key);

            break;

        case 4:

            display(table);

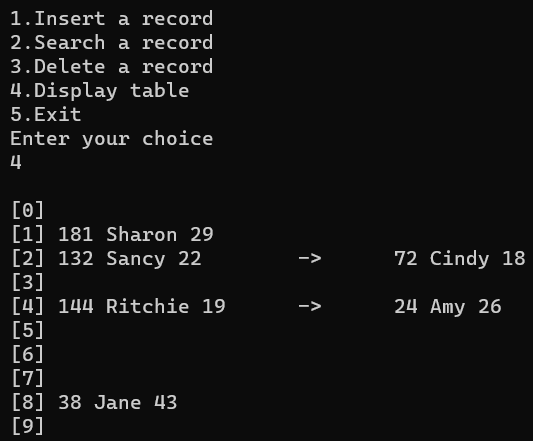
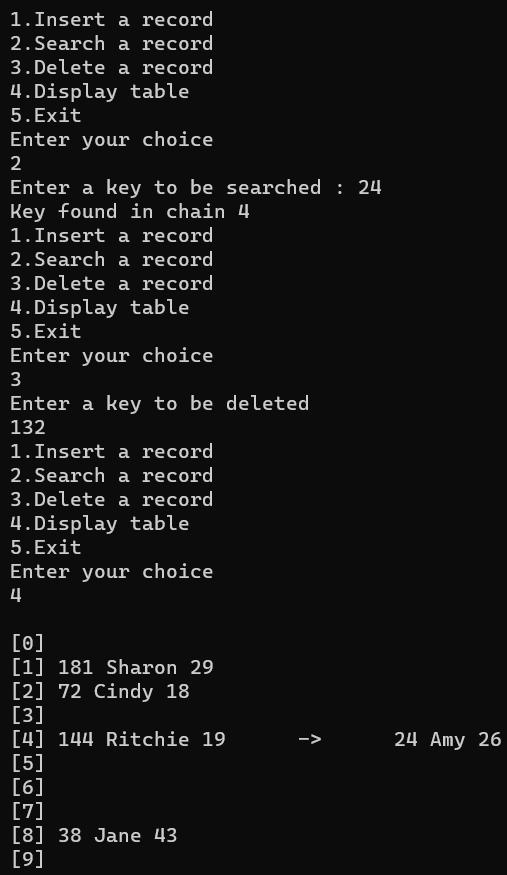
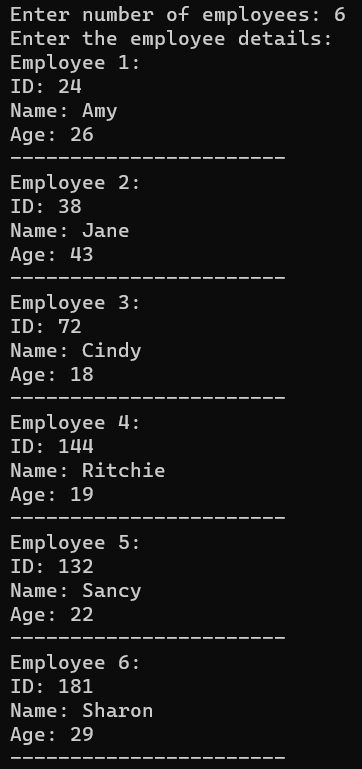
            break;

        case 5: break;

        }

    } while(choice != 5);

}



**Conclusion**: Three programs based on linear & binary searching, and hashing were written and outputs successfully obtained.