Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.: 16	Hashing	Date:30/05/2024

Write a C program to create a hash table and perform collision resolution using the following techniques.

(i) Open addressing

(ii) Closed Addressing

(iii) Rehashing

Algorithm:

1. Open Addressing:

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, move to the next slot in the hash table.
5. Repeat the process until an empty slot is found.

2. Closed Addressing (Separate Chaining):

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, add the new key to the linked list at that slot.

3. Rehashing:

1. When the load factor of the hash table reaches a certain threshold (typically > 0.7), create a new hash table of larger size.
2. Compute the hash of each key in the old table.
3. Insert each key into the new table.
4. Delete the old table.

PROGRAM:

A. OPEN ADDRESSING:

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 }; int
b[10];

void merging(int low, int mid, int high) { int
l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
if(a[l1] <= a[l2]) b[i] = a[l1++]; else
b[i] = a[l2++]; }

while(l1 <= mid) b[i++]
= a[l1++];

while(l2 <= high) b[i++]
= a[l2++];

for(i = low; i <= high; i++)
a[i] = b[i]; }

void sort(int low, int high) { int
mid;

if(low < high) { mid =
(low + high) / 2;
sort(low, mid);
sort(mid+1, high);
merging(low, mid, high);
} else { return;
}
}

int main() { int
i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);

sort(0, max);
```

```
printf("\nList after sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);
}
```

B. CLOSED ADDRESSING;

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 }; int
b[10];

void merging(int low, int mid, int high) {
int l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
if(a[l1] <= a[l2]) b[i] = a[l1++]; else b[i] = a[l2++];
}
while(l1 <= mid) b[i++]
= a[l1++]; while(l2 <=
high) b[i++] = a[l2++];

for(i = low; i <= high; i++)
a[i] = b[i];
}

void sort(int low, int high) { int
mid;

if(low < high) { mid =
(low + high) / 2;
sort(low, mid);
sort(mid+1, high);
merging(low, mid, high);
} else { return;
}
}

int main() { int
i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);

sort(0, max);

printf("\nList after sorting\n");
```

```
for(i = 0; i <= max; i++) printf("%d
", a[i]);
}
```

C. REHASHING:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
int key; int value;
struct Node* next;
} Node;

typedef struct HashTable {
int size; int count; Node**
table;
} HashTable;

Node* createNode(int key, int value) { Node*
newNode = (Node*)malloc(sizeof(Node));
newNode->key = key; newNode->value = value;
newNode->next = NULL; return newNode;
}

HashTable* createTable(int size) {
HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
newTable->size = size; newTable->count = 0;
newTable->table = (Node**)malloc(sizeof(Node*) * size); for
(int i = 0; i < size; i++) {

newTable->table[i] = NULL;
}
return newTable;
}

int hashFunction(int key, int size) { return
key % size;
}

void insert(HashTable* hashTable, int key, int value);

void rehash(HashTable* hashTable) {
int oldSize = hashTable->size; Node**
oldTable = hashTable->table;

// New size is typically a prime number or double the old size int
newSize = oldSize * 2;
hashTable->table = (Node**)malloc(sizeof(Node*) * newSize);
hashTable->size = newSize; hashTable->count = 0;
for (int i = 0; i < newSize; i++) {
hashTable->table[i] = NULL;
}
```

```
for (int i = 0; i < oldSize; i++) {
Node* current = oldTable[i]; while
(current != NULL) {
insert(hashTable, current->key, current->value);
Node* temp = current;

current = current->next; free(temp);
}
}

free(oldTable);
}

void insert(HashTable* hashTable, int key, int value) { if
((float)hashTable->count / hashTable->size >= 0.75) {
rehash(hashTable);
}

int hashIndex = hashFunction(key, hashTable->size);
Node* newNode = createNode(key, value); newNode->next
= hashTable->table[hashIndex]; hashTable-
>table[hashIndex] = newNode; hashTable->count++;
}

int search(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex];
while (current != NULL) { if
(current->key == key) {
return current->value;
}
current = current->next;
}

return -1;
}

void delete(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex]; Node*
prev = NULL;
while (current != NULL && current->key != key) { prev
= current;
current = current->next;
}
if (current == NULL) { return;
}
if (prev == NULL) {
hashTable->table[hashIndex] = current->next;
} else {
prev->next = current->next;

}
free(current);
```

```
hashTable->count--;
}

void freeTable(HashTable* hashTable) {
for (int i = 0; i < hashTable->size; i++) {
Node* current = hashTable->table[i];
while (current != NULL) { Node* temp =
current; current = current->next;

free(temp);
}
}
free(hashTable->table); free(hashTable);
}

int main() {
HashTable* hashTable = createTable(5);

insert(hashTable, 1, 10); insert(hashTable,
2, 20); insert(hashTable, 3, 30);
insert(hashTable, 4, 40); insert(hashTable,
5, 50);
insert(hashTable, 6, 60); // This should trigger rehashing

printf("Value for key 1: %d\n", search(hashTable, 1)); printf("Value
for key 2: %d\n", search(hashTable, 2)); printf("Value for key 3:
%d\n", search(hashTable, 3)); printf("Value for key 4: %d\n",
search(hashTable, 4)); printf("Value for key 5: %d\n",
search(hashTable, 5)); printf("Value for key 6: %d\n",
search(hashTable, 6));

delete(hashTable, 3);
printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

freeTable(hashTable);
return 0;

}
```

Data Structures Odyssey: Exploring the Foundations of Computing

OUTPUT:

```
aiml231501167@cselab:~$ gcc program16C.c
aiml231501167@cselab:~$ ./a.out
Value for key 1: 10
Value for key 2: 20
Value for key 3: 30
Value for key 4: 40
Value for key 5: 50
Value for key 6: 60
Value for key 3 after deletion: -1
aiml231501167@cselab:~$
```

```
Value for key 1: 10
Value for key 2: 20
Value for key 12: 30
Value for key 3: -1
Value for key 2 after deletion: -1
aiml231501167@cselab:~$
```

```
aiml231501167@cselab:~$ gcc program16A.c
aiml231501167@cselab:~$ ./a.out
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44 aiml231501167@cselab:~$
```

RESULT: Thus, the program was successfully executed.