Chapter 7

Searching and Hashing

Searching

- Searching refers to the process of finding a desired item in a set of items. The desired item is called "target". The set of items to be searched can be stored using any suitable data structure such as array, linked list, tree, graph etc.
- Here, we focus on searching techniques to search items that are stored in arrays.
- **Choice of algorithms:**
  - **Linear or sequential search** (unsorted or sorted array)
  - **Binary search** (sorted array)

Searching

- **Linear Search (unsorted array):**
  - Let us assume that the array is unsorted. This algorithm compares each array component in turn with target value. As soon as it finds a array component with value that equals the target value, the algorithm terminates and returns the component's index as its answer. If it finds no such component, it returns a special value (say -1) as its answer.
  - The linear search (or sequential search) algorithm is so called because it compares each array component in tern with the target value.
  - **Analysis:** If the search is successful, the algorithm compare target with any number of components from **1** through **n**. so, average number of comparisons is **(n + 1) / 2**. If the search is unsuccessful, all components will be compared. So, number of comparisons is **n**. In either case, the linear search algorithm has time complexity **O(n)**.

Searching

❖ **C-Function:**

```
int linearsearch(int a[], int n, int target)
{
    for(int i = 0; i < n; i++)
    {
            if(target == a[i])
                    return i;
    }
    return -1;
}
```

Searching

- **Linear Search (sorted array):**
  - Let us assume that the array is sorted. This enables us to improve the linear search algorithm slightly. During the search process, if target is less than a value in the array, the algorithm returns a special value (say -1) as its answer.
  - **Analysis:** Whether the search is successful or unsuccessful, this algorithm's average number of comparison is **(n + 1) / 2**. So, time complexity is **O(n)**.
  - However, binary search is much better.

Searching

❖ **C-Function:**

```
int linearsearchsorted(int a[], int n, int target)
{
    for(int i = 0; i < n; i++)
    {
            if(target == a[i])
                    return i;
            else if(target < a[i])
                    return -1;
    }
    return -1;
}
```

Searching

- **Binary Search:**
  - The most efficient method of searching a sorted array is the binary search.
  - In this searching technique, the target value is compared with the value of the middle of the array. If they are equal, the search ends successfully. Otherwise, either upper or lower half of the array must be searched in a similar manner.
  - **Analysis:** Let *n* be the length of the array. Assume that each execution of the loop perform a single comparison. If the search is **unsuccessful**, the loop is repeated as often as we must halve *n* to reach 1. So, number of comparisons is $\lfloor \log_2 n \rfloor + 1$. If the search is **successful**, the loop is repeated at most that many times. So, maximum number of comparisons is also $\lfloor \log_2 n \rfloor + 1$. In either case, the time complexity of binary search is $O(\log n)$.

Searching

❖ **C-Function:**

```
int binarysearch(int a[], int n, int target)
{
      int l = 0, r = n-1, m;
      while(l <= r)
      {
                  m  = (l + r) / 2;
                  if(target == a[m])
                              return m;
                  else if(target < a[m])
                              r = m -1;
                  else
                              l = m + 1;
      }
      return -1;
}
```

Hashing

- **Hashing** is a technique or process of mapping each key to a small integer and use that integer to index an array. This mapping is done using a **hash function**.
- The efficiency of hashing depends on the efficiency of the hash function used.
- **Hash table** is an array of size m together with a hash function *h*(*k*) that translates each key *k* to a array index (in the range 0…*m*–1).
- Hashing is a technique used for performing insertions, deletions, and searching in constant average time.
- To **insert** a key *k* into the hash table, insert that key to the array index given by the hash function *h(k)*.

Hashing

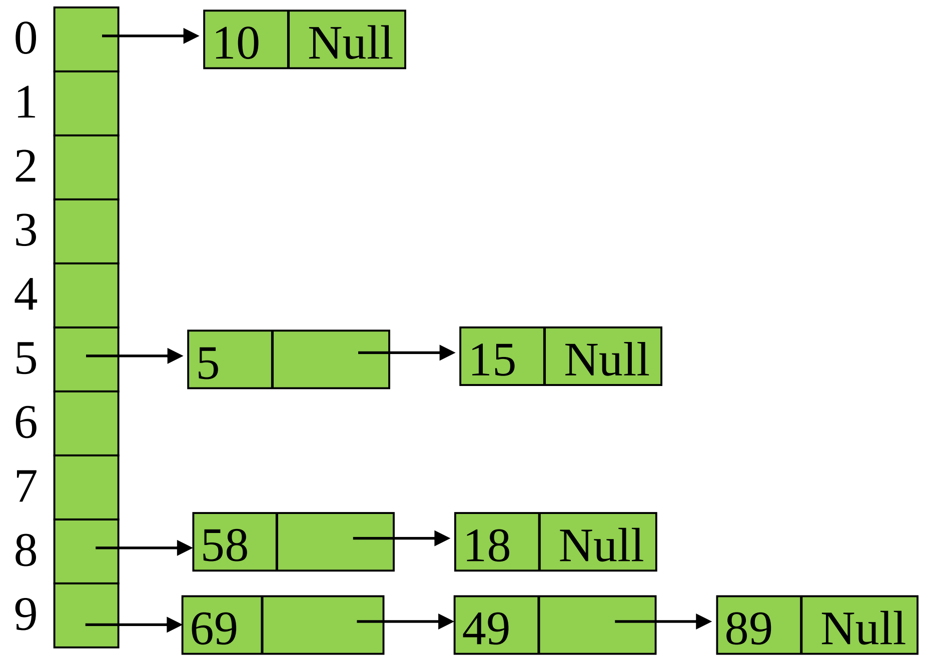
- To **delete** a key *k* from the hash table, look it in the array index given by hash function *h(k)* and delete it from there.
- To **search** a key *k* into the hash table, look that key to the array index given by the hash function *h(k)*.
- The hash function must be consistent, that is, $k_1 = k_2$ implies $h(k_1) = h(k_2)$.
- In general, the hash function is many-to-one. Therefore different keys may share the same array index, that is, $k_1 \neq k_2$ but $h(k_1) = h(k_2)$. This is called a **hash collision** or **hash clash**.
- Always prefer a hash function that makes collisions relatively infrequent.
- There are several methods for dealing with collision. The most common are **open hashing** and **closed hashing**.

Open Hashing

- **Open hashing** is also called **separate chaining**. This technique builds a linked list of all keys that hash to the same array index.
- To perform **search**, we use the hash function to determine which linked list to traverse. We than traverse this linked list in the normal manner, returning the position where the key found.
- To perform an **insert**, we traverse down the appropriate linked list to check whether the key is already in place. If the key turns out to be new, it is inserted either at the front or the end of the linked list.
- To perform **delete**, we use hash function to determine which linked list to traverse. We then delete the node from the linked list with the given key.

Open Hashing

- Suppose, the set of keys is {10, 15, 89, 18, 49, 58, 69, 5} and hash function is ***h(x) = x mod 10***. After inserting all these keys, chaining yields



Open Hashing

- **Analysis of search/insertion/deletion algorithms:**
  - Let the number of entries be n.
  - In the **best case**, no array index contains more than (say) 2 entries and maximum number of comparisons = 2. Hence, best-case time complexity is $O(1)$.
  - In the **worst case**, one array index contains all n keys and maximum number of comparisons = n. Hence, Worst-case time complexity is $O(n)$.

Open Hashing

- **C Program:**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int key;
        struct node *next;
};
struct node *table[10];
int hash(int k)
{
        return k % 10;
}
```

Open Hashing

```
void insert(int k)
{
	int i = hash(k);
	struct node *curr = table[i];
	while(curr != NULL)
	{
			if(k == curr->key)
				return;
			curr = curr->next;
	}
	struct node *newnode = (struct node*) malloc(sizeof(struct node));
	newnode->key = k;
	newnode->next = NULL;
	newnode->next = table[i];
	table[i] = newnode;
}
```

Open Hashing

```
node* search(int k)
{
    int i = hash(k);
    struct node *curr = table[i];
    while(curr != NULL)
    {
            if(k == curr->key)
                    return curr;
            curr = curr->next;
    }
    return NULL;
}
```

Open Hashing

```
void del(int k)
{
    int i = hash(k);
    struct node *curr = table[i];
    struct node *pred = NULL;
    while(curr != NULL)
    {
            if(k == curr->key)
            {
                    if (pred == NULL)
                            table[i] = curr->next;
                    else
                            pred->next = curr->next;
                    free(curr);
                    return;
            }
            pred = curr;
            curr = curr->next;
    }
}
```

Open Hashing

```
int main()
{
        insert(10);
        insert(15);
        insert(89);
        insert(18);
        insert(49);
        insert(58);
        insert(69);
        insert(5);
        del(89);
        printf("%d\n", search(89));
        printf("%d\n", search(58));
}
```

Closed Hashing

- Open hashing has the disadvantage of requiring pointers. This tends to slow the algorithm down a bit because of the time required to allocate new cells, and it also essentially requires the implementation of a second data structure.
- **Closed hashing**, also known as **open addressing**, is an alternative to resolve collision.
- In a closed hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. More formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$,…are tried in succession where $\mathbf{h_i(x) = (h(x) + f(i)) \bmod m}$, where m is the size of array. The function f(i) depends on the collision resolution strategy we use. The three common collision resolution strategies are: **linear probing**, **quadratic probing**, and **double hashing**.

Closed Hashing

- **Linear Probing:**
  - In linear probing, we use **f(i) = i**. Hence, the hash function in this case becomes **$h_i(x) = (h(x) + i) \bmod m$**.
  - Suppose, the set of keys is {10, 15, 89, 18, 49, 58, 69, 5}, m = 10, and ***$h(x) = x \bmod 10$***. After inserting these keys, linear probing yields.

	After 10	After 15	After 89	After 18	After 49	After 58	After 69	After 5
0	10	10	10	10	10	10	10	10
1					49	49	49	49
2						58	58	58
3							69	69
4								
5		15	15	15	15	15	15	15
6								5
7								
8				18	18	18	18	18
9			89	89	89	89	89	89

Closed Hashing

- **Quadratic Probing:**
  - Here, we use **f(i) = i^2**. Hence, the hash function in this case becomes **$h_i(x) = (h(x) + i^2) \bmod m$**.
  - Suppose, the set of keys is {10, 15, 89, 18, 49, 58, 69, 5}, m = 10, and ***h(x) = x mod 10***. After inserting these keys, quadratic probing yields.

	After 10	After 15	After 89	After 18	After 49	After 58	After 69	After 5
0	10	10	10	10	10	10	10	10
1								
2						58	58	58
3					49	49	49	49
4							69	69
5		15	15	15	15	15	15	15
6								5
7								
8				18	18	18	18	18
9			89	89	89	89	89	89

Closed Hashing

- **Double Hashing:**
  - Here, one popular choice is $\mathbf{f(i) = i.h_2(x)}$. Hence, the hash function in this case becomes $\mathbf{h_i(x) = (h(x) + i.h_2(x)) \bmod m}$.
  - A poor choice of $h_2(x)$ would be disastrous. This function must never evaluate to zero. A function such as $\mathbf{h_2(x) = R - (x \bmod R)}$, where R is a prime smaller than m, will work well. If we chose R = 7, then the result of inserting the keys {10, 15, 89, 18, 49, 58, 69, 5}, m = 10, and ***h(x) = x mod 10*** is given below.

Closed Hashing

	After 10	After 15	After 89	After 18	After 49	After 58	After 69	After 5
0	10	10	10	10	10	10	10	10
1							69	69
2								
3						58	58	58
4								
5		15	15	15	15	15	15	15
6					49	49	49	49
7								5
8				18	18	18	18	18
9			89	89	89	89	89	89

Closed Hashing

- **Analysis of search/insertion/deletion algorithms:**
  - Let the number of entries be *n*.
  - In the **best case**, no cluster contains more than (say) 4 entries and maximum number of comparisons = 4. Hence, best-case time complexity is $O(1)$.
  - In the **worst case**, one cluster contains all *n* keys and maximum number of comparisons = *n*. Hence, Worst-case time complexity is $O(n)$.

Closed Hashing

- **C-Program (Linear Probing):**

```
#include<stdio.h>
int a[10] = {NULL};
int hash(int k)
{
		return k % 10;
}
void insert(int k)
{
		int i = hash(k), index;
		for(int j = 0; j < 10; j++)
		{
				index = (i + j) % 10;
				if(a[index]==NULL)
				{
						a[index] = k;
						break;
				}
		}
}
```

Closed Hashing

```
int search(int k)
{
        int i = hash(k), index;
        for(int j = 0; j < 10; j++)
        {
                index = (i + j) % 10;
                if(a[index]==k)
                {
                        return index;
                }
        }
        return -1;
}
```

Closed Hashing

```
void del(int k)
{
        int i = hash(k), index;
        for(int j = 0; j < 10; j++)
        {
                index = (i + j) % 10;
                if(a[index]==k)
                {
                        a[index]=NULL;
                        break;
                }
        }
}
```

Closed Hashing

```
int main()
{
        insert(10);
        insert(15);
        insert(89);
        insert(18);
        insert(49);
        insert(58);
        insert(69);
        insert(5);
        del(58);
        printf("%d\n", search(5));
        printf("%d\n", search(58));
}
```

Hash Table Design

- The **load factor** of a hash table is the average number of entries per array index, *n*/*m*.
- If *n* is (roughly) predictable, choose *m* such that the load factor is likely to be between 0.5 and 0.75.
  - A low load factor wastes space.
  - A high load factor tends to increase collision.
- Choose *m* to be a prime number.
  - Typically the hash function performs modulo-*m* arithmetic. If *m* is prime, the entries are more likely to be distributed evenly in the array, regardless of any pattern in the keys.

Hash Table Design

- The hash function should be efficient (performing few arithmetic operations).
- The hash function should distribute the entries evenly in the array, regardless of any patterns in the keys.
- Possible trade-off:
  - Speed up the hash function by using only part of the key.
  - But beware of any patterns in that part of the key.