Unit 5 - Trie Trees & Hashing

Class notes by Vibha Masti

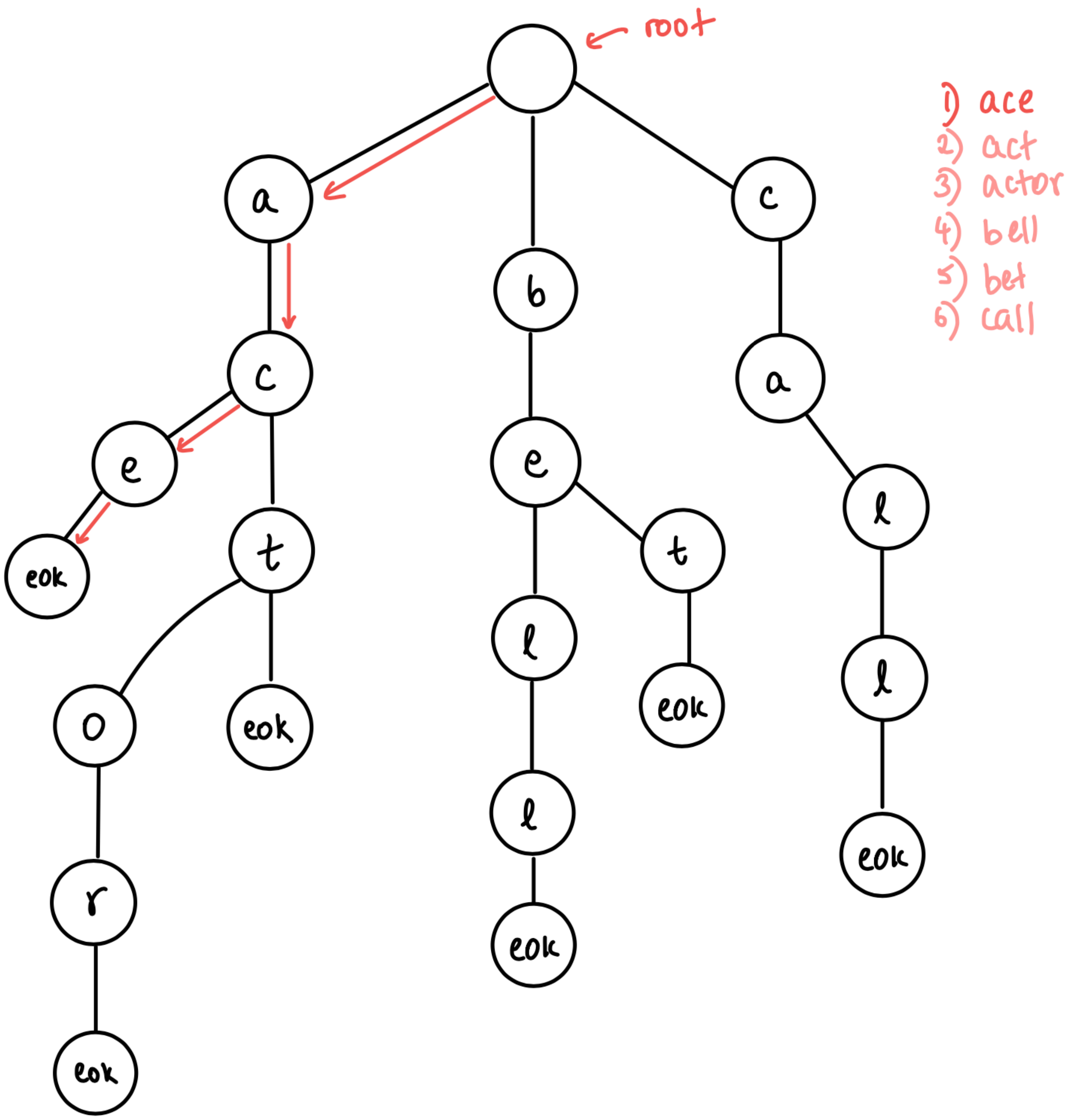
Feedback/corrections: vibha@pesu.pes.edu

Trie Trees

- Focus on accessing data faster
- Large databases

Introduction

- Principle of BST but as many children as possible
- Comes from the word re***trie***val
- Also called prefix tree, digital tree
- Generally used to store things
- Each node can contain m pointers corresponding to m possible symbols in each position of the key
- Mobile phone databases
- If keys are strings, BSTs will compare entire strings while trie compares individual characters
- A trie is a tree where each node stores a bit indicating wherether the string spelled out to this point is in the set
- Example (tracing out the word ace)



- Looking for word patterns in text

Code Implementation

- Every character in the pattern is inndicated by a **link** and not a node
- If a connection exists, then the previous node exists in the pattern
- Note: I hate global variables so I have avoided using them, making the code slightly more complicated

Structure for Trie

- ASCII value of each character is the index in the `child` array of pointers
- `endofword` indicates whether that character is the end of the word in that trie pattern
- `create_node` creates a new `TrieNode` and initialises all the members of the `child` array to `NULL`

```
#define NO_OF_CHARACTERS 255
#define MAX 255

typedef struct trienode {
    /* Child can have upto 255 children (pointers) */
    struct trienode *child[NO_OF_CHARACTERS]; /* Array of pointers */
    int endofword;
} TrieNode;


TrieNode *create_node() {
    /* Create a new TrieNode */
    TrieNode *temp;
    temp = (TrieNode *) malloc(sizeof(TrieNode));
    temp->endofword = 0;

    /* Initialise all pointers to NULL */
    for (int i = 0; i < NO_OF_CHARACTERS; ++i) {
        temp->child[i] = NULL;
    }
    return temp;
}
```

Insert a pattern/key/word

```
void insert(TrieNode *root, char *key) {
    /* ASCII values are indices */
    TrieNode *cursor = root;
    int i;

    for (i = 0; key[i]; ++i) {
        /* Find if node for that letter exists */
        if (cursor->child[key[i]] == NULL) {
            cursor->child[key[i]] = create_node();
        }
        cursor = cursor->child[key[i]];
    }
    cursor->endofword = 1;
}
```

Search for a pattern/key/word

```
int search(TrieNode *root, char *key) {
    TrieNode *cursor = root;

    for (int i = 0; key[i]; ++i) {
        /* Character in that pattern absent */
        if (cursor->child[key[i]] == NULL) {
            /* Not found */
            return 0;
        }
        cursor = cursor->child[key[i]];
    }

    /* Prefixes are not end of words */
    return cursor->endofword;
}
```

Display all patterns/keys/words

```
void display(TrieNode *root) {
    /* DFS traversal */
    char word[MAX] = "";
    int length = 0;
    dfs_word(root, word, &length);
}

void dfs_word(TrieNode *root, char *word, int *plength) {
    for (int i = 0; i < NO_OF_CHARACTERS; ++i) {
        /* If char exists in a pattern */
        if (root->child[i] != NULL) {
            /* Add it to the word array */
            word[*plength] = i;
            ++*plength;
            /* If end of word */
            if (root->child[i]->endofword) {
                /* Print word */
                word[*plength] = 0;
                printf("%s\n", word);
            }
            dfs_word(root->child[i], word, plength);
        }
    }
    --*plength;
```

```
}
```

Using global variables

```
/* Global variables */
char word[100];          // Word
int length;              // Length of the word (search)
int top;                 // Top pf the stack
Stack s[100];            // Stack


void display(TrieNode *curr) {
  for (int i = 0; i < 26; ++i) {
    /* If char exists in a pattern */
    if (curr->child[i] != NULL) {
      /* Add it to the word array */
      word[length] = ('a' + i);
      ++length;
      /* If end of word */
      if (curr->child[i]->endofword) {
        /* Print word */
        for (int j = 0; j < length; ++j) {
            printfun(word[j]);
        }
        printnewline();
      }
      display(curr->child[i]);
    }
  }
  --length;
}
```

Delete a pattern/key/word

- Stack needed to keep track of traversal path (all the letters in the word to be deleted)

```
/* struct for deletion */
typedef struct stack {
    TrieNode *trie_node;    /* TrieNode in the path */
    int index;              /* Index for removing edge */
} Stack;
```

- Helper functions for stack (push and pop) and check (for checking if node has connections)

```
void push(Stack *stack_array, TrieNode *curr, int index, int *ptop) {
    Stack new_el;
    new_el.index = index;
    new_el.trie_node = curr;

    stack_array[++(*ptop)] = new_el;
}

Stack pop(Stack *stack_array, int *ptop) {
    return stack_array[(*ptop)--];
}

int check(TrieNode *curr) {
    /* Number of connections */
    int count = 0;
    for (int i = 0; i < NO_OF_CHARACTERS; ++i) {
        if (curr->child[i] != NULL) {
            ++count;
        }
    }
    return count;
}
```

- Function to delete

```
void delete_word(TrieNode *root, char *key) {
    TrieNode *curr = root;
    Stack stack_array[MAX], stack_el;
    int index, top = -1, other_keys;

    for (int i = 0; key[i]; ++i) {
        /* Index of connection */
        index = key[i];
        if (curr->child[index] == NULL) {
            /* No connection at that letter's index */
            printf("Word not found\n");
            return;
        }
        /* Push the node and index (current letter) to stack */
        push(stack_array, curr, index, &top);
        curr = curr->child[index];
    }
    /* No longer end of word */
    curr->endofword = 0;

```

```c
    /* Push final node to stack with letter index -1 (no letter follows)
*/
    push(stack_array, curr, -1, &top);

    /* Remove edges (connections) from nodes on stack */
    while (1) {
        stack_el = pop(stack_array, &top);

        /* Check if not last char of word (only from second iteration) */
        if (stack_el.index != -1) {
            /* Make connection of node index NULL (delete last letter) */
            stack_el.trie_node->child[stack_el.index] = NULL;
        }

        /* Root element - all chars have been deleted */
        if (stack_el.trie_node == root) {
            break;
        }

        /* If the trie node has other key connections */
        other_keys = check(stack_el.trie_node);

        /* If the trie node has other key
        connections or is the end of another word */
        if (other_keys >= 1 || stack_el.trie_node->endofword) {
            break;
        }
        else {
            /* Delete node if no connections */
            free(stack_el.trie_node);
        }
    }
}
```

Using global variables

```c
/* Global variables */
char word[100];         // Word
int length;             // Length of the word (search)
int top;                // Top pf the stack
Stack s[100];           // Stack


void delete_trie(TrieNode *root, char *key) {
  TrieNode *curr = root;
  int index, other_keys;
  Stack stack_el;
```

```

  top = -1;
  for (int i = 0; key[i]; ++i) {
    index = key[i] - 'a';
    if (curr->child[index] == NULL) {
      return;
    }

    push(curr, index);
    curr = curr->child[index];
  }
  curr->endofword = 0;
  push(curr, -1);

  while (1) {
    stack_el = pop();

    /* Remove next connection */
    if (stack_el.index != -1) {
      stack_el.m->child[stack_el.index] = NULL;
    }

    /* Root element - all chars have been deleted */
    if (stack_el.m == root) {
      break;
    }

    other_keys = check(stack_el.m);

    /* If the trie node has other key
    connections or is the end of a word */
    if (other_keys >= 1 || stack_el.m->endofword) {
      break;
    }
    else {
      free(stack_el.m);
    }
  }
}

int check(TrieNode *x) {
  int count = 0;
  for (int i = 0; i < 26; ++i) {
    if (x->child[i] != NULL)
      ++count;
  }
  return count;
}

```

```
void push(TrieNode *x, int k) {
  Stack new_el;
  new_el.index = k;
  new_el.m = x;

  s[++top] = new_el;
}

Stack pop() {
  return s[top--];
}
```

Hashing & Hash Tables

- Retrieving data in constant time $O(1)$ instead of linear or logarithmic time
- Fast retrieval, more efficient than arrays
- Direct mapping (one-to-one) works only for numeric data of similar range, not too many gaps
- In other words, element 6 will be at 6th index of array; retrival will be in constant time
- For more versatile data (different ranges, datatypes), direct mapping will not work as efficiently
- Too many empty locations and large ranges in such an implementation
- Need a good hash table & hash function

Hash Function

- Any function that can be used to map data of arbitrary size to fixed-size values
- Value returned by a hash function: *hash value*, *hash code*, *digest*, or simply *hash*
- For example, mod function, $identity$ function (one-to-one), multiplicative hashing, folding and adding (adding all digits), truncation etc
- A good hash function is one that distributes keys evenly among all slots/indices (locations)

Collision

- When multiple keys generate the same hash value after being entered in to a hash function
- For example, if hash function is $mod\ 10$ of a number and the two numbers are 21 and 411, both result in the same hash value (1) and they collide
- Collision resolution techniques
- Hash functions that have minimum number of collisions are good
- Look up birthday paradox (24 for probability > 0.5)

Load factor

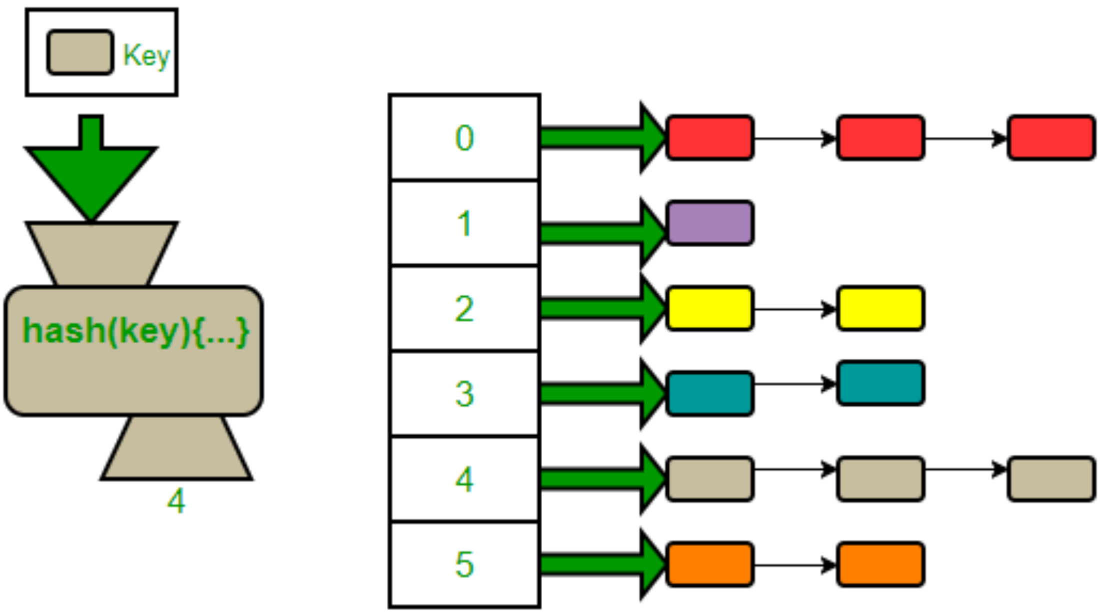
- $load\ factor = \frac{number\ of\ records\ filled}{total\ capacity}$

Collision Resolution Techniques

- Two techniques for resolving collisions
  - **Open hashing** - separate chaining (linked list at position)
  - **Closed hashing** - open addressing (all keys stored inside hash table; no new memory created)
    - Linear probing
    - Quadratic Probing
    - Double Hashing
- Still almost constant time with collisions
- Hash tables can store data in the form of key-value pairs
- For simplicity, implementation is only key
- Value can be added to the definition of the struct node

Separate Chaining

- New memory allocated during collisions



- Worst case scenario is if all the keys get mapped to the same bucket and we have a linked list of n (size of hash table) size from one single bucket, with all the other buckets empty

Code Implementation

Structure for the hash table and nodes

```
#define STR_LEN 20

/* Every element of the linked list */
typedef struct node {
    int key;            /* int value representing the key */
    struct node *next;  /* Link */
} Node;

/* Every element of the hash table */
typedef struct hash {
    struct node *head;  /* Head of linked list */
    int count;          /* Number of nodes in linked list */
} HashTable;
```

Initialise and destroy tables

```
HashTable *initialise_table(int size) {
    /* Initialise array of size size */
    HashTable *temp = calloc(size, sizeof(HashTable));

    for (int i = 0; i < size; ++i) {
        temp[i].head = NULL;
        temp[i].count = 0;
    }
    return temp;
}

void destroy_hash(HashTable *hashtable, int size) {
    /* Destroy hash table - delete all open chains */
    Node *temp = NULL, *to_del = NULL;

    for (int i = 0; i < size; ++i) {
        temp = hashtable[i].head;
        hashtable[i].head = NULL;

        for (int j = 0; j < hashtable[i].count; ++j) {
            to_del = temp;
            temp = temp->next;
            free(to_del);
        }
    }
    free(hashtable);
```

```
}
```

Insert into hash table with hash function key % size

```c
void insert_hash(HashTable *hashtable, int size, int key) {

    /* Hash function: key % size */
    int hash = key % size;

    /* Initialise new node */
    Node *new_node = (Node *)malloc(sizeof(Node));
    new_node->key = key;

    /* Insert to front */
    new_node->next = hashtable[hash].head;

    /* Make new node the new head */
    hashtable[hash].head = new_node;

    /* Increment count */
    ++hashtable[hash].count;
}
```

Delete from hash table

```c
void delete_hash(HashTable *hashtable, int size, int key) {
    /* Hash function: key % size */
    int hash = key % size;

    Node *temp = hashtable[hash].head, *prev = NULL;

    while (temp != NULL) {
        if (temp->key == key) {

            /* Delete node */
            if (prev != NULL) {
                /* Not first node */
                prev->next = temp->next;
                --hashtable[hash].count;
                free(temp);
                return;
            }
            else {
                hashtable[hash].head = temp->next;
```

```c
                --hashtable[hash].count;
                free(temp);
                return;
            }
        }
        prev = temp;
        temp = temp->next;
    }
    printf("Key not found in hash table\n");
}
```

Display hash table

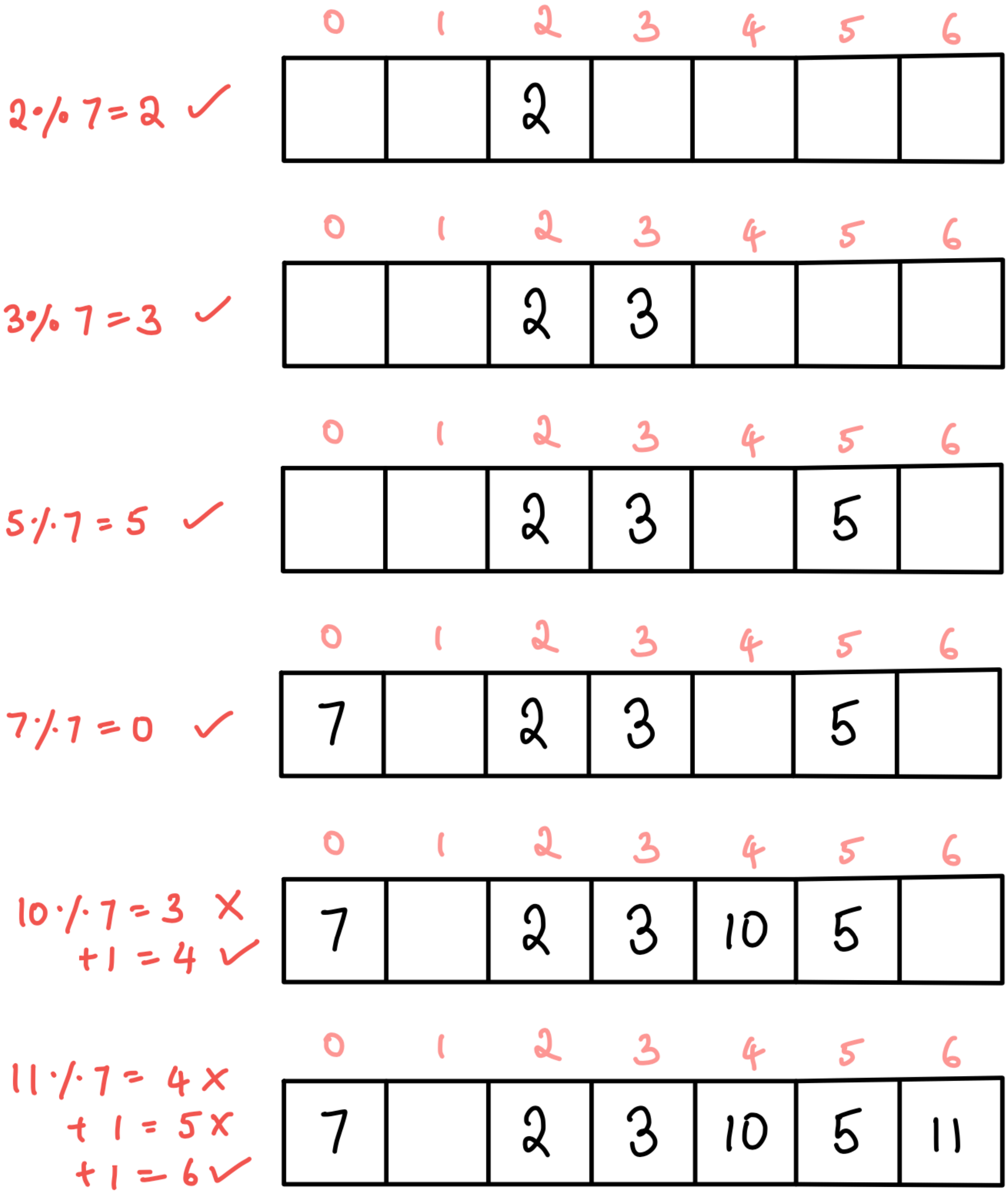
```c
void display_hash(HashTable *hashtable, int size) {
    /* Hash function: key % size */

    Node *temp = NULL;
    printf("\n");
    for (int i = 0; i < size; ++i) {
        temp = hashtable[i].head;
        printf("%d: ", i);
        for (int j = 0; j < hashtable[i].count; ++j, temp = temp->next) {
            printf("%d->", temp->key);
        }
        printf("NULL\n");
    }
    printf("\n");
}
```

Open Addressing

- All elements are stored in the hash table itself
- In case of collision, searches for a new empty spot

Linear Probing

- Linearly probe for next empty spot
- If $hash(key) = h$ is occupied, $h + 1$ is checked and so on until $(h + i)\ \%\ n$ is found to be empty
- Example: let $hash(key) = key\ \%\ 7$ and let the elements to insert be 2, 3, 5, 7, 10, 11



Challenges in Linear Probing

1. **Primary Clustering**: many consecutive elements form groups and time taken to find a free slot or to search an element increases
2. **Secondary clustering**: two records have the same collision chain (Probe Sequence) if their initial position is the same

Code Implementation

Structure for hash table

```c
typedef struct {
    int *table;     /* int array - hash table */
    int size;       /* size */
} HashTable;
```

Creating and Destroying the table

```c
HashTable *create_table(int size) {
    /* Initialise HashTable pointer */
    HashTable *temp = malloc(sizeof(HashTable));
    /* Initialise array of size size */
    temp->table = calloc(size, sizeof(int));

    /* Initialise to -1 (empty) */
    for (int i = 0; i < size; ++i) {
        temp->table[i] = -1;
    }
    temp->size = size;
    return temp;
}

void destroy_table(HashTable *htable) {
    /* Destroy hash table - free up int array */
    htable->size = 0;
    free(htable->table);
}
```

Insert an element

```c
void insert(HashTable *htable, int element) {
    int hash = element % htable->size;
    int count = 0;

    while (count < htable->size) {
        if (htable->table[hash] == -1) {
            /* Empty spot found */
            htable->table[hash] = element;
            break;
        }

        /* hash = hash + 1*/
        ++hash;
        ++count;
```

```c

        if (hash == htable->size) {
            hash = 0;
        }
    }
    /* Hash table is full */
}
```

Search for an element

```c
int search(HashTable *htable, int element) {
    int hash = element % htable->size;
    int count = 0;

    while (count < htable->size) {
        /* element found */
        if (htable->table[hash] == element) {
            return 1;
        }
        ++hash;
        ++count;
    }
    return 0;
}
```

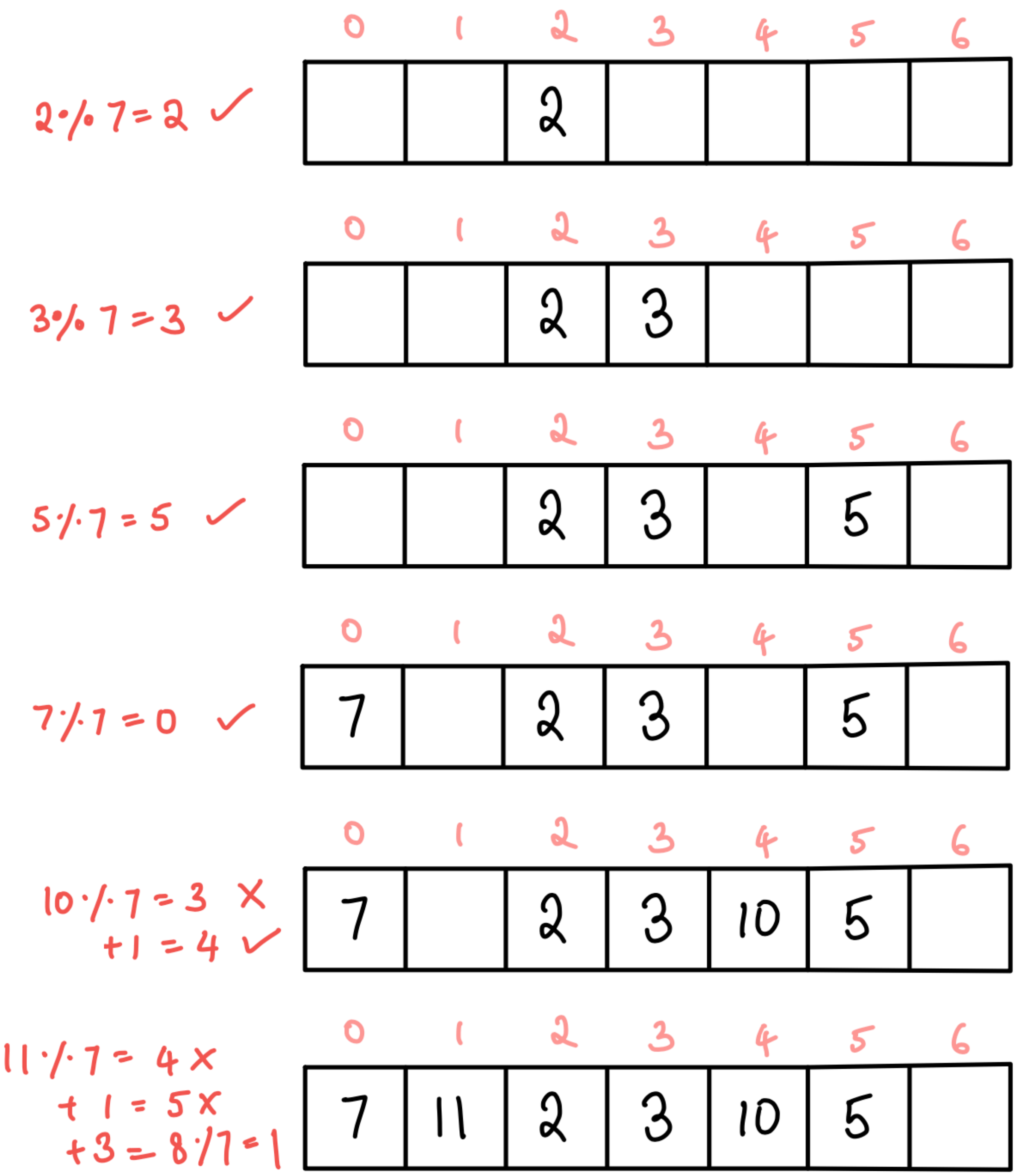
Delete an element

```c
void delete (HashTable *htable, int element) {
    int hash = element % htable->size;
    int count = 0;

    while (count < htable->size) {
        if (htable->table[hash] == element) {
            htable->table[hash] = -1;
            return;
        }
        ++hash;
        ++count;
    }
}
```

Quadratic Probing

- If $hash(key) = h$ is occupied, $h + 1^2$ is checked and so on until $(h + i^2)\ \%\ n$ is found to be empty
- The difference between consecutive squares is an odd number (the difference between consecutive numbers is 1, so every iteration we add 1 in linear probing)
- In quadratic probing, if $hash(key) = h$ is occupied, we add 1 and check, then add 3 and check, then 5, 7, 9 and so on (making sure to $\%\ n$). In this manner we visit all $(h + i^2)\ \%\ n$ elements

- Example: let $hash(key) = key\ \%\ 7$ and let the elements to insert be 2, 3, 5, 7, 10, 11 (same example)



Challenges in Linear Probing

1. Not all hash table slots will be visited, leaving some slots possibly empty but not checked

Double Hashing

- Applying a second hash function to key when a collision occurs
- If $hash(key) = h$ is occupied, $(h + 1 \times hash\_2(key)) \,\%\, n$ is checked and so on until $(h + i \times hash\_2(key)) \,\%\, n$ is found to be empty
- A popular second hash function is $hash\_2(key) = PRIME - (key \,\%\, PRIME)$ where $PRIME$ is a prime number smaller than table size n

Rehashing & Load Factor

- $load\ factor = \frac{number\ of\ records\ filled}{total\ capacity}$
- When the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases
- To overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity