CBCS SCHEME

USN | | | | | | | | | | |

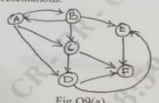
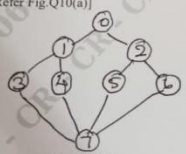
22MCA13

First Semester MCA Degree Examination, Dec.2023/Jan.2024
Data Structures with Algorithms

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	What are data structures? Explain the classifications of data structures.	08	L1	CO1
	b.	Write a 'C' program to convert postfix to infix expression.	08	L3	CO2
	c.	Evaluate the following postfix expression using stack: 5, 6, 2, +, *, 12, 4, 1, –	04	L3	CO2
		OR			
Q.2	a.	Write a 'C' program to convert infix to postfix expression using applications of stack.	10	L3	CO2
	b.	Define STACK. Write a C program to implement stack operations using arrays.	10	L3	CO2
		Module – 2			
Q.3	a.	Write a 'C' program to implement tower of Honai problem using recursion and trace the output for 3 disks.	10	L3	CO2
	b.	Write a 'C' recursive functions to implement GCD of 2 numbers and generating Fibonacci sequence.	10	L3	CO2
		OR			
Q.4	a.	Define circular queues. Write a 'C' program to implement circular queue operations.	10	L3	CO3
	b.	What are priority queues? Write a program to simulate priority queues with priority 1 and 2.	10	L3	CO3
		Module – 3			
Q.5	a.	What are Linked lists? Write a program to implement the following options: (i) Insert a node at the beginning of the list. (ii) Delete a node at the end of the list.	10	L3	CO3
	b.	Give an account of: (i) Static and dynamic memory allocation (ii) Getnode( ) and freenode( ) operations	10	L3	CO3
		OR			

Q.6	a.	Write a 'C' program to implement STACK operations using linked lists.	10	L3	CO3
	b.	Give an account of : (i) Memory management functions (ii) Array implementation of lists	10	L3	CO3
		Module – 4			
Q.7	a.	Explain the array and linked representation of binary trees with suitable examples.	06	L3	CO3
	b.	Construct the binary search tree for the following array items: 40, 60, 50, 33, 55, 11	06	L3	CO3
	c.	Write a C function to create binary search tree.	08	L3	CO3
		OR			
Q.8	a.	Explain binary tree traversal methods with 'C' functions and examples.	10	L3	CO3
	b.	Give an account of threaded binary trees.	10	L3	CO3
		Module – 5			
Q.9	a.	Define a graph. For a graph shown in Fig.Q9(a), write the adjacency matrix and adjacency list representations.	08	L3	CO3
		 Fig.Q9(a)			
	b.	Suppose an array contains 8 elements such as 77, 33, 44, 11, 88, 22, 66, 55. Sort the array using insertion sort algorithm.	08	L3	CO4
	c.	What is hashing? Explain any two hash functions with proper examples.	06	L3	CO4
		OR			
Q.10	a.	Briefly explain Breadth-First-Search (BFS) and Depth-First-Search (DFS) traversal of a graph. Also, show the BFS and DFS traversals for the following graphs. [Refer Fig.Q10(a)]	10	L3	CO4
		 Fig.Q10(a)			
	b.	Explain the working operation of Radix sort for the following set of data: 348, 143, 361, 423, 538, 128, 321, 543, 366	05	L3	CO4
	c.	Explain Address Calculation Sorting method with suitable example.	05	L3	CO4

* * * * *

CBCS SCHEME
22MCA13
First Semester MCA Degree Examination. Dec.2023/Jan.2024
Data Structures with Algorithms

Time: 3 hrs.
Max Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.

Q1. What are data structures? Explain the classifications of data structures.

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

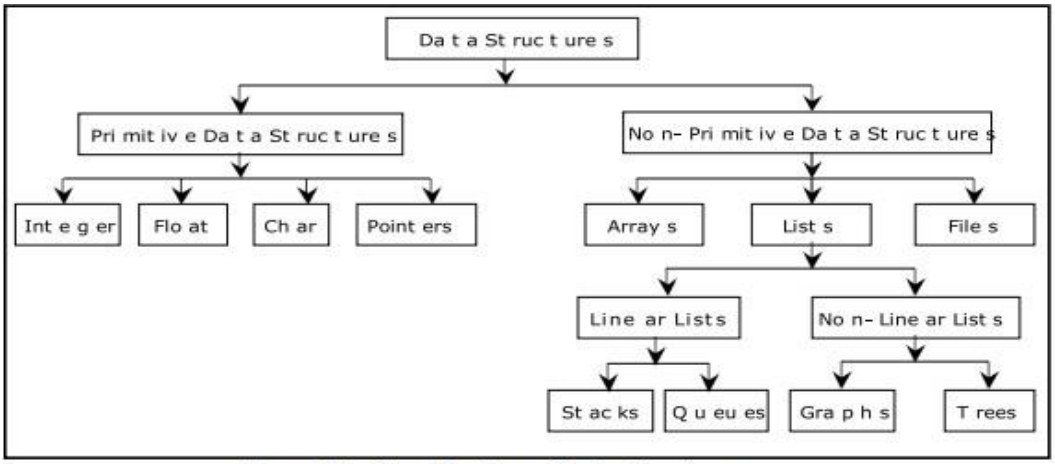


Fig ure 1. 1. C lass if icat io n of Da t a St ruct ures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matte how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non- contiguous structures.

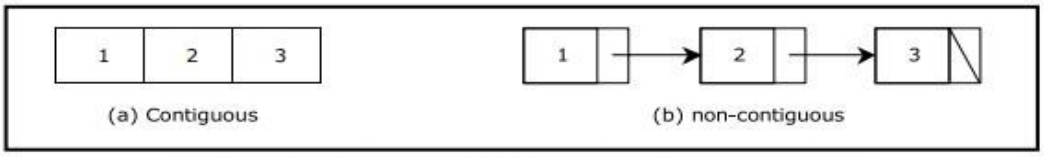


Figure 1.2 Contiguous and Non-contiguous structures compared

OR

Q2. Write a 'C' program to convert postfix to infix expression.

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

#define MAX 100

typedef struct {
   char data[MAX][MAX];
   int top;
} Stack;

void push(Stack *s, char *str) {
   strcpy(s->data[++(s->top)], str);
}
```

```
char* pop(Stack *s) {
   return s->data[(s->top)--];
}

void postfixToInfix(char* exp) {
   Stack s;
   s.top = -1;
   char temp[MAX];
   for (int i = 0; exp[i] != '\0'; i++) {
      if (isalnum(exp[i])) {
         temp[0] = exp[i];
         temp[1] = '\0';
         push(&s, temp);
      } else {
         char op1[MAX], op2[MAX];
         strcpy(op1, pop(&s));
         strcpy(op2, pop(&s));
         sprintf(temp, "(%s%c%s)", op2, exp[i], op1);
         push(&s, temp);
      }
   }
   printf("Infix Expression: %s\n", pop(&s));
}

int main() {
   char exp[MAX] = "ABC/-AK/L-*";
   postfixToInfix(exp);
   return 0;
}
```

Q3. Evaluate the following postfix expression using stack: 5, 6, 2, +, *, 12, 4, /, -

Invalid Expression

Q4. Define STACK. Write a 'C' program to implement stack operations using arrays.

STACK:
A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The insertion and deletion operations are performed at the same end called the "top" of the stack.

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

#define MAX 100

typedef struct {
   int data[MAX];
   int top;
} Stack;
```

```
void push(Stack *s, int value) {
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    s->data[++(s->top)] = value;
}

int pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return s->data[(s->top)--];
}

int peek(Stack *s) {
    if (s->top == -1) {
        printf("Stack is Empty\n");
        return -1;
    }
    return s->data[s->top];
}

int main() {
    Stack s;
    s.top = -1;

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    printf("Top element: %d\n", peek(&s));

    printf("Popped element: %d\n", pop(&s));
    printf("Popped element: %d\n", pop(&s));

    return 0;
}
```

Q5(a). What are Linked Lists? Write a program to implement the following options:

(i) Insert a node at the beginning of the list. (ii) Delete a node at the end of the list.

Linked Lists: A linked list is a linear data structure where each element is a separate object called a node. Each node contains two items: the data and a reference to the next node in the sequence. This structure allows for efficient insertion and deletion operations.

(i) Insert a Node at the Beginning of the List

```
void beginsert() { struct node *ptr; int
item; ptr = (struct node *) malloc(sizeof(struct
node *));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter value\n");
scanf("%d",&item);
ptr->data = item; ptr-
>next = head; head = ptr;
printf("\nNode inserted");
}
```

(ii) Delete a node at the end of the list.

```
last_delete() { struct
node *ptr,*ptr1;
if(head == NULL)
{
printf("\nlist is empty");
}
else if(head -> next == NULL)
{
head = NULL; free(head); printf("\nOnly
node of the list deleted ...\n");
}
else
{
ptr = head;
while(ptr->next != NULL)
{
ptr1 = ptr;
ptr = ptr ->next;
}
ptr1->next = NULL;
free(ptr); printf("\nDeleted Node from the
last ...\n");
}
}
```

Q6

a. Write a 'C' program to implement STACK operations using linked lists.

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

// Define the structure for the stack node
struct Node {
```

```
    int data;
    struct Node* next;
};

struct Node* top = NULL;

// Function to push an element onto the stack
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed to stack\n", value);
}

// Function to pop an element from the stack
int pop() {
    if (top == NULL) {
        printf("Stack underflow\n");
        return -1;
    }
    int value = top->data;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return value;
}

// Function to display the elements of the stack
void display() {
    struct Node* temp = top;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    printf("Popped element: %d\n", pop());
    display();
    return 0;
}
```

b. Give an account of:

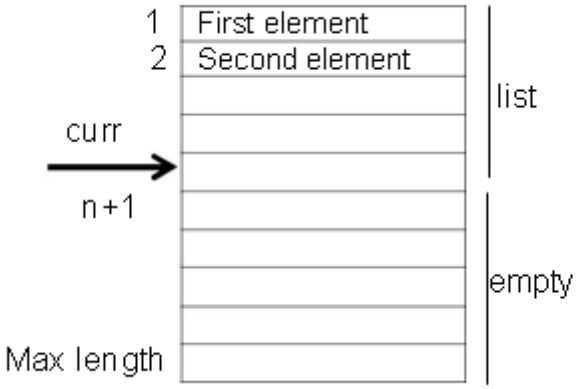
i) Memory management functions**

Memory management functions in C include:
- `malloc()`: Allocates a block of memory.
- `calloc()`: Allocates memory for an array and initializes all bytes to zero.
- `realloc()`: Reallocates a block of memory to a new size.
- `free()`: Frees a previously allocated block of memory.

(ii) Array implementation of lists

An array implementation of lists involves using a contiguous block of memory to store list elements. Each element is accessed using its index. The main disadvantage is the fixed size, requiring resizing when the list grows.
The simplest method to implement a List ADT is to use an array that is a“linear list” or a “contiguous list” where elements are stored in contiguous array positions. The implementation specifies an array of a particular maximum length, and all storage is allocated before run-time. It is a sequence of n-elements where the items in the array are stored with the index of the array related to the position of the item in the list.

In array implementation,elements are stored in contiguous array positions (Figure 3.1). An array is a viable choice for storing list elements when the elements are sequential, because it is a commonly available data type and in general algorithm development is easy.



List Implementation using arrays

In order to implement lists using arrays we need an array for storing entries – **listArray[0,1,2…..m],** a variable **curr**to keep track of the number of array elements currently assigned to the list**,** thenumber of items in the list, or current size of the list **size**and a variable to record the maximum length of the array, and therefore of the list .

Advantages of Array-Based Implementation of Lists

Some of the major advantages of using array implementation of lists are:

- Array is a natural way to implement lists
- Arrays allow fast, random access of elements
- Array based implementation is memory efficient since very little memory is required other than that needed to store the actual contents

Some of the disadvantages of using arrays to implement lists are:

- The size of the list must be known when the array is created and is fixed (static)
- Array implementations of lists use a static data structure. Often defined at compile-time. This means the array size or structure cannot be altered while program is running. This requires an accurate estimate of the size of the array.
- This fixing of the size beforehand usually results in overestimation of size which means we tend to usually waste space rather than have program run out.
- The deletion and insertion of elements into the list is slow since it involves shifting of elements. It also means that data must be added to the end of the list for insertion and deletion to be efficient. If insertion and deletion is towards the front of the list, all other elements must shuffle down. This is slow and inefficient. This inefficiency is even more pronounced when the size of the list is large.

Fix one end of the list at index 0 and elements will be shifted***as needed*** when an element is added or removed. Therefore insert and delete operations will take O(n). That is if we want to insert at position 0 (insert as first element),this requires first pushing the entire array to the right one spot to make room for the new element. Similarly deleting the element at position 0requires shifting all the elements in the list left one spot.

Q7 a. **Explain the array and linked representation of binary trees with suitable examples.**

Array Representation:
In array representation, a binary tree is stored in a single-dimensional array. The root element is at index 0. For any node at index `i`, its left child is at `2i + 1` and its right child is at `2i + 2`.

Example:

```
      1
     / \
    2   3
   / \
  4   5
```

Array: `[1, 2, 3, 4, 5]`

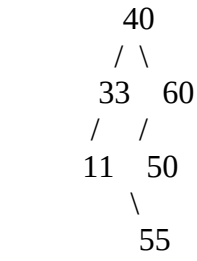
Linked Representation:
In linked representation, each node of the binary tree is represented as a structure containing data, a pointer to the left child, and a pointer to the right child.

```
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};
```

b. Construct the binary search tree for the following array items: 40, 60, 50, 33, 55, 11

1. Insert 40 (root)

2. Insert 60 (right of 40)
3. Insert 50 (left of 60)
4. Insert 33 (left of 40)
5. Insert 55 (right of 50)
6. Insert 11 (left of 33)



c. **Write a C function to create a binary search tree.**

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
```

```
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    int elements[] = {40, 60, 50, 33, 55, 11};
    for (int i = 0; i < 6; i++) {
        root = insert(root, elements[i]);
    }
    inorderTraversal(root);
    return 0;
}
```

Q8

a. **Explain binary tree traversal methods with 'C' functions and examples.**

Binary tree traversal methods include:
-Inorder Traversal (Left, Root, Right)
-Preorder Traversal (Root, Left, Right)
- Postorder Traversal (Left, Right, Root)

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

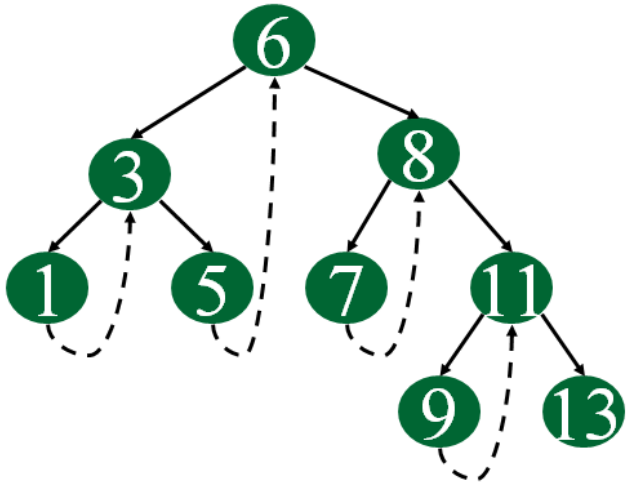
```
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder Traversal: ");
    inorder(root);
    printf("\nPreorder Traversal: ");
    preorder(root);
    printf("\nPostorder Traversal: ");
    postorder(root);
    return 0;
}
```

b. **Give an account of threaded binary trees**.

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.
Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if

successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
The threads are also useful for fast accessing ancestors of a node.
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



.
Q9
a. **Define a graph. For a graph shown in Fig.Q9(a), write the adjacency matrix and adjacency list representations**.

A graph is a collection of nodes (vertices) and edges connecting pairs of nodes.

For the graph in Fig.Q9(a):

Adjacency Matrix:

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	1	0	0
C	0	0	0	1	0	0
D	0	0	0	0	1	1
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Adjacency List:
A -> B -> C
B -> D
C -> D
D -> E -> F
E -> F
F

b. Suppose an array contains 8 elements such as 77, 33, 44, 11, 88, 22, 66, 55. Sort the array using insertion sort algorithm.

Initial array: 77,33,44,11,88,22,66,5577, 33, 44, 11, 88, 22, 66, 5577,33,44,11,88,22,66,55
Iteration 1 (key = 33): 33,77,44,11,88,22,66,5533, 77, 44, 11, 88, 22, 66, 5533,77,44,11,88,22,66,55
Iteration 2 (key = 44): 33,44,77,11,88,22,66,5533, 44, 77, 11, 88, 22, 66, 5533,44,77,11,88,22,66,55
Iteration 3 (key = 11): 11,33,44,77,88,22,66,5511, 33, 44, 77, 88, 22, 66, 5511,33,44,77,88,22,66,55
Iteration 4 (key = 88): 11,33,44,77,88,22,66,5511, 33, 44, 77, 88, 22, 66, 5511,33,44,77,88,22,66,55
Iteration 5 (key = 22): 11,22,33,44,77,88,66,5511, 22, 33, 44, 77, 88, 66, 5511,22,33,44,77,88,66,55
Iteration 6 (key = 66): 11,22,33,44,66,77,88,5511, 22, 33, 44, 66, 77, 88, 5511,22,33,44,66,77,88,55
Iteration 7 (key = 55): 11,22,33,44,55,66,77,8811, 22, 33, 44, 55, 66, 77, 8811,22,33,44,55,66,77,88

c. What is hashing? Explain any two hash functions with proper examples.

Hashing is a technique used to uniquely identify a specific object from a group of similar objects. In computer science, hashing is used to convert data (such as strings or numbers) into a fixed-size hash value (or hash code) using a hash function. This hash value can then be used to index and retrieve items in a database because it is faster to find the shorter hash value than the original value.

Hash Functions

A hash function takes an input (or 'message') and returns a fixed-size string of bytes. The output is typically a 'digest' that is unique to each unique input.

Example 1: Division Method

One of the simplest hash functions is the division method. It involves dividing the key by a number (often a prime number) and taking the remainder as the hash value.

Formula: hash=keymod table_size\text{hash} = \text{key} \mod \text{table_size}hash=keymodtable_size

Example:

- Suppose the keys are 50, 700, 76, 85, 92 and the table size is 10.
- For key = 50, hash value = 50 % 10 = 0
- For key = 700, hash value = 700 % 10 = 0
- For key = 76, hash value = 76 % 10 = 6
- For key = 85, hash value = 85 % 10 = 5
- For key = 92, hash value = 92 % 10 = 2

Example 2: Multiplication Method

The multiplication method involves multiplying the key by a constant fractional value, extracting the fractional part of the result, and then multiplying that fractional part by the table size.

Formula: hash=⌊table_size×(kAmod 1)⌋\text{hash} = \left\lfloor \text{table_size} \times (kA \mod 1) \right\rfloorhash=⌊table_size×(kAmod1)⌋ where kkk is the key, AAA is a constant (0 < A < 1), and ⌊⋅⌋\left\lfloor \cdot \right\rfloor⌊⋅⌋ denotes the floor function.

Example:

- Suppose the key is 50, the table size is 10, and AAA is 0.618033 (which is roughly the golden ratio).
- First, calculate kAmod 1kA \mod 1kAmod1: 50×0.618033≈30.9016550 \times 0.618033 \approx 30.9016550×0.618033≈30.90165

- Extract the fractional part: 0.90165
- Multiply by the table size: 0.90165×10=9.01650.90165 \times 10 = 9.01650.90165×10=9.0165
- Apply the floor function: ⌊9.0165⌋=9\left\lfloor 9.0165 \right\rfloor = 9⌊9.0165⌋=9

Thus, the hash value for the key 50 is 9.