Searching Algorithms

Searching algorithms are methods or procedures used to find a specific item or element within a collection of data. These algorithms are widely used in computer science and are crucial for tasks like searching for a particular record in a database, finding an element in a sorted list, or locating a file on a computer.

These are some commonly used searching algorithms:

1. **Linear Search:** In this simple algorithm, each element in the collection is sequentially checked until the desired item is found, or the entire list is traversed. It is suitable for small-sized or unsorted lists, but its time complexity is O(n) in the worst case.
2. **Binary Search:** This algorithm is applicable only to sorted lists. It repeatedly compares the middle element of the list with the target element and narrows down the search range by half based on the comparison result. Binary search has a time complexity of O(log n), making it highly efficient for large sorted lists.
3. **Hashing:** Hashing algorithms use a hash function to convert the search key into an index or address of an array (known as a hash table). This allows for constant-time retrieval of the desired item if the hash function is well-distributed and collisions are handled appropriately. Common hashing techniques include direct addressing, separate chaining, and open addressing.
4. **Interpolation Search:** Similar to binary search, interpolation search works on sorted lists. Instead of always dividing the search range in half, interpolation search uses the value of the target element and the values of the endpoints to estimate its approximate position within the list. This estimation helps in quickly narrowing down the search space. The time complexity of interpolation search is typically O(log log n) on average if the data is uniformly distributed.
5. **Tree-based Searching:** Various tree data structures, such as binary search trees (BST), AVL trees, or B-trees, can be used for efficient searching. These structures impose an ordering on the elements and provide fast search, insertion, and deletion operations. The time complexity of tree-based searching algorithms depends on the height of the tree and can range from O(log n) to O(n) in the worst case.
6. **Ternary Search:** Ternary search is an algorithm that operates on sorted lists and repeatedly divides the search range into three parts instead of two, based

on two splitting points. It is a divide-and-conquer approach and has a time complexity of O($\log_3$ n).

7. **Jump Search:** Jump search is an algorithm for sorted lists that works by jumping ahead a fixed number of steps and then performing linear search in the reduced subarray. It is useful for large sorted arrays and has a time complexity of O(sqrt(n)), where n is the size of the array.
8. **Exponential Search:** Exponential search is a technique that combines elements of binary search and linear search. It begins with a small range and doubles the search range until the target element is within the range. It then performs a binary search within that range. Exponential search is advantageous when the target element is likely to be found near the beginning of the array and has a time complexity of O(log n).
9. **Fibonacci Search:** Fibonacci search is a searching algorithm that uses Fibonacci numbers to divide the search space. It works on sorted arrays and has a similar approach to binary search, but instead of dividing the array into halves, it divides it into two parts using Fibonacci numbers as indices. Fibonacci search has a time complexity of O(log n).
10. **Interpolation Search for Trees:** This algorithm is an extension of interpolation search designed for tree structures such as AVL trees or Red-Black trees. It combines interpolation search principles with tree traversal to efficiently locate elements in the tree based on their values. The time complexity depends on the tree structure and can range from O(log n) to O(n) in the worst case.
11. **Hash-based Searching (e.g., Bloom Filter):** Hash-based searching algorithms utilize hash functions and data structures like Bloom filters to determine whether an element is present in a set or not. These algorithms provide probabilistic answers, meaning they can occasionally have false positives (indicating an element is present when it is not), but no false negatives (if an element is not present, it will never claim it is). Bloom filters have a constant-time complexity for search operations.
12. **String Searching Algorithms:** Searching algorithms specific to string data include techniques like Knuth-Morris-Pratt (KMP) algorithm, Boyer-Moore algorithm, Rabin-Karp algorithm, and many others. These algorithms optimize the search for patterns within text or strings and are widely used in text processing, pattern matching, and string matching tasks.

Implementation of Searching Algorithms

Linear Search:

Code in C++:

```cpp
#include <iostream>
#include <vector>

int linearSearch(const std::vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            return i; // Return the index of the element
        }
    }
    return -1; // If the target is not found, return -1
}

int main() {
    std::vector<int> myVector = {4, 7, 2, 1, 9, 5};
    int targetValue = 9;

    int result = linearSearch(myVector, targetValue);
    if (result != -1) {
        std::cout << "The target value " << targetValue << " is found at index " << result << "." << std::endl;
    } else {
        std::cout << "The target value is not present in the vector." << std::endl;
    }

    return 0;
}
```

Output:

```
The target value 9 is found at index 4.
```

Code in C:

```c
#include <stdio.h>

```

```
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index of the element
        }
    }
    return -1; // If the target is not found, return -1
}

int main() {
    int myArray[] = {4, 7, 2, 1, 9, 5};
    int targetValue = 9;
    int size = sizeof(myArray) / sizeof(myArray[0]);

    int result = linearSearch(myArray, size, targetValue);
    if (result != -1) {
        printf("The target value %d is found at index %d.\n", targetValue, result);
    } else {
        printf("The target value is not present in the array.\n");
    }

    return 0;
}
```

Output:

```
The target value 9 is found at index 4.
```

Explanation:

1. The linearSearch function takes three parameters: arr (the array), size (the size of the array), and target (the value to be searched).
2. Inside the linearSearch function, a for loop is used to iterate through each element of the array. The loop variable i is initialized to 0, and the loop continues as long as i is less than the size of the array.
3. Inside the loop, each element of the array is compared with the target value using the expression arr[i] == target. If a match is found, the function

immediately returns the index i, indicating the position of the element in the array.

4. If the loop finishes without finding a match, the function returns -1, indicating that the target value was not found in the array.
5. In the main function, an example array myArray is created and initialized with values. The target value is set to 9.
6. The size of the array is calculated by dividing the total size of the array (sizeof(myArray)) by the size of an individual element (sizeof(myArray[0])). This ensures that the size of the array is correctly passed to the linearSearch function.
7. The linearSearch function is called with the array, size, and target value as arguments. The return value is stored in the result variable.
8. If the result is not equal to -1, it means that the target value was found in the array. In this case, a message is printed using printf() to indicate the index at which the target value was found.
9. If the result is -1, it means that the target value was not found in the array. In this case, a message is printed to indicate that the target value is not present in the array.
10. The program terminates by returning 0 from the main function.

Binary Search:

Recursive Code

```
// Binary Search in C

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
```

```c
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Output:

```
Element is found at index 1
```

Explanation:

- The binarySearch function takes an array, the element to search for (x), the lower index (low), and the higher index (high) as parameters.
- It first checks if high is greater than or equal to low. If not, it means the element was not found, and it returns -1.
- If high is greater than or equal to low, it calculates the middle index mid using the formula low + (high - low) / 2.
- It then checks if the element at the middle index (array[mid]) is equal to x. If they are equal, it means the element is found, and it returns the index mid.
- If array[mid] is greater than x, it means the element lies in the left half of the array. In this case, the binarySearch function is called recursively with low unchanged and high set to mid - 1.

- If array[mid] is less than x, it means the element lies in the right half of the array. In this case, the binarySearch function is called recursively with low set to mid + 1 and high unchanged.
- If the element is not found after the recursive calls, the function returns -1.
- In the main function, an array {3, 4, 5, 6, 7, 8, 9} is defined, and the size of the array n is calculated using the formula sizeof(array) / sizeof(array[0]).
- The element x to search for is set to 4.
- The binarySearch function is called with the array, x, and the range of indices 0 to n - 1.
- The returned result is stored in the result variable.
- Finally, the result is checked. If it is -1, it means the element was not found, and "Not found" is printed. Otherwise, the index where the element was found is printed as "Element is found at index result".

Iterative

```c
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

```

```c
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Output:

```
Element is found at index 1
```

Explanation:

- The binarySearch function takes an array, the element to search for (x), the lower index (low), and the higher index (high) as parameters.
- It uses a while loop that continues until the low pointer is less than or equal to the high pointer.
- Inside the loop, it calculates the middle index mid using the formula low + (high - low) / 2.
- It then checks if the element at the middle index (array[mid]) is equal to x. If they are equal, it means the element is found, and it returns the index mid.
- If array[mid] is less than x, it means the element lies in the right half of the array. In this case, the low pointer is updated to mid + 1.
- If array[mid] is greater than x, it means the element lies in the left half of the array. In this case, the high pointer is updated to mid - 1.
- If the element is not found after the loop, it means the element is not present in the array, and the function returns -1.
- In the main function, an array {3, 4, 5, 6, 7, 8, 9} is defined, and the size of the array n is calculated using the formula sizeof(array) / sizeof(array[0]).
- The element x to search for is set to 4.
- The binarySearch function is called with the array, x, and the range of indices 0 to n - 1.

- The returned result is stored in the result variable.
- Finally, the result is checked. If it is -1, it means the element was not found, and "Not found" is printed. Otherwise, the index where the element was found is printed as "Element is found at index result".

Hashing

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

#define TABLE_SIZE 10

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

struct Node* table[TABLE_SIZE];

int hash_function(int key) {
    return key % TABLE_SIZE;
}

void insert(int key, int value) {
    int index = hash_function(key);

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->value = value;
    newNode->next = NULL;

    if (table[index] == NULL) {
        table[index] = newNode;
    } else {
        struct Node* current = table[index];
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

int search(int key) {
    int index = hash_function(key);

    struct Node* current = table[index];
    while (current != NULL) {
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }

    return -1;  // Key not found
}

int main() {
    // Initialize the hash table
    for (int i = 0; i < TABLE_SIZE; i++) {
        table[i] = NULL;
    }

    // Insert elements into the hash table
    insert(5, 100);
    insert(2, 200);
    insert(7, 300);

    // Search for a key in the hash table
    int searchKey = 2;
    int result = search(searchKey);
    if (result != -1) {
        printf("Value for key %d: %d\n", searchKey, result);
    } else {
        printf("Key %d not found\n", searchKey);
    }

    return 0;
}
```

Output:

```
Value for key 2: 200
```

Explanation:

1. The C implementation starts by including the necessary header files: stdio.h for standard input/output operations and stdlib.h for memory allocation.
2. A constant TABLE_SIZE is defined to determine the size of the hash table array.
3. A struct Node is defined to represent a key-value pair. It has three members: key, value, and next. The next member is a pointer to the next node in case of collisions.
4. A global variable table is declared as an array of struct Node* to store the hash table.
5. The hash_function function takes a key as input and returns the index in the hash table array. It uses the modulo operator % with TABLE_SIZE to calculate the index.
6. The insert function takes a key and value as input. It calculates the index using the hash_function and inserts a new node containing the key-value pair into the linked list at the calculated index.
7. The search function takes a key as input and returns the value associated with the key if found in the hash table. It calculates the index using the hash_function and traverses the linked list at the calculated index to search for the key.
8. The main function initializes the hash table, inserts elements into it, and performs a search operation.

Code in Java:

```java
import java.util.ArrayList;
import java.util.List;

class Node {
    int key;
    int value;

    Node(int key, int value) {
        this.key = key;
        this.value = value;
    }
}

class HashTable {
    private int size;
    private List<List<Node>> table;

    HashTable(int size) {
        this.size = size;
        this.table = new ArrayList<>(size);
        for (int i = 0; i < size; i++) {
            table.add(new ArrayList<>());
        }
    }

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

    void insert(int key, int value) {
        int index = hashFunction(key);
        List<Node> list = table.get(index);
        list.add(new Node(key, value));
    }

    int search(int key) {
        int index = hashFunction(key);
        List<Node> list = table.get(index);
        for (Node node : list) {
            if (node.key == key) {
                return node.value;
            }
        }
        return -1;  // Key not found
    }
}

public class Main {
    public static void main(String[] args) {
        // Initialize the hash table
        int tableSize = 10;
        HashTable hashTable = new HashTable(tableSize);

        // Insert elements into the hash table
        hashTable.insert(5, 100);
        hashTable.insert(2, 200);
        hashTable.insert(7, 300);

        // Search for a key in the hash table
        int searchKey = 2;
        int result = hashTable.search(searchKey);
        if (result != -1) {
            System.out.println("Value for key " + searchKey + ": " + result);
        } else {
            System.out.println("Key " + searchKey + " not found");
        }
    }
}
```

Output:

```
Value for key 2: 200
```

Explanation:

1. The Java implementation defines a Node class with key and value fields to represent a key-value pair.
2. The HashTable class encapsulates the hash table functionality.

3. The class has a constructor that takes the size of the hash table as input and initializes an ArrayList of empty lists to represent the buckets.
4. The hashFunction method takes a key as input and returns the index in the hash table array. It uses the division method (key % size) to calculate the index.
5. The insert method takes a key and value as input. It calculates the index using the hashFunction and inserts a new Node object into the list at the calculated index.
6. The search method takes a key as input and returns the value associated with the key if found in the hash table. It calculates the index using the hashFunction and iterates over the list at the calculated index to search for the key.
7. The main method creates an instance of HashTable, inserts elements into it, and performs a search operation.

Interploation Search:

```
#include <stdio.h>

int interpolationSearch(int arr[], int n, int target) {
   int low = 0, high = n - 1;

   while (low <= high && target >= arr[low] && target <= arr[high]) {
      if (low == high) {
         if (arr[low] == target)
            return low;
         return -1;
      }

      // Calculate the position using interpolation formula
      int pos = low + (((double)(high - low) / (arr[high] - arr[low])) * (target - ar
r[low]));

      if (arr[pos] == target)
         return pos;

      if (arr[pos] < target)
         low = pos + 1;
      else
```

```c
            high = pos - 1;
    }

    return -1; // Element not found
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 16;

    int index = interpolationSearch(arr, n, target);

    if (index != -1)
        printf("Element %d found at index %d\n", target, index);
    else
        printf("Element %d not found in the array\n", target);

    return 0;
}
```

Output:

```
Element 16 found at index 4
```

Explanation:

1. The interpolation search algorithm assumes that we have an input array in sorted in ascending order.
2. The algorithm consists of two variables: They are low, which points to the first index of the array, and high, initialized to the last index of the array.
3. The algorithm uses the target values arr[low] and arr[high] to calculate the probable location of the target element in the array where the probability of finding an element is high. It uses the projection formula: *pos = low + (((double) (high - low) / (arr[high] - arr[low])) * (value - arr[low])* The above given formula calculates the pos position as the weighted average of the lower and upper indices, depending on the values of the corresponding array elements. The objective is to calculate the location of the target detection.

4. The algorithm then compares the target value with arr[pos] to determine the next step:
    - If arr[pos] is equal to the given target value, then the algorithm searches for the target element and returns pos.
    - If arr[pos] is less than the given target value, the algorithm does not create another pos + 1, because the target value is likely at the top of the array.
    - If arr[pos] is greater than the target value, the algorithm updates up at pos - 1, since the target value is likely in the lower half of the array.
5. The algorithm repeats steps 3 and 4 until one of the following conditions is satisfied:
   The target element is found at position pos, in which case the algorithm returns the position.
   The low value is greater than high, indicating that the target is not in the system. In this case, the algorithm returns -1 indicating that the element was not found.
6. The main function in the example demonstrates the usage of the interpolationSearch function. It initializes an array, its size, and a target value. It then calls the interpolationSearch function to search for the target value in the array.
7. Finally, based on the returned index, the main function prints a corresponding message indicating whether the target element was found or not.

Code in Java:

```java
public class InterpolationSearch {
    public static int interpolationSearch(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high && target >= arr[low] && target <= arr[high]) {
            if (low == high) {
                if (arr[low] == target)
                    return low;
                return -1;
            }

            // Calculate the position using interpolation formula
            int pos = low + (((target - arr[low]) * (high - low)) / (arr[high] - arr[low])
);

            if (arr[pos] == target)
                return pos;

            if (arr[pos] < target)
                low = pos + 1;
            else
                high = pos - 1;
        }

        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int target = 16;

        int index = interpolationSearch(arr, target);

        if (index != -1)
            System.out.println("Element " + target + " found at index " + index);
        else
            System.out.println("Element " + target + " not found in the array");
    }
}
```

Output:

```
Element 16 found at index 4
```

Tree based searching:

Depth-First Search (DFS): DFS is a recursive search technique that explores the tree by going as deep as possible before backtracking. In the case of a binary tree, there are three common variants of DFS:

- **Pre-order traversal:** Visit the current node, then recursively visit the left subtree, and finally the right subtree.
- **In-order traversal:** Recursively visit the left subtree, visit the current node, and then recursively visit the right subtree. In a binary search tree, this traversal visits the nodes in ascending order.
- **Post-order traversal:** Recursively visit the left subtree, then the right subtree, and finally visit the current node.

Breadth-First Search (BFS): BFS explores the tree level by level, visiting all the nodes at the same depth before moving to the next level. It uses a queue data structure to keep track of the nodes to visit. BFS ensures that nodes closer to the root are visited before deeper nodes.

DFS:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX_NODES 100

// Structure to represent a graph node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
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;
}

// Depth-First Search (DFS) recursive function
void DFS(struct Node* node) {
    if (node == NULL) {
        return;
    }
    printf("%d ", node->data);  // Process the current node
    DFS(node->left);  // Recursively visit the left subtree
    DFS(node->right); // Recursively visit the right subtree
}

int main() {
    // Create a sample binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Depth-First Traversal (DFS): ");
    DFS(root);

    return 0;
}
```

Output:

```
Depth-First Traversal (DFS): 1 2 4 5 3
```

Explanation:

- The necessary header files stdio.h and stdbool.h are included, which provide input/output functionality and support for boolean values, respectively.
- The constant MAX_NODES is defined with a value of 100, representing the maximum number of nodes in the binary tree. This value is not currently used in the code.
- The structure Node is defined to represent a node in the binary tree. It contains an integer data to store the node's value, and two pointers left and right pointing to the left and right child nodes, respectively.
- The function createNode is a utility function that takes an integer data as input and creates a new node with the given data value. It allocates memory for the

new node using malloc and initializes the left and right pointers to NULL. The function returns a pointer to the newly created node.

- The function DFS is the recursive implementation of Depth-First Search. It takes a pointer to a node as input and performs the DFS traversal. The base case is when the current node is NULL, in which case the function simply returns. Otherwise, it processes the current node by printing its data value using printf. Then, it recursively calls DFS on the left subtree (node->left) and the right subtree (node->right).
- In the main function, a sample binary tree is created. The root node is created using createNode with a data value of 1. Two child nodes are created for the root node with data values 2 and 3, respectively. Further, two child nodes are created for the left child of the root node with data values 4 and 5, respectively.
- Finally, the DFS traversal is performed on the binary tree starting from the root node by calling DFS(root). The values of the nodes are printed in the depth-first order.

BFS:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX_NODES 100

// Structure to represent a graph node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Structure to represent a queue
struct Queue {
    struct Node* items[MAX_NODES];
    int front;
    int rear;
};

```

```c
// Function to create a new node
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;
}

// Function to create an empty queue
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

// Function to check if the queue is empty
bool isEmpty(struct Queue* queue) {
    return queue->rear == -1;
}

// Function to enqueue an item
void enqueue(struct Queue* queue, struct Node* item) {
    if (queue->rear == MAX_NODES - 1) {
        printf("Queue Overflow!\n");
        return;
    }
    if (queue->front == -1) {
        queue->front = 0;
    }
    queue->rear++;
    queue->items[queue->rear] = item;
}

// Function to dequeue an item
struct Node* dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow!\n");
        return NULL;
    }
    struct Node* item = queue->items[queue->front];
    queue->front++;
    if (queue->front > queue->rear) {
        queue->front = queue->rear = -1;
    }
    return item;
}

// Breadth-First Search (BFS) iterative function
void BFS(struct Node* root) {
    if (root == NULL) {
        return;
    }
    struct Queue* queue = createQueue();
    enqueue(queue, root);  // Enqueue the root node

    printf("Breadth-First Traversal (BFS): ");
    while (!isEmpty(queue)) {
        struct Node* node = dequeue(queue);
        printf("%d ", node->data);  // Process the current node

        // Enqueue the left and right child nodes if they exist
        if (node->left != NULL) {
            enqueue(queue, node->left);
        }
        if (node->right != NULL) {
            enqueue(queue, node->right);
        }
    }
    printf("\n");
}

int main() {
    // Create a sample binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

            BFS(root);

            return 0;
        }
```

Output:

```
Breadth-First Traversal (BFS): 1 2 3 4 5
```

Explanation:

- The necessary header files stdio.h and stdbool.h are included, which provide input/output functionality and support for boolean values, respectively.
- The constant MAX_NODES is defined with a value of 100, representing the maximum number of nodes in the binary tree. This value is used to define the size of the queue in the Queue structure.
- The structure Node is defined to represent a node in the binary tree. It contains an integer data to store the node's value, and two pointers left and right pointing to the left and right child nodes, respectively.
- The structure Queue is defined to represent a queue that will be used for BFS. It contains an array items of Node pointers to store the nodes in the queue, and two integers front and rear to keep track of the front and rear indices of the queue.
- The function createNode is a utility function that takes an integer data as input and creates a new node with the given data value. It allocates memory for the new node using malloc and initializes the left and right pointers to NULL. The function returns a pointer to the newly created node.
- The function createQueue creates an empty queue by allocating memory for the Queue structure and initializing the front and rear indices to -1.

- The function isEmpty checks if the queue is empty by checking if the rear index is -1. If it is, the function returns true, indicating that the queue is empty.
- The function enqueue adds a node to the queue. It first checks if the queue is already full (rear index is equal to MAX_NODES - 1). If so, it prints a "Queue Overflow!" message and returns. Otherwise, it increments the rear index, adds the node to the items array at the new rear index, and updates the front index if it was -1 (indicating an empty queue).
- The function dequeue removes and returns a node from the queue. It first checks if the queue is empty by calling isEmpty. If the queue is empty, it prints a "Queue Underflow!" message and returns NULL. Otherwise, it retrieves the node from the front index, increments the front index, and checks if the front index has surpassed the rear index. If so, it sets both the front and rear indices to -1, indicating an empty queue. Finally, it returns the dequeued node.
- The function BFS is the iterative implementation of Breadth-First Search. It takes a pointer to the root node of the binary tree as input. It starts by creating a queue using createQueue and enqueues the root node. Then, it enters a loop where it dequeues a node from the queue, processes the node by printing its data value using printf, and enqueues its left and right child nodes if they exist. The loop continues until the queue becomes empty, i.e., there are no more nodes to process.
- In the main function, a sample binary tree is created. The root node is created using createNode with a data value of 1. Two child nodes are created for the root node with data values 2 and 3, respectively. Further, two child nodes are created for the left child of the root node with data values 4 and 5, respectively.
- Finally, the BFS traversal is performed on the binary tree starting from the root node by calling BFS(root). The values of the nodes are printed in the breadth-first order.

Ternary Search:

```
#include <stdio.h>

// Function to perform ternary search
int ternarySearch(int arr[], int left, int right, int key)
{
    if (right >= left) {
```

```c
        int mid1 = left + (right - left) / 3;
        int mid2 = right - (right - left) / 3;

        // If the key is present at any of the mid points
        if (arr[mid1] == key)
            return mid1;

        if (arr[mid2] == key)
            return mid2;

        // If the key is smaller than the mid1 element, recur on the left segment
        if (key < arr[mid1])
            return ternarySearch(arr, left, mid1 - 1, key);

        // If the key is greater than the mid2 element, recur on the right segment
        if (key > arr[mid2])
            return ternarySearch(arr, mid2 + 1, right, key);

        // If the key is between mid1 and mid2, recur on the middle segment
        return ternarySearch(arr, mid1 + 1, mid2 - 1, key);
    }

    // Key not found
    return -1;
}

// Test the ternarySearch function
int main()
{
    int arr[] = { 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 23;
    int result = ternarySearch(arr, 0, n - 1, key);
    if (result == -1)
        printf("Element not found in the array.\n");
    else
```

```
        printf("Element found at index %d.\n", result);

    return 0;
}
```

Output:

```
Element found at index 5.
```

Explanation:

- In this example, we have an array arr containing some elements, and we want to search for a key value (in this case, 23). The ternarySearch function takes the array, the left and right indices of the current segment, and the key to search for.
- The function recursively divides the array into three segments: mid1 = left + (right - left) / 3 and mid2 = right - (right - left) / 3. It checks if the key is equal to arr[mid1] or arr[mid2]. If either condition is true, it returns the respective index.
- If the key is smaller than arr[mid1], the function recurs on the left segment (left to mid1 - 1). If the key is greater than arr[mid2], the function recurs on the right segment (mid2 + 1 to right). Otherwise, if the key is between arr[mid1] and arr[mid2], the function recurs on the middle segment (mid1 + 1 to mid2 - 1).
- If the function doesn't find the key in the array, it returns -1. The main function calls ternarySearch with the appropriate arguments and displays the result accordingly.

Exponential Search

```
#include <stdio.h>

// Binary search implementation
int binarySearch(int arr[], int left, int right, int key)
{
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == key)
```

```
            return mid;
        else if (arr[mid] < key)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;  // Key not found
}

// Exponential search implementation
int exponentialSearch(int arr[], int size, int key)
{
    // If key is found at index 0
    if (arr[0] == key)
        return 0;

    // Find the range for binary search
    int i = 1;
    while (i < size && arr[i] <= key)
        i *= 2;

    // Perform binary search within the identified range
    return binarySearch(arr, i / 2, (i < size) ? i : size - 1, key);
}

// Test the exponentialSearch function
int main()
{
    int arr[] = { 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 23;
    int result = exponentialSearch(arr, n, key);
    if (result == -1)
        printf("Element not found in the array.\n");
    else
        printf("Element found at index %d.\n", result);

    return 0;
}
```

Output:

```
Element found at index 5.
```

Explanation:

- We have an array arr containing some elements, and we want to search for a key value (in this case, 23). The exponentialSearch function takes the array, the size of the array, and the key to search for.
- The function starts by checking if the key is present at index 0. If so, it immediately returns 0. Otherwise, it determines the range for the binary search by doubling the index i until either the end of the array is reached or arr[i] becomes greater than the key.
- After identifying the range, the function calls the binarySearch function to perform a binary search within that range. The binarySearch function implements the standard binary search algorithm and returns the index of the key if found, or -1 if not found.
- The main function calls exponentialSearch with the appropriate arguments and displays the result accordingly.

Code in Java:

```java
public class ExponentialSearch {
    public static int exponentialSearch(int[] arr, int target) {
        if (arr[0] == target) {
            return 0;
        }

        int n = arr.length;
        int bound = 1;
        while (bound < n && arr[bound] <= target) {
            bound *= 2;
        }

        return binarySearch(arr, target, bound / 2, Math.min(bound, n - 1));
    }

    private static int binarySearch(int[] arr, int target, int left, int right) {
        if (left > right) {
            return -1;
        }

        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] > target) {
            return binarySearch(arr, target, left, mid - 1);
        } else {
            return binarySearch(arr, target, mid + 1, right);
        }
    }

    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10, 12, 14, 16};
        int target = 10;
        int index = exponentialSearch(arr, target);
        if (index != -1) {
            System.out.println("Element found at index " + index);
        } else {
            System.out.println("Element not found");
        }
    }
}
```

Output:

```
Element found at index 4
```

Explanation:

1. The exponentialSearch method takes an array arr and a target value target as input and returns the index of the target value in the array. If the target value is not found, it returns -1.

2. The method starts by checking if the first element of the array (arr[0]) is equal to the target value. If it is, the method returns 0 since the target is found at the first index.
3. If the target value is not found at the first index, the method proceeds with the exponential search algorithm. It initializes a variable bound to 1. This variable represents the upper bound of the search range.
4. The algorithm then enters a loop that doubles the bound value until either the bound exceeds the array size (n) or the element at the bound index is greater than the target value. This loop narrows down the search range exponentially.
5. After determining the bound, the method calls the binarySearch method to perform a binary search within the determined search range. It passes the array, target value, lower bound (bound / 2), and upper bound (Math.min(bound, n - 1)) to the binarySearch method.
6. The binarySearch method is a recursive implementation of the binary search algorithm. It takes the array, target value, left index, and right index as parameters.
7. In the binarySearch method, it checks if the left index is greater than the right index. If it is, it means the target value is not present in the search range, so the method returns -1.
8. Otherwise, it calculates the middle index (mid) as the average of the left and right indices.
9. It then compares the element at the mid index with the target value. If they are equal, the method returns the mid index since the target value is found.
10. If the element at the mid index is greater than the target value, it recursively calls the binarySearch method with the left part of the search range (left to mid - 1).
11. If the element at the mid index is smaller than the target value, it recursively calls the binarySearch method with the right part of the search range (mid + 1 to right).
12. The recursion continues until the target value is found or the search range is narrowed down to the point where the left index becomes greater than the right index.
13. In the main method, an example array arr is created with sorted elements. The target value is set to 10.

14. The exponentialSearch method is called with the array arr and the target value. The returned index is stored in the index variable.
15. Finally, the index variable is checked. If it is not -1, it means the target value was found, and the corresponding message is displayed. Otherwise, if the index is -1, it means the target value was not found, and the appropriate message is displayed.

Fibonnaci Search

Code in C:

```c
#include <stdio.h>

int fibonacciSearch(int arr[], int n, int key)
{
    // Initialize Fibonacci numbers
    int fib2 = 0; // (m-2)th Fibonacci number
    int fib1 = 1; // (m-1)th Fibonacci number
    int fib = fib2 + fib1; // mth Fibonacci number

    // Find the smallest Fibonacci number greater than or equal to n
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib2 + fib1;
    }

    // Marks the eliminated range from front
    int offset = -1;

    // Perform a comparison-based search
    while (fib > 1) {
        // Check if fib2 is a valid index
        int i = (offset + fib2 < n) ? offset + fib2 : n - 1;

        // If key is greater than the value at index fib2, cut the subarray from offs
et to i
```

```
        if (arr[i] < key) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            offset = i;
        }
        // If key is smaller than the value at index fib2, cut the subarray after i+1
        else if (arr[i] > key) {
            fib = fib2;
            fib1 = fib1 - fib2;
            fib2 = fib - fib1;
        }
        // If the key is found
        else {
            return i;
        }
    }

    // Compare the last element with the key
    if (fib1 == 1 && arr[offset + 1] == key) {
        return offset + 1;
    }

    // Element not found
    return -1;
}

int main()
{
    int arr[] = { 2, 5, 9, 13, 18, 21, 28, 35, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 18;
    int index = fibonacciSearch(arr, n, key);

    if (index >= 0) {
        printf("Element found at index %d\n", index);
    }
```

```c
    else {
        printf("Element not found\n");
    }

    return 0;
}
```

Output:

```
Element found at index 4.
```