**Data structure and algorithm**

**Tropic: Linked-list**

**Traverse linked list:**

Traversing a linked list means going through each element or node in the list, one by one, in order to access or process the data contained in each node. It is like walking along a path and examining the objects or information at each point.

|  |  |
| --- | --- |
| Pseude code | Program |
| //pseudo code  /\*  Structure Node:  data  next  Function traverseLinkedList(head):  current = head  Print "Linked List: "  while current is not NULL:  Print current.data  current = current.next  Function main():  node1 = Allocate memory for Node  node2 = Allocate memory for Node  node3 = Allocate memory for Node  node1.data = 1  node2.data = 2  node3.data = 3  node1.next = node2  node2.next = node3  node3.next = NULL  head = node1  traverseLinkedList(head)  Free memory for node1  Free memory for node2  Free memory for node3 | #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to traverse the linked list and print its elements  void traverseLinkedList(struct Node\* head) {  struct Node\* current = head;  printf("Linked List: ");  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  // Create nodes  struct Node\* node1 = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* node2 = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* node3 = (struct Node\*)malloc(sizeof(struct Node));  // Assign data to nodes  node1->data = 1;  node2->data = 2;  node3->data = 3;  // Connect nodes  node1->next = node2;  node2->next = node3;  node3->next = NULL;  // Set the head of the linked list  struct Node\* head = node1;  // Traverse and print the linked list  traverseLinkedList(head);  // Free memory  free(node1);  free(node2);  free(node3);  return 0;  } |

**Linked list searching algorithm:**

Linked list searching involves finding a specific element within a linked list data structure. The goal is to locate the node that contains the desired data value.

To perform a search in a linked list, you typically iterate through each node starting from the head node and compare the data value of each node with the target value you are searching for. The search process continues until the target value is found or until the end of the linked list is reached (i.e., the current node becomes NULL

Here is the basic concept of linked list searching:

Start at the head of the linked list.

Set a pointer to the current node, initially pointing to the head.

Enter a loop that continues until the current node becomes NULL.

Within each iteration of the loop:

Compare the data value of the current node with the target value you are searching for.

If a match is found, the search is successful, and you can take appropriate action (e.g., return the node, print a message).

If the data value does not match, move the pointer to the next node in the list (by updating the current node pointer to the next node).

If the loop completes without finding a match (i.e., the current node becomes NULL), the search is unsuccessful, indicating that the target value is not present in the linked list.

Linked list searching can be implemented using various algorithms, such as linear search, binary search (if the list is sorted and supports random access), or other specialized search techniques depending on specific requirements and constraints.

Remember that the efficiency of searching in a linked list is typically O(n), where n is the number of nodes in the list. This is because, in the worst case, you may need to traverse the entire list to find the desired element.

|  |  |
| --- | --- |
| Psedu code | Program |
| Function searchLinkedList(head, target):  current = head  while current is not NULL:  if current.data is equal to target:  Return true  current = current.next  Return false | #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to search for an element in the linked list  int searchLinkedList(struct Node\* head, int target) {  struct Node\* current = head;  while (current != NULL) {  if (current->data == target) {  return 1; // Element found  }  current = current->next;  }  return 0; // Element not found  }  int main() {  // Create nodes  struct Node\* node1 = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* node2 = (struct Node\*)malloc(sizeof(struct Node));  struct Node\* node3 = (struct Node\*)malloc(sizeof(struct Node));  // Assign data to nodes  node1->data = 1;  node2->data = 2;  node3->data = 3;  // Connect nodes  node1->next = node2;  node2->next = node3;  node3->next = NULL;  // Set the head of the linked list  struct Node\* head = node1;  // Search for target element  int target = 2;  int isFound = searchLinkedList(head, target);  // Print the result  if (isFound) {  printf("%d is found in the linked list.\n", target);  } else {  printf("%d is not found in the linked list.\n", target);  }  // Free memory  free(node1);  free(node2);  free(node3);  return 0;  } |

Explanation:

1. Start with the head node of the linked list.
2. Set a pointer current to point to the head node.
3. Enter a loop that continues until the current pointer becomes NULL.
4. Inside the loop:

Check if the current node's data is equal to the target value.

If they are equal, return true to indicate that the target element is found in the linked list.

If they are not equal, move the current pointer to the next node in the list by assigning current.next to current.

1. If the loop completes without finding the target element, return false to indicate that the target element is not present in the linked list.

**Linked list node insertion**

Linked list insertion involves adding a new node to a linked list at a specific position or location. This operation allows for dynamic modification of the linked list structure by inserting new elements.

Here is the basic concept of linked list insertion:

1. Create a new node and assign the desired data value to it.
2. Identify the location in the linked list where you want to insert the new node.

* If inserting at the beginning of the list, update the head pointer to point to the new node, making it the new head.
* If inserting at the end of the list, find the last node (the node with next pointer as NULL) and update its next pointer to point to the new node.
* If inserting at a specific position in the middle of the list, locate the node preceding the desired position.
  + Update the next pointer of the preceding node to point to the new node, making the new node the successor of the preceding node.
  + Update the next pointer of the new node to point to the node originally at that position, making it the successor of the new node.

1. Adjust the appropriate pointers to ensure the connectivity and integrity of the linked list.

|  |  |
| --- | --- |
| Pseudo code | Program |
| Function insertNode(head, position, newData):  Create a new node and assign newData to its data field  If position is 0:  Set the next pointer of the new node to the current head  Update the head pointer to point to the new node  Else:  Set a pointer current to point to the head node  Initialize a variable count to 0  While count is less than position - 1:  Move current to the next node in the list  Increment count  Set the next pointer of the new node to the node currently pointed to by current  Set the next pointer of the node pointed to by current to the new node | #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to insert a new node at a specific position in the linked list  void insertNode(struct Node\*\* head, int position, int newData) {  // Create a new node  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = newData;  if (position == 0) {  // Insert at the beginning of the list  newNode->next = \*head;  \*head = newNode;  } else {  // Insert at a specific position  struct Node\* current = \*head;  int count = 0;  while (count < position - 1 && current->next != NULL) {  current = current->next;  count++;  }  newNode->next = current->next;  current->next = newNode;  }  }  // Function to print the linked list  void printLinkedList(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  printf("\n");  }  int main() {  // Initialize an empty linked list  struct Node\* head = NULL;  // Insert nodes at different positions  insertNode(&head, 0, 10); // Insert 10 at the beginning  insertNode(&head, 1, 20); // Insert 20 at position 1  insertNode(&head, 2, 30); // Insert 30 at position 2  // Print the updated linked list  printf("Linked List: ");  printLinkedList(head);  return 0;  } |

**Explanation:**

1. The algorithm takes the head pointer of the linked list, the position at which the new node should be inserted, and the data for the new node as input.
2. A new node is created, and the newData value is assigned to its data field.
3. If the position is 0, it means the new node should be inserted at the beginning of the list:

* The next pointer of the new node is set to the current head.
* The head pointer is updated to point to the new node, making it the new head of the list.

1. If the position is not 0, the new node should be inserted at a specific position in the list:

* A pointer current is set to point to the head node.
* A count variable is initialized to 0 to keep track of the current position while traversing the list.
* A loop is entered that continues until the count is less than position - 1 (reaching the node preceding the desired position).
* Within each iteration of the loop, the current pointer is moved to the next node in the list, and the count is incremented.
* After exiting the loop, the new node's next pointer is set to the node currently pointed to by current.
* The next pointer of the node pointed to by current is updated to point to the new node, establishing the new node in the list at the desired position.