Linked List

This C++ file is an exhaustive, practical guide to the **Linked List data structure**, implemented in a procedural, C-style. It systematically covers the creation, manipulation, and application of several types of linked lists, making it an excellent resource for mastering this fundamental concept.

The file begins with the basics of a **Singly Linked List**, demonstrating core operations like creation, display (iteratively and recursively), counting, searching, insertion, and deletion. It then progresses to more advanced algorithms, providing multiple, distinct solutions for common problems like reversing a list (using an auxiliary array, sliding pointers, and recursion) and removing duplicates.

The scope then expands to cover more complex list variations. It includes a clear implementation of a **Circular Linked List**, showing how to handle insertions and deletions where the last node points back to the first. It also provides a thorough implementation of a **Doubly Linked List**, where each node points to both the next and previous nodes, allowing for more efficient bidirectional traversal and reversal.

Finally, the file showcases practical applications, such as merging two sorted lists, detecting a loop using Floyd's "tortoise and hare" algorithm, and representing a mathematical **polynomial**. This final example brilliantly illustrates how linked lists can be used to efficiently handle sparse data, where only non-zero terms need to be stored.

**Creating and Displaying LinkedList**

This section lays the foundation for a Singly Linked List. It shows how to create a list from an array and includes several methods for displaying its contents, including both iterative and recursive approaches.

* struct Node{ int data; struct Node \*next; }\*first = NULL;

This defines the blueprint for a node, the basic building block of a linked list. Each node contains an integer data and a pointer next that holds the memory address of the subsequent node.

* for(int i = 1; i<n; i++){ ... last->next = t; last = t; }

This is the core of the create function. It iterates through the input array, creating a new node (t) for each element. The line last->next = t; links the previous node to the newly created one, and last = t; moves the last pointer forward to keep track of the end of the list.

* RDisplayRev(p->next); printf("%d ", p->data);

This RDisplayRev function cleverly prints the list in reverse order. The recursive call happens before the printf statement. This means the printing only occurs during the "returning phase" of the recursion, starting from the last element and moving back to the first.

**Searching in a LinkedList**

This code demonstrates three different ways to perform a linear search to find a node with a specific key. One of the methods includes an optimization to speed up future searches.

* struct Node \* LinearSearch(struct Node \*p, int key){ ... }

This is the standard iterative linear search. It uses a while loop to traverse the list node by node, returning a pointer to the node if the key is found, or NULL if it reaches the end.

* return RLinearSearch(p->next, key);

This shows the recursive version of the linear search. The function checks the current node, and if it's not a match, it calls itself on the rest of the list (p->next).

* q->next = p->next; p->next=first; first = p;

This is an optimization called Move-to-Head. When the key is found, these lines of code rewire the pointers to move the found node to the very beginning of the list. This makes subsequent searches for the same key extremely fast (an O(1) operation).

**Inserting in a Linked List**

This section provides a general-purpose function to insert a new node at any valid position within the linked list (at the beginning, in the middle, or at the end).

* if(index<0 || index>count(p)){ return; }

This is an important boundary check. It ensures the provided index is valid before proceeding, preventing potential crashes.

* if(index==0){ t->next=first; first = t; }

This handles the special case of inserting at the beginning of the list. The new node t is made to point to the current first node, and then the global first pointer is updated to point to t.

* for(int i = 0; i<index-1; i++){ p=p->next; }

For insertions in the middle or at the end, this loop traverses the list to stop at the node just before the desired insertion point.

**Creating Linked List using insert function**

This snippet demonstrates that the insert function developed in the previous section is versatile enough to build a complete linked list from scratch, one element at a time.

* insert(first, 100, 0); insert(first, 200, 1);

This shows the process. The first call inserts 100 at index 0 of an empty list. The second call inserts 200 at index 1, placing it after the 100. This process can be repeated to construct a list of any length.

**Inserting in a sorted linked list**

This code implements a specialized insertion function that places a new element into a **sorted linked list** while maintaining its sorted order.

* struct Node \*t, \*q = NULL;

This function uses a "tailing pointer" (q). While the main pointer p moves forward, q always stays one step behind it.

* while(p && p->data<x){ q=p; p=p->next; }

This while loop is the core of the algorithm. It traverses the list as long as the current node's data is less than the new value x. This effectively finds the correct position to insert the new node. The tailing pointer q is essential because it allows us to access the node that will come before our new node.

**Deleting an element from linked list**

This section provides a function to remove a node from a specific position (index) in the list.

* if(index==1){ ... first=first->next; ... free(p); }

This handles the special case of deleting the first node. The global first pointer is simply moved to the second node, and the memory of the original first node is deallocated using free(p).

* for(i = 0; i<index-1; i++){ q = p; p=p->next; }

To delete a node from the middle or end, this loop traverses the list until p points to the node to be deleted and q points to the node right before it.

* q->next = p->next; ... free(p);

This is the key step. The next pointer of the previous node (q) is rewired to bypass the node being deleted (p), effectively removing it from the chain. The memory for p is then freed.

**Checking weather linked list is sorted or not**

This code provides a simple and efficient function to verify if the elements in a linked list are in non-decreasing sorted order.

* int x = -65536;

A variable x is initialized to a very small integer. This variable will be used to store the value of the previously visited node.

* while(p!=NULL){ if(p->data < x){ return 0; } x = p->data; ... }

This loop traverses the list. In each iteration, it compares the current node's data (p->data) with the previous node's data (x). If it ever finds a value that is smaller than the previous one, the list is not sorted, and the function immediately returns 0 (false).

**Deleting duplicate elements**

This function removes consecutive duplicate elements from a **sorted linked list**.

* struct Node \*q; q = p->next;

The algorithm uses two pointers. p is the main iterator, and q always points to the node immediately following p.

* if(p->data!=q->data){ p=q; q=q->next; }

If the data in the two nodes is different, it means there's no duplicate, so both pointers simply advance to the next position.

* else{ p->next = q->next; free(q); q = p->next; }

If the data is the same, a duplicate is found. The next pointer of p is updated to skip over the duplicate node q. The memory for q is then freed, and q is reset to the new p->next to continue checking for more duplicates.

**Reversing a linked list**

This section is a great comparison of three distinct algorithms for reversing a linked list.

* void Reverse1(struct Node \*p){ ... }

This method reverses the list by copying the data, not the pointers. It first iterates through the list to copy all data into an auxiliary array A. Then, it iterates through the list again, writing the data back from the array in reverse order. It's simple but uses extra memory.

* void Reverse2(struct Node \*p){ ... }

This is the classic iterative "sliding pointers" method. It uses three pointers (p, q, r) to walk down the list and reverse the direction of the next pointers one by one. It's highly efficient as it uses constant extra space.

* void Reverse3(struct Node \*q, struct Node \*p){ ... }

This is the recursive solution. The reversal of the pointers happens during the returning phase of the recursion. The line p->next = q; does the actual reversal, making the next node point back to the previous one.

**Concatination and Merging of 2 linked lists**

This code demonstrates two ways to combine two linked lists: concat simply appends one list to the end of another, while merge combines two *sorted* lists into a single new sorted list.

* void concat(struct Node \*p, struct Node \*q){ ... }

Concatenation is straightforward. The while loop traverses the first list (p) until it reaches the very last node. Then, it sets the next pointer of that last node to point to the beginning of the second list (q).

* void merge(struct Node \*p, struct Node \*q){ ... }

This function implements the merge logic, which is a core part of the Merge Sort algorithm. It compares the nodes from both lists and, in each step, appends the smaller of the two to the new merged list (third). It continues until one list is exhausted, then appends the remainder of the other list.

**Checking weather the Linked List is looped or not**

This code implements **Floyd's Cycle-Finding Algorithm**, also known as the "tortoise and hare" algorithm, to efficiently detect if a linked list contains a loop.

* struct Node \*p, \*q; p = q = f;

Two pointers are used: p is the "tortoise" (slow pointer), and q is the "hare" (fast pointer). Both start at the head of the list.

* do{ p=p->next; q=q->next; q=q?q->next:q; }while(p && q && p!=q);

In each step of the loop, the slow pointer p moves one node forward, while the fast pointer q moves two nodes forward.

* if(p==q){ return 1; }

If there is a loop, the fast pointer will eventually run around the cycle and "lap" the slow pointer, causing them to meet at the same node. If they meet, a loop is confirmed. If either pointer becomes NULL, it means the end of the list was reached, and there is no loop.

**Circular Linked List**

This section defines and implements a **Circular Linked List**, where the last node does not point to NULL but instead points back to the head of the list.

* Head->next = Head;

This is the defining characteristic of a circular list with a single node. The first node points to itself. In a larger list, last->next will point to Head.

* void Display(struct Node \*h){ do{ ... } while (h != Head); }

Displaying a circular list requires a do-while loop instead of a standard while loop. This ensures that the code inside the loop runs at least once (to print the head node) before checking if the traversal has come back around to the Head.

* static int flag = 0; if (h != Head || flag == 0){ flag = 1; ... }

The recursive display function needs a static flag to handle the starting condition. The condition h != Head would be false for the very first call, so flag == 0 ensures the code enters the block on the first call and is then set to 1 to prevent an infinite loop.

**Doubly Linked List and inserting in Doubly Linked List**

This code introduces a **Doubly Linked List**, where each node has pointers to both the next and the previous nodes in the sequence.

* struct Node \*Prev; int data; struct Node \*next;

The Node structure is updated to include a Prev pointer in addition to the next pointer.

* t->next=last->next; t->Prev = last; last->next = t; last = t;

When creating the list, setting up the links for a new node t requires updating both pointers. t->Prev is set to the previous node (last), and last->next is updated to point to t.

* t->next = p->next; t->Prev = p; if (p->next) p->next->Prev = t; p->next = t;

Inserting a node t in the middle of a doubly linked list is more complex and requires re-wiring four pointers to correctly splice the new node into the list without breaking the chain.

**Deleting an element in Doubly Linked List**

This section shows how to delete a node from a Doubly Linked List. The presence of the Prev pointer makes this operation more straightforward than in a singly linked list.

* p->Prev->next = p->next; if(p->next) p->next->Prev = p->Prev;

This is the core of the deletion logic for a middle node. It tells the node before p to point its next to the node after p. Then, it tells the node after p to point its Prev back to the node before p. This effectively bypasses p from both directions.

**Reversing a doubly linked list**

This code demonstrates a very elegant and efficient way to reverse a Doubly Linked List.

* temp = p->next; p->next = p->Prev; p->Prev = temp;

This is the key operation. For each node, it simply swaps the next and Prev pointers. A temp variable is used to hold the original next pointer so it's not lost.

* p = p->Prev;

After the swap, the original Prev pointer now points to the next node in the original sequence, so we use it to traverse the list.

**Polynomial representation using Linked List**

This final section provides a practical application of linked lists: representing a mathematical polynomial. Each node in the list represents a single term of the polynomial.

* struct Node { int coeff; int exp; struct Node \*next; };

Each node stores the coefficient and exponent of a term, making it a perfect way to represent polynomials where many exponents might be missing (e.g., 5x100+2x3−10).

* long Eval(struct Node \*p, int x) { ... }

The Eval function calculates the value of the polynomial for a given x. It traverses the list, and for each term (node), it calculates coeff \* pow(x, exp) and adds it to a running total.