Linked Lists

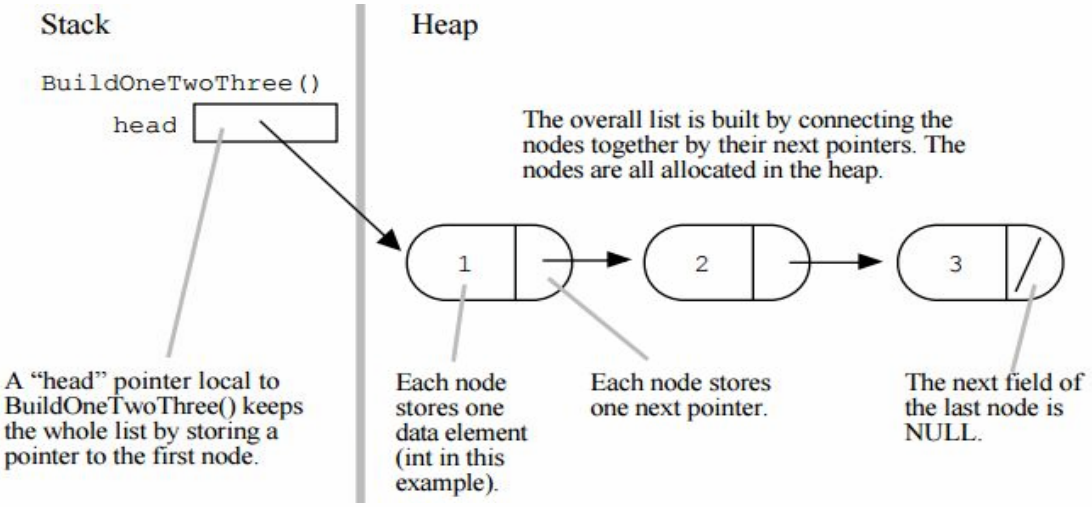
Topics

1. What is a Linked List?
2. Types of Linked Lists
3. Advantages & Disadvantages of a Linked List
4. Implementation of Doubly Linked List in C/C++
5. Problems & Solutions
6. Important References

What is a Linked List?

A Linked List [LL] allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain. This is in contrast to an array, where memory is allocated for all its elements lumped together as one block of memory.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node. Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like:



Disadvantages of arrays:

1. The size of the array is fixed. Most often this size is specified at compile time with a simple declaration. With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. (additional information for experts) We can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with realloc(), but that requires some real programming effort.

2. Because of the aforementioned reason, the most convenient thing for programmers to do is to allocate arrays which seem "large enough". Although convenient, this strategy has two disadvantages:
    a. Most of the time there are just 20 or 30 elements in the array and 70% of the space in the array is not really utilised.
    b. If the program ever needs to process more elements, the code breaks.

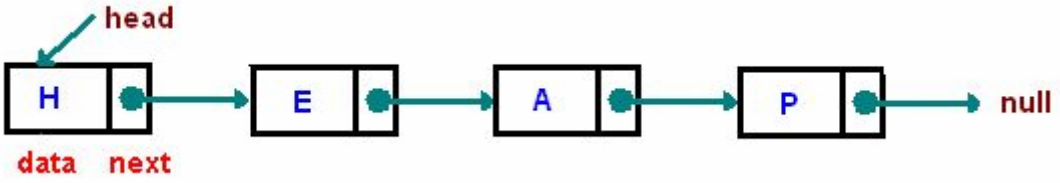
3. Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room, thereby, increasing the overall complexity.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. All arrays follow a simple strategy and that is allocating the memory for all

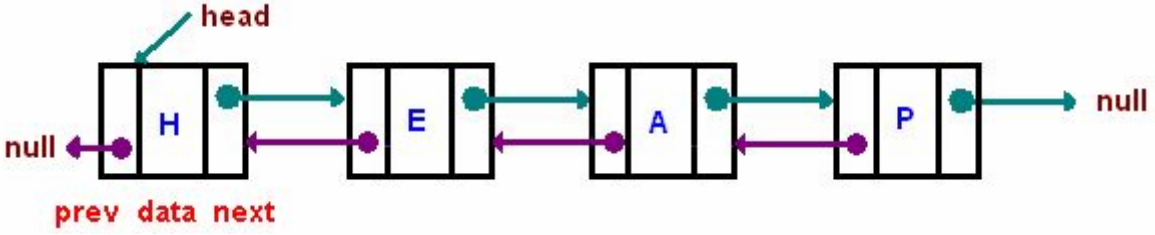
its elements in one block of memory. Linked lists use an entirely different strategy. As we will see, linked lists allocate memory for each element separately and only when necessary.

Types of Linked Lists

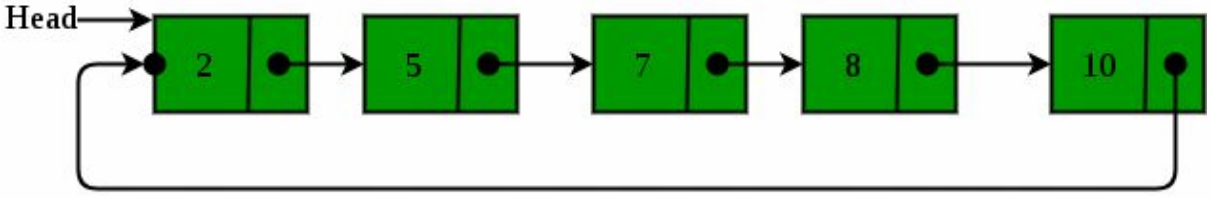
Singly Linked List: Singly linked lists contain nodes which have a **data** part as well as an **address** part *i.e.*, next, which points to the next node in the sequence of nodes. The operations we can perform on a singly linked list are insertion, deletion and traversal.



Doubly Linked List: In a doubly linked list, each node contains two links the first link points to the previous node (**prev**) and the next link (next) points to the next node in the sequence. Compared to SLL, the added advantage of a DLL is that, from a given node we can traverse backward and forward using a DLL. But this costs us *extra memory* for using an additional pointer for every node to also point to the previous node.



Circular Linked List: In the circular linked list the last node of the list contains the address of the first node and forms a **circular chain**. A CLL can be implemented using a SLL or a DLL. The last node of the CLL doesn't point to NULL. A simple example where CLL's are used is when we try to keep track of whose turn it is in a multi-player board game.



Further Reading: **Carnegie Melon University Notes on Linked Lists.**

Advantages of Linked Lists

1. They are dynamic in nature *i.e.*, runtime memory allocation, only when required.
2. Insertion and deletion operations can be easily implemented.
3. Stack and Queue operations can be easily executed once they're implemented.
4. Linked Lists reduce the access time of an element.
5. Can grow arbitrarily large, without re-allocating the entire structure.

Disadvantages of Linked Lists

1. Additional memory usage, since pointers require extra memory.
2. No element can be accessed randomly; it has to access each node sequentially, therefore list search becomes linear, when compared to a sorted array and Balanced BST, where, search only takes logarithmic time.
3. Reverse Traversing is difficult in a singly linked list.
4. The Cache footprint is higher, in case of Linked Lists.
5. In case of languages like C/C++, dynamically allocated memory has to be taken care by the programmer themselves, which is a programming overhead, and if it is not taken care properly, then memory leaks can cause problems.

Applications of Linked Lists

1. Linked lists are used to implement stacks, queues, graphs, etc.
2. Linked lists let you insert elements at the beginning and end of the list.
3. Linked lists are used in separate chaining, while handling collisions in a Hash Table.
4. Linked Lists are used in implementing certain page replacement policies in OS.
5. Linked Lists are used in implementing binary/non-binary trees.

Applications of Circular Linked List

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue. We don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4. Circular Doubly Linked Lists are used for implementation of advanced data structures like **Fibonacci Heap**.

Implementation of Doubly Linked List in C/C++

```
#include <iostream>
#include <cstdlib>
using namespace std;

// Structure of node
typedef struct node {
	int data;
	struct node *next, *prev;
} node;

// Structure of LL, where root is a dummy node of LL, and last is the tail of LL
typedef struct LinkedList {
	node *root, *last;
	int size;
} LinkedList;


// Function that creates a new node for the LL
node* createNode(int num) {
	node* newNode =  (node*)malloc(sizeof(node*));
	newNode->data = num;
	newNode->next = NULL;
	newNode->prev = NULL;
	return newNode;
}

// Function that creates a dummy node as the first node of the LL
LinkedList* createLinkedList() {
	LinkedList* newLinkedList =  (LinkedList*)malloc(sizeof(LinkedList*));
	node* newNode =  createNode(0);
	newLinkedList->root = newNode;
	newLinkedList->last = newNode;
	newLinkedList->size = 0;
	return newLinkedList;
}

// Function that appends a node at the tail of LL
void push_back(LinkedList* linkedList, int num) {
	node* newNode = createNode(num);
	linkedList->last->next = newNode;
	newNode->prev = linkedList->last;
	linkedList->last = newNode;;
	linkedList->size = linkedList->size + 1;
}

// Function that appends a node at the head of LL
void push_front(LinkedList* linkedList, int num) {
	node* newNode = createNode(num);
	newNode->next = linkedList->root->next;
	newNode->prev = linkedList->root;
	linkedList->root->next = newNode;
	if(newNode->next != NULL)
		newNode->next->prev = newNode;
	else
		linkedList->last = newNode;
```

```
	linkedList->size = linkedList->size + 1;
}

// Function that deletes a node from the tail of LL
int pop_back(LinkedList* linkedList) {
	if(linkedList->size == 0)
		return -1;

	node* last = linkedList->last;
	linkedList->last = last->prev;
	linkedList->last->next = NULL;

	int data = last->data;
	free(last);
	linkedList->size = linkedList->size - 1;
	return data;
}

// Function that deletes a node from the head of LL
int pop_front(LinkedList* linkedList) {
	if(linkedList->size == 0)
		return -1;

	node* front = linkedList->root->next;
	linkedList->root->next = front->next;

	if(front->next != NULL)
		front->next->prev = linkedList->root;

	int data = front->data;
	free(front);
	linkedList->size = linkedList->size - 1;
	return data;
}

// Function that deletes all nodes with the a particular data value in the LL
void deleteNode(LinkedList* linkedList, int num) {
	node* current = linkedList->root;
	while(current->next != NULL) {
		if(current->next->data == num) {
			node* tmp = current->next;
			tmp->next->prev = current;
			current->next = tmp->next;
			free(tmp);
			linkedList->size = linkedList->size - 1;
		}
		else
			current=current->next;
	}
}

// Function that returns the data present at the tail node of the LL
int back(LinkedList* linkedList) {
	if(linkedList->size == 0)
		return -1;
	return linkedList->last->data;
}
```

```
// Function that returns the data present at the head node of the LL
int front(LinkedList* linkedList) {
	if(linkedList->size == 0)
		return -1;
	return linkedList->root->next->data;
}

// Function that checks whether a LL is empty or not
bool isEmpty(LinkedList* linkedList) {
	return linkedList->size == 0;
}

// Function that returns the number of nodes in the LL
int length(LinkedList* linkedList) {
	return linkedList->size;
}

// Function that prints all the node values in the LL sequentially
void printList(LinkedList* linkedList) {
	node* root = linkedList->root->next;
	while(root != NULL) {
		cout<<root->data<<"->";
		root = root->next;
	}
	cout<<"NULL"<<endl;
}

// Driver Program
int main() {
	LinkedList* linkedList = createLinkedList();
	cout<<"isEmpty:"<<(isEmpty(linkedList)?"True":"False")<<endl;

	push_front(linkedList, 2);
	push_back(linkedList, 3);

	printList(linkedList);
	pop_front(linkedList);
	printList(linkedList);

	push_back(linkedList, 4);
	push_front(linkedList, 1);
	push_back(linkedList, 5);
	push_back(linkedList, 4);

	printList(linkedList);
	deleteNode(linkedList, 4);
	printList(linkedList);
	pop_back(linkedList);
	printList(linkedList);
	push_back(linkedList, 5);

	cout<<"isEmpty:"<<(isEmpty(linkedList)?"True":"False")<<endl;
	cout<<"Length:"<<length(linkedList)<<endl;
	printList(linkedList);
	return 0;
}
```

Problems & Solutions

1. Find the length of a linked list – Iterative and Recursive.
   Solution:

```
int iterativeLength(node* root) {
    int length = 0;
    while(root) {
        root = root->next;
        length++;
    }
    return length;
}

int recursiveLength(node* root) {
    if(root == NULL)
        return 0;
    return 1 + recursiveLength(root->next);
}
```

2. Reverse a singly linked list – Iterative and Recursive.
   Example: `6->20->3->14->5->NULL`; Ans: `5->14->3->20->6->NULL`;
   Solution:

Iterative	Recursive

`node *reverse(node *head) {`
`  node *prev = NULL, *tmp;`
 	
`  while(head != NULL) {`
`  tmp = head->next;`
`  head->next = prev;`
`  prev = head;`
`  head = tmp;`
`  }`
`  return prev;`
`}`
	`node *reverse(node *head) {`
	`  if(head == NULL || head->next == NULL)`
	`  return head;`
	
	`  node *ret = reverse(head->next);`
	`  head->next->next = head;`
	`  head->next=NULL;`
	`  return ret;`
	`}`

3. Find the midpoint of a linked-list.
   Example: `1->2->3->4->5->NULL`; Ans: `3`
   Solution: Take two pointers, slow and fast. Move slow pointer by 1 node and fast pointer by 2 nodes at a time. At the end, when fast pointer reaches the last node, the slow pointer is at the middle element of the linked list.

4. Sort a singly linked list.
   Solution: Use merge sort
   a. Split the list into 2 halves, by finding mid using slow and fast pointers.
   b. Sort the left and right sublist individually
   c. Merge the two sorted lists

5. Union and Intersection of two Linked Lists.
   Example: `10->15->4->20->NULL ;  8->4->2->10->NULL`
   Intersection: `4->10->NULL` ; Union: `2->4->8->10->15->20->NULL`
   Solution: Sort the two linked lists and do linear traversal to find Union/Intersection.

6. Rearrange a given linked list in-place, *i.e.*, "L_0 -> L_1 -> L_2 -> ... L_{n-2} -> L_{n-1} -> L_n" to "L_0 -> L_n -> L_1 -> L_{n-1} -> L_2 -> L_{n-2} ..."
   Example: `1->2->3->4->5->NULL`; Ans: `1->5->2->4->3->NULL`
   Solution: Find the midpoint, split the list into two parts, reverse the right part and create a list by taking alternate elements from 1st part and 2nd part.

7. You are given a Doubly Linked List with one pointer of each node pointing to the next node just like in a singly linked list. The second pointer however can point to any node in the list and not just the previous node. WAP to clone such a linked list, *i.e.*, create an exact clone of the given linked list with next and random pointers pointing to the same valued elements (not the addresses) in the original linked list.
   Solution:
   a. Create a copy of each node and insert it between the current and the next node.
   b. Update random pointer for the new nodes by:
      ```
      original->next->random = original->random->next;
      ```
   c. Restore the original and copy linked lists using:
      ```
      original->next = original->next->next;
      copy->next = copy->next->next;
      ```

8. Add two numbers represented by linked lists into a new Linked List.
   Example: `5->6->3->NULL + 8->4->2->NULL` ; Ans: `1->4->0->5->NULL`, *i.e.*,

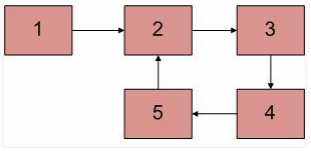
```
    5->6->3->NULL   ~> Given Linked List L1.
(+) 8->4->2->NULL   ~> Given Linked List L2.
-----------------
 1->4->0->5->NULL   ~> L1 + L2 is the required final solution.
-----------------
```

   Solution: Reverse the two given linked lists, add node by node and maintain a carry pointer into a new list and finally reverse the new list to obtain the final answer.

9. Implement Stack with `getMiddle()` operation in O(1).
   Solution: Use a Doubly Linked List to implemet a stack. At the same time, keep a mid pointer that points to the middle element in the list. Adjust the mid pointer whenever there are changes (push or pop) in the stack.

10. Implement **LRU** cache: Given the order of access of page numbers and the cache (or memory) size, **LRU** caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache.
Solution: Use doubly linked list and a **HashMap** of **<PageNumber, Node>** to perform operations efficiently. Keep a **dummy** node to ease things up.
    a. Keep two pointers, front and rear – front always points to **LRU** page/node and rear always points to the **MRU** page/node.
    b. When a page is accessed:
        i. If it's present in the LL (check using **HashMap**), move it to the end of the list and update the prev/next pointers. Update rear pointer as well.
        ii. If it's not present in the LL (check using **HashMap**):
            1. If size of LL is less than cache size, insert it at the rear of the LL, update rear pointer and insert in the **HashMap** as well.
            2. Else, remove the LRU (front of the LL) node, both from the LL and the **HashMap**, and insert new node at the rear of the LL, update rear pointer and insert in the **HashMap** as well.

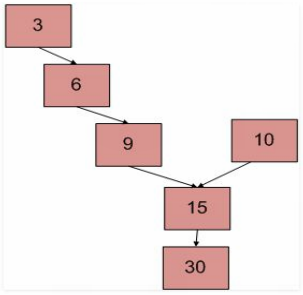
11. Detect and Remove Loop in a Linked List.



In this LL, there's loop from node 2 to node 5.

Solution:
    a. Slow and fast pointer to get the pointer to a loop node. (node where they meet)
    b. Take two pointers - one pointing at head and the other at the intersection point.
    c. Move both pointers one at a time, their intersection point is the last/first node of the loop.

12. Write a function to get the intersection point of two Linked Lists, given their head pointers.



In this LL, the intersection point is node 15.

Solution:
    a. Find the lengths of the two linked lists, say L_1 and L_2
    b. Traverse the larger list by a distance of abs($L_1 - L_2$)
    c. Traverse each list one node at a time and compare.

13. Given a linked list, WAF to reverse every **k** nodes.

    Example:
    Input$_1$: **1->2->3->4->5->6->7->8->NULL** and **k=3**
    Output$_1$: **3->2->1->6->5->4->8->7->NULL**

    Input$_2$: **1->2->3->4->5->6->7->8->NULL** and **k=5**
    Output$_2$: **5->4->3->2->1->8->7->6->NULL**

    Solution:

Recursive	Iterative

`node* reverseK(node *root, int k)`
`{`
`  if(root == NULL)`
`  return NULL;`
 	
`  node* prev=NULL, *next=NULL;`
`  node* head = root;`
`  int cnt = 0;`
`  while(root != NULL && cnt<k) {`
`  next = root->next;`
`  root->next = prev;`
`  prev = root;`
`  root = next;`
`  cnt++;`
`  }`
 	
`  head->next = reverseK(root, k);`
`  return prev;`
`}`
	`node* reverseK(node *root, int k)`
	`{`
	`  node *head = createNode(-1);`
	`  node *p2 = root, *p1 = head;`
	`  node *prev = NULL;`
	`  int cnt = 0;`
	
	`  while(root != NULL) {`
	`  node* tmp = root->next;`
	`  root->next = prev;`
	`  prev = root;`
	`  root = tmp;`
	`  cnt++;`
	`  if(cnt == k) {`
	`  cnt = 0;`
	`  p1->next = prev;`
	`  prev = NULL;`
	`  p1 = p2; p2 = root;`
	`  }`
	`  }`
	`  p1->next = prev;`
	`  return head->next;`
	`}`

14. Given a pointer to the head of a circular linked list, split the list in two equal sized circular linked lists and return the head of the 2nd circular list.
    Solution:
    a. Store the mid and last pointers of the circular linked list using **tortoise and hare algorithm**.
    b. Make the second half circular.
    c. Make the first half circular.
    d. Set head (or start) pointers of the two linked lists.

15. Insert an element in a sorted circular linked list, given a pointer to the head of the list.
    Solution: Iterate and find the position for insertion, carefully handle edge cases. One of the edge case is that, we can insert a smaller element than the first element.
    Another edge case is, we can insert a greater element than the last element. Therefore, these edge cases are to be handled efficiently.

Important References

1. **Linked List vs Array**
2. **Singly Linked List & Doubly Linked List Implementations in Java**
3. **Singly Linket List & Doubly Linked List Implementations in Python**
4. **XOR Linked Lists - A Memory Efficient Linked List - Part-1 & Part-2**