Linked Lists

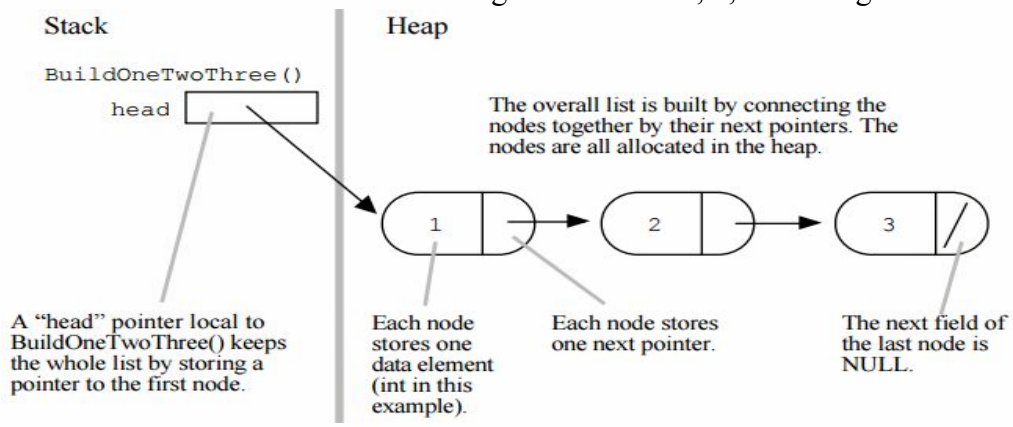
Topics

1. Definition
2. Advantages & Disadvantages
3. Types of Linked List
4. Implementation
5. Problems & Solutions
6. Notes
7. References

Linked List

A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is 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 such as in the example above. 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. (extra for experts) You 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 programmer effort.

2. Because of above, 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 really is wasted.
    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.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of 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.

Advantages of Linked Lists

1. They are dynamic in nature which allocates the memory when required.
2. Insertion and deletion operations can be easily implemented.
3. Stacks and queues can be easily executed.
4. Linked List reduces the access time.

Disadvantages of Linked Lists

1. The memory is wasted as pointers require extra memory for storage.
2. No element can be accessed randomly; it has to access each node sequentially.
3. Reverse Traversing is difficult in linked list

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. In Linked Lists we don't need to know the size in advance.

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 sequence of nodes. The operations we can perform on singly linked lists 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 and the next link points to the next node in the sequence.

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 simple example is keeping track of whose turn it is in a multi-player board game.

Code for Doubly Linked List:

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

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

typedef struct LinkedList {
    node *root, *last;
    int size;
} LinkedList;


node* createNode(int num) {
node* newNode =  (node*)malloc(sizeof(node*));
newNode->data = num;
newNode->next = NULL;
newNode->prev = NULL;
return newNode;
}

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

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;
}
```

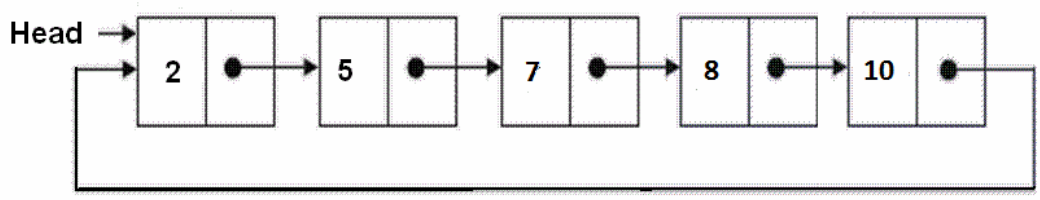
```
void push_front(LinkedList* linkedList, int num) {
    node* newNode = createNode(num);
    newNode->next = linkedList->root->next;
    newNode->prev = linkedList->root;
    linkedList->root->next = newNode;
    if(newNode->next != NULL)
        newNode->next->prev = newNode;
    else
        linkedList->last = newNode;

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

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;
}

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;
}
```

```
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;
            break;
        }
        current=current->next;
    }
}

int back(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;
    return linkedList->last->data;
}

int front(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;
    return linkedList->root->next->data;
}

bool isEmpty(LinkedList* linkedList) {
    return linkedList->size == 0;
}

int length(LinkedList* linkedList) {
    return linkedList->size;
}

void printList(LinkedList* linkedList) {
    node* root = linkedList->root->next;
    while(root != NULL) {
        cout<<root->data<<"->";
        root = root->next;
    }
    cout<<"NULL"<<endl;
}
```

```
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);

    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;
}
```

Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Applications

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.

Problems

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

```
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 linked list – Iterative and Recursive.
   Example: 1->2->3->4->5->NULL, Ans: 5->4->3->2->1->NULL

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 mid-point of a linked-list.
   Example: 1->2->3->4->5->NULL, Ans: 3

   Solution: Take 2 pointers, slow and fast. Move slow pointer by 1 node and fast pointer by 2 nodes. At the end, 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->8->20->4->15->10->NULL
   Solution: Sort the 2 linked lists and do linear traversal to find Union/Intersection.

6. Rearrange a given linked list in-place, ie, "L_0 -> L_1 -> L_2 -> … L_{n-2} -> Ln-1 -> Ln" 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 mid, split the list into 2 parts, reverse the right part and create a list by taking alternate elements from 1st part and 2nd part.

7. You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link 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,ie, a linked list with next and random pointer.
   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;

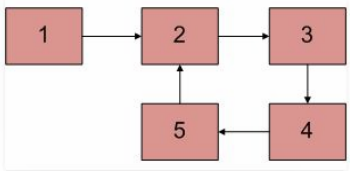
8. Add two numbers represented by linked lists into a new Linked List.
   Example: 5->6->3->NULL + 8->4->2->NULL, Ans: 1->4->->0->5->NULL

   Solution: Reverse the 2 given linked list, add node by node and maintain a carry pointer into a new list and finally reverse the new list.

9. Implement Stack with getMiddle operation in O(1)
   Solution: Use Doubly Linked List and keep a mid pointer, adjust when you perform push and pop operations.

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 map of <PageNumber, Node> to perform operations efficiently. Keep a dummy node to ease things.
    a. Keep 2 pointer front and rear – front always points to LRU page and rear always points to the RU node.
    b. When a page is accessed:
        i. If it's present in the LL (check using HashTable), 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 HashTable):
            1. If size of LL less than cache size, insert it at the rear of the LL, update rear pointer and insert in the HashTable as well.
            2. Else, remove the LRU (front of the LL) node, both from the LL and the HashTable, and insert new node at the rear of the LL, update rear pointer and insert in the HashTable as well.

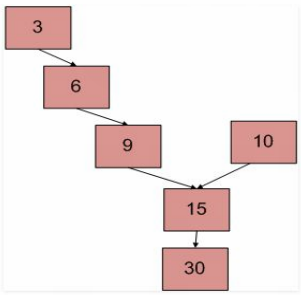
11. Detect and Remove Loop in a Linked List.



Solution

a. Slow and fast pointer to get the pointer to a loop node.
b. Find the length of the loop, say k.
c. Take 2 pointers pointing at head, move one of them by k.
d. Move both pointers one at a time, their intersection point is the last/first node of the loop, make node->next = NULL.
e. Ref: http://www.geeksforgeeks.org/detect-and-remove-loop-in-a-linked-list/

12. Write a function to get the intersection point of two Linked Lists.



Solution

a. Find the lengths of the 2 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:
Input1: 1->2->3->4->5->6->7->8->NULL and k = 3
Output1: 3->2->1->6->5->4->8->7->NULL.

Input2: 1->2->3->4->5->6->7->8->NULL and k = 5
Output2: 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 list and return the head 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 insert, carefully handle edge case like smaller than the first element, greater than the last element.

References

1. http://www.geeksforgeeks.org/linked-list-vs-array/
2. http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/
3. http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/