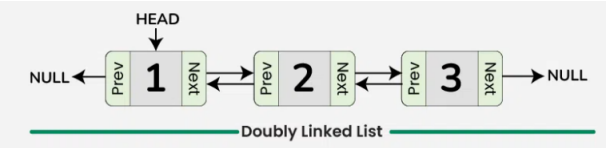
Linked List 3

Intro to Doubly Linked List

- A **Doubly Linked List (DLL)** is a two-way list in which each node has two pointers, the next and previous that have reference to both the next node and previous node respectively. Unlike a singly linked list where each node only points to the next node, a doubly linked list has an extra previous pointer that allows traversal in both the forward and backward directions.

```
Class Node {
    int data;
    Node next;
    Node prev;
    Node (int data) {
        this.data = data;
        this.next = NULL;
        this.prev = NULL;
    }
};
```



Advantages of Doubly Linked List

- **Efficient traversal in both directions:** Doubly linked lists allow for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.
- **Easy insertion and deletion of nodes:** The presence of pointers to both the previous and next nodes makes it easy to insert or delete nodes from the list, without having to traverse the entire list.
- **Can be used to implement a stack or queue:** Doubly linked lists can be used to implement both stacks and queues, which are common data structures used in programming.

Disadvantages of Doubly Linked List

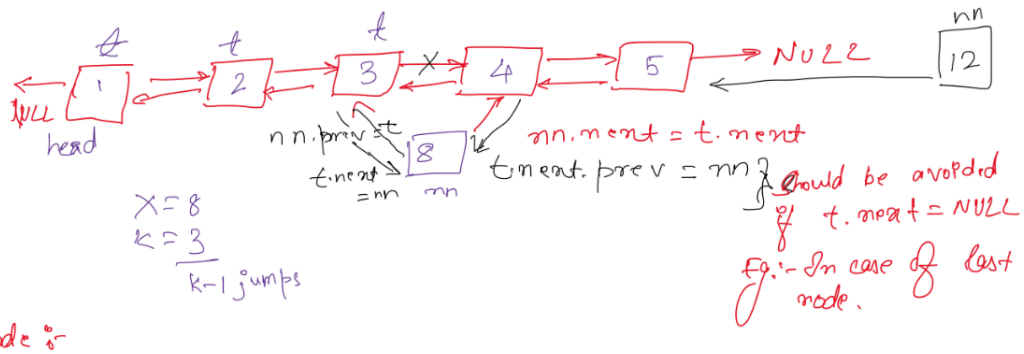
- **More complex than singly linked lists:**
- **More memory overhead:**

Applications of Doubly Linked List

- Implementation of undo and redo functionality in text editors.
- Cache implementation where quick insertion and deletion of elements are required.
- Browser history management to navigate back and forth between visited pages.
- Music player applications to manage playlists and navigate through songs efficiently.
- Implementing data structures like Deque (double-ended queue) for efficient insertion and deletion at both ends.

Insertion

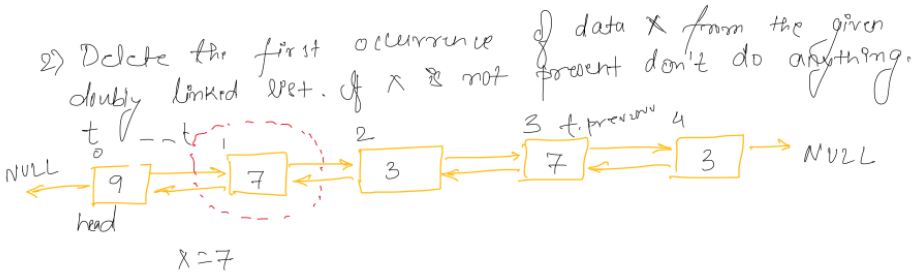
Insert a node with value X at position k.



Code :-

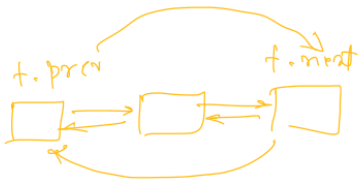
```
if (k > size(head)) {
    return head;        //Invalid Scenario.
}
{ Node nn = new Node (x);
  if (head == NULL)     //Empty list
      return nn;
  if (k == 0) {         //Insert at head
      nn.next = head;
      head.prev = nn;
      head = nn;
      return head;
  }
  Node t = head;
  for (int i=0; i < k ; i++) {  //Insert at kth loc
      t = t.next;
  }
  nn.next = t.next ;
  nn.prev = t;
  if (t.next != NULL) {     //Insertion at end
      t.next.prev = nn;
  }
    t.next = nn;
  return head;
}
```

Deletion

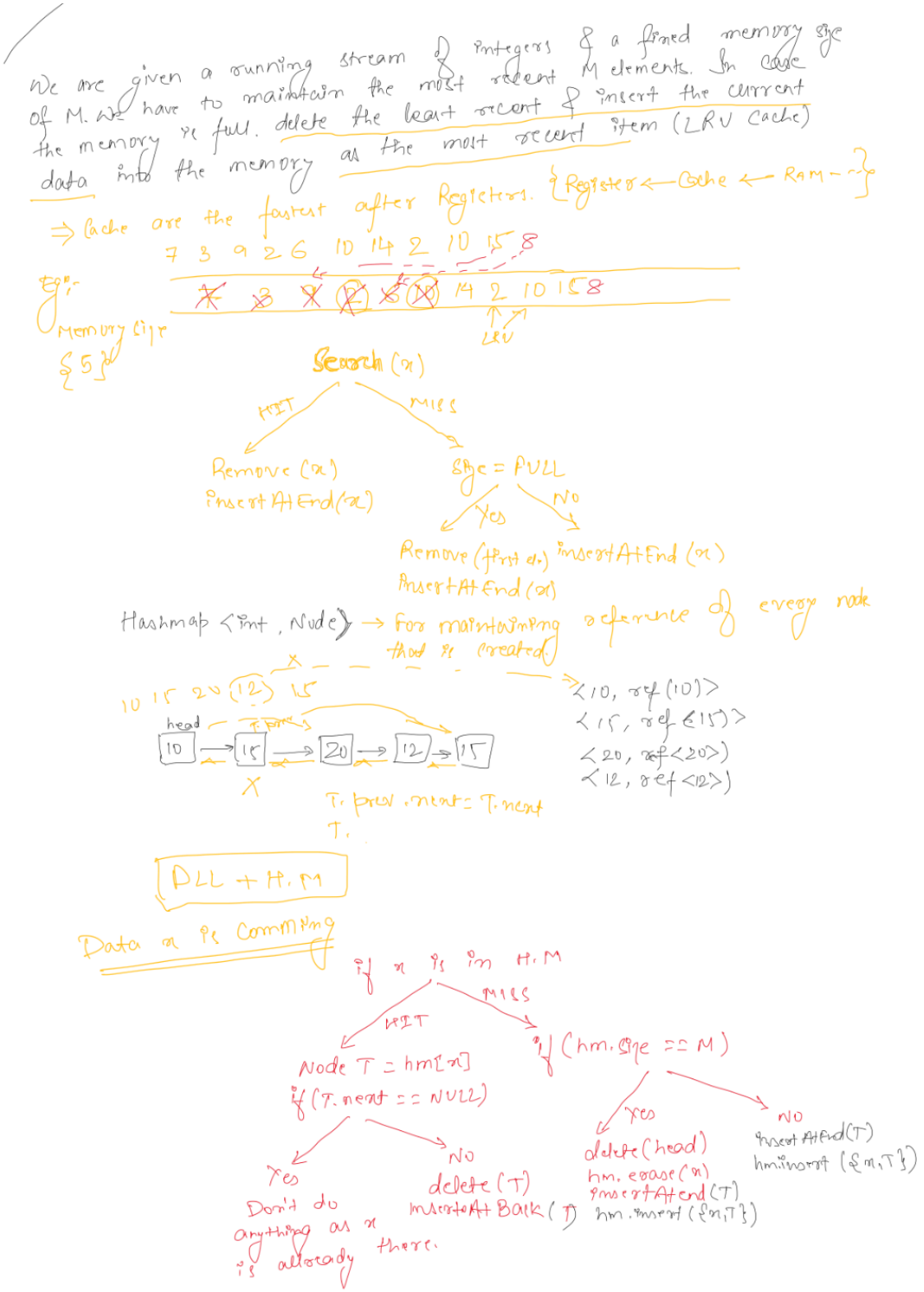


Code:-

```
Node t = head
while(t != NULL)
{   if(t.data == x)
        break;
    t = t.next;
}
// If data x is not present
if(t == NULL) {
    return head;
}
//delete head in case of single head
if(t.next == NULL && t.prev == NULL) {
    return NULL;
}
//delete head in other cases.
if(t.prev == NULL) {
    t.next.prev = NULL;
    head = t.next;
    return head;
}
//delete last node
if(t.next == NULL) {
    t.prev.next = NULL;
    return head;
}
else {
    t.prev.next = t.next;
    t.next.prev = NULL;
    return head;
}
```



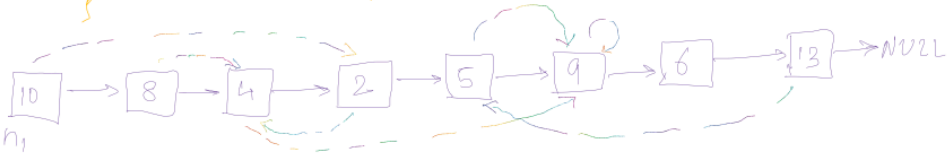
LRU Cache


We are given a running stream of integers & a fixed memory size of M. We have to maintain the most recent M elements. In case the memory is full, delete the least recent & insert the current data into the memory as the most recent item (LRU Cache)
⇒ Cache are the fastest after Registers. {Register ← Cache ← Ram - ...}
7 3 9 2 6 10 14 2 10 15 8
Eg:-
Memory Size {5}
LRU
Search (x)
HIT
Miss
Remove (x)
insertAtEnd(x)
Size = FULL
Yes
No
Remove (first el.)
insertAtEnd (x)
insertAtEnd (x)
Hashmap <int, Node> → for maintaining reference of every node that is created.
<10, ref(10)>
<15, ref(15)>
<20, ref<20>)
<12, ref<12>)
10 15 20 12 15
head
10 15 20 12 15
T.prev.next = T.next
T.
DLL + H.M
Data is Coming
if x is in H.M
Miss
HIT
Node T = hm[x]
if (T.next == NULL)
if (hm.size == M)
Yes
No
Don't do anything as x is already there.
delete (T)
insertAtBack (T)
Yes
No
delete (head)
hm.erase(x)
insertAtEnd(T)
hm.insert({x,T})
insertAtEnd(T)
hm.insert({x,T})

Clone a doubly Linked List

Create a deep copy of the given doubly linked list.

```
class Node {
    int data;
    Node next; (points to next node)
    Node random; (Anywhere in the Linked List)
}
```




```
Node h2 = h1; // Shallow Copy;
```

Deep copy

1) Create a new linked list with next pointer fixed.

```
Node T1 = h1, T2 = new Node(T1.data)
T1 = T1.next;
while (T1 != NULL) {
    Node x = new Node(T1.data)
    T2.next = x;
    T1 = T1.next;
    T2 = T2.next;
}
T1 = h1; T2 = h2;
Hashmap<Node, Node> hm;
while (T1 != NULL) {
    hm.insert({T1, T2})
    T1 = T1.next;
    T2 = T2.next;
}
T1 = h1, T2 = h2;
while (T1 != NULL) {
    T2.random = hm[T1.random]
    T1 = T1.next;
    T2 = T2.next;
}
```