Linked Lists

Ajit Rajwade

A limitation of arrays

- We have all seen arrays and used them many times.
- The one advantage of an array is that you can access the i-th element of array A by means of a constant time, i.e. O(1), operation, via the syntax A[i].
- However suppose you have to insert a new element somewhere in the middle of the array at some index j.
- You have to first right shift all elements of the array from index j onwards.
- For example, let int A[10] = {10,20,30,40,50};
- If you want to insert a number 25 at index 2, then you have to right shift the elements 30,40,50 and then set A[2] = 25 to produce A = {10,20,25,30,40,50};
- Thus insertion of a new element into an array with n elements is an O(n) operation.

Linked List

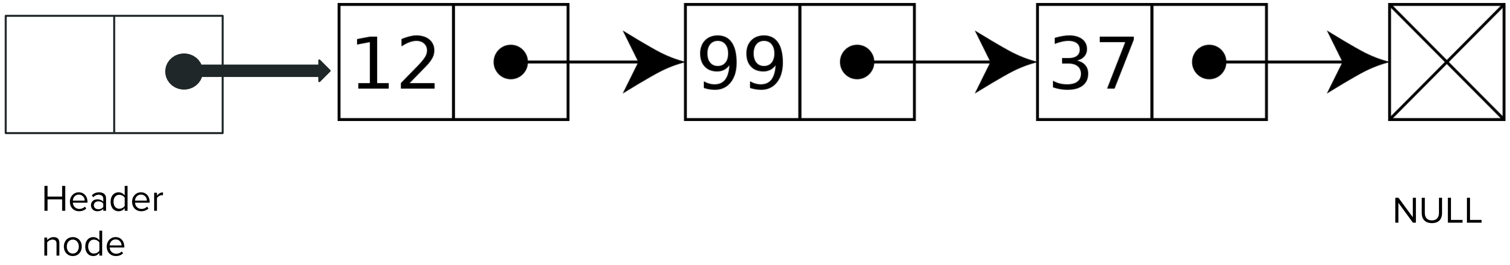
- A (singly) linked list (LL or SLL) will allow you to insert a new element in O(1) time without having to move so many elements around.
- A linked list contains many elements called **nodes** arranged as a chain.
- Each node contains a **value** (say a number, but could be anything else), and a pointer to its next **node**, i.e. the address of the next node.
- This is represented as:

```
struct node{

    int data; // could be any other datatype, including a structure

    node* next; // address of the next node

};
```



- The convention we will adopt is that the first node of a linked list is a **header** node which does **not** contain any valid data.
- However its next pointer points to a node containing some valid data, and is truly the **first data node** of the list.
- An empty linked list upon creation contains just the header node, with its next pointer pointing to NULL.
- After insertion of new nodes, the last node of the linked list should always point to NULL. This NULL marks the **end** of the linked list.
- Memory for each node is allocated dynamically via: `node *q = new node;`

Creation of a linked list

- We allocate memory for the header node.
- The data of the header is meaningless.
- The next is set to NULL.
- The header node is returned to the calling function.

```
node* createLinkedList ()
{
        node* header = new node;
        header->data = 0;
        header->next = NULL;
        return header;
}
```

Appending a node to the linked list

- We consider adding a node to the end of the linked list.
- Starting from the header, determine which is the last node by moving through all the next pointers until you encounter NULL.
- Save the previous pointers during your traversal.
- Allocate memory for a new node and set the next pointer of the last node of the list to this new node.
- See next slide for code.
- Corner case: ensure that the header exists (i.e. it is non-NULL)
- Time complexity: O(n) where n is the number of nodes of the linked list, space complexity: O(1)

```
void appendNode (node* header, int dataval)
{
        node *q = header, *r;

        node *p = new node;
        p->data = dataval;
        p->next = NULL;

        while (q!= NULL)
        {
                r = q;  // previous node
                q = q->next;
        }
         // p becomes the “next” of the
         // last node of the list
        r->next = p;
}
```

Displaying the contents of a linked list

- Starting from the header, traverse the list via the next pointers.
- Print the data at every node, except the data of the header which is invalid.
- Continue this process until you hit the end of the list (NULL).
- Time complexity: O(n) where n is the number of nodes of the linked list, space complexity: O(1)

```
void displayList (node* header)
{
        node *q = header;

        if (header == NULL) { cout << "This list does not
exist"; return;}
        if (header->next == NULL) { cout << "The list is
empty"; return;}

        cout << endl;
        while (q != NULL) {
                if (q!=header) cout << q->data << " ";
                q = q->next;
        }
}
```

Deleting the last node of the list

- Starting from the header, traverse the list using next pointers till you hit the last node (i.e., node for which the next points to NULL)
- All along, maintain a pointer for the previous node (called `prev` – say)
- When you hit the last node, you need to delete `prev->next` and set `prev->next` to NULL
- Corner case: if `header->next` is NULL, there is nothing to delete
- Time complexity: O(n) where n is the number of nodes of the linked list, space complexity: O(1)

```
bool deleteLastNode (node* header)
{
        node *curr = header, *prev;

        if (header->next == NULL) return
false;
        while (curr->next != NULL)
        {
                prev = curr;
                curr = curr->next;
        }
        prev->next = NULL;
        delete curr;

        return true;
}
```

Note: a previous (prev) pointer was not needed in appending a new node to the linked list. But we need it here for deletion to maintain list continuity.

Deleting a linked list

- Delete every node except the header
- Then delete the header itself

```
void deleteLinkedList (node *header)
{
        do
        {
                bool flag = deleteLastNode (header);
                if (flag == false) break;
                displayList (header);
        } while (true);

        delete header;
        cout << "deleting the header node: no more operations
can be performed on this list";
}
```

Concatenating one linked list to another

- Traverse list 1 from its header till its last node (call that node curr1)
- Set curr1->next equal to header2->next (header2 is the header of list 2)
- Time complexity: O(n1) where n1 is the number of nodes of the list 1, space complexity: O(1)
- Note: the time complexity is not O(n1+n2) which it would have been given an array!
- Implement this one yourself!

Reversing a linked list

- Maintain three pointers: current, next and previous
- Initialize current to header initially, and previous to NULL initially
- Set next = current->next, current->next = previous, previous = current, current = next
- Keep doing this as long as current is not NULL
- Time complexity: O(n), space complexity: O(1)

```c
void reverseLinkedList(node *header)
{
        if (header == NULL) return;
        // Initialize current, previous and next pointers
        node* curr = header->next;
        node *prev = NULL, *next = NULL;

        while (curr != NULL) {
            // Store next
            next = curr->next;
            // Reverse current node's pointer
            curr->next = prev;
            // Move pointers one position ahead.
            prev = curr;
            curr = next;
        }
        header->next = prev;
}
```

Deleting a node with a certain data value

- Let us say we want to delete a node with a certain data value
- Starting from the header, you traverse the list till you encounter a node with a matching data value (call it `current`)
- Maintain pointers to the immediate previous node in the linked list (call it `previous`).
- `previous->next` should be set to `current->next`
- Delete the current node
- If there is no node with matching node, there is nothing left to be done.
- Implement this one yourself!

Deleting a node with a certain data value

- Note that the time complexity of this operation is O(n) just to locate the node with matching data value.
- This would have been the case with an array as well.
- However once the node is located, the actual deletion just takes a constant time, as no rearrangement is required (unlike with an array)
- Similarly write code to insert a new node between a node containing a matching data value and then its next node. Note once you locate a node after which you need to insert a new node, the actual insertion takes only O(1) time unlike an array which would require O(n) time for rearrangement.
- Here again, you will see that no re-arrangement of the linked list is needed (unlike with an array).

Linked Lists: disadvantage

- Given the header of a linked list, you cannot access any element in O(1) time.
- To access the k-th node, you need k steps starting from the header node.
- In contrast, for an array, every element had the same access time irrespective of its position in the array.
- Given a sorted array, you can therefore do binary search.
- This is impossible in a linked list even if its nodes are in sorted order of data values.

Linked Lists: Pitfalls

- Make sure you allocate memory for each node via `new`.
- When you delete the linked list or a node, you should use `delete` to actually free the memory. Otherwise you will get memory leaks.
- Avoid mistakes such as `p->next = p;` which will produce infinite loops.
- Make sure that all next pointers are correctly set.
- When you insert a new node in the middle or delete a node, make sure that the continuity of the linked list is maintained.
- You can confirm the continuity by displaying the linked list after each operation.