Tecnológico de Monterrey

Reflection Activity 2.2

Liliana Solórzano Pérez A01641392

11 de October del 2022

Programming of Data Structures and Fundamental Algorithms

(Gpo 613)

Professors

Jorge Enrique González

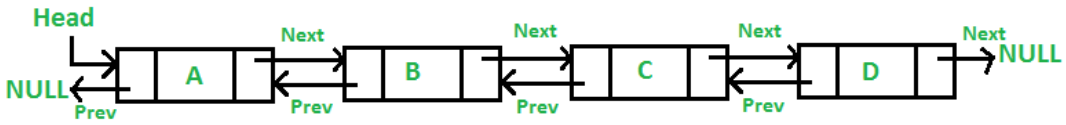
Zapata Luis Ricardo Peña Llamas

What is a linked list?

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list. Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element. The first node is always used as a reference to traverse the list and is called HEAD. The last node points to NULL.

This difference in the data storage scheme decides which data structure would be more suitable for a given situation. A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

The elements in a linked list are linked using pointers as shown in the below image:



Why we preferred to use a Double Linked List

The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, the allocated memory is generally equal to the upper limit irrespective of usage, and in practical uses, the upper limit is rarely reached.

Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create a room existing elements have to be shifted.

From a memory allocation point of view, linked lists are more efficient than arrays. Unlike arrays, the size of a linked list is not predefined, allowing the linked list to increase or decrease in size as the program runs. This is possible because to insert or delete from a linked list, the pointers need to be updated accordingly.

Function - Create a linked list

Inserting a new node to the beginning or end of a linked list takes constant time (O(1)) as the only steps are to initialize a new node and then update the pointers. Likewise, if there were a tail pointer (similar to the head pointer) insertion to the end of the linked list would also be O(1). However, insertion to the middle of the linked list takes linear time (O(n)) as iteration over n elements is required to get to the correct location before inserting the node. Similarly, the deletion of the nodes at the beginning and end of the linked list takes constant time while deleting a node in the middle of the linked list takes linear time.

```
// Insert a node at the beginning of the list

// Complexity - O(1)

Node *insert_node_begin(Node *head, int val)

{

   Node *new_node = new Node; // Create a new node

   new_node->val = val;        // Set the value of the new node

    new_node->next = head;      // Set the next pointer of the new node to
the head

    new_node->prev = NULL;      // Set the previous pointer of the new node
to NULL



    // If the list is not empty, set the previous pointer of the head to
the new node

   if (head)

   {
```

```
        new_node->next = head;

        head->prev = new_node;

    }

    return new_node;

}



// Insert a node at the end of the list

// Complexity - O(n)

Node *insert_node_end(Node *head, int val)

{

    Node *new_node = new Node; // Create a new node

    new_node->val = val;        // Set the value of the new node

    new_node->next = NULL;      // Set the next pointer of the new node to
NULL



    // If the list is not empty, set the next pointer of the last node to
the new node

    if (head)

    {

        Node *current = head;

        while (current->next)

        {

            current = current->next;

        }

        current->next = new_node;

        new_node->prev = current;
```

```
    }
    else
    {
        // else If the list is empty, set the new node as the head
        new_node->prev = NULL;
        head = new_node;
    }
    return head;
}

// Insert a node after a given node
// Complexity - O(n)
Node *insert_node_after(Node *head, int val, int after)
{
    Node *new_node = new Node; // Create a new node
    new_node->val = val;       // Set the value of the new node

    // If the list is not empty...
    if (head)
    {
        Node *current = head;

        // Find the given node
        while (current->val != after)
        {
```

```
            current = current->next;

        }



         // Set the next pointer of the new node to the next pointer of the
given node

        new_node->next = current->next;

        new_node->prev = current;

        current->next = new_node;



        // If the given node is not the last node, set the previous pointer
of the next node to the new node

        if (new_node->next)

        {

            new_node->next->prev = new_node;

        }

    }

    return head;

}
```

Double-Linked lists have slower search times than arrays as random access is not allowed. Unlike arrays where the elements can be searched by index, linked lists require iteration. This means that if you want to get the data on the tenth node, the head pointer can be used to get to the first node, the pointer on the first node can be used to get to the second node, and so forth until the tenth node is reached. This means that the more nodes that must be iterated while searching for a node, the longer the search time. This implies that the search time for a linked list is linear time or Big O of n (O(n)).

Function - read/search an element in a double-linked list

We used a function to read each node of the linked list from the file. We use these nodes to create a new linked list and after reading all the nodes from the file we return the head of the newly created linked list. First, the function checks if the list is empty. If it is empty, the program shows a message that our list is empty. Otherwise, it will read each node of the list, starting from the head until it reaches the NULL value.

```
// Search position of a node

// Complexity - O(n)

Node *search_position(Node *head, int pos)

{

   Node *current = head; // Set the current node to the head

   int i = 0;             // Set the position counter to 0

    while (i < pos)         // While the position counter is less than the
given position

   {

       current = current->next; // Set the current node to the next node

       i++;                     // Increment the position counter

   }



   return current;

}



// Search value of a node

//  Complexity - O(n)

Node *search_value(Node *head, int val)
```

```
{

   Node *current = head;        // Set the current node to the head

    while (current->val != val) // While the value of the current node is
not equal to the given value

   {

       current = current->next; // Set the current node to the next node

   }



   return current;

}



// Update a value of a node

// Complexity - O(n)
```

Function - Update an element from the double linked list

Approach: The following steps are:

- Split the list from the middle. Perform front and back split. If the number of elements is odd, the extra element should go in the 1st(front) list.
- Reverse the 2nd(back) list.
- Perform the required subtraction while traversing both lists simultaneously.
- Again reverse the 2nd list.
- Concatenate the 2nd list back to the end of the 1st list.

```
// Update a value of a node

// Complexity - O(n)
```

```
void update_value(Node *head, int val, int new_val)

{

   Node *current = head; // Set the current node to the head



   // While the value of the current node is not equal to the given value

   while (current->val != val)

   {

       current = current->next;

   }



   // Set the value of the current node to the new value

   current->val = new_val;

}
```

Function - Delete an element

Delete from a double Linked List:

You can delete an element in a list from:

- Beginning
- End
- Middle

We used a function to make it more friendly for the user to run the program. The complexity of deleting an element by position is O (n).

```
// Delete a node

// Complexity - O(n)

Node *delete_node(Node *head, int val)

{

   // If the list is not empty...

   if (head)

   {

       Node *current = head; // Set the current node to the head



         // While the value of the current node is not equal to the given
value

       while (current->val != val)

       {

           current = current->next;

       }



       if (current->prev)

        {                                          // If the previous node
exists

           current->prev->next = current->next; // Set the next pointer of
the previous node to the next pointer of the current node

       }

       else

       {
```

```
            head = current->next; // else Set the head to the next node

        }

        if (current->next)

        {                                           // If the next node exists

                current->next->prev = current->prev; // Set the previous
pointer of the next node to the previous pointer of the current node

        }



        delete current; // Delete the current node

    }

    return head;

}
```

Output of the code

```
++++ The program begins ++++
NULL <-> 3 <-> 6 <-> 7 <-> 4 <-> 5 <->  NULL
NULL <-> 3 <-> 6 <-> 7 <-> 4 <-> 5 <-> 2 <-> 1 <-> 8 <-> 10 <-> 9 <->  NULL
NULL <-> 3 <-> 6 <-> 7 <-> 4 <-> 5 <-> 11 <-> 2 <-> 1 <-> 13 <-> 8 <-> 10 <-> 12 <-> 9 <->  NULL
Position 10: 10
Value 1: 1
NULL <-> 3 <-> 6 <-> 7 <-> 4 <-> 5 <-> 11 <-> 11 <-> 1 <-> 13 <-> 8 <-> 10 <-> 12 <-> 9 <->  NULL
NULL <-> 3 <-> 6 <-> 7 <-> 4 <-> 5 <-> 11 <-> 11 <-> 1 <-> 13 <-> 8 <-> 12 <-> 9 <->  NULL
++++ The program ends ++++
```

Conclusion of using ADT (in this case double linked list)

In conclusion there are many different data structures. Each data structure has strengths and weaknesses which affect performance depending on the task. Today, we explored two data structures: arrays and linked lists. Arrays allow random access and require less memory per element (do not need space for pointers) while lacking efficiency for insertion/deletion

operations and memory allocation. On the contrary, linked lists are dynamic and have faster insertion/deletion time complexities. However, linked lists have a slower search time and pointers require additional memory per element in the list. Figure 10 below summarizes the strength and weaknesses of arrays and linked lists.

Citas en Formato APA

- "Doubly Linked List: Set 1 (Introduction and Insertion)." GeeksforGeeks, 2 Sept. 2022, https://www.geeksforgeeks.org/doubly-linked-list/.
- Krohn, Hermann. "Linked Lists vs. Arrays." Medium, Towards Data Science, 14 July 2019, https://towardsdatascience.com/linked-lists-vs-arrays-78746f983267.