DoublyLinkedLists

August 8, 2020

1 Doubly Linked Lists

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ListDouble.html
- https://en.cppreference.com/w/cpp/container/list

1.0.1 Table of Contents

1.1 Introduction

- **Singly Linked List** allows for direct access from a list node only to the next node in forward direction
- **Doubly Linked List** allows access in both directions – forward and backward
  - giving easy access to next node and previous node

1.2 Doubly Linked List

- also called two-way list
- each node is depicted with three boxes (members) each holding:
  1. data (middle box)
  2. address/pointer to the next node (right box)
  3. address/pointer to the previous node (left box)
- diagonal slash (see last and first node) represents NULL pointer meaning it's not pointing to another node
- head or first is a special pointer pointing to the first (header) node
- tail or last is a special pointer pointing to the last (trailer) node
- use pointer to traverse through the linked list (unlike index in array-based list)

1.3 Common Operations

- inserting and deleting nodes are common operations but need to deal with many cases.
- if header and trailer nodes are used without actually storing the data, simplifies many special cases
  - see visualization at: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ListDouble.html

1.4 Implemenation of Node

- since a node is a complex type with data (of various type) and pointers, we use struct or class to implement it

[1]:

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

[2]:

```
struct Int_Node {
    int data; // int data
    Int_Node * next; // address of the next node
    Int_Node * prev; // address of the previous node
};
```

[ ]:

```
// better implementation
template <class T>
struct Node {
    T data; // data of some type T
    Node<T> * next;
    Node<T> * prev;
};
```

1.5 Creating a Doubly Linked List

- add elements 10, 20, 30, etc.
- doubly linked list of: 10 <-> 20 <-> 30

[ ]:

```
Int_Node *head, *tail, *temp;
```

[ ]:

```
// create empty header and trailer nodes as shown in figure above
temp = new Int_Node;
temp->data = 0;
temp->prev = NULL;
temp->next = NULL;
head = temp; // head points to header node

temp = new Int_Node;
temp->data = 0;
temp->prev = head; // trailer points to header
temp->next = NULL;
tail = temp;

head->next = tail; // header points to trailer
```

1.6 Push Back Element

- inserting element at the end of the doubly linked list
- algorithm steps:

1. create a new node with data
   - make new node’s next point to trailer node
   - make new node’s prev point to trailer’s prev node
   - make trailer node’s prev next point to the new node
   - make trailer node’s prev point to the new node

[ ]:

```
// create and add the new node with 10 at the end
temp = new Int_Node;
temp->data = 10;
temp->next = tail;
temp->prev = tail->prev;
tail->prev->next = temp;
tail->prev = temp;
```

[ ]:

```
// create and add the new node with 20 at the end
temp = new Int_Node;
temp->data = 20;
temp->next = tail;
temp->prev = tail->prev;
tail->prev->next = temp;
tail->prev = temp;
```

[ ]:

```
// create and add the new node with 20 at the end
temp = new Int_Node;
temp->data = 30;
temp->next = tail;
temp->prev = tail->prev;
tail->prev->next = temp;
tail->prev = temp;
```

1.7 Traversing Doubly Linked List

- visiting every node of the linked list
  - access data, check and or update data
- can be traversed both in forward and backward directions

[ ]:

```
void traverseForward(Int_Node *head) {
    // start from header's next and go through every node
    // stop before trailer
    Int_Node * curr = head->next;
    cout << "[";
    while (curr != tail) {
        cout << " " << curr->data;
        curr = curr->next;
    }
    cout << " ]";
}
```

[11]:

```
traverseForward(head);
```

```
[ 10 20 30 ]
```

[12]:

```
void traverseBackward(Int_Node *tail) {
    // start from trailers's prev and go through every node
    // stop before header
    Int_Node * curr = tail->prev;
    cout << "[";
    while (curr != head) {
        cout << " " << curr->data;
        curr = curr->prev;
    }
    cout << " ]";
}
```

[13]:

```
traverseBackward(tail);
```

```
[ 30 20 10 ]
```

1.8 Push Front Element

- inserting element at the beginning of the doubly linked list
- similar to push back operation
- algorithm steps:
    1. create a new node with data
    - make new node->next point to the head->next
    - make new node->prev point to the head
    - make head->next point to the new node
    - make new node->next->prev point to the new node

[14]:

```
// insert a new node at the beginning (push_front)
temp = new Int_Node;
temp->data = 100;
temp->next = head->next;
temp->prev = head;
head->next = temp;
temp->next->prev = temp;
```

[15]:

```
traverseForward(head);
```

```
[ 100 10 20 30 ]
```

[16]:

```
traverseBackward(tail);
```

```
[ 30 20 10 100 ]
```

[17]:

```
// insert a new node at the beginning (push_front)
temp = new Int_Node;
temp->data = 200;
temp->next = head->next;
temp->prev = head;
head->next = temp;
temp->next->prev = temp;
```

[18]:

```
traverseForward(head);
```

```
[ 200 100 10 20 30 ]
```

1.9 Doubly Linked List Remove

- remove an element/node from the linked list
- algorithm steps:
  1. use a pointer, current
     - current is the node that needs to be deleted if found
  2. if node is found delete it
     - update the doubly linked list

[19]:

```
Int_Node * curr;
```

[23]:

```
// delete 2nd node from the list
// NOTE: header is not an actual node!
curr = head->next->next;
curr->prev->next = curr->next;
curr->next->prev = curr->prev;
delete curr;
```

[24]:

```
traverseForward(head);
```

```
[ 200 20 30 ]
```

1.10 Doubly Linked List Insert

- insert an element/node after certain node in the linked list
- similar to push front operation
- algorithm steps:
  1. create a new node with the data
  - find the location where the new node needs to be inserted after, say curr
  - insert the new node at that location
  - update doubly linked list

[ ]:

```
// insert element as the 2nd node (after the first node) with key value 100
// NOTE: header node is not an actual node!
curr = head->next;
temp = new Int_Node;
```

```
temp->data = 100;
temp->next = curr->next;
temp->prev = curr;
curr->next = temp;
temp->next->prev = temp;
```

[ ]:

```
traverseForward(head);
```

1.11 Doubly Linked List Implementation as ADT

- following Doulby Linked list as ADT works for integer data
- it can be easily converted into a template class
  - this is left as an exercise

[3]:

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

[4]:

```
struct Int_Node {
    int data; // int data
    Int_Node * next; // address of the next node
    Int_Node * prev; // address of the previous node
};
```

```
input_line_12:1:8: error: redefinition of

'Int_Node'
struct Int_Node {
       ^

input_line_9:1:8: note: previous definition is here
struct Int_Node {
       ^



        Interpreter Error:
```

[5]:

```
class IntDoublyList {
    private:
        Int_Node * head;
        Int_Node * tail;
        size_t count;
        // removes curr node
        void remove(Int_Node* curr) {
            curr->prev->next = curr->next;
            curr->next->prev = curr->prev;
            delete curr;
            this->count--;
        }

    public:
        IntDoublyList() {
            this->count = 0;
            // create empty header and trailer nodes as shown in figure above
            Int_Node * temp = new Int_Node; //create header node
            temp->data = 0;
            temp->prev = NULL;
            temp->next = NULL;
            head = temp; // head points to header node

            temp = new Int_Node; // create trailer node
            temp->data = 0;
            temp->prev = head; // trailer points to header
            temp->next = NULL;
            tail = temp;

            head->next = tail; // header points to trailer
        }

        bool empty() const {
            return this->count == 0;
        }

        // adds an element to the end
        void push_back(int data) {
            Int_Node * node = new Int_Node;
            node->data = data;
            node->next = tail;
            node->prev = tail->prev;
            tail->prev->next = node;
            tail->prev = node;
            this->count++;
        }
        // inserts an element to the beginning
        void push_front(int data) {
            // FIXME
        }
        // access the last element
        int back() {
            return tail->prev->data;
        }

        // return the size of the list
        size_t size() {
            return this->count;
        }

        // access the first element
        // FIXME - implement method to access the data in first node

        // removes the last element
        void pop_back() {
            // nothing to do in an empty list
            if (empty()) return;
            this->remove(tail->prev);
        }

        // removes the first element
        // FIXME - implement a method to remove the first node

        // visits every node and prints the data
        // traverse in forward direction
        void traverseForward() {
            cout << "[";
            Int_Node * curr = head->next;
            while (curr != tail) {
                cout << " " << curr->data;
                curr = curr->next;
            }
            cout << " ]";
        }

        // traverseBackward
        // visits every node and prints the data in backward direction
        void traverseBackward() {
            // FIXME...
        }

        // insert a node with a given data after the node with the after_key value
        // if the element with after_key not found, insert data at the end
        void insert_after(int after_key, int data) {
            // FIXME:
        }

        // clears the linked list deleting all the nodes
        // except for the header and trailer nodes
        void clear() {
            // FIXME...
        }
};
```

[6]:

```
// test IntDoublyList with some data
IntDoublyList ilist;
```

[8]:

```
ilist.traverseForward();
```

```
[ ]
```

[9]:

```
ilist.push_back(10);
ilist.traverseForward();
```

```
[ 10 ]
```

[10]:

```
ilist.push_back(20);
ilist.push_back(30);
ilist.traverseForward();
```

```
[ 10 20 30 ]
```

[11]:

```
ilist.pop_back();
ilist.traverseForward();
```

```
[ 10 20 ]
```

1.11.1 Exercises

1. Linked lists are better than array-based lists when the final size of the list is known in advance.
    1. True
    - False
2. Fix all the FIXMEs and test the fixes of doubly linked list ADT.
3. Convert Doubly Linked List ADT as a template class to store data of any type in the node.

[ ]:

```
```