LinkedLists

August 8, 2020

1 Linked Lists

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ListLinked.html

1.0.1 Table of Contents

1.1 Introduction

- fundamentally different way of storing a large collection of data as list
- consists of list of nodes connected ("linked") together via pointers
- uses dynamic memory allocation
  - allocates memory for new list element/node as needed
- commonly two types:
  1. singly linked list (STL forward_list)
  2. doubly linked list (STL list)

1.2 Singly Linked List

- also called one-way list
- each node is depicted with two boxes (members) each holding:
  1. data (left box)
  2. address/pointer to the next node in the list (right box)
- diagonal slash (see last 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)
- inserting and deleting node are common operations but need to deal with many cases.
  - see visualization at: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ListLinked.html

1.3 Implemenation of Node

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

[1]:
```
struct Int_Node {
    int data; // int data
    Int_Node * next; // address of the next node
};
```

[2]:
```
// better implementation
template <class T>
struct Node {
    T data; // data of some type T
    Node<T> * next;
};
```

[3]:
```
// linked list of: 10 -> 20 -> 30
#include <iostream>
using namespace std;
```

1.4 Creating a linked list

- add elements 10, 20, 30, etc.

[4]:
```
Int_Node *head, *tail, *temp;
```

[5]:
```
// empty linked list
head = tail = NULL;
```

1.5 Push Back Element

- inserting element at the end of the linked list
- algorithm steps:
    1. create a new node with data
    - handle special case where linked list is empty
    - make tail->next point to new node
    - make tail point to the new node

[6]:
```
// create and add the first node
temp = new Int_Node;
temp->data = 10;
temp->next = NULL;

// list is empty so far so add the first node
head = temp; //update head
tail = temp; // update tail
```

[6]: @0x7ffee81618d0

[7]:
```
//push_back more elements
temp = new Int_Node;
temp->data = 20;
temp->next = NULL;

tail->next = temp;
// update tail
tail = temp;
```

[8]:
```
//push_back more elements
temp = new Int_Node;
temp->data = 30;
temp->next = NULL;

tail->next = temp;
// update tail
tail = temp;
```

1.6 Traversing Linked List

- visiting every node of the linked list
  - access data, check and or update data

[9]:
```
void traverse(Int_Node *head) { // start from head and go through every node
    Int_Node * curr = head;
    cout << "[";
    while (curr != NULL) {
        cout << " " << curr->data;
        curr = curr->next;
    }
    cout << " ]";
}
```

[10]:
```
traverse(head);
```

```
[ 10 20 30 ]
```

1.7 Visualize Push Back using pythontutor.com: https://goo.gl/iQ8xJH

1.8 Push Front Element

- inserting element at the beginning of the linked list
- algorithm steps:
  1. create a new node with data
  - make new node->next point to the head
  - update head to point to the new node

[11]:
```
// insert a new node at the beginning (push_front)
temp = new Int_Node;
temp->data = 200;
temp->next = head;

head = temp;
```

[12]:
```
traverse(head);
```

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

[13]:
```
// insert a new node at the beginning (push_front)
temp = new Int_Node;
temp->data = 100;
temp->next = head;

head = temp;
```

[14]:
```
traverse(head);
```

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

1.9 Visualize Push Front using pythontutor.com: https://goo.gl/Qnq51b

1.10 Linked List Remove

- remove an element/node from the linked list
- algorithm steps:
    1. use two pointers, previous and current
        - current is the node that needs to be deleted if found
        - previous is the node right before current
    2. if node is found delete it
        - update the linked list
    - if head is deleted, update head
    - if tail is deleted, update tail

[15]:
```
Int_Node * curr;
Int_Node * pre;
```

[16]:
```
// delete 2nd node from the list
curr = head->next;
pre = head;
pre->next = curr->next;
delete curr;
```

[17]:
```
traverse(head);
```

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

1.11 Linked List Insert

- insert an element/node after certain node in the linked list
- algorithm steps:
    1. create a new node
    - find the location where the new node needs to be inserted
    - insert the new node at that location
    - if the new node is inserted at the beginning, update head
    - if the new node is inserted at the end, update tail

[18]:
```
// insert element as the 2nd node (after the first node) with key value 100
temp = new Int_Node;
temp->data = 200;
temp->next = NULL;
curr = head; // pointing to the first node
temp->next = curr->next;
curr->next = temp; // insert temp after current
```

[18]: @0x7ffee81618d0

[19]:
```
traverse(head);
```

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

1.12 Detect and remove cycle from a linked list

- let's say the next of some node points to some earlier nodes creating a cycle (by error)
- determine if the linked list has a cycle
- return 1 if there's a cycle; 0 otherwise
- use unordered_set to keep track of visited node's addresses

[20]:
```
# include <unordered_set>
```

[21]:
```
int detectAndRemoveCycle(Int_Node * head) {
    Int_Node * last = nullptr;
    unordered_set< Int_Node* > visited;
    bool cycle = false;
    while (!cycle && head != nullptr) {
        // if the node is not in visited set, add it into the set
        if (visited.find(head) == visited.end()) {
            visited.insert(head);
            last = head;
            head = head->next;
        }
        else {
            cycle = true;
            last->next = nullptr;
        }
    }
```

```
    return cycle?1:0;
}
```

[22]:
```
// test detectAndRemoveCycle by adding a cycle
cout << tail->data;
```

30

[23]:
```
tail->next = head->next->next;
// not there's a cycle
cout << tail->next->data;
```

10

[24]:
```
cout << detectAndRemoveCycle(head);
```

1

[25]:
```
traverse(head);
```

[100 200 10 20 30]

1.13 A Better Approach to Represent Linked List

- insert and delete algorithms stated above have many special cases to worry about
  - makes it harder to understand, and increases the chance of introducing bugs
- many special cases can be eliminated if **header node** and **trailer node** are used
  - these nodes do not store actual data/value
- the following diagram shows empty linked list with header and trailer nodes
- the following diagram shows linked lists with some nodes with data, header and trailer nodes

1.14 Remove

- algorithm steps to remove a node
  1. find the node to be deleted
  - delete the node

1.15 Insert

- algorithm steps to insert an element after certain node
  1. find the node to insert after
  2. insert new node after the node

1.16 Linked List Implementation as ADT

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

[26]:
```
#include <iostream>
#include <sstream>
#include <cassert>
using namespace std;
```

[ ]:
```
struct Int_Node {
    int data; // int data
    Int_Node * next; // address of the next node
};
```

[28]:
```
class IntLinkedList {
    private:
        Int_Node * head; // pointer to the header node
        Int_Node * tail; // pointer to the trailer node
        size_t nodeCount; // keep track of number of nodes

    public:
        IntLinkedList() {
            this->nodeCount = 0;
            Int_Node *temp = new Int_Node;
            temp->data = 0;
            temp->next = nullptr;
            this->head = temp;
            temp = new Int_Node;
            temp->data = 0;
            temp->next = nullptr;
            this->tail = temp;

            // link head with tail
            this->head->next = this->tail;
        }

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

        // adds an element to the end (insert end)
        void push_back(int data) {
            Int_Node * node = new Int_Node;
            node->data = 0;
            node->next = NULL;

            this->tail->data = data; // update trailer node's data
            this->tail->next = node; // link new node at the end
            this->tail = node; // make new node the trailer node

            this->nodeCount++;
```

```
    }

    // inserts an element to the beginning
    void push_front(int data) {
        // FIXME
    }

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

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

    // return string representation of linked list
    string toString() {
        stringstream ss;
        ss << "[";
        Int_Node * curr = head->next; // ignore header
        while (curr != tail) { // stop before trailer
            ss << " " << curr->data;
            curr = curr->next;
        }
        ss << " ]";
        return ss.str();
    }


    // remove node with given key
    void remove(int key) {
        assert(!empty());
        Int_Node *curr = head; //start from header node
        bool found = false;
        while (curr->next != tail) {
            if (curr->next->data == key) {
                found = true;
                break;
            }
            curr = curr->next;
        }
        if (found) {
            // node found delete it
            Int_Node *node = curr->next; //copy curr->next to properly␣
↪delete it
            // point around unneeded node
            curr->next = curr->next->next;
```

```
                delete node;
                this->nodeCount--;
            }
        }


        // 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
        }

};
```

[29]:
```
// test IntLinkedList with some data
IntLinkedList ilist;
```

[30]:
```
cout << ilist.toString();
```

```
[ ]
```

[31]:
```
ilist.push_back(10);
cout << ilist.toString();
```

```
[ 10 ]
```

[32]:
```
ilist.push_back(20);
ilist.push_back(30);
cout << ilist.toString();
```

```
[ 10 20 30 ]
```

[33]:
```
cout << "Linkedlist has " << ilist.size() << " nodes." << endl;
```

```
Linkedlist has 3 nodes.
```

[34]:
```
ilist.remove(20);
cout << ilist.toString();
```

```
[ 10 30 ]
```

1.17 Array-based List Vs Linked List

1.17.1 Disadvantages of array-based lists

- list size must be predetermined before the array can be allocated
- list cannot grow beyond their predetermined size
- memory is potentially wasted if array is not completely filled

- insertion operation requires shifting elements down (at most $O(n)$ operations in the worst case)
- deletion operation requires shifting elements up (at most $O(n)$ operations in the worst case)

1.17.2 Advantages of linked lists

- list size must not be known before hand
- list can grow dynamically as more data need to be stored
- no over allocating space and waste of memory
- if pointer is predetermined, insertion and deletion takes constant time $O(1)$ operation

1.17.3 Advantages of array-based lists

- no extra space is required to store pointer in each node
- better suited if the size of the data is known and array is guaranteed to be filled
- provides random access to data via index

1.17.4 Disadvantages of linked lists

- memory overhead for storing pointer in each node
- no random access to data is possible… must traverse the list to access elements

1.17.5 Exercises

1. In a linked list, the successive elements in the list:
    1. Need not occupy contiguous space in memory
    - Must not occupy contiguous space in memory
    - Must occupy contiguous space in memory
    - None of the above
2. Fix all the FIXMEs of Singly Linked List ADT above and test the fixes
3. Convert Singly Linked List ADT to a template class to store data of any type in the node

[ ]:

```

```