CSCI 262
Data Structures
5 – Linked Lists, part 2

REVIEW
Since it has been a week...

Overview
- Our first data structure: Linked List
- Practice some old (or recently new) concepts:
  - Pointers
  - Classes & objects
  - Encapsulation
- Introduce some new concepts at high level:
  - Dynamic memory allocation
  - Analysis of algorithms

Describe, create, and analyze a very simple linked list

Linked Lists
- A linked structure composed of node objects
- Each node contains one data value
- Each node contains a pointer to the next node
- Nodes can be independently created/destroyed
  - ...through the use of dynamic allocation – more soon!

Linked Lists Illustrated

Node Class
Here’s a very simple implementation of a node:

class node {
    public:
        int data;
        node* next;
};

Graphical representation
Note NULL pointer in tail node!
(Should use nullptr, but NULL fit better.)
Creating Nodes

Where do nodes come from…?
Don’t worry about where just yet, but here’s how we do it:

node* ptr = new node;

Linked List Class: Declaration

class linked_list {
  public:
    void add_to_back(int val);
    void print();
    // more to come!
  private:
    class node {
      public:
        int _data;
        node* _next;
      };
      node* _head = nullptr;
      node* _tail = nullptr;
      int _size = 0;
  };

This declares a class that can only be used within the linked_list class – great for our purposes, because user doesn’t need to know about it!

Note the initializers on these variables; only allowed in C++ 11 and later. We’re using these in place of a default constructor.

Linked List Class: Methods

void linked_list::add_to_back(int val) {
  // make new tail node
  node* p = new node;
  p->_data = val;
  p->_next = nullptr;
  // if list is empty, new node becomes both head and tail
  if (_head == nullptr) {
    _head = _tail = p;
  } else {
    _tail->_next = p;
    _tail = p;  // update tail only
    _size++;
  }
}

void linked_list::print() {
  for (node* p = _head; p != nullptr; p = p->_next) {
    cout << p->_data << endl;
  }
}

In which we add some extra operations, and talk more about dynamic allocation and analysis

PART 2

Easy and Hard Operations

Consider: given that we are keeping pointers to head and tail, what is involved in the following operations?
  ▪ Adding to the front
  ▪ Adding to the back (already done)
  ▪ Removing from the front
  ▪ Removing from the back

Add to Front

void linked_list::add_to_front(int val) {
  // make new head node
  node* p = new node;
  p->_data = val;
  p->_next = nullptr;
  // if list is empty, new node becomes both head and tail
  if (_head == nullptr) {
    _head = _tail = p;
  } else {
    // attach old head to p, p becomes head
    _head->_next = p;
    _head = p;
    _size++;
  }
}
Add to Front Illustration

list.add_to_front(81);

Destroying Nodes

Remember that we used new to create a node:
node* ptr = new node;

When we are done with a node, we should use delete to properly dispose of it:
delete ptr;

We’ll discuss more in depth later about why this step is important!

Remove from Front

void linked_list::remove_from_front() {
    // make sure we don’t operate on an empty list!
    if (_head == nullptr) return;

    // save a pointer to head
    node* p = _head;

    // move head pointer down one node
    _head = _head->next;

    // destroy the old head
    delete p;
    _size--;
}

Remove from Front Illustration

Remove from Back

How do we go about this?
  • We have a tail pointer — so we know what node we are going to destroy
  • But, how do we update the tail pointer?
    • No pointers going in other direction (singly-linked list*)
    • Only solution: start from head and find the next-to-last node.

Remove from Back Code

void linked_list::remove_from_back() {
    // make sure we don’t operate on an empty list!
    if (_head == nullptr) return;

    // special case: only one node
    if (_head == _tail) {
        delete _head;
        _head = _tail = nullptr;
    }
    else {
        // find the next-to-last node
        node* p = _head;
        while (p->next != _tail) p = p->next;
        delete _tail;
        _tail = p;
    }
    _size--;
}

Remove from Back

*We are implementing a singly-linked list, but doubly-linked lists are a popular alternative. More maintenance and storage, but some operations are easier.
Remove from Front vs Remove from Back

- Remove from front:
  - Do a couple of variable assignments
  - Destroy head node
  - Decrement size

- Remove from back:
  - Loop over most of list (lots of pointer assignments)
  - Destroy tail node
  - Do a pointer assignment
  - Decrement size

Big O Introduction

- Remove from front operation:
  - Cost to remove front element is unaffected by size of list
  - Number of operations is constant – no loops
  - We say: remove from front is an $O(1)$ operation

- Remove from back:
  - Number of operations depends on size of list
  - If N elements in list, we do roughly N iterations of a loop
  - We say: remove from back is an $O(N)$ operation

  Basic idea: big O measures how an operation’s cost grows with the size of the input (in this case, constant vs linearly)

Revisit: Why Linked Lists?

Seems overly complex: why not just use a vector?

It’s all about trade-offs:

- Vectors are built on arrays
  - On the plus side: random access!
  - Low cost to get/set values at a particular index – $O(1)$ operation
  - However:
    - (Relatively) difficult to grow and shrink
    - Insert/remove operations expensive – $O(N)$ operation on average

- Linked lists built on independent nodes
  - Grow/shrink trivial
  - Insert/remove inexpensive-ish – $O(1)$, if we have the right pointers!
  - However, we lose random access – access to $k^{th}$ element is $O(N)$

Applications

- Lower-level data structure for implementing:
  - Stacks – a data structure that only allows adding/removing elements at one end
  - Queues – a data structure that allows adding to one end and removing from the other

- Linear containers for use when lots of cutting/splicing operations are important
  - E.g., modeling DNA manipulation operations
  - Keeping track of Undo chains (e.g., Photoshop, Word)

- Lots of uses in other data structures:
  - Hashables
  - Graphs

Up Next

- Today
  - Lab 2 due
  - Reading: Chapter 14.4 – 14.6

- Friday
  - Lab 3 – TBA
  - APT 2 assigned
  - Project 1 due