2020/09/16 - Dealing Cards (Part 2)

2020年9月16日 7:01

SYNOPSIS

- Go over LAB 2B.

LAB 2B

- On Canvas, go to the "Lab 2.1" assignment and read the "lab2.html" file attached. All lab details will be there.
- There are 2 parts. There are 2 due dates:
  - Sept 12 (SAT): Prog2a (BASIC CARD DEALING)
  - Sept 19 (SAT): Prog2b (DITTO w/ LINKED LIST)
- For LAB 2B, simply COPY "Prog2a.cpp" over to "Prog2b.cpp". You will simply be modifying it.

SUBMISSION COMMANDS

LAB 2.1

```
tar -cvf lab2_1.tar Prog2a.cpp
```

LAB 2.2

```
tar -cvf lab2_2.tar Prog2b.cpp
```

PROG 2B

- An introduction to singly-linked lists... with a twist!
- Do NOT use #include <list>. This is cheating and we will check for it (regardless of "deals").
- Make a list class with a node sub-class (private). It must at least follow the following requirements:

LIST CLASS

- Constructor + Destructor
- insert function which inserts an int in the list in sorted order. If element already exists, delete it from the list instead. Return 1 if an insertion took place. Return 0 if a deletion took place.
- A function prototype for operator<< to be overloaded. Mark it as a friend so ostream (cout) can access class private variables.
- Private node* which will be the sentinel node (head of the list). This is created in constructor and deleted in destructor.
- A private sub-class named "node". Define this ABOVE the private node* head.

NODE SUB-CLASS

- Everything in here can be public tbh...
  - Yes, it can be a struct instead.
- Just needs an int to hold a card face, and a node* to point to the next element in the list (or NULL otherwise).
- A constructor to set the pointer to NULL. This will make your life easier... seriously.

SOMETHING LIKE THIS

```
class list {
  private:
      class node {
        public:
            node();
            node* next;
            int element;
      };
  public:
      list();
      ~list();
      bool insert(int);

      friend ostream &operator<<(ostream &, const list &);
  private:
      node *head;
};
```

NODE INSERTION

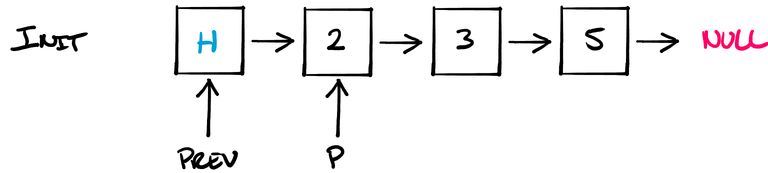
- Insert if possible. Delete if element already exists.
- Return 1 upon an insertion taking place. Return 0 if a deletion took place.

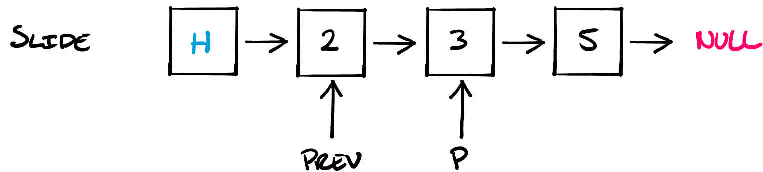
(Ex. Insert Ace, 3, 2, 3, 4.

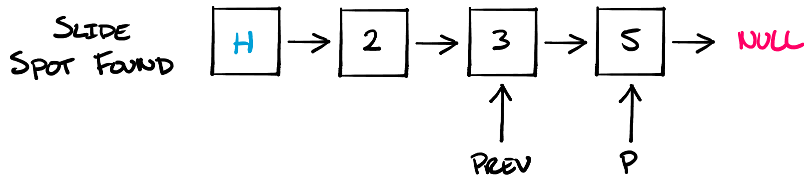
1. [ ]
2. [Ace]
3. [Ace, 3]
4. [Ace, 2, 3]
5. [Ace, 2]
6. [Ace, 2, 4]

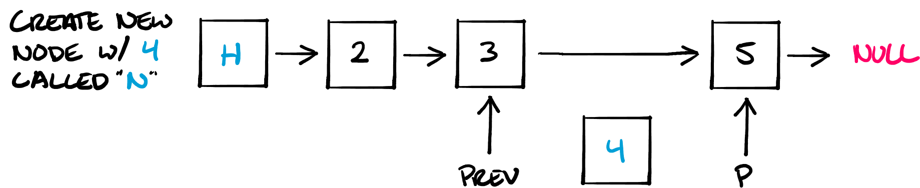
- Notice that it's sorted.
- Procedure:
  - Have two pointers (node*) for traversing: p, prev.
  - prev = head and p = prev→next.
  - As long as p isn't NULL, just loop and update p and prev to go to their next elements.
  - If p→element is more than what we are inserting, insert a new node between prev and p.
  - If p→element is equal to what we are inserting, set prev→next to p→next and obliterate p.
- Use the return value to determine incrementing table[I][] (from Prog2a) or decrement it instead.

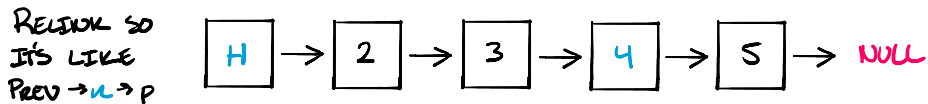
- Visually ... Assume L is a list that has {2, 3, 5}, and we want to insert 4 ...



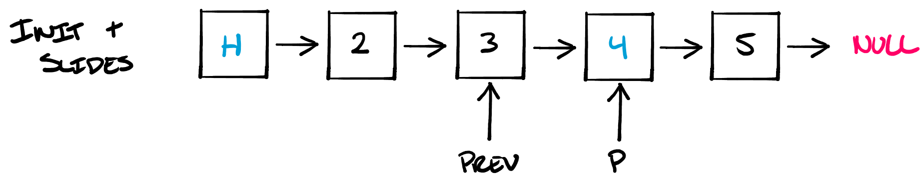


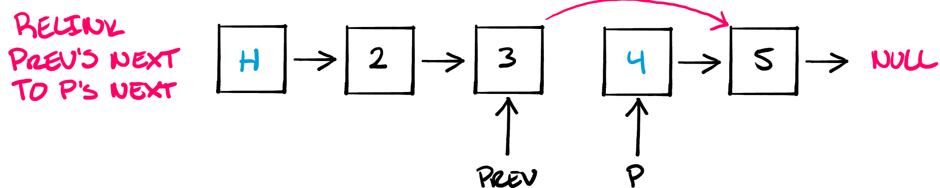


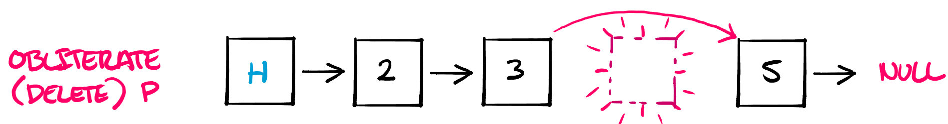




- Furthermore, if we tried to insert another 4...







OPERATOR <<

- Lets you print out a class (in this case...).

- Friend ostream and list so you can use list private members like the node sub-class and the head of the list.

- ```
  ostream &operator<< (ostream &out, const list &L) {
      /* YOUR CODE */
      return out; //why?
  }
  ```

    - Think about cout << "Hello World!" << endl; It's basically a chain of operator<< calls.
      aka. operator<<(operator<<(cout, "Hello World!"), endl);

    - Thus, that inner function must return cout (out) so the outer call will know where to print to.

    - Evaluation:

      operator<<(operator<<(cout, "Hello World!"), endl);

      operator<<(cout, endl);

- To make a node pointer here, you need to type list::node*, as we are outside the list class now.

- Set the list node pointer to the first element in the list (head → next).

- Loop through until the pointer is NULL. Print every node's element to out (NOT cout) and set the pointer to its next one (ex. p = p→next).
  ```