Exam 1

Data Structures
Thursday, October 7, 2021

You must justify all your answers to recieve full credit

1. Consider a code fragment using the bitwise XOR.

```
int a = -3;
int x = ???  // replace this ??? with a number
int b = a^x;
cout << hex << b;
// This outputs the hexadecimal representation of "b": "ffffabcd".
```

   (a) Write the hexadecimal representation of variable `a`, if its decimal value is -3.

   Written as twos complement, -3 is represented as `0xFFFFFFFD`. See `https://bit.ly/3Bng7hE` for details.

   □

   (b) Write the binary representation of the same variable `a`.

   We expand each digit of `0xFFFFFFFD` as 4 bits:

   `1111.1111.1111.1111.1111.1111.1111.1101`

   □

   (c) Write the hexadecimal representation of variable `x`. That is, rewrite line 2 of the code snippet above to make variable `b` equal to `0xffffabcd`.

   We can take $x = a \oplus b$ (XOR of a and b). Indeed $b = a \oplus x = a \oplus (a \oplus b) = (a \oplus a) \oplus b = b$.

```
a = fffffffd = 1111.1111.1111.1111.1111.1111.1111.1101
b = ffffabcd = 1111.1111.1111.1111.1010.1011.1100.1101
a XOR b = 0000.0000.0000.0000.0101.0100.0011.0000
```

   Rewrite the result into hexadecimal notation: `0x00005430`.

   □

2. A doubly linked list has 3 structures of type `Node` (see Figure 1): it has `prev` and `next` pointers of type `Node*` and also a constant `info` field storing constant positive integers. That is, the `info` field does not change over the lifetime of the `Node`.

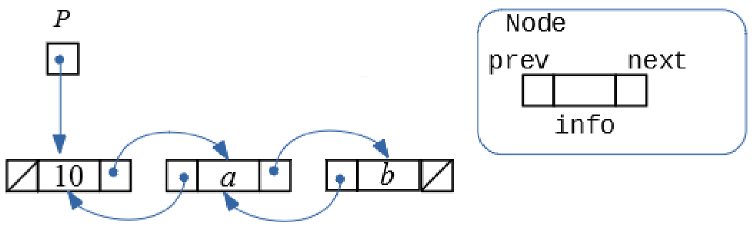


Figure 1: Doubly linked list

Variable `P` is a pointer of type `Node*`, initially `P` points to the first node in this list. The `info` fields in this list have values 10, a and b.

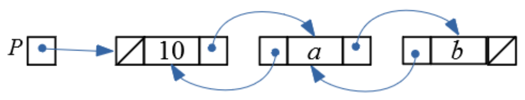
Write code that modifies this doubly linked list in the following way:

- If $a > b$, the pointers in the list change so that the `Node` with `info` field value 10 points to b (that `Node` is now second), and the `Node` with `info` field b points to a (that `Node` is now third).
- If $a \leq b$, then your code should leave the list unchanged.

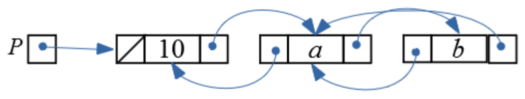
```
int a = P -> next -> info;
int b = P -> next -> next -> info;
if (a > b) {
    P -> next -> next -> next = P -> next;
        P -> next = P -> next -> next;
        P -> next -> next -> next = NULL;

        P -> next -> prev = P;
        P -> next -> next -> prev = P -> next;
}
```

The above code snippet could rearrange the nodes. The first 3 assignments under the **if** statement should come in certain order to avoid information loss. If you want to change this order, you might need to introduce some temporary pointer variables (which is perfectly fine as long as the resulting linked list is not broken). Figure 2 shows step-by-step process of reassigning pointers.

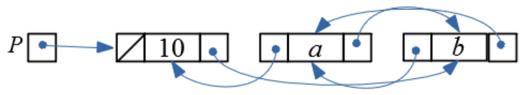
Note: We cannot simply reassign `info` fields, since they are constant and do not change over the lifetime of the nodes. There are some other situations when it is easier to leave the pointers as they are, but swap the contents of the `info` fields. □



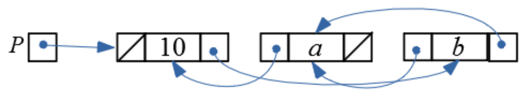
```
P -> next -> next -> next = P -> next;
```



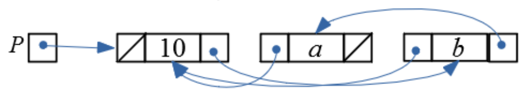
```
P -> next = P -> next -> next;
```



```
P -> next -> next -> next = NULL;
```



```
P -> next -> prev = P;
```



```
P -> next -> next -> prev = P -> next;
```

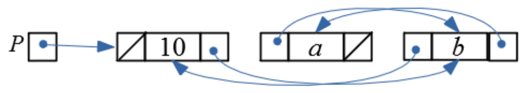


Figure 2: Doubly linked list being rearranged

3. The following C++ program declares a class `Pair`. Pairs are *lexicographically ordered*:

$$(x_1, y_1) < (x_2, y_2) \text{ iff } x_1 < x_2 \lor (x_1 = x_2 \land y_1 < y_2).$$

Complete the code below so that, when compiled and executed, in order:

- a positive integer n is input,
- n pairs from the standard input are input,
- the n pairs are pushed on the stack, skipping pairs which are not lexicographically larger than the current top element of the stack,
- the remaining pairs are output to the standard output as separate lines in their original order.

Use the overloaded operators "`cout << pair`", "`cin >> pair`", "`p1 < p2`" for input, output and comparisons. The only data structure to use is STL stack. If necessary, you may use several stacks.

```
#include <iostream>
#include <stack>
using namespace std;
class Pair { public: int x; int y; };
istream &operator>>(istream  &input, Pair &p ) {
  input >> p.x >> p.y;   return input;
}
ostream &operator<<(ostream &output, const Pair &p ) {
  output << "(" << p.x << "," << p.y << ")";   return output;
}
bool operator<(const Pair &left, const Pair &right) {
  // implement the lexicographic comparison operator.
}
int main() {
  // Input the total number of pairs, then 2*n integers (the pairs).
  // Output those pairs which are in lexicographically increasing order.
}
```

Sample input

```
5
4 17
4 17
5 1000
7 12
7 9
```

Sample output

```
(4,17)
(5,1000)
(7,12)
```

One possible solution is shown below. □

```
#include <iostream>
#include <stack>
using namespace std;
class Pair {
  public: int x; int y;
  Pair() {
    cout << "Default constructor" << endl;
  }
  Pair(const Pair& arg) {
    cout << "Copy constructor on (" << x << "," << y << ")" << endl;
    x = arg.x;
    y = arg.y;
  }
  ~Pair() {
    cout << "Destructor on (" << x << "," << y << ")" << endl;
  }

};
istream &operator>>(istream  &input, Pair &p ) {
  input >> p.x >> p.y;   return input;
}
ostream &operator<<(ostream &output, const Pair &p ) {
  output << "(" << p.x << "," << p.y << ")";   return output;
}
bool operator<(const Pair &left, const Pair &right) {
  return (left.x < right.x) || (left.x == right.x && left.y < right.y);
}
int main() {
    int n;
    cin >> n;
    stack<Pair> myStack;
    for (int i = 0; i < n; i++) {
        Pair p;
        cin >> p;
        if (myStack.empty() || myStack.top() < p) {
            myStack.push(p);
        }
    }
    stack<Pair> otherStack;
    while (!myStack.empty()) {
        otherStack.push(myStack.top());
        myStack.pop();
    }

    while (!otherStack.empty()) {
        cout << otherStack.top() << endl;
        otherStack.pop();
    }
}
```

4. In your code from Question 3, define the default (no argument) constructor, copy constructor and destructor. Describe the order in which these are called when your compiled code is executed with the input below.

```
2
2 3
1 4
```

In your answer you should list the order how the constructors of both types (and also destructor) are invoked for every `Pair` object. Describe why this behavior makes sense and generalize – how many invokations would happen for n pairs in the input.

```
Default constructor
Copy constructor on (8914240,8913088)
Destructor on (2,3)
Default constructor
Destructor on (1,4)
Copy constructor on (8984152,8978256)
Destructor on (2,3)
(2,3)
Destructor on (2,3)
```

A sample order on the calls of constructors/destructors is shown in the above plaintext fragment. The first pair $(2, 3)$ is constructed three times (first, it is read from the input, then added to a stack – which causes a copy constructor, finally it is also copied to another stack to reverse order). The second pair $(1, 4)$ is skipped (so it is only constructed once). For every constructor (either default or copy constructor) there should be one destructor.

In general, if there are n pairs in the input and $k \leq n$ of these pairs are not skipped, then the number of constructor calls would be $3k + (n - k)$ – namely, every non-ignored pair is constructed three times, but every ignored pair is constructed just once.

For your implementation the number of constructor calls may differ, but some principles should still apply (every constructor has a matching destructor, etc.) □

5. Consider the functions $f_1(n)$, $f_2(n)$, $f_3(n)$, $f_4(n)$ below, mapping positive integers $n \geq 5$ to positive real numbers $t > 0$:

$$\begin{aligned} f_1(n) &= (1 + \cos n)\sqrt{2^{7\cdot\log_2(n)}}, \\ f_2(n) &= 13^{\log_2(n)}, \\ f_3(n) &= \frac{1}{n^2} \cdot \binom{n}{5}, \\ f_4(n) &= f_1(n) + f_2(n) + f_3(n). \end{aligned}$$

For each function $f_i(n)$, $i = 1, 2, 3, 4$, find functions

- $g_i(n)$ such that $f_i(n)$ is $O(g_i(n))$,
- $h_i(n)$ such that $f_i(n)$ is $\Omega(h_i(n))$,
- $k_i(n)$ such that $f_i(n)$ is $\Theta(k_i(n))$.

Give justifications for all your answers.

1. $f_1(n)$ is in $O(n^{3.5})$. To see this, rewrite like this:

$$\sqrt{2^{7\cdot\log_2(n)}} = \sqrt{(2^{\log_2(n)})^7} = \sqrt{n^7} = n^{3.5}.$$

Moreover, $f_1(n)$ is in $\Omega(0)$ (in fact, $1+\cos n$ returns to the value 0 infinitely often). So the bottom estimate of how this function grows cannot be any bigger than $O(0)$.
Function $g(n)$ such that $f_1(n)$ is in $\Theta(g(n))$ does not exist (as Big-O and Big-Omega estimates differ).

2. $f_2(n)$ is in $O(n^{\log_2 13})$ this (same as $O(n^{3.7})$). Same answer about Big-Omega and Big-Theta.

3. $f_3(n)$ is in $O(n^3)$. This is because $\binom{n}{5}$ is the 5th degree polynomial:

$$\binom{n}{5} = \frac{n(n-1)(n-2)(n-3)(n-4)}{5!}.$$

If you divide 5th degree polynomial by n^2, it ends up being $O(n^3)$. Same answer applies for Big-Omega and Big-Theta.

4. $f_4(n)$ is in $O(n^{\log_2 13}) \approx O(n^{3.7})$ (as $\log_2 13$ is the largest exponent among all the functions $f_1(n)$, $f_2(n)$, $f_3(n)$, so we can ignore all the smaller exponents, including the fluctuating $f_1(n)$). Same answer for Big-Omega and Big-Theta.

$\square$