Assignment 2

Due: Friday Mar 4th, 11:59 pm

Sorting, searching (60 pts)

1. Build a data structure of 300 elements. Use pointer, vector and linked list, each of 300 integers elements. The elements are integers from 150 to 449. Even numbers are added into the data structures from the start. Odd numbers are added in the reverse order. Meaning, the data structure would look like 150, 152, 154, 156, … , 444, 446, 448, 449, 447, 445, … ,157, 155, 153, 151.
    a. Design a pseudocode for how you will do this exercise using a pointer, vector/array, and a linked list. What kind of a linked list do you plan to use? Singly, Doubly, Circular? Notice that the way to access pointers, arrays and vectors is similar. **Due 18th Feb, 11:59 pm**.
    b. Once you have the list, vector/array and the pointer ready, print the data in the data structures out with one space between the elements, and print each data structure on a separate line.
2. Use the following methods to sort the pointer data (ascending):
    a. Insert an element into its correct position by moving all the others to the right by one. Be sure to correctly identify the algorithm, print the sorted elements.
    b. Go through the whole data structure, comparing the rest of the elements with the element at the current position, and swapping whenever necessary. Be sure to correctly identify the algorithm, print the sorted elements.
3. Use the following methods to sort the vector/array (descending):
    a. Select the maximum element, and place it at the beginning of the remaining vector/array. Be sure to correctly identify the algorithm, print the sorted elements.
    b. Go through the vector/array, swapping consecutive elements whenever necessary. Keep repeating this until the array is sorted. In short, repeatedly swap the adjacent elements if they are in the wrong order. Be sure to correctly identify the algorithm, print the sorted elements.
4. Use the following methods to sort the linked list (ascending):
    a. Insert an element into its correct position by inserting a node between two nodes in a linked list. Be sure to correctly identify the algorithm, print the elements.
    b. Go through the whole data structure, comparing the rest of the elements with the element at the current position, and swapping whenever necessary. Be sure to correctly identify the algorithm, print the sorted elements.
5. Write a pseudocode for 2, 3 and 4. There are sources online which have the pseudocode, and we will check your answers with them for plagiarism. **If using algorithms from a textbook or slides, explain what the lines are doing in your own words.** Please maintain Academic Integrity. **Due 18th Feb, 11:59 pm**.
6. Measure the runtime of your algorithms in 2, 3 and 4. For 2 and 4, report the run time for each algorithm.
    a. Is there a difference between the two timings? Why? This subsection will go in the final PDF submitted.

Example usage:

```
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
      …          /* Some code */
      struct timeval start, end;

      gettimeofday(&start, NULL);

      /* code to measure the run time of, example to measure the run
time of the for loop below
      */
      for (int i = 0; i < iterations; i++)
      {
      }

      gettimeofday(&end, NULL);

      printf("%ld\n", ((end.tv_sec * 1000000 + end.tv_usec)
         - (start.tv_sec * 1000000 + start.tv_usec)));

      return 0;
}
```

Once the pointer, vector and linked list are sorted, use the following methods to search for the element 279 in **each** of them:

7. Linear Search
    a. Pseudocode **Due 18th Feb, 11:59 pm**.
    b. C++ code
8. Binary Search
    a. Pseudocode **Due 18th Feb, 11:59 pm**.
    b. C++ code

Print the address of the pointer and the vector/array indices, followed by the index. For Linked List, print the address of the node.

Stacks and Queues **(40 pts)**

9. Each arithmetic operation is denoted by the following numbers (op_num):

op_num	Operation	opnd_num	operand
0 (0000b)	nop	0 (0000b)	none
1 (0001b)	+	1 (0001b)	a
2 (0010b)	-	2 (0010b)	b
3 (0011b)	*	3 (0011b)	c
4 (0100b)	/	4 (0100b)	d
5 (0101b)	pop	5 (0101b)	e
6 (0110b)	push	6 (0110b)	acc

The following series of operations takes place, which is explained in detail in the text that follows:

Seq: 601	push a
Seq: 602	push b
Seq: 604	push d
Seq: 504	pop d
Seq: 502	pop b
Seq: 442	/ (acc=d/b)
Seq: 603	push c
Seq: 536	pop (c=acc)
Seq: 605	push e
Seq: 603	push c
Seq: 503	pop c
Seq: 505	pop e
Seq: 335	* (acc=c*e)
~~Seq: 601~~	~~push a~~
Seq: 516	pop (a=acc)

The input to your code will be sequences like the one above. The sequences can be in an array, such as **opcodes[] = {0x601, 0x602, 0x604, 0x504, 0x502, 0x442, 0x603, 0x536, 0x605, 0x603, 0x503, 0x505, 0x335, 0x601, 0x516}.**
These are the 2 initial sample states you can use to test with the opcode sequence provided in the question:

1. a = 55, b = 20, c = 276, d = 100, e = 203, acc = 0
2. a = 101, b = 225, c = 3, d = 14625, e = 113, acc = 0

This is one more simple sequence:

opcodes[]={0x602, 0x604, 0x504, 0x502, 0x442, 0x516, 0x605, 0x556, 0x605, 0x601, 0x501, 0x505, 0x115}

The list of operands can be as simple as: **unsigned long long int operands[7];**

You can set the values that you want to be able to test your code. Make sure the arithmetic is correct. Create some test cases of your own to test the correctness of your implementation. We will provide some of our own midway through the assignment.
Description: The most significant hexadecimal digit is for the operation. The second digit is the first operand variable/destination variable. The third digit (least significant hexadecimal digit) is the second operand variable/source variable. The explanation of the sequences is as follows:

1. An operation like 60X is a *push* operation, and it will have a middle digit as 0. X is the operand number. It pushes the operand onto a stack. If implemented as an array, you can push the index of the operands array onto the stack.
2. An operation like 5XX is a *pop* operation that can be interpreted in two ways:
    a. A pop operation with a 0 or 6 in the middle returns only the operand denoted by the rightmost digit. When popping the operand, compare the operand denoted by the rightmost digit with the operand at the top of the stack (**Hint:** top()).:
        i. If they are the same, pop the operand, and **push it into a queue.** Print **popS, pushQ**
        ii. If not, print the error "**Stack Wrong operand requested**", and use **exit()** to terminate the program**.**
    b. A pop operation with a middle digit other than 0 and 6 puts the value of *acc* into the operand denoted by the middle digit. **The rightmost digit won't matter**. You should first pop the corresponding (middle digit denoted) operand from a queue (if it is at the start of the queue), and then put the value of *acc* into it. **Important:** If the operand is not at the start of the queue (**Hint:** front() to check the head of the queue), check whether it is at the top of the stack (scenario highlighted in red). Use *top()* for it.
        i. If yes, pop the operand and put the value of *acc* into it directly. Print the following sequence: **popS, BypassQ**
        ii. If not, meaning the operand has not yet been pushed to the stack either (look at the **strikethrough** case), then push the operand denoted by the middle digit onto the stack, immediately pop it, and that operand then has the value held in *acc*. In this scenario, print the following sequence: **pushS, popS, BypassQ**
        iii. **We will be testing** if you account for this fail-safe (look at the instruction that has a strikethrough). The code should run with or without it. The *push* operation in red is where we have pushed the operand in a similar situation. Code should run for both scenarios.
3. The arithmetic operation pops the first two operands in the queue, verifies whether they match the middle and the rightmost digits, respectively, and puts the result of the

arithmetic operation in the *acc* operand (number 6). In case any one of the checks fail, throw the error message: **Arithmetic Wrong operand requested,** and use **exit()** to terminate the program.

When implementing a stack, you can **only** pop the last operand. In order to pop the operand, you remove it from the data structure, and delete the node representing it. **Use linked lists**. You will need a list of pushed operands and a list of popped operands. **The popped operands from the stack will act like a queue**. Consider the sequence from above: 504, 502, 442. As you push the operands *d* and *b* (602, 604), *b* is at the bottom of the stack, and *d* at the top. Then 504 pops *d*, and pushes it into a queue. 502, which is now the top operand, is then pushed into the queue behind *d*. The arithmetic operation pops *d* from the queue, followed by *b*, and divides the first operand by the second, i.e. *d/b*. Similarly for 503, 505 and 335.

516 checks whether *a* is present in front of the queue. If not, it checks whether it is on top of the stack. Since it does not find it there, it pushes *a* onto the stack, then pops *a* from the stack, and puts the value of *acc* into it. In 536, *c* is present on top of the stack after being pushed by 603. In this case, check the front of the queue for *c*. When not found, check whether it is on top of the stack. Since it is, pop *c* and put the value of *acc* into it.

Hint: Before calling *pop()* for queue or stack, make sure to check whether the first operand is correct using *front()* or *top()*.

Print the final state of all your operands. Further details are given in the **Instructions**.

Use C++ comments to explain the important parts of your algorithm in the code. All reportables must be in a document (txt, pdf, doc).

Instructions:

- Each question should be done in a separate file. The naming convention is <question_no_subdivision>.cpp i.e., 1b.cpp, 2a.cpp, 2b.cpp, 3a.cpp etc.
- All pseudo codes can be written in the same file called pseudo_code.pdf. Make sure to include the correct question number and subdivision for the pseudo codes. **Follow the instructions in the Course Syllabus regarding pseudocode submission for more details.** Add the name of the peer reviewer in the pdf file. For example:
  Name: ABC
  Peer Reviewer: XYZ
  Reminder - This is due 18th Feb, 11:59 pm.
  You have to submit the final pseudocode in the final PDF file, with comments explaining the pseudocode wherever necessary.
- For the search questions (7 and 8), print address and index of the element searched for the data structures (pointer, vector/array) in the same order. Then, print the address of the linked list node containing the data.
- Refer to the example outputs given below for each question.

Outputs:

1.

b). The output should contain 3 lines, in the order of pointer, array/vector and linked list.

Example output :

```
150 152 154 156 …… 155 153 151
150 152 154 156 …… 155 153 151
150 152 154 156 …… 155 153 151
```

2.

a). Example output:

```
150 151 152 153 …… 447 448 449
```

b). Example output:

```
150 151 152 153 …… 447 448 449
```

3.

a). Example output:

```
150 151 152 153 …… 447 448 449
```

b). Example output:

```
150 151 152 153 …… 447 448 449
```

4.

a). Example output:

```
150 151 152 153 …… 447 448 449
```

b). Example output:

```
150 151 152 153 …… 447 448 449
```

6. The time taken to run 2a, 2b, 4a and 4b should be reported in this order. Print each time in a new line. **In the final pdf that you submit, answer if you see any differences in timings, and why. This is an open ended question.**

Example:

```
13
25
45
12
```

Note: These numbers are just examples and not representative of the actual time taken to run.

7.

b). Example output:

```
0x6df1234 115
0x6df7654 95
0x6df1987
```

Note: Linked list has no index, just address

8.
b). Example output:

```
0x6df1234 115
0x6df7654 95
0x6df1987
```

Note: Linked list has no index, just address

9. Follow the printing instructions in **Description**. Print the sentences just as it is, whether it is pertaining to the errors or sequence of operations in the corresponding edge cases. Print the final state of operands in the following way (You don't need to print the alphabet):
<operand_name> <value> -> You can print it as, for example: `cout << "Operand" << i << " " << operand[i] << endl;`

Example output:
`Operand0 0 Operand1 10 Operand2 15 Operand3 20 Operand4 0 Operand5 99 Operand6 1000`

Important Tips:

1. Implement a stack with a linked list. Test the *pushS(), popS(), top(), isEmptyS()* to see whether the stack is implemented correctly.
2. Implement a queue with a linked list. Test the *pushQ(), popQ, front(), isEmptyQ()* to see whether the queue is implemented correctly.
   (You can call your stack and queue ADT whatever you want, this is just an example.)
3. Implement an arbiter function which checks for which opcode is the next in line (from the opcode array). Send it one opcode at a time from your main function. The arbiter function could have if statements or switch statements. That design choice is up to you.
4. Whenever making comparisons to check the individual digits in the opcode, I would advise to use **shift** operator and the **bitwise AND (&)** operator, to check every digit. For example, if I want to check the Most Significant digit (leftmost digit), I would use something like `if ((num >> 8) & 0xf == 0x6)`. This shifts a number, like 0x602 to the right by 8 bits, making it 0x6, does a bitwise AND with 0xf (0b0110 & 0b1111), and checks whether it is equal to 0x6 (0b0110).

Here are some links to learn more about the rules of bitwise AND, shift operations and binary/hexadecimal representations:

- https://www.programiz.com/c-programming/bitwise-operators
- https://learn.sparkfun.com/tutorials/binary/counting-and-converting
- https://www.sciencedirect.com/topics/engineering/hexadecimal
- https://docs.microsoft.com/en-us/cpp/cpp/left-shift-and-right-shift-operators-input-and-output?view=msvc-170