CS2710 - Programming and Data Structures Lab

Lab 8 (graded)

Sept 18, 2024

Instructions

- You are expected to solve ALL the problems in the lab, using the local computer, C++ language, and g++ compiler. (Bonus problems can fetch 5% capped bonus, and Optional problems are just for fun and won't be graded.)
- You should submit your code to the course **moodle** on time, i.e., on/before 4.45pm (so that TAs can subsequently grade your submissions for graded lab assignments using the private test cases).
- You must strictly adhere to the following naming convention for your .cpp files and the single .zip file submission. For example, for a Lab Session 8 consisting of 2 questions, a student with roll number CS23B000 should
  - Name their .cpp files as **CS23B000_LAB8_Q1.cpp** and **CS23B000_LAB8_Q2.cpp**
  - Put both these .cpp files in a directory named **CS23B000_LAB8**
  - Zip this directory into a file named **CS23B000_LAB8.zip**
  - Submit only this single .zip file to moodle.
- If you need assistance, ask your TA, not your classmate.
- Internet access and mobile devices are prohibited in the lab.
- Check course Moodle for the public test cases and the evaluation script, which you can use to test your programs.
- Solve each problem in this lab session using *queue or binary trees or stack* as the primary data structure (do not use any other data structure; string data type is fine - please clarify with the TAs if you have any questions about this).

---

1. [DEQUE IMPLEMENTATION USING CIRCULAR ARRAY] A deque (double-ended queue) extends a queue by supporting insertion (aka addition) and removal at both ends of the queue, and enjoys applications in certain sliding window and job scheduling problems. In this assignment, you will use a circular array/vector to implement a Deque class and its operations below:

   - **push_front(x)**: Add element `x` to the front of the deque. If the deque is full, print `"Deque is full"` and do not perform the insertion.
   - **push_back(x)**: Add element `x` to the back of the deque. If the deque is full, print `"Deque is full"` and do not perform the insertion.
   - **pop_front()**: Remove and print the front element from the deque. If the deque is empty, print `"Deque is empty"`
   - **pop_back()**: Remove and print the back element from the deque. If the deque is empty, print `"Deque is empty"`
   - **isEmpty()**: Check if the deque is empty and print `True` if it is, otherwise `False`.

   Feel free to extend the circular array implementation of queue seen in lecture to implement this Deque class.

   Input Format:

   - The first input is an integer `N` representing the maximum size of the deque.
   - The second input is an integer `M` representing the number of operations.
   - Next, M lines where each line represents an operation in the following format:
     - `push_front x`: Add element `x` to the front of the deque.
     - `push_back x`: Add element `x` to the back of the deque.
     - `pop_front`: Remove and print the front element of the deque
     - `pop_back`: Remove and print the back element of the deque
     - `isEmpty`: Print `True` if the deque is empty, otherwise `False`.

     Warning messages as indicated above are printed when the deque is full or empty.

   Output Format:

   - For each operation that produces output (`pop_front`, `pop_back`, `isEmpty`), print the result.

   Constraints:

   - Deque size (`N`) will be a positive integer ($1 \leq N \leq 1000$).
   - The operations `push_front`, `push_back`, `pop_front`, and `pop_back` should be performed in **O(1)** time.
   - The deque should operate using a circular array structure.

   Examples:
   Input1:
   4
   7
   push_back 10
   push_front 20
   push_back 30
   push_front 40

pop_front
pop_back
isEmpty

Output1:
40
30
False

Explanation:
`push_back 10`: Adds 10 to the back of the deque. Deque: [10]
`push_front 20`: Adds 20 to the front. Deque: [20, 10]
`push_back 30`: Adds 30 to the back. Deque: [20, 10, 30]
`push_front 40`: Adds 40 to the front. Deque: [40, 20, 10, 30]
`pop_front`: Removes 40 from the front. Deque after removal: [20, 10, 30]. Output: `40`
`pop_back`: Removes 30 from the back. Deque after removal: [20, 10]. Output: `30`
`isEmpty`: Checks if the deque is empty. It's not, so output: `False`

Input2:
3
5
push_back 5
push_back 6
push_back 7
push_back 8
pop_front

Output2:
Deque is full
5

Explanation:
`push_back 5`: Adds 5 to the back. Deque: [5]
`push_back 6`: Adds 6 to the back. Deque: [5, 6]
`push_back 7`: Adds 7 to the back. Deque: [5, 6, 7]
`push_back 8`: Attempt to add 8 fails since the deque is full. Output: `"Deque is full"`
`pop_front`: Removes 5 from the front. Deque after removal: [6, 7]. Output: `5`

2. [UNIX DIRECTORY-RELATED COMMANDS] Using std::filesystem, your friend was able to write the code below to list the contents of a directory. Use this code as a starting point to implement two more functions:

   - `list_all(dirname)`: to list the contents of not only the directory `dirname`, but also its subdirectory, subsubdirectory, etc., with proper indentation (two spaces for subdirectory, four for subsubdirectory, etc.). (e.g., `list_all(".")`).
   - `find_suff(dirname, suffix)`: to print filename (with full path) of all files in `dirname` whose suffix matches the provided suffix (e.g., `findsuff("/home/test", ".cpp")`).

   Do not use any other functions in std::filesystem or the UNIX shell.

```
#include <iostream>
#include <filesystem>
#include <cassert>
#include <string>
using namespace std;
using namespace std::filesystem;

void list_dir_contents(string dirname)
{
  assert(exists(dirname));
  assert(is_directory(dirname));
  cout << dirname << " [dir]" << endl;

  for (const auto& curEntry : directory_iterator(dirname))
  {
    const string fullname = curEntry.path().string();
    const string filename = curEntry.path().filename().string();
    if (curEntry.is_directory()) {
      cout << "  " << filename << " [dir]"<<endl;
    }
    else if (curEntry.is_regular_file()) {
      cout << "  " << filename << endl;
    }
    else {
      cout << "  " << filename << " [unknown_type]"<<endl;
    }
  }
  return;
}

int main()
{
  string dirname;
  cout << "Enter directory name:";  cin >> dirname;
  list_dir_contents(dirname);
  return 0;
}
```

   Please find below an example output for these two functions. Since output depends on the current folder, TAs will manually run/verify your code (you/TAs can also manually check your code's output against the similar UNIX commands: `tree <dirname>` and `find <dirname> | grep ".cpp$"`).

Input Format:
dirname (directory name whose contents are to be printed - string)
suffix (filenames with this suffix are printed - string)

Output Format:
Output of `list_all` function, followed by a newline, and then the output of `find_suff` function as discussed above, and illustrated via examples below. If there is no file with matching suffix, then `find_suff` prints nothing.

Examples:

Input1:
/home/pds/test/stack .cpp

Output1:

```
/home/pds/test/stack [dir]
  stack_impl_typedef.cpp
  infixfp2postfix_postfixeval_etc.cpp
  stack_usage_std.cpp
  test.c
  prev [dir]
    postfixeval.cpp
    postfixeval.hpp

/home/pds/test/stack/stack_impl_typedef.cpp
/home/pds/test/stack/infixfp2postfix_postfixeval_etc.cpp
/home/pds/test/stack/stack_usage_std.cpp
/home/pds/test/stack/prev/postfixeval.cpp
```

3. [BONUS: EXPRESSION TREE] Given a postfix expression, your task is to construct an expression tree from it. Once the tree is built, output both the infix and prefix expressions by performing inorder and preorder traversals, respectively.

   Input Format:

   - A single postfix expression as a string where operands are single-letter variables or single-digit integers, and operators are +, -, *.

   Output Format: For each test case, print:

   - The infix expression obtained through inorder traversal of the expression tree. The infix expression should be parenthesized.
   - The prefix expression obtained through preorder traversal of the expression tree.

   Constraints:

   - $3 \leq$ length of expression ≤ 40
   - All the operators used in expression are binary operators(need 2 operands).

   Examples:

   Input1:
   ab+ef*g*-

   Output1:
   ((a+b)-((e*f)*g))
   - +ab**efg

   Explanation:
   Postfix Expression: ab+ef*g*-
   Construct Expression Tree: Perform inorder traversal to get the infix expression: ((a + b) - ((e * f) * g)). Perform preorder traversal to get the prefix expression: - + a b * * e f g.

   Input2:
   ab+cd+*

   Output2:
   ((a+b)*(c+d))
   *+ab+cd

   Explanation:
   Postfix Expression: ab+cd+*
   Construct Expression Tree: Perform inorder traversal to get the infix expression: ((a + b) * (c + d)). Perform preorder traversal to get the prefix expression: * + a b + c d.