Assignment 2: Postfix Calculator

● Graded

Student

Arushi Yadav

Total Points

63 / 100 pts

Autograder Score

63.0 / 100.0

Failed Tests

1.2) test_2 (test_simple.PostfixCalculator) (0/2.25)
1.3) test_3 (test_simple.PostfixCalculator) (0/2.25)
1.4) test_4 (test_simple.PostfixCalculator) (0/2.25)
1.5) test_5 (test_simple.PostfixCalculator) (0/2.25)
2.5) test_25 (test_simple.PostfixCalculator) (0/24)
3.5) test_35 (test_simple.PostfixCalculator) (0/4)
4.1) test_41 (test_simple.PostfixCalculator) (0/0)

Passed Tests

Check submitted files (0/0)
1.1) test_1 (test_simple.PostfixCalculator) (2/2)
1.6) test_6 (test_simple.PostfixCalculator) (3/3)
1.7) test_7 (test_simple.PostfixCalculator) (3/3)
1.8) test_8 (test_simple.PostfixCalculator) (1/1)
1.9) test_9 (test_simple.PostfixCalculator) (1/1)
1.10) test_10 (test_simple.PostfixCalculator) (1/1)
2.1) test_21 (test_simple.PostfixCalculator) (4/4)
2.2) test_22 (test_simple.PostfixCalculator) (4/4)
2.3) test_23 (test_simple.PostfixCalculator) (4/4)
2.4) test_24 (test_simple.PostfixCalculator) (4/4)
3.1) test_31 (test_simple.PostfixCalculator) (4/4)
3.2) test_32 (test_simple.PostfixCalculator) (4/4)
3.3) test_33 (test_simple.PostfixCalculator) (4/4)
3.4) test_34 (test_simple.PostfixCalculator) (4/4)
3.6) test_36 (test_simple.PostfixCalculator) (16/16)
3.7) test_37 (test_simple.PostfixCalculator) (4/4)

Autograder Results

Check submitted files (0/0)

All required files submitted!

1.1) test_1 (test_simple.PostfixCalculator) (2/2)

1.2) test_2 (test_simple.PostfixCalculator) (0/2.25)

Test Failed: 'Testcase failed division instruction 14\nFailed\n' != 'Passed\n'
+ Passed
- Testcase failed division instruction 14
- Failed
 :

Testcase failed division instruction 14
Failed

1.3) test_3 (test_simple.PostfixCalculator) (0/2.25)

Test Failed: 'Testcase failed division instruction 14\nFailed\n' != 'Passed\n'
+ Passed
- Testcase failed division instruction 14
- Failed
 :

Testcase failed division instruction 14
Failed

1.4) test_4 (test_simple.PostfixCalculator) (0/2.25)

Test Failed: 'Testcase failed division instruction 14\nFailed\n' != 'Passed\n'
+ Passed
- Testcase failed division instruction 14
- Failed
 :

Testcase failed division instruction 14
Failed

1.5) test_5 (test_simple.PostfixCalculator) (0/2.25)

Test Failed: 'Testcase failed division instruction 14\nFailed\n' != 'Passed\n'
+ Passed
- Testcase failed division instruction 14
- Failed
 :

Testcase failed division instruction 14
Failed

1.6) test_6 (test_simple.PostfixCalculator) (3/3)

1.7) test_7 (test_simple.PostfixCalculator) (3/3)

1.8) test_8 (test_simple.PostfixCalculator) (1/1)

1.9) test_9 (test_simple.PostfixCalculator) (1/1)

1.10) test_10 (test_simple.PostfixCalculator) (1/1)

2.1) test_21 (test_simple.PostfixCalculator) (4/4)

2.2) test_22 (test_simple.PostfixCalculator) (4/4)

2.3) test_23 (test_simple.PostfixCalculator) (4/4)

2.4) test_24 (test_simple.PostfixCalculator) (4/4)

2.5) test_25 (test_simple.PostfixCalculator) (0/24)

Test Failed: '' != 'Passed\n'
+ Passed
 :
---
terminate called after throwing an instance of 'std::runtime_error'
  what(): Not Enough Arguments

---
terminate called after throwing an instance of 'std::runtime_error'
  what(): Not Enough Arguments

Stress Tests Failed

3.1) test_31 (test_simple.PostfixCalculator) (4/4)

3.2) test_32 (test_simple.PostfixCalculator) (4/4)

3.3) test_33 (test_simple.PostfixCalculator) (4/4)

3.4) test_34 (test_simple.PostfixCalculator) (4/4)

3.5) test_35 (test_simple.PostfixCalculator) (0/4)

Test Failed: 'First node is not a sentinel node\nFailed\nDLL invariants violated.\n' != 'Passed\n'
+ Passed
- First node is not a sentinel node
- Failed
- DLL invariants violated.
 :

First node is not a sentinel node
Failed
DLL invariants violated.

3.6) test_36 (test_simple.PostfixCalculator) (16/16)

3.7) test_37 (test_simple.PostfixCalculator) (4/4)

4.1) test_41 (test_simple.PostfixCalculator) (0/0)

Test Failed: 1 != 0 : Bonus Submitted

Submitted Files

list.cpp

Download

```cpp
// #pragma once


#include "node.h"
#include "list.h"

/* PART B */
/* Stacks using Linked Lists */

/*
Linked Lists with Sentinels
[X]<->[7]<->[3]<->[2]<->[X]
The head and tails are dummy elements ([X]) that do not have valid values.
These are called sentinel elements.
*/


   List::List(){
      size=0;
      sentinel_head = new Node(true);
      sentinel_tail = new Node(true);
      sentinel_head->next = sentinel_tail;
      sentinel_tail->prev = sentinel_head;
   }

    List::~List(){
      while (sentinel_head->next != sentinel_tail) {
         Node* temp = sentinel_head->next;
         sentinel_head->next = temp->next;
         delete temp;
      }


      delete sentinel_head;
      delete sentinel_tail;
   }

   // Insert an element at the tail of the linked list
   void  List::insert(int v){
      Node* node1= new Node(v,sentinel_tail, sentinel_tail ->prev);
      sentinel_tail->prev->next= node1;
      sentinel_tail->prev= node1;
      size++;

   }

   // Delete the tail of the linked list and return the value
   // You need to delete the valid tail element, not the sentinel
   int  List::delete_tail(){
```

```
50
51      int val= sentinel_tail->prev->get_value();
52      Node* temp = sentinel_tail->prev;
53      sentinel_tail->prev= temp->prev;
54      temp->prev->next= sentinel_tail;
55
56      delete temp;
57      size--;
58      return val;
59
60
61
62  }
63
64  // Return the size of the linked list
65  // Do not count the sentinel elements
66  int List:: get_size(){
67      return size;
68  }
69
70  // Return a pointer to the sentinel head of the linked list
71  Node*  List::get_head(){
72      return sentinel_head;
73  }
74
75
76
```

node.cpp

Download

```cpp
// #pragma once
#include<iostream>
#include"node.h"
using namespace std;

/* PART B */
/* Stacks using Linked Lists */

// class Node {
// private:
//     int value;
//     bool is_sentinel;

// public:
//     Node* next;
//     Node* prev;

    // Use to construct a sentinel node (see list.h)
    Node::Node(bool sentinel){
        this->value=-1;
        this->next= nullptr;
        this->prev= nullptr;
    }

    // Use to construct a regular node
    Node::Node(int v, Node* nxt, Node* prv){
        this->value= v;
        this->next=nxt;
        this->prev= prv;

    }




    // Return whether a node is a sentinel node
    // Use it to check if we are at the ends of a list
    bool Node::is_sentinel_node(){
        return is_sentinel;

    }

    // Return the value of a node
    int Node::get_value(){
        return this->value;

    }

```

stack_a.cpp

Download

```cpp
#include<iostream>
#include<stdexcept>
using namespace std;
#include "stack_a.h"


    Stack_A::Stack_A(){
        size= 0;
    }

    void  Stack_A:: push(int data){


    if((size)==1024){
        throw runtime_error("Stack Full");
    }

    stk[size++]=data;
    }

    int  Stack_A:: pop(){
        if(size>0){



            return stk[--size];

        }
        else{
            throw runtime_error("Empty Stack");
            // return top;
        }
    }

    int  Stack_A::get_element_from_top(int idx){
        if(idx>=size && idx<0){
            throw runtime_error("Index out of range");
        }

        return stk[size-idx-1];
    }

    int Stack_A:: get_element_from_bottom(int idx){
        if(idx>=size && idx<0){
            throw runtime_error("Index out of range");
        }
        return stk[idx];
    }

```

```
50
51  void  Stack_A::print_stack(bool top_or_bottom){
52      if (top_or_bottom){
53          for (int i= (size-1); i>=0;i--){
54              cout<<stk[i]<<endl;
55          }
56
57      }
58      else{
59          for(int i=0;i<size;i++){
60              cout<<stk[i]<<endl;
61          }
62      }
63  }
64
65  int  Stack_A::add(){
66      if(size<2){
67          throw runtime_error("Not Enough Arguments");
68
69      }
70      int op2=pop();
71
72      int op1=pop();
73
74      int result= op1+op2;
75
76      push(result);
77      return result;
78
79
80  }
81
82  int  Stack_A::subtract(){
83      if(size<2){
84          throw runtime_error("Not Enough Arguments");
85
86      }
87      int op2= pop();
88
89      int op1= pop();
90
91      int result= op1-op2;
92
93      push(result);
94
95      return result;
96  }
97
98  int  Stack_A::multiply(){
99      if(size<2){
100         throw runtime_error("Not Enough Arguments");
101
```

```
102         }
103         int op2= pop();
104         int op1= pop();
105         int result= op1 * op2;
106 
107         push(result);
108         return result;
109     }
110 
111     // int floorf(double& x){
112     // if (x>0){
113     //     return x;
114 
115     // }
116     // else {
117     //     return (x-1);
118     // }
119 
120 
121 
122     int  Stack_A::divide(){
123         if(size<2){
124             throw runtime_error("Not Enough Arguments");
125 
126         }
127         int op2= pop();
128         int op1= pop();
129 
130 
131         if (op2 !=0){
132             int c= op1/op2;
133             if ( op1>0 && op2<0 || op1<0 && op1>0){
134                 c=c-1;
135             }
136 
137             int result= c;
138             // result= floorf(result);
139 
140 
141             push(result);
142             return result;
143 
144         }
145         else{
146             throw runtime_error("Divide by Zero Error");
147         }
148 
149 
150     }
151 
152     int*  Stack_A::get_stack(){
153         int* arr= stk;
```

```
154         return arr;
155     }
156 
157     int  Stack_A::get_size(){
158         return size;
159     }
160 
161 
162 
163 
164 
165 
166 // int main(){
167 //     Stack_A stack;
168 //     int n;
169 //     cin>>n;
170 
171 
172 //     while(n--){
173 
174 //         string command;
175 //         cin>>command;
176 
177 //         try {
178 //             if (command == "push" ){
179 
180 //                 int data;
181 //                 cin>>data;
182 //                 stack.push(data);
183 //             }
184 //             else if(command=="pop"){
185 //                 cout<<stack.pop();
186 //             }
187 
188 //             else if (command =="add"){
189 //                 cout<<stack.add();
190 //             }
191 //             else if (command =="subtract"){
192 //                 cout<< stack.subtract();
193 //             }
194 //             else if (command =="multiply"){
195 //                 cout<<stack.multiply();
196 //             }
197 //             else if (command =="divide"){
198 //                 cout<<stack.divide();
199 //             }
200 //             else if(command=="get_size"){
201 //                 cout<<stack.get_size();
202 //             }
203 
204 //             else if(command == "get_element_from_top"){
205 //                 int idx;
```

```
206 //             cin>>idx;
207 //             cout<<stack.get_element_from_top(idx);
208 //         }
209 
210 //         else if(command=="get_element_from_bottom"){
211 //             int idx;
212 //             cin>>idx;
213 //             cout<<stack.get_element_from_bottom(idx);
214 //         }
215 
216 //         else if(command=="get_stack"){
217 //             cout<<stack.get_stack();
218 
219 //         }
220 
221 
222 //         else if (command == "print_stack") {
223 //             string direction;
224 //             cin >> direction;
225 
226 //             if (direction == "top") {
227 //                 stack.print_stack(true);
228 //             } else if (direction == "bottom") {
229 //                 stack.print_stack(false);
230 //             }
231 // }
232 
233 
234 
235 
236 //     }
237 //     catch( runtime_error e){
238 //         cout<<e.what()<<endl;
239 //     }
240 //  }
241 
242 
243 
244 //  return 0;
245 // }
```

stack_b.cpp

Download

```cpp
#include<iostream>
#include<stdexcept>
#include "stack_b.h"

using namespace std;



    // Constructor
    Stack_B::Stack_B()
        {
        this->size=0;
        this->capacity= 1024;


        try{
            stk = new int[capacity];
        }
        catch(bad_alloc&){
            throw runtime_error("Out of Memory");
        }
    }

    // Destructor
    Stack_B::~Stack_B(){
        delete[] stk;
    }



    void Stack_B::push(int data){
      if (size == capacity) {

            int newCapacity = capacity * 2;
            int* newStk;

            try {
                newStk = new int[newCapacity];
            }
            catch(bad_alloc&){
               throw runtime_error("Out of Memory");
            }
            for (int i = 0; i < size; i++) {
                newStk[i] = stk[i];
            }
            delete[] stk;
            stk = newStk;
            capacity = newCapacity;
        }
```

```
50
51
52              stk[size] = data;
53             size++;
54
55
56      }
57
58      int Stack_B::pop(){
59          if (size<=0){
60              throw runtime_error("Empty Stack");
61          }
62
63          else{
64              return stk[--size];
65          }
66      }
67
68
69
70      int Stack_B::get_element_from_top(int idx){
71        if (idx >= size || idx < 0) {
72         throw std::runtime_error("Index out of range");
73        }
74        return stk[size - idx - 1];
75      }
76
77      int Stack_B::get_element_from_bottom(int idx){
78          if (idx >= size || idx < 0) {
79          throw std::runtime_error("Index out of range");
80          }
81          return stk[idx];
82      }
83
84
85      void Stack_B::print_stack(bool top_or_bottom){
86          if (top_or_bottom==1){
87              for (int i= size-1; i>=0;i--){
88                  cout<<stk[i]<<endl;
89              }
90
91          }
92          else{
93              for(int i=0;i<size;i++){
94                  cout<<stk[i]<<endl;
95              }
96          }
97      }
98
99      int Stack_B::add(){
100         if(size<2){
101             throw runtime_error("Not Enough Arguments");
```

```
102
103         }
104         int op2=pop();
105
106         int op1=pop();
107
108         int result= op1+op2;
109
110         push(result);
111         return result;
112
113
114     }
115
116     int Stack_B::subtract(){
117         if(size<2){
118             throw runtime_error("Not Enough Arguments");
119
120         }
121         int op2= pop();
122
123         int op1= pop();
124
125         int result= op1-op2;
126
127         push(result);
128
129         return result;
130     }
131
132     int Stack_B::multiply(){
133         if(size<2){
134             throw runtime_error("Not Enough Arguments");
135
136         }
137         int op2= pop();
138         int op1= pop();
139         int result= op1 * op2;
140
141         push(result);
142         return result;
143     }
144
145
146     int  Stack_B::divide(){
147         if(size<2){
148             throw runtime_error("Not Enough Arguments");
149
150         }
151         int op2= pop();
152         int op1= pop();
153
```

```
154
155         if (op2 !=0){
156             int c= op1/op2;
157             if ( (op1>0 && op2<0) || (op1<0 && op2>0)){
158                 c=c-1;
159             }
160
161             // int  result= c;
162             // result= floorf(result);
163
164
165             push(c);
166             return c;
167
168         }
169         else{
170             throw runtime_error("Divide by Zero Error");
171         }
172     }
173
174     int* Stack_B::get_stack(){
175
176         return stk;
177     }
178
179     int Stack_B::get_size(){
180         return size;
181     }
182
183
184
185
186
187
```

stack_c.cpp

Download

```cpp
#include<iostream>
#include "stack_c.h"
using namespace std;


    // Constructor
    Stack_C::Stack_C(){
        stk = new List();
    }

    // Destructor
    Stack_C::~Stack_C(){
        delete stk;

    }

    void Stack_C::push(int data){
        stk->insert(data);
    }

    int Stack_C::pop(){
        if (stk->get_size()==0){
            throw std::runtime_error ("Empty Stack");
        }
        else{
          return stk-> delete_tail();
        }
    }

    int Stack_C::get_element_from_bottom(int idx){


    // if (idx >= 0 && idx < stk->get_size()) {
    //     Node* node = stk->get_head()->next;
    //     int count = 0;

    //     while (node != nullptr) {
    //         if (count == stk->get_size() - idx - 1) {
    //             return node->get_value();
    //         }
    //         node = node->next;
    //         count++;
    //     }

    //     throw std::runtime_error("Index out of range");
    // } else {
    //     throw std::runtime_error("Index out of range");
    // }
    if (idx<0 || idx>stk->get_size()){
```

```
50          throw runtime_error("Index out of range");
51
52      }
53      Node* node = stk->get_head()->next;
54      for(int i=0; i<idx;i++){
55          node= node->next;
56      }
57      int a= node->get_value();
58      return a;
59
60
61      }
62
63
64
65      int Stack_C::get_element_from_top(int idx){
66
67
68
69          if(idx >= 0 && idx < stk->get_size()){
70              Node* node = stk->get_head()->next; // Start from the first actual element (not sentinel)
71              for(int i = 0; i < stk->get_size()- idx-1; i++) {
72                  node = node->next;
73              }
74              return node->get_value();
75      }
76
77          else {
78              throw std::runtime_error("Index out of range");
79          }
80
81      }
82
83      void Stack_C::print_stack(bool top_or_bottom){
84
85
86          if (stk->get_size() == 0) {
87              std::cout << "Stack is empty." << std::endl;
88              return;
89          }
90
91          if (top_or_bottom) {
92              Node* node = stk->get_head()->next;
93              while (node != nullptr) {
94                  std::cout << node->get_value() << " ";
95                  node = node->next;
96              }
97          } else {
98              Node* node = stk->get_head();
99              while (node->next != nullptr) {
100                 node = node->next;
101             }
```

```
102             while (node != stk->get_head()) {
103                 std::cout << node->get_value() << " ";
104                 node = node->prev;
105             }
106         }
107     }
108 
109     int Stack_C:: add(){
110         if(stk->get_size()<2){
111             throw std::runtime_error("Not Enough Arguments");
112 
113         }
114         int op2=pop();
115 
116         int op1=pop();
117 
118         int result= op1+op2;
119 
120         push(result);
121         return result;
122 
123     }
124 
125 
126     int Stack_C::subtract(){
127         if(stk->get_size()<2){
128             throw std::runtime_error("Not Enough Arguments");
129 
130         }
131         int op2= pop();
132 
133         int op1= pop();
134 
135         int result= op1-op2;
136 
137         push(result);
138 
139         return result;
140     }
141 
142     int Stack_C::multiply(){
143         if(stk->get_size()<2){
144             throw std::runtime_error("Not Enough Arguments");
145 
146         }
147         int op2= pop();
148         int op1= pop();
149         int result= op1 * op2;
150 
151         push(result);
152         return result;
153 
```

```
154
155     }
156     // int floorf(double& x){
157     //     if (x>0){
158     //         return x;
159
160     //     }
161     //     else {
162     //         return (x-1);
163     //     }
164     // }
165
166     int  Stack_C::divide(){
167         if(stk->get_size()<2){
168             throw runtime_error("Not Enough Arguments");
169
170         }
171         int op2= pop();
172         int op1= pop();
173
174
175         if (op2 !=0){
176             int c= op1/op2;
177             if ( op1>0 && op2<0 || op1<0 && op1>0){
178                 c=c-1;
179             }
180
181           int  result= c;
182             // result= floorf(result);
183
184
185             push(result);
186             return result;
187
188         }
189         else{
190             throw runtime_error("Divide by Zero Error");
191         }
192     }
193
194
195     List* Stack_C::get_stack(){
196         return stk;
197     } // Get a pointer to the linked list representing the stack
198
199     int Stack_C::get_size(){
200         return stk->get_size();
201     } // Get the size of the stack
202
203
204
205
```

206
207
208