Lecture 7

Goals:
- stacks
  Implementation of stack applications
- Postfix expression evaluation
- Convert infix to postfix
Stack Overview

Stack ADT
Basic operations of stack
push, pop, top, isEmpty etc.

Implementations of stacks using
Array
Linked list

Application to arithmetic expression evaluation
A stack is a list in which insertion and deletion take place at the same end.
This end is called top.
The other end is called bottom.

Stacks are known as LIFO (Last In, First Out) lists.
The last element inserted will be the first to be retrieved.
Push and Pop

Primary operations: Push and Pop

Push
Add an element to the top of the stack

Pop
Remove the element at the top of the stack
Implementation of Stacks

Any list implementation could be used to implement a stack
- arrays (static: the size of stack is given initially)
- Linked lists (dynamic: never becomes full)

We will explore implementations based on array
class Stack {
public:
    Stack(int size = 10); // constructor
    ~Stack() { delete [] values; } // destructor
    bool IsEmpty() { return top == -1; }
    bool IsFull() { return top == maxTop; }
    double Top();    // examine, without popping
    void Push(const double x);
    double Pop();
    void DisplayStack();
private:
    int maxTop;        // max stack size = size - 1
    int top;           // current top of stack
    double* values;    // element array
};
Stack class

Attributes of Stack

- maxTop: the max size of stack
- top: the index of the top element of stack
- values: point to an array which stores elements of stack

Operations of Stack

- IsEmpty: return true if stack is empty, return false otherwise
- IsFull: return true if stack is full, return false otherwise
- Top: return the element at the top of stack
- Push: add an element to the top of stack
- Pop: delete the element at the top of stack
- DisplayStack: print all the data in the stack
Create Stack

The constructor of Stack

Allocate a stack array of size. By default, size = 10.
Initially top is set to -1. It means the stack is empty. When the stack is full, top will have its maximum value, i.e. size - 1.

```cpp
Stack::Stack(int size /*= 10*/) {
    values = new double[size];
    top = -1;
    maxTop = size - 1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed.
void Push(const double x);
Push an element onto the stack
Note top always represents the index of the top element. After pushing an element, increment top.

void Stack::Push(const double x) {
    if (IsFull()) // if stack is full, print error
        cout << "Error: the stack is full." << endl;
    else
        values[++top] = x;
}
Pop Stack

double Pop()

Pop and return the element at the top of the stack

Don’t forget to decrement top

double Stack::Pop() {
    if (IsEmpty()) { //if stack is empty, print error
        cout << "Error: the stack is empty." << endl;
        return -1;
    }
    else {
        return values[top--];
    }
}
Stack Top

```cpp
double Top() {
    if (IsEmpty()) {
        cout << "Error: the stack is empty." << endl;
        return -1;
    }
    else
        return values[top];
}
```
Printing all the elements

```cpp
void DisplayStack()
{
    Print all the elements
}

void Stack::DisplayStack()
{
    cout << "top -->";
    for (int i = top; i >= 0; i--)
        cout << "\t|\t" << values[i] << "\t|" << endl;
    cout << "\t|---------------|" << endl;
}
```

```text

<table>
<thead>
<tr>
<th>top --&gt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-8</td>
</tr>
<tr>
<td></td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

---------------
```
int main(void) {
    Stack stack(5);
    stack.Push(5.0);
    stack.Push(6.5);
    stack.Push(-3.0);
    stack.Push(-8.0);
    stack.DisplayStack();
    cout << "Top: " << stack.Top() << endl;

    stack.Pop();
    cout << "Top: " << stack.Top() << endl;
    while (!stack.IsEmpty()) stack.Pop();
    stack.DisplayStack();
    return 0;
}
Implementation based on Linked List

Now let’s implement a stack based on a linked list.
To make the best out of the code of List, we implement Stack by inheriting List.
To let Stack access private member head, we make Stack a friend of List.

```cpp
class List {
public:
    List(void) { head = NULL; } // constructor
    ~List(void); // destructor
    bool IsEmpty() { return head == NULL; }
    Node* InsertNode(int index, double x);
    int FindNode(double x);
    int DeleteNode(double x);
    void DisplayList(void);

private:
    Node* head;
    friend class Stack;
};
```
Implementation based on Linked List

class Stack : public List {
public:
    Stack() {} // constructor
    ~Stack() {} // destructor
    double Top() {
        if (head == NULL) {
            cout << "Error: the stack is empty." << endl;
            return -1;
        }
        else
            return head->data;
    }
    void Push(const double x) { InsertNode(0, x); }
    double Pop() {
        if (head == NULL) {
            cout << "Error: the stack is empty." << endl;
            return -1;
        }
        else {
            double val = head->data;
            DeleteNode(val);
            return val;
        }
    }
    void DisplayStack() { DisplayList(); }
};

Note: the stack implementation based on a linked list will never be full.
Array implementation versus linked list implementations

push, pop, top are all constant-time operations in both array implementation and linked list implementation

For array implementation, the operations are performed in very fast constant time
Application 1: Balancing Symbols

To check that every right brace, bracket, and parentheses must correspond to its left counterpart e.g. [()]{()} is legal, but {[( ])} is illegal

Algorithm

(1) Make an empty stack.
(2) Read characters until end of file
   i. If the character is an opening symbol, push it onto the stack
   ii. If it is a closing symbol, then if the stack is empty, report an error
   iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error
(3) At end of file, if the stack is not empty, report an error
Application 2: Expression evaluation

Given an arithmetic expression such as:

\[ x + y \times (z + w) \]

(Given values assigned to \( x = 23 \), \( y = 12 \), \( z = 3 \) and \( w = -4 \))

What is the value of the expression?

Goal: Design a program that takes as input an arithmetic expression and evaluates it.

This task is an important part of compilers. As part of this evaluation, the program also needs to check if the given expression is correctly formed.

Example of bad expressions:

\( (3 + 12 \times (5 - 3) \)

\( A + B \times + C \) etc.
Postfix expression

Instead of writing the expression as $A + B$, we write the two operands, then the operator.

Example: $a \ b \ c \ + \ *$

It represents $a \cdot (b + c)$

Question: What is the postfix form of the expression $a + b \cdot c$?
Postfix expression

Instead of writing the expression as $A + B$, we write the two operands, then the operator.

Example: $a \ b \ c \ +*$

It represents $a \cdot (b + c)$

Question: What is the postfix form of the expression $a + b \cdot c$?

Answer: $a \ b \ c \ * \ +$
Algorithm for evaluating postfix expression

- Use a stack.
- Push operands on the stack.
- When you see an operator, pop off the top two elements of the stack, apply the operator, push the result back.
- At the end, there will be exactly one value left on the stack which is the final result.
Algorithm for evaluating postfix expression

- Use a stack.
- Push operands on the stack.
- When you see an operator, pop off the top two elements of the stack, apply the operator, push the result back.
- At the end, there will be exactly one value left on the stack which is the final result.

Example: 12 8 3 * +

Finally, the result 36 is pushed back on the stack.
Implementation of exp evaluation

Token class:

class token {
private: int op_type;
    double value;
public:
    token(int x, int y) {
        op_type = x; op_value = y;
    }
    int get_op_type() {
        return op_type;
    }
    double get_value() {
        return value;
    }
    void set_op_type(int x) { op_type = x; }
    void set_value(double y) { value = y; }
};

Op_type:
1 → +
2 → -
3 → *
4 → /
5 → **
6 → operand
-1 → token represents end of expression

Op_value: value of the operand.
Input

Look at the main program:

```c
int main(void) {
    string str = "908 100 200+ 23 19 * +/ 123 *";
    Expr ex(str, 0);
    double rslt = ex.eval();
    cout << "The result of evaluation is " << rslt << endl;
    return 0;
}
```

There must be a space between successive operands. There need not be a space when an operand follows an operator, and after an operator. There can be more than one space after any token, including the last.

```
C:\PROGRA~1\dm\bin>stack_eval
The result of evaluation is 151.539
```
Implementation of exp evaluation

double eval() {
    Stack st(MAX_SIZE);
    token tok = get_token();  // gets the next token
    while (tok.get_op_type() != -1) {
        if (tok.get_value() != 0)
            st.Push(tok.get_value());
        else {
            double opd2 = st.Pop();
            double opd1 = st.Pop();
            double op = apply(tok.get_op_type(), opd1, opd2);
            st.Push(op);
        }
        current++; tok = get_token();
    }
    double result = st.Pop(); return result;
} // eval
}; // end Expr
token get_token() {
    token tok( -1, 0);
    if (current > exp.length() - 1)
        return tok;
    while (exp[current] == ' ') current++;
    if (current > exp.length() - 1) return tok;
    if (exp[current] == '+') tok.set_op_type(1);
    else if (exp[current] == '-') tok.set_op_type(2);
    else if (exp[current] == '/') tok.set_op_type(4);
    else if (exp[current] == '*') {
        if (exp[current+1] != '*') tok.set_op_type(3);
        else {tok.set_op_type(5); current++;
    }
    } else { // token is an operand
        double temp = 0.0;
        while (!(exp[current] == ' ') && !optr(exp[current])) {
            temp= 10*temp+val(exp[current]); current++;
        }
        if (optr(exp[current])) current--;
        tok.set_op_type(6);
        tok.set_value(temp);
    }
    return tok;
} //end get_token