Stacks

August 8, 2020

1 Stacks

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/StackArray.html
- https://en.cppreference.com/w/cpp/container/stack

1.1 Table of Contents

1.2 Introduction

- stack is a list-like data structure in which elements may be inserted or removed from only one end
  - less flexible than list
  - more efficient and easy to implement
- **LIFO** (Last-In, First-Out) data structure
- many applications require the limited form of insert and remove operations that stacks provide
- uses real-world analogy of stacks e.g., stacks of coins, books, boxes, plates, etc.

1.2.1 Applications

- stack structure is used in memory management for local variables
- used to evaluate prefix, postfix and infix expressions
- can be used to convert expression in one form to anther
- parse syntax in programming language to verify grammar
- backtracking in solving games and puzzles
- checking for valid parenthesis

1.2.2 Operations

- push : insert element at the top of the stack
- pop : remove and return element from the top of the stack

1.3 Implementations of Stacks as ADT

- Stacks can be implemented using array or linked-list

1.3.1 Array implementation of Stack

- array-based stack is as simple as array-based list
- below is the array-based stack

1.3.2 Visualization of Array-based Stack

https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/StackArray.html

[ ]:
```
#include <iostream>
#include <cassert>

using namespace std;
```

[ ]:
```
template<class T>
class ArrayStack {
  private:
    size_t maxSize;
    size_t top;
    T * stack;
  public:
    ArrayStack(size_t mSize=100) { //constructor
      assert(mSize > 0);
      maxSize = mSize;
      stack = new T[maxSize];
      top = 0;
    }

    // clear the stack
    void clear() { top = 0; }

    // get the size of the stack
    size_t size() { return top; }

    // check if stack is empty
    bool empty() { return top == 0; };

    // check if stack is full
    bool full() { return top == maxSize; }

    // return the max size
    size_t max_size() { return maxSize; }

    //push data on the top of the stack
    void push(T value) {
        assert(!full());
        stack[top++] = value;
    }
```

```
    // remove and return the element from the top of the stack
    T pop() {
        assert(!empty());
        return stack[--top];
    }
    // return the top element
    T get_top() {
        assert(!empty());
        return stack[top-1];
    }
};
```

1.4 Test ArrayStack Implementation

[ ]:
```
ArrayStack<int> iStack;
```

[4]:
```
iStack.push(10);
iStack.push(20);
iStack.push(30);
cout << "size of iStack = " << iStack.size();
```

```
size of iStack = 3
```

[5]:
```
cout << "top of the stack is: " << iStack.pop() << endl;
cout << "now the size = " << iStack.size() << endl;
```

```
top of the stack is: 30
now the size = 2
```

[6]:
```
iStack.push(40);
```

[7]:
```
while(!iStack.empty()) {
    cout << iStack.pop() << endl;
}
```

```
40
20
10
```

1.5 Linked Stack Implementation

- elements are inserted and removed only from the head of the list
- header node is not used because no special-cases need to be handled
- below is linked stack implementation

[1]:
```
#include <iostream>
#include <cassert>

using namespace std;
```

[2]:
```
template<class T>
struct Node {
  T data;
  Node<T> * next;
};
```

[3]:
```
template<class T>
class LinkedStack{
  private:
    size_t nodeCount;
    Node<T> * head;
  public:
    //constructor
    LinkedStack() {
      nodeCount = 0;
      head = nullptr;
    }

    // clear the stack
    void clear() {
        nodeCount = 0;
        Node<T> * curr = head;
        while ( curr != nullptr ) {
            head = head->next;
            delete curr;
            curr = head;
        }
    }

    // get the size of the stack
    size_t size() { return nodeCount; }

    // check if stack is empty
    bool empty() { return nodeCount == 0; };

    //push data on the top of the stack
    void push(T value) {
        Node<T> * node = new Node<T>;
        node->data = value;
        node->next = head;
        head = node;
        nodeCount++;
```

```
    }

    // remove and return element from top of the stack
    T pop() {
        assert(!empty());
        T data = head->data;
        // adjust head pointer and delete head node
        Node<T> * curr = head;
        head = head->next;
        delete curr;
        nodeCount--;
        return data;
    }

    T top() {
        assert(!empty());
        return head->data;
    }
};
```

1.6 Test Linked Stack Implementation

[4]:
```
LinkedStack<int> lStack;
```

[5]:
```
lStack.push(10);
lStack.push(20);
lStack.push(30);
cout << "size of lStack = " << lStack.size() << endl;
```

```
size of lStack = 3
```

[6]:
```
cout << "top element = " << lStack.pop() << endl;
cout << "now the size = " << lStack.size() << endl;
```

```
top element = 30
now the size = 2
```

[7]:
```
cout << boolalpha;
cout << "is lStack empty? " << lStack.empty();
```

```
is lStack empty? false
```

[8]:
```
while(!lStack.empty()) {
    cout << lStack.top() << " ";
    lStack.pop();
}
```

```
20 10
```

1.7 Exercises

1. Backspace problem: https://open.kattis.com/problems/backspace

- Game of throwns: https://open.kattis.com/problems/throwns
- Even Up Solitaire: https://open.kattis.com/problems/evenup
- Working at the Restaurant: https://open.kattis.com/problems/restaurant
- Pairing Socks: https://open.kattis.com/problems/pairingsocks
- Find stack-based problems in Kattis: https://cpbook.net/methodstosolve search for stack

[ ]: