

ECE2800J

Programming and Elementary Data Structures

Linear List; Stack

Learning Objectives:

Understand what is a linear list and what is a stack

Know how they can be implemented

Discover some applications of the stack data structure

Outline

- Linear List
- Stack
 - Implementation
 - Application

Linear List ADT

- Recall the IntSet ADT
 - A collection of zero or more integers, with **no duplicates**.
 - It supports insertion and removal, but by value.
- A related ADT: linear list
 - A collection of zero or more integers; **duplicates possible**.
 - $L = (e_0, e_1, \dots, e_{N-1})$
 - It supports insertion and removal **by position**.

Linear List ADT

Insertion

```
void insert(int i, int v) // if 0 <= i <= N  
// (N is the size of the list), insert v at  
// position i; otherwise, throws BoundsError  
// exception.
```

Example: L1 = (1, 2, 3)

```
L1.insert(0, 5) = (5, 1, 2, 3);  
L1.insert(1, 4) = (1, 4, 2, 3);  
L1.insert(3, 6) = (1, 2, 3, 6);  
L1.insert(4, 0) throws BoundsError
```

Linear List ADT

Removal

```
void remove(int i) // if 0 <= i < N (N is  
// the size of the list), remove the i-th  
// element; otherwise, throws BoundsError  
// exception.
```

Example: L2 = (1, 2, 3)

```
L2.remove(0) = (2, 3);  
L2.remove(1) = (1, 3);  
L2.remove(2) = (1, 2);  
L2.remove(3) throws BoundsError
```



Which Answers Are Correct?

Suppose we want to implement a linear list ADT so that it can grow as large as the user wants. It can be implemented by:

- A. a static array
- B. a dynamic array
- C. a singly-linked list
- D. a doubly-linked list



Outline

- Linear List
- Stack
 - Implementation
 - Application

Stack

- A “pile” of objects where new object is put on **top** of the pile and the top object is removed first.
 - Restricted form of a **linear list**: insert and remove only at the end of the list.
 - LIFO access: last in, first out.



Methods of Stack

- **size ()** : number of elements in the stack.
- **isEmpty ()** : checks if stack has no elements.
- **push (Object o)** : add object **o** to the top of stack.
- **pop ()** : remove the top object if stack is not empty;
otherwise, throw **stackEmpty**.
- **Object &top ()** : return a reference to the top element.

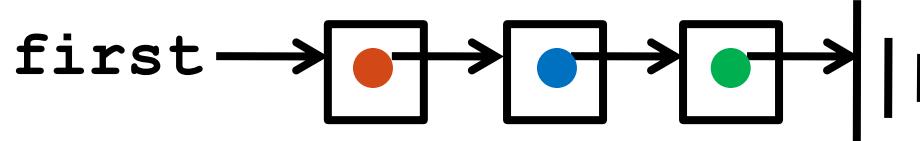
Stacks Using Arrays

Array[MAXSIZE] :

2	3	1	4		
---	---	---	---	--	--

- Maintain an integer **size** to record the size of the stack.
- **size(): return size;**
- **isEmpty(): return (size == 0);**
- **push(Object o)**: add object **o** at index **size** of the array and increment **size**. Allocate more space if necessary.
- **pop()**: If **isEmpty()**, throw **stackEmpty**; otherwise, decrement **size**.
- **Object &top()**: return a reference to the top element
Array[size-1]

Stacks Using Linked Lists

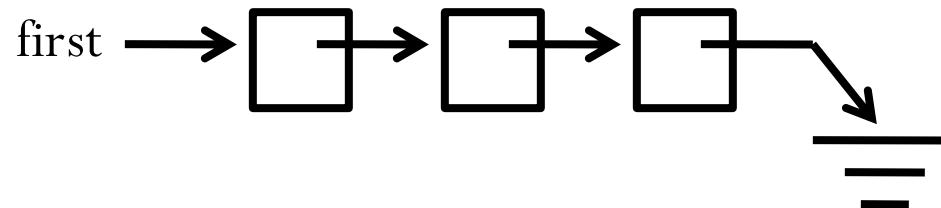


For single-ended linked list,
which end is preferred
to be the top? Why?

- **size () : LinkedList::size () ;**
- **isEmpty () : LinkedList::isEmpty () ;**
- **push (Object o) :** insert object at the beginning
LinkedList::insertFirst (Object o) ;
- **pop () :** remove the first node
LinkedList::removeFirst () ;
- **Object &top () :** return a reference to the object stored
in the first node.

LinkedList::size()

- How to get the size of a linked list?



```
int LinkedList::size() {  
    int count = 0;  
    node *current = first;  
    while(current) {  
        count++;  
        current = current->next;  
    }  
    return count;  
}
```

A fast version: add a size data member

Linked List with a size data member

```
class LinkedList {  
    node *first;  
    int size;  
public:  
    ...  
};
```

```
int LinkedList::size()  
{  
    return size;  
}
```

- Question: do we need to change any other parts of the code?
 - We need to increment/decrement **size** when nodes are inserted/removed.

Array vs. Linked List: Which is Better?

- Linked list **with a size data member** is better
- Array
 - not memory-efficient: need to allocate a big enough array
- Linked list with a size data member
 - memory-efficient: a new item just needs extra constant amount of memory
 - All operations are of constant runtime, same as array

Outline

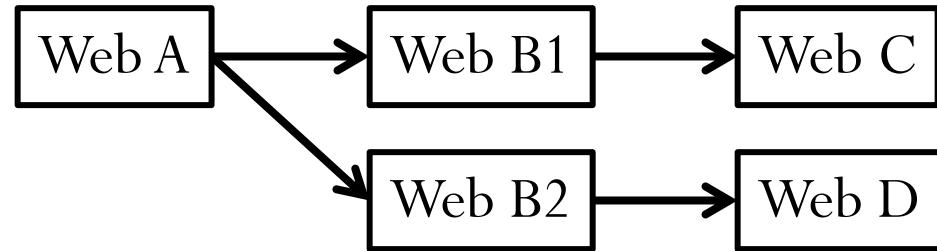
- Linear List
- Stack
 - Implementation
 - Application

Application of Stacks

- Function calls in C++
- Web browser's "back" feature
- Parentheses Matching

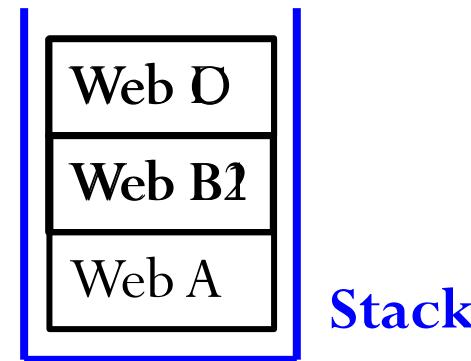
Web Browser's “back” Feature

- Stack stores URL



Visiting order

- Web A
- Web B1
- Web C
- Back (to Web B1)
- Back (to Web A)
- Web B2
- Web D



Parentheses Matching

- Output pairs (u, v) such that the left parenthesis at position u is matched with the right parenthesis at v .

$$((\mathbf{a} + \mathbf{b}) * \mathbf{c} + \mathbf{d} - \mathbf{e}) / (\mathbf{f} + \mathbf{g})$$

0 1 2 3 4 5 6 7 8 9 10 12 14 16 18

- Output is: (1, 5); (0, 12); (14, 18);

$$(\mathbf{a} + \mathbf{b})) * ((\mathbf{c} + \mathbf{d})$$

0 1 2 3 4 5 6 7 8 9 10 12

- Output is

(0, 4);

Right parenthesis at 5 has no matching left parenthesis;

(8, 12);

Left parenthesis at 7 has no matching right parenthesis

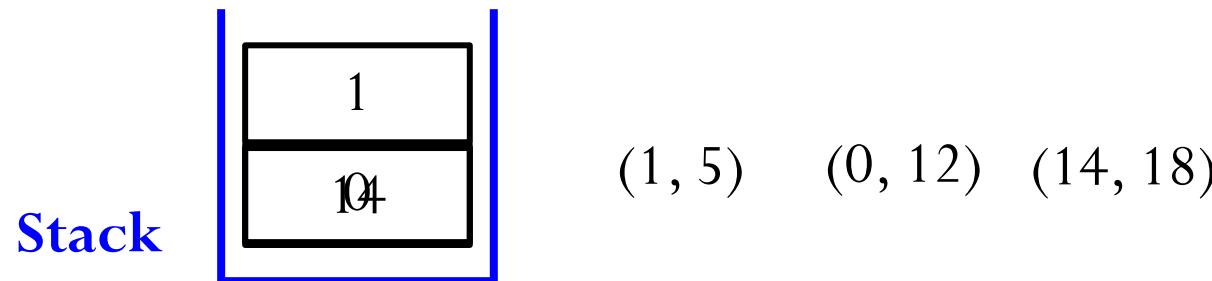
How to Realize Parentheses Matching?

((a + b) * c + d - e) / (f + g)
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18

- Scan expression from left to right.
- When a **left** parenthesis is encountered, push its position to the stack.
- When a **right** parenthesis is encountered, pop the top position from the stack, which is the position of the **matching left** parenthesis.
 - If the stack is empty, the **right** parenthesis is not matched.
 - If string is scanned over but the stack is not empty, there are not-matched **left** parentheses.

Parentheses Matching

((a + b) * c + d - e) / (f + g)
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18



Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
 - Chapter 13.2 Stack