Lecture No.06

Data Structures

Stack Using Linked List

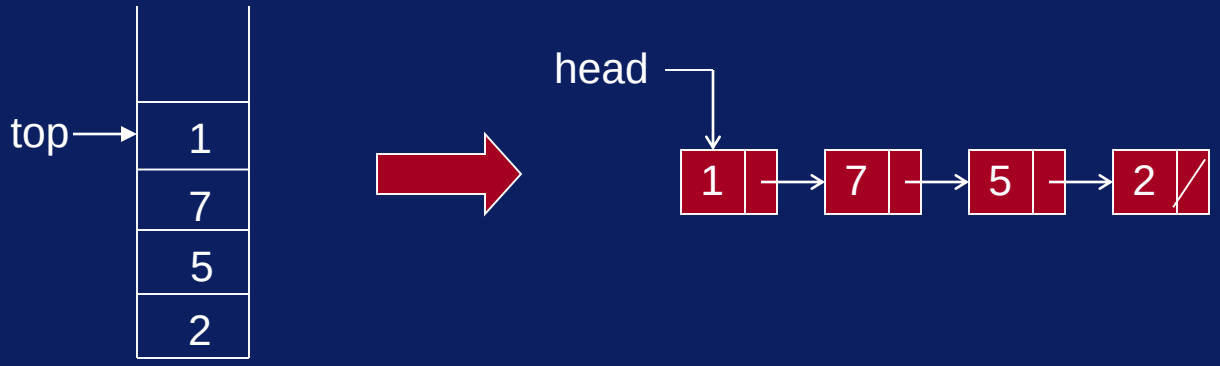
- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.
- As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

Stack Using Linked List

- For a singly-linked list, insert at start or end takes constant time using the head and current pointers respectively.
- Removing an element at the start is constant time but removal at the end required traversing the list to the node one before the last.
- Make sense to place stack elements at the start of the list because insert and removal are constant time.

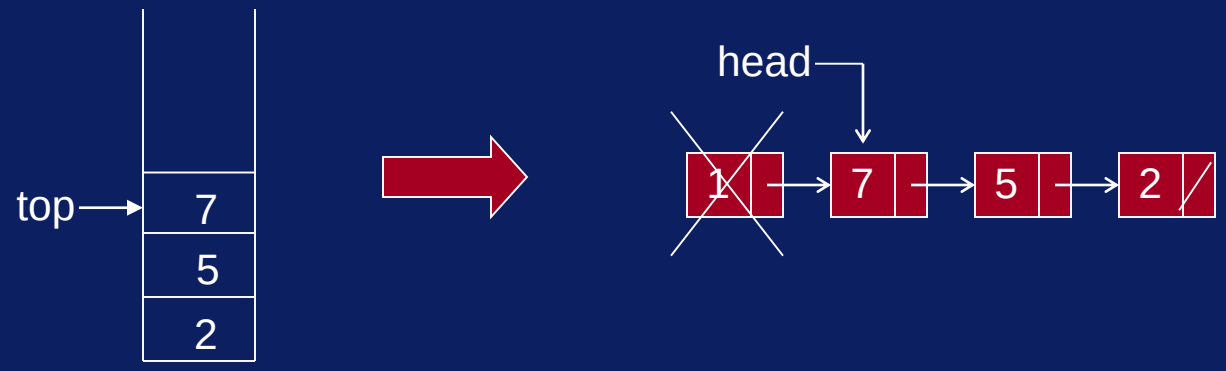
Stack Using Linked List

- No need for the current pointer; head is enough.



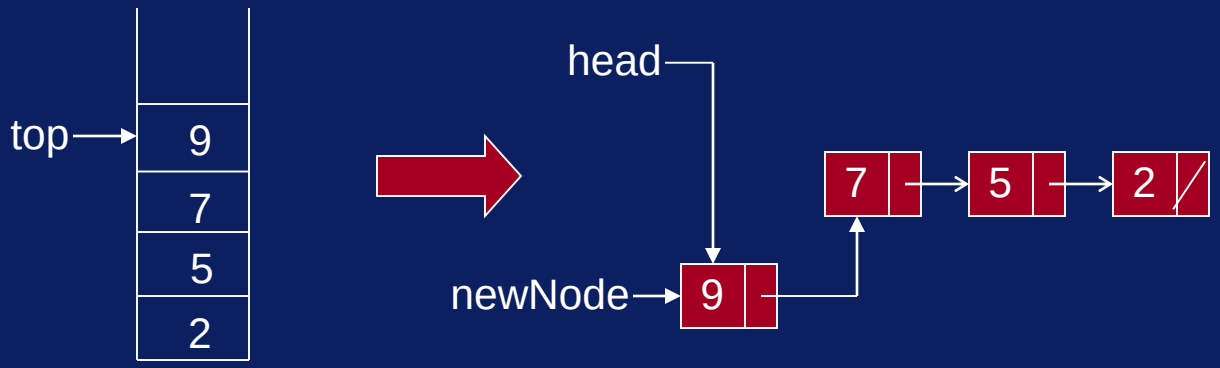
Stack Operation: List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    delete p;
    return x;
}
```



Stack Operation: List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
}
```



push(9)

Stack Operation: List

```
int top()
{
    return head->get();
}
int IsEmpty()
{
    return ( head == NULL );
}
```

- All four operations take constant time.

Stack: Array or List

- Since both implementations support stack operations in constant time, any reason to choose one over the other?
- Allocating and deallocating memory for list nodes does take more time than preallocated array.
- List uses only as much memory as required by the nodes; array requires allocation ahead of time.
- List pointers (head, next) require extra memory.
- Array has an upper limit; List is limited by dynamic memory allocation.

Use of Stack

- Example of use: prefix, infix, postfix expressions.
- Consider the expression A+B: we think of applying the *operator* “+” to the *operands* A and B.
- “+” is termed a *binary operator*: it takes two operands.
- Writing the sum as A+B is called the *infix* form of the expression.

Prefix, Infix, Postfix

- Two other ways of writing the expression are

+ A B	*prefix*
A B +	*postfix*

- The prefixes “pre” and “post” refer to the position of the operator with respect to the two operands.

Prefix, Infix, Postfix

- Consider the infix expression

  A + B * C
- We “know” that multiplication is done before addition.
- The expression is interpreted as

  A + (B * C)
- Multiplication has *precedence* over addition.

Prefix, Infix, Postfix

- Conversion to postfix

  A + (B * C) infix form

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition
A B C * +	postfix form

Prefix, Infix, Postfix

- Conversion to postfix

  (A + B) * C infix form

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication

Prefix, Infix, Postfix

- Conversion to postfix

(A + B) * C	infix form
(A B +) * C	convert addition
(A B +) C *	convert multiplication
A B + C *	postfix form

Precedence of Operators

- The five binary operators are: addition, subtraction, multiplication, division and exponentiation.
- The order of precedence is (highest to lowest)
- Exponentiation ↑
- Multiplication/division *, /
- Addition/subtraction +, -

Precedence of Operators

- For operators of same precedence, the left-to-right rule applies:

  A+B+C means (A+B)+C.

- For exponentiation, the right-to-left rule applies

  $A \uparrow B \uparrow C$ means $A \uparrow (B \uparrow C)$

Infix to Postfix

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+