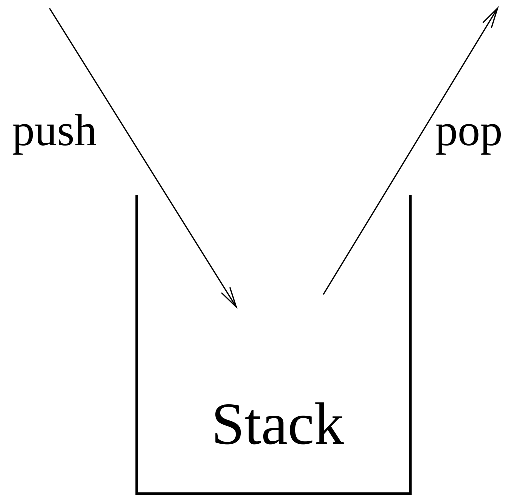
Stacks

- A **Stack** is a sequential organization of items in which the last element inserted is the first element removed. They are often referred to as LIFO, which stands for "last in first out."
- **Examples:** letter basket, stack of trays, stack of plates.
- The *only* element of a stack that may be accessed is the one that was most recently inserted.
- There are only two basic operations on stacks, the *push* (insert), and the *pop* (read and delete).



Stacks: Push and Pop

- The operation **push(x)** places the item x onto the top of the stack.
- The operation **pop()** removes the top item from the stack, and returns that item.
- We need some way of detecting an empty stack (This is an *underfull* stack).
  - In some cases, we can have **pop()** return some value that couldn't possibly be on the stack.
  - **Example:** If the items on the stack are positive integers, we can return "-1" in case of underflow.
  - In other cases, we may be better off simply keeping track of the size of the stack.
- In some cases, we will also have to worry about filling the stack (called *overflow*). One way to do this is to have **push(x)** return "1" if it is successful, and "0" if it fails.

An Example Stack Operations

Assume we have a stack of size 3 which holds integers between -100 and 100. Here is a series of operations, and the results.

Operation	Stack Contents	Return
create	()	
push(55)	(55)	1
push(-7)	(-7,55)	1
push(16)	(16,-7,55)	1
pop	(-7,55)	16
push(-8)	(-8,-7,55)	1
push(23)	(-8,-7,55)	0
pop	(-7,55)	-8
pop	(55)	-7
pop	()	55
pop	()	101

Implementing Stacks: Array

- Stacks can be implemented with an array and an integer top that stores the array index of the top of the stack.
- Empty stack has $top = -1$, and a full stack has $top = n - 1$, where n is the size of the array.
- To push, increment the top counter, and write in the array position.
- To pop, decrement the top counter.

Example of Array Implementation of Stack

- We use an array $E[0..4]$ to store the elements and a variable p to keep track of the top. The last column, labeled "R" is the result of the function call.

Operation	p	E0	E1	E2	E3	E4	R
create	-1	?	?	?	?	?	
push(55)	0	55	?	?	?	?	1
push(-7)	1	55	-7	?	?	?	1
push(16)	2	55	-7	16	?	?	1
pop	1	55	-7	16	?	?	16
push(-8)	2	55	-7	-8	?	?	1
pop	1	55	-7	-8	?	?	-8
pop	0	55	-7	-8	?	?	-7

- Notice that some values are still in the array, but are no longer considered to be in the stack. In general, elements $E[i]$ are "garbage" if $i > p$. Why don't we erase the element (i.e. set it to some default value.)?

Example Application of Stacks

- Stacks can be used to check for balanced symbols (such as $\{\}$,(),[ ]).
- **Example:** $\{()\}$ is legal, as is $\{()(\{\})\}$, whereas $\{((\}$ and $\{(\})$ are not.
- If the symbols are balanced correctly, then when a closing symbol is seen, it should match the "most recently seen" unclosed opening symbol. Therefore, a stack will be appropriate.

The following algorithm will do the trick:

- While there is still input:

```
    $s$ = next symbol
    if ($s$ is an opening symbol) push($s$)
    else                //$s$ is a closing symbol
        if (Stack.Empty) report an error
        else $r$ = pop()
        if(! Match($s$,$r$)) report an error
```

- If (! Stack.Empty) report an error

Examples

1. Input: { ( ) }
   - Read {, so push {
   - Read (, so push (. Stack has { (
   - Read), so pop. popped item is (which matches). Stack has now {.
   - Read }, so pop; popped item is { which matches }.
   - End of file; stack is empty, so the string is valid.
2. Input: { ( ) ({) } } (This will fail.)
3. Input: { ({ }){ } ( ) } (This will succeed.)
4. Input: { ( ) }) (This will fail.)

Stack: C++ Array Implementation

- The following is a C++ stack that holds items of type *ItemType*

```
class Stack {
   private:
     ItemType *stack;
     int p;
   public:
     Stack(int max=100)
       {stack = new ItemType[max];
        p = 0; }
    ~Stack()
       {delete stack; }
     void push(ItemType v)
       { stack[p++] = v; }
     ItemType pop()
       {return stack[--p]; }
     int empty()
       {return !p; }
   };
```

Stack: JAVA Array Implementation

- The following is a JAVA stack that holds items of type *ItemType*

```
public class Stack {
     private ItemType[] stack;
     private int p;
     public Stack(int max) {
        stack = new ItemType[max];
        p = 0;
     }
     public void push(ItemType v) {
        stack[p++] = v;
     }
     public ItemType pop() {
        return stack[--p];
     }
     public int empty() {
        return !p;
     }
}
```

Operator Precedence Parsing

- We can use the stack class we just defined to parse and evaluate mathematical expressions like: $5 * (((9 + 8) * (4 * 6)) + 7)$
- First, we transform it to postfix notation (How would you do this?):

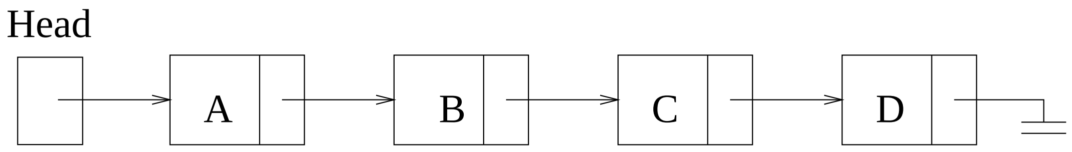
$$5\ 9\ 8 + 4\ 6 * * 7 + *$$

- The following code uses a stack to perform this evaluation:

```
while ((c = nextChar()) !=null) {
   x = 0;
   while (c == ’ ’ ) c = nextChar();
   if (c == ’+’) x = stack.pop() + stack.pop();
   if (c == ’*’) x = stack.pop() * stack.pop();
   while (c>=’0’ && c<=’9’)
     {x = 10*x + (c-’0’); c = nextChar(); }
   stack.push(x);
}
print(stack.pop());
```

Stack Implementation: C++ Linked Lists

- We can use linked lists to implement stacks.
- The head of the list represents the top of the stack.
- **Example** After $push(D)$, $push(C)$, $push(B)$, $push(A)$, we have:



-
  ```
  ItemType pop () {
      ItemType x = head->key;
      head=head->next;
      return x; }
  ```

-
  ```
  void push(ItemType x) {
      node *x;
      x=new node;
      x->key = X;
      insert(x); }
  ```
- **Note:** Slight error in `pop`. What is it?

Stack Implementation: JAVA Linked Lists

We assume *head* points to the first element of the list, and is the top of the stack.

-
  ```
  public ItemType pop () {
          ItemType x = head.key;
          head=head.next;
          return x;
  }
  ```

-
  ```
  public void push(ItemType x) {
          node A=new node();
          A.key = x;
          A.next=head;
          head=A;
  }
  ```

Stack Implementation: Array or Linked List?

Linked Lists

- Use 1 pointer extra memory per item. If an item is an integer, that means twice as much space is used. If an item is a structure/class consisting of many objects, it is only a small price to pay.
- Are unlimited in size.

Arrays

- Allocate a constant amount of space, some of which may never be used. The amount of wasted memory is the number of unused elements times the size of the item, which could be large in some cases.
- The maximum size is determined when the stack is created.

Which is better? Why?

Stack Applications

- Recursion removal can be done with stacks.
- Reversing things is easily done with stacks.
- Procedure call and procedure return is similar to matching symbols:
  - When a procedure returns, it returns to the most recently active procedure.
  - When a procedure call is made, save current state on the stack. On return, restore the state by popping the stack.