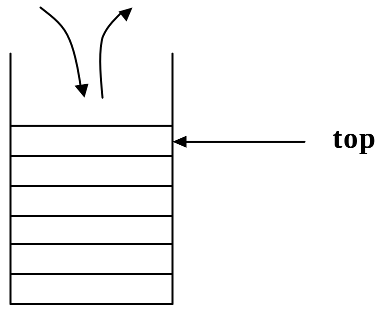
STACK

- A *stack* is a collection of items into which new items are inserted and from which items are deleted at one end (called the *top* of the stack).

- Different implementations are possible; although the concept of a stack is unique.

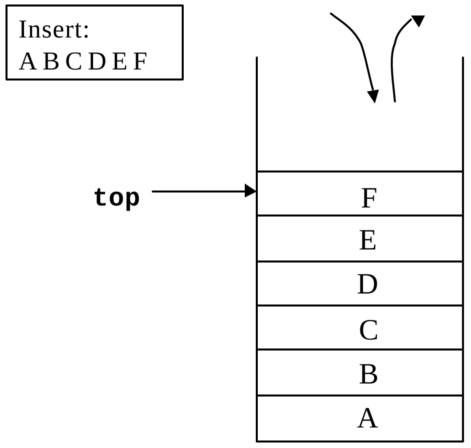
Example: Trays in the cafeteria.

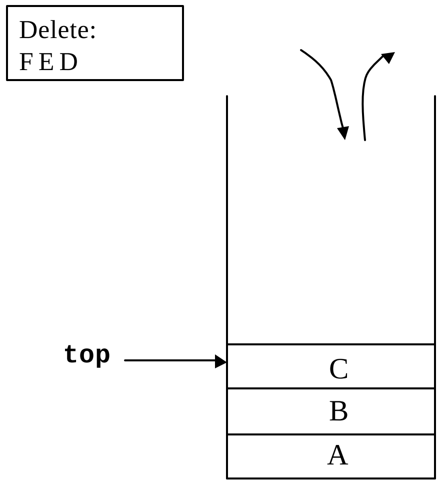


- Two primary operations:
  1. **push**: adds a new item on top of a stack.
  2. **pop**: removes the item on the top of a stack

- Stack is also known as push-down list
- LIFO (Last In First Out): order of addition and deletion of items from a stack.

A stack is a dynamic structure. It changes as elements are added to and removed from it.

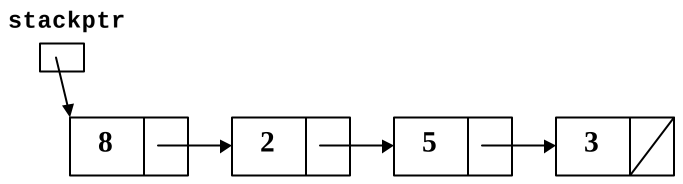




Data Structure

A stack can be implemented as a constrained version of a linked list. A stack is referenced via a pointer to the top element of the stack. The link member in the last node of the stack is set to NULL to indicate the bottom of the stack.

Example:



– `stackptr` points to the top of the stack.

Note that stacks and linked lists are represented identically. The difference is that insertions and deletions occur anywhere in a linked list, but only at the top of a stack.

- Function `push` creates a new node and places it on top of the stack.

- Function `pop` removes a node from the top of the stack, frees the memory that was allocated to the popped node, and returns the popped value.

Stack Operations

- **Implementation of a simple stack of integers**

```
struct stackNode{
    int data;
    struct stackNode *nextPtr;
};

/* Insert a node at the top of the stack */
void push(struct stackNode **topPtr, int info)
{
    struct stackNode *newPtr;

    newPtr = (struct stackNode *)
             malloc(sizeof (struct stackNode));
    if (newPtr != NULL) {
        newPtr ->data = info;
        newPtr->nextPtr = *topPtr;
        *topPtr = newPtr;
    }
    else
        printf(“%d not inserted. No memory”
               “available.\n”, info);
}
```

```
/*Remove a node from the stack top */
int pop(struct stackNode **topPtr)
{
   struct stackNode *tempPtr;
   int popValue;

   tempPtr = *topPtr;
   popValue = (*topPtr)->data;
   *topPtr = (*topPtr)->nextPtr;
   free(tempPtr);
   return popValue;
}
```

```
/*Is the stack empty? */
int isEmpty(struct stackNode *topPtr)
{
   return topPtr == NULL;
}
```

Example Applications

- Reading a line of text and writing it out backwards.

```
int main()
{
   struct stackNode *top = NULL;
   int c;

   while ((c=getchar() )!=’\n’)
      push(&top, c);

   while (!isEmpty(top))
      printf(“%c”, pop(&top));
   printf(“\n”);
}
```

Evaluation of arithmetic expressions

- Notation can be infix, postfix or prefix.

  Infix: operator is between operands
  A + B

  Postfix : operator follows operands
  AB+

  Prefix: operator precedes operands
  +AB

- Operators in a postfix expression are in correct evaluation order.

Postfix Expressions

Infix	**Postfix**
a + b * c	abc*+

(precedence of * is higher than of +)

a + b * c / d	abc*d/+

(precedence of * and / are same and they are left associative)

- Parentheses override the precedence rules:

(a + b) * c
ab+c*

- More examples

Infix	**Postfix**
(a + b) * (c – d)	ab+cd-*
a – b / (c + d * e)	abcde*+/-
((a + b) * c – (d – e))/(f + g)	ab+c*de - - fg+/

Order of precedence for 5 binary operators:
power (^)
multiplication (*) and division (/)
addition (+) and subtraction (-)

The association is assumed to be left to right except in the case of power where the association is assumed from right to left.
i.e. a + b + c = (a+b)+c = ab+c+
a^b^c = a^(b^c) = abc^^

Converting an Infix Expression to Postfix

```
while there are more characters in the input {
    Read next symbol ch in the given infix expression.
    If ch is an operand put it into the output.
    If ch is an operator (i.e.*,/,+,-, or ^) {
        check the item op at the top of the stack
        while (more items in the stack && precedence(ch) <=
                                                 precedence (op))
        {
            pop op and append it to the output.
            op becomes the next top element
        }
        push ch onto stack
    }
}
```

Evaluating a Postfix Expression

Each operator in a postfix string refers to the previous two operands in the string. Each time we read an operand we push it onto a stack. When we reach an operator its operands will be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

• Implementation of a stack as an array

```
#define maxstack 100

struct stack{
    int items[maxstack];
    int top;
};

int isEmpty(struct stack s){
    return (s.top < 0);
}

int isFull(struct stack s){
    return (s.top >= maxstack-1);
}

void push (struct stack *s, int x){
    if (s->top >= maxstack-1)
        printf("The stack is full.\n");
    else {
        s->top = s->top +1;
        s->items[s->top] = x;
    }
}

int pop (struct stack *s){
    int x;
    if (s->top < 0)
        printf("Stack is empty.\n");
    else{
        x = s->items[s->top];
        s->top = s->top –1;
        return x;
    }
}
```

```c
int main()
{
    struct stack S;
    int c, i;

    S.top = -1;

    while ((c=getchar() )!='\n')
        push(&S, c);

    while (!isEmpty(S))
        printf("%c", pop(&S));

    printf("\n");
}
```