# CS11001/CS11002
# Programming and Data Structures (PDS)
## (Theory: 3-1-0)

## The Stack ADT

- A stack is an ordered list of elements in which elements are always inserted and deleted at one end, say the beginning.
- In the terminology of stacks, this end is called the **top** of the stack, whereas the other end is called the **bottom** of the stack. A
- lso the insertion operation is called **push** and the deletion operation is called **pop**.
- The element at the top of a stack is frequently referred, so we highlight this special form of getElement.

## The Stack ADT operations

- A stack ADT can be specified by some basic operations.
- Once again we assume that we are maintaining a stack of characters.
- In practice, the data type for each element of a stack can be of any data type.
- An element popped out of the stack is always the last element to have been pushed in. Therefore, a stack is often called a **Last-In-First-Out** or a **LIFO** list

## The Stack ADT operations

- S = init();
  - Initialize S to an empty stack.
- isEmpty(S);
  - Returns "true" if and only if the stack S is empty, i.e., contains no elements.
- isFull(S);
  - Returns "true" if and only if the stack S has a bounded size and holds the maximum number of elements it can.

## The Stack ADT operations

- top(S);
  - Return the element at the top of the stack S, or error if the stack is empty.
- S = push(S,ch);
  - Push the character ch at the top of the stack S.
- S = pop(S);
  - Pop an element from the top of the stack S.
- print(S);
  - Print the elements of the stack S from top to bottom.

## Implementations of the stack ADT

```
#include <stdio.h>

#define MAXLEN 100

typedef struct {
   char element[MAXLEN];
   int top;
} stack;
```

## The init and isEmpty function

```
stack init ()
{
   stack S;

   S.top = -1;
   return S;
}
```

```
int isEmpty ( stack S )
{
return (S.top == -1);
}
```

```
int isFull ( stack S )
{
  return (S.top == MAXLEN - 1);
}
```

## The push procedure

```
stack push ( stack S , char ch )
{
   if (isFull(S)) {
      fprintf(stderr, "push: Full stack\n");
      return S;
   }
   ++S.top;
   S.element[S.top] = ch;
   return S;
}
```

## The pop Procedure

```
stack pop ( stack S )
{
   if (isEmpty(S)) {
      fprintf(stderr, "pop: Empty stack\n");
      return S;
   }
   --S.top;
   return S;
}
```

## Printing the stack

```
void print ( stack S )
{
   int i;

   for (i = S.top; i >= 0; --i)
    printf("%c",S.element[i]);
}
```

## The top procedure

```
char top ( stack S )
{
   if (isEmpty(S)) {
      fprintf(stderr, "top: Empty stack\n");
      return '\0';
   }
   return S.element[S.top];
}
```

## The main function

```
int main ()
{
   stack S;

   S = init(); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'d'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'f'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'a'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'x'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
}
```

## Output

- top: Empty stack
- Current stack : with top = .
- Current stack : d with top = d.
- Current stack : fd with top = f.
- Current stack : afd with top = a.
- Current stack : fd with top = f.
- Current stack : xfd with top = x.
- Current stack : fd with top = f.
- Current stack : d with top = d.
- top: Empty stack
- Current stack : with top = .
- pop: Empty stack
- top: Empty stack
- Current stack : with top = .

## An Application of Stack

- It can be used to ensure that the parenthesis of a mathematical expression is well-formed:
  - it has an equal number of right and left parenthesis
  - every right parenthesis is preceded by a matching left parenthesis

# A sketch of a program

```
valid=true;
s=init();
while(the entire string is not read){
  read the next symbol (sym) of the stringg;
  if(sym==‘(‘||sym==‘[‘||sym==‘{‘)
      push(s,sym);
 if(sym==‘)‘||sym==‘]‘||sym==‘}‘)
     if(isEmpty(s)) valid=false;
else{ i=pop(s); //i has the top element
         if(i is not the matching opener of sym)
              valid=false;
      }
}//end reading the string
if(!isEmpty(s)) valid=false;
```

# Exercise

- Complete the program using the previous definitions.

## An interesting definition for stack element

- The stack element could be of any data type

```
struct stackelement{
  int etype;
  union{
      int ival;
      float fval;
      char *sval;
    }element;
 };
struct stack{
  int top;
  struct stackelement items[100];
}
```

## Printing the top element of the stack

```
#define INGR 1
#define FLT 2
#define STRING 3
struct stack S;
struct stackelement se;
se=s.items[s.top];
switch(se.etype){
   case INGR: printf("%d\n",se.ival);
   case FLT: printf("%f\n",se.fval);
   case STRING: printf("%s\n",se.sval);
}
```