COMP2321 — DATA STRUCTURES

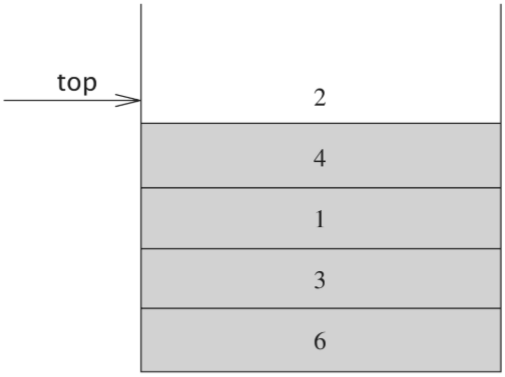
Stacks

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

جامعة بيرزيت
BIRZEIT UNIVERSITY

Stacks

- A stacks is a list with the restriction that insertion and deletion can be performed only at one position: The end of the list (called TOP)
- Meaning it is a list with only one end accessible

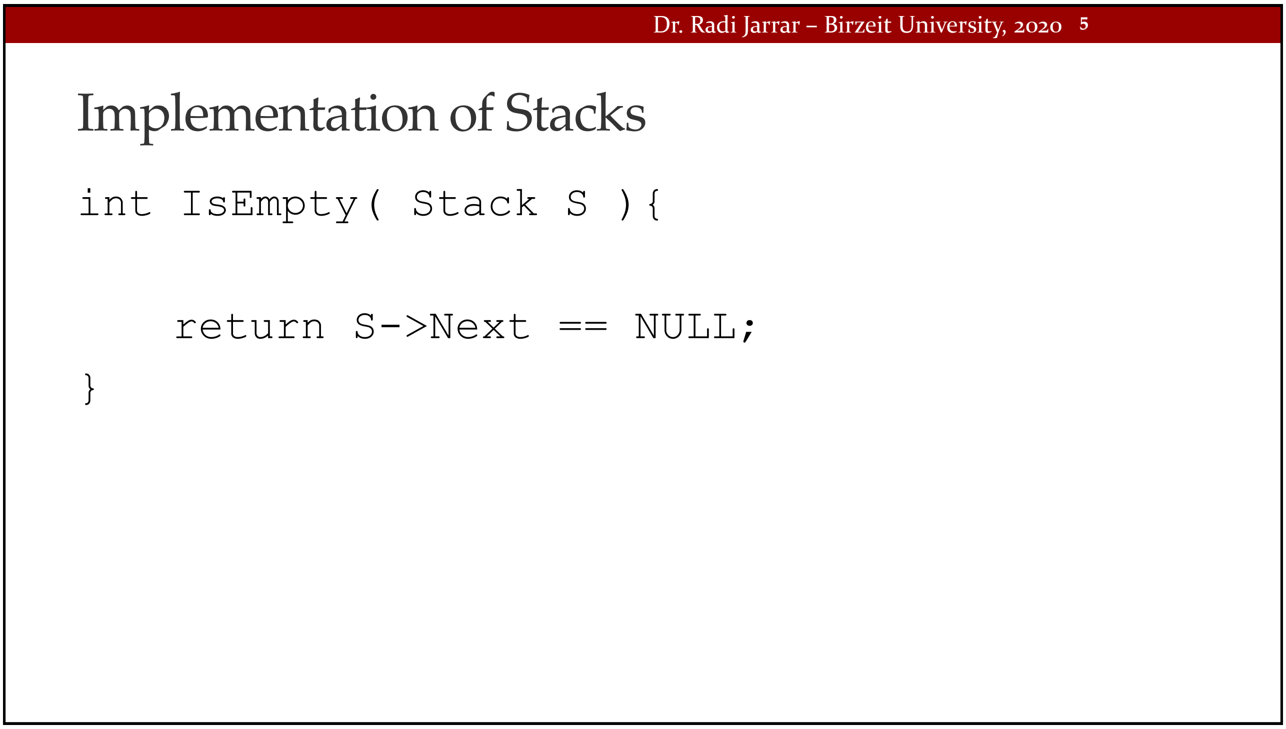


Stacks

- The fundamental operations of the stack are
  - *Push*: equivalent to insert. Inserts element at top
  - *Pop*: return and delete the most recently added element
  - *Top*: examines (returns) the most recently added element
- Pop or top on empty stack is generally considered an error
- Stacks are known as LIFO (Last In First Out) lists

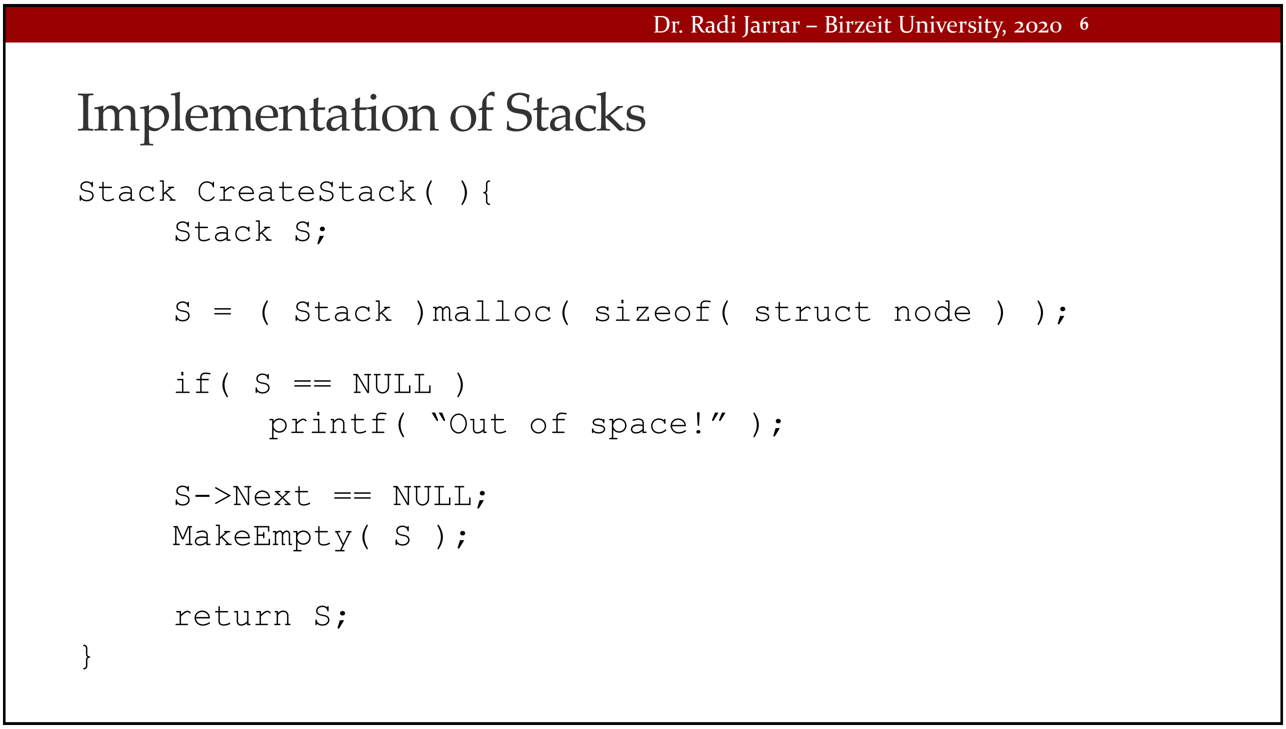
Implementation of Stacks

```
//struct node

typedef struct node* PtrToNode;
typedef PtrToNode Stack;

struct node{
    int Element;
    PtrToNode Next;
};
```



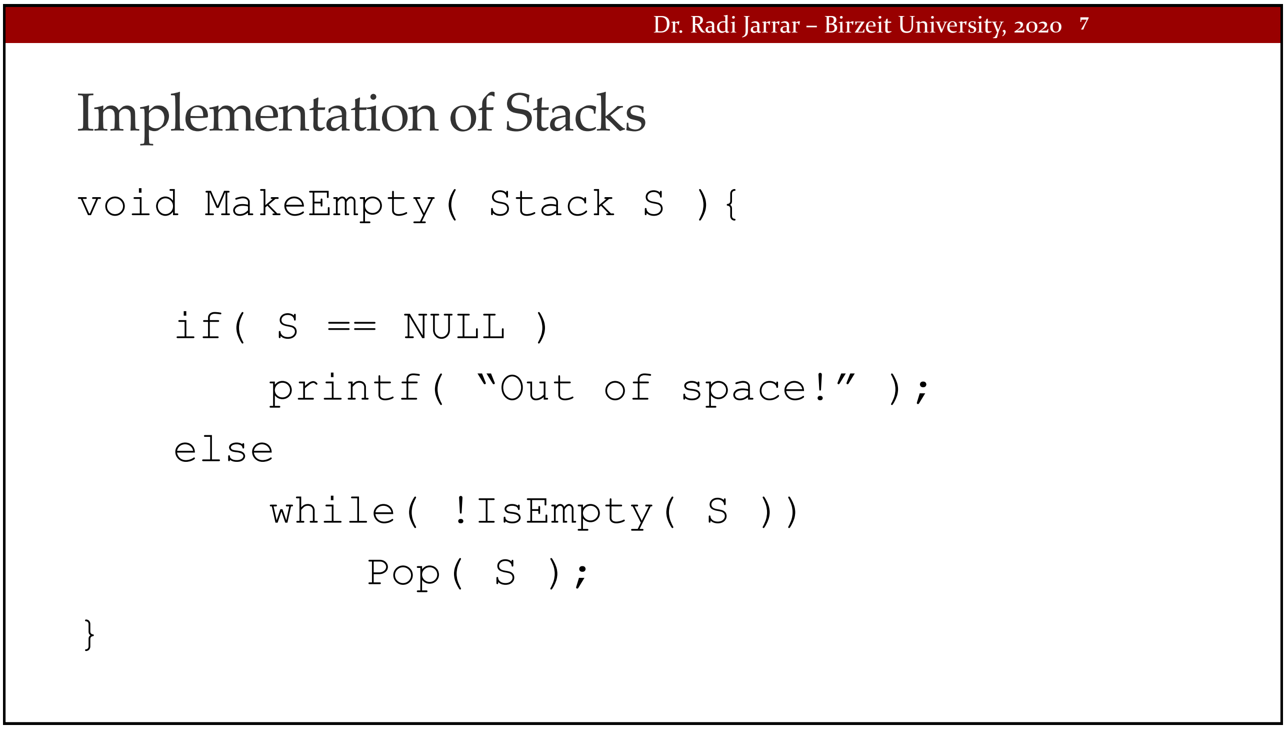
Implementation of Stacks

```
int IsEmpty( Stack S ){

    return S->Next == NULL;
}
```



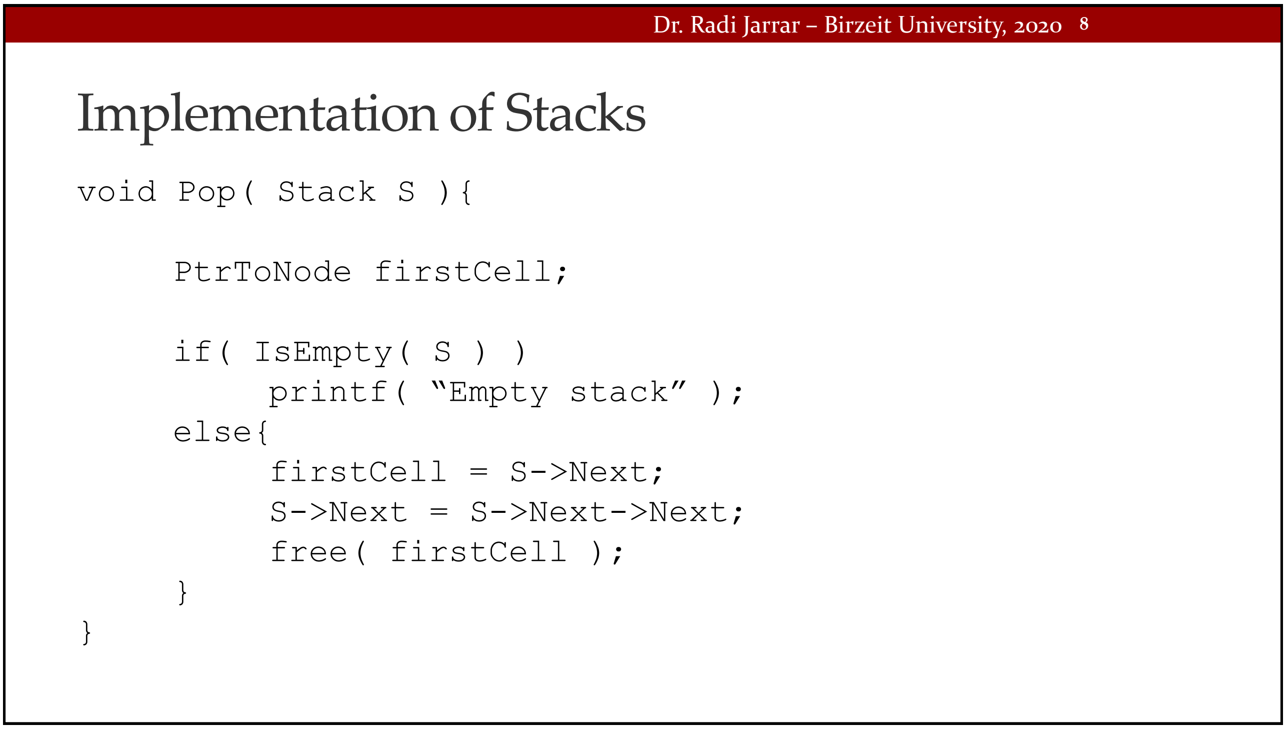
Implementation of Stacks

```
Stack CreateStack( ){
     Stack S;

     S = ( Stack )malloc( sizeof( struct node ) );

     if( S == NULL )
          printf( “Out of space!” );

     S->Next == NULL;
     MakeEmpty( S );

     return S;
}
```



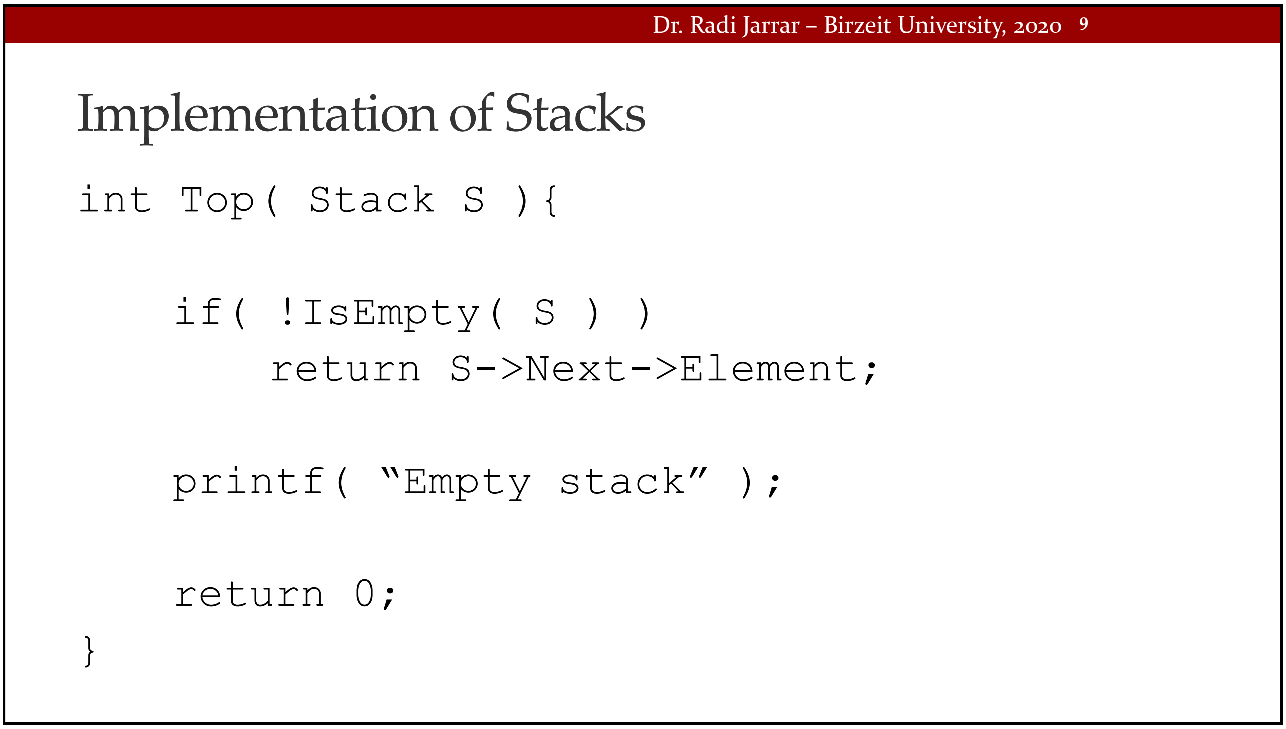
Implementation of Stacks

```
void MakeEmpty( Stack S ){

    if( S == NULL )
        printf( “Out of space!” );
    else
        while( !IsEmpty( S ))
            Pop( S );
}
```



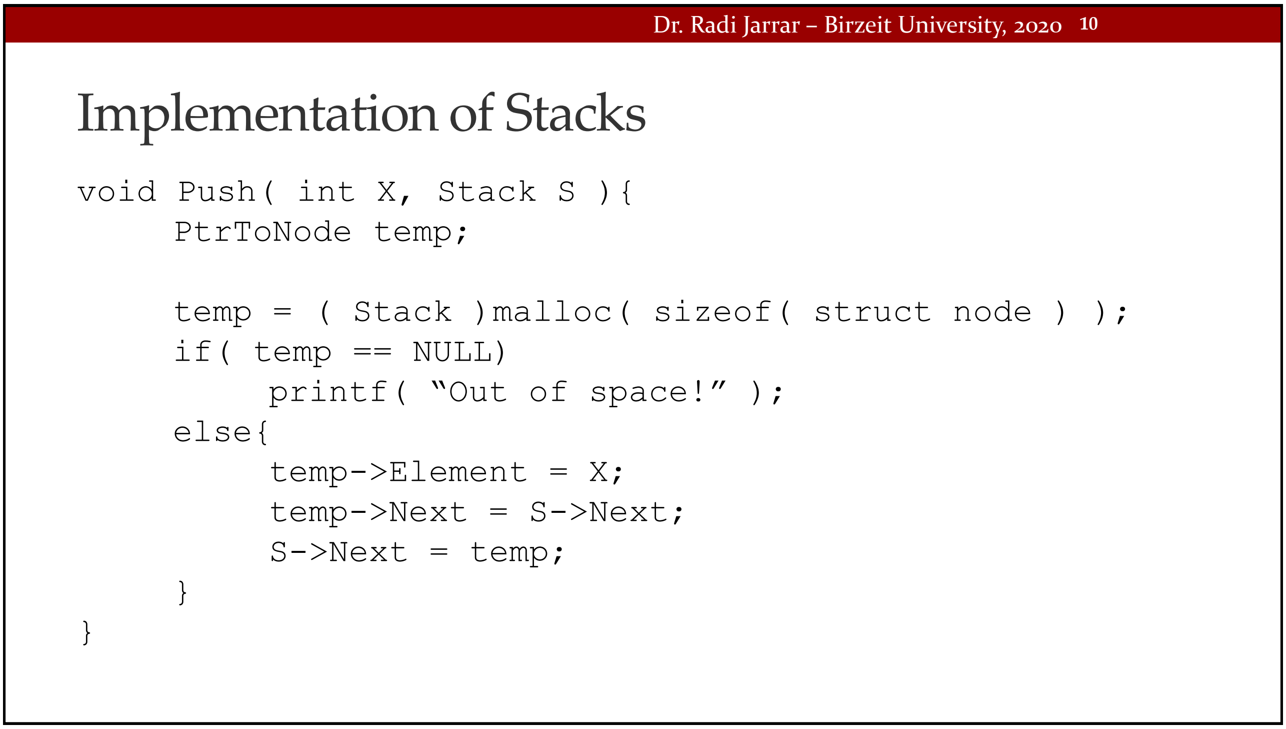
Implementation of Stacks

```
void Pop( Stack S ){

    PtrToNode firstCell;

    if( IsEmpty( S ) )
        printf( “Empty stack” );
    else{
        firstCell = S->Next;
        S->Next = S->Next->Next;
        free( firstCell );
    }
}
```



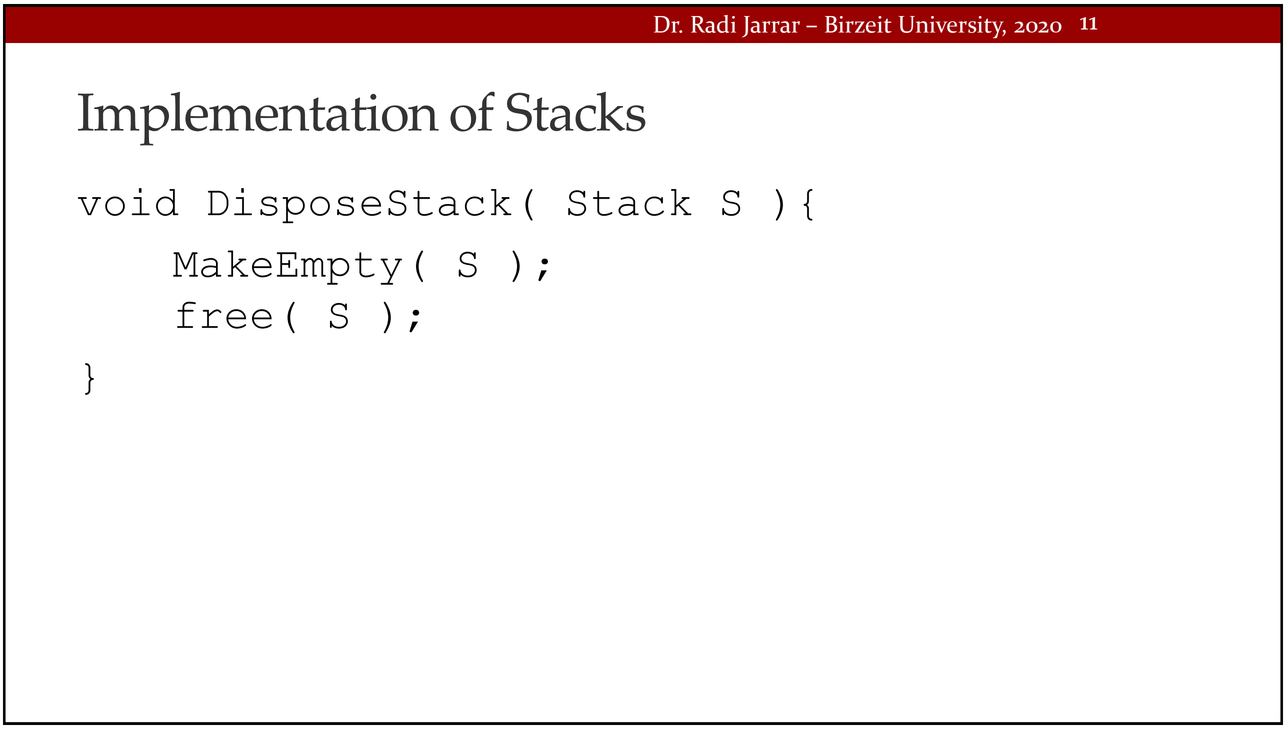
Implementation of Stacks

```
int Top( Stack S ){

    if( !IsEmpty( S ) )
        return S->Next->Element;

    printf( “Empty stack” );

    return 0;
}
```



Implementation of Stacks

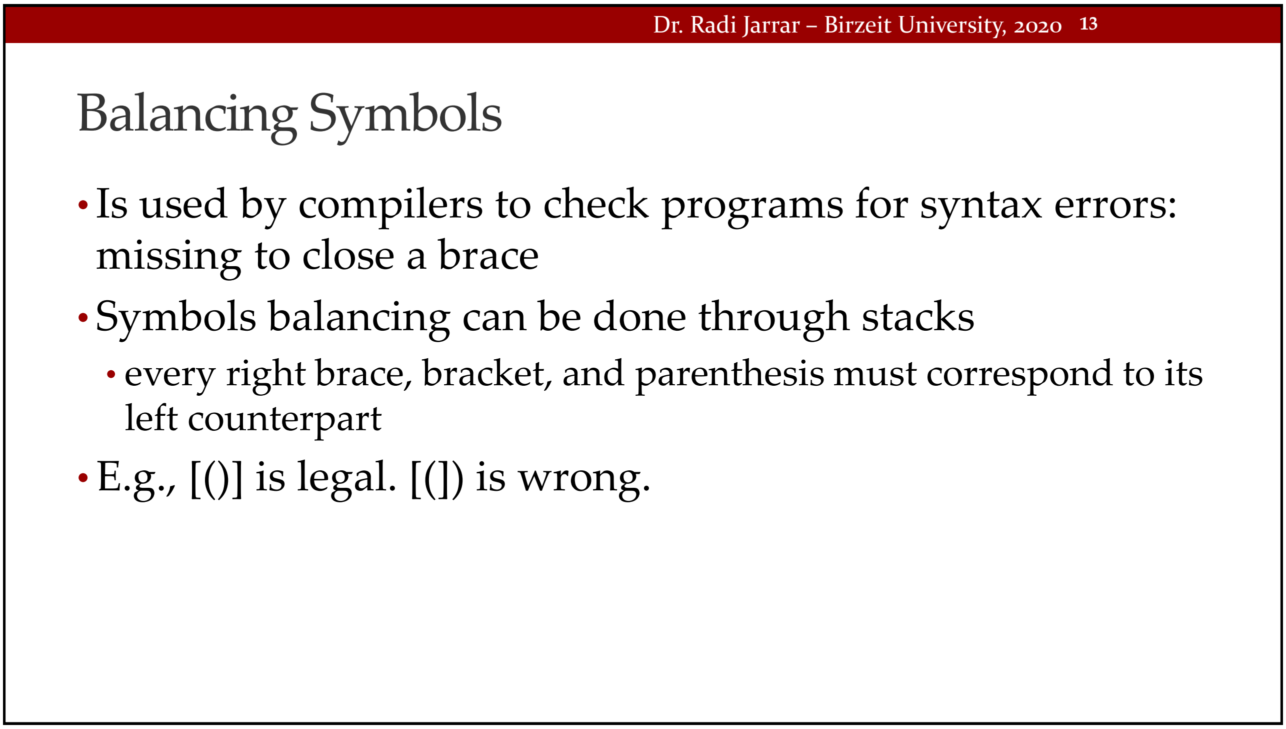
```
void Push( int X, Stack S ){
     PtrToNode temp;

     temp = ( Stack )malloc( sizeof( struct node ) );
     if( temp == NULL)
          printf( “Out of space!” );
     else{
          temp->Element = X;
          temp->Next = S->Next;
          S->Next = temp;
     }
}
```



Implementation of Stacks

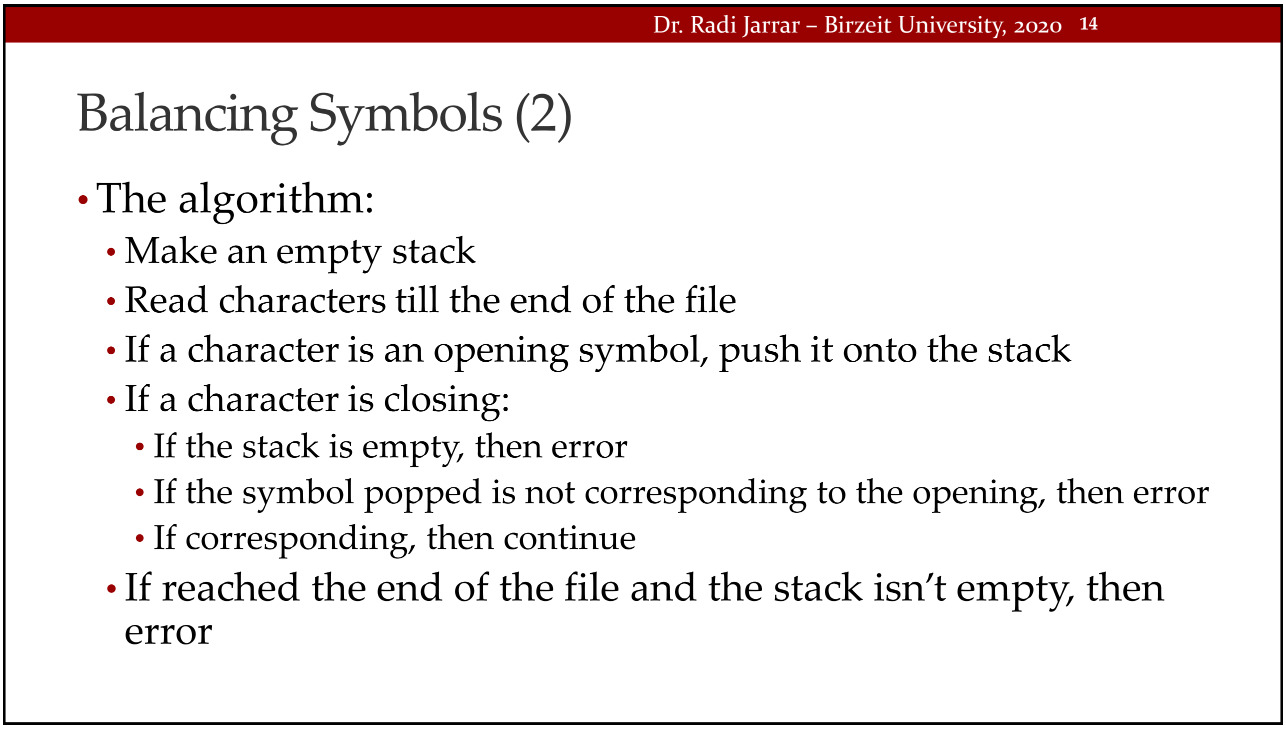
```
void DisposeStack( Stack S ){
    MakeEmpty( S );
    free( S );
}
```

APPLICATIONS OF STACKS



Balancing Symbols

- Is used by compilers to check programs for syntax errors: missing to close a brace
- Symbols balancing can be done through stacks
    - every right brace, bracket, and parenthesis must correspond to its left counterpart
- E.g., [()] is legal. [(]) is wrong.



Balancing Symbols (2)

- The algorithm:
    - Make an empty stack
    - Read characters till the end of the file
    - If a character is an opening symbol, push it onto the stack
    - If a character is closing:
        - If the stack is empty, then error
        - If the symbol popped is not corresponding to the opening, then error
        - If corresponding, then continue
    - If reached the end of the file and the stack isn’t empty, then error

Balancing Symbols (2)

- E.g., check if the following is correct: [(< ( ]))]

- [([( )])]

Postfix evaluation

- Stacks are also used to evaluate postfix expressions and to convert infix into postfix
  - Infix expression is on the form AOB, where O is operator (+, -, *, /, %)
  - Postfix expression is on the form ABO
  - Prefix expression is on the form OAB
- Postfix expressions are easy to compute using stacks. That is the reason why we convert infix into postfix

Postfix evaluation (2)

- Algorithm
  - Create an empty stack
  - Read the expression: when a number is seen, push into the stack
  - When operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack
  - The result is pushed back into the stack

Postfix evaluation (3)

- E.g., `1 2 + 4 × 5 + 3 −`

Postfix evaluation (3)

- E.g., `10 2 8 × 3 −`

Postfix evaluation (5)

- E.g., `1 2 + 3 × 6 + 2 3 + /`

Infix to Postfix conversion

- Algorithm
  - When a number is read, place it into the output
  - When an operation is read, push into a stack
  - For operation O, if the top of the stack is lower priority, then insert it. Else pop the top elements in the stack until there is no operator having higher priority than O, then push O into the stack

Infix to Postfix conversion (2)

- **Notes**
- High-precedence operation can be on top of low. The opposite is not true
- Empty stack for operations
- Empty list for output
- When reaching an opening parenthesis, treat the stack as empty & execute it first until you reach the closing parenthesis. Then pop the remaining elements

Infix to Postfix conversion (3)

- E.g., `A × B^C + D`

Infix to Postfix conversion (4)

- E.g., `(A + B) × (C + D)`

Infix to Postfix conversion (5)

- E.g., `A × (B + C × D) + E`

Infix to Postfix conversion (6)

- E.g., `A × (B + C / D)`