Stack "Abstract Data Type"

- **A stack object maintains a set of values of some type.**
- **The distinguishing characteristic of a stack are the way in which elements may be inserted into or deleted from the stack**
- **Stack: The last element inserted will be the first element deleted.**
- **Model: A cafeteria rack of plates**
- **Thus the insert operation is called a push**
- **The delete operator is called a pop.**
- **The next element to be deleted from a stack is said to be at the top of the stack**

“Generic” Element Type

- The ***operations*** on a Stack ***do not depend on the <u>type</u> of elements*** that the structure contains.
- However, the type of the element determines the type of the Stack as a C variable – **intStack**, **charStack**, etc.
- Rather than rewrite the same code again and again with only the element type declarations changed, we write our stack code assuming a generic element type.
- An appropriate ***typedef*** for the generic type must be included before the code for the stack in order for the code to compilable.

Stacks

- **In order to be considered a stack, an ADT must follow the LIFO insertion-deletion discipline described above.**
- **In order to be useful, certain other operations must be provided for *initialization*, *testing* for being *empty* and for being *full*.**
- **Another useful operation is one to *clear* the stack in a single step.**
- **Other operations are optional, but almost always include a function to *retrieve* the *top* element value without removing it from the stack.**

Implementing the Stack

- The header file for the Stack ADT provides the interface of the ADT.

- It should provide all the information needed to use stacks in a program or module

- Implementation will be *private*.
  - Other parts of the program will have no knowledge of how the stack is implemented.
  - Access the stack only through the functions defined in stack.h.

- Principle of *encapsulation*.
  - Information hiding.

Stack Header File

- **Next we show the header files for the Stack ADT.**
- **#include “stackDefs.h”**
  (provides the implementation typedefs for the Stack ADT)
-
```
void InitStack(Stack * s);
/*  pre: None.
   post: The stack s has been
         initialized to be empty.
*/
```

Stack Header File

- `int Push( ItemType item, Stack * s );`

```
/* pre:  The stack exists and is not full.
   post: The argument item has been stored at
         the top of the stack.
*/
```

- `int Pop( ItemType *item, Stack * s );`

```
/* pre:  The stack exists and is not empty.
   post: The top of the stack has been removed

         and returned in *item.
*/
```

- `int StackTop(ItemType *item, const Stack *s);`

```
/* pre:  The stack exists and is not empty.
   post: The top of the stack is returned in
         *item without being removed; the
         stack s is unchanged.
*/
```

Stack Header File

- ```
  int StackEmpty( const Stack * s );
  /* pre:  The stack exists and has been initialized.
     post: Returns TRUE if the stack is empty, FALSE
           otherwise.
  */
  ```
- ```
  int   StackFull( const Stack * s );
  /* pre:  The stack exists and has been initialized.
     post: Returns TRUE if the stack is full, FALSE
           otherwise.
  */
  ```
- ```
  void    ClearStack( Stack * s );
  /* pre:  The stack exists and has been initialized.
     post: All entries in the stack have been
           deleted; the stack is empty.
  */
  ```
  ```

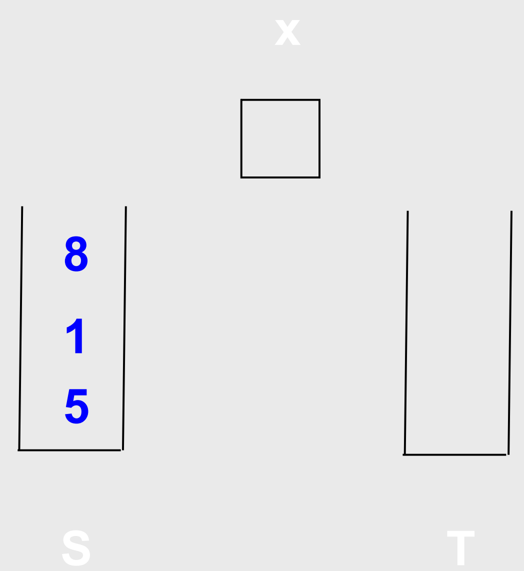
Using the Stack ADT

- Next we show how to perform actions on a stack without knowing how the Stack is implemented (or even knowing the element type).
- Our task is to print the elements of the stack in order from bottom to top, i.e., the order in which there were inserted.
- Since we do not know the element type, we assume that we have available a function *PrintItemType* that prints a value of type *ItemType*.
- We will use a local variable of type Stack in our function

The PrintBottomToTop Function

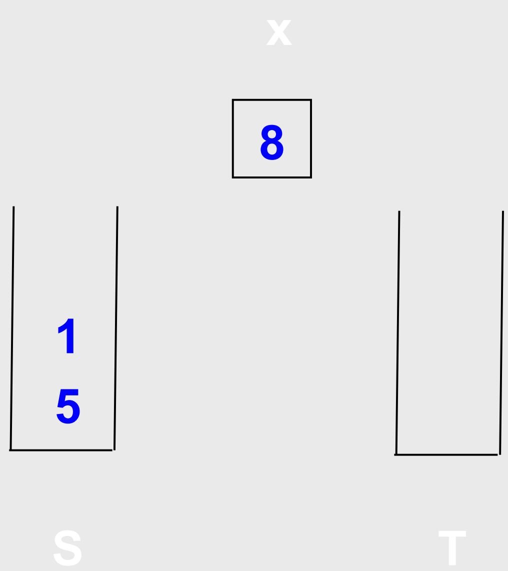
- ```
  void  printBottomToTop( Stack *S )
  {
    Stack T;
    ItemType x;
    InitStack(&T);
    while (!StackEmpty( S ))
    {
      Pop(&x, S);
      Push(x, &T);
    }
  ```
- ```
  while (!StackEmpty( &T ))
    {
      Pop(&x, &T);
      PrintItemType(x);
      Push(x, S);
      printf("\n");
    }
  }
  ```
- We next illustrate this algorithm with a stack of 3 integers

PrintBottomToTop() Execution Trace



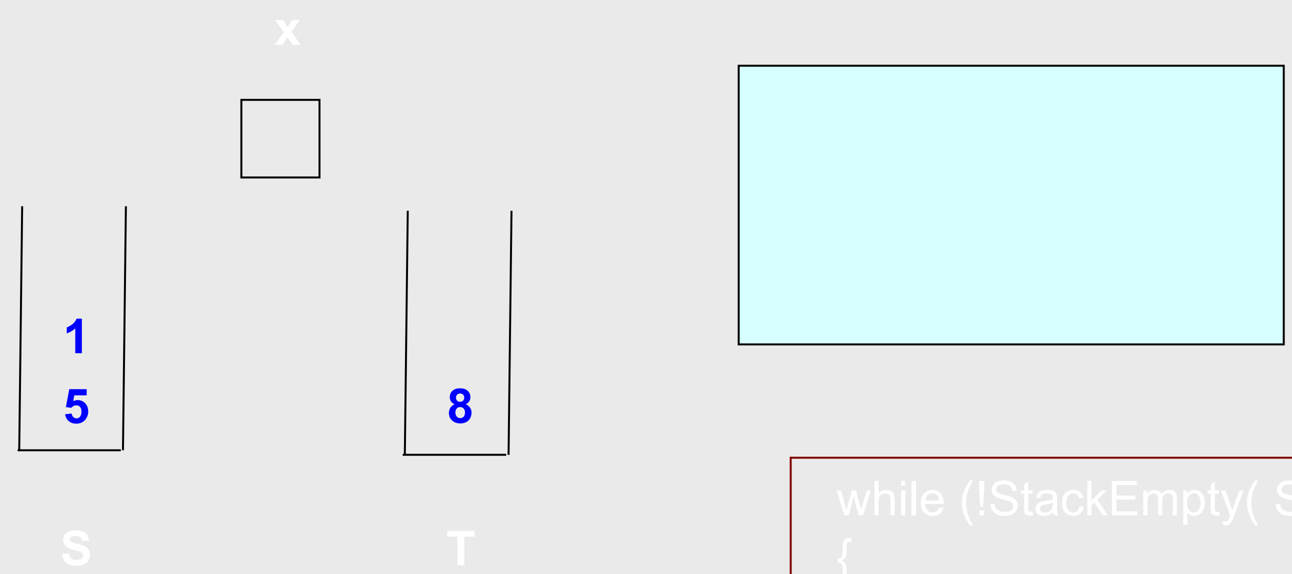

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
 }
```

PrintBottomToTop() Execution Trace



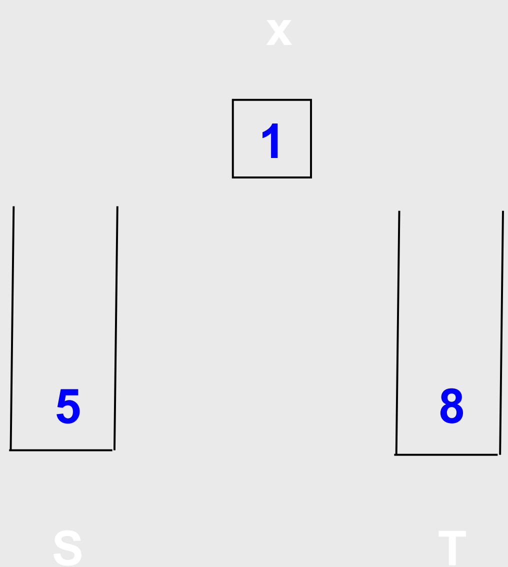

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace



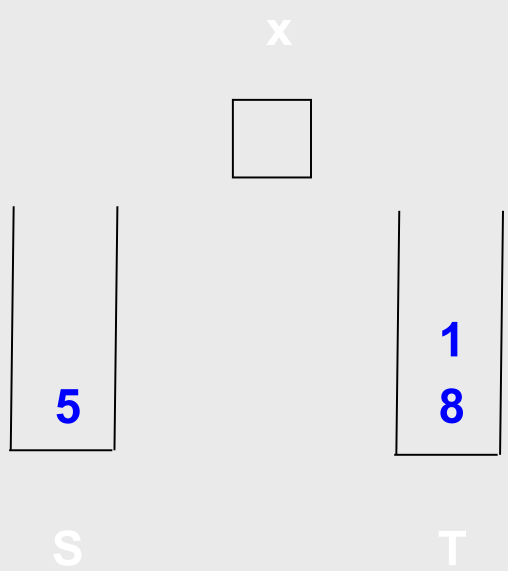

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace




```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

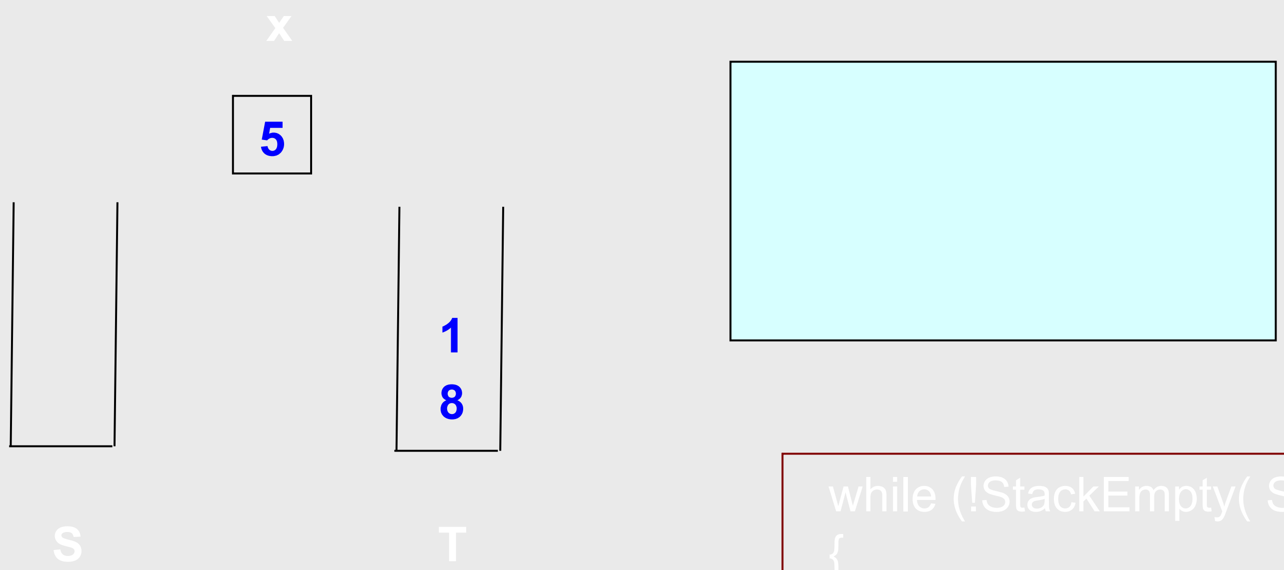
PrintBottomToTop() Execution Trace




```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);

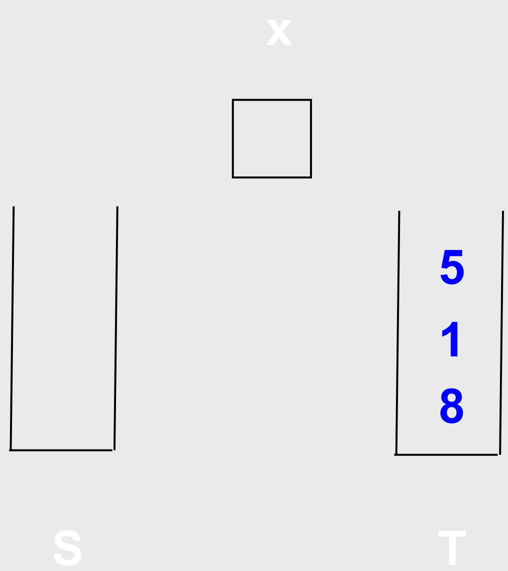
}
```

PrintBottomToTop() Execution Trace



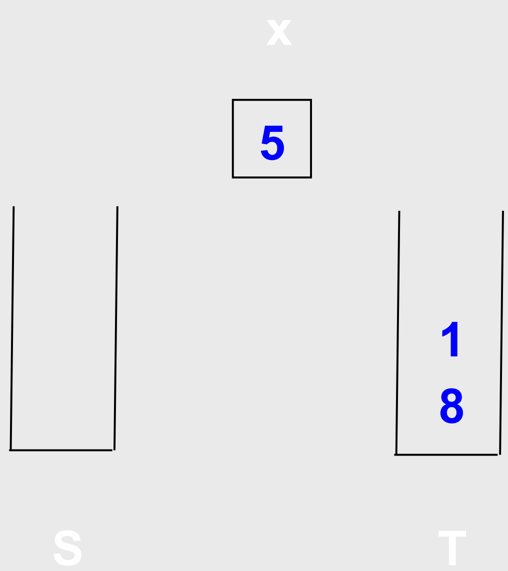

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace



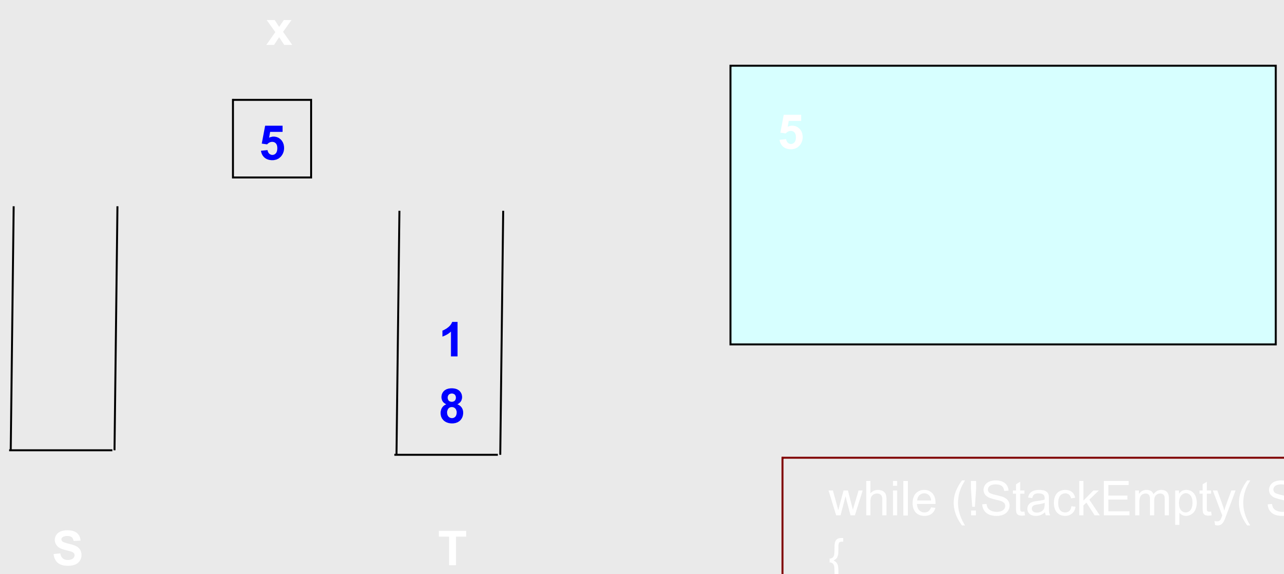

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace



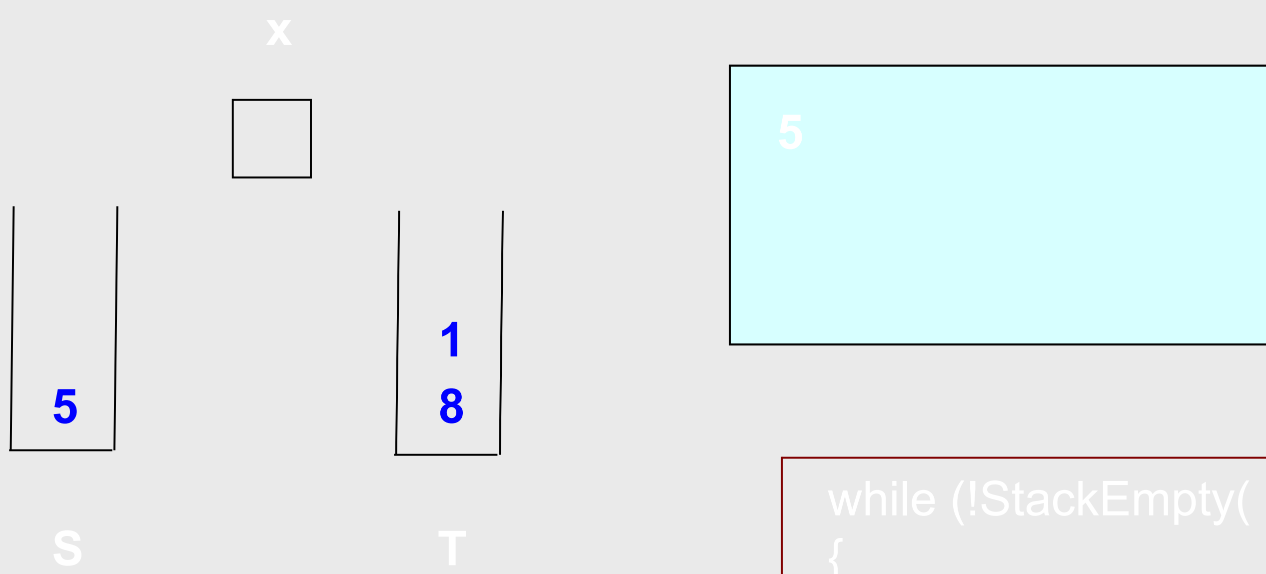

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace



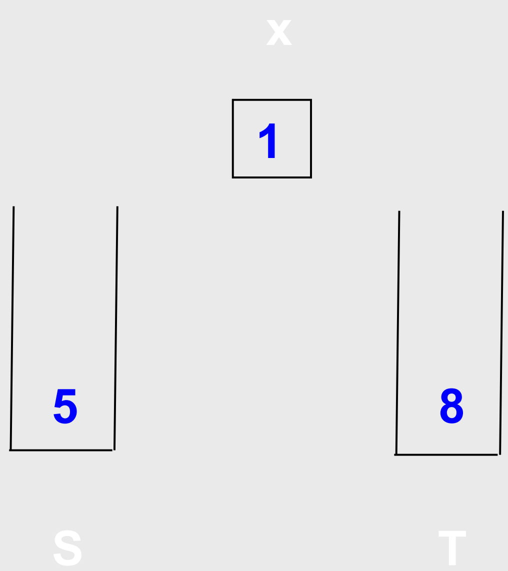

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace




```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

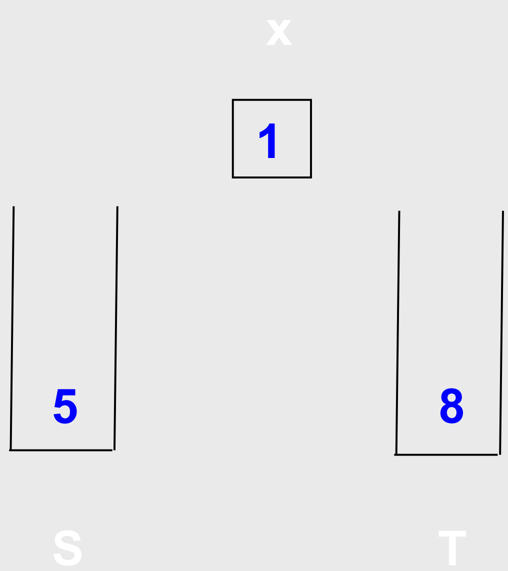
PrintBottomToTop() Execution Trace



5

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

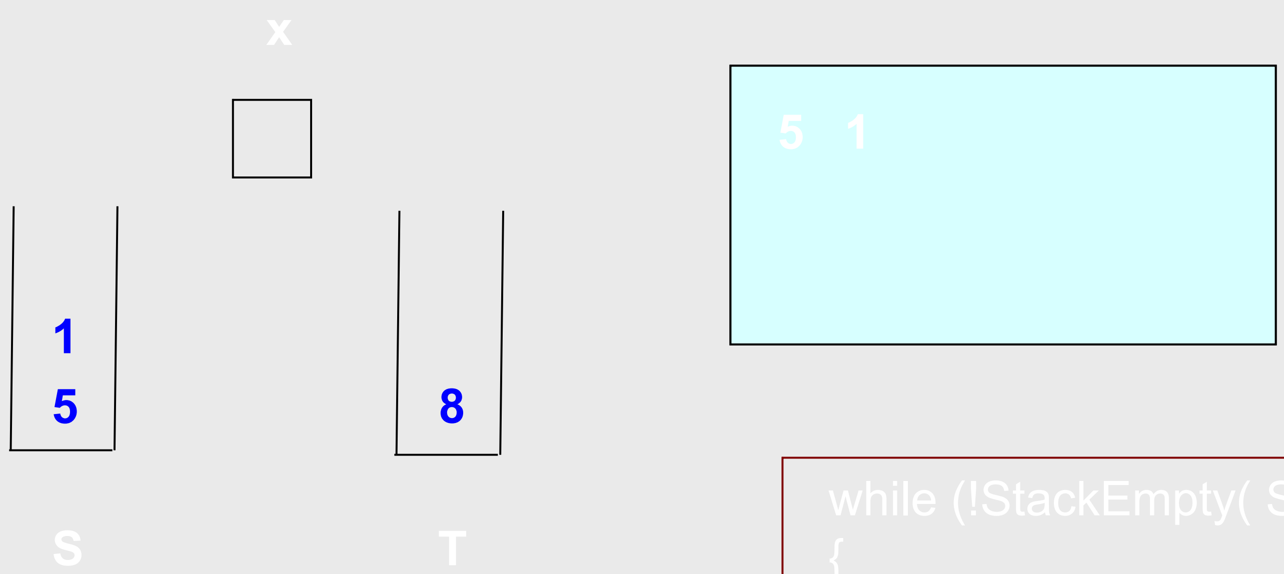
PrintBottomToTop() Execution Trace



5 1

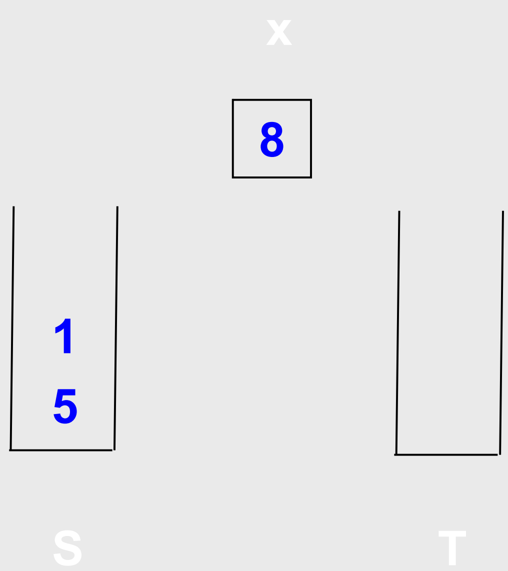
```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

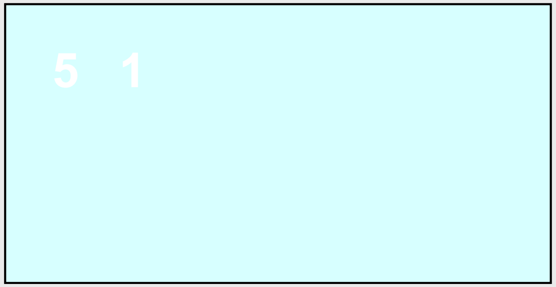
PrintBottomToTop() Execution Trace




```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

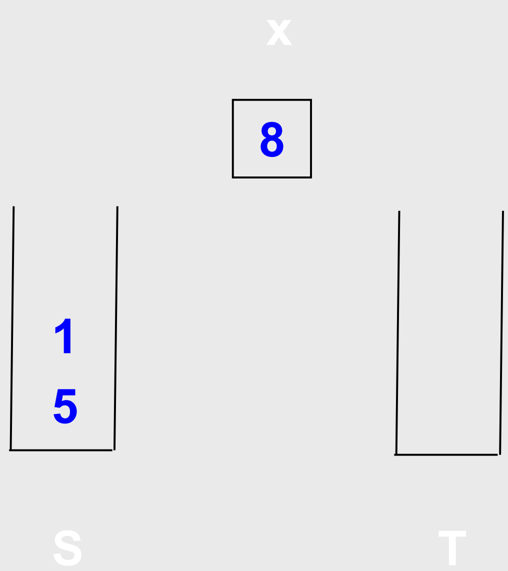
PrintBottomToTop() Execution Trace






```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

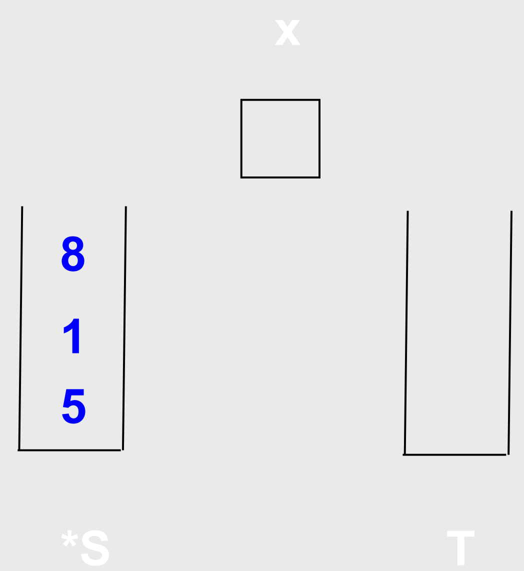
PrintBottomToTop() Execution Trace



5 1 8

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

PrintBottomToTop() Execution Trace



5 1 8

```
while (!StackEmpty( S ))
{
  Pop(&x, S);
  Push(x, &T);
}
while (!StackEmpty( &T ))
{
  Pop(&x, &T);
  PrintItemType(x);
  Push(x, S);
}
```

S has been restored to its original state and its contents have been printed in bottom-to-top order