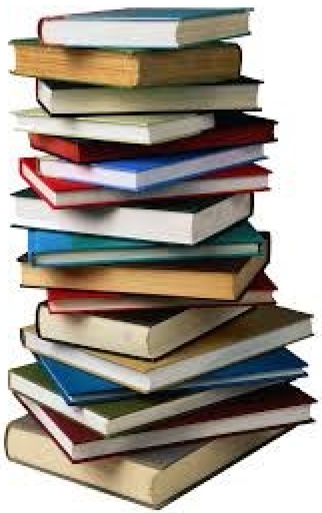
Stacks (Στοίβες)

Manolis Koubarakis

Stacks and Queues

- **Linear data structures** are collections of components arranged in a straight line.
- If we restrict the growth of a linear data structure so that new components can be added and removed only at one end, we have a **stack (στοίβα).**
- If new components can be added at one end but removal of components must take place at the opposite end, we have a **queue (ουρά αναμονής).**

Examples of Stacks in Real Life



Stacks in Computer Science

- Stacks are used in many areas of Computer Science:
  - Parsing algorithms
  - Pushdown automata
  - Expression evaluation algorithms
  - Backtracking algorithms
  - Activation records in run-time stack

Stacks

- Stacks are sometimes called **LIFO lists** where LIFO stands for “last-in, first-out”.
- When we add a new object to the top of a stack, this is called “**pushing**”.
- When we remove an object from the top of a stack, this is called “**popping**”.
- Pushing and popping are inverse operations.

Sequences

- A **finite-length sequence (ακολουθία πεπερασμένου μήκους)** $S=(s_1, s_2, \ldots, s_n)$ is just an ordered arrangement of finitely many components $s_1, s_2, \ldots, s_n$.
- The **length** of a sequence is the number of its components.
- There is a special sequence with length 0 called the **empty sequence**.

An Abstract Data Type for Stacks

- A **stack** *S* of items of type *T* is a sequence of items of type *T* on which the following operations can be defined:
  1. Initialize the stack *S* to be the **empty stack**.
  2. Determine whether or not the stack *S* is **empty**.
  3. Determine whether or not the stack *S* is **full**.
  4. **Push** a new item onto the top of stack *S*.
  5. If *S* is nonempty, **pop** an item from the top of stack *S*.

An Interface for Stacks

- Using **separately compiled C files**, we can define **C modules** that specify the underlying representation for stacks and implement the abstract stack operations.

The Stack ADT Interface

```
/* This is the file StackInterface.h
*/
#include "StackTypes.h"

void InitializeStack(Stack *);
int Empty(Stack *);
int Full(Stack *);
void Push(ItemType, Stack *);
void Pop(Stack *, ItemType *);
```

Using the Stack ADT to Check for Balanced Parentheses

- The first application of the Stack ADT that we will study involves determining whether parentheses and brackets balance properly in algebraic expressions.

- Example:

$$\{a^2 - [(b+c)^2 - (d+e)^2] * [\sin(x-y)]\} - \cos(x+y)$$

- This expression contains parentheses, square brackets, and braces in balanced pairs according to the pattern

$$\{[()()][()]\}()$$

The Algorithm

- We can start with an **empty stack** and scan a string representing the algebraic expression from left to right.
- Whenever we encounter a left parenthesis (, a left bracket [or a left brace {, we **push** it onto the stack.
- Whenever we encounter a right parenthesis), a right bracket] or a right brace }, we **pop** the top item off the stack and check to see that its type matches the type of right parenthesis, bracket or brace encountered.
- If the stack is **empty** by the time we get to the end of the expression string and if all pairs of matched parentheses were of the same type, the expression has properly balanced parentheses. Otherwise, the parentheses are not balanced properly.

The Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char InputExpression[100];  // Fixed-size array for
input expression

int Match(char c, char d) {
    switch (c) {
        case '(': return d == ')';
        case '[': return d == ']';
        case '{': return d == '}';
        default: return 0;
    }
}
```

The Program (cont'd)

```
void ParenMatch(void) {
  int n, i = 0;
  char c, d;
  Stack ParenStack;

  InitializeStack(&ParenStack);

  n = strlen(InputExpression);
  while (i < n) {
    d = InputExpression[i];

    // If opening bracket, push onto stack
    if (d == '(' || d == '[' || d == '{') {
      Push(d, &ParenStack);
    }
    // If closing bracket, check stack
    else if (d == ')' || d == ']' || d == '}') {
      if (Empty(&ParenStack)) {
        printf("More right parentheses than left parentheses\n");
        return;
      }
      Pop(&ParenStack, &c);
      if (!Match(c, d)) {
         printf("Mismatched Parentheses: %c and %c\n", c, d);
         return;
      }
    }
    ++i;
  }

  // Final stack check
  if (Empty(&ParenStack)) {
    printf("Parentheses are balanced properly\n");
  } else {
    printf("More left parentheses than right parentheses\n");
  }
}
```

The Program (cont'd)

```
int main(void) {
   printf("Give Input Expression without blanks: ");
   scanf("%99s", InputExpression); // Prevent buffer
overflow
   ParenMatch();
   return 0;
}
```

Infix and Postfix Notation

- Expressions are usually written in **infix** notation **(ενθεματικό συμβολισμό)** e.g., *(a+b)*2-c*. Parentheses are used to denote the order of operation.
- **Postfix (μεταθεματικές)** expressions are used to specify algebraic operations using a parentheses free notation. For example, *ab+2*c-.*
- The postfix notation *L R op* corresponds to the infix notation *L op R*.

Examples

Infix	Postfix
(a + b)	a b +
(x – y – z)	x y – z –
(x – y – z)/(u + v)	x y – z – u v + /
(a^2 + b^2) * (m – n)	a 2 ^ b 2 ^ + m n – *

Prefix Notation

- There is also **prefix (or Polish) notation (προθεματικός συμβολισμός)** in which the operator precedes the operands.
- **Example**: `+ 3 * 2 5` is the prefix form of `(2 * 5) + 3`

- Prefix and postfix notations **do not need parentheses** for denoting the order of operations and there are **no precedence rules** to learn.
- Postfix notation is sometimes called **reverse Polish** for obvious reasons.

Using the Stack ADT to Evaluate Postfix Expressions

- To evaluate a postfix expression *P*, you **scan from left to right**.
- When you encounter an **operand (όρισμα)** *X*, you **push** it onto an evaluation stack *S*.
- When you encounter an **operator (τελεστή)** *op*, you **pop** the topmost operand stacked on *S* into a variable *R* (which denotes the right operand), then you **pop** another topmost operand stacked on *S* onto a variable *L* (which denotes the left operand).
- Finally, you **perform the operation** *op* on *L* and *R*, getting the value of the expression *L op R*, and you **push** the value back onto the stack *S*.
- When you finish scanning *P*, the value of *P* is the only item remaining on the stack *S*.

Example

- Let us see slides 4 to 14 from the following nice presentation:
  - http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial2.pdf

The Program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "StackInterface1.h"

Stack EvalStack;
char PostfixString[20];

void InterpretPostfix(void)
{
    float LeftOperand, RightOperand, Result;
    int i;

    InitializeStack(&EvalStack);
```

The Program (cont’d)

```
for (i = 0; i < strlen(PostfixString); ++i) {
  if (isdigit(PostfixString[i])) {  // Check if character is a digit
     Push((float)(PostfixString[i] - '0'), &EvalStack); // Convert char to float
  }
  else if (PostfixString[i] == '+' || PostfixString[i] == '-' || PostfixString[i] == '*' ||
            PostfixString[i] == '/' || PostfixString[i] == '^')  {

     Pop(&EvalStack, &RightOperand);
     Pop(&EvalStack, &LeftOperand);

     switch (PostfixString[i]) {
        case '+': Push(LeftOperand + RightOperand, &EvalStack); break;
        case '-': Push(LeftOperand - RightOperand, &EvalStack); break;
        case '*': Push(LeftOperand * RightOperand, &EvalStack); break;
        case '/': Push(LeftOperand / RightOperand, &EvalStack); break;
        case '^': Push(pow(LeftOperand, RightOperand), &EvalStack); break;
        default: break;
     }
   }
  }

  Pop(&EvalStack, &Result);
  printf("Value of postfix expression = %f\n", Result);
}
```

Notes

- The function `isdigit` takes as input a `char` or an `int`. If it is a char, it is implicitly transformed into an int before checking whether it is a digit.
- The expression `PostfixString[i] - '0'` is used to transform a character (`'0' – '9'`) which is in the i-th position of the postfix string into its `int` value.

Understanding Characters as ASCII values

- In C, characters are stored as ASCII values (integer codes).
- For example, `‘0’` is stored as integer `48`, `‘1’` as `49`, …, `‘9’` as `57`.
- Therefore, for example,

```
ASCII(‘5’)-ASCII(‘0’)=53-48=5.
```

- This is what the expression

```
PostfixString[i] - ‘0’
```

does.

Notes (cont'd)

- The function `strlen` takes as argument a pointer to a null-terminated character string (`const char *`).

- The function `strlen` cannot directly take a character array as argument. We need to pass the name of the array, which decays to a pointer to the first element (the first character) automatically.

The Program (cont'd)

```
int main(void) {
    printf("Give input postfix string without
blanks: ");
    scanf("%19s", PostfixString);  // Read input
safely
    InterpretPostfix();
    return 0;
}
```

Implementing the Stack ADT

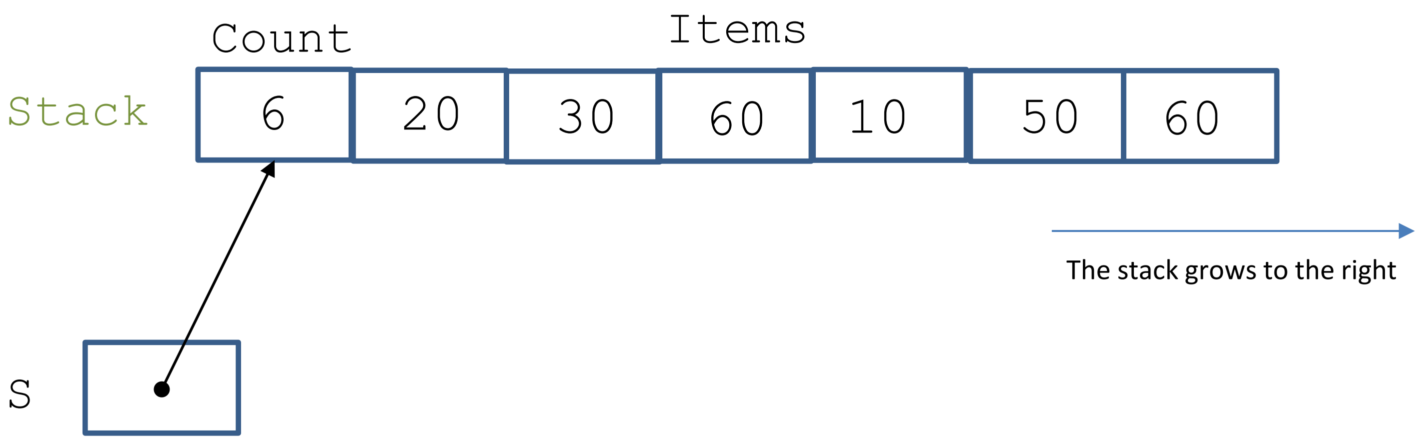
- We will present two implementations of the stack ADT based on:
  - arrays (sequential representation)
  - linked lists (linked representation)

- Both implementations can be used to realize the two applications we presented earlier.

The Implementation Based on Arrays

```
/* This is the file StackTypes.h   */

#define MAXSTACKSIZE 100

typedef char ItemType;
/* char is the type for our first application */
/* float is the type for our second application */

typedef struct{
           int Count;
           ItemType Items[MAXSTACKSIZE];
        } Stack;
```

Example



- We assume that **the top of the stack is the last position of the array.**
- `Count` tells us how many elements we have in the stack. It also contains the **index of the element after the one at the top of the stack**.
- `S` is a pointer to a stack that we will use as parameter in the functions that will implement the stack operations (see the following slides). `S` will correspond to the argument stack that we will use when calling these functions from our main program (e.g., the stack `EvalStack` in the previous example).

The Implementation Based on Arrays (cont'd)

```
/* This is the file StackImplementation.c */

#include <stdio.h>
#include <stdlib.h>
#include "StackInterface.h"

void InitializeStack(Stack *S)
{
   S->Count=0;
}

int Empty(Stack *S)
{
   return (S->Count == 0);
}
```

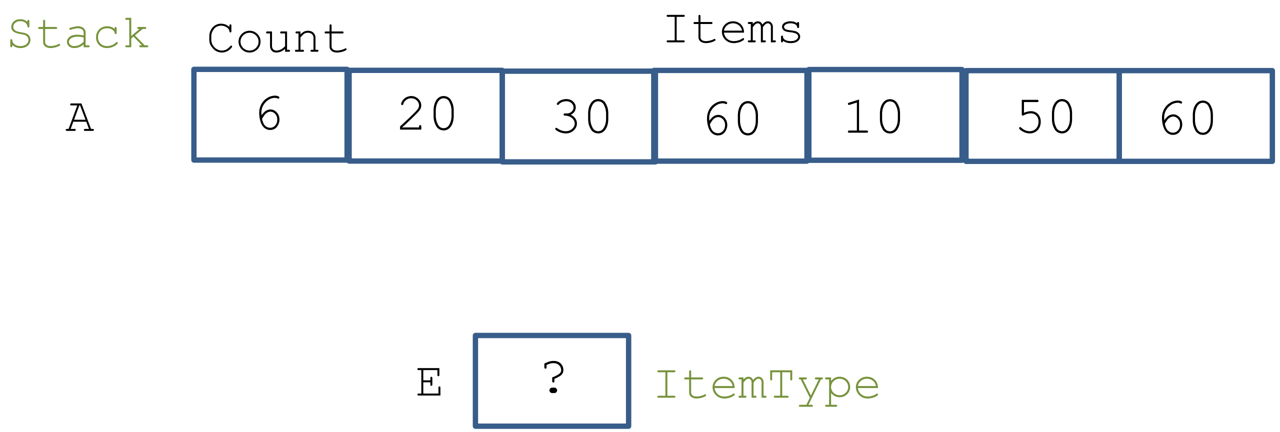
The Implementation Based on Arrays (cont’d)

```
int Full(Stack *S){
   return(S->Count == MAXSTACKSIZE);
}

void Pop(Stack *S, ItemType *X)
{  if (S->Count ==0){
      printf(“attempt to pop the empty stack”);
   } else {
      --(S->Count);
      *X=S->Items[S->Count];
   }
}
```

Example

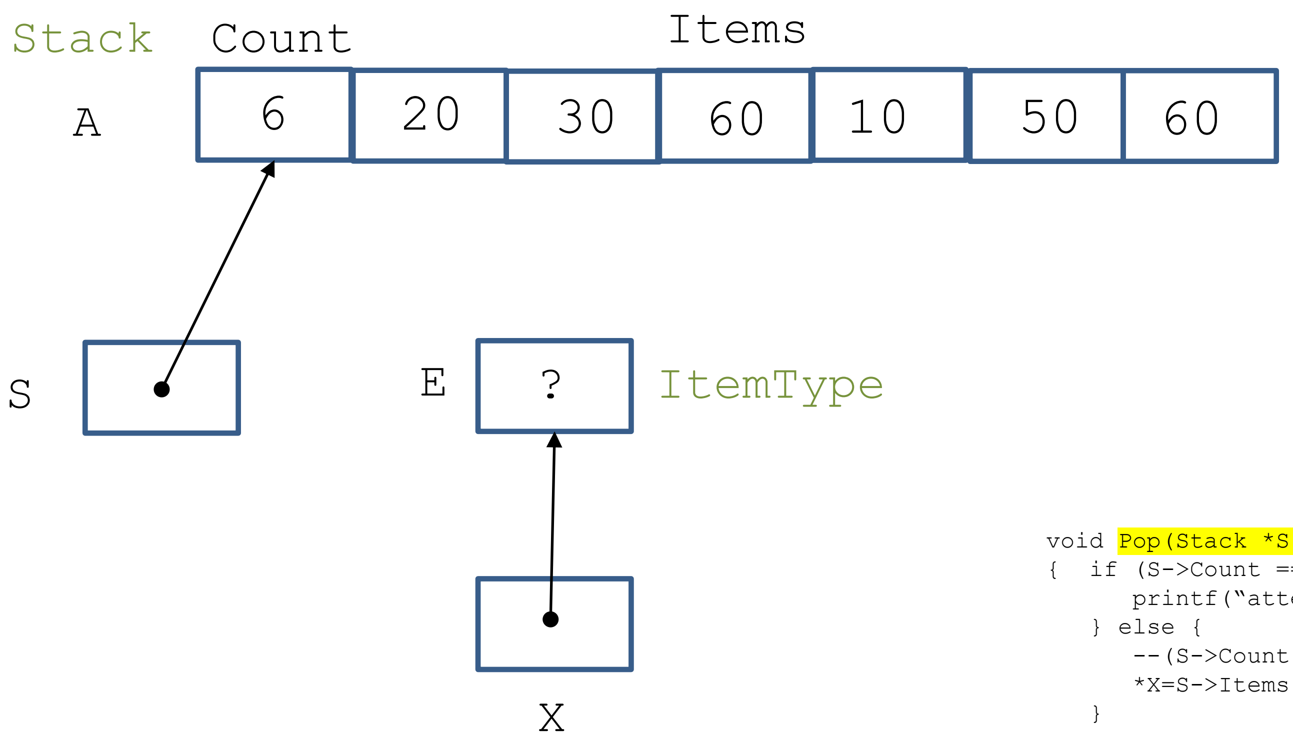
- Let us assume that we execute the call `Pop(&A, &E)` where arguments `A` and `E` correspond to the parameters `S` and `X` of the function `Pop`.
- The following slides show how this call to `Pop` proceeds.
- **Notation** for this and subsequent examples in this lecture: in each slide, we show the statement that is executed in yellow color. The accompanying figure shows the result of executing the statement.

Example (cont'd)



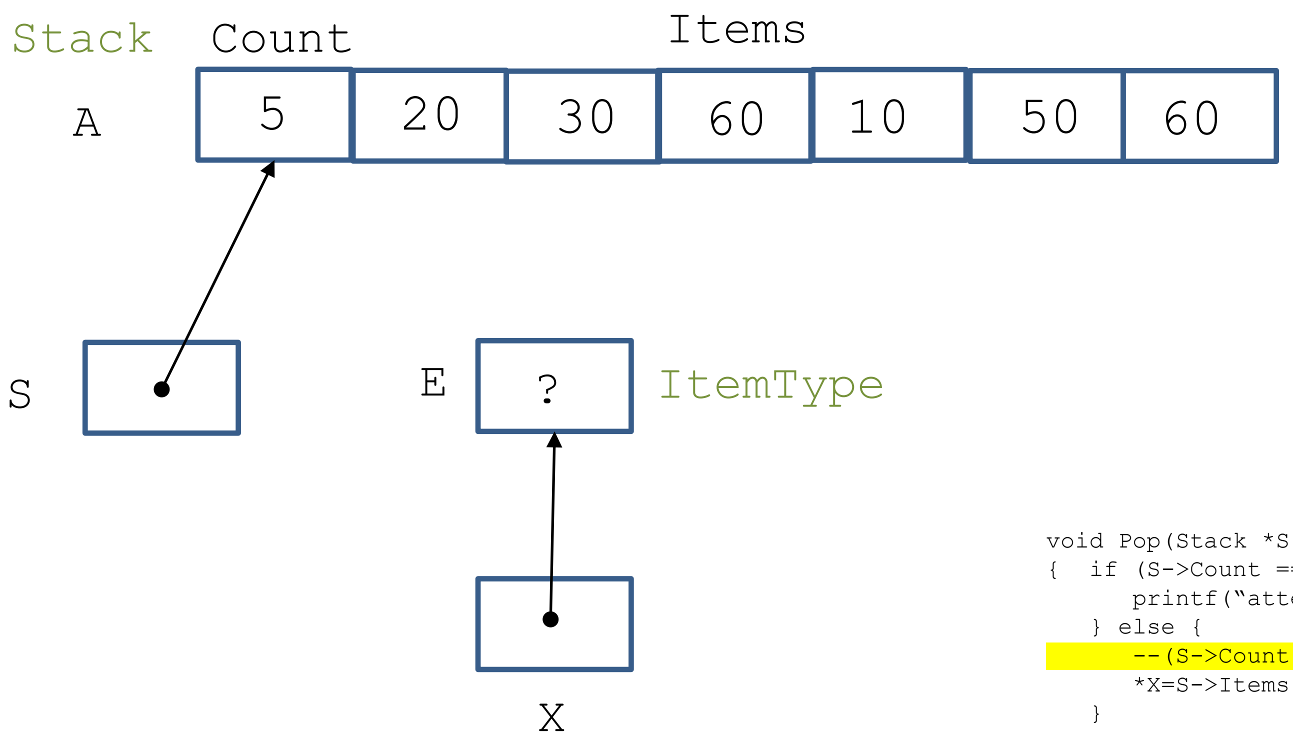
`Pop(&A, &E)`

Example (cont'd)



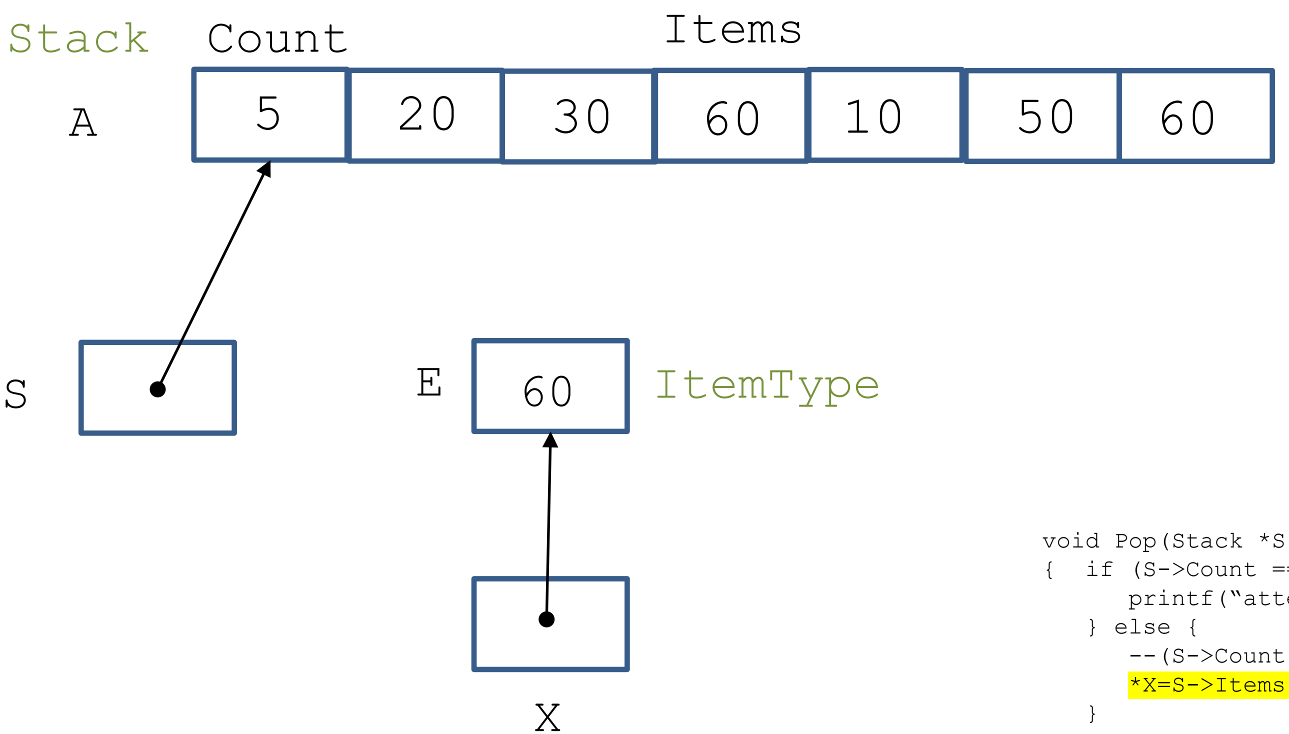

```
void Pop(Stack *S, ItemType *X)
{   if (S->Count ==0){
        printf("attempt to pop the empty stack");
    } else {
        --(S->Count);
        *X=S->Items[S->Count];
    }
}
```

Example (cont'd)



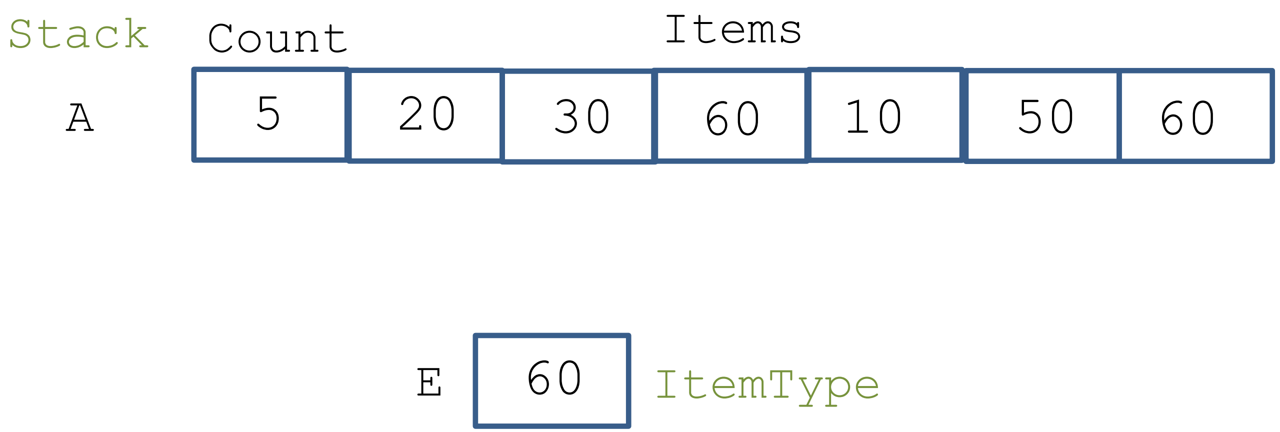

```
void Pop(Stack *S, ItemType *X)
{   if (S->Count ==0){
          printf("attempt to pop the empty stack");
    } else {
          --(S->Count);
          *X=S->Items[S->Count];
    }
}
```

Example (cont'd)




```
void Pop(Stack *S, ItemType *X)
{   if (S->Count ==0){
        printf("attempt to pop the empty stack");
    } else {
        --(S->Count);
        *X=S->Items[S->Count];
    }
}
```

Result After Executing `Pop(&A, &E)`



The Implementation Based on Arrays (cont’d)

```
void Push(ItemType X, Stack *S)
{
   if (S->Count == MAXSTACKSIZE){
      printf(“attempt to push new item on a full
stack”);
   } else {
      S->Items[S->Count]=X;
      ++(S->Count);
   }
}
```

Example

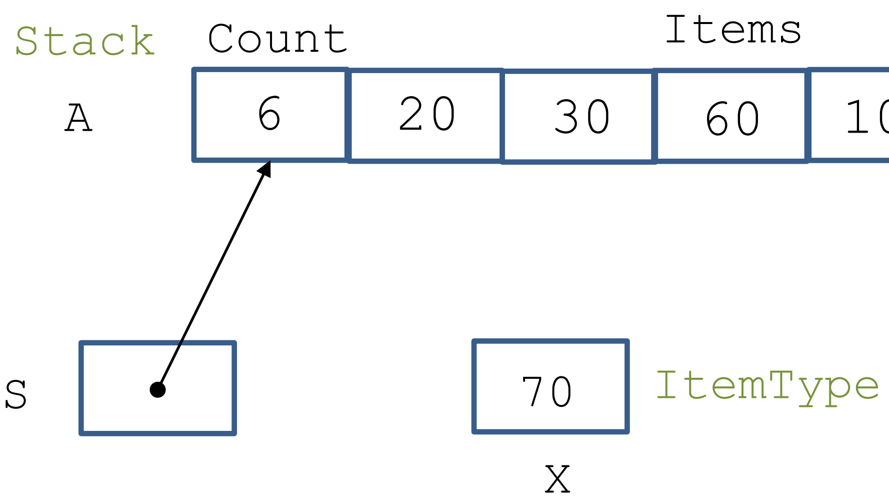
- Let us assume that we execute the call `Push(70, &A)` where argument `A` is of type `Stack` and `70` and `A` correspond to the parameters `X` and `S` of the function `Push`.
- The following slides show how this call to `Push` proceeds.

Example (cont'd)

Stack	Count	Items					
A	6	20	30	60	10	50	60

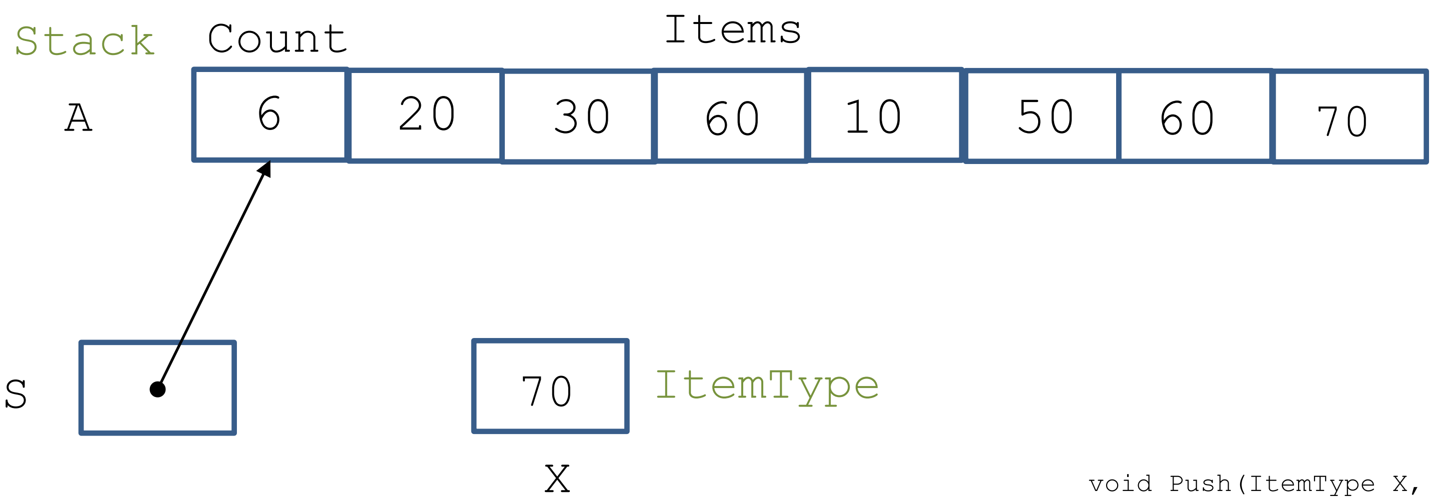
`Push(70, &A)`

Example (cont'd)



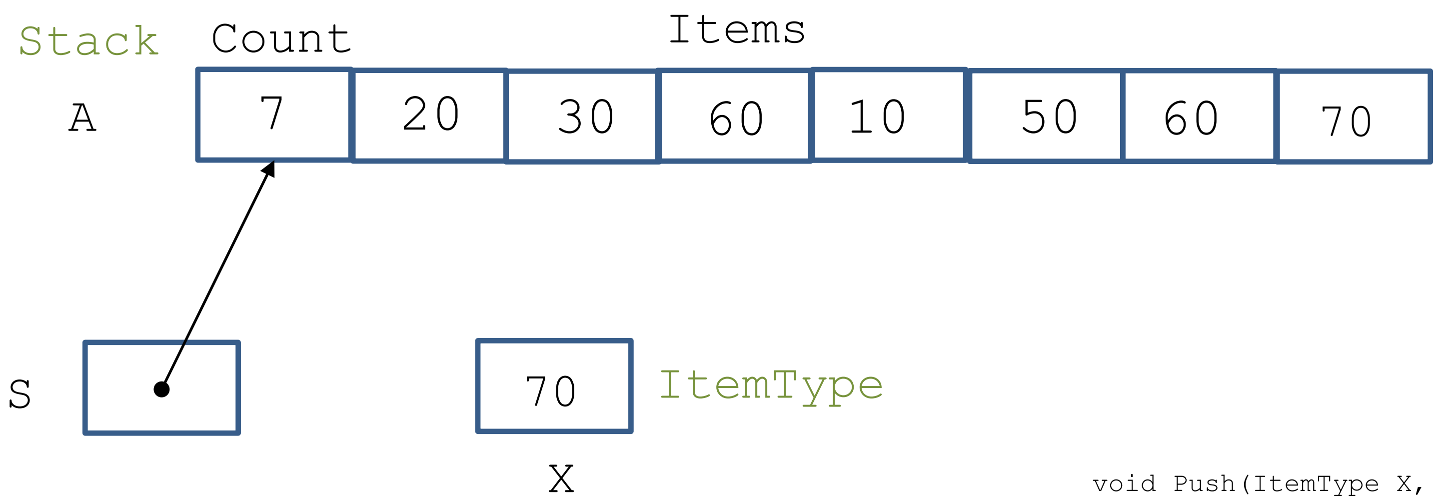

```
void Push(ItemType X, Stack *S)
{
   if (S->Count == MAXSTACKSIZE){
      printf("attempt to push new item
on a full stack");
   } else {
      S->Items[S->Count]=X;
      ++(S->Count);
   }
}
```

Example (cont'd)




```
void Push(ItemType X, Stack *S)
{
   if (S->Count == MAXSTACKSIZE){
      printf("attempt to push new item
on a full stack");
   } else {
      S->Items[S->Count]=X;
      ++(S->Count);
   }
}
```

Example (cont'd)




```
void Push(ItemType X, Stack *S)
{
   if (S->Count == MAXSTACKSIZE){
      printf("attempt to push new item
on a full stack");
   } else {
      S->Items[S->Count]=X;
      ++(S->Count);
   }
}
```

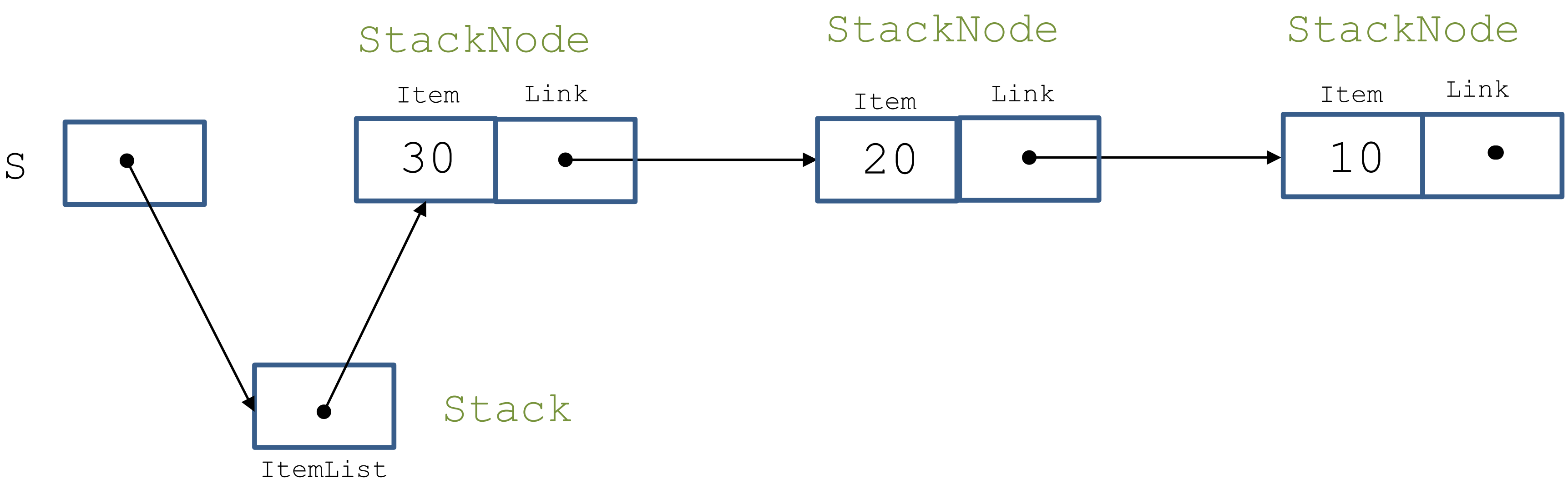
Result After Executing `Push(70, &A)`

Stack	Count			Items				
A	7	20	30	60	10	50	60	70

The Implementation Based on Linked Lists

```
/* This is the file StackTypes.h   */

typedef char ItemType;
/* char is the type for our first application */
/* float is the type for our second application */

typedef struct StackNodeTag {
           ItemType Item;
           struct StackNodeTag *Link;
        } StackNode;

typedef struct {
           StackNode *ItemList;
        } Stack;
```

Example


StackNode
StackNode
StackNode
Item
Link
Item
Link
Item
Link
S
30
20
10
Stack
ItemList

Notes

- In this implementation, **we keep the stack in the reverse order of the array implementation**; from most recently inserted elements to least recently inserting elements.
- We assume that **the top of the stack is the node to which pointer `ItemList` points.**
- We have no field `Count` in the structure `Stack`.
- `S` is a pointer to a stack that we will use as parameter in the functions that will implement the stack operations (see the following slides). `S` will correspond to the argument stack that we will use when calling these functions from our main program (e.g., the stack `EvalStack` in the above example).

Comment

- With the previous definition of type `Stack` using linked lists, we have **uniformity** with our earlier definition using arrays.
- We could have defined the type `Stack` differently if we don't care about uniformity: as a pointer to a structure of type `StackNode`. Do it as an exercise!

The Implementation Based on Linked Lists (cont'd)

```
/* This is the file StackImplementation.c */

#include <stdio.h>
#include <stdlib.h>
#include "StackInterface.h"

void InitializeStack(Stack *S)
{
   S->ItemList=NULL;
}

int Empty(Stack *S)
{
   return (S->ItemList==NULL);
}

int Full(Stack *S)
{
  return 0;
}
/* We assume an already constructed stack is not full since it can potentially */
/* grow as a linked structure  */
```

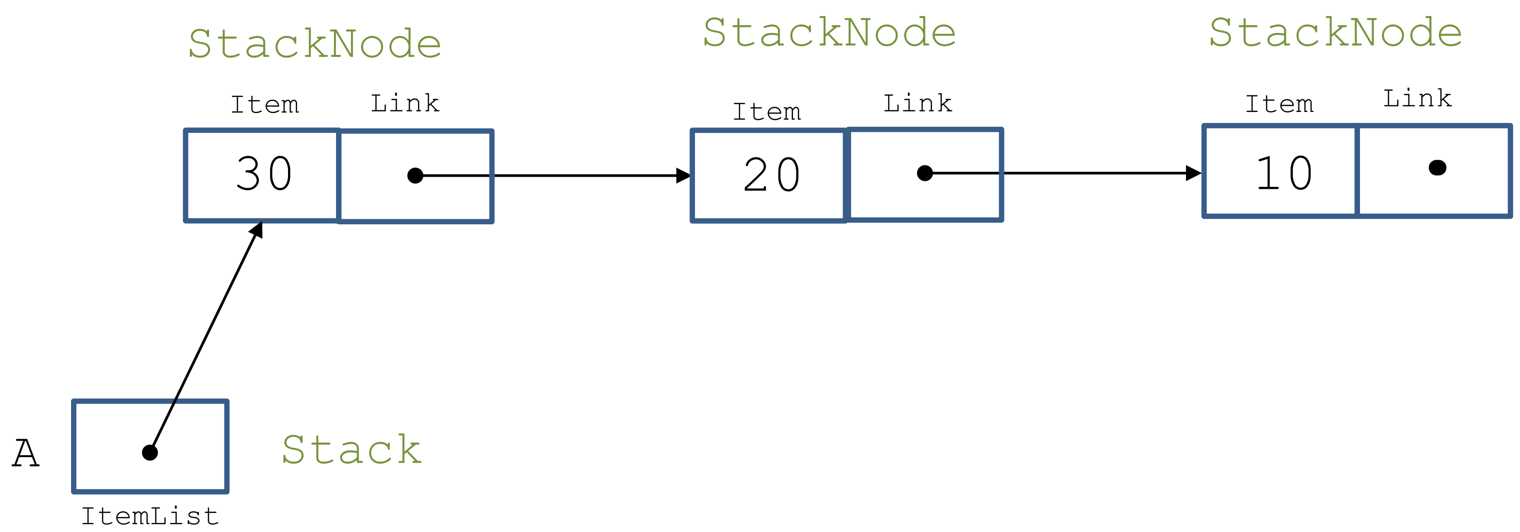
The Implementation Based on Linked Lists (cont'd)

```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Example

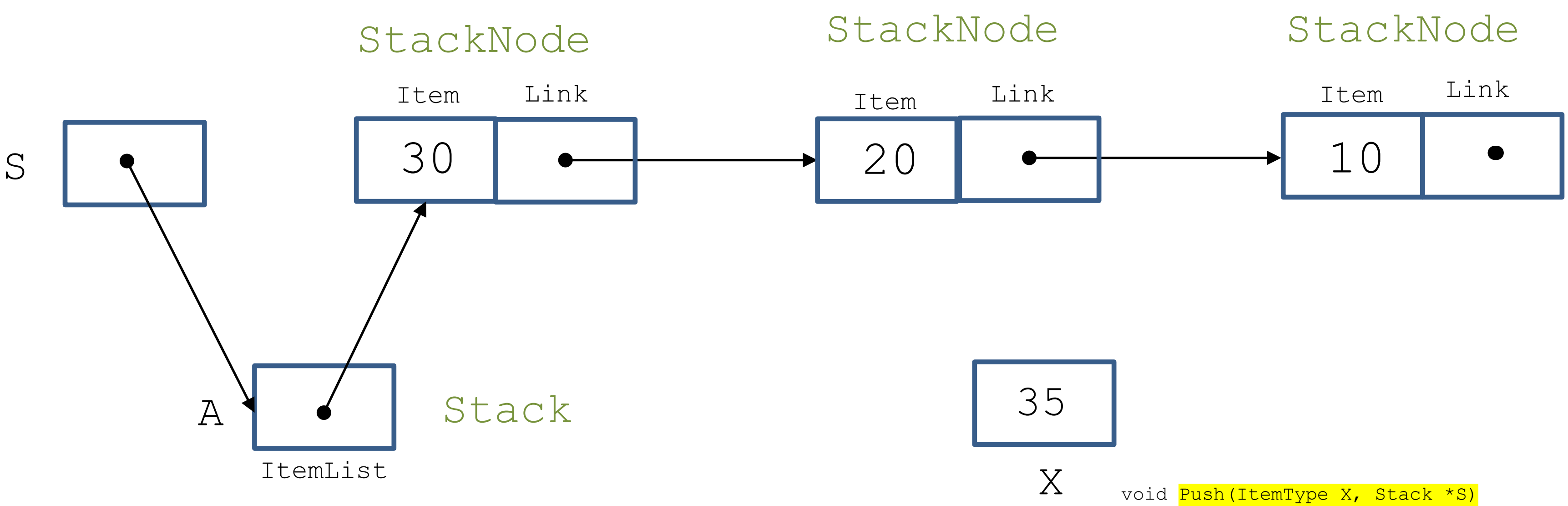
- Now assume that we execute the call `Push(35,&A)` where `A` is of type `Stack` and `35` and `A` correspond to parameters `X` and `S` of the function `Push`.
- The following slides show how this call to `Push` proceeds.

Example (cont'd)



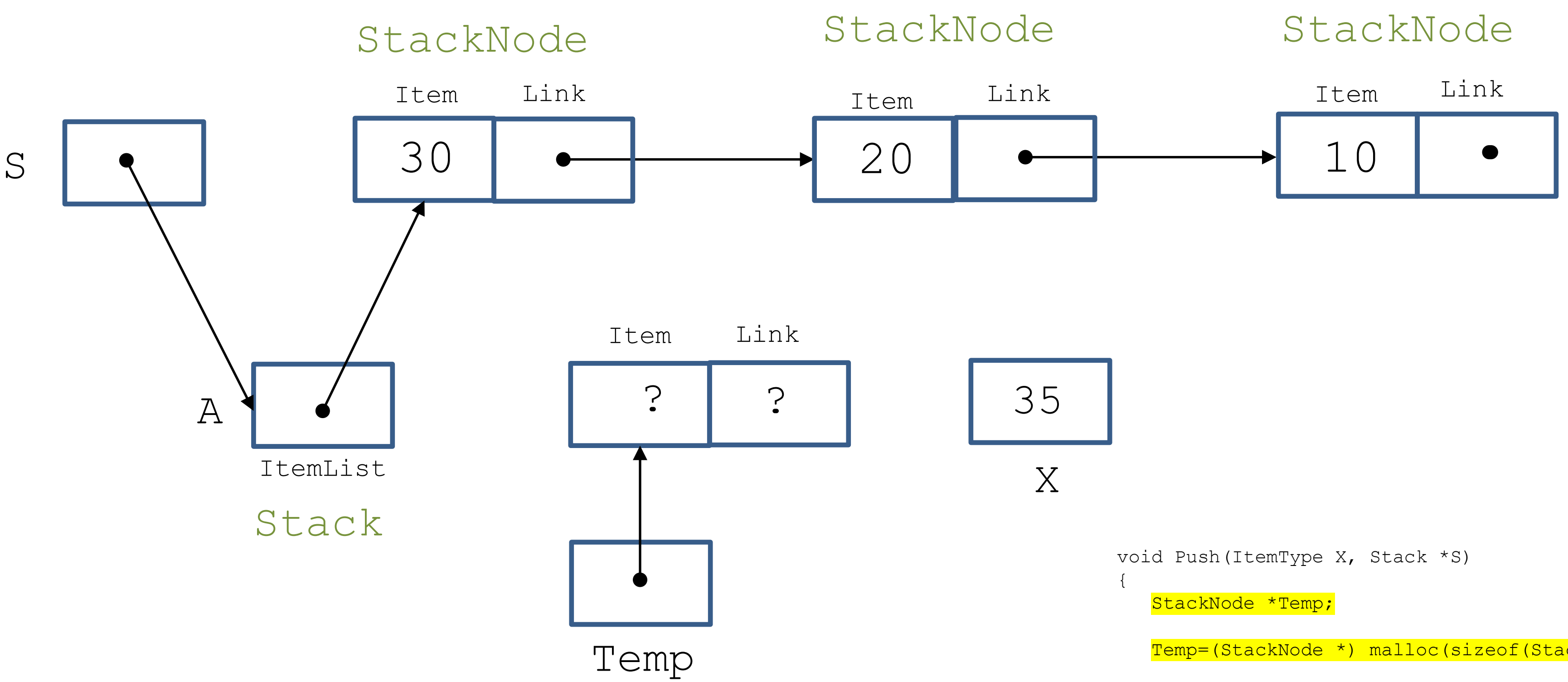
`Push(35, &A)`

Example (cont'd)



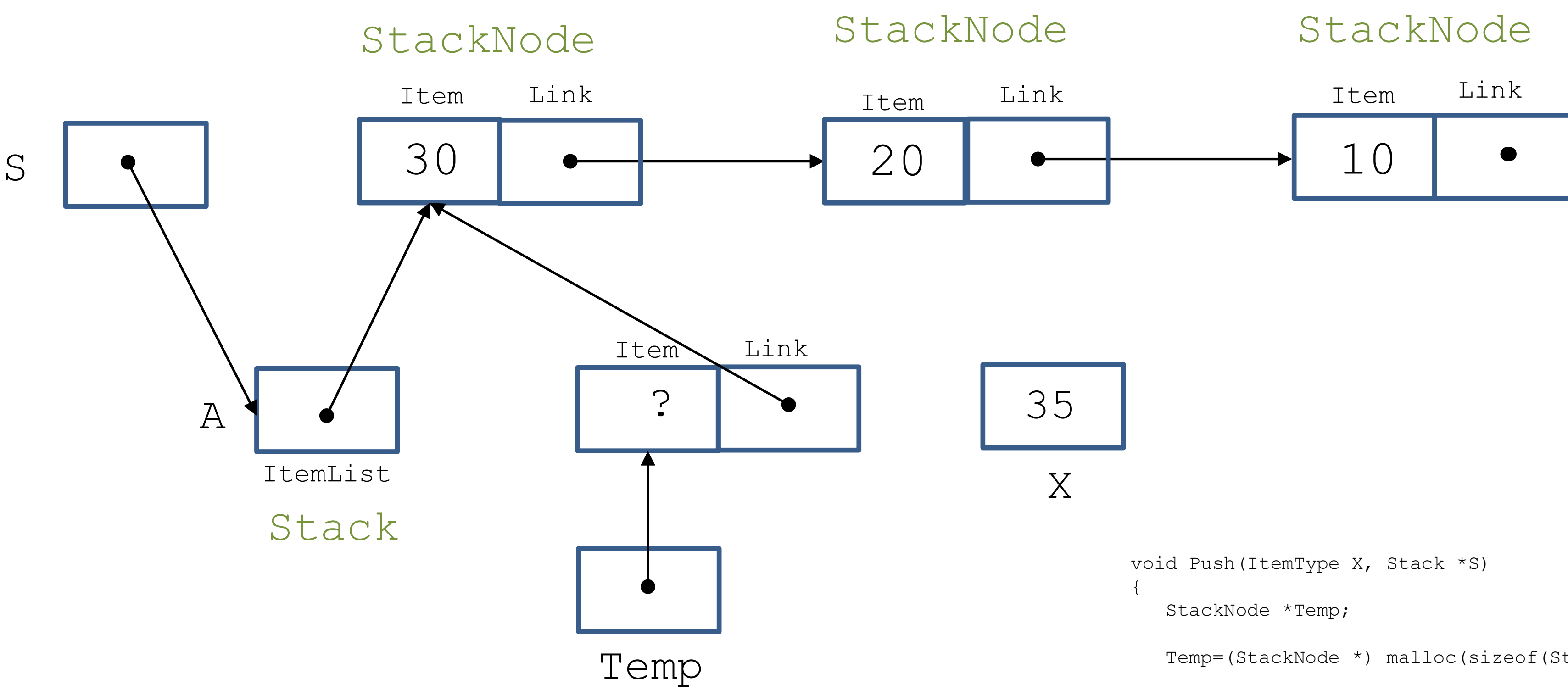

```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Example (cont'd)



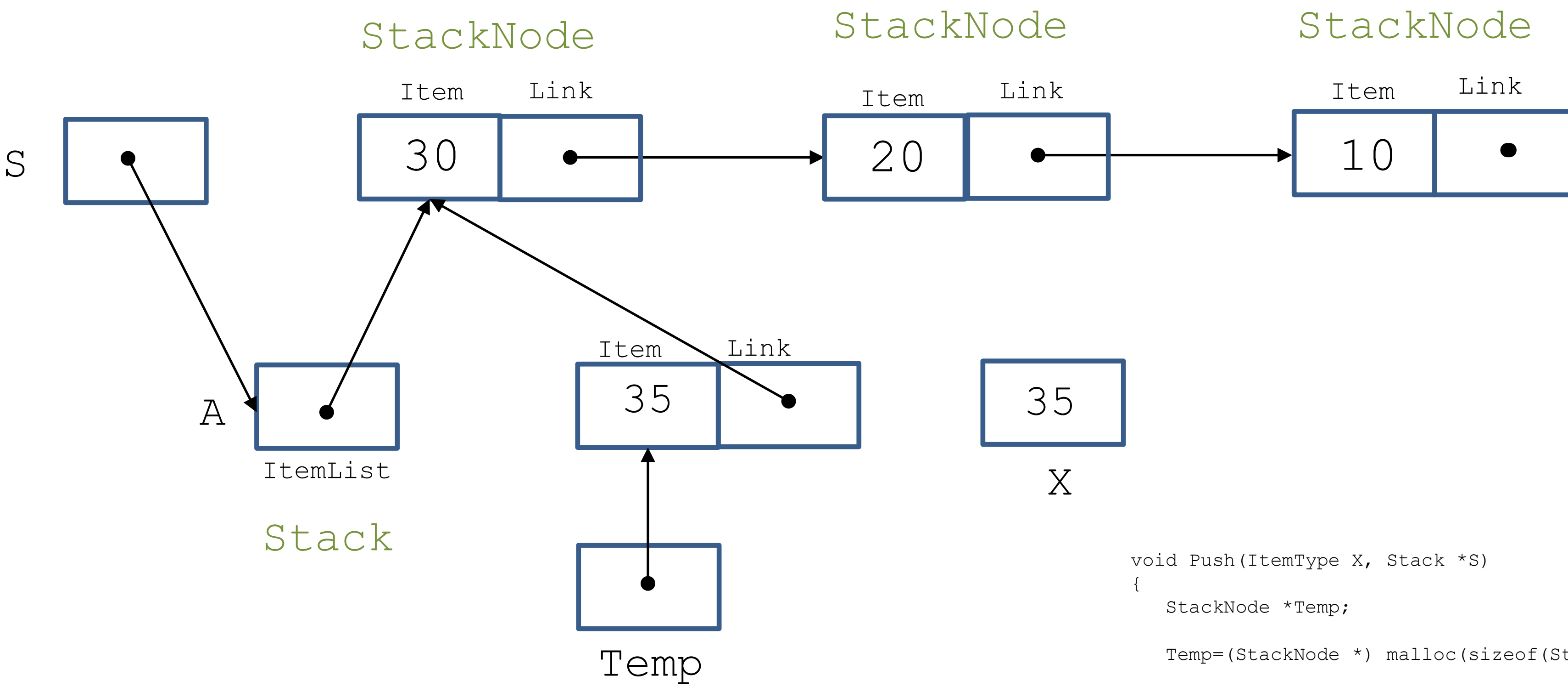

```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Example (cont'd)



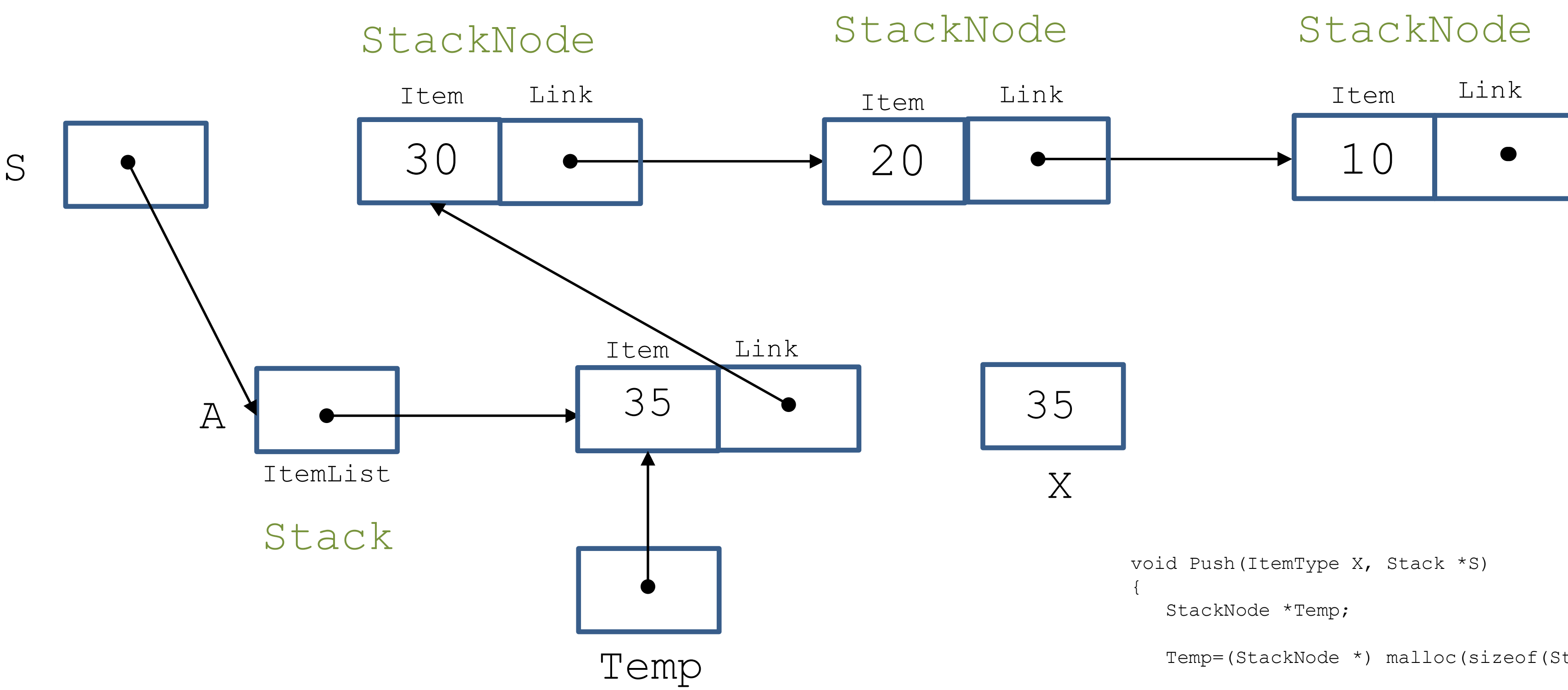

```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Example (cont'd)



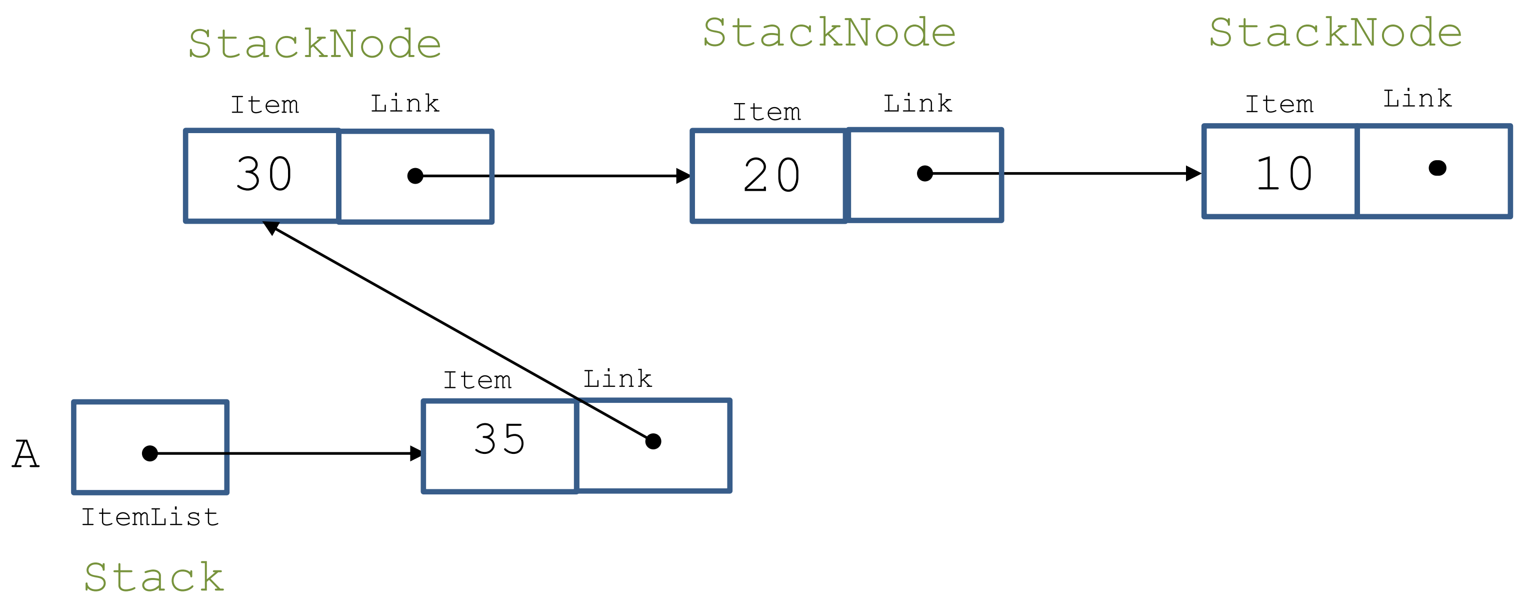

```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Example (cont'd)




```
void Push(ItemType X, Stack *S)
{
   StackNode *Temp;

   Temp=(StackNode *) malloc(sizeof(StackNode));

   if (Temp==NULL){
      printf("system storage is exhausted");
   } else {
      Temp->Link=S->ItemList;
      Temp->Item=X;
      S->ItemList=Temp;
   }
}
```

Result After Executing `Push(&A, 35)`



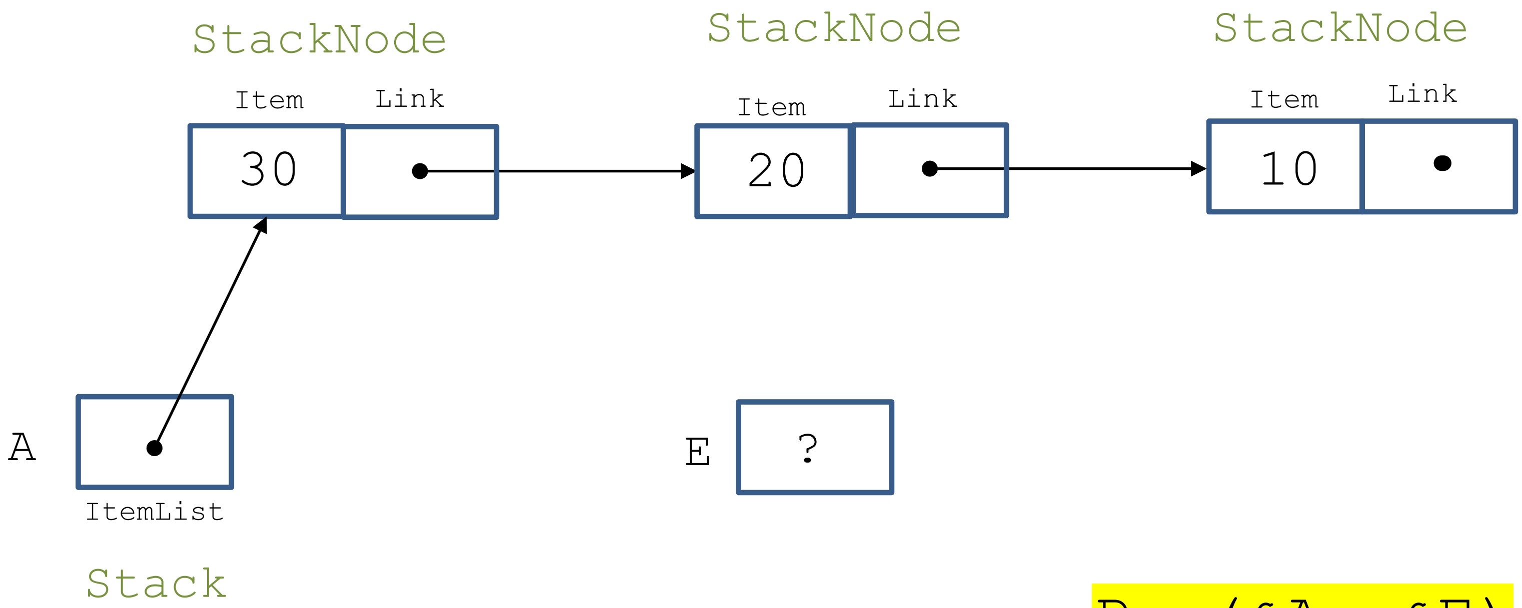
The Implementation Based on Linked Lists (cont’d)

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf(“attempt to pop the empty stack”);
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example

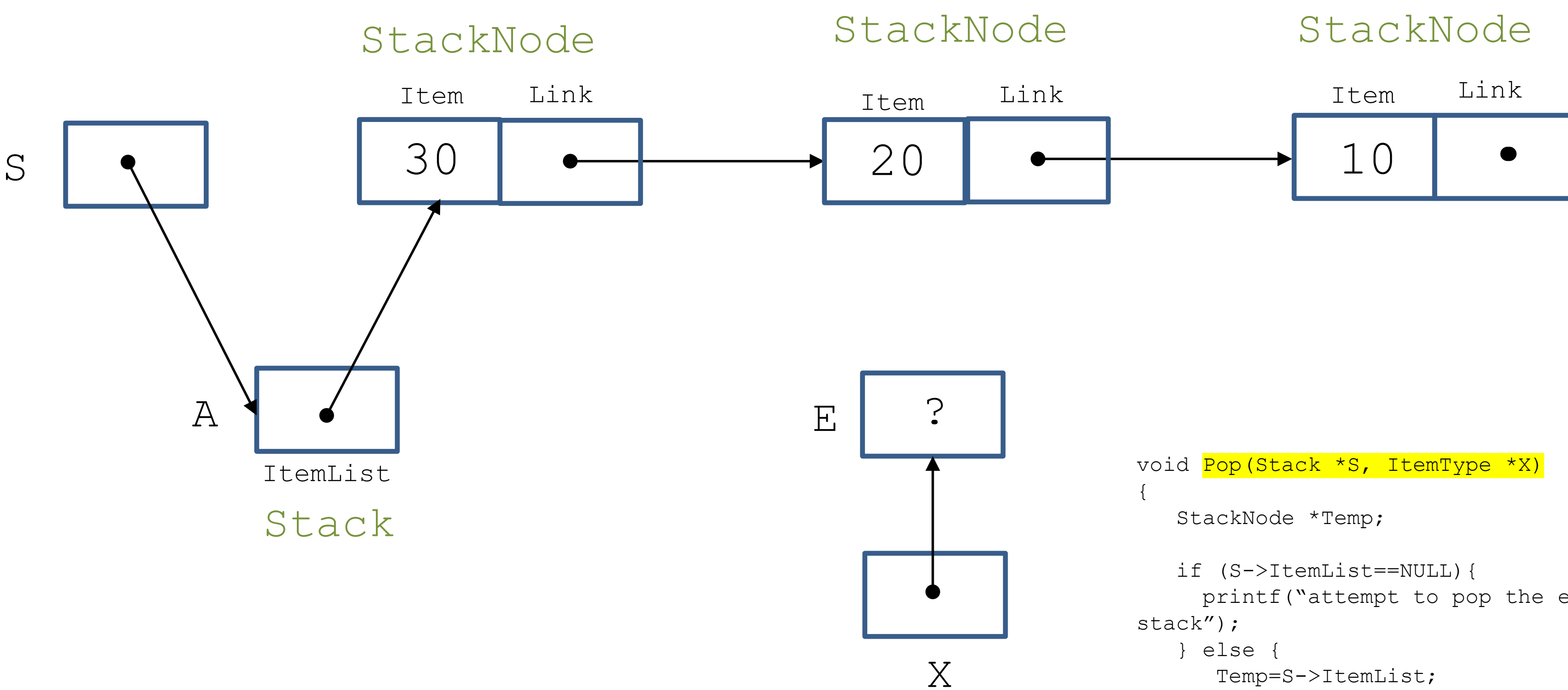
- Now assume that we execute the call `Pop(&A, &E)` where `A` and `E` are of type `Stack` and `int` respectively and correspond to parameters `S` and `X` of the function `Pop`.
- The following slides show how this call to `Pop` proceeds.

Example (cont'd)



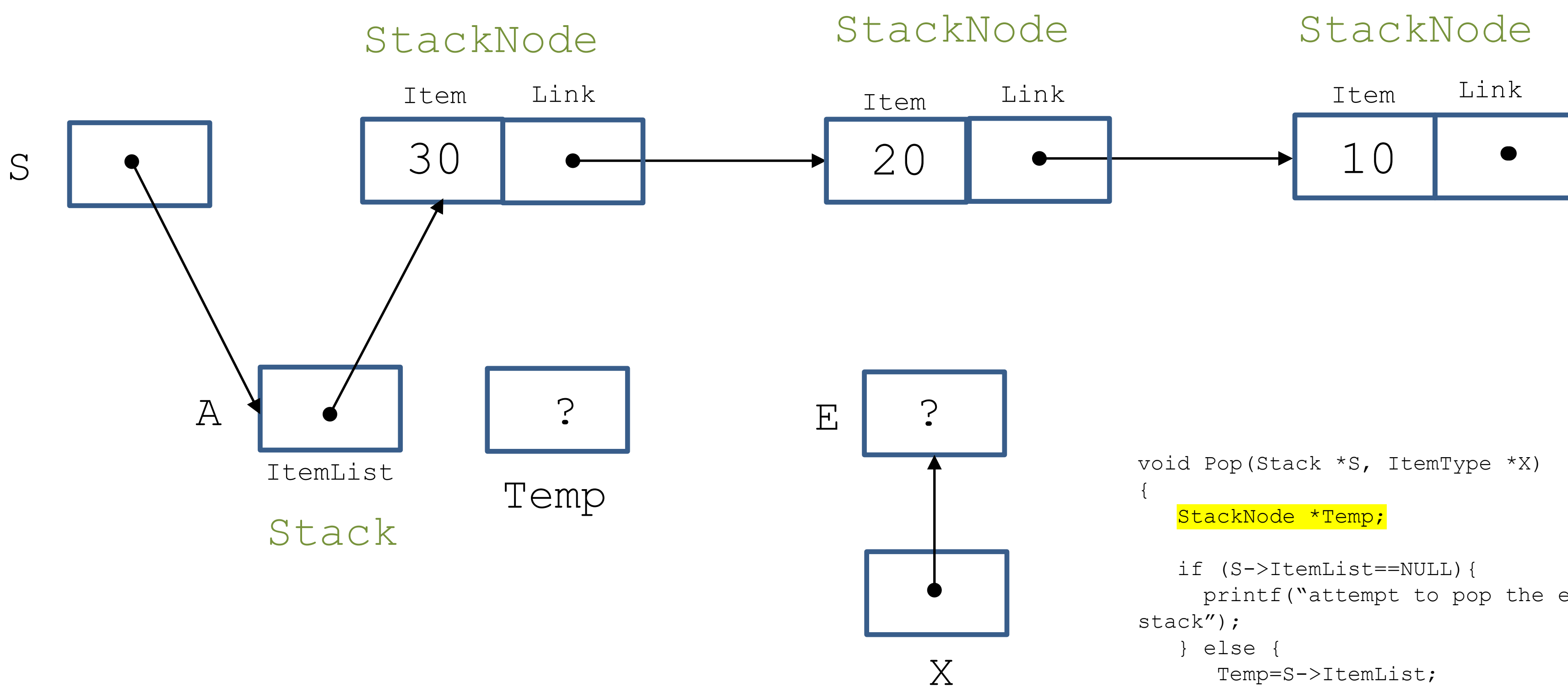
`Pop(&A, &E)`

Example (cont'd)



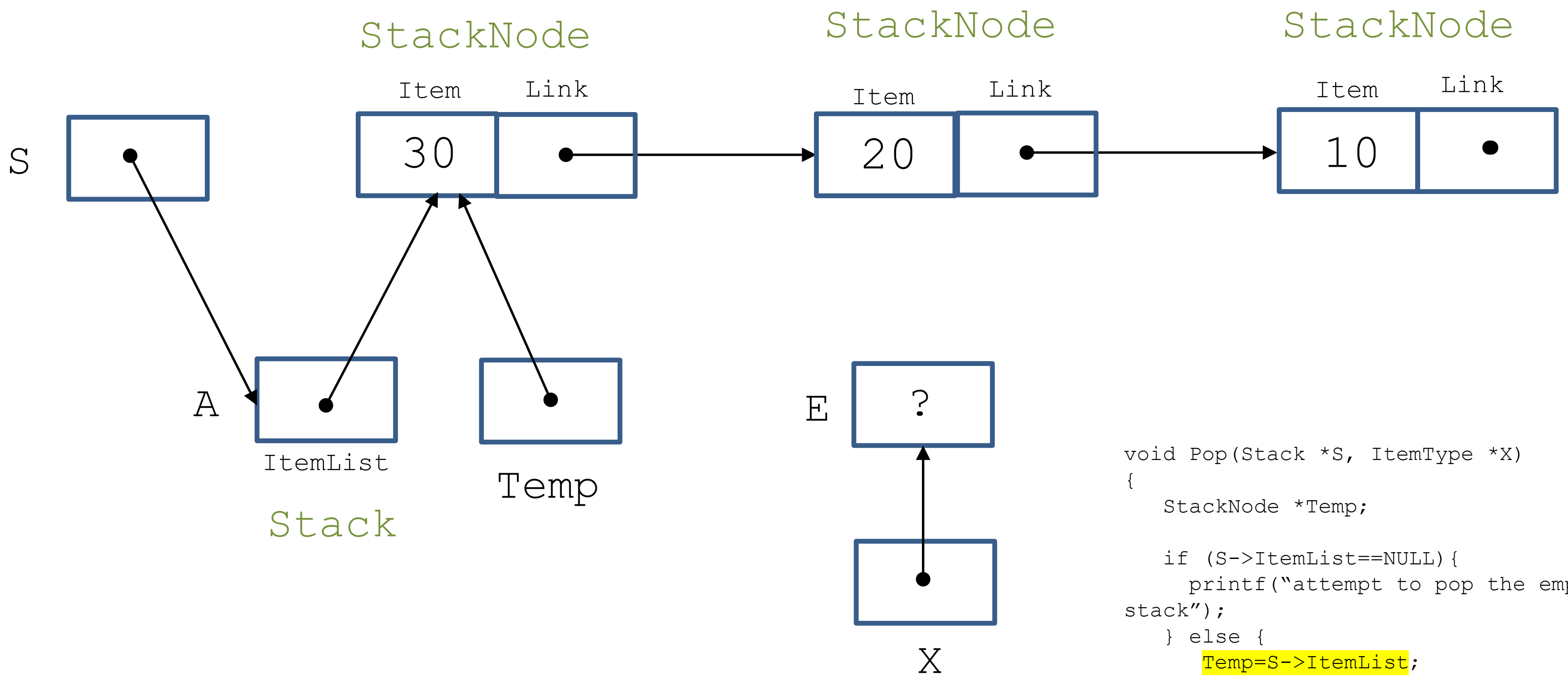

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example (cont'd)



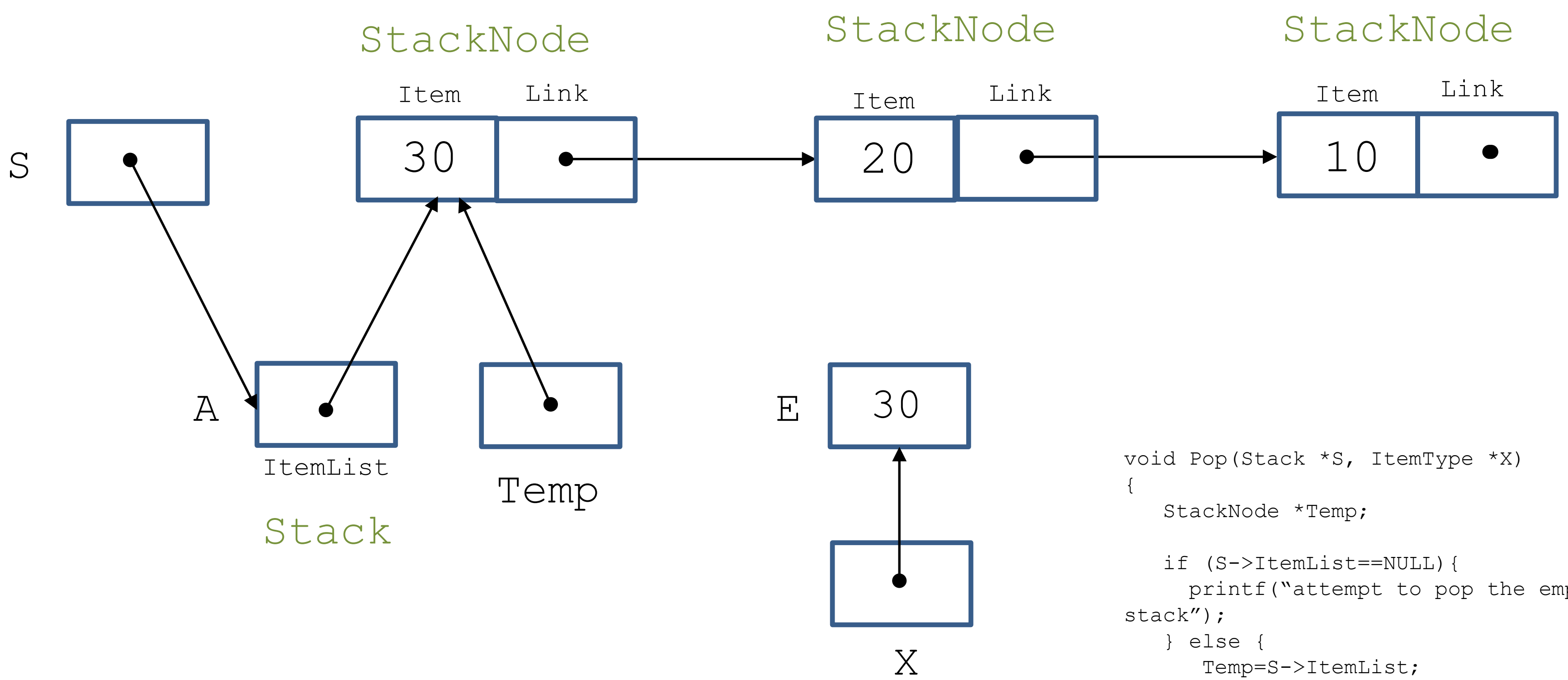

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example (cont’d)



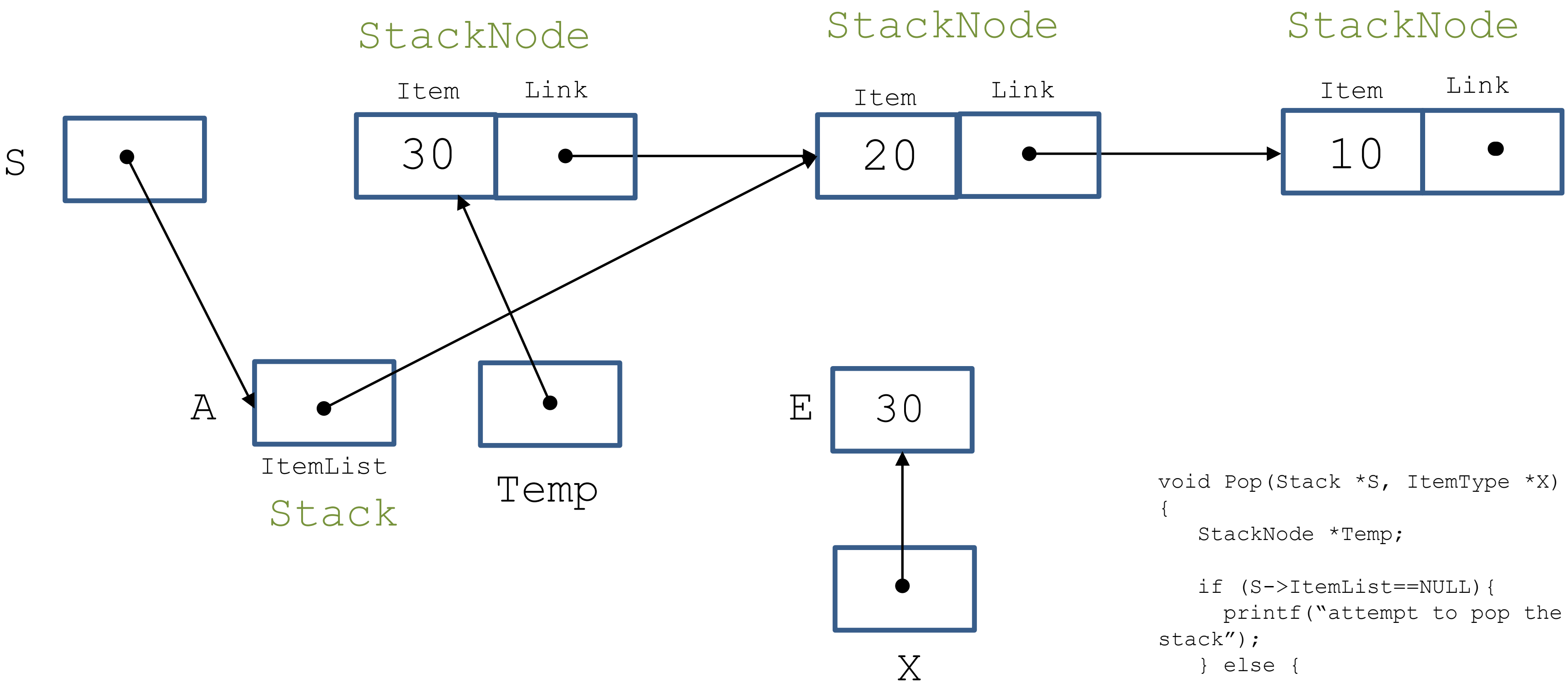

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example (cont'd)



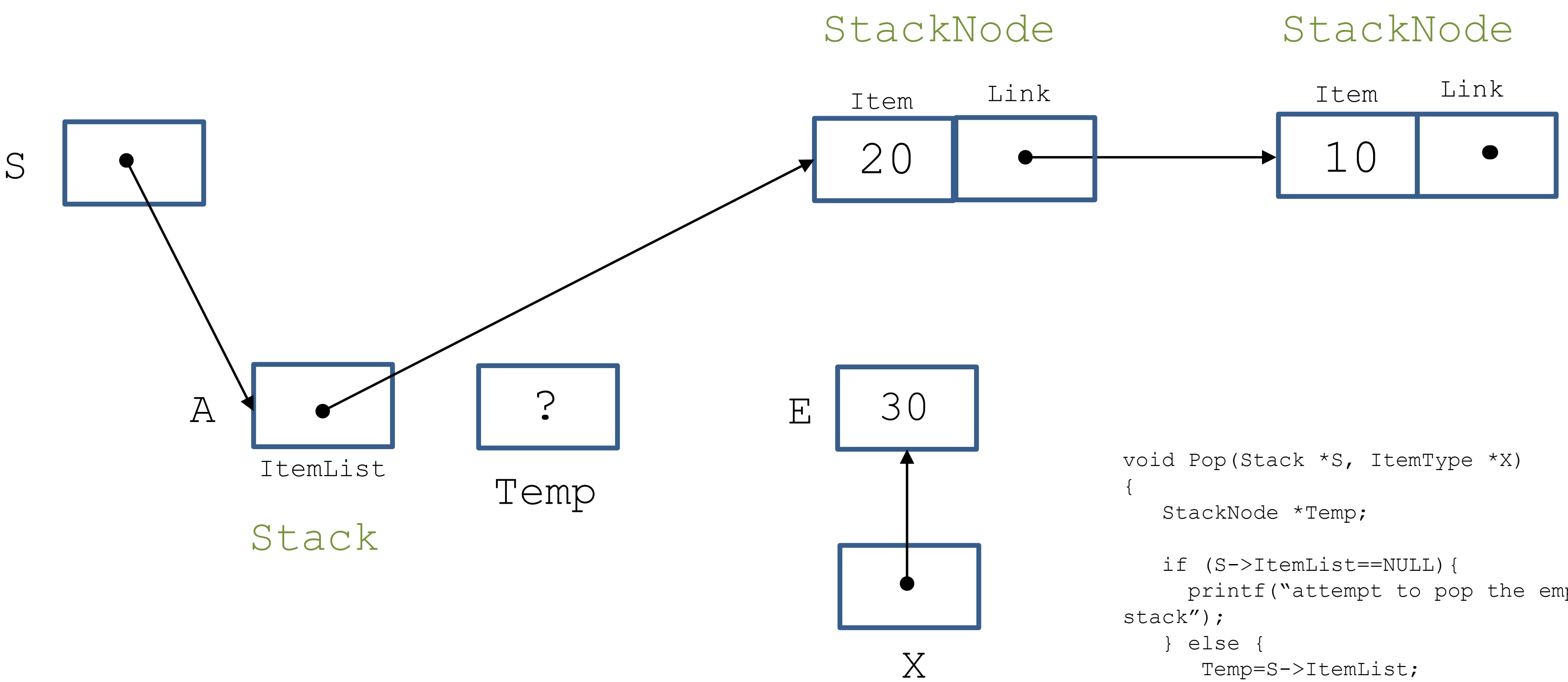

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example (cont'd)



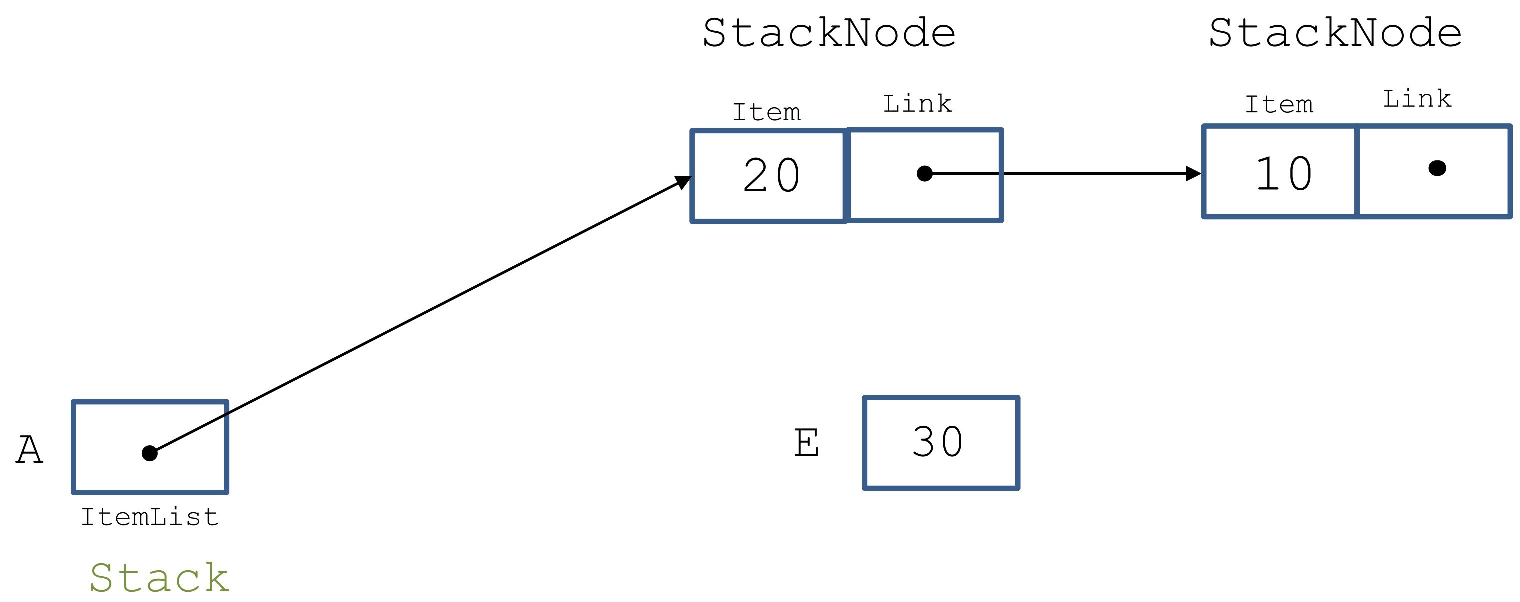

```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Example (cont'd)




```
void Pop(Stack *S, ItemType *X)
{
   StackNode *Temp;

   if (S->ItemList==NULL){
     printf("attempt to pop the empty
stack");
   } else {
      Temp=S->ItemList;
      *X=Temp->Item;
      S->ItemList=Temp->Link;
      free(Temp);
   }
}
```

Result After the Call `Pop(&A, &E)` Terminates



Information Hiding Revisited

- The two previous specifications of the ADT stack **do not hide the details of the representation** of the stack since a client program can access the array or the list data structure because it includes `StackInterface.h` and therefore `StackTypes.h`.
- We will now present another specification which does a better job in hiding the representation of the stack.

The Interface File `STACK.h`

```
void STACKinit(int);
int STACKempty();
void STACKpush(Item);
Item STACKpop();
```

The type `Item` will be defined in a header file `Item.h` which will be included in the implementation of the interface and the client programs.

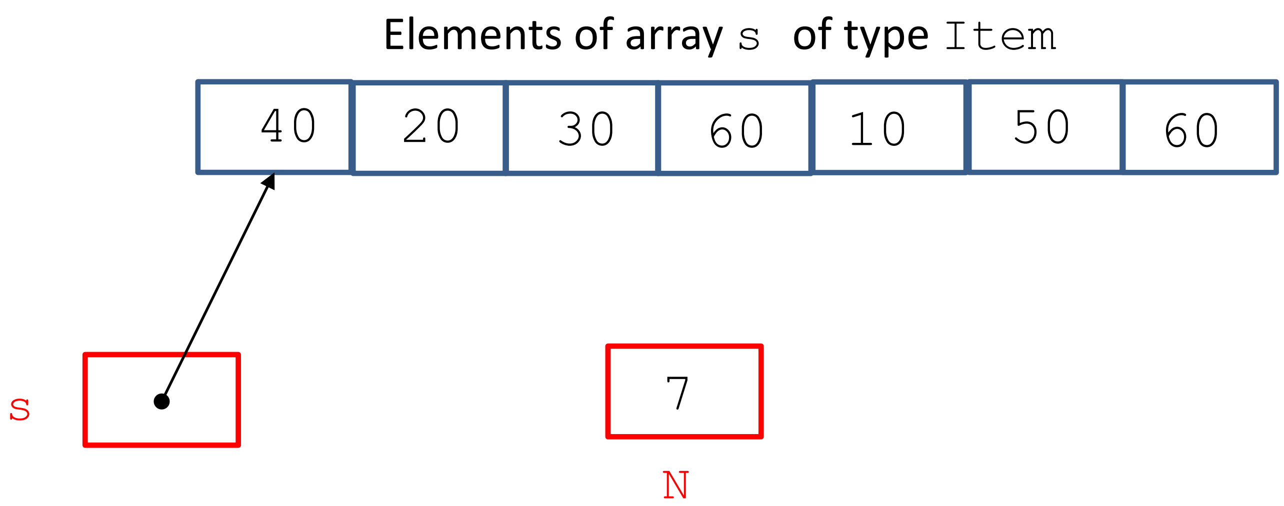
The Implementation of the Interface

- As previously, we will consider an **array implementation** and a **linked list implementation** of the ADT stack.

The Array Implementation

```
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"
static Item *s;
static int N;

void STACKinit(int maxN)
{ s = malloc(maxN*sizeof(Item)); N = 0; }

int STACKempty()  { return N == 0; }

void STACKpush(Item item)
{ s[N++] = item; }

Item STACKpop()  { return s[--N]; }
```

Example



Notation: in this and subsequent examples, static variables are shown in red.

Notes

- The variable `s` is a pointer to an item (equivalently, the name of an array of items defined by `Item s[]`).
- When there are `N` items in the stack, the implementation keeps them in array elements `s[0], …, s[N-1]`.
- The variable `N` **shows the top of the stack (where the next item to be pushed will go).** It also tells us how many elements exist in the stack. It is like the variable `Count` in the other implementation with arrays which we presented earlier.
- `s` and `N` are defined as **static** variables i.e., they **retain their values throughout calls** of the various functions that access them, **but their scope is restricted to the file they have been defined in**; they cannot be accessed by other files like global variables.
- The client program passes the maximum number of items expected on the stack as an argument to `STACKinit`.
- The previous code **does not check for errors** such as pushing onto a full stack or popping an empty one.

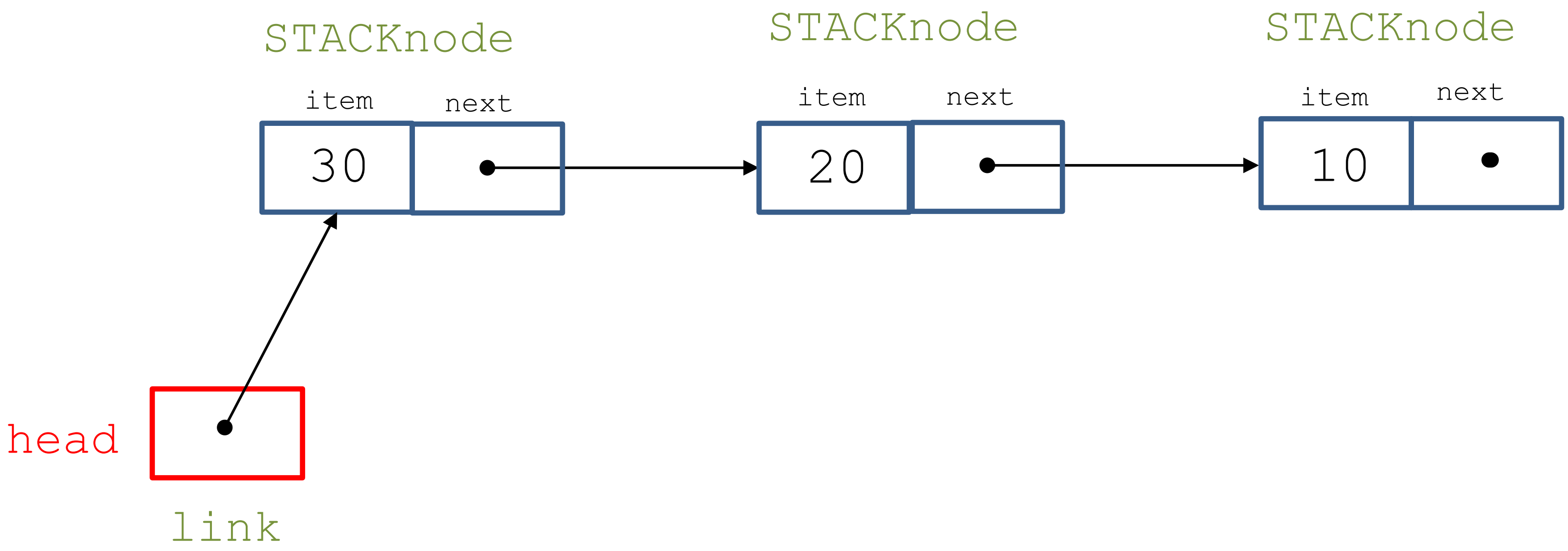
The Linked List Implementation

```
#include <stdlib.h>
#include "Item.h"

typedef struct STACKnode* link;
struct STACKnode { Item item; link next; };
static link head;


void STACKinit(int maxN)  { head = NULL; }

int STACKempty()
{ return head == NULL; }
```

The Linked List Implementation (cont'd)

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}


STACKpush(Item item)
{ head = NEW(item, head); }
```

Example



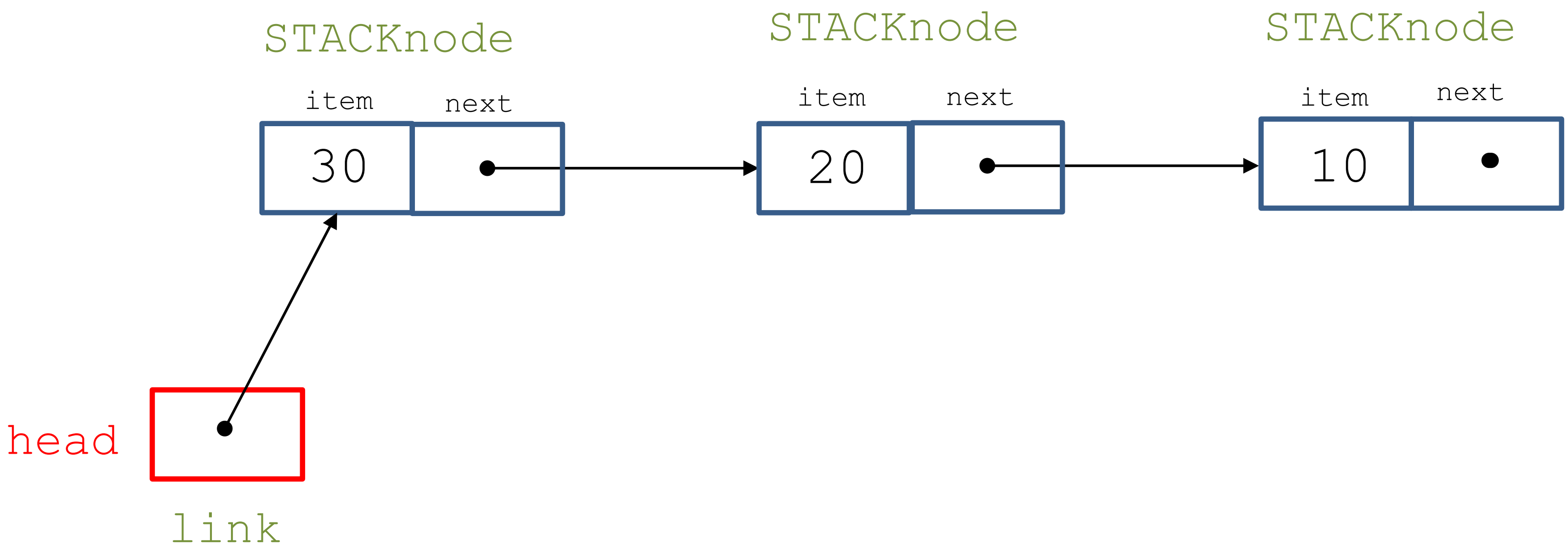
STACKnode
item
next
30
STACKnode
item
next
20
STACKnode
item
next
10
head
link

Example (cont'd)

- Let us call `STACKpush(40)` now.
- `STACKpush` will call `NEW(40,head)`.

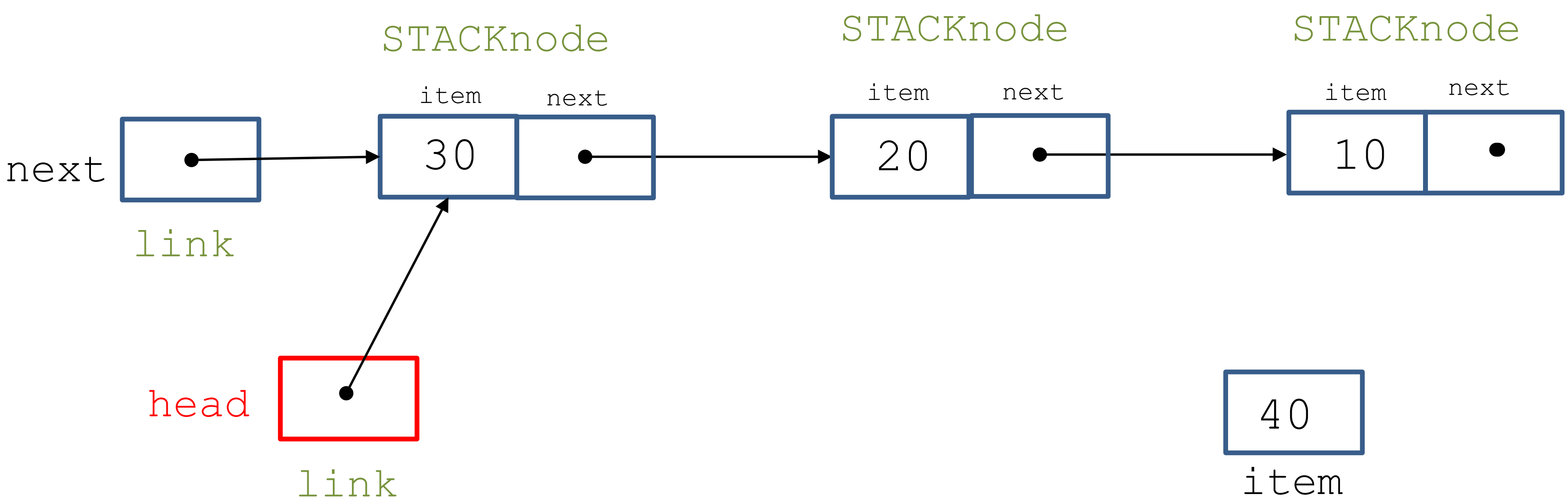
- In the following slides we simulate the execution of `NEW(40,head)`.

Example



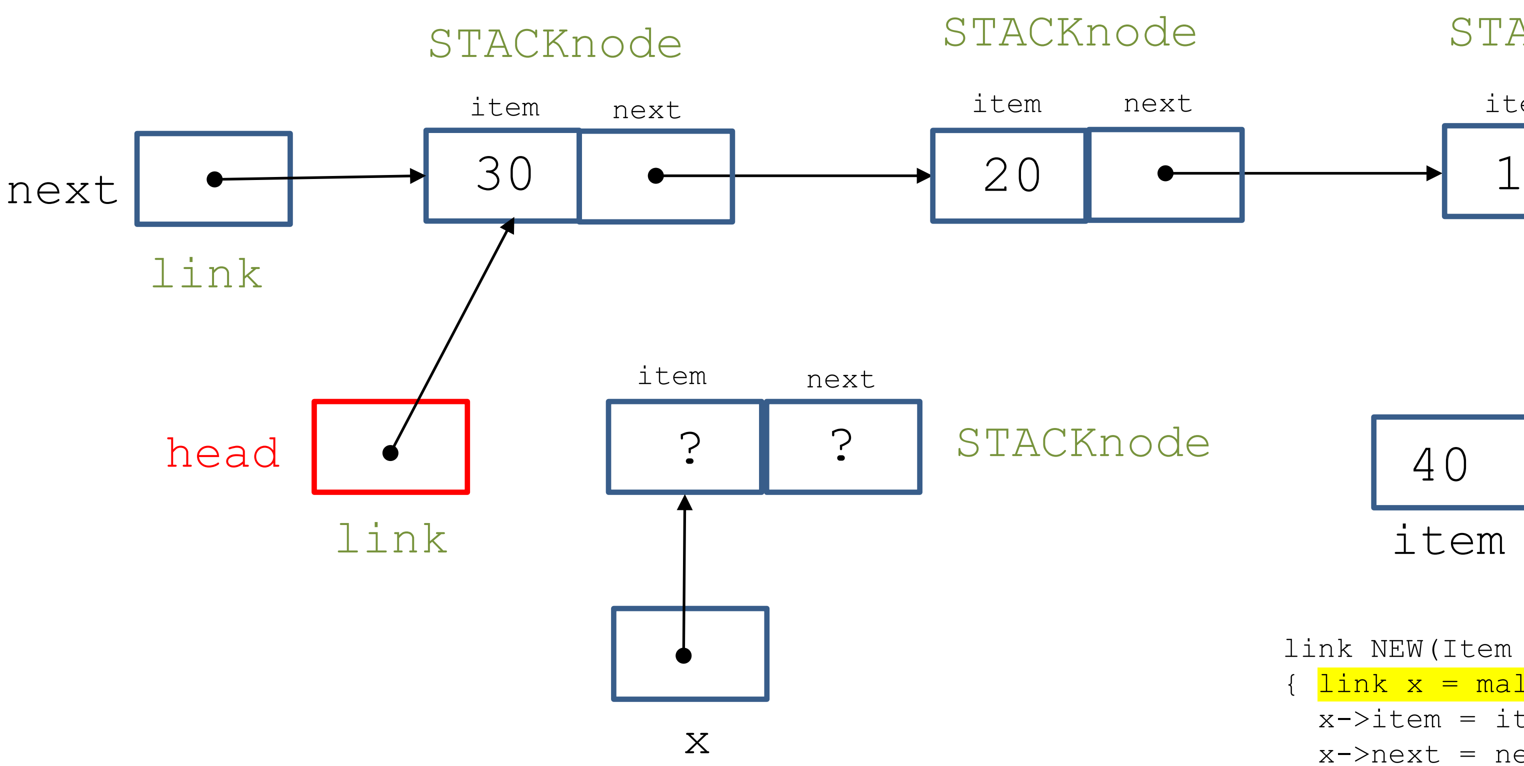
`head = NEW(item, head);`

Example



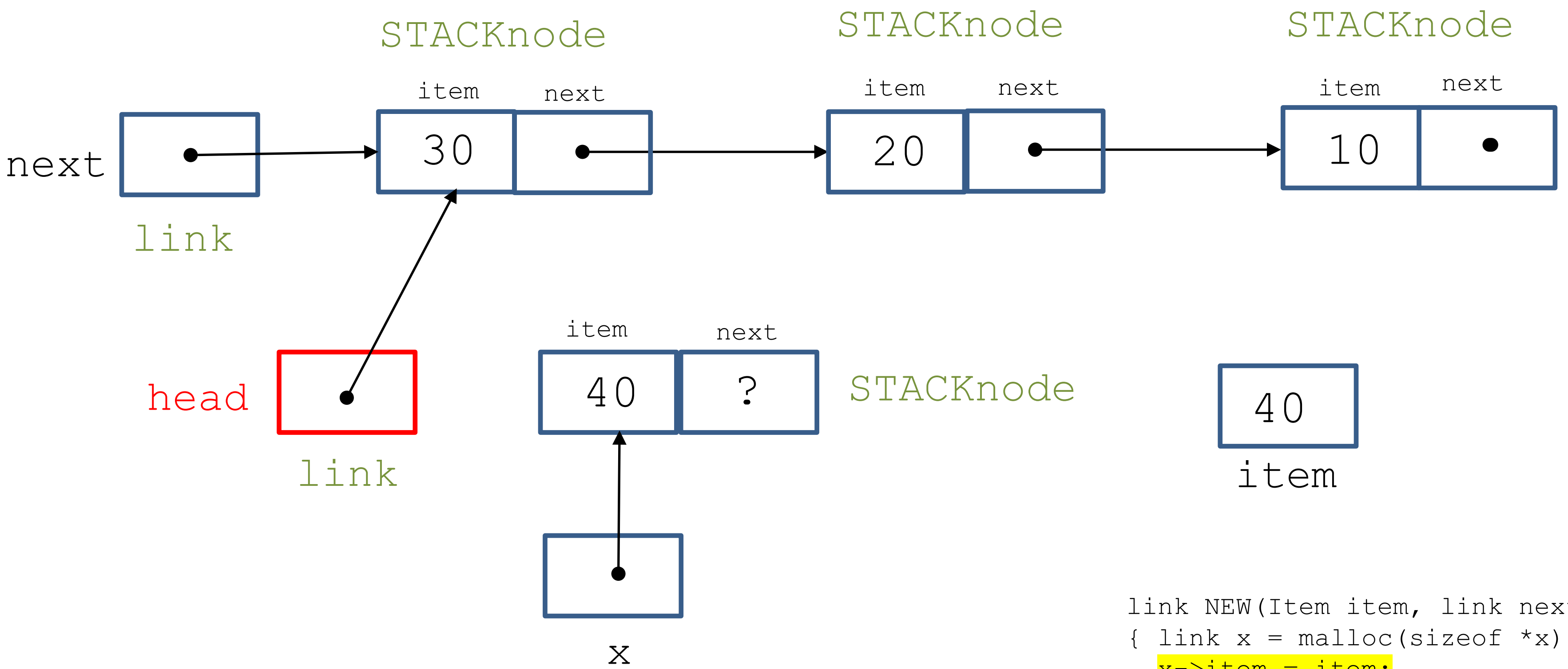

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

Example (cont'd)



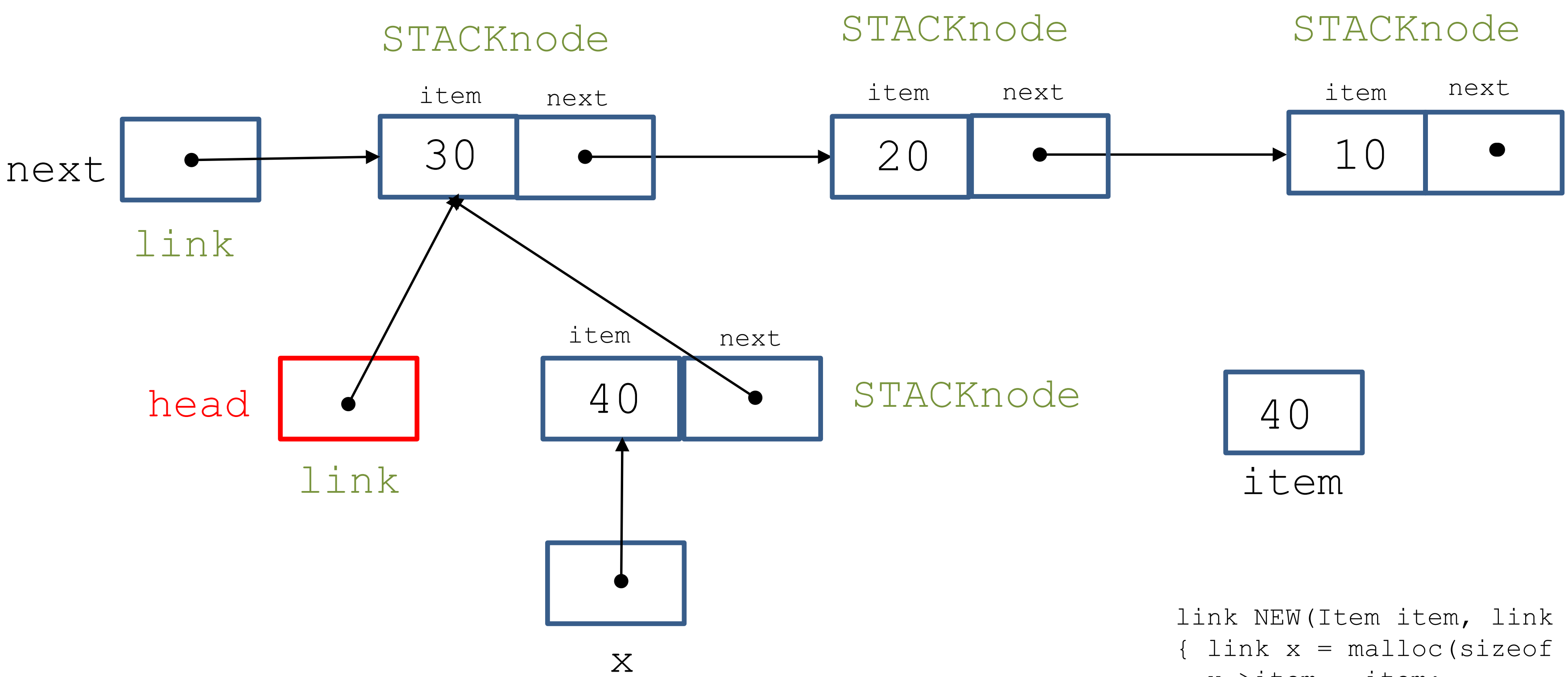

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

Example (cont'd)



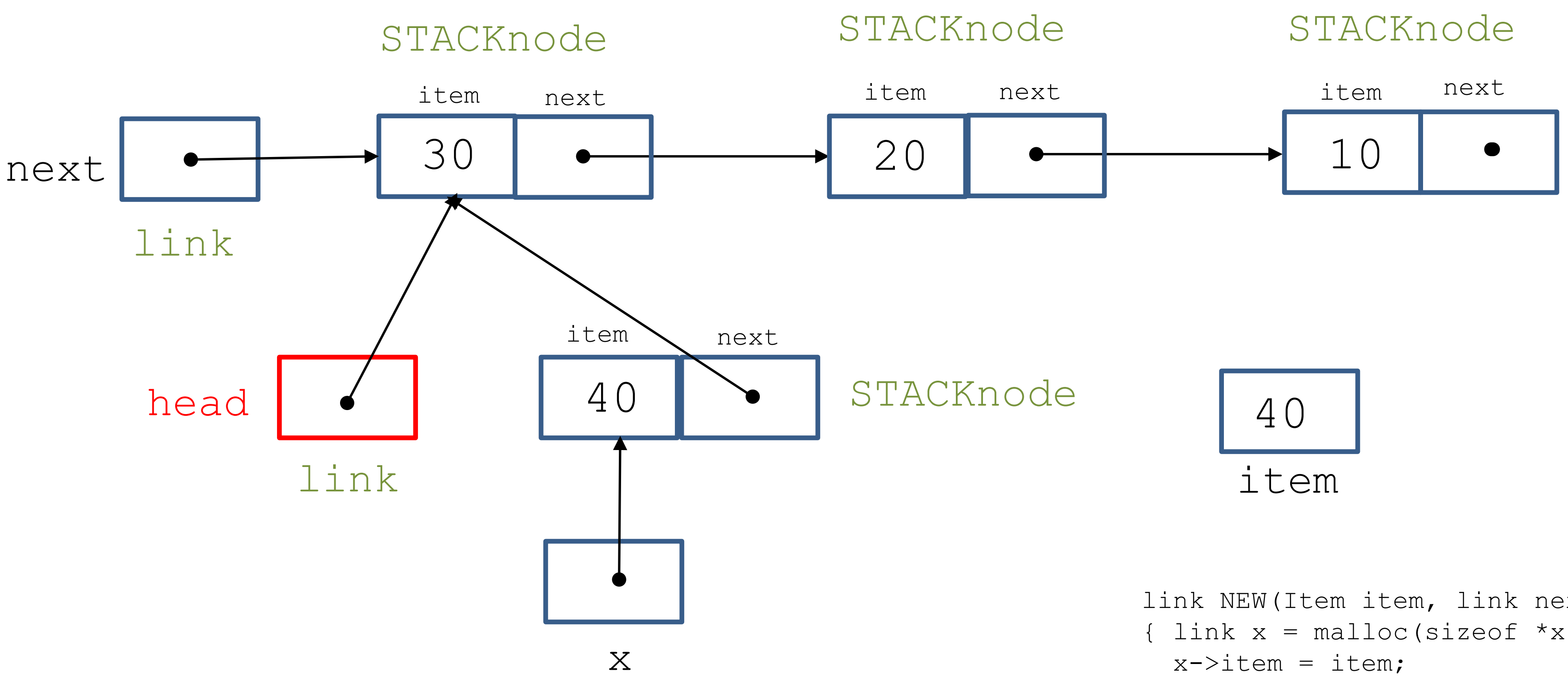

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

Example (cont'd)



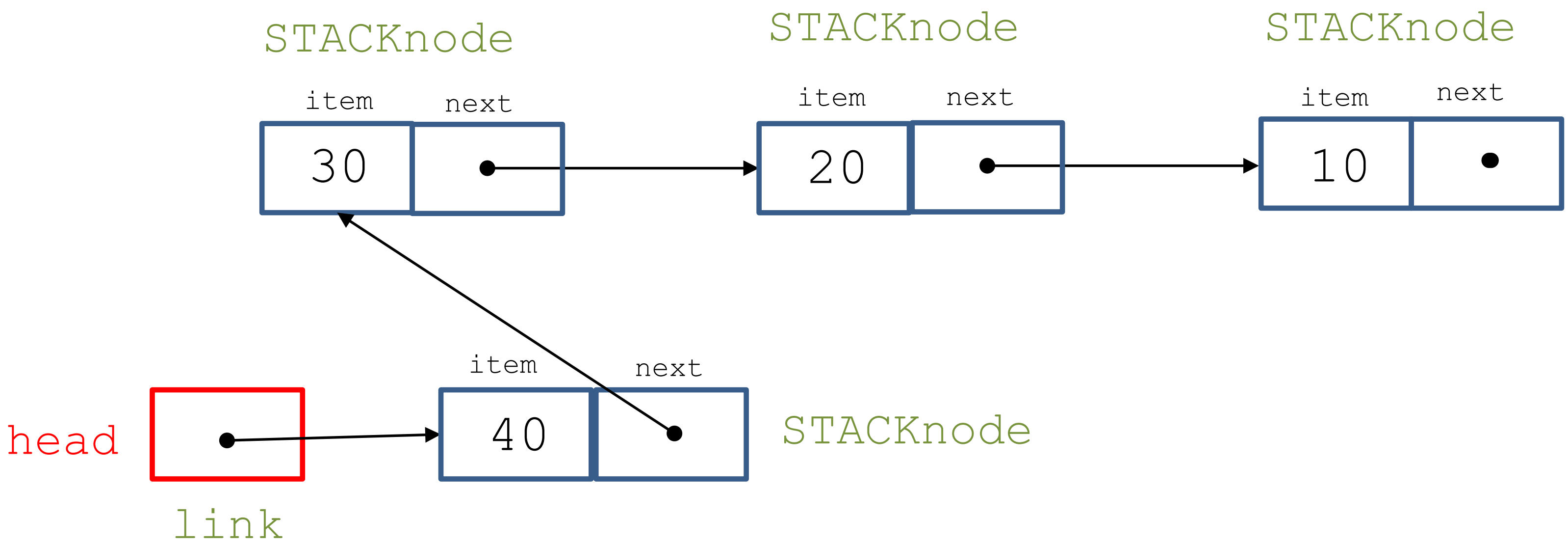

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

Example (cont’d)



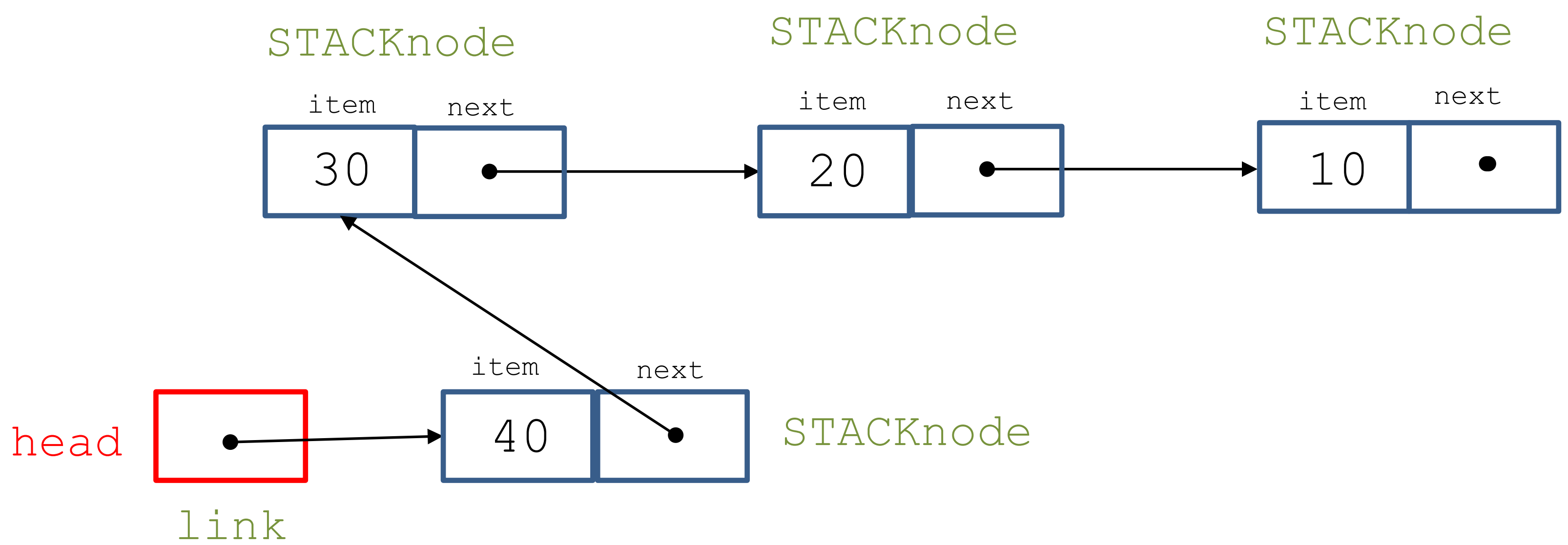

```
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

Example (cont'd)



`head = NEW(40, head);`

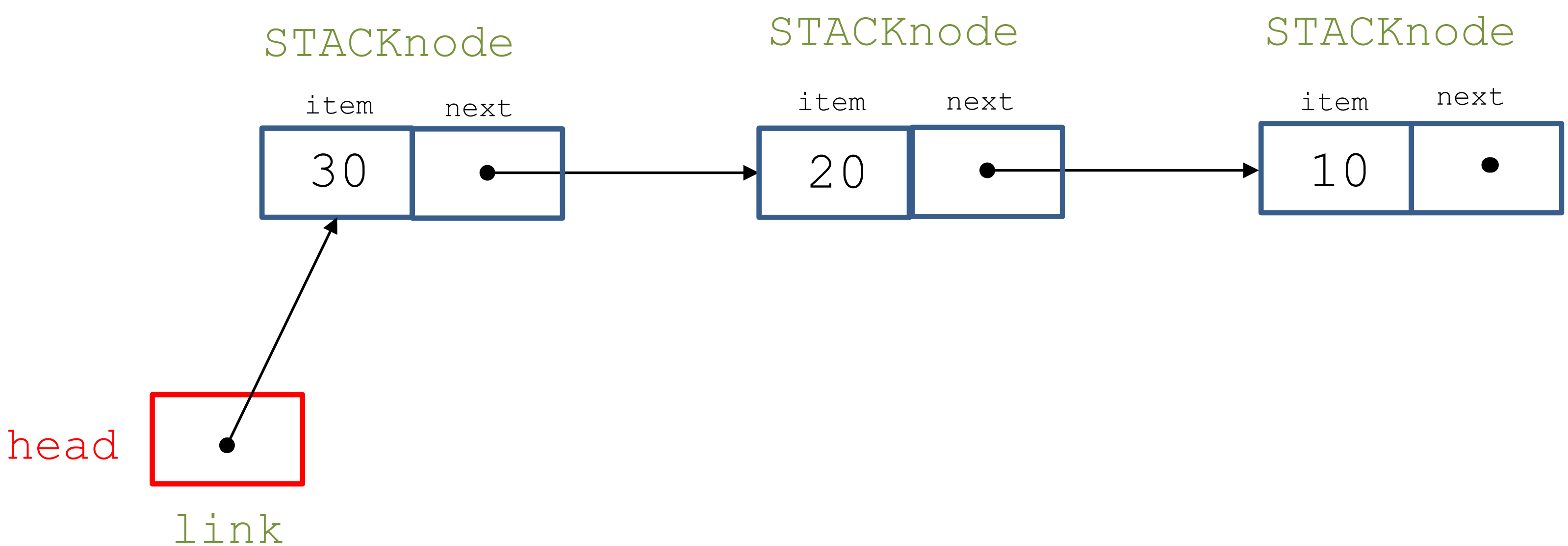
Result


STACKnode
item
next
30
STACKnode
item
next
20
STACKnode
item
next
10
item
next
head
40
STACKnode
link

The Linked List Implementation (cont’d)

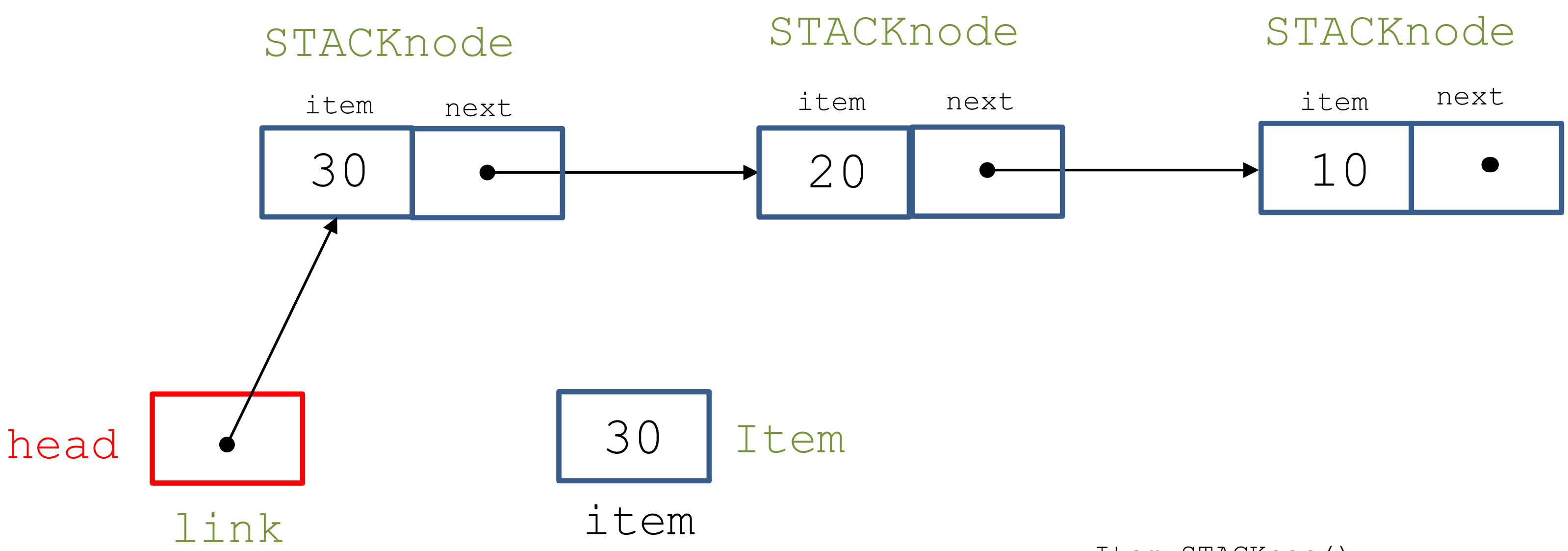
```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Example



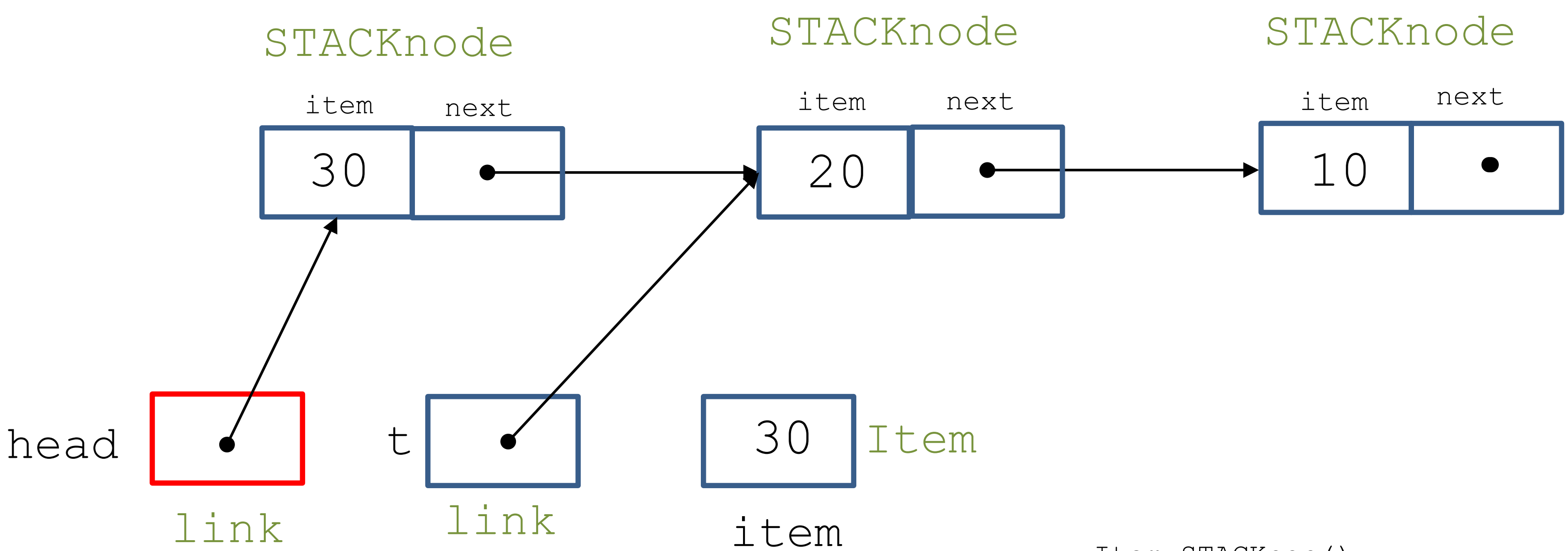
STACKnode
STACKnode
STACKnode
item
next
item
next
item
next
30
20
10
head
link

Example of the Execution of the Call `STACKpop()`



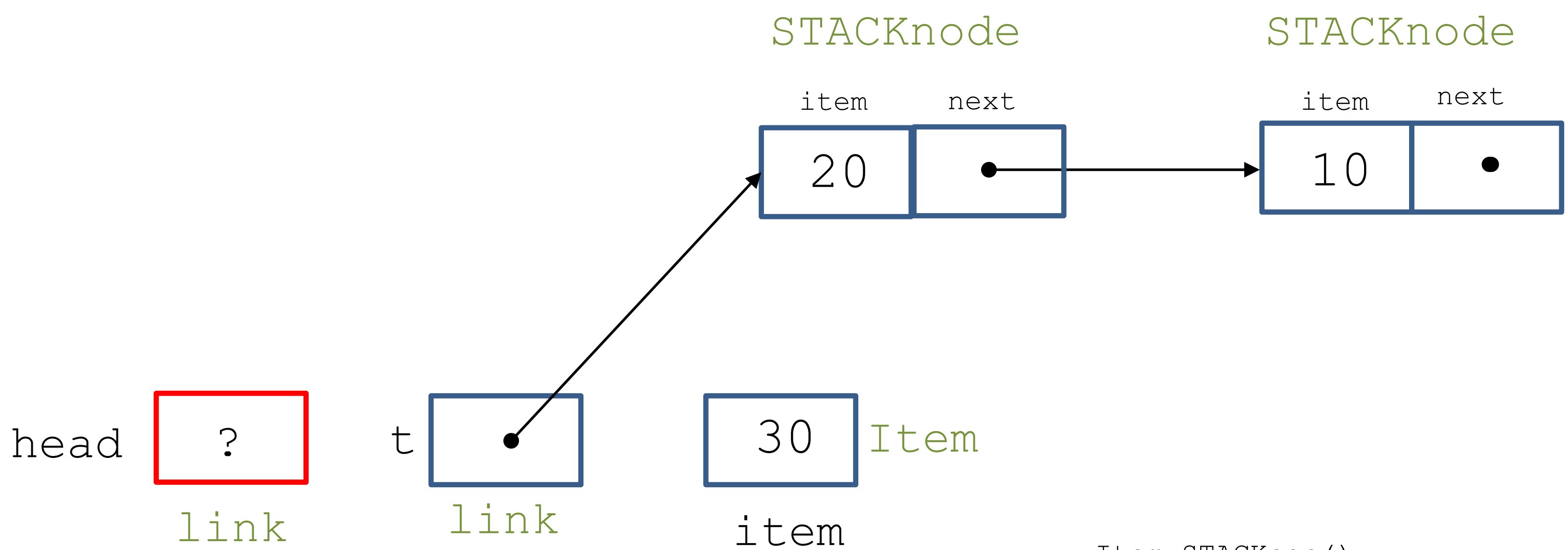

```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Example (cont'd)



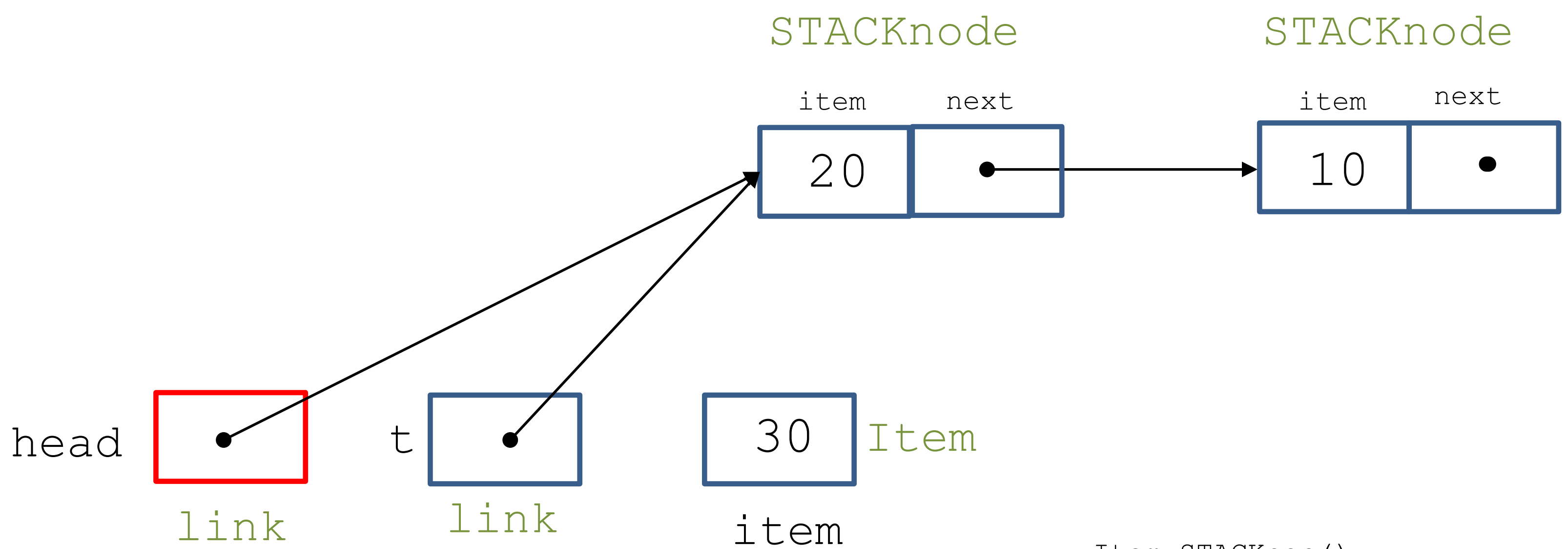

```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Example (cont'd)



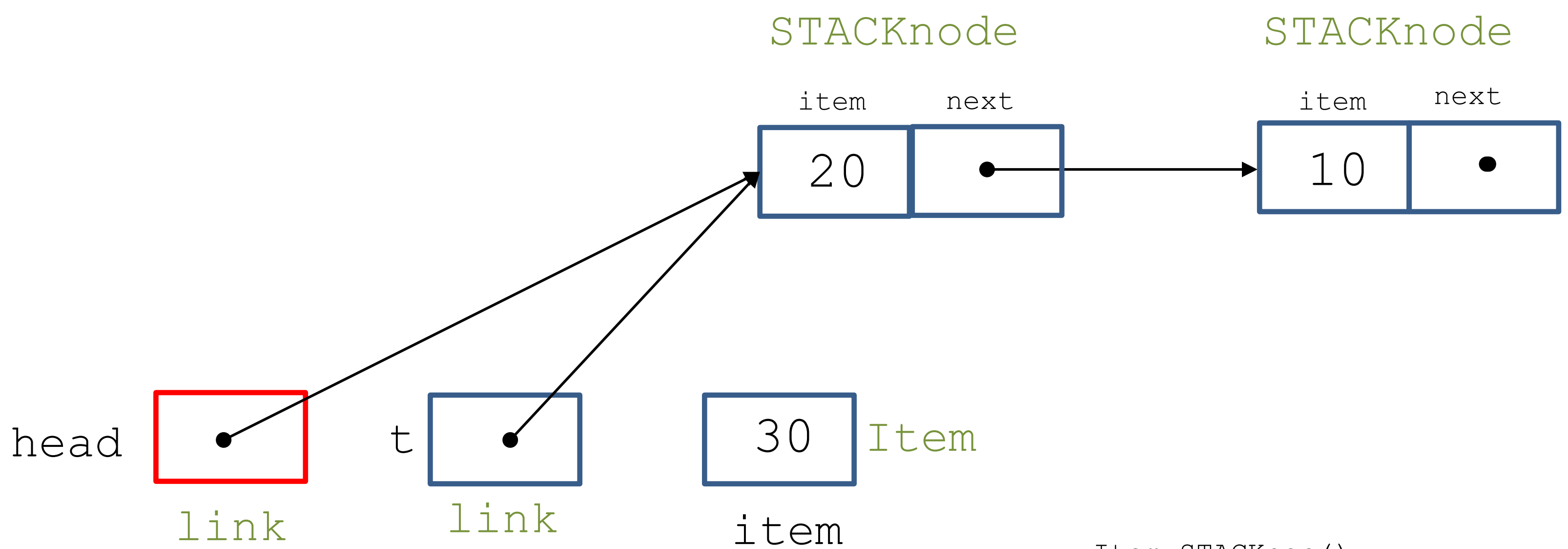

```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Example (cont'd)



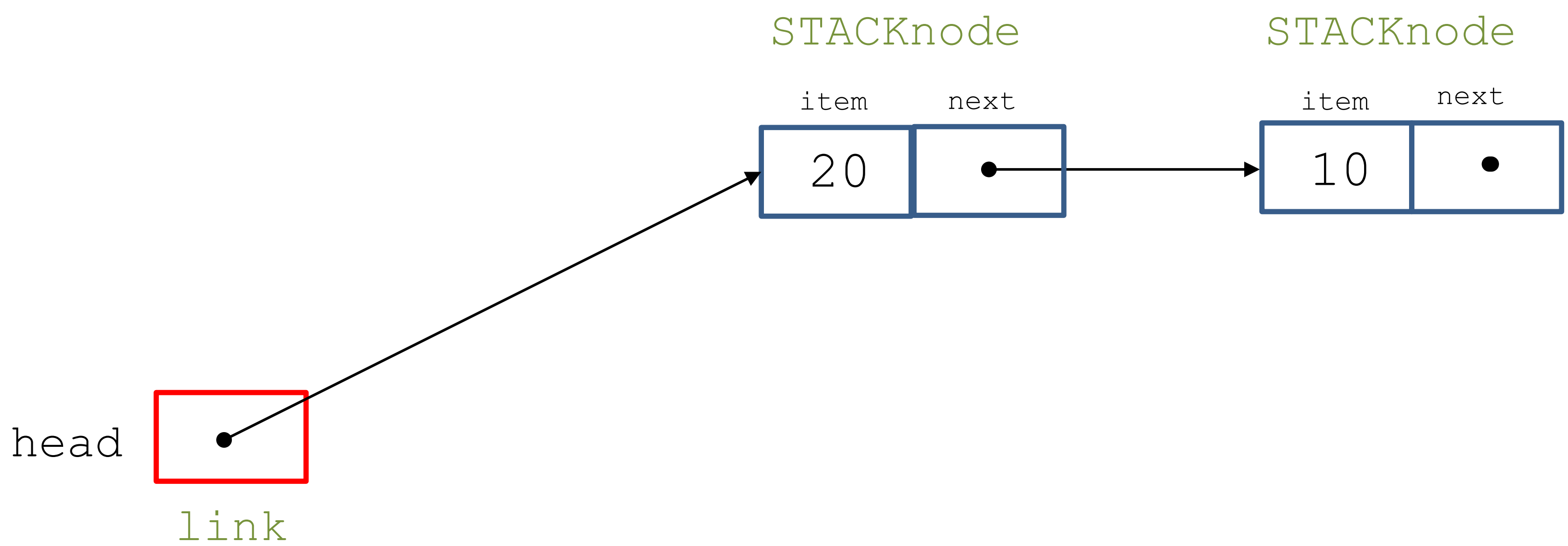

```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Example (cont'd)




```
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Result



Notes

- The implementation of stacks shown in the previous slides uses an auxiliary function `NEW` to allocate memory for a node, set its fields from the function arguments, and return a link to the node.
- In this implementation, **we keep the stack in the reverse order of the array implementation**; from most recently inserted elements to least recently inserting elements.
- **Information hiding:** For both implementations (with arrays or linked lists), the data structure for the representation of the stack (array or linked list) is defined **only** in the implementation file thus it is not accessible to client programs.

Translating Infix Expressions to Postfix

- Let us now use the latest implementation of the stack ADT to implement a translator of **fully parenthesized** infix arithmetic expressions to postfix.
- The **recursive algorithm** for doing this is as follows. To convert (A+B) to the postfix form AB+, we ignore the left parenthesis, convert A to postfix, save the + on the stack, convert B to postfix, then, on encountering the right parenthesis, pop the stack and output the +.

Example

- We want to translate the infix expression `((5*(9+8))+7)` into postfix.
- What is the result?

Example (cont'd)

- We want to translate the infix expression `((5*(9+8))+7)` into postfix.
- The result will be `5 9 8 + * 7 +`.

Executing the Algorithm

Input	Output	Stack
(		
(		
5	5	
*		*
(		*
9	9	*
+		* +
8	8	* +
)	+	*
)	*	
+		+
7	7	+
)	+	

The Client Program

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"

main(int argc, char *argv[])
{
    char *a = argv[1];
    int i, N = strlen(a);

    STACKinit(N);
    for (i = 0; i < N; i++)
    {
       if (a[i] == ')')  printf("%c ", STACKpop());
       if ((a[i] == '+')  ||  (a[i] == '*'))   STACKpush(a[i]);
       if ((a[i] >= '0')  && (a[i] <= '9'))  printf("%c ", a[i]);
    }
    printf("\n");
}
```

The File `Item.h`

- The file `Item.h` can only contain a `typedef` which defines the type of items in the stack.
- For the previous program, this can be:

```
typedef char Item;
```

A Weakness of the 2nd Solution

- The 2nd solution for defining and implementing a stack ADT is weaker than the 1st one since it allows the construction and operation of a **single stack** by a client program.
- Conversely, the 1st solution allows us to define **many stacks** in the client program.

Exercise

- Modify the 1st solution so that it does better information hiding without losing the capability to be able to define many stacks in the client program.

Question

- Which implementation of a stack ADT should we prefer?

Answer

- **It depends on the application.**

- In the linked list implementation, push and pop take more time to allocate and de-allocate memory.
- If we need to do these operations a huge number of times then we might prefer the array implementation.
- On the other hand, the array implementation uses the amount of space necessary to hold the maximum number of items expected. This can be wasteful if the stack is not kept close to full.
- The list implementation uses space proportional to the number of items but always uses extra space for a link per item.

- Note also that the running time of push and pop in each implementation is constant.

How C Implements Recursive Function Calls Using Stacks

- When calling an instance of a function $F(a_1,a_2,\ldots,a_n)$ with actual parameters $a_1, a_2,\ldots,a_n$, C uses a **run-time stack**.
- A collection of information called a **stack frame** or **call frame** or **activation record** is prepared to correspond to the call and it is placed on top of other previously generated stack frames on the run-time stack.

Stack Frames

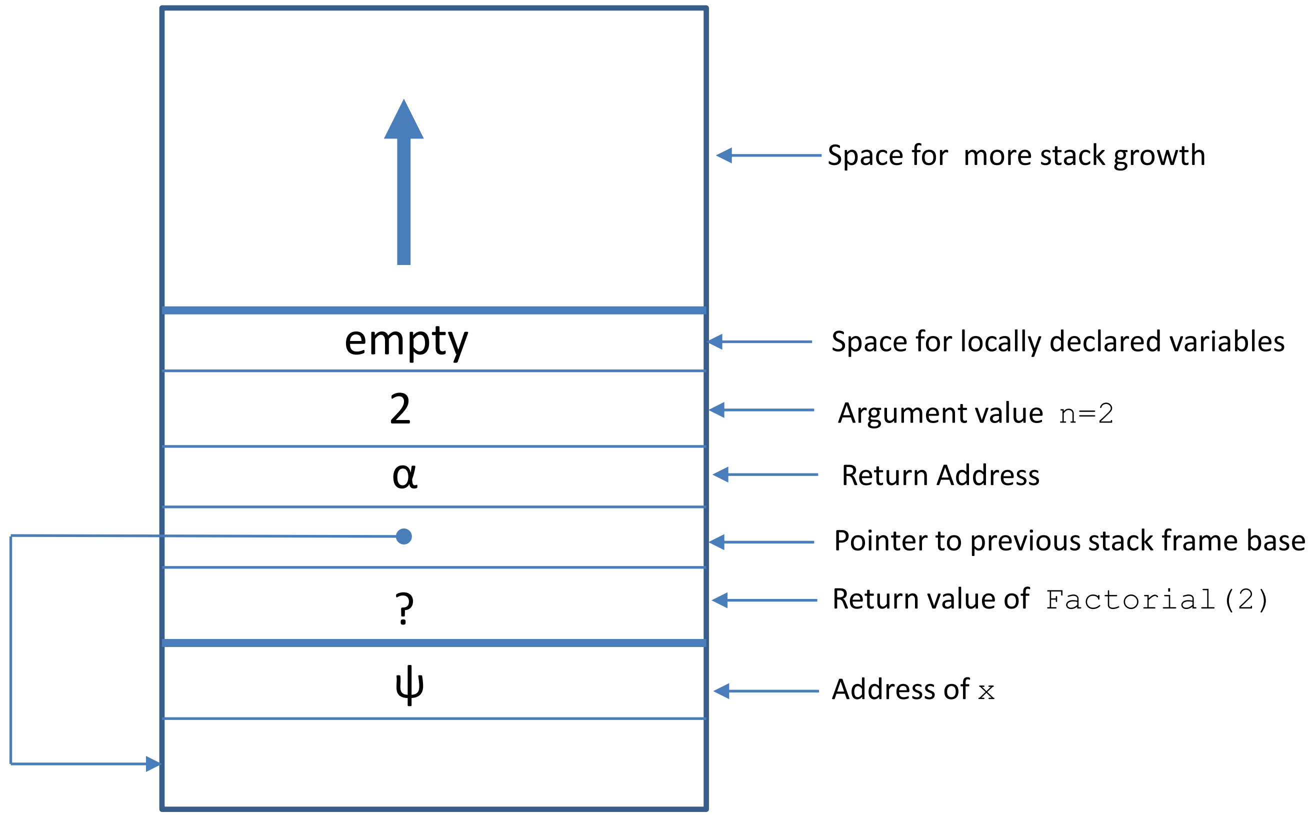
- The information in a stack frame consists of:
  - Space to hold the **value returned by the function**.
  - A **pointer to the base of the previous stack frame** in the stack.
  - A **return address**, which is the address of an instruction to execute in order to resume the execution of the caller of the function when the call has terminated.
  - Parameter storage sufficient to hold **the values of the arguments** used in the call.
  - A set of storage locations sufficient to hold the values of the **variables declared locally** in the function.

Example - Factorial

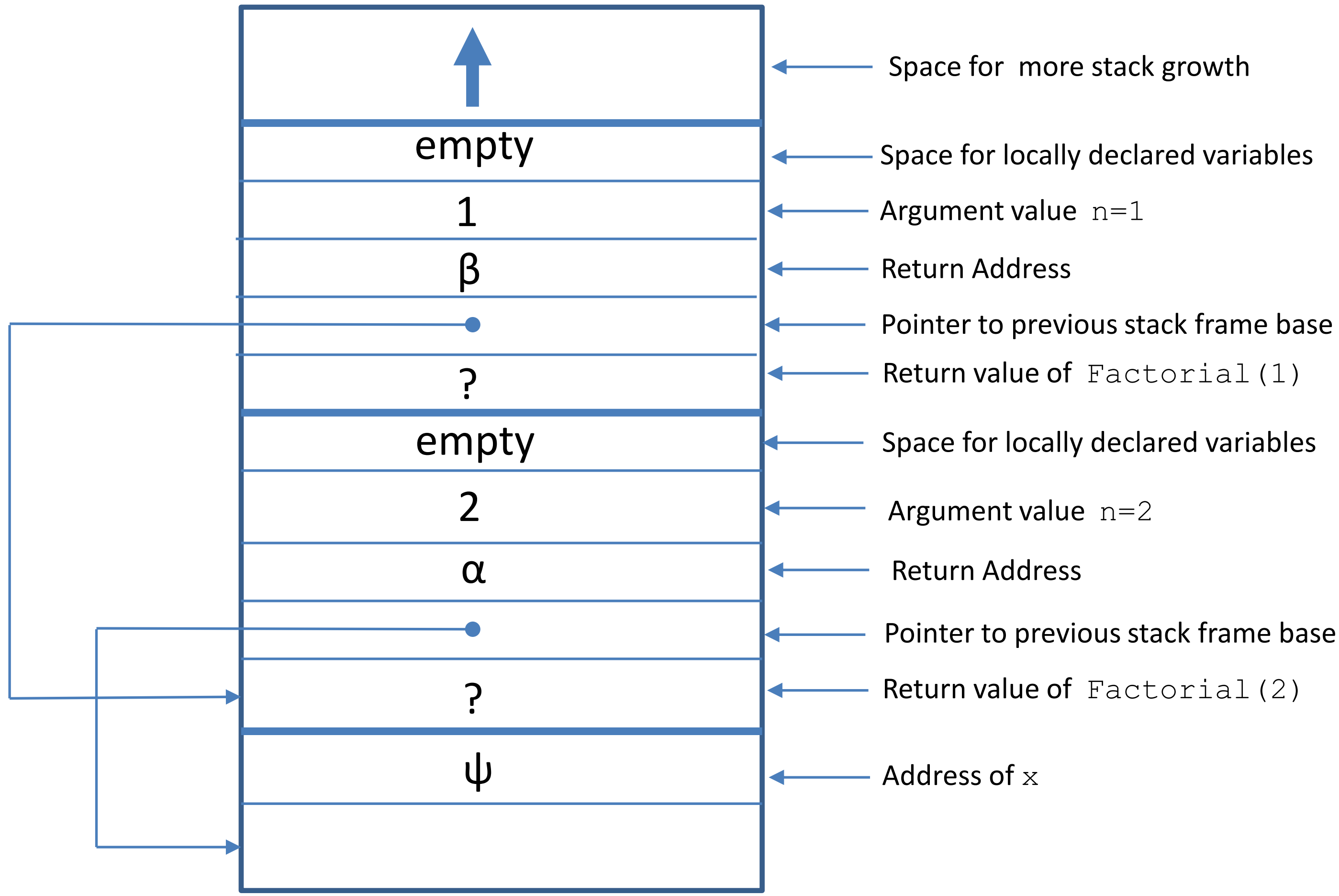
```
int Factorial(int n);
{
   if (n==1) {
      return 1;
   } else {
      return n*Factorial(n-1);
   }
}
```

Let us consider the call `x=Factorial(2).`

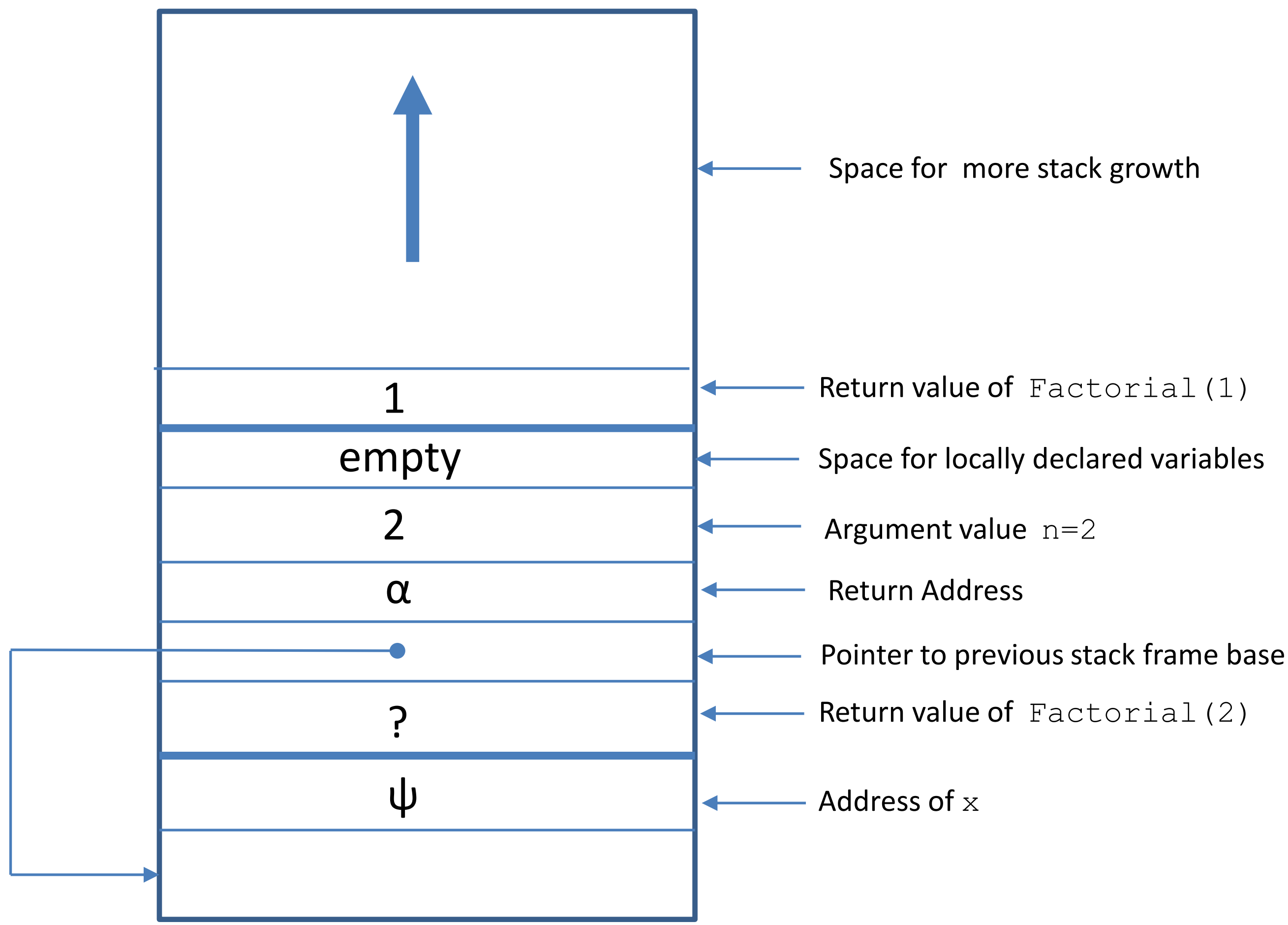
Stack Frame for `Factorial(2)`



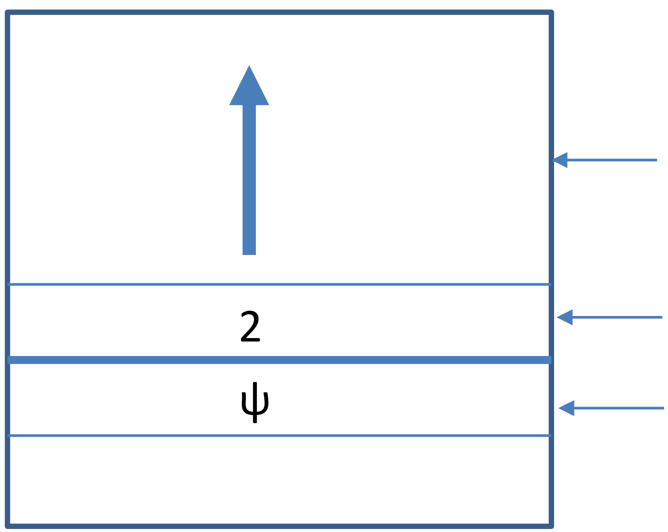
Stack Frame for `Factorial(2)` and `Factorial(1)`



Stack After Return from `Factorial(1)`



Stack After Return from `Factorial(2)`



Space for more stack growth

Return value of `Factorial(2)`

Address of `x`

More Details

- **Stack + Iteration** can implement **Recursion**.

- Run-time stacks are discussed in more details in the courses Computer Architecture and Compilers.

Using Stacks

- Generally speaking, stacks can be used to implement any kind of **nested structure**.
- When processing nested structures, we can start processing the outermost level of the structure, and if we encounter a nested substructure, we can interrupt the processing of the outer layer to begin processing an inner layer by putting a record of the interrupted status of the outer layer's processing on top of a stack.
- In this way the stack contains **postponed obligations** that we should resume and complete.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

  Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C.

  Κεφ. 4

- As we also said for other lectures, the code that does not do good information hiding is from the first book, while the code that does good information hiding is from the second one.