Lecture 5.2 – Data Structures

Linked Stacks

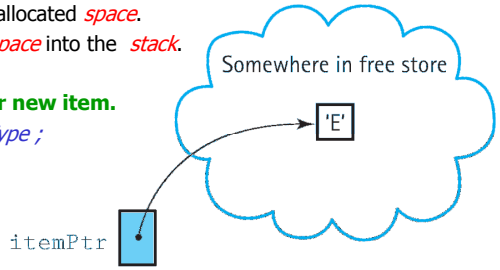
Stack as a Linked Structure

- The implementation of a *stack in an array* is simple but the *size* of the array must be determined when a stack *object is declared*.
- *Dynamic allocation of each stack element* – one at a time – allows us to do get space for stack element when we need it.

Function *Push*

- *Allocate space* for new *item*.
- *Put new item* into allocated *space*.
- *Put* the allocated *space* into the *stack*.

// **Allocate space for new item.**
ItemPtr = new ItemType ;



The *new* operator allocates a block of memory big enough to hold a value of type *ItemType* (the type of data contained in the stack) and returns the block's address, which is copied into the variable *ItemPtr*. We can now use the dereference operator (** ItemPtr*) to put *newItem* into the space that was allocated: **itemPtr = newItem* - *newItem* is '*E*'.

Third part of the *push* operator

- result of calling push to add the characters '*D*', '*A*', and '*L*' to the stack.

•Data are in the dynamically allocated space, but is *not a stack*. There is *no order.*

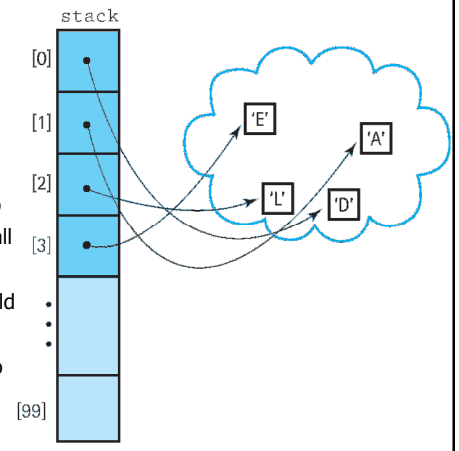
•Even worse, because we haven't returned the pointers to the dynamically allocated space from function *push*, we have *no way to access any of the elements* any more.

•Clearly, the *third part* of the *push* operation *needs to do something* to fix this situation.

• *Where* can we *store the pointers* to our data?

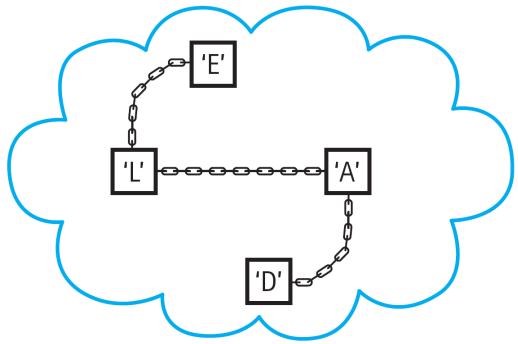
Stack as an array of pointers

- One possibility is to declare a stack as an *array of pointers* and to put the pointer to each new item into this array.
- This solution would keep track of the pointers to all the elements in the correct order, but it would *not solve our original problem:* we still need to declare an *array* of a *particular size.*



Linked stack

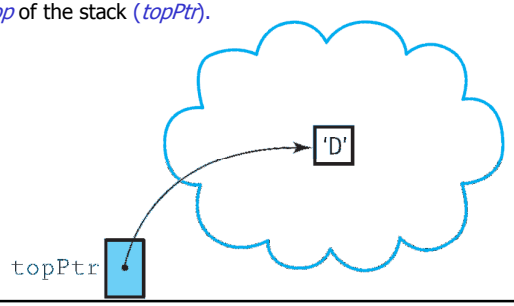
- Another possibility is to *chain all the elements together*.
- We call each *element* in this *linked stack* a ***node***.



Push the character '*D*'

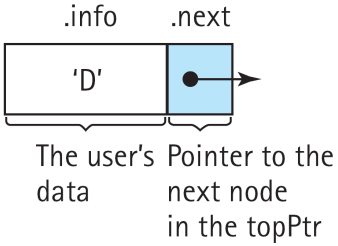
Use link idea to implement the stack.

- First we *push* the character '*D*'. *Push* uses operator *new* to allocate space for the new node, and *puts 'D' into the space*.
- We don't want to lose the pointer to this element, so we *need a data member* in our stack class in which to store the *pointer to the top* of the stack (*topPtr*).

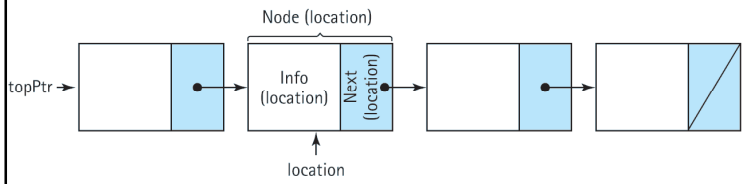


Chain '*A*' to '*D*'

- Next we want to chain '*A*' to '*D*', the old stack element.
- Let's make each *node* in the stack contain *two parts*: *info* and *next*.
- The *info* member contains the stack's *user data* – e.g. *character*.
- The *next* element contains the *address of the next node* in the stack.

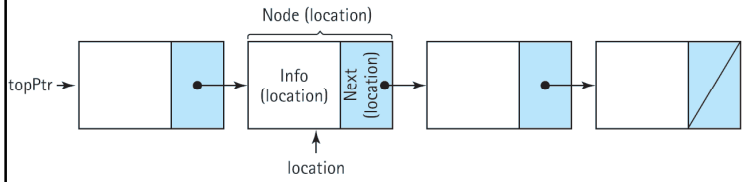


- The *next* member of each member points to the *next node* in the stack.
- The next member of the *last node points to NULL*. It marks the *end of the stack.*



Terminology

Node(location)	refers to *all the data* at *location*.
Info(location)	refers to the *user's data* at *location*.
Info(last)	refers to the *user's data* at the *last location*.
Next(location)	gives the location of the node following *Node(location).*



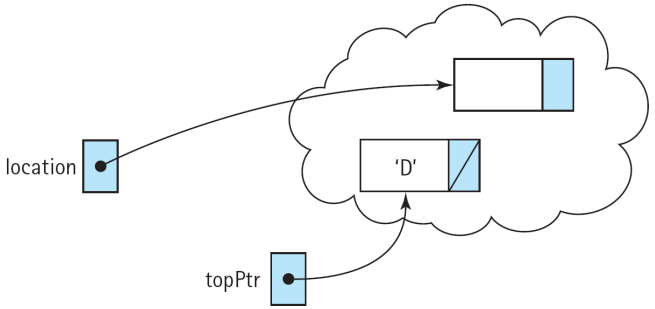
- In our pointer-based implementation *location* must be a *pointer to a record that contains both the user's information and a pointer to the next node* on the stack.

push algorithm

- Now let's return to our *push* algorithm. We have allocated a node to contain the new element *'A'* using operator *new* .

Set location to address of a new node of type ***ItemType***

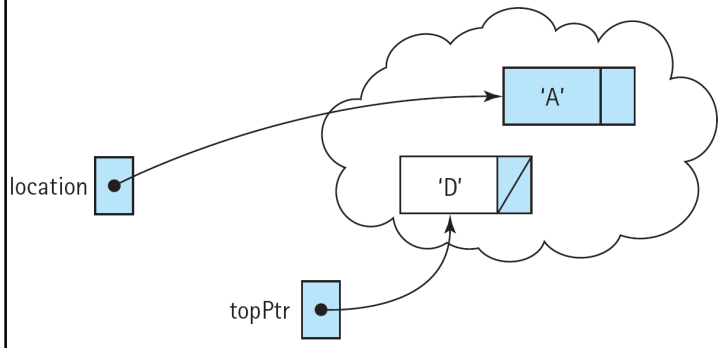
// Allocate space for new item



Then the new value, 'A', is put into the node

Set Info(location) to *newItem*

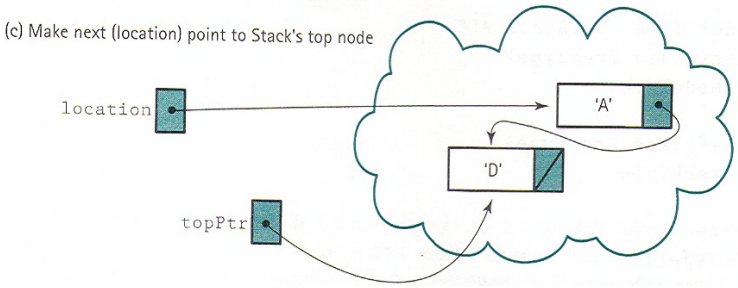
// Put new item into the allocated space

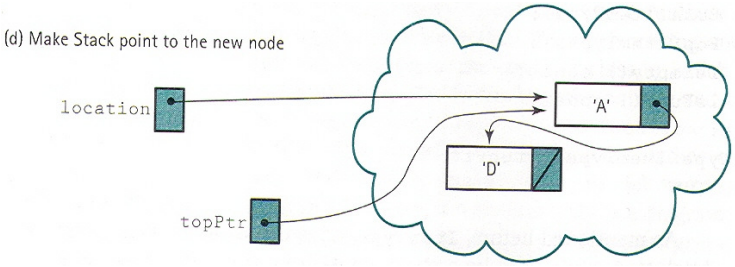


Linking the new node to the (previous) top node in the stack is a two-step process:

Make ***Next(location)*** **point to the stack's top node**

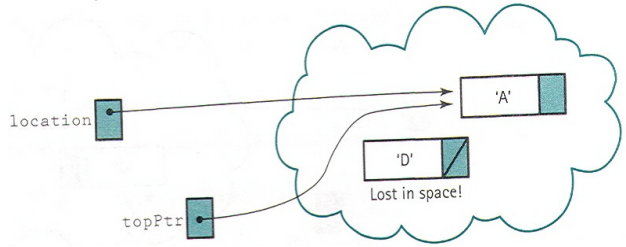
Make *topPtr* point to the new node





Note that the order of these tasks is critical

If we *change* the *topPtr* pointer *before* making *Next(location)* point to the *top* of the stack, we would have *lost access to the stack nodes*.



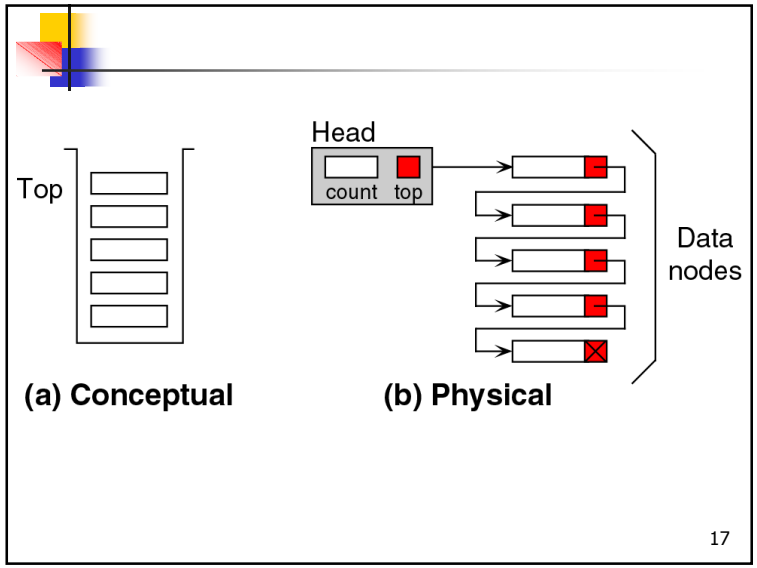
•From the stack user's point of view, nothing has changed.
•The prototype for member function *push* is the *same* as it was for the *array-based implementation*.

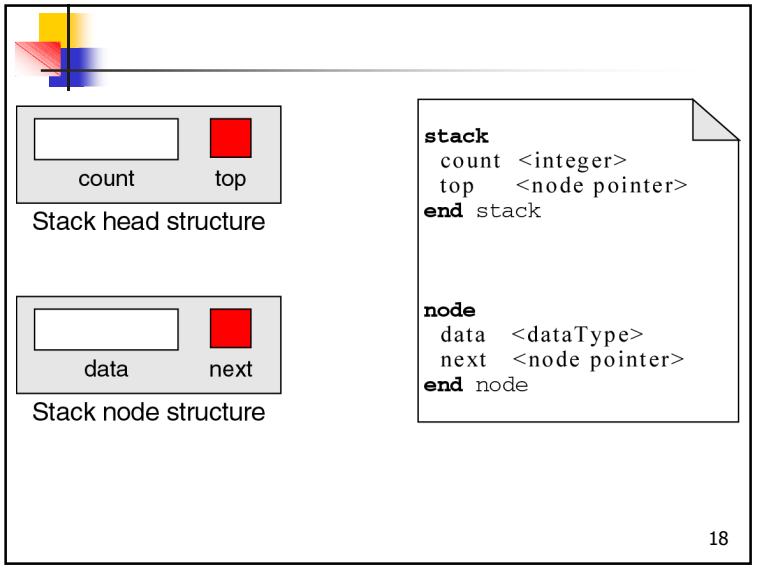
void push(ItemType newItem) ;

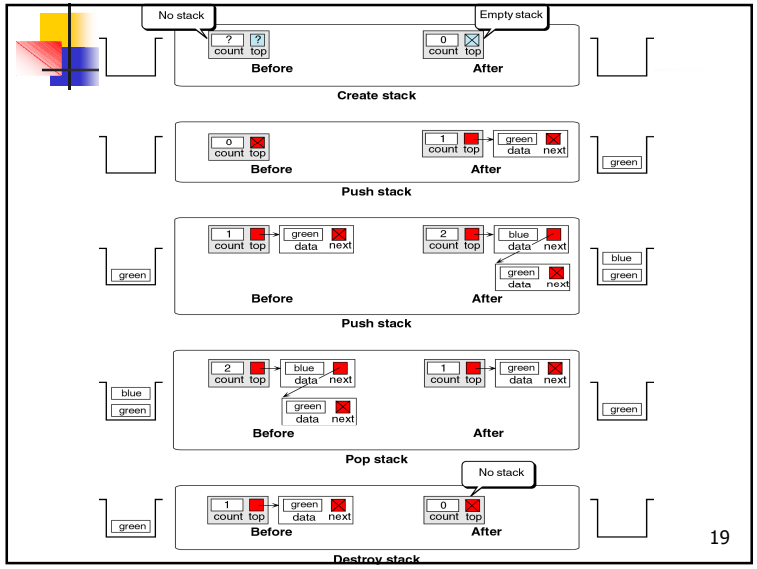
ItemType is still the *type of data* that the user wants to put in the stack.

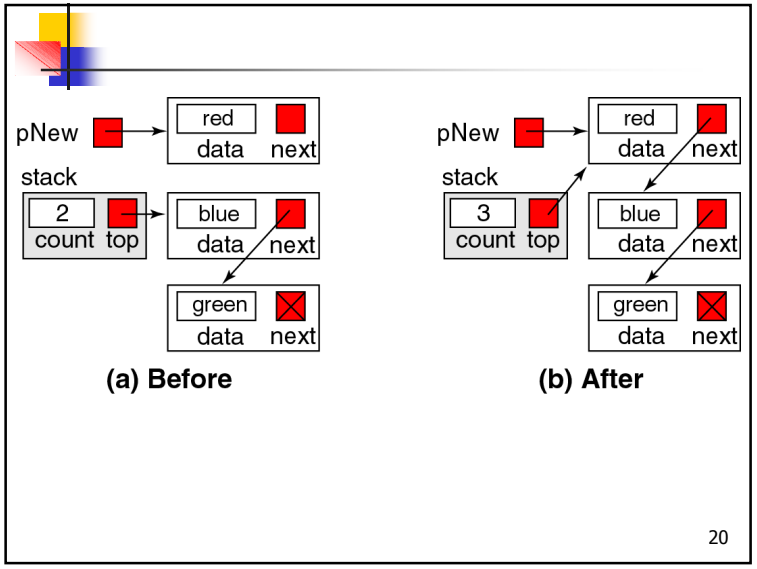
Class *StackType*, needs *new definitions*.

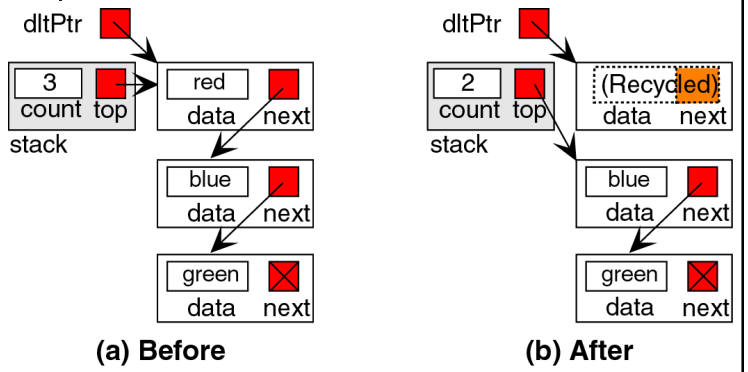
- It no longer is a class with a *top* member and an array member to hold the items; its only data member is *topPtr*, the pointer to a single node, the top of the stack.
- The node to which *topPtr* points has two parts, *info* and *next*, which suggests a C++ *struct* or *class* representation.
- We choose to make *NodeType* a *struct* rather than a class, because the *nodes* in the structure are *passive*.
- They are acted upon by the member functions of *StackType*.


Top
Head
count top
Data nodes
(a) Conceptual
(b) Physical
17


count top
Stack head structure
data next
Stack node structure
stack
count <integer>
top <node pointer>
end stack
node
data <dataType>
next <node pointer>
end node
18


No stack
Empty stack
Before
After
Create stack
Push stack
Push stack
Pop stack
Destroy stack
19


pNew
stack
red
blue
green
data next
2
3
count top
(a) Before
(b) After
20



// Stack4.h: header file for Stack ADT

```
// Stack4.h: header file for Stack ADT
// Class is templated.
template <class ItemType>
struct NodeType
{
   ItemType info ;
   NodeType  *next ;
} ;
```

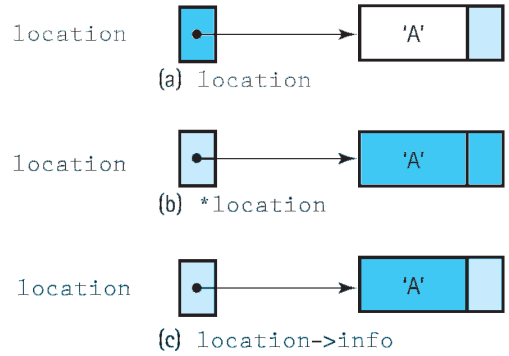


```
template<class ItemType>
class StackType
{
public:
   StackType();
   // Class constructor.
   ~StackType();
   // Class destructor.
   void MakeEmpty();
   // Function:  Sets stack to an empty state.
   // Post: Stack is empty.
   bool IsFull() const;
   // Function: Determines whether the stack is full.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is full)
   bool IsEmpty() const;
   // Function: Determines whether the stack is empty.
   // Pre:  Stack has been initialized.
   // Post: Function value = (stack is empty)
```



```
   void Push(ItemType item);
      // Function: Adds newItem to the top of the stack.
      // Pre:  Stack has been initialized and is not full.
      // Post: newItem is at the top of the stack.
      void Pop(ItemType& item);
      // Function: Removes top item from the stack and returns it in item.
      // Pre:  Stack has been initialized and is not empty.
      // Post: Top element has been removed from stack. item is a copy of
      the removed item.
private:
   NodeType<ItemType>* topPtr;
};
#include "stack4.cpp"
```

- *location* is a pointer to a node containing an *info* member and a *next* member.
- **location* - reference this node; it is a *struct* with two members. We can *access its members* in the usual way – (**location*).*info* or the → operator, which (dereferences the pointer and accesses a member) *location* → *info*.

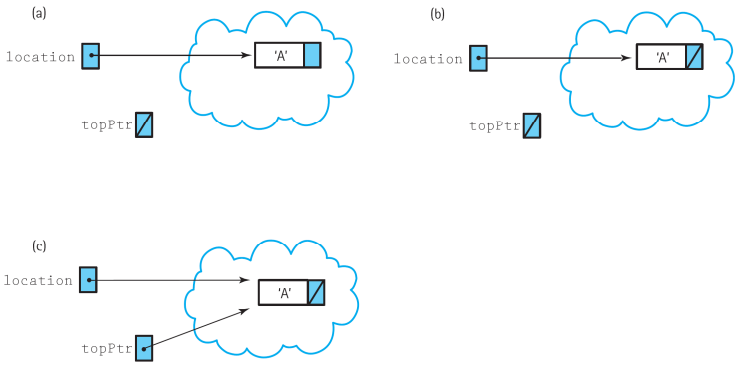


Summary:

- *StackType* is a *class* with only *one data member:* a *pointer to the top* node in the stack. With our dynamically allocated linked structure, there is no specific limit on the size of the stack.

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  NodeType<ItemType>* ptr;
  ptr = new NodeType<ItemType>;
  ptr->info = newItem;
  ptr->next = topPtr;
  topPtr = ptr;
}
```

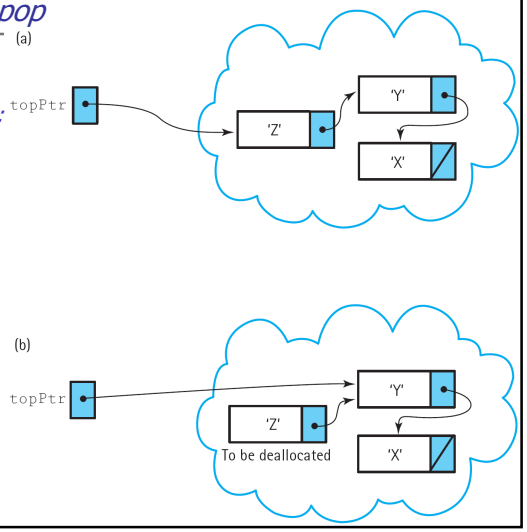

code on an empty stack



Function *pop*

The algorithm for *pop:*

- Set item to *Info(top_node).*
- *Unlink* the *top_node* from the stack.
- *Deallocate* the old *top_node.*

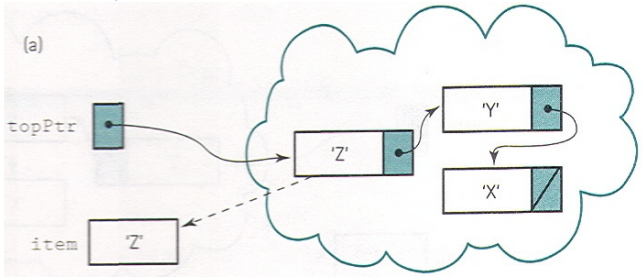


Function *pop* – problem in the algorithm

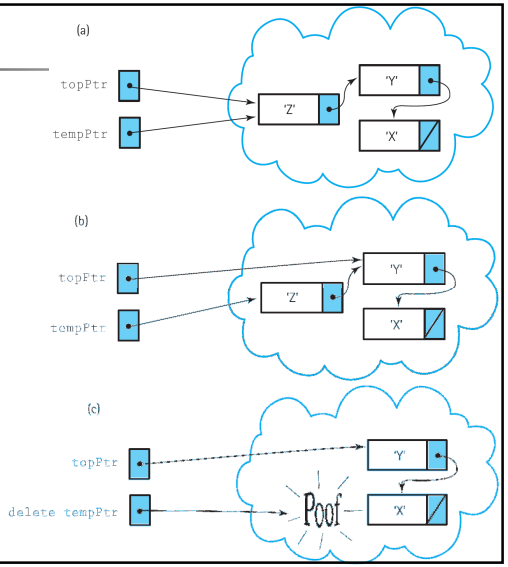
The algorithm for *pop* is as follows:

- Set *item* to Info(*top node*).
- *Unlink* the top node from the stack.
- Deallocate the old top node.

- *No pointer to deallocate the node*

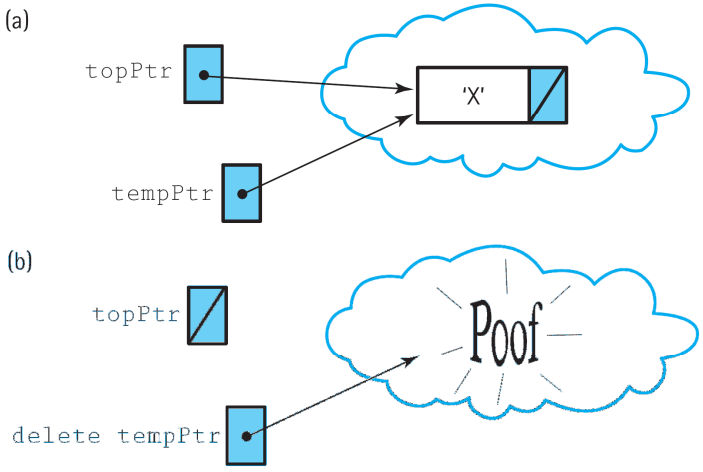


- Need *temporary pointer* to deallocate the node




```
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
    NodeType<ItemType>* tempPtr;
    tempPtr = topPtr;
    item = topPtr ->info;
    topPtr = topPtr->next;
    delete tempPtr;
}
```


Only one node in the stack when *pop* is called



the stack is *empty* when *pop* is called

- If *topPtr* contains *NULL*, then the assignment statement

  topPtr = topPtr → next;

results in a run-time error. On some systems you get a message ATTEMPT TO DEREFFERENCE NULL POINTER; on other systems the screen freezes

- So the *pop* function is not required or expected to *protect the client* from this situation.
- The client is responsible for checking for an *empty* stack before the call to *pop*. In fact, this is why we provide the *IsEmpty* function.

Other Stack Functions

```
template<class ItemType>
StackType<ItemType>::StackType()
{
   topPtr = NULL;
}
template <class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
   return (topPtr == NULL);
}
template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
   NodeType<ItemType>* ptr;
   ptr = new NodeType<ItemType>;
   if (ptr == NULL)
      return true;
   else
   {
      delete ptr;
      return false;
   }
}
```


MakeEmpty

While more nodes in the stack

- ***Unlink top node***
- ***Deallocate the node***

```
template <class ItemType>
void StackType<ItemType>::MakeEmpty()
{
   NodeType<ItemType>* tempPtr;
   while (topPtr != NULL)
   {
      tempPtr = topPtr;
      topPtr = topPtr->next;
      delete tempPtr;
    }
}
```


Destructor

```
template <class ItemType>
StackType<ItemType>::~StackType()
{
   NodeType<ItemType>* tempPtr;
   while (topPtr != NULL)
   {
      tempPtr = topPtr;
      topPtr = topPtr->next;
      delete tempPtr;
    }
}
```

```
// Implementation file for Stack ADT.
// Class definition is in "Stack4.h".
// Class is templated; implementation is linked.
#include <stddef.h>
template <class ItemType>
struct NodeType
{
   ItemType info;
   NodeType<ItemType>* next;
};
template<class ItemType>
StackType<ItemType>::StackType()
{
    topPtr = NULL;
}
```

37

```
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
   NodeType<ItemType>* tempPtr;
   tempPtr = topPtr;
   item = topPtr ->info;
   topPtr = topPtr->next;
   delete tempPtr;
}
template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
   NodeType<ItemType>* ptr;
   ptr = new NodeType<ItemType>;
   if (ptr == NULL)
      return true;
   else
   {
      delete ptr;
      return false;
   }
}
```

38

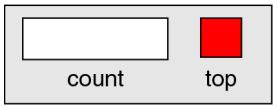
```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
   NodeType<ItemType>* ptr;
   ptr = new NodeType<ItemType>;
   ptr->info = newItem;
   ptr->next = topPtr;
   topPtr = ptr;
}
template <class ItemType>
void StackType<ItemType>::MakeEmpty()
{
   NodeType<ItemType>* tempPtr;
   while (topPtr != NULL)
   {
      tempPtr = topPtr;
      topPtr = topPtr->next;
      delete tempPtr;
    }
}
```

39

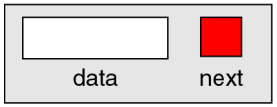
```
template <class ItemType>
StackType<ItemType>::~StackType()
{
   NodeType<ItemType>* tempPtr;
   while (topPtr != NULL)
   {
      tempPtr = topPtr;
      topPtr = topPtr->next;
      delete tempPtr;
    }
}
template <class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
   return (topPtr == NULL);
}
```

40

Other implementations

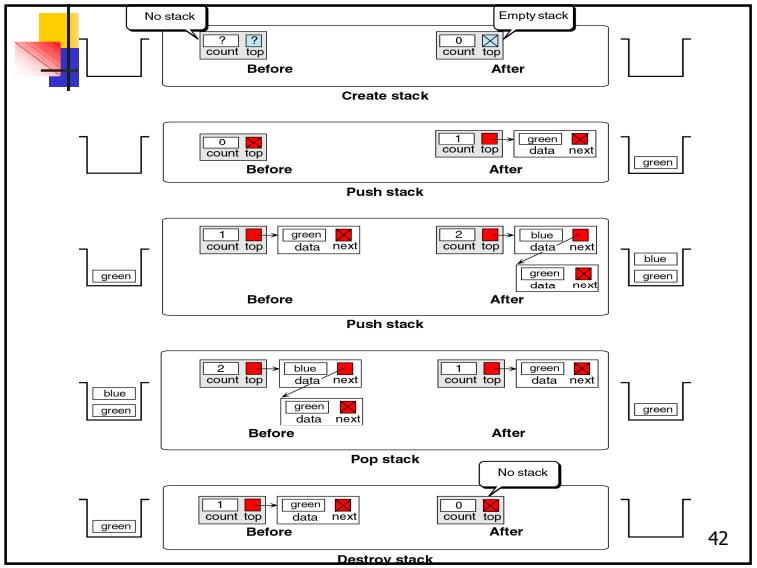


Stack head structure

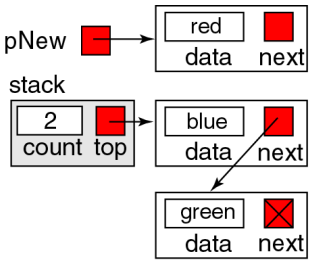


Stack node structure

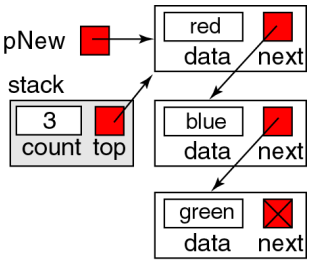
```
stack
  count  <integer>
  top    <node pointer>
end stack


node
  data   <dataType>
  next   <node pointer>
end node
```

Push

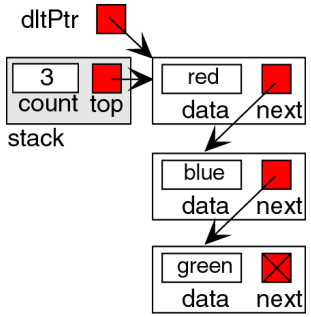


(a) Before

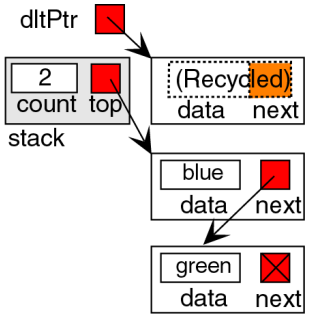


(b) After

Pop



(a) Before



(b) After

Stack Applications

Stack Applications

- Reversing Data.
  - Changing the sequence of data.
  - Converting from decimal to Bin, Hex and Oct.
- Parsing Data.
  - Matching Parenthesis.
- Postponing Data.
  - Prefix, Infix and Postfix.
- Backtracking.
  - Goal seeking.

Category of algorithms

- **Backtracking** - Backtracking is a *refinement of the brute force approach*, which systematically *searches for a solution to a problem among all available options*.
- It does so by *assuming* that the solutions are represented by *vectors* (v_1, ..., v_m) of *values* and by *traversing, in a depth first manner*, the domains of the vectors *until the solutions are found*.

Convert decimal to binary

```
1  read (number)
2  loop (number > 0)
     1  digit = number modulo 2
     2  print (digit)
     3  number = number / 2
```

Converting From Decimal

Convert to Binary

56/2 = 28 rem 0
28/2 = 14 rem 0
14/2 = 7 rem 0
7/2 = 3 rem 1
3/2 = 1 rem 1
½ = 0 rem 1

= 1 1 1 0 0 0

Convert to Hexadecimal

196/16 = 12 rem 4
12/16 = 0 rem 12

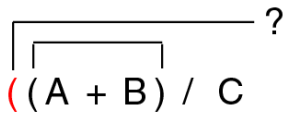
= C 4

- Converting Decimal to Binary(B), Octal(Q), Hexadecimal(H).

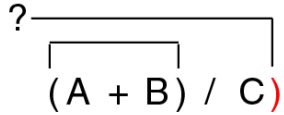
49

```
program decimalToBinary
   This algorithm reads an integer from the keyboard and
   prints its binary equivalent. It uses a stack to reverse
   the order of 0's and 1's produced.

 1 stack = createStack
 2 prompt (Enter a decimal to convert to binary)
 3 read (number)
 4 loop (number > 0)
   1  digit    = number modulo 2
   2  pushOK   = push (stack, digit)
   3  if (pushOK false)
      1  print (Stack overflow creating digit)
      2  quit algorithm
   4  number = number / 2

   Binary number created in stack. Now print it.
 5 loop (not emptyStack(stack))
   1  popStack (stack, digit)
   2  print (digit)

   Binary number created. Destroy stack and return.
 6 destroy (stack)
end decimalToBinary
```

Matching parenthesis



(a) Opening parenthesis not matched



(b) Closing parenthesis not matched

51

```
program parseParens
   This algorithm reads a source program and parses it to make
   sure all opening-closing parentheses are paired.
 1 loop (more data)
   1  read (character)
   2  If (character is an opening parenthesis)
      1  pushStack (stack, character)
   3  else
      1  If (character is closing parenthesis)
         1  if (emptyStack (stack))
            1  print (Error: Closing parenthesis not matched)
         2  else
            1  popStack(stack, token)
 2 if (not emptyStack (stack)
   1  print (Error: Opening parenthesis not matched)
end parseParens
```

Infix to Postfix notation:

Prefix: + a b Infix: a + b Postfix: a b +

A + B * C

- Step 1: put parenthesis

  (A + (B * C))

- Step 2: starting from the innermost blocks, apply the rule

  (A + (B C *))

  (A(BC*)+)

- Step 3: remove the parentheses

  ABC*+

Infix to postfix using a stack

1. *Scan* the expression from *left to right*. One pass is sufficient.
2. If the *next* scanned *symbol is an operand*, *append* it to the *postfix* expression.
3. If the next scanned symbol is a *left parenthesis*, *push* it onto the *stack*.
4. If the next scanned symbol is a *right parenthesis*, *pop* and *append all* the symbols from the stack *until the most recent left parenthesis*. *Pop and discard the left parenthesis.*
5. If the next scanned symbol is an *operator*:
   - *Pop* and *append* to the postfix expression *every operator from the stack that is above the most recently scanned left parenthesis and that has precedence greater than or equal* to the new operator.
   - *Push* the *new operator* onto the stack.
6. After the infix *string is completely processed*, *pop* and *append* to the postfix string *everything from the stack*.

Input buffer	Operator stack	Output string
A*B-(C+D)+E		

	Infix	Stack	Postfix
(a)	A + B * C − D / E		
(b)	+ B * C − D / E		A
(c)	B * C − D / E	+	A
(d)	* C − D / E	+	A B
(e)	C − D / E	* +	A B
(f)	− D / E	* +	A B C
(g)	D / E	−	A B C * +
(h)	/ E	−	A B C * + D
(i)	E	/ −	A B C * + D
(j)		/ −	A B C * + D E
(k)			A B C * + D E / −

Examples

- A+B-C
- (A+B)*C
- (A+B)*(C-D)
- A+((B+C)*(E-F)-G)/(H-I)
- A+B*(C+D)-E/F*G+H

Table 7-2 Infix Expressions and Their Equivalent Postfix Expressions

Infix Expression	Equivalent Postfix Expression
a + b	a b +
a + b * c	a b c * +
a * b + c	a b * c +
(a + b) * c	a b + c *
(a – b) * (c + d)	a b – c d + *
(a + b) * (c – d / e) + f	a b + c d e / – * f +

57

- Infix: A+B-C;
- Postfix: AB+C-

- Infix: (A+B)*C;
- Postfix: AB+C*

- Infix: (A+B)*(C-D);
- Postfix: AB+CD-*

- Infix: A+((B+C)*(E-F)-G)/(H-I);
- Postfix: ABC+EF-*G-HI-/+

- Infix: A+B*(C+D)-E/F*G+H;
- Postfix: ABCD+*+EF/G*-H+

58

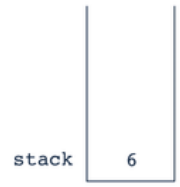
Application of Stacks: Postfix Expression Calculator

Table 7-2 Infix Expressions and Their Equivalent Postfix Expressions

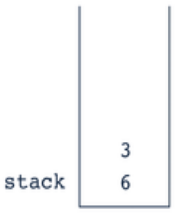
Infix Expression	Equivalent Postfix Expression
a + b	a b +
a + b * c	a b c * +
a * b + c	a b * c +
(a + b) * c	a b + c *
(a – b) * (c + d)	a b – c d + *
(a + b) * (c – d / e) + f	a b + c d e / – * f +

59

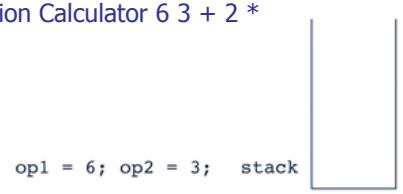
Application of Stacks: Postfix Expression Calculator 6 3 + 2 *



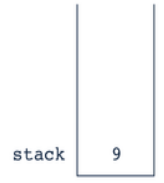
1. Stack after pushing 6



2. Stack after pushing 3



3. Stack after retrieving the top two elements and popping twice

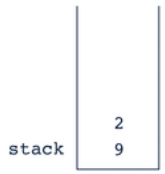


4. Stack after pushing the result of `op1 + op2`, which is `9`

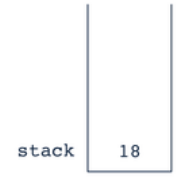
60

Application of Stacks: Postfix Expression Calculator

6 3 + 2 *



5. Stack after pushing 2



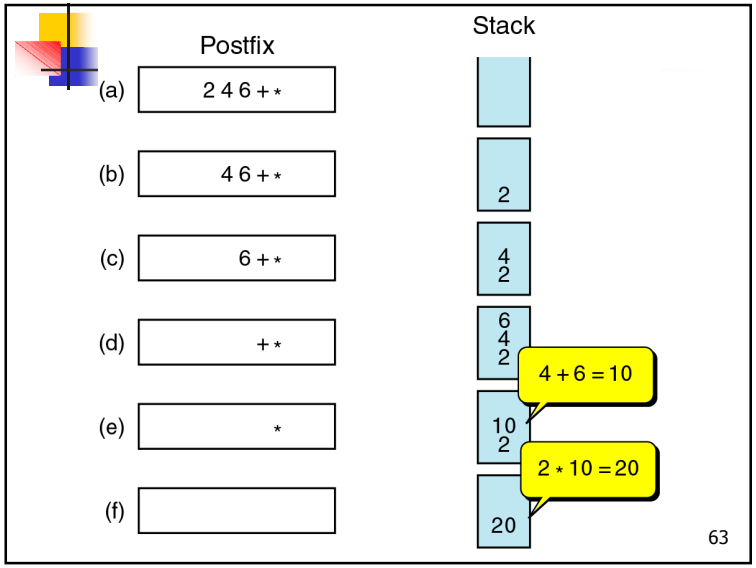
7. Stack after pushing the result of op1 * op2, which is 18

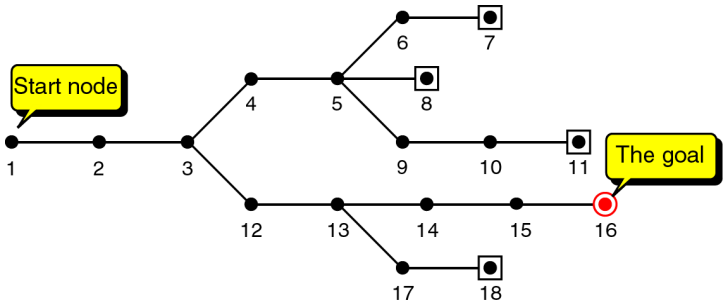
6. Stack after retrieving the top two elements and popping twice

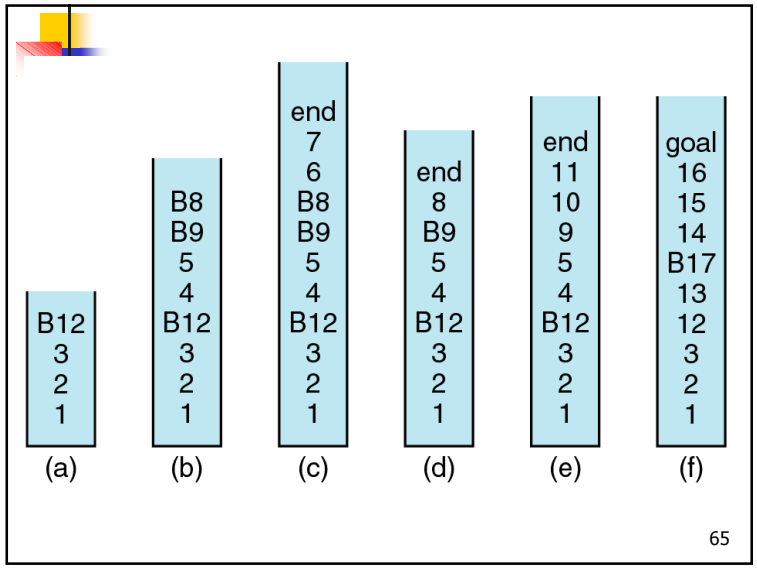
8. Stack after popping the element

Postfix Expression Calculator (Main Algorithm)

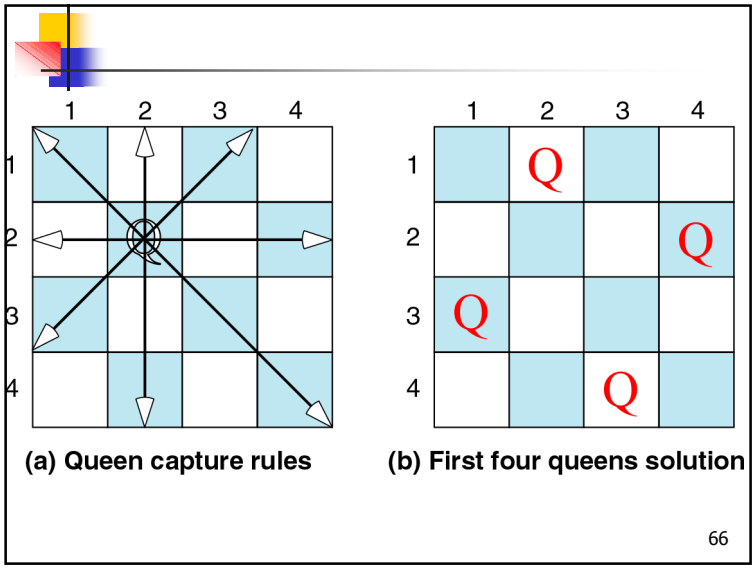
```
read the first ch;
while more data to process
{
    clear the stack;
    output ch;

    while(ch is not = '=') //process each expression
                           //= marks the end of an expression
    {
       switch(ch)
       {
       case '#': read a number;
                 output the number;
                 push the number onto the stack;
                 break;
       default: Assume that ch is an operation
                evaluate the operation;
       }//end switch

       if no error is found, then
       {
          read next ch;
          output ch;
       }
    }   //end while

    if the expression does not contain any error(s), then
       output the result;
    else
      discard the result;

    start processing the next expression;
}
```

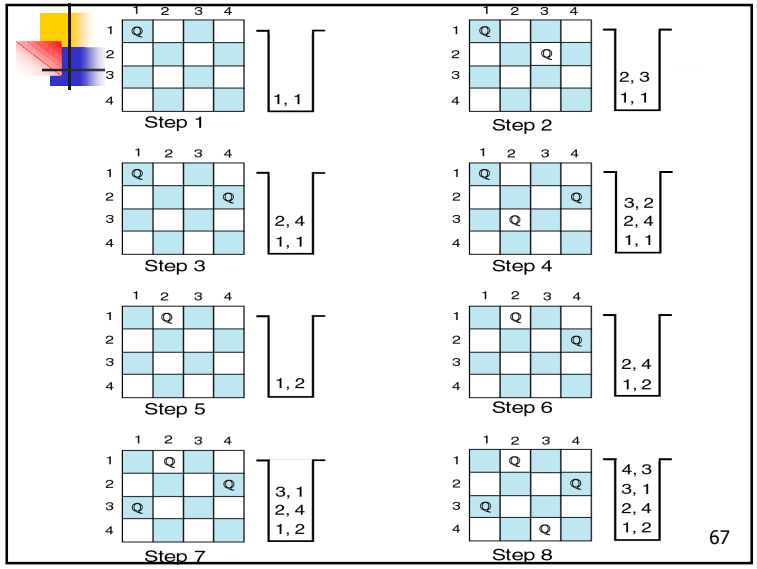



B12
3
2
1
(a)
B8
B9
5
4
B12
3
2
1
(b)
end
7
6
B8
B9
5
4
B12
3
2
1
(c)
end
8
B9
5
4
B12
3
2
1
(d)
end
11
10
9
5
4
B12
3
2
1
(e)
goal
16
15
14
B17
13
12
3
2
1
(f)

65


1 2 3 4
1
2
3
4
(a) Queen capture rules
Q
(b) First four queens solution

66


Step 1
1, 1
Step 2
2, 3
1, 1
Step 3
2, 4
1, 1
Step 4
3, 2
2, 4
1, 1
Step 5
1, 2
Step 6
2, 4
1, 2
Step 7
3, 1
2, 4
1, 2
Step 8
4, 3
3, 1
2, 4
1, 2

67