
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering


# Data Structures and Algorithms – C++ Implementation

Huỳnh Tấn Đạt

Email: htdat@cse.hcmut.edu.vn

Home Page: http://www.cse.hcmut.edu.vn/~htdat/

# Stacks

- Basic stack operations
- Linked-list implementation
- Stack applications
- Array implementation

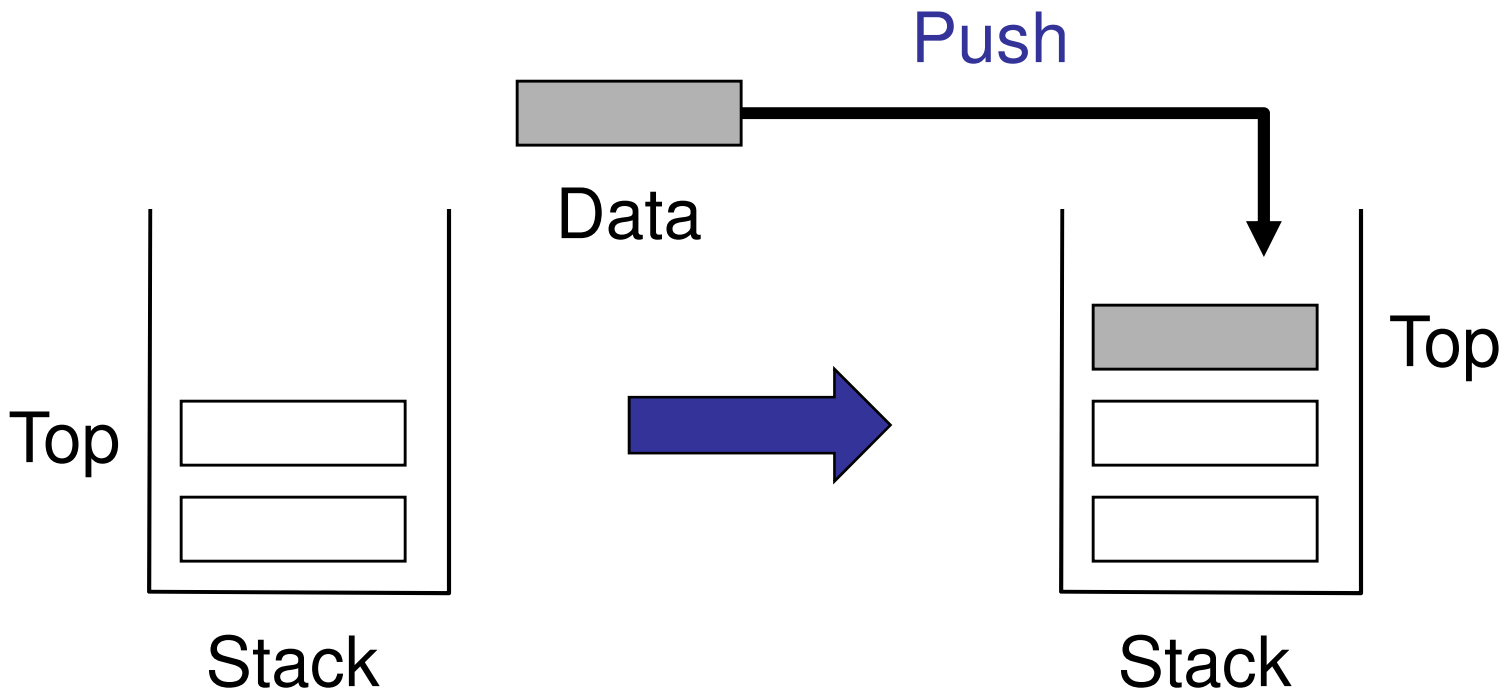
# Linear List Concepts

- General list: no restrictions on where data can be inserted/deleted, and on which operations can be used on the list
- Restricted list: data can be inserted/deleted and operations are performed only at the ends of the list

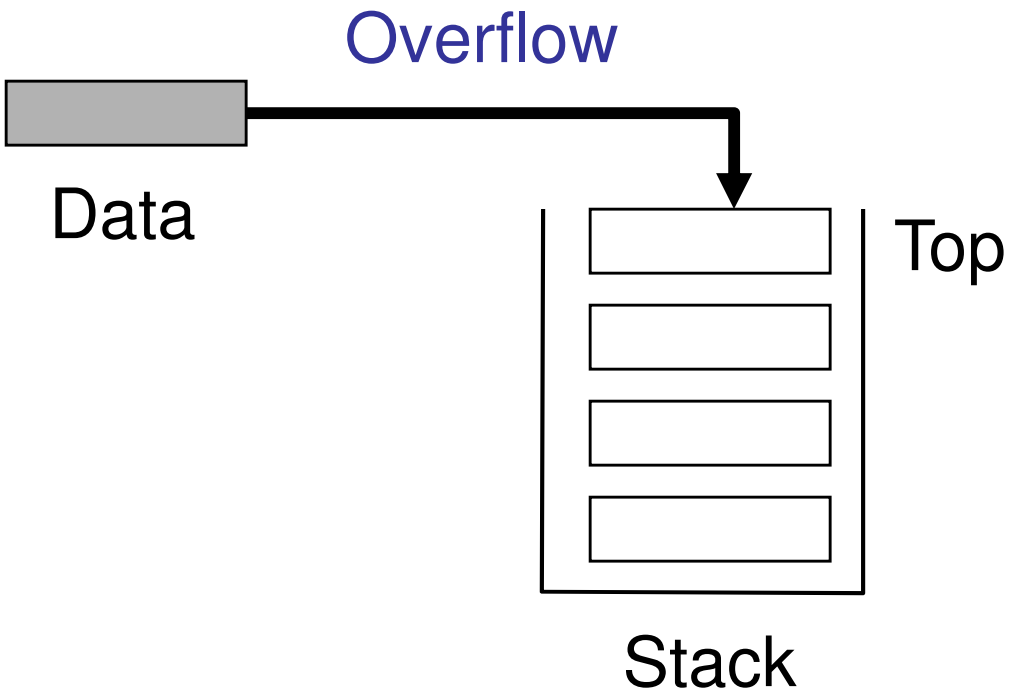
# Stack

- All insertions and deletions are restricted to one end called the top
- Last-In First-Out (LIFO) data structure

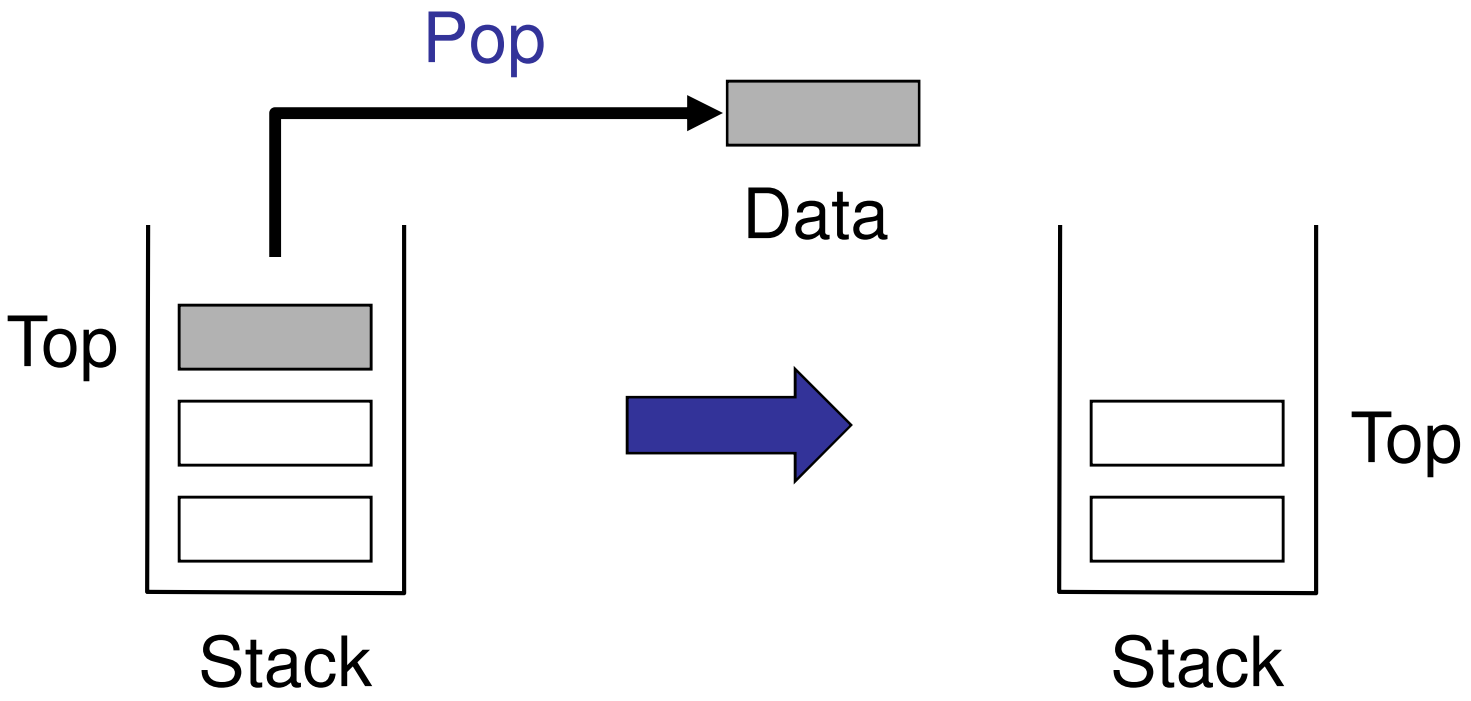
# Basic Stack Operations



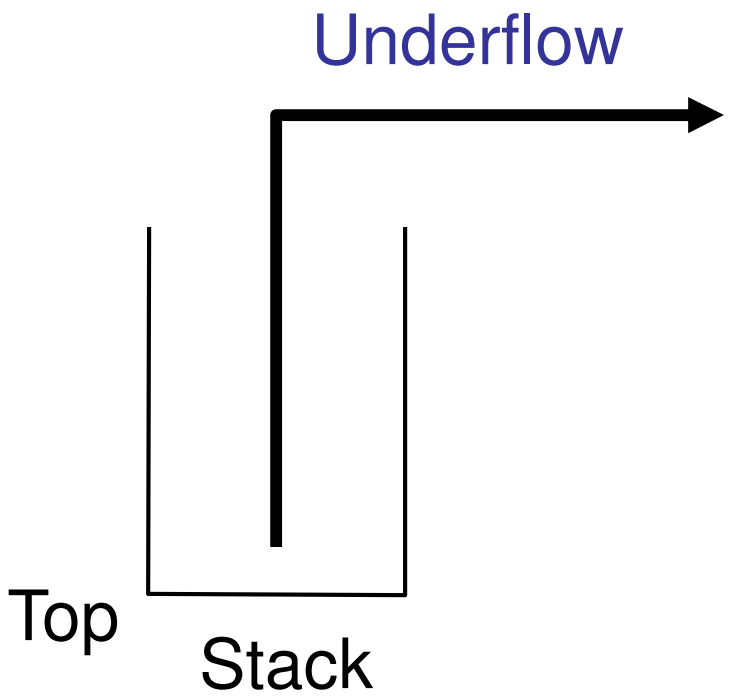
# Basic Stack Operations



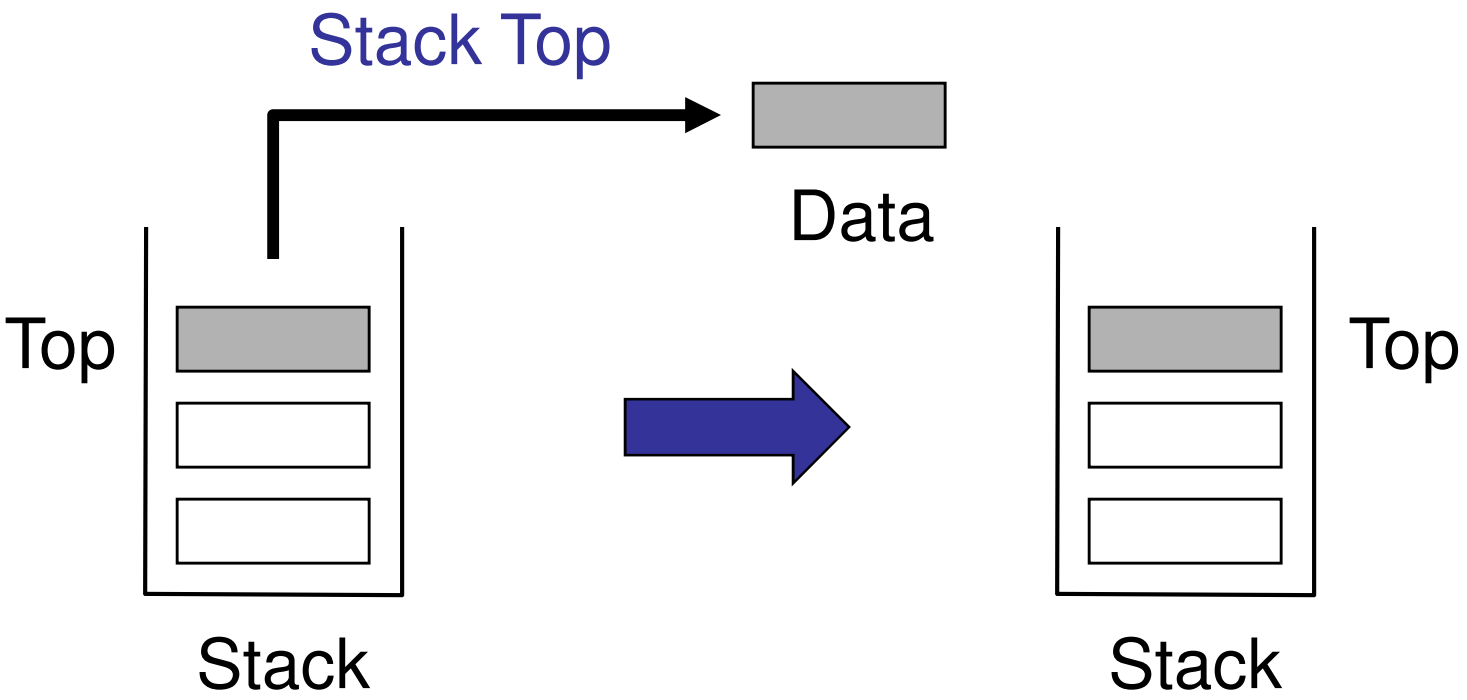
# Basic Stack Operations



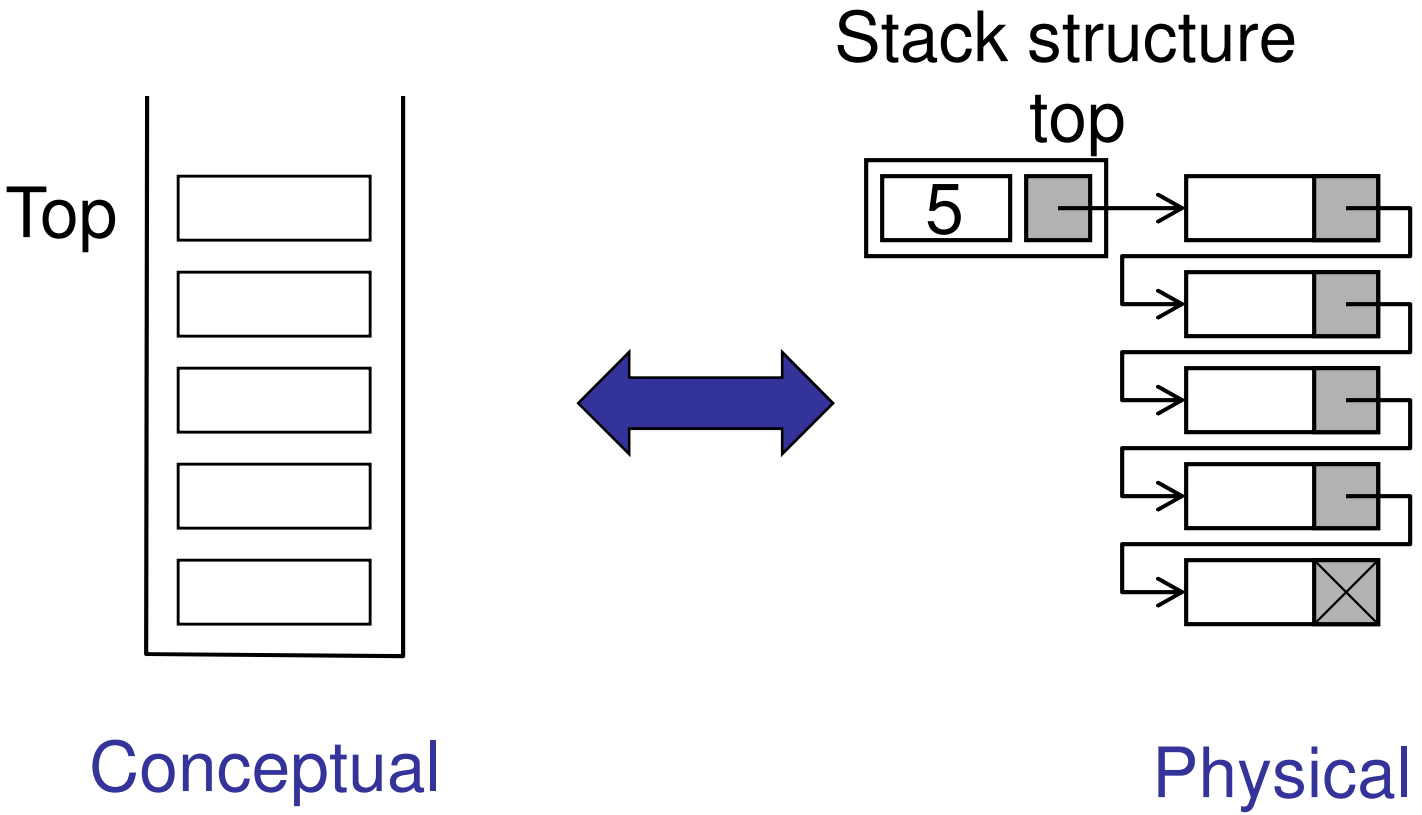
# Basic Stack Operations



# Basic Stack Operations

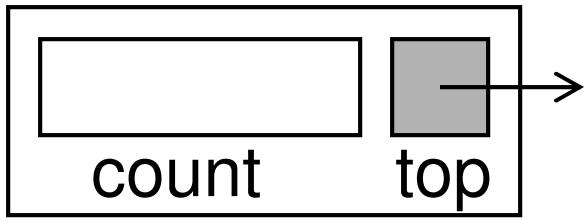


# Linked-List Implementation



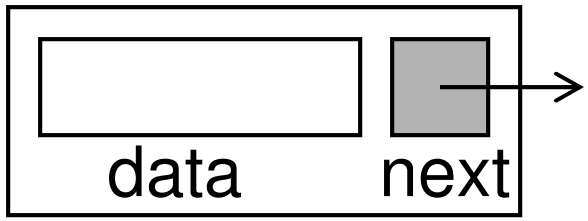
# Linked-List Implementation

Stack structure




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

Stack node structure




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

# Linked-List Implementation

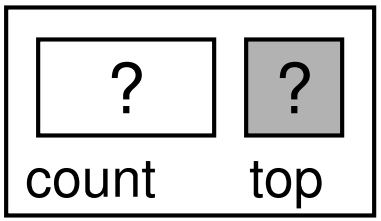
```
template <class ItemType>
struct Node {
  ItemType data;
  Node<ItemType> *next;
};

template <class List_ItemType>
class Stack {
public:
  Stack();
  ~Stack();
```

# Linked-List Implementation

```
  void Push(List_ItemType dataIn);
  int Pop(List_ItemType &dataOut);
  int GetStackTop(List_ItemType &dataOut);
  void Clear();
  int IsEmpty();
  int GetSize();
  Stack<List_ItemType>* Clone();
  void Print2Console();
private:
  Node<List_ItemType>* top;
  int count;
};
```

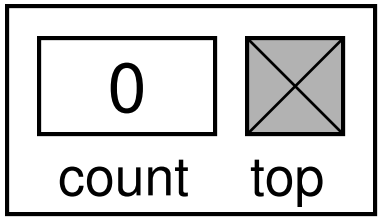
# Create Stack

Before




(no stack)

After




(empty stack)

# Create Stack

**Algorithm** createStack (ref stack <metadata>)

Initializes metadata for a stack

**Pre** stack is structure for metadata

**Post** metadata initialized

1 stack.count = 0

2 stack.top = null

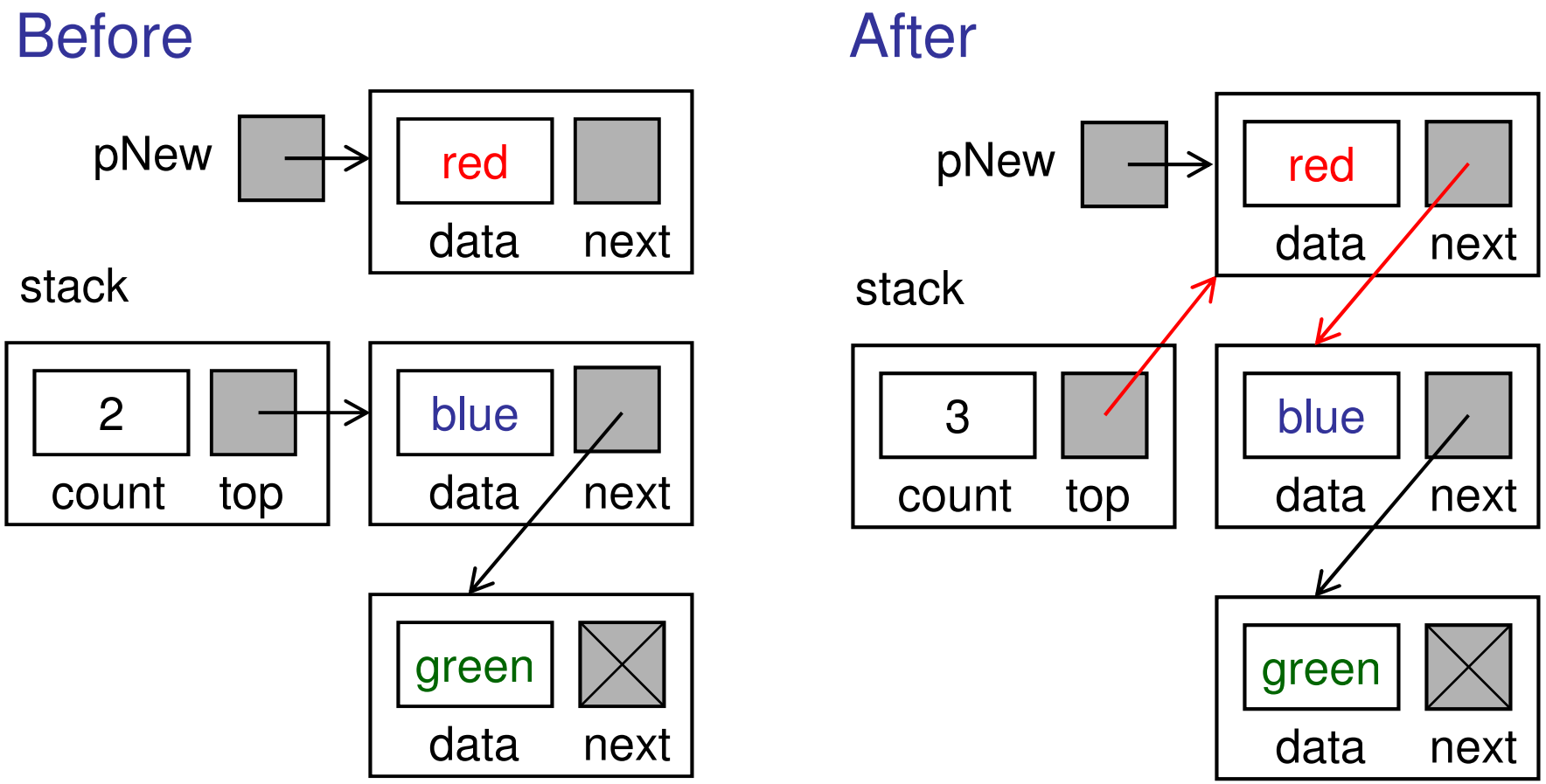
3 return

**End** createStack

# Linked-List Implementation

```
template <class List_ItemType>
Stack<List_ItemType>::Stack(){
  this->top = NULL;
  this->count = 0;
}

template <class List_ItemType>
Stack<List_ItemType>::~Stack(){
  this->Clear();
}
```

# Push Stack


Before
pNew
red
data
next
stack
2
count
top
blue
data
next
green
data
next
After
pNew
red
data
next
stack
3
count
top
blue
data
next
green
data
next

# Push Stack

**Algorithm** pushStack (ref stack <metadata>,
val data <dataType>)

Inserts (pushes) one item into the stack

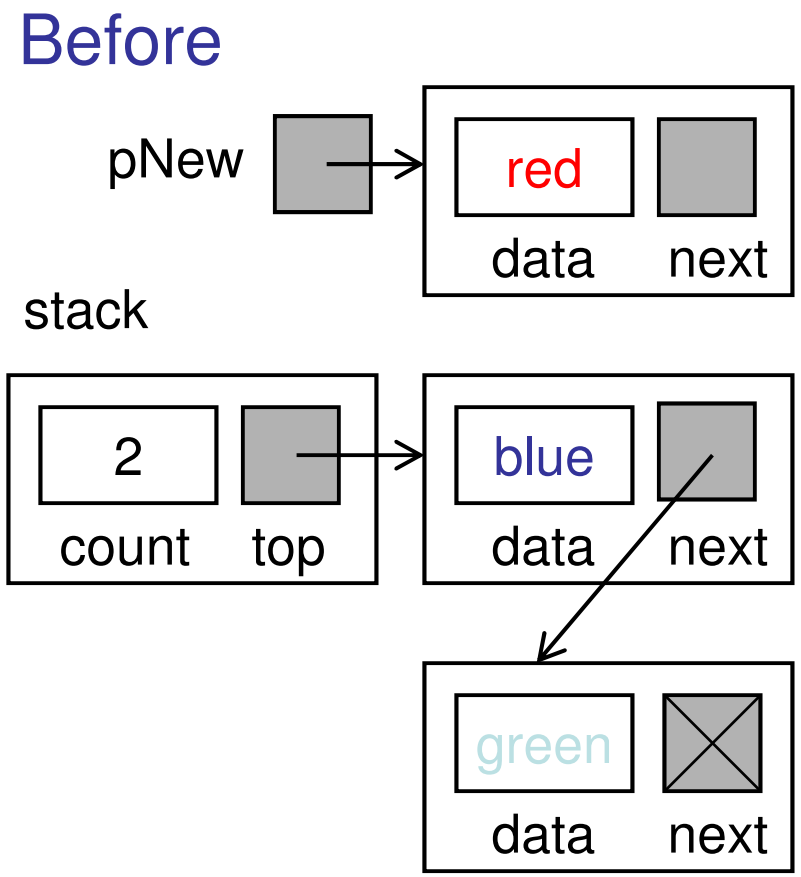
**Pre** stack is a metadata structure to a valid stack
data contains data to be pushed into stack

**Post** data have been pushed in stack

**Return** true if successful; false if memory overflow

# Push Stack

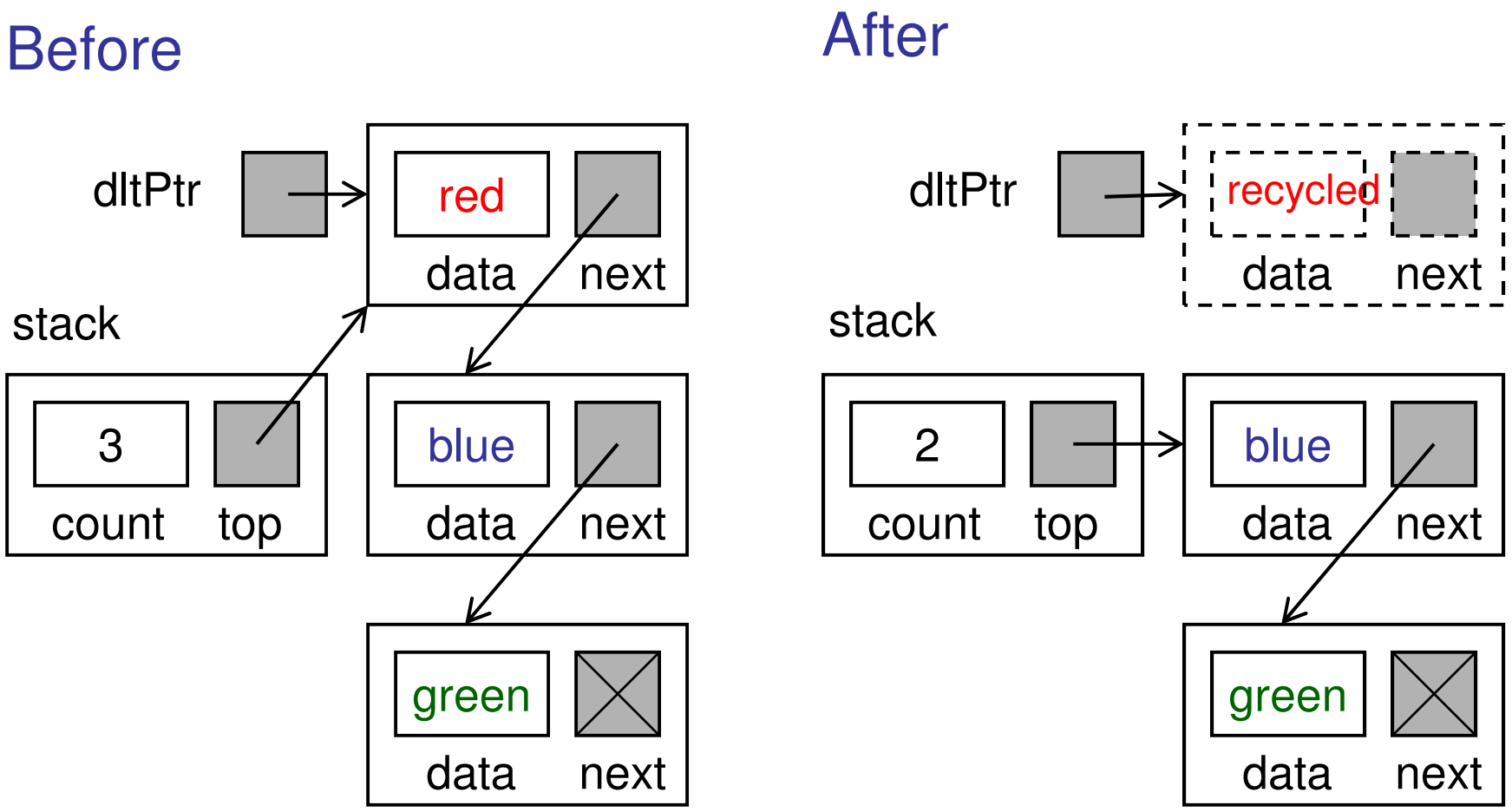
```
1  if (stack full)
   1  success = false
2  else
   1  allocate (pNew)
   2  pNew -> data = data
   3  pNew -> next  = stack.top
   4  stack.top = pNew
   5  stack.count = stack.count + 1
   6  success = true
3  return   success
End  pushStack
```



# Push Stack

```
template <class List_ItemType>
void Stack<List_ItemType>::Push
  (List_ItemType value){
  Node<List_ItemType>* pNew = new
                      Node<List_ItemType>();
  pNew->data = value;
  pNew->next = top;
  this->top = pNew;
  this->count++;
}
```

# Pop Stack


Before
dltPtr
red
data
next
stack
3
count
top
blue
data
next
green
data
next
After
dltPtr
recycled
data
next
stack
2
count
top
blue
data
next
green
data
next

# Pop Stack

**Algorithm** popStack (ref stack <metadata>,
ref dataOut <dataType>)

Pops the item on the top of the stack and returns it to caller

**Pre** stack is a metadata structure to a valid stack
dataOut is to receive the popped data

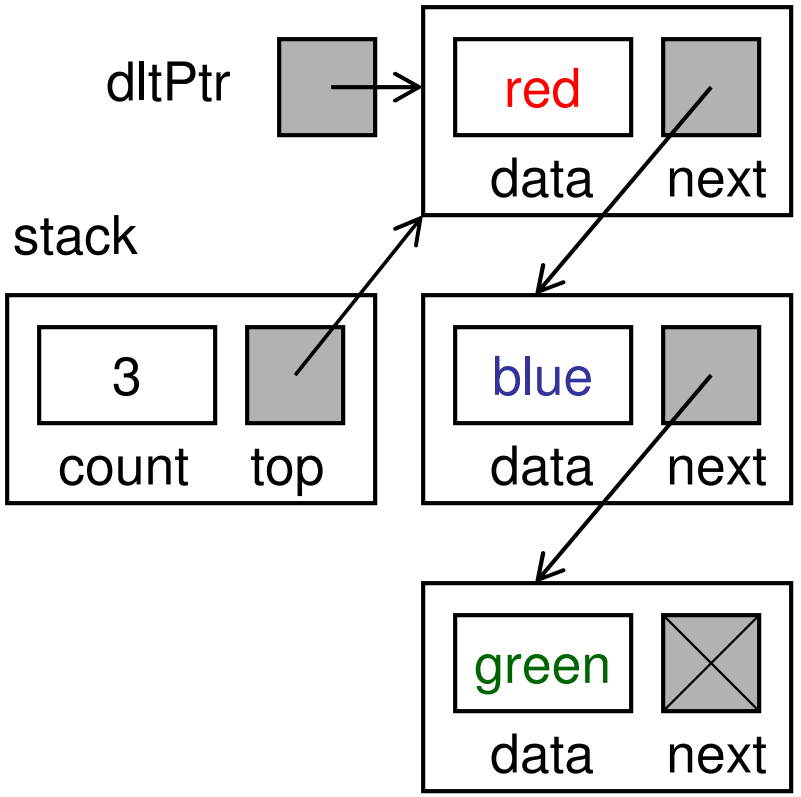
**Post** data have been returned to caller

**Return** true if successful; false if underflow

# Pop Stack

```
1  if (stack empty)
   1  success = false
2  else
   1  dltPtr  = stack.top
   2  dataOut = stack.top -> data
   3  stack.top = stack.top -> next
   4  stack.count = stack.count - 1
   5  recycle (dltPtr)
   6  success = true
3  return success
End  popStack
```

Before



# Pop Stack

```
template <class List_ItemType>
int Stack<List_ItemType>::Pop(List_ItemType
  &dataOut){
  if (count == 0)
     return 0;
  Node<List_ItemType>* dltPtr = this->top;
  dataOut = this->top->data;
  this->top = this->top->next;
  this->count--;
  delete dltPtr;
  return 1;
}
```

# Stack Top

**Algorithm** stackTop (val stack <metadata>,
ref dataOut <dataType>)

Retrieves the data from the top of the stack without changing the stack

**Pre** stack is a metadata structure to a valid stack
dataOut is to receive top stack data

**Post** data have been returned to caller

**Return** true if successful; false

# Stack Top

```
1  if (stack empty)
   1  success = false
2  else
   1  dataOut = stack.top -> data
   2  success = true
3  return success
End  stackTop
```

# Stack Top

```
template <class List_ItemType>
int Stack<List_ItemType>::GetStackTop
  (List_ItemType &dataOut){
  if (count == 0)
     return 0;
  dataOut = this->top->data;
  return 1;
}
```

# Destroy Stack

**Algorithm** destroyStack (ref stack <metadata>)

Releases all nodes back to memory

**Pre** stack is a metadata structure to a valid stack

**Post** stack empty and all nodes recycled

```
1  if (stack not empty)
   1  loop (stack.top not null)
      1  temp = stack.top
      2  stack.top = stack.top -> next
      3  recycle (temp)
2  stack.count = 0
3  return
```

**End** destroyStack

# Destroy Stack

```
template <class List_ItemType>
void Stack<List_ItemType>::Clear() {
  Node<List_ItemType>* temp;
  while (this->top != NULL){
     temp = this->top;
     this->top = this->top->next;
     delete temp;
  }
  this->count = 0;
}
```

# Stack Empty

```
template <class List_ItemType>
int Stack<List_ItemType>::IsEmpty() {
  return (count == 0);
}
template <class List_ItemType>
int Stack<List_ItemType>::GetSize() {
  return count;
}
```

# Print a stack

```
template <class List_ItemType>
void Stack<List_ItemType>::Print2Console() {
      Node<List_ItemType>* p;
  p = this->top;
  while (p != NULL){
      printf("%d\t", p->data);
      p = p->next;
  }
  printf("\n");
}
```

# Using Stacks

```
int main(int argc, char* argv[]){
  Stack<int> *myStack = new Stack<int>();
  int val;
  myStack->Push(7);
  myStack->Push(9);
  myStack->Push(10);
  myStack->Push(8);
  myStack->Print2Console();
  myStack->Pop(val);
  myStack->Print2Console();
  delete myStack;
  return 0;
}
```

# Exercises

```
template <class List_ItemType>
Stack<List_ItemType>*
          Stack<List_ItemType>::Clone() {
  // ...
}
```