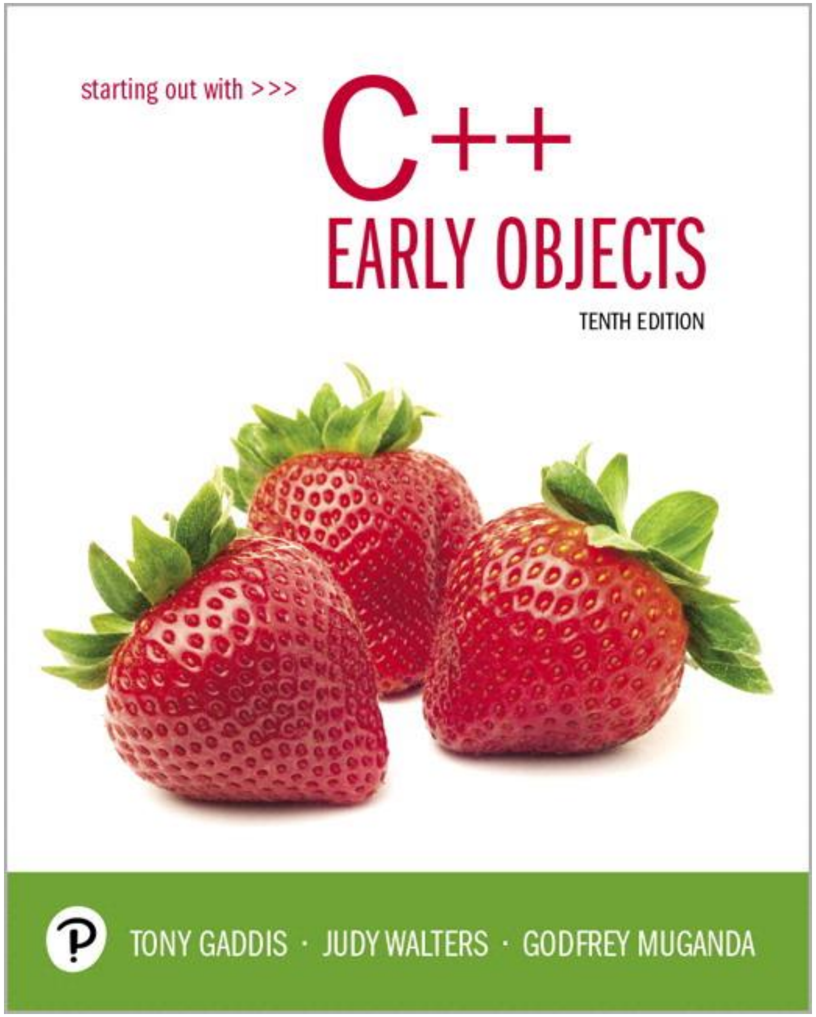
Starting Out with C++ Early Objects

Tenth Edition



Chapter 19

Stacks and Queues

Topics

19.1 Introduction to Stacks

19.2 Dynamic Stacks

19.3 The STL **`stack`** Container

19.4 Introduction to Queues

19.5 Dynamic Queues

19.6 The STL **`deque`** and **`queue`** Containers

19.7 Eliminating Recursion

Pearson

18.1 Introduction to Stacks

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria serving area
  - return addresses for function calls

Stack Basics

- Stack is usually implemented as a list, with additions and removals taking place at one end of the list
- The active end of the list implementing the stack is the top of the stack
- Stack types:
  - Static – fixed size, often implemented using an array
  - Dynamic – size varies as needed, often implemented using a linked list

Stack Operations and Functions

Operations:

- –push: add a value at the top of the stack
- –pop: remove a value from the top of the stack

Boolean function:

- –isEmpty: true if the stack currently contains no elements

Static Stack Implementation

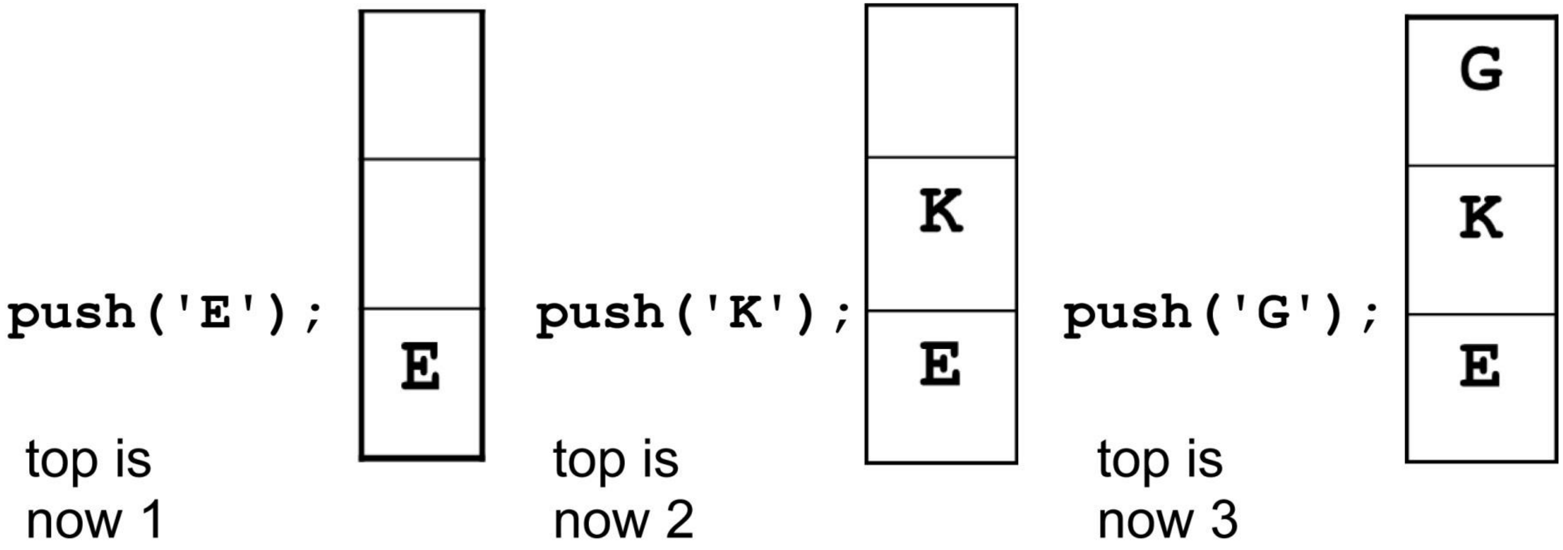
- Uses an array of a fixed size
- Bottom of stack is at index 0. A variable called top tracks the current top of the stack

```
const int STACK_SIZE = 3;
char s[STACK_SIZE];
int top = 0;
```

top is where the next item will be added

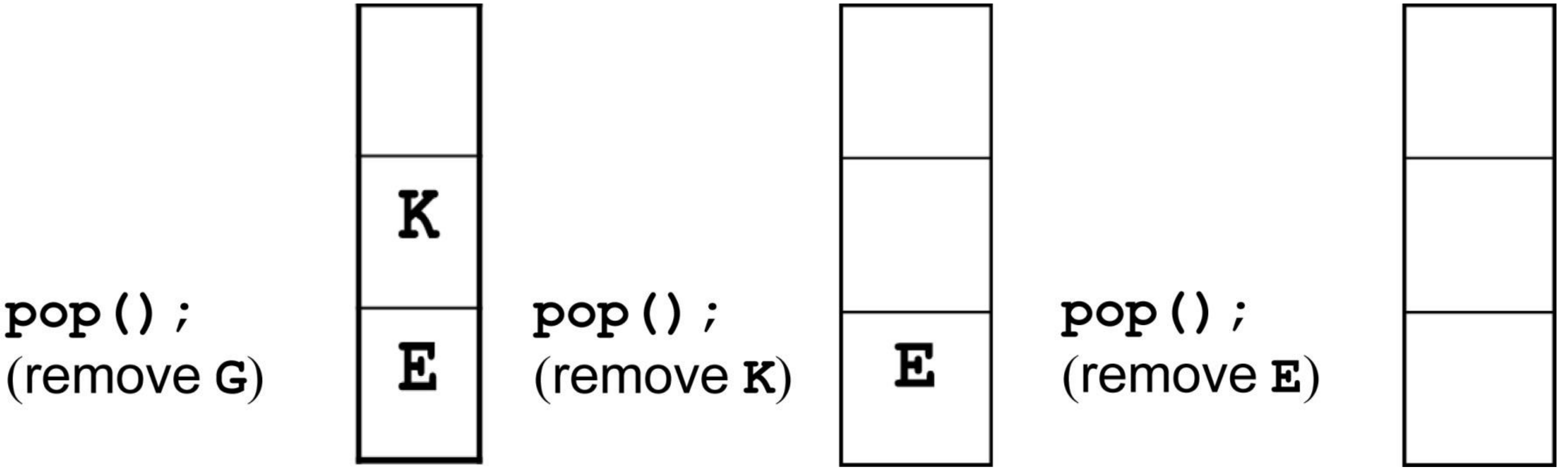
Array Implementation Example

This stack has max capacity 3, initially top = 0 and stack is empty.



Stack Operations Example

After three pops, **`top`** is **`0`** and the stack is empty



Class Implementation Using an Array 1 of 5

```
class STACK
{
 private:
      unique_ptr<char []> s;  // for the array
      int capacity, top;
   public:
      void push(char x);
      void pop(char &x);
      bool isEmpty();
      STACK(int capacity);
      ~STACK()
};
```

Class Implementation Using an Array 2 of 5

Use exception classes as members of the STACK class to signal that an underflow or overflow condition has occurred:

```
class Overflow {};
class Underflow {};
```

Class Implementation Using an Array 3 of 5

To check if the stack is empty:

```
bool STACK::isEmpty()
{

        return (top == 0);

}
```

Class Implementation Using an Array 4 of 5

To add an item to the stack

```
void STACK::push(char x)
{
  if (top==capacity)
     throw STACK::Overflow();
  s[top] = x;
  top++;
}
```

Class Implementation Using an Array 5 of 5

To remove an item from the stack

```
void STACK::pop(char &x)
{
 if (isEmpty())
     throw STACK::Underflow();
 top--;
 x = s[top];
}
```

Exceptions from Stack Operations

- The preceding example uses exception classes to handle cases where an attempt is made to push onto a full stack (overflow) or to pop from an empty stack (underflow)
- Programs that use **`push`** and **`pop`** operations should do so from within a **`try`** block.
- **`catch`** block(s) should follow the **`try`** block to interpret what occurred and to inform the user.

19.2 Dynamic Stacks

- The storage for a stack can be implemented as a linked list
- There is no need to indicate the initial capacity of the stack. It can grow and shrink as necessary.
- It can't ever be full as long as memory is available, so there is no need to test for overflow.
- Testing for underflow (empty stack) is still needed.

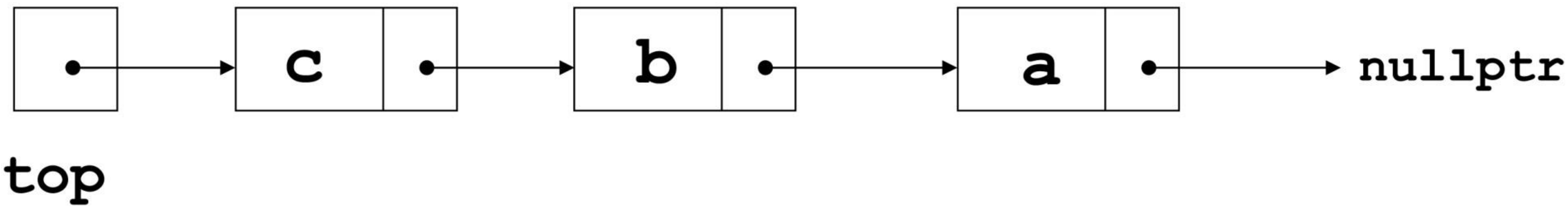
Dynamic Linked List Implementation

- Define a class for a dynamic linked list
- Within the class, define a private member class (**`LNode`**) for dynamic nodes in the list
- Define a node pointer to the beginning of the linked list, which will serve as the top of the stack

Linked List Implementation

A linked stack after three push operations:

```
push('a'); push('b'); push('c');
```



Operations on a Linked Stack 1 of 3

Check if stack is empty:

```
bool isEmpty()
{
        return (top == nullptr);
}
```

Operations on a Linked Stack 2 of 3

Add a new item to the stack

```
void push(char x)
{
   top = new LNode(x, top);
}
```

Operations on a Linked Stack 3 of 3

Remove an item from the stack

```
 void pop(char &x)
{
  if (isEmpty())
     throw STACK::Underflow();
  x = top->value;
  LNode *oldTop = top;
  top = top->next;
  delete oldTop;
}
```

19.3 The STL `stack` Container

- Stack template can be implemented using a **`vector`**, **`list`**, or a **`deque`**
- Implements **`push`**, **`pop`**, and **`empty`** member functions
- Implements other member functions:
  - **`size`**: number of elements on the stack
  - **`top`**: reference to element on top of the stack (must be used with **`pop`** to remove and retrieve top element)

Container Adapters

- A class that provides a new interface to an existing class is a container adapter.
- The purpose of a container adapter is to provide a specialized use of the existing class.
- The STL **`stack`** container, using either a **`vector`**, a **`list`**, or a **`deque`**, is an example of a container adapter.

Defining an STL-based Stack

- Defining a stack of **`char`**, named **`cstack`**, implemented using a **`vector`**:

```
stack< char, vector<char> > cstack;
```

- Implemented using a list:

```
stack< char, list<char> > cstack;
```

- Implemented using a **`deque`** (default):

```
stack< char > cstack;
```

- Prior to C++ 11, spaces are required between consecutive **`> >`** symbols to distinguish from stream extraction

18.4 Introduction to Queues

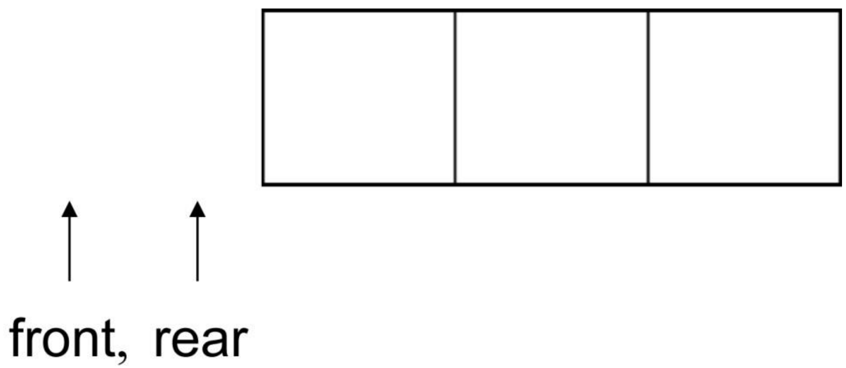
- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people waiting to use an ATM
  - cars lined up to pay and exit a parking structure
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Queue Locations and Operations

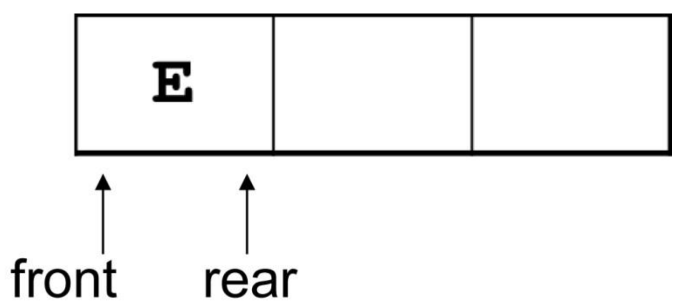
- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Array Implementation of Queue 1 of 3

An empty queue that can hold **`char`** values:

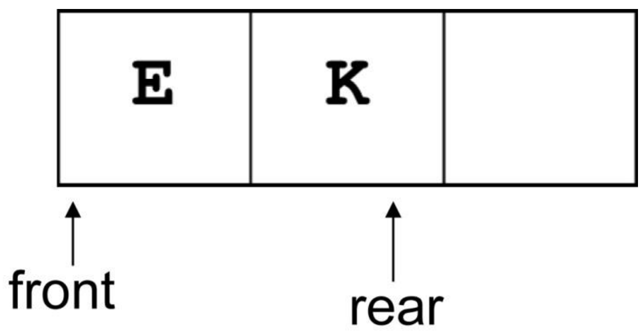


`enqueue('E');`

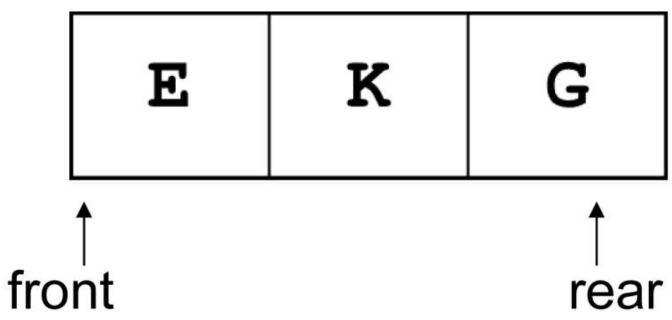


Array Implementation of Queue 2 of 3

```
enqueue('K');
```

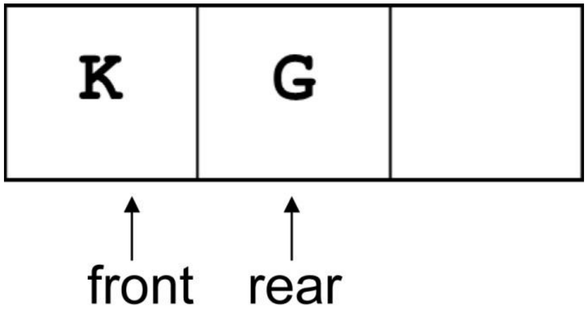



```
enqueue('G');
```

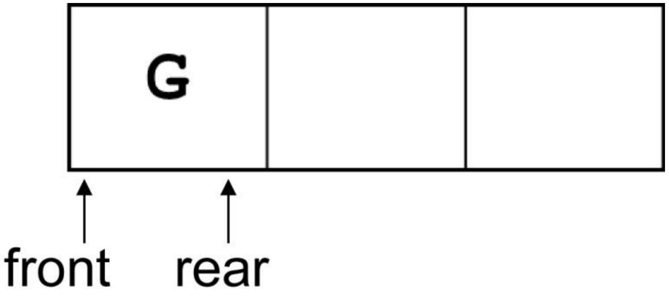


Array Implementation of Queue 3 of 3

`dequeue(); // remove E`



`dequeue(); // remove K`



Array Implementation Issues 1 of 2

- In the preceding example, front never moves.
- Whenever **`dequeue`** is called, all remaining queue entries move up one position. These moves takes time.
- Alternate approach:
  - –Use a ‘circular’ array: **`front`** and **`rear`** both move when items are added and removed. Both can ‘wrap around’ from the end of the array to the front if warranted.
- Other solutions are possible

Array Implementation Issues 2 of 2

- queue variables needed:
  - `int qSize;`
  - `unique_ptr<int []> q;`
  - `int front = -1;`
  - `int rear = -1;`
  - `int number = 0; //how many in queue`
- These could be members of a queue class, and queue operations would be member functions

`isEmpty` Member Function

Check if queue is empty:

```
bool isEmpty()
{
    return (number == 0);
}
```

isFull Member Function

Check if queue is full:

```
bool isFull()
{

    return (number == qSize);

}
```

enqueue and dequeue 1 of 4

- To enqueue, we need to add an item `x` to the rear of the queue
- Queue convention says `q[rear]` is already occupied. Execute

```
if(!isFull)
 { rear = (rear + 1) % qSize;
// mod operator for wrap-around
   q[rear] = x;
   number ++;
}
```

enqueue and **dequeue** 2 of 4

- To dequeue, we need to remove an item **x** from the front of the queue
- Queue convention says **q[front]** has already been removed. Execute

```
   if(!isEmpty)
{   front = (front + 1) % qSize;
    x = q[front];
    number--;
}
```

`enqueue` and `dequeue` 3 of 4

- `enqueue` moves `rear` to the right as it fills positions in the array
- `dequeue` moves `front` to the right as it empties positions in the array
- When `enqueue` gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When `dequeue` gets to the end, it wraps around to the beginning use those positions that have been filled

enqueue and dequeue 4 of 4

- Enqueue wraps around by executing

```
rear = (rear + 1) % qSize;
```

- Dequeue wraps around by executing

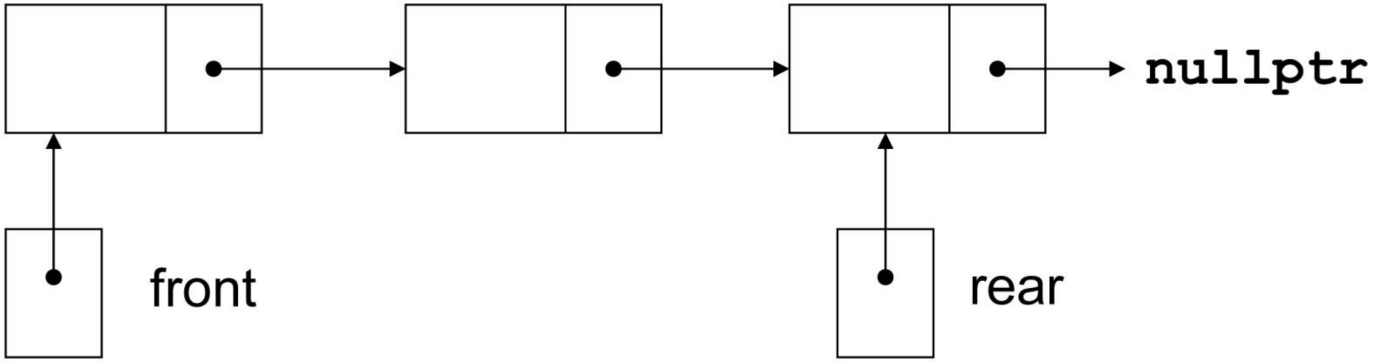
```
front = (front + 1) % qSize;
```

Exception Handling in Static Queues

- As presented, the static queue class will encounter an error if an attempt is made to enqueue an element to a full queue, or to dequeue an element from an empty queue
- A better design is to throw an underflow or an overflow exception and allow the programmer to determine how to proceed
- Remember to throw exceptions from within a **`try`** block, and to follow the **`try`** block with a **`catch`** block

19.5 Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- This allows dynamic sizing and avoids the issue of wrapping indices



Dynamic Queue Implementation Data Structures

- Define a class for the dynamic queue
- Within the dynamic queue, define a private member class for a dynamic node in the queue
- Define node pointers to the front and rear of the queue

Dynamic queue: isEmpty Member Function

To check if queue is empty:

```
bool isEmpty()
{

    return (front == NULL);

}
```

enqueue Member Function Details

To add item at rear of queue

```
if (isEmpty())
    {
      front = new QNode(x);
      rear = front;
    }
  else
    {
      rear->next = new QNode(x);
      rear = rear->next;
    }
```

dequeue Member Function

To remove item from front of queue

```
if (isEmpty())
{
    // throw exception or print
    // a message
} else {
x = front->value;
QNode *oldfront = front;
front = front->next;
delete oldfront;
}
```

19.6 The STL `deque` and `queue` Containers

- **`deque`**: a double-ended queue (DEC). Has member functions to enqueue (**`push_back`**) and dequeue (**`pop_front`**)

- **`queue`**: container ADT that can be used to provide a queue based on a **`vector`**, **`list`**, or **`deque`**. Has member functions to enqueue (**`push`**) and dequeue (**`pop`**)

Pearson

Defining a Queue

- Defining a queue of **`char`**, named cQueue, based on a **`deque`**:

```
deque<char> cQueue;
```

- Defining a **`queue`** with the default base container

```
queue<char> cQueue;
```

- Defining a queue based on a **`list`**:

```
queue<char, list<char> > cQueue;
```

- Prior to C++ 11, spaces are required between consecutive **`> >`** symbols to distinguish from stream extraction

19.7 Eliminating Recursion

- Recursive solutions to problems are often elegant but inefficient
- A solution that does not use recursion is more efficient for larger sizes of inputs
- Eliminating the recursion: re-writing a recursive algorithm so that it uses other programming constructs (stacks, loops) rather than recursive calls

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.

Pearson