Chapter 3: Restricted Structures

Overview Of Restricted Structures

• The two most commonly used restricted structures are Stack and Queue
  – Both can be implemented as array-based structures
  – Their restrictions are consistent with some applications, and the performance of Stack and Queue is excellent
• Coding generic data structures can be methodized
• A priority queue is a restricted structure used in scheduling applications
• Java’s API Stack class is a generic structure

Restrictions on Restricted Structures

• Operation restrictions
  – Update is not supported
  – Insert is supported (not restricted)
  – Fetch and Delete are combined into one operation
• Access mode restrictions
  – Key field mode is not supported
  – Node number mode is severely restricted
• Still, they are ideally suited for some applications

Insert Operation

• A Restricted Structure after Nodes A, then B, the C and Finally D Have Been Inserted

Restrictions on Node Number Access Mode

• Can only say “Fetch-and-Delete” the next node (e.g. cannot “Fetch-and-Delete the 3rd node”)
  – For a Stack: the node returned and deleted is the node in the structure the shortest time
  – For a Queue: the node returned and deleted is the node in the structure the longest time

Combined Fetch-and-Delete Operation

(assumes Node A was inserted first, Node D last)
• A Queue and a Stack before and after a Fetch-and-Delete Operation is performed

Stack

• Stack has its own terminology
• When based on an array, its operation algorithms are simple and fast
• It is implemented as a separate class to promote software reusability and generic conversion
• Used in many applications
• Its two operations are sometime expanded and it can be made dynamic

Terminology of Stacks

• Operations
  – The Insert operation is called Push
    • An item is pushed “onto” a stack
  – The Fetch-and-Delete operation is called Pop
    • An item is popped “off of” or “from” a stack
• Errors
  – A Pop from an empty stack is an underflow error
  – A push onto a full stack is an overflow error
• The last item pushed onto the stack is said to be at the “top” of the stack
• Stack is a Last-In-First-Out structure (LIFO)
Stack Operations on a Three Member Stack

Stack Operation Algorithms for an Array Based Stack
- Initialization, Push, and Pop algorithms assume
  - An array, data, stores references to the nodes
  - An integer variable top stores the index of the last node pushed
  - An integer variable size stores the size of the array

Stack Initialization Algorithm
For a Stack of Size s

top = -1;
size = s;

Push Algorithm
(assumes newNode is being pushed)

if (top == size - 1)
    return false; // ** overflow error **
else
{
    top = top + 1;
data[top] = newNode.deepCopy();
return true; // push operation successful
}

Memory Action of the Push Algorithm

Pop Algorithm

if(top == -1)
    return null; // ** underflow error **
else
{
    topLocation = top;
    top = top - 1;
    return data[topLocation];
    // returns a shallow copy
}

Memory Action of the Pop Algorithm

Stack Implementation
- Implemented as a class
- Private data members
  - data[], top and size
- Methods
  - A constructor to initialize data and top and allocate the array
  - push and pop
    - The Java equivalent of the pseudocode algorithms
    - showAll to output all nodes
    - Invokes node definition class’ toString method
- The method pop returns a shallow copy

Stack Applications
- Any algorithm that requires a LIFO structure
  - Artificial intelligence algorithms e.g., backtracking
  - Tree traversals
Graph traversals
In compilers
  • For passing information to, and from, subprograms
  • Remembering return addresses
  • Evaluation of arithmetic expressions

Evaluation of Arithmetic Strings
• Programmers write in infixed notation:
  – Operators in between operands: 2 + 3 * 4
  – Ambiguity of operand order requires precedence rules (slow runtime evaluation)
• To improve speed, compilers
  – Change infixed notation to postfixed 2 3 4 * + using a Stack and a Queue structure
  – Use a Stack to evaluate the expression at run time

Evaluating Postfixed Expressions
1. Begin at left side of the postfixed expression
2. Move to right
3. if an operand is found
   Push it onto a stack
else when an operator is found
   Pop two operands off the stack
   Apply the operator to these two operands
   Push the result onto the stack
4. Repeat steps 2 and 3 until the end of the expression is reached. Then pop the value of the expression off the stack

Postfixed Expression Evaluation Algorithm

Stack Progression During The Evaluation Of 2 3 4 * +
Progression → → →

Expanded Stack Operations (Page 1)
• Reinitialize the stack to empty, init
  top = -1;
• Test for an empty stack (underflow condition), isEmpty
  if (top == -1) // stack empty
• Test for a full stack (overflow condition), isFull
  if (top == size - 1) // stack full

Expanded Stack Operations (Page 2)
• Pop a node from the stack without deleting it from the structure, peek
  – Same as Pop algorithm, but top is not decremented
• Expand the stack at run time within the limits of system memory (inside of the Push algorithm)
  – Use Java’s arraycopy and set size to the expanded size

Performance of the Stack Structure
• Speed
  – Push performs 4 memory access to: fetch top and size, rewrite top, and write the node location into data[top], O(1)
  – Pop performs 3 memory accesses to: fetch top, rewrite top, and to fetch the contents of data[topLocation], O(1)
• Density > 0.8 for node widths >= 16 bytes
• Outperforms the Unsorted-Optimized structure
Density of a Stack
- Density = information bytes / total bytes
  - Information bytes = n \times w
    - n is the number of nodes, w is the bytes per node
  - Overhead = 4n + 8
    - 4 bytes per array element + 8 bytes for next and size
- Density = n \times w / (n \times w + 4n + 8)
  - As n gets large, \( n \times w / (n \times w + 4n + 8) \rightarrow 1/(1 + 4/w) \)

Variation in the Density of a Stack With Node Width

Comparison of the Stack and Unsorted-Optimized Structures

Queue
- Queue has its own terminology
- When based on an array, operation algorithms are simplistic and demonstrate good performance
- It is implemented as a separate class to promote software reusability and generic conversion
- It is used in many applications
- Its two operations are sometime expanded, and it can be made dynamic

Terminology of Queues
- Operations
  - The Insert operation is called Enqueue
    - We say an item is “entered into” a queue
  - The Fetch-and-Delete operation is called Dequeue
    - We say an item is “removed from” a queue
- Errors
  - A Dequeue from an empty queue is an underflow error
  - An enqueue into a full queue is an overflow error
- The first and last items entered into a queue is said to be at the “front” and “rear” of the queue
- Queue is a First-In-First-Out structure (FIFO)

Queue Operations

Queue Operation Algorithms
For An Array Based Stack
- Initialization, Enqueue, and Dequeue algorithms assume:
  - An array, data, stores references to the nodes
  - An integer variable size stores the size of the array
  - Integer variables front and rear keep track of the front and rear of the queue
  - An integer variable numOfNodes stores the number of nodes in the queue

Queue Initialization Algorithm
for a Queue of Size s
\[
\text{size} = s; \quad \text{numOfNodes} = 0; \quad \text{front} = 0; \quad \text{rear} = 0;
\]

Enqueue Algorithm
(assumes newNode is being inserted)
if(numOfNodes == size)
  return false; // ** overflow error **
else
{
    numOfNodes = numOfNodes + 1;
    data[rear] = newNode.deepCopy();
    rear = (rear + 1) % size; // circular
    return true;  // Enqueue successful
}

---

**Memory Action Of The Enqueue Algorithm**

**Dequeue Algorithm**

if (numOfNodes == 0)
    return null;  // ** underflow error **
else
{
    frontLocation = front;
    front = (front + 1) % size; // circular
    numOfNodes = numOfNodes - 1;
    return data[frontLocation];
}

---

**Memory Action of the Dequeue Algorithm**

**Queue Operations on a Three Member Queue (Page 1)**

**Queue Operations on a Three Member Queue (Page 2)**

---

**Performance of a Queue**

- Speed
  - Enque performs 6 memory access to: fetch numOfNodes, size, and rear; rewrite top and numOfNodes, and write the node location into data[rear], O(1)
  - Dequeue performs 6 memory accesses to: fetch numOfNodes, front, and size; rewrite front and numOfNodes, and to fetch the contents of data[front], O(1)
- Density > 0.8 for node widths >= 16 bytes
- Outperforms the Unsorted-Optimized structure

---

**Density Of A Queue**

- Density = information bytes / total bytes
  - Information bytes = n * w
  - n is the number of nodes, w is the bytes per node
  - Overhead = 4n + 16
  - 4 bytes per array element + 16 bytes for front, rear, size, and numOfNodes
- Density = n * w / (n * w + 4n + 16)
  - As n gets large n * w/(n * w + 4n + 16) → 1/(1 + 4/w)
- Density → 1/(1 + 4/w) is same as that of a Stack

---

**Variation in the Density of a Stack With Node Width**

**Comparison of Stack’s Performance With Previously Developed Structures**

---

**Implementation Of A Queue**

- Implemented as a class
- Private data members
  - data, numberOfNodes, front, rear and size
- Methods
  - A constructor initialize the data members and allocate the array
  - enqueue and dequeue
  - the Java equivalent of the pseudocode algorithms
– showAll to output all nodes
  • invokes the node definition class’ toString method
  • The method dequeue returns a shallow copy

Queue Applications
• Any algorithm that process data in the order it is received, FIFO
  – Print spoolers (several tasks sharing a printer)
  – Queuing theory algorithms in Operations Research
  – Artificial intelligence algorithms
  – Sorting algorithms
  – Graph traversals
  – Conversion of infixed expressions to postfixed

Methodized Generic Coding Process
• Code a node definition class and a non-generic data structure class that complies to a set of coding guidelines (e.g., our Stack and Listing classes)
• Test and debug the two classes
• Use a four-step methodology to convert the data structure class to a generic implementation

Generic Coding Guidelines
• The node definition and the data structure are coded as two separate classes
• The data structure cannot mention the names of the data fields that make up a node
• If the structure is going to be encapsulated, a method to perform a deep copy of a node must be coded in the node definition class
• If the structure is going to be accessed in the key field mode, a method to determine if a given key is equal to the key of a node must be coded in the node definition class
• The data structure class cannot mention the key field’s type

Four-Step Methodology To Convert a Data Structure Class To A Generic Structure
• Step 1
  – Add a generic placeholder at the end of the class’ heading e.g., <T>
    public class GenericStack <T>
• Step 2
  – Replace all occurrences of the name of the node definition class with the generic placeholder T
    e.g., public boolean push( T newNode)
• Step 3
  – Wherever the operator new operates on the placeholder T, substitute Object for T, include coercion
    e.g., data = (T[ ]) new Object[100]

Four-Step Generic Conversion Methodology (continued)
• Step 4
  – Wherever a method operates on an instance of type T (e.g., newNode)
    • The method’s signature is added to an interface (named, e.g., GenericNode), which must be implemented by the node definition class
    • An instance of the interface is declared and assigned the instance’s address
      Generic node = (GenericNode) newNode;
    • The method invocation is changed to operate on the instance of the generic type
      e.g., data[top] = node.deepCopy();
Priority Queue

Java’s API Stack Class

- A generic data structure
  - Access is restricted to the node at the top of the stack
  - Supports five Operations
  - Not encapsulated
  - Stores objects, but primitive types are wrapped automatically in Wrapper objects
  - Expandable

- Client codes
  Stack <Car> garage = new Stack<Car>()
  to create an empty homogeneous stack named garage that can store only Car objects

Five Basic Operations in the Class Stack

Table assumes the heterogeneous structure boston was declared as:
Stack boston = new Stack();
And the code is executed in the order shown