Stacks and Queues

Dr.Haitham A. El-Ghareeb

Information Systems Department
Faculty of Computers and Information Sciences
Mansoura University

helghareeb@gmail.com

October 21, 2012

Stacks and Queues

Stacks and Queues

- Data in a stack are added and removed from only one end of the list.

Stacks and Queues

- Data in a stack are added and removed from only one end of the list.
- Data in a queue are added at one end and removed from the other end of a list.

Stacks and Queues

- Data in a stack are added and removed from only one end of the list.
- Data in a queue are added at one end and removed from the other end of a list.
- Stacks are used extensively in programming language implementations, from everything from expression evaluation to handling function calls.

Stacks and Queues

- Data in a stack are added and removed from only one end of the list.
- Data in a queue are added at one end and removed from the other end of a list.
- Stacks are used extensively in programming language implementations, from everything from expression evaluation to handling function calls.
- Queues are used to prioritize operating system processes and to simulate events in the real world, such as teller lines at banks and the operation of elevators in buildings.

Stacks and Queues

- Data in a stack are added and removed from only one end of the list.
- Data in a queue are added at one end and removed from the other end of a list.
- Stacks are used extensively in programming language implementations, from everything from expression evaluation to handling function calls.
- Queues are used to prioritize operating system processes and to simulate events in the real world, such as teller lines at banks and the operation of elevators in buildings.
- C# provides two classes for using these data structures: the Stack class and the Queue class.

Stacks

- One of the most frequently used data structures.
- Stack is a list of items that are accessible only from the end of the list, which is called the top of the stack. Known as a Last-in, First-out (LIFO) data structure.

Stack Operations

The two primary operations of a stack are adding items to the stack and taking items off the stack.

- The Push operation adds an item to a stack.
- We take an item off the stack with a Pop operation.
- The other primary operation to perform on a stack is viewing the top item (Peek). The Pop operation returns the top item, but the operation also removes it from the stack.
- A stack is completed emptied by calling the Clear operation.
- t is also useful to know how many items are in a stack at any one time. We do this by calling the Count property.

Stack Operations (Cont.)

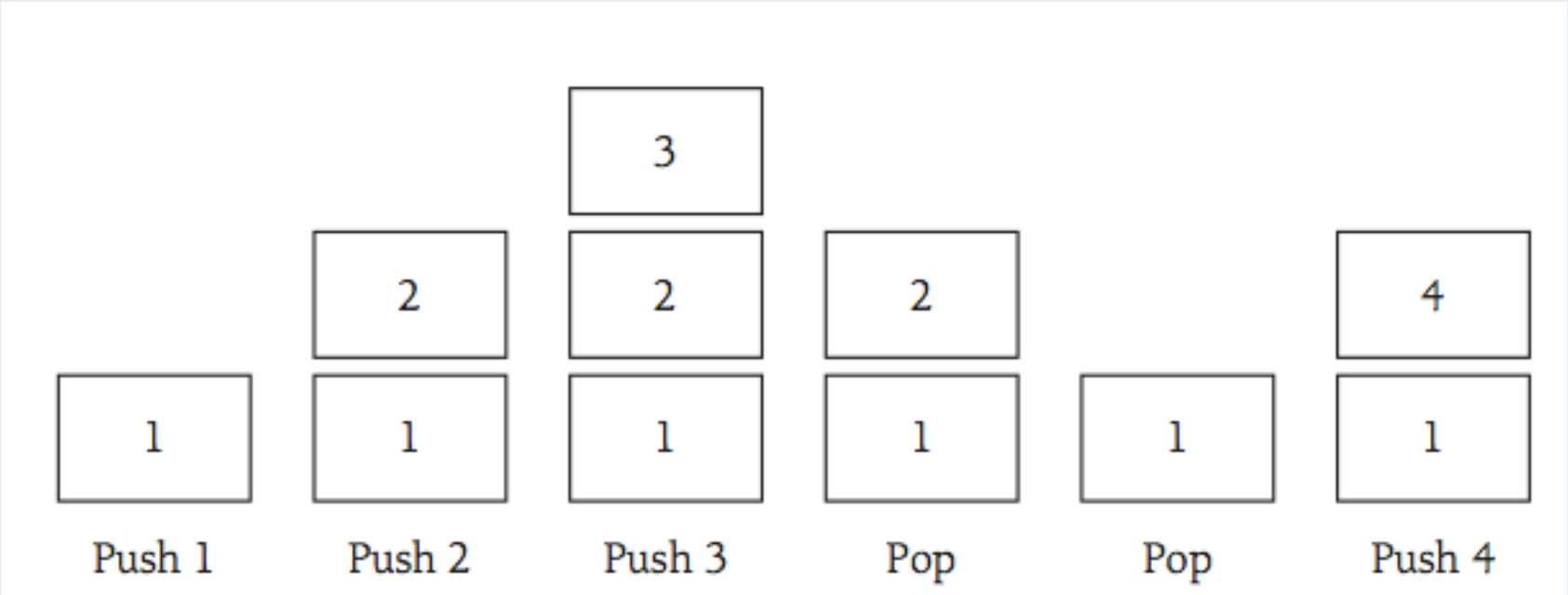

Figure: Stack Two Main Operations: Push, and Pop

Stack Constructors

- Stack(): Initializes a new instance of the Stack class that is empty and has the default initial capacity.
- Stack(ICollection): Initializes a new instance of the Stack class that contains elements copied from the specified collection and has the same initial capacity as the number of elements copied.
- Stack(Int32): Initializes a new instance of the Stack class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

Stack Properties

- Count: Gets the number of elements contained in the Stack.
- IsSynchronized: Gets a value indicating whether access to the Stack is synchronized (thread safe).
- SyncRoot: Gets an object that can be used to synchronize access to the Stack.

Stack Properties

- Clear: Removes all objects from the Stack.
- Clone: Creates a shallow copy of the Stack.
- Contains: Determines whether an element is in the Stack.
- CopyTo: Copies the Stack to an existing one-dimensional Array, starting at the specified array index.
- Equals(Object): Determines whether the specified object is equal to the current object. (Inherited from Object.)
- Finalize: Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
- GetEnumerator: Returns an IEnumerator for the Stack.

Stack Properties (Cont.)

- GetHashCode: Serves as a hash function for a particular type. (Inherited from Object.)
- GetType: Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone: Creates a shallow copy of the current Object. (Inherited from Object.)
- Peek: Returns the object at the top of the Stack without removing it.
- Pop: Removes and returns the object at the top of the Stack.
- Push: Inserts an object at the top of the Stack.
- Synchronized: Returns a synchronized (thread safe) wrapper for the Stack.
- ToArray: Copies the Stack to a new array.
- ToString: Returns a string that represents the current object. (Inherited from Object.)

Stack Remarks

- Stack is implemented as a circular buffer.
- The capacity of a Stack is the number of elements the Stack can hold. As elements are added to a Stack, the capacity is automatically increased as required through reallocation.
- If Count is less than the capacity of the stack, Push is an O(1) operation. If the capacity needs to be increased to accommodate the new element, Push becomes an O(n) operation, where n is Count. Pop is an O(1) operation.
- Stack accepts null as a valid value and allows duplicate elements.

Stack Example

```
using System;
using System.Collections;
public class SamplesStack  {
    public static void Main()  {
        // Creates and initializes a new Stack.
        Stack myStack = new Stack();
        myStack.Push("Hello");
        myStack.Push("World");
        myStack.Push("!");

        // Displays the properties and values of the Stack.
        Console.WriteLine( "myStack" );
        Console.WriteLine( "\tCount:    {0}", myStack.Count );
        Console.Write( "\tValues:" );
        PrintValues( myStack );
    }

    public static void PrintValues( IEnumerable myCollection )  {
        foreach ( Object obj in myCollection )
            Console.Write( "    {0}", obj );
        Console.WriteLine();
    }
}
```

Building Our Own Stack

```
class CStack {
private int p_index;
private ArrayList list;
public CStack( ) {
list = new ArrayList();
p_index = -1;
}
public int count {
get {
     return list.Count;
     }
    }
    public void push(object item) {
     list.Add(item);
     p_index++;
    }
```

Building Our Own Stack (Cont.)

```
    public object pop( ) {
     object obj = list[p_index];
     list.RemoveAt(p_index);
     p_index--;
     return obj;
    }
    public void clear( ) {
     list.Clear();
     p_index = -1;
    }
    public object peek( ) {
     return list[p_index];
    }
}
```

Using Our Own Code

```
static void Main(string[ ] args) {
CStack alist = new CStack();
string ch;
string word = "sees";
bool isPalindrome = true;
for(int x = 0; x < word.Length; x++)
alist.push(word.Substring(x, 1));
int pos = 0;
while (alist.count > 0) {
ch = alist.pop().ToString();
if (ch != word.Substring(pos,1)) {
   isPalindrome = false;
break;
}
pos++;
}
    if (isPalindrome)
     Console.WriteLine(word + " is a palindrome.");
    else
     Console.WriteLine(word + " is not a palindrome.");
    Console.Read();
}
```

Task

Decimal to Multiple-Base Conversion

Queues

- Queue is a data structure where data enters at the rear of a list and is removed from the front of the list.
- Queues are used to store items in the order in which they occur.
- Queues are an example of a first-in, first-out (FIFO) data structure.
- Queues are used to order processes submitted to an operating system or a print spooler, and simulation applications use queues to model customers waiting in a line.

Queue Constructors

- Queue( ): Initializes a new instance of the Queue class that is empty, has the default initial capacity, and uses the default growth factor.
- Queue(ICollection): Initializes a new instance of the Queue class that contains elements copied from the specified collection, has the same initial capacity as the number of elements copied, and uses the default growth factor.
- Queue(Int32): Initializes a new instance of the Queue class that is empty, has the specified initial capacity, and uses the default growth factor.
- Queue(Int32, Single): Initializes a new instance of the Queue class that is empty, has the specified initial capacity, and uses the specified growth factor.

Queue Properties

- Count: Gets the number of elements contained in the Queue.
- IsSynchronized: Gets a value indicating whether access to the Queue is synchronized (thread safe).
- SyncRoot: Gets an object that can be used to synchronize access to the Queue.

Queue Methods

- Clear: Removes all objects from the Queue.
- Clone: Creates a shallow copy of the Queue.
- Contains: Determines whether an element is in the Queue.
- CopyTo: Copies the Queue elements to an existing one-dimensional Array, starting at the specified array index.
- Dequeue: Removes and returns the object at the beginning of the Queue.
- Enqueue: Adds an object to the end of the Queue.
- Equals(Object): Determines whether the specified object is equal to the current object. (Inherited from Object.)
- Finalize: Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.

Queue Methods (Cont.)

- GetEnumerator: Returns an enumerator that iterates through the Queue.
- GetHashCode: Serves as a hash function for a particular type. (Inherited from Object.)
- GetType: Gets the Type of the current instance. (Inherited from Object.)
- MemberwiseClone: Creates a shallow copy of the current Object. (Inherited from Object.)
- Peek: Returns the object at the beginning of the Queue without removing it.
- Synchronized: Returns a Queue wrapper that is synchronized (thread safe).
- ToArray: Copies the Queue elements to a new array.
- ToString: Returns a string that represents the current object. (Inherited from Object.)
- TrimToSize Sets the capacity to the actual number of elements in the

Queue Remarks

- Queues are useful for storing messages in the order they were received for sequential processing. This class implements a queue as a circular array. Objects stored in a Queue are inserted at one end and removed from the other.
- The capacity of a Queue is the number of elements the Queue can hold. As elements are added to a Queue, the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling TrimToSize.
- The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the Queue is constructed. The default growth factor is 2.0. The capacity of the Queue will always increase by at least a minimum of four, regardless of the growth factor. For example, a Queue with a growth factor of 1.0 will always increase in capacity by four when a greater capacity is required.
- Queue accepts null as a valid value and allows duplicate elements.

Queue Example

```
using System;
using System.Collections;
public class SamplesQueue  {

   public static void Main()  {

      // Creates and initializes a new Queue.
      Queue myQ = new Queue();
      myQ.Enqueue("Hello");
      myQ.Enqueue("World");
      myQ.Enqueue("!");

      // Displays the properties and values of the Queue.
      Console.WriteLine( "myQ" );
      Console.WriteLine( "\tCount:    {0}", myQ.Count );
      Console.Write( "\tValues:" );
      PrintValues( myQ );
   }
}
```

Queue Example (Cont.)

```
public static void PrintValues( IEnumerable myCollection )  {
        foreach ( Object obj in myCollection )
            Console.Write( "    {0}", obj );
        Console.WriteLine();
    }
```

Two Queue Approaches

- First come, first served
- Priority-based processing

Memory Management of Queues

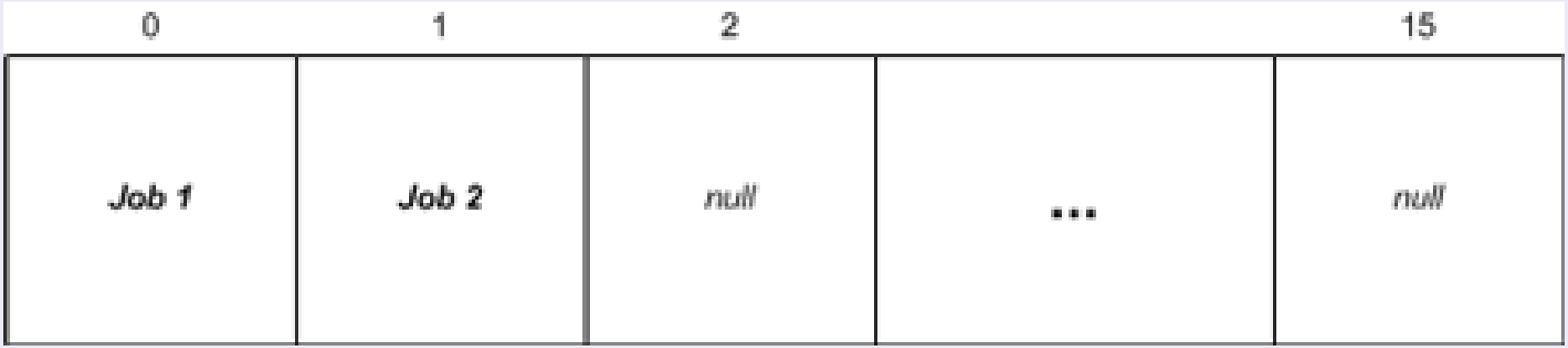

Figure: The ArrayList after the first two lines of code

Memory Management of Queues

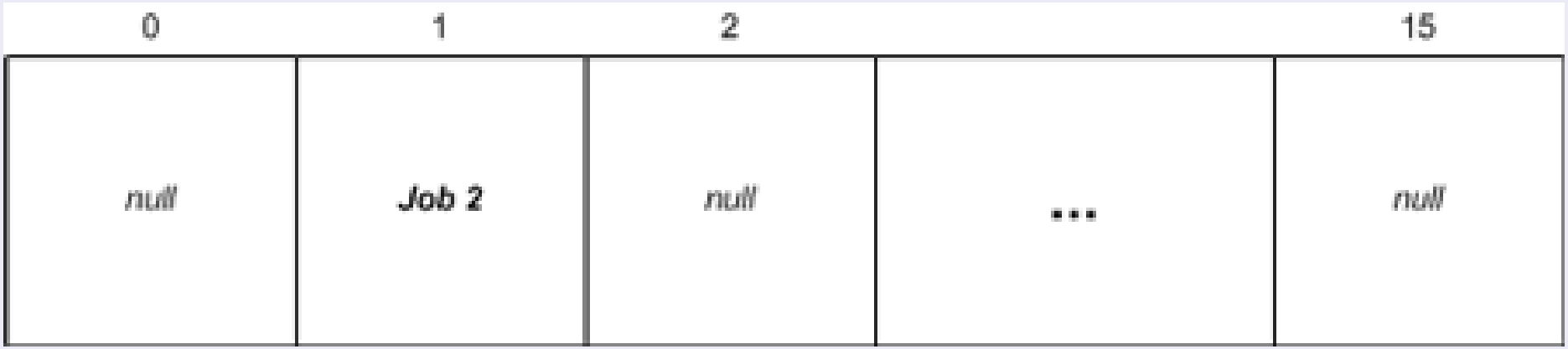

Figure: Program after the GetNextJob() method is invoked

Memory Management of Queues

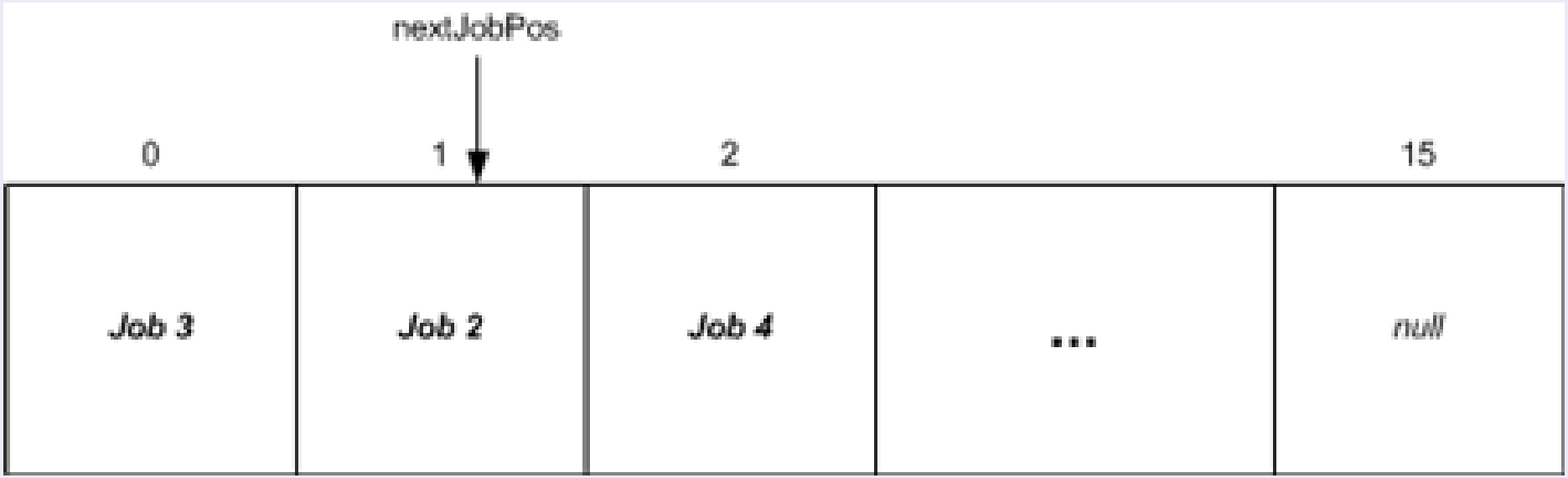

Figure: Issue Created by placing jobs in the 0 index

Memory Management of Queues

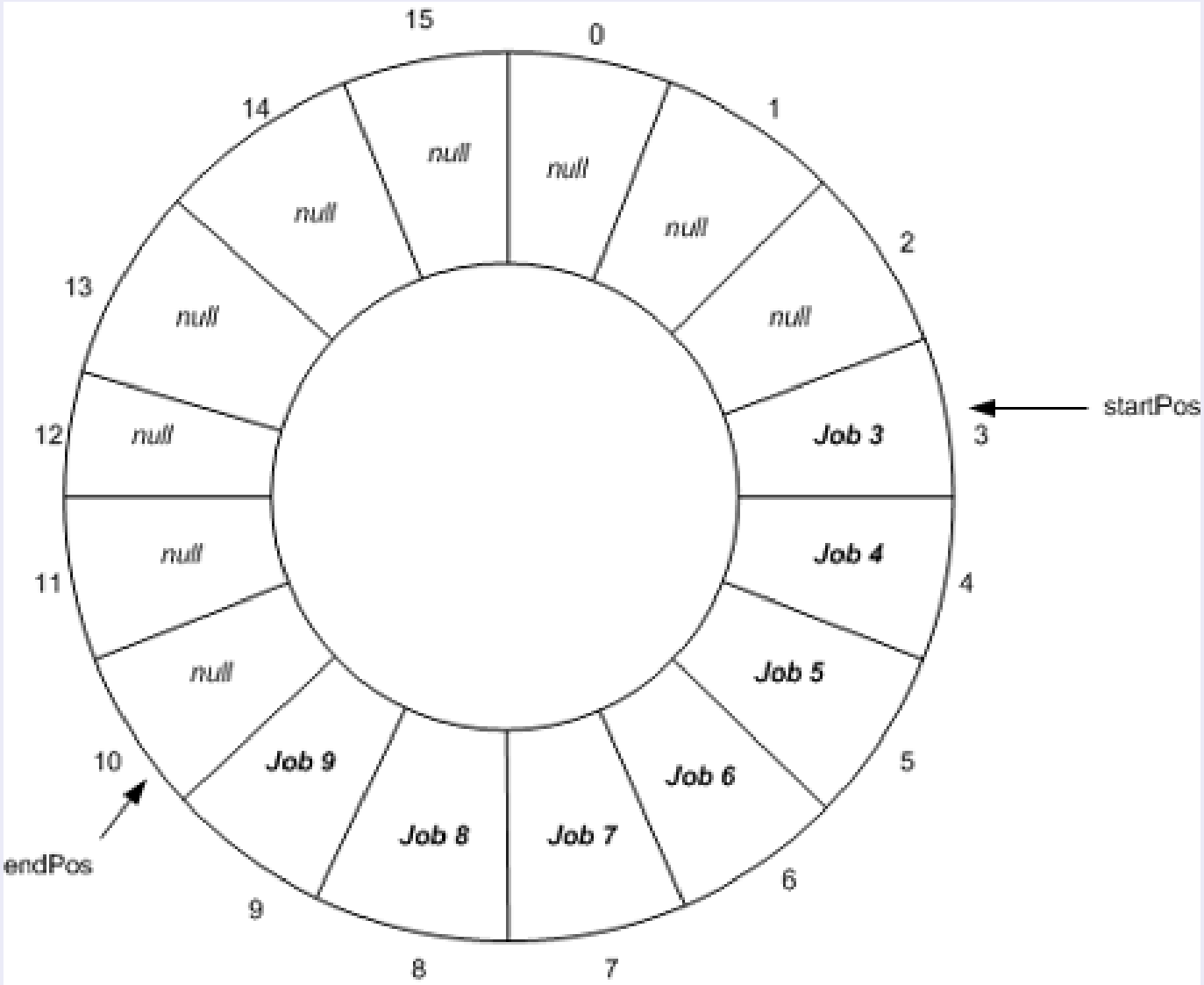

Figure: Example of a Circular Array

Queue Operations

- Enqueu
- Dequeu

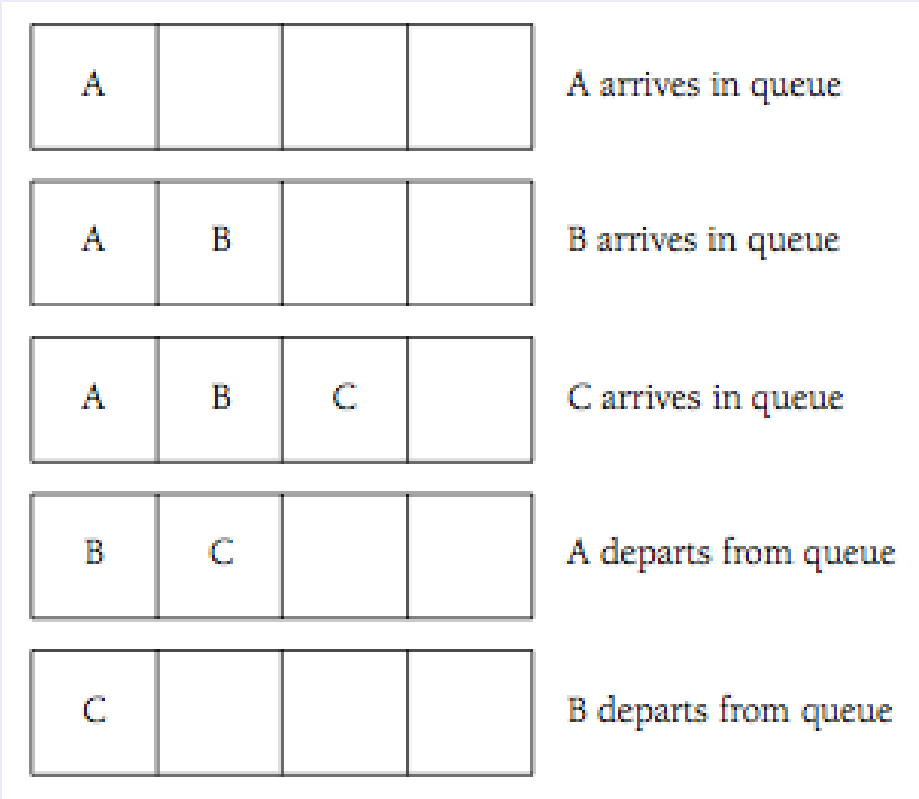

Figure: Queue Operations

Priority Queues

- For many applications, a data structure is needed where an item with the highest priority is removed first, even if it isnt the oldest item in the structure.
- There are many applications that utilize priority queues in their operations.
- A good example is process handling in a computer operating system. Certain processes have a higher priority than other processes, such as printing processes, which typically have a low priority.
- Processes (or tasks) are usually numbered by their priority, with a Priority 0 process having a higher priority than a Priority 20 task.
- Items stored in a priority queue are normally constructed as keyvalue pairs, where the key is the priority level and the value identifies the item.

Happy Eid