Queues

Queues

- Queue is a linear data structure that only allows items to be inserted at the end or tail of the queue and removed from the front or head of the queue.
- “Queue” is the British word for a line (or line-up)
- Queues are **FIFO** (First In First Out) data structures.

What Can You Use a Queue For?

- Processing inputs and outputs to screen (console)
- Server requests
  - Instant messaging servers queue up incoming messages
  - Database requests
- Print queues
  - One printer for dozens of computers
- Operating systems use queues to schedule CPU jobs

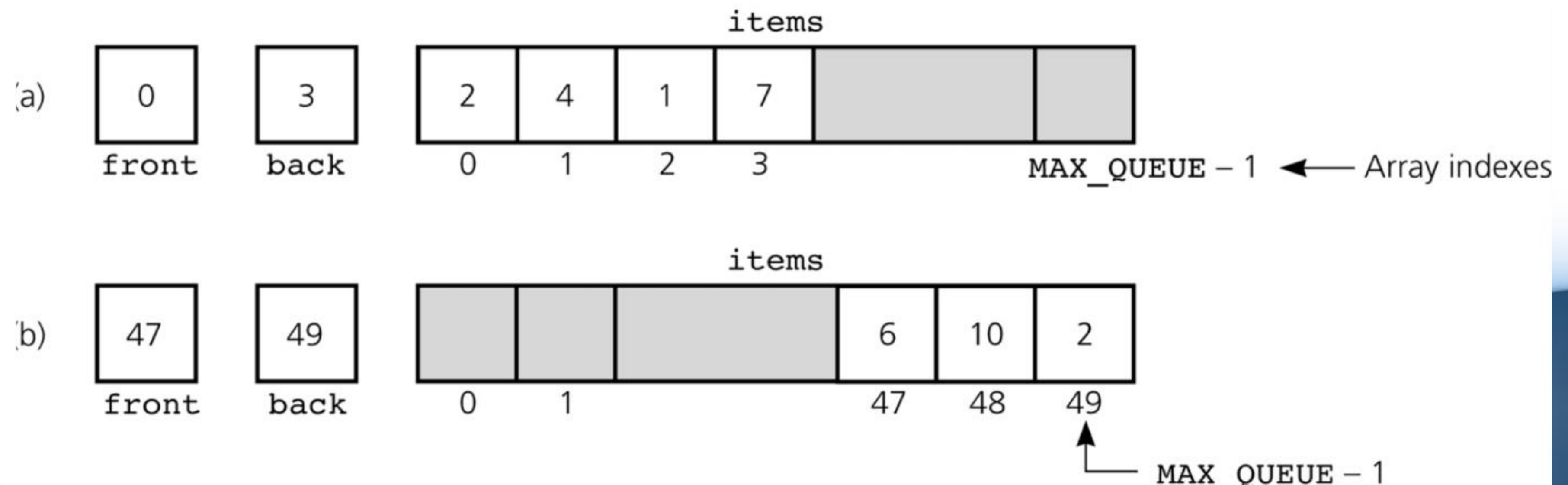
Queue Operations

- A queue should implement (at least) these operations:
  - **enqueue** – insert an item at the back of the queue
  - **dequeue** – remove an item from the front
  - **peek** – return the item at the front of the queue without removing it

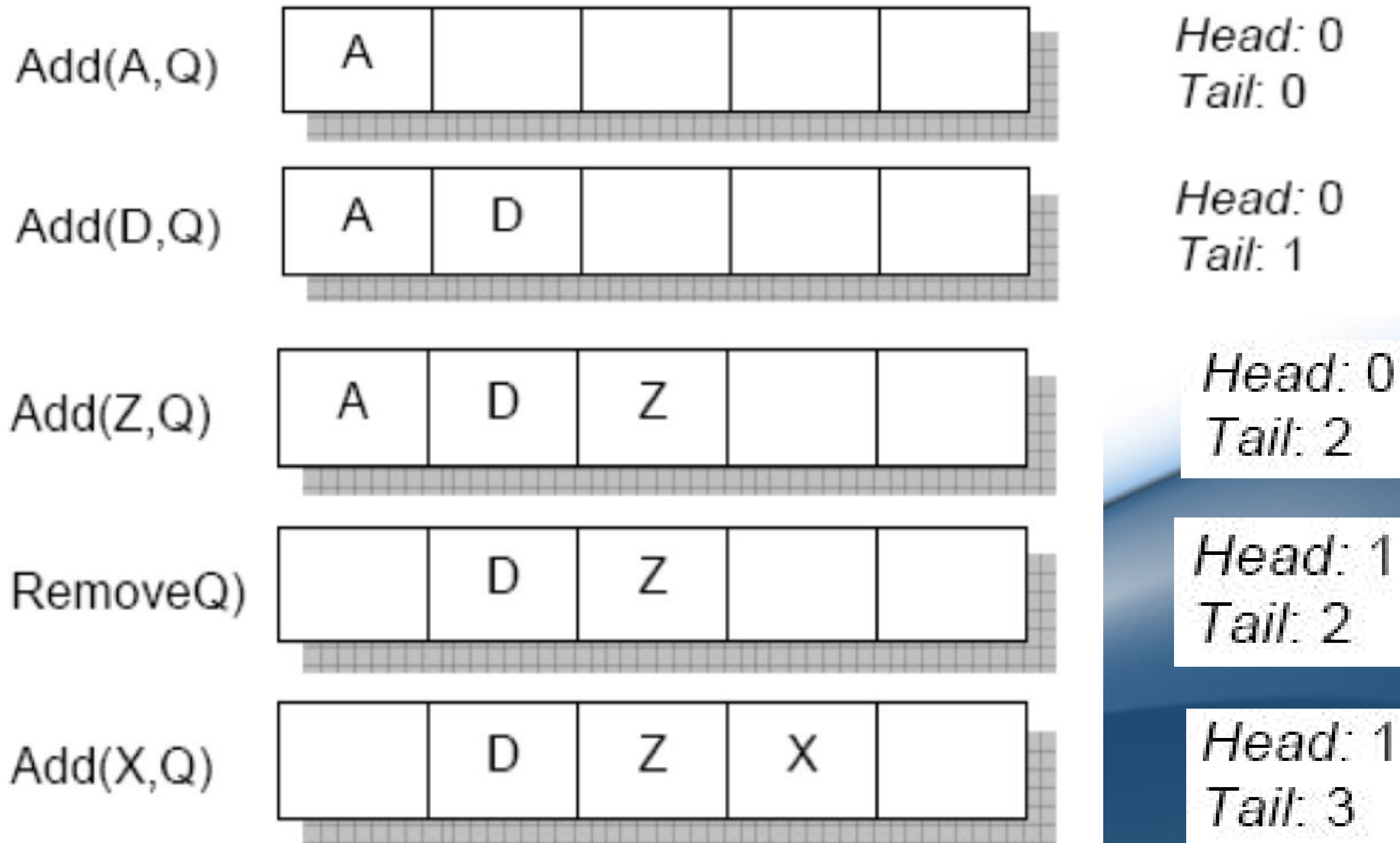
Queue: Indexed Implementation

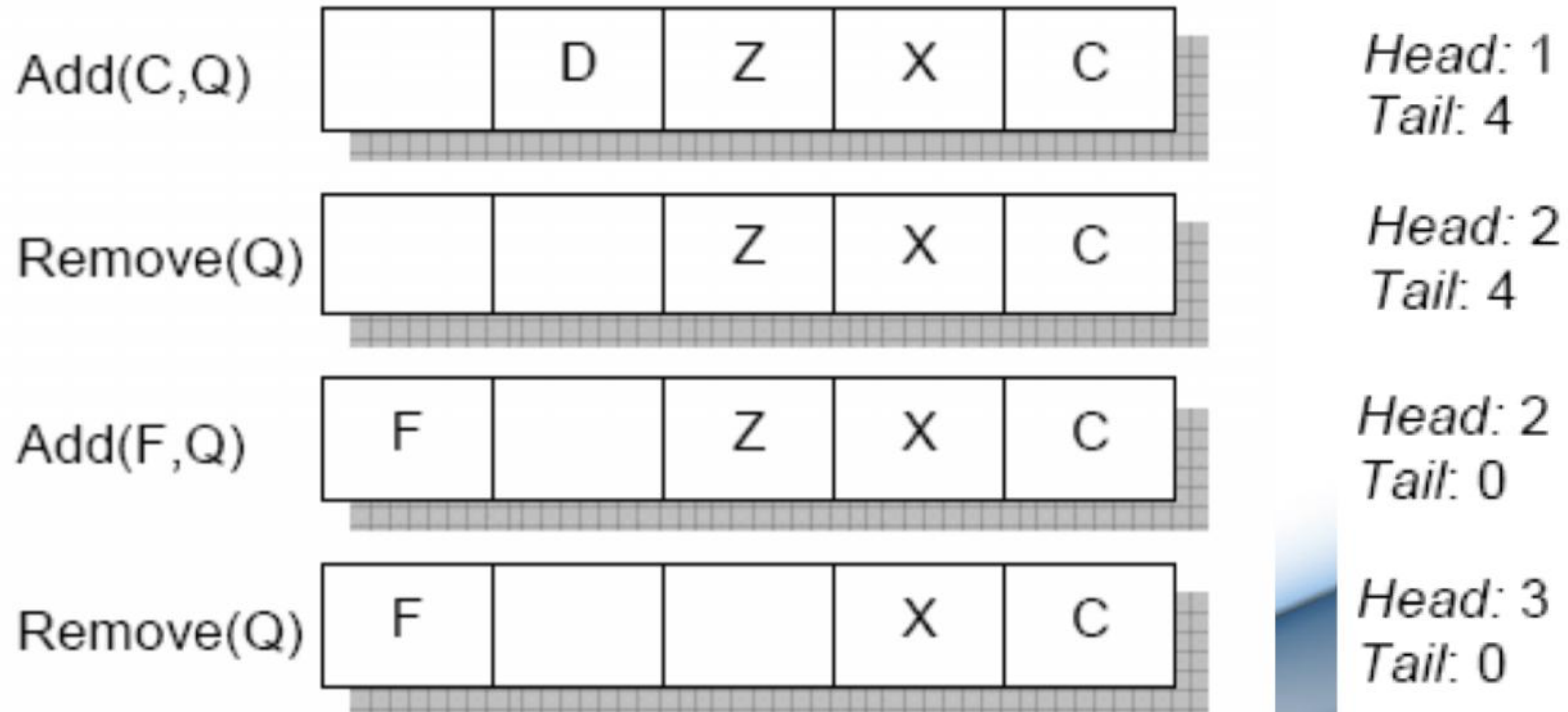
- Consider using an array as the underlying structure for a queue, one plan would be to
  - Make the back of the queue the current size of the queue (i.e., the number of elements stored)
  - Make the front of the queue index 0
  - Inserting an item can be performed in constant time
  - But removing an item would require shifting all elements in the queue to the left which is **too slow**!
- **Therefore we need to find another way**
- The array indices at which the head and tail of the queue are currently stored must be maintained.

An Array-Based Implementation



- Example – storing a queue in an array of length 5



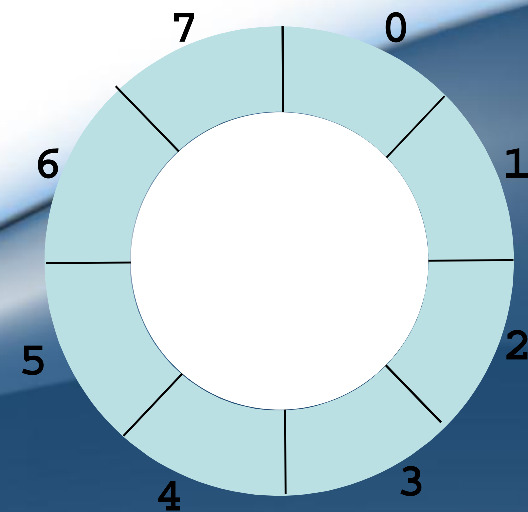

Add(C,Q)
D
Z
X
C
Head: 1
Tail: 4
Remove(Q)
Z
X
C
Head: 2
Tail: 4
Add(F,Q)
F
Z
X
C
Head: 2
Tail: 0
Remove(Q)
F
X
C
Head: 3
Tail: 0

```
public class SimpleQueue
{
    private Object[] queue;
    private int size;       // size of the queue
    private int head;    // for next removal
    private int tail;       // for next insertion
    public SimpleQueue(int capacity)
    {
          queue = new Object[capacity];
          size = head = tail = 0;
    }
    public boolean isEmpty()
    { return size == 0; }

     public boolean isFull()
    { return size == queue.length; }
```

```
public void enqueue(Object item)
 {
     if (isFull())
     { throw new RuntimeException("queue overflow"); }
     else
       { queue[tail++] = item;
         ++size;
     }
}
public Object dequeue()
{
     if (isEmpty())
     { throw new RuntimeException("queue underflow"); }
      else
      {
        Object item = queue[head++];
        --size;
        return item;
      }
 }
}
```

Circular Arrays

- Use a **circular array** to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array "wraps around" to the start of the array.



The mod Operator

- The `mod` operator (%) is used to calculate remainders:
  - **`1%5 = 1, 2%5 = 2, 5%5 = 0, 8%5 = 3`**
- `mod` can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size
  - The back(rear) of the queue is:
    - **`rear=(front + count) % items.length`**
    - where `count` is the number of items currently in the queue
  - After removing an item the front of the queue is:
    - **`front=(front) % items.length;`**
  - insert item at :
    - (front + count) % items.length

Array Queue Example

front =	1
count =	1

6					

	6				
0	1	2	3	4	5

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
```

insert item at (front + count) % items.length
Rear=front+count % items.length

Array Queue Example

front =	1
count =	5

	6	4	7	3	8
0	1	2	3	4	5

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
```

insert item at (front + count) % items.length
Rear=front+count % items.length

Array Queue Example

front	=	2
count	=	4

6	4	7	3	8	9
0	1	2	3	4	5

make front = (1) % 6 = 1

make front = (2) % 6 = 2

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);
```

Array Queue Example

front	=	2
count	=	5

5		7	3	8	9
0	1	2	3	4	5

```
//Java Code
Queue q = new Queue();
q.enqueue(6);
q.enqueue(4);
q.enqueue(7);
q.enqueue(3);
q.enqueue(8);
q.dequeue();//front = 1
q.dequeue();//front = 2
q.enqueue(9);
q.enqueue(5);
```

insert at (front + count) % 6
=
(2 + 4) % 6
= 0

```
public interface Queue
{
void insert (Object o);
boolean isEmpty ();
boolean isFull ();
Object remove ();
}
```

```
public class ArrayCircularQueue implements Queue
{
private int front= 1,rear = 0,count=0;
private Object []queue;

public ArrayCircularQueue (int maxElements)
{queue=new Object [maxElements];}

public void insert(Object o)
{
  if(count==queue.length)
  {
    throw new RuntimeException("queue full");
  }
   rear=(front+count)%queue.length;
   queue[rear]= o;
   ++count;
}
```

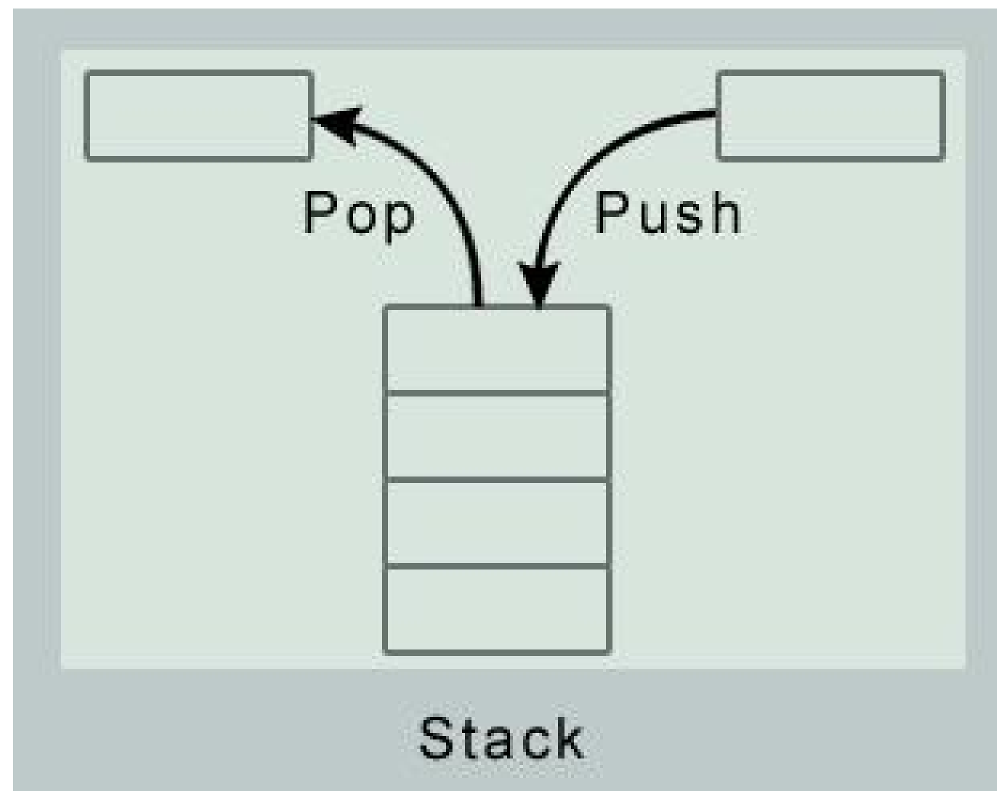
```
public boolean isEmpty()
{
return count==0;
}
public boolean isFull()
{
return count==queue.length;
}

public Object remove()
{
if(isEmpty())
    throw new RuntimeException("Queue empty");
front=(front)%queue.length;
count--;
return queue[front++];
}
```

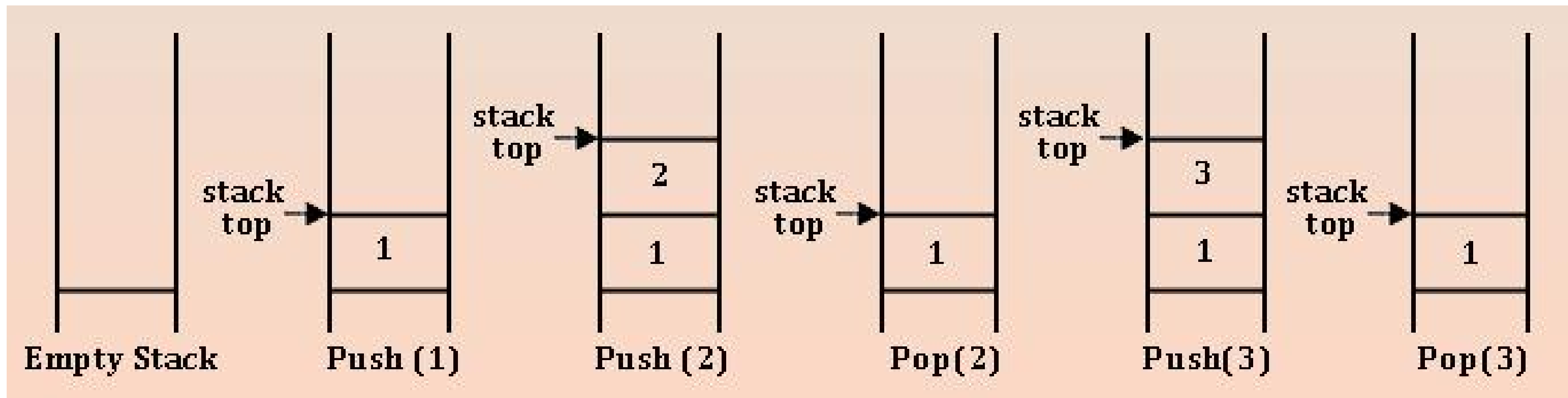
```
public static void main(String[] a)
{
    ArrayCircularQueue ac=new ArrayCircularQueue(5);
    ac.insert(new Integer(1));
    ac.insert(new Integer(2));
    ac.insert(new Integer(3));
    ac.insert(new Integer(4));
    ac.insert(new Integer(5));
    System.out.println(ac.remove());
    System.out.println(ac.remove());
    System.out.println(ac.remove());
    System.out.println(ac.remove());
    ac.insert(new Integer(6));
    System.out.println(ac.remove());
}
}
```

Stacks

Overview

- **A stack is a last-in first-out data structure (LIFO).**
- Mainly two action are performed by stack one is push and other is pop.
- The last thing which we placed or push on stack is the first thing we can get when we pop.
- A stack is a list of element with insertion and deletion at one end only.
- The drawback of implementing stack is that the size of stack is fixed and it cannot grow or shrink.


Pop
Push
Stack


Empty Stack
stack top
1
Push (1)
stack top
2
1
Push (2)
stack top
1
Pop(2)
stack top
3
1
Push(3)
stack top
1
Pop(3)

Implementation

- Implementation of array-based stack is very simple.
- It uses **top** variable to point to the topmost stack's element in the array.

1. Initially **top = -1;**
2. **push** operation increases top by one and writes pushed element to **storage[top];**
3. **pop** operation checks that top is not equal to -1 and decreases top variable by 1;
4. **peek** operation checks that top is not equal to -1 and returns **storage[top];**
5. **isEmpty** returns boolean **(top == -1).**

Stack class

- Subclass of Vector that implements a standard last-in, first-out stack.
- Stack only defines the default constructor, which creates an empty stack.

Methods

- **boolean empty()**
  Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
- **Object peek( )**
  Returns the element on the top of the stack, but does not remove it.
- **Object pop( )**
  Returns the element on the top of the stack, removing it in the process.
- **Object push(Object element)**
  Pushes element onto the stack. Element is also returned.

Methods contd…

- **int search(Object element)**
  Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

```
import java.io.*;
import java.util.*;

public class StackImplement
{
    Stack<Integer> stack;
    String str;
    int num, n;
    public static void main(String[] args)
    {
        StackImplement q = new StackImplement();
    }
```

```
public StackImplement()
{
        try{
        stack = new Stack<Integer>();
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(ir);
        System.out.print("Enter number of elements : ");
        str = bf.readLine();
        num = Integer.parseInt(str);
        for(int i = 1; i <= num; i++){
                System.out.print("Enter elements : ");
                str = bf.readLine();
                n = Integer.parseInt(str);
                stack.push(n);
        }}
        catch(IOException e){  }
        System.out.print("Retrieved elements from the stack : ");
        while (!stack.empty()){
                System.out.print(stack.pop()+"  ");
        }}}
```

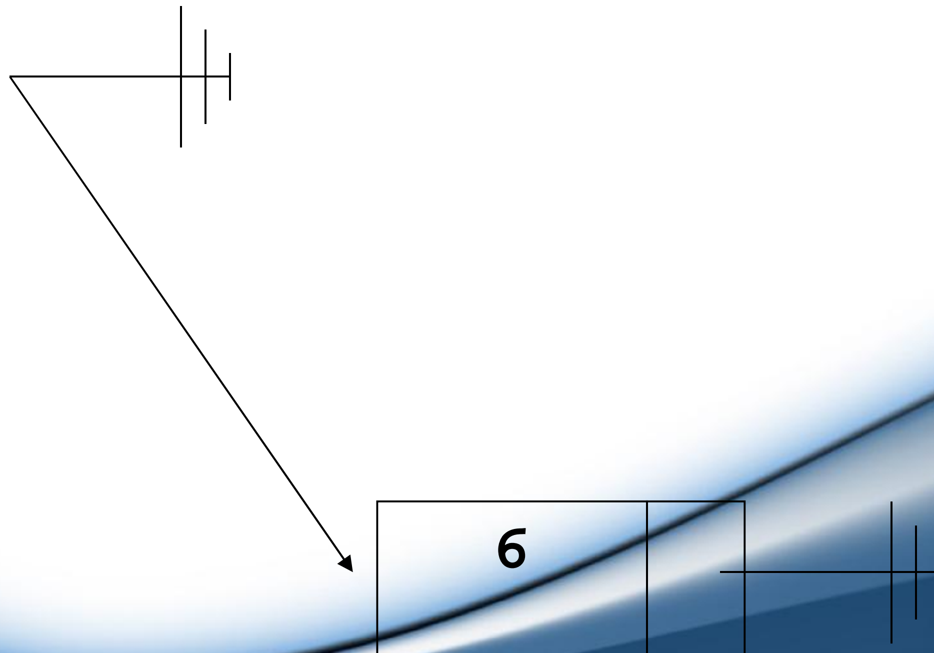
Linked List Implementation

- Using a linked list is one way to implement a stack so that it can handle *essentially* any number of elements.
- **Push and pop at the head of the list**
  - New nodes should be inserted at the front of the list, so that they become the top of the stack
  - Nodes are removed from the front (top) of the list.

List Stack Example

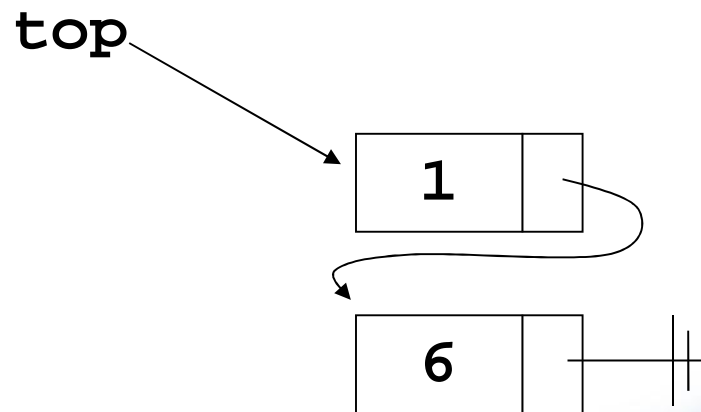
Java Code

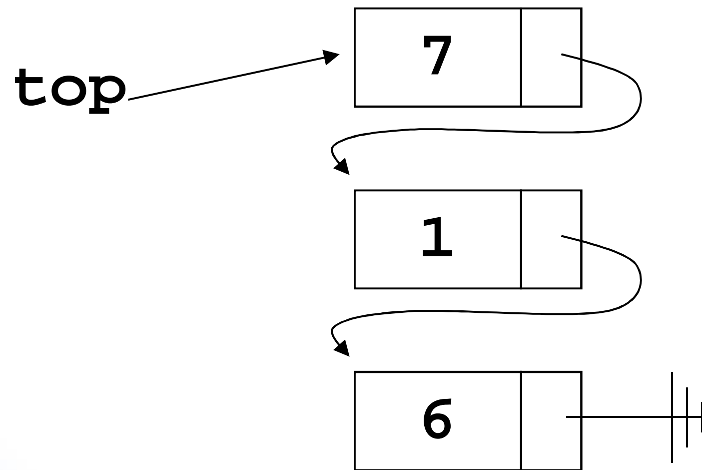
```
Stack st = new Stack();
st.push(6);
```



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
```



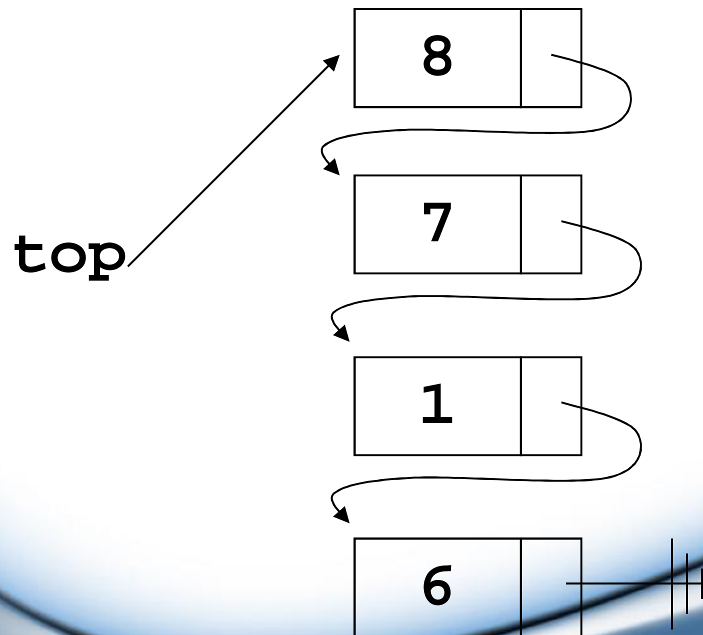


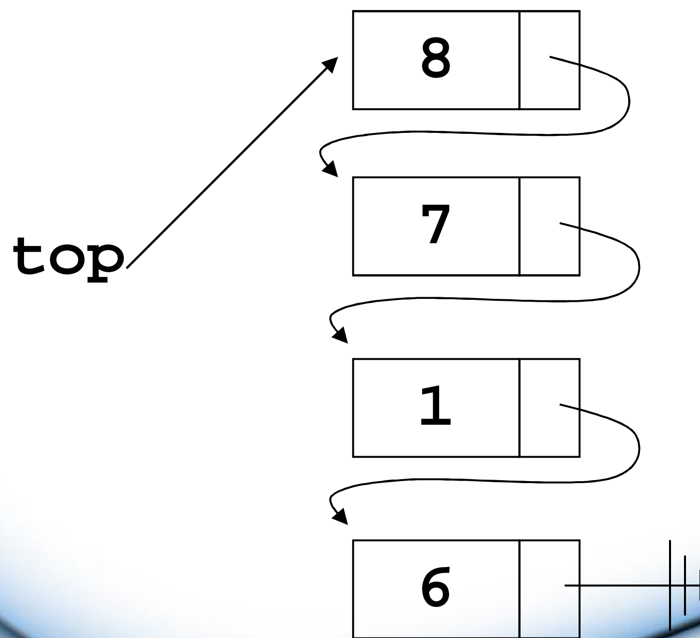
Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
```

Java Code

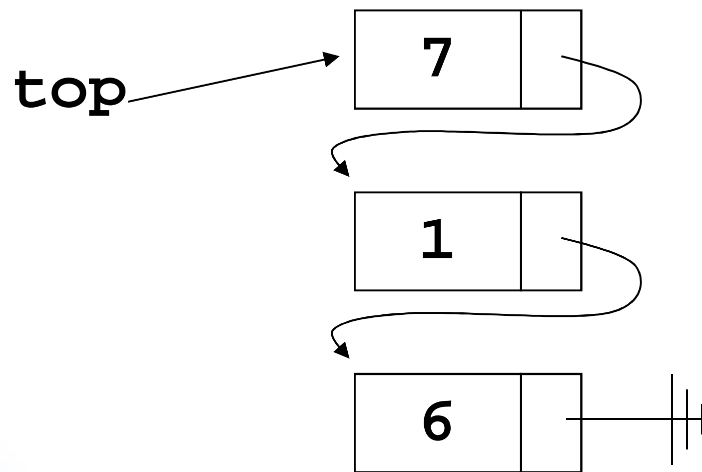
```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```





Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

```
/** Cell models a cell of a linked list. */
public class Cell
{
    private Object val; // value in the cell
    private Cell next; // the address of the next cell in the list

    /** Constructor Cell builds a new cell
    public Cell(Object value, Cell link)
    {
            val = value;
              next = link;
       }
    /** getVal returns the value held in the cell */
    public Object getVal()
    { return val; }
```

```
/** getNext returns the address of the cell
   chained to this one */
   public Cell getNext()
   { return next; }

   /** setNext resets the address of the cell
   chained to this one */
   public void setNext(Cell link)
   { next = link; }
 }
```

```
/** Stack implements a stack as a linked list */
 public class Stack
{
    private Cell top; // marks the topmost Cell of the stack

/** Constructor Stack creates an empty stack */
   public Stack()
   { top = null; }

 /** push inserts a new element onto the stack */
   public void push(Object ob)
   { top = new Cell(ob, top); }
```

```
/** pop removes the most recently added element */
public Object pop()
      {
     if ( top == null )
      { throw new RuntimeException("Stack error: stack empty"); }
     Object answer = top.getVal();
     top = top.getNext();
      return answer;
      }
public Object top()
{
     if ( top == null )
     { throw new RuntimeException("Stack error: stack empty"); }
     return top.getVal();
 }
public boolean isEmpty()
 { return (top == null); }
}
```

Applications of Stacks

- Searching networks, traversing trees (keeping a track where we are).

Examples:

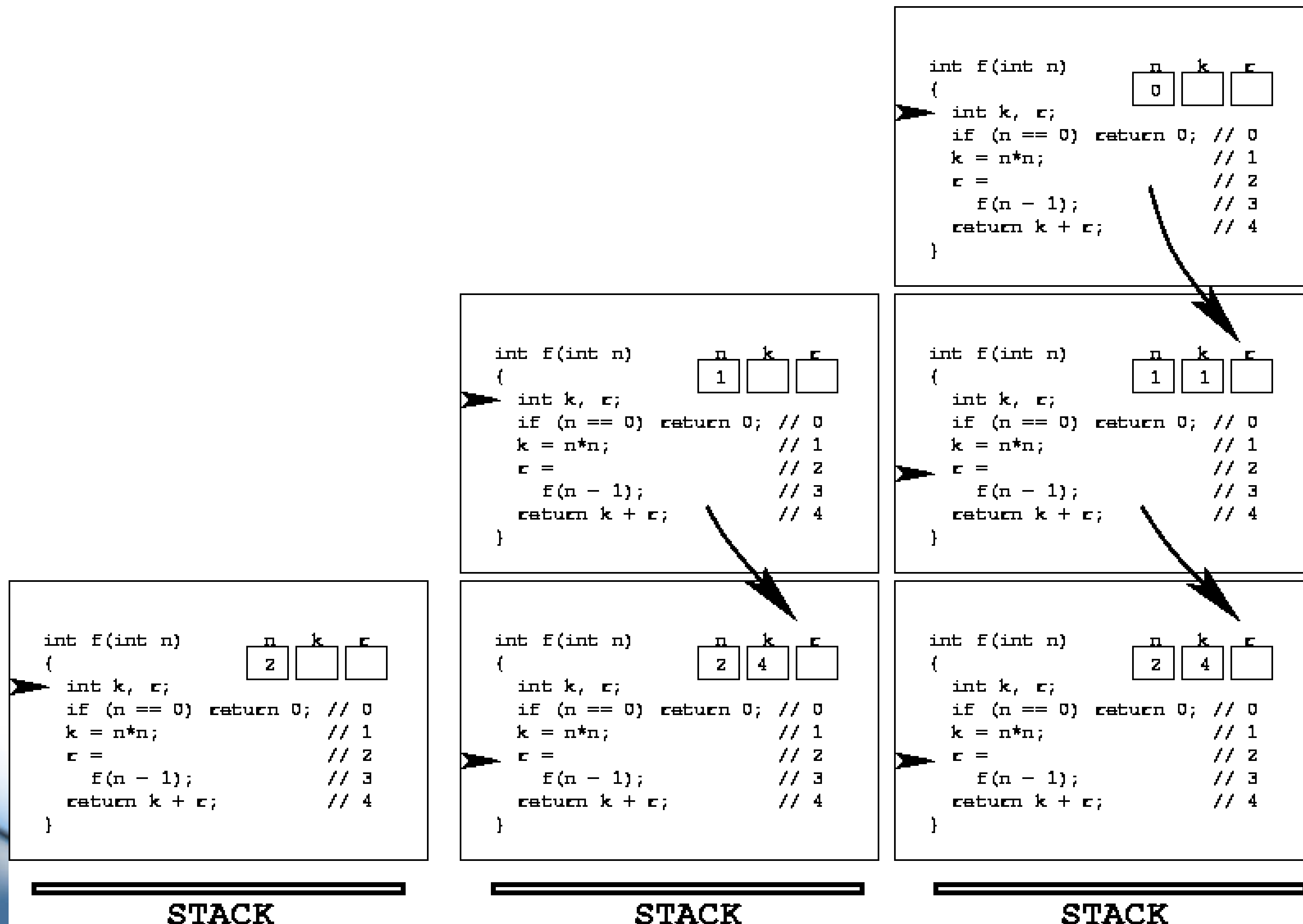
- Checking balanced expressions
- Recognizing palindromes
- Evaluating algebraic expressions

Recursion Using Stacks

```
#include <iostream>

int main()
{
  // Get Input
  int x, y;
  cout << "Enter value x: ";
  cin >> x;

  // Call f
  y = f(x);

  // Print results
  cout << y << endl;
  return 0;
}
```

```
int f(int n)
{
  int k, r;
  if (n == 0)
    return 0;         // 0
  k = n*n;            // 1
  r =                 // 2
      f(n - 1);       // 3
  return k + r;       // 4
}
```

What's Going On…

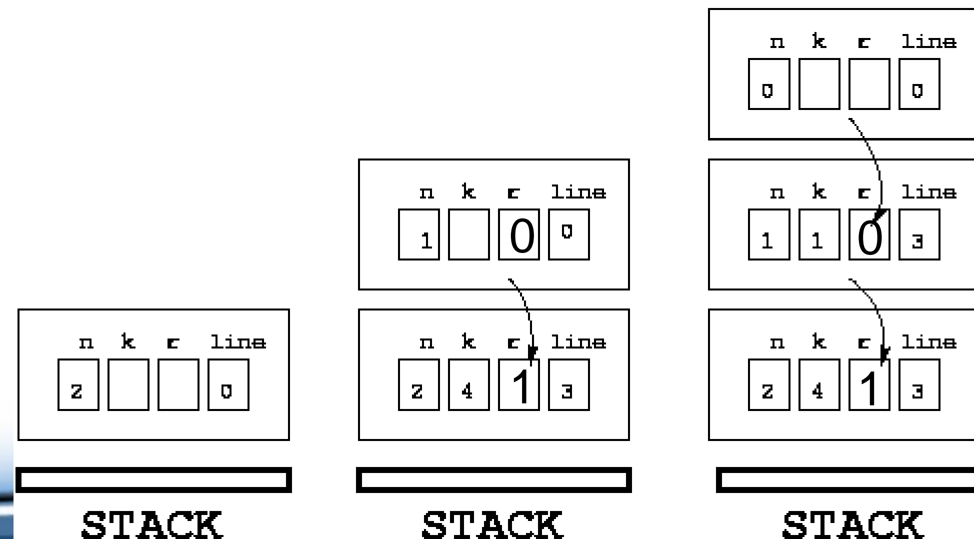


What’s Going On

- Recursive calls to f get pushed onto stack
  - Multiple copies of function f, each with different arguments & local variable values, and at different lines in function
- All computer really needs to remember for each active function call values of arguments & local variables and the location of the next statement to be executed when control goes back
  - “The stack" looks a little more like :

Real Stack

- Looks sort of like bunch of objects
  - Several pieces of data wrapped together
- We can make our own stack & simulate recursive functions ourselves



Data Structures

Overview

- A data structure is the organization of data in a computer's memory or in a disk file for better algorithm efficiency.
- The correct choice of data structure allows major improvements in program efficiency.
- Examples of data structures are arrays, stacks, linked lists etc.

3 steps in the study of data structures

- Logical or mathematical description of the structure.
- Implementation of the structure on the computer.
- Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

Overview

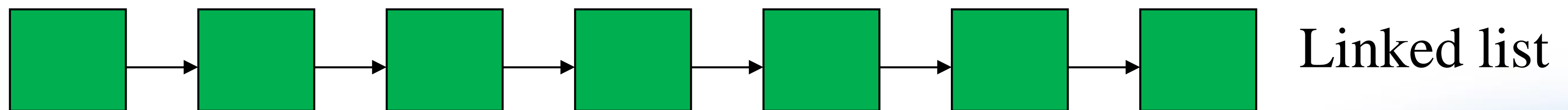
- Data may be organized in many ways
  - E.g., arrays, linked lists, trees etc.
- The choice of particular data model depends on two consideration:
  - It must be rich enough in structure to mirror the actual relationships of data in the real world.
  - The structure should be simple enough that one can effectively process the data when necessary.

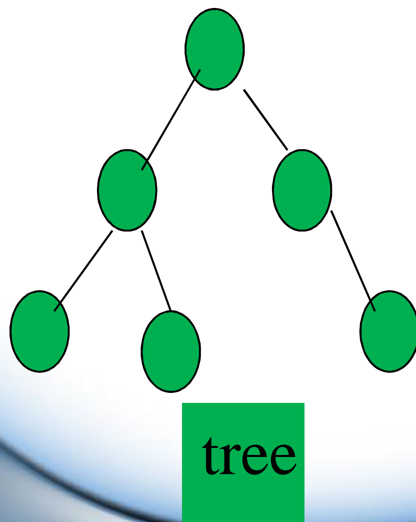
Overview

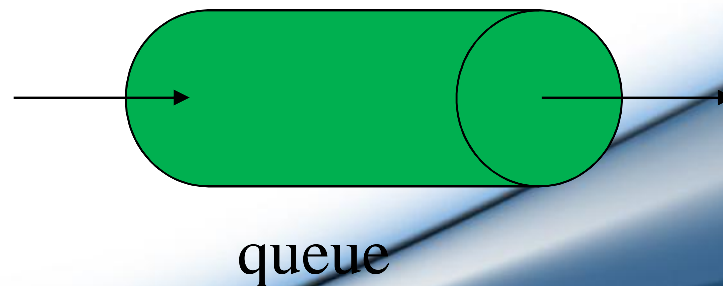
- Data structure for storing data of students:-
  - Arrays
  - Linked Lists
- Issues
  - Space needed
  - Operations efficiency (Time required to complete operations)
    - Retrieval
    - Insertion
    - Deletion
  - Frequency of usage of above operations

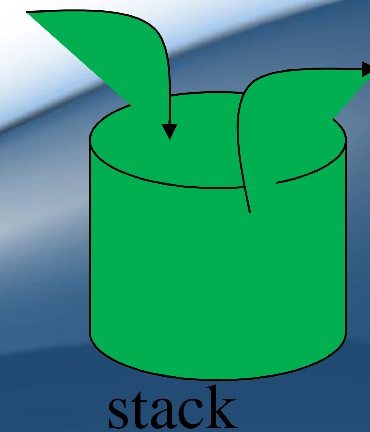
What data structure to use?

Data structures let the input and output be represented in a way that can be handled efficiently and effectively.









What's the difference

- Different types of values.
- Different structures
  - No structure – just a collection of values
  - Linear structure of values – the order matters
  - Set of key-value pairs
  - Hierarchical structures
  - Grid/table
  - ….
- Different access disciplines
  - get, put, remove anywhere
  - get, put, remove only at the ends, or only at the top, or …
  - get, put, remove by position, or by value, or by key, or …
  - ….

Algorithm Review

- An algorithm is a definite procedure for solving a problem in finite number of steps.
- Algorithm is a well defined computational procedure that takes some value (s) as input, and produces some value (s) as output.
- Algorithm is finite number of computational statements that transform input into the output.

Good Algorithms?

- Run in less time
- Consume less memory

But computational resources (time complexity) is usually more important.

Complexity

- In examining algorithm efficiency we must understand the idea of complexity
- Complexity is the consumptions of resources.
- Most important aspect of complexity are
  – Space complexity
  – Time Complexity

Space Complexity

- When memory was expensive we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc.)

Time Complexity

- Is the algorithm “fast enough” for my needs?
- How much longer will the algorithm take if I increase the amount of data it must process?
- Given a set of algorithms that accomplish the same thing, which is the right one to choose?

Algorithm Efficiency

- A measure of the amount of resources consumed in solving a problem of size n
  - time
  - space
- Benchmarking: implement algorithm,
  - run with some specific input and measure time taken
  - better for comparing performance of processors than for comparing performance of algorithms
- Big Oh (Asymptotic analysis)
  - associates n, the problem size, with t, the processing time required to solve the problem

Table 1.1: Characteristics of Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced).	Deletion algorithm is complex.

Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion	Slow access to other items. Access to largest item.
Graph	Models real-world situations.	Some algorithms are slow and complex.

The Basics of Arrays

- Simplest type of data structure is linear array.
- List of finite number **n** of similar data elements referenced by a set of **n** consecutive numbers.
- If A is array then the elements are denoted by subscript notation

A[1], A[2], A[3]……A[N]

The number K in A[K] is called a subscript and A[K] is subscripted variable.

The Basics of Arrays

- Arrays have a length field, which you can use to find the size, in bytes, of an array.
- int arrayLength = intArray.length; // find array length

Accessing Array Elements

- Array elements are accessed using square brackets.
- temp = intArray[3]; // get contents of fourth element of array
- intArray[7] = 66; // insert 66 into the eighth cell

```
import java.io.*;
class ArrayApp
{
public static void main(String[] args) throws
   IOException
{
int[] arr; // reference
arr = new int[100]; // make array
int nElems = 0; // number of items
int j; // loop counter
int searchKey; // key of item to search for
```

```
arr[0] = 77; // insert 10 items
arr[1] = 99;
arr[2] = 44;
arr[3] = 55;
arr[4] = 22;
arr[5] = 88;
arr[6] = 11;
arr[7] = 00;
arr[8] = 66;
arr[9] = 33;
nElems = 10; // now 10 items in array
```

```
for(j=0; j<nElems; j++) // display items
System.out.print(arr[j] + " ");
System.out.println("");
searchKey = 66; // find item with key 66
for(j=0; j<nElems; j++) // for each element,
if(arr[j] == searchKey) // found item?
break; // yes, exit before end
if(j == nElems) // at the end?
System.out.println("Can’t find " + searchKey); //
  yes
else
System.out.println("Found " + searchKey); // no
```

```
searchKey = 55; // delete item with key 55
for(j=0; j<nElems; j++) // look for it
if(arr[j] == searchKey)
break;
for(int k=j; k<nElems; k++) // move higher ones down
arr[k] = arr[k+1];
nElems--; // decrement size

for(j=0; j<nElems; j++) // display items
System.out.print( arr[j] + " ");
System.out.println("");
} // end main()
} // end class ArrayApp
```

Properties of Arrays

- Arrays stores **similar data** types. That is, array can hold data of same data type values. This is one of the limitations of arrays compared to other data structures.
- Each value stored, in an array, is known as an **element** and all elements are indexed. The first element added, by default, gets 0 index. That is, the 5th element added gets an index number of 4.
- Elements can be retrieved by their index number.

Properties of Arrays

- Array elements are stored in **contiguous (continuous) memory locations**.
- Once array is created, its **size is fixed**. That is, at runtime if required, more elements cannot be added. This is another limitation of arrays compared to other data structures.
- One array name can represent multiple values. Array is the easiest way to store a large quantity of data of same data types. For example, to store the salary of 100 employees, it is required to declare 100 variables. But with arrays, with one array name all the 100 employees salaries can be stored.

Properties of Arrays

- Arrays can be multidimensional.
- At the time of creation itself, array size should be declared (array initialization does not require size).
- In Java, arrays are predefined objects. With methods, the array elements can be manipulated.
- As Java does not support garbage values, **unassigned** elements are given **default values**; the same values given to unassigned instance variables.

Copying Arrays (Duplicating Arrays)

- The System class has an **arraycopy** method that can be used to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

```
class ArrayCopyDemo
{
 public static void main(String[] args)
{
char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i',
   'n', 'a', 't', 'e', 'd' };
char[] copyTo = new char[7];
System.arraycopy(copyFrom, 2, copyTo, 0, 7);
System.out.println(new String(copyTo));
} }
```

The output from this program is:

caffein

Thank You