Problem solving technique

Lesson 4: Other Data Structures and techniques

Speed.Agility.Imagination

Lesson Objectives

- **You will learn the following topics in this lesson:**
  - Introduction to Stack
  - Operations on Stacks
    - Push
    - Pop
  - Different ways of implementation
    - Using Arrays
    - Using Linked List
  - Applications of Stack

Lesson Objectives

- Introduction to Queues
- Operation On Queues
  - Insert
  - Delete
- Implementation of Queue
  - Queues using Arrays
  - Queues Using Linked List
- Circular Queues
- Applications of Queues
- What is Hashing
- What is recursion
- Various collection classes in JAVA

IGATE
Speed.Agility.Imagination

Definition

- **A "stack" is an ordered collection of items into which new items may be inserted and elements may be deleted at one end, which is called as "top" of the stack.**

Explanation

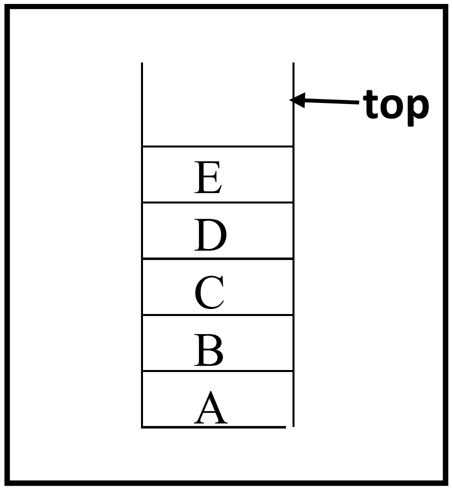
- **A stack implements the concept of Last In First Out (LIFO).**
  - It means last inserted item will be the first item to be deleted.
- **For example: Stack of books**
  - One can place the book or take out the book only from the top.

Explanation

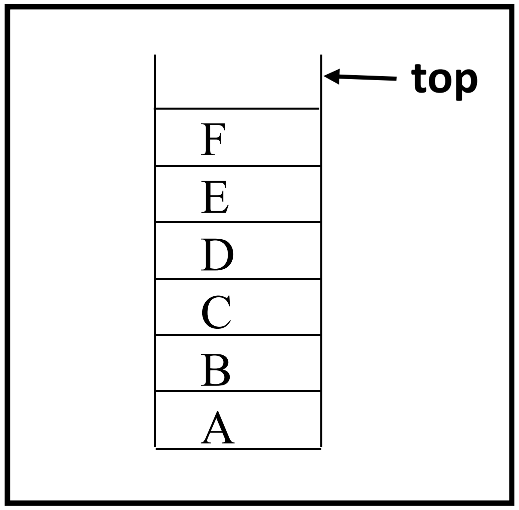
- **Two Basic operations on Stacks are:**
  - **Push:** It inserts item at the TOP of the stack.
  - **Pop:** It deletes item from the TOP of the stack.
- **Other Operations on Stacks are:**
  - **Initstack:** It initializes the stack top.
  - **Isfull:** It checks for overflow of stack.
  - **Isempty:** It checks whether stack is empty (useful to check underflow of the stack).

4.1: Stacks

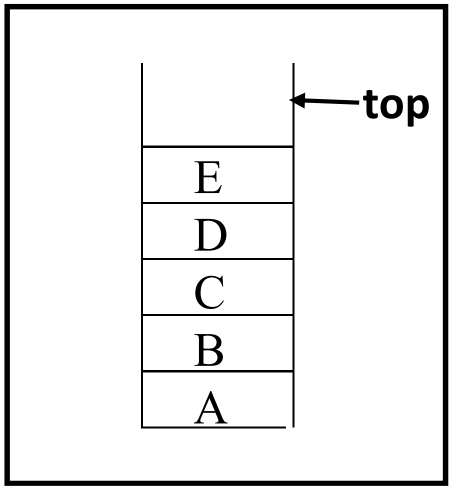
Illustration



Before insertion of item 'F' in the stack
(INITIAL STACK)



After insertion of item 'F' in the stack
(PUSH 'F')



After deletion of item 'F' in the stack
(POP)

Explanation

- **Stack can be implemented in different ways, namely:**
  - By using arrays (Static implementation)
    - There is a limit on maximum number of elements to be stored.
  - By using linked list (Dynamic Implementation)
    - There is no limit on maximum number of elements to be stored.

Explanation

- **Using Arrays:**
  - Using arrays is called as “static implementation”.
  - You can implement the following modules, namely:
    - Initialize – to initialize the stack
    - Push
    - Pop
  - Write pseudocode for
    - Isempty – to check whether stack is empty
    - Isfull – to check whether stack is full

Pseudocode

- **Comment: Assume that the array elements begin at 0**
- **Comment: Assume that the maximum elements of the stack is MAX**

```
Declare
   stack[MAX] : Integer;
Module Initialize
        top = 0;
Module End
Module Push
        if top >= MAX then
              return error stack is full
        else
         begin
              stack[top] = data;
               top = top + 1;
        end
Module End
```

Pseudocode

```
Module pop
 if stack_pointer <= 0 then
       return error Stack is empty
 else
  begin
              top = top - 1;
              data = stack[top];
 end
 Module End
```

Explanation

- **Using Linked List:**
  - Using Linked List is called as “dynamic implementation”.
  - Since it increases and shrinks dynamically, **isfull** function is not required.
  - You can implement the following functions, namely:
    - Initstack
    - isempty
    - Push
    - pop

Applications

- **Stack can be used for the following applications:**
  - To reverse data (e.g., strings, files)

    (Demo on Reverse.txt)
    - This is also useful in decimal to binary conversion because we always print result in reverse order.
  - To convert infix expression to postfix expression
  - To process function call (operating system uses stack for this task)

Definition

- A "queue" is an ordered collection of items from which items may be deleted from one end (called the "front" of the queue) and items may be inserted at the other end (called the "rear" of the queue).

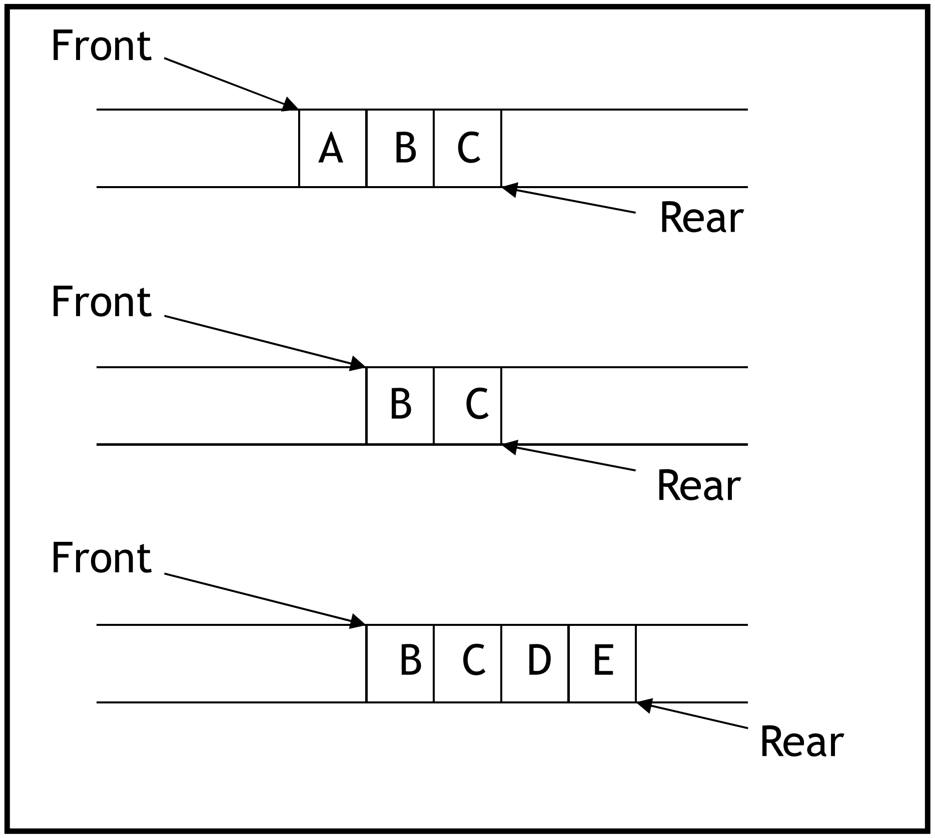
Explanation

- **A queue implements the concept of FIFO (First In First Out). It means the first inserted item will be the first item to be deleted.**
- **For example: Queues for ticket reservation, for paying bill**
  - The first person in the queue gets served first at the beginning of the queue called as front. If a new person joins the queue, he stands at the end of the queue called rear.

Explanation

- **Two basic operations performed on a Queue are:**
  - **Insert:** Insertion of item at the REAR of the queue.
  - **Delete:** Deletion of item from the FRONT of the queue.
- **Other operations on Queue are as follows:**
  - **Isfull:** checks queue is full (rear=maxsize-1)
  - **Isempty:** checks queue is empty (rear < front)
  - **Initqueue:** initializes the queue (front=0 and rear=-1)

Illustration



Before deletion of item 'A' in the Queue
(INITIAL QUEUE)

After deletion of item 'A' in the queue
(remove)

After insertion of items 'D' and 'E' in the queue
(Insert 'D')
(Insert 'E')

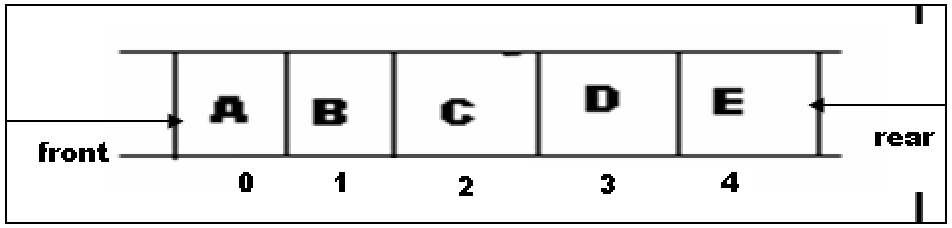
Explanation

- **Using arrays is a static implementation of the queue.**
- **To implement a queue we need the following functions to be implemented:**

– **Initqueue**	: It initializes the queue.
– **Isfullq**	: It returns TRUE if queue is full, and FALSE otherwise.
– **Isemptyq**	: It returns TRUE if queue is empty and FALSE otherwise.
– **Insert**	: It inserts element at rear.
– **Remove**	: It removes element from front of queue.

Explanation

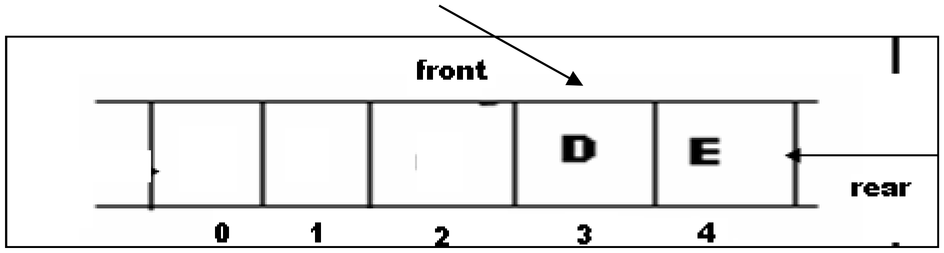
- Consider linear queue using arrays.
- Consider that queue is full (front=0 and rear=maxsize-1).



- Suppose we have removed some elements from the queue (front > 0).

Explanation

- If we remove three elements A, B, C from the queue, then front =3 and rear =4.



- In the above queue, we have space on the left hand side to store elements which we cannot reuse for inserting elements in linear queue.
- However, it can become possible by using circular queue.

Pseudocode

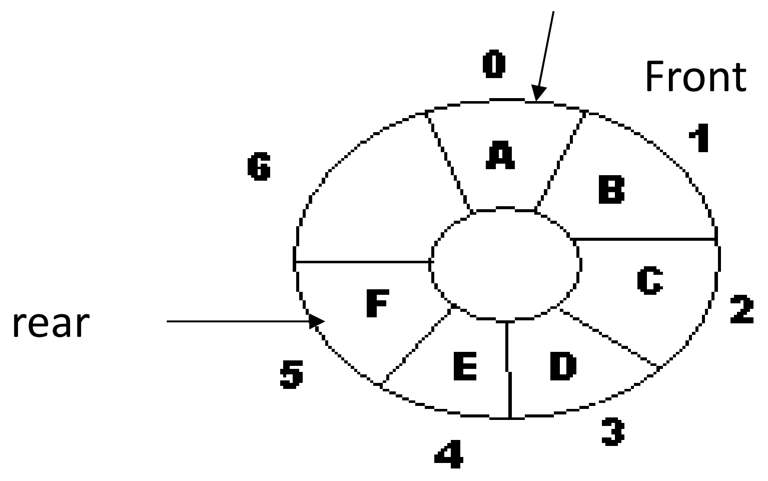
```
Comment : Assume MAX represents size of array
module initialize
       front = 0
       rear = 0
 end module initialize
 module insert
 If rear =  MAX
          return error queue is full and do not insert
     else
       begin
          queue[rear]=data
           rear =rear+1
 End if
 end module insert
```

Pseudocode

```
module remove
     if front=rear
         return error queue is empty and do not remove
     else
     begin
         data = queue[front]
         front =front +1
     end if
end module remove
```

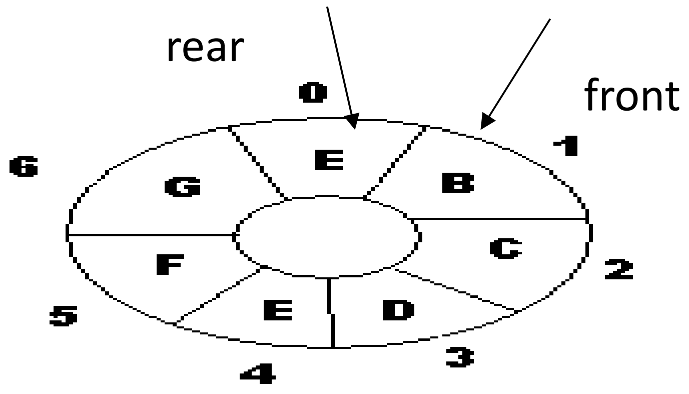
Explanation

- In the above example even though first few elements of array are removed from the queue and space is empty still we cannot add elements in the array.
- Solution for that is represent array as circular queue
- Initially front is 0 and rear =-1.
- As we insert element in the queue rear, it will be increased by 1.



Explanation

- Remove A from the queue, and insert G and E in the circular queue.



- In circular queue, if rear =maxsize-1, then next value of rear = 0, and then 1, 2, and so on.
- We can reuse the space.

Explanation

- **You can perform same operations on circular queue.**
  - Only the condition for checking whether queue is full or empty will change. (Note : After reading element from front end it will be marked as NULL)
- **Operations on queue are described below:**
  - **Isfull:** It is useful to check the queue overflow.
    - if front== (rear+1 mod maxsize) && queue[front]!=NULL then the queue is full
  - **Isempty:** It is used to check for underflow of the queue.
    - If rear==-1 || (rear+1 mod maxsize)==front && queue[front]!=NULL
  - **Initqueue:**
    - Initialize front=0 and rear =-1

Process Steps

- **Step 1: Suppose the "front" and "rear+1 mod maxsize" is pointing at same location, and the queue[front] holds a not NULL value. It means the queue is full; it is a "queue overflow" state. So quit; else go to step 2**
- **Step 2: If "rear" of the queue is pointing to the last position, then go to step 3 or else step 4.**
- **Step 3: Make the "rear" value as 0.**
- **Step 4: Increment the "rear" value by one.**
- **Step 5: Insert the new value in the queue at "rear".**

Process Steps

- Step 1: If the queue is empty, then display message "empty queue" and quit; else continue.
- Step 2: Remove the element at "front".
- Step 3: If the "front" is pointing to the last position of the queue, then perform step 4 else step 5.
- Step 4: Reinitialize "front" to 0 (i.e., pointing at first position) and quit.
- Step 5: Increment the "front" position by one.

IGATE
Speed.Agility.Imagination

Illustration

- **In circular queues, while increasing the value of rear and front, use mod operator as shown below:**
  - rear=(rear+1) mod maxsize
  - front=(fornt+1) mod maxsize
- **This is because mod operator always assigns the value between 0 and maxsize-1.**

Usage

- **Queues are used for the following:**
  - Resource scheduling, CPU scheduling by Opearting system
  - Organizing multiple jobs to a printer
  - Vehicles on toll-tax bridge: The vehicle that comes first to the toll tax booth leaves the booth first. The vehicle that comes last leaves last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
  - Luggage checking machine: Luggage checking machine checks the luggage first that comes first. Therefore, it follows FIFO principle of queue.
  - Patients waiting outside the doctor's clinic: The patient who comes first visits the doctor first, and the patient who comes last visits the doctor last. Therefore, it follows the first-in-first-out (FIFO) strategy of queue.

What is Hashing ?

- **Hashing is the transformation of a string into a small fixed-length value or key**
- **Searching the item using the shorter hashed key than to find it using the original value is faster.**
- **Uses of hashing**
  - Hashing is used in indexing and retrieve items from the database tables
  - Java uses hashcode for searching objects faster

Example

Suppose we want to store following data using hashing

Key	Value
Cuba	Havana
England	London
France	Paris
Spain	Madrid
Switzerland	Berne

What is Hashing?

- Decide hashing function

e.g Hashing function : length of the string

- Decide number of buckets in hash table (equal to 11- maximum length of string)
- Apply hash function on each value
- Create hash table using arrays

4.3 Hashing
Example

Position (hash code = key length)	Keys array	Values array
1		
2		
3		
4	Cuba	Havana
5	Spain	Madrid
6	France	Paris
7	England	London
8		
9		
10		
11	Switzerland	Berne

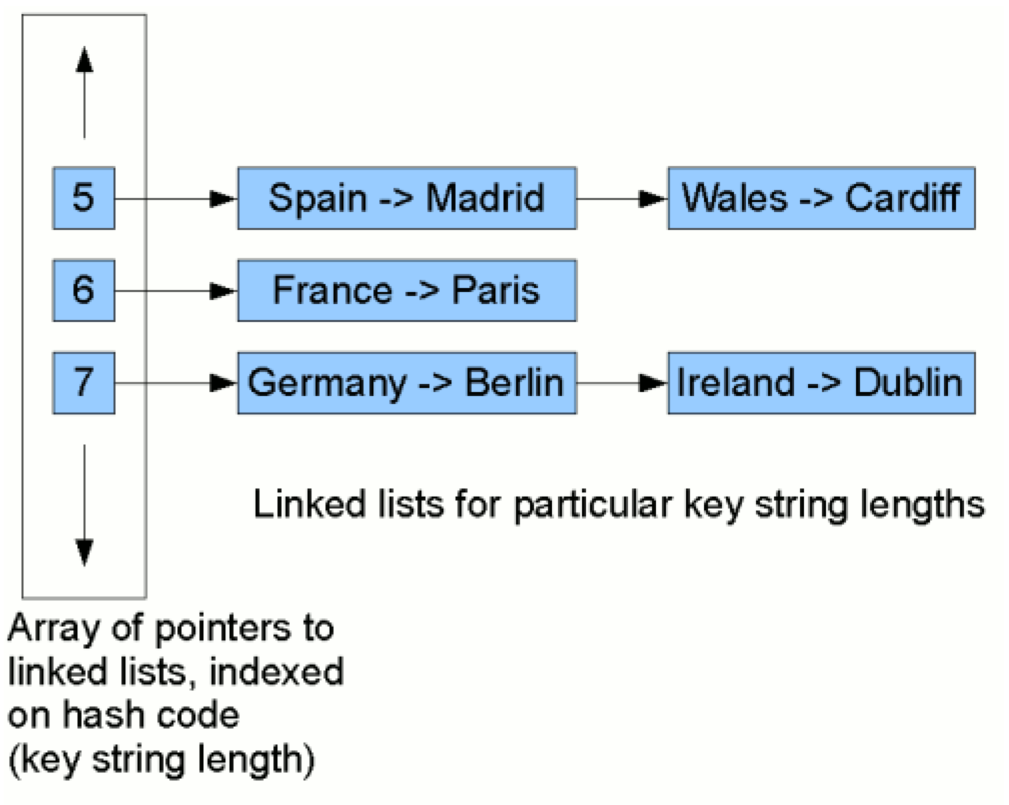
Problems in using arrays

- Array size needs to be big enough to accommodate the longest string. If the longest string is 700 characters then the number of buckets will be 700.
- **Problems**
  - **Waste of space**
    - **There may be certain places which do not contain data.**
    - **Certain places may contain large amount o**
  - **Collision**
    - **When we try to store 2 strings of same length collision will occur**

Solution

- **To avoid collisions**
  - use array of pinters to **linked lists**[2] rather than simply an array of keys/values. Each little list is generally called a **bucket**.
  - To solve the problem of using an array that is too large find better hash function
  - E.g hash code *modulo* array size[3].
  - E.g if the array size is 20 then length of string mod 20
  - This will distribute the keys within 20 buckets numbered 0-19
  - If length is 5 then (5 mod 20 =5) store the key value pair in bucket 5
  - If length is 23 then (23 mod 20=3) store the key value pair in bucket 5

Solution


5
6
7
Spain -> Madrid
Wales -> Cardiff
France -> Paris
Germany -> Berlin
Ireland -> Dublin
Linked lists for particular key string lengths
Array of pointers to linked lists, indexed on hash code (key string length)

Properties of hash function

- It can distribute the keys more or less *evenly* over the range of positions in the Hash Table
- We want to avoid the situation where position 9 has a list of 12,000 entries and position 15 has a list of only 21 entries.

What is Recursion

- **If a function calls itself is called as recursion**
- **E.g – Calculate factorial of a number**
- N! = N × (N-1) × (N-2) × … × 2 × 1
- **The recurssive program**

```
public static int factorial(int N)
{ if (N == 1) return 1;
return N * factorial(N-1); }
```

Recursion

- Example -Winding
  - factorial(5) - > 5 * factorial(4)
  - factorial(4) -> 4 * factorial(3)
  - factorial(3) -> 3* factorial(2)
  - factorial(2) -> 2 * factorial(1)
  - return 1
- Rewinding
  - return 2*1 = 2
  - return 3*2 = 6
  - return 4*6 = 24
  - return 5*24 = 120

Recursion

- Write a recursive program for finding GCD of 2 numbers
- If *p* > *q*, the gcd of *p* and *q* is the same as the gcd (*q*,*p* % *q*)
- return 2*1 = 2
- Winding
  - gcd(1440, 408)= return (gcd(408, 216))
  - (gcd(408, 216)) = return (gcd(216, 192))
  - gcd(216, 192) =return(gcd(192, 24))
  - (gcd(192, 24))=return (gcd(24, 0))
  - gcd(24,0)= return 24
- rewinding
  - return 24
  - return 24
  - return 24
  - return 24

Recursion

- Advantages
  - **Algorithm becomes easy**
- **Disadvantages**
  - **Program becomes slow**
  - **Uses more stack memory**

LAB

- Write a recursive function to reverse a string. Write a recursive function to reverse the words in a string, i.e., “I am recursive code" becomes “code recursive am I".
- Write a program to reverse a string using stack . Write a function to reverse the words in a string, i.e., “I am recursive code" becomes “code recursive am I".
- Compare time and space efficiency for both the programs.

Collections in Java

Collections Framework

- **A Collection is a group of objects.**
- **Collections framework provides a set of standard utility classes to manage collections.**
- **Collections Framework consists of three parts:**
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

Advantages of Collections

- **Collections provide the following advantages:**
  - Reduces programming effort
  - Increases performance
  - Provides interoperability between unrelated APIs
  - Reduces the effort required to learn APIs
  - Reduces the effort required to design and implement APIs
  - Fosters Software reuse

Concept of Interfaces and Implementation

Fig:1

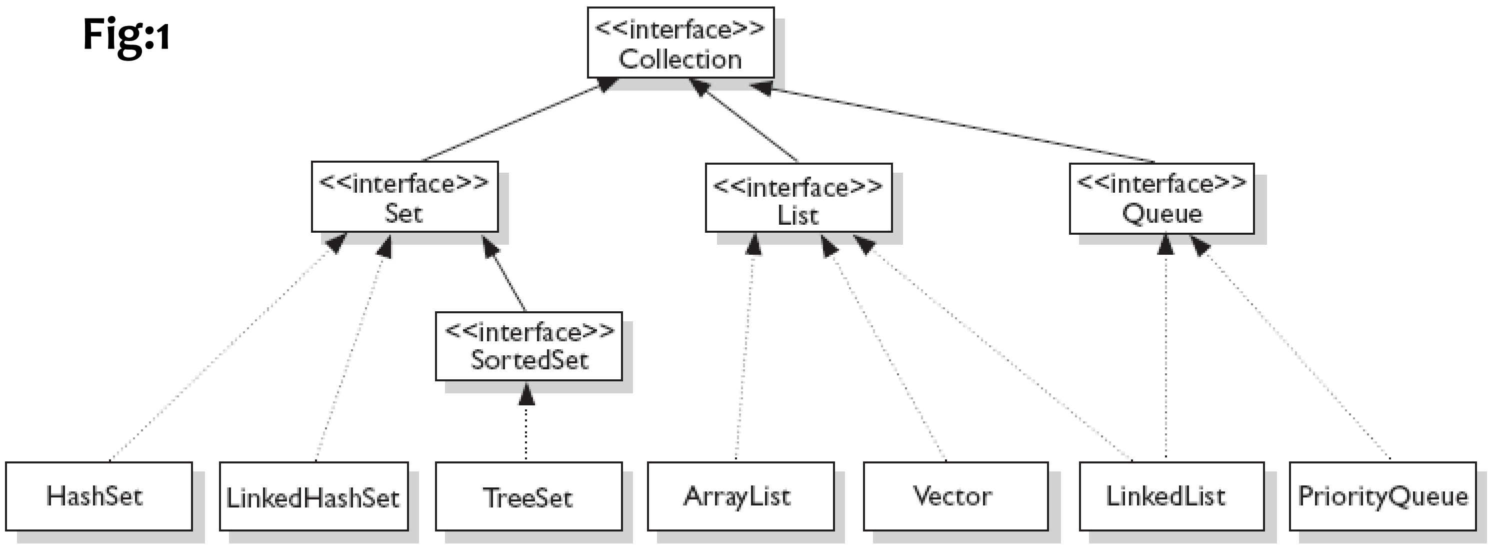


Fig:2

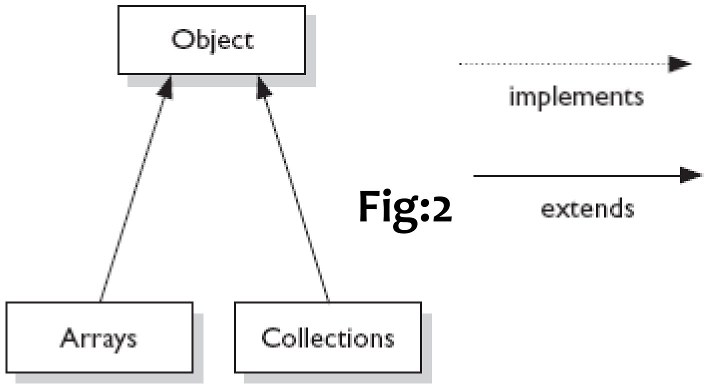
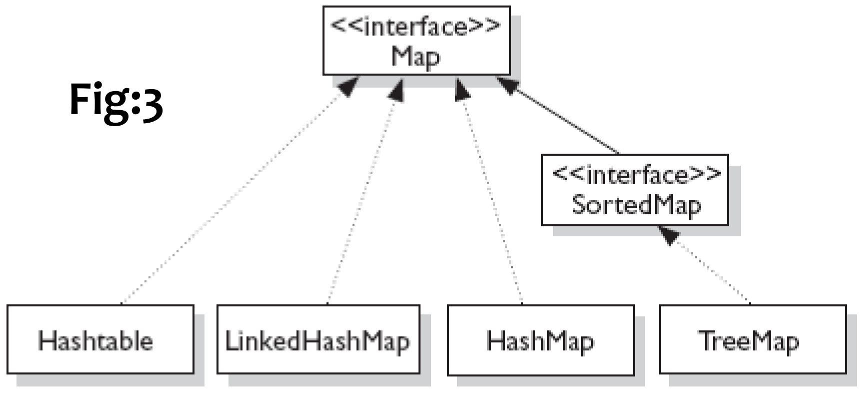


Fig:3



Collection Interfaces

➤ **Let us discuss some of the collection interfaces:**

Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

Collection Implementations

- **Collection Implementations:**

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection Implementations

- Notes Page

Collection Interface methods

Method	Description
`int size();`	Returns number of elements in collection.
`boolean isEmpty();`	Returns true if invoking collection is empty.
`boolean contains(Object element);`	Returns true if element is an element of invoking collection.
`boolean add(Object element);`	Adds element to invoking collection.
`boolean remove(Object element);`	Removes one instance of element from invoking collection
`Iterator iterator();`	Returns an iterator fro the invoking collection
`boolean containsAll(Collection c);`	Returns true if invoking collection contains all elements of *c*; false otherwise.
`boolean addAll(Collection c);`	Adds all elements of c to the invoking collection.
`boolean removeAll(Collection c);`	Removes all elements of c from the invoking collection
`boolean retainAll(Collection c);`	Removes all elements from the invoking collection except those in c.
`void clear();`	Removes all elements from the invoking collection
`Object[] toArray();`	Returns an array that contains all elements stored in the invoking collection
`Object[] toArray(Object a[]);`	Returns an array that contains only those collection elements whose type matches that of *a*.

AutoBoxing with Collections

- **Boxing conversion converts primitive values to objects of corresponding wrapper types.**

```
int intVal = 11;
Integer iReference = new Integer(i); // prior to Java 5, explicit Boxing
iReference = intVal;                 // In Java 5,Automatic Boxing
```

- **Unboxing conversion converts objects of wrapper types to values of corresponding primitive types.**

```
int intVal = iReference.intValue();   // prior to Java5, explicit unboxing
intVal = iReference;                  // In Java 5, Automatic Unboxing
```

Iterating through a collection

- **Iterator is an object that enables you to traverse through a collection.**
- **It can be used to remove elements from the collection selectively, if desired.**

```
public interface Iterator<E>
 {
 boolean hasNext();
 E next();
void remove();
 }
```

- **Iterable is an superinterface of Collection interface, allows to iterate the elements using foreach method**

```
Collection.forEach(Consumer<? super T> action)
```

Enhanced for loop

- **Iterating over collections looks cluttered:**

```
void printAll(Collection<Emp> employees) {
    for (Iterator<Emp> iterator = employees.iterator(); iterator.hasNext(); )
        System.out.println(iterator.next()); } }
```

- **Using enhanced** for loop**, we can do the same thing as:**

```
void printAll(Collection<Emp> employees) {
     for (Emp empObj : employees) )
        System.out.println( empObj ); }}
```

  - When you see the colon (:) read it as “in.”
  - The loop above reads as “for each emp ‘t’ in collection ‘e’.”

ArrayList Class

- **An ArrayList Class can grow dynamically.**
- **It provides more powerful insertion and search mechanisms than arrays.**
- **It gives faster Iteration and fast random access.**
- **It uses Ordered Collection (by index), but not Sorted.**

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

HashSet Class

- HashSet Class does not allow duplicates.
- A HashSet is an unsorted, unordered Set.
- It can be used when you want a collection with no duplicates and you do not care about the order when you iterate through it.

4.5 Collection Framework
TreeSet class

- **TreeSet does not allow duplicates.**
- **It iterates in sorted order.**
- **Sorted Collection:**
  - By default elements will be in ascending order.
- **Not synchronized:**
  - If more than one thread wants to access it at the same time, then it must be synchronized externally.

Comparator Interface

- **The** java.util.Comparator **interface can be used to sort the elements of an Array or a list in the required way.**
- **It gives you the capability to sort a given collection in any number of different ways.**
- **Methods defined in Comparator Interface are as follows:**
  - int compare(Object o1, Object o2)
    - It returns true if the iteration has more elements.
  - boolean equals(Object obj)
    - It checks whether an object equals the invoking comparator.

Comparable Interface

- **Java.util.Comparable interface imposes a total ordering on the objects of each class that implements it.**
- **This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.**

- **Methods defined in Comparable Interface are as follows:**
  - public int compareTo(Object o)
  - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparable Interface Example

```
class Emp implements Comparable {
   int empID;
   String empName;
   double empSal;
    public Emp(String ename, double sal) { …  }
    public String toString() {  …  }

     public int compareTo(Object o) {
        if (this.empSal == ((Emp) o).empSal)    return 0;
          else if (this.empSal > ((Emp) o).empSal)   return 1;
             else   return -1;
   }}
```

Comparable Interface Example (ctnd…)

```
class ComparableDemo {
    public static void main(String[] args) {
      TreeSet tset = new TreeSet();
      tset.add(new Emp("harry", 40000.00));
      tset.add(new Emp("Mary", 20000.00));
      tset.add(new Emp("Peter", 50000.00));

      Iterator iterator = tset.iterator();
      while (iterator.hasNext()) {
        Object empObj = iterator.next();
        System.out.println(empObj + "\n");
      }
}}
```

Output:

```
Ename : Mary    Sal : 20000.0
Ename : harry   Sal : 40000.0
Ename : Peter   Sal : 50000.0
```

HashMap Class

- HashMap uses the hashcode value of an object to determine how the object should be stored in the collection.
- Hashcode is used again to help locate the object in the collection.
- HashMap gives you an unsorted and unordered Map.
- It allows one null key and multiple null values in a collection.

Vector Class

- **The** java.util.Vector **class implements a growable array of Objects.**
- **It is same as ArrayList. However, Vector methods are synchronized for thread safety.**
- **New java.util.Vector is implemented from List Interface.**
- **Creation of a Vector:**
  - Vector v1 = new Vector(); // allows old or new methods
  - List v2 = new Vector(); // allows only the new (List) methods.

Hashtable Class

- **It is a part of** java.util package**.**
- **It implements a hashtable, which maps keys to values.**
  - Any non-null object can be used as a key or as a value.
  - The Objects used as keys must implement the **hashcode** and the **equals method.**
- **Synchronized class**

Best Practices

- **Let us discuss some of the best practices on Collections:**
  - Use for-each liberally.
  - Presize collection objects.
  - Note that Vector and HashTable is costly.
  - Note that LinkedList is the worst performer.

Best Practices

- Choose the right Collection.
- Note that adding objects at the beginning or in between of the collections is considerably slower than adding at the end, if collection class is internally using arrays.
- **Encapsulate collections.**
- Use thread safe collections only if your application is multithreaded.

Best Practices

- Notes page

Summary

- **In this lesson, you have learnt that:**
  - In a stack, data can be inserted or deleted from the top.
  - It implements LIFO.
  - Different operations performed on stack are as follows:
    - Initstack
    - isfull
    - Isempty
    - Push
    - pull

Summary

- In a Queue, data can be inserted from rear and deleted from front.
- It implements FIFO.
- Different operations performed on stack are as follows:
  - Initqueue
  - isfullq
  - Isemptyq
  - insert
  - remove

Summary

- The various Collection classes and Interfaces
- Generics
- Best practices in Collections

Review – Questions

- **Question 1: Which of the following condition indicates that stack is empty.**
  - **Option 1:** Top==-1
  - **Option 2:** Top=-1
  - **Option 3:** Top==NULL
- **Question 2: To check stack overflow we need to use initstack function.**
  - True / False

Review – Questions

- **Question 3: In pop function, to avoid underflow error we use ___ function.**
- **Question 4: Queue implements LIFO.**
  - True / False
- **Question 5: In a circular queue, ___ condition indicates that the queue is full.**

Review – Questions

- Consider a hash table of size seven, with starting index zero, and a hash function (3x + 4)mod7. Assuming the hash table is initially empty. Contents of the table when the sequence 1, 5,8, 10,15,7 is inserted into the table using hashing? How the numbers will be arranged.
- Given the following input (4322, 1334, 1571, 9636, 1986, 6271, 6173, 4196) and the hash function x mod 10, which of the following statements are true?

A. 9636, 1986, 4196 hash to the same value

B. 1571, 6271 has to the same value

C. All elements hash to the same value

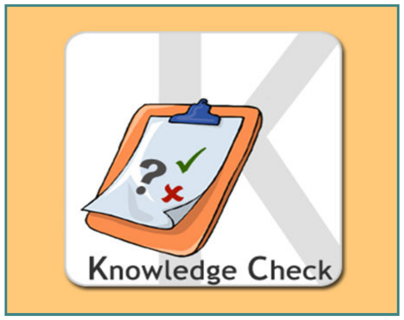
D. Each element hashes to a different value

Review Questions

- **Question 1:** Consider the following code:

```
TreeSet map = new TreeSet();
  map.add("one");
  map.add("two");
  map.add("three");
  map.add("one");
  map.add("four");
  Iterator it = map.iterator();
  while (it.hasNext() )
     System.out.print( it.next() + " " );
```

  - **Option 1:** Compilation fails
  - **Option 2:** four three two one
  - **Option 3:** one two three four
  - **Option 4:** four one three two



IGATE
Speed.Agility.Imagination

Review Questions

- **Question 2:** Which of the following statements are true for the given code?

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
   while (it.hasNext())
       System.out.print(it.next() + " ");
}
```

  - **Option 1:** The before() method will print 1 2
  - **Option 2:** The before() method will print 1 2 3
  - **Option 3:** The before() method will not compile.
  - **Option 4:** The before() method will throw an exception at runtime.