Implementing Stacks

|  |  |
| --- | --- |
| //this is the java.util implementation    public class myArrayListStack<E>  {  private ArrayList<E> a;  public myArrayListStack()   {   a = new ArrayList<E>();  }  public E push(E obj) //O( 1 )  {   a.add(obj);  }  public E pop() //O( 1 )  {   return a.remove(a.size()-1);  }  public E peek() //O( 1 )  {   return a.get(a.size()-1);  }  public boolean isEmpty() //O( 1 )  {    **return a.size()==0;**  }  } | public class myRawArrayStack  {  private Object[] a = new Object [10];  private int mySize = 0;  public myRawArrayStack()   { }  public Object push(Object obj) //O( 1)  {   if(mySize==a.length){  Object[]b = new Object[2\*mySize];  for(int i=0;i<mySize;i++)  b[i]=a[i];  a=b;  }  a[++mySize]=obj;  **return obj**  }  public Object pop() //O( 1 )  {  **Object o = a[mySize-1];**  a[--mySize]=null;  return o;  }  public Object peek() //O( 1 )  {    **return a[mySize-1];**  }  public boolean isEmpty() //O( **1**  )  {  return mySize==0;  }  } |
| public class myLinkedListStack<E>   {  private LinkedList<E> ll;  public myLinkedListStack<E>()   {   ll **= new Queue<E>();**  }  public E push(E obj) //O( 1 )  {    **ll.addFirst(obj);**  return obj;  }  public E pop() //O( 1 )  {    **return ll.removeFirst();**  }  public E peek () //O( 1 )  {    **return ll.getFirst();**  }  public boolean isEmpty() //O( 1 )  {    **return ll.isEmpty();**  }  } | public class myListNodeStack<E>  {  private ListNode head;  public myListNodeStack<E>()   { }  public E push(E obj) //O( 1 )  {    **head = new ListNode(obj, head);**  **return obj;**  }  public E pop() //O( 1 )  {  ListNode t = head;  head = head.getNext();  return (E)t.getValue();  }  public E peek() //O( 1 )  {  **return (E)head.getValue();**  }  public boolean isEmpty() //O( 1 )  {    **return head.getNext()==null;**  }  } |
|  |  |

Implementing Queues

|  |  |
| --- | --- |
| public class myArrayListQueue<E>   {  private ArrayList<E> a;  public myArrayListQueue<E>()   {  **a = new ArrayList<E>();**  }  public boolean add(E obj) //O( 1 )  {    **a.set(a.size(),obj);**  return obj;  }  public E remove() //O(1 )  {    **return a.remove(0);**  }  public E peek () //O( 1 )  {  **return a.get(0);**  }  public boolean isEmpty() //O( 1 )  {    **return a.size()==0;**  }  } | public class myRawArrayQueue<E>   {  private Object[] a = new Object[10];  private int mySize = 0; public myRawArrayQueue()   { }  public boolean add(Object obj)//O( n)  {  **//resizing code, if needed**  **//shift every element up by 1**  **mySize++;**  **a[0]=obj;**  **return true;**  }  public Object remove() //O( 1 )  {    **Object o =a[--mySize];**  **a[mySize] = null;**  **return o;**  }  public Object peek() //O( 1 )  {    **return a[mySize];**  }  public boolean isEmpty() //O( 1 )  {    **return mySize==0;**  } |
| //this is the java.util implementation public class myLinkedListQueue<E>   {  private LinkedList<E> ll;  public myLinkedListQueue<E>()   {  **ll = new LinkedList<E>();**  }  public boolean add(E obj) //O( 1 )  {    **ll.add(obj)**  }  public E remove() //O( 1 )  {    **return ll.remove();**  }  public E peek() //O( 1 )  {   return ll.peek();  }  public boolean isEmpty() //O( 1 )  {    **return ll.isEmpty();**  }  } | public class myListNodeQueue<E>  {  private ListNode head, tail;  public myListNodeQueue<E>()   { }  public boolean add(E obj) //O()  {  **if (head == null){**  **head = new ListNode(obj,null);**  **tail = head;**  **}  else{  tail.setNext(new ListNode(obj),null);  tail = tail.getNext();   }  return true;  }**  public E remove() //O(  )  {   **E o = head.getValue();**  **head = head.getNext();**  **return o;**  public E peek () //O(  )  {  **return (E) head.getValue();**  }  public boolean isEmpty() //O(  )  {  **return head == null;**  **}** |
|  |  |

Implementing Priority Queues

as ArrayLists

|  |  |
| --- | --- |
| public class ArrayListPriorityQueue<E   extends Comparable<E>>  {  //elements in random order  private ArrayList<E> a; public ArrayListPriorityQueue<E>()   {    **a = new ArrayList<E>();**  }  public boolean add(E obj) //O(  )  {    **a.add(obj);**  }  public E remove() //O( n )  {   int minIndex=0;  for (int i=1;i<a.size();i++){  if (a.get(i).compareTo(  a.get(minIndex)<0)  minIndex = i;  }  return a.remove(minIndex);  }  public E peek() //O( n )  {  int minIndex=0;  for (int i=1;i<a.size();i++){  if (a.get(i).compareTo(  a.get(minIndex)<0)  minIndex = i;  }  return a.get(minIndex)  }  public boolean isEmpty() //O( )  {  return a.size()==0;  }  } | public class ArrayListPriorityQueue<E   extends Comparable<E>>  {  //elements sorted, Highest Priority at // the front private ArrayList<E> a; public ArrayListPriorityQueue<E>()   {   a = new ArrayList<E>();  }  public boolean add (E obj) //O( **n**  )  {  **for(int i =0; i<a.size();i++){**  **if(obj.compareTo(a.get(i))<0){  a.add(i,a.get(i));**  **return true;  }**  **}   a.add(obj);  return true;**  }  public E remove() //O( n )  {    **return a.remove(0);**  }  public E peek() //O(  )  {    **return a.get(0);**  }  public boolean isEmpty() //O( )  {    **return a.size()==0;**  }  } |

Implementing Priority Queues

as LinkedLists

|  |  |
| --- | --- |
| public class LLPriorityQueue<E   extends Comparable<E>>  {  //elements in random order private LinkedList<E> ll;  public LLPriorityQueue<E>()   {  **ll = new LinkedList<E>();**  }  public boolean add(E obj) //O(  )  {   return **ll.add(obj);**  }  public E remove() //O( n2 )  {   int minIndex=0;  for (int i=1;i<ll.size();i++){  if (ll.get(i).compareTo(  ll.get(minIndex)<0)  minIndex = i;  }  return ll.remove(minIndex);  }  public E peek() //O( n2 )  {  int minIndex=0;  for (int i=1;i<ll.size();i++){  if (ll.get(i).compareTo(  ll.get(minIndex)<0)  minIndex = i;  }  return ll.get(minIndex);  }  public boolean isEmpty() //O( )  {  return ll.size()=0;  }  } | public class LLPriorityQueue<E   extends Comparable<E>>  {  //elements sorted, Highest Priority at // the front  private LinkedList<E> ll; public LLPriorityQueue<E>()   {   ll = new LinkedList<E>();  }    public boolean add (E obj) //O( )  {  **for(int i =0; i<ll.size();i++){**  **if(obj.compareTo(ll.get(i))<0){  ll.add(i,obj);**  **return true;  }**  **}   ll.add(obj);  return true;**  }  public E remove() //O(  )  {    **return ll.removeFirst();**  }  public E peek() //O(  )  {    **return ll.getLast();**  }  public boolean isEmpty() //O( )  {    **return ll.size()==0;**  }  } |

N.B.: java.util.PriorityQueue is actually implemented by a *heap*. The heap's access time is O(log n).