III. Data Structures

A. Motivation:

1. Recall, most computers are used for:

a) Storage of data

b) Retrieval of data

c) Processing of data (algorithms)

2. We have discussed some "processing of data" in terms of searching and sorting but we must address some issues that determine how we should organize and implement data into various "structures". Specifically, we recognize the two major issues of speed verus storage in different situations:

a) Adding an element to the data structure --- in front/beginning or at back/end or in middle

b) Removing an element from the data structure--- in front/beginning or at back/end or in middle

c) Locating an element in the data structure --- sorted versus unsorted

d) Dealing with duplicate data

B. Standard Structures & Java Collections Framework (interfaces in java.util.*)

1. Overview

A) Standard Data Structures In Computer Science

1) arrays and matrices (linear or "bilinear" data structures)
2) linked lists (usually linear)
    a. stacks
    b. queues
    c. double
    d. circular
3) trees (nonlinear data structure)
    a. generic
    b. binary
    c. decision & searching
4) heaps
    a. combination of arrays and trees to take advantage of benefits of both structures
    b. *aka* PriorityQueues
5) maps
    a. *aka* tables, dictionaries
    b. generalizaton of an array using a (key,value) pair rather than (index,value) pair

B) General Concepts and Useful Details for Data Structures and Java Collections

1) hashing – the process of converting data into a key or index to locate the data in a structure

a. example: To store the names of students in a String array, convert the characters into integers and add the resulting values into one integer. This "total integer" could be used as an index to store and retrieve each name.

b. public int hashCode(String name)

```
{       int total = 0;
        for (int i = 0; i < name.length; i++)
                total += convertToInt(name.getAt(i));  // this converts a character into a value
        return total;                                   // where A=1, B=2, C=3,....
}

public store(String name)
{       String [] nameArray = new String[100];      // 100 slots to hold names
        String nameTest = "Bob";
        int index = hashCode(nameTest);             // index is now "Bob" as an integer = 2+15+2 =19
        nameArray[index] = nameTest                 // Bob is now stored in slot 19
}
```

c. pros – data itself is used to "locate" and "retrieve" itself

d. cons – can have collisions if data is "similar"... what would happen if nameTest was "BBo"?

e. Java uses a hashCode() method to store objects in memory... could a "collision" occur for "similar" objects?...Yes but extremely unlikely since the hashCode() method for Objects is very sophisticated.

2) iteration – used instead of an integer index when "moving" or "traversing" each element of a data structure...

a. ***for arrays we use:***

```
for (int i = 0; i < myArray.length; i++)
      Object data = myArray[i];
```

b. ***for collections we can use***:

```
Iterator i = anyDataStructure.iterator(); LOOK AT    C:/j2sdk1.4/src
while (i.hasNext())                       //see if there is more data
      Object data = i.next();             //get the element
```

c. or using for-each looping with Generics

```
for (String  o: anyDataStructure)         //automatically loops through
      System.out.println(o);              //data and gets String
```

d. searching for an item in a data structure using iterators:

```
public boolean search(Collection c, Object o)
{
      Iterator i = c.iterator();    // Collections have method that returns an iterator on it
      while (!found) && (i.hasNext())     // while not found and there is a next element
            if (o.equals(i.next()))       // check if object is equal to this element
                  return true;            // Note: i.next() returns current object and moves
      return false;
}                                         //       iterator to next element in Collection.
```

i. Iterator Interface –

- boolean hasNext() // returns true if there is another element, false otherwise
- Object next() // returns the next element and moves iterator to that element
- void remove() // deletes current element...most difficult to define.

ii. ListIterator class // same as Iterator but is bidirectional

iii. Notes
1. Iterators by their very nature linearizes any data structure it traverses.
2. May have more than 1 iterator on a data structure but only ONE my be used to modify

e. searching for an item using for-each and Generics

```
public boolean search(Collection<String> c, String o)
{
      for (String n : c)
            if (n.equals(o)) return true;
      return false;
}
```

3) AutoBoxing Lecture (Java5—jre1.5)

Why:

It is a hassle to convert or cast between primitive types and their wrapper classes. This is especially true when using JFC collections or data structures (ArrayLists, Maps, etc.) The compiler will do this casting for us. It is especially useful when combined with Generics.

Example using the <u>pre-Java5</u> way:

```
//instantiate an array of "number" Objects
ArrayList numbers = new ArrayList();

Integer   xInt    = new Integer(13);  //instantiate an Integer object;
numbers.add(xInt);                    //add the Integer to the ArrayList
...
xInt  = (Integers)numbers.get(0);     //get Object then cast it back to Integer
int z =  y.intValue();                //get the value (13) from the Integer
```

Example using the Java5 <u>AutoBoxing way:</u>

```
//instantiate an array of Integers using generics
ArrayList<Integer> numbers = new ArrayList<Integer>();    //declaration using Generics

int x = 13;                           //create a primitive variable;
numbers.add(x);                       //this "autoboxes" the int to an Integer then
...                                   //    adds the Integer object to the array
int z = numbers.get(0);               //gets an Object from array slot 0 and then
                                      //    the JRE "unboxes" the Integer to an int
```

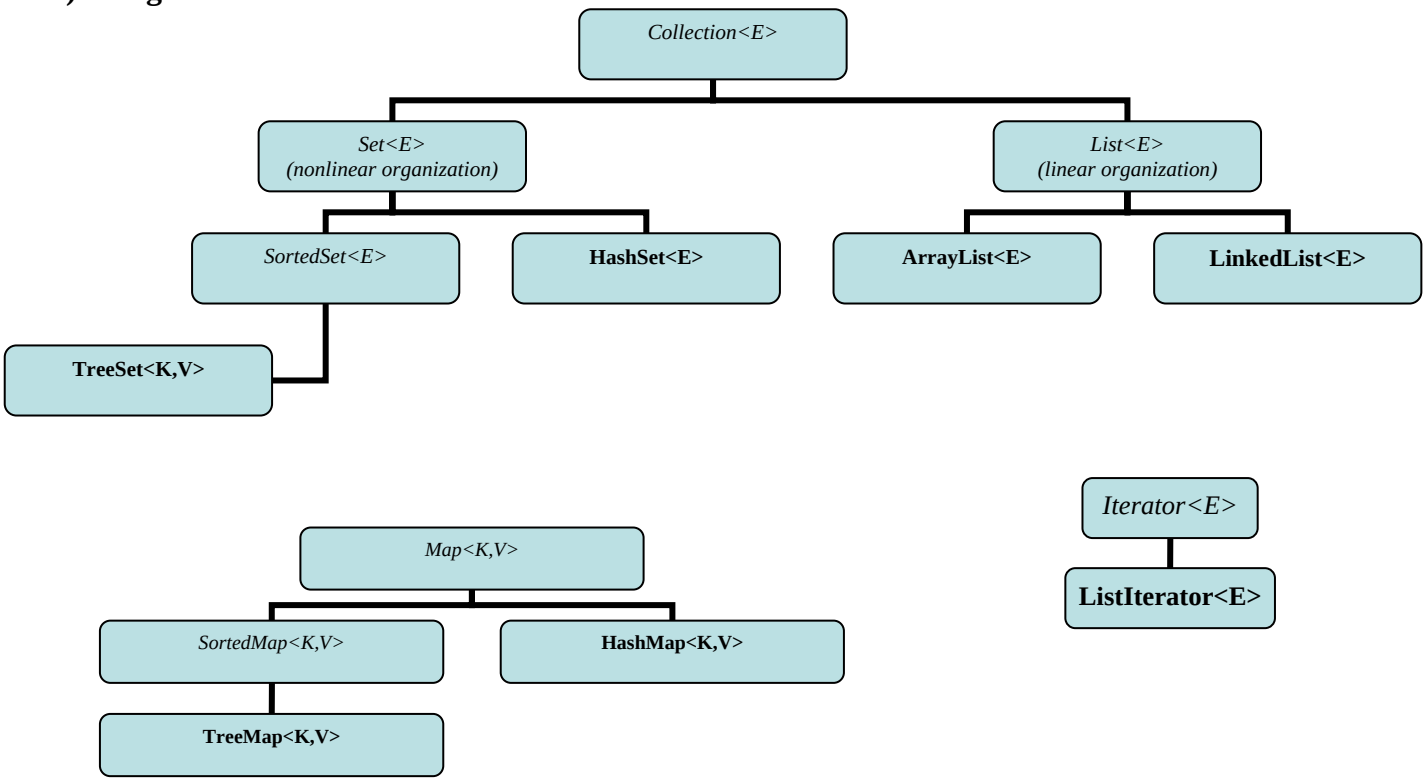
Notes:

1. Use Autoboxing/Unboxing *only* when there is an "impedance mismatch" between reference types and primitives. Usually this happens when you have to put numerical values into a collection.
2. It is *not* appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code since the autoboxing/unboxing requires a fair amount of computational complexity on the JRE.
3. An `Integer` is *not* a substitute for an `int`; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.
4. If the Integer returns "null" then the JRE throws a "`NullPointerException`" error

The == operator will automatically detect & use comparisons between two reference types (Objects) and will return true only if the two references are the same object. However, the JRE will use == operator with "Unboxing" when comparing primitive types and their corresponding wrapper classes.

C) Java API Collections Implementation of the above Data Structures

1) Diagrams

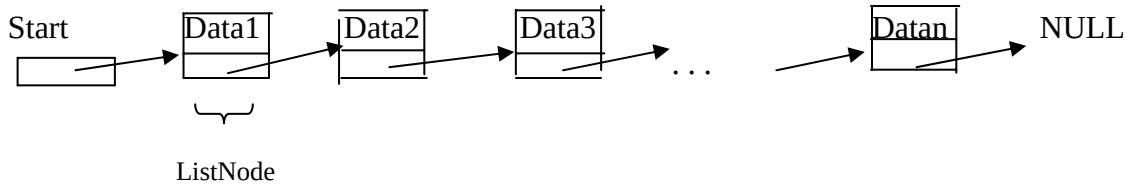


2) Definitions

Linked Lists

a) An overview - Remember the idea is to NOT waste memory . But large programs invariably need many variable names, a problem previously handled by arrays, i.e. many memlocs accessed by one name . After all can one realistically keep track of only so many names at a time . The answer is of course to come up with the a Linked List of pointers/references with only one name .

The picture -



b) Pointers/references require 2 memlocs per piece of data, so we tie these together into one structure which we will refer to as a ListNode. Thus the definition would be -

```
public class ListNode                  // Java API uses an inner class called "Entry" that is a
{                                      // double linked list.  This is not easy
      private Object   value;
      private ListNode next;           // self reference!!!!  "begging for recursion"

      public ListNode (Object initValue, ListNode initNext)
                                                  {value = initValue; next = initNext;}

      public Object     getValue()                {return value;}
      public ListNode   getNext()                 {return next;}
      public void       setValue (Object newValue)    {value = newValue;}
      public void       setNext (ListNode newNext)    {next  = newNext;}
}
```

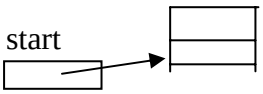
c) Remember at this stage there is no ListNode object, much less a Linked List,
so, we meed another class—called LinkedList—that will create ListNodes as needed.

```
public class LinkedList
{
```

ListNode start = new ListNode() ; **// private or public??? initialized here or constuctor??**

Allocates one Node and
places the address/reference of this structure in Start .

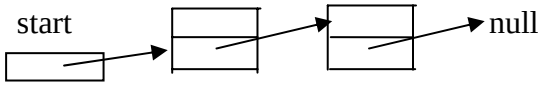
Thus the picture is now -



where *start* is the reference (the address of the ListNode) and
start.getValue() is the object at that address , a structure, so
start.getNext() is the reference to the 2nd ListNode (or null if it doesn't exist)
start.getValue().getName() is the is accessing the name field of the object in the ListNode

d) To use any of these Linked Lists we clearly need methods to -

i) push() a new ListNode with the appropriate Object onto the List
ii) pop() a ListNode from the List and retrive the object in the ListNode .



And before either of these we need the following method-

iii) To test for an empty list use the method-

```
boolean   isEmpty()
{
    boolean emptyList = false;
    if ( start = null)   emptyList = true ;
    return emptyList ;
}
```

iv) To add a ListNode onto the front of the LinkedList

```
public void   push(Object o)
{
      // create a new ListNode with new object and pointing to front of List
      ListNode temp = new ListNode(o,start);

      // point the start of the List to the new ListNode
      start = temp;
}
```

Question: what happens to temp????? Ans: local variables are out of scope

v) To remove a ListNode from the front of the Linked List

```
public Object pop()
{
      // make sure we aren't popping from an empty list
      if (isEmpty) return null;

      // create a temporary reference to the Object in front of List
      Object temp = start.getValue();

      // point start to 2nd ListNode in the List
      start = start.getNext();
      return temp;
}
```

Question: what happens to the ListNode at the front???? Ans: garbage collected

```
public class LinkedList             // technically, this is an implementation of a Stack
{
      private ListNode start;

      public LinkedList()
      {     start = null; }

      boolean   isEmpty()
      {
          boolean emptyList = false;
          if ( start = null)   emptyList = true ;
          return emptyList ;
      }

      public void   push(Object o)
      {
            ListNode temp = new ListNode(o,start);
            start = temp;
      }

      public Object pop()
      {
            if (isEmpty) return null;
            Object temp = start.getValue();
            start = start.getNext();
            return temp;
      }

      // other methods needed
}
```

e) Java API and Methods to be tested on the AP Exam

List Interface- sequence of elements, non-unique (not necessarily sorted*)*

List	ArrayList	LinkedList
`boolean add(Object)`	`void add(int index, Object)`	`void addFirst(Object)`
`int size()`	`void remove(int index)`	`void addLast(Object)`
`Object get(int index)`	`...`	`Object removeFirst()`
`Object set(int index, Object)`	*`(along with all`*	`Object removeLast()`
`Iterator iterator()`	*`List methods`*`)`	`void itr.add()`
`ListIterator listIterator()`		`void itr.remove()`

(1) Notes for ArrayList and LinkedList Classes:

(a) Most-used generic data structure. Good for "first approach" to a problem because it offers a balance between generalization and optimization.

(b) Easy manipulation. ArrayList is a combination of arrays (integer index) and Lists (Iterators).

(c) Pros - "organized" for faster insertion and deletion than Collections; O(n) = 1 once an element has been found as compared with arrays .

(d) Cons - not always the fastest organizations, non-sorted searching is linear, O(n) = n using Iterator or index unless List is Ordered in some way.

(e) Use ArrayList unless using frequent addition of elements to front--O(1) vs O(n)—or to “ends” (i.e. queue, stack)

(2) ArrayList

(a) used to store "ranked" data since--like arrays--uses integer indexes

(b) has the benefits of Lists (i.e. quick element addition and deletion)--hence better than arrays

f) There a 2 general categories of Linked Lists :

i) Simple (aka “Linear” or “Singly”) L.L.s which a the kind pictured above -

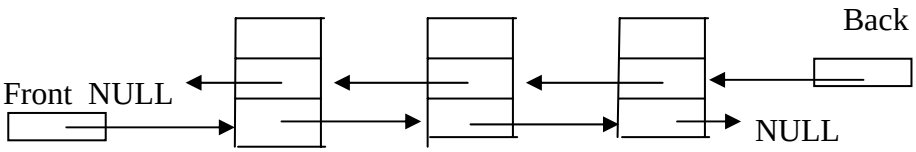
a) A Stack - where components are PUSHed onto and POPed from the Front of the list .
This is a Last On First Out list (LIFO List) .
Like a Dixie Cup dispenser .

b) A Queue - where components are PUSHed onto one end but POPed from the other end .
A FIFO List .
Like a waiting line at a theater .
Requiring a pointer to both the Front and the Back of the list .

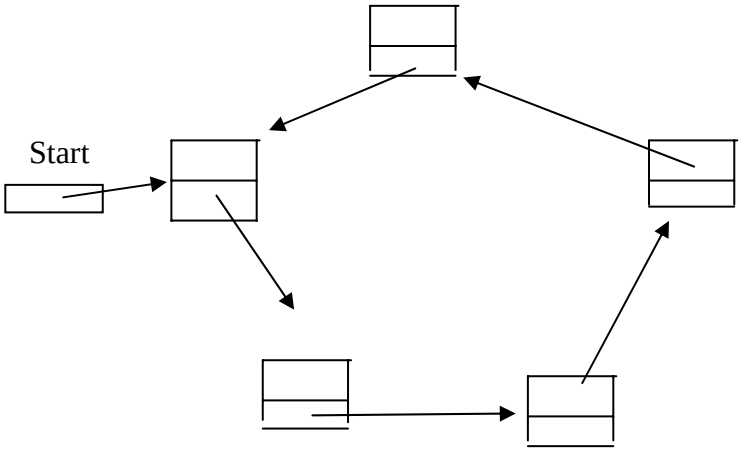
c) An Ordered List.
Here the Data is in some given order, such as ascending by Idnum or Alpha by name
Thus may be PUSHed onto and POPed from anywhere in the list .

ii) Complex L.L.s because the problem with Singly Linked Lists is that they may be traversed in only one direction since the arrows only point one direction . This is a major problem for large lists since when retrieving data one must start at the Front, so -

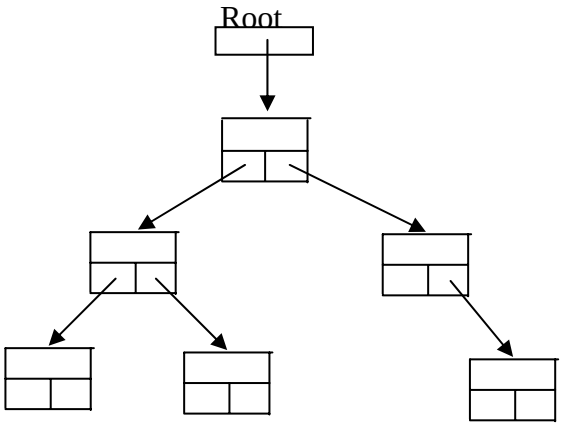
a) Double Linked - where one can transverse the list in both directions .



b) Circular List - while one cannot traverse backward, one can always continue onward from ones present location, without the necessity of starting over -



c) Trees - multiply Linked Lists
i.e. a list whose nodes (components) may point to more than one other node.



This is called a binary tree since each node can point to 2 others.
The connotation of a tree is of course when viewed in a flipped position.

Special Types of Linear (Simple) Linked Lists

b) Stacks - *Last In, First Out (LIFO) Linked List*

(1) Diagrams

(2) Applications

(a) modelling "plates" in a restaurant

(b) postfix math notation used in HP calculators...makes it easy to do math operations on calculator and is actually how our brain "evaluates" arithmetic 3*(4+5) in infix notation is 3 4 5 + * in postfix (also called reverse polish notation (RPN))

(c) recursion functions work on a "stack" basis

(3) methods - besides being a Collection & List

```
(a) boolean empty()    - should have been isEmpty() but original design problem

(b) Object push(Object o) - adds an element to end of stack

(c) Object pop()       -  takes element off the top of the stack

(d) Object peek()      - gives top element and leaves element on the top of stack

(e) int search()       -  returns position of element on stack... -1 if not
                          found...top position = 1!!!!
```

c) **Queues -** *First In, First Out (FIFO) -- no Java interface or class...must build one from Collections*

(1) Diagrams

(2) Applications

(a) Simulation of people in lines at a movie theatre.

(b) Simulation of demands by clients for too few services (Internet), printers, etc.)

(3) methods needed - since NO Java Implementation we must create following interface

```
public inteface Queue extends Collection
{
      boolean isEmpty();
      Object dequeue()        // remove element from front of queue
      Object enqueue()        // add an element to end of queue
      Object getBack()        // return the element on the back of the queue
      Object getFront()       // return the element on the front of the queue
}

// to follow Java Collections Framework, should implement Queue interface as an Abstract class
public AbstractQueue extends AbstractCollection implements Queue
{
      public abstract Object dequeue();
      public abstract Object enqueue(Object o);
      public abstract Object getBack();
      public abstract Object getFront();

      // other abstract methods to be declared are:
      public abstract Iterator iterator();
      public abstract int size();

      // other concrete methods to be defined are:
      public boolean equals(Object o) {...}
      public int hashCode() {...}

      /* other concrete methods inherited from Object and AbstractCollections are
      /* boolean addAll(Collection c);
      /* boolean contains (Object c)
      /* void clear()
      /* boolean isEmpty()
      /* boolean remove(Object o)
      /* String toString()          ... and others
      */
```

Binary Trees – the Java API diagrams given before are examples of trees

(a Linked List with nodes (components) which point/refer to multiple nodes)

a) Dfn : A Tree is a LinkedList with 2 properties -

i) Any node may point to 1 or more other nodes.

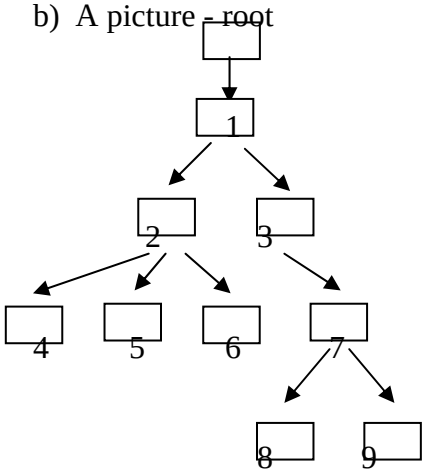
ii) One node pointed to by "root" has the properties

a) It is unique.

b) No node points to it.

c) All nodes may be reached from it by only one path. (no circling back up->graphs!)

b) A picture - root



A Tree called, "root", with 9 nodes.

c) Terminology -

i) Nodes 2 and 3 are Children of node 1 .

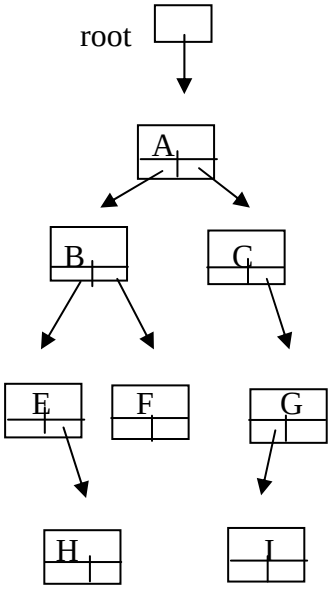
ii) Node 2 is the Parent of nodes 4, 5 and 6 .

iii) Nodes 4, 5, 6, 8 and 9 are Leaves (i.e. have no children) .

iv) The Level or Order of a node is the # of (super) parents + 1 .
Ergo node 5 is of order 3 .

d) Due to the complexity of implementation trees are invariable **binary** (i.e. 2 children possible.)

The picture - root



iii) A node and all of its (sub) children are called a subtree.
Which leads to an alternate definition for a tree -

Dfn : A Binary Tree is a set of connected nodes each pointing
0, 1, or 2 subtrees.

iv) In any "Ordered Binary Tree" the data is always stored so that for any node

either a) *leftchild.getData()* < *parent.getData()* < *rightchildgetData()*

Note : this implies there are **no** duplications of data— a TreeSet not a TreeMap

or b) *leftchild.getData()* <= *parent.getData()* <= *rightchildgetData()*

Note : this implies there **may** be duplications of data— a TreeMap not a TreeSet

v) In any traverse, the Leftchild is visited before the Rightchild (except 4) below).
This leads to 4+1 methods of Traversing the Data (each name comes from where "Parent" is visited) -

1) Preorder PLR
2) Inorder LPR
3) Postorder LRP
4) ReverseInOrder RPL

+

1) LevelOrder --- where each "level" is from top to bottom and from left to right

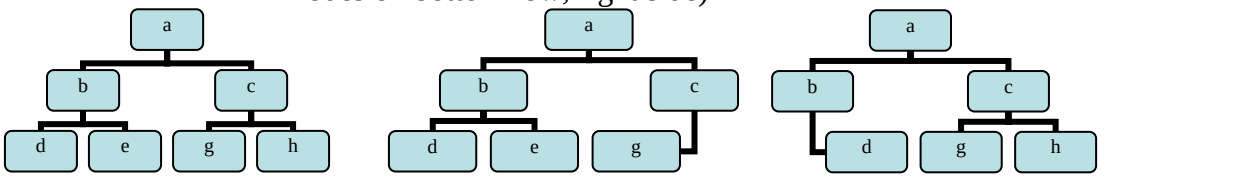
And the recursive function to implement the Inorder transverse for the class below -

```
public void     traverse_Inorder     (TreeNode root)
      {
            if ( !Empty() )
            {
                  traverse_Inorder (root.getLeft ());

                  // use the Parent's Data (i.e. root.getData())

                  traverse_Inorder (root.getRight ());
            }  // End If
      }  // End Transverse
```

Note: the first 4 traversals can be "easily" implemented using recursion. The "LevelOrder" is usually implemented by loops/iterations.

vi) Various Structures of Binary Trees

a. Full Binary Tree – all the tree's leaves are at the same level and every "interior" node has two children ("best possible" - most efficient use of space & time)
b. Complete Binary Tree – full tree or "almost full" (i.e. maybe missing some nodes on bottom row, right side)



Full (ideal) Complete (ok) neither (poor implementation)

e) Basic implementation of Tree

```
public class TreeNode               // Java API uses an inner class called “Entry” that is a
{                                   // is a Map (TreeMap)
      private Object value;
      private TreeNode left, right;

      public TreeNode (Object initValue, TreeNode initLeft, TreeNode initRight)
                                    { value = initValue; left = initLeft; right=initRight;}

      public Object     getValue()                    { return value;}
      public TreeNode   getLeft()                     {return left;}
      public TreeNode   getRight()                    {return right;}
      public void       setValue (Object newValue)    {value = newValue;}
      public void       setLeft (TreeNode newLeft)    {left = newLeft;}
      public void       setRight (TreeNode newRight)  {right = newRight;}
}

public class Tree                          // Binary Search Tree...must have comparable objects
{
    private TreeNode myRoot;

    public Tree()
    {
      myRoot = null;
    }

    // need a generic implementation to add an object to the tree.
    private TreeNode addValue(TreeNode r, Comparable cmp)
    {
            // if the TreeNode is empty then create a new Node at the root, r
            // else check if the object is smaller than the current object at TreeNode, r
            //    if smaller then recursively call “addValue” to attach the object to the
            //          left side of current TreeNode, r
            //    if bigger then recursively call “addValue” to attach object to right side

            if (r == null)
                  r = new TreeNode(cmp, null, null);
            else
                  if ( cmp.compareTo( (Comparable) (r.getValue()) ) < 0)
                        r.setLeft(addValue(r.getLeft(), cmp));
                  else
                        r.setRight(addValue(r.getRight(), cmp));
            return r;   // reference to this TreeNode (needed to keep track of the real root)
    }

    // the public interface to add an object to the tree
    public void add(Comparable cmp)
    {
        myRoot = addVal( myRoot, cmp );
    }
...         // see the Data Structures directory for implementation for AP Test
}
```

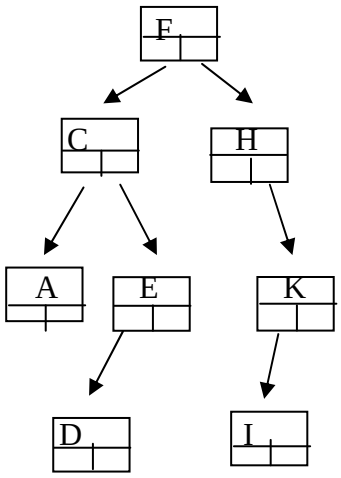
Questions:

How would you implement a “search()” method using recursion?
How would you implement a “height()” method?
How would you implement a “toString()” method?
How would you implement a “remove(Object)” method?

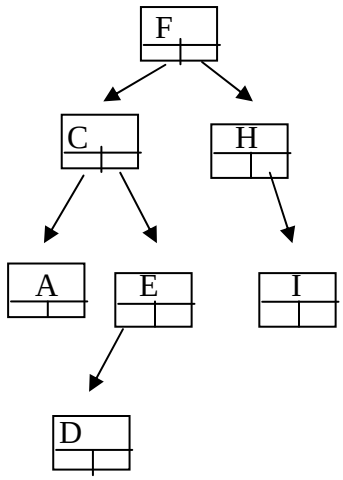
Answers:

```
    public int height()
    {
      return height(myRoot);
    }



    private int height(TreeNode root)
    {
      if (root == null) return 0;
      return 1 + Math.max(height( root.getLeft() ), height( root.getRight() ) );
    }



    public String toString()
    {
      return toString(myRoot);
    }

    private String toString(TreeNode root)
    {
            if (root == null) return "";
            else
                  return "(" + toString(root.getLeft()) +
                         " " + root.getValue() +
                         " " + toString(root.getRight()) + ")";
    }

    // Does the following work on binary trees as well as binary search Trees????
    private boolean search(Comparable c, TreeNode r)
    {
            if (r == null) return null;
            else
                  if (c.equals((Comparable)r.getValue()))
                        return true;
                  else
                        return (search(c, r.getLeft()) || search(c. r.getRight()))
    }

    public boolean search(Comparable c)
    {
            return search(c, root);
    }
```

f) The function to remove the object from the Tree, then close and reorder the remaining nodes.

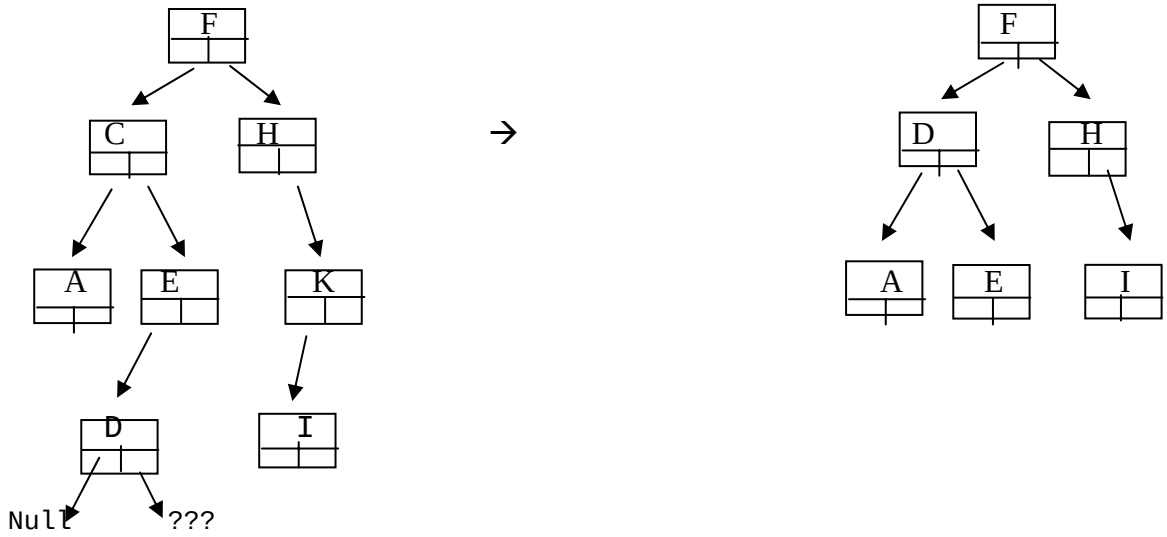


case 1: deleting node that has at most 1 child (i.e. delete K)
attach node below (i.e. node I) to parent (Node H) of deleted Node



case 2: deleting node that has 2 children (i.e. delete Node C from original)

```
step 1: find smallest element to right of node (“D”) to be deleted & its parent (“E”)
step 2: unlink this smallest element (has max 1 right branch) from parent
step 3: attach right side of smallest element to left of parent
                          “E”.setLeft(“D”.getRight())
         (NOTE: if parent is to be deleted, attach right side of
             of smallest to right side of parent)
step 4: let this smallest link take the place of deleted node
        by attaching left & right sides of old deleted node to
        left & right sides of smallest node
step 5: delete node
```




```
boolean remove(Comparable data)
      // postcondition: return true if data removed from tree
      //                      restructures tree to maintain order
      //                return false if data not found
{
      // search to find parent of node containing data
      // if not found then return false, otherwise

      // get reference to node to be deleted
      // and reference to parent node above node to be deleted

      // case 1:  node to be deleted has at most 1 child
      //          a) check if right side or left
                  b) connect parent reference to single child node below node to be deleted
      // case 2:  node to be deleted has 2 children
      //          step 1: find smallest element to right of node to be deleted & its parent
      //          step 2: unlink this smallest element (has max 1 right branch) from parent
      //          step 3: attach right side of smallest element to left of parent
      //                   (if parent is to be deleted, attach right side of
      //                       of smallest to right side of parent)
      //          step 4: let this smallest link take the place of deleted node
      //                  by attaching left & right sides of old deleted node to
      //                  left & right sides of smallest node
      //          step 5: delete node
}
```

3) Java API and AP Exam Details

a) arrays

(1) already discussed and used

(2) easy implementation

(3) used for "ranked" data since using integer indexes (not necessarily all in order)

(4) fast searching, O(n) = n

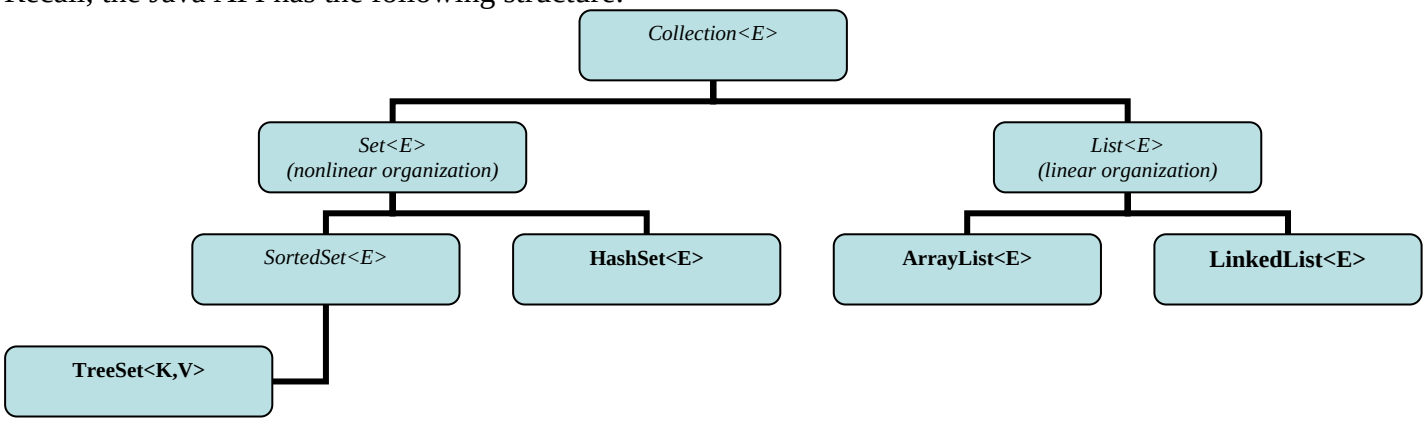
(5) slow addition/deletion O(n) = n since inserting or deleting at first position requires moving all data.

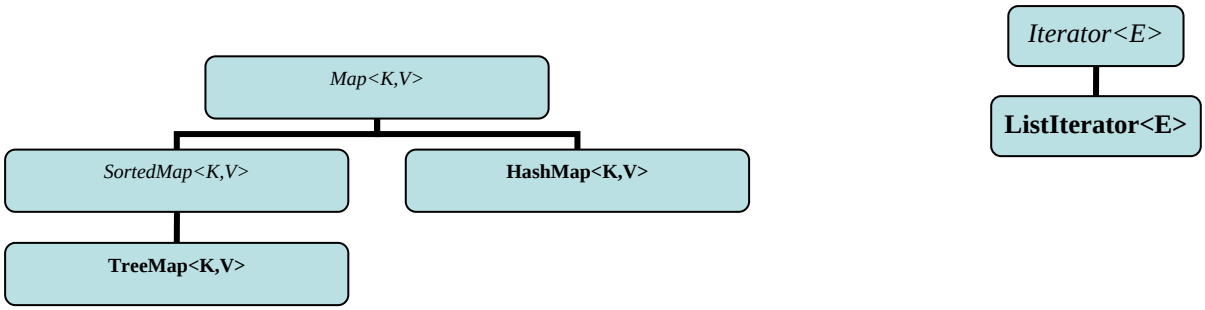
b) matrices (2 dimensional arrays) - an array of arrays

(1) int [] [] [] m = new int [3][2][5]; // 3x2x5

(2) int [] [] n = { {1,2,3}, null, {3,3,3}, {0,1,1}} // 4x3

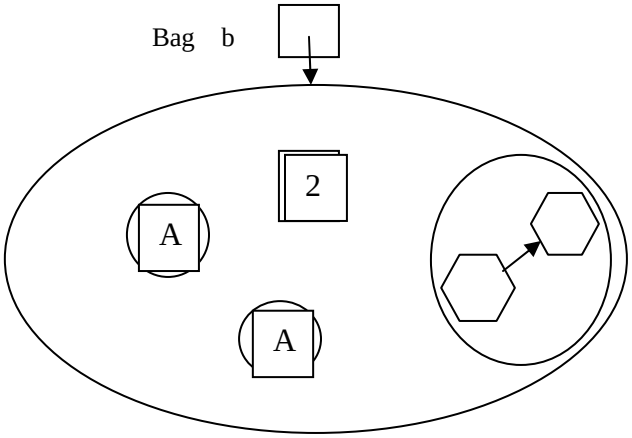
Recall, the Java API has the following structure:





c) Collections (aka "A Bag", "multiset") - *generic group of elements (repeats allowed, no order)* – technically an interface for other interfaces

(1) Drawing



(2) Notes:

(a) Useful for situations when the data/objects are "unknown" (i.e. in Adventure Games, this would be used to implement a "Bag of Holding" which can hold any type and number of items without order.

(b) Pros - most general type of grouping

(c) Cons - not efficient in terms of storage (duplicates?) or retrieval (no organization)

(3) methods for a Collection

```
(a)   boolean  add(Object o)
(b)   boolean  addAll(Collection c)
(c)   void     clear()
(d)   boolean  contains (Object o)
(e)   boolean  containsAll(Collection c)
(f)   boolean  equals(Object o)
(g)   int      hashCode()
(h)   boolean  isEmpty()
(i)   boolean  remove(Object o)
(j)   boolean  removeAll(Collection c)
(k)   boolean  retainAll(Collection c)
(l)   int      size();
(m)   Object[] toArray()
(n)   Object[] toArray(Object[] o)
(o)   Iterator iterator()
```

d) Set - ***unordered collection of unique elements (no repeats) this is an Interface!***

(1) Drawing

HashSet - unsorted Set

(uses **HashMap** instance object)

O(1) for add, remove, contains, size

O(n) for search

TreeSet – a Set that is sorted into a Tree (comparable)

(uses a **TreeMap** instance)

O(logN) for add, remove, contains & search

HashCode()	**array index**	**Data value stored in array[index]**
	0	
hashCode("B") =	**1**	"B"
	2	
	3	
hashCode("BC") =	**4**	"BC"
hashCode("E") = ?	??	where to store this?? ---> lecture later

(2) Notes:

(a) Two sets are equal if and only if both have the exact same elements regardless of implementation... (== means that they both have same hashCode())

(b) Pros – both efficient storage and easy programming. TreeSet for ordered items, HashSet is faster otherwise

(c) Cons – no duplicates...user defined Sets are difficult because of overriding equals() and hashCode() methods

(3) Set Interface methods

```
(a)   boolean  add(Object o)
(b)   int      size()
(c)   boolean  contains (Object o)
(d)   Iterator iterator()
(e)   boolean  remove(Object o)
(f)   ...
```

methods for

HashSet (no order)

```
void remove(Object)
HashSet()
HashSet(Collection)
...
```

TreeSet (ascending order)

```
Comparator comparator()
Object     first()
SortedSet  headSet(Object) // a set with elements < Object
SortedSet  tailSet(Object) // a set with elements >= Object
..
```

e) Map (aka Dictionary, Table, & Associative Array) - *collection of (key,value) pairs.*

HashMap - unsorted Map TreeMap – a Map that is organized into a Tree (Comparator or Comparable)

(1) Notes:

(a) Keys and Values can be any Objects

(b) Keys must be unique but Values must not necessarily be unique (if unique values then use Sets!)

(c) Arrays are simple Maps with the keys being the integer indexes.

(d) Map interface allows us to:

(i) insert key/value pair into map

(ii) retrieve any value given its key

(iii) test if a given key is in the map

(iv) iterate over the keys, values or key/value pairs

(v) don't remove while iterating except via i.remove() method

HashSet (unsorted, unique data) **TreeSet** (sorted, unique data)	**O() Hash**	**O() Tree**	**HashMap** (unsorted, unique keys, duplicate data) **TreeMap** (sorted, unique keys, duplicate data)	**O() Hash**	**O() Tree**
boolean contains(Object)	O( )	O( )	boolean containsKey(Object k)	O( )	O( )
boolean addValue(Object)	O( )	O( )	Object put(Object k, Object v)	O( )	O( )
boolean remove(Object)	O( )	O( )	Object remove (Object k)	O( )	O( )
			Object get(Object k)	O( )	O( )
			boolean containsValue(Object v)	O( )	O( )
Iterator iterator()	O( )	O( )			
			void putAll(Map m)	O( )	O( )
			Collection values()	O( )	O( )
			Set keySet()	O( )	O( )
			Object lastKey()		O( )
			SortedMap headMap(Object k)		O( )
			SortedMap subMap(Object fromK, Object toK)		O( )
			SortedMap tailMap(Object fromKey)		O( )

(2) General uses for Maps

(a) Anywhere an array or tree could be used since the arrays just use integer keys, more complicated to program

(b) Example – Frequency of word use in a report

```
public class WordFreq
{
        private Map m;

        public WordFreq()
        {
                m = new HashMap();
                loadMap(m);
                System.out.println(m);        // print the values in the map
        }

        public void loadMap(Map m)
        {
                < code to open the report file called "inFile" >

                while ( <there are still words in "inFile"  > )
                {
                        String work = inFile.readWord();      // readWord() reads one word in...may have to tokenize it

                        Integer i = (Integer) m.get(word);    // get the value associated with the key word

                        if (i == null)                        // new word
                                m.put(word, new Integer(1));  // put new word in the map since we've seen it once
                        else
                                m.put(word, new Integer(i.intValue()+1));  // add 1 to the number of times we've seen it
                }
        }
}
```

Hashing

Suppose we wish to store a record for each student in a school with 700 students.
We will use an array to contain the info comprised of say:

```
public class Student
{
            private String name;
            private long   ssnum ;

            //and any other appropiate info & methods ;
}
```

Questions:

1) Should we store the information by SSN or Name? Ans: SSN as may have duplicate names.
2) should we use an array or tree? Ans: array as SSN may not be "in order"

Since an array must be subscripted by an integer, the logical choice would of course be the SSN.
i.e. 371-525-6042 a 9 digit integer without the dashes.

So we have the possible values 0 - 999999999 = 1 Billion possibilities .
This is not tenable, i.e. to allocate 1 Billion "blocks" of memory to contain these large chunks of data is a tremendous waste of resources, after all we will only use 700 of these "blocks".

But, (because we look slightly ahead) to allocate for possible expansion we will allocate

```
static final   maxSize = 1000 ;        // A global for the schools size
Block          student [maxSize] ;     // whatever a Block is composed of ? (Student class)
```

So how do we decide which student to put in which Block using their SSN ? One way of doing this is using just the last 3 digits of the SSN . The possible values range from 0 – 999 (i.e. 1,000 possibilities)
.
Now a function to return this subscript which we will refer to as hashKey

```
public int hashKey   (int ssnum)
{
      int last3Digits ;

      last3Digits = ssnum % maxSize ;     // Where maxSize is a static final field

      return last3Digits ;
}
```

Why a method? Because the process might become more complicated in the future .
For instance, someone might decide to store via the Student's Name - so how does one turn a name into an integer ? How about -

```
public int   hashKey   (String name)
{
   int theKey ;
   int j  ;

   for (j = 0; j < name.getSize(); j++)
     theKey = theKey + (int)name.getAt(j);// This adds the ASCII (Unicode) values of the chars
   theKey = theKey % maxSize ;            // This sum might be too large
   return   theKey ;
}
```

Hashing

Ergo

Suppose we have the declaration *float student [10];* // A school, 10 Students Max

Store the following GPA's via their **last** digit of SSN-

GPA's: 3.97 2.54 0.01 3.40 1.59 1.58 2.00

SSN's: 397-68-3156, 398-98-1583, 458-96-0207, 227-64-4669, -2511, -3720, -2673

	0	1	2	3	4	5	6	7	8	9
Student	1.58	1.59		2.54			3.97	0.01		3.40

Either method can lead to a problem called - Collision i.e. 2 Blocks with the same hashKey() value .

One method of handling collisions when they occur is look at the next consecutive location if it is empty use this location otherwise look at the next, etc. until a empty location is found . For this reason the size of the array should be larger than the number of Blocks to be stored . In fact, the optimal size needed to prevent as many collisions as possible is a little less than 2 times the number of Blocks . But, first we need to initialize the array so that one can check to determine if the location is empty .

```
//Global
private static final Empty = -1 ;         // Some value that cannot equal the Hashkey

public void   init_array   (Block student[maxSize])
{
      int J ;

      for (J = 0; J < maxSize; J++)
         student[J] = Empty ;

}
```

Now the function to store in the next open slot

```
      public void   hashStore   (int     student [maxSize];
                                 int     item ;
                                 boolean placeFound ) ;
         {
            int startKey ;
            int tryKey ;

            placeFound = False ;
            startKey = hashKey(Item) ;
            tryKey = startKey;

            do
               {
                  if (student[tryKey] = Empty)
                     {
                        A[tryKey] = Item ;
                        placeFound = True ;
                     } //  end if

                  else
                     {
                        tryKey ++ ;
                        tryKey = tryKey % maxSize;    // Why ?  What ?
                     } //  end else

                  } // end while

            while (!placeFound) & (tryKey != startKey)

         } //  end fcn Hashstore
```

And to find a Student one can essentially use an adaptation of the above called hashRetrieve()

Other methods of collision avoidance also exist since “linear probing” is easy to program but poor performance. Quadratic Probing is another technique...see text.

<<<<<<<<<<<<<

DEMONSTRATION OF HASHSET AND HASHMAP in Folder ..>examples>data structures>Sets & Maps

1. take Student.java and Course.java get the Course application to run;
2. do as tree – won't run – must implement comparable

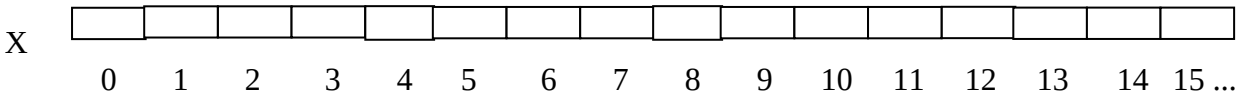
Teaching Notes on Student.java and Course.java:

(1) classes are immutable. No modifiers
(2) toString(): write this because otherwise System.out.printlong will only print the address (i.e. hashCode())
(3) Note the toString() method calls the accessors. Don't use the private instance fields – if you ever update the data structure, have to modify the whole class**????**
(4) Do we need a compareTo()? Yes if we want to add students in a tree.
(5) If we use a hash table then need equals() and hashCode() methods!!!!
(6) equals() method must be used correctly!!! (i.e. use overriding–boolean equals(Object o)--rather than overloading– boolean equals(OtherClass o)
(7) If you add something to a HashMap, you must use a hashCode.

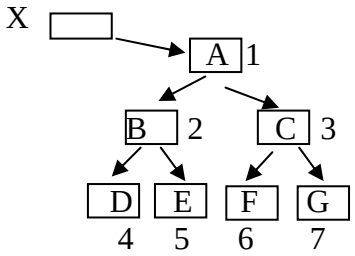
>>>>>>>>>>>>>>>>>>>

HEAPS - combining trees with arrays
(this is where "Full" and "Complete" binary trees are important)

All memory can be considered to be linear -
i.e. in line and addressed thusly -

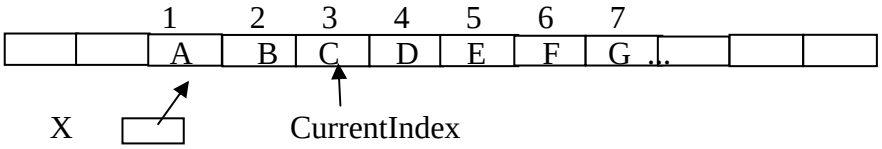


Essentially, RAM is one huge, long array.
And we can treat any part of this array as a Binary Tree in the following way -



NOTE: Numbers are same as array index!
Letters are the Data elements in a Node!

I will refer to each Letter as a "node" for convenience.
The array, X, is composed of nodes somewhere in memory. And the name X is a reference to the first node, thus cannot be changed (it is the same as the root, not root node, of the tree).



We need to adjust the subscripting so that the first node has subscript 1.
If CurrentIndex is an adjusted subscript of some node, then that node is accessed by

```
X[CurrentIndex];
```

and the adjusted subscripts of Current's children become

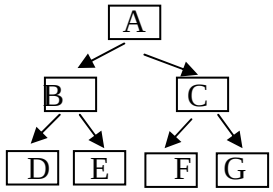
```
LeftchildIndex  = 2 * CurrentIndex;
RightchildIndex = 2 * CurrentIndex + 1;
```

So, for the node C -

```
CurrentIndex = 3
Leftchild    = 2*3 = 6          node F , and
Rightchild   = 2*3 + 1 = 7      node G.
```

A) Definition: a **Heap** is a Binary Tree with 2 properties

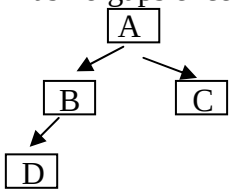
1) The parent's data is greater than either child's (Thus NonOrdered).
2) The tree must be either
   a) full - all nonleafs have 2 children



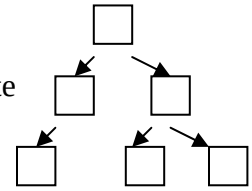
or

   b) complete - full thru the next to the last level and last level has no gaps except on right side of level.

complete



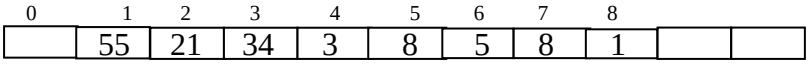
NOT complete



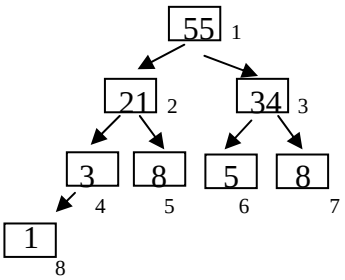
B) When we implement the Heap as the array, X,
   Property 2) is automatically implemented, and
   Property 1) the largest data must be stored in the first node so the adjusted subscript of the Maxdata = 1.

C)mapping heaps with arrays: NOTE: 0^{th} place is not used for programming convenience

array X



heap X



NOTE: heaps CAN store duplicate data!

D) heap class

```
class heap
{
      public static int MaxSize = 1024;   // reserve a maximum array/heap size

      private     int heapMaxSize;        // maximum array size
      private int numNodes;               // current number of nodes in heap
      private     Object[] buffer         // array to hold heap

      public      Heap(maxSize);          // constructor, pass default maxSize
      public      boolean isEmpty();      // tell if heap is empty
      public      boolean isFull();       // array is full
      public      boolean insert(Object newData)      // insert a new node
      public      boolean remove(Object badData);     // delete a node
};
```

E) "easy" public member functions

```
public Heap(int mS)      {heapMaxSize = mS; buffer=new Object[mS], numNodes=0;}
public boolean isEmpty() { return (numNodes == 0);}
public boolean isFull()  { return (numNodes == heapMaxSize);}
```

F) inserting a node into heap:

Two considerations:

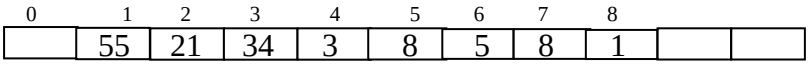
1. preserve order and
2. preserve "completeness"

Approach:

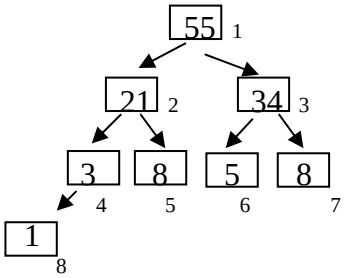
1. attach new node to "end" of heap
2. move new node to correct spot by repeatatively
   swapping with each smaller parents until new node reaches final spot

Visualization:

array X

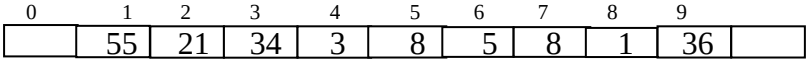


heap X

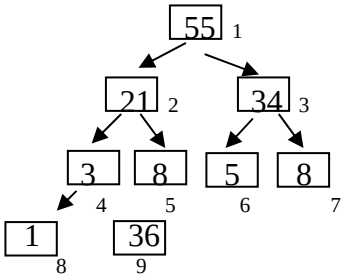


Step 1: attach to end of heap **(insert new node, say 36)**

array X



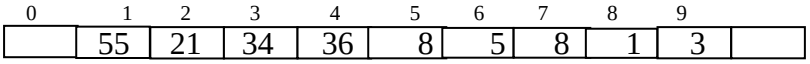
heap X



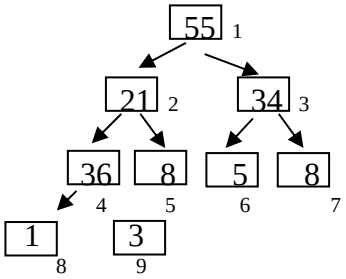
Step 2: move new node up by swapping with successive smaller parents

Note: for the array, swap index "9" with "9/2 = 4" index

array X

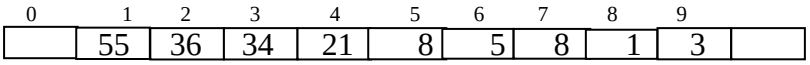


heap X

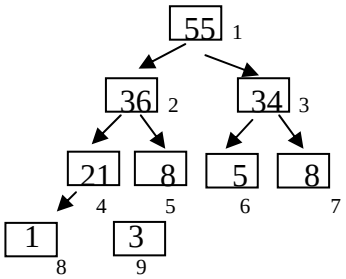


Note: now swap index "4" with "4/2 = 2" index

array X



heap X



// **member function for insert into heap**

```
boolean insert(Object newNode)
{
      // postcondition:  return true if d is inserted into heap
                         otherwise return false

      if (isFull())
            return false;

      numNodes++;       // use next slot in array

      // starting at last node, go from node i (last node) to
      // its parent node (pi) and swap with any parent smaller

      int i = numNodes;
      int pi;

      while (i > 1)
      {
            ip = i/2;
            if (d <= buffer[ip])     // if data is at right location
                  break;             // skip out of loop
            buffer[i] = buffer[ip];  // move parent down
            i = ip;
      }
      buffer[i] = d;                 // insert new data into correct spot
      return true;
}
```

G) deleting a node from top of heap -- (may use a modified version to delete any node)

Two considerations:

1. preserve order and
2. preserve "completeness"

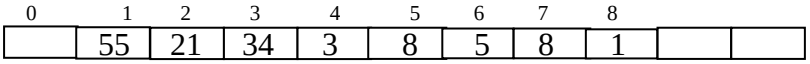
Approach:

1. remove node from root
2. move last node to root
3. move root down by swapping with larger nodes below it until in place

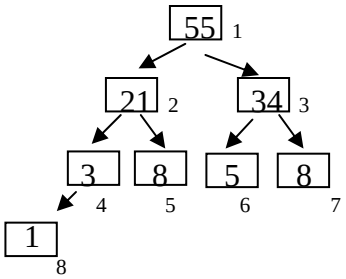
Visualization:

Original heap array

array X

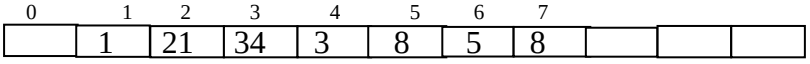


heap X

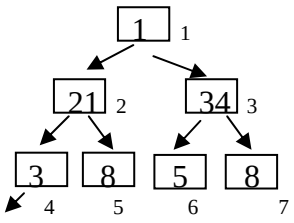


Steps 1 & 2: remove node (55) from root & move last node (1) to root

array X



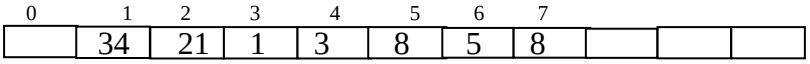
heap X



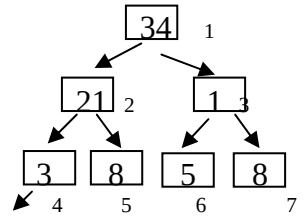
Step 3: move "small" root node down by swapping with largest node below it

move down 1 level

array X

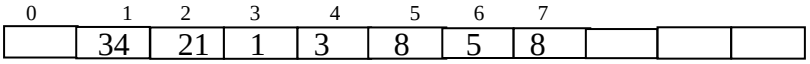


heap X

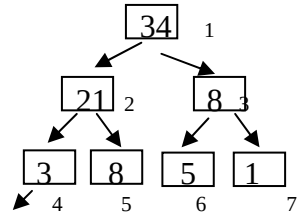


move down 2nd level

array X



heap X

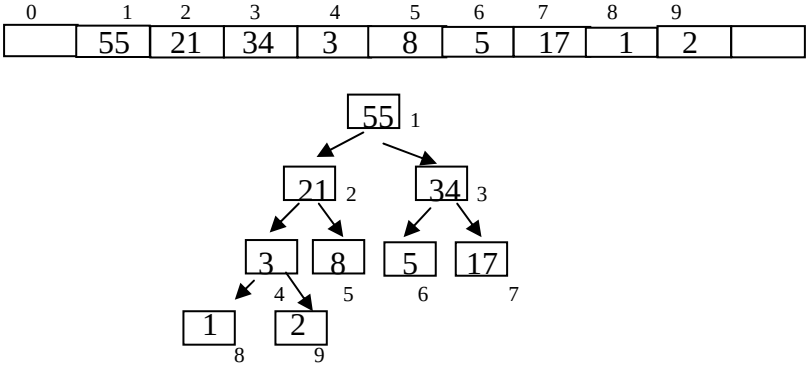


// **member function to delete root node of heap**

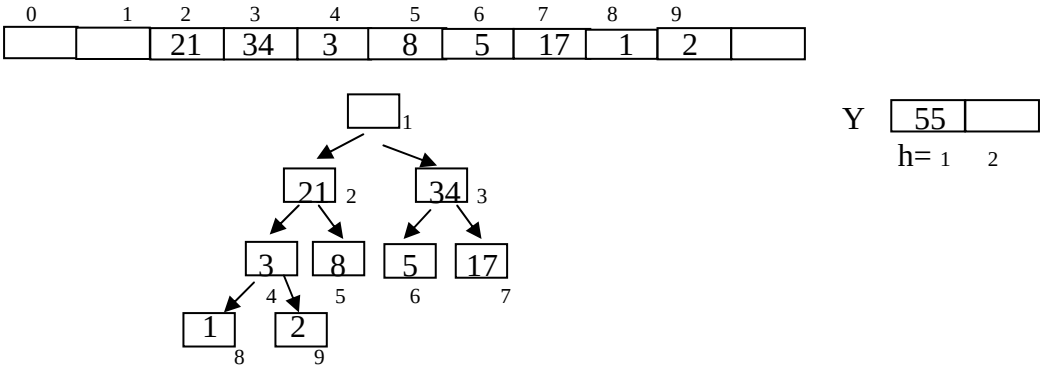
```
boolean remove()
{
      // postcondition:  returns true if largest element (root node) is deleted
      //                 false otherwise

      if (isEmpty())    return false;

      // get top element
      d = buffer[1];

      // starting from vacant root (ip), go from parent node (ip) to its
      // largest child (i) and, as long as ip has a larger child than last
      // element of heap, move child up

      int ip = 1;       // root
      int i  = 2;       // start at left child

      while (i <= numNodes)
      {
            // set i to right child (i+1) if it exists and is larger
            if ((i < numNodes) && (buffer[i] < buffer[i+1])) i++;

            // if this last node is bigger than largest child then get out of loop
            if (buffer[i] <= buffer[numNodes]   break;

            buffer[ip] = buffer[i];       // move large child up
            ip = i;                       // look at node down one level
            i *= 2;                       // i now is at left child
      } // while

      // move last node to the correct slot in heap
      if (numNodes > 1)
            buffer[ip] = buffer[numNodes];
      numNodes--;                                 // one less node
      return true;                                // deleted a node
}
```

Heap Sort – Time and Space Analysis

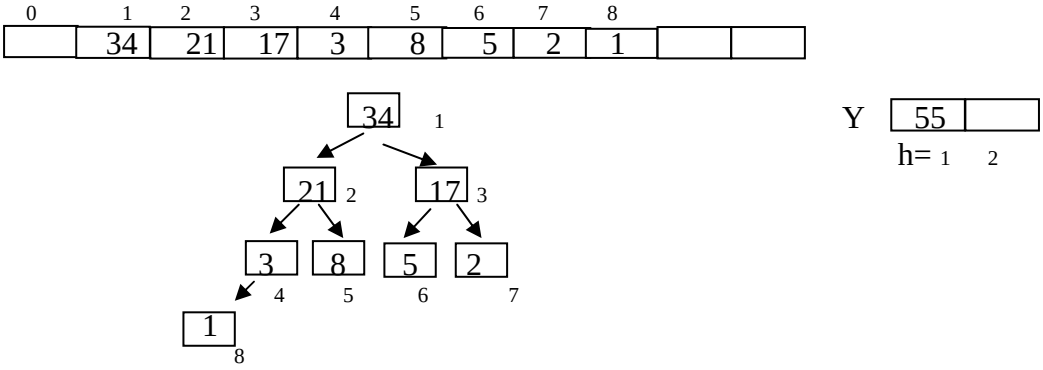
Step 1: Create the heap --> Time analysis is O(N), Space analysis has an array size of N = 9



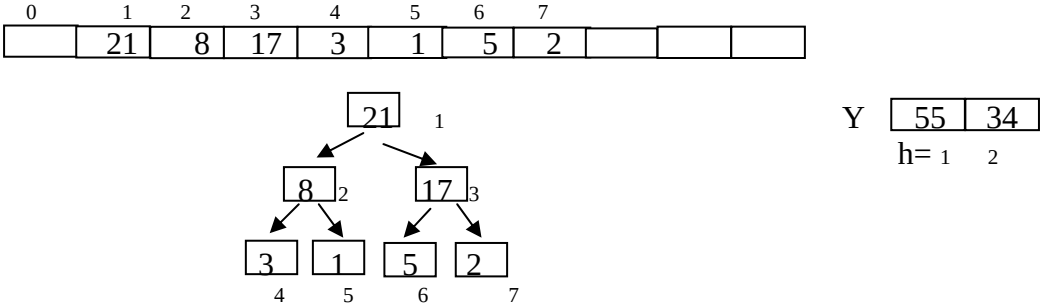
Step 2: Find maximum at X[1] ---> O(1)
Step 3a: Remove maximum;
Step 3b: Put maximum in a new array at Y[h] then increase h++ --> O(1); Space analysis means double memory



Step 3c: Re-heap ---> O(logN), no extra Space



Step 4: Repeat step 3 until no elements in heap --> Step 3 is done N times



Overall: Big-Oh is N times logN or O(NlogN) and space analysis means need 2 x amount of memory
(Won't need extra array if keeping track of "deleted" end element of original array)

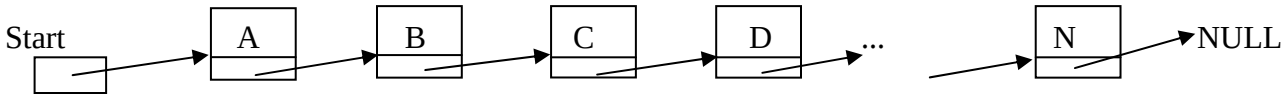
f) Priority Queues (aka Ordered Linked Lists) – implemented as Heaps

1) The overview -

A) Definition : A Priority Queue is a Simple Linked List where the data is ordered or "prioritized" or "ordered" in some fashion. – implemented as a Heap (ArrayList or Tree)

B) An example would be information stored for the IRS, which would necessarily be ordered by SS number.

C) The picture of a Priority Queue. named Start -



where A < B < C < ... < N

D) Since Java does not come with it's own PriorityQueue class, we declare an interface which will be tested on the AP Exam.

```
public interface PriorityQueue
{
    // postcondition:         Returns true if the number of elements in priority queue is 0;
    //                        Otherwise, returns false
    boolean isEmpty();

    // postcondition:         x has been added to the priority queue;
    //                        Number of elements in the priority queue is increased by 1.
    void add(Object x);

    // postcondition:         The smallest item in the priority queue is removed and returned;
    //                        The number of elements in the priority queue is decreased by 1.
    //                        Throws an unchecked exception if the priority queue is empty
    Object removeMin();

    // postcondition:         The smallest item in the priority queue is returned;
    //                        The priority queue is unchanged
    //                        Throws an unchecked exception if the priority queue is empty
    Object peekMin();
}
```

E) To use the interface, we create a “concrete class” using an ArrayList.

```
import java.util.*;

public class ArrayPriorityQueue implements PriorityQueue
{
  private List items;            // not an ArrayList since more General and add() method is O(1) for
                                 // List interface as opposed to O(n) for ArrayList class

  public ArrayPriorityQueue()
  {
    items = new ArrayList();     // items is an ArrayList but can only use List interface methods!
  }

  public boolean isEmpty()
  {
    return (items.size() == 0);
  }

  public void add(Object x)
  {
    items.add(x);                // Qst:  What class has the “add()” method?
                                 // Ans:  Declared in List but defined in ArrayList.
  }

  public Object removeMin()
  {
    Object min = peekMin();
    items.remove(min);
    return min;
  }

  public Object peekMin()
  {
    int minIndex = 0;
    for (int i = 1; i < items.size(); i++)     //...SEE LECTURE/DEMONSTRATION ON ITERATOR VS INDEXING
    {

      // Must cast "calling object" but do not cast the argument
      // ...the compareTo() method will do the casting...
      // If in doubt about whether the argument can be "Comparable" to the calling object,
      // ...encase in a "try" block and catch the "illegalClassCastException"

      // alternatively, we might try using Generics
      if (((Comparable) items.get(i)).compareTo(items.get(minIndex)) < 0)
      {
         minIndex = i;
      }
    }
    return items.get(minIndex);
  }

}
```

Iterator versus Indexing >>>>>>>>>>>>>> USE DEMONSTRATION PROGRAM

```
import java.util.*;

public class IteratorTest
{
       public IteratorTest()
       {
           ArrayList list = initialize();
           Integer ZERO = new Integer(0);             // need a “0” object

           System.out.println("Original List");
           printArray(list);

           // This works and is the recommended way
           System.out.println("Iterator WHILE remove() of all ZEROS ");
           Iterator it = list.iterator();
           while (it.hasNext())
              if ( ZERO.equals(it.next()) )
                  it.remove();
           printArray(list);


           // works but is awkward
           System.out.println("Iterator FOR with remove() of all ZEROS ");
           list = initialize();
           for ( it = list.iterator(); it.hasNext(); )
              if ( ZERO.equals(it.next()) )
                  it.remove();
           printArray(list);


           // Won't work for "zeros that are sequential"...see initializer() method
           System.out.println("Indexed FOR with remove() of all ZEROES");
           list = initialize();
           for ( int i = 0; i < list.size(); i++ )
              if ( ZERO.equals(list.get(i)) )
                 list.remove(i);
           printArray(list);

           // works since List moves deleted items to "decreasing" index
           System.out.println("Reversed Indexed FOR with remove() of all ZEROES");
           list = initialize();
           for ( int i = list.size() - 1; i >= 0; i-- )
             if ( ZERO.equals(list.get(i)) )
                 list.remove(i);
           printArray(list);
       }

       public void printArray(List l)
       {
            System.out.println("");
            Iterator i = l.iterator();
            while (i.hasNext())
                System.out.print(" "+i.next());
       }

       public ArrayList initialize()
       {
            ArrayList l = new ArrayList();
            for (int i = 0; i<10; i++)
            {
               l.add(new Integer(i-5));
               if (i==5) l.add(new Integer(0));      // multiple sequential “0”s may cause problems
            }
            return l;
       }

       public static void main(String a[])
       {     new IteratorTest(); }
}
```