Arrays

Definition

An array is a sequence of variables of the same type arranged contiguously in a block of memory.

Basic Knowledge in Java API

1. Declare and initialize an array in java:

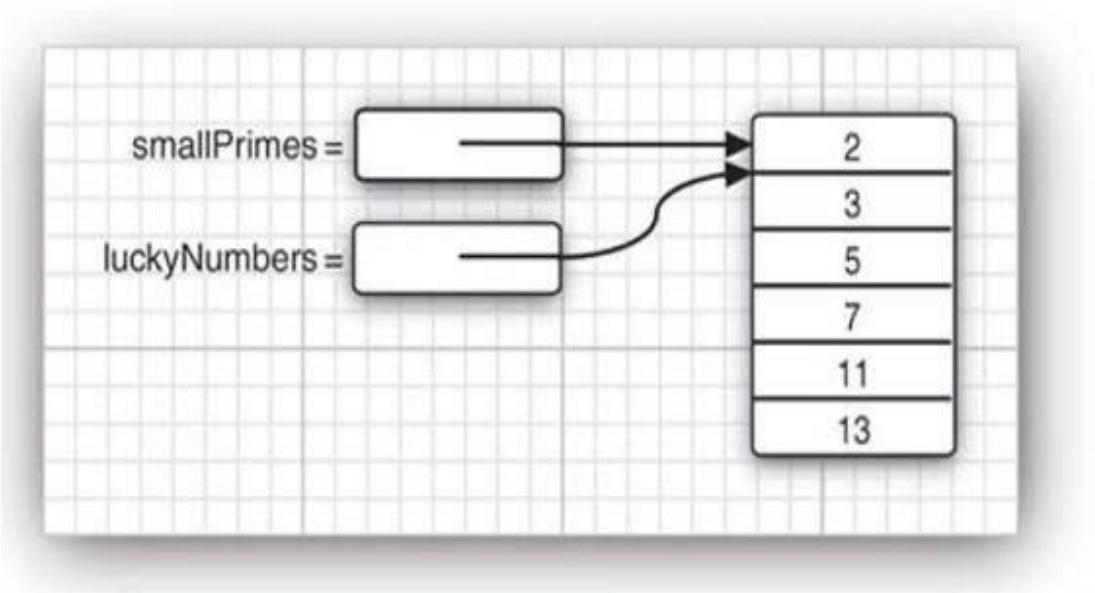
int[] array1 = new int[100];
String array2 = new String[100];
int[] arrays={1,2,3,4,5};

2. Array Length

A very important parameter about array is length:
for(int i=0;i<array1.length;i++)

3. Array copying.

You can copy one array variable into another, but then both variables refer to the same array:
int[] luckyNumbers=smallPrimes;
luckyNumbers[5]=12;//now smallPrimes[5] is also 12



If you want a new copy of current array, you could use the api provided by Arrays:
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);

copyOf

```
public static double[] copyOf(double[] original,
                              int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain 0d. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

4. Array Sorting

You could also sort the array with quick sort:

Arrays.sort(array1);

Method Detail

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

Big O analysis (lookup O(1), Insertion/Deletion O(N))

Array lookup is O(1) as long as you know the index of the element you want. The provision of index is important, that is to say, if you only know the value, lookup is still O(N).

The price for this improved lookup is significantly decreased efficiency for insertion and deletion of data in the middle of the array. Because an array is essentially a block of contiguous memory, it's not possible to create or eliminate storage between any two elements as it is with a linked list. Instead, you must physically move data within the array to make room for an insertion or to close the gap left by a deletion; this is an

O(N) operation.

Implementation

Binary Search (O(logN))

```
public int binarysearch(int[] nums, int start, int end, int target){
    int mid=(start+end)/2;
    if(end<start)
        return -1;
    if(nums[mid]==target)
        return mid;
    if(nums[mid]>target)
        return binarysearch(nums,start,mid-1,target);
    if(nums[mid]<target)
        return binarysearch(nums,mid+1,end,target);
    return -1;
}
```

Summary

Arrays are not dynamic data structures: They have a finite, fixed number of elements. Memory must be allocated for every element in an array, even if only part of the array is used. Arrays are best used when you know how many elements you need to store before the program executes. When the program needs a variable amount of storage, the size of the array imposes an arbitrary limit on the amount of data that can be stored. Making the array large enough so that the program always operates below the limit doesn't solve the problem: Either you waste memory or you won't have enough memory to handle the largest data sizes possible.

Strings

Definition

Conceptually, Java strings are sequences of Unicode characters. characters are two bytes in java.

Basic Knowledge in Java API

1. Declaration and Initialization

Java does not have a built-in string type. Instead, the standard java library contains a predefined class called String. Each quoted string is an instance of the String class:

String e = ””; // an empty string

String greeting = “hello”;

2. Substrings

You can extract a substring from a larger string with the substring method of the String class. For example,

String greeting = “Hello”;

String s = greeting.substring(0,3);

substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

Examples:

```
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"
```

Parameters:

`beginIndex` - the beginning index, inclusive.

`endIndex` - the ending index, exclusive.

Returns:

the specified substring.

Throws:

`IndexOutOfBoundsException` - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String` object, or `beginIndex` is larger than `endIndex`.

3. Concatenation

Java, like most programming languages, allows you to use + to join (concatenate) two strings.

String expletive = "Expletive";

String PG13 = "deleted";

String message = expletive + PG13;

4. Strings are Immutable

They cannot be changed after the string has been constructed. Methods that appear to modify a string actually return a new string instance.

String is immutable for several reasons.

- **Security**: parameters are typically represented as `String` in network connections, database connection urls, usernames/passwords etc. If it were mutable, these parameters could be easily changed.
- **Synchronization** and concurrency: making String immutable automatically makes them thread safe thereby solving the synchronization issues.
- **Caching**: when compiler optimizes your String objects, it sees that if two objects have same value (a="test", and b="test") and thus you need only one string object (for both a and b, these two will point to the same object).
- **Class loading**: `String` is used as arguments for class loading. If mutable, it could result in wrong class being loaded (because mutable objects change their state).

5. Testing Strings for Equality

To test whether two strings are equal, use the equals method.

s.equals(t);

equals

`public boolean equals(Object anObject)`

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

Overrides:

`equals` in class `Object`

Parameters:

`anObject` - The object to compare this `String` against

Returns:

`true` if the given object represents a `String` equivalent to this string, `false` otherwise

See Also:

`compareTo(String)`, `equalsIgnoreCase(String)`

6. Empty and Null Strings

The empty string is “” is a string of length 0. An empty string is a java object which holds the string length (namely 0) and an empty contents. However, a String variable can also hold a special value, called null, which indicates that no object is currently associated with the variable.

Sometimes, you need to test that a string is neither null nor empty.

if(str!=null&&str.length()!=0)

7. Code Points and Code Units

charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

8. String Building

Building a new string each time is time-consuming and wastes of memory. The StringBuffer and StringBuilder classes (the former is in all versions of Java and is thread-safe; the latter is newer and higher performance, but nonthread-safe) create mutable strings that can be converted to a String instance as necessary.

It would be more efficiently coded as:

```
StringBuffer b = new StringBuffer();
for( int i = 0; i < 10; ++i ){
b.append( i );
b.append( ' ' );
}
String s = b.toString();
```

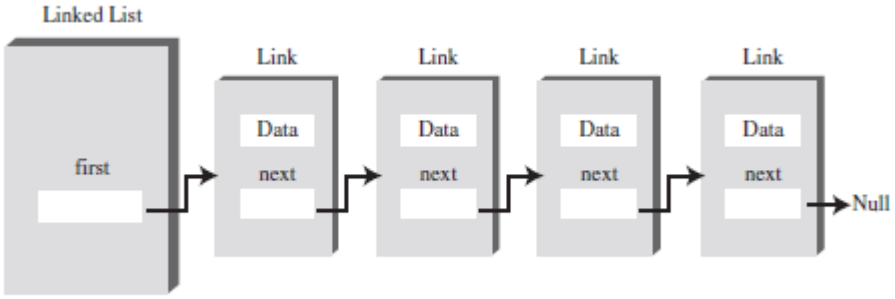
Summary

Strings are one of the most common applications of arrays. String objects can be converted to and from character arrays; make sure you know how to do this in the languages you'll be using because the operations required by programming problems are often more convenient with arrays. Basic string objects are immutable (read-only) in C# and Java; other classes provide writeable string functionality. Careless concatenation of immutable strings can lead to inefficient code that creates and throws away many string objects.

LinkedLists

Definition

An ordered sequence that allows efficient insertion and removal at any location.



My Own Implementation in Java

```
/**
 * ClassName: GenericSingleLinkedList
 * Description: A class to implement single linkedlist in java using generic
 * Date: 2017-4-17
 *
 * Update History:
 *
 * @version 1.0
 * @author Steve Shao
 /
/**
 *Node class, which holds data and contains next which points to next Node.
 */
class Node<T> {
    public T data; // data in Node.
    public Node<T> next; // points to next Node in list.
    /**
     * Constructor
     */
    public Node(T data){
    this.data = data;
    }

    public void displaydata(){
    System.out.print(data+" ");
    }
}
/**
 * generic Single LinkedList class (Generic implementation)
```

```
 */
class LinkedList<T> {
    private Node<T> first;
    /**
     * generic Single LinkedList constructor
     */
    public LinkedList(){
          first=null;
    }

    /*
     * check the single linkedlist is empty
     */
     public boolean isEmpty(){
     return first==null;
     }
    /**
     * Insert New Node at first position of generic Single LinkedList
     */
    public void insertFirst(Node<T> newNode) {
          //Node<T> newNode = new Node<T>(data);  //Creation of New Node.
          newNode.next=first;   //newLink ---> old first
          first = newNode;  //first ---> newNode
    }

    /**
     * Deletes first Node of generic Single LinkedList
     */
    public Node<T> deleteFirst()
    {
          if(first==null){
           throw new IllegalArgumentException("Cannot remove the empty
linkedlist");

          }
          Node<T> tempNode = first; // save reference to first Node in
tempNode- so that we could return saved reference.
          first = first.next; // delete first Node (make first point to
second node)
          return tempNode; // return tempNode (i.e. deleted Node)
    }

    /**
     * Display generic Single LinkedList
```

```
     */
    public void displayLinkedList() {
          System.out.print("Displaying LinkedList [first--->last]: ");
          Node<T> tempDisplay = first; // start at the beginning of
linkedList
          while (tempDisplay != null){ // Executes until we don't find end of
list.
                tempDisplay.displaydata();
                tempDisplay = tempDisplay.next; // move to next Node
          }
          System.out.println(" ");
    }
}
```

Big O analysis (lookup O(N), Insertion/Deletion O(1))

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time. Finding, deleting, or inserting next to a specific item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted. The increased efficiency can be significant, especially if a copy takes much longer than a comparison. Of course, another important advantage of linked lists over arrays is that a linked list uses exactly as much memory as it needs and can expand to fill all of available memory. The size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small. Vectors, which are expandable arrays, may solve this problem to some extent, but they usually expand in fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). This solution is still not as efficient a use of memory as a linked list.

Basic API in Java

1. Declaration and Initialization

```
List<Integer> newlist = new LinkedList<Integer>();
```

2. Iterator/ ListIterator

Objects containing references to items in data structures, used to traverse these structures, are commonly called iterators (or sometimes, as in certain Java classes, enumerators).

```
List<String> staff = new LinkedList<>(); // LinkedList implements List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();//ListIterator<String> iter=staff.listIterator();
String first = iter.next(); // visit first element
```

String second = iter.next(); // visit second element
iter.remove(); // remove last visited element

3. get, add and remove

get

`E get(int index)`

Returns the element at the specified position in this list.

Parameters:

`index` - index of the element to return

Returns:

the element at the specified position in this list

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

---

add

`boolean add(E e)`

Appends the specified element to the end of this list (optional operation).

Lists that support this operation may place limitations on what elements may be added to this list. In particular, some lists will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. List classes should clearly specify in their documentation any restrictions on what elements may be added.

Specified by:

`add` in interface `Collection<E>`

Parameters:

`e` - element to be appended to this list

Returns:

`true` (as specified by `Collection.add(E)`)

Throws:

`UnsupportedOperationException` - if the `add` operation is not supported by this list

`ClassCastException` - if the class of the specified element prevents it from being added to this list

`NullPointerException` - if the specified element is null and this list does not permit null elements

`IllegalArgumentException` - if some property of this element prevents it from being added to this list

remove

`boolean remove(Object o)`

Removes the first occurrence of the specified element from this list, if it is present (optional operation). If this list does not contain the element, it is unchanged. More formally, removes the element with the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))` (if such an element exists). Returns `true` if this list contained the specified element (or equivalently, if this list changed as a result of the call).

Specified by:

`remove` in interface `Collection<E>`

Parameters:

`o` - element to be removed from this list, if present

Returns:

`true` if this list contained the specified element

Throws:

`ClassCastException` - if the type of the specified element is incompatible with this list (optional)

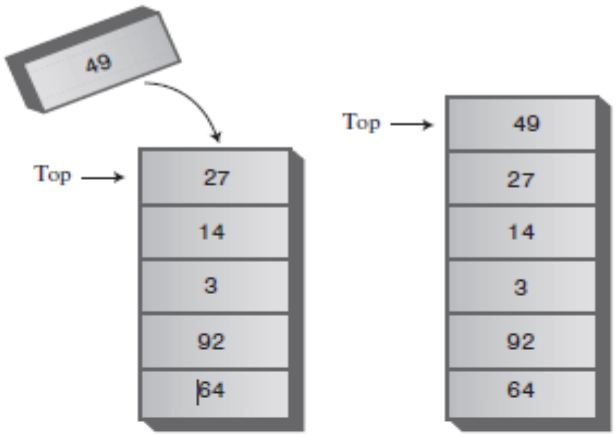
`NullPointerException` - if the specified element is null and this list does not permit null elements (optional)

`UnsupportedOperationException` - if the `remove` operation is not supported by this list

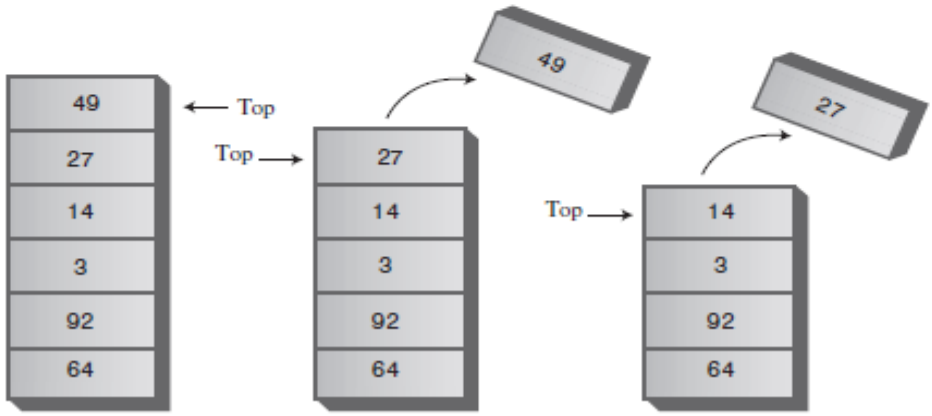
Stack

Definition

A stack is a last-in-first-out (LIFO) data structure: Elements are always removed in the reverse order in which they were added, much in the same way that you add or remove a dish from a stack of dishes.



New item pushed on stack



Two items popped from stack

My Own Implementation in java

```
/**
 * ClassName: Mystack
 * Description: A class to implement stack in java using singly linked list
 * Date: 2017-4-18
 *
 * Update History:
 *
 *
 * @version 1.0
 * @author Steve Shao
```

```
 */
/*
 *Stack element class, which holds data and contains next which points to
next Node.
 */
class Element<T>{
    private T data;
    public Element<T> next;

    public Element(T data){
        this.data=data;
    }

    public void setdata(T data){
        this.data=data;
    }
    public T getdata(){
        return data;
    }
    public void display(){
        System.out.print(data+" ");
    }
}
/*
 * Stack class that has three functions pop() push() and peek()
 */
class Stack<T>{
    private Element<T> first;
    private int currentsize;
   private int capacity;

    public Stack(int capicity){
        first=null;
        currentsize=0;
        this.capacity=capicity;
    }

    public Element<T> pop(){
        if(first==null)
            throw new IllegalArgumentException("cannot pop empty stack");
        else{
            Element<T> tempelement;
           tempelement=first;
           first=tempelement.next;
```

```
            currentsize--;
            return tempelement;

        }
    }
    public void push(T data){
     if(peek()){
         throw new IllegalArgumentException("cannot push more data");
     }
     else{
         Element<T> newelement=new Element<T>(data);
        newelement.next=first;
        first=newelement;
        currentsize++;
     }
    }
    public boolean peek(){
     return currentsize==capacity;
    }

    public void display(){
     Element<T> tempelement;
     if(first!=null){
         tempelement=first;
         System.out.print("Stack has: ");
         while(tempelement!=null){
             tempelement.display();
             tempelement=tempelement.next;
         }
     }
     else
         throw new IllegalArgumentException("cannnot display empty stack");
    }
}
```

Big O analysis (O(1))

Items can be both pushed and popped from the stack implemented in the Stack class in constant O(1) time. That is, the time is not dependent on how many items are in the stack and is therefore very quick. No comparisons or moves are necessary.

Basic API in Java

1. Declaration and Initialization

Stack<Integer> mystack = new Stack<Integer>();

2. Pop and Push

push

```
public E push(E item)
```

Pushes an item onto the top of this stack. This has exactly the same effect as:

```
addElement(item)
```

Parameters:

`item` - the item to be pushed onto this stack.

Returns:

the `item` argument.

See Also:

`Vector.addElement(E)`

pop

```
public E pop()
```

Removes the object at the top of this stack and returns that object as the value of this function.

Returns:

The object at the top of this stack (the last item of the `Vector` object).

Throws:

`EmptyStackException` - if this stack is empty.

Queue

Definition

In computer science a queue is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, FIFO).

My own Implementation in java

```
/**
 * ClassName: MyQueue
 * Description: My own implementation of queue in java
 *
 * Date: 2017-5-14
 *
 * Update History:
 *
 * @version 1.0
 * @author Steve Shao
 */
class Queue
{
        private int maxSize;
        private long[] queArray;
        private int front;
        private int rear;

        public Queue(int s) // constructor
        {
        maxSize = s;
        queArray = new long[maxSize];
        front = 0;
        rear = -1;
        }

        public void insert(long j) // put item at rear of queue
        {
        if(rear == maxSize-1) // deal with wraparound
            rear = -1;
        queArray[++rear] = j; // increment rear and insert
        }

        public long remove() // take item from front of queue
        {
        long temp = queArray[front++]; // get value and incr front
```

```
        if(front == maxSize) // deal with wraparound
            front = 0;
        return temp;
        }
}
```

Big O analysis (O(1))

As with a stack, items can be inserted and removed from a queue in O(1) time.

Basic API in Java

1. Declaration and Initialization

Queue<Integer> myqueue = new Queue<Integer>();

2. Add and Remove

add

`boolean add(E e)`

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.

Specified by:

`add` in interface `Collection<E>`

Parameters:

`e` - the element to add

Returns:

`true` (as specified by `Collection.add(E)`)

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

`ClassCastException` - if the class of the specified element prevents it from being added to this queue

`NullPointerException` - if the specified element is null and this queue does not permit null elements

`IllegalArgumentException` - if some property of this element prevents it from being added to this queue

remove

`E remove()`

Retrieves and removes the head of this queue. This method differs from `poll` only in that it throws an exception if this queue is empty.

Returns:

the head of this queue

Throws:

`NoSuchElementException` - if this queue is empty

3. Priority Queue

A priority queue retrieves elements in sorted order after they were inserted in arbitrary order. That is, whenever you call the remove method, you get the smallest element currently in the priority queue. However, the priority queue does not sort all its elements. If you iterate over the elements, they are not necessarily sorted. The priority queue makes use of an elegant and efficient data structure called a heap. A heap is a self-organizing binary tree in which add and remove operations cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

PriorityQueue

```
public PriorityQueue(int initialCapacity,
                     Comparator<? super E> comparator)
```

Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.

Parameters:

`initialCapacity` - the initial capacity for this priority queue

`comparator` - the comparator that will be used to order this priority queue. If `null`, the natural ordering of the elements will be used.

Throws:

`IllegalArgumentException` - if `initialCapacity` is less than 1

```
/*
        * Creates a PriorityQueue with the specified initial capacity that
orders its elements according to the specified comparator.
        */
       PriorityQueue<ListNode> minHeap = new
PriorityQueue<ListNode>(lists.length, new Comparator<ListNode>() {
           public int compare(ListNode l1, ListNode l2) {
               return l1.val - l2.val;
           }
       });
```

Trees (Resursion)

Definition

A tree is made up of nodes (data elements) with zero, one, or several references (or pointers) to other nodes. Each node has only one other node referencing it.

My own Implementation in java

```
/**
 * ClassName: MyBST
 * Description: A class that implement BST which has find, delete and insert
functions
 * Date: 2017-4-21
 *
 * Update History:
 *
 * @version 1.0
 * @author Steve Shao
 */

/*
 * Tree Node of BST
 */
class TreeNode{
    private int data;
    public TreeNode leftchild;
    public TreeNode rightchild;

    //constructor
    public TreeNode(int data){
        this.setData(data);
        leftchild=null;
        rightchild=null;
    }
    public int getData() {
        return data;
    }
    public void setData(int data) {
        this.data = data;
    }
}
/*
 * My own BST
 */
```

```
class BST{
    public TreeNode root;

    //constructor
    public BST(){
        root=null;
    }

    public TreeNode find(int data){
        TreeNode newnode = new TreeNode(data);
        /*
         * use two node pointer to keep track of nodes here
         */
        TreeNode currentnode=root;
        if(root==null)
            throw new IllegalArgumentException("Cannot find TreeNode in a
empty BST");
        while(currentnode!=null){
          if(newnode.getData()>currentnode.getData())
              currentnode=currentnode.rightchild;
          if(newnode.getData()<currentnode.getData())
              currentnode=currentnode.leftchild;
          if(currentnode.getData()==newnode.getData())
              return currentnode;
        }
        throw new IllegalArgumentException("Cannot find TreeNode in a empty
BST");
    }

    public void insert(int data){
        TreeNode newnode = new TreeNode(data);

        if(root==null)//when the tree is empty
            root=newnode;
        else{
            /*
             * use two node pointer to keep the track of node need to be
inserted
             */
            TreeNode currentnode=root;
            TreeNode parentnode;
            while(currentnode!=null){
              parentnode=currentnode;
              if(newnode.getData()>=currentnode.getData()){
```

```
                currentnode=currentnode.rightchild;
                if(currentnode==null)
                    parentnode.rightchild=newnode;
            }
            else{
                currentnode=currentnode.leftchild;
                if(currentnode==null)
                    parentnode.leftchild=newnode;
            }
        }
    }
}
/*
 * delete has three different situations here
 * 1.the delete node has no child
 * 2.the delete node has one child
 * 3.the delete node has two children
 */
public void delete(int data){
    TreeNode deletednode=new TreeNode(data);
    TreeNode currentnode=root;
    TreeNode parentnode = null;
    boolean isright = false;
    boolean isleft = false;
    /*
     * find the node firstly
     * And also record the parent node of current deleted node
     */
    if(root==null)
        throw new IllegalArgumentException("cannot delete a empty BST");
    else{
        while(currentnode!=null){
            parentnode=currentnode;
            if(deletednode.getData()>currentnode.getData()){
                currentnode=currentnode.rightchild;
                isright=true;
            }
            if(deletednode.getData()<currentnode.getData()){
                currentnode=currentnode.leftchild;
                isleft=true;
            }
            if(deletednode.getData()==currentnode.getData())
                break;
        }
```

```
            if(currentnode==null)
                throw new IllegalArgumentException("cannot find that data in
BST");
        }
        /*
         * first case
         */
        if(currentnode.leftchild==null&currentnode.rightchild==null){
            if(isright)
                parentnode.rightchild=null;
           if(isleft)
            parentnode.leftchild=null;
        }
        /*
         * second case
         */
        if(currentnode.leftchild!=null&currentnode.rightchild==null){
            if(isright)
                parentnode.rightchild=currentnode.leftchild;
           if(isleft)
            parentnode.leftchild=currentnode.leftchild;
        }
        if(currentnode.leftchild==null&currentnode.rightchild!=null){
            if(isright)
                parentnode.rightchild=currentnode.rightchild;
           if(isleft)
            parentnode.leftchild=currentnode.rightchild;
        }
        /*
         * the third case
         * find the submission of current node and then replace that
         */
        if(currentnode.leftchild!=null&currentnode.rightchild!=null){
            /*
             * find the successor firstly
             */
            TreeNode parent = currentnode;
            TreeNode successor=currentnode.rightchild;
            while(successor.leftchild!=null){
                parent=successor;
                successor=successor.leftchild;
                }
            /*
             * if successor is not the rightchild of currentnode then make
```

```
connection
                */
               if(successor!=currentnode.rightchild){
                   parent.leftchild=null;
                   successor.leftchild=currentnode.leftchild;
                   successor.rightchild=currentnode.rightchild;
               }
               else
                   successor.leftchild=currentnode.leftchild;
               /*
                * make connection to the parent node here
                */
               if(isright)
                   parentnode.rightchild=successor;
              if(isleft)
               parentnode.leftchild=successor;
              if(currentnode==root)//no parent node update root
               root=successor;

          }
     }
}
```

Big O analysis (lookup is O(logN) Deletion and insertion are O(logN))

One important caveat exists in saying that lookup is *O*(log(*n*)) in a BST: Lookup is only *O*(log(*n*)) if you can guarantee that the number of nodes remaining to be searched will be halved or nearly halved on each iteration.

Tree Represented as Arrays

Trees can be represented in the computer's memory as an array, although the reference-based approach is more common.If a node's index number is index, this node's left child is 2*index + 1, its right child is 2*index + 2 and its parent is (index-1) / 2.

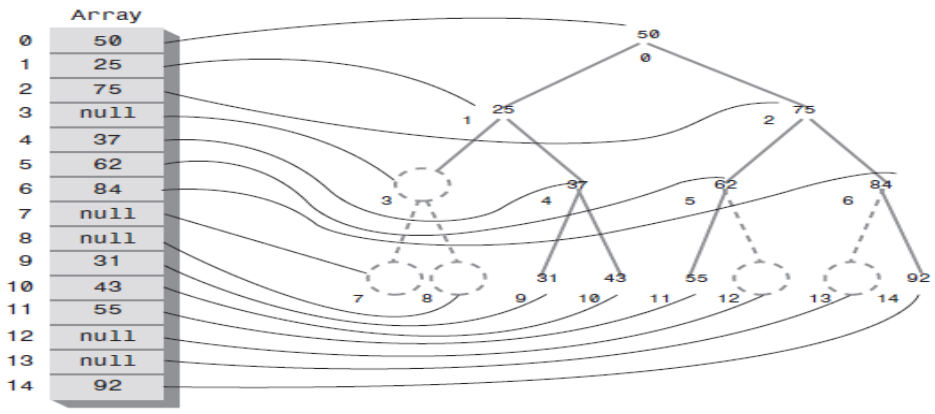


FIGURE 8.21 Tree represented by an array.

Common Searches

BFS (Heap/Queue)

One way to search a tree is to do a *breadth-first search (BFS)*. In a BFS you start with the root, move left to right across the second level, then move left to right across the third level, and so forth. You continue the search until either you have examined all the nodes or you find the node you are searching for. A BFS uses additional memory because it is necessary to track the child nodes for all nodes on a given level while searching that level.

DFS (Stack)

Another common way to search for a node is by using a *depth-first search (DFS)*. A depth-first search follows one branch of the tree down as many levels as possible until the target node is found or the end is reached. When the search can't go down any farther, it is continued at the nearest ancestor with unexplored children. DFS has much lower memory requirements than BFS because it is not necessary to store all the child pointers at each level. If you have additional information on the likely location of your target node, one or the other of these algorithms may be more efficient. For instance, if your node is likely to be in the upper levels of the tree, BFS is most efficient. If the target node is likely to be in the lower levels of the tree, DFS has the advantage that it doesn't examine any single level last. (BFS always examines the lowest level last.)

Traversals

➤➤ **Preorder — P**erforms the operation first on the node itself, then on its left descendants, and finally on its right descendants. In other words, a node is always visited *before* any of its children.

➤➤ **Inorder — P**erforms the operation first on the node's left descendants, then on the node itself, and finally on its right descendants. In other words, the left subtree is visited first, then the node itself, and then the node's right subtree.

➤➤ **Postorder — P**erforms the operation first on the node's left descendants, then on the node's right descendants, and finally on the node itself. In other words, a node is always visited *after* all its children.

Heaps

Definition

A heap is a binary tree with these characteristics:

- ✓ It's **complete**. This means it's completely filled in, reading from left to right across each row, although the last row need not be full.
- ✓ It's (usually) implemented as an **array**.
- ✓ Each node in a heap satisfies the heap condition, which states that every node's key is larger than (or equal to) the keys of its children.

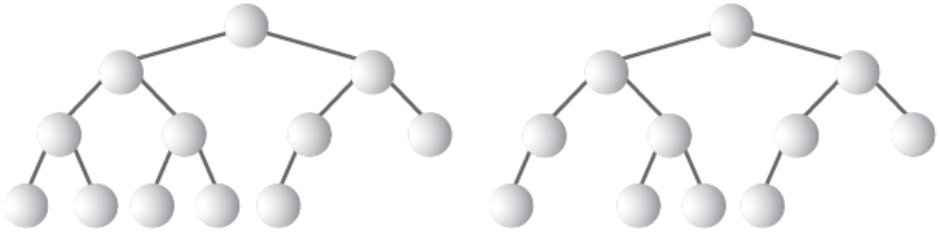


FIGURE 12.1 Complete and incomplete binary trees.

My Own Implementation in Java

```
/**
 * ClassName: Myheap
 * Description: A class that implement heap in java
 * Date: 2017-4-23
 *
 * Update History:
 *
 * @version 1.0
 * @author Steve Shao
 */
/*
 * Heap node for my own designed heap
 */
class heapnode{
    private int data;

    //constructor
    public heapnode(int data){
        this.setData(data);
    }
    public int getData() {
        return data;
```

```
    }
    public void setData(int data) {
        this.data = data;
    }
}
/*
 * Heap class
 * Heap is actually an array that has a weakly ordered completed tree
 */
class Heap{
    private heapnode[] heaparray;
    private int maxsize;
    private int currentsize;

    //constructor
    public Heap(int maxsize){
        currentsize=0;
        this.maxsize=maxsize;
        heaparray=new heapnode[maxsize];
    }

    /*
     * insert function
     * insert at the last index of an array and then trickle it up
     */
    public void insert(int data){
        heapnode newnode = new heapnode(data);
        if(currentsize==maxsize)
            throw new IllegalArgumentException("cannot insert into a full
heap");
        else{
            heaparray[currentsize]=newnode;
           trickleup(currentsize++);
        }
    }
    private void trickleup(int index) {
        int parent= (index-1)/2;
        heapnode bottom= heaparray[index];
        while(index>0&&heaparray[parent].getData()<bottom.getData()){
            heaparray[index]=heaparray[parent];
            index=parent;
            parent=(parent-1)/2;
        }
        heaparray[index]=bottom;
```

```
    }

    /*
     * delete function
     * delete the first and use the last of this array to trick it down
     */
    public heapnode delete(){
        if(currentsize==0)
            throw new IllegalArgumentException("cannot delete an empty
heap");
        else{
            heapnode root=heaparray[0];
            currentsize--;
            heaparray[0]=heaparray[currentsize];
            trickledown(0);
            return root;
        }
    }
    private void trickledown(int index) {
        int largerchild;
        heapnode top=heaparray[index];
        while(index<currentsize/2){
            int leftchild=2*index+1;
            int rightchild=leftchild+1;

    if(rightchild<currentsize&heaparray[leftchild].getData()<heaparray[rightc
hild].getData())
                largerchild=rightchild;
            else
                largerchild=leftchild;
            if(top.getData()>=heaparray[largerchild].getData())
                break;

            heaparray[index]=heaparray[largerchild];
            index=largerchild;
        }
        heaparray[index]=top;

    }
}
```

Basic API in Java

In java, **Priority queue** makes use of elegant and efficient data structure called heap.

Big O analysis (deletion and insertion are O(logN))

A heap is a special kind of binary tree, and as we saw in Chapter 8, the number of levels L in a binary tree equals log2(N+1), where N is the number of nodes. The trickleUp() and trickleDown() routines cycle through their loops L-1 times, so the first takes time proportional to log2N, and the second somewhat more because of the extra comparison. Thus, the heap operations we've talked about here all operate in O(logN) time.

Hash Tables

Definition

A hash table is a data structure that offers very fast insertion and searching.

Basic API in Java

1. Declaration and Initialization

Hashtable<String, Integer> numbers = new Hashtable<String,Integer>();

HashMap<Integer,Integer> myhashmap = new HashMap<Integer,Integer>();

2. put and get

put

```
public V put(K key,
             V value)
```

Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.

Specified by:

`put` in interface `Map<K,V>`

Overrides:

`put` in class `AbstractMap<K,V>`

Parameters:

`key` - key with which the specified value is to be associated

`value` - value to be associated with the specified key

Returns:

the previous value associated with `key`, or `null` if there was no mapping for `key`. (A `null` return can also indicate that the map previously associated `null` with `key`.)

get

```
public V get(Object key)
```

Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.

More formally, if this map contains a mapping from a key `k` to a value `v` such that `(key==null ? k==null : key.equals(k))`, then this method returns `v`; otherwise it returns `null`. (There can be at most one such mapping.)

A return value of `null` does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to `null`. The `containsKey` operation may be used to distinguish these two cases.

Specified by:

`get` in interface `Map<K,V>`

Overrides:

`get` in class `AbstractMap<K,V>`

Parameters:

`key` - the key whose associated value is to be returned

Returns:

the value to which the specified key is mapped, or `null` if this map contains no mapping for the key

See Also:

`put(Object, Object)`

3. containskey

containsKey

```
public boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key.

Specified by:

`containsKey` in interface `Map<K,V>`

Overrides:

`containsKey` in class `AbstractMap<K,V>`

Parameters:

`key` - The key whose presence in this map is to be tested

Returns:

`true` if this map contains a mapping for the specified key.

Big O analysis (deletion and insertion are O(1))

We've noted that insertion and searching in hash tables can approach O(1) time. If no collision occurs, only a call to the hash function and a single array reference are necessary to insert a new item or find an existing item. This is the minimum access time.

Difference between hashmap and hashtable

1. Hashtable is **synchronized**, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
3. One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

HashSet(No duplicate)

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

1. Declaration and Initialization

HashSet<String> myhashset = new HashSet<>();

2. add(Return Boolean)

add

```
public boolean add(E e)
```

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if this set contains no element e2 such that `(e==null ? e2==null : e.equals(e2))`. If this set already contains the element, the call leaves the set unchanged and returns `false`.

Specified by:

`add` in interface `Collection<E>`

Specified by:

`add` in interface `Set<E>`

Overrides:

`add` in class `AbstractCollection<E>`

Parameters:

e - element to be added to this set

Returns:

`true` if this set did not already contain the specified element

3. contains

contains

```
public boolean contains(Object o)
```

Returns `true` if this set contains the specified element. More formally, returns `true` if and only if this set contains an element e such that `(o==null ? e==null : o.equals(e))`.

Specified by:

`contains` in interface `Collection<E>`

Specified by:

`contains` in interface `Set<E>`

Overrides:

`contains` in class `AbstractCollection<E>`

Parameters:

o - element whose presence in this set is to be tested

Returns:

`true` if this set contains the specified element

Summary

A hash table is based on an array. The range of key values is usually greater than the size of the array. A key value is hashed to an array index by a **hash function**. The hashing of a key to an already-filled array cell is called a collision. Collisions can be handled in two major ways: **open addressing and separate chaining**. In open addressing, data items that hash to a full array cell are placed in another cell in the array. In separate chaining, each array element consists of a linked list. All data items hashing to a given array index are inserted in that list. Three kinds of open addressing: linear probing, quadratic probing, and double hashing.

Basic Sorting

Selection Sort(O(N^2), In-place and not Stable)

It starts with the first element in the array (or list) and scans through the array to find the element with the smallest key, which it swaps with the first element. The process is then repeated with each subsequent element until the last element is reached.

Java Code:

```
    public int[] selectionsort(int[] array){
        int[] orderedarray=new int[array.length];
        for(int i=0;i<array.length;i++){
            for(int j=i+1;j<array.length;j++){
                if(array[j]<array[i]){
                    int temp=array[i];
                    array[i]=array[j];
                    array[j]=temp;
                }
            }
            orderedarray[i]=array[i];
        }
        return orderedarray;
    }
```

Insertion Sort

It builds a sorted array (or list) one element at a time by comparing each new element to the already-sorted elements and inserting the new element into the correct location, similar to the way you sort a hand of playing cards.

Java Code:

```
    public static int[] insertionsort(int[] array){
        for(int which=1;which<array.length;which++){
            int val = array[which];
            for(int i=0;i<which;i++){
                if(array[i]>val){
                    System.arraycopy(array, i, array, i+1, which-i);
//public static void arraycopy(Object src, int srcPos, Object dest, int
destPos, int length)
                    array[i]=val;
                    break;
                }
            }
        }
        return array;
    }
```

Big O analysis(O(N) in best case, O(N^2) worst and average, stable and

in-place)

Unlike selection sort, the best-case running time for insertion sort is **O(N)**, which occurs when the list is already sorted. T**his means insertion sort is an efficient way to add new elements to a presorted list.** The average and worst cases are both **O(N^2)**, however, so it's not the best algorithm to use for large amounts of randomly ordered data. Insertion sort is a **stable, in-place** sorting algorithm especially suitable for sorting small data sets and is often used as a building block for other, more complicated sorting algorithms.

QuickSort

Quicksort is a divide-and-conquer algorithm that involves choosing a pivot value from a data set and splitting the set into two subsets: a set that contains all values less than the pivot and a set that contains all values greater than or equal to the pivot. The pivot/split process is recursively applied to each subset until there are no more subsets to split. The results are combined to form the final sorted set.

Java Code:

```
/*
 * quick sort class
 */
class Arrayquicksort {
    private int[] theArray;
    private int nElems;

    public Arrayquicksort(int[] theArray){
        this.theArray=theArray;
        nElems=theArray.length;
    }
    public void display() {
        for(int i=0;i<nElems;i++)
            System.out.print(theArray[i]);
    }
    /*
     * quick sort
     */
    public void quicksort(){
        requicksort(0,nElems-1);
    }

    public void requicksort(int left,int right){
        if(right-left<=0)
            return;
        else{
            int pivot = theArray[right];
```

```
            int partition = partitionIt(left,right,pivot);
            requicksort(left,partition-1);
            requicksort(partition+1,right);
        }
    }

    private int partitionIt(int left, int right, int pivot) {
        int leftPtr = left-1;
        int rightPtr = right;
        while(true){
            while(theArray[++leftPtr]<pivot)
                ;
            while(rightPtr>0&&theArray[--rightPtr]>pivot)
                ;
            if(leftPtr>=rightPtr)
                break;
            else
                swap(leftPtr,rightPtr);
        }
        swap(leftPtr,right);
        return leftPtr;
    }
    private void swap(int leftPtr, int rightPtr) {
        int temp = theArray[leftPtr];
        theArray[leftPtr]=theArray[rightPtr];
        theArray[rightPtr]=temp;
    }

}
```

Big O analysis(Average case O(NlogN), Worst O(N^2), In-place, not stable)

In the best case, the size of a sublist is halved on each successive recursive call, and the recursion terminates when the sublist size is 1. This means the number of times you operate on an element is equal to the number of times you can divide n by 2 before reaching one: log(n). Performing log(n) operations on each of n elements yields a combined best case complexity of **O(n log(n))**. In the worst case, the pivot is the *minimum* value in the data set, which means that one subset is empty and the other subset contains *n*- 1 items (all the items except for the pivot). The number of recursive calls is then *O*(*n*) (analogous to a completely unbalanced tree degrading to a linked list), which gives a combined worst-case complexity of ***O*(*n*^2)**. This is the same as selection sort or insertion sort.

MergeSort

Merge sort is another divide-and-conquer algorithm that works by splitting a data set

into two or more subsets, sorting the subsets, and then merging them together into the final sorted set.

Java Code:

```
public int[] mergeSortSimple( int[] data ){
    if( data.length < 2 ){
        return data;
    }
    // Split the array into two subarrays of approx equal size.
    int mid = data.length / 2;
    int[] left = new int[ mid ];
    int[] right = new int[ data.length - mid ];
    System.arraycopy( data, 0, left, 0, left.length );
    System.arraycopy( data, mid, right, 0, right.length );
    // Sort each subarray, then merge the result.
    mergeSortSimple( left );
    mergeSortSimple( right );
    return merge( data, left, right );
}
// Merge two smaller arrays into a larger array.
private int[] merge( int[] dest, int[] left, int[] right ){
    int dind = 0;
    int lind = 0;
    int rind = 0;
// Merge arrays while there are elements in both
    while ( lind < left.length && rind < right.length ){
        if ( left[ lind ] <= right[ rind ] ){
            dest[dind] = left[lind];
            dind++;
            lind++;
        }
        else {
            dest[dind] = right[rind];
            dind++;
            rind++;
        }
    }
    // Copy rest of whichever array remains
    while ( lind < left.length )
        dest[ dind++ ] = left[ lind++ ];
    while ( rind < right.length )
        dest[ dind++ ] = right[ rind++ ];
    return dest;
}
```

Big O analysis(O(NlogN), Not In-place, but stable)

The best, average, and worst-case running times for merge sort are all O(Nlog(N)), which is great when you need a guaranteed upper bound on the sorting time. However, merge sort requires O(N) additional memory — substantially more than many other algorithms. Typical (maximally efficient) merge sort implementations are stable but not in-place.

Summary

When facing a sorting problem, think into three aspects:

- ✓ What do we know about the data? Is the data already sorted or mostly sorted? How large are the data sets likely to be? Can there be duplicate key values?
- ✓ What are the requirements for the sort? Do you want to optimize for best-case, worst-case, or average-case performance? Does the sort need to be stable?
- ✓ What do we know about the system? Is the largest data set to be sorted smaller than, the same size as, or larger than available memory?