* The **abstract** class modifier describes a class that has abstract methods. Abstract methods are declared with the **abstract** keyword and are empty (that is, they have no block defining a body of code for this method). A class that has nothing but abstract methods and no instance variables is more properly called an interface (see Section 2.4), so an **abstract** class usually has a mixture of abstract methods and actual methods. (We discuss abstract classes and their uses in Section 2.4.)
* The **final** class modifier describes a class that can have no subclasses.
* /\*\*

This is a block comment.

\*/

* // This is an inline comment.
* A new object is dynamically allocated in memory, and all instance variables are initialized to standard default values. The default values are **null** for object variables and 0 for all base types except **boolean** variables (which are **false** by default).
* We sometimes want to store numbers as objects, but base type numbers are not themselves objects, as we have noted. To get around this obstacle, Java defines a wrapper class for each numeric base type

Byte

n = new Byte((byte)34);

n.byteValue( )

**short**

Short

n = new Short((short)100);

n.shortValue( )

**int**

Integer

n = new Integer(1045);

n.intValue( )

**long**

Long

n = new Long(10849L);

n.longValue( )

**float**

Float

n = new Float(3.934F);

n.floatValue( )

**double**

Double

n = new Double(3.934);

* + n.doubleValue( )
* Every object reference variable must refer to some object, unless it is null, in which case it points to nothing.
* Java does not allow two methods with the same signature to return different types.
* **static**: The **static** keyword is used to declare a variable that is associated with the class, not with individual instances of that class.
* **Enum**
  + 
* Static methods can also be used to change the state of static variables associated with a class (provided these variables are not declared to be **final**).
* If the return type is **void**, the method is called a ***procedure***; otherwise, it is called a ***function***
* Of course, the method can use the reference h to change the internal state of the object, and this will change g's object as well (since g and h are currently referring to the same object).
* Java functions can return only one value. To return multiple values in Java, we should instead combine all the values we wish to return in a ***compound object***
* All parameters in Java are passed by value, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body. So if we pass an int variable to a method, then that variable's integer value is copied. The method can change the copy but not the original. If we pass an object reference as a parameter to a method, then the reference is copied as well. Remember that we can have many different variables that all refer to the same object. Changing the internal reference inside a method will not change the reference that was passed in. For example, if we pass a Gnome reference g to a method that calls this parameter h, then this method can change the reference h to point to a different object, but g will still refer to the same object as before. Of course, the method can use the reference h to change the internal state of the object, and this will change g's object as well (since g and h are currently referring to the same object).
  + G -> h ->

Name = “Giri”

* + h.name = “sure” then print g.name is Suri
  + if h -> other object then G still points to name = “Giri”, now changing h will not change G ‘s attributes value
* Constructor modifiers, shown above as *modifiers*, follow the same rules as normal methods, except that an **abstract, static**, or **final** constructor is not allowed.
* A ***literal*** is any "constant" value that can be used in an assignment or other expression.
* Unlike C++, Java does not allow operator overloading.
* &&, || -> first fails, then second condition will not check

&,| - > checks all irresspective of first is pass or fail

* < < shift bits left, filling in with zeros , > > shift bits right, filling in with sign bit

>>> shift bits right, filling in with zeros

* Operators on the same line are evaluated in left-to-right order (except for assignment and prefix operations, which are evaluated right-to-left
* **int** i1 = 3;

**int** i2 = 6;

dresult = (**double**)i1 / (**double**)i2;// dresult has value 0.5

sdresult = i1 / i2; // dresult has value 0.0

* Autoboxing and unboxing
  + Consider the following code:

List<Integer> li = new ArrayList<>();

for (int i = 1; i < 50; i += 2)

li.add(i);

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because li is a list of Integer objects, not a list ofint values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates anInteger object from i and adds the object to li. Thus, the compiler converts the previous code to the following at runtime:

List<Integer> li = new ArrayList<>();

for (int i = 1; i < 50; i += 2)

li.add(Integer.valueOf(i));

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.

Consider the following method:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i: li)

if (i % 2 == 0)

sum += i;

return sum;

}

Because the remainder (%) and unary plus (+=) operators do not apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i : li)

if (i.intValue() % 2 == 0)

sum += i.intValue();

return sum;

}

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxings. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.
* switch statement, however, so flow of control "falls through" to other cases if the code for each case is not ended with a break statement (which causes control flow to jump to the next line after the switch statement).
* In for loop, *initialization, condition*, and *increment* can be empty.
* in Java, a **while** loop cannot have an empty Boolean condition, whereas a **for** loop can.
* Break statement is used to "break" out of the innermost **switch, for, while**, or **do-while** statement body.
* **break** *label;*

where *label* is a Java identifier that is used to label a loop or **switch** statement. Such a label can only appear at the beginning of the declaration of a loop. There are no other kinds of "go to" statements in Java

* break statetement breaks the innermost loop where as break with label breaks loop with label specified
* break leaves a loop, continue jumps to the next iteration
* break skips all iterations once the condition is matched, continue skips current iteration and continues the next iterations
* The fact that arrays in Java are objects has an important implication when it comes to using array names in assignment statements. For when we write something like

b = a;

in a Java program, we really mean that b and a now both refer to the same array. So, if we then write something like

b[3] = 5;

then we will also be setting the number a [3] to 5

* If instead, we wanted to create an exact copy of the array, a, and assign that array to the array variable, b, we should write

b = a.clone();

which copies all of the cells of a into a new array and assigns b to point to that new array. In fact, the clone method is a built-in method of every Java object, which makes an exact copy of that object. In this case, if we then write

b[3] = 5;

then the new (copied) array will have its cell at index 3, assigned the value 5, but a[3] will remain unchanged.

* System.in, for performing input from the Java console window. Technically, the input is actually coming from the "standard input" device, which by default is the computer keyboard echoing its characters in the Java console.
* The System.in object is an object associated with the standard input device. A simple way of reading input with this object is to use it to create a Scanner object

Scanner s = **new** Scanner(System.in);

System.out.print("Enter your height in centimeters: ");

**float** height = s.nextFloat();

hasNext(): Return **true** if and only if there is another token in the input stream.

next(): Return the next token string in the input stream; generate an error if there are no more tokens left.

hasNext*Type*(): Return **true** if and only if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.

next*Type*(): Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

hasNextLine(): Returns **true** if and only if the input stream has another line of text.

nextLine(): Advances the input past the current line ending and returns the input that was skipped.

findInLine(String s): Attempts to find a string matching the (regular expression) pattern s in the current line. If the pattern is found, it is returned and the scanner advances to the first character after this match. If the pattern is not found, the scanner returns **null** and doesn't advance.

Scanner input = **new** Scanner(System.in);

System.out.print("Please enter an integer: ");

**while** (!input.hasNextInt()) {

input. nextLine();

System.out.print("That' s not an integer; please enter an integer: ");

}

* One technical point regarding nested classes is that the nested class should be declared as **static**. This declaration implies that the nested class is associated with the outer class, not an instance of the outer class, that is, a specific object
* If we provide an **import** statement for both packages, then we must specify which class we mean as follows:

Gnomes.Mushroom shroom = **new** Gnomes.Mushroom ("purple");

Cooking.Mushroom topping = **new** Cooking.Mushroom ();

* The primary javadoc tags are the following:

@author *text*: Identifies each author (one per line) for a class.

@exception *exception-name description*: Identifies an error condition that is signaled

by this method (see Section 2.3).

@param *parameter-name description*: Identifies a parameter accepted by this method.

@return *description*: Describes the return type and its range of values for a

method.

* Dynami dipatch – finding method from subclass to superclass.
* Specialization -> subclass has either no methods or overrideen methods of superclass
* Extension -> subclass having its own methods
* Replacement -> completely overrides method
* Refinement -> adds additional code in sublass, acheved in constructor by using super().



When this program is executed, it prints the following:

The dog local variable =5.0

The dog field = 2

* if we try to delete the tenth element from a sequence that has only five elements, the code may throw a BoundaryViolationException.
* Note that exceptions that are not subclasses of RuntimeException must be declared in the **throws** clause of any method that can throw them
* Another legitimate way of handling exceptions is to create and throw another exception, possibly one that specifies the exceptional condition more precisely. The following is an example of this approach:

**catch** (ArrayIndexOutOfBoundsException aioobx) {

**throw new** ShoppingListTooSmallException(

"Product index is not in the shopping list");

}

* In Java, multiple inheritance is allowed for interfaces but not for classes.
* Interfaceextends one or more interfaces

**public interface** InsurableItem **extends** Transportable, Sellable {

/\*\* Returns insured Value in cents \*/

**public int** insuredValue();

}

* Like an interface, an abstract class may not be instantiated, that is, no object can be created from an abstract class.
* Abstract class method can be called by creating subclass object which extends. Because Abstarct class object cannot be created and extend means superclass(abstract class) methods are visible in subclass, create subclass object and call the abstract class method
* Use **try-catch** blocks every time we perform a cast, Java provides a way to make sure an object cast will be correct. Namely, it provides an operator, **instanceof**, that allows us to test whether an object variable is referring to an object of a certain class (or implementing a certain interface).
* Widening(implicit) conversion -> assigning subclass object to superclass object, here no explicit type cast required.

Narrowing(explicit) conversion -> assigning Superclass object to subclass object, here explicit conversion required.

* == returns true if two objects points to same memory locations, equals returns true if two object contents are same.
* String obj1 = new String("xyz");
* String obj2 = new String("xyz");
* if(obj1 == obj2)
* System.out.println("obj1==obj2 is TRUE");
* else
* System.out.println("obj1==obj2 is FALSE");

obj1==obj2 is FALSE

String obj1 = new String("xyz");

// now obj2 and obj1 reference the same place in memory

String obj2 = obj1;

if(obj1 == obj2)

System.out.printlln("obj1==obj2 is TRUE");

else

System.out.println("obj1==obj2 is FALSE");

obj1==obj2 is TRUE

String obj1 = new String("xyz");

String obj2 = new String("xyz");

if(obj1.equals(obj2))

System.out.printlln("obj1==obj2 is TRUE");

else

System.out.println("obj1==obj2 is FALSE");

obj1==obj2 is TRUE

* Compareto
* public int compareTo( NumberSubClass referenceName )
* If the Integer is equal to the argument then 0 is returned.
* If the Integer is less than the argument then -1 is returned.
* If the Integer is greater than the argument then 1 is returned.
* public class Test{
* public static void main(String args[]){
* Integer x = 5;
* System.out.println(x.compareTo(3));
* System.out.println(x.compareTo(5));
* System.out.println(x.compareTo(8));
* }
* }

1

0

-1

* Generics
  + All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
  + Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
* public class GenericMethodTest
* {
* // generic method printArray
* public static < E > void printArray( E[] inputArray )
* {
* // Display array elements
* for ( E element : inputArray ){
* System.out.printf( "%s ", element );
* }
* System.out.println();
* }
* public static void main( String args[] )
* {
* // Create arrays of Integer, Double and Character
* Integer[] intArray = { 1, 2, 3, 4, 5 };
* Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
* Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
* System.out.println( "Array integerArray contains:" );
* printArray( intArray ); // pass an Integer array
* System.out.println( "\nArray doubleArray contains:" );
* printArray( doubleArray ); // pass a Double array
* System.out.println( "\nArray characterArray contains:" );
* printArray( charArray ); // pass a Character array
* }
* }
  + Bounded Type Parameters:
    - There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
    - To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

public class MaximumTest

{

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z)

{

T max = x; // assume x is initially the largest

if ( y.compareTo( max ) > 0 ){

max = y; // y is the largest so far

}

if ( z.compareTo( max ) > 0 ){

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main( String args[] )

{

System.out.printf( "Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ) );

System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

System.out.printf( "Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum( "pear", "apple", "orange" ) );

}

}

## Generic Classes:

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

* Fouth Chapter(Recursion,stacks,queues,single linked list,double linked list)
  + Linear Recursion(each time atmost one recursive call)
    - Sum of array elements



* + - Swapping array elements



* + - Computation of power of x



* + - Try above three examples without using recursion
  + Binary Recursion(2 calls each time)
    - Sum of array elements where n is multiple of 2



* + - Fibanacci



* + - Check how to check number is multiple of 2
    - Solve fibonacci using linear recursion
  + Mutliple recursion



* + - For above example, try writing the code which accepts abc as array elements and prints all combination of abc
* Stacks
  + Push(Object e).pop(),top(),isEmpty(),size()
  + Stack interface and its implementation using array
  + Ex: Reverse the array elements, JVM, Java methods call,arithmatic problem solve,Recusrion, Paranthesis matching,html tag matching
  + Code examples : Reverse array, Parenthesis match, Html tag match
* Queues
  + Enqueue(Object e), Dequeue(), size(), isEmpty()
  + Queue interface and its implemetation using array
  + Circular queue interface and its implementation
    - Modulo usgae
  + Ex: Round robin schedular, memory alloocation,Josephs problem solve, Thread execution
  + Code examples : Josephs problem solve
* Single linked list
  + Single linked interface(head)
  + Stack implementation using linked list
  + Queue implementaion using linked list
* Double linked list
  + removeFirst(),removeLast(), insertFirst(Object e),insertLast(Object e), first(), last(),size(), isempty()
  + Double linked list interface and implememtaions

**New Book**

* **Procedural programming draw back :** when several functions needed to access the same data. To be available to more than one function, such variables had to be global, but global data

could be accessed inadvertently by any function in the program. This lead to frequent

programming errors.

* **Why class is required :** You might think that the idea of an object would be enough for one programming revolution, but there's more. Early on, it was realized that you might want to make several

objects of the same type.

* int intVar; // an int variable called intVar

BankAccount bc1; // reference to a BankAccount object

In the first statement, a memory location called intVar actually holds a numerical value

such as 127 (assuming such a value has been placed there). However, the memory

location bc1 does not hold the data of a BankAccount object. Instead, it contains the

*address* of a BankAccount object that is actually stored elsewhere in memory.

he array is stored at an address elsewhere in memory

* there are no overloaded operators in Java.
* println() method as the last

statement in a series to actually display everything. It causes the contents of the buffer to

be transferred to the display:

system.out.flush() to cause the buffer to be displayed withoutb going to a new line:

System.out.print("Enter your name: ");

System.out.flush();

* **console read :**

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

* Suppose you want your program's user to enter a character.

Therefore, the safest way to read a character involves

reading a String and picking off its first character with the charAt() method:

public static char getChar() throws IOException

{

String s = getString();

return s.charAt(0);

}

* The difference is that in the ordered array, the search quits if an item with a

larger key is found

* **Binary serach find method**

public int find(double searchKey)

{

int lowerBound = 0;

int upperBound = nElems-1;

int curIn;

while(true)

{

curIn = (lowerBound + upperBound ) / 2;

if(a[curIn]==searchKey)

return curIn; // found it

else if(lowerBound > upperBound)

return nElems; // can't find it

else // divide range

{

if(a[curIn] < searchKey)

lowerBound = curIn + 1; // it's in upper half

else

upperBound = curIn - 1; // it's in lower half

} // end else divide range

} // end while

} // end find()

* Ordered arrays are therefore useful in situations in which searches are frequent, but

insertions and deletions are not.

* log10(100) = 2, so log2(100) = 2 times 3.322, or 6.644. Rounded up to the whole number

7

* linear search times are proportional to the size of the array : T = N
* Binary search : T = log2(N)
  + Where N is array size

Linear search O(N)

Binary search O(log N)

Insertion in unordered array O(1)

Insertion in ordered array O(N)

Deletion in unordered array O(N)

Deletion in ordered array O(N)

* O(1) is excellent, O(log N) is good, O(N) is fair, and O(N e2) is poor. O(N e2) occurs in the

bubble sort

* why not use arrays for all data storage? We've already

seen some of their disadvantages. In an unordered array you can insert items quickly, in

O(1) time, but searching takes slow O(N) time. In an ordered array you can search

quickly, in O(logN) time, but insertion takes O(N) time. For both kinds of arrays, deletion

takes O(N) time, because half the items (on the average) must be moved to fill in the

hole.

It would be nice if there were data structures that could do everything—insertion,

deletion, and searching—quickly, ideally in O(1) time, but if not that, then in O(logN) time.

Another problem with arrays is that their size is fixed

* Bubble sort :
  + Outerloop backward directions and inner loop in forward direction
  + Each neighour elements compared and swapped.
  + Inner loop, loop until outer loop index
  + Its slowest
  + Here are the rules you're following:

1. Compare two players.

2. If the one on the left is taller, swap them.

3. Move one position right.

* + In each loop, one tallest person will be sorted
    - First loop one person
    - Second loop two persons and so on
  + public void bubbleSort()

{

int out, in;

for(out=nElems-1; out>1; out--) // outer loop

(backward)

for(in=0; in<out; in++) // inner loop (forward)

if( a[in] > a[in+1] ) // out of order?

swap(in, in+1); // swap them

} // end bubbleSort()

}

}

//-------------------------------------------------------------

-

private void swap(int one, int two)

{

double temp = a[one];

a[one] = a[two];

a[two] = temp;

}

* + bubble sort runs in O(N2)
* Selection sort
  + Both outer and inner are looped in forward directions.
  + In each iteration, shortest value stored in min and replaced with starting index
  + Outter loop, from front to end
  + Inner loop, start at outer loop index to avoid sorted value condiderations as loop proceeds
  + this algorithm the sorted players accumulate

on the left (lower indices), while in the bubble sort they accumulated on the right.

* + public void selectionSort()

{

int out, in, min;

for(out=0; out<nElems-1; out++) // outer loop

{

min = out; // minimum

for(in=out+1; in<nElems; in++) // inner loop

{

if(a[in] < a[min] ) // if min greater,

min = in; // we have a new min

}

swap(out, min); // swap them

} // end for(outer)

} // end selectionSort()

* + selection sort runs in O(N\*N) time, just as the bubble sort did. However, it is unquestionably

faster because there are so few swaps.

* Insertion sort
  + partial sorting did not take place in the bubble sort and selection sort. In these

algorithms a group of data items was completely sorted at any given time; in the insertion

sort a group of items is only partially sorted

* + In the outer for loop, out starts at 1 and moves right. It marks the leftmost unsorted

data. In the inner while loop, in starts at out and moves left, until either temp is

smaller than the array element there, or it can't go left any further. Each pass through the

while loop shifts another sorted element one space right.

* + public void insertionSort()

{

int in, out;

for(out=1; out<nElems; out++) // out is dividing line

{

double temp = a[out]; // remove marked item

in = out; // start shifts at out

while(in>0 && a[in-1] >= temp) // until one is smaller,

{

a[in] = a[in-1]; // shift item right,

--in; // go left one position

}

a[in] = temp; // insert marked item

} // end for

} // end insertionSort()

* a[in-1].getLast().compareTo(temp.getLast()) > 0

The compareTo() method returns different integer values depending on the

lexicographical (that is, alphabetical) ordering of the String for which it's invoked and

the String passed to it as an argument, as shown in Table 3.1.

s1 < s2 < 0

s1 equals s2 0

s1 > s2 > 0

For example, if s1 is "cat" and s2 is "dog", the function will return a number less than

0. In the program this method is used to compare the last name of a[in-1] with the

last name of temp.

* **Example to try : insertion, deletion, search(binary search) for primitives and objects**
* **Bubble sort,selection sort, insertion sort**
* Satcks
  + Stacks general int array example
  + Reverse word (stack by character array)
  + Brackets
  + O(1) – push and pop operations irrespective of size, because at a time one element or pushed or poped
  + Stack methods : push(value), pop(), peek() -> top element without removing, ieEmpty(),isFull(),size(),display()
* Queue
  + Queue methods : insert(value), remove,peekFront(), isEmpty(),isFull(), size(),display()
  + Dequeue methods : insertFront(), insertRear(),removeFront(),removeRear()
  + Queue general example and circular queue example implementation by using modulo operator(browse the code)
  + Circular queue
    - 
* Priority queue
  + Insertion occurs by sorting it
  + Insertion time : 0(N) and deletion: O(1)
  + Code example
* Infix to postfix expression
  + You can't evaluate the 3+4 until you see what operator follows the 4. If it's a \* or / you

need to wait before applying the + sign until you've evaluated the \* or /.

* + Tried examle

a+b\*(c-a)/d

for all characters

swict(ch)

a

output = a

stack =

+

output = a

stack = +

b

output = ab

stack = +

\*

output = ab opthis = \*, optop = +, prec1 = 2, prec2 = 1

stack = +\*

(

output = ab

stack = +\*(

c

output = abc

stack = +\*(

-

output = abc opthis = -, optop (

stack = +\*(-

a

output = abca

stack = +\*(-

)

output = abca opthis ) optop -

output = abca-

stack = +\*(

/

output = abca- opthis / optop (

stack = +\*(/

d

output = abca-d

no more

output = abca-d/\*+

a+b\*(c-a)/d = abca-d/\*+

* Postfix expression evalution
  + Operands push to stack
  + When operator found, pop top two values, perform operator operation on it and push the result to stack
  + At the end, pop the final result
* **Example to try : Stack, queue (circular queue) implementation, Priority queue implementation**
* **Reverse a word, brackets verification using stacks**
* **Infix to postfix and postfix evalution expression**
* Linked list
  + Link class with two data attrributes and one Link object. Methods paramterised constructor, displayLink
  + LinkList class, first Link object, methods -> default constructor with first to null, isEmpty,

insertFirst,deleteFirst,display,insertAfter(key), delete(key)

* Double ended list
  + LinkList class has two link objects -> first,last
  + Methods : InsertLast, DeleteLast(), DeleteFist, InsertFirst(involves setting last also)
  + Link list insertionfirst,deletefirst takes O(1) and serach , find, delete particular time takes O(N)
* Abstract data types
  + What is an ADT? Roughly speaking, it's a way of looking at

a data structure: focusing on what it does, and ignoring how it does it.

* + Stack by linked list : theList.insertFirst(data) , data = theList.deleteFirst()
  + Stack by double linked list : theList.insertLast(data) , data = theList.deleteFirst()
  + One consideration is how accurately you can predict the amount of data

the stack or queue will need to hold(use arrays) . If this isn't clear, the linked list gives you

more flexibility than an array. Both are fast, so that's probably not a major consideration.

* + In object-oriented programming, then, an abstract data type is a class considered without

regard to its implementation. It's a description of the data in the class (fields), a list of

operations (methods) that can be carried out on that data, and instructions on how to use

these operations. Specifically excluded are the details of how the methods carry out their

tasks.

* + An ADT specification is often called an *interface*.
* Sorted single linked list
  + Currnt not equal to null and curr data nt greatrer than key, move the current and previous), after that check if no element and make a link change
  + O(N2)
  + Pass the unsorted array to list and pass each value to insert to list by sorting order

, store back to array by calling remove .So that array will be in sorted order

* Double linked list (double ended double linked list)
  + Link have data attribute, two link objects(previous and next)
  + DoubleEndedDoubleLinkList methods : isEmpty,insertFirst,insertLast,insertAfter(key),insertBefore(key), deleteFirst,deleteLast,deletekey(key),displayBackward(),displayForward()
* **Example to try : DoubleEndedSingleList(methods : insertFirst(),insertLast(),insertAfter(key),deleteFirst(),deleteKey(key),insertbySort(),isEmpty(),display()**
* **Example to try : DoubleEndedDoubleList(methods : insertFirst(),insertLast(),insertAfter(key),insertBefore(),deleteFirst(),deleteLast(),deleteKey(key),displayForward(),displayBackward(), isEmpty()**
* Iterators
  + Objects containing *references* to items in data structures, used to traverse data

structures, are commonly called *iterators* (or sometimes, as in certain Java classes,

*enumerators*).

* DoubleEndedSingleLinkList three class data structure and DoubleEndedDoubleLinkList class data structure

*DoubleEndedSingleLinkList*

*Private SingleLinkData first;*

*Private SingleLinkData next ;*

*Constructor : DoubleEndedSingleLinkList()*

*Methods : isEmpty()*

*insertFirst(value)*

*insertLast(value)*

*insertAfter(existvalue, value)*

*insertBySort(value)*

*deleteFirst()*

*deleteKey(value)*

*display()*

*SingleLinkData*

*Public int data;*

*Public SingleLinkData next;*

*Constructor : SingleLinkData(int valure)*

*Methods : display()*

*DoubleEndedSingleLinkList*

*Private SingleLinkData first;*

*Private SingleLinkData next ;*

*Constructor : DoubleEndedSingleLinkList()*

*Methods : isEmpty()*

*insertFirst(value)*

*insertLast(value)*

*insertAfter(existvalue, value)*

*insertBySort(value)*

*deleteFirst()*

*deleteKey(value)*

*display()*

*SingleLinkData*

*Public int data;*

*Public SingleLinkData next;*

*Constructor : SingleLinkData(int valure)*

*Methods : display()*

*DoubleLinkData*

*Public SingleLinkData previous;*

*Public int data;*

*Public SingleLinkData next;*

*Constructor : DoubleLinkData(int valure)*

*Methods : display()*

*DoubleEndedDoubleLinkList*

*Private DoubleLinkData first;*

*Private DoubleLinkData next ;*

*Constructor : DoubleEndedSingleLinkList()*

*Methods : isEmpty()*

*insertFirst(value)*

*insertLast(value)*

*insertAfter(existvalue, value)*

*insertBefore(existvalue, value)*

*deleteFirst()*

*deleteLast()*

*deleteKey(value)*

*displayForward()*

*displayBackward()*

* Recursion
  + Factorial

int factorial(int n)

{

if(n==0)

return 1;

else

return (n \* factorial(n-1) );

}

* + Combination of letters ex : abc

void comb(int[] arr) {

Display(arr);

for(int i=0;i<fact(arr.length)-1;i++) {

for(j=i+1;j>=0;j--) {

swap(arr[j-1],arr[j])

display(arr)

}

}

}

* + Tower of annoi (from to inner, right shift => inner from to)
    - From inter to
      * From to inter
      * Inter from to
    - **public class**MainClass {  
        **public static void**main(String[] args) {  
          **int**nDisks = 3;  
          doTowers(nDisks, 'A', 'B', 'C');  
        }  
        
        **public static void**doTowers(**int**topN, **char**from, **char**inter, **char**to) {  
          **if**(topN == 1){  
            System.out.println("Disk 1 from " + **from**+ " to " + to);  
          }**else**{  
            doTowers(topN - 1, from, to, inter);  
            System.out.println("Disk " + topN + " from " + **from**+ " to " + to);  
            doTowers(topN - 1, inter, from, to);  
          }  
        }  
      }
  + Merge sort
    - Merging two sorted array elements to other array with sorting
    - While the bubble, insertion, and selection sorts take O(N2) time, the mergesort is

O(N\*logN).

public static void main(String[] args)

{

int[] arrayA = {23, 47, 81, 95};

int[] arrayB = {7, 14, 39, 55, 62, 74};

int[] arrayC = new int[10];

merge(arrayA, 4, arrayB, 6, arrayC);

display(arrayC, 10);

} // end main()

//----------------------------------------------------------

-

// merge A and B into C

public static void merge( int[] arrayA, int sizeA,

int[] arrayB, int sizeB,

int[] arrayC )

{

int aDex=0, bDex=0, cDex=0;

while(aDex < sizeA && bDex < sizeB) // neither array empty

{

if( arrayA[aDex] < arrayB[bDex] )

arrayC[cDex++] = arrayA[aDex++];

else

arrayC[cDex++] = arrayB[bDex++];

}

while(aDex < sizeA) // arrayB is empty,

arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn't

while(bDex < sizeB) // arrayA is empty,

arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn't

} // end merge()

//----------------------------------------------------------

-

-

* **Example to try : Factorial, tower of annoi, merge sort**
* Advance sorting
  + Shell sort
  + O(N\*(logN)2)
  + Shell sort is same like insertion sort, shift happens in interval, not like one interval by h\*3=1 formula.
  + This helps if arrays are partila sorted in between. It avoids left shift
    - Sequence : h = 3\*h + 1
    - h = (h–1) / 3

//-------------------------------------------------------------

-

public void shellSort()

{

int inner, outer;

double temp;

int h = 1; // find initial value of h

while(h <= nElems/3)

h = h\*3 + 1; // (1, 4, 13, 40, 121,...)

while(h>0) // decreasing h, until h=1

{

// h-sort the file

for(outer=h; outer<nElems; outer++)

{

temp = theArray[outer];

inner = outer;

// one subpass (eg 0, 4,8)

while(inner > h-1 && theArray[inner-h] >= temp)

{

theArray[inner] = theArray[inner-h];

inner -= h;

}

theArray[inner] = temp;

} // end for

h = (h-1) / 3; // decrease h

} // end while(h>0)

} // end shellSort()

* + - //-------------------------------------------------------------
  + Particioning
    - Chose the pivot value(say 99)
    - Left pointer at startng index and right pointer at end
    - Increase left pointer and decrease right pointer until left >= right
      * In loops , swap the left ponter number if its less than 99 with right parameter value .
      * At the end, two halfs one half with greater than pivot and other half with less than pivot

public int partitionIt(int left, int right, double pivot)

{

int leftPtr = left - 1; // right of first elem

int rightPtr = right + 1; // left of pivot

while(true)

{

while(leftPtr < right && // find bigger item theArray[++leftPtr] < pivot)

; // (nop)

while(rightPtr > left && // find smaller itemtheArray[--rightPtr] > pivot)

; // (nop)

if(leftPtr >= rightPtr) // if pointers cross,

break; // partition done

else // not crossed, so

swap(leftPtr, rightPtr); // swap elements

} // end while(true)

return leftPtr; // return partition

} // end partitionIt()

* + Quick sort
    - Works based on particion logic
    - Selects right most value as pivot value
    - O(n\*logn)

public void recQuickSort(int left, int right)

{

if(right-left <= 0) // if size <= 1,

return; // already sorted

else // size is 2 or larger

{

double pivot = theArray[right]; // rightmost item

// partition range

int partition = partitionIt(left, right, pivot);

recQuickSort(left, partition-1); // sort left side

recQuickSort(partition+1, right); // sort right side

}

} // end recQuickSort()

* + - Selecting pivot has right most is bad idea



public void recQuickSort(int left, int right)

{

int size = right-left+1;

if(size <= 3) // manual sort if small

insertionSort(left, right); //for less than three elements

else // quicksort if large

{

double median = medianOf3(left, right);

int partition = partitionIt(left, right, median);

recQuickSort(left, partition-1);

recQuickSort(partition+1, right);

}

} // end recQuickSort()

public double medianOf3(int left, int right)

{

int center = (left+right)/2;

// order left & center

if( theArray[left] > theArray[center] )

swap(left, center);

// order left & right

if( theArray[left] > theArray[right] )

swap(left, right);

// order center & right

if( theArray[center] > theArray[right] )

swap(center, right);

swap(center, right-1); // put pivot on right

return theArray[right-1]; // return median value

} // end medianOf3()

public int partitionIt(int left, int right, double pivot)

{

int leftPtr = left; // right of first elem

int rightPtr = right - 1; // left of pivot

while(true)

{

while(theArray[++leftPtr] < pivot) // find bigger

; // (nop)

while(theArray[--rightPtr] > pivot) // find smaller

; // (nop)

if(leftPtr >= rightPtr) // if pointers cross,

break; // partition done

else // not crossed, so

swap(leftPtr, rightPtr); // swap elements

} // end while(true)

swap(leftPtr, right-1); // restore pivot

//\*\*\*

return leftPtr; // return pivot location

} // end partitionIt()

* **Example to try : Quick sort**
* Binary Tree
  + Ordered array are slow insertion and deletion
  + Linked list are slow in searcing
  + It would be nice if there were a data structure with the quick insertion and deletion of a

linked list, and also the quick searching of an ordered array. Trees provide both these

characteristics, and are also one of the most interesting data structures.

* + Binary tree search takes O(log2N)
  + Binary tree orders
    - Inorder

1.Call itself to traverse the node's left subtree

2. Visit the node

3. Call itself to traverse the node's right subtree

private void inOrder(node localRoot)

{

if(localRoot != null)

{

inOrder(localRoot.leftChild);

localRoot.displayNode();

inOrder(localRoot.rightChild);

}

}

* + - Preorder

1. Visit the node

2. Call itself to traverse the node's left subtree

3. Call itself to traverse the node's right subtree

* + - Postorder

1. Call itself to traverse the node's left subtree

2. Call itself to traverse the node's right subtree

3. Visit the node

* + - Finding Successor of the node

First the program goes to the original node's right child, which must have a

key larger than the node. Then it goes to this right child's left child (if it has

one), and to this left child's left child, and so on, following down the path of left children. The last left child in this path is the successor of the original node,



* + - Binary tree takes O(logN) time.(because it involves number of level(first level – 1 node, second = 3, 3rd = 7, 4th = 17..so on..
* **All this try with picture example for each functions and try the code**

insertion

================

.create a newnode

.if root is null, then root is newnode

.else current = root

while true

{

if(less) {

current to left child

if(current null) { attach to left node and return }

else {

current to right child

if(current null) { attach to right child and return }

}

}

find

==========

while(current.data != key) {

if(less) {

move to left

}

else {

move to right

}

if(current null) {

return null

}

}

return current;

traversals(recursion code)

==========================

preorder

-----------

private void inOrder(node localRoot)

{

if(localRoot != null)

{

localRoot.displayNode();

inOrder(localRoot.leftChild);

inOrder(localRoot.rightChild);

}

}

Inorder(displays the sorted elements in ascending order)

-----------

private void inOrder(node localRoot)

{

if(localRoot != null)

{

inOrder(localRoot.leftChild);

localRoot.displayNode();

inOrder(localRoot.rightChild);

}

}

postorder

-----------

private void inOrder(node localRoot)

{

if(localRoot != null)

{

inOrder(localRoot.leftChild);

inOrder(localRoot.rightChild);

localRoot.displayNode();

}

}

finding maximum

==============

while current != null {

move current to right

}

return current

finding minimum

==============

while current != null {

move current to left

}

return current

traversals(recursion code)

==========================

Node to be delated which has no children

----------------------------------------------------



Node to be delated which has one children

----------------------------------------------------



Node to be delated which has two children

----------------------------------------------------

Involves finding sucessor : successor is next immediate highest number of deleted element in tree (simple : next element can be found by using inorder traversal, as it gives ellemnts of tree in sorted order)





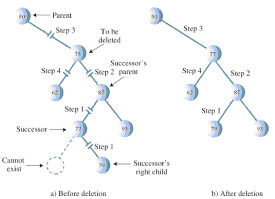
Logic : go to delete node right child first, traverse left from there until no more lement.

Last element is succesor.

Delete when succesor is right child of deleted element



Delete when succesor is left child of deleted element



No

* **Example to try : Binary tree : find(int value), insert(int value, doble depth), preOrder(Node localRoot), inOrder(Node localRoot), postOrder(Node localRoot), min(), max()**
* **deleting a node (which has no children, which has one children(either right or left),which has two children(involves finding successor) – remember the logic and diagrams, how to do it, not how to code it. Its difficult**

**Red Black Trees**

* It's important to keep a binary search tree balanced to ensure that the time necessary to find a given node is kept as short as possible.
* Inserting data that has already been sorted can create a maximally unbalanced tree, which will have search times of O(N).
* In the red-black balancing scheme, each node is given a new characteristic: a color that can be either red or black. It always provides balance tree even for elements which are sorted.
* A color flip changes a black node with two red children to a red node with two black children.
* In a rotation, one node is designated the top node.
* A right rotation moves the top node into the position of its right child, and the top

node's left child into its position.

* A left rotation moves the top node into the position of its left child, and the top node's right child into its position.

•

* Color flips, and sometimes rotations, are applied while searching down the tree to find where a new node should be inserted. These flips simplify returning the tree to redblack correctness following an insertion.
* After a new node is inserted, red-red conflicts are checked again. If a violation is

found, appropriate rotations are carried out to make the tree red-black correct.

* These adjustments result in the tree being balanced, or at least almost balanced.
* For sorted elements, binary tree search takes 0(N) where as for RBT takes O(logN)

**Red-Black Rules**

1. Every node is either red or black.

2. The root is always black.

3. If a node is red, its children must be black (although the converse isn't necessarily

true).

4. Every path from the root to a leaf, or to a null child, must contain the same number

Of black nodes.

Insertion involves colour change and Rotation(Rotaion of right(ROR) or rotaion

of left(ROL)

Colour Change

* Root node colour is always black
* During insertion, parent colour has to be block, if not change the parent colour to block provided block node count from root on both the side has to be same. If not perform rotatiton and change colour.

Rotation

* Performed always from root node
* Outside grandchild(1 rotaion required)
* Inside grandchild (2 rotaion required )
* For a left rotation, the top node must have a right child. For a right rotation, the top node must have a left child.
* The inside grandchild, if it's the child of the node that's going up (which is the left child of the top node in a right rotation) is always disconnected from its parent and reconnected to its former grandparent as a left child if its right child of parent or as a right child if its left child of parent(first fighure, grand child is only node, second figure grand child is subtree, rules same for both. Instead of child, complete sub tree moves)





* Three possible cases during insertion(P parent, G grand parent, X current elment)
  + P is black.
  + P is red and X is an outside grandchild of G.
  + P is red and X is an inside grandchild of G.

**Other Balanced Trees**

* AVL(Adelson-Velskii and Landis) tree : In AVL trees each node stores an additional piece of data: the difference between the heights of its left and right subtrees
* The other important kind of balanced tree is the *multiway tree*, in which each node can have more than two children. (2-3-4 tree)
* Solve the Red block tree examples by inserting many items(check in net where all colour flip, rotation we do), outside grand child, inside grandchild

**2-3-4 Trees**

* A node with one data item always has two children
* A node with two data items always has three children
* A node with three data items always has four children













* A node with one data item must always have two links, unless it's a leaf,

RB v/s 2-3-4 Trees

* In a 2-3-4 tree the tree is kept balanced using node splits.
* In a red-black tree the two balancing methods are color flips and rotations
* RB tree has more levels than 2-3-4 tree
* The height of a 2-3-4 tree would be about half that of a red-black tree, provided that all the nodes were full. Also, every node contains the maximum number of data items, which is 1. This makes red-black trees more efficient than 2-3-4 trees in terms of memory usage.

B- Tree

* Will trees work with files?

They will, but a different kind of tree must be used for external data than for in-memory data. The appropriate tree is a multiway tree somewhat like a 2-3-4 tree, but with many more data items per node; it's called a B-tree. B-trees were first conceived as appropriate structures for external storage by R. Bayer and E. M. McCreight in 1972

* For a tree stored in a disk file, the links are block numbers in a file
* In fact, the structure of a B-tree is similar to that of a 2-3-4 tree, except that there are more data items per node and more links to children.
* The insertion process differs from that of 2-3-4 trees in three ways:
  + A node split divides the data items equally: half go to the newly created node, and half remain in the old one.
  + Node splits are performed from the bottom up rather than from the top down.
  + It's not the middle item in a node that's promoted upward, but the middle item in the sequence formed from the items in the node plus the new item.

20, 40, 70, 80

60

10,30

15

75 85 90

25 35 50

22 27

29

case 1: 20 40 70 80

case 2(60) : 20 40 60 70 80

60

20 40 70 80

case 3(10 30) : 60

10 20 30 40 70 80

case 4(15): 60

10 15 20 30 40 70 80

20 60

10 15 30 40 70 80

case 5(75 85 90): 20 60

10 15 30 40 70 75 80 85 90

20 60 80

10 15 30 40 70 75 85 90

case 6(25 35 50) 20 60 80

10 15 25 30 35 40 50 70 75 85 90

20 35 60 80

10 15 25 30 40 50 70 75 85 90

case 7(22 27) 20 35 60 80

10 15 22 25 27 30 40 5 70 75 85 90

case 8 (29) 20 35 60 80

10 15 22 25 27 29 30 40 50 70 75 85 90

20 27 35 60 80

10 15 22 25 29 30 40 50 70 75 85 90

35

20 27 60 80

10 15 22 25 29 30 40 50 70 75 85 90

* Indexing in B trees : Note that we don't need to move any records in the main file; we simply append the new record at the end of the file
* Mergesort is the preferred algorithm for sorting external data.
* To transform a 2-3-4 tree into a red-black tree, make each 2-node into a black node,

make each 3-node into a black parent with a red child, and make each 4-node into a

black parent with two red children.

* The 2-3-4 tree wastes space because many nodes are not even half full.
* Data in external storage is typically transferred to and from main memory a block at a

time.

* Solve the 2-3-4 examples by inserting many items(split from the root, so all leaves are on same level), browse more to undersgtand implementaions, check the code snippet and understand
* Look for B tree examples how to form a B tree, browse B tree implemetaions

Compare B tree with Sequential file, indexing in B trees, Merge sort of external data

Hashing

* No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) in Big O notation.
* Hash tables do have several disadvantages. They're based on arrays, and arrays are difficult to expand once they've been created. For some kinds of hash tables, performance may degrade catastrophically when the table becomes too full, so the programmer needs to have a fairly accurate idea of how many data items will need to be stored (or be prepared to periodically transfer data to a larger hash table, a timeconsuming process). Also, there's no convenient way to visit the items in a hash table in any kind of order (such as from smallest to largest). If you need this capability, you'll need to look elsewhere. However, if you don't need to visit items in order, and you can predict in advance the size of your database, hash tables are unparalleled in speed and convenience.
* word cats to a number. We convert the digits to numbers as shown earlier. Then we multiply each number by the appropriate power of 27, and add the results:

3\*27pow3 + 1\*27pow2 + 20\*27pow1 + 19\*27pow0

This gives a unique number for every word.

The largest 10-letter word, zzzzzzzzzz, translates into

26\*27pow9 + 26\*27pow8 + 26\*27pow7 + 26\*27pow6 + 26\*27pow5 + 26\*27pow4 + 26\*27pow3 + 26\*27 pow 2 + 26\*27 pow 1 +26\*27 pow 0

Just by itself, 279 is more than 7,000,000,000,000, so you can see that the sum will be

huge. An array stored in memory can't possibly have this many elements.

* Hash Function
  + arrayIndex = hugeNumber % arraySize;
* We convert a word into a huge number by multiplying each character in the

word by an appropriate power of 27.

hugeNumber = ch0\*27pow9 + ch1\*27pow8 + ch2\*27pow7 +

ch3\*27pow6 + ch4\*27pow5 + ch5\*27pow4 + ch6\*27pow3 + ch7\*27pow2 + ch8\*27pow1 + ch9\*27pow0

Then, using the modulo (%) operator, we squeeze the resulting huge range of numbers

into a range about twice as big as the number of items we want to store. This is an

example of a hash function:

arraySize = numberWords \* 2;

arrayIndex = hugeNumber % arraySize;

* Perhaps you want to insert the word melioration into the array. You hash the word to

obtain its index number, but find that the cell at that number is already occupied by the

word demystify, which happens to hash to the exact same number (for a certain size

array). This situation, shown in Figure 11.4, is called a collision.

* when a collision occurs, is to search the array in some systematic way for an empty cell, and insert the new item there, instead of at the index specified by the hash function. This approach is called open addressing.
* A second approach (mentioned earlier) is to create an array that consists of linked lists of words instead of the words themselves. Then when a collision occurs, the new item is simply inserted in the list at that index. This is called separate chaining.
* We'll explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are linear probing, quadratic probing, and double hashing.
* **Linear Probing**

In linear probing we search sequentially for vacant cells. If 5,421 is occupied when we try to insert cats there, we go to 5,422, then 5,423, and so on, incrementing the index until we find an empty in a general-purpose hash table,

the array size should be a prime number, so 59 would be a better choice.

Following a collision, the Find algorithm simply steps along the array looking at each cell in sequence. If it encounters an empty cell before finding the key it's looking for, it knows the search has failed.

The Del Button The Del button deletes an item whose key is typed by the user. Deletion isn't accomplished by simply removing a data item from a cell, leaving it empty. Why not? Remember that during insertion the probe process steps along a series of cells, looking for a vacant one. If a cell is made empty in the middle of this sequence of full cells, the Find routine will give up when it sees the empty cell, even if the desired cell can eventually be reached.

For this reason a deleted item is replaced by an item with a special key value that identifies it as deleted. In this applet we assume all legitimate key values are positive, so the deleted value is chosen as –1.

The Insert button will insert a new item at the first available empty cell or in a \*Del\* item.

The Find button will treat a \*Del\* item as an existing item for the purposes of searching for another item further along.

If there are many deletions, the hash table fills up with these ersatz \*Del\* data items, which makes it less efficient. For this reason many hash table implementations don't allow deletion.

* Quadratic Probing

The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site

In a linear probe, if the primary hash index is x, subsequent probes go to x+1, x+2, x+3, and so on. In quadratic probing, probes go to x+1, x+4, x+9, x+16, x+25, and so on.

* **Double Hashing(rehashing)**

the algorithm that generates the sequence of steps in the quadratic probe always generates the same steps: 1, 4, 9, 16, and so on.

What we need is a way to generate probe sequences that depend on the key instead of being the same for every key.

The solution is to hash the key a second time, using a different hash function, secondary hash function must have certain characteristics:

* + It must not be the same as the primary hash function.
  + It must never output a 0 (otherwise there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop).

stepSize = constant - (key % constant);

where constant is prime and smaller than the array size. For example,

stepSize = 5 - (key % 5);

* **Separate Chaining**

item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list;

* **Buckets**

Another approach similar to separate chaining is to use an array at each location in the hash table, instead of a linked list. Such arrays are called buckets.

* (382 to 389 , 392 to 398 , 400 to 406 , 408 , 409 page numb) Selection of Good hash key in hash function, Linear probe hash code, quadratic proble hash code, double hashing hash code, sepaerate chaining hash code

Heaps

* Can be used to implement a priority queue: the heap, its a kind of binary tree
* insert : at appropriate postion

remove : root elemet removed, next higher priority element moved to root position

* What happens if, while a program is running, too many items are inserted for the size of the heap array? A new array can be created, and the data from the old array copied into it. (Unlike the situation with hash tables, changing the size of a heap doesn't require reordering the data.) The copying operation takes linear time, but enlarging the array size shouldn't be necessary very often, especially if the array size is increased substantially each time it's expanded (by doubling it, for example).
* In Java, a Vector class object could be used instead of an array; vectors can be expanded dynamically.
* Heapsort
  + The basic idea is to insert the unordered items into a heap using the normal insert() routine. Repeated application of the remove() routine will then remove the items in sorted order.

Here's how that might look:

for(j=0; j<size; j++)

theHeap.insert( anArray[j] ); // from unsorted array

for(j=0; j<size; j++)

anArray[j] = theHeap.remove(); // to sorted array

* + The entire sort requires O(N\*logN) time, it's not quite as fast as quicksort. Partly this is because there are more operations in the inner while loop in trickleDown() than in the inner loop in quicksort.

* (419 to 429 , 429 to 437)

Graphs

* Graphs
  + Graphs are data structures rather like trees. In fact, in a mathematical sense, a tree is a kind of graph. In computer programming, however, graphs are used in different ways than trees
* Adjacency
  + Two vertices are said to be adjacent to one another if they are connected by a single edge
* Connected Graphs
  + A graph is said to be connected if there is at least one path from every vertex to every other vertex
* Our example programs store only a letter (like A), used as a label for identifying the vertex, and a flag for use in search algorithms, as we'll see later. Here's how the Vertex class looks:

class Vertex

{

public char label; // label (e.g. 'A')

public boolean wasVisited;

public Vertex(char lab) // constructor

{

label = lab;

wasVisited = false;

}

} // end class Vertex

* The Adjacency Matrix
  + An adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices. If a graph has N vertices, the adjacency matrix is an NxN array

**Adjacency Matrix**

**A B C D**

**A** 0 1 1 1

**B** 1 0 0 1

**C** 1 0 0 0

**D** 1 1 0 0

* The Adjacency List
  + The other way to represent edges is with an adjacency list. The list in adjacency list refers to a linked list of the kind we examined in Chapter 6, "Recursion." Actually, an adjacency list is an array of lists (or a list of lists). Each individual list shows what vertices a given vertex is adjacent to

**Adjacency List**

**Vertex List Containing Adjacent VerticesA**

**A** B->C->D

**B** A->D

**C** A

**D** A->B

* There are two common approaches to searching a graph: depth-first search (DFS) and breadth-first search (BFS).The difference is that the depth-first search is implemented with a stack, whereas the breadthfirst search is implemented with a queue.

**dfs**

visit root, push root, visit roots adjascent , visit if not visited, push to stack

if no adjascent, pop it(check adjascent, if exist perform similiar operations like root visit)

Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

Rule 2: If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

Rule 3: If you can't follow Rule 1 or Rule 2, you're finished.

**bfs**

visit the root, for all its adjascent push to queue until node with no adjascent is found. Once found remove from the queue , visit those nodes

Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex,

mark it, and insert it into the queue.

Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices,

remove a vertex from the queue (if possible) and make it the current vertex.

Rule 3: If you can't carry out Rule 2 because the queue is empty, you're finished.

**Minimum spanning tree**

five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them. This constitutes a minimum spanning tree.

The arithmetically inclined will note that the number of edges E in a minimum

spanning tree is always one less than the number of vertices V:

E = V – 1

**Directed graph**

a directed graph has only one entry in the adjacency matrix. Figure 13.12

shows a small directed graph

**Toplogical sorting**

Step 1: Find a vertex that has no successors.

The successors to a vertex are those vertices that are directly "downstream" from it—that

is, connected to it by an edge that points in their direction. If there is an edge pointing

from A to B, then B is a successor to A. In Figure 13.11, the only vertex with no

successors is H.

Step 2: Delete this vertex from the graph, and insert its label at the beginning of a list.

Steps 1 and 2 are repeated until all the vertices are gone. At this point, the list shows the

vertices arranged in topological order.

One kind of graph the topological-sort algorithm cannot handle is a graph with cycles.

What's a cycle? It's a path that ends up where it started.



the path B-C-DB forms a cycle. (Notice that A-B-C-A is not a cycle because you

can't go from C to A.)

graph v/s tree

A graph with no cycles is called a tree. However, the trees that arise in graphs are more

general than binary and multiway trees, which have a fixed maximum number of child

nodes. In a graph, a vertex in a tree can be connected to any number of other vertices,

provided that no cycles are created.

In a binary tree, each node has a maximum of two children, but in a graph each vertex may

be connected to an arbitrary number of other vertices.

A minimum spanning tree (MST) consists of the minimum number of edges necessary

to connect all a graph's vertices.

* (445,449-454, 456-460 ,461-466,469-475 ,perform dfs and bfs on few graphs )

Waited Graphs

* For unweighted graphs the depth-first search with adjacency lists requires O(V+E) time, where V is the number of vertices and E is the number of edges. For weighted graphs, both the minimum spanning tree and the shortest-path algorithm require O((E+V)logV) time.
* An algorithm using a priority queue can be used to find the minimum spanning tree of a weighted graph.
* The shortest-path problem in a nonweighted graph involves finding the minimum number of edges between two vertices.
* The shortest-path problem for weighted graphs can be solved with Dijkstra's Algorithm.
* Logic for Minimum spanning tree for waited undirected graph, for waited directed graphs , understand the dijkstraws algorothm logic, 483-490, 500-509