**Review of Java**

java

Java Compiler

javac

java source program .class file   
 (aka Machine

Language/ JVM byte codes)

-Java programs are portable to platforms to which the Java Virtual Machine (interpreter) has been ported

-Java Virtual Machine (JVM) = a virtual processor that can execute Java byte code

-Java Platform = Java’s predefined set of classes that exist on every Java installation (also referred to as the Java runtime environment)

-The classes on Java Platform are organized into related groups known as packages (java.io

package, java.lang package, java.util package, etc.)

*Primitive Types:*

Integer types: byte (8 bits), short (16 bits), int (32 bits), long (64 bits)

Real types: float (32 bits), double (64 bits)

Boolean: true and false (each 1 bit)

Character: char (16 bits) (e.g., Unicode characters)

All of these primitive types have “wrapper classes” that can be used to treat primitive values as objects.

Ex:

int i = 16;

List numbers = new ArrayList();

numbers.add( new Integer(i) ); //The int is stored as a wrapped primitive. It is stored as an Integer object

int k = ( (Integer) numbers.get(0)).intValue();//Extract primitive value from instance of wrapper class

*Classes:*

-Named collection of fields that hold data values and methods that operate on those values

-Objects act as instance of classes

-Memory to store an object is dynamically allocated on the heap when the object is created. This memory is automatically “garbage-collected” when the object is no longer needed

Differences between primitive types and classes

|  |  |
| --- | --- |
| Primitive types | Classes |
| With values  Always handled exclusively by value  Example:  int i = 6;  6  i | With Objects (Created with “new” keyword)  Handled exclusively by reference  Example:  String s = “hello”;  s hello |

Example:

String s = “hello”;

String p = s; // p now has a pointer to the String object

p = “hi”;

System.out.println( s ); //This would print hi since p and s are pointing to the same place in memory

//any change you make to the object through p is visible through p and vice versa

**Javadoc**:

- doc comments are used to embed documentation about the class and each of its methods directly into the source code. A program named *javadoc* extracts these comments and processes them to create online documentation for your class. A doc comment can contain HTML tags and can use additional syntax understood by *javadoc*.

Example:

/\*\* This is an example

\* of a doc comment.

\*/

**Orphaned Objects and Garbage Collection:**

**-** any object is considered garbage when no references to it are stored in any variables, the fields of any objects, or the elements of any arrays.

For example:

Point p = new Point(1,2); // Create an object

p = new Point(2,3); // The previous object has been orphaned

At some point, the garbage collector discovers this orphaned object and reclaims the memory. It does so automatically (unlike C++).

**Classwide vs. Instance Methods:**

- Any method with the *static* modifier is considered a class method. Therefore, it is associated with the class in which it is defined rather than with an instance of the class.

- Any method without the static modifier is considered an instance method. Therefore, it is associated with the instance of the class (object), rather than the class itself.

Example:

public class Circle{

public static final double PI = 3.14159; //class field

private double r;

public static double area( double r ){ //class method

return PI \* r \* r;

}

public double circumference(){ // instance method

return 2 \* PI \* r;

}

}

**Procedural/ Data Abstraction (using the example above):**

area() is a class method and would be invoked by saying Circle.area(5); (i.e., ClassName.classMethod(parameters))

circumference() is an instance method and would be invoked by first creating an instance of the class (object). It is invoked by calling objectName.instanceMethod().

Circle c = new Circle( 5);

double thisCircumference = c.circumference();

\* A good indicator as to whether or not something is a classwide method or an instance method, is whether or not the word before the period is capitalized (Circle.area(5) is a classwide method and c.circumference() is an instance method)

**Program args vs. standard I/O (from Dorin’s website):**

Reading from Standard Input:

Here’s the basic mantra:

java.io.BufferedReader stdIn

= new java.io.BufferedReader ( new java.io.InputStreamReader ( System.in ) );

...

String s = stdIn.readLine();// grab the first line (or null, if end-of-file)

while ( s != null ) { // while not end-of-file

process ( s ); // process this line

s = stdIn.readLine(); // grab the next line (or null)

}

This reads successive lines from the so-called standard input device (aka System.in in Java, stdin

in C, handle #0 in assembler, etc.) Technically, your program is making a request (aka a system

call) for the operating system to get data from a standard place, which is invariably initialized

as the keyboard. Note that if you actually enter input from the keyboard, at some point you will

have to indicate an end-of-file condition, which, depending on your system, can probably be done

by typing ctrl-d.

The real point of all this is that you get to decide, at run-time, where the standard input is;

so, e.g., instead of using the keyboard, you can re-direct an arbitrary text file to your program

like this:

java MapColorer < MapOfNorthAmerica.txt

Now, every time your program requests a line of input, the operating system will deliver it fromthe textfile instead. Note that, in this case, the operating system can detect the end of the

input file, so there is no need to simulate it. You should not put any special characters at the

end or they are likely to be treated as part of the input.

Since the readLine() method can throw an exception- e.g., if some extraordinary event occurs while

you’re trying to access the file- you must either put it inside some try-catch code, or else you

must indicate, in the method's signature, that an exception might be thrown, as follows:

public static void main ( String[] args ) throws Exception {

...

}

Writing to Standard Output:

Just use System.out.print()!

**Encapsulation:**

Classes allow for the *encapsulation* of data and operations (i.e., grouping of attributes and operations)

Access rules for members of a class:

*public* modifier: indicates that the member is accessible anywhere the containing class is available

*private* modifier: the member is only accessible within its containing class

*protected* modifier: the member is accessible to all classes within the package as well as to subclasses of containing class. (even if the subclass is within a different package)

no modifier: the member has default package-accessibility. It is accessible to code within all classes that are defined in the same package.

**Inheritance:**

Classes in Java, either indirectly or directly, extend java.lang.Object. (Inheritance hierarchy)

When your own object is created, an Object object is first created and then enhanced with your own specific attributes/operations

**Abstract Classes:**

- any class w/ an abstract method is automatically abstract itself and must be declared as such

- can NOT be instantiated (cannot create instances of the class (i.e., objects))

- a subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides implementation for each (called a concrete subclass)

- static, private, and final methods cannot be abstract (since these kinds cannot be overridden by subclasses). Similarly, final classes (classes that cannot be subclassed) cannot have abstract methods.

- abstract methods can have partial implementation

Example:

public abstract class Shape {

public abstract double area(); //semicolon instead of body

}

**Interfaces:**

- provide NO implementation

-a class that implements an interface must provide the necessary method bodies

-all methods are implicitly abstract—abstract modifier typically omitted per java convention

-an interface defines a public API (Application Programming Interface)

-cannot be instantiated

- can only contain method signatures (but no bodies for implementation), compile-time constants and nested types. The 2 latter of the 3 were not discussed in class.

-an interface can extend another interface (with extends clause)

Example:

public interface Centered{

void setCenter( double x, double y);

}

**Differences between Abstract Classes and Interfaces**

|  |  |
| --- | --- |
| Abstract Class | Interface |
| -can contain partial implementation  - a class can only extend one other class  -can add concrete methods to class later on | -contains NO implementation  - a class can implement as many interfaces as you would like (Java’s solution to multi-inheritance)  -if you add another abstract method later on, it breaks all classes that previously implemented the interface |

**Superclass constructors:**

When you define a constructor in a class, the class automatically invokes the super constructor.

Example:

public class Point extends Object{

private double x, y;

public Point (double x, double y){

super();// IT DOES THIS IMPLICITLY! i.e., creates an Object object and then enhances it

this.x = x;

this.y = y;

}

}

**Using iterators explicitly vs. implicitly:**

**public** Iterator iterator () {

**return** **new** return new MyIterator( this, super.iterator());//pass this as argument

//so that it will know

// to iterate over

//current MSet

}

**public class MyIterator implements Iterator{**

**private int nextIndex;**

**private int lastIndex;**

**private Iterator superIterator;**

**public MyIterator( MSet m, Iterator superIterator){**

**this.nextIndex = 0;**

**this.lastIndex = m.unique() - 1;**

**this.superIterator = superIterator;**

**}**

**public boolean hasNext(){**

**return nextIndex <= lastIndex;**

**}**

**public Object next(){**

**nextIndex++;**

**return superIterator.next();**

**}**

**public void remove(){**

**throw new UnsupportedOperationException();**

**}**

**}**

/\*\* BE CAREFUL OF SIZE WHEN IMPLEMENTING THE SUPER ITERATOR! IT USES THE SIZE() METHOD\*/

Using iterators explicitly and implicitly accomplish the same task.

*Explicitly:*

Iterator it = m.iterator();

do{

System.out.println(it.next.toString());

} while (it.hasNext());

*Implicitly: //for-each loop*

MSet m;

for (Object o : m){

System.out.println(o);

}

**Methodology for Unit Testing:**

Assert statements- Boolean- if false the program will throw an AssertionError (when assertions are enabled)

Run program with ProgramName -ea java

Good manner in which to test with assertion statements:

try {

some code here…

…

…

} catch (AssertionError ae) {

System.out.println(“Assertion error occurred during this test.”);

} catch (Exception e){

System.out.println(“Abnormal exception thrown during this test.”);

}

**Sequential (Array-Based) vs. Linked (Pointer-Based) Structures**

Physical memory (RAM) – Random Access Memory attached to CPU

Sequentially numbered collection of bytes

0

1

2

3

4

1 byte

An array of 10 ints would take up 40 cells (4 bytes per int \* 10 ints in the array)

array[i] will be at starting address of array + 4i.

In arrays, you cannot add an element because the next place in memory may not be free. Thus, it must find another large chunk of memory and copy the data over.

-16

3

18

Linked Structures contain pointers to next place in memory. Thus, the physical memory does not need to be all together. This makes adding an element easier because, for an int for example, you can just find any 4 available cells that are together.

3

-16

18

Example of Linked Structure:

class Node{

private int data;

private Node next;

public Node(){

next = null;

}  
 public int getData() { return data; }

public void setData(int i) { data = i; }

public Node getNext() { return next; }

public void setNext ( Node n) {next = n; }

}

public class CustomLinkedList {

private Node first;

public CustomLinkedList (){

first = null;

}

}

**Sequential vs. Linked (Worst-case)**

|  |  |
| --- | --- |
| Array-based | Pointer-based |
| -remove/add elements – O(n) / linear  -get an element—O(1) / constant | -remove/add elements – O(1) / constant  -get an element—O(n)/ linear |

**Asymptotics:**

Focuses on what becomes dominant as n approaches infinity

**Big Oh:**

f(n) \in \!\, Og(n) if there exists two constants c and no such that for all n ≥ no

f(n) ≤ c \* g(n)

Example: n + 3 \in \!\, O(n) when c = 3 and no = 1

**Big Omega:**

f(n) \in \!\, Ωg(n) if there exists two constants c and no such that for all n ≥ no

f(n) ≥ c \* g(n)

**Big Theta:**

If f(n) \in \!\, Og(n) and f(n) \in \!\, Ωg(n), then f(n) \in \!\, **Θ**g(n).

The function is said to be tightly bound.

Side note: n! grows faster asymptotically than 2n

In terms of asymptotic growth (using k as a constant):

n! > kn (exponential) > nk (polynomial) > nlogn > n > logn > 1

**Comparative vs. Non-Comparative Sorts**

Comparative sorts: the basic operation is a comparison including to data, to see which is smaller/bigger (or whether they’re equivalent). Examples: insertion sort, bubble sort, merge sort, quick sort

Fact: **Θ**(nlogn) is optimal for comparative sorts

Non-Comparative Sorts: Hashing

**BubbleSort:**

This algorithm works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

public static void bubblesort(int[] a) {

for (int i = 1; i < a.length; i++) {

boolean is\_sorted = true;

for (int j = 0; j < a.length - i; j++) { // skip the already sorted largest elements

if (a[j] > a[j + 1]) {

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

is\_sorted = false;

}

}

if (is\_sorted) {

return;

}

}

Worst-case: # of exchanges = (n(n-1))/2

(n-1) + (n-1) + (n-3) + … + 3 + 2 + 1

T(n) = T(n-1) + (n-1)

Worst-case **Θ**(n2) – this is the expected-case as well

**Merge Sort:**

Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

public void MergeSort(int low, int high) {

if (low < high) { // If there are more than one element

int mid = (low + high)/2;

MergeSort(low, mid);

MergeSort(mid + 1, high);

Merge(low, mid, high);

}

}

void Merge(int low, int mid, int high) {

int h = low, i = low, j = mid+1, k;

while ((h <= mid) && (j <= high)) {

if (a[h] <= a[j]) { b[i] = a[h]; h++; }

else { b[i] = a[j]; j++; } i++;

}

if (h > mid) for (k=j; k<=high; k++) {

b[i] = a[k]; i++;

}

else for (k=h; k<=mid; k++) {

b[i] = a[k]; i++;

}

for (k=low; k<=high; k++) a[k] = b[k];

}

**Θ**(n log n)

T(n) = 2 T(n/2) + **Θ**(n) // **Θ**(n) is the amount of work needed for merge

**Insertion Sort:**

Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted sublist, and inserts it there.

public static void InsertionSort (int[] a){

for (int i = 1; I < a.length; i++){

int temp = a[i];

int j;

for (j = I – 1; j >= 0 && temp < a[j]; j--){

a[j +1] = a[j];

a[j + 1] = temp;

}

}

}

Worst-case: # of exchanges = (n(n-1))/2

(n-1) + (n-1) + (n-3) + … + 3 + 2 + 1

T(n) = T(n-1) + (n-1)

Worst-case **Θ**(n2) – this is the expected-case as well

**Master Theorem:** Divide and conquer

T(n) = aT(n/b) + f(n)

f(n) \in \!\, **Θ**(nd)

**Θ**(nd) if a < bd

T(n) \in \!\,  **Θ**(ndlogn) if a = bd

**Θ**(nlogba) if a > bd

**Log Rules:**

logac = (logab)(logbc)

logab = 1

logba

loga(b/c) = logab – logac

|  |  |
| --- | --- |
| n | Estimate of  nlogn  (appx.) |
| 1 | 0 |
| 2 | .3 |
| 3 | .48 |
| 4 | .6 |
| 5 | .7 |
| 6 | .78 |
| 7 | .85 |
| 8 | .9 |
| 9 | .95 |
| 10 | 1 |

**Stacks**:

-Last In First Out (LIFO)

-(linear) list for which all accesses (push/pop) access at one and the same end known as the top

-input/output restricted

push—input (s.push())

pop—output (s.pop())

isEmpty (s.isEmpty())

peekAtTop (s.peekAtTop()) //no change to structure here

**Array-based vs. Linked-based implementation of a Stack**

|  |  |
| --- | --- |
| Array-based | Linked-based |
| -push – amortized **Θ**(1) (bad case where **Θ**(n))  -pop—worst-case **Θ**(1)  -isEmpty— worst-case **Θ**(1)  -peekAtTop— worst-case **Θ**(1)  -size— **Θ**(1)  -advantage—more space efficient  -disadvantage—resizing can take **Θ**(n) running time in the worst-case | -push – worst-case **Θ**(1)  -pop—worst-case **Θ**(1)  -isEmpty— worst-case **Θ**(n)  -peekAtTop— worst-case **Θ**(1)  -size— **Θ**(n) unless a size field is kept  -advantage—cells can come from anywhere in memory  -disadvantage—space inefficient—every node has values and address (pointer) to next cell |

Common use for stacks: Practical expression-stack machines

x = a + b \* 2 x a b 2 \* + =

-post-fixed/polished notation

-Every time we see a variable/constant, we push on to stack

-Operator—pop top two off, perform operation with 2 operands, push it back on

-say b = 13 and a = 6

32

32

x

26

a

x

2

b

a

x

2 \* b = 26 26 + a = 32 x = 32

-What happens to the x? – the semicolon throws away the top of stack

x = y[2] = 12 + m // completely legal in Java

**Computer uses memory in 2 ways:**

1. Memory given to program for construction of objects – heap
2. Rest of memory comes from the stack

Your programm

Runtime stack

M -- used for storing “activation records” for the methods

heap

-- used for construction of objects

The stack grows downwards and heap grows upwards on RAM (for space efficiency)

q—return address to p

p—return address to main

main

Always return to who called them (LIFO structure)

Activation record—storage for parameters, storage for local variables, storage for return address

**Priority Queues:**

-List (queue) FIFO

-Can insert items with a priority. A priority can be anything (but is typically a number)

-Ex: multi-programming OS—takes CPU and decides which process to give it to

**Θ**(1)—remove operation. By definition, it takes the highest priority item

**Θ**(n)—insert – adds an item with some states “priority”

data

priority

-The OS that makes decisions as to who gets resources is a process itself. If OS takes linear time to decide what process get resources, there is less resources/time for actual processes

**Heap:**

-A complete binary tree implemented in an array and which satisfies, everywhere, the “heap” property.

The “heap” property—every node has a higher priority than its two children (if any).

Complete binary tree—every level full except possibly leaf nodes (which must be left to right)

**Binary Tree:**

-No notion of leftness and rightness with plain trees

-Tree-like structure consisting of nodes:

-a set of nodes can be empty

-otherwise, one node is distinguished and called the root and the remaining nodes can be partitioned into 2 sets, each of which is a binary tree and are referred to as left and right sub-trees of the root

Heap:

Build Heap/Sort – worst case **Θ**(nlogn)

Insert—worst case **Θ**(logn)

Remove—worst case **Θ**(logn)

Can store a heap in an array

n—sons at 2n and 2n + 1

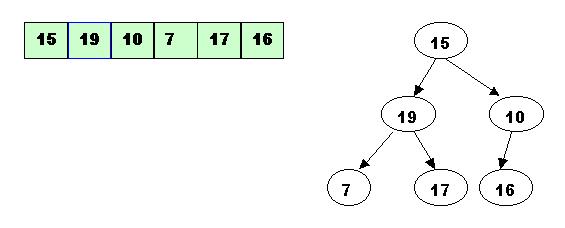
n—father at Math.floor(n/2)

Ex:

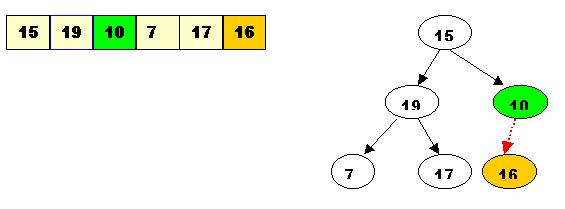
Here is the array: 15, 19, 10, 7, 17, 6

Building the heap tree:

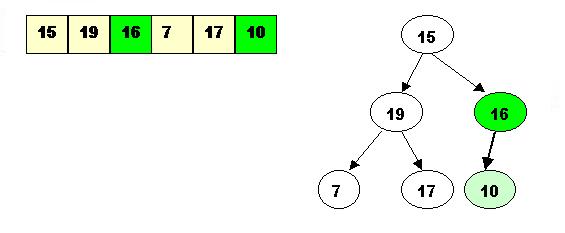
The array represent as a tree, complete but not ordered:



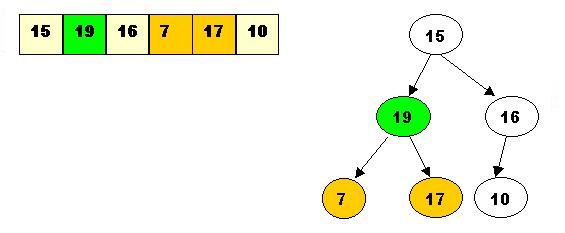
10 has one greater child and has to be percolated down



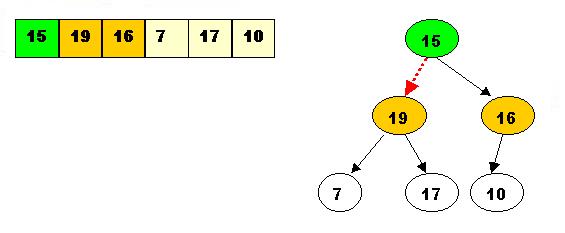
Change the array accordingly



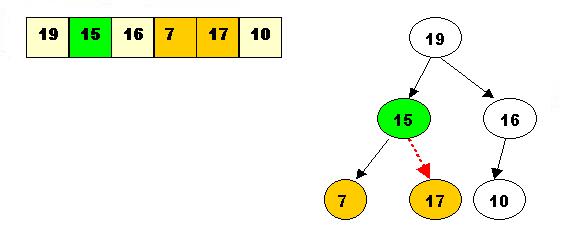
19 has all smaller children, so no percolation is needed.



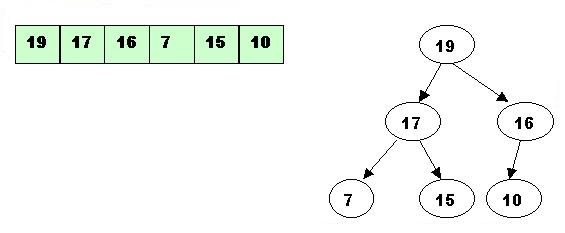
15 has a right child that is smaller than himself. 15 must be percolated down.



The result situation is:

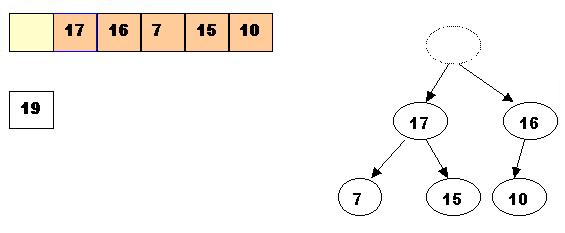


15 must percolate down further

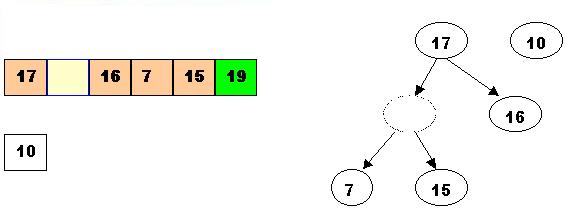


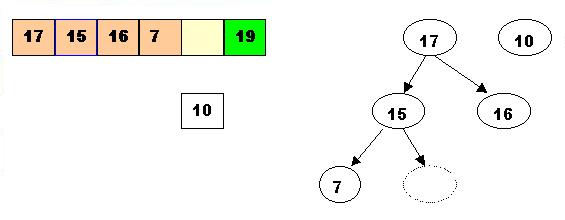
Sorting:

1. Delete the top element 19.
2. Store 19 in a temporary place. A hole is created at the top

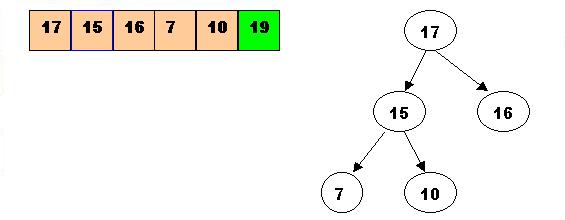


Swap 19 with the last element (since you know 19 is the max). 10 will be adjusted in the heap.





Now 10 can be inserted



Repeat this cycle with the root each time.

Example Java code for HeapSort:

public class HEAP\_SORT {

private static int[] arr;

private static int size;

private static int random;

private static int n;

private static int left;

private static int right;

private static int largest;

private static void printArray() {

for(int i=0; i<size; i++) {

System.out.println("a[" + i + "] = " + arr[i]);

}

}

public static void buildheap(){

n=arr.length-1;

for(int i=n/2;i>=0;i--){

maxheap(i);

}

}

public static void maxheap(int i){

left=2\*i;

right=2\*i+1;

if(left <= n && arr[left] > arr[i]){

largest=left;

}

else{

largest=i;

}

if(right <= n && arr[right] > arr[largest]){

largest=right;

}

if(largest!=i){

exchange(i,largest);

maxheap(largest);

}

}

public static void exchange(int i, int j){

int t=arr[i];

arr[i]=arr[j];

arr[j]=t;

}

public static void sort(){

buildheap();

for(int i=n;i>0;i--){

exchange(0, i);

n=n-1;

maxheap(0);

}

}

**Quicksort:**

-partition around some pivot element

-quicksort the data to the left (of the pivot element)

-quicksort the data to the right( of the pivot element)

Randomly choose pivot and swap it with the right-most element

Compare from left to right

If the item on the left is smaller, leave it and increment pointer

If the item on left is larger, swap them and go from the other side

**Θ**(n) to partition worst-case

void quicksort(int[] a, int lb, int rb){

if (n <= 15) {

then just insert sort them; // constant time **Θ**(152)

} else {

int p = partition around some pivot element; // **Θ**(n)

quicksort( a, lb, p-1);

quicksort(a, p+1, rb);

}

**Select:**

Given n data

1 ≤ k ≤ n

Find the kth smallest elements

Ex: n = 35 k = 12

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 35 | 12 | 13 | 90 | 67 | 61 | 1012 |
| 17 | 26 | 109 | 27 | 33 | 54 | 365 |
| 42 | 7 | 6 | 2 | 666 | 622 | 768 |
| 41 | 72 | 88 | 21 | 92 | 19 | 13037 |
| 11 | 101 | 3 | 106 | 1024 | 10 | 2048 |

1. Organize in columns of 5
2. Sort each column. Specifically, insert sort each column **Θ**(25)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 11 | 7 | 3 | 2 | 33 | 10 | 365 |
| 17 | 12 | 6 | 21 | 67 | 19 | 768 |
| 35 | 26 | 13 | 27 | 92 | 54 | 1012 |
| 41 | 72 | 88 | 90 | 666 | 61 | 2058 |
| 42 | 101 | 109 | 106 | 1024 | 622 | 13037 |

Pivot guaranteed to be 30% from end

1. Find median of medians—another select

n/5 medians. Partition using this median

recurse

continue as before

Why are we guaranteed a good pivot?

Have to be at least 30% of numbers on each side of the pivot

**Θ**(n) to insert sort

T(n/5) to find median of medians

**Θ**(n) to partition

3 cases:

-p ended in slot k (done!)

-p ended in slot < k –recurse

-p ended in slot > k – recurse

**Binary tree:**

Set of “nodes” which is either empty or has specially-designated “root” and remaining nodes can be partitioned into sets, called “left subtree” and “right subtree” of the root, each of which is itself a binary tree (recursive definition)

**Graph:**

A set of “nodes” and a set of “edges” where each edge indicates (connects) some pair of nodes

2 collections:

Collection of vertices

Collection of edges

G = (V, E)

Directed graph—collection of nodes and edges with presumed direction

Density of graph = average vertex degree

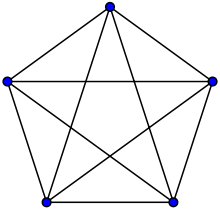
Dense: only missing ≈ |v|log|v| edges

Sparse: only has ≈ |v|log|v| edges

Complete: all pairs of nodes are adjacent (maximum size)

Maximum size = (n (n-1)) / 2

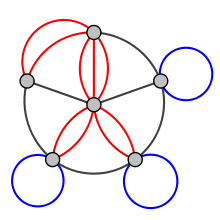
Example:



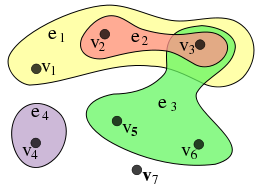
Can be represented in memory as “adjacency matrix” or “edge lists”

|  |  |
| --- | --- |
| directed | undirected |
| Edges = ordered pairs  (a,b) | Edges = two element sets  {a,b}  -(problems arise with {a,a} because sets do not allow duplicates) |

Multigraphs—permitted to have multiple edges



Hypergaph—extension/generalization of undirected graphs—edges can have n nodes (rather than just 2)



Tree—connected, loop-free graph with distinguished root node

Connected, loop-free graph without one root = free tree

Array-based storage of trees is not very effective if the tree is not complete

Linked-based

|  |  |
| --- | --- |
| info | father |
| Left son | Right son |

Traverse the structure:

Start at the root and visit each node at least once

pre(T){

if (isEmptyTree(T)){ return; }

else {

visit (root(T));

pre(T.leftSubtree());

pre(T.rightSubtree());

}

in(T){

if (isEmptyTree(T)){ return; }

else {

in(T.leftSubtree());

visit (root(T));

in(T.rightSubtree());

}

post(T){

if (isEmptyTree(T)){ return; }

else {

post(T.leftSubtree());

post(T.rightSubtree());

visit (root(T));

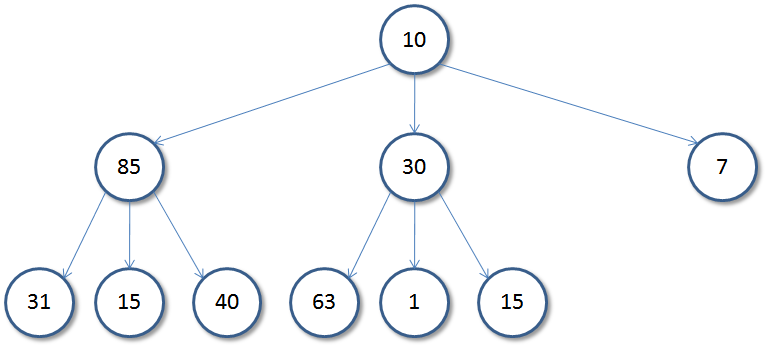
}

-Recursive implementation allows for you to not explicitly deal with stacks (happens under the hood)

-Also know how to traverse trees in preorder, inorder and postorder iteratively

**N-ary Trees:**

Example: n = 3



info

father

2

3

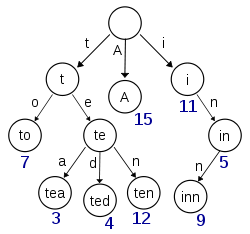
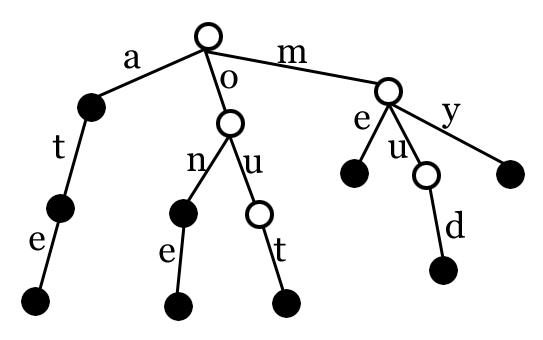
1

In an array, sons at indices 3i-1, 3i, and 3i+1 (if complete)

**Tries:**

Retreival Trees

n-ary trees

Allows you to quickly tell whether a word is or is not in a given set

Every node has n possible descendants

count

dad

-(index array as letters)

a b c d etc….

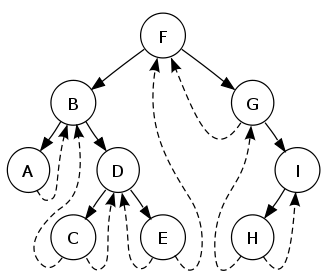
\*decorate edges with letters

**Threading:**

N nodes implies ? nulls

Links: 2n-(n-1) actually used

= n+1 nulls



See if a node *k* that has a right child *r*. Then the left pointer of *r* must be either a child or a thread back to *k*. In the case that *r* has a left child, that left child must in turn have either a left child of its own or a thread back to *k*, and so on for all successive left children. So by following the chain of left pointers from *r*, we will eventually find a thread pointing back to *k*.

**Binary Search Tree:**

BST property—

At each node n:

Info(node n) > infor(any node in n’s left subtree)

And

< info (any node in n’s right subtree)

\*They are not unique

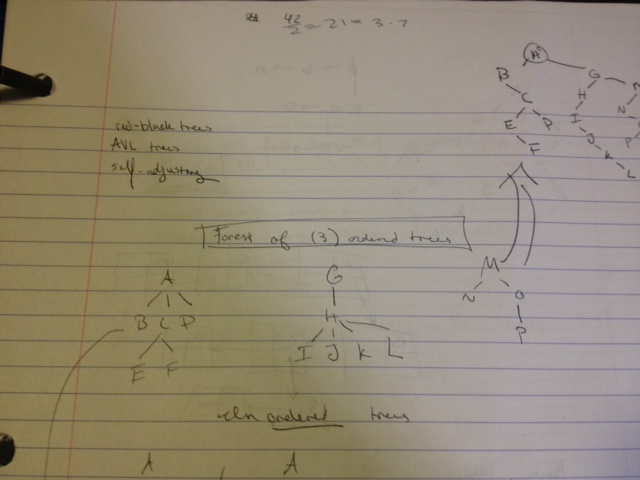
Can find any given element in **Θ**(logn) IFF the tree is balanced

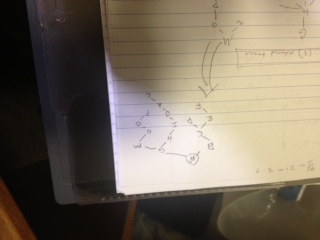
**Binary tree representation for forest of ordered trees:**

Binary Tree = Forest .

Root Root of left-most tree

|  |  |
| --- | --- |
| **Binary Tree** | **Forest** |
| Root | Root of left most tree |
| Left Subtree of Root | Subtree of leftmost tree |
| Right Subtree of Root | Remaining Trees |





**Hashing:**

Hash tables

Map: K V

map(k) = v

See packet for more information on hashing