INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 4

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Last Class
  - Linked Lists, Queues, Stacks, Recursion
- Today
  - Merge & Quick Sort
  - JavaDocs
  - Testing
  - Trees

Linked List

Big-O

Operation Implementation	get set	add remove last	insert remove front	insert remove middle	search
Singly Linked List	N	1 or N*	1	N	N
Doubly Linked List	N	1	1	N	N

* Add is 1 for double-ended, remove last is still N

- Only use memory proportional to N
- “Locality of reference” poor compared to array based

Last Class: Node Data Structure

- What **fields** do we need to store a node?

- What **methods** do we need?

Last Class: List Data Structure

- What **fields** do we need?

- What **methods** do we need?

List Methods

- Minimum Required to Emulate an Array
  - Get / Set element
  - Append
  - Get size
- Cool new things
  - Remove
  - Insert
  - Append can't (usually) run out of space (just like a dynamic list)

Questions?

Last Class: Stacks & Queues

- Stacks
  - ____ in ____ out?
  - 4 common methods?
- Queues
  - ____ in ____ out?
  - 4 common methods?

Last Class: Stacks & Queues

- Stacks
  - last in first out (LIFO)
  - push(), pop(), peek(), isEmpty()
- Queues
  - first in first out (FIFO)
    - BUT THERE ARE OTHER TYPES!
  - enqueue(), dequeue(), peek(), isEmpty()

Stack

Big-O

Operation Implementation	push	pop	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

Easiest to implement Dynamic Array, constants are better

Queue

Big-O

Operation Implementation	enqueue	dequeue	peek	isEmpty	size
Dynamic Array	1	1	1	1	1
Doubly Linked List	1	1	1	1	1

For Queue, Dynamic Array is called a “Circular Queue”
Easiest to implement Singly (double-ended) Linked List

Last Class: Priority Queues

- _________ comes out first
- Implementations
    - __________
    - __________

Last Class: Priority Queues

- Highest priority comes out first
  - Not FIFO or LIFO
  - Priority could be max or min
- Implementations
  - Multiple queues
  - Single queue
- Naive Approaches
  - Unordered Array
  - Ordered Array

Questions?

Last Class: Recursion

- Idea: keep doing the ______ thing, ______ the problem
- Key components
  - _______ case (when to ______)
  - ______ case (when to ______)

Last Class: Recursion

- Idea: keep doing the same thing, reducing the problem
  - smaller subset of the problem
  - one step closer to the answer
- Key components
  - recursive case (when to keep going)
  - base case (when to stop)

Questions?

Last Class: Binary Search

- Requires?
- Method?
- Big-O?
    - worst case?
    - best case?

Last Class: Binary Search

- Requires: sorted list
- Method, see:
  - http://en.wikipedia.org/wiki/Binary_search_algorithm
- Big-O
  - worst case: O(log n)
    - why?
  - best case: O(1)
    - why?

Sorted Array Binary Search Code

```
public static int search( Comparable findItem, Comparable[] items )
{
    int index = -1;
    int lo = 0;
    int hi = items.length-1;
    while( lo <= hi )
    {
        // Find half way position between begin and end
        int mid = lo + (hi - lo) / 2;
        Comparable midItem = items[mid];

        if(      findItem.compareTo(midItem) < 0 ) hi = mid-1; // Move left
        else if( findItem.compareTo(midItem) > 0 ) lo = mid+1; // Move right
        else
        {
            // Must be equal, narrowed to one index, exit loop
            index = mid;
            break;
        }
    }
    return index;
}
```

Divide and Conquer

- Divide the problem
  - in half or some smaller portion
- Keep doing that (based on recursion)
  - until the problem is small enough to solve (conquer)
- If needed, use the smaller solved problems to solve the big one (conquer)
- Traditionally, “divide-and-conquer” algorithms have two or more recursive calls.

Recursion Alternatives

- Recursion can make a seemingly difficult problem easy to solve, is often the most elegant approach, and generally easier to verify
- However, it has non-trivial overhead to add activation records to the stack for each method call
- Factorials and triangle problems use recursion only for teaching, in practice use iterative approach
- Common recursion alternatives (Look at Triangle.java)
  - Iterative, not always possible
  - Explicit stack (call stack is implicit), used to be important, now compilers quite efficient and approach is less often useful

Questions?

Sorting

- Often desirable to have output sorted
- More importantly, sorting input can make algorithms much more efficient (e.g. binary search arrays)
- Because it is a building block for other algorithms, must understand performance
- Three sub-quadratic algorithms (many more)
  - Shell Sort (will not cover, easy to implement)
  - Merge Sort, invented by Von Neumann
  - Quick Sort, invented by Hoare

Stable Sorting

- Relative ordering for duplicate values is maintained, important for objects

Original Sorted by Name		
Arda	8	PA1
Arda	7	PA2
Arda	10	PA3
James	8	PA1
James	3	PA2
James	8	PA3
Mary	6	PA1
Mary	6	PA2
Mary	6	PA3

Not Stable Sorted by Grade		
James	3	PA2
Mary	**6**	**PA3**
Mary	**6**	**PA1**
Mary	**6**	**PA2**
Arda	7	PA2
James	**8**	**PA3**
Arda	**8**	**PA1**
James	**8**	**PA1**
Arda	10	PA3

Stable Sorted by Grade		
James	3	PA2
Mary	**6**	**PA1**
Mary	**6**	**PA2**
Mary	**6**	**PA3**
Arda	7	PA2
Arda	**8**	**PA1**
James	**8**	**PA1**
James	**8**	**PA3**
Arda	10	PA3

Merge Sort

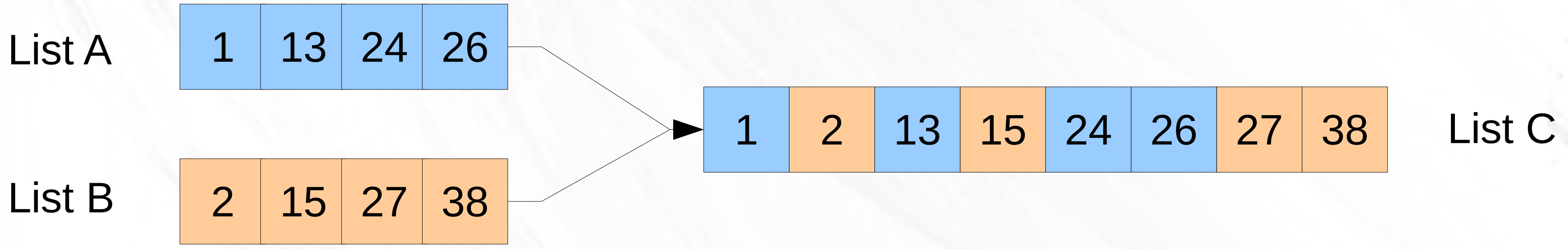
- If the problem is too big

  … find a smaller problem.
- Demo
- Resource with animations if you forget this:
  - http://en.wikipedia.org/wiki/Merge_sort
  - http://www.sorting-algorithms.com/merge-sort

Merge Operation

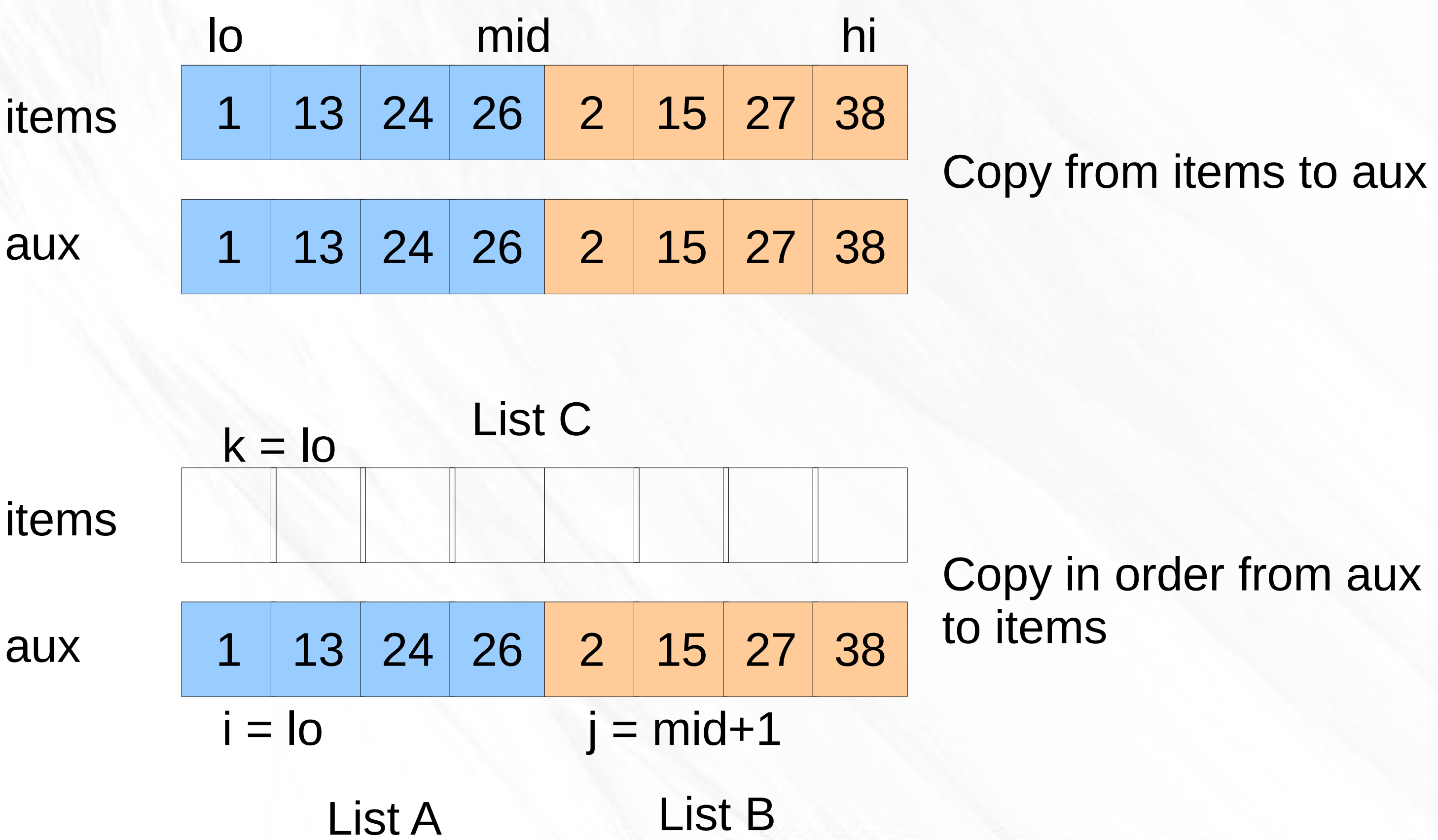
Weiss 8.5.1

- Given two sorted lists, merge into one sorted list in linear time O(n)



Merge Operation Memory

- Memory to copy from and memory to copy to.



Merge Operation Outline

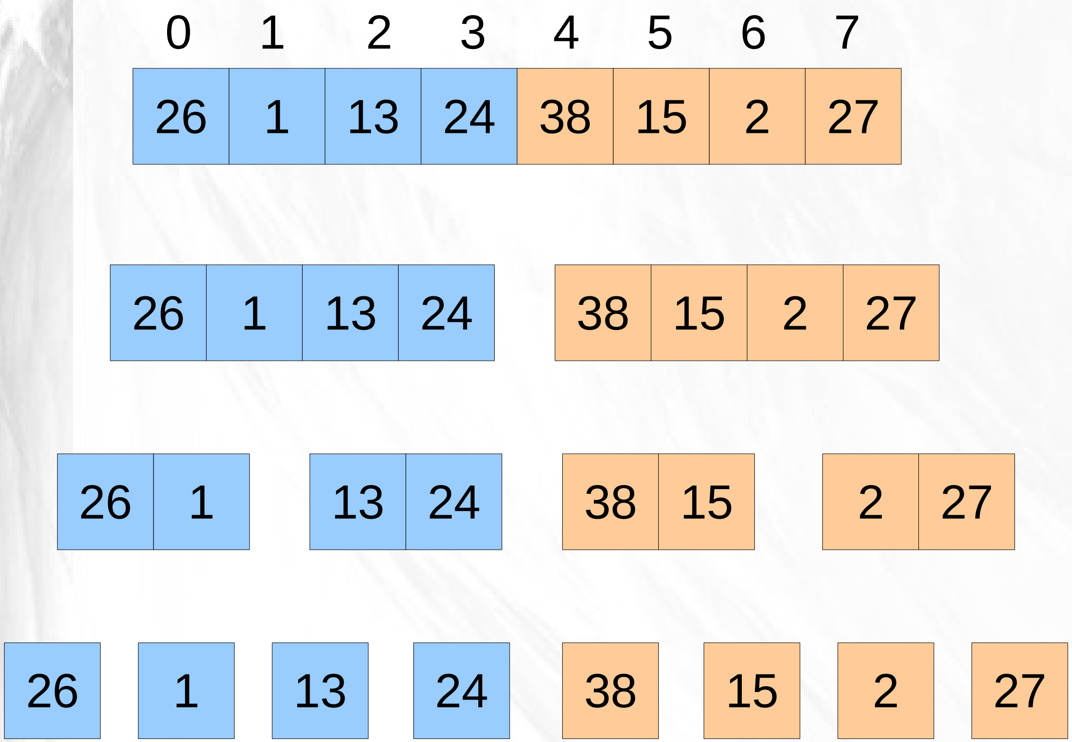
“Top-down Mergesort”

```
public static void merge( Comparable[] items, Comparable[] aux,
                          int lo, int mid, int hi )
{
    // Copy items from lo to hi to the aux array

    // Now copy back into item, file out in order
    // Consider [lo,mid] list a, [mid+1,hi] list b
    int i = lo;     // start index for list a
    int j = mid+1; // start index for list b
    for( int k = lo; k <= hi; k++ ) // k index into items
    {
        // Handle cases where a list is empty
        //      If a is empty,   take from b
        // Else If b is empty,   take from a
        // Else If b is smaller, take from b
        // Else a is smaller or equal, take from a

        // Note order, if equal, get from a,
        // comes first to keep stable
    }
}
```

Merge Sort Outline

```
// Note: not real code...
list mergeSort(list)
{
   if(list is empty or contains 1 element)
      return list
   list1 = mergeSort(first half of list)
   list2 = mergeSort(second half of list)
   return merge(list1, list2)
}
```

Merge Sort Tree



Merge Sort Trace

```
lo=0 hi=7
lo=0 hi=3
lo=0 hi=1
lo=0 hi=0
lo=1 hi=1
          Merge: [0,1]
lo=2 hi=3
lo=2 hi=2
lo=3 hi=3
          Merge: [2,3]
          Merge: [0,3]
lo=4 hi=7
lo=4 hi=5
lo=4 hi=4
lo=5 hi=5 Merge: [4,5]
lo=6 hi=7
lo=6 hi=6
lo=7 hi=7
          Merge: [6,7]
          Merge: [4,7]
          Merge: [0,7]
```

Merge Sort Heuristics 1/2

Sedgewick/Wayne 2.2

- Cutoff value below which simple sorting is used (e.g. InsertionSort)
  - Recursive call overhead to sort just a few items items is relatively significant
  - Common technique for other recursive sorting algorithms
  - For small n (e.g. 8), simple sorts are faster
- Avoid copy to auxilliary array
  - Tricky, essentially reverse roles of the items and auxilliary arrays on recursive calls

Merge Sort Heuristics 2/2

Sedgewick/Wayne 2.2

- Check if two lists are already in sorted order to avoid the merge O(N)
    - Left list max value at items[mid]
    - Right list min value at items[mid+1]
- Use iterative approach (“bottom-up mergesort) which avoids recursion altogether

Properties of Merge Sort

- Not in-place
  - requires O(n) additional memory space
- Stable
  - relative order of equal elements preserved

Operation Implementation	worst	average	best	in place	stable	remarks
Selection Sort	N^2	N^2	N^2	yes	no	
Insertion Sort	N^2	N^2	N	yes	yes	
Merge Sort	N lg N	N lg N	N lg N	no	yes	

Questions?

The Quick Sort Algorithm

Weiss 8.6.1

The basic algorithm Quicksort(S) consists of the following four steps.

1. If the number of elements in S is 0 or 1, then return.

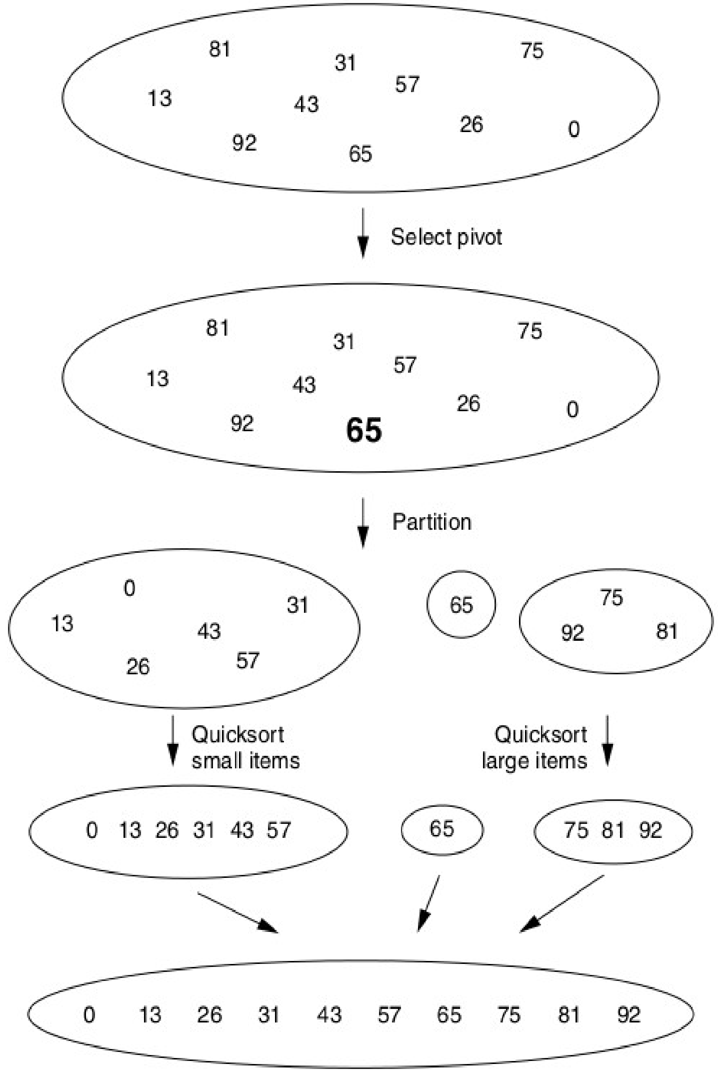
2. Pick any element v in S. It is called the pivot.

3. Partition S – {v} (the remaining elements in S) into two disjoint groups:
$L = \{ x \in S - \{ v \} \, x \leq v \}$ and
$R = \{ x \in S - \{ v \} \, x \geq v \}$.

4. Return the result of Quicksort(L) followed by v followed by Quicksort(R).

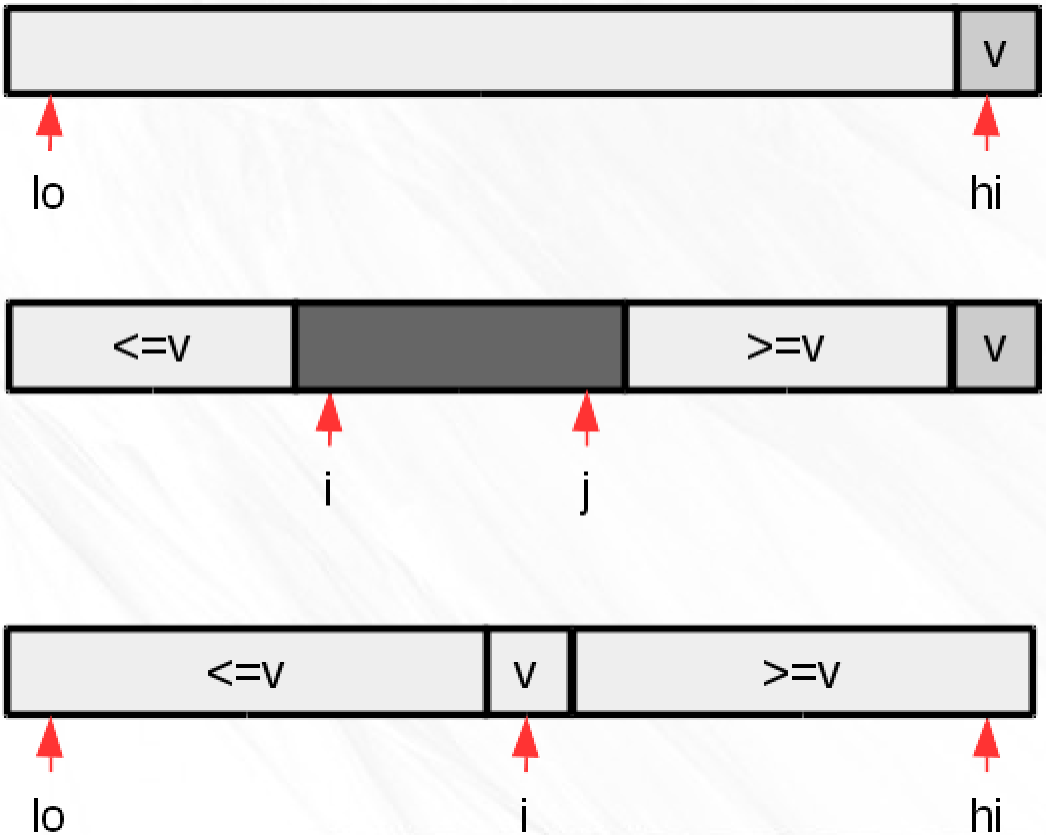
Steps of Quick Sort

Weiss Figure 8.11



The Quick Sort Partition 1/2

Invariant: For a given pivot, no larger values to the left of the pivot and no smaller values to the right of the pivot.



The Quick Sort Partition 2/2

Weiss 8.6.4

8	1	4	9	0	3	6	2	7	5	Step0: Pick pivot (6)
lo									hi	
8	1	4	9	0	3	5	2	7	6	Step1: Move out of way
i = lo									j = hi-1	
8	1	4	9	0	3	5	2	7	6	Step2: Small elements to left of array and large elements to right of array
i							j			
2	1	4	9	0	3	5	8	7	6	Swap 8 and 2
i							j			
2	1	4	9	0	3	5	8	7	6	
			i			j				
2	1	4	5	0	3	9	8	7	6	Swap 9 and 5
			i			j				
2	1	4	5	0	3	9	8	7	6	
					j	i				
2	1	4	5	0	3	6	8	7	9	Step3: Swap 6 and 9
					j	i				

Quick Sort

- There is a good chance that

  … randomness is your friend
- Quick sort does work (partition) then recursion
- Merge sort does recursion then work (merge)
- Demo

- Resource with animations if you forget this:
  - http://en.wikipedia.org/wiki/Quicksort
  - http://www.sorting-algorithms.com/quick-sort

Quick Sort

```
// Note: not real code...
int quickSort(list)
{
   if(list is empty or contains 1 element)
      return list
   int pivot = some item in the list
   for(each item in the list)
   {
      if(item smaller than pivot)
         put in first “section” of list
      if(item larger than pivot)
         put in last “section” of list
   }
   put pivot in between two sections
   quickSort(first “section” of list)
   quickSort(last “section” of list)
   return list
}
```

Quick Sort Variants

Weiss 8.6 and 9.4

- Pick middle
- Estimate median (3 samples)
- Randomly shuffle input before sorting
    - Statistically guarantees average performance

```
// Similar to Knuth / Fisher-Yates shuffling algorithm
// If generator independent, uniform, so is output
public static void shuffle(Object[] items)
{
    for(int j = 1; j < items.length; j++)
    {
        swap( items, j, random.nextInt(j) ); // [0,j)
    }
} // returns some permutation of items, adds O( N )
```

Space Complexity: Quicksort

- Space complexity
  - O(log n) or O(n)
- Why?
- Just add more memory?
  - Is this stack or heap memory?
- Other variants to reduce memory to log(N)
  - Tail recursion / Iteration
- Duplicate values, can degrade to O(N^2) !!!
  - Quick 3-Way solves, [< v] [= v] [> v]

Properties of Quick Sort

- Space complexity?
- In-place/Not in-place
- Unstable/Stable

Operation Implementation	worst	average	best	in place O(1)	stable	remarks
Selection Sort	N^2	N^2	N^2	yes	no	
Insertion Sort	N^2	N^2	N	yes	yes	
Merge Sort	N lg N	N lg N	N lg N	no	yes	
Quick Sort	N^2	N lg N	N lg N	yes*	no	fast practice
???	N lg N	N lg N	N	yes	yes	Unknown

* Depending on variant, will assume O(lg(N)) ~ O(1)

Sorting Summary

- Merge sort (Java objects, C++/Python stable)
  - Stable and efficient but requires extra memory
- Quick sort (Java primitives, Python, Matlab)
  - Little extra memory and efficient, not stable
- Which one to use?
  - If memory is limited (in-place), use quicksort
  - If stability is required, use mergesort
- Example: Java system programmers assume if using primitives then memory is limited and if using objects then stability is more desirable

Questions?

JavaDocs

- Use javadoc style for this class
  - Always use @param and @return for all methods with parameters and return types
  - Use these on your assignments!
- http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Example Class Comment

```
/**
 * Model of a banana
 */
public class Banana
{

}
```

Javadoc Annotations

```java
/**
 * Model of a banana
 *
 * @author Your Name Here
 * @version 0.1-alpha
 */
public class Banana
{

}
```

HTML!

```
/**
 * <p>Model of a banana.</p>
 *
 * <p>Another paragraph about the awesome
 * banana class</p>
 *
 * @author Your Name Here
 * @version 0.1-alpha
 */
public class Banana
{

}
```

Example Method Comment

```
/**
 * myMethod provides users with some
 * cool functionality.
 */
public void myMethod()
{

}
```

Return Annotation

```
/**
 * Counts how many bananas we have.
 *
 * @return number of bananas in the system
 */
public int countBananas()
{

}
```

Param Annotation

```
/**
 * Divides two numbers.
 *
 * @param a the numerator
 * @param b the denominator
 * @return the quotient of a and b
 */
public double div(int a, int b)
{

}
```

Param Annotation

```java
/**
 * Divides two numbers.
 *
 * @param a the denominator
 * @param b the numerator
 * @return the quotient of a and b
 */
public double div(int a, int b)
{

}
```

Throws/Exception Annotation

```java
/**
 * My method throws that exceptions...
 *
 * @throws UnsupportedOperationException
 *          Always throws this
 */
public void myMethod()
{

}
```

Member Comment

```
/**
 * Model of a banana
 */
public class Banana
{
   /**
    * Keeps count of number of bananas
    * in existence
    */
   public static int count = 0;
}
```

Questions?

Getting the Bugs Out

- Most development time is spent testing and debugging code
- How do you know when your code works?
- How do you catch bugs?
- How do you fix bugs?
  - If you fix a bug, how do you know it stays fixed?

Software Engineering: Testing

- Quality Assurance (QA) process – manner in which a company ensures its product is acceptable
- Code Reviews – meeting in which several people examine a design document or section of code
- For medium and large systems, testing must be a carefully managed process
  - Many organizations have a separate QA department to lead testing efforts

Unit and Integration Testing

Liskov and Guttag 10.7

- Unit testing checks each individual module for correctness in isolation. Typically software engineers are responsible for unit testing.
- Integration testing checks entire program correctness when all the modules are put together. Usually harder than unit testing and may be carried out by software or system engineers.

Automated Tests

- Test Case – set of inputs and actions coupled with expected results
- Test Suite – test cases and/or testing framework which is stored and reused as needed
- Defect Testing – execution of test cases to find errors
- Regression Testing – running previous test suites to ensure new errors have not been introduced

How is Testing Done

- Testing can mean many different things:
  - running a program on various inputs
  - human or computer assessment of quality
  - evaluations before writing code
- The earlier we find an problem, the easier and cheaper it is to fix
  - Some software engineering approaches advocate writing test cases **BEFORE** you even write any code (Termed "Test Driven Development (TDD)")
    - Why?

Unit Testing 1/2

- Main test method (some languages do not allow more than one main). Simple, black box view.
- Java assert. Typically used to maintain internal invariants, not black box view.
- Testing framework. More complicated but more powerful black box view. Most common in industry.

Unit Testing 2/2

- Commonly used frameworks to test code
  - JUnit (http://junit.org) for java
  - NUnit for c#
- Idea:
  - Write code & test method
    - @Test (Java Annotation, not Javadoc)
  - Test methods do something & check if answer is correct, using assert-type methods
    - assertTrue(..), assertFalse(..), assertEquals(x,y)

Example: PA3

```
/**
 * Check HeapSort, if takes longer than timeout
 * test case will fail
 */
@Test(timeout=2000)
public void heapSort()
{
   .. .. ..
   int[] sortedList = PA2.heapsort(list);
   list = insertionSort(list);
   assertArrayEquals(sortedList, list);
}
```

Example: PA3

```
/**
 * Expected exception, test case succeeds if and
 * only if the exception is thrown
 */
@Test(expected=RuntimeException.class,
       timeout=2000)
public void myMethod()
{

}
```

Assert Things!

```
import org.junit.*; //junit
import static org.junit.Assert.*; //assert
```

- junit provides convenience functions for testing
  - Assert.assertEquals()
    - 2 params: expected value & actual value
    - lot of versions of this function
  - Assert.assertTrue() & Assert.assertFalse()
  - Assert.assertNull() & Assert.assertNotNull()
- http://junit.sourceforge.net/javadoc/org/junit/Assert.html

JUnit Assert vs Java Assert

- Java has a keyword “assert”
  - assert <boolean>, e.g. assert isSorted(array);
  - If false, AssertionError is thrown
  - Have to enable “java -ea MergeSort” (default disabled)
- JUnit also has similar behavior
  - http://junit.org/apidocs/org/junit/Assert.html
    - assertTrue(...)
    - assertEquals(...)
  - Throws same AssertionError
- JUnit is considered more flexible and black box view of code.
- Either approach no running cost for production code
  - Java assert in compiled code, JUnit is not

Java Annotations

http://docs.oracle.com/javase/tutorial/java/annotations/basics.html

- Information for the compiler
  - Like comments but the compiler may not completely ignore them
- Metadata that summarizes the intent of code

Java Annotation Examples

- Java Annotations - http://en.wikipedia.org/wiki/Java_annotation
- Examples
  - @Test This code tests other code (compiler may just ignore)
  - @Deprecated This code is old, unsupported, may disappear
  - @Override Warn if not overriding parent method

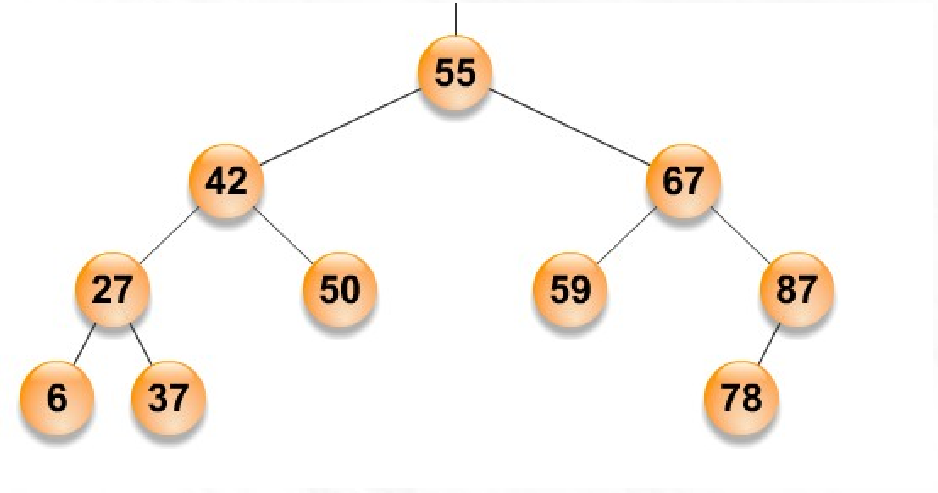
What Testing Is...

- Unit tests are just more code
  - Intended to test other code
  - Can have bugs (what’s testing the tests?)
  - Can miss critical stuff
- Very common job in industry
- To really learn testing, write some tests!

Questions?

Trees

- Data structure which looks like an upside down tree (or the root system of a tree)
  - Nodes have parents and children
  - No loops



Trees

- Collection of nodes and edges
  - These nodes are different!
  - Any shape, but can't have a loop
  - Acyclic means “no cycles” (i.e. no loops)
- Nodes have:
  - data
  - (possibly) a “key” to sort/search by
  - (possibly) pointer to children
  - (possibly) pointer to parent

Types of Nodes

- By Relationship
  - Parent/Child Nodes
  - Ancestor/Descendent Nodes
- By Tree Location
  - Inner/Branch/Internal Nodes
  - Outer/Leaf/External/Terminal Nodes
  - Root Node (one 1!)
- Null Links

Tree Definitions 1/2

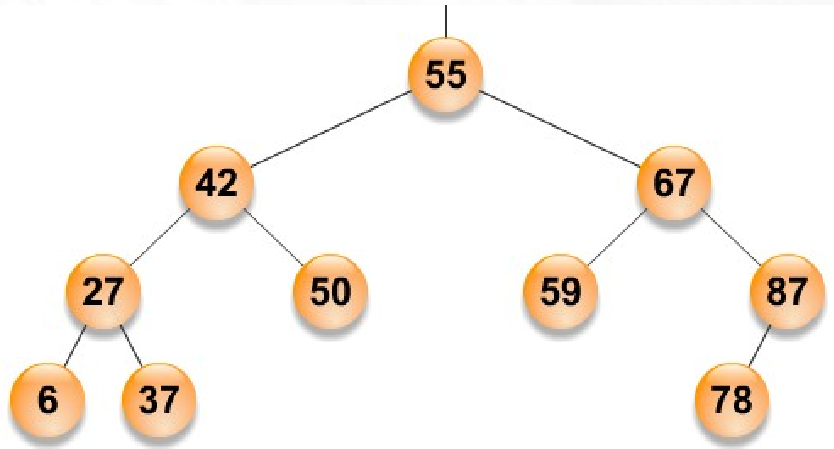
- The descendants of a node are all the nodes below it and includes itself
- The ancestors of a node are the nodes on the path from the node to the root
- The leaf nodes have no children, the root node has no parent
- Nodes are siblings if they have the same parent

Tree Definitions 2/2

- The depth of a node is the length (number of edges) of the path from the node to the root
- The tree height is the maximum depth of any node in the tree
- An inner node has at least one child (i.e. not a leaf)

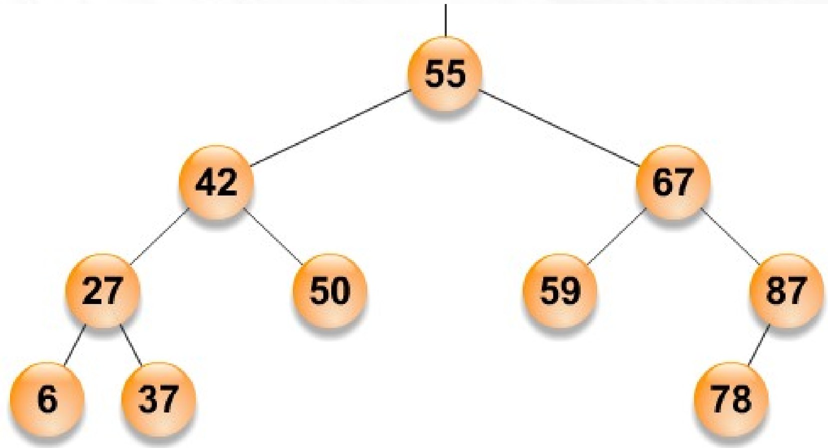
Examples

- What is the parent of 27?
- What are the children of 67?
- What are the ancestors of 59?
- What are the descendants of 55?



Examples

- What is the root?
- Which nodes are leaf nodes?
- Which nodes are inner nodes?
- Where are the null links?



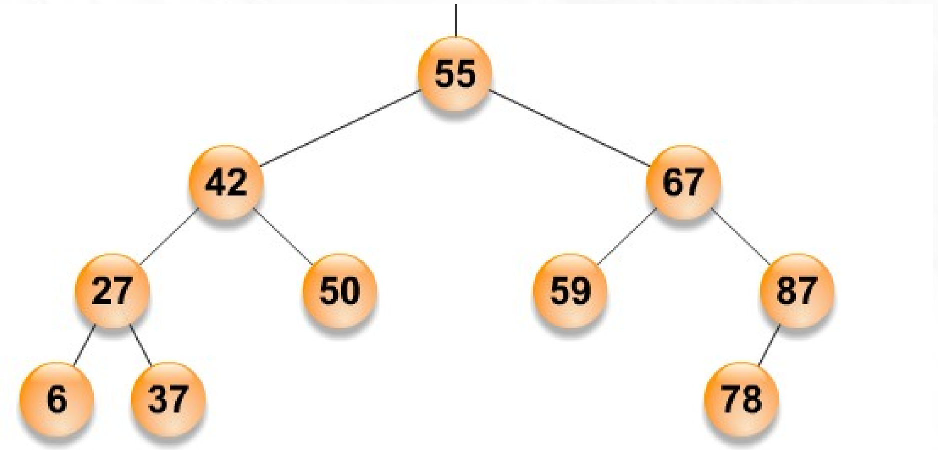
Types of Trees: Fullness

Note: Ambiguous Definitions

- full/perfect tree
  - every node other than the leaves has the max number of children
- perfect/complete tree
  - all leaves have the same depth
  - every node other than the leaves has the max number of children
- almost complete or nearly complete tree
  - last level is not completely filled

Examples

- Is this tree full?
- Is this tree complete?
- Is this tree nearly complete?
- What is the tree height?

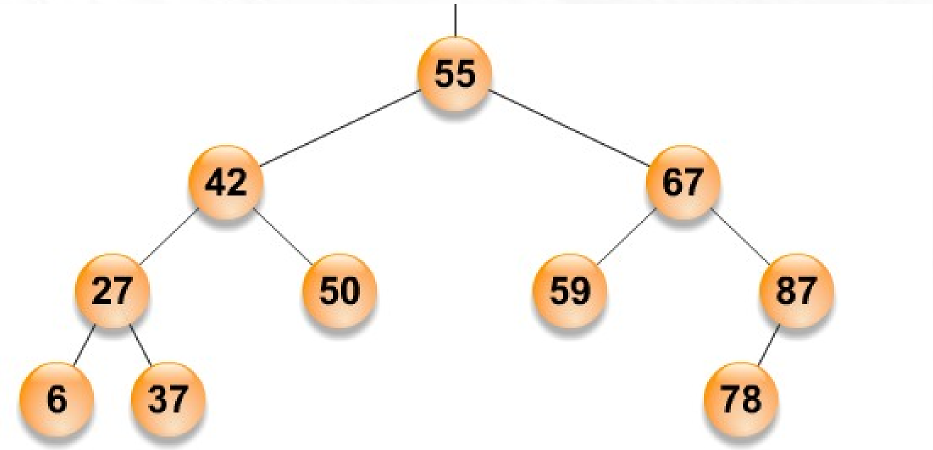


Types of Trees: Arrangement

- balanced tree
  - height of the left and right sub trees of every node differ by 1 or less
- degenerate tree
  - each parent node has only one associated child node
  - equiv. to linked list, maximum height?

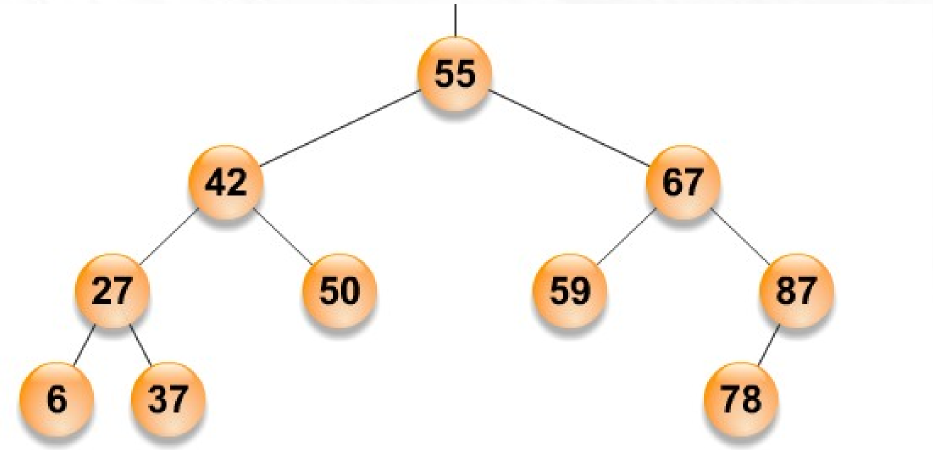
Examples

- Is this tree balanced?
- Is this tree degenerate?



Examples

- Is this tree balanced?
- Is this tree degenerate?
  - what nodes could we remove to make it degenerate?



Questions?

Common Tree Operations

- Iterating (a.k.a. Enumerating) = mention things one by one
  - all the items
  - a section of a tree
- Searching for an item
- Adding/Deleting items
- Pruning/Grafting
- Balancing

Tree Data Structures

- Arrays
  - Need to know where each item is
    - How? Need to limit number of children
  - Most common for balanced binary trees
  - Fast memory access (compared to linked list)
- Linked Data Structures
  - easy to add, remove, and swap around parts of the tree

Questions?

Assignments: PA2 / PA3 / PA4

- PA2: Grades posted
- PA3: Due Tomorrow
- PA4
  - Practice recursive techniques
  - Implement a sorting algorithm

Assignments: PA2

- Generics
- Array List (Java Collections)
- Compiling
- Exceptions (ranges negative) throw vs try/catch
- Edge case – reducing size to zero
- Insert
- Null checking
- Submission format
- Generics

Let's Look at the Files

Free Question Time!