INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 5

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Last Class
    - Merge Sort, Quick Sort, Trees
- Today
    - Last Lecture Review
    - Heaps
    - More Trees
    - Midterm Review

Questions?

Divide and Conquer

- Three Steps...

Divide and Conquer

- Divide the problem
  - in half or some smaller portion
- Keep doing that
  - until the problem is small enough to solve (conquer)
- If needed, use the smaller solved problems to solve the big one (Conquer)

Why divide and conquer is lg(n)...

- with n items in a list
  - let's pretend n is a power of 2 ($n = 2^h$)
- every iteration you divide the problem, when you're down to a single element to work on...
  - how many times do you need to multiply by 2 to get to a level where the problem size $= 2^h$?
    - h....
  - $lg(2^h) = lg(n)$, so $h = lg\ n$
- n is not a power of 2? that's ok, it's lg(n) +/- 1
  - big-O does what with smaller terms?

Questions?

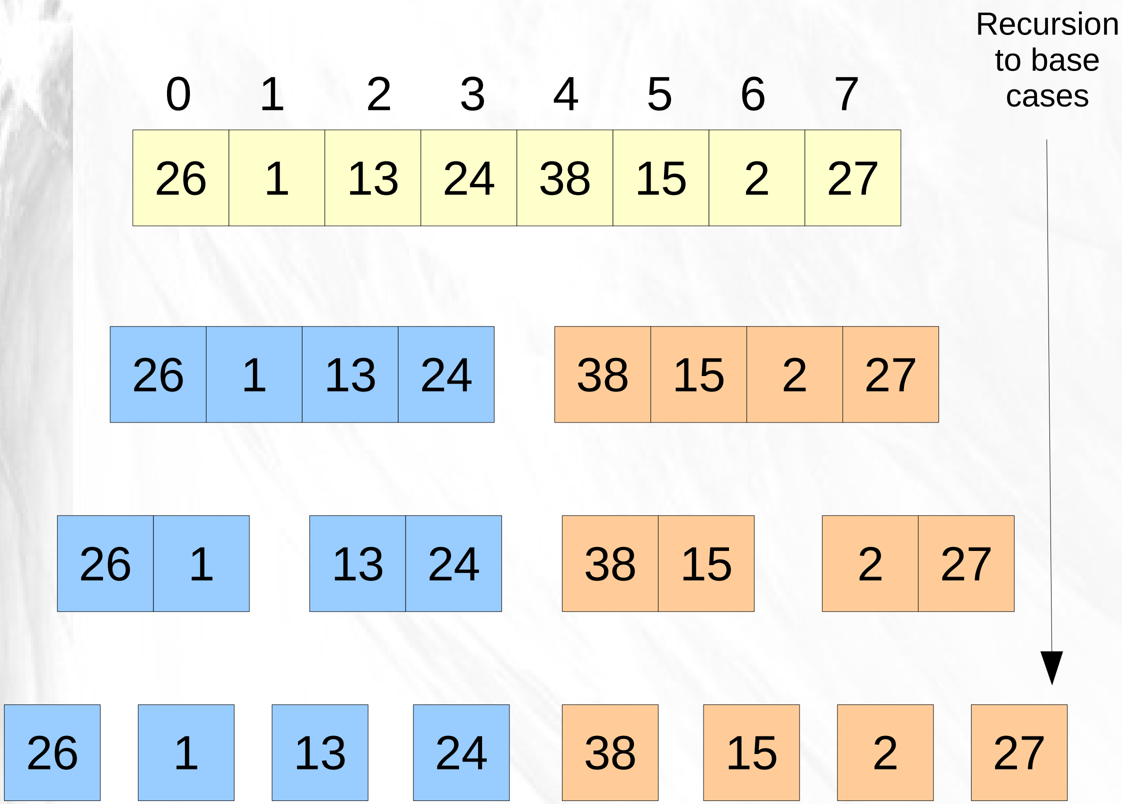
Last Class: Merge Sort

- https://www.youtube.com/watch?v=XaqR3G_NVoo

Merge Sort Method 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)
}
```

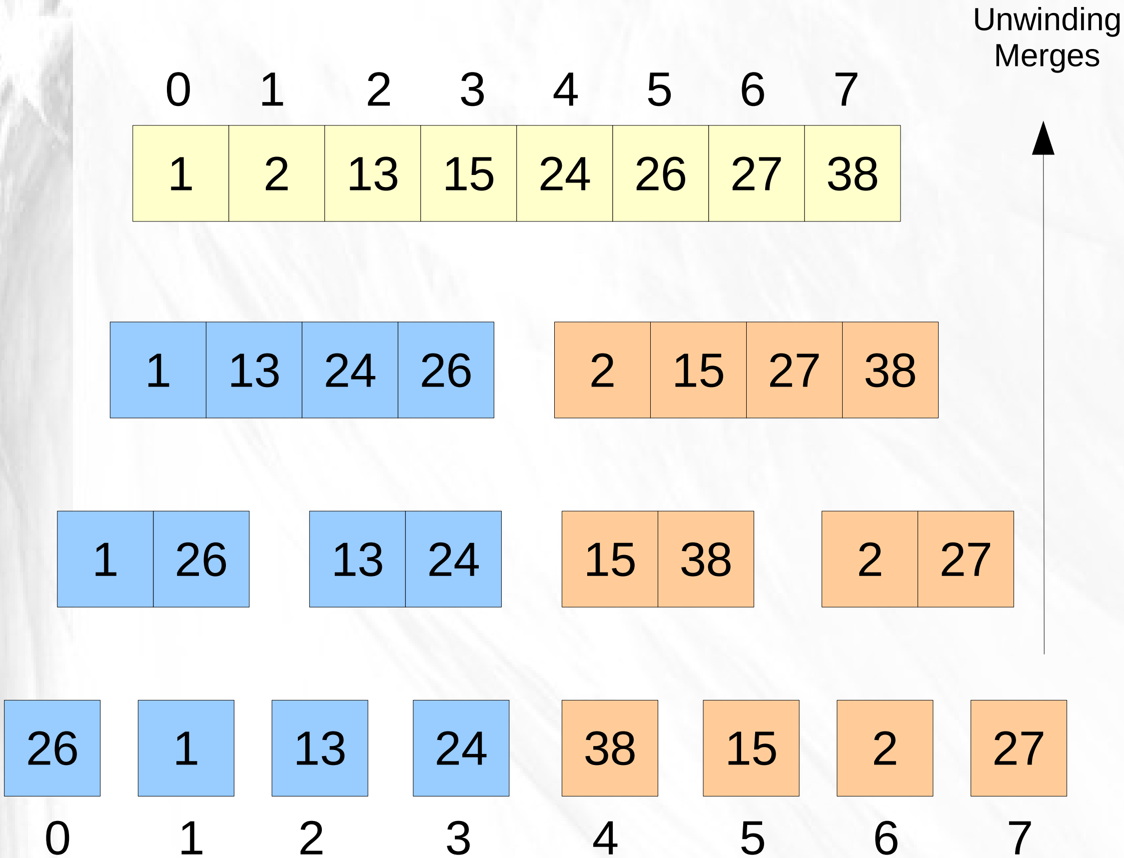
Merge Sort Tree 1/2



<u>Merge Sort Trace</u>

```
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 Tree 2/2



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]
```

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?

Last Class: Quick Sort

- https://www.youtube.com/watch?v=ywWBy6J5gz8

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).

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
   // Partition
   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
}
```

The Quick Sort Partition

Weiss 8.6.4

Step0: Pick pivot (6)

8	1	4	9	0	3	6	2	7	5
lo									hi

Step1: Move out of way

8	1	4	9	0	3	5	2	7	6
i = lo								j = hi-1	

Step2: Small elements to left of array and large elements to right of array

8	1	4	9	0	3	5	2	7	6
i							j		

Swap 8 and 2

2	1	4	9	0	3	5	8	7	6
i							j		

2	1	4	9	0	3	5	8	7	6
			i			j			

Swap 9 and 5

2	1	4	5	0	3	9	8	7	6
			i			j			

2	1	4	5	0	3	9	8	7	6
					j	i			

Step3: Swap 6 and 9

2	1	4	5	0	3	6	8	7	9
					j	i			

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)

Questions?

Last Class: Trees

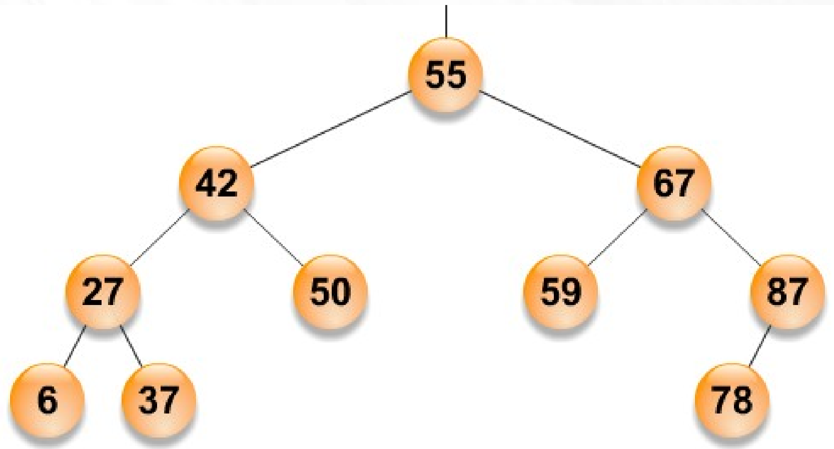
- Collection of _____ and _____
- Any shape, but can't have a _____
- Common operations?
  - 5....

Last Class: Trees

- Collection of nodes and edges
- Any shape, but can't have a loop
- Common operations:
  - Enumerating
  - Searching for an item
  - Adding/Deleting items
  - Pruning/Grafting
  - Balancing

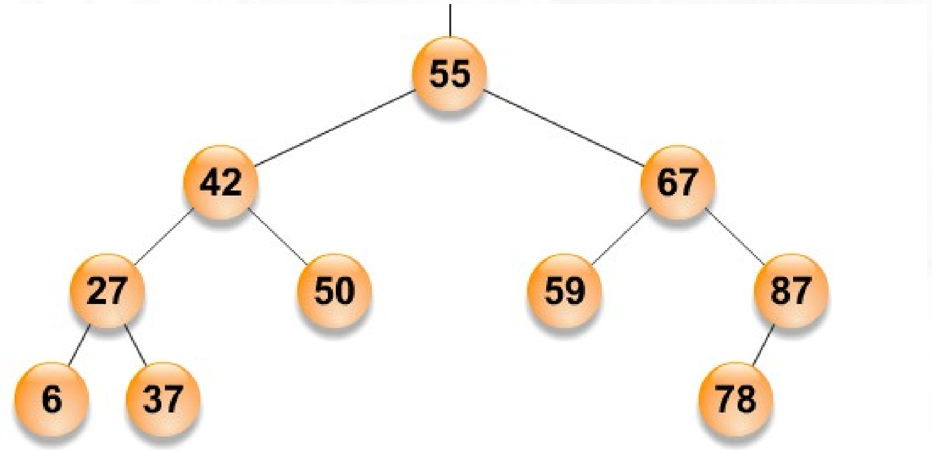
Examples

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



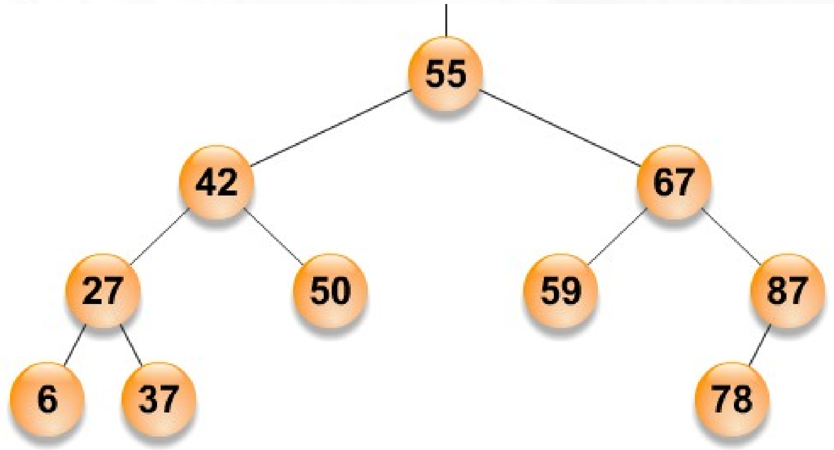
Examples

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



Examples

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



Questions?

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)
- Linked Data Structures
  - easy to add, remove, and swap around parts of the tree

Tree Implementations
Array vs Linked

- Generally prefer linked data structures as they offer the most flexibility (specifically inserting and removing) but can become unbalanced.
- Thought: Nice constants for arrays. Is it possible to use an array to implement a tree that remains efficient (balanced)?
  - Yes, if we limit the operations
  - Specifically, priority queue operations allow an array implementation that is very efficient
  - Called a binary heap

Questions?

Trees With Rules

- How do we find things in a tree?
- We'd like to search from the root
  - Like we started from “head” in a list
  - If we do that...
    - “root” pointer for linked trees
    - Index 0 for array-based trees
- How do we do this? Rules!

Binary Tree

- each parent can only have two children
- number of nodes n in a binary tree

  between $h+1$ and $2^{h+1}-1$

  - h is the height of the tree
- number of internal nodes in a complete binary tree of n nodes

  $\lfloor n/2 \rfloor$
- height of a **balanced** binary tree

  $\lfloor \lg(n) \rfloor$

Binary Tree Storage: Arrays

- Root at index 0
- Children at index:
  - parentIndex*2+1
  - parentIndex*2+2
  - e.g. root at 0, children of root at index 1 and 2
- Parent at index
  - $\lfloor(\text{childIndex}-1)/2\rfloor$
  - e.g. parent of item at index 2 = $\lfloor 1/2 \rfloor = 0$

Binary Tree Storage: Links

- (optional) key
- value
- link to child 1
- link to child 2
- (optional) link to parent
  - why is this optional?

Binary Tree Operations

- Adding/Deleting items
- Pruning/Grafting sections
- Balancing (to provide performance guarantees)
  - later
- Enumerating & Searching
  - later

Questions?

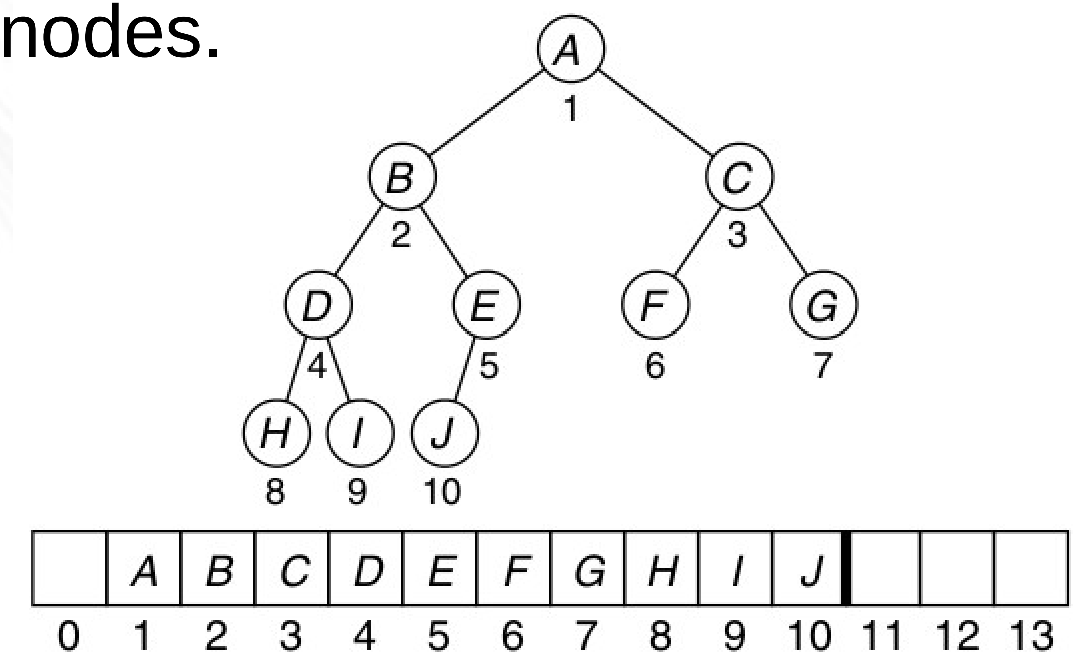
Heaps

- Also a tree!
  - usually binary when learning... there are others
- Relationship maintained between...
  - parent and child
  - e.g. min and max heaps
- Removing items
  - removes the root (“top” item)
- Items “bubble” up and down to maintain order
- Maintains two properties
  - Structure property
  - Heap order property

Structure Property

Weiss 21.1.2

- Want the logical tree represented by the heap to be balanced
  - A “complete binary tree” is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right and has no missing nodes.



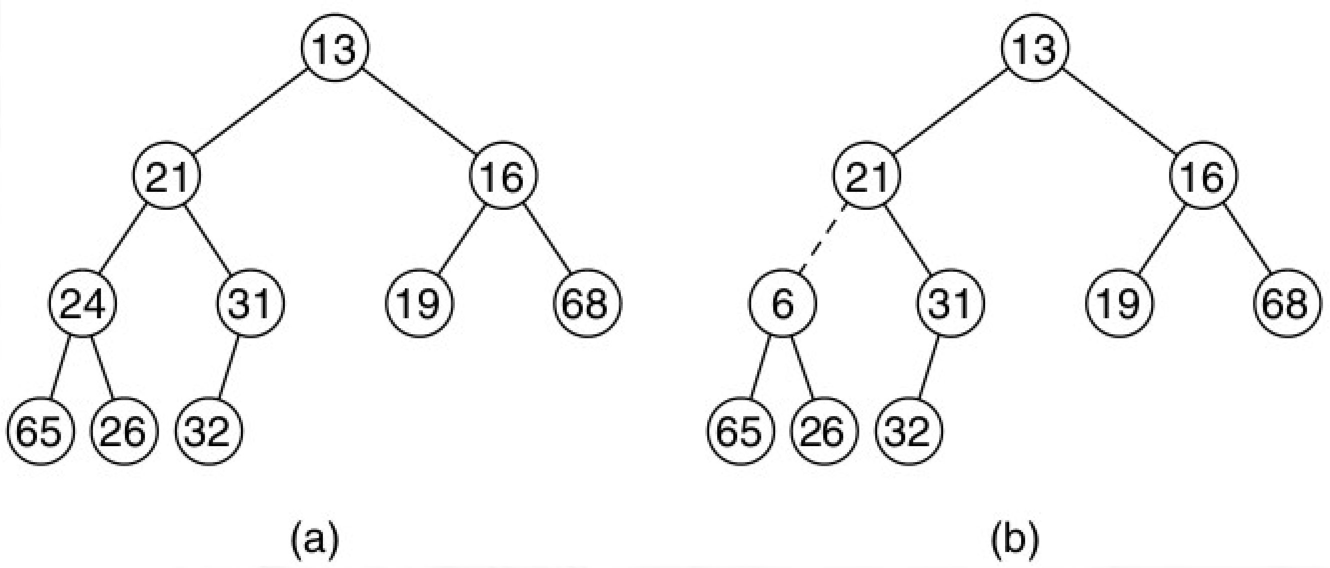
Heap Order Property

Weiss 21.1.2

- Want to find max (or min) quickly so keep at the root of the tree
- Repeating this idea, every subtree should have the subtree root as the largest item in the subtree (i.e. than any descendants)
- Heap-order Property
  - In a heap, for every node X with parent P, the key in P is larger than or equal to the key in X.
- Detail: Note we use a sentinel value at index [0], makes calculations less messy

Heap Order Property

- Do these satisfy the (min) heap order property?



Weiss Figure 21.3

Heap Demo

http://people.ksp.sk/~kuko/gnarley-trees/

- Min/Max
  - java -jar gt.jar

(Max) Heap Implementation

- Insert by adding to the end of the array, which likely violates the heap order property.
  - Increment the size
  - Fix by swimming the new last value
- Delete (the max value) by swapping the last value and the root value
  - Null out the last value, prevent loitering
  - Decrement size
  - Sink the new root value

(Max) Heap Swim

Weiss a.k.a. percolate up, Sedgewick/Wayne 2.4

- Scenario: Heap order violated, child greater than parent (possibly ancestors)
  - Peter principle, rise to highest level

```
private void swim( int k )
{
    // Look at parent, if we are greater, swap
    while( k > 1 && less(k/2, k) )
    {
        // swap parent and child
        swap(k/2,k);
        k = k/2; // move up
    }
}
```

(Max) Heap Sink

Weiss a.k.a. percolate down, Sedgewick/Wayne 2.4

- Scenario: Heap order violated, parent less than at least one child (possibly both)
  - Power struggle, better subordinate promoted

```
private void sink( int k )
{
    // While we have another level of children within size
    while( 2*k <= this.size ) // equals is important to get right child
    {
        // Find larger of children, exchange with them
        // Why? Know after swap heap invariant met for k
        int j = 2*k; // left child index
        // Be careful, don't look at right child if null
        // We know we have a left child, use size instead of null
        if( j < this.size && less(j, j+1) ) j++;
        // Now j points to largest child
        if( less(k, j) == false ) break;

        // Otherwise, swap and continue sink
        swap(k,j);
        k = j; // move down
    }
}
```

What can we use this for?

- Hint... two things we've already seen!
  - Priority Queue
  - Sorting

What can we use this for?

- Priority Queues!
  - maintain order by “priority”
  - highest priority at the top
  - removing an item puts the next highest priority at the top

Priority Queue Summary

- Binary heap supports insertion and deletion of the max (min) item in logarithmic worst-case time. Uses an array, easy to implement, and elegant. Often best choice.

Operation Implementation	insert	delMax	findMax
Unordered Array	1	N	N
Ordered Array	N	1	1
Binary Heap	lg(N)	lg(N)	1
???*	1	1	1

Impossible: Lower bounds (omega) for compare sorting is N lg N. If O(1), then heap sort O(N).

What else can we use heaps for?

Weiss 21.5

- Client provides array not heap ordered
- Sorting! Two steps
  - Build heap, referred to as “heapify”
  - Remove max/min to attain sorted order
- Keep big-O in mind for this...

Heapsort

- Demo

- Resource with animations if you forget this:
  - http://en.wikipedia.org/wiki/Heapsort

Big-O!

- heapify()
  - repeatedly insert items? O(n log n)
  - heapify? O(n)
- delMin/Max() … n times
  - O(n log n)
- that makes heapsort O(____)?

Why is heapify() O(n)?

- Count the work done at each level...
  - at the bottom there are 2^h nodes
    - we do not do anything, so the work is 0
  - 2nd level has 2^{h-1} nodes
    - each might move down (at most) 1 level
  - 3rd level has 2^{h-2} nodes
    - each might move down (at most) 2 levels

Build (Max) Heap

Weiss 21.3

- Use bottom up approach ensures lower sub-heaps have already been heapified
- Thus, only need to sink root for sub heaps
- The leafs are trivially heaps, so skip

```
private void buildHeap( )
{
    // Bottom up, starting at first non-leaf
    for( int i = this.size / 2; i > 0; i-- )
    {
        sink( i );
    }
}
```

Complexity of Heapify Methods

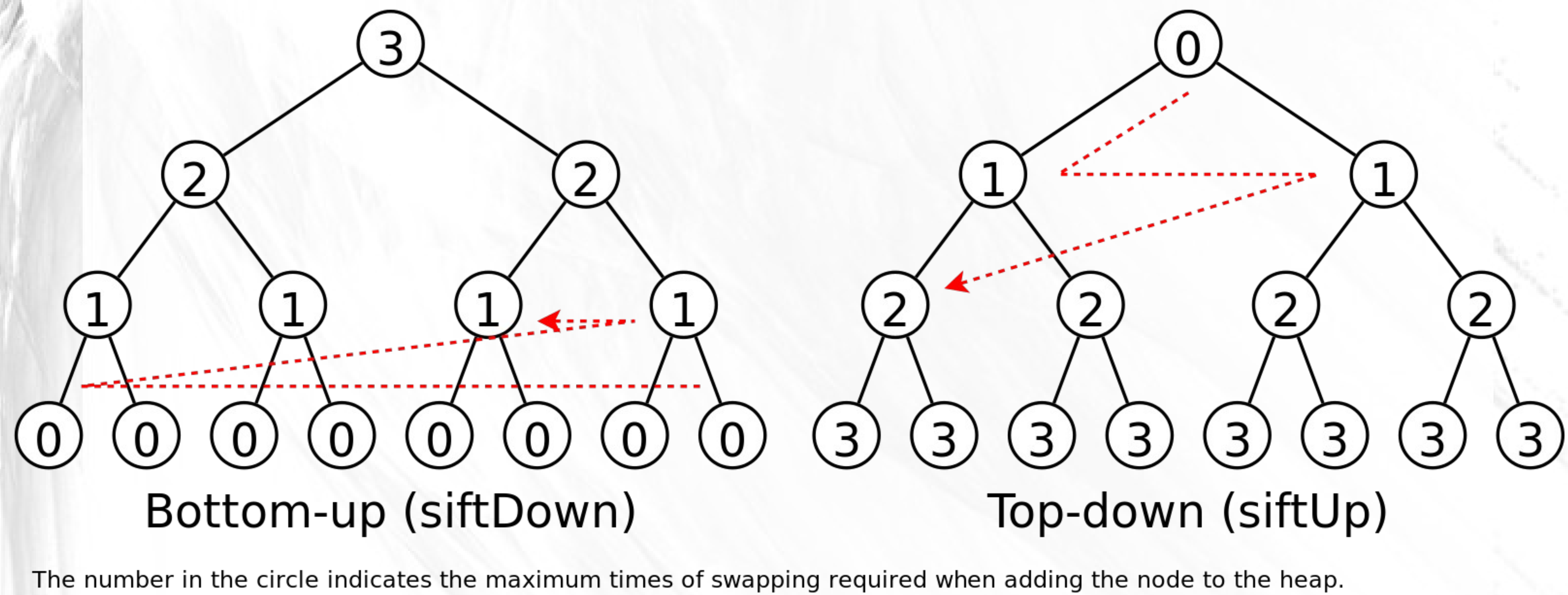


Image Source:
http://commons.wikimedia.org/wiki/File:Binary_heap_bottomup_vs_topdown.svg#mediaviewer/File:Binary_heap_bottomup_vs_topdown.svg

Why is heapify() O(n)?

Weiss 21.3

- Theorem 21.1: For a perfect tree of height H containing N = 2^{H+1} - 1 nodes, the sum of the heights of the nodes is N - H - 1.
- Confirmed by inspection on previous heap.
- Algebraically:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}}\right)$$
$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h}}\right)$$
$$= O(n)$$

Analysis of Heapsort

- Worse Case / Best Case
  - O(n log n)
- Space complexity / In-place?
  - Have to delete and put into second array, so it requires additional O(N) memory?
  - Clever solution, delete from heap, position opens at the end of the array, place there. Max Heaps produce ascending order
- Unstable
  - relative order of equal elements not preserved

Sorting Summary

- Heap sort is in place and guarantees N lg N performance, so why not used more?
- Heap sort poor cache relative to merge/quick (compares with values far apart).

Operation Implementation	worst	average	best	in place O(1)	stable	remarks
Selection Sort	N^2	N^2	N^2	yes	no	never use
Insertion Sort	N^2	N^2	N	yes	yes	small n
Merge Sort	N lg N	N lg N	N lg N	no	yes	extra memory
Quick Sort	N^2	N lg N	N lg N	yes*	no	fast practice
Heap Sort	N lg N	N lg N	N lg N	yes	no	poor cache
???	N lg N	N lg N	N lg N	yes	yes	Unknown

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

Questions?

Tree Traversals

Weiss 18.4 Described Iterators, Today we describe operations

- Two common types
  - breadth first
  - depth first
- Three common depth first
  - In-order
  - Pre-order
  - Post-order
- Level-order is breadth first

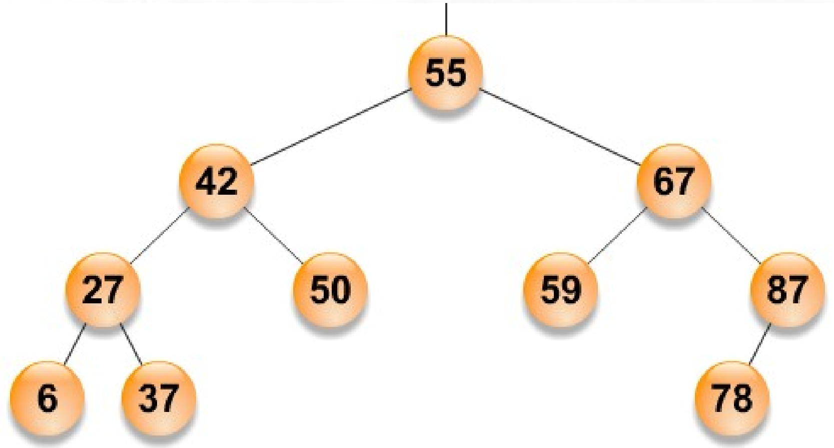
Tree Search Order Property

Weiss 19.1

- A useful property is to have the tree nodes stored such that
  - all keys less than a node's key are in the left subtree and
  - all keys greater than a node's key are in the right subtree
- Search order property
  - Typically used with binary trees, but can apply to other trees as well
- Makes searching tree similar to binary search in an ordered array

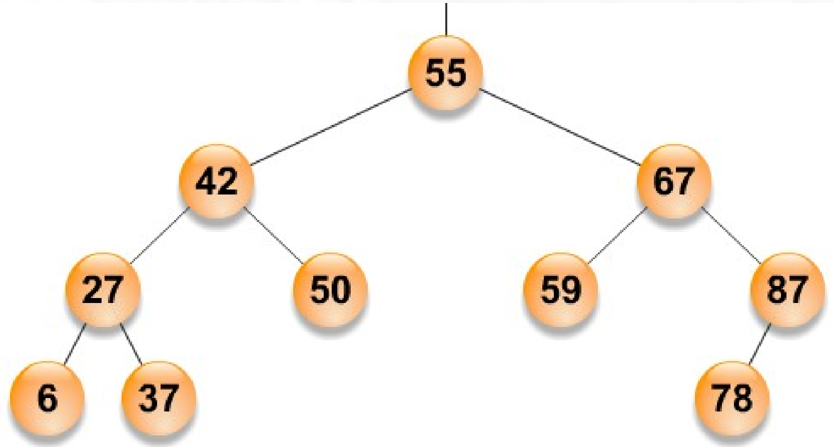
Tree Traversals: Pre Order

- process order 55, 42, 27, 6, 37, 50, 67, 59, 87, 78
  - self
  - left children
  - right children



Tree Traversals: Post Order

- process order 6, 37, 27, 50, 42, 59, 78, 87, 67, 55
  - left children
  - right children
  - self

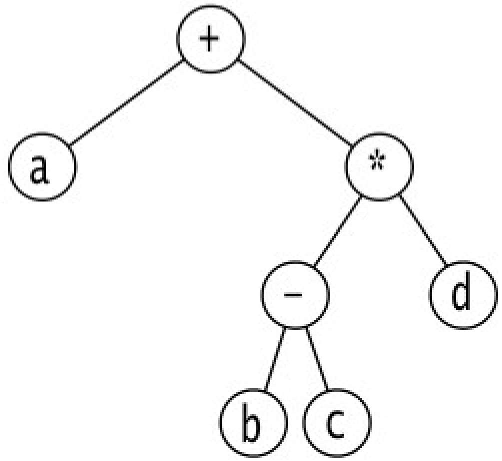


Tree Traversals: Post Order

https://en.wikipedia.org/wiki/Stack_machine

- Conversion from syntax tree into stack machine instructions

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



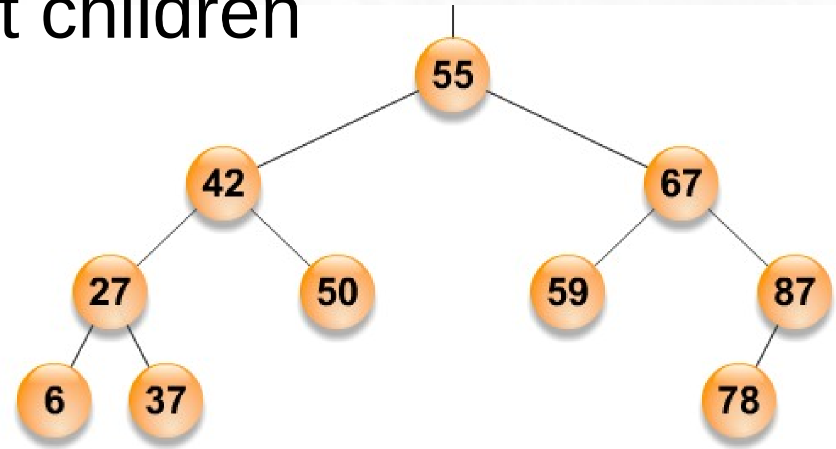
Weiss Figure 18.11

Operations act on top two operands and push result back onto stack

push a
push b
push c
minus
push d
multiply
add

Tree Traversals: In Order

- process order 6, 27, 37, 42, 50, 55, 59, 67, 78, 87
  - left children
  - self
  - right children



If tree satisfies search order property,
Let gravity generate sorted items

Questions?

Next Week: Midterm!

- Schedule:
  - 7:20-8:50 Midterm
  - 9:00-10:00 Lecture
- Bring photo ID!
  - GMU ID or Driver's/Walker's License

Midterm: What have we covered?

- Analysis
  - Big O (Omicron), Big Ω (Omega), Θ (Theta), Little o (Omicron), Little ω (Omega)
  - Worst, Average, Best Case
  - Amortized Analysis
- Data Structures
  - Arrays, Dynamic Arrays
  - Stacks, Queues, Priority Queues
  - Linked Lists, Trees, Heaps

Midterm: What have we covered?

- Programming Topics
  - recursion
  - iterators
- Searching/Sorting
  - binary search arrays
  - tree traversals
  - insertion sort, merge sort, quick sort, heap sort

Practice Problems Templates

- What is the big-O of ___ and why?
- What is the difference between ___ and ___?
- When/why would you use ___?
- In class we did ___, explain why/how.
- Describe the algorithm for ___.
- Do algorithm ___ on the following data: ___.

Questions?

Group Practice: Big-O

- Get with a partner
- Each person take out a piece of paper your partner can write on
- Write 5 functions on it
- Trade papers
- Write the Big-O of each function on your new paper
- Together with your partner, form a single list ordering them by computational complexity

Group Practice: Traversals

- Draw a 5 node tree on your piece of paper
- Pick one of the three tree traversals we did and write that on a paper (pre-order, in-order, post-order)
- Trade papers
- Perform the requested traversal on your partner's tree
- Go over it together

Group Practice: Sorting

- Get with a partner
- Get a piece of paper you can write on
- Write 4 numbers on it
- Look at your partner's paper and add his 4 numbers to your list
  - So that you both have the same list!
- Each person do insertion sort on it
- Compare your results

Group Practice: Sorting

- Get in groups of three

Group Practice: Sorting

- Pick who is person 1, who is person 2, and who is person 3
- Person 1: describe the algorithm of quick sort to your group
- Person 2: describe the algorithm of heap sort to your group
- Person 3: describe the algorithm of merge sort to your group
- You will each have 3 minutes

Group Practice: Linked Lists

- Get with a partner
- Get a piece of paper you can write on
- Draw a single linked list 6 nodes long used to store shapes
  - circles, squares, triangles
- Trade papers
- On your new list, show your partner how you would search for a circle
- Show how you would find the last triangle in the list

Group Practice: Heaps

- Get with a partner
- Get a piece of paper you can both write on
- Decide who is person 1 and who is person 2
  - Person 1: pick a number
    Person 2: insert it into a heap
  - Person 1: pick a number
    Person 2: insert it into a heap
  - Person 1: pick a number from the heap
    Person 2: remove that number
- Repeat the above on the same heap, switching who is person 1 and who is person 2

Week After Next Week

- Recess Break

Free Question Time!