COSC160: Data Structures
Heaps

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

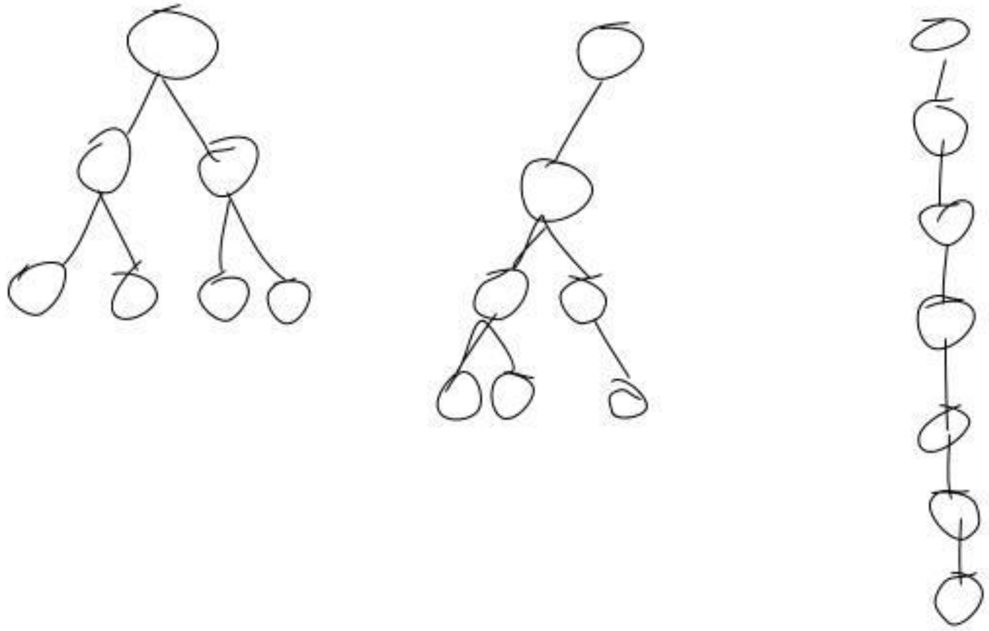
Outline

I. Priority Queue
II. Heaps
    I. Binary Heaps
    II. Skew Heaps

Balancing a Tree

- Binary trees lack a depth constraint. As a result the worst case insertion, removal and retrieval times are O(n).

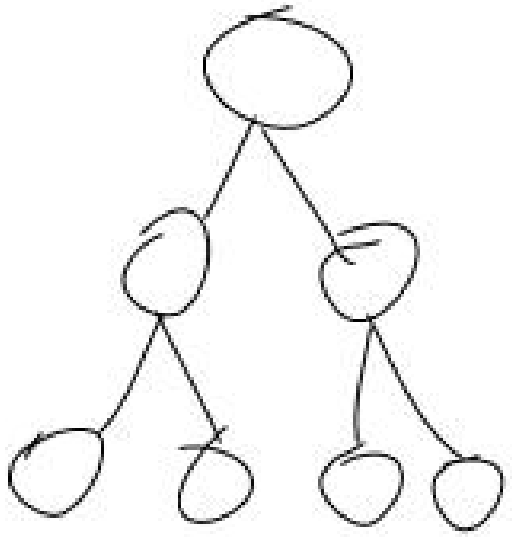
Some interesting cases of a
tree with n nodes



GEORGETOWN UNIVERSITY

Balanced Trees: Time Complexity Implications

- If a tree is “balanced” in some sense. The time complexity of insertions, removals and retrievals may have a logarithmic bound.



Heaps

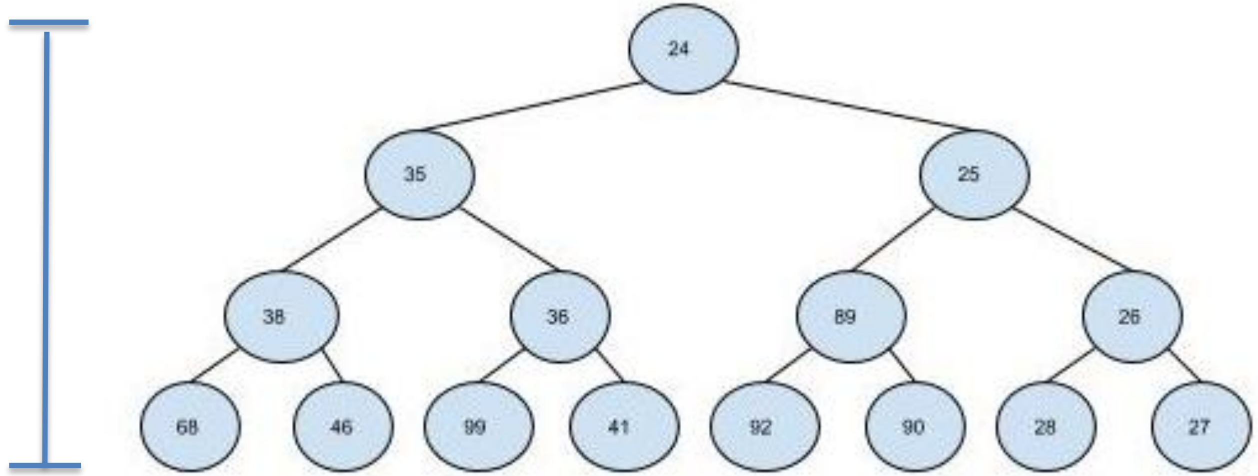
- Another tree structure well suited for a priority queue is a Heap.
  – Similar to other BSTs, BUT key order is determined by parent-child relationship alone, not necessarily which child (left and right child have no order constraint).

- A binary **heap** of size n, is a complete, binary tree where key order is determined by the heap-order property.
  – A complete binary tree of size n has a height of $\lfloor \log n \rfloor$ and the tree is completely filled up to depth $\lfloor \log n \rfloor - 1$.
  – A complete tree, has all nodes at the maximum depth in leftmost position.
    - If the tree was implemented as an array and ordered by a BFS, all nodes would be contiguous (no vacant spaces until BFS is complete).
  – (min) Heap-order: a parents key is less than its children's key
    - min-heap or max-heap

Heap Example: Heap Order

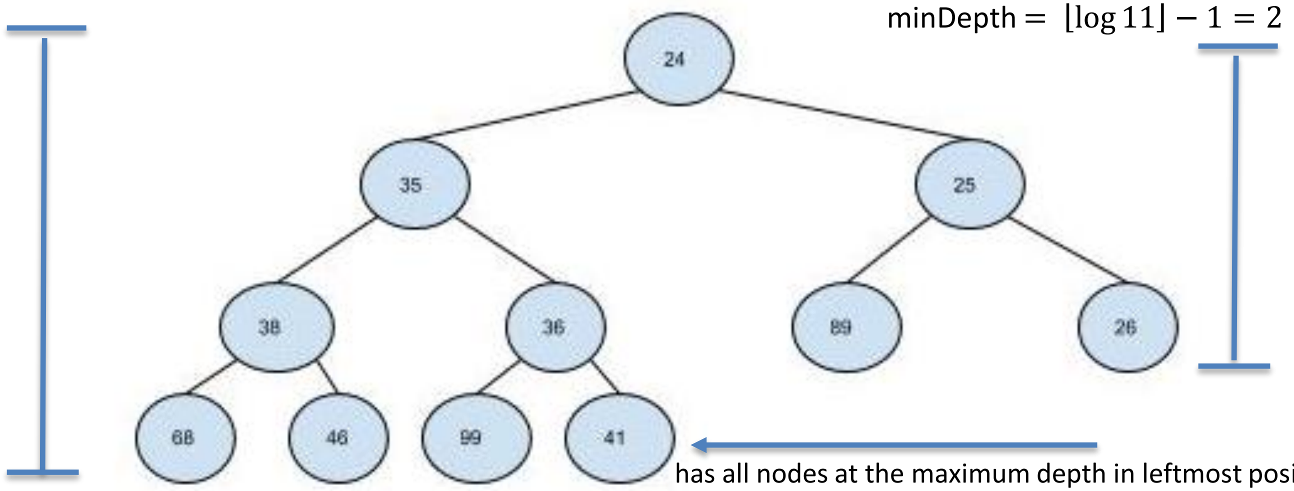
- (min) Heap-order: a parents key is less than its children's key

- Structural constraints
  - A complete binary tree of size n has a height of $\lfloor \log n \rfloor$ and the tree is completely filled up to depth $\lfloor \log n \rfloor - 1$.
  - A complete tree, has all nodes at the maximum depth in leftmost position.
    - If the tree was implemented as an array and ordered by a BFS, all nodes would be contiguous (no vacant spaces until BFS is complete).



$h = \lfloor \log 15 \rfloor = 3$

minDepth $= \lfloor \log 11 \rfloor - 1 = 2$

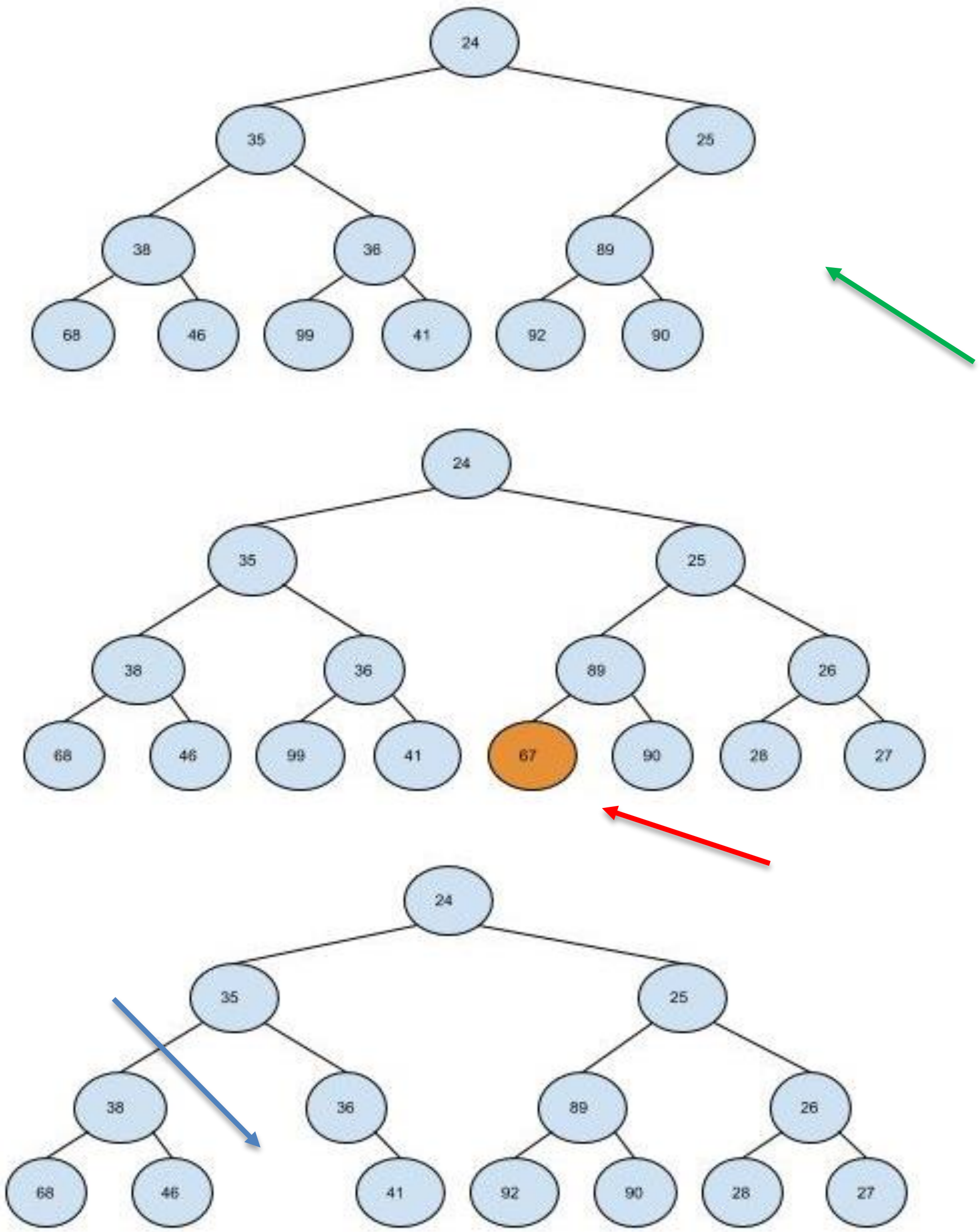


has all nodes at the maximum depth in leftmost position.

$h = \lfloor \log 11 \rfloor = 3$

Not Binary Heaps

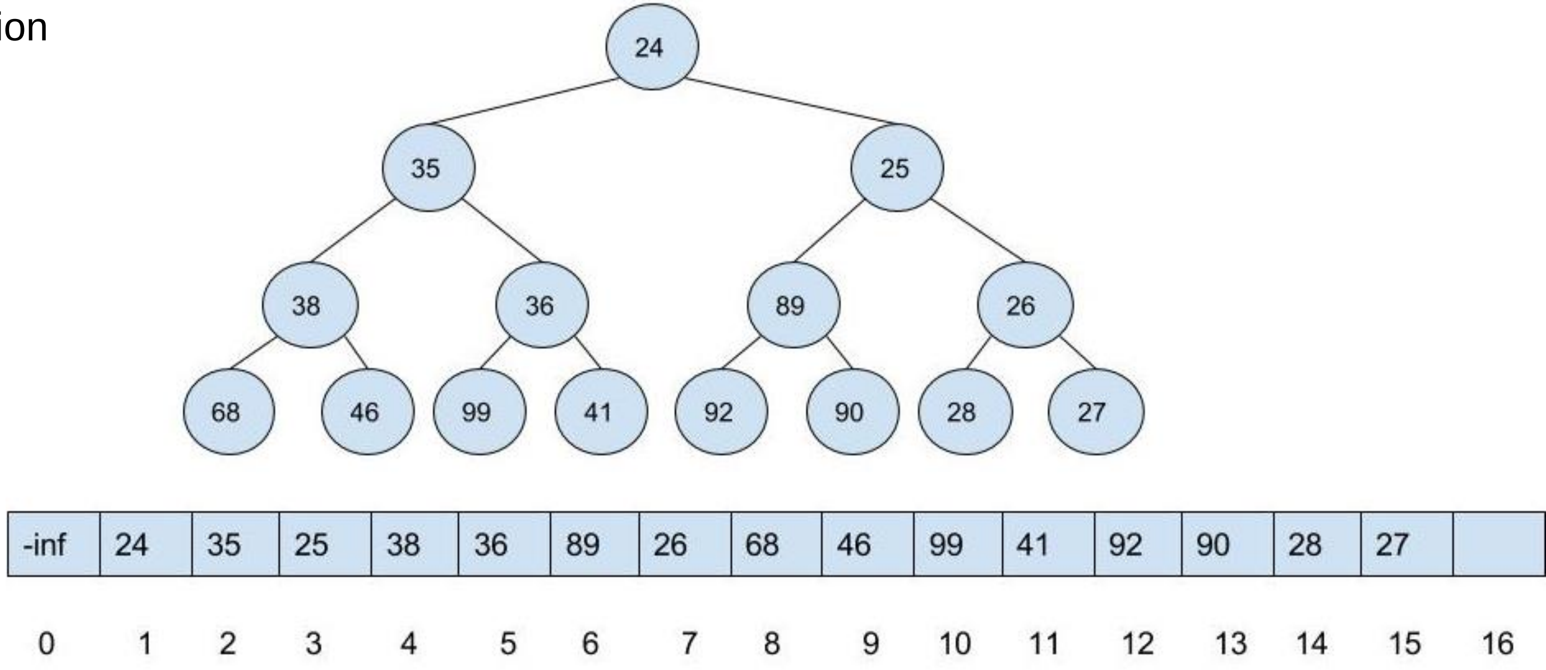
- Violations of
  - Min filled depth
  - Heap order
  - Leftmost position



Implementation

- Given that the heap is complete in leftmost position, it is reasonable to implement as an array.
  - In the simple case, no need for left and right child pointers
  - Con: static maximum size (use dynamic array)
  - Attributes:
    - Array (for keys / priorities)
      - Some use sentinel at 0^{th} position
    - currentSize
    - maxSize
    - inOrder

- Chaining implementation
  - Binary Tree Node

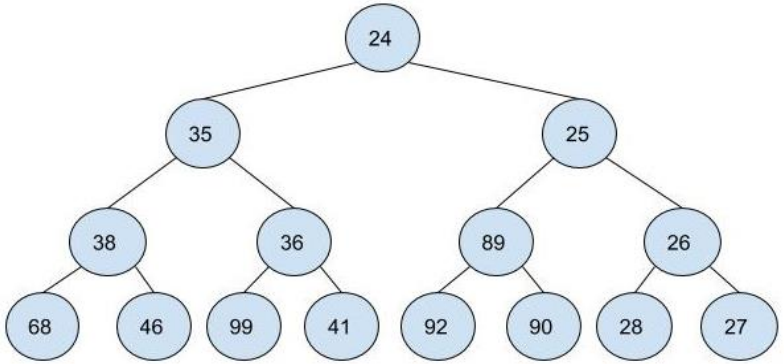


Traversing an array implementation of a tree

- Assume a (BF) level-ordering of keys in the array.

- Review and Examples of array implementation of trees:
  - Implement a method to traverse and print keys in BF order
  - Implement a method to traverse and print keys in a DF order
    - Observations
      - Doubling parentIndex: leftChild index
      - Double parentIndex + 1: rightChild index
      - childIndex/2 : parentIndex
        - » Assume floor when odd

Try this at home!



-inf	24	35	25	38	36	89	26	68	46	99	41	92	90	28	27	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Insertion

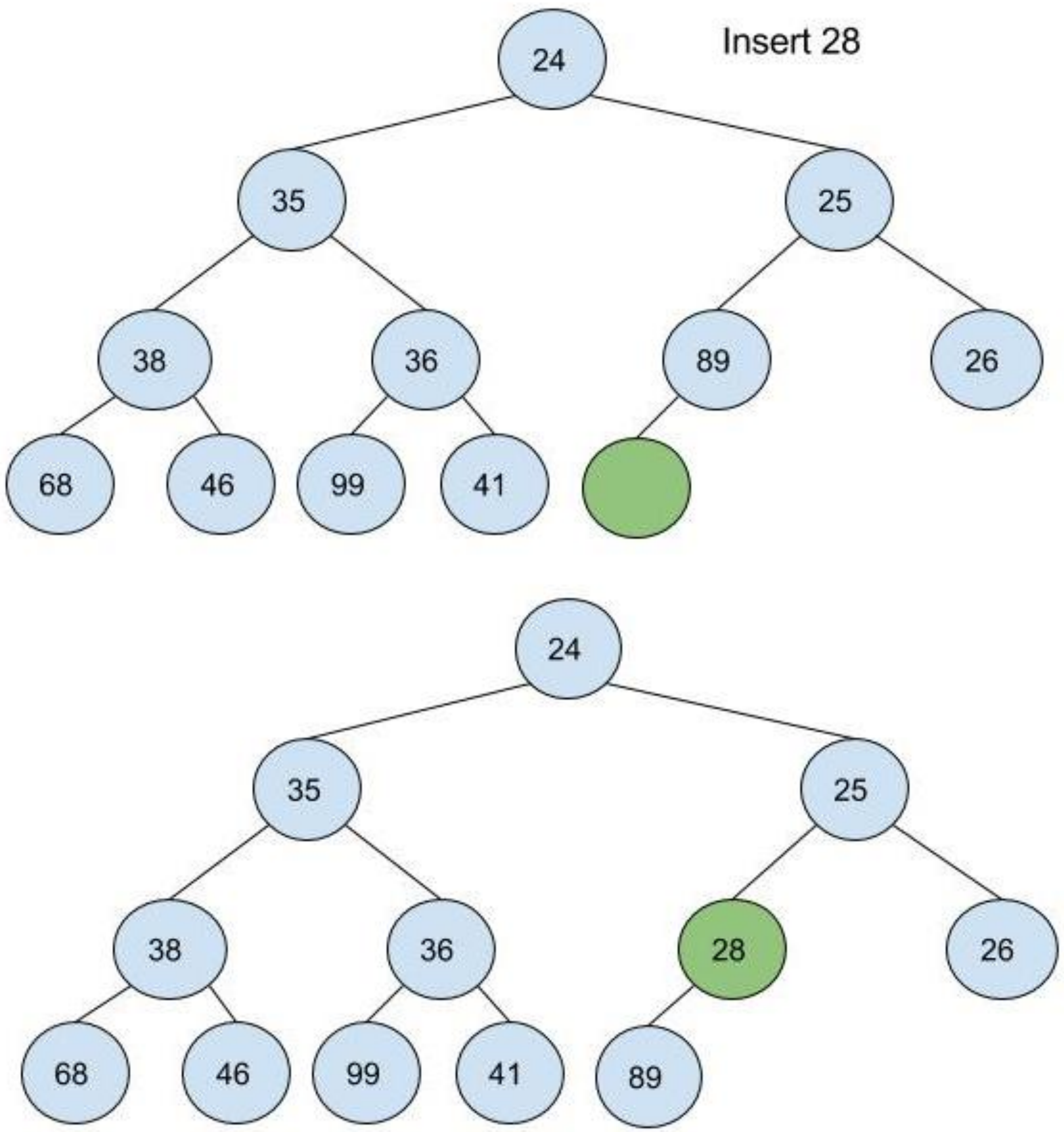
- Conceptually
  1. Insert the key k in the leftmost available position at height $\lfloor \log n \rfloor$.
  2. If heap order is not violated by k, then done.
  3. Else, swap k with parent key
  4. Repeat at step 2.

- Implementation
  – Inserting and swapping occurs in array.
  – Must be able to determine parents index.

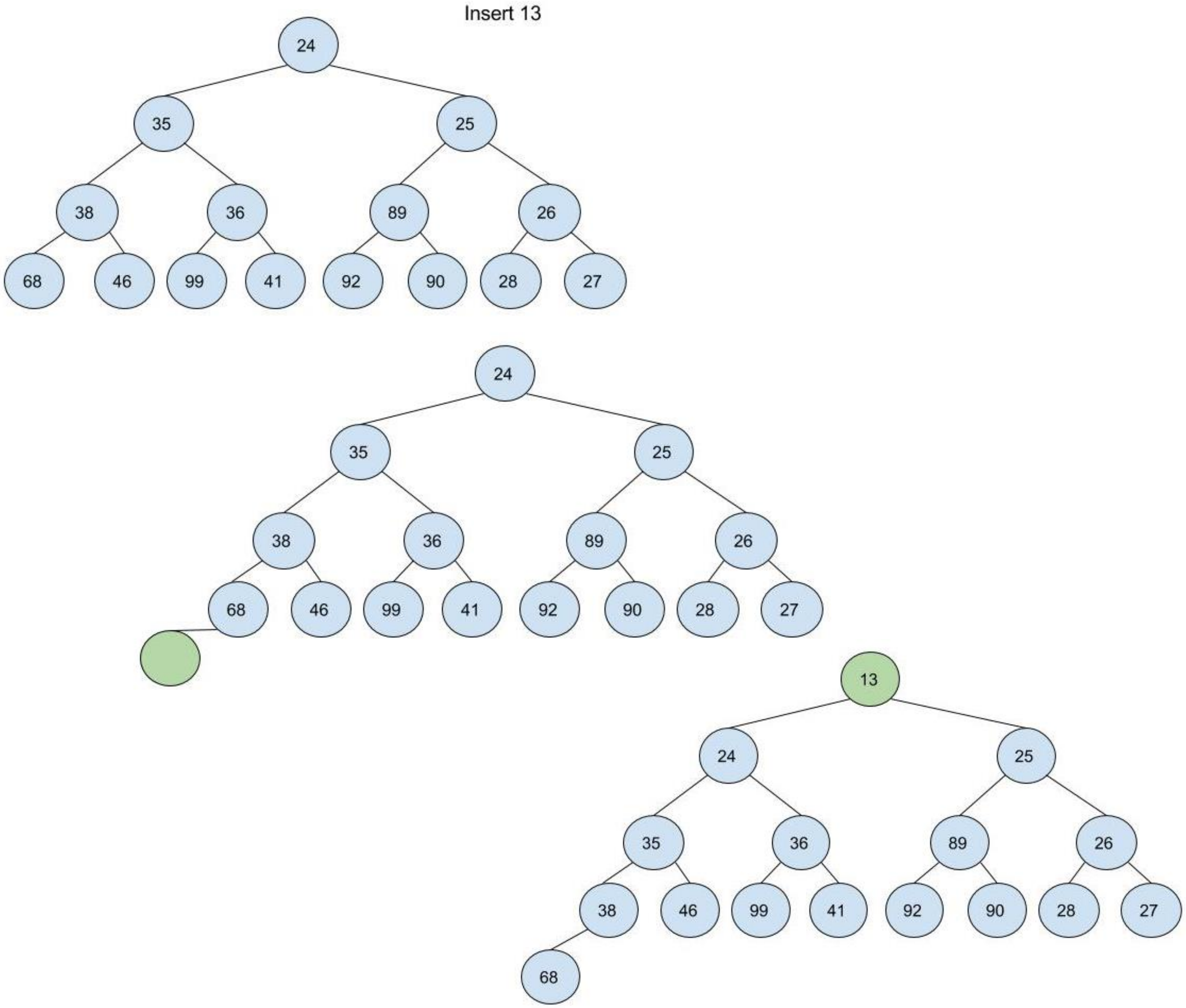
Heap Insertion

// Assumes array size sufficient
// insert hole into next leftmost location
// swaps hole up to maintain heap order
// inserts val in hole

$function\ insertHeap(heap, val)$
    $hole := heap.currentSize := heap.currentSize + 1$
    $while\ val < heap.array[hole/2]$ // swap keys/priorities down
        $heap.array[hole] := heap.array[hole/2]$
        $hole := hole/2$
    $heap.array[hole] := val$ // inserts val at stopped hole location



Insertion Example



1. Insert hole in leftmost position

2. Repeatedly swap up until insertion does not violate heap order

3. Then insert

Removal

- Find min
  - The min value (in a minHeap) is simply the root of the tree (if heap is inOrder)

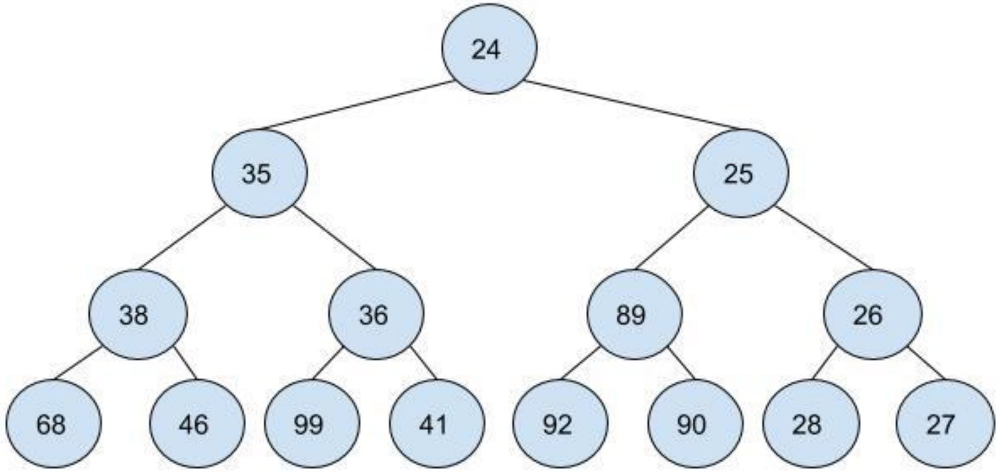
- Remove Min
  - The item to remove from the priority queue is the item with “highest” priority (always the root in a heap!)
  - The “last” item is detached (for future insertion at hole) since the number of elements is decremented
  - Result will be a tree with a hole for a root
    - Fix: propagate hole down and insert the “last” item into the hole when possible
    - “last” refers to order in a BF scan.

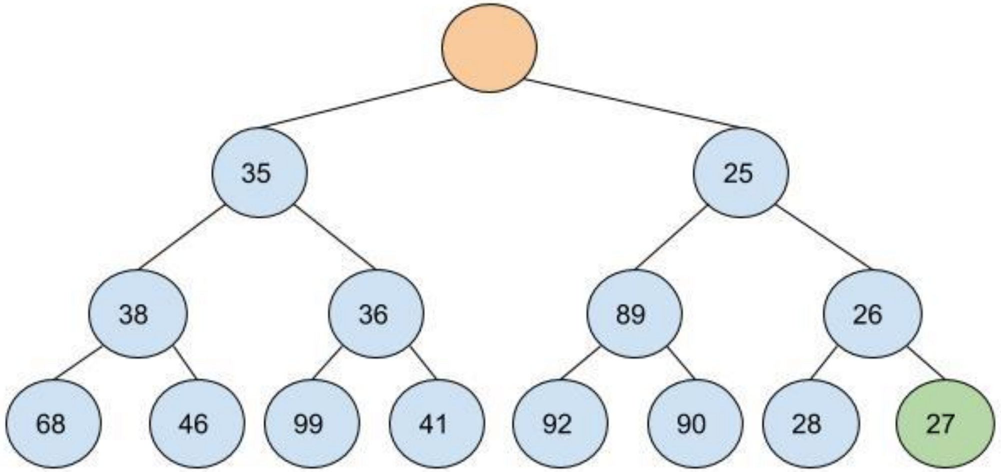
Heap Removal Example

1. Remove root (always root if heap is inOrder)

2. Detach “last” item from tree to insert into new location
   – Last in BF scan
   – Right most item at max depth

3. Swap hole down until we can insert the “last” item



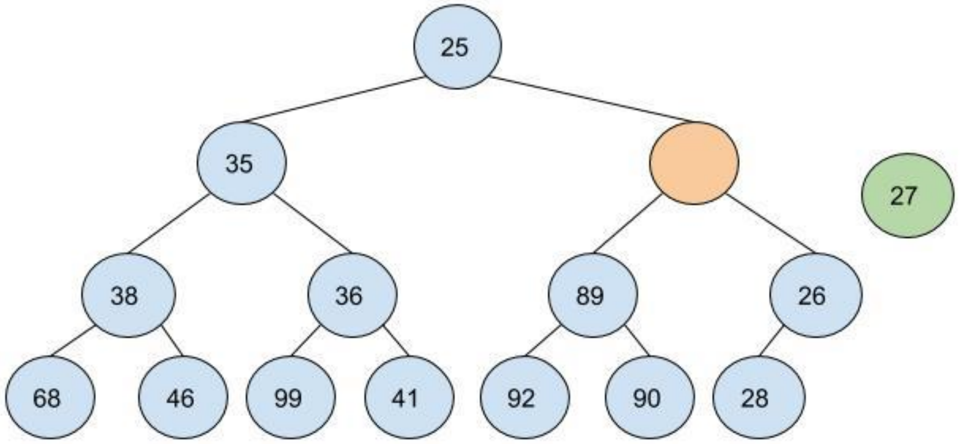


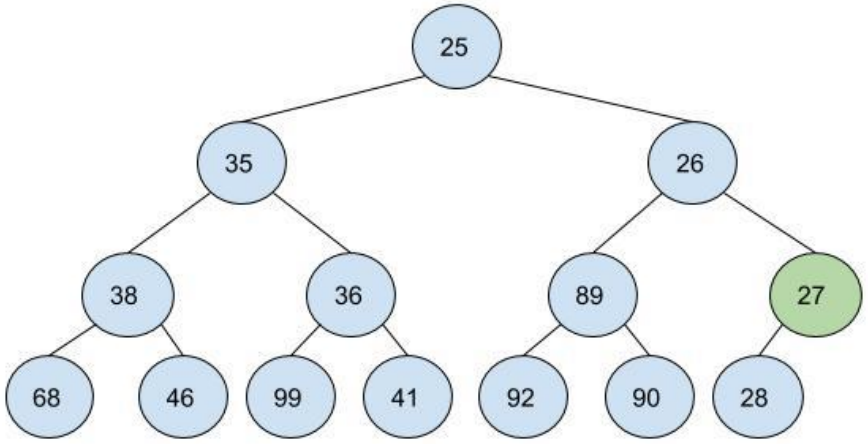
Heap Removal Example (cont)

1. Remove root (always root if heap is inOrder)

2. Detach “last” item from tree to insert into new location
   - Last in BF scan
   - Right most item at max depth

3. Swap hole down until we can insert the “last” item





Heap Removal

```
// Removes item with highest priority
// detaches lastItem
// Swaps hole down and reinserts lastItem so heap is inOrder

function Remove(heap)
        returnVal := heap.array[1]
        temp := heap.array[heap.currentSize]
        heap.currentSize := heap.currentSize – 1
        hole := 1
        while hole <= heap.currentSize  // while there exists a child , traverse down
                child := hole * 2   // check left child
                if child != heap.currentSize  AND heap.array[child+1] < heap.array[child]  // swap down to lesser of two children
                        child := child + 1
                if heap.array[child] < temp   // not ready to insert … continue swapping down
                        heap.array[hole] := heap.array[child]
                else
                        break // ready  to insert
        heap.array[hole] := toInsert
        return returnVal
```

RE-write removal using helper function

```
function swapDown(heap, hole)
    temp :=  heap.array[hole] // assumes last item was copied to hole location
    while hole <= heap.currentSize  // while there exists a child , traverse down
        child := hole * 2   // check left child
        if child != heap.currentSize  AND heap.array[child+1] < heap.array[child]  // swap down to lesser of two children
            child := child + 1
        if heap.array[child] < temp   // not ready to insert … continue swapping down
            heap.array[hole] := heap.array[child]
            hole := child
        else
            break // ready  to insert
        hole := child  // update and continue traversal
    heap.array[hole] := temp
```

```
// Removes item with highest priority
// detaches lastItem
// Swaps hole down and reinserts lastItem so heap is inOrder

function Remove(heap)
    returnVal := heap.array[1]
     heap.array[1] := heap.array[heap.currentSize]// copy last item to hold location
    heap.currentSize := heap.currentSize – 1
    swapDown(heap , 1)
    return returnVal
```

GEORGETOWN UNIVERSITY

Analysis of Insert and Remove

- Insert
  1. Add hole to next leftmost position
  2. Swap up hole until insertion does not violate heap order
  - Worst case: O(log n)

- Remove
  1. Replace root with hole
  2. Detach rightmost item at max depth
  3. Swap hole down until insertion of detached item does not violate heap order
  - Worst case: O(log n)

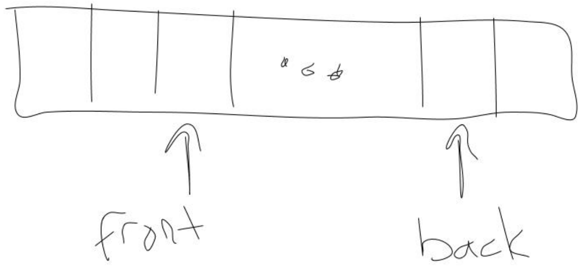
Heap uses

- Observe we have sacrificed “order” for “balance”

- When can we make use of a heap?

- Only partial order matters
  – Min or max

Queue



- FIFO and LIFO
  - Removal from queue is based on order of entry
  - As a result, access can be restricted to head or tail (base or top) only

- Priority Queue
  - Removal is based on priority ranking
  - Tracking priority rankings is necessary to determine removal order

GEORGETOWN UNIVERSITY

Priority Queue

- Using a list implementation
  - Array or linked list
  - Option 1 unordered:
    - Insert: O(1)
    - Remove: O(n)
  - Option 2 ordered:
    - Insert: O(n)
    - Remove: O(1)

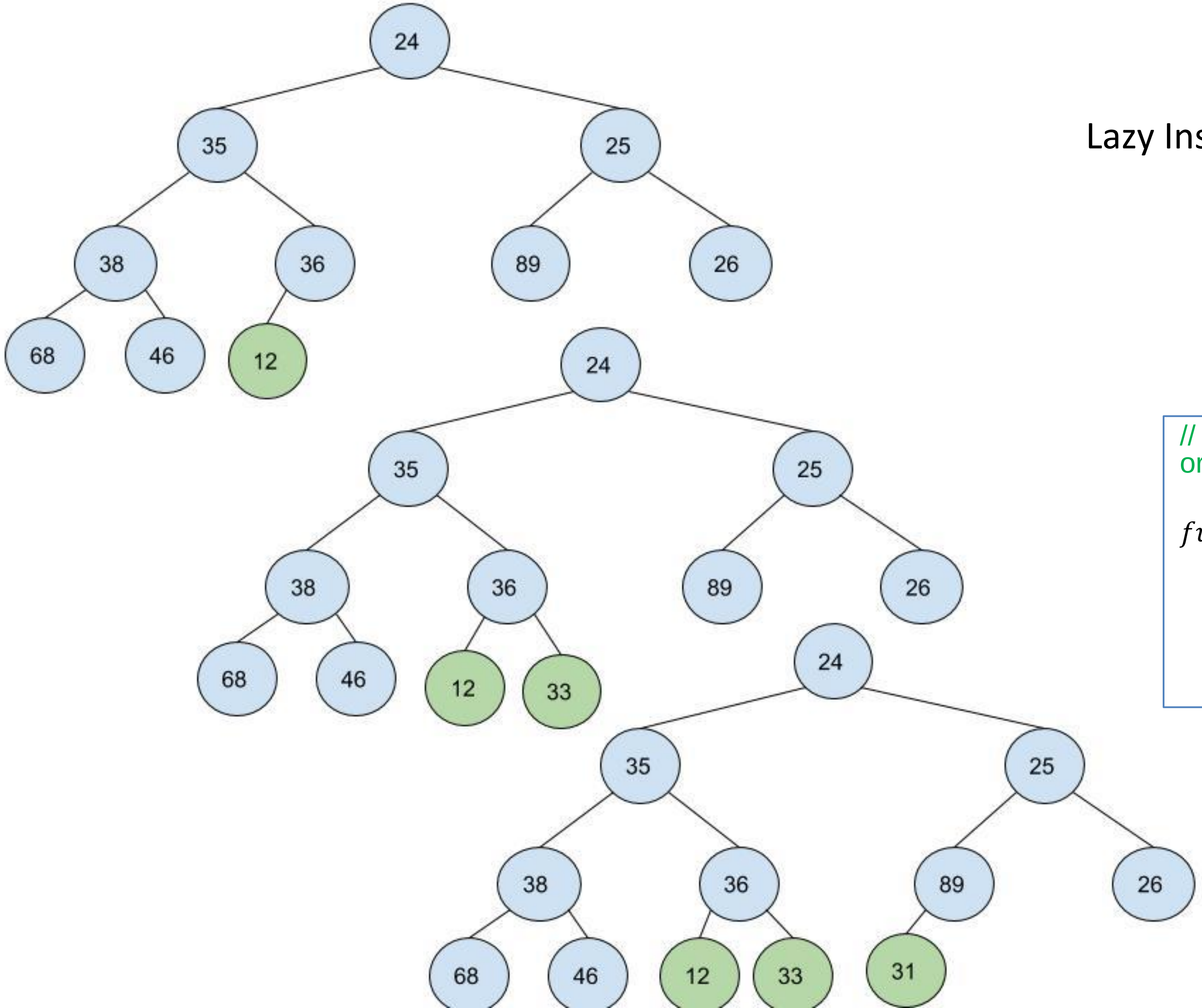
- Using a BST
  - Insert, remove, search: Average case, O(log n)
  - AVL: worst case O(log n) ** (later!)

“Lazy” Insertion: Heap

- O(log n) insertion time is not bad.

- Can we do better … ? Sometimes.
  - Lazy insertion.
    - Simply place the item in the heap, without consideration for heap order.
    - Time: O(1)
    - Why? … If we plan to perform many insertions, before attempting any removals, then why waste time maintaining hash order. Instead delay ordering until the next removal.
    - Before a removal must “fixHeap” aka “heapify” – update heap such that heap order is maintained.

Lazy insert or "tossIn"



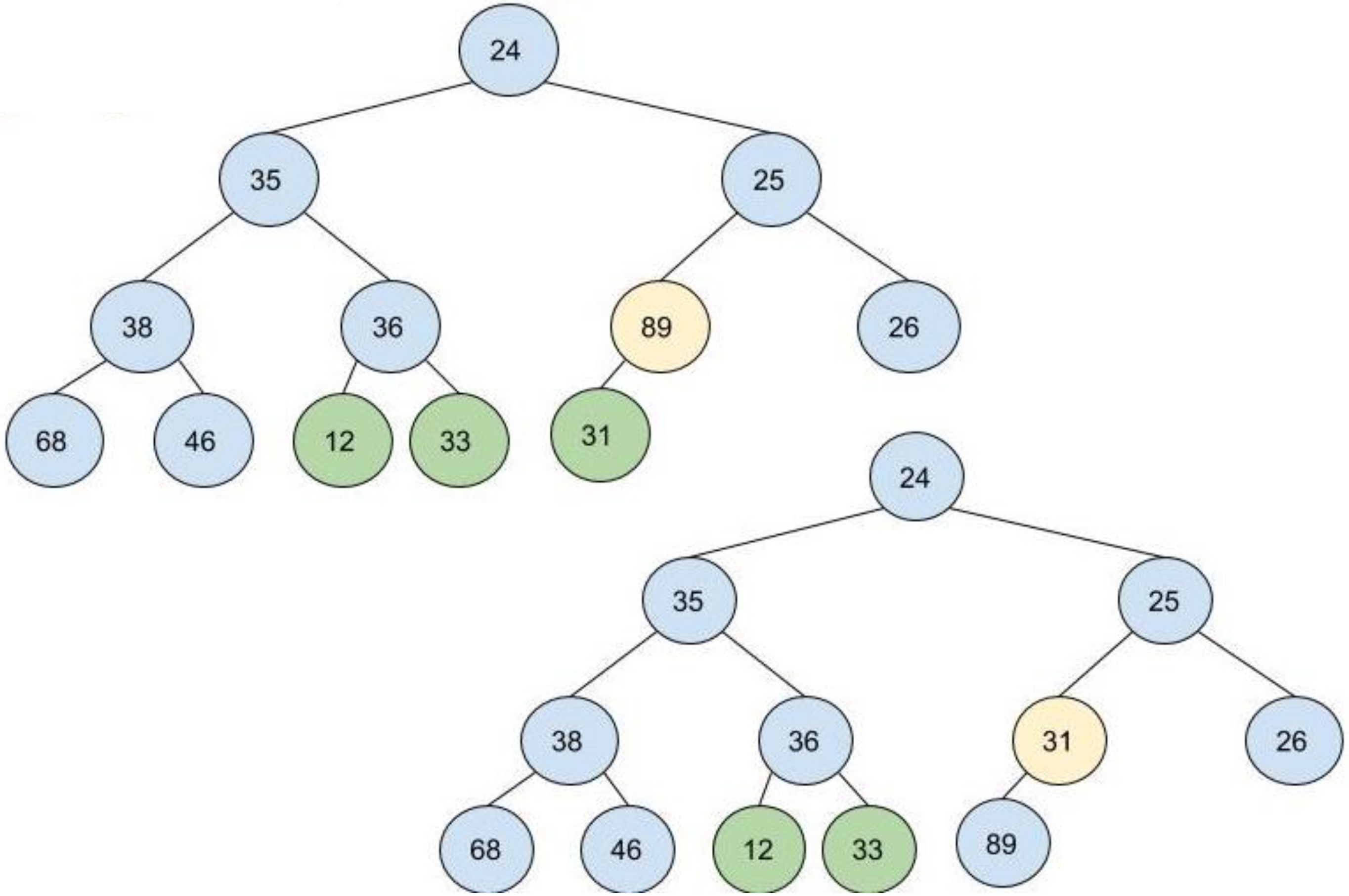
Lazy Insert: 12, 33, 31

```
// insert vals into next leftmost location, no check for heap order

function tossIn(heap, val)
        heap.currentSize := heap.currentSize + 1
        heap.array[currentSize] := val
        if(val < heap.array[currentSize/2])
                heap.inOrder := false // set flag
```

GEORGETOWN UNIVERSITY

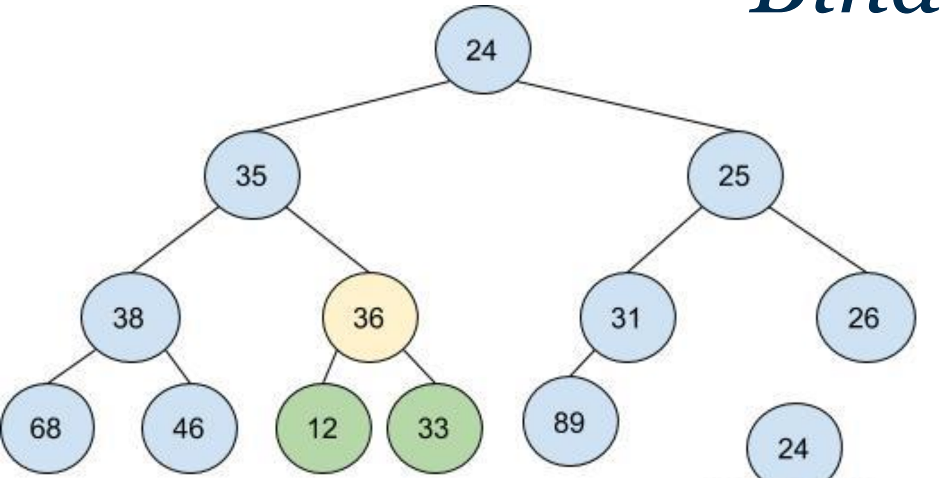
Binary Heap Example: fixHeap aka heapify

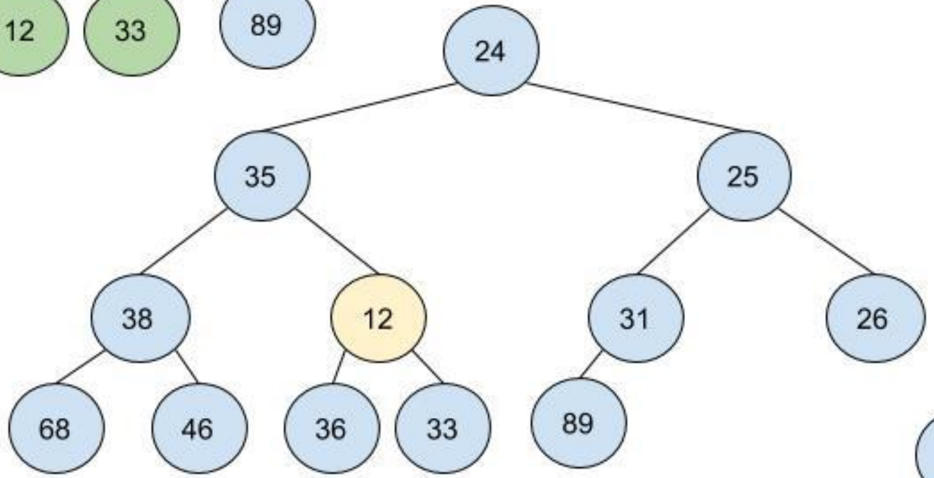


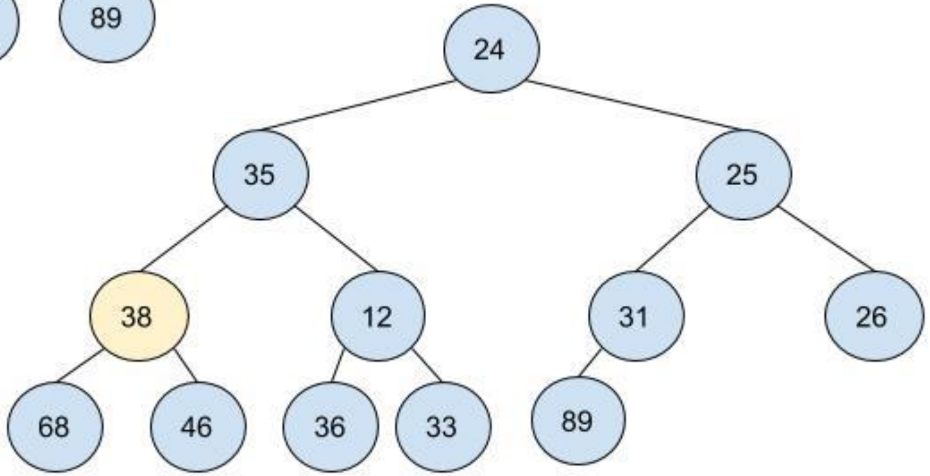
// for use with lazy insert
// repeatedly check with swap down

function fixHeap(heap)
i := heap.currentSize / 2
while i > 0
swapDown(heap, i)
i := i - 1
heap.inOrder = true

Binary Heap Example: fixHeap aka heapify





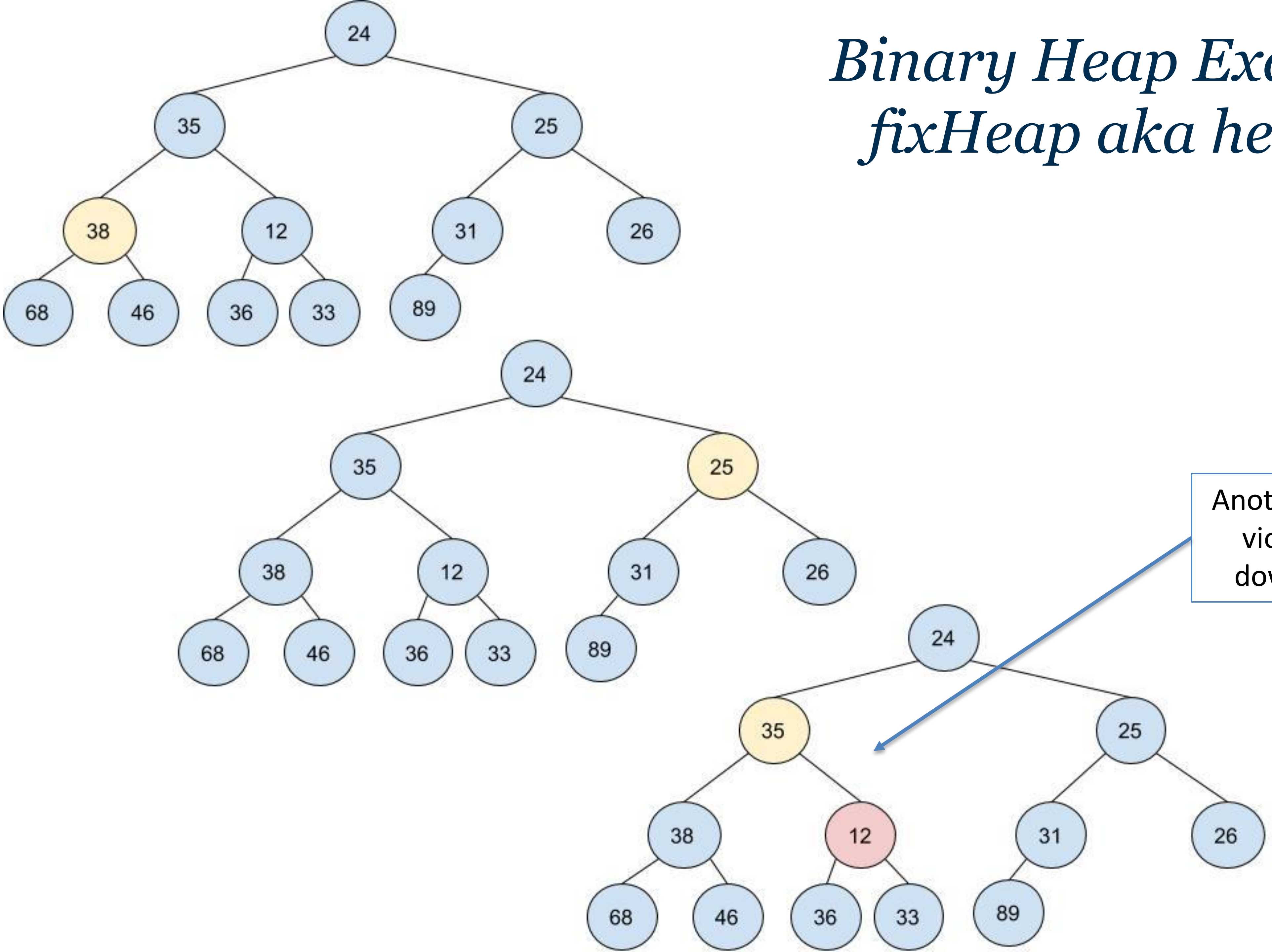



```
// for use with lazy insert
// repeatedly check with swap down

function fixHeap(heap)
    i := heap.currentSize / 2
    while i > 0
        swapDown(heap, i)
        i := i - 1
    heap.inOrder = true
```

Try at home: create a worst case unordered heap. How many swaps are needed?

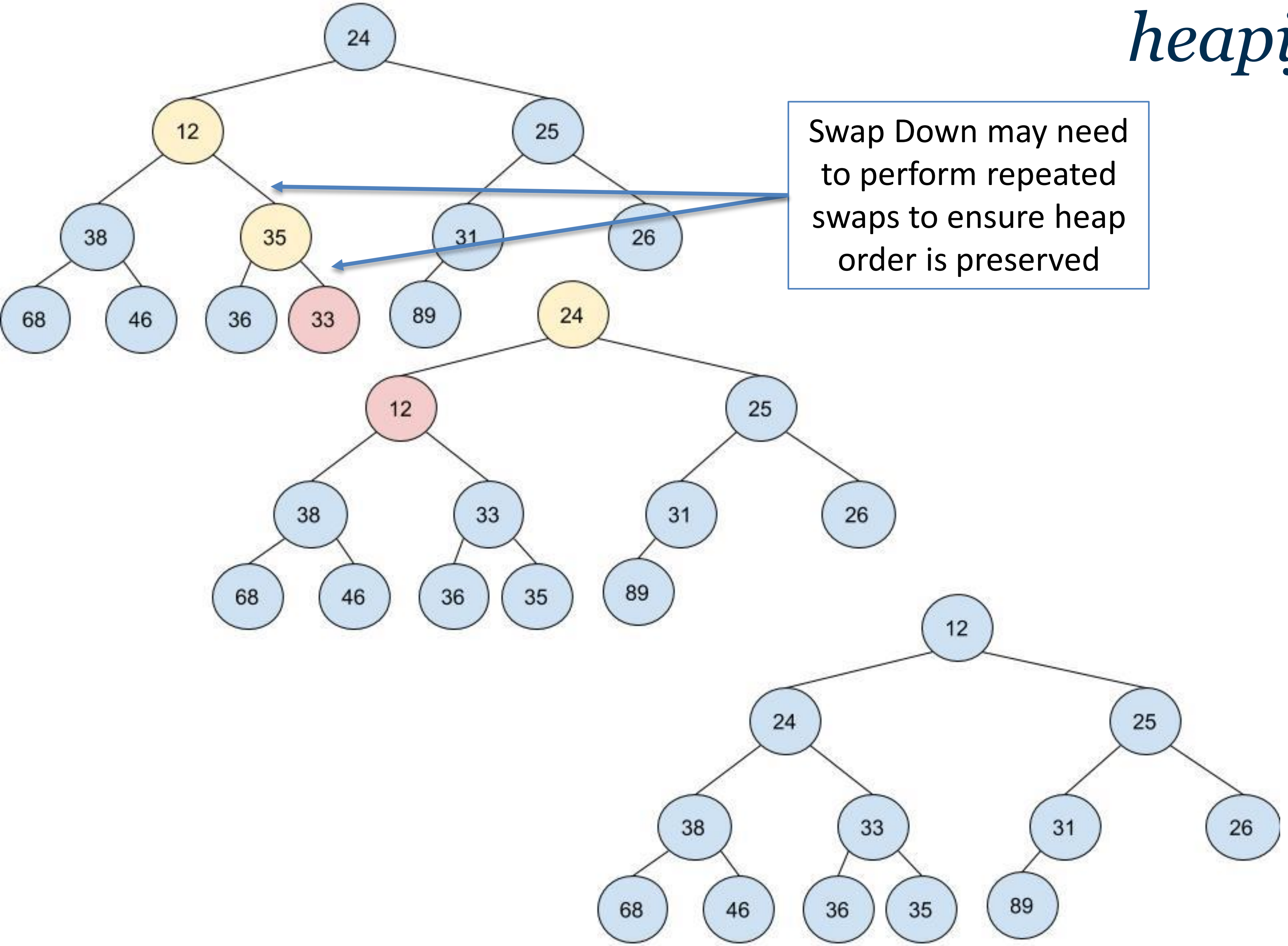
Binary Heap Example: fixHeap aka heapify



Another heap order violation. Swap down is applied.

GEORGETOWN UNIVERSITY

Binary Heap Example: fixHeap aka heapify



- Observations: The initial swap of 12 and 35 is progress, however, there is a violation still with 35 and 33. The swap down method will repeatedly swap down, thus assuring heap property order.
- Result: Swap down in the worst case will result in O(log n); swapping down the height of the tree.

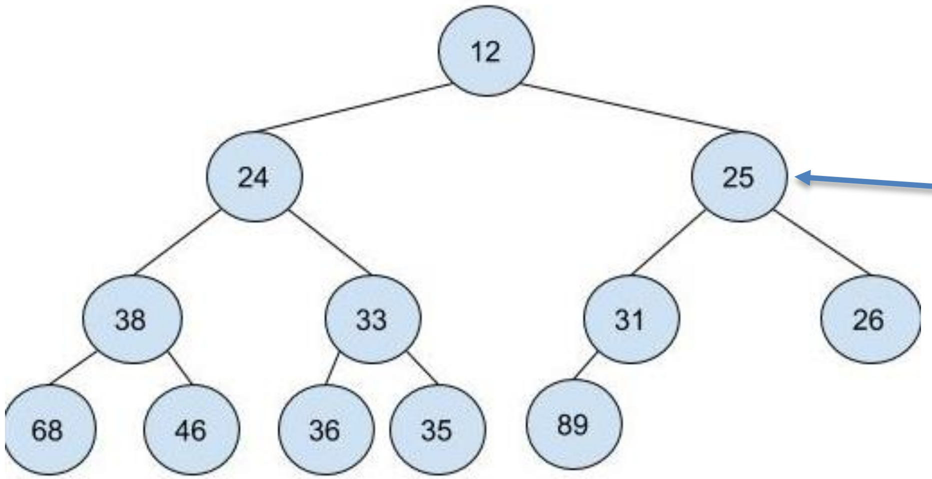
Analysis of tossIn and fixHeap (heapify)

- tossIn
  - all cases time (assuming size is sufficient): O(1)

- fixHeap
  - Fix heap performs a reverse BFS O(n) and potentially swaps down at each step of the traversal O(log n), for a total upperbound of O(n log n)
    - Swap down has a worst case of O(log n), BUT is this worst case possible in every iteration of our reverse BFS in fixHeap **– NO.**
    - In fact we can limit the total number of swaps performed by swap down when used in conjunction with this reverse BFS.
    - **The Result: O(n)**

Bounding fixHeap

- How can we more tightly bound fixHeap by O(n) ?
- Proof idea:
  - Observe: Each node can be swapped down, at maximum the distance to its furthest leaf node (the height of that specific node)
  - Thus we need only compute the sum of the heights of all nodes to compute an upper bound (teaser alert! Its O(n)).



The maximum number of swaps 25 could potentially have is 2 (the distance to 89), this is its height

GEORGETOWN UNIVERSITY

Proof idea: fixHeap is O(n)

- Show the sum of the heights of all nodes in a BST is linear.
  - 1 node (the root node), will have the maximum height $\lfloor \log_2 n \rfloor$, its children (2 nodes) will have heights $\lfloor \log_2 n \rfloor$ - 1, continuing on in this fashion we have the following sum of heights:

- Sum of heights =

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i * (\lfloor \log_2 n \rfloor - i) \quad = \sum_{i=0}^{h} 2^i (h-i) = 2^h(0) + 2^{h-1}(1) + \ldots + 2^0(h) =$$

$$2^h[(0) + 2^{-1}(1) + \ldots + 2^{-h}(h)] =$$

$$= n \boxed{\sum_{i=0}^{h} \frac{i}{2^i}} \leq 2n$$

Note that when h approaches inf, this sum converges to 2, thus we can upperbound it.

Side note: this proof shows that we can upper bound the number of total swaps linearly! In fact we can more strictly bound this expression:
Sum of heights = n – h.
Try a proof by induction.
See our example on previous slide:
Sum of heights = 9
n – h = 12 - 3

Repeated inserts vs. toss and heapify

- Which is more efficient?

- Assume we start with an empty heap and perform n inserts.
  - tossIn and fixHeap
    - O(n) + O(n) = O(n)
  - Repeated Inserts
    - O(n log n)

Priority Queues (with changing priorities)

- Array implementation permits efficient implementation
  - Previous implementation priorities were fixed

- Permitting priority updates
  - Concerns
    1. Identifying item to be updated
    2. Updating item and updating heap order

updatePriority

- Updating a nodes priority
  - Assume location of priority to be updated is known.
  - **If location is not assumed, must search heap before update

```
// updates priority value at node index with newVal
function updateP(heap, index, newVal)
    if heap.array[index] < newVal // lowering priority
        heap.array[index] := newVal
        swapDown(heap, index)
    else // increasing priority
        heap.array[index] := newVal
        swapUp(heap, index)
```

Side Note: Applications

- Priority Queues

- Sorting
  - Data structures are used to store data
  - Data structures are also used to help facilitate computational tasks efficiently
  - Heaps are useful for sorting
    - Takes advantage of efficiency of fixHeap (aka heapify)
    - First weakly orders elements, then removes min

Heap Sort

- Input: n orderable items
- Heap sort algorithm
  1. Build heap
     1. tossIn all n items , O(n)
     2. fixHeap , O(n)
  2. Iteratively disassemble heap and build ordered list
     - Repeatedly Remove O(n log n)
       1. remove root (min thus far) and add to tail of growing list
       2. swapDown(heap,1)

Analysis of some sorting algorithms

- Insertion Sort
  - Time
    - Worst Case: $O(n^2)$
    - Best Case: $O(n)$
  - Space:
    - In-place: YES
- Bubble Sort
  - Time
    - Worst Case: $O(n^2)$
    - Best Case: $O(n)$
  - Space:
    - In-place: YES
- Heap Sort
  - Time
    - Worst Case: ?
    - Best Case: ?
  - Space:
    - In-place: YES
- Selection
  - Time
    - Worst Case: $O(n^2)$
    - Best Case: $O(n)$
  - Space:
    - In-place: YES

Merging Heaps

- In some applications, merging or combining priority queues is fundamental
  - Eg a set of queues is formed for set of resources and a resource is eliminated.

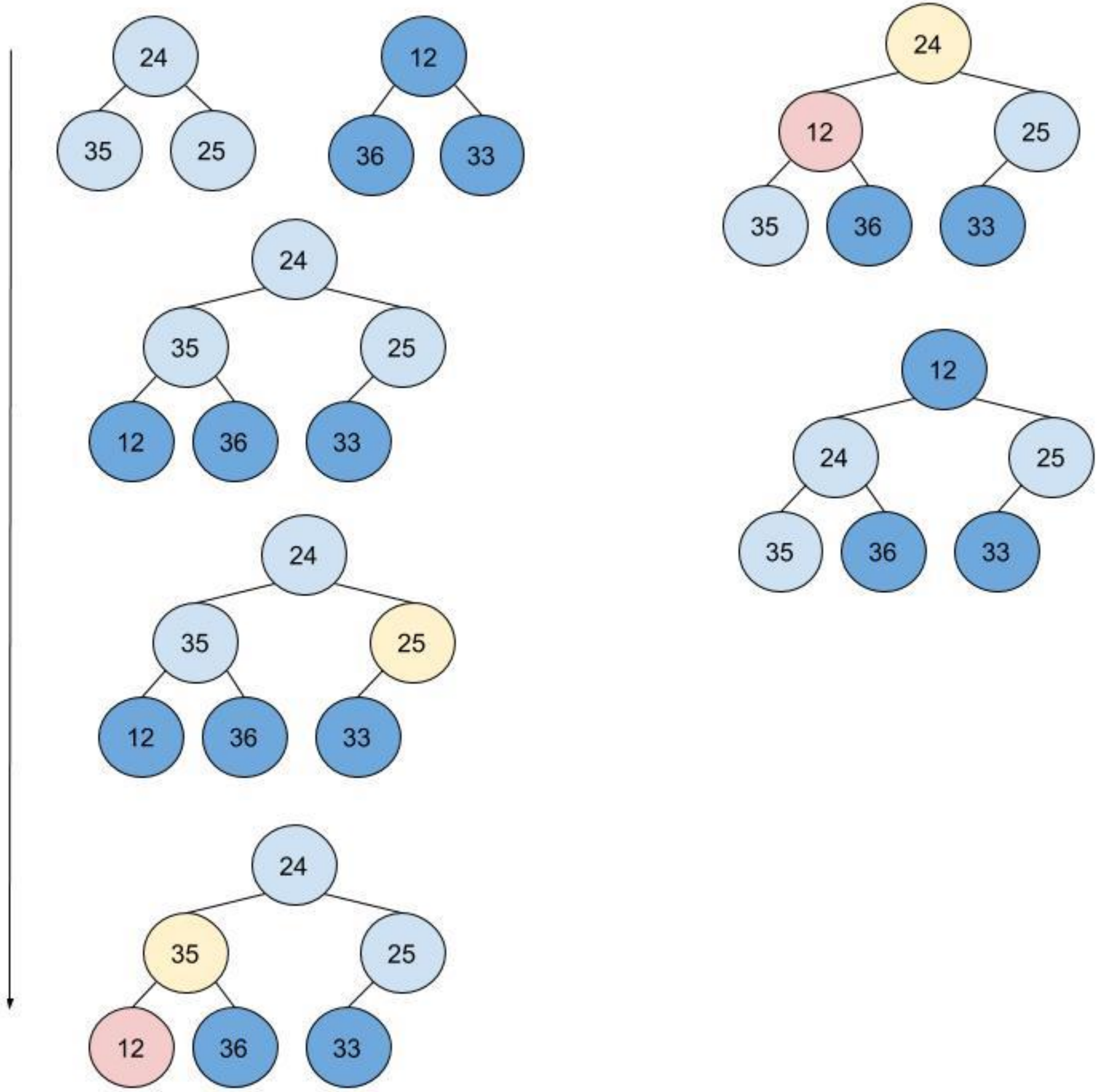
- Implementation Implications
  - Array
    - Idea: copy items from array into appropriate locations into other array (likely with some extra rearrangements)
    - At best, linear time given contiguous allocation (must at least copy over one of the hashes)
    - One simple scheme
      1. newHeap := concatenate(heap1 ,heap2) // repeated lazy insertions
      2. fixHeap(newHeap)

GEORGETOWN UNIVERSITY

Merging Examples

One simple scheme (assuming array implementation)

1. newHeap := concatenate(heap1 ,heap2)
   // repeated lazy insertions
2. fixHeap(newHeap)



Merging Heaps

- Array vs. Chaining
  - Simple merging for array implementation required O(sizeHeap1 + sizeHeap2)

- Chaining
  - If the priority queue requires a merging operation, then a chaining implementation may be more efficient.
  - Merging can be done with O(log n) with the following design changes
    - Chaining with nodes (not array)
    - Relax height constraint (really ?!?)
  - The skew heap!

Skew heaps

- A skew heap is a binary tree with heap order (and no balance constraint) and a skew merging scheme
  - Result: may have linear depth in worst case (though as we have shown, with BST, the average depth is logarithmic)
  - All operations will have logarithmic time for the average case.

- Implementation (structurally)
  - Chaining with pointers: leftChild and rightChild
  - Node similar to node used in BST

Merging Skew Heaps

- Scheme to merge $heap_1$ and $heap_2$, with root nodes r_1 and r_2
  - Repeatedly merge
    1. Base Case: if one tree is empty, return the other
    2. Recursive Case:
       1. temp := r_1.rightChild
       2. r_1.rightChild := r_1.leftChild
       3. r_1.leftChild := merge(temp, r_2)
       4. return r_1

- Remove right child of left tree
- Make left child of left tree, the right child
- Make new left child of left tree, the result of merging the right tree with the old right child.

Merge Skew Heap

// assume r_1 and r_2 are the roots of heap$_1$ and heap$_2$ initially
// recursively merges the heaps using skew scheme

$function\ mergeSkewHeap(r_1, r_2)$

$\quad if\, r_1\ is\ NULL\ , return\ r_2$

$\quad if\, r_2\ is\ NULL\ , return\ r_1$

$\quad if\ r_1.priority < r_2.priority$

$\qquad temp := r_1.rChild$

$\qquad r_1.rChild := r_1.lChild$

$\qquad r_1.lChild := mergeSkewHeap(r_2, temp)$

$\qquad return\ r_1$

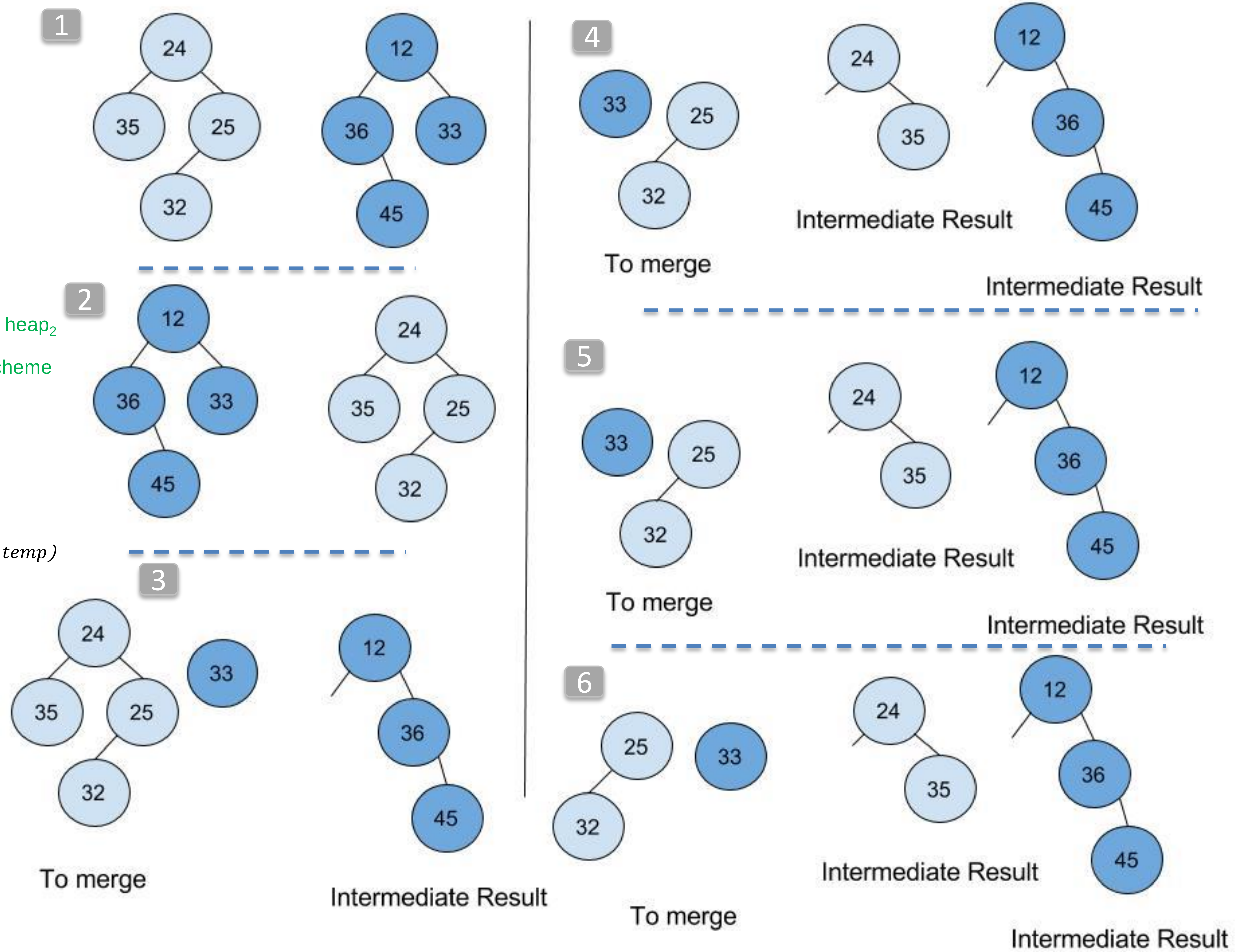
$\quad else$

$\qquad return\ merge(r_2, r_1)$

Merge Skew Heaps Example

// assume r_1 and r_2 are the roots of heap$_1$ and heap$_2$ initially
// recursively merges the heaps using skew scheme

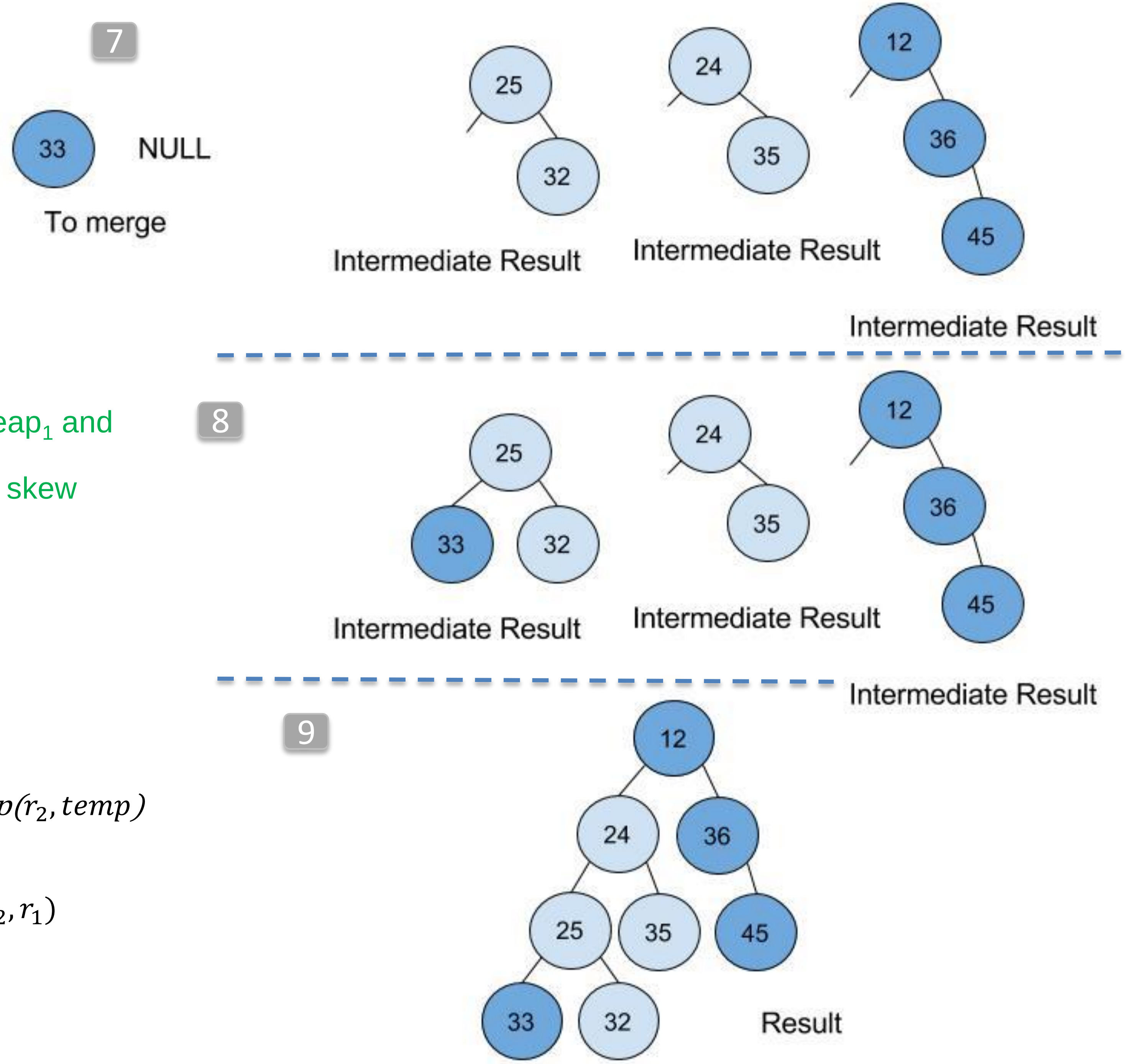
$function\ mergeSkewHeap(r_1, r_2)$
$if\ r_1\ is\ NULL, return\ r_2$
$if\ r_2\ is\ NULL, return\ r_1$
$if\ r_1.priority < r_2.priority$
$temp := r_1.rChild$
$r_1.rChild := r_1.lChild$
$r_1.lChild := mergeSkewHeap(r_2, temp)$
$return\ r_1$
$else$
$return\ mergeSkewHeap(r_2, r_1)$



Merge Skew Heaps Example

// assume r_1 and r_2 are the roots of heap$_1$ and heap$_2$ initially
// recursively merges the heaps using skew scheme

function $mergeSkewHeap(r_1, r_2)$
    $if\, r_1$ *is NULL , return* r_2
    $if\, r_2$ *is NULL , return* r_1
    if $r_1.priority < r_2.priority$
        $temp := r_1.rChild$
        $r_1.rChild := r_1.lChild$
        $r_1.lChild := mergeSkewHeap(r_2, temp)$
        return r_1
    else
        return $mergeSkewHeap(r_2, r_1)$



Notes

- Other operations on skew heaps
  - Since Skew Heaps are not array implementations, nor are they balanced, the insertion operation and update priority operations are somewhat different as compared to the Binary Heap
  - Interestingly, these operations can be managed with using merges
    - Insert(val): merge(heapRoot, newNode(val))
    - Remove:
      1. newRoot : =merge(root.lChild, root.rChild)
      2. return oldRoot
    - updatePriority(node, newVal):
      1. node.priority := newVal
      2. detach node from parent // this requires pointer to parent
      3. newRoot := merge(root, node)

GEORGETOWN UNIVERSITY

Summary

- Binary Heaps
  - Relaxed order compared to BSTs
  - Strict balance and leftmost structure
  - Provides for log time worst case for all operations except merge
  - Array implementation provides for efficient space and time
  - Not bad for sorting

- Skew Heaps
  - Chaining implementation
  - Binary Tree with Heap order (but no balance constraint)
  - Provides for average logarithmic time for all operations, including merge