Outline

In this topic, we will:

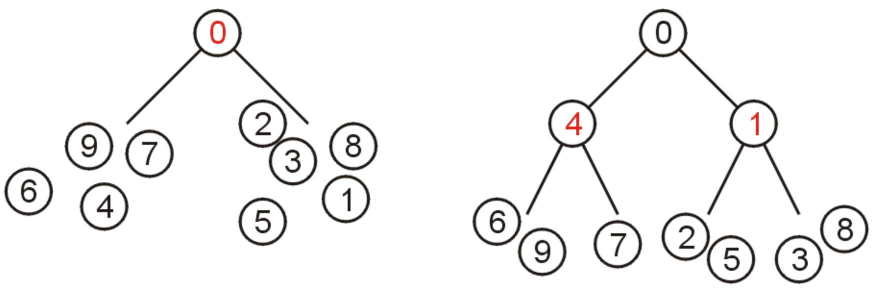
- Define a binary min-heap
- Look at some examples
- Operations on heaps:
  - Top
  - Pop
  - Push
- An array representation of heaps
- Define a binary max-heap
- Using binary heaps as priority queues

7.2

Definition

A non-empty binary tree is a min-heap if

- The key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



From this definition:

- A single node is a min-heap
- All keys in either sub-tree are greater than the root key

7.2

Definition

Important:

THERE IS NO OTHER RELATIONSHIP BETWEEN

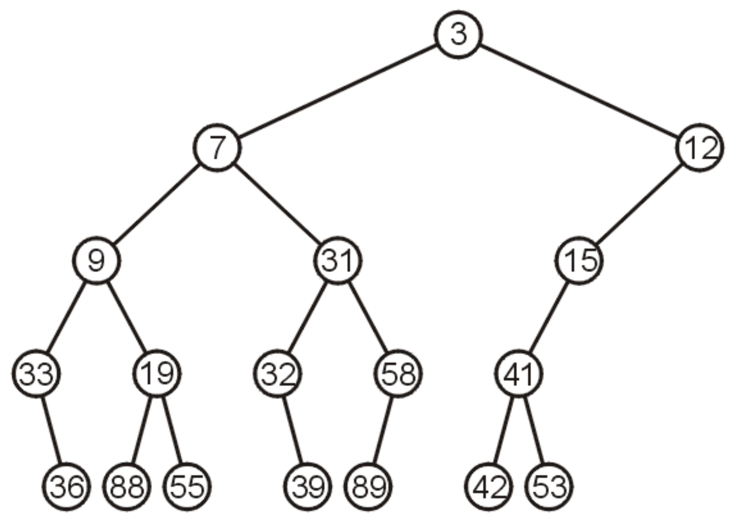
THE ELEMENTS IN THE TWO SUBTREES

Failing to understand this is the greatest mistake a student makes

7.2

Example

This is a binary min-heap:



7.2.2

Implementations

With binary search trees, we introduced the concept of *balance*

From this, we looked at:
- AVL Trees
- B-Trees
- Red-black Trees (not course material)

How can we determine where to insert so as to keep balance?

7.2.2

Implementations

There are multiple means of keeping balance with binary heaps:

- Complete binary trees
- Leftist heaps
- Skew heaps

We will look at using complete binary trees

- In has optimal memory characteristics but sub-optimal run-time characteristics

7.2.2

Complete Trees

By using complete binary trees, we will be able to maintain, with minimal effort, the complete tree structure

We have already seen

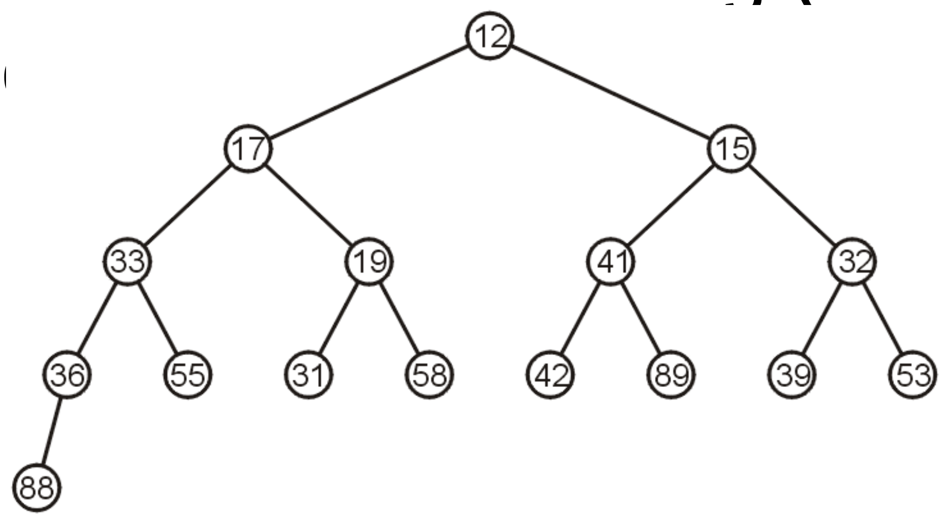
– It is easy to store a complete tree as an array

If we can store a heap of size n as an array of size $\Theta(n)$, this would be great!

7.2.2

Complete Trees

For example, the previous heap may be represented as the following (non-unique!) complet

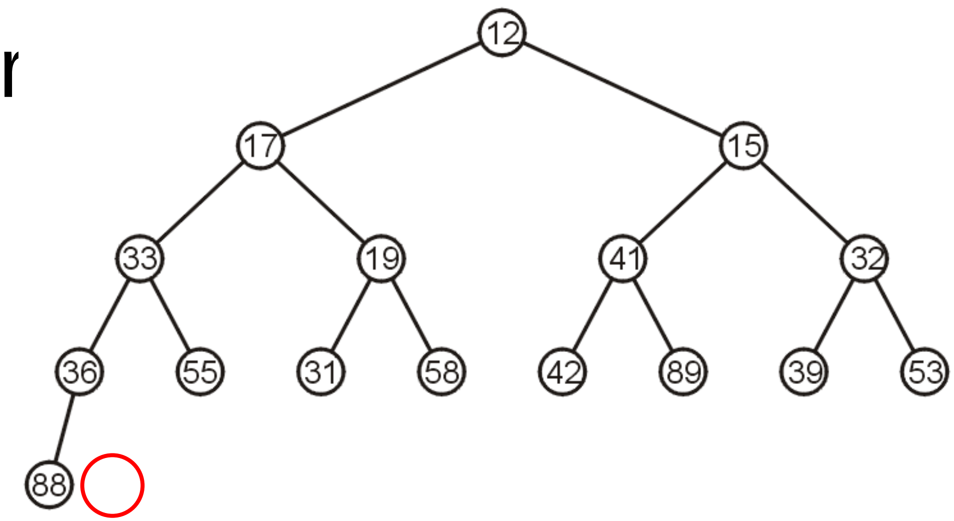


7.2.2

Complete Trees: Push

If we insert into a complete tree, we need only place the new node as a leaf node in the appr

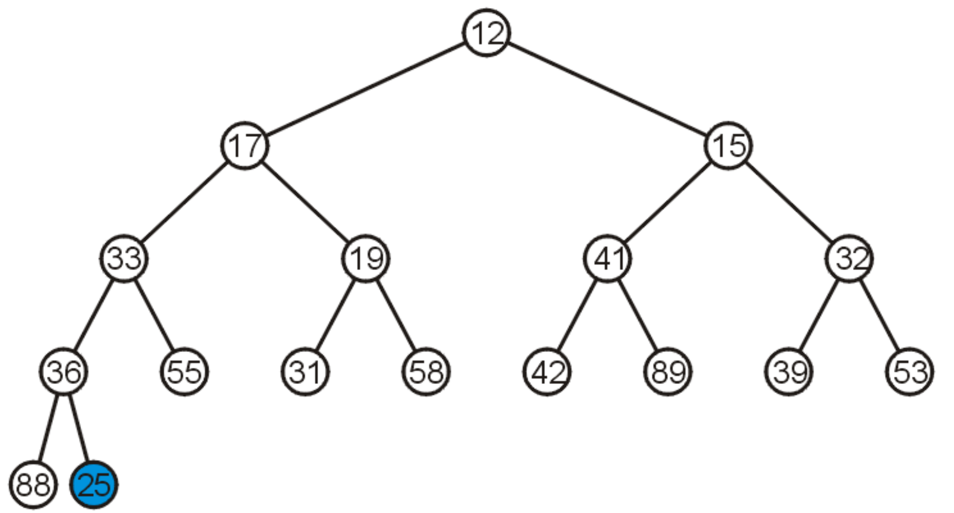
olate up



7.2.2

Complete Trees: Push

For example, push 25:

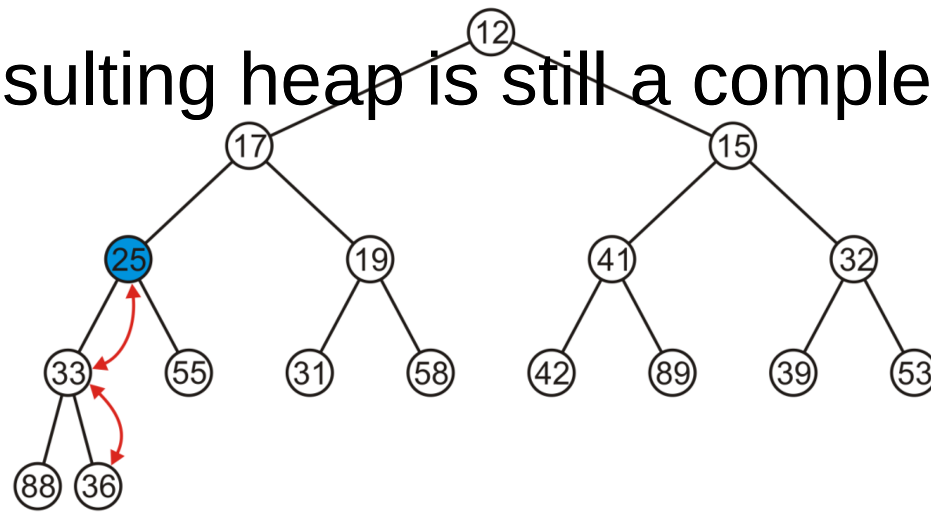


7.2.2

Complete Trees: Push

We have to percolate 25 up into its appropriate location

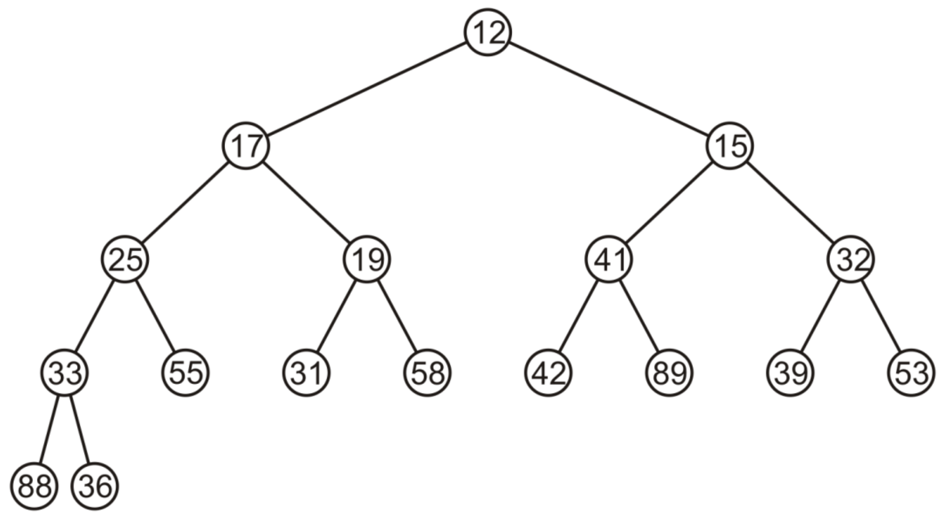
– The resulting heap is still a complete tree



7.2.2

Complete Trees: Pop

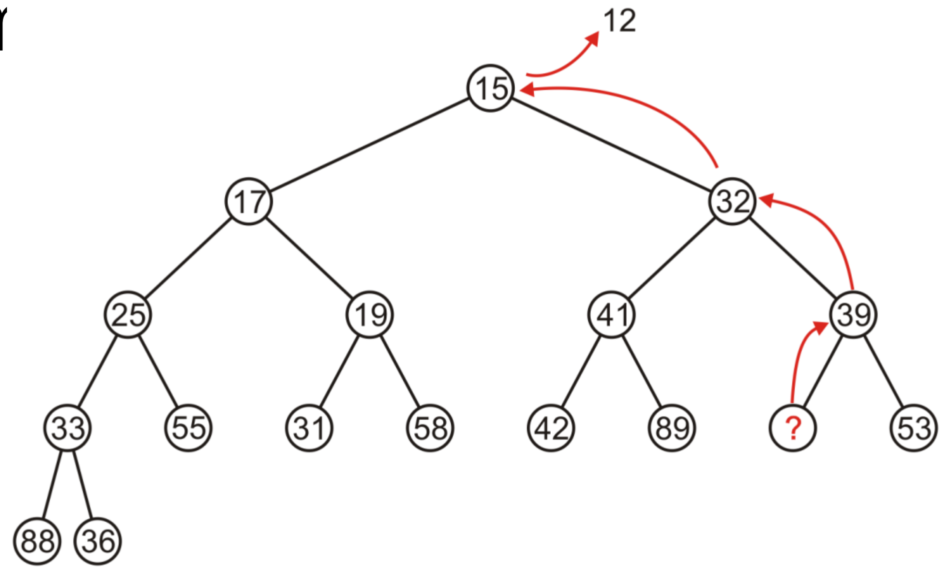
Suppose we want to pop the top entry: 12



7.2.2

Complete Trees: Pop

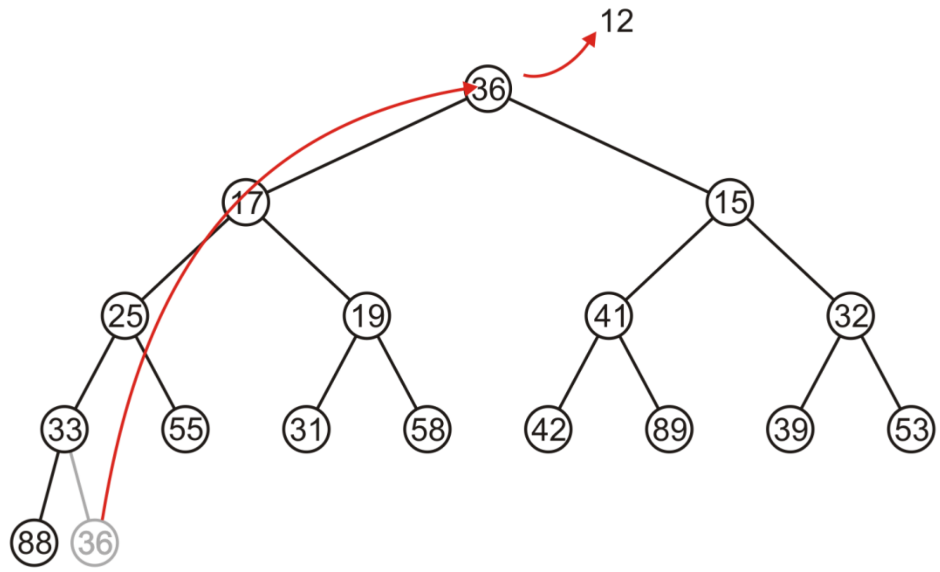
Percolating up creates a hole leading to a
non-com



7.2.2

Complete Trees: Pop

Alternatively, copy the last entry in the heap to

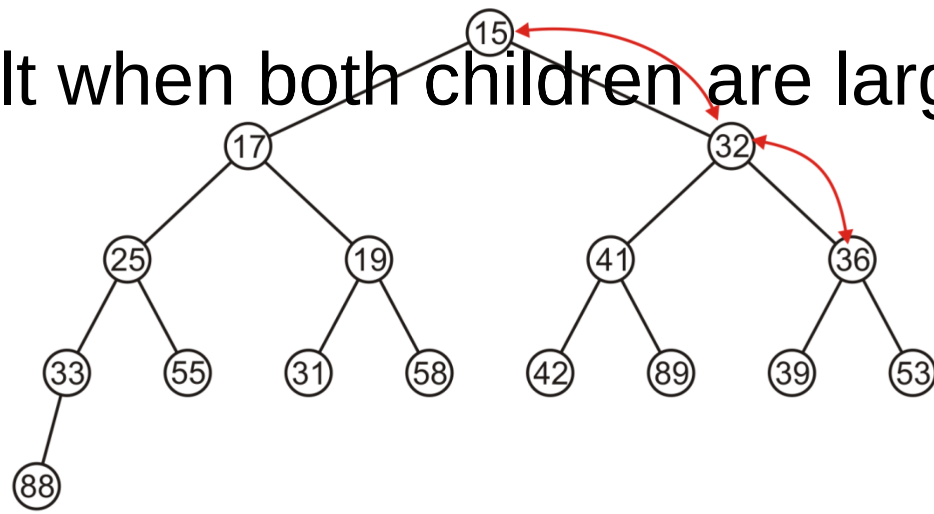


7.2.2

Complete Trees: Pop

Now, percolate 36 down swapping it with the smallest of its children

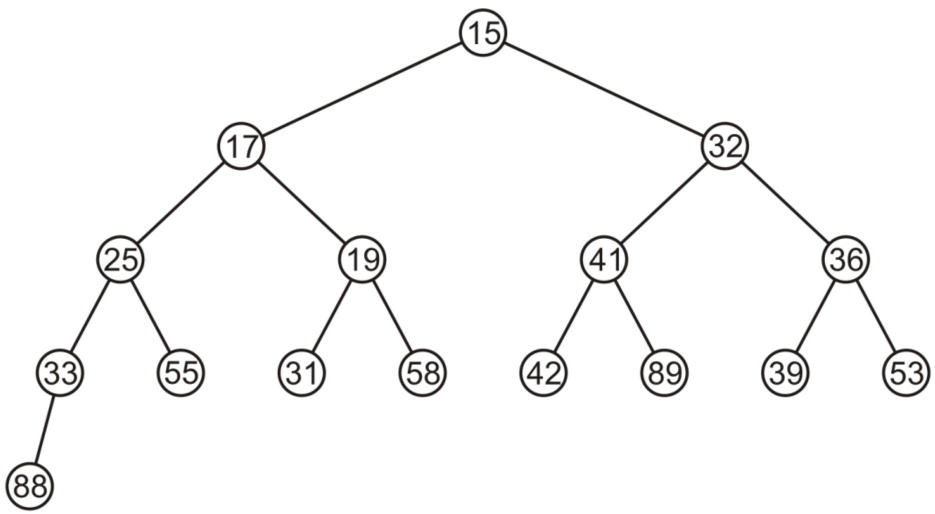
– We halt when both children are larger



7.2.2

Complete Trees: Pop

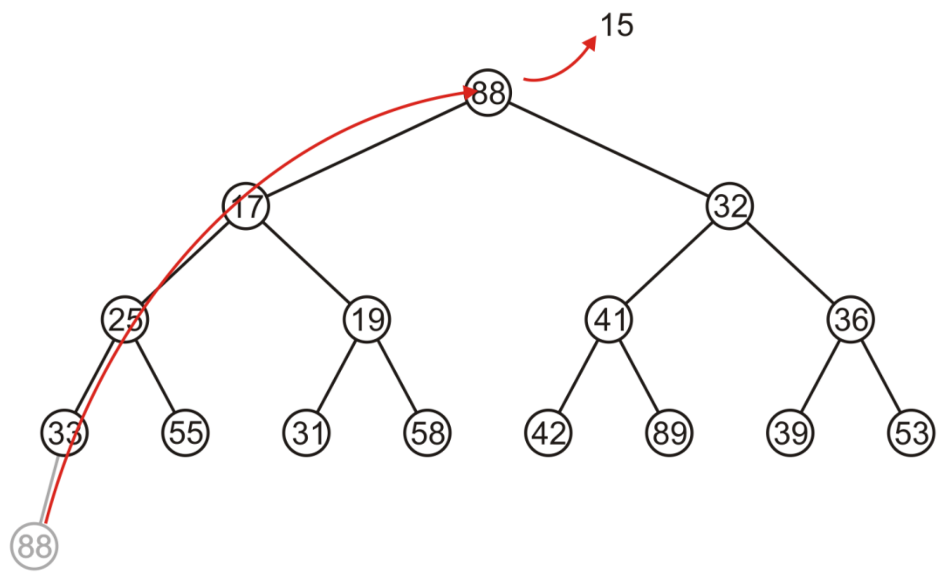
The resulting tree is now still a complete tree:



7.2.2

Complete Trees: Pop

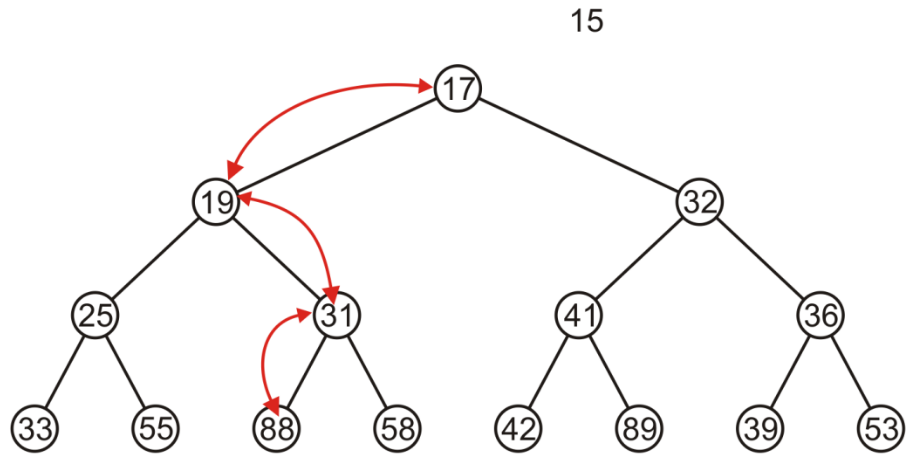
Again, popping 15, copy up the last entry: 88



7.2.2

Complete Trees: Pop

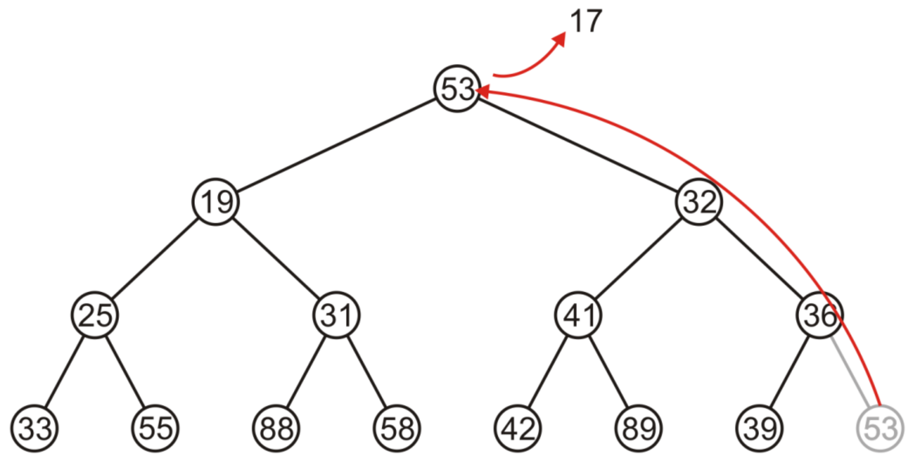
This time, it gets percolated down to the point wh



7.2.2

Complete Trees: Pop

In popping 17, 53 is moved to the top

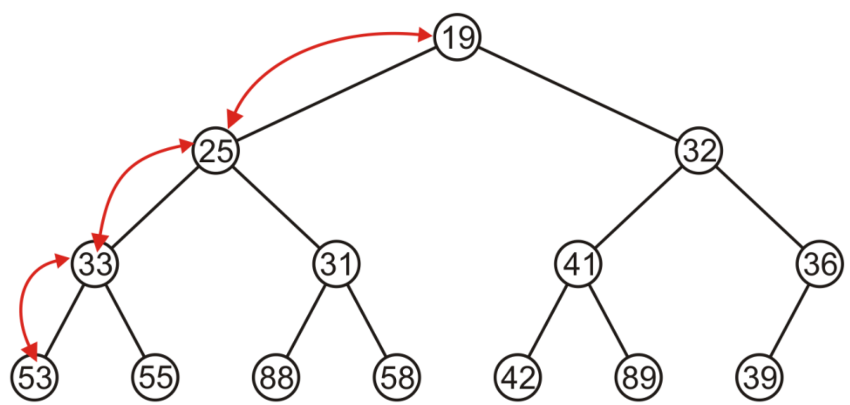


7.2.2

Complete Trees: Pop

And percolated down, again to the deepest

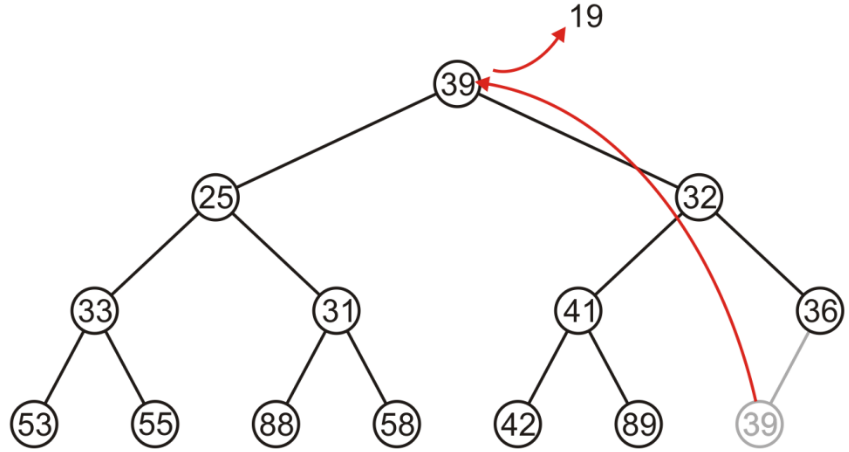
17



7.2.2

Complete Trees: Pop

Popping 19 copies up 39

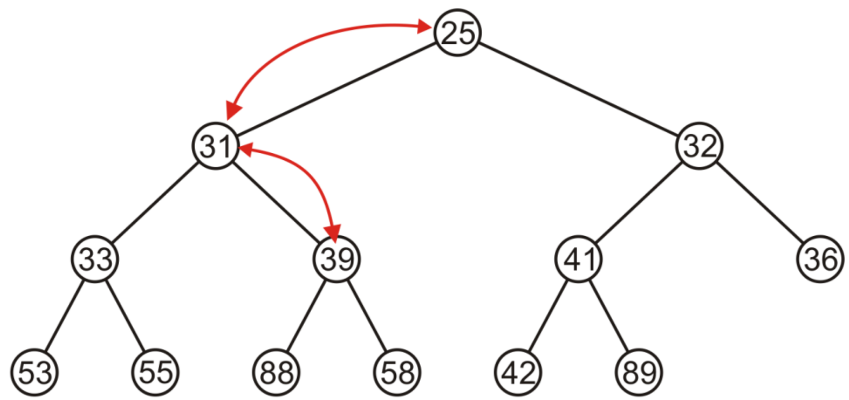


7.2.2

Complete Trees: Pop

Which is then percolated down to the second

19



7.2.3

Complete Tree

Therefore, we can maintain the complete-tree shape of a heap

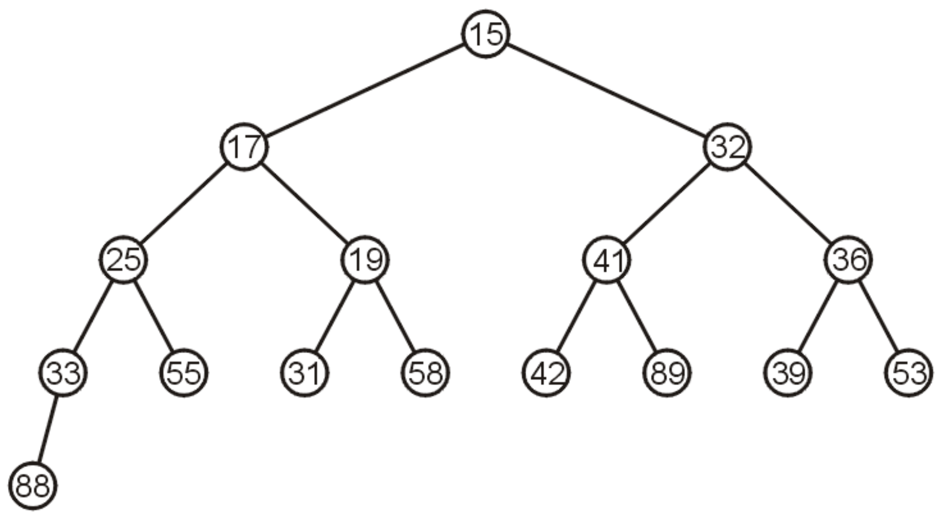
We may store a complete tree using an array:

- A complete tree is filled in breadth-first traversal order
- The array is filled using breadth-first traversal

7.2.3.1

Array Implementation

For the heap



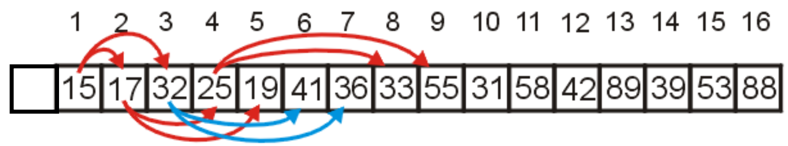
	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

a breadth-first traversal yields:

7.2.3.1

Array Implementation

Recall that If we associate an index–starting at 1–with each entry in the breadth-first traversal, we get:



Given the entry at index k, it follows that:

- The parent of node is a $k/2$
- the children are at $2k$ and $2k + 1$

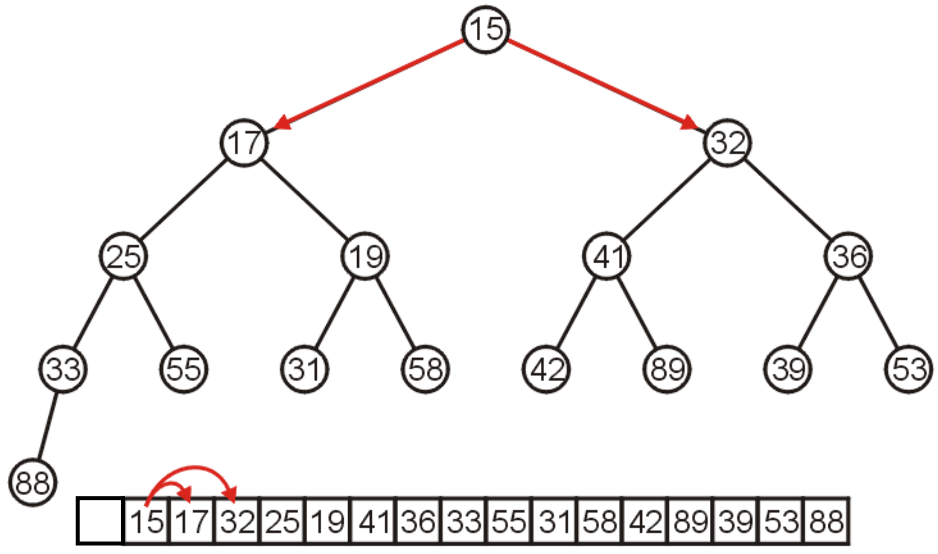
```
parent = k >> 1;

left_child = k << 1;
right_child = left_child | 1;
```

Cost (trivial): start array at position 1 instead of position 0

7.2.3.1

Array Implementation

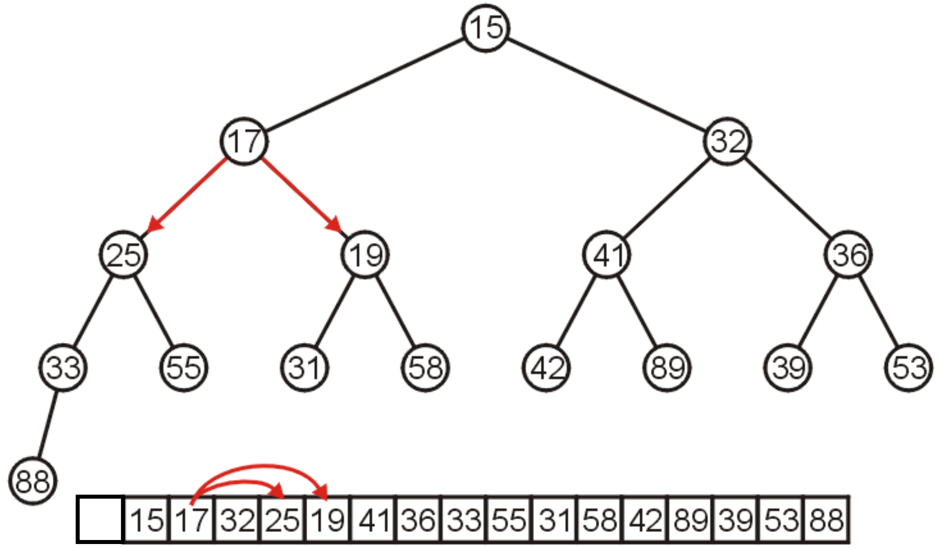
The children of 15 are 17 and 32:



7.2.3.1

Array Implementation

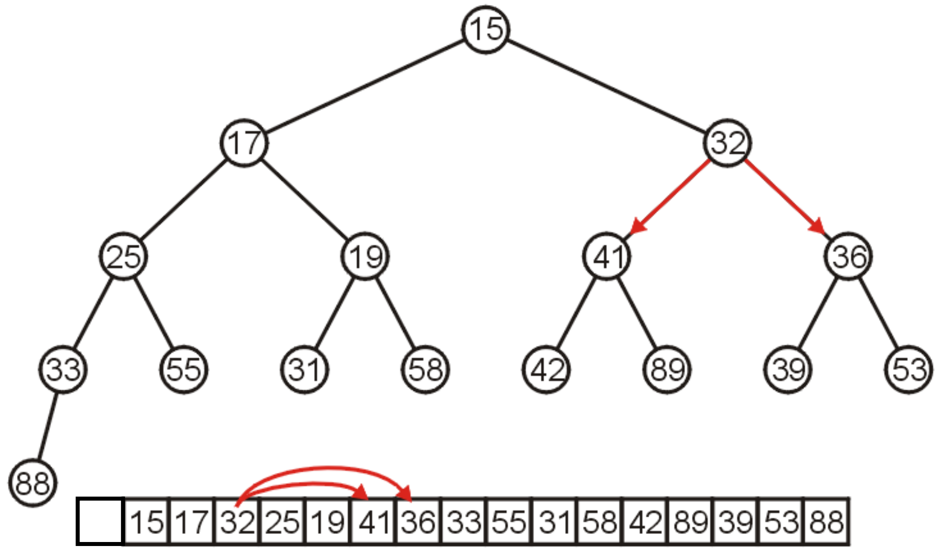
The children of 17 are 25 and 19:



7.2.3.1

Array Implementation

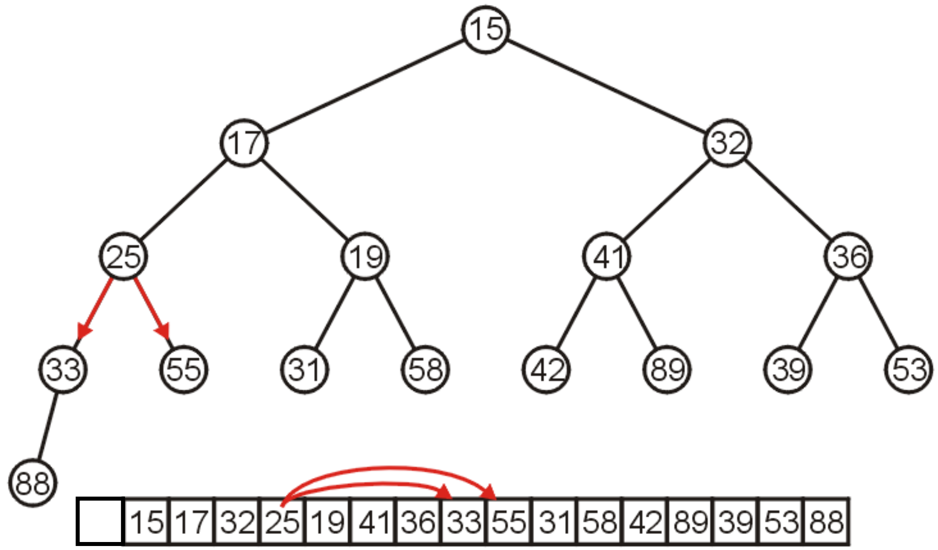
The children of 32 are 41 and 36:



7.2.3.1

Array Implementation

The children of 25 are 33 and 55:



7.2.3.1

Array Implementation

If the heap-as-array has **`count`** entries, then the next empty node in the corresponding complete tree is at location **`posn = count + 1`**

We compare the item at location **`posn`** with the item at **`posn/2`**

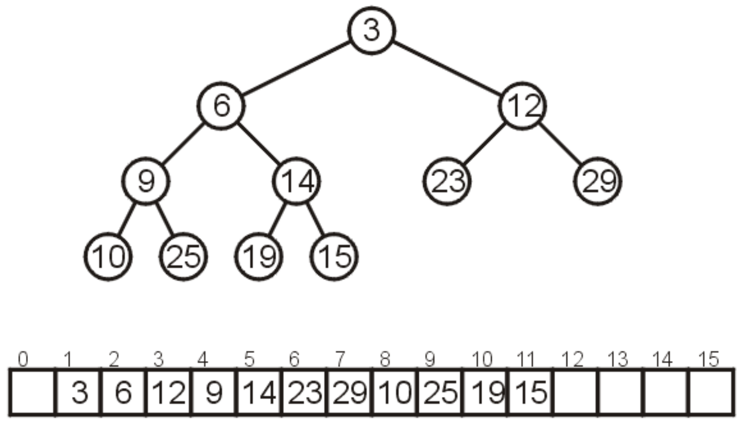
If they are out of order

– Swap them, set **`posn /= 2`** and repeat

7.2.3.2

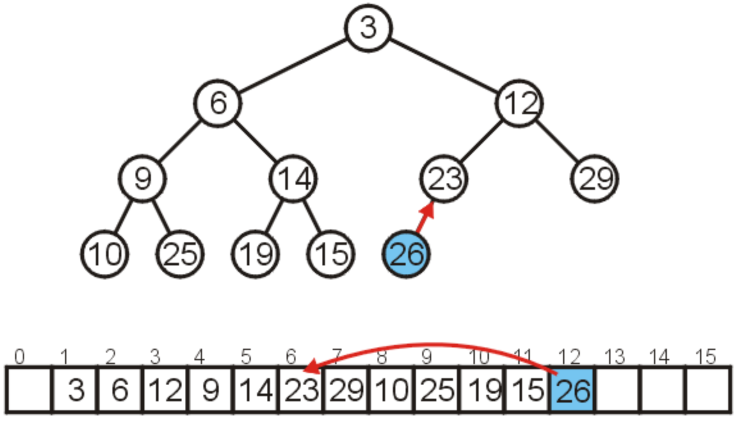
Array Implementation

Consider the following heap, both as a tree and in its array representation



7.2.3.2.1 Array Implementation: Push

Inserting 26 requires no changes



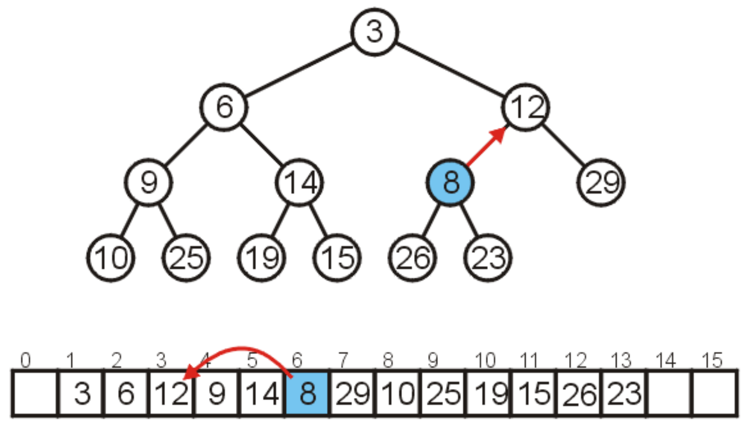
7.2.3.2.1 Array Implementation: Push

Inserting 8 requires a few percolations:

– Swap 8 and 23

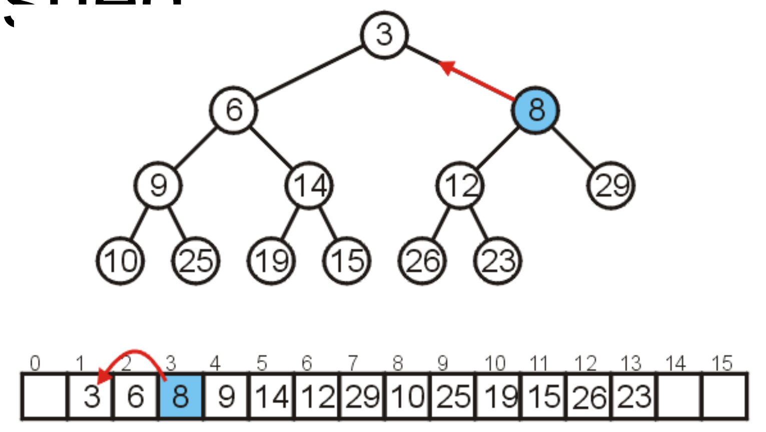
7.2.3.2.1 Array Implementation: Push

Swap 8 and 12



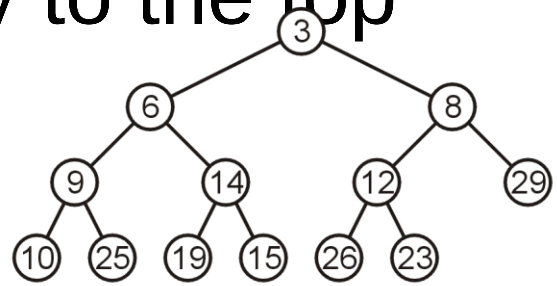
7.2.3.2.1 Array Implementation: Push

At this point, it is greater than its parent, so we are finished



7.2.3.2.2 Array Implementation: Pop

As before, popping the top has us copy the last entry to the top

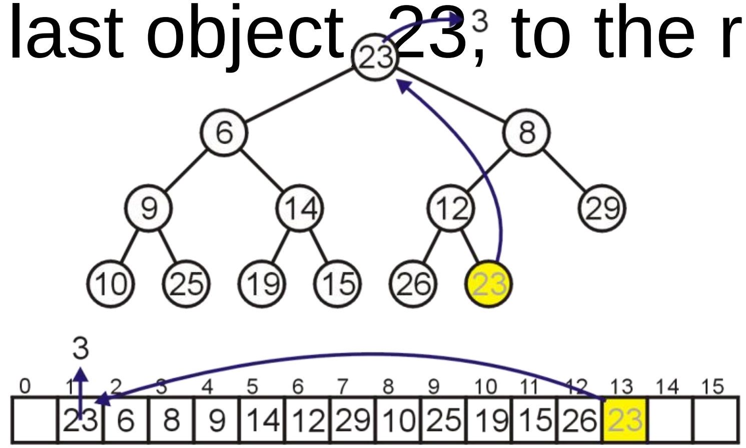


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

7.2.3.2.2 Array Implementation: Pop

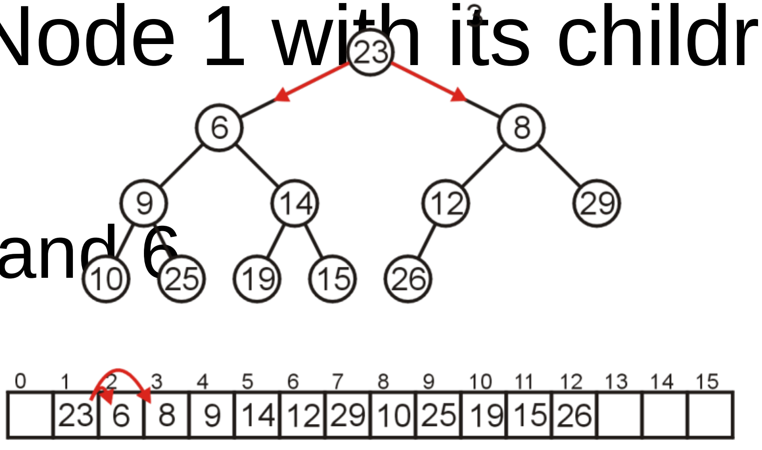
Instead, consider this strategy:

– Copy the last object, 23, to the root



7.2.3.2.2 Array Implementation: Pop

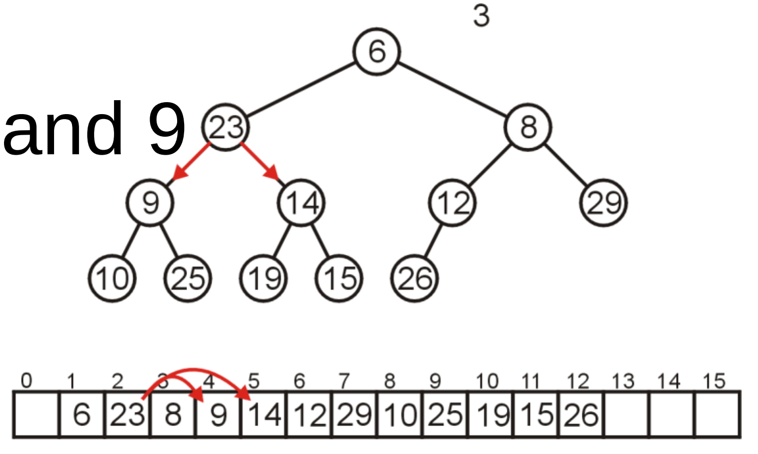
- Now percolate down
- Compare Node 1 with its children: Nodes 2 and 3
  - Swap 23 and 6



7.2.3.2.2 Array Implementation: Pop

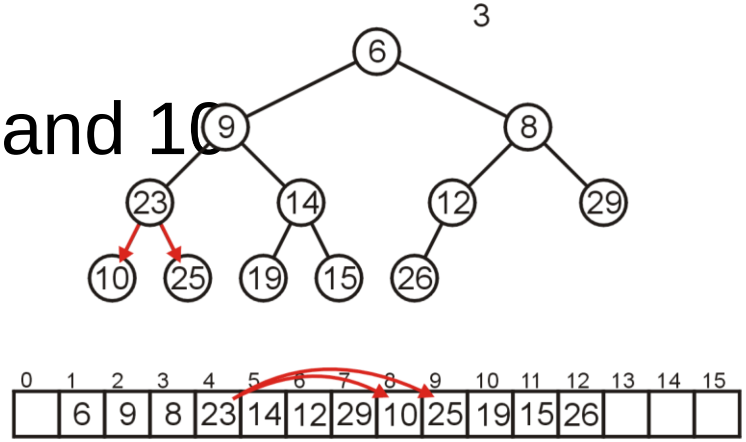
Compare Node 2 with its children: Nodes 4 and 5

– Swap 23 and 9



7.2.3.2.2 Array Implementation: Pop

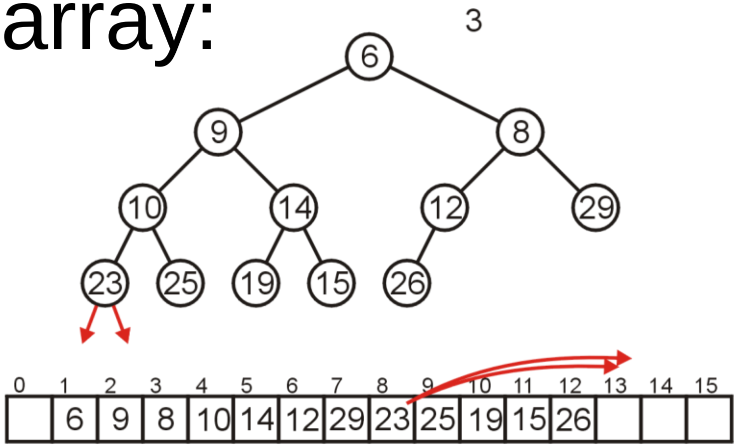
- Compare Node 4 with its children: Nodes 8 and 9
  - Swap 23 and 10



7.2.3.2.2 Array Implementation: Pop

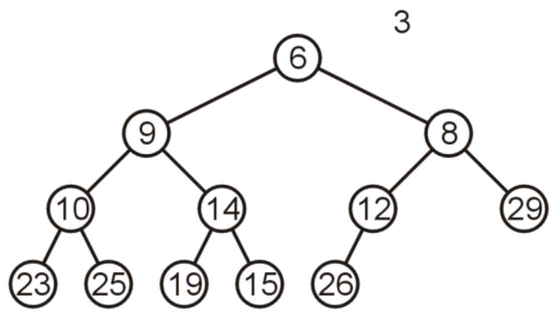
The children of Node 8 are beyond the end of the array:

– Stop



7.2.3.2.2 Array Implementation: Pop

The result is a binary min-heap

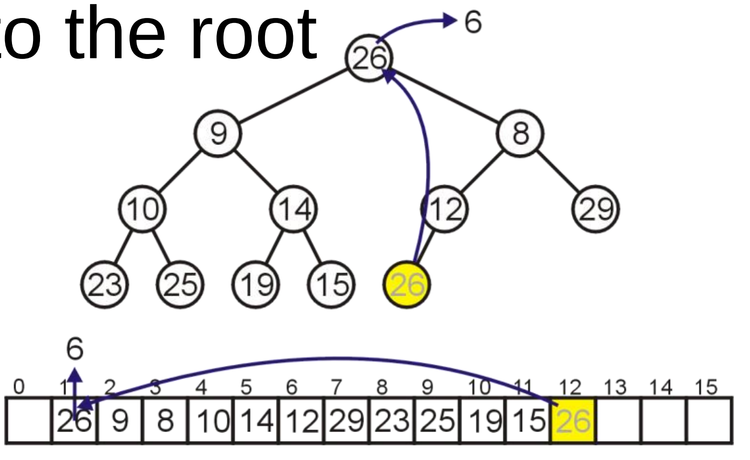


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

7.2.3.2.2 Array Implementation: Pop

Dequeuing the minimum again:

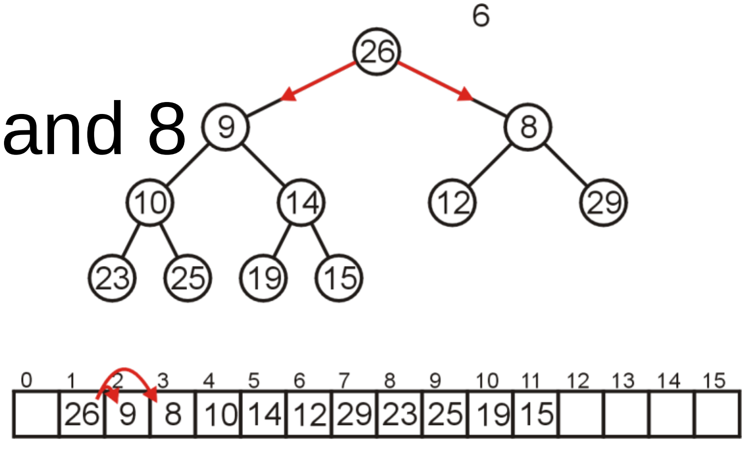
– Copy 26 to the root



7.2.3.2.2 Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

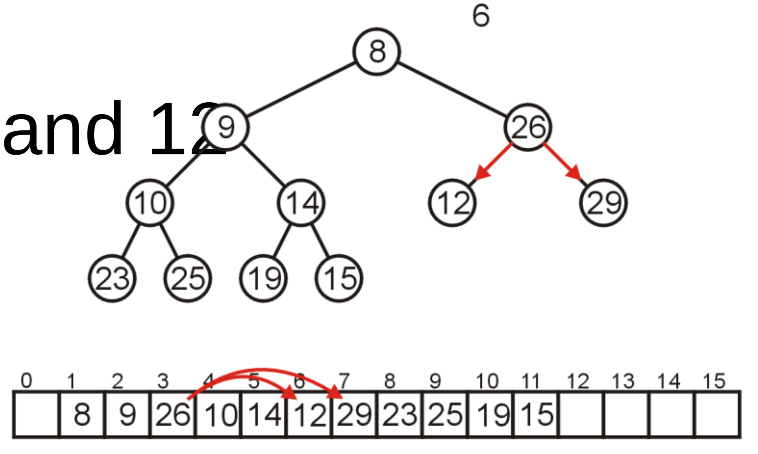
– Swap 26 and 8



7.2.3.2.2 Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

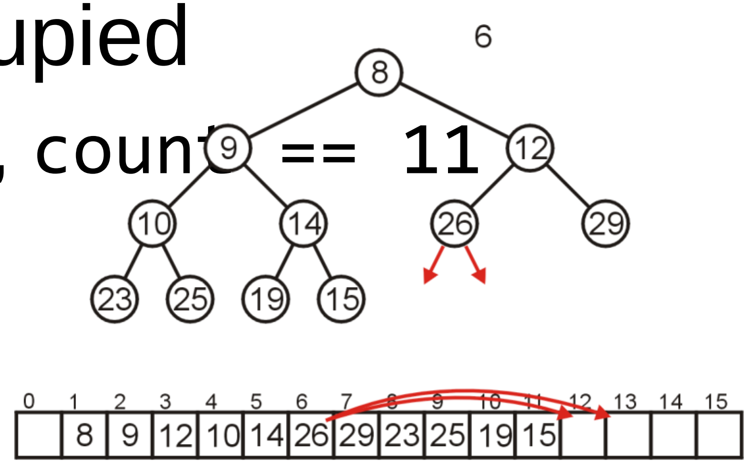
– Swap 26 and 12



7.2.3.2.2 Array Implementation: Pop

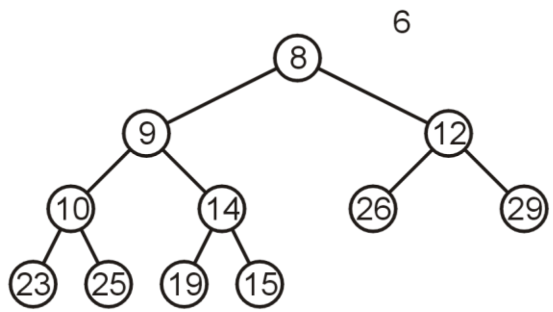
The children of Node 6, Nodes 12 and 13 are unoccupied

– Currently, count == 11



7.2.3.2.2 Array Implementation: Pop

The result is a min-heap

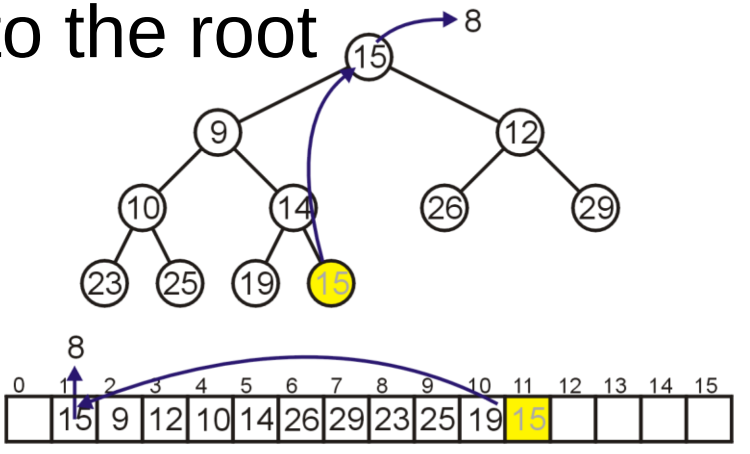


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	8	9	12	10	14	26	29	23	25	19	15				

7.2.3.2.2 Array Implementation: Pop

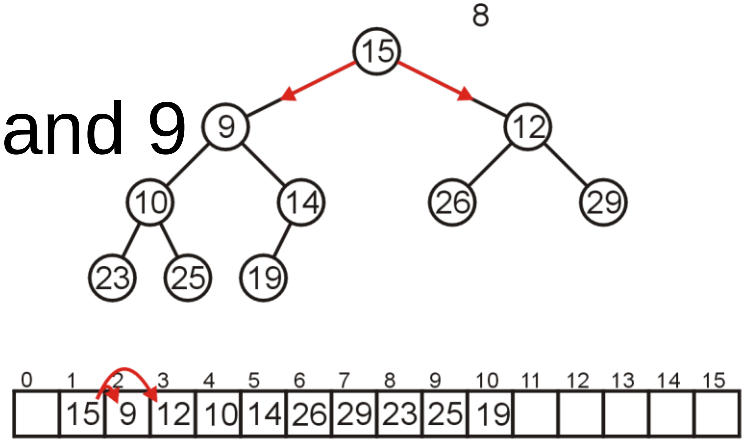
Dequeuing the minimum a third time:

– Copy 15 to the root



7.2.3.2.2 Array Implementation: Pop

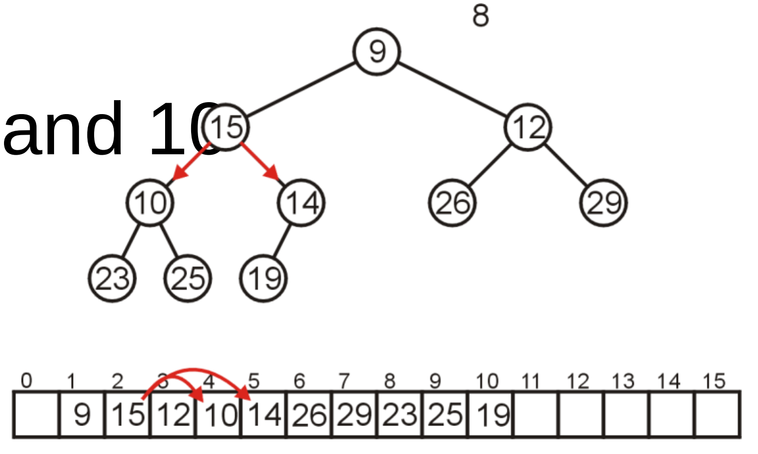
- Compare Node 1 with its children: Nodes 2 and 3
  - Swap 15 and 9



7.2.3.2.2 Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

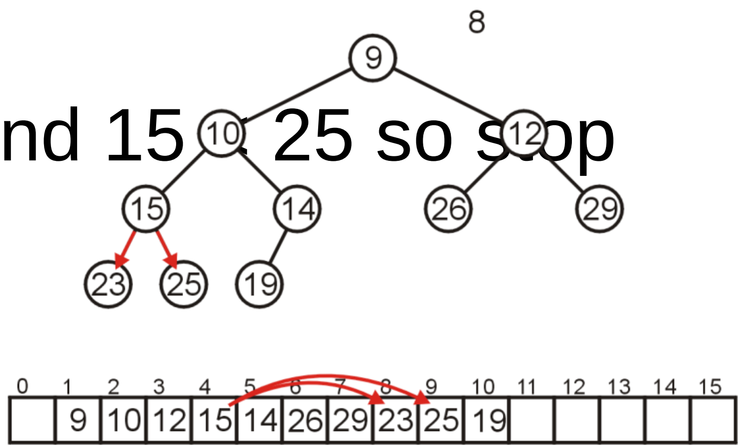
– Swap 15 and 10



7.2.3.2.2 Array Implementation: Pop

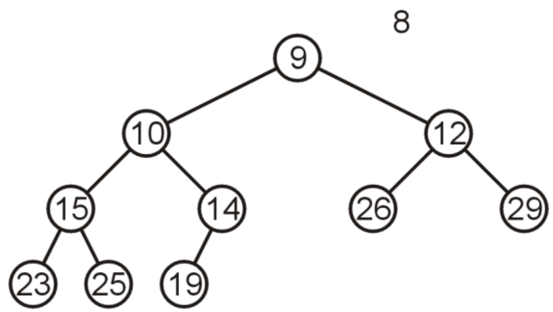
Compare Node 4 with its children: Nodes 8 and 9

– 15 < 23 and 15 < 25 so stop



7.2.3.2.2 Array Implementation: Pop

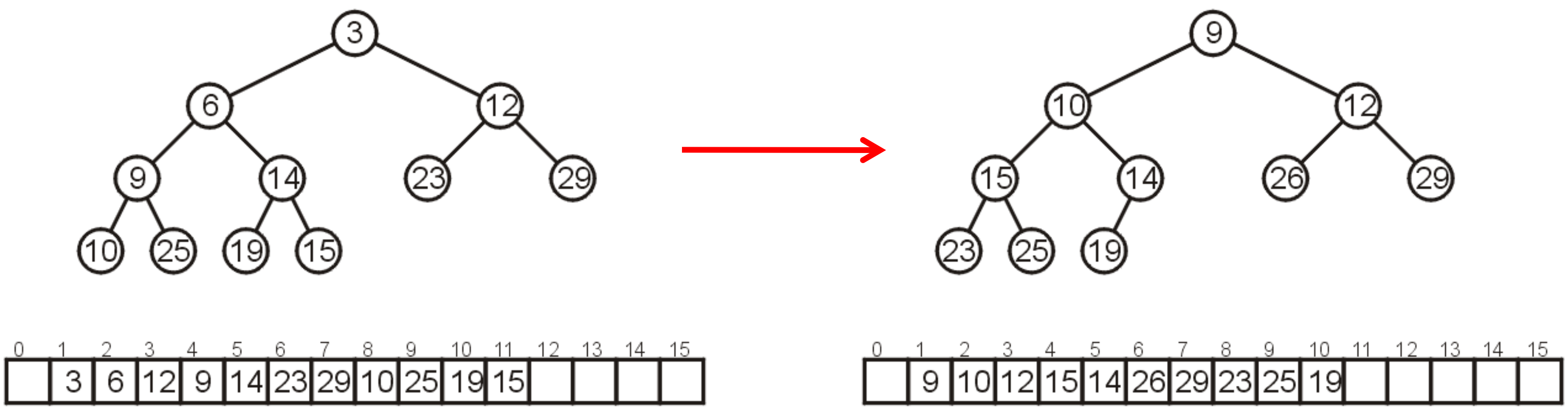
The result is a properly formed binary min-heap



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	9	10	12	15	14	26	29	23	25	19					

7.2.3.2.2 Array Implementation: Pop

After all our modifications, the final heap is



7.2.4

Run-time Analysis

Accessing the top object is $\mathbf{\Theta}(1)$

Popping the top object is $\mathbf{O}(\ln(n))$

– We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

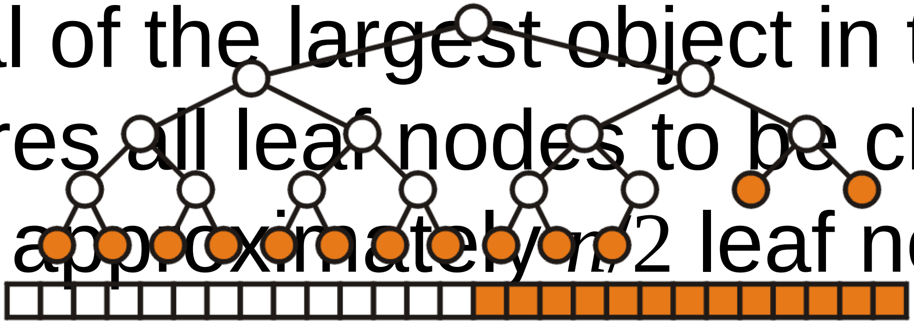
How about push - $\mathbf{O}(\ln(n))$

7.2.4

Run-time Analysis

An arbitrary removal requires that all entries in the heap be checked: $\mathbf{O}(n)$

A removal of the largest object in the heap still requires all leaf nodes to be checked – there are approximately $n/2$ leaf nodes: $\mathbf{O}(n)$

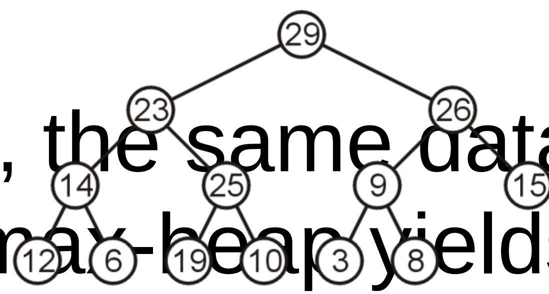


7.2.5

Binary Max Heaps

A binary max-heap is identical to a binary min-heap except that the parent is always larger than either of the children

For example, the same data as before stored as a max-heap yields



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	29	23	26	14	25	9	15	12	6	19	10	3	8		

7.2.6

Priority Queues

Now, does using a heap ensure that that object in the heap which:

- has the highest priority, and
- of that highest priority, has been in the heap the longest

Consider inserting seven objects, all of the same priority (colour indicates order):

2, 2, 2, 2, 2, 2, 2

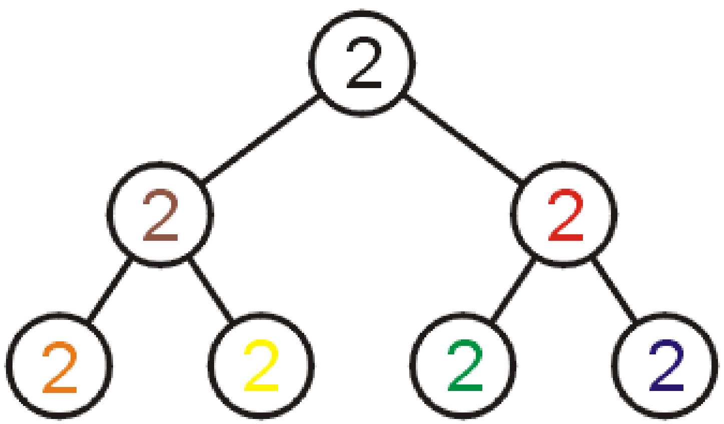
7.2.6

Priority Queues

Whatever algorithm we use for promoting must ensure that the first object remains in the root position

– Thus, we must use an insertion technique where we only percolate up if the priority is lower

The result:



Challenge:

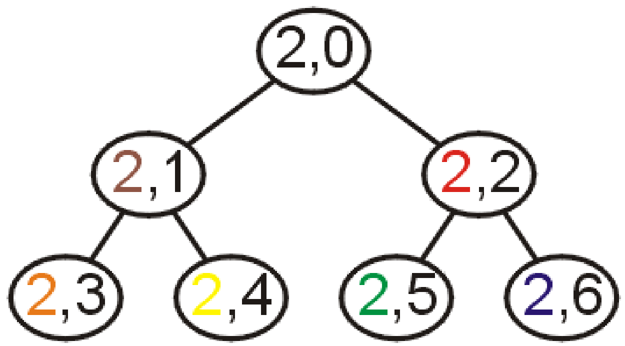
– Come up with an algorithm which removes all seven objects in the original order

7.2.6

Lexicographical Ordering

A better solution is to modify the priority:

- Track the number of insertions with a counter k (initially 0)
- For each ins… …n, create a hybrid priorit…

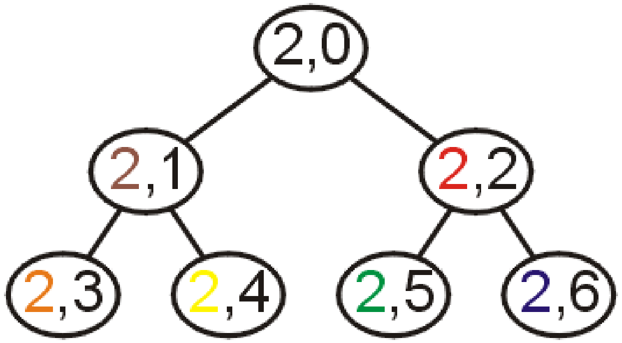


$(n_1, k_1) < (n$ … $_2$ and $k_1 < k_2)$

7.2.6

Priority Queues

Removing the objects would be in the following order:

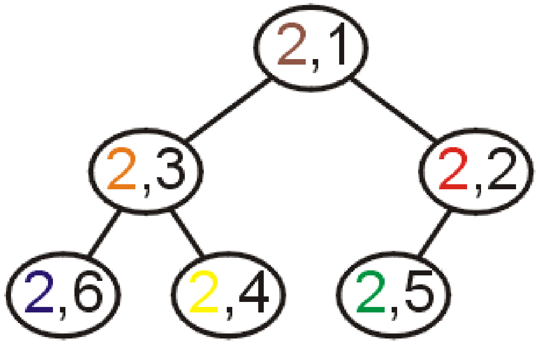


7.2.6

Priority Queues

Popped: 2

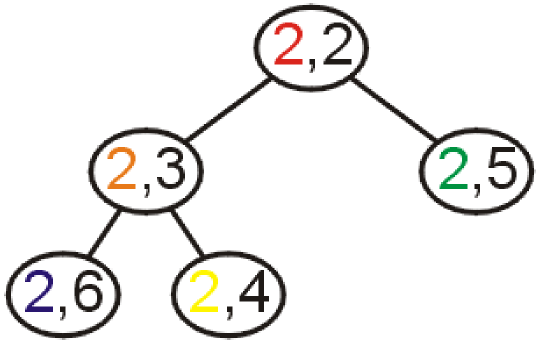
– First, (2,1) < (2, 2) and (2, 3) < (2, 4)



7.2.6

Priority Queues

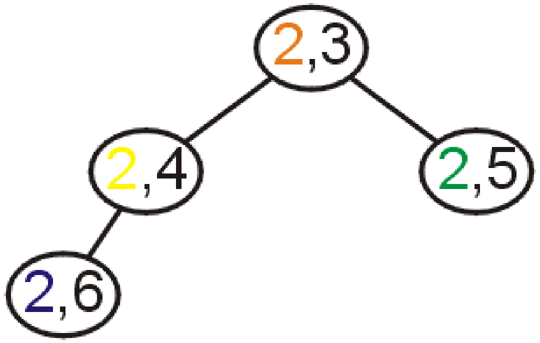
Removing the objects would be in the following order:



7.2.6

Priority Queues

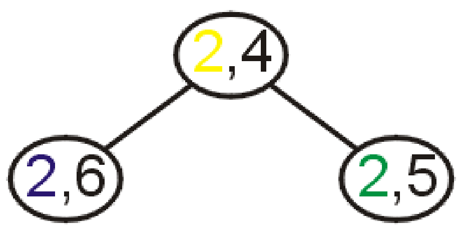
Removing the objects would be in the following order:



7.2.6

Priority Queues

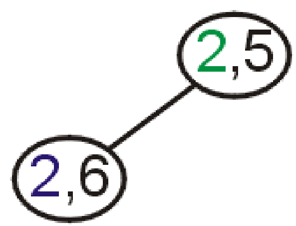
Removing the objects would be in the following order:



7.2.6

Priority Queues

Removing the objects would be in the following order:



References

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §7.2.3, p.144.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §7.1-3, p.140-7.

[3] Weiss, *Data Structures and Algorithm Analysis in C++*, *3rd Ed.*, Addison Wesley, §6.3, p.215-25.

Usage Notes

- These slides are made publicly available on the web for anyone to use
- If you choose to use them, or a part thereof, for a course at another institution, I ask only three things:
  - that you inform me that you are using the slides,
  - that you acknowledge my work, and
  - that you alert me of any mistakes which I made or changes which you make, and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides

Sincerely,

Douglas Wilhelm Harder, MMath

dwharder@alumni.uwaterloo.ca