



a place of mind

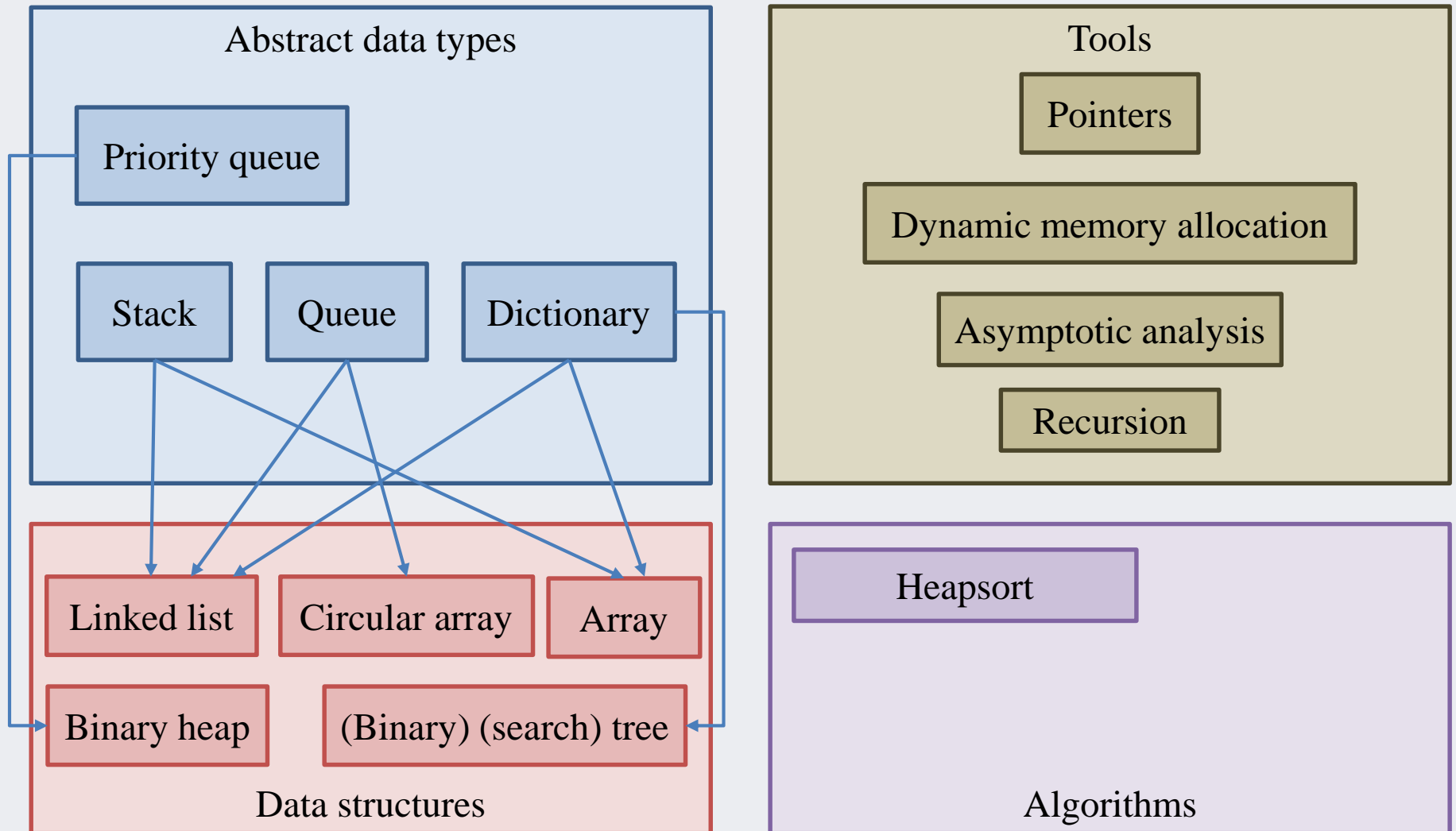
THE UNIVERSITY OF BRITISH COLUMBIA

Priority queue and binary heap

Priority queue ADT
Binary heap properties

The story thus far...

CPSC 259 topics up to this point



Priority queues?

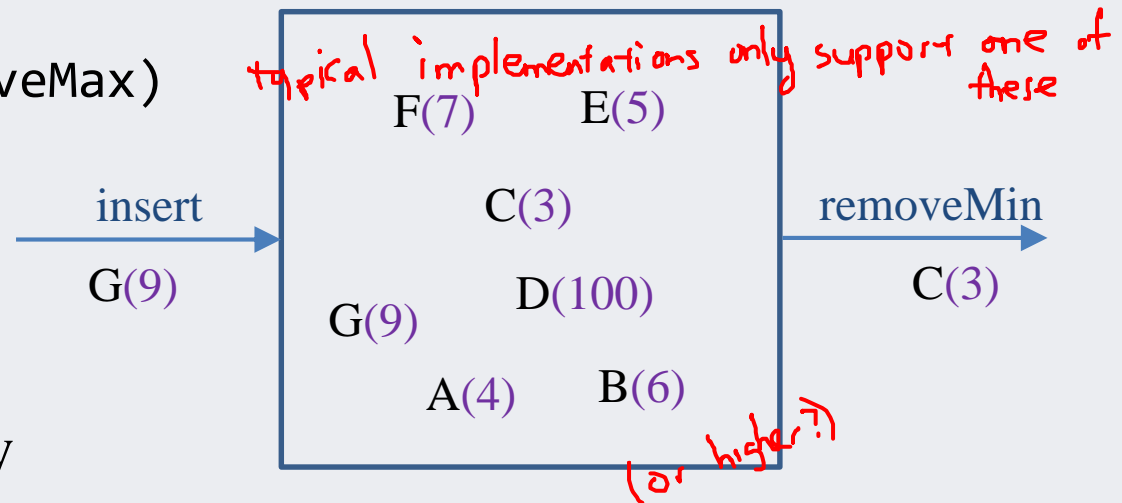
- Suppose Geoff has made a to-do list (with priority values):
 - 7 – Grade lab quiz
 - 2 – Vacuum home
 - 8 – Pay electricity bill
 - 1 – Sleep
 - 9 – Extinguish computer on fire
 - 2 – Eat
 - 8 – Publish lecture slides
 - 1 – Bathe
- We are interested in quickly finding the task with the highest priority

Priority queue ADT

- A collection organised so as to allow fast access to and removal of the largest (or smallest) element
 - *Prioritisation* is a weaker condition than *ordering*
 - Order of insertion is irrelevant
 - Element with the highest priority (whatever that means) is the first element to be removed
 - Not really a queue: not FIFO!

Priority queue ADT

- Priority queue operations
 - create
 - destroy
 - insert
 - removeMin (or removeMax)
 - isEmpty



- Priority queue property
 - For two elements x and y in the queue, if x has a lower priority value than y , x will be removed before y

Priority queue properties

- A priority queue is an ADT that maintains a multiset of items
 - vs set: a multiset allows duplicate entries
- Two or more distinct items in a priority queue may have the same priority
- If all items have the same priority, will it behave FIFO like a queue?
 - not necessarily! Due to implementation details

Priority queue applications

- Hold jobs for a printer in order of size
- Manage limited resources such as bandwidth on a transmission line from a network router
- Sort numbers
- Anything *greedy*: an algorithm that makes the "locally best choice" at each step

Data structures for priority queues

Worst case complexities

Structure	insert	removeMin
Unordered list	$O(1)$	$O(n)$
Ordered list	$O(n)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$

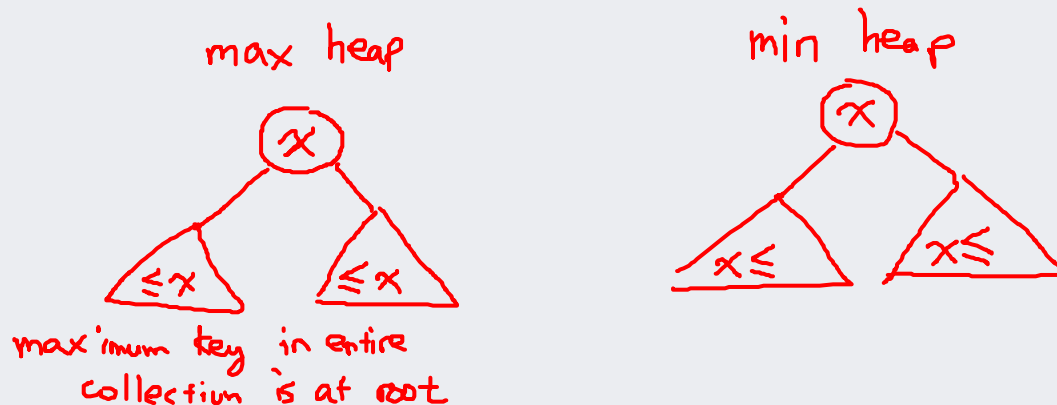
A balanced BST can do insert and removeMin in $O(\log n)$, but we cannot guarantee balance

Binary heap	$O(\log n)$	$O(\log n)$
-------------	-------------	-------------

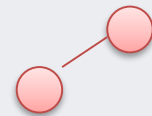
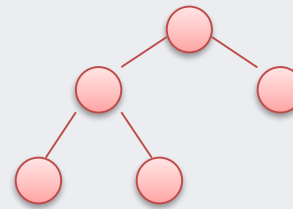
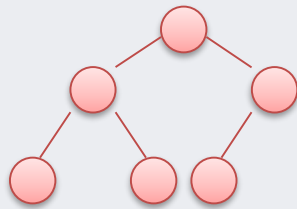
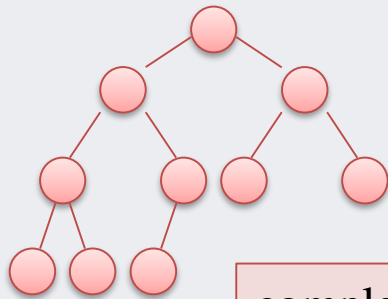
Binary heap

A complete, partially-ordered binary tree

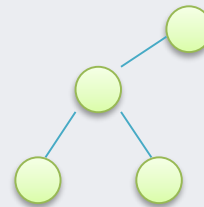
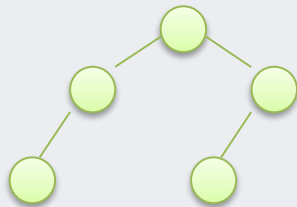
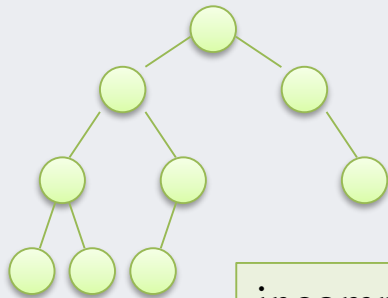
- A *heap* is binary tree with two properties
- Heaps are *complete*
 - All levels, except the bottom, must be completely filled in
 - The leaves on the bottom level are as far to the left as possible
- Heaps are *partially ordered*
 - For a *max* heap – the value of a node is at least as large as its children's values
 - For a *min* heap – the value of a node is no greater than its children's values



Review: complete binary trees



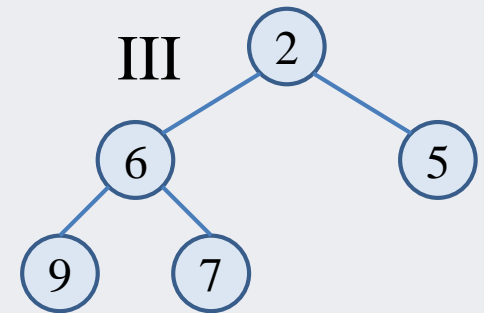
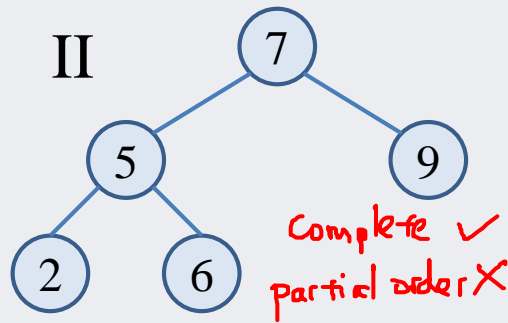
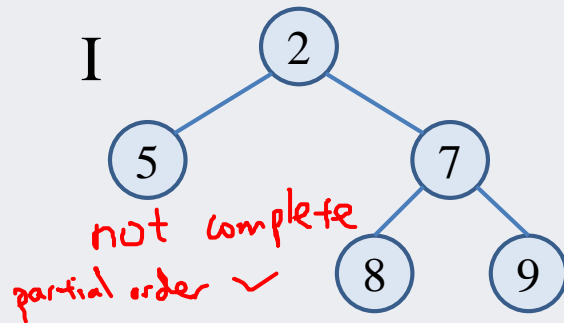
complete binary trees



incomplete binary trees

iClicker 10.1

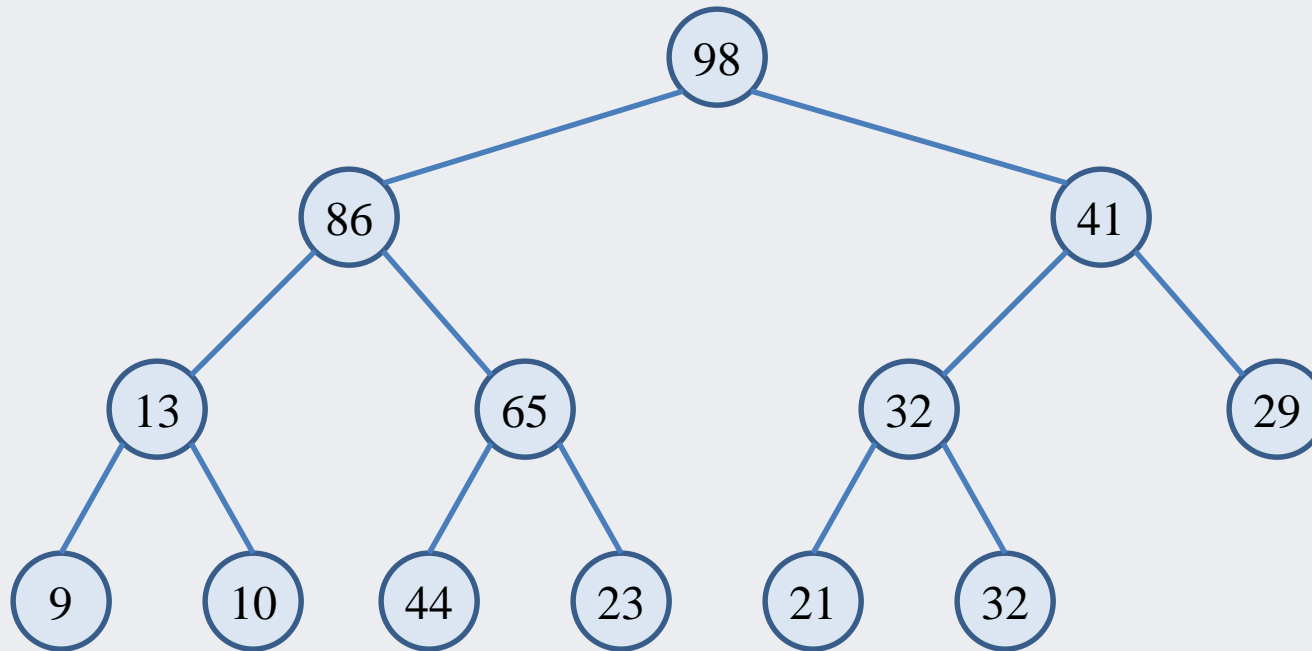
- Which of the following represent a valid min-heap? *complete partial order*



- A. Only II
- ☒ B. Only III
- C. Both I and III
- D. Both II and III
- E. All of I, II, and III

Partially ordered tree

max heap example



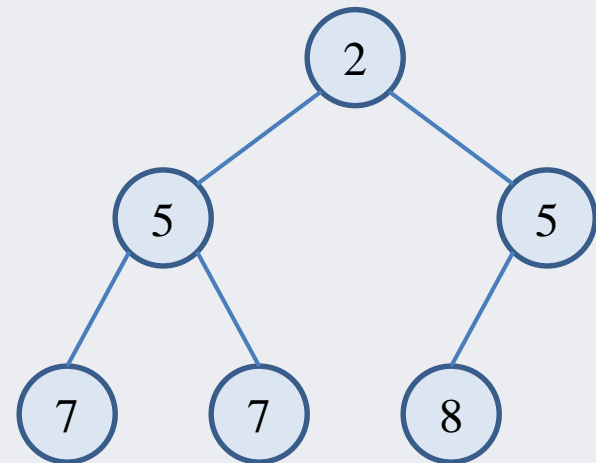
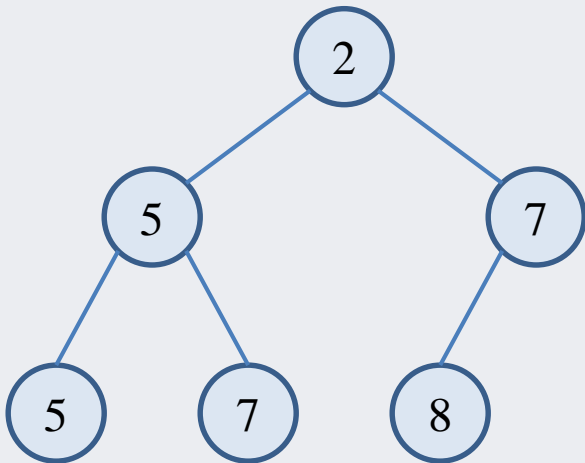
Heaps are *not* fully ordered – an in-order traversal would result in:

9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 32, 41, 29

Duplicate priority values

Min heap examples

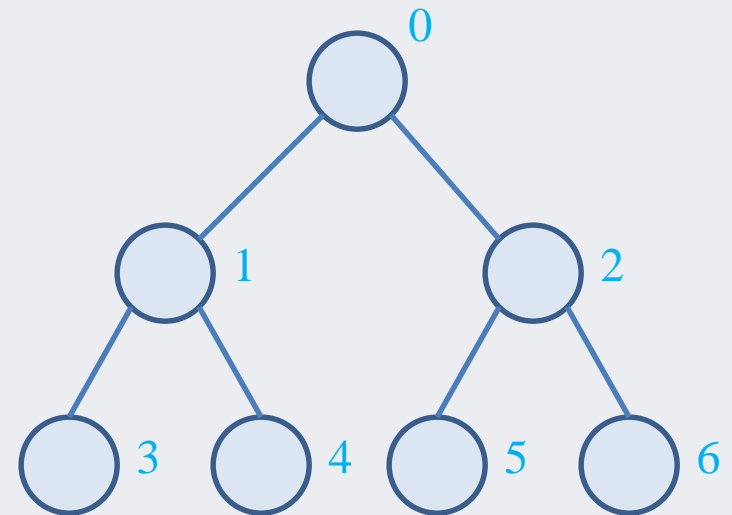
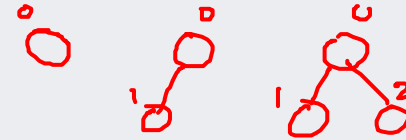
- It is important to realise that two binary heaps can contain the same data, but items may appear in different positions in the structure



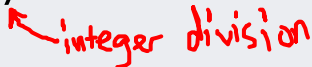
Heap implementation

Using an array

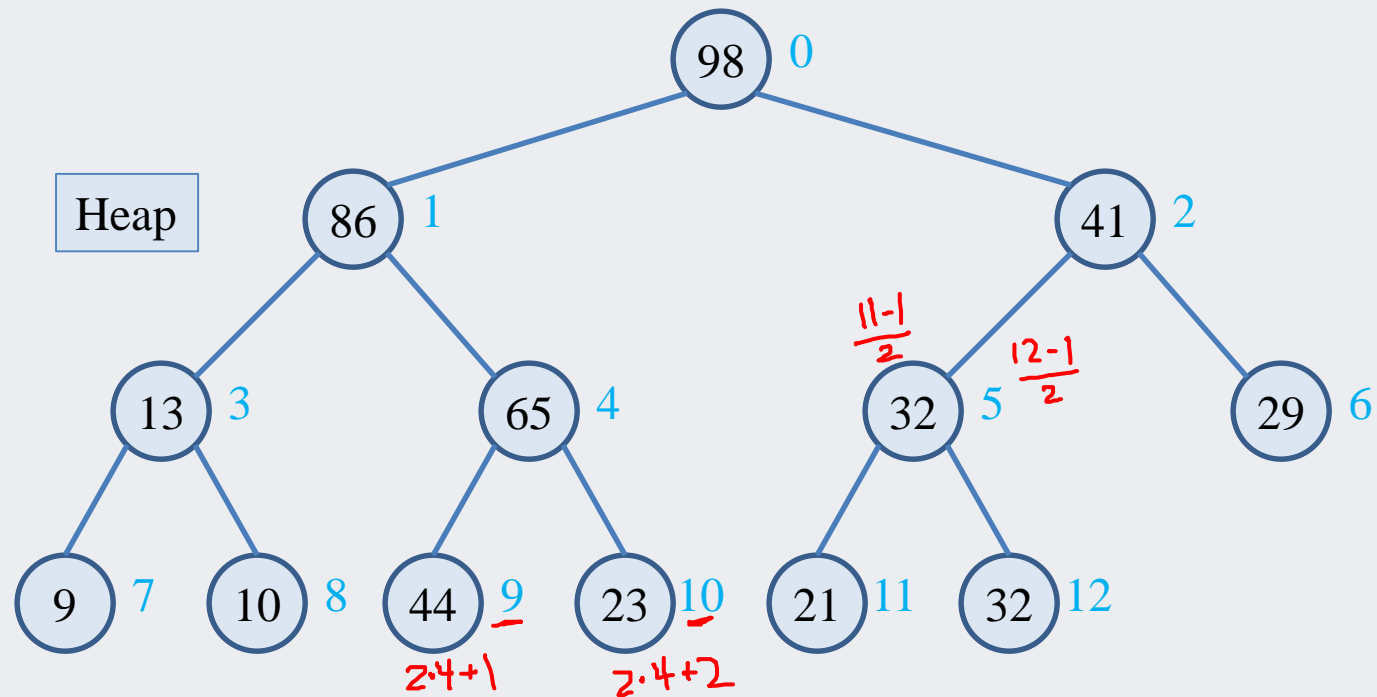
- Heaps can be implemented using *arrays*
- There is a natural method of indexing tree nodes
 - Index nodes from top to bottom and left to right as shown on the right
 - Because heaps are *complete* binary trees there can be no gaps in the array



Referencing nodes

- To move around in the tree, it will be necessary to find the index of the parents of a node
 - or the children of a node
- The array is indexed from 0 to $n - 1$
- Each level's nodes are indexed from:
 - $2^{\text{level}} - 1$ to $2^{\text{level}+1} - 2$ (where the root is level 0)
 - The children of a node i , are the array elements indexed at $2i + 1$ and $2i + 2$
 - The parent of a node i , is the array element indexed at $(i - 1)/2$


Array heap example



Underlying array

value	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Heap implementation

```
typedef struct MinHeap {  
    int size;        // number of stored elements  
    int capacity;    // maximum capacity of array  
    int* arr;        // array in dynamic memory  
} MinHeap;
```

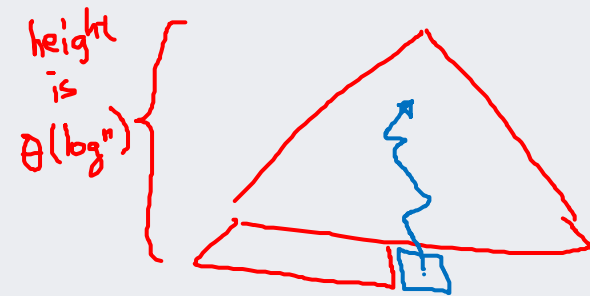
```
void initializeMinHeap(MinHeap* h, int initcapacity) {  
    h->size = 0;  
    h->capacity = initcapacity;  
    h->arr = (int*) malloc(h->capacity * sizeof(int));  
}
```

Heap insertion

- On insertion the heap properties have to be maintained, remember that
 - A heap is a complete binary tree and
 - A partially ordered binary tree
- There are two general strategies that could be used to maintain the heap properties
 - ✓▪ Make sure that the tree is complete and then fix the ordering or
 - Make sure the ordering is correct first
 - Which is better?

Heap insertion sketch

- The insertion algorithm first ensures that the tree is complete
 - Make the new item the first available (left-most) leaf on the bottom level
 - i.e. the first free element in the underlying array
- Fix the partial ordering
 - Compare the new value to its parent
 - Swap them if the new value is greater than the parent
 - Repeat until this is not the case
 - Referred to as *heapify up*, *percolate up*, or *trickle up*, *bubble up*, etc.

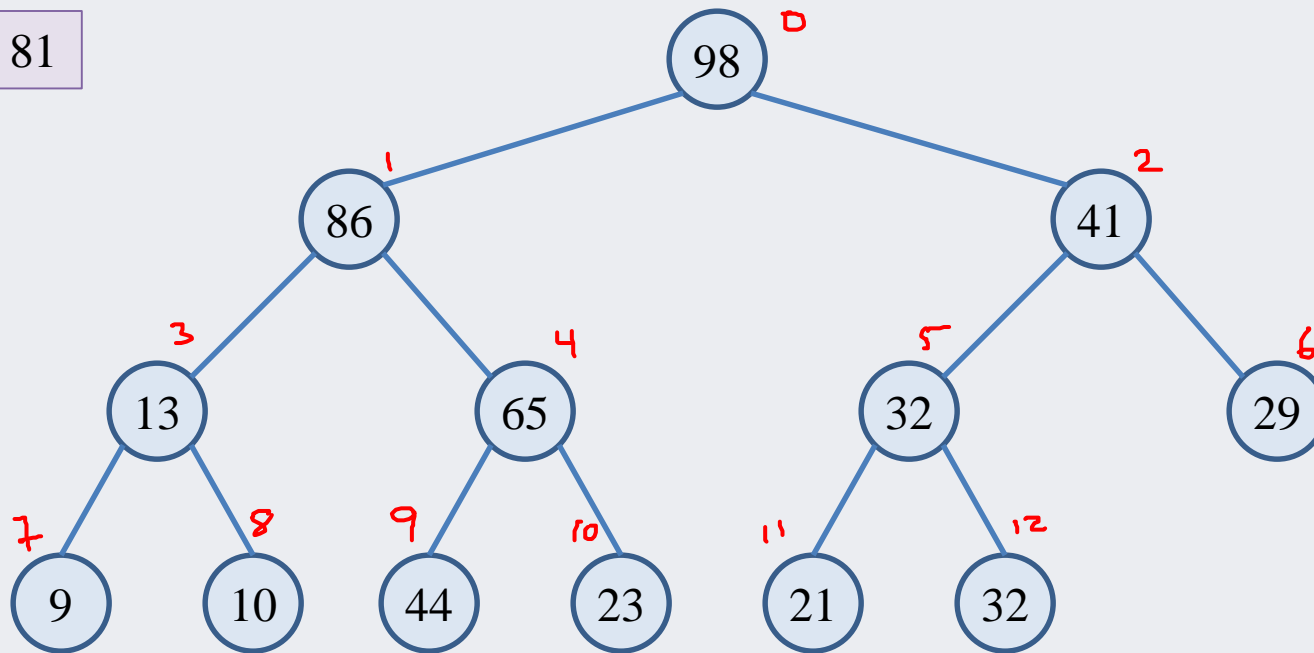


cost of insertion : $O(\log n)$

Heap insertion example

max heap

Insert 81

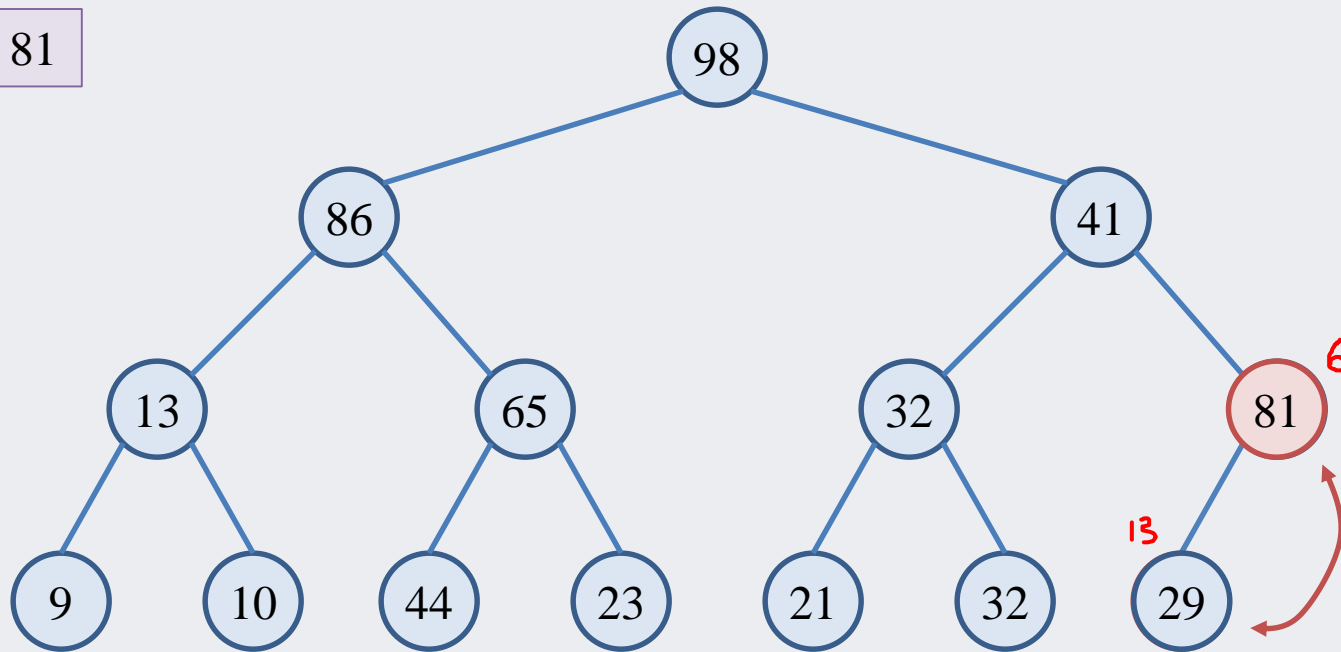


value	98	86	41	13	65	32	29	9	10	44	23	21	32	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap insertion example

max heap

Insert 81



parent: $(13 - 1) / 2 = 6$

value	98	86	41	13	65	32	81	9	10	44	23	21	32	29
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap insertion example

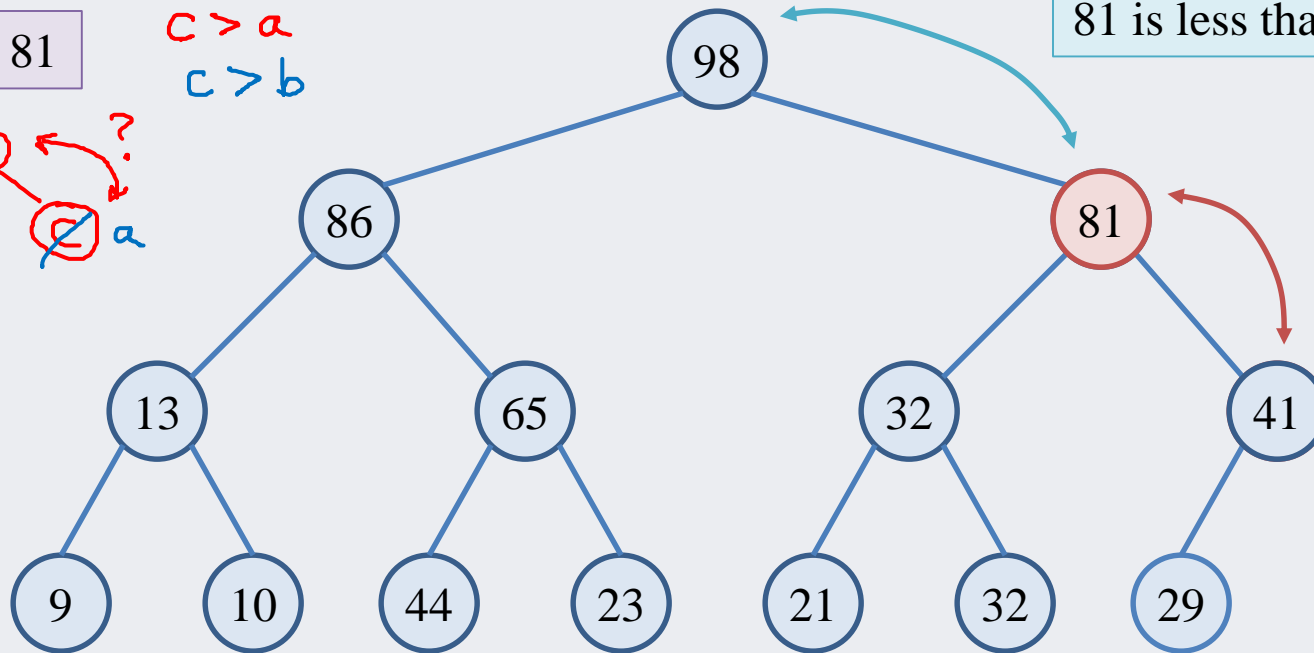
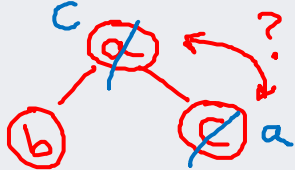
max heap

$$a \geq b$$

Insert 81

$$c > a$$
$$c > b$$

81 is less than 98 so finished

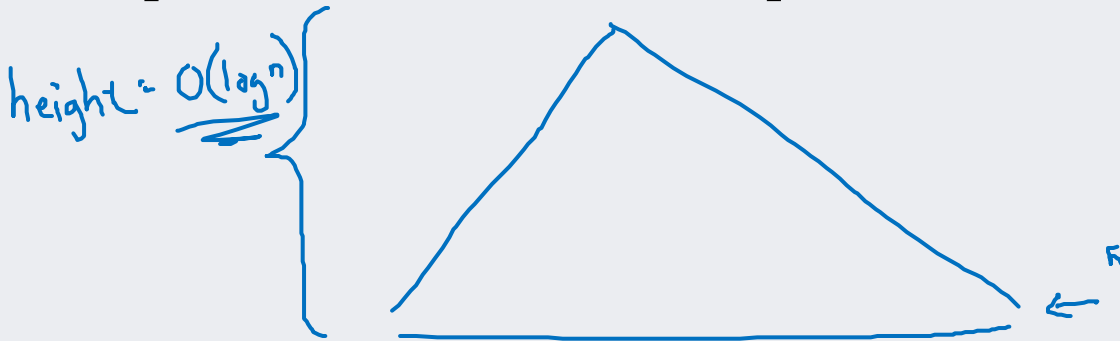


$$\text{parent: } (6 - 1) / 2 = 2$$

value	98	86	81	13	65	32	41	9	10	44	23	21	32	29
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap insertion complexity

- Item is inserted at the bottom level in the first available space
 - this can be tracked using the heap `size` attribute
 - $O(1)$ access using array index
- Repeated heapify-up operations. How many?
 - each heapify-up operation moves the inserted value up one level in the tree
 - Upper limit on the number of levels in a complete tree? $O(\log n)$
- Heap insertion has worst-case performance of $O(\log n)$



Exercise

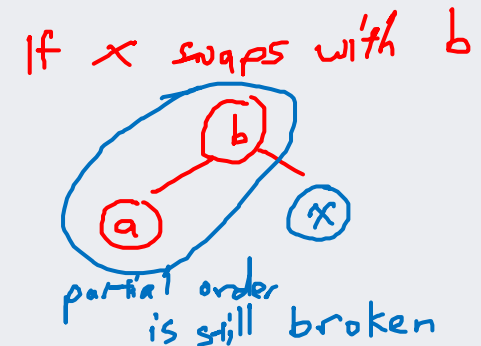
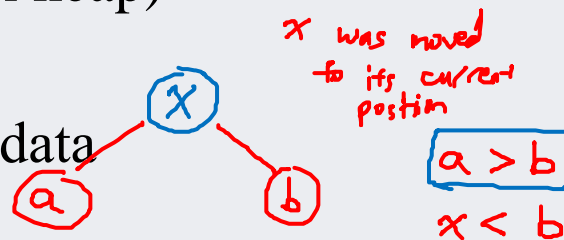
- Insert the following sequence of items into an initially empty *max heap*:
34, 76, 11, 32, 73, 9, 50, 65, 41, 27, 3, 88
- Draw the underlying array after all insertions have completed
- Insert the same sequence of items into an initially empty *min heap* and draw the array after all insertions have been completed.
- For inserting n items, what is the total complexity?
- Implement recursive and iterative implementations:
 - `void heapifyUp(MinHeap* h, int index);`

Building a heap (an heap?)

- A heap can be constructed by repeatedly inserting items into an empty heap. Complexity?
 - $O(\log n)$ per item, n items
 - $O(n \log n)$ total
- More about this later

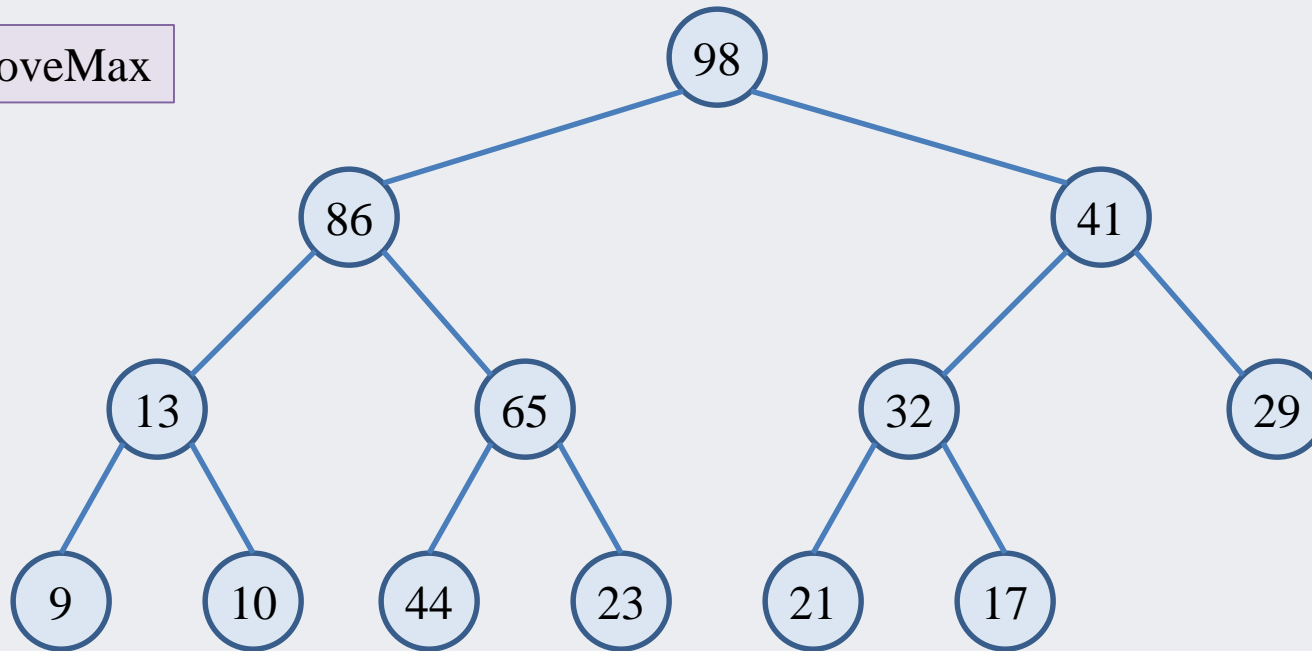
Removing the priority item

- Heap properties must be satisfied after removal
- Make a temporary copy of the root's data
- Similarly to the insertion algorithm, first ensure that the heap remains complete
 - Replace the root node with the right-most leaf
 - i.e. the highest (occupied) index in the array
- Swap the new root with its *largest valued child* until the partially ordered property holds (for a max heap)
 - i.e. heapifyDown
- Return the copied root's data



Heap removal example

removeMax

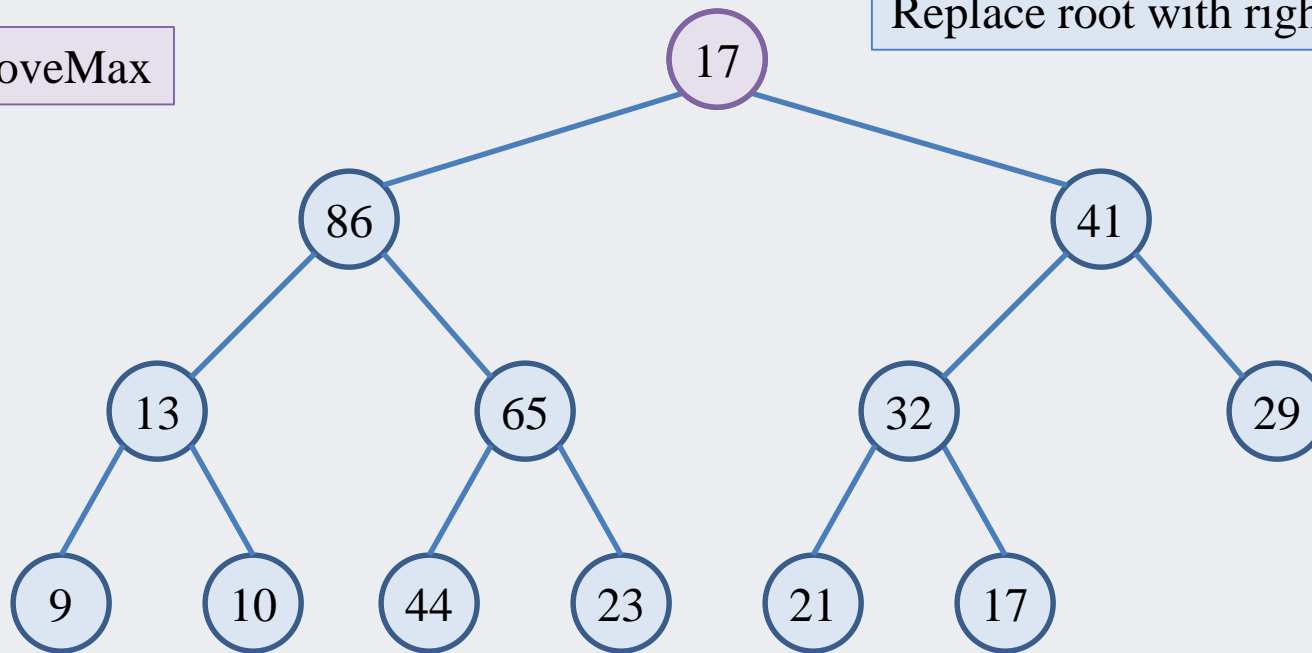


value	98	86	41	13	65	32	29	9	10	44	23	21	17
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Heap removal example

removeMax

Replace root with rightmost leaf

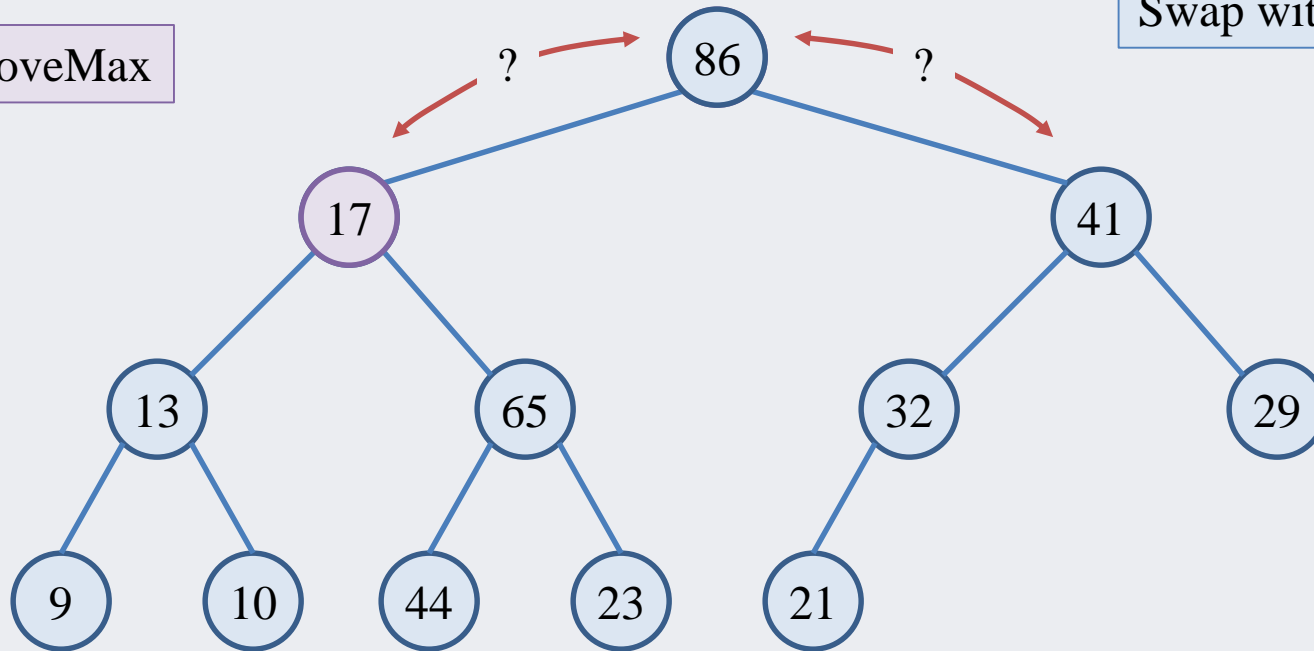


value	17	86	41	13	65	32	29	9	10	44	23	21	17
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Heap removal example

removeMax

Swap with largest child



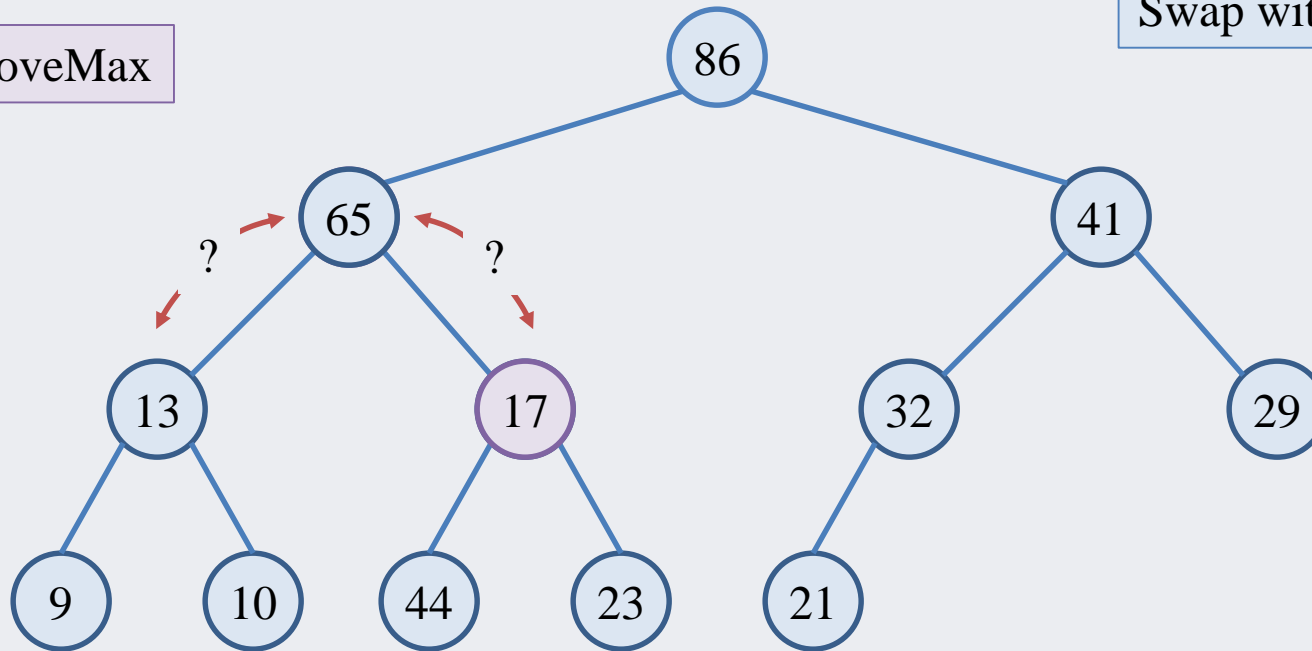
children of root: $2 \cdot 0 + 1, 2 \cdot 0 + 2 = 1, 2$

value	86	17	41	13	65	32	29	9	10	44	23	21	
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Heap removal example

Swap with largest child

removeMax



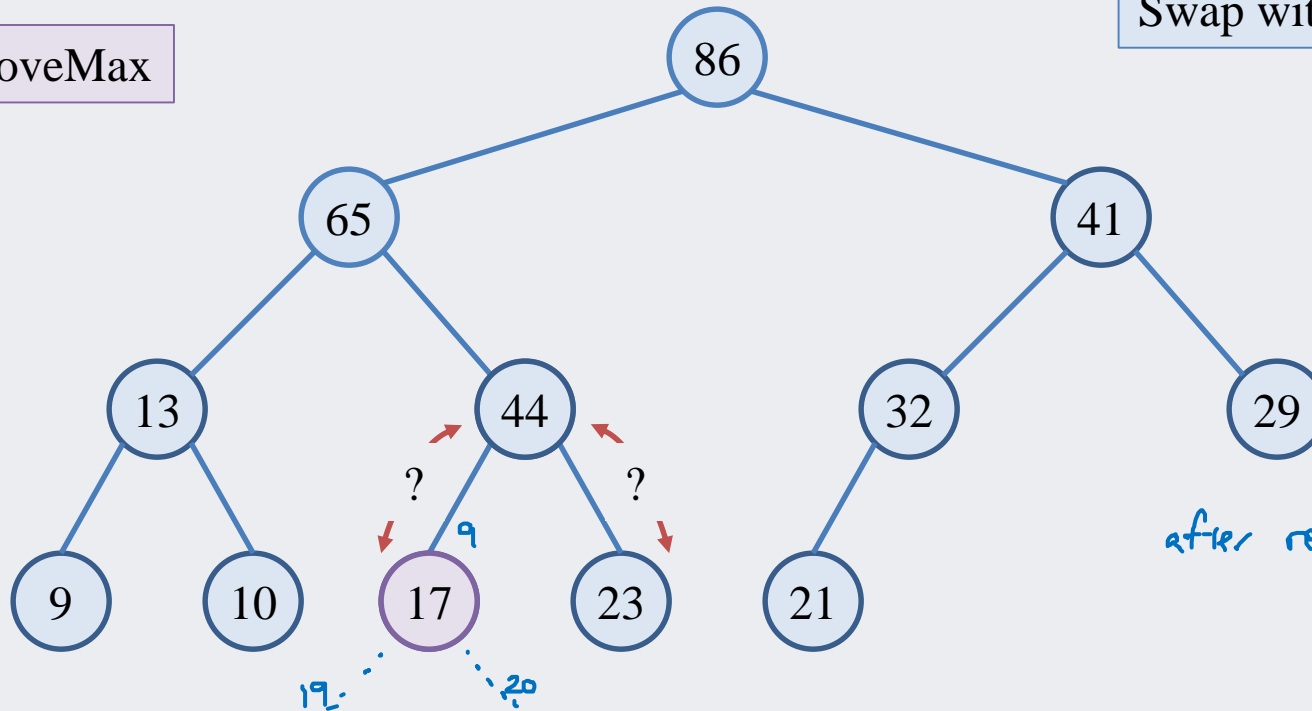
children of 1: $2 \cdot 1 + 1, 2 \cdot 1 + 2 = 3, 4$

value	86	65	41	13	17	32	29	9	10	44	23	21	
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Heap removal example

Swap with largest child

removeMax



size: 12

children of 4: $2*4+1$, $2*4+2 = 9, 10$

value	86	65	41	13	44	32	29	9	10	17	23	21	
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Complexity of removeMin / removeMax

- Analysis is similar to insertion
- Replace root with last element – $O(1)$
- Repeated heapify-down operations starting from root level
 - each heapify-down moves one level closer to the bottom of the tree
 - How many levels? $O(\log n)$
- Removing the priority item from a heap is also $O(\log n)$ in the worst case

Array implementation and insertion

- Note that like other array-based structures, we have a limited capacity at creation time
 - What to do when the array is full? Expand it!
- Since array indices correspond exactly to node positions in the tree, and nodes should remain in their original positions after expanding the array, we can simply copy into the same indices

value	3	27	9	31	46	12
index	0	1	2	3	4	5

value	3	27	9	31	46	12	62					
index	0	1	2	3	4	5	6	7	8	9	10	11

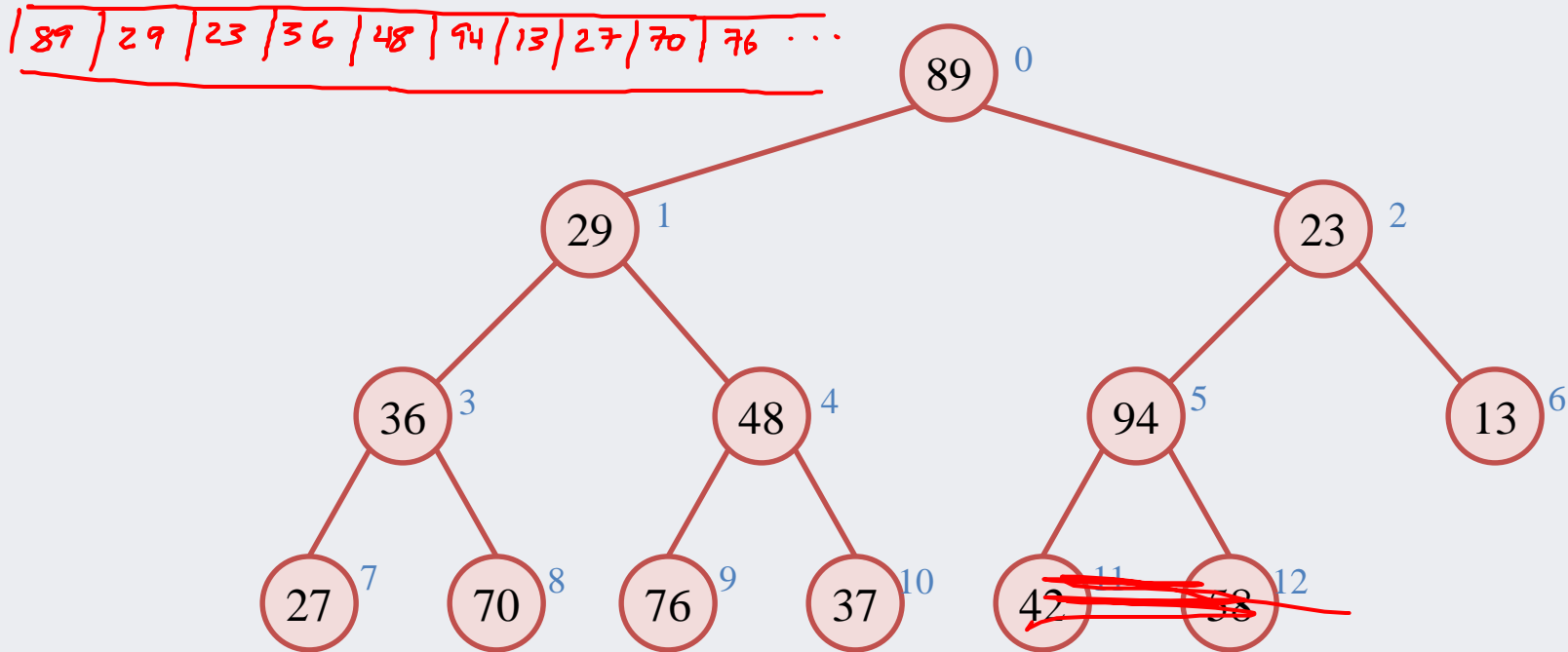
Sorting with heaps?

- **Observation 1:** Removal of a node from a heap can be performed in $O(\log n)$ time
- **Observation 2:** Nodes are removed in order
- **Conclusion:** Removing all of the nodes one by one would result in sorted output
- **Analysis:** Removal of *all* the nodes from a heap is a $O(n \log n)$ operation

Sorting with heaps

- A heap is (usually) an array, and can be used to return sorted data in $O(n \log n)$ time
- However, we can't assume that the data to be sorted just happens to be a heap to begin with!
 - But we can *put* it in a heap.
 - Inserting an item into a heap is a $O(\log n)$ operation so inserting n items is $O(n \log n)$
 - But we can do slightly better than just repeatedly calling the insertion algorithm

buildHeap



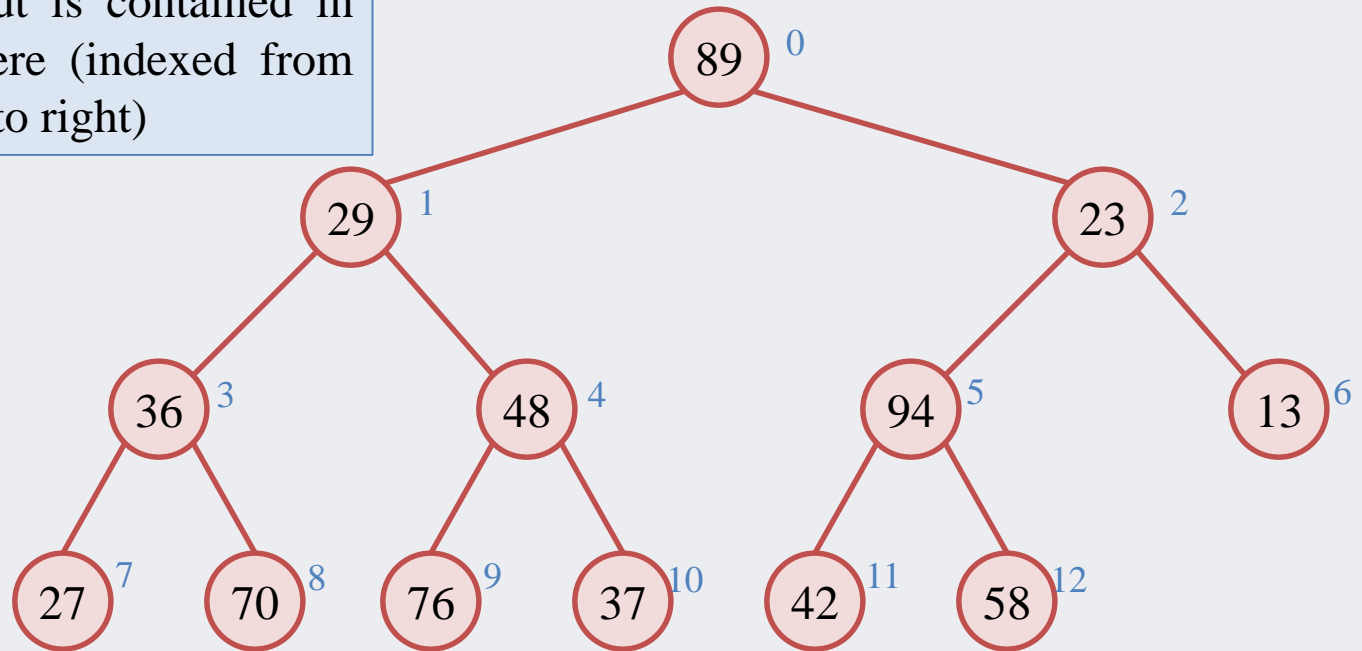
- (Max) Heap invariant:
 - Key value in every node must be larger than key value of children
- Given the tree representation of some unordered array
 - Where can the heap invariant initially be violated? *may be violated at internal nodes*
invariant holds at leaf nodes

buildHeap

- To create a heap from an unordered array repeatedly call `heapifyDown`
 - Any subtree in a heap is itself a heap
 - Call `heapifyDown` on elements in the upper $1/2$ of the array
 - Since lower half are leaf nodes and are already heaps
 - Start with index $n/2$ and work up to index 0
 - i.e. from the last non-leaf node to the root
- `heapifyDown` does not need to be called on the lower half of the array (the leaves)
 - Since `heapifyDown` restores the partial ordering from any given node down to the leaves, the heap property remains satisfied at deeper levels of the tree

buildHeap example

Assume unsorted input is contained in an array as shown here (indexed from top to bottom and left to right)

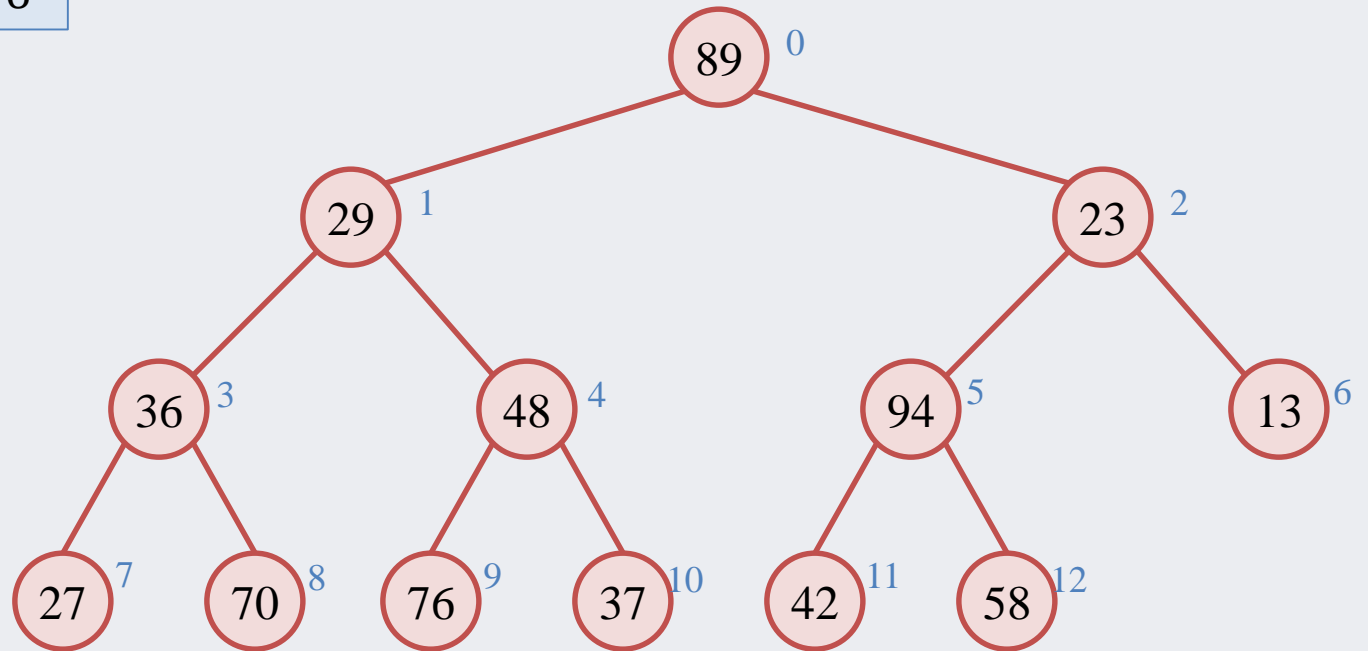


buildHeap example

$n = 13, (n - 1)/2 = 6$

heapifyDown(6)

heapifyDown(5)



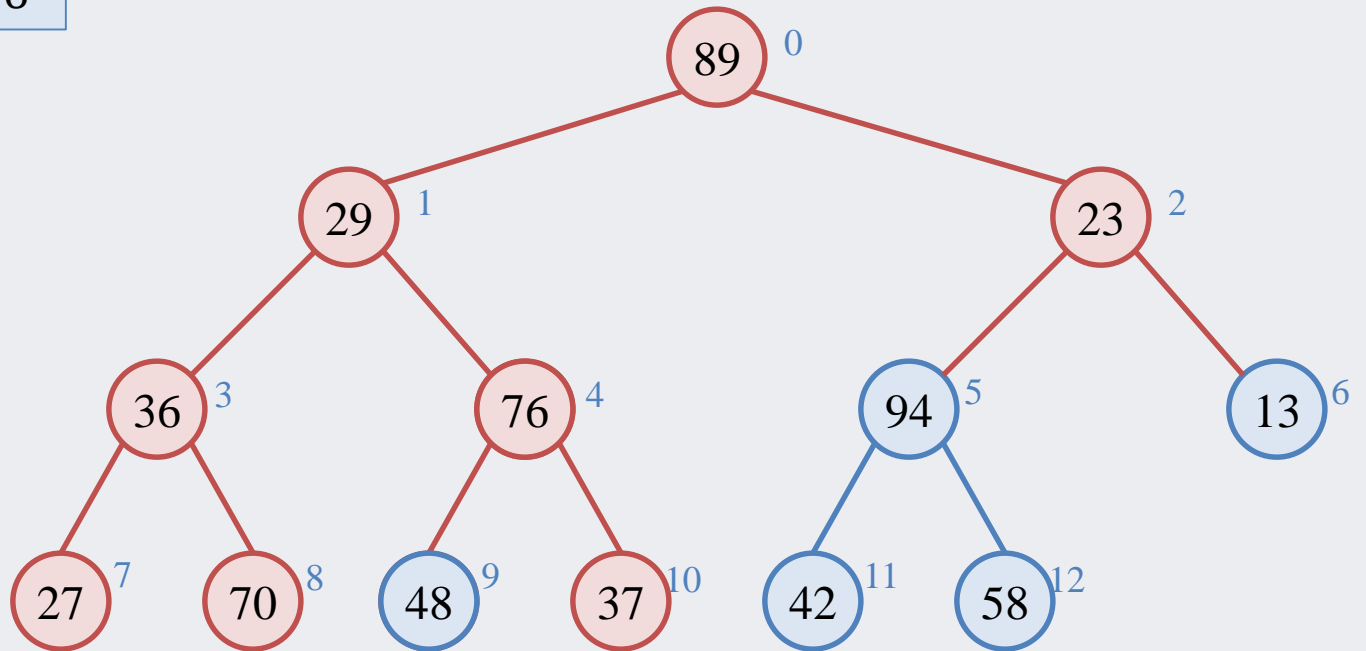
buildHeap example

$n = 13, (n - 1)/2 = 6$

heapifyDown(6)

heapifyDown(5)

heapifyDown(4)



buildHeap example

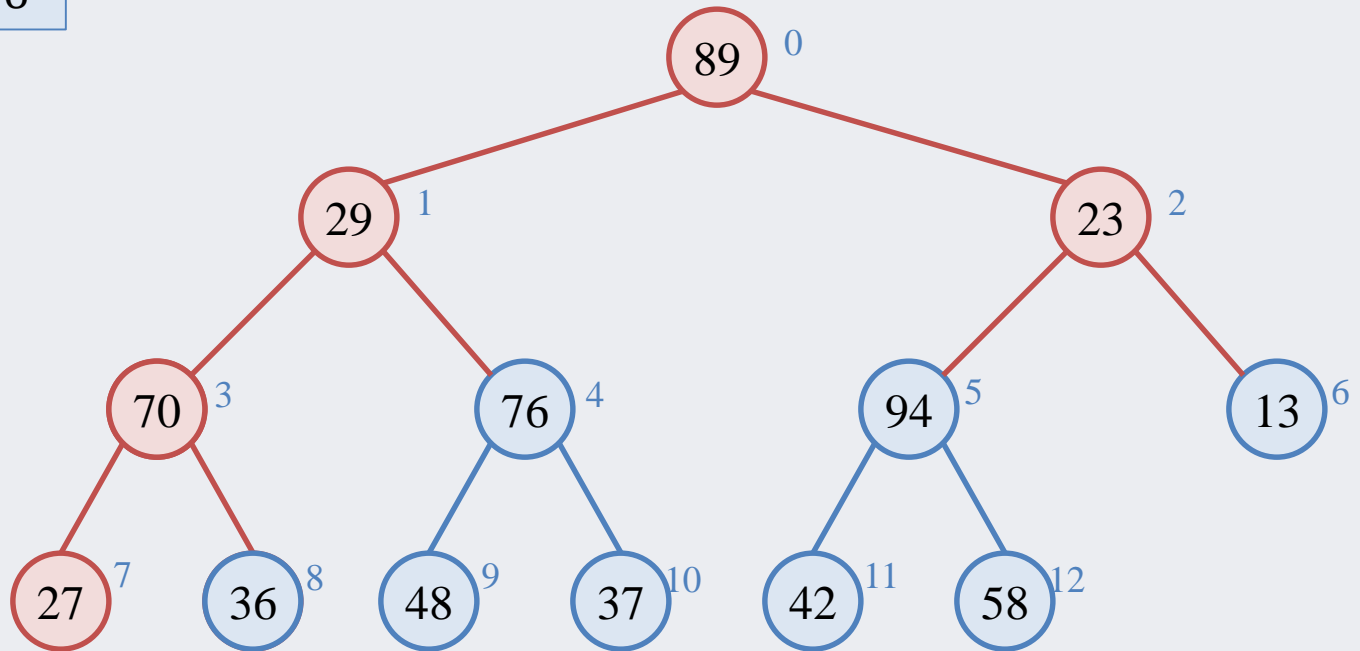
$n = 13, (n - 1)/2 = 6$

heapifyDown(6)

heapifyDown(5)

heapifyDown(4)

heapifyDown(3)



buildHeap example

$n = 13, (n - 1)/2 = 6$

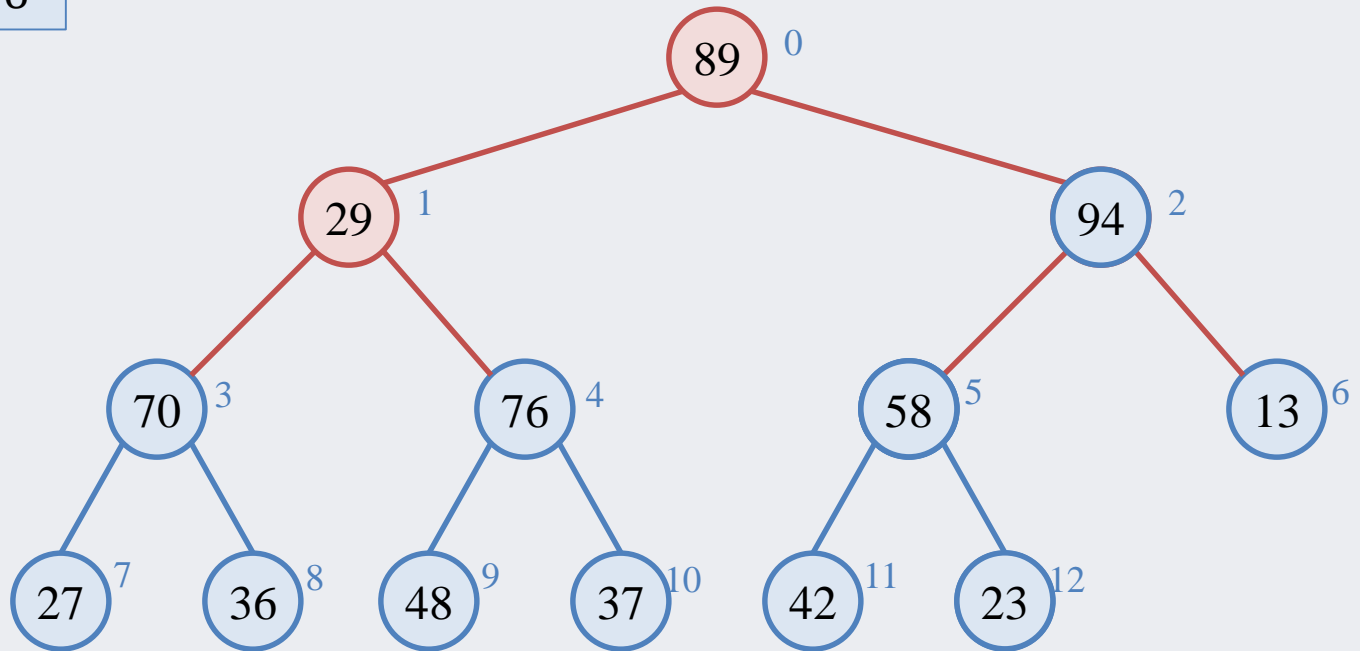
heapifyDown(6)

heapifyDown(5)

heapifyDown(4)

heapifyDown(3)

heapifyDown(2)



buildHeap example

$n = 13, (n - 1)/2 = 6$

heapifyDown(6)

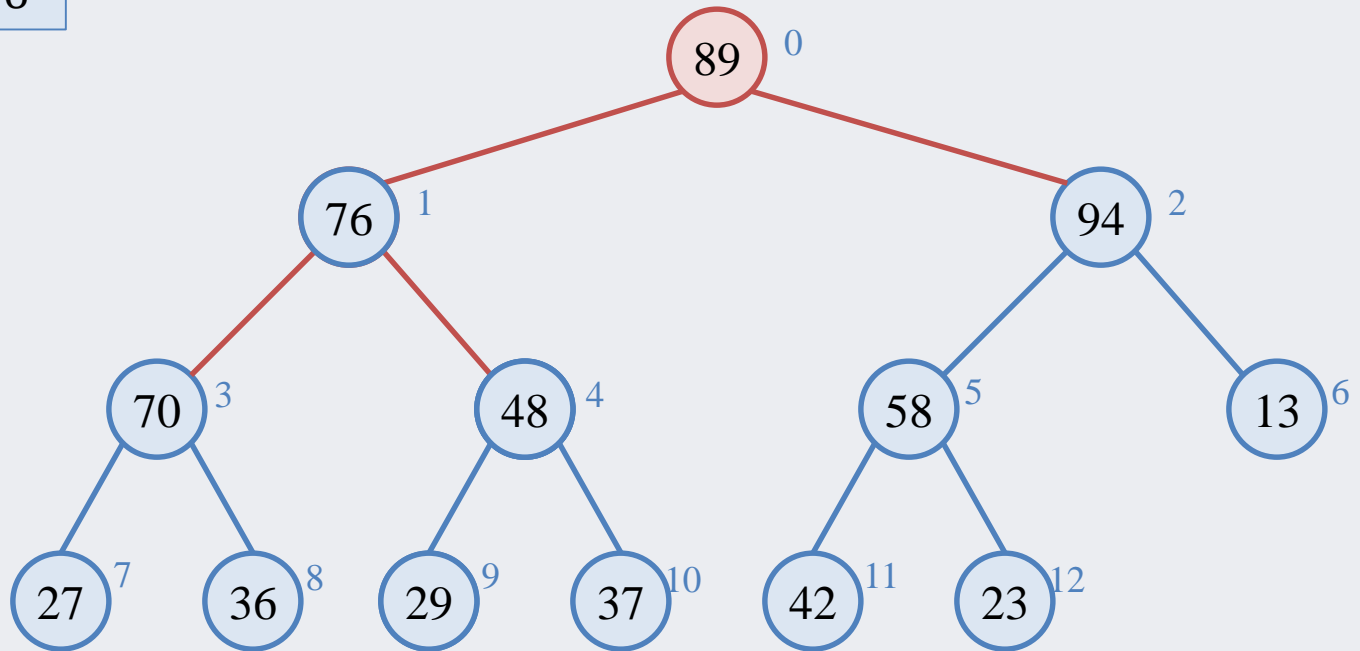
heapifyDown(5)

heapifyDown(4)

heapifyDown(3)

heapifyDown(2)

heapifyDown(1)



buildHeap example

$n = 13, (n - 1)/2 = 6$

heapifyDown(6)

heapifyDown(5)

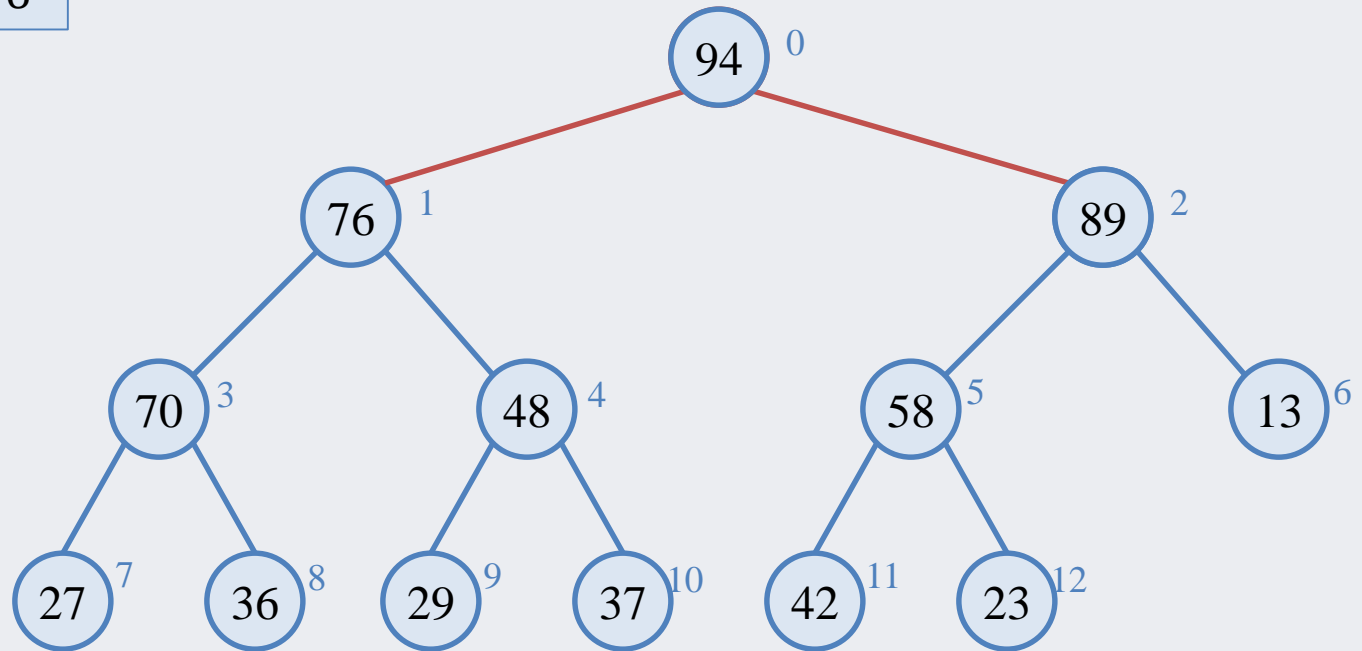
heapifyDown(4)

heapifyDown(3)

heapifyDown(2)

heapifyDown(1)

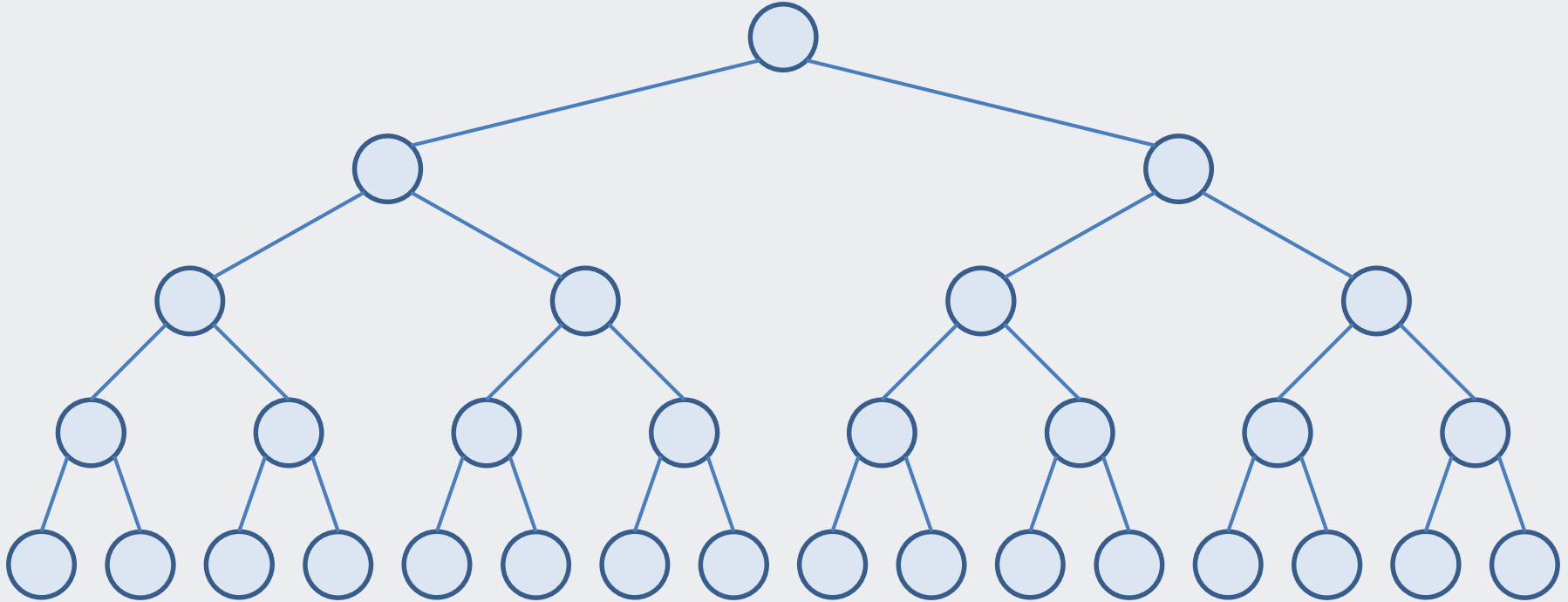
heapifyDown(0)



buildHeap complexity

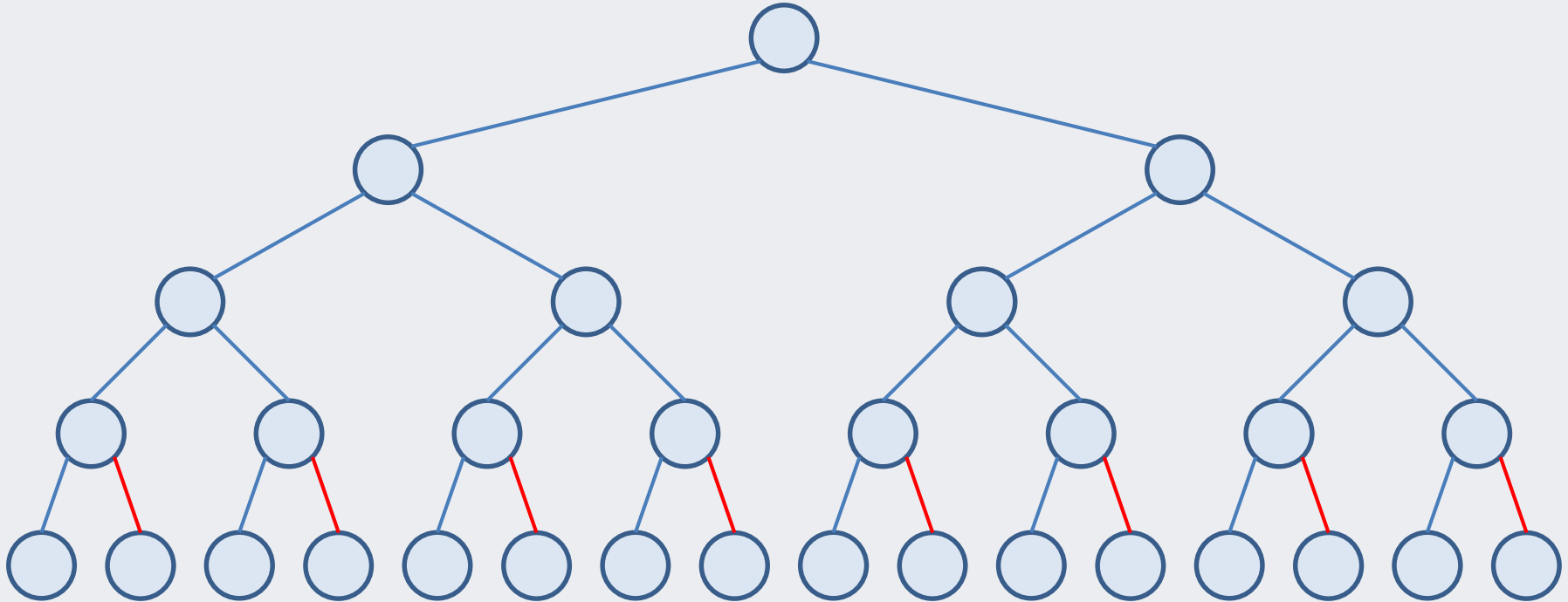
- `heapifyDown` is called on half the array
 - The cost for `heapifyDown` is $O(\text{height})$
 - It would appear that `buildHeap` cost is $O(n \log n)$
- In fact the cost is $O(n)$
- Let's look at the total number of swaps that can occur in the worst case
 - the bottom level is full, and contains all the highest priority values (for a max heap)

buildHeap, number of swaps

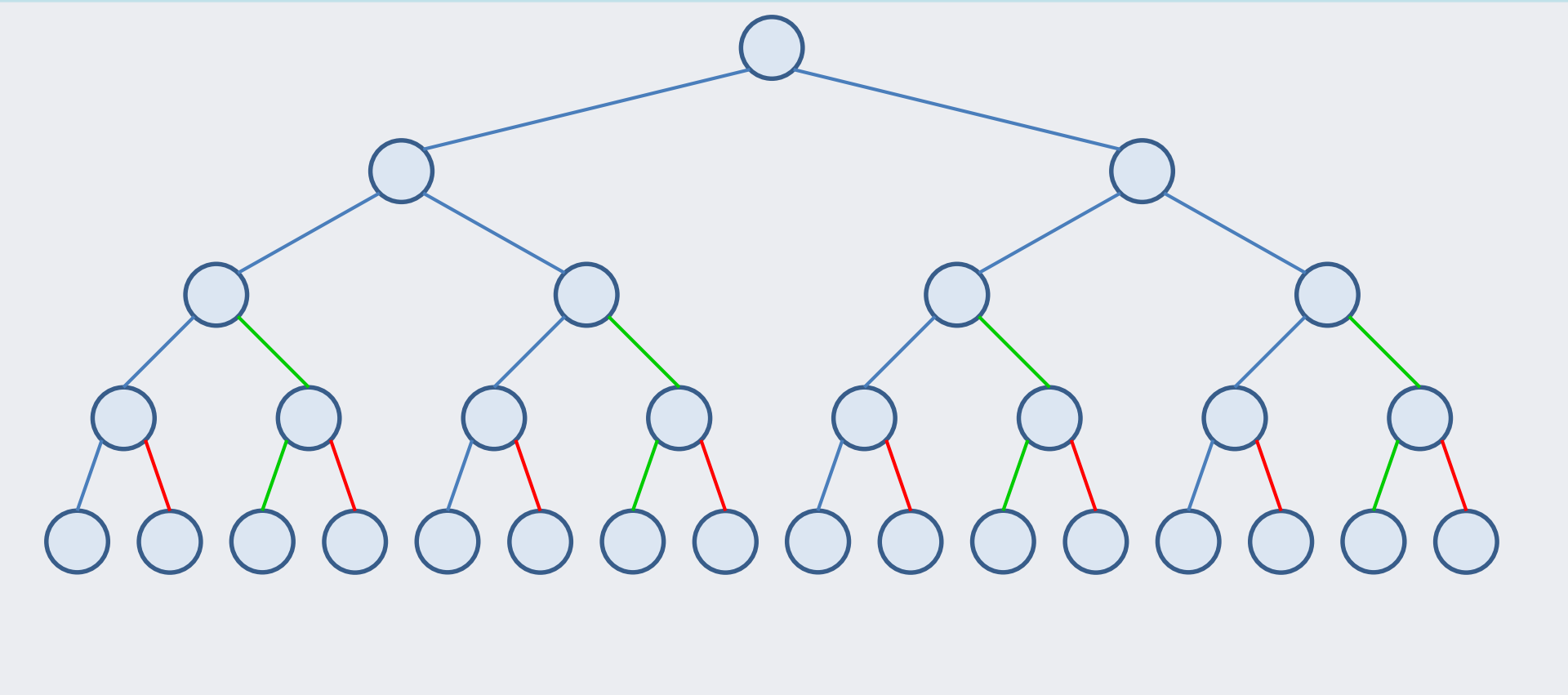


- heapifyDown must follow some path (along edges) to the bottom of the tree
 - let's count the total number of edges that can be followed in the worst case

buildHeap, number of swaps

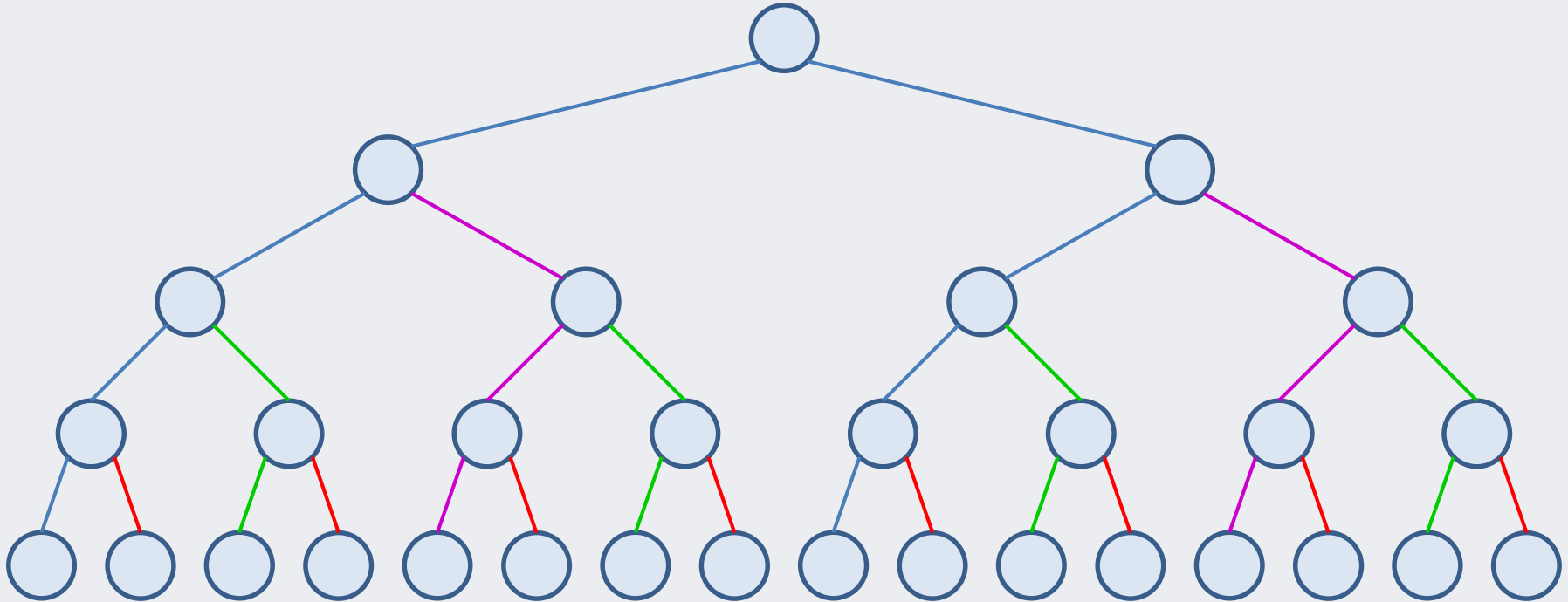


- At the next level, `heapifyDown` must still follow a single path down to the bottom level
 - for illustration, let's follow a path along edges that have not been coloured yet



- ...let's continue doing the same at the next level...

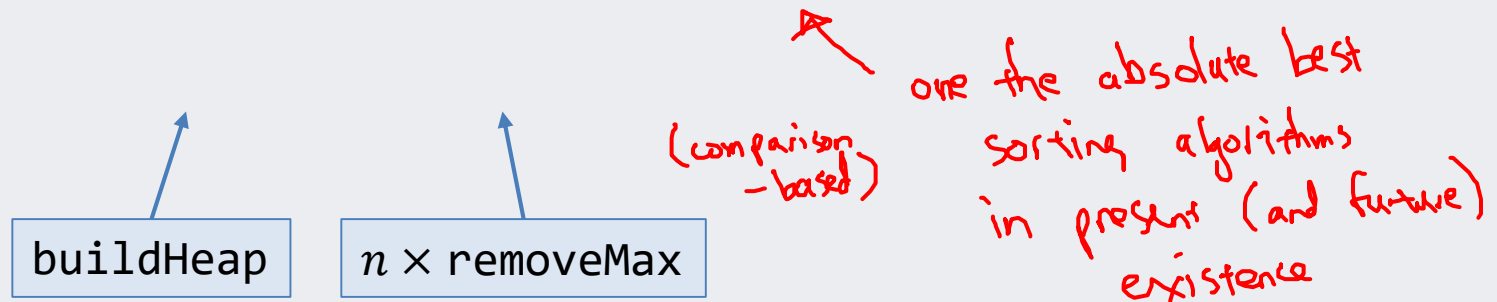
buildHeap, number of swaps



- ...and also at the root level
 - we have coloured the maximum number of edges that heapifyDown can use
 - what is the upper bound on the number of edges in a tree with n nodes?
 - exactly $n - 1$ edges, thus the worst case number of swaps is $O(n)$

Heapsort

- Heapify the array (max heap)
- Repeatedly remove the root (largest item)
 - After each removal swap the root with the last element in the tree
 - The array is divided into a heap part and a sorted part
- At the end of the sort the array will be sorted in ascending order
- Complexity: $O(n) + O(n \log n) = O(n \log n)$



Heapsort example

Starting array:

7	27	96	48	58	23	21	76	87	44
---	----	----	----	----	----	----	----	----	----

heap items

96	87	23	76	58	7	21	27	48	44
----	----	----	----	----	---	----	----	----	----

→

87	76	23	48	58	7	21	27	44	96
----	----	----	----	----	---	----	----	----	----

→

76	58	23	48	44	7	21	27	87	96
----	----	----	----	----	---	----	----	----	----

58	48	23	27	44	7	21	76	87	96
----	----	----	----	----	---	----	----	----	----

48	44	23	27	21	7	58	76	87	96
----	----	----	----	----	---	----	----	----	----

44	27	23	7	21	48	58	76	87	96
----	----	----	---	----	----	----	----	----	----

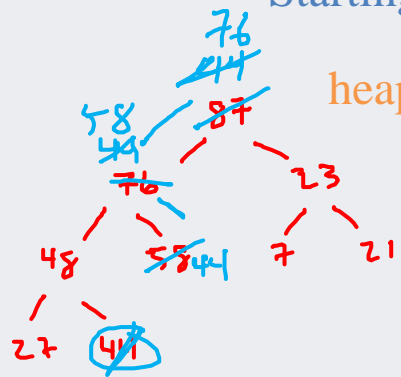
27	21	23	7	44	48	58	76	87	96
----	----	----	---	----	----	----	----	----	----

23	21	7	27	44	48	58	76	87	96
----	----	---	----	----	----	----	----	----	----

21	7	23	27	44	48	58	76	87	96
----	---	----	----	----	----	----	----	----	----

7	21	23	27	44	48	58	76	87	96
---	----	----	----	----	----	----	----	----	----

removed items



Exercise

- Given the following array representation of a heap:

value	81	76	43	63	55	39	41	7	58	25	32	9	22	37	11
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Remove three items from the top of the heap. Show the state of the array after the three items have been removed.

Readings for this lesson

- Thareja
 - Chapter 12.1, 12.5 (Binary heaps and applications)
- Next class:
 - Thareja chapters 14.9, 14.8 (Selection sort, Insertion sort)