Data Structures and Algorithms

Part 10: Binary Heap

Dr Robert Holton
Horton D4.12
d.r.w.holton@bradford.ac.uk

Overview

1. What is a Binary Heap
2. What is it used for?
3. Insertion
4. Removal
5. Heap Build
6. Heap Sort
7. Array-based Implementation

What is a Binary Heap

A Binary Heap is

1. a complete binary tree. . .
2. where the nodes contain objects with keys. . .
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

What is a Binary Heap

A Binary Heap is

1. a complete binary tree. . .
2. where the nodes contain objects with keys. . .
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

What is a Binary Heap

A Binary Heap is

1. a complete binary tree. . .
2. where the nodes contain objects with keys. . .
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

What is a Binary Heap

A Binary Heap is

1. a complete binary tree...
2. where the nodes contain objects with keys...
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

What is a Binary Heap

A Binary Heap is

1. a complete binary tree...
2. where the nodes contain objects with keys...
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

What is a Binary Heap

A Binary Heap is

1. a complete binary tree...
2. where the nodes contain objects with keys...
3. that satisfy a heap-order property

A Binary Heap is used

1. to implement a priority queue
2. for heap sort

Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order

Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right

Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right

Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right

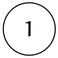
1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right

Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right

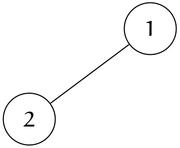
Complete Binary Tree

1. In a complete binary tree of height h:
    - all leaves are at depth h or depth $h - 1$; and
    - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
    - top to bottom
    - left to right



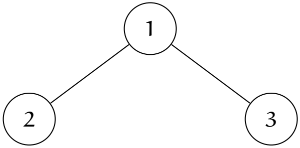
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



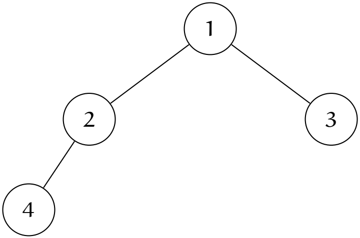
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



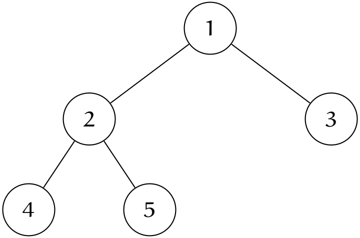
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



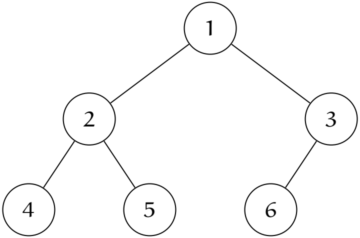
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



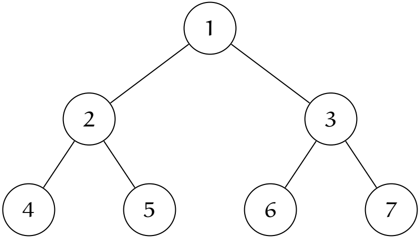
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



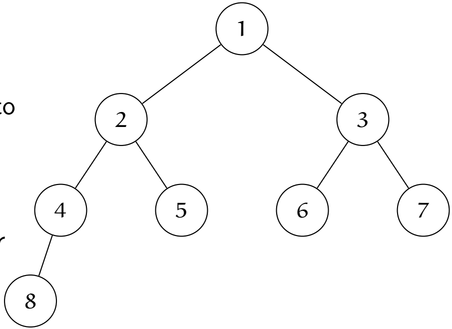
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



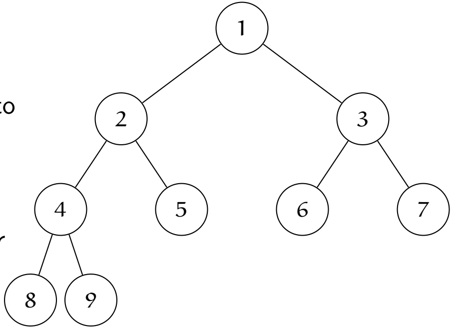
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



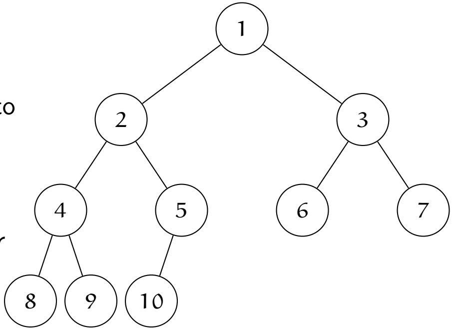
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



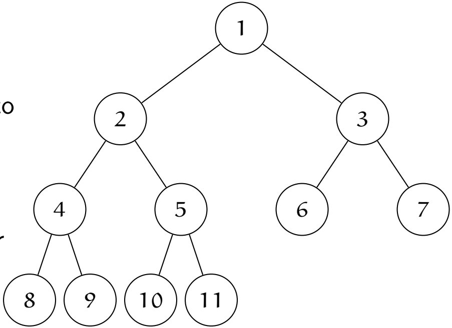
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



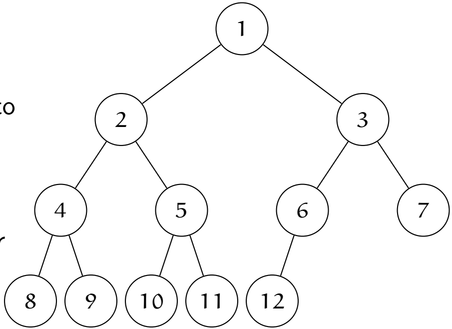
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



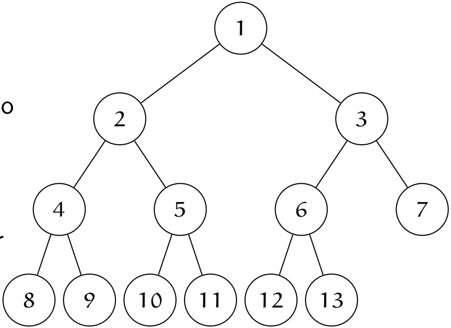
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



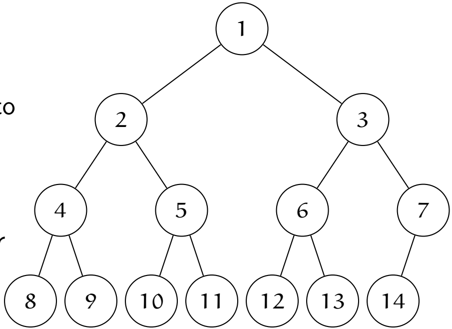
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



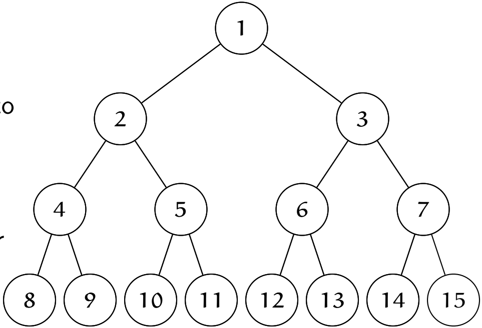
Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h - 1$; and
   - leaves at depth h are to the left of leaves at depth $h - 1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



Complete Binary Tree

1. In a complete binary tree of height h:
   - all leaves are at depth h or depth $h-1$; and
   - leaves at depth h are to the left of leaves at depth $h-1$
2. A complete binary tree is built in breadth-first order
   - top to bottom
   - left to right



min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

otherwise If you are not told whether the heap is a max-heap or a min-heap

Heap-Order Property

min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

otherwise If you are not told whether the heap is a max-heap or a min-heap

Heap-Order Property

min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

otherwise If you are not told whether the heap is a max-heap or a min-heap

- it is a min-heap

Heap-Order Property

min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

otherwise If you are not told whether the heap is a max-heap or a min-heap

- it is a min-heap

Heap-Order Property

min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

otherwise If you are not told whether the heap is a max-heap or a min-heap

- it is a min-heap

Heap-Order Property

min-heap For every node, the key is greater than or equal to the key in the node's parent

- The smallest key is at the root

max-heap For every node, the key is less than or equal to the key in the node's parent

- The largest key is at the root

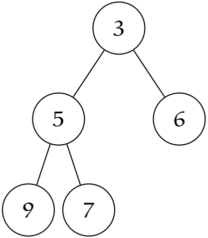
otherwise If you are not told whether the heap is a max-heap or a min-heap

- it is a min-heap

Insertion – add an object to the heap

Example: add key 2 to the heap shown:

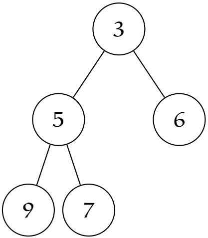
1. Find the insertion point (the next node in the complete binary tree) and create a new node (the "hole")
2. Check whether putting the new object into the "hole" would breach heap order
3. If it would, promote the "hole" by moving the object in the parent downwards and repeat from 2. Otherwise, store the object in the "hole"

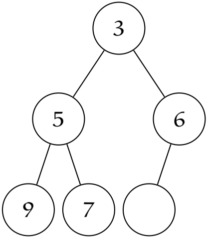


Insertion – add an object to the heap

Example: add key 2 to the heap shown:

1. Find the insertion point (the next node in the complete binary tree) and create a new node (the “hole”)
2. Check whether putting the new object into the “hole” would breach heap order
3. If it would, promote the “hole” by moving the object in the parent downwards and repeat from 2
4. Otherwise, store the object in the “hole”

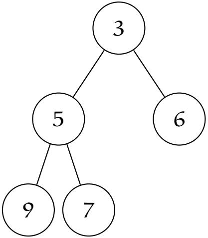


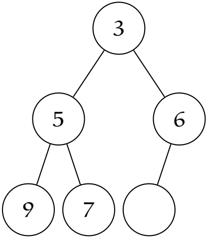


Insertion – add an object to the heap

Example: add key 2 to the heap shown:

1. Find the insertion point (the next node in the complete binary tree) and create a new node (the "hole")
2. Check whether putting the new object into the "hole" would breach heap order
   a) If it would, promote the "hole" by moving the object in the parent downwards and repeat from 2
   b) Otherwise, store the object in the "hole"





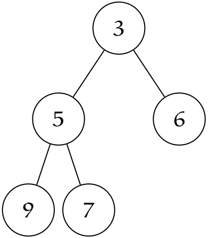
Insertion – add an object to the heap

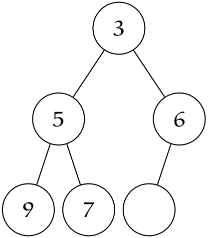
Example: add key 2 to the heap shown:

1. Find the insertion point (the next node in the complete binary tree) and create a new node (the “hole”)
2. Check whether putting the new object into the “hole” would breach heap order

   a) If it would, promote the “hole” by moving the object in the parent downwards and repeat from 2

   b) Otherwise, store the object in the “hole”

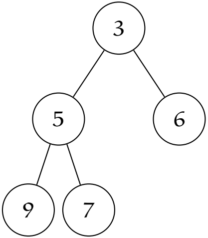


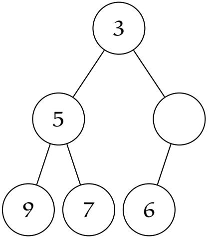


Insertion – add an object to the heap

Example: add key 2 to the heap shown:

1. Find the insertion point (the next node in the complete binary tree) and create a new node (the "hole")
2. Check whether putting the new object into the "hole" would breach heap order
   a) If it would, promote the "hole" by moving the object in the parent downwards and repeat from 2
   b) Otherwise, store the object in the "hole"

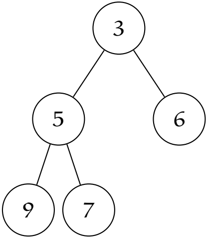


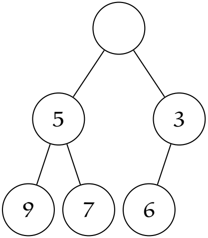


Insertion – add an object to the heap

Example: add key 2 to the heap shown:

1. Find the insertion point (the next node in the complete binary tree) and create a new node (the "hole")
2. Check whether putting the new object into the "hole" would breach heap order
   a) If it would, promote the "hole" by moving the object in the parent downwards and repeat from 2
   b) Otherwise, store the object in the "hole"

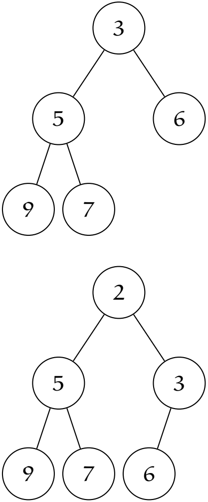




Insertion – add an object to the heap

Example: add key 2 to the heap shown:

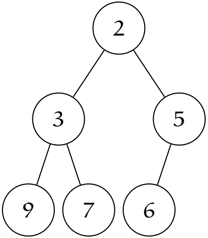
1. Find the insertion point (the next node in the complete binary tree) and create a new node (the "hole")
2. Check whether putting the new object into the "hole" would breach heap order
   a) If it would, promote the "hole" by moving the object in the parent downwards and repeat from 2
   b) Otherwise, store the object in the "hole"



Removal – always from the root of the heap

Example: remove from the heap shown:

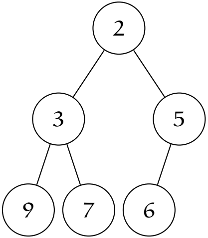
1. Remove (and output) the object at the root – leaving a "hole"
2. If putting the object from the last node of the heap into the "hole" would violate heap order, migrate the "hole" down the tree by promoting from the child with the smaller key
3. Repeat until the object from the last node may be put into the "hole"
4. Then delete the (now empty) node at the last position

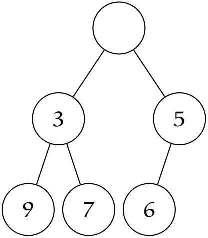


Removal – always from the root of the heap

Example: remove from the heap shown:

1. Remove (and output) the object at the root – leaving a "hole"
2. If putting the object from the last node of the heap into the "hole" would violate heap order, migrate the "hole" down the tree by promoting from the child with the smaller key
3. Repeat until the object from the last node may be put into the "hole"
4. Then delete the (now empty) node at the last position

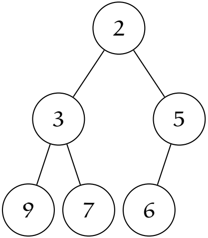


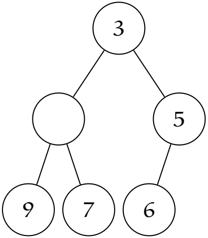


Removal – always from the root of the heap

Example: remove from the heap shown:

1. Remove (and output) the object at the root – leaving a "hole"
2. If putting the object from the last node of the heap into the "hole" would violate heap order, migrate the "hole" down the tree by promoting from the child with the smaller key
3. Repeat until the object from the last node may be put into the "hole"
4. Then delete the (now empty) node at the last position

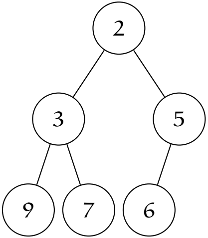


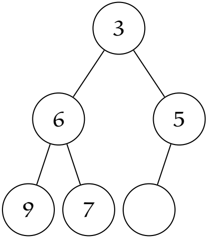


Removal – always from the root of the heap

Example: remove from the heap shown:

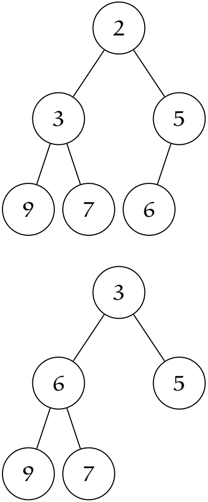
1. Remove (and output) the object at the root – leaving a "hole"
2. If putting the object from the last node of the heap into the "hole" would violate heap order, migrate the "hole" down the tree by promoting from the child with the smaller key
3. Repeat until the object from the last node may be put into the "hole"
4. Then delete the (now empty) node at the last position





Example: remove from the heap shown:

1. Remove (and output) the object at the root – leaving a "hole"
2. If putting the object from the last node of the heap into the "hole" would violate heap order, migrate the "hole" down the tree by promoting from the child with the smaller key
3. Repeat until the object from the last node may be put into the "hole"
4. Then delete the (now empty) node at the last position



Run Times for insertion and removal

In the worst cases:

Inserting one item involves the "hole" being moved from the lowest leaf to the root

In both cases the run time is (asymptotically) proportional to the height of the tree

Removing one item involves the "hole" being moved from the root down to a leaf

If the heap contains n items, the times are $\mathcal{O}(\log_2 n)$

Run Times for insertion and removal

In the worst cases:

Inserting one item involves the "hole" being moved from the lowest leaf to the root

Removing one item involves the "hole" being moved from the root down to a leaf

In both cases the run time is (asymptotically) proportional to the height of the tree

If the heap contains n items, the times are $O(\log_2 n)$

Run Times for insertion and removal

In the worst cases:

Inserting one item involves the "hole" being moved from the lowest leaf to the root

In both cases the run time is (asymptotically) proportional to the height of the tree

Removing one item involves the "hole" being moved from the root down to a leaf

If the heap contains n items, the times are $\mathcal{O}(\log_2 n)$

Run Times for insertion and removal

In the worst cases:

Inserting one item involves the "hole" being moved from the lowest leaf to the root

In both cases the run time is (asymptotically) proportional to the height of the tree

Removing one item involves the "hole" being moved from the root down to a leaf

If the heap contains n items, the times are $\mathcal{O}(\log_2 n)$

Run Times for insertion and removal

In the worst cases:

Inserting one item involves the "hole" being moved from the lowest leaf to the root

In both cases the run time is (asymptotically) proportional to the height of the tree

Removing one item involves the "hole" being moved from the root down to a leaf

If the heap contains n items, the times are $\mathcal{O}(\log_2 n)$

Heap Build

- The Heap ADT includes a method for bulk loading of many objects into the heap at one time
- The method is (usually) called `heapBuild`
- Its run time is $\mathcal{O}(n)$, where n is the number of objects in the heap (after all the new ones have been loaded)
- Adding n objects one-by-one (using insertion) would take $\mathcal{O}(n \log_2 n)$ time
- So `heapBuild` is good for heap sort but not useful for priority queue (since items are generally added one at a time to a queue)

Heap Build

- The Heap ADT includes a method for bulk loading of many objects into the heap at one time
- The method is (usually) called `heapBuild`
- Its run time is $\mathcal{O}(n)$, where n is the number of objects in the heap (after all the new ones have been loaded)
- Adding n objects one-by-one (using insertion) would take $\mathcal{O}(n \log_2 n)$ time
- So `heapBuild` is good for heap sort but not useful for priority queue (since items are generally added one at a time to a queue)

Heap Build

- The Heap ADT includes a method for bulk loading of many objects into the heap at one time
- The method is (usually) called `heapBuild`
- Its run time is $\mathcal{O}(n)$, where n is the number of objects in the heap (after all the new ones have been loaded)
- Adding n objects one-by-one (using insertion) would take $\mathcal{O}(n \log_2 n)$ time
- So `heapBuild` is good for heap sort but not useful for priority queue (since items are generally added one at a time to a queue)

Heap Build

- The Heap ADT includes a method for bulk loading of many objects into the heap at one time
- The method is (usually) called `heapBuild`
- Its run time is $\mathcal{O}(n)$, where n is the number of objects in the heap (after all the new ones have been loaded)
- Adding n objects one-by-one (using insertion) would take $\mathcal{O}(n \log_2 n)$ time
- So `heapBuild` is good for heap sort but not useful for priority queue (since items are generally added one at a time to a queue)

Heap Build

- The Heap ADT includes a method for bulk loading of many objects into the heap at one time
- The method is (usually) called `heapBuild`
- Its run time is $\mathcal{O}(n)$, where n is the number of objects in the heap (after all the new ones have been loaded)
- Adding n objects one-by-one (using insertion) would take $\mathcal{O}(n \log_2 n)$ time
- So `heapBuild` is good for heap sort but not useful for priority queue (since items are generally added one at a time to a queue)

Heap Build Algorithm

1. Load all the input objects into a complete binary tree (top to bottom, left to right)
2. For each node that is a parent of leaves, apply heap order to the sub-tree based on that node (by swapping the smaller child with its parent)
3. Repeat the process with the grandparents of leaves, and so on, up to the root
4. The maximum total number of swaps is $2^{h+1} - h - 2 \approx O(n)$ (Because there are $2^h = n$ nodes in a tree of height h)

Heap Build Algorithm

1. Load all the input objects into a complete binary tree (top to bottom, left to right)
2. For each node that is a parent of leaves, apply heap order to the sub-tree based on that node (by swapping the smaller child with its parent)
3. Repeat the process with the grandparents of leaves, and so on, up to the root
4. The maximum total number of swaps is $2^{h+1} - h - 2 \approx O(n)$ (Because there are $2^h = n$ nodes in a tree of height h)

Heap Build Algorithm

1. Load all the input objects into a complete binary tree (top to bottom, left to right)
2. For each node that is a parent of leaves, apply heap order to the sub-tree based on that node (by swapping the smaller child with its parent)
3. Repeat the process with the grandparents of leaves, and so on, up to the root
4. The maximum total number of swaps is $2^{h+1} - h - 2 \approx \mathcal{O}(n)$ (Because there are $2^h = n$ nodes in a tree of height h)

Heap Build Algorithm

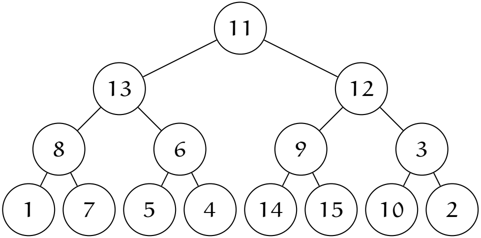
1. Load all the input objects into a complete binary tree (top to bottom, left to right)
2. For each node that is a parent of leaves, apply heap order to the sub-tree based on that node (by swapping the smaller child with its parent)
3. Repeat the process with the grandparents of leaves, and so on, up to the root
4. The maximum total number of swaps is $2^{h+1} - h - 2 \approx \mathcal{O}(n)$ (Because there are $2^h = n$ nodes in a tree of height h)

For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

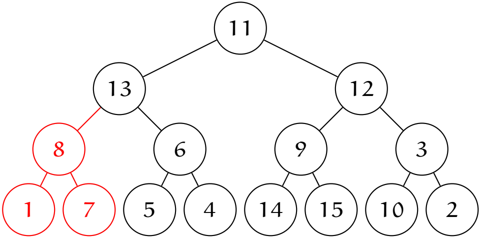
Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

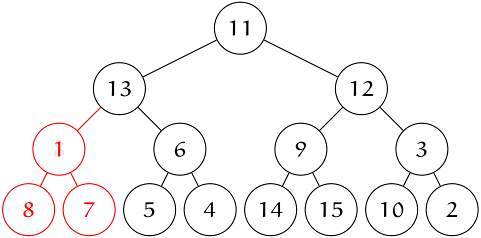


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

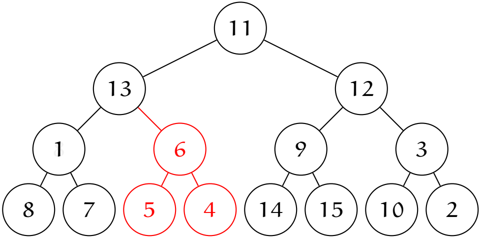
Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

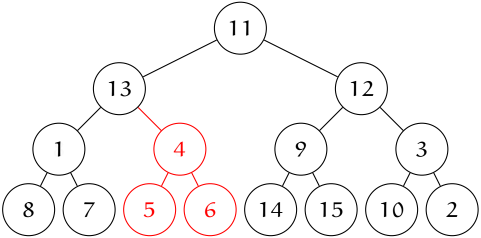


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

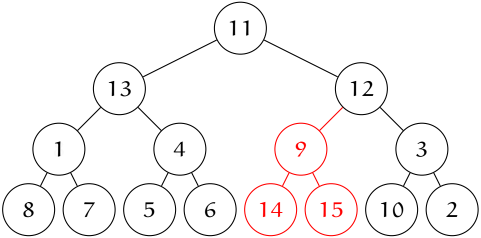


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

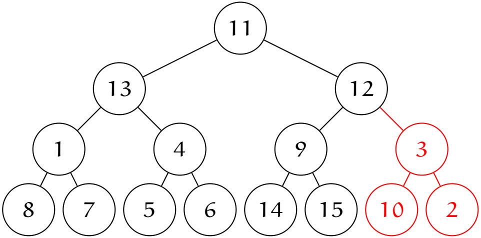


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

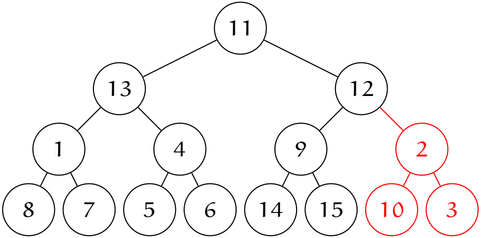
Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

Heap Build Algorithm

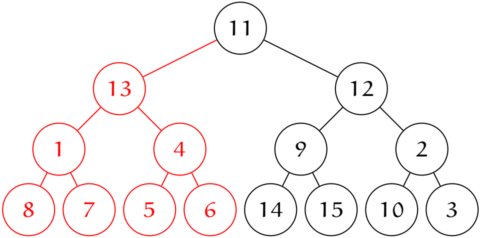


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

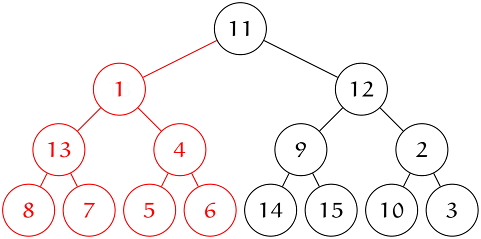


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

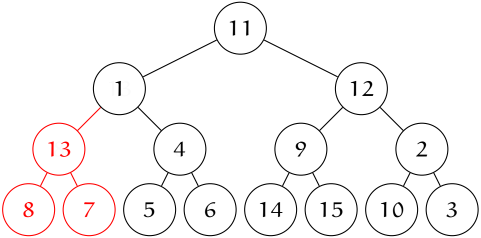


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

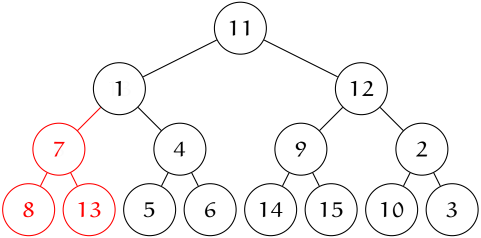


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

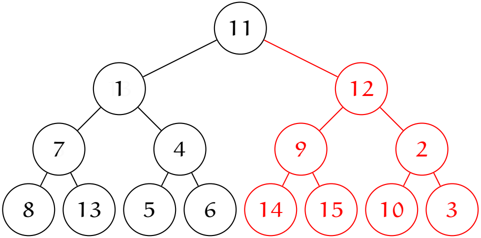


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

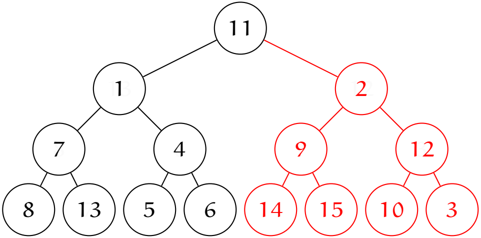
Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

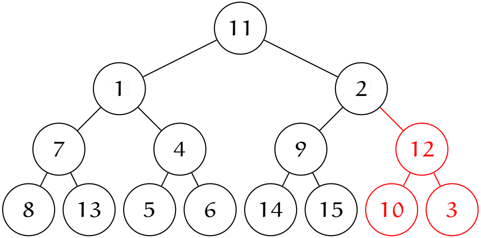
Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

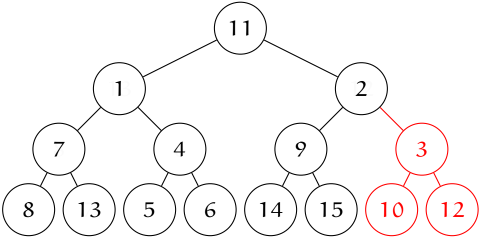


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

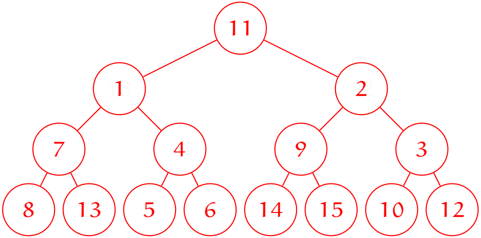


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

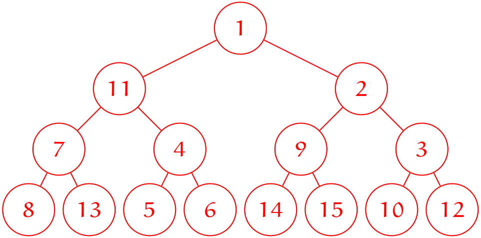
Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

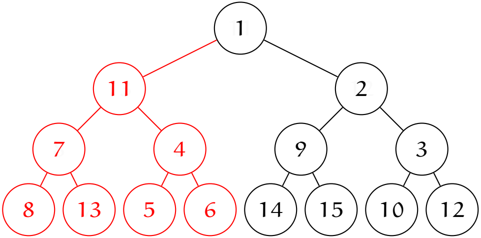
Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

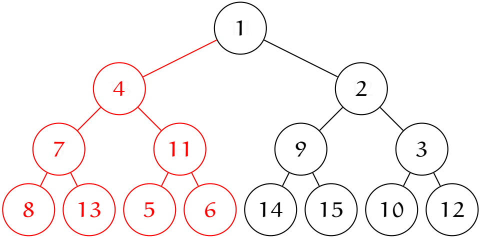


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

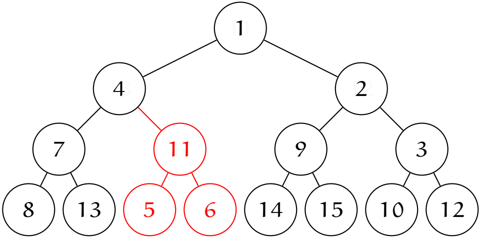


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

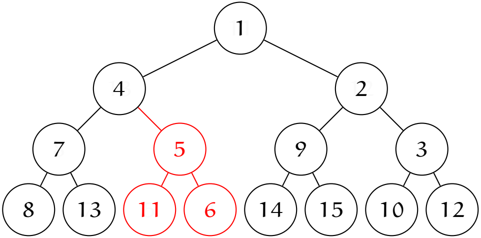


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

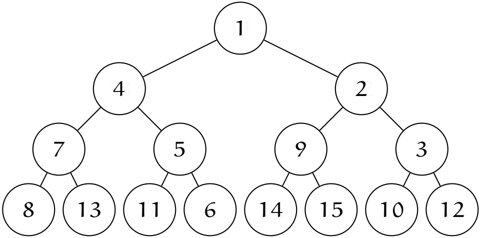


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

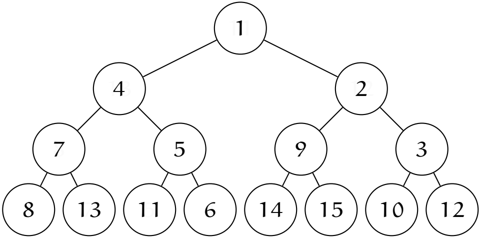


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm

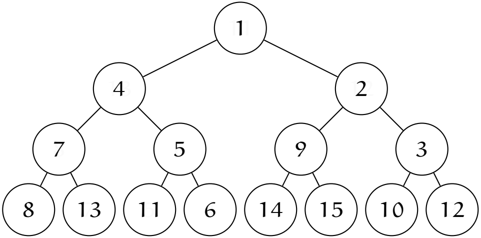


For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Build Algorithm



For example, insert the random sequence 11, 13, 12, 8, 6, 9, 3, 1, 7, 5, 4, 14, 15, 10, 2 into a heap

$\sum_{i=1}^{h} i \cdot 2^{h-i}$

Depth	Number of Nodes	Max. swaps per node
0	2^0	h
...	...	...
$h-3$	2^{h-3}	3
$h-2$	2^{h-2}	2
$h-1$	2^{h-1}	1

Heap Sort

1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

Heap Sort

1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

Heap Sort

1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

Heap Sort

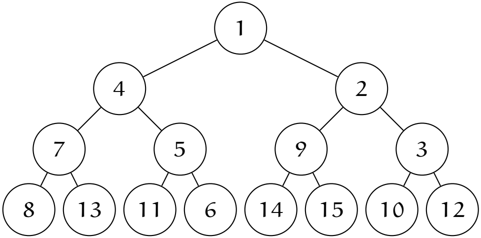
1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

Heap Sort

1. Load all input objects into heap using `heapBuild`
   - run time $\mathcal{O}(n)$
2. Output objects in order, one-by-one using `removeFirst`
   - run time $\mathcal{O}(n \log_2 n)$
3. Overall run time $\mathcal{O}(n \log_2 n)$
4. Much better than $\mathcal{O}(n^2)$ run times of insertion sort and selection sort

Heap Sort

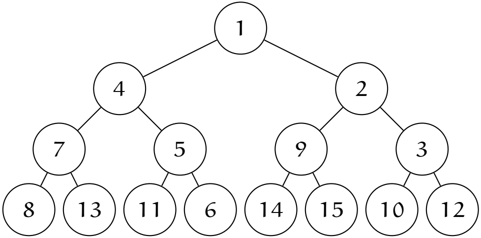
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

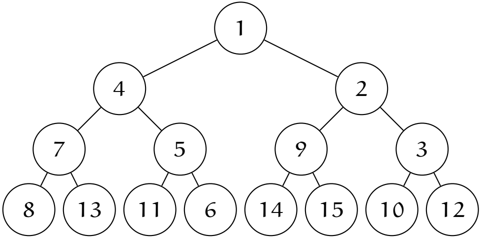
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

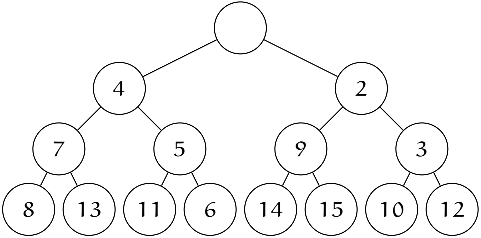
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

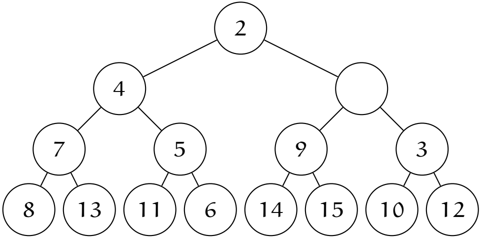
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1

Heap Sort

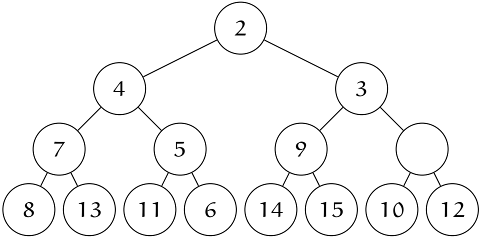
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

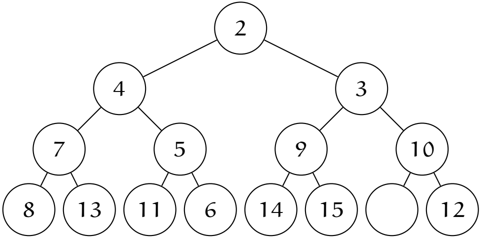
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1

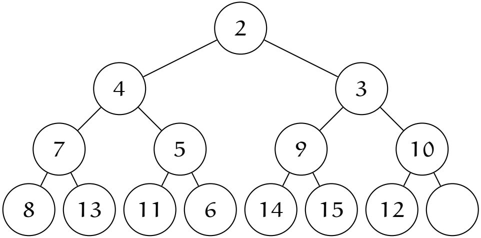
Heap Sort

- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

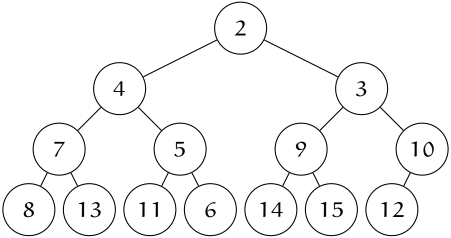
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

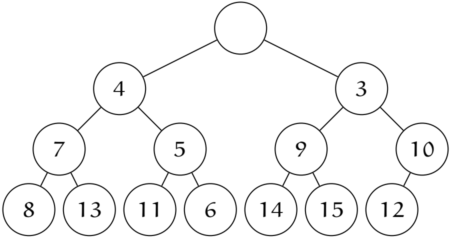
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

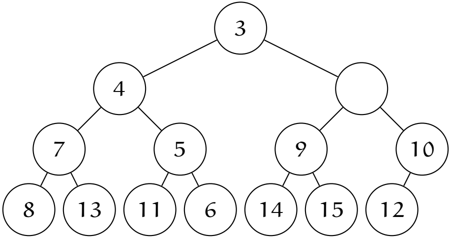
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2

Heap Sort

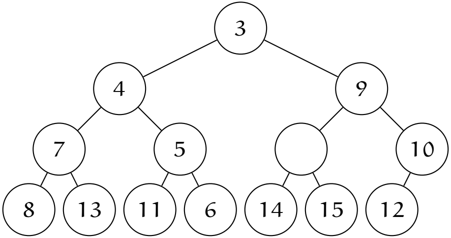
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2

Heap Sort

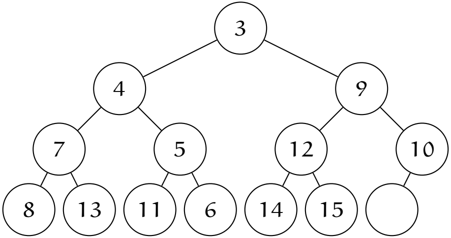
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

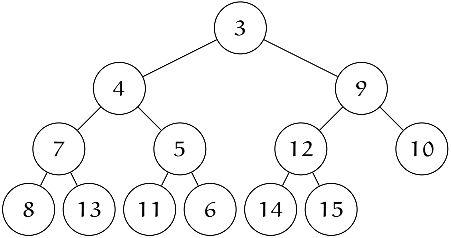
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

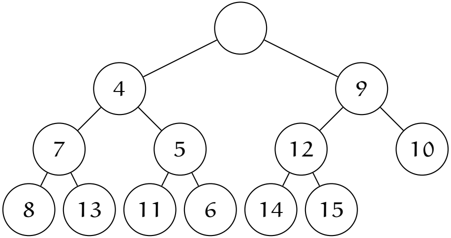
- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Heap Sort

- The random sequence from the previous example has been loaded into a heap
- Now empty the heap by removing (and outputting) the first member of the heap until it is empty
- The sequence has been sorted



1 2 3 . . .

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

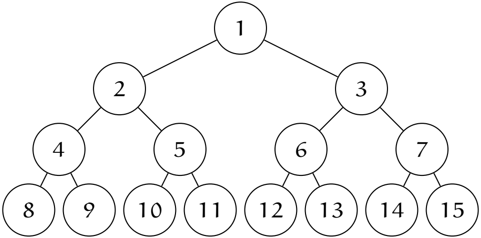
1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

Binary Heap ADT Interface

1. Constructors – may accept
   - indicator for min-heap or max-heap
   - Comparator
2. `insert` or `add`
3. `remove` or `removeFirst` or `removeMin`
4. `heapBuild`
5. `size`
6. `isEmpty`

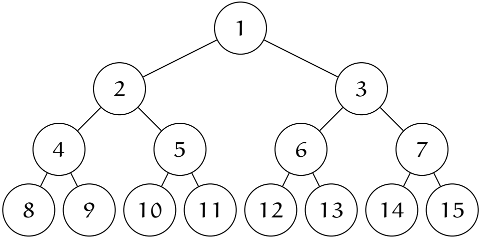
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty



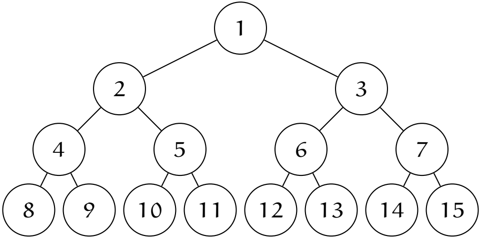
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
    - its parent is at $\lfloor k \div 2 \rfloor$
    - its left child at $2 \times k$
    - its right child at $2 \times k + 1$
4. Position index 0 is kept empty



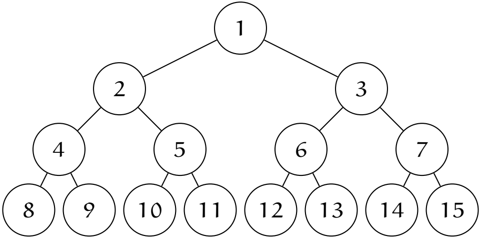
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty



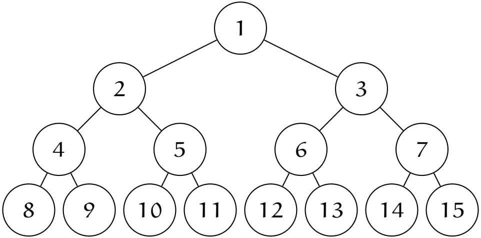
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty



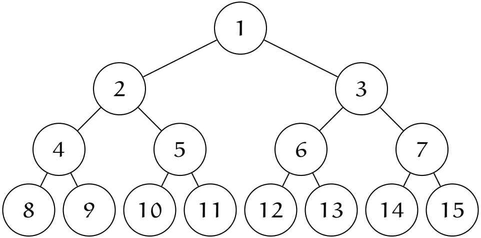
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty
   - though could be used to store the size attribute



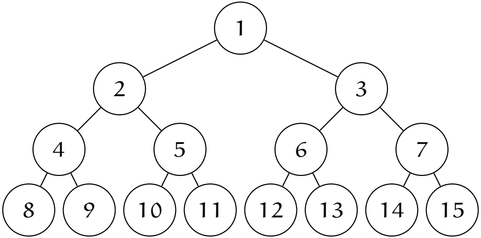
Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty
   - though could be used to store the size attribute



Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
    - its parent is at $\lfloor k \div 2 \rfloor$
    - its left child at $2 \times k$
    - its right child at $2 \times k + 1$
4. Position index 0 is kept empty
    - though could be used to store the `size` attribute



Binary Heap Implementation

1. The heap is not implemented as a tree but as an array
2. The root is at index 1
3. In general, if a node is at index k
   - its parent is at $\lfloor k \div 2 \rfloor$
   - its left child at $2 \times k$
   - its right child at $2 \times k + 1$
4. Position index 0 is kept empty
   - though could be used to store the `size` attribute