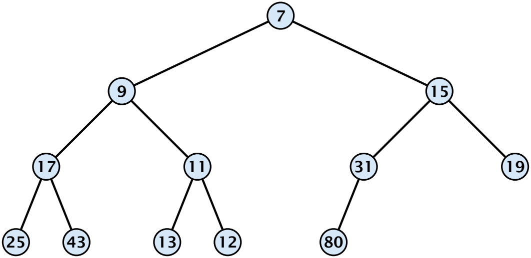
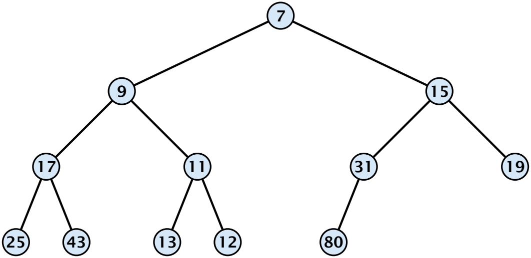
8.1 Binary Heaps



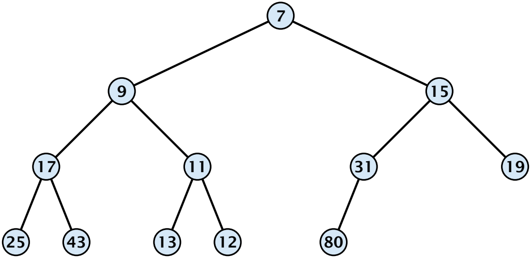
8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.



8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.
- Heap property: A node's key is not larger than the key of one of its children.



Binary Heaps

Operations:

- **minimum()**: return the root-element. Time $\mathcal{O}(1)$.
- **is-empty()**: check whether root-pointer is **null**. Time $\mathcal{O}(1)$.

Binary Heaps

Operations:

- **minimum()**: return the root-element. Time $\mathcal{O}(1)$.
- **is-empty()**: check whether root-pointer is null. Time $\mathcal{O}(1)$.

Binary Heaps

Operations:

- **minimum()**: return the root-element. Time $\mathcal{O}(1)$.
- **is-empty()**: check whether root-pointer is null. Time $\mathcal{O}(1)$.

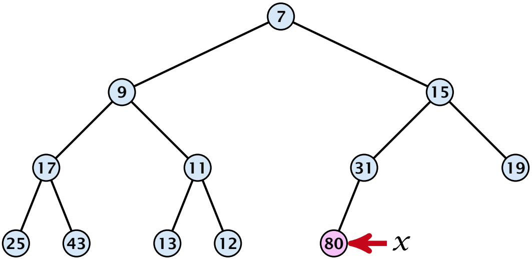
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



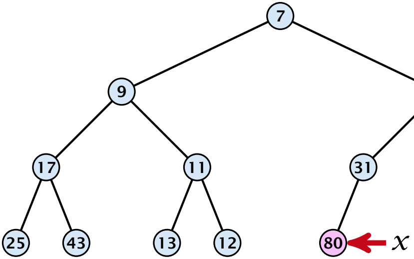
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



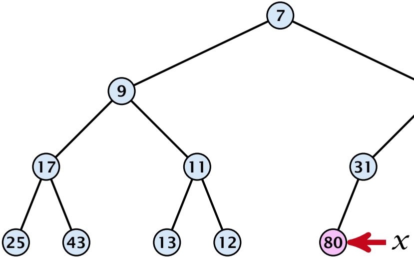
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



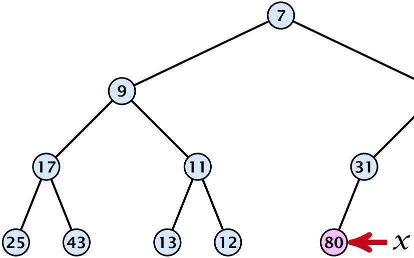
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

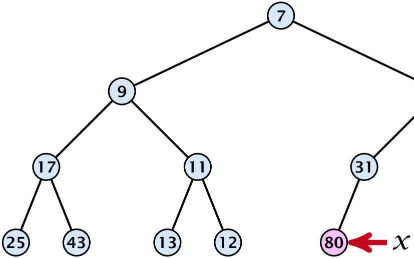
  if you hit the root on the way up, go to the rightmost element



8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.



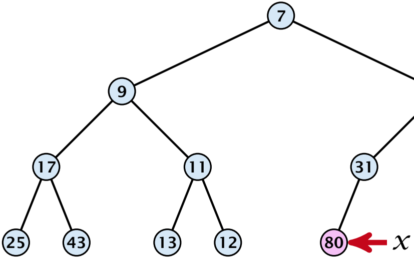
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



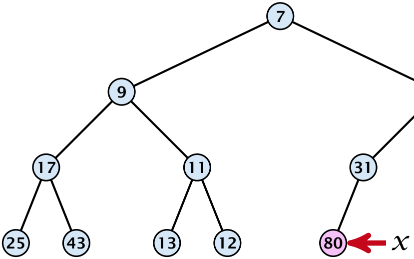
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



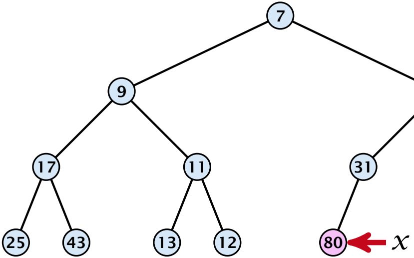
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

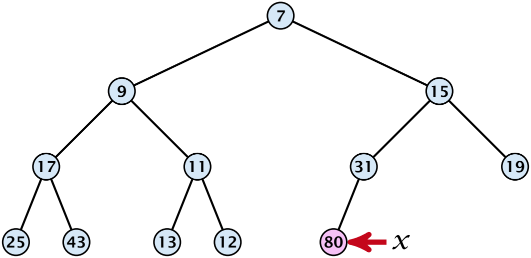
  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



Insert

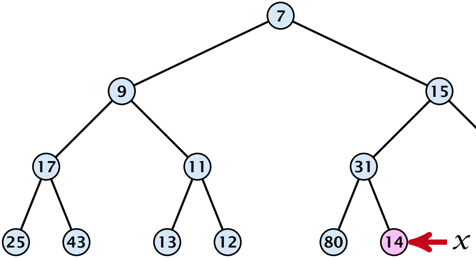
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

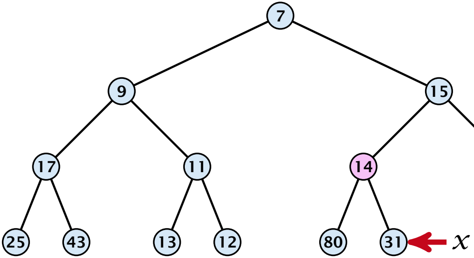
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

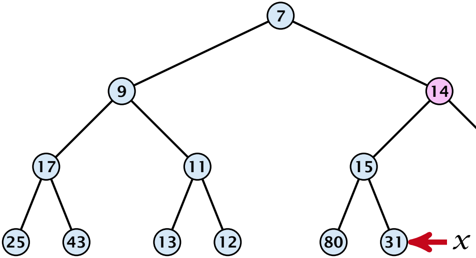
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

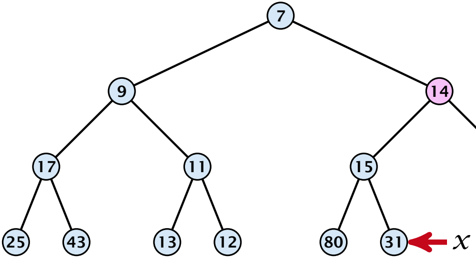
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

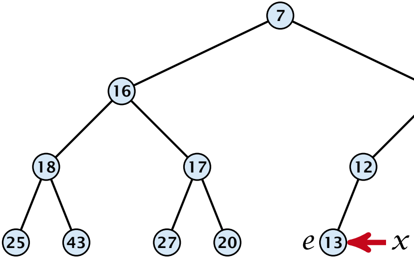
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Delete

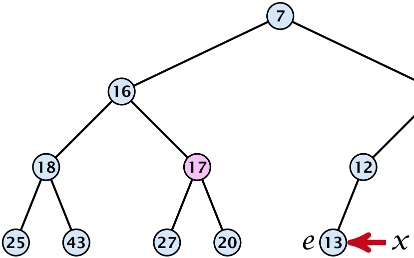
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

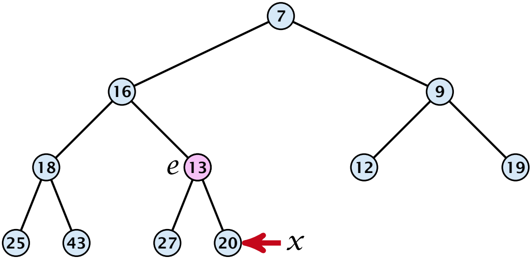
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

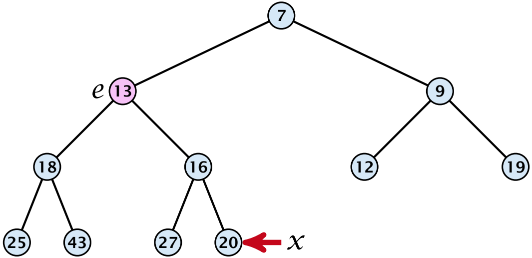
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

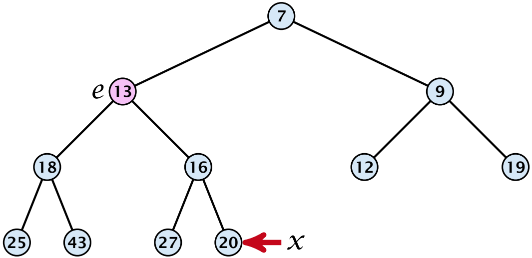
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

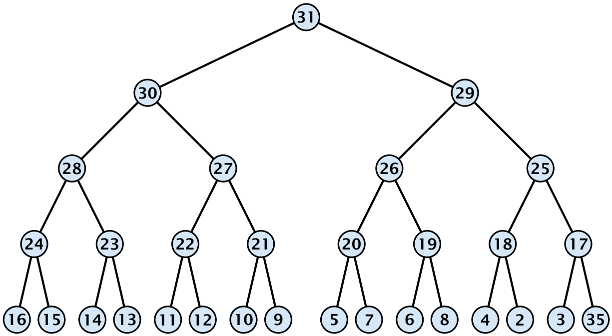
Binary Heaps

Operations:

- **minimum()**: return the root-element. Time $\mathcal{O}(1)$.
- **is-empty()**: check whether root-pointer is null. Time $\mathcal{O}(1)$.
- **insert(k)**: insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- **delete(h)**: swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.

Build Heap

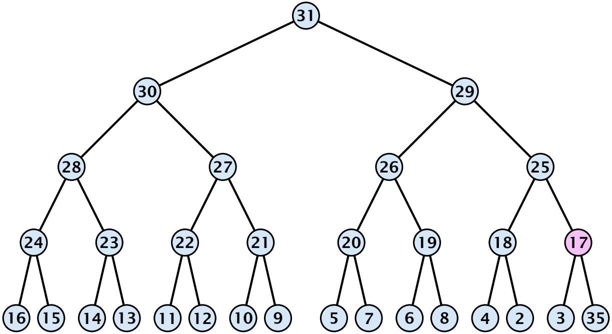
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

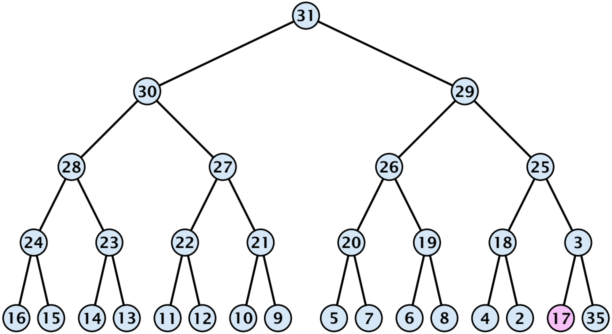
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

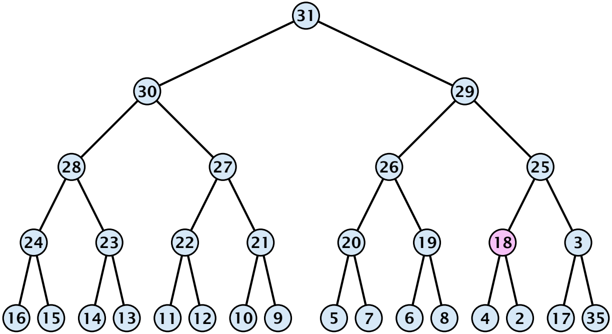
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

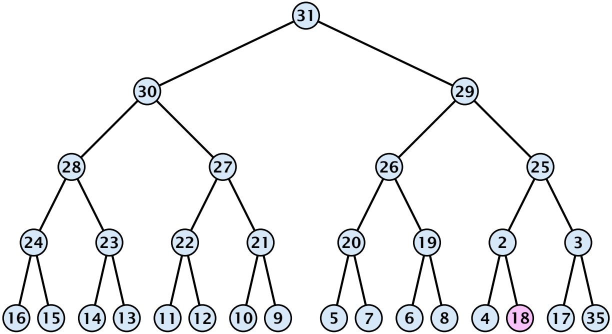
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

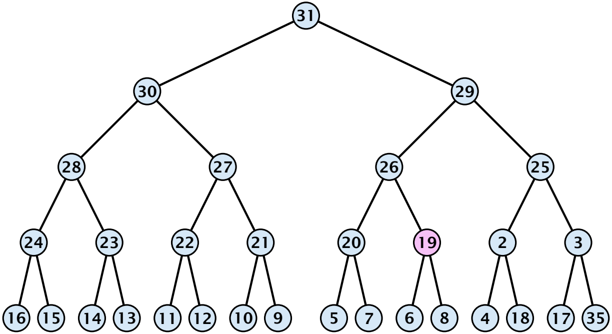
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

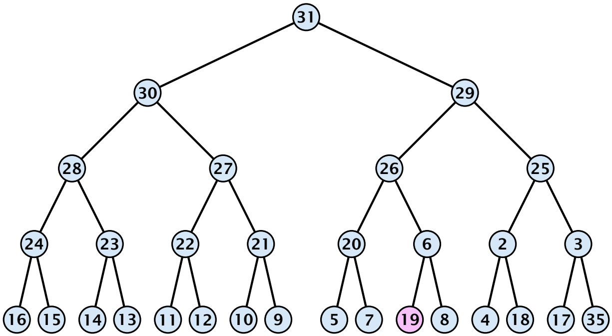
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

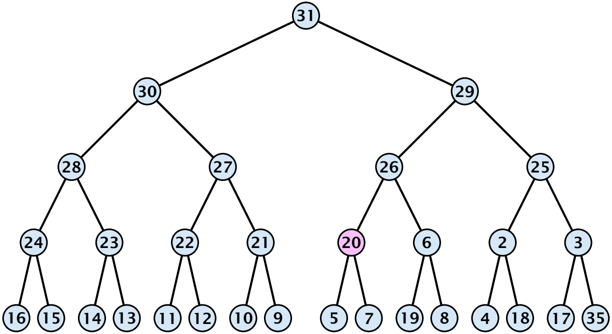
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

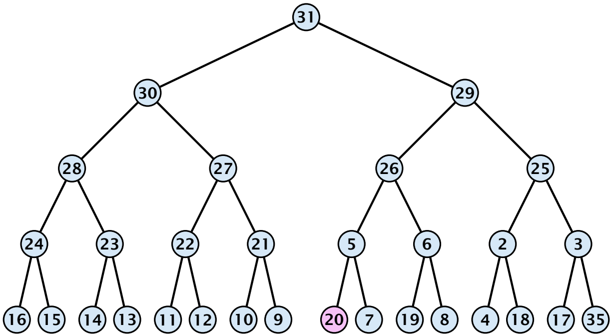
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

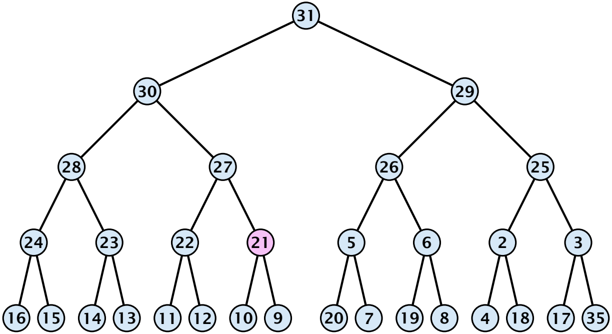
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

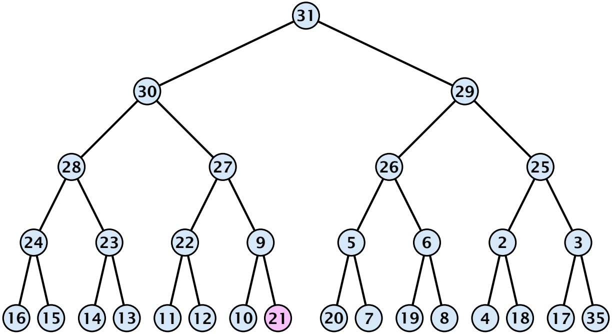
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

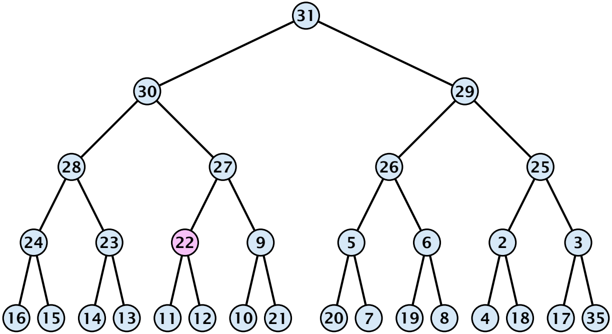
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

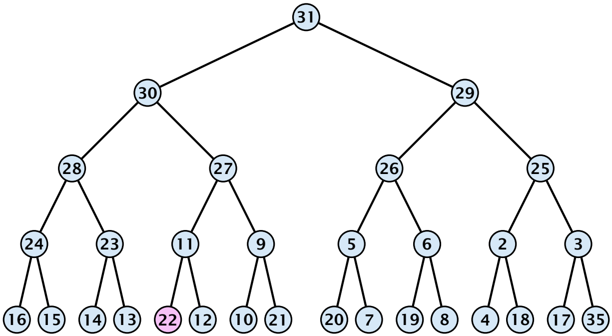
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

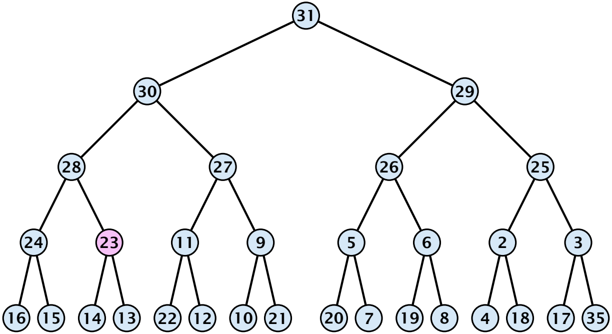
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

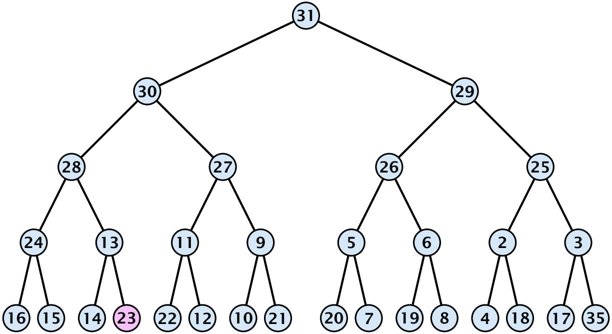
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

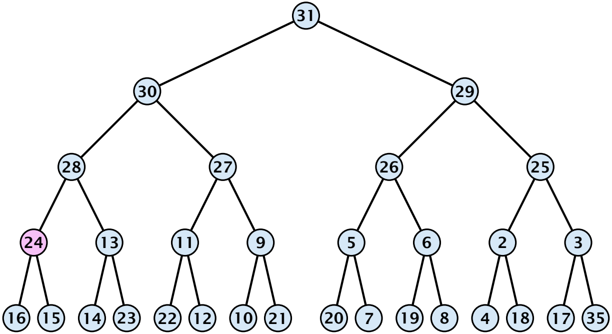
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

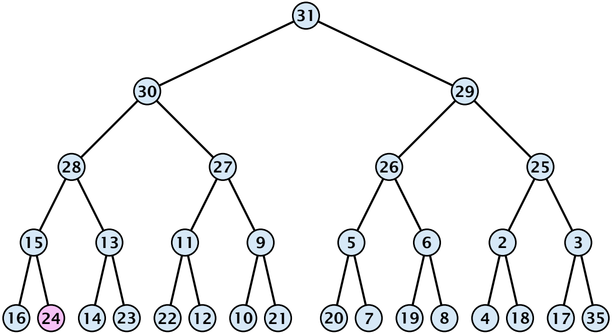
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

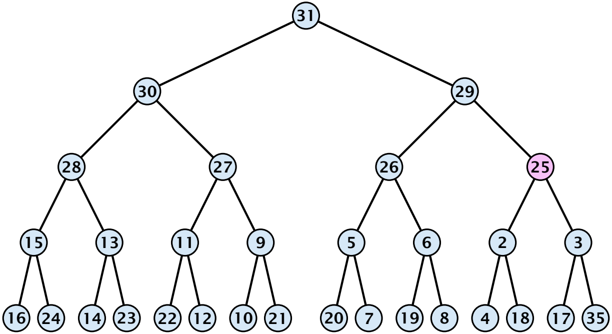
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

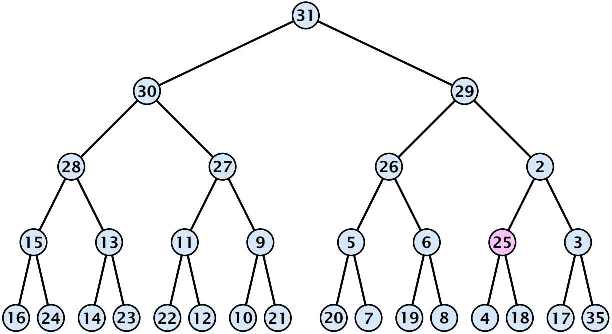
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

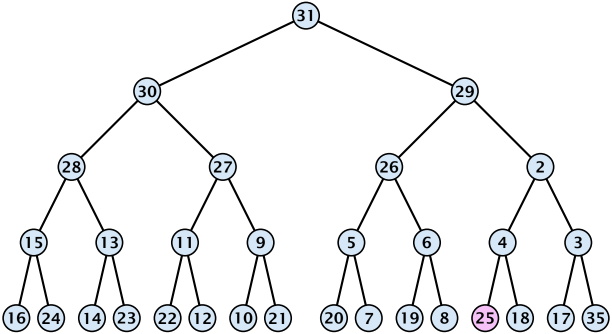
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

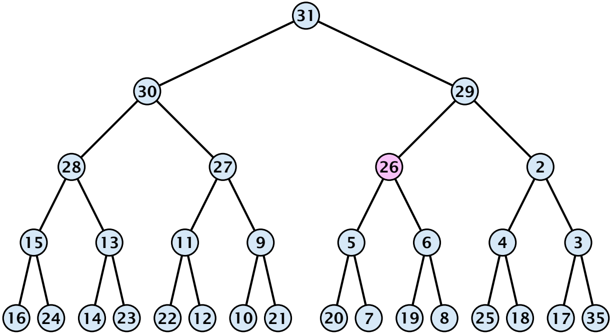
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

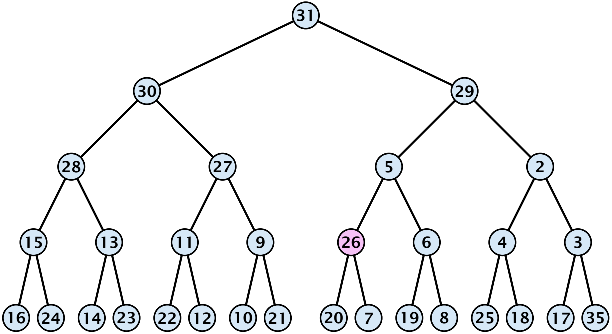
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

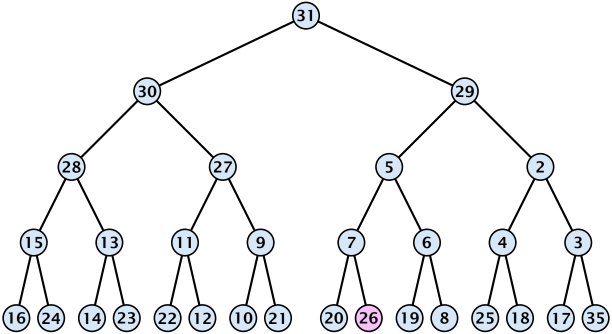
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

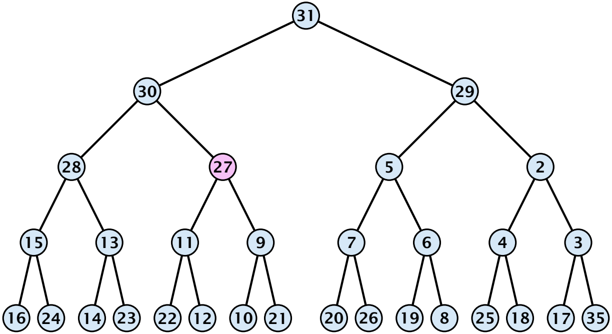
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

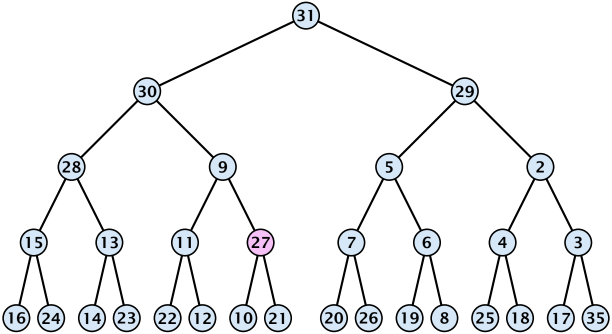
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

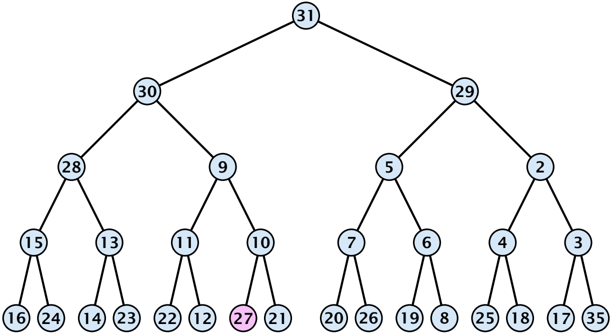
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

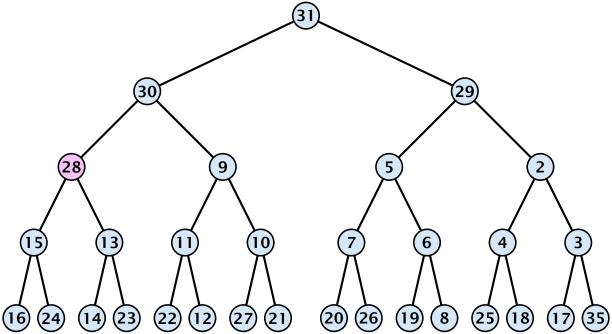
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

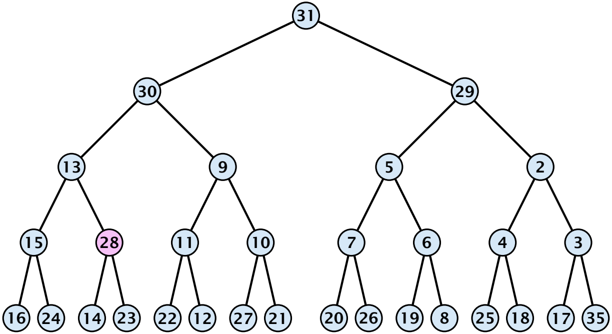
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

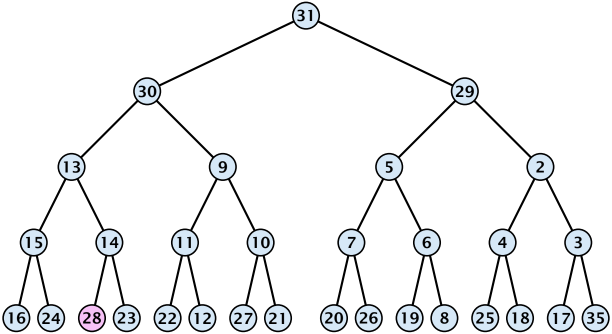
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

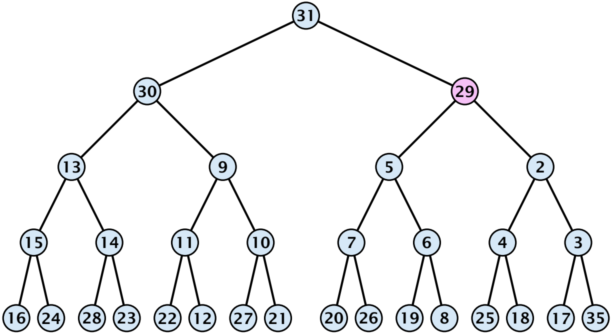
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

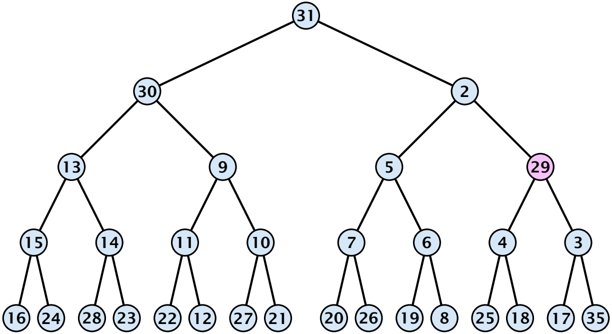
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

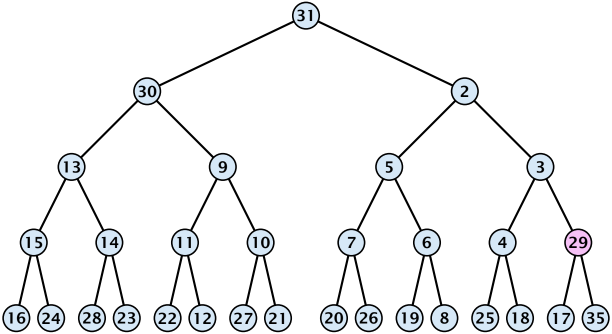
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

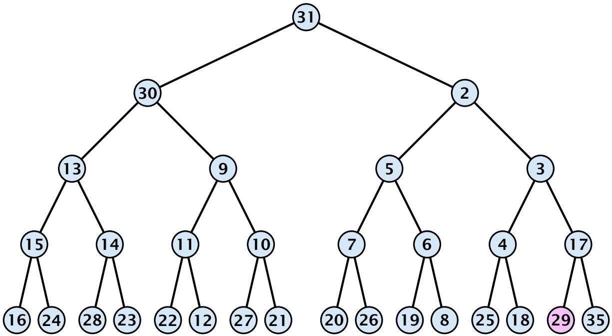
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

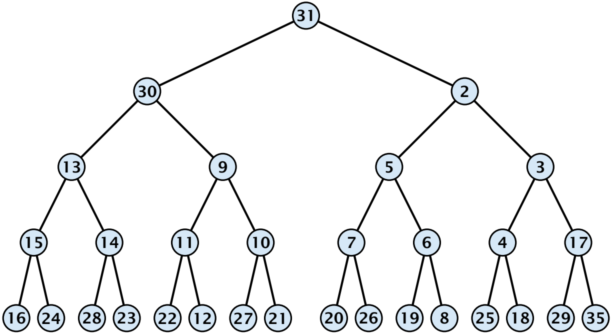
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

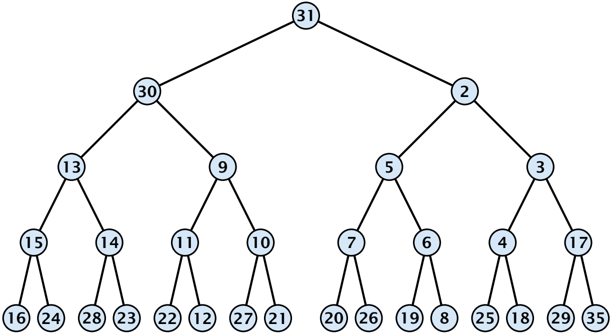
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Binary Heaps

Operations:

- $\textbf{minimum()}$: Return the root-element. Time $\mathcal{O}(1)$.
- $\textbf{is-empty()}$: Check whether root-pointer is null. Time $\mathcal{O}(1)$.
- $\textbf{insert}(\boldsymbol{k})$: Insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- $\textbf{delete}(\boldsymbol{h})$: Swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.
- $\textbf{build}(\boldsymbol{x_1, \dots, x_n})$: Insert elements arbitrarily; then do sift-down operations starting with the lowest layer in the tree. Time $\mathcal{O}(n)$.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\ldots,n-1]$ be an array

- The parent of i-th element is at position $\lfloor\frac{i-1}{2}\rfloor$.
- The left child of i-th element is at position $2i+1$.
- The right child of i-th element is at position $2i+2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0, \ldots, n-1]$ be an array

- The parent of i-th element is at position $\lfloor \frac{i-1}{2} \rfloor$.
- The left child of i-th element is at position $2i + 1$.
- The right child of i-th element is at position $2i + 2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0, \ldots, n-1]$ be an array

- The parent of i-th element is at position $\lfloor \frac{i-1}{2} \rfloor$.
- The left child of i-th element is at position $2i + 1$.
- The right child of i-th element is at position $2i + 2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0, \dots, n-1]$ be an array

- The parent of i-th element is at position $\lfloor \frac{i-1}{2} \rfloor$.
- The left child of i-th element is at position $2i + 1$.
- The right child of i-th element is at position $2i + 2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.