Recap – Basic Data Structures

Fundamental algorithms & data structures

Stacks, queues, priority queues

Complete binary trees

Heaps and heap operations

Java class for (integer) heaps

Heapsort

The stack abstract data type

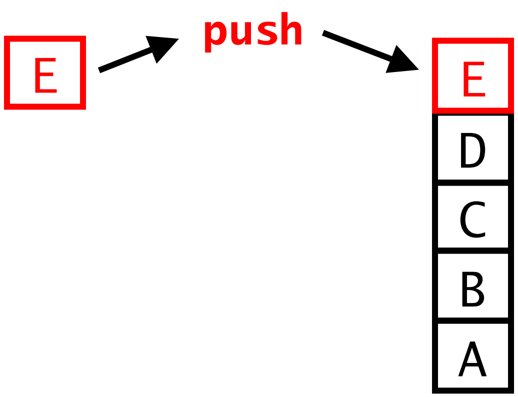
Basic operations are

- **`create`** (create an empty stack)
- **`isEmpty`** (check if stack is empty)
- **`push`** (insert a new item on the top of the stack)
- **`pop`** (delete and return the item on the top of the stack)

The stack abstract data type

Basic operations are

- **`create`** (create an empty stack)
- **`isEmpty`** (check if stack is empty)
- **push** (insert a new item on the top of the stack)
- **pop** (delete and return the item on the top of the stack)

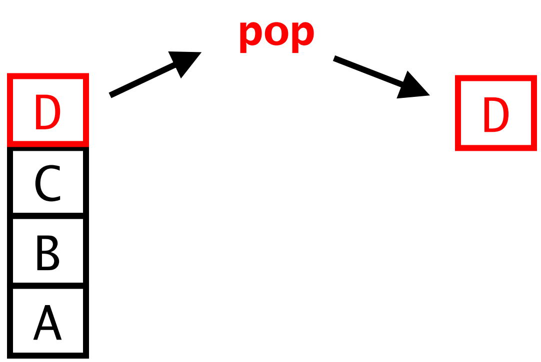


new item inserted on the top of the stack

The stack abstract data type

Basic operations are

- **`create`** (create an empty stack)
- **`isEmpty`** (check if stack is empty)
- **push** (insert a new item on the top of the stack)
- **pop** (delete and return the item on the top of the stack)



items removed from the top of the stack

The stack abstract data type

Basic operations are

- **`create`** (create an empty stack)
- **`isEmpty`** (check if stack is empty)
- **push** (insert a new item on the top of the stack)
- **pop** (delete and return the item on the top of the stack)

Order of removal of elements: last in first out

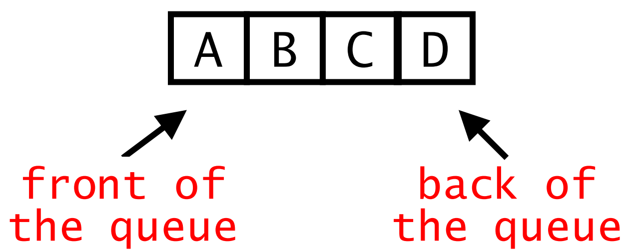
Representations of a stack

- as an array
  - the bottom of the stack is “anchored” to one end of the array
  - all operations are O(1)
- as a linked list
  - again all operations are O(1)

The queue abstract data type

Basic operations are

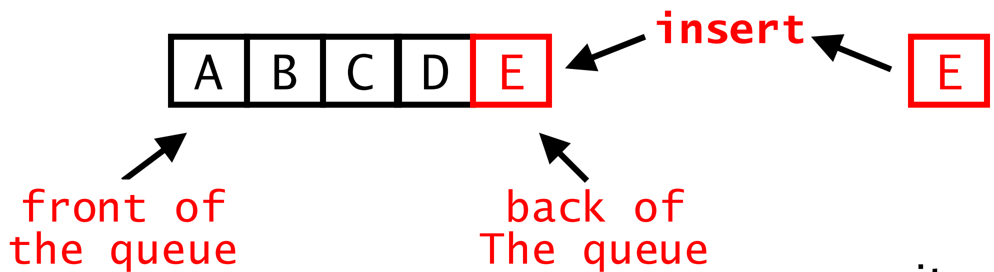
- **create** (create an empty queue)
- **isEmpty** (check if queue is empty)
- **insert** (insert a new item at the back of the queue)
- **delete** (delete and return the item at the front of the queue)



The queue abstract data type

Basic operations are

- **`create`** (create an empty queue)
- **`isEmpty`** (check if queue is empty)
- **`insert`** (insert a new item at the back of the queue)
- **`delete`** (delete and return the item at the front of the queue)

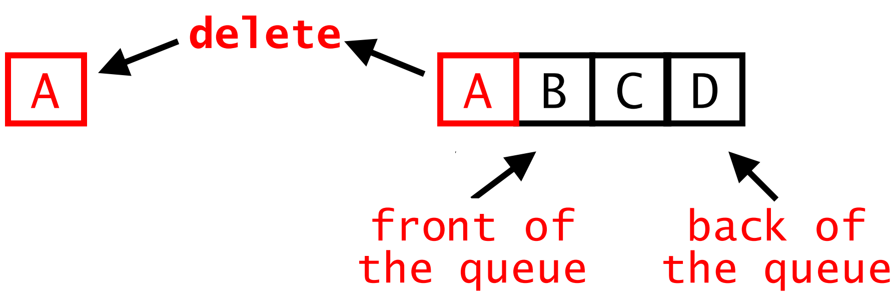


items added to the back of the queue

The queue abstract data type

Basic operations are

- **create** (create an empty queue)
- **isEmpty** (check if queue is empty)
- **insert** (insert a new item at the back of the queue)
- **delete** (delete and return the item at the front of the queue)



items removed from the front of the queue

The queue abstract data type

Basic operations are

- `create` (create an empty queue)
- `isEmpty` (check if queue is empty)
- `insert` (insert a new item at the back of the queue)
- `delete` (delete and return the item at the front of the queue)

Order of removal of elements: first in first out

Representations of a queue

- as an array
  - all operations are O(1) but care is needed (see tutorial sheet 1)
  - the queue must be “wrapped around”, treating the array as circular
- as a linked list
  - again all operations are O(1)

The priority queue abstract data type

Basic operations are

- **`create`** (create an empty queue)
- **`isEmpty`** (check if queue is empty)
- **`insert`** (insert a new item at the back of the queue)
- **`delete`** (delete and return the item at the front of the queue)

Order of removal of elements: highest priority first

Representations of a priority queue

- as an unordered list (**`insert`** is `O(1)` while **`delete`** is `O(n)`)
- as an ordered list (**`insert`** is `O(n)` while **`delete`** is `O(1)`)
- as a heap (**`insert`** and **`delete`** are `O(log n)`)
- in all cases **`create`** and **`isEmpty`** are `O(1)`

Fundamental algorithms & data structures

Stacks, queues, priority queues

Complete binary trees

Heaps and heap operations

Java class for (integer) heaps

Heapsort

Complete binary trees

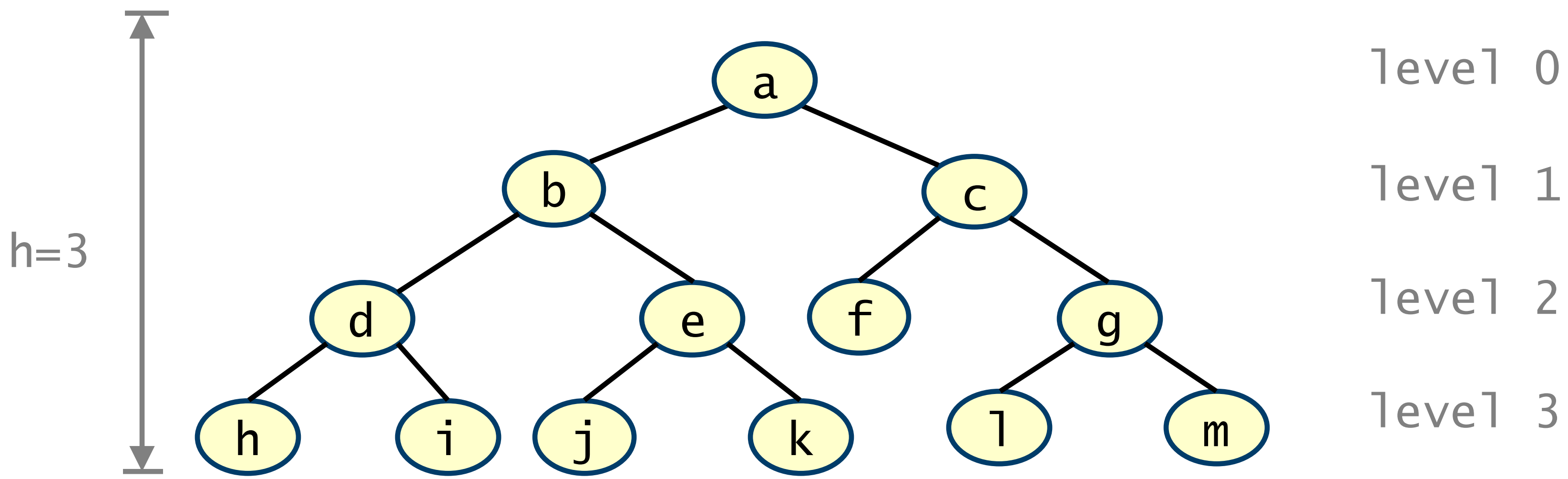
A complete binary tree with **n** nodes has

- the minimum possible height
  - the height of a node is longest path from the node to a leaf
  - the height of the tree is the height of its root node
- the maximum possible number of nodes at each level except the last
  - having minimum height actually follows from this requirement
- the nodes on the last level are as far to the left as possible

Complete binary trees

A complete binary tree with **n** nodes has

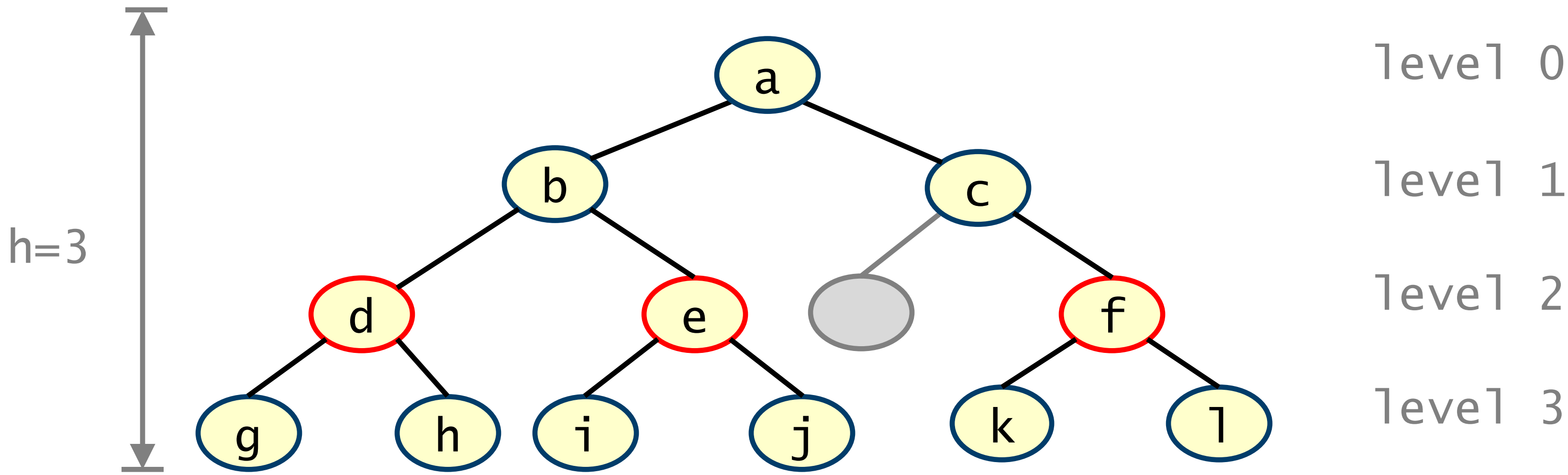
- the minimum possible height
- the maximum possible number of nodes at each level except the last
- the nodes on the last level are as far to the left as possible
  - a binary tree of height h can contain at most $2^{h+1}-1$ nodes
  - therefore the height of a complete binary tree with n nodes is the smallest h such that $n \le 2^{h+1}-1$, i.e. `h = ceil(log`$_2$`(n+1)-1)`



Complete binary trees

A complete binary tree with **n** nodes has

- the minimum possible height
- the maximum possible number of nodes at each level except the last
- the nodes on the last level are as far to the left as possible
  - i.e. for `i=0,…,h-2`, level `i` has 2^i nodes

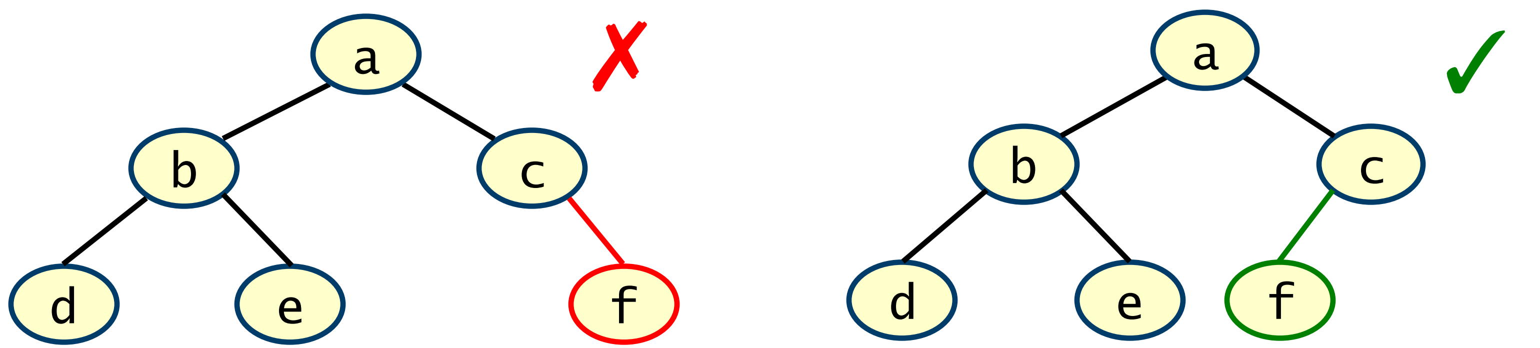


only $3<2^2=4$ nodes on level 2: not a complete binary tree

Complete binary trees

A complete binary tree with n nodes has

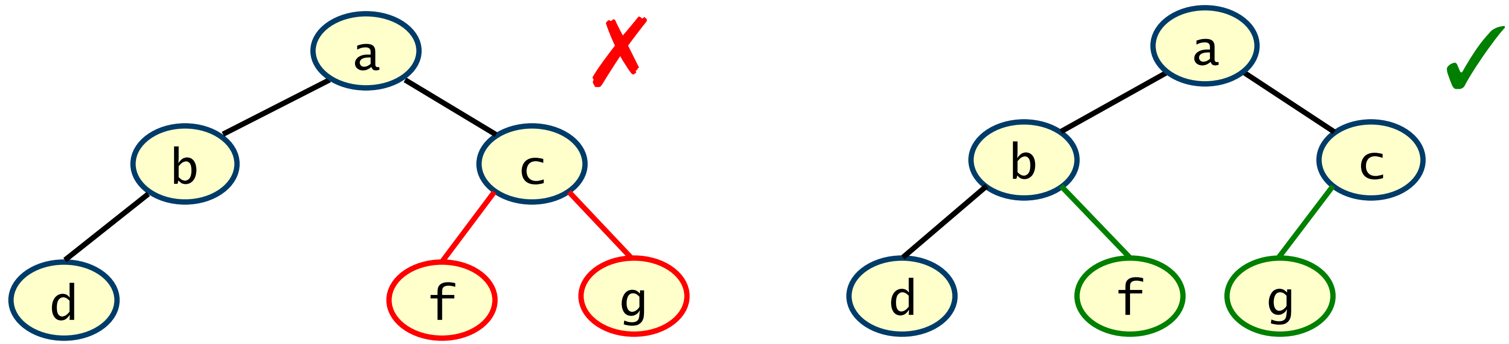
- the minimum possible height
- the maximum possible number of nodes at each level except the last
- the nodes on the last level are as far to the left as possible



Complete binary trees

A complete binary tree with n nodes has

- the minimum possible height
- the maximum possible number of nodes at each level i<k
- the nodes on the last level are as far to the left as possible



Complete binary trees – Properties

Let **T** be a complete binary tree of height **h** with **n** nodes

T has at most $2^{h+1}-1$ nodes

T height is `ceil(log`$_2$`(n+1) - 1)`

If **T** is proper (full), the number of leaf nodes is `ceil(n/2)`

If **T** is proper (full), the number of branch nodes is `floor(n/2)`

Fundamental algorithms & data structures

Stacks, queues, priority queues

Complete binary trees

Heaps and heap operations

Java class for (integer) heaps

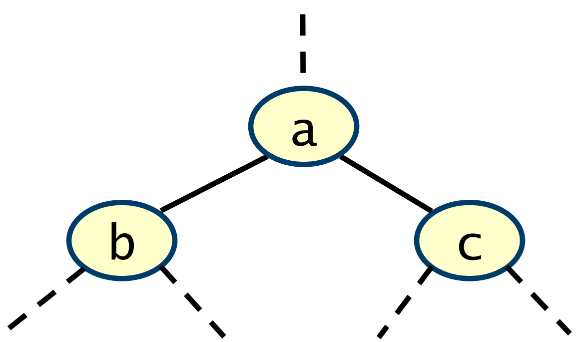
Heapsort

Heaps

A (binary) heap is a complete binary tree with an item stored at each node and each item has a value (or priority)

Heap property: for every node, the value of its item is greater than or equal to (≥) the value of all items in descendent nodes

- therefore the largest item is stored at the root



heap property: $a \geq b$ and $a \geq c$

Heaps

A (binary) **heap** is a **complete binary tree** with an **item** stored at each node and each item has a **value** (or priority)

Heap property: for every **node**, the value of its item is **greater than or equal** to ($\geq$) the value of all items in descendent nodes

- therefore the largest item is stored at the root

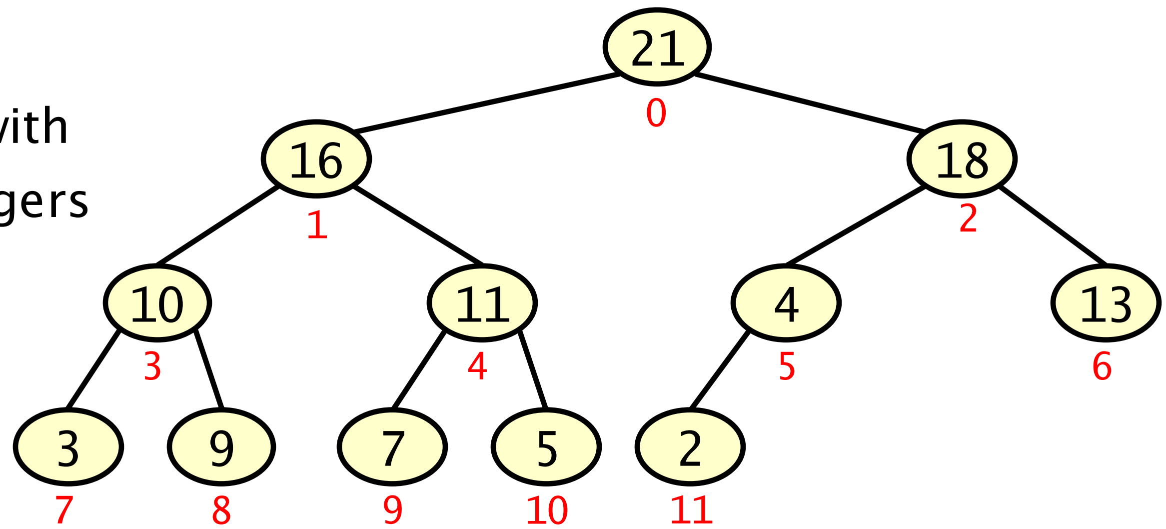
A **min-heap** is similar except that the value of each item is less than or equal to the value of all items in descendent nodes

- hence the smallest item is stored at the root in this case

Heaps – Example

A heap with 12 items

- items: integer values with usual ordering on integers



There is a natural and useful correspondence between the nodes and positions `0,…,n-1` of an array

- children of node `i` (if they exist) are nodes `2i+1` and `2i+2`
- conversely parent of node `i` is the node **`floor`**`((i-1)/2)`

Heaps – Operations

Fundamental heap operation

- **`insert`** a new item
- **`build`** a heap containing a given set of items
- **`delete`** the item with largest value (i.e. the item contained in the root)

Auxiliary operation

- **`impose`** the heap property on a particular node
  - assuming that all its descendent nodes have the heap property

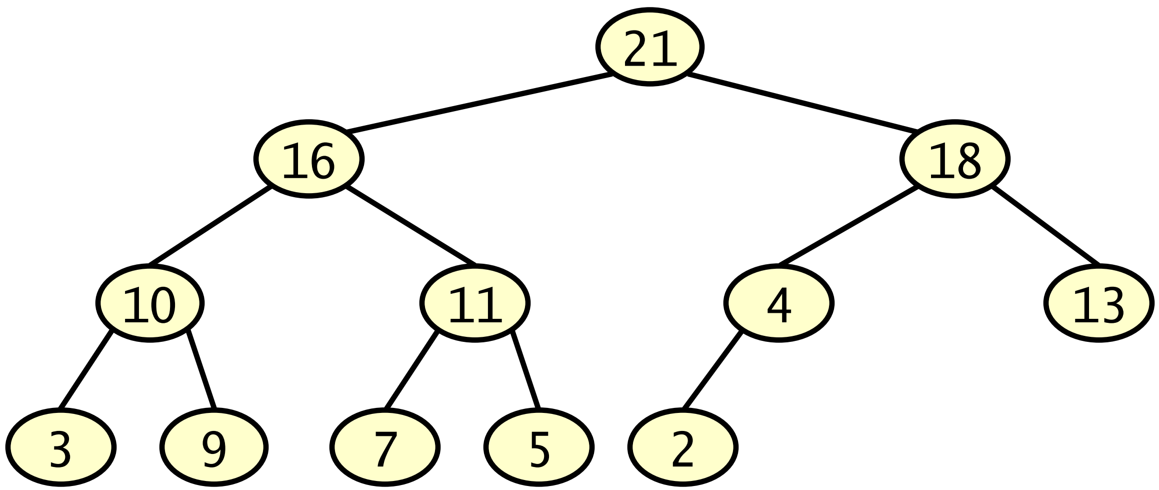
Complexity of heap operations

- a heap with `n` nodes has height `O(log n)`
- any algorithm involving `O(1)` steps at each level has complexity `O(log n)`
  - holds for **`insert`**, **`impose`**, and **`delete`** operations
- **`build`** has an easy `O(n log n)` version (just repeated insertions) and a clever `O(n)` alternative which we will introduce

Heaps – Insertion algorithm

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
    swap new_value with parent_value;
```

For example we will insert **20** into the following heap

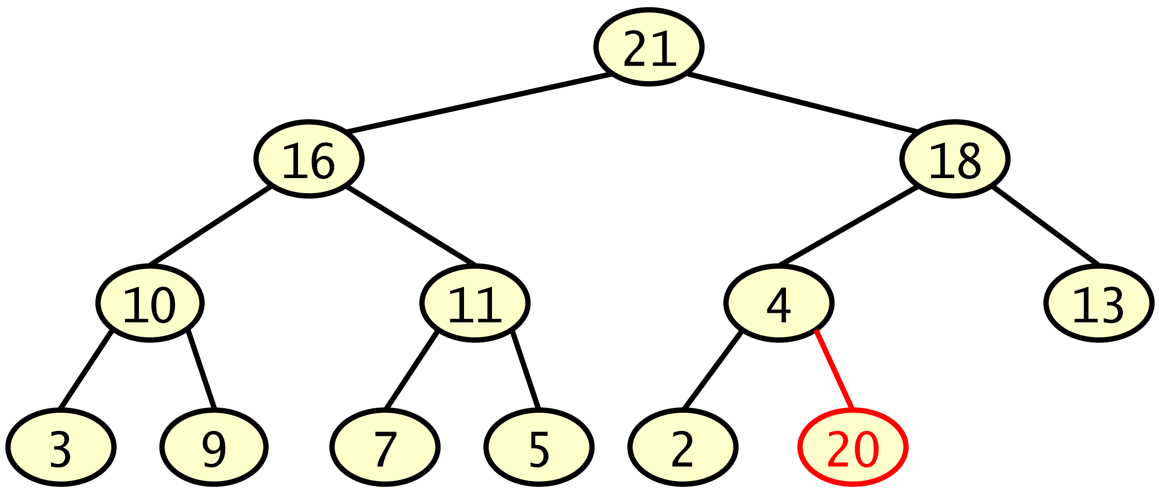


Heaps – Insertion algorithm

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
  swap new_value with parent_value;
```

For example we will insert 20 into the following heap

– insert the item 20 into a new leaf node

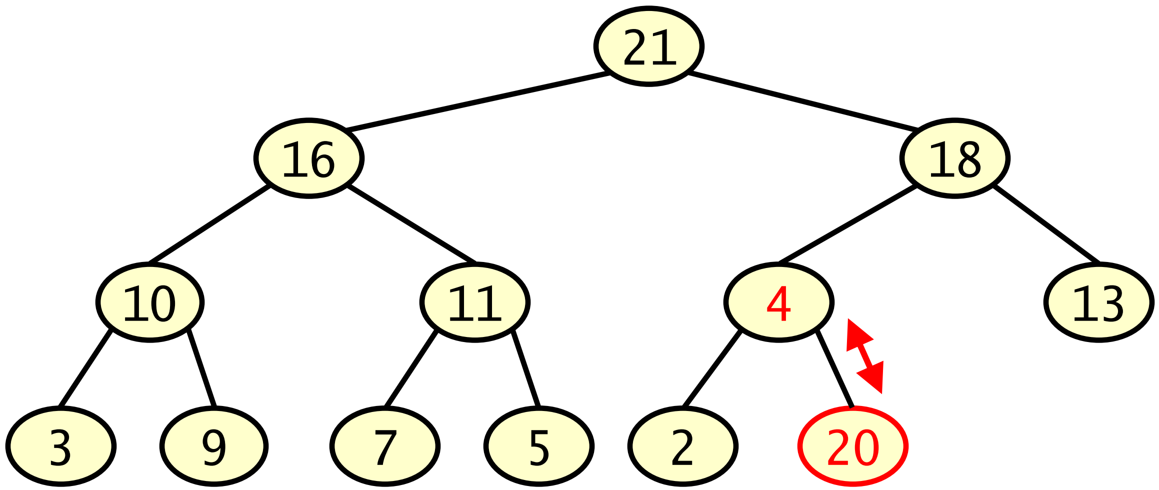


Heaps – Insertion algorithm

```
insert item in new leaf node;
 while (new_value not in root && new_value > parent_value)
   swap new_value with parent_value;
```

For example we will insert 20 into the following heap

- new value greater than parent so swap

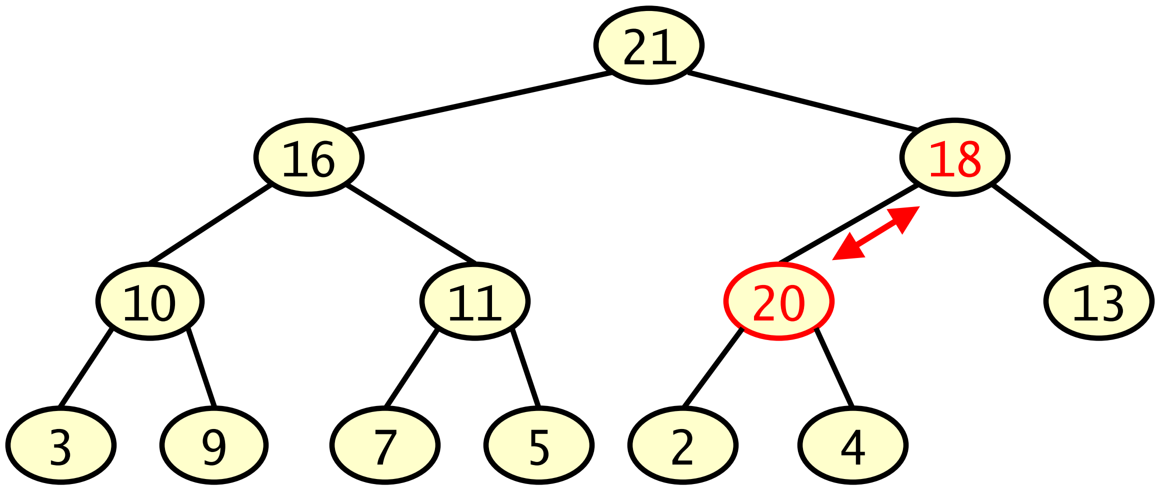


Heaps – Insertion algorithm

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
   swap new_value with parent_value;
```

For example we will insert 20 into the following heap

- again new value greater than parent so swap

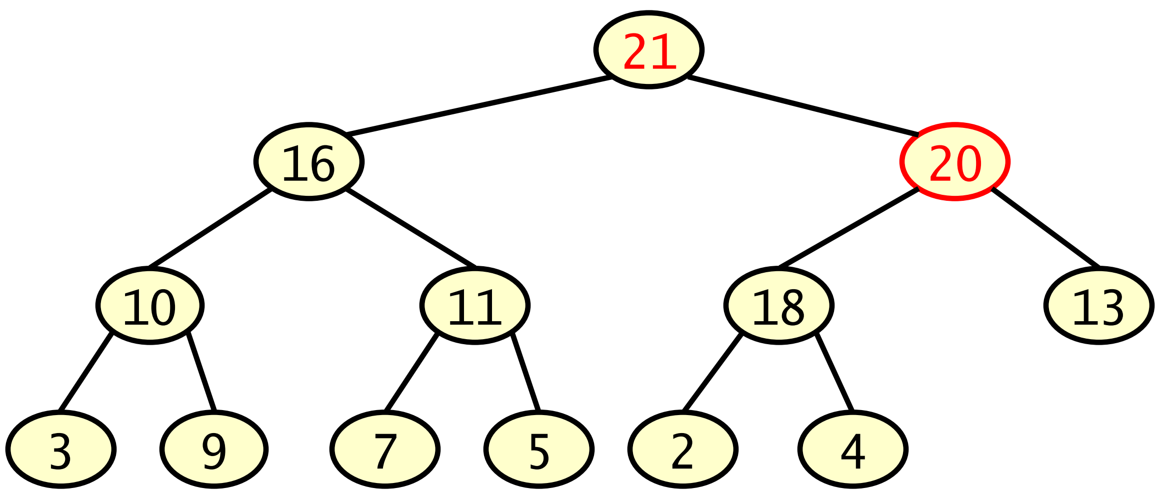


Heaps – Insertion algorithm

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
  swap new_value with parent_value;
```

For example we will insert 20 into the following heap

- new value less than parent so exit (finished insertion)

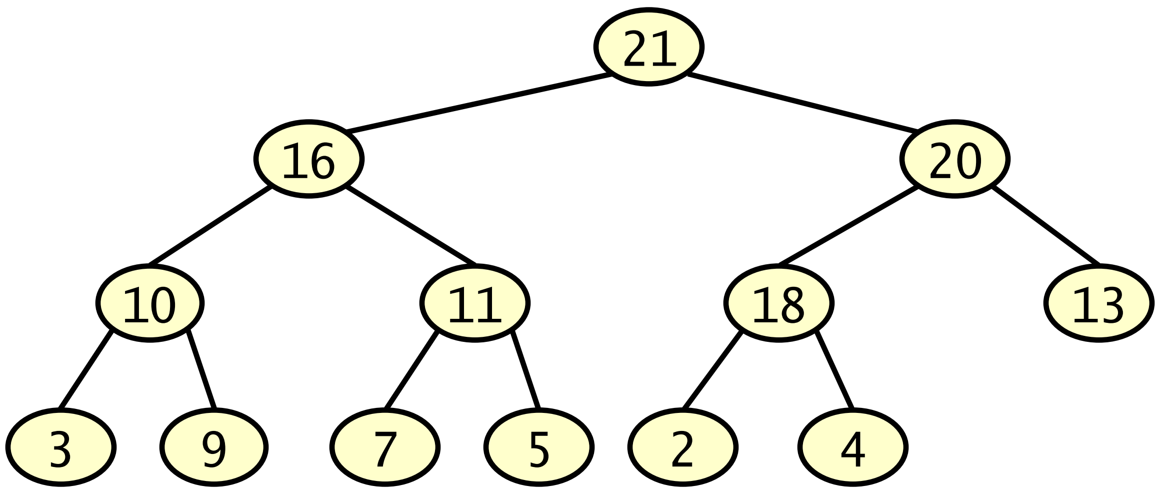


Heaps – Insertion algorithm

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
  swap new_value with parent_value;
```

For example we will insert 20 into the following heap

- new value less than parent so exit (finished insertion)



Heap property now holds

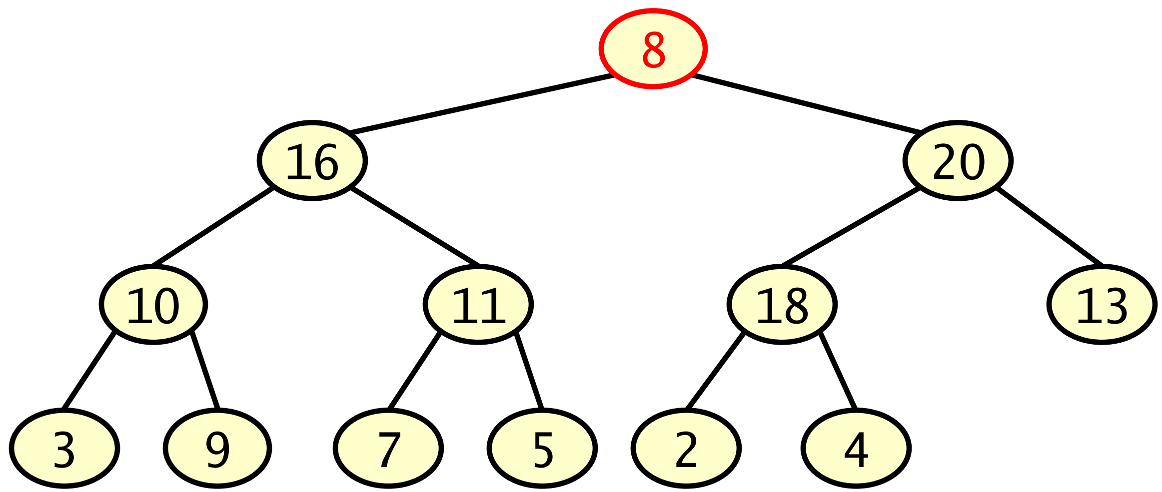
Heaps – Impose algorithm

```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

8 is a bad value in root



Heaps – Impose algorithm

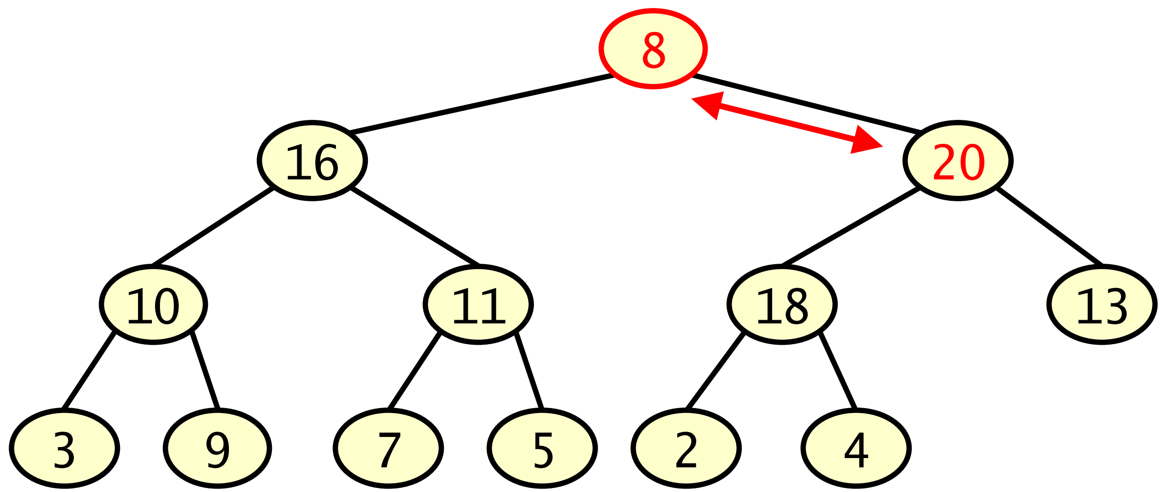
```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

8 is a bad value in root

- swap with larger child



Heaps – Impose algorithm

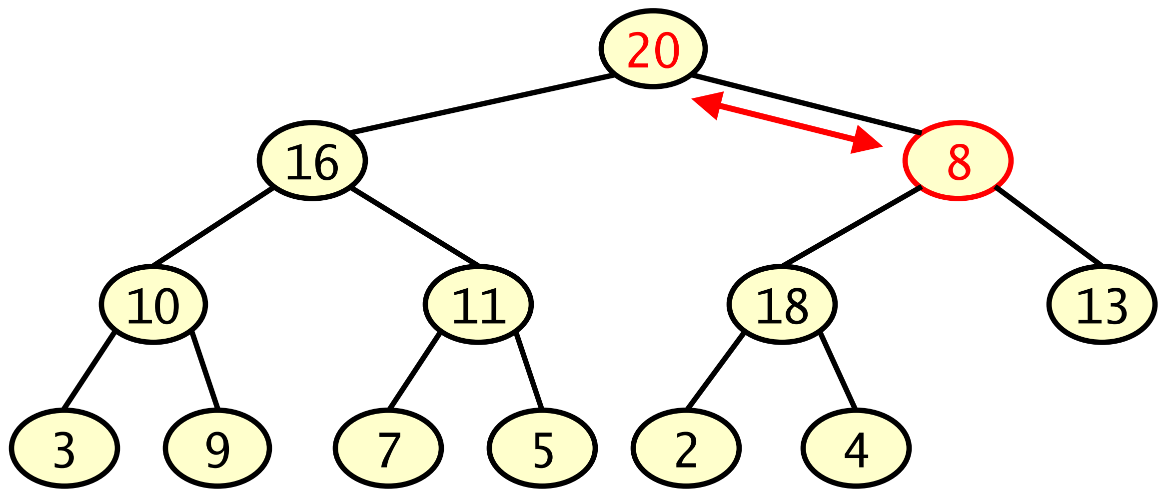
```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

8 is a bad value in root

– swap with larger child



Heaps – Impose algorithm

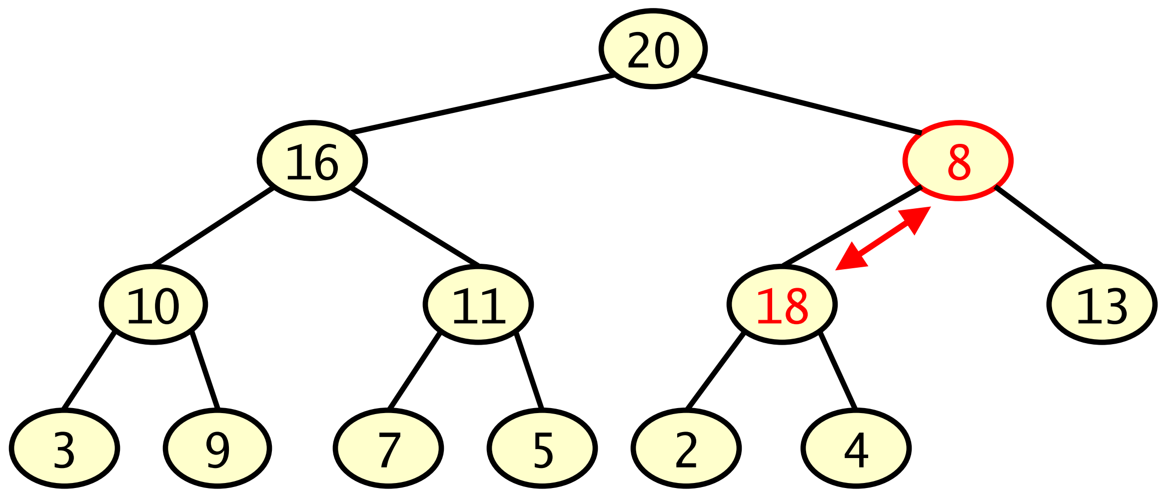
```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

8 is a bad value in root

– not leaf and smaller than larger child so swap again



Heaps – Impose algorithm

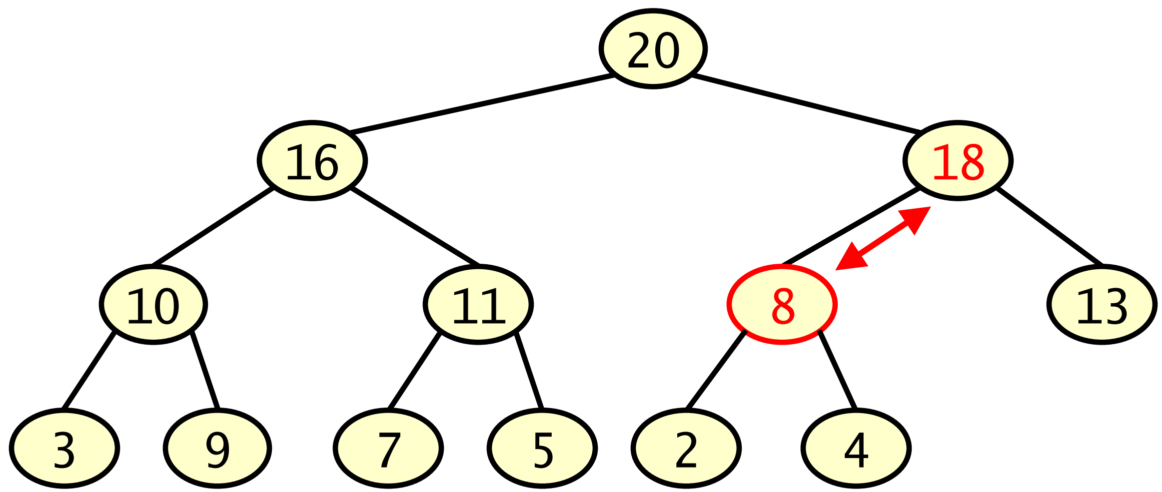
```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

8 is a bad value in root

– no longer smaller than larger child so exit



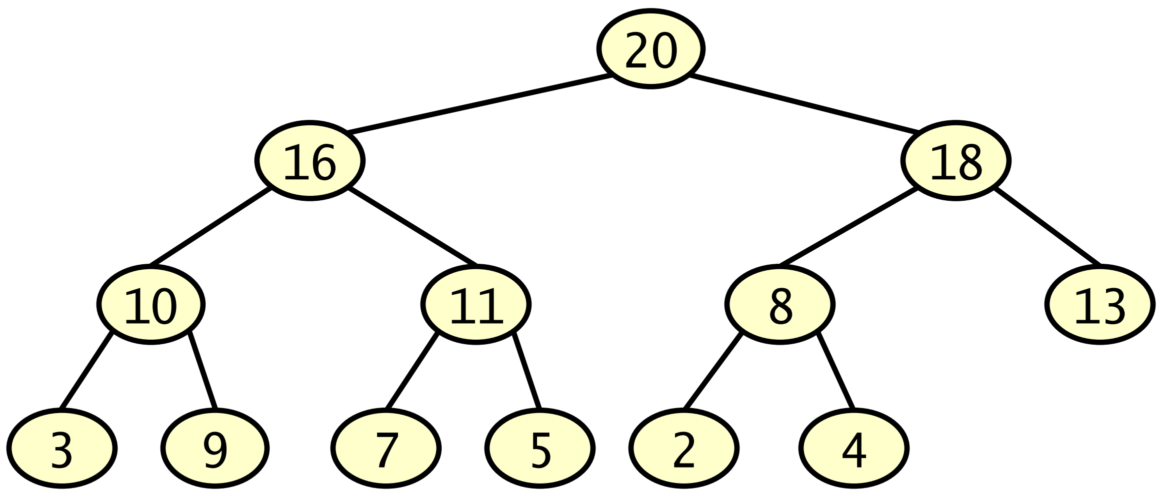
Heaps – Impose algorithm

```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Pre-condition: a specified node n may violate the heap property; all descendents of n satisfy the heap property

Post-condition: node n and all of its descendents satisfy the heap property, i.e. subtree rooted at n is a heap

Heap property imposed

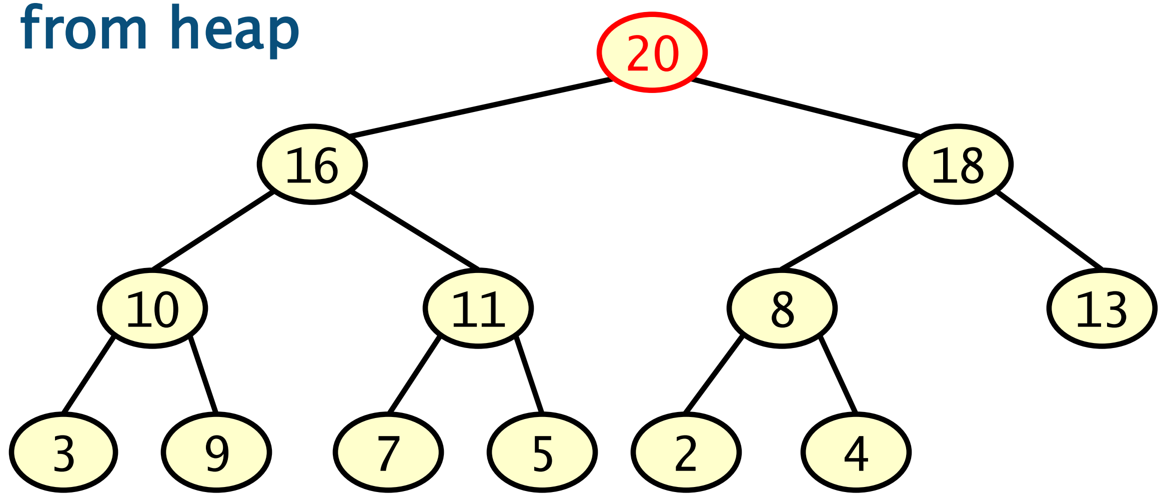


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

– by definition of heap maximum value of heap is the is root

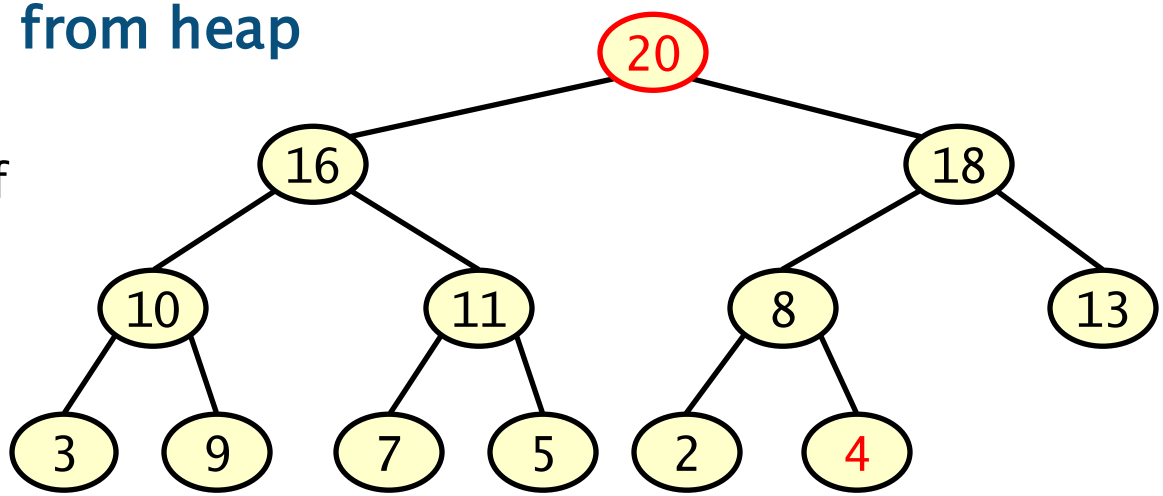


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- swap root value with last (bottom-right) leaf

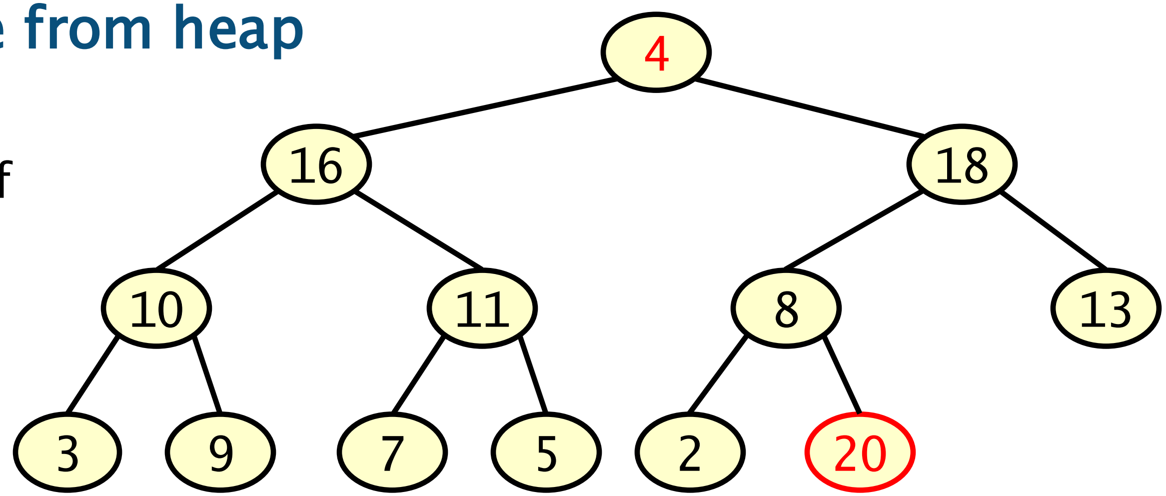


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- swap root value with last (bottom-right) leaf

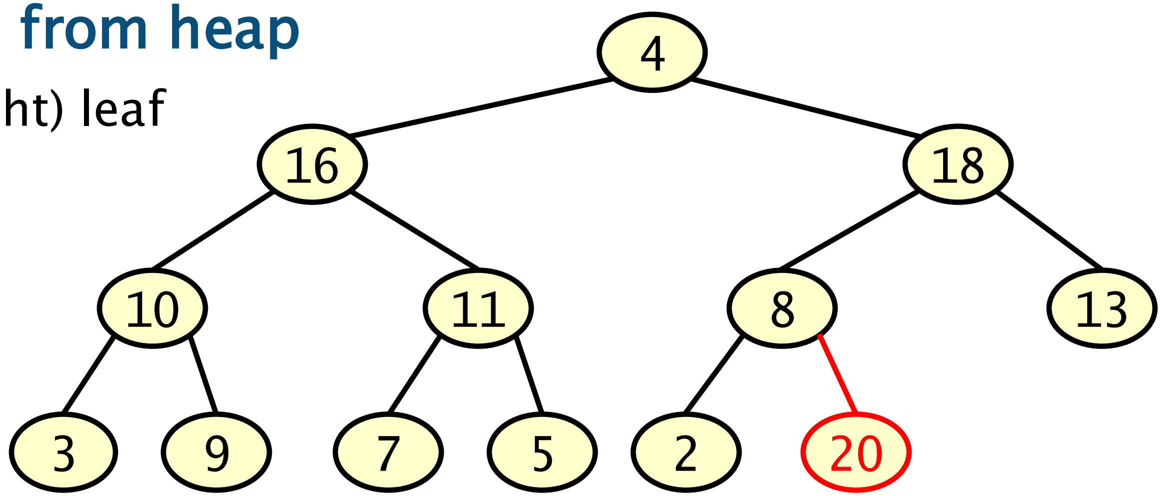


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- delete last (bottom-right) leaf

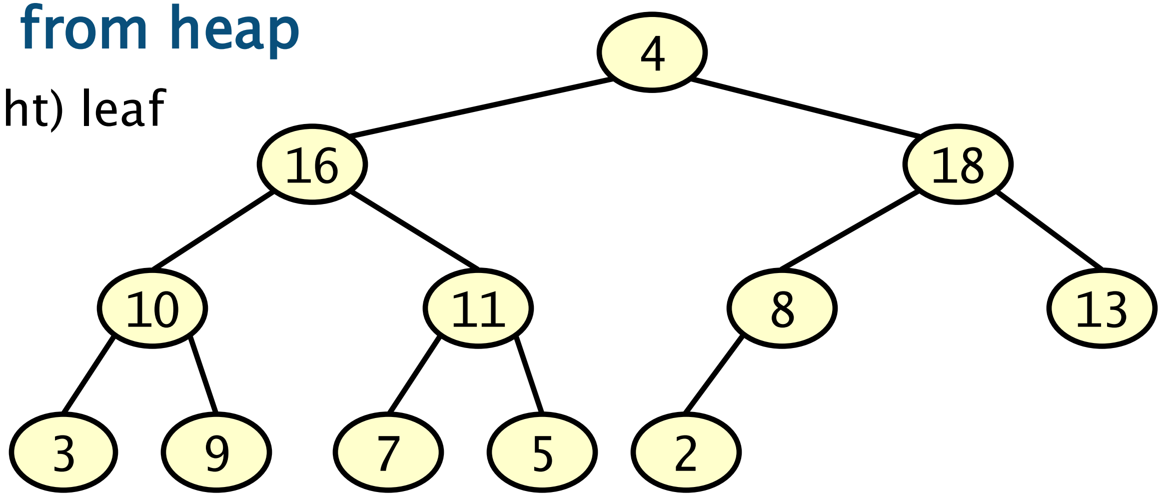


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- delete last (bottom-right) leaf

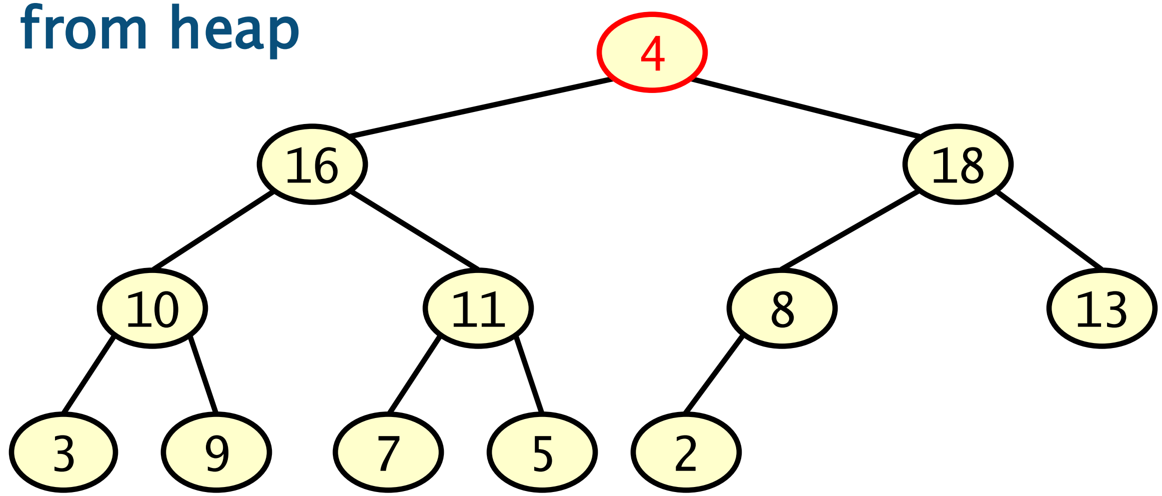


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- impose heap property on bad value in root

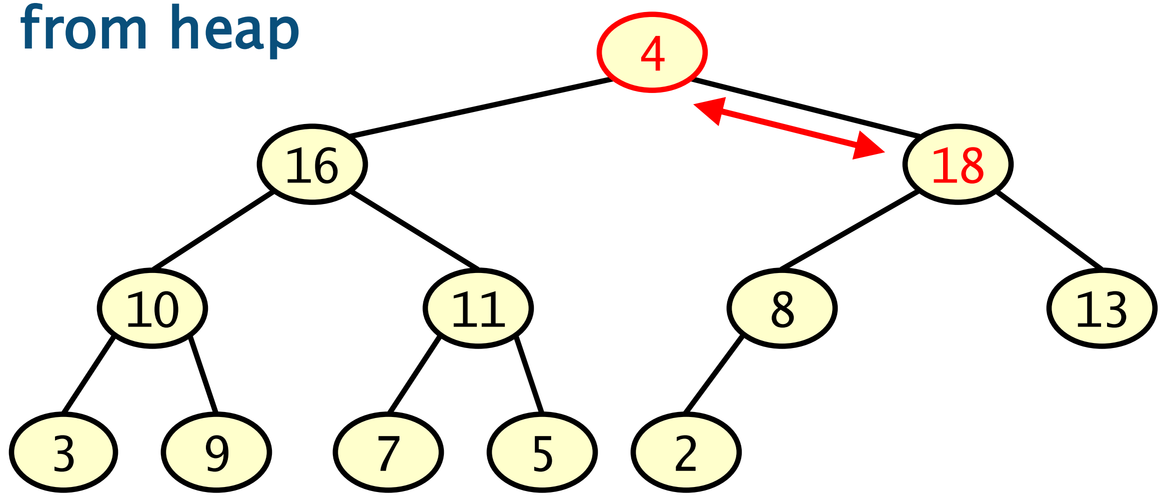


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- impose heap property on bad value in root
- swap with larger child

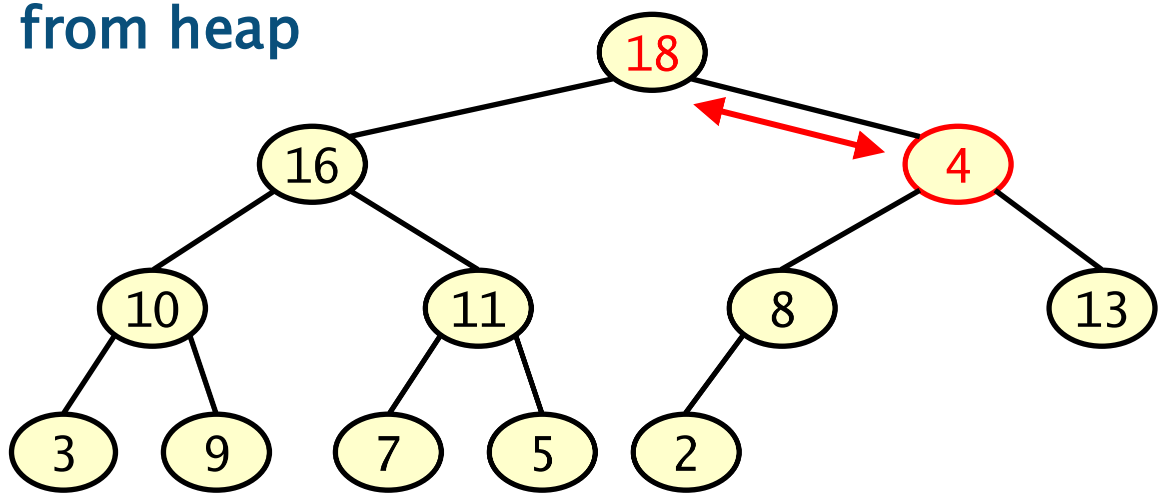


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- impose heap property on bad value in root
- swap with larger child

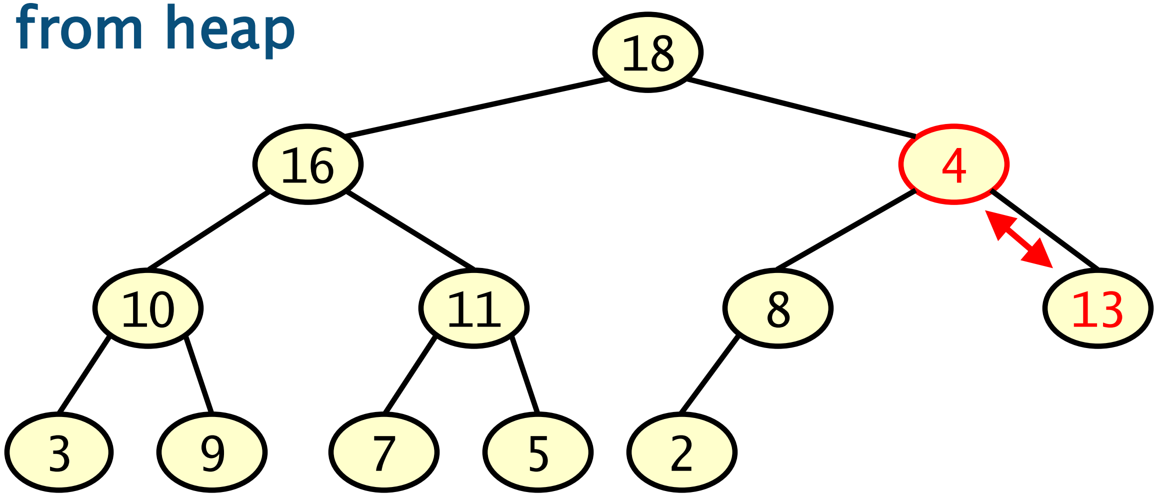


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- impose heap property on bad value in root
- 4 is still a bad value (smaller than children) so swap again with larger child

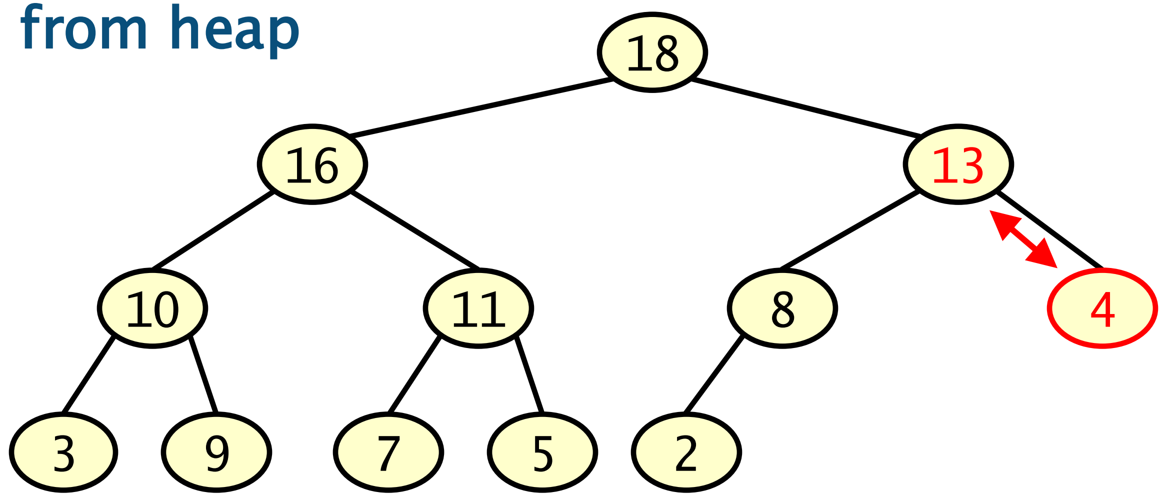


Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- impose heap property on bad value in root
- 4 is still a bad value (smaller than children) so swap again with larger child



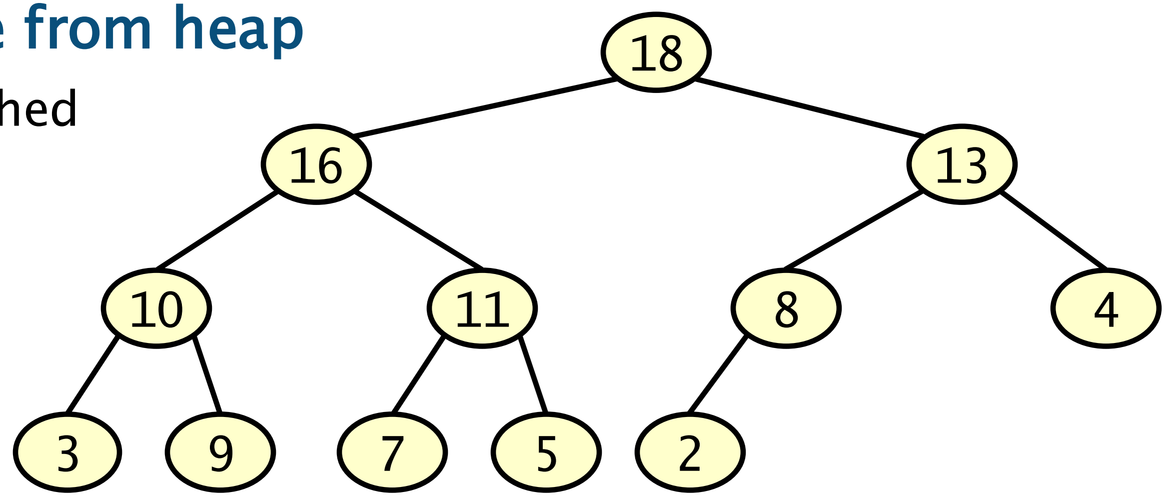
Heaps – Deletion algorithm

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

Removes maximum value from heap

- 4 is now a leaf so finished

Heap property imposed



Heaps – Build algorithm

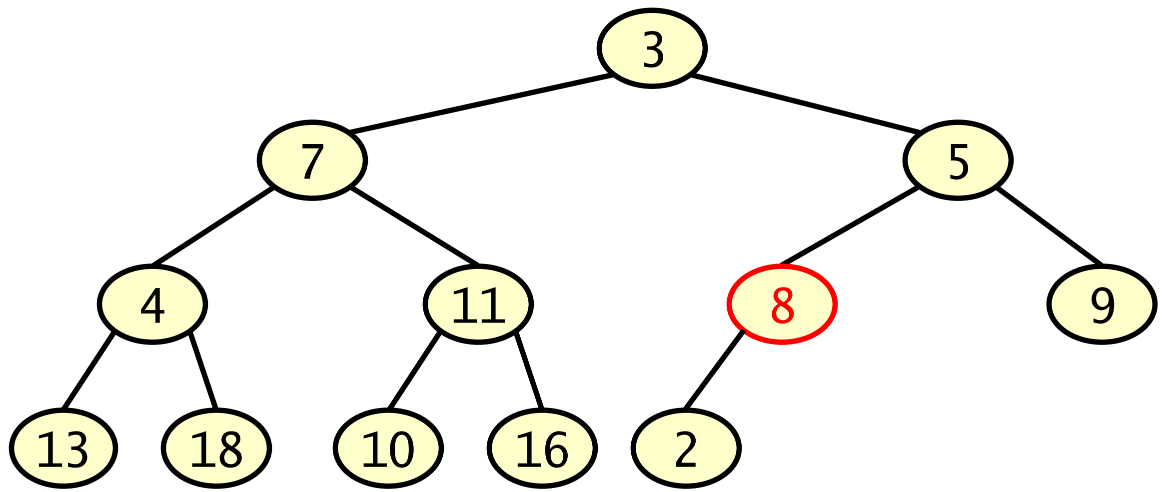
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

First non-leaf node in bottom-to-top right-to-left order: 8

– heap property holds on node so move to next node



Heaps – Build algorithm

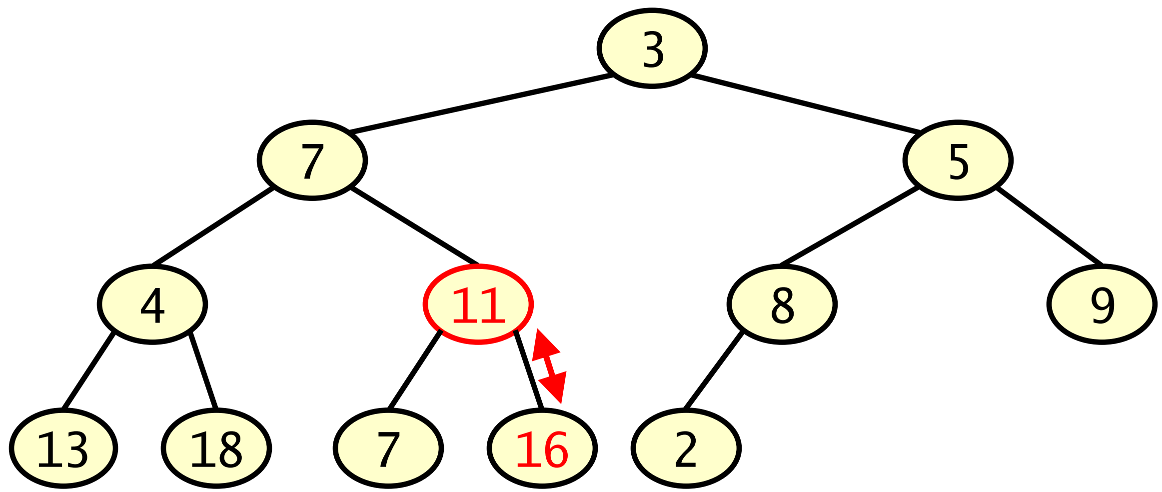
```
for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 11

- bad node so swap with larger child



Heaps – Build algorithm

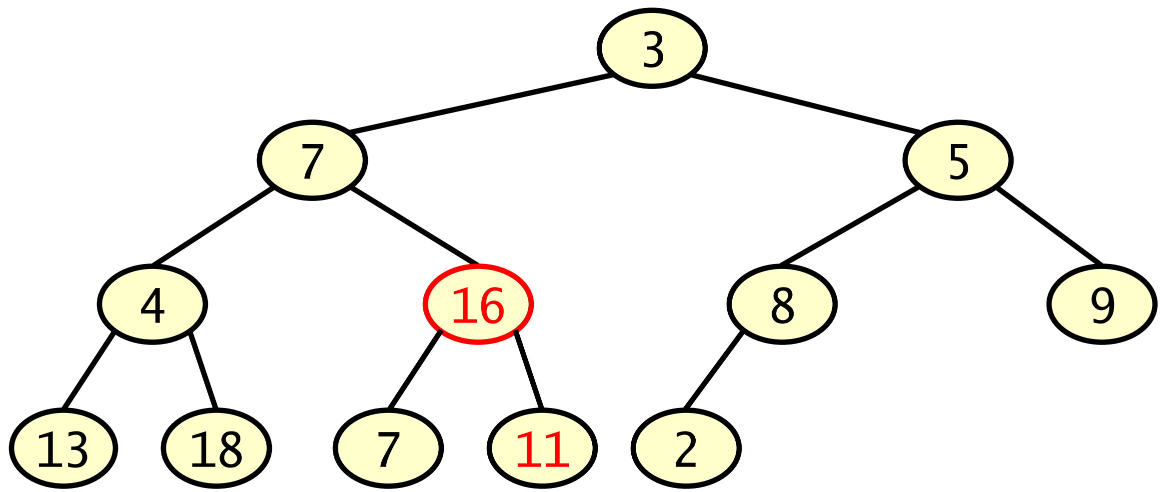
```
for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 11

- bad node so swap with larger child
- no longer a bad node



Heaps – Build algorithm

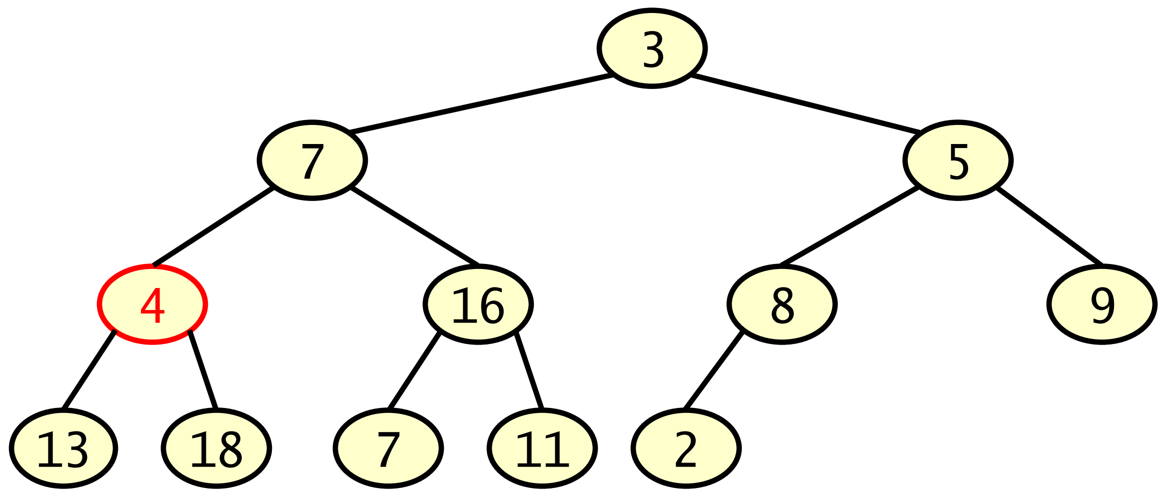
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 4

- bad node so swap with larger child



Heaps – Build algorithm

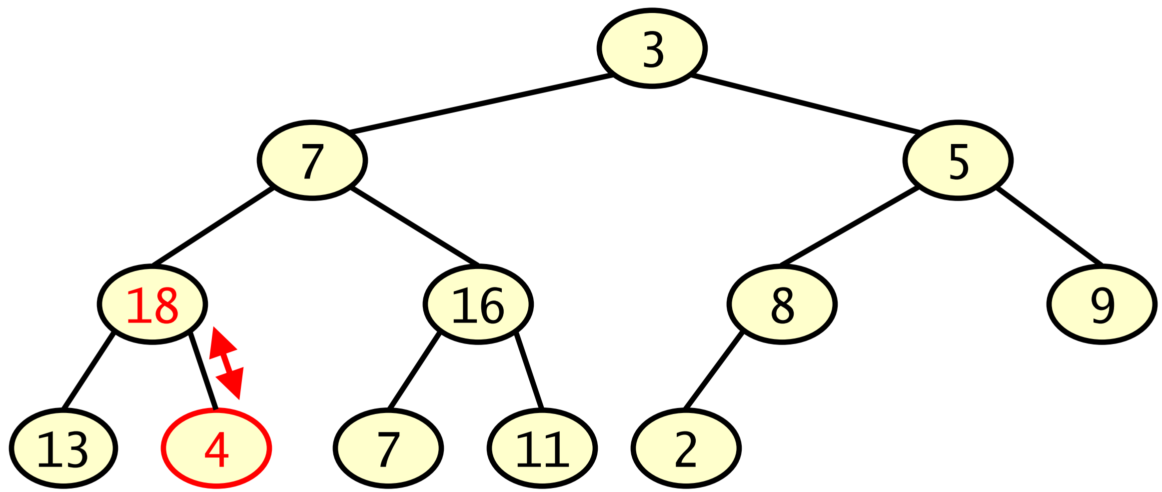
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 4

- bad node so swap with larger child
- no longer a bad node



Heaps – Build algorithm

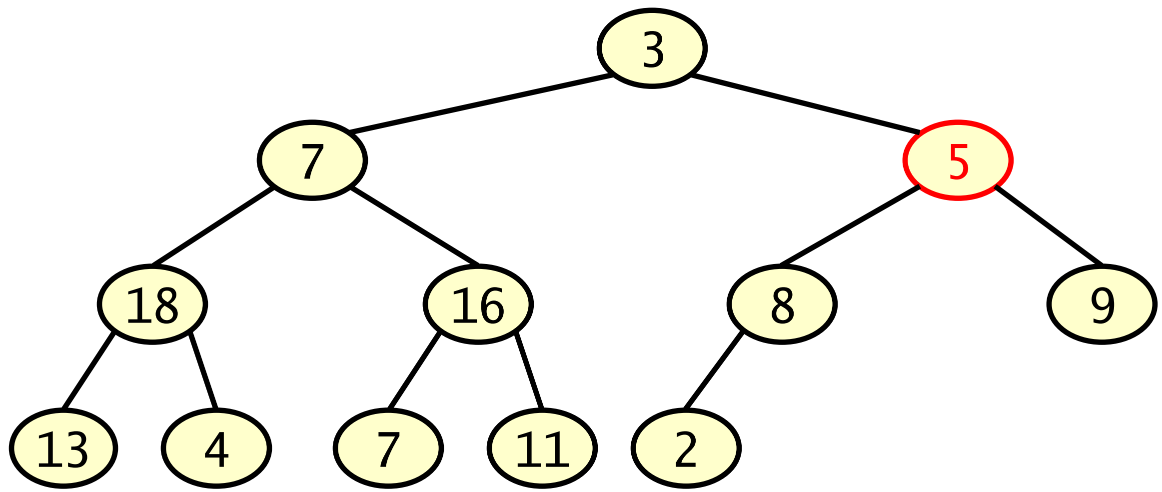
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 5

- bad node so swap with larger child



Heaps – Build algorithm

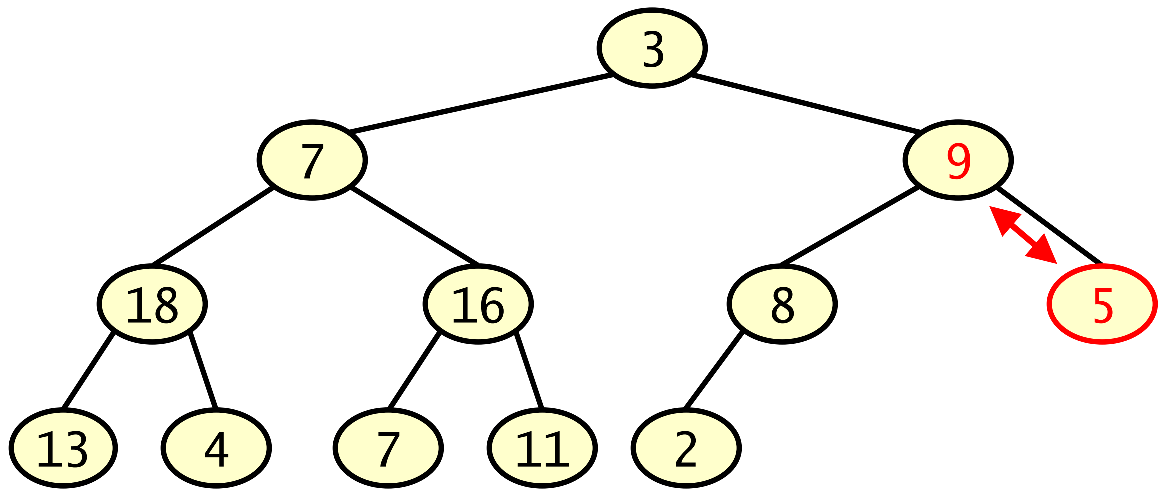
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 5

- bad node so swap with larger child
- no longer a bad node



Heaps – Build algorithm

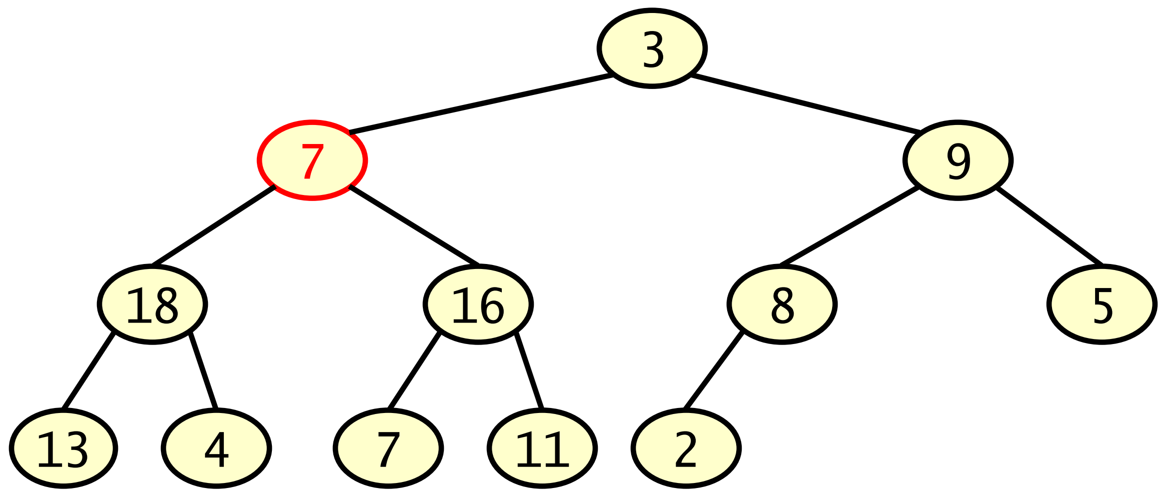
```
for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 7

- bad node so swap with larger child



Heaps – Build algorithm

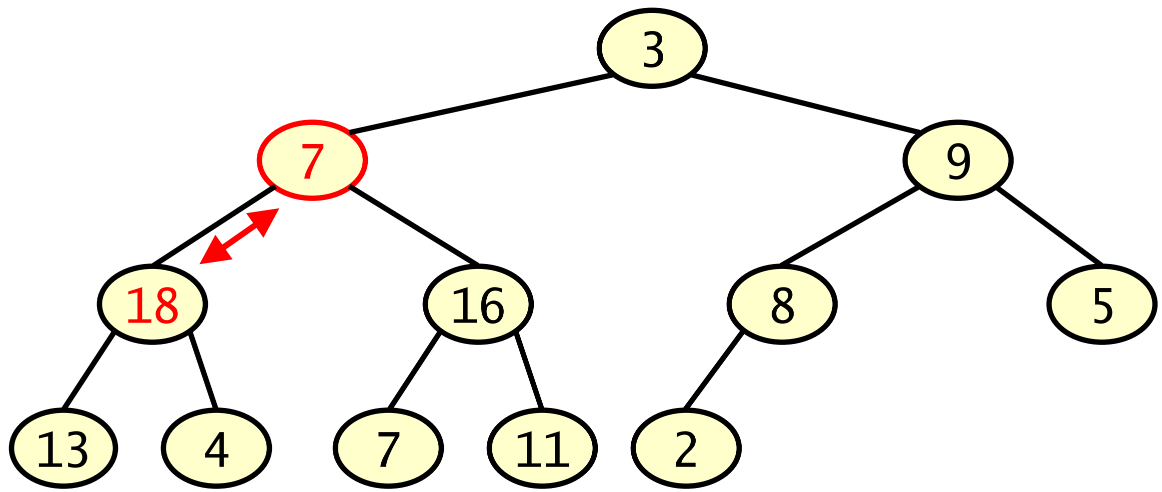
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 7

– bad node so swap with larger child



Heaps – Build algorithm

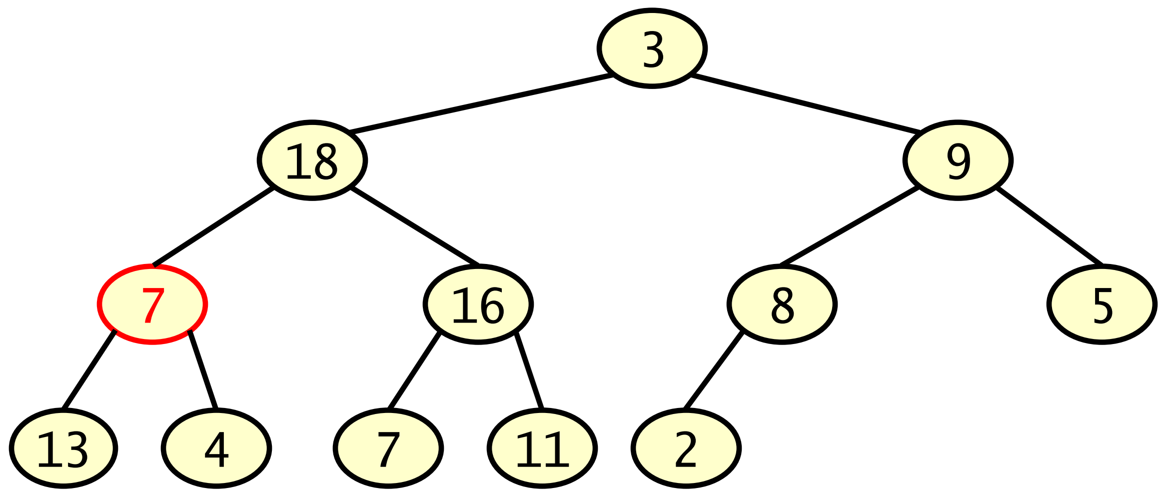
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 7

– 7 is still a bad node so swap with larger child



Heaps – Build algorithm

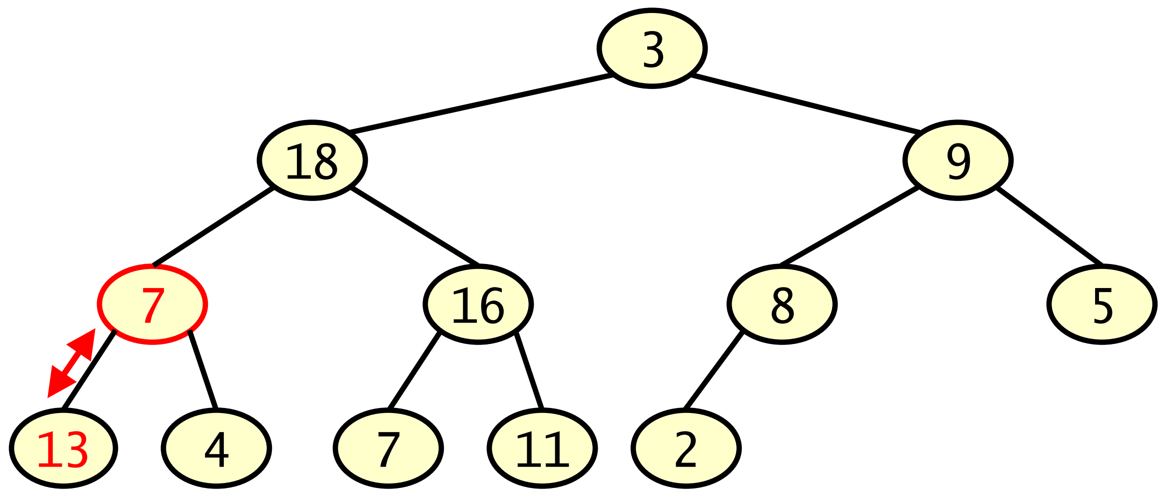
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Next non-leaf node in bottom-to-top right-to-left order: 7

- 7 is still a bad node so swap with larger child

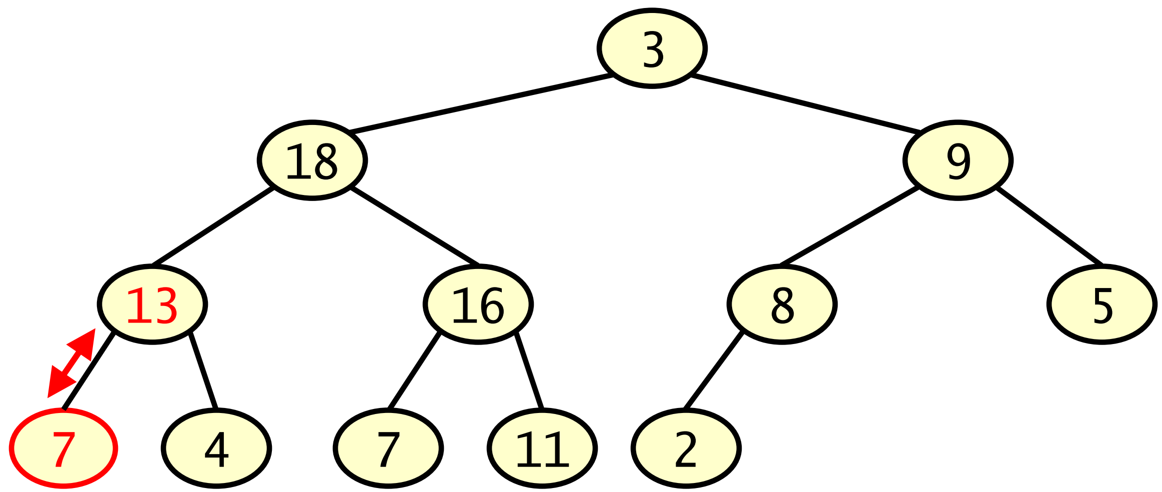


Heaps – Build algorithm

```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap



Next non-leaf node in bottom-to-top right-to-left order: 7

- 7 is still a bad node so swap with larger child
- no longer a bad node

Heaps – Build algorithm

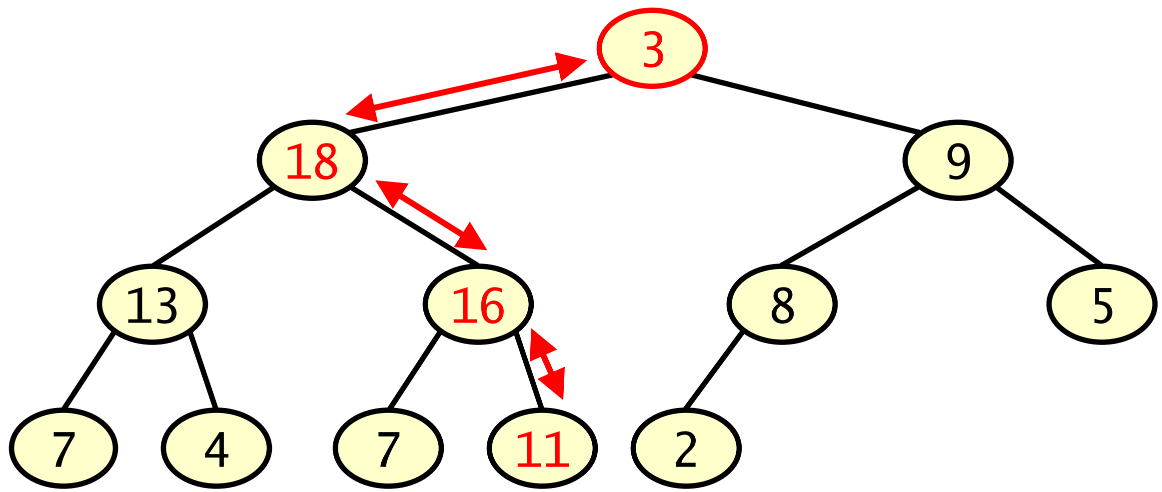
```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Final non-leaf node in bottom-to-top right-to-left order: 3

- bad node, in this case we need to swap 3 with 18, then with 16 and finally with 11

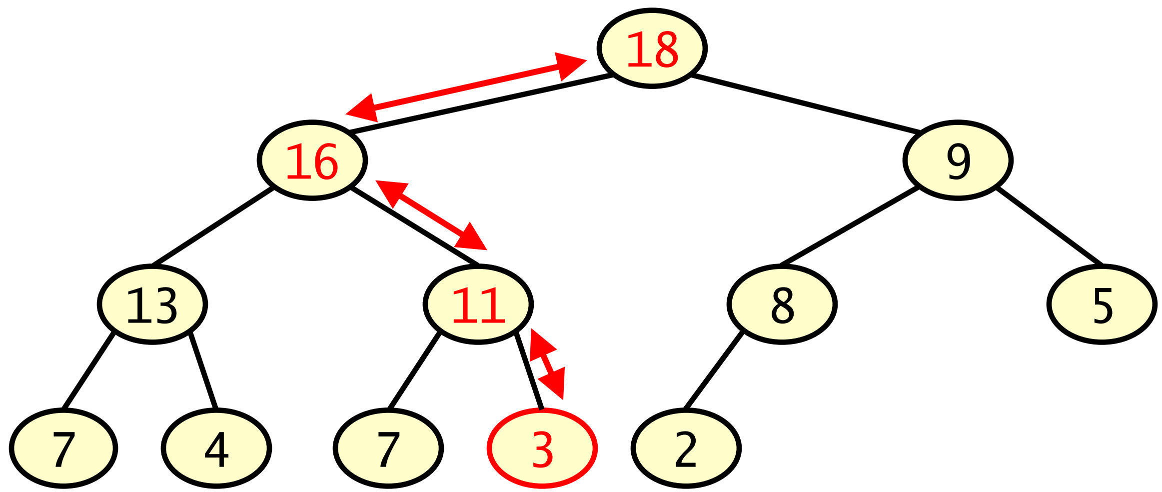


Heaps – Build algorithm

```
for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap



Final non-leaf node in bottom-to-top right-to-left order: 3

- bad node, in this case we need to swap 3 with 18, then with 16 and finally with 11

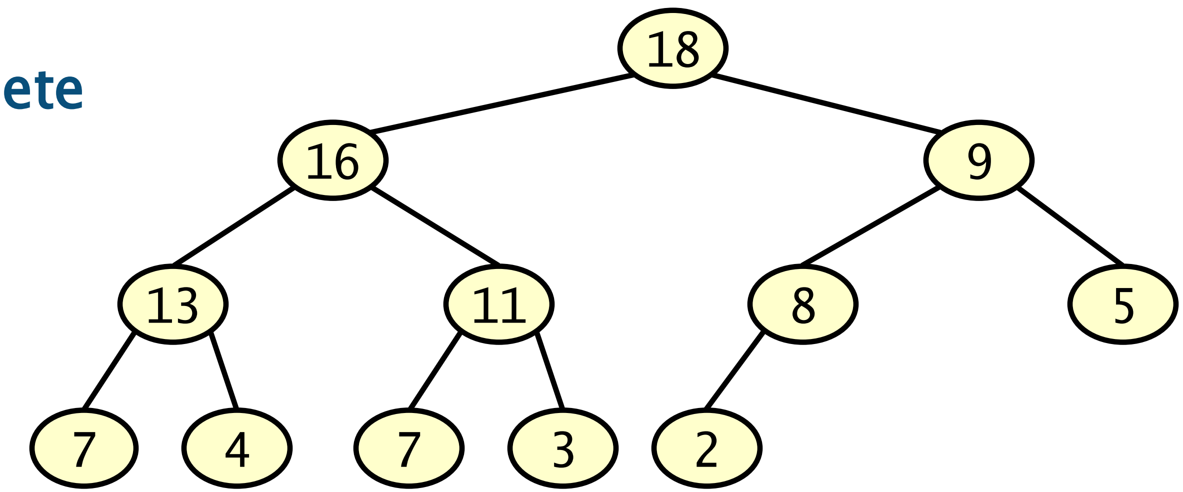
Heaps – Build algorithm

```
for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

Heap build is now complete and heap property holds



Fundamental algorithms & data structures

Stacks, queues, priority queues

Complete binary trees

Heaps and heap operations

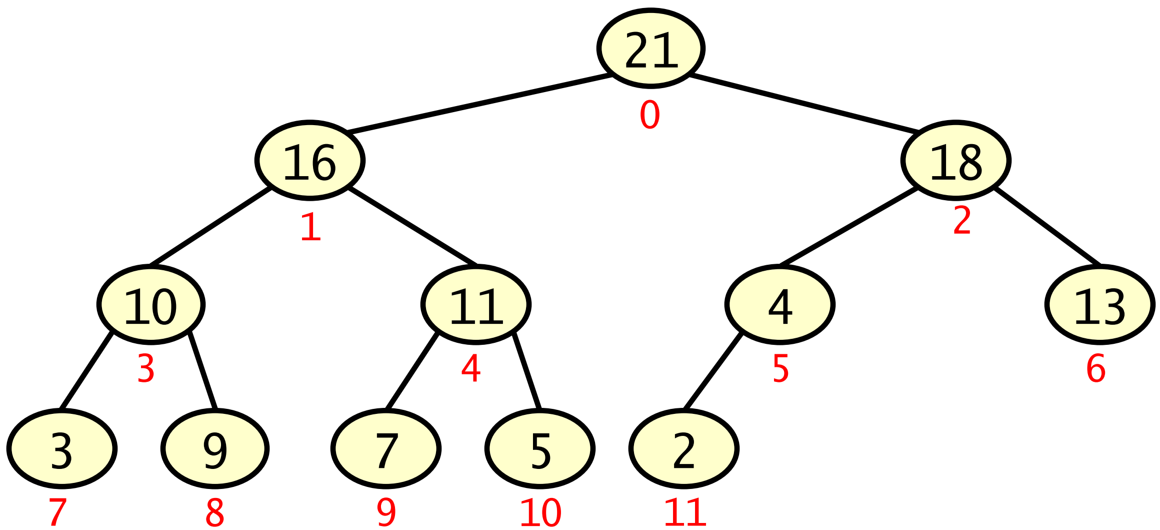
Java class for (integer) heaps

Heapsort

Heaps – An integer heap class

Representation a heap of size n as an array of size n

– uses the natural correspondence between the nodes and positions `0,…,n-1` of an array
  - i.e. children of node `i`, if they exist, are nodes `2i+1` and `2i+2`
  - conversely parent of node `i` is the node **floor**`((i-1)/2)`
    - use the fact under Java's integer arithmetic `(i-1)/2 = ` **floor**`((i-1)/2)`



Heaps – An integer heap class

```
/** Class (abbreviated) to represent heaps of integer valued items */
public class Heap {

  int size; // the size of the heap
  int[] items; // the heap items (stored as an array of integers)

  // create a new empty heap of maximum capacity n
  public Heap(int n) {
    size = 0; // heap is empty
    items = new int[n]; // array for heap items (max capacity n)
  }
  /** create new heap of capacity n containing items from an array a */
  public Heap(int n, int[] a) {
    size = a.length; // size of heap equals the size of the array
    items = new int[n]; // create array for heap items
    for (int i = 0; i < size; i++)
       items[i] = a[i]; // add values in arbitrary order
    build(); // use build algorithm to impose heap property
 }
```

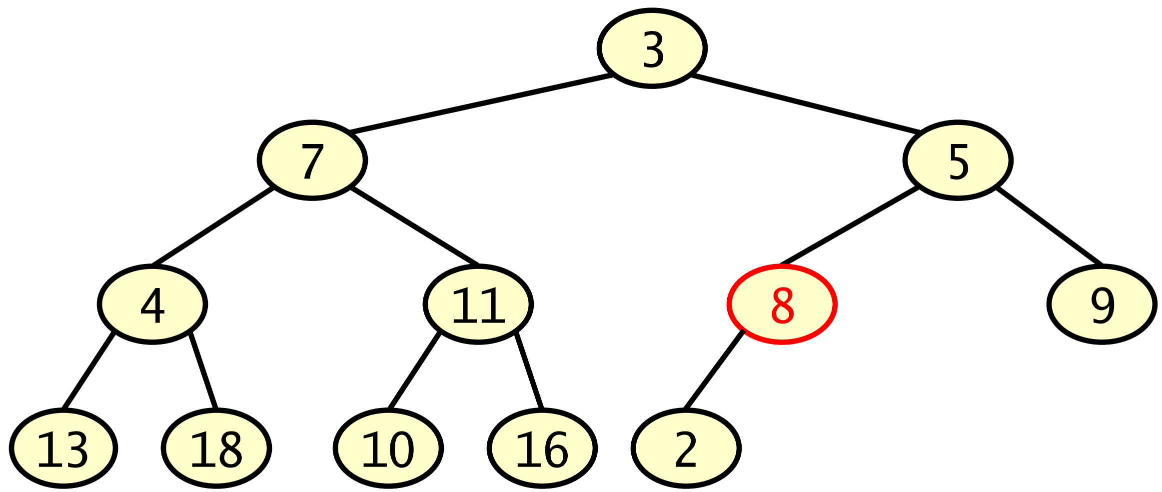
Heaps – Build algorithm

```
for (each non-leaf node in bottom-to-top right-to-left order)
  impose heap property on that node;
```

Pre-condition: values are in arbitrary order

Post-condition: values form a heap

First non-leaf node in bottom-to-top right-to-left order: **8**



Heaps – An integer heap class

```
 for (each non-leaf node in bottom-to-top right-to-left order)
   impose heap property on that node;
```

```
/** build a heap on current items */
private void build() {
   // for each non-leaf node in bottom-to-top right-to-left order
   for (int i = (size-1)/2; i >= 0; i--) // start at parent of final leaf
      impose(i); // impose heap property on that node
}
```

Heaps – An integer heap class

```
insert item in new leaf node;
while (new_value not in root && new_value > parent_value)
  swap new_value with parent_value;
```

```
/** insert item k into the heap */
public void insert(int k) {

   size++; // increase size of the heap
   int i = size-1; // current position (start at new leaf node)
   // while current position not root and parent smaller
   while (i > 0 && items[(i-1)/2] < k) {
     items[i] = items[(i-1)/2]; // swap with parent
     i = (i-1)/2; // new position is position of parent
   }
   items[i] = k; // finalise location of the item
}
```

Heaps – An integer heap class

```
// removes largest value (i.e. root) from the heap
swap root value with value in last (bottom-right) leaf;
delete last (bottom-right) leaf;
impose heap property on bad value in root;
```

```
/** delete and return maximum item */
public int deleteMax() {

  int k = items[0]; // maximum value (value at root)
  items[0] = items[size-1]; // swap root with last (bottom-right) leaf
  size--; // and delete last (bottom-right) leaf
  impose(0); // impose heap property on bad value in root
  return k; // return the maximum value
}
```

Heaps – Impose algorithm

```
// bad value violates the heap property
while (bad_value not in leaf && bad_value < larger_child)
  swap bad_value with larger_child;
```

Heaps – An integer heap class

```
/** impose the heap property on node i */
private void impose(int i) {

  int temp = items[i]; // copy item at position i
  int current = i; // current position to change (i.e. bad value)
  boolean finished = false; // not finished yet

  while (2*current+1 < size && !finished) { // not finished/reached leaf
    // find the larger child
    int next = 2*current+1; // assume initially it is the left child
    if (next+1 < size && items[next+1] > items[next])
      next++; // change if right child exists and is larger
    if (temp < items[next]) { // bad node (value < larger_child)
      items[current] = items[next]; // swap (child become parent)
      current = next; // new position (bad value moved to child node)
    }
    else finished = true; // not a bad node so finished
  }
  items[current] = temp; // finalise location of the item
}
```

Fundamental algorithms & data structures

Stacks, queues, priority queues

Complete binary trees

Heaps and heap operations

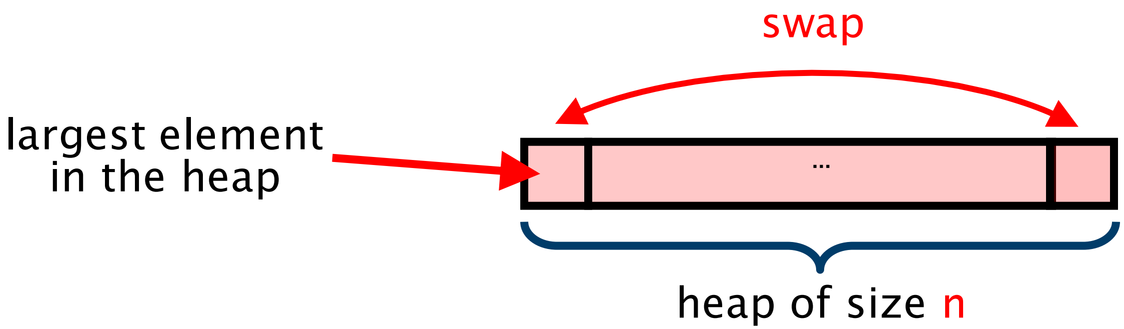
Java class for (integer) heaps

Heapsort

Heapsort

Like selectionsort but more efficient

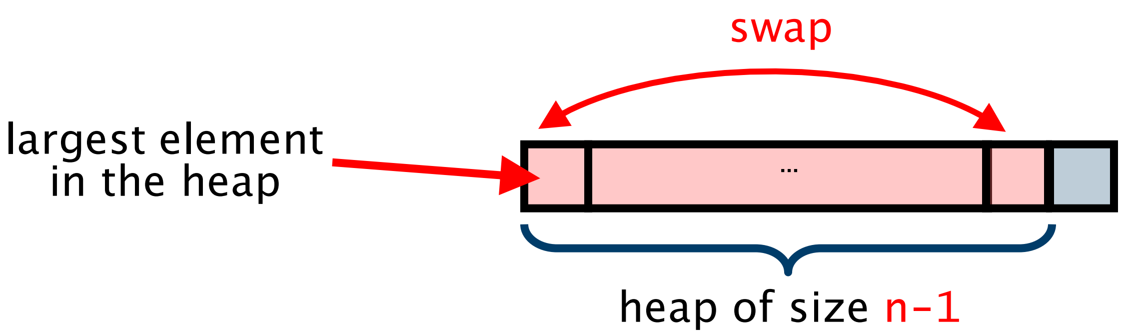
build heap and repeatedly remove largest element restoring heap structure



Heapsort

Like selectionsort but more efficient

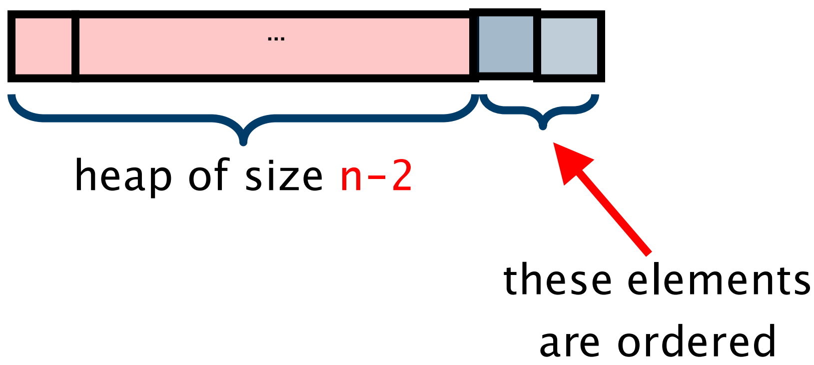
build heap and repeatedly remove largest element restoring heap structure



Heapsort

Like selectionsort but more efficient

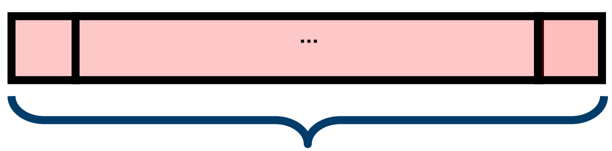
build heap and repeatedly remove largest element restoring heap structure



Heapsort

Like selectionsort but more efficient

```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){








 }
```

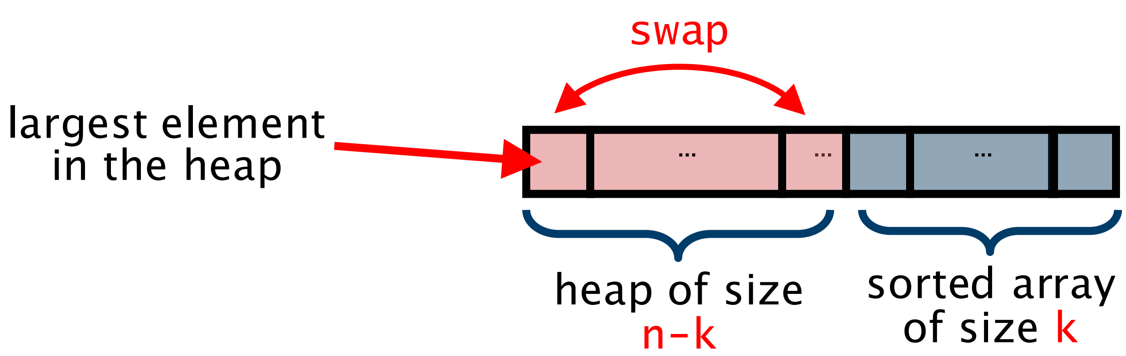


heap of size n

Heapsort

Like selectionsort but more efficient

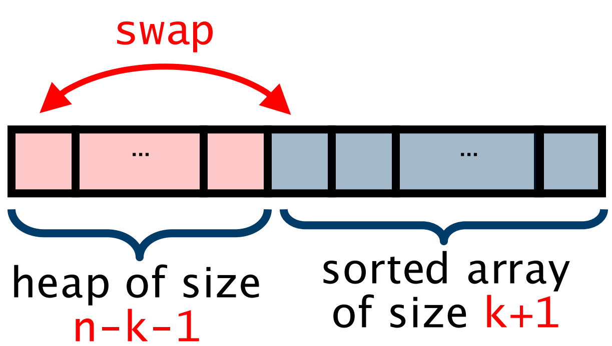
```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){
   // invariant: items 0,…,n-k-1 form a heap
   // invariant: items n-k,…,n-1 are sorted
   find the largest unsorted item; // is in position 0, so O(1)
   swap it into position n-1-k; // its correct place O(1)


 }
```



Heapsort

Like selectionsort but more efficient

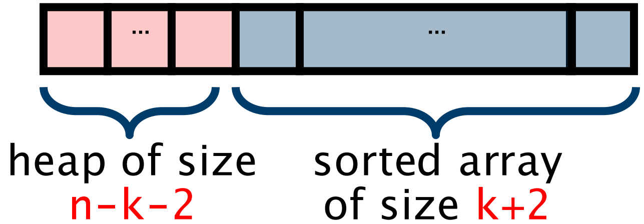
```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){
   // invariant: items 0,…,n-k-1 form a heap
   // invariant: items n-k,…,n-1 are sorted
   find the largest unsorted item; // is in position 0, so O(1)
   swap it into position n-1-k; // its correct place O(1)
   reduce the size of the heap by 1; // O(1)
   impose the heap property on position 0; // this is O(log n)
 }
```



Heapsort

Like selectionsort but more efficient

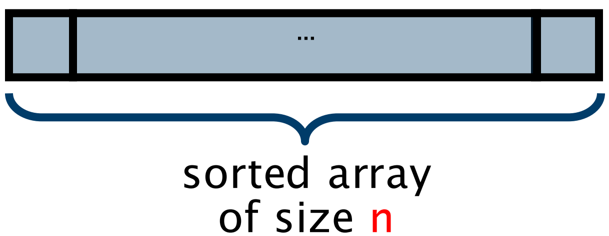
```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){
   // invariant: items 0,…,n-k-1 form a heap
   // invariant: items n-k,…,n-1 are sorted
   find the largest unsorted item; // is in position 0, so O(1)
   swap it into position n-1-k; // its correct place O(1)
   reduce the size of the heap by 1; // O(1)
   impose the heap property on position 0; // this is O(log n)
 }
```



Heapsort

Like selectionsort but more efficient

```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){
   // invariant: items 0,…,n-k-1 form a heap
   // invariant: items n-k,…,n-1 are sorted
   find the largest unsorted item; // is in position 0, so O(1)
   swap it into position n-1-k; // its correct place O(1)
   reduce the size of the heap by 1; // O(1)
   impose the heap property on position 0; // this is O(log n)
 }
restore size to original value;
```



Heapsort

Like selectionsort but more efficient

```
build sequence into a heap; // O(n)
 for (int k = 0; k < n-1; k++){
   // invariant: items 0,…,n-k-1 form a heap
   // invariant: items n-k,…,n-1 are sorted
   find the largest unsorted item; // is in position 0, so O(1)
   swap it into position n-1-k; // its correct place O(1)
   reduce the size of the heap by 1; // O(1)
   impose the heap property on position 0; // this is O(log n)
 }
restore size to original value;
```

Loop is iterated **n-1** times and each iteration takes **O(log n)** time

Hence heapsort is **O(n log n)** in the worst case