JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

# Heapsort

This topic teaches the Heapsort algorithm

- Figure 16.12 demonstrates the Standard Library algorithms for performing the heapsort sorting algorithm, in which an array of elements is arranged into a data structure called a *heap*
- Heapsort is discussed in detail in computer science courses called "Data Structures" and "Algorithms"

## `make_heap( a.begin(), a.end() );`

- Uses the `make_heap` algorithm to take a sequence of values in the range from `a.begin()` up to, but *not* including, `a.end()` and *create a heap* that can be used to produce a *sorted sequence*
- The two iterator arguments must be *random-access iterators*, so this algorithm will work only with arrays, `vector`s and `deque`s
- A second version of this algorithm takes as a third argument a *binary predicate function* for *comparing* values

JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

## `sort_heap( a.begin(), a.end() );`

- Uses the `sort_heap` algorithm to *sort a sequence of values* in the range from `a.begin()` up to, but *not* including, `a.end()` that are already arranged in a heap
- The two iterator arguments must be *random-access iterators*
- A second version of this algorithm takes as a third argument a *binary predicate function* for *comparing* values

## `push_heap( v.begin(), v.end() );`

- Uses the `push_heap` algorithm to *add a new value into a heap*
- We take one element of array `init` at a time, *append it* to the *end* of `vector` `v` and perform the `push_heap` operation
- If the appended element is the only element in the `vector`, the `vector` is already a heap
- Otherwise, `push_heap` rearranges the `vector` elements into a heap

- Each time `push_heap` is called, it assumes that the last element currently in the `vector` (i.e., the one that is appended before the `push_heap` call) is the element being added to the heap and that all other elements in the `vector` are already arranged as a heap
- The two iterator arguments to `push_heap` must be random-access iterators
- A second version of this algorithm takes as a third argument a *binary predicate function* for comparing values

## ***`pop_heap( v.begin(), v.end() - j );`***

- Uses `pop_heap` to remove the *top* heap element
- This algorithm assumes that the elements in the range specified by its two *random-access iterator* arguments are already a heap
- Repeatedly removing the *top* heap element results in a sorted sequence of values
- Algorithm `pop_heap` *swaps* the *first* heap element (`v.begin()`) with the *last* heap element (the element before `v.end() - j`), then ensures that the elements up to, but *not* including, the last element still form a heap
- Notice in the output that, after the `pop_heap` operations, the `vector` is *sorted* in *ascending order*
- A second version of this algorithm takes as a third argument a *binary predicate function* for comparing values

- `is_heap`
  - Returns `true` if the elements in the specified range represent a heap
  - A second version of this algorithm takes as a third argument a *binary predicate function* for comparing values

- `is_heap_until`
  - Checks the specified range of values
  - Returns an iterator pointing to the last item in the range for which the elements up to, but *not* including, that iterator represent a heap

This topic taught the Heapsort algorithm