CSC212

Data Structure

Computer Science
City College of New York

Lecture 21

Recursive Sorting, Heapsort & STL Quicksort

Instructor: George Wolberg
Department of Computer Science
City College of New York

Topics

- Recursive Sorting Algorithms
  - Divide and Conquer technique
- An O(NlogN) Sorting Alg. using a Heap
  - making use of the heap properties
- STL Sorting Functions
  - C++ sort function
  - Original C version of qsort

The Divide-and-Conquer Technique

- Basic Idea:
  - If the problem is small, simply solve it.
  - Otherwise,
    - **divide** the problem into two smaller sub-problems, each of which is about half of the original problem
    - **Solve** each sub-problem, and then
    - **Combine** the solutions of the sub-problems

The Divide-and-Conquer Sorting Paradigm

1. Divide the elements to be sorted into two groups of (almost) equal size
2. Sort each of these smaller groups of elements (by recursive calls)
3. Combine the two sorted groups into one large sorted list

Mergesort

- Divide the array in the middle
- Sort the two half-arrays by recursion
- Merge the two halves

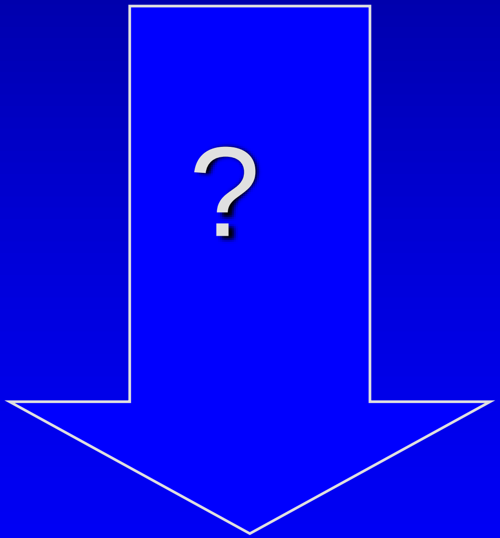
```
void mergesort(int data[ ], size_t n)
{
    size_t n1; // Size of the first subarray
    size_t n2; // Size of the second subarray

    if (n > 1)
    {
        // Compute sizes of the subarrays.
        n1 = n / 2;
        n2 = n - n1;

       // Sort from data[0] through data[n1-1]
        mergesort(data, n1);
       // Sort from data[n1] to the end
        mergesort((data + n1), n2);

        // Merge the two sorted halves.
        merge(data, n1, n2);
    }
}
```

Mergesort – an Example

16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Mergesort – an Example

16	12	7	6	3	2	18	10



2	3	6	7	10	12	16	18

Mergesort – an Example

16	12	7	6	3	2	18	10

divide

16	12	7	6

3	2	18	10

Mergesort – an Example

16	12	7	6	3	2	18	10

divide

16	12	7	6

3	2	18	10

divide

16	12

7	6

3	2

18	10

Mergesort – an Example

16	12	7	6	3	2	18	10

divide

16	12	7	6

3	2	18	10

divide

16	12

7	6

3	2

18	10

divide

16 | 12 | 7 | 6 | 3 | 2 | 18 | 10

Mergesort – an Example

16	12	7	6	3	2	18	10

divide: [16 | 12 | 7 | 6] [3 | 2 | 18 | 10]

divide: [16 | 12] [7 | 6] [3 | 2] [18 | 10]

divide: [16] [12] [7] [6] [3] [2] [18] [10]

merge: [12 | 16] [6 | 7] [2 | 3] [10 | 18]

Mergesort – an Example

| 16 | 12 | 7 | 6 | 3 | 2 | 18 | 10 |

divide | 16 | 12 | 7 | 6 | | 3 | 2 | 18 | 10 |

divide | 16 | 12 | | 7 | 6 | | 3 | 2 | | 18 | 10 |

divide | 16 | | 12 | | 7 | | 6 | | 3 | | 2 | | 18 | | 10 |

merge | 12 | 16 | | 6 | 7 | | 2 | 3 | | 10 | 18 |

merge | 6 | 7 | 12 | 16 | | 2 | 3 | 10 | 18 |

Mergesort – an Example

16	12	7	6	3	2	18	10

divide

[16 | 12 | 7 | 6] [3 | 2 | 18 | 10]

divide

[16 | 12] [7 | 6] [3 | 2] [18 | 10]

divide

[16] [12] [7] [6] [3] [2] [18] [10]

merge

[12 | 16] [6 | 7] [2 | 3] [10 | 18]

merge

[6 | 7 | 12 | 16] [2 | 3 | 10 | 18]

merge

2	3	6	7	10	12	16	18

Mergesort – two issues

- Specifying a subarray with pointer arithmetic
  - int data[10];
  - (data+i)[0] is the same as data[i]
  - (data+i][1] is the same as data[i+1]
- Merging two sorted subarrays into a sorted list
  - need a temporary array (by new and then delete)
  - step through the two sub-arrays with two cursors, and copy the elements in the right order

Mergesort - merge

data

6	7	12	16	2	3	10	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
c1				c2			

temp

?	?	?	?	?	?	?	?
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d							

Mergesort - merge

data

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
6	7	12	16	2	3	10	18

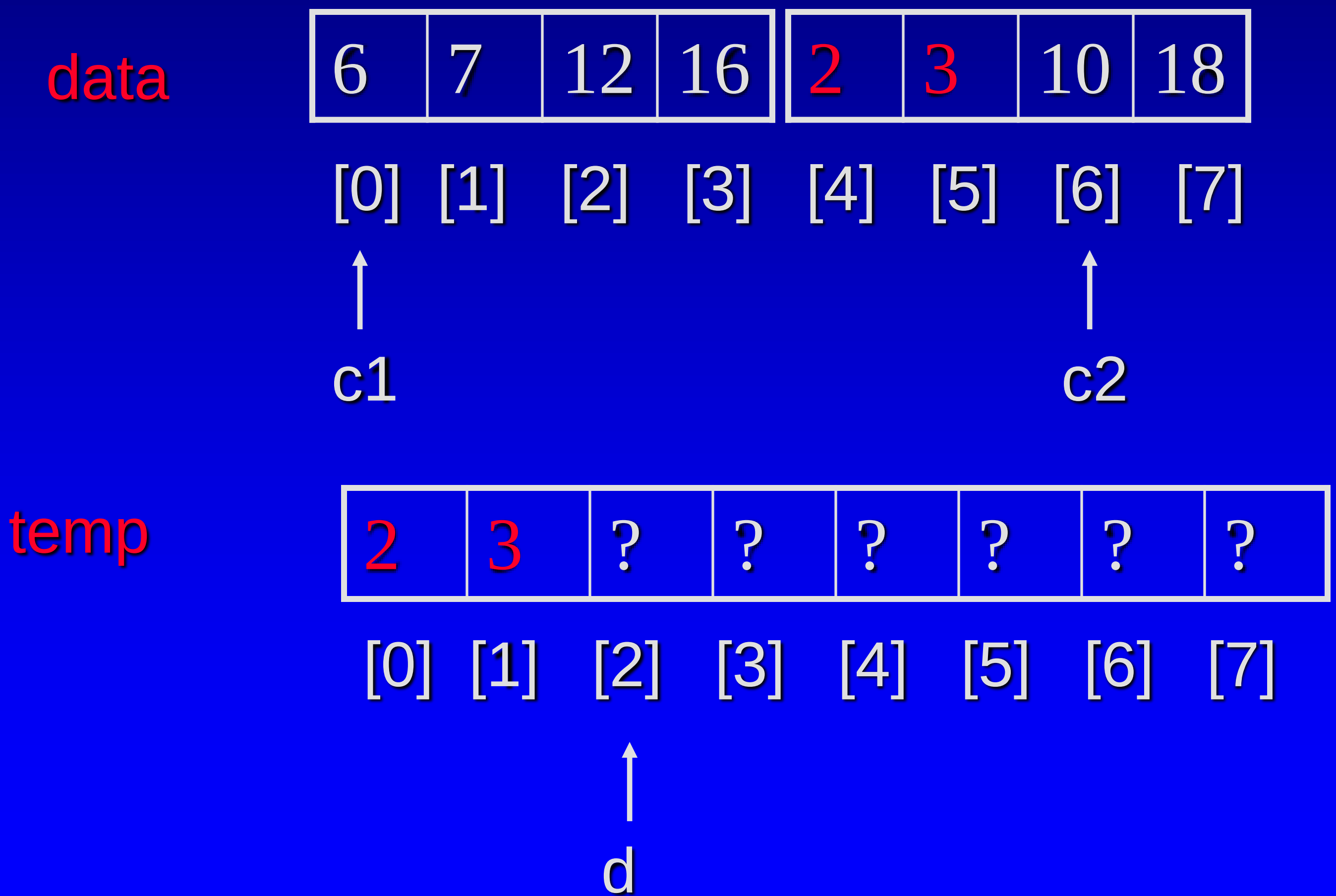
c1 c2

temp

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	?	?	?	?	?	?	?

d

Mergesort - merge



Mergesort - merge

data

6	7	12	16	2	3	10	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	c1					c2	

temp

2	3	6	?	?	?	?	?
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
			d				

Mergesort - merge

data

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
6	7	12	16	2	3	10	18

c1 c2

temp

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	6	7	?	?	?	?

d

Mergesort - merge

data

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
6	7	12	16	2	3	10	18

c1 → [2]

c2 → [7]

temp

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	6	7	10	?	?	?

d → [5]

Mergesort - merge

data

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
6	7	12	16	2	3	10	18

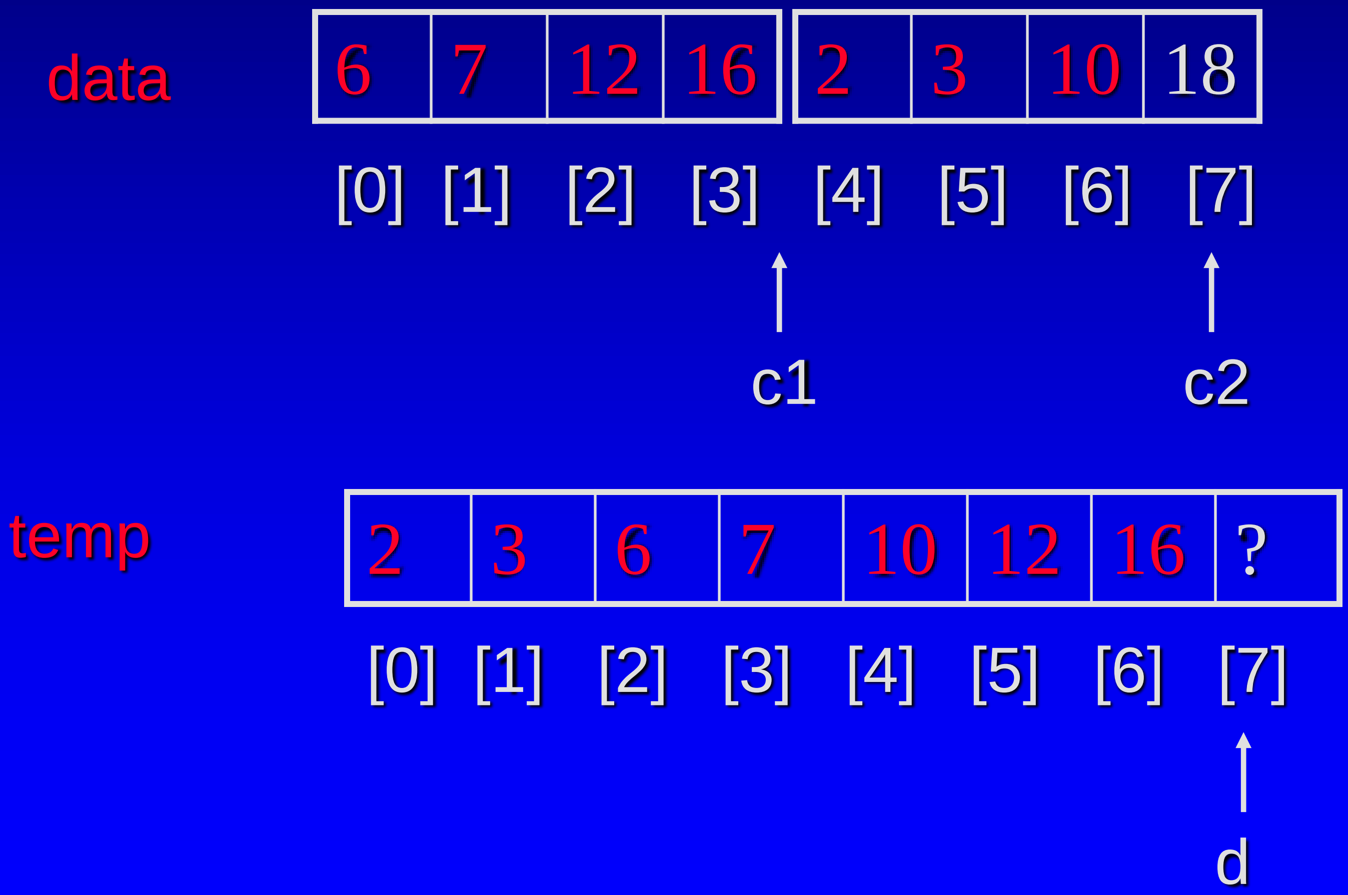
c1 → [3], c2 → [7]

temp

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	6	7	10	12	?	?

d → [6]

Mergesort - merge



Mergesort - merge

data

6	7	12	16	2	3	10	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

c1

c2

temp

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

d

Mergesort - merge

data

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	6	7	10	12	16	18

temp

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	6	7	10	12	16	18

Mergesort - merge

data

2	3	6	7	10	12	16	18

[0] [1] [2] [3] [4] [5] [6] [7]

Mergesort – Time Analysis

- The worst-case running time, the average-case running time and the best-case running time for mergesort are all O(n log n)

Mergesort – an Example

16 12 7 6 3 2 18 10

divide 16 12 7 6 | 3 2 18 10

divide 16 12 | 7 6 | 3 2 | 18 10

divide 16 | 12 | 7 | 6 | 3 | 2 | 18 | 10

merge 12 16 | 6 7 | 2 3 | 10 18

merge 6 7 12 16 | 2 3 10 18

merge 2 3 6 7 10 12 16 18

Mergesort – Time Analysis

- At the top (0) level, 1 call to merge creates an array with n elements
- At the 1st level, 2 calls to merge creates 2 arrays, each with n/2 elements
- At the 2nd level, 4 calls to merge creates 4 arrays, each with n/4 elements
- At the 3rd level, 8 calls to merge creates 8 arrays, each with n/8 elements

- At the *d*th level, 2^d calls to merge creates 2^d arrays, each with $n/2^d$ elements

- Each level does total work proportional to n => c n, where c is a constant
- Assume at the dth level, the size of the subarrays is $n/2^d = 1$, which means all the work is done at this level, therefore
  - the number of levels $d = \log_2 n$
- The total cost of the mergesort is $c\,nd = c\,n \log_2 n$
  - therefore the Big-O is $O(n \log_2 n)$

Heapsort

- Heapsort – Why a Heap? (two properties)
- Heapsort – How to? (two steps)
- Heapsort – How good? (time analysis)

Heap Definition

- A heap is a binary tree where the entries of the nodes can be compared with the *less than* operator of a strict weak ordering.
- In addition, two rules are followed:
  - The entry contained by the node is NEVER *less than* the entries of the node's children
  - The tree is a COMPLETE tree.

Why a Heap for Sorting?

- Two properties
  - The largest element is always at the root
  - Adding and removing an entry from a heap is O(log n)

Heapsort – Basic Idea

- Step 1. Make a heap from elements
  - add an entry to the heap one at a time
  - reheapification upward n times – O(n log n)
- Step 2. Make a sorted list from the heap
  - Remove the root of the heap to a sorted list and
  - Reheapification downward to re-organize into an updated heap
  - n times – O(n log n)

Heapsort – Step 1: Make a Heap

16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

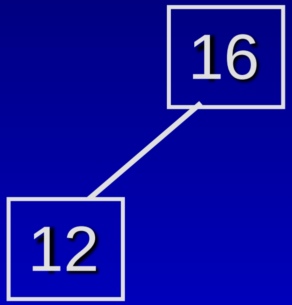
16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

16

add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

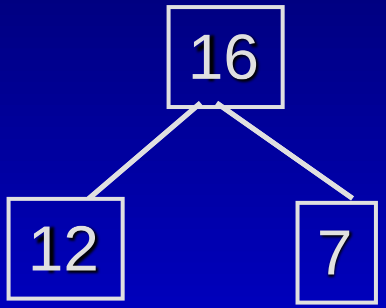
16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

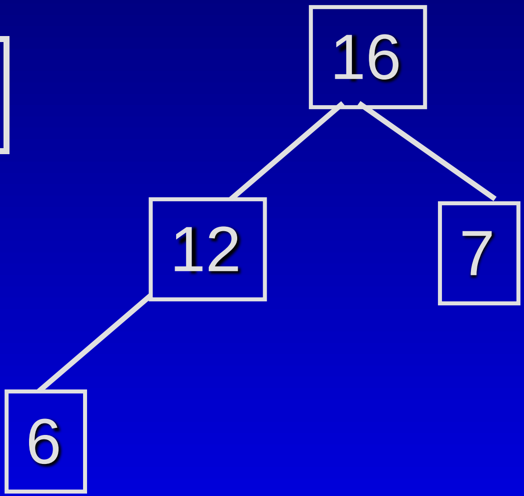
16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

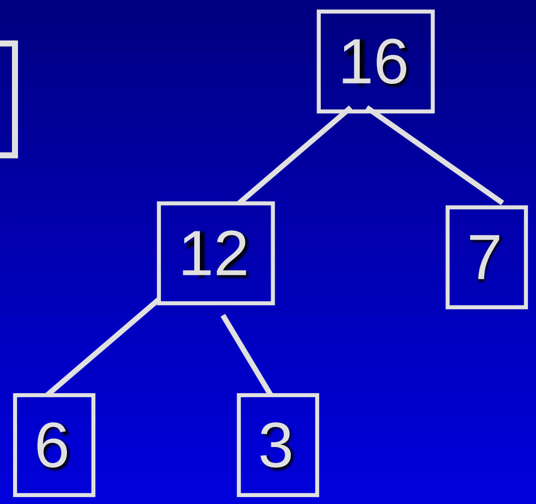
16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

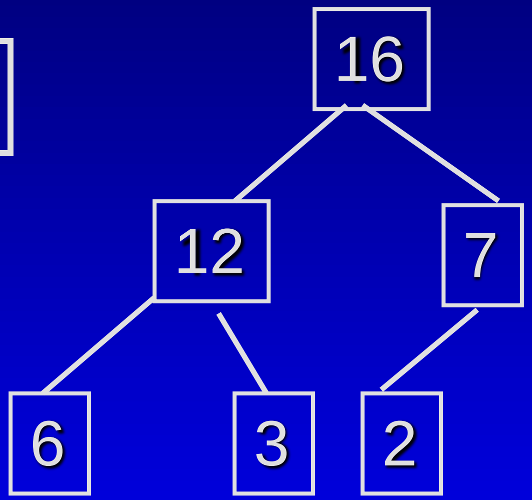
16	12	7	6	3	2	18	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

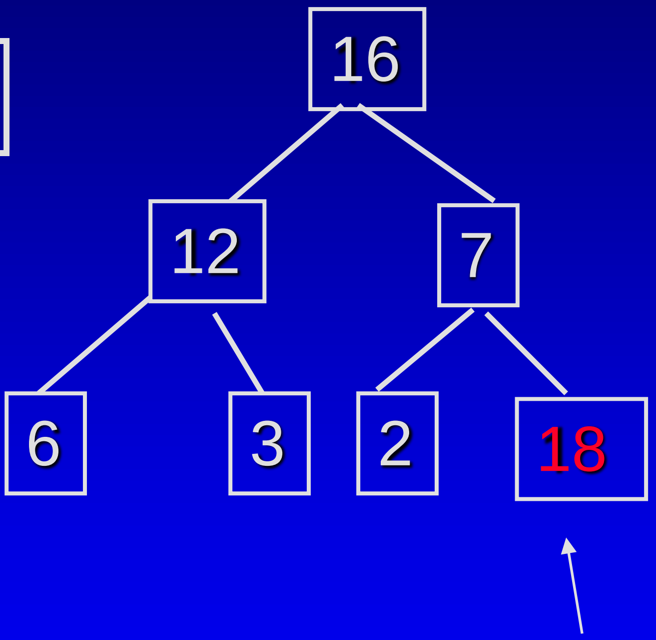
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
16	12	7	6	3	2	18	10



add an entry
to the heap
one at a time

Heapsort – Step 1: Make a Heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
16	12	7	6	3	2	18	10

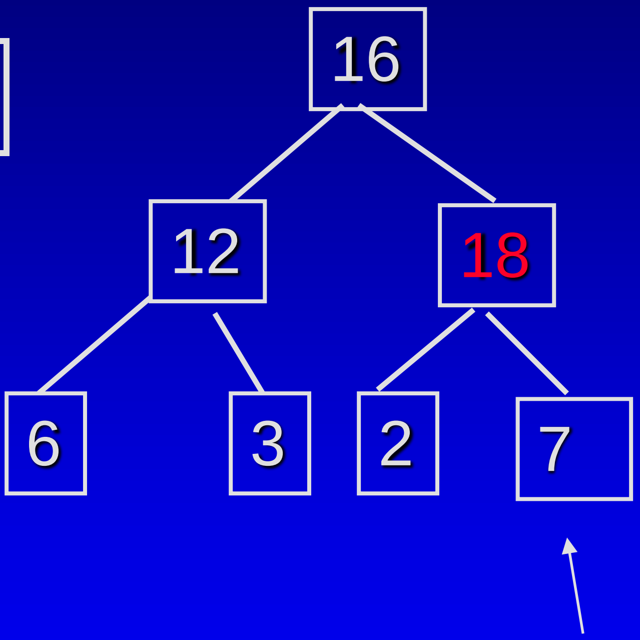


add an entry
to the heap
one at a time

reheapification upward: push the out-of-place node upward

Heapsort – Step 1: Make a Heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
16	12	18	6	3	2	7	10

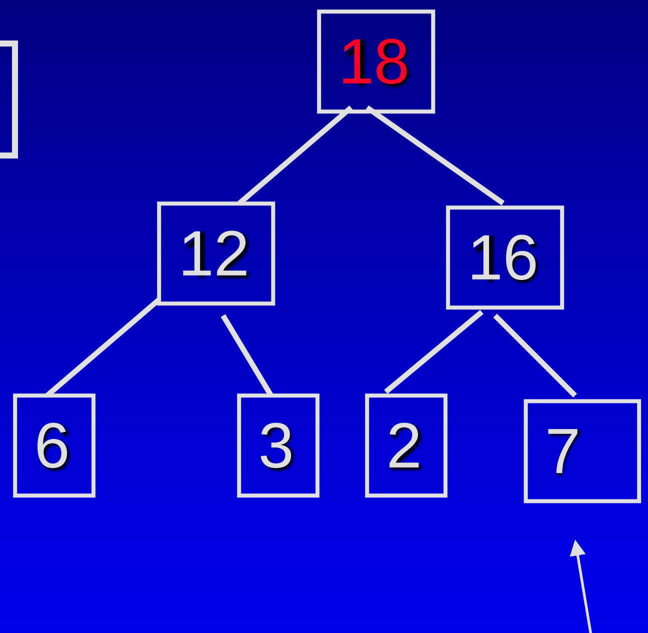


add an entry
to the heap
one at a time

reheapification upward: push
the out-of-place node upward

Heapsort – Step 1: Make a Heap

18	12	16	6	3	2	7	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

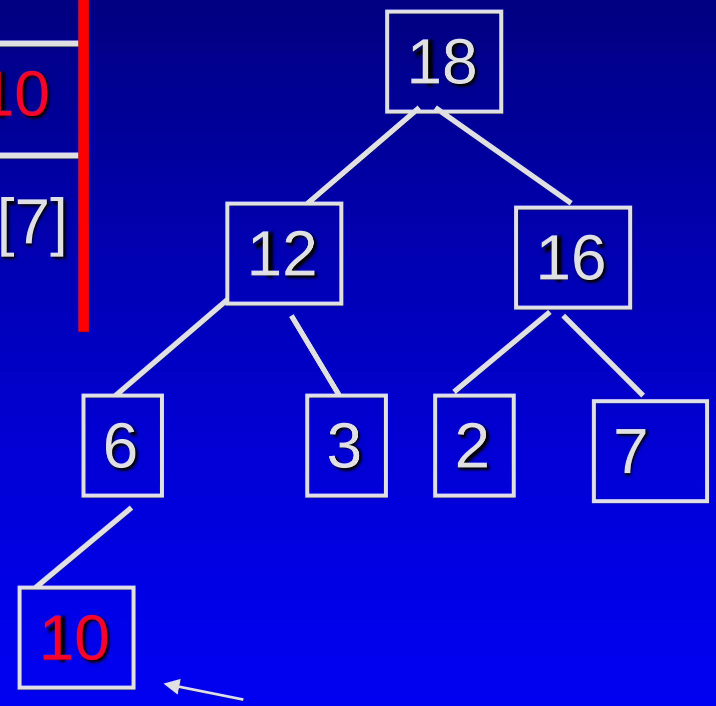


add an entry
to the heap
one at a time

reheapification upward: push the out-of-place node upward until it is in the right place

Heapsort – Step 1: Make a Heap

18	12	16	6	3	2	7	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

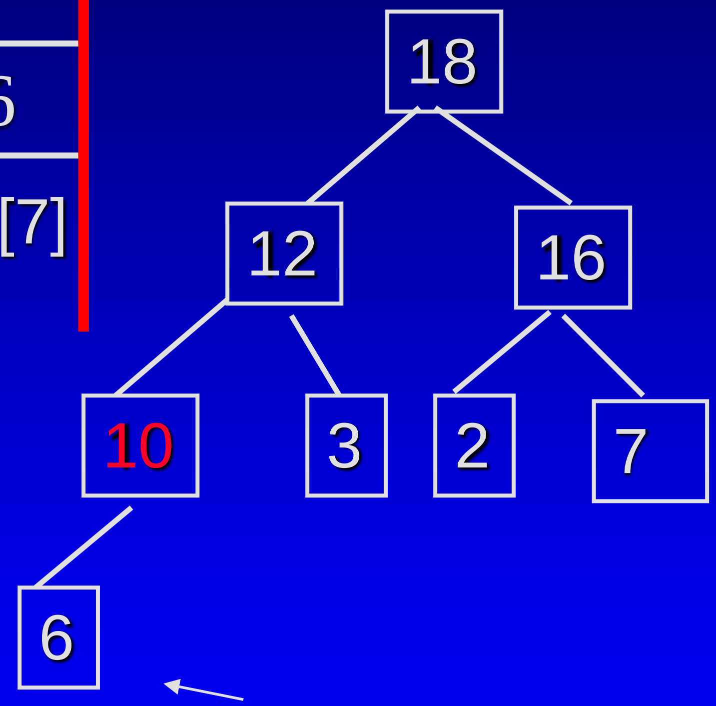


add an entry
to the heap
one at a time

reheapification upward: push the out-of-place node upward until it is in the right place

Heapsort – Step 1: Make a Heap

18	12	16	10	3	2	7	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

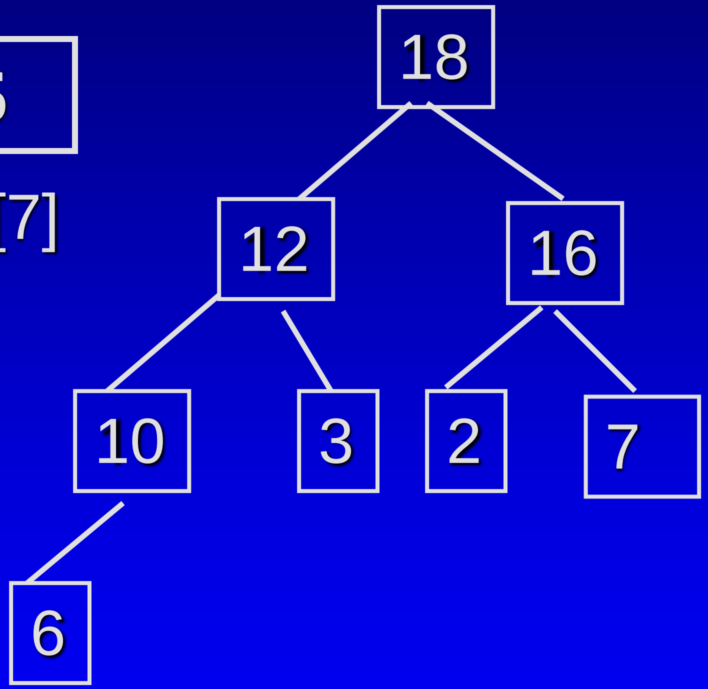


add an entry
to the heap
one at a time

reheapification upward: push the out-of-place node upward until it is in the right place

Heapsort – Step 1: Make a Heap

18	12	16	10	3	2	7	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

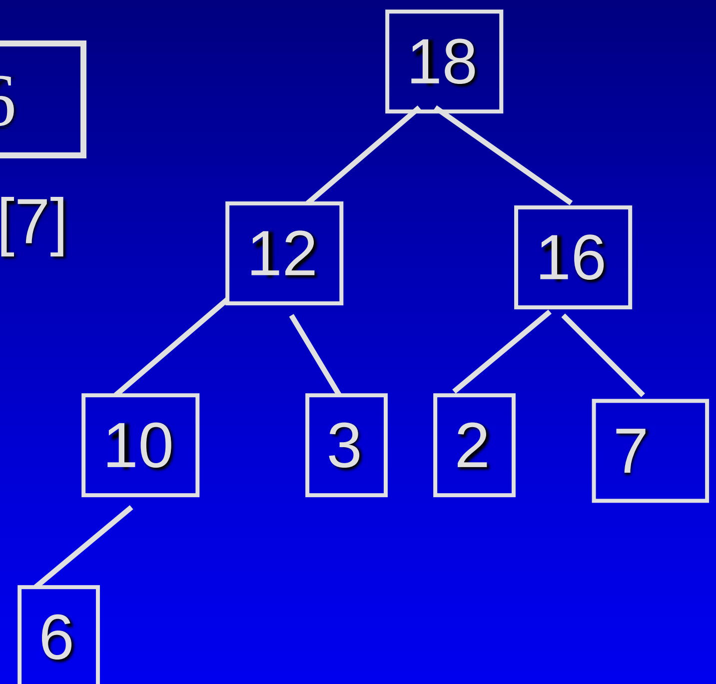


A heap is created: it is saved in the original array- the tree on the right is only for illustration!

Sorted???

Heapsort – Step 2: Sorting from Heap

18	12	16	10	3	2	7	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



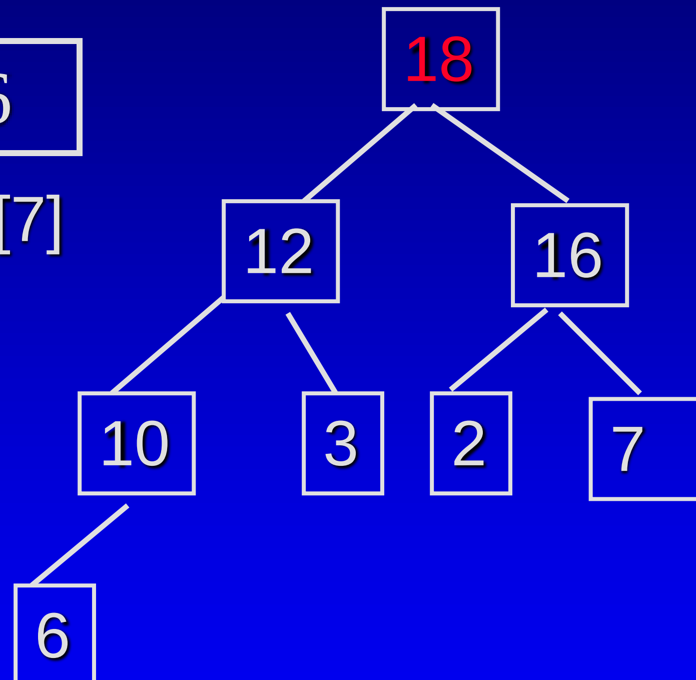
heap ->

sorted list from smallest to largest

Q: where is the largest entry?

Heapsort – Step 2: Sorting from Heap

18	12	16	10	3	2	7	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



Idea: remove the root of the heap and place it in the sorted list

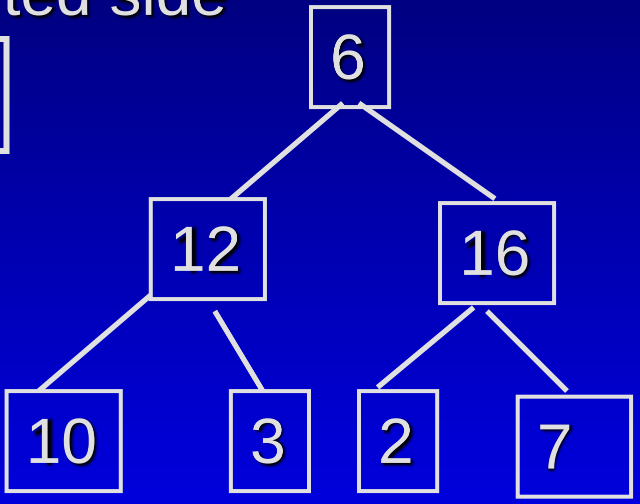
=> recall: how to remove the root?

Heapsort – Step 2: Sorting from Heap

almost a heap...

sorted side

6	12	16	10	3	2	7	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



How to remove the root?

move the last entry in the root...

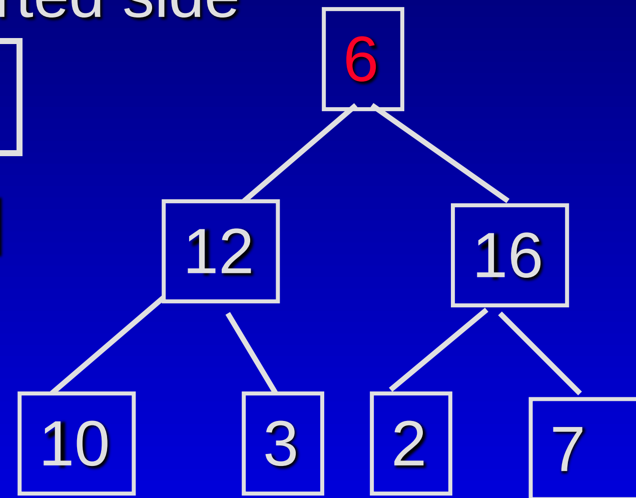
and for the sake of sorting, put the root entry in the "sorted side"

Heapsort – Step 2: Sorting from Heap

almost a heap...

sorted side

6	12	16	10	3	2	7	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



How to remove the root?

move the last entry in the root...

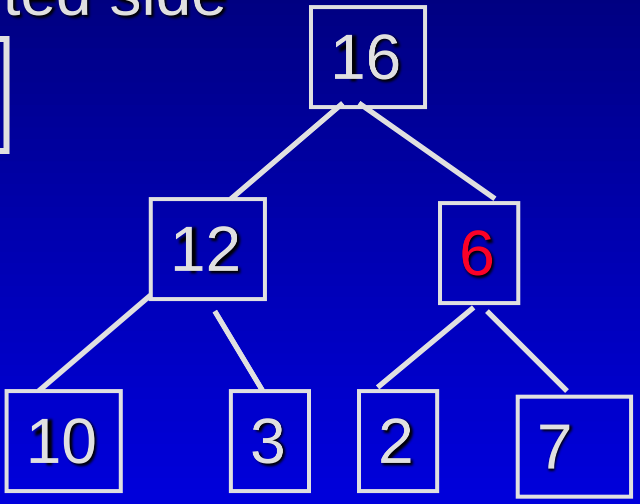
then reposition the out-of place node to update the heap

Heapsort – Step 2: Sorting from Heap

almost a heap...

sorted side

16	12	6	10	3	2	7	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



How to remove the root?

move the last entry in the root...

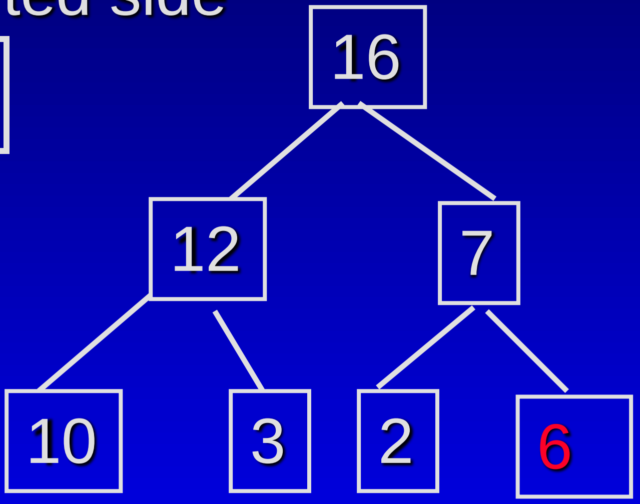
then reposition the out-of place node to update the heap

reheapification downward

Heapsort – Step 2: Sorting from Heap

a heap in the unsorted side | sorted side

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
16	12	7	10	3	2	6	18



How to remove the root?

move the last entry in the root...

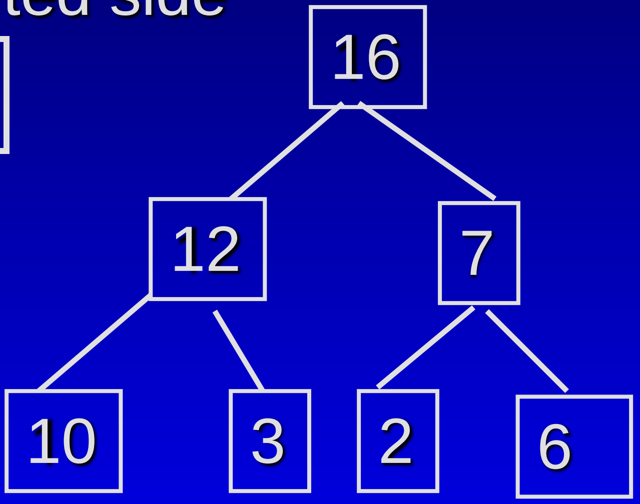
then reposition the out-of place node to update the heap

reheapification downward

Heapsort – Step 2: Sorting from Heap

a heap in the unsorted side | sorted side

16	12	7	10	3	2	6	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

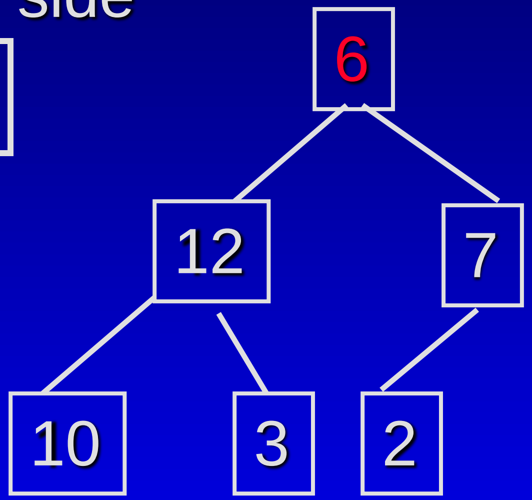


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap... sorted side

6	12	7	10	3	2	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

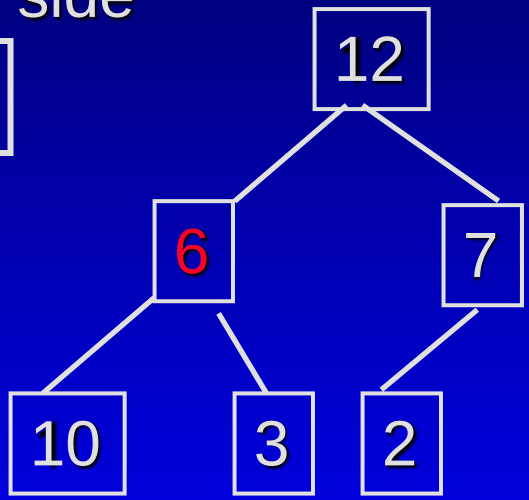


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap... | sorted side

12	6	7	10	3	2	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

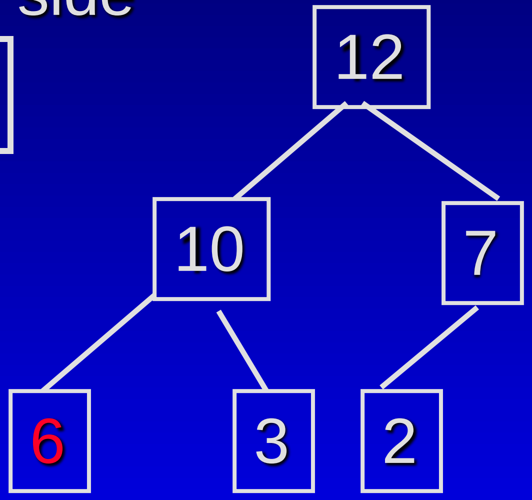


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap... | sorted side

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
12	10	7	6	3	2	16	18

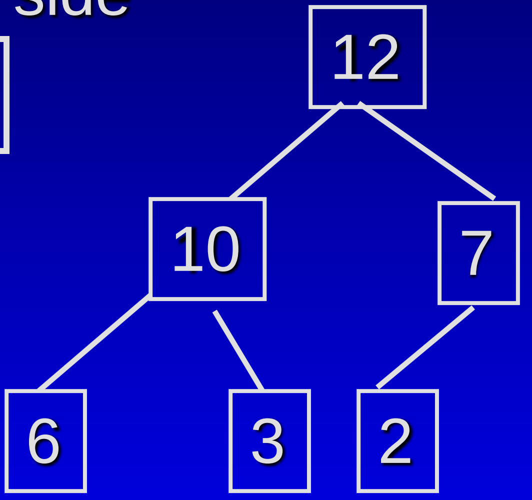


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

a heap again! sorted side

12	10	7	6	3	2	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

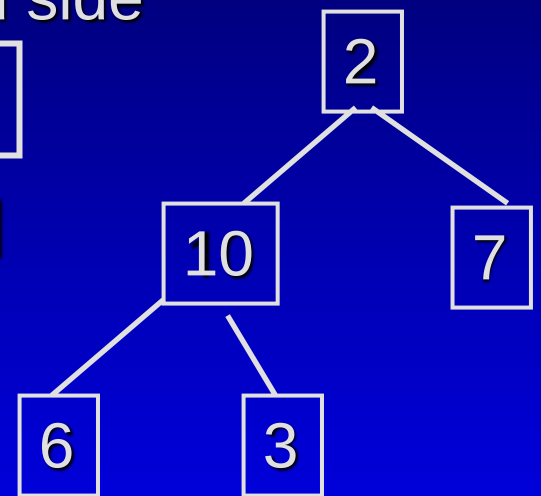


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap...   sorted side

2	10	7	6	3	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

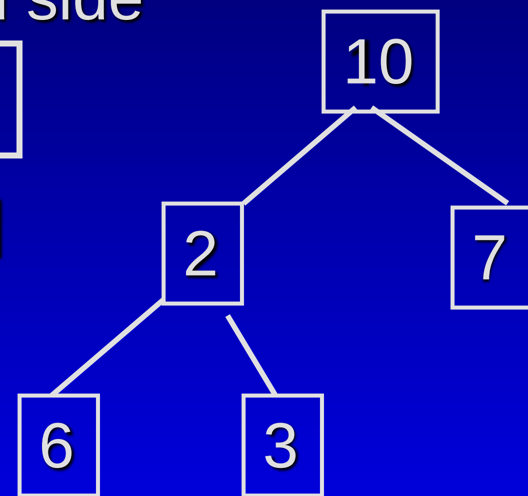


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap... sorted side

10	2	7	6	3	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

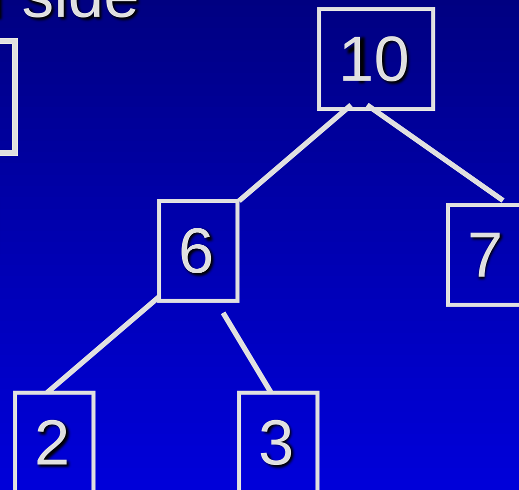


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

a heap again! sorted side

10	6	7	2	3	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

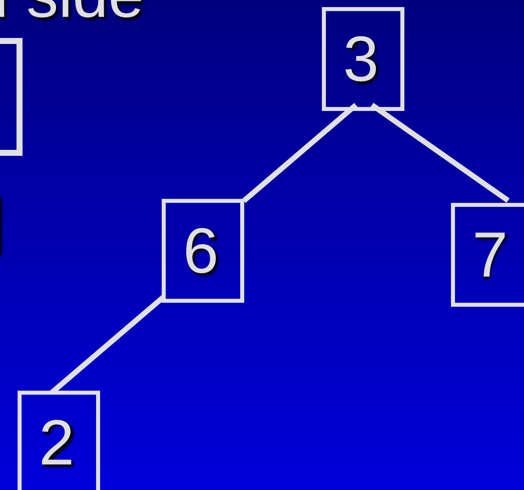


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

almost a heap... sorted side

3	6	7	2	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

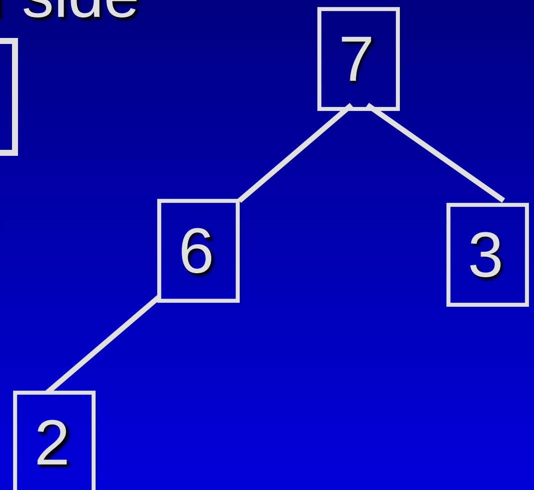


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

a heap again ! sorted side

7	6	3	2	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

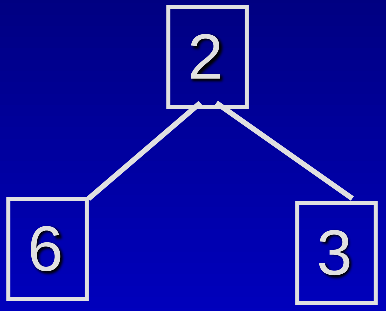


do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

a heap ??　　sorted side

2	6	3	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



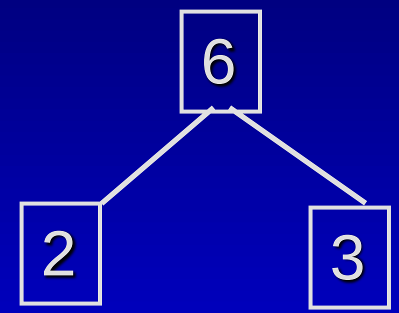
do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

a heap !!

sorted side

6	2	3	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



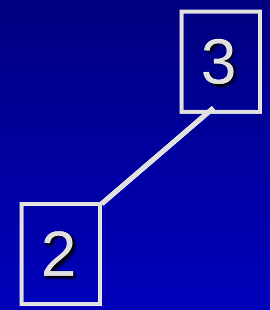
do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

heap !

sorted side

3	2	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

heap ! sorted side

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

$\boxed{2}$

do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

Heapsort – Step 2: Sorting from Heap

sorted side

2	3	6	7	10	12	16	18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

do the same thing again for the heap in the unsorted side until all the entries have been moved to the sorted side

DONE!

Heapsort – Time Analysis

- Step 1. Make a heap from elements
  - add an entry to the heap one at a time
  - reheapification upward n times – O(n log n)
- Step 2. Make a sorted list from the heap
  - Remove the root of the heap to a sorted list and
  - Reheapification downward to re-organize the unsorted side into an updated heap
  - do this n times – O(n log n)
- The running time is O(n log n)

C++ STL Sorting Functions

- The C++ sort function
  - void sort(Iterator begin, Iterator end);
- The original C version of qsort

```
void qsort(
    void *base,
    size_t number_of_elements,
    size_t element_size,
    int compare(const void*, const void*)
);
```

Summary & Homework

- Recursive Sorting Algorithms
  - Divide and Conquer technique
- An O(NlogN) Sorting Algorithm using a Heap
  - making use of the heap properties
- STL Sorting Functions
  - C++ sort function
  - Original C version of qsort
- Homework
  - use your heap implementation to implement a heapsort!