CPSC 221
Basic Algorithms and Data Structures

June 10, 2015

Administrative stuff

Theory Assignment 1 has been marked and feedback has been sent.

Theory Assignment 2 has been posted.

Programming Assignment 2 will be posted soon.

Exam progress is incremental.

Heapsort revisited

We got a lot of questions about heapify and heapsort. Let’s review, then see another example and another version of heapify.

First the review of Monday’s heapsort slides...

Heapsort

We can use a heap as the foundation for a very efficient sorting algorithm called heapsort.

Heapsort consists of two phases:

Heapify: build a heap using the elements to be sorted

Sort: Use the heap to sort the data

This can all be done in place in the array that holds the heap, but it's easier to see if we draw the trees instead of the array. Once you see how it works with trees, make sure you understand how it works in place in the array.

Heapsort

Here’s the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:

5	2	1	4	3

Implementing a heap

6	14	12	28	18	17	33	41	52	47	19	22	__	__	__
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

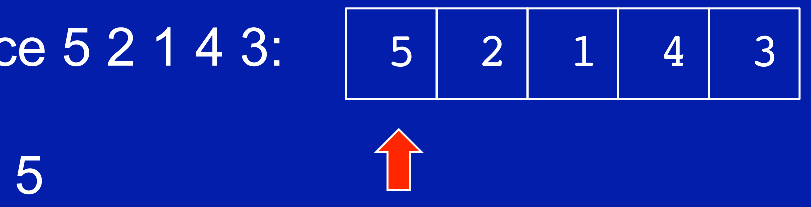
we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heapsort

Here’s the heapify component:

```
for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:

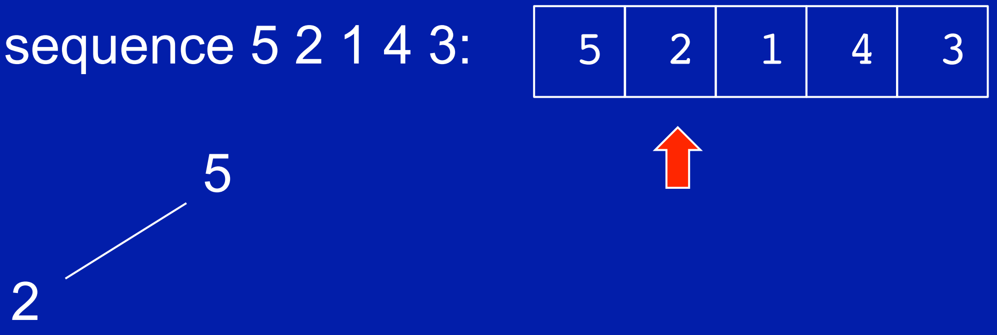


Heapsort

Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:

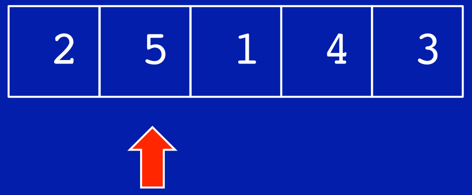


Heapsort

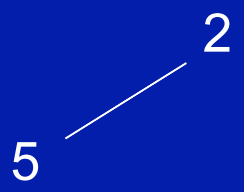
Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



2	5	1	4	3

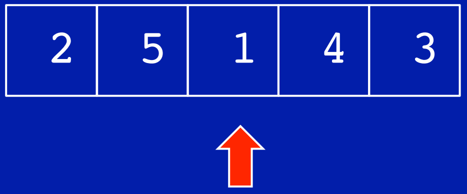


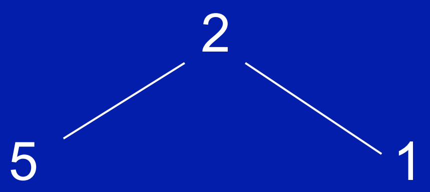
Heapsort

Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



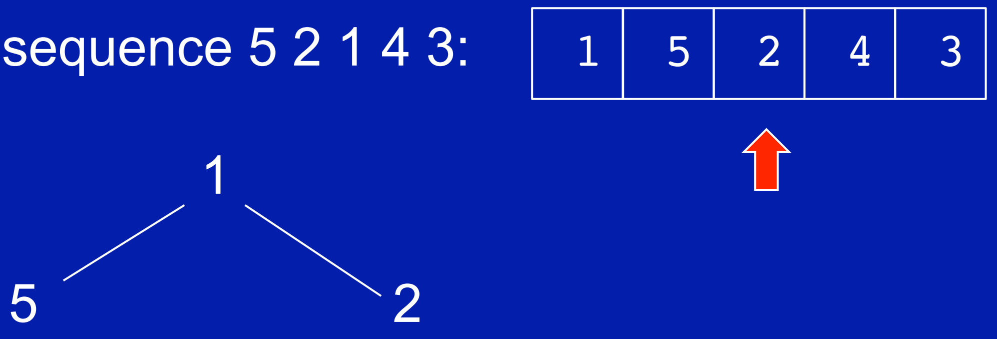


Heapsort

Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:

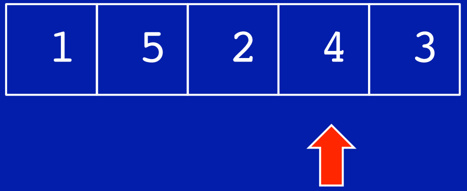


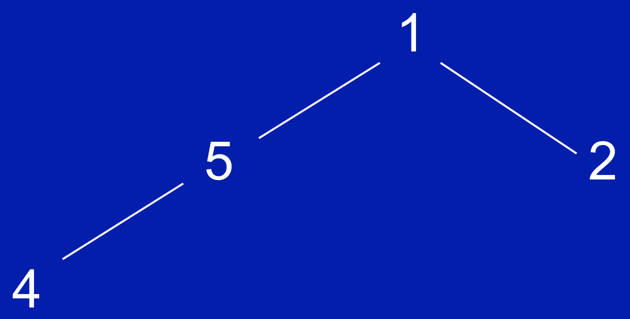
Heapsort

Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



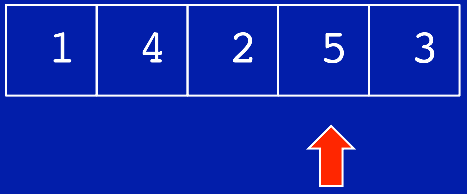


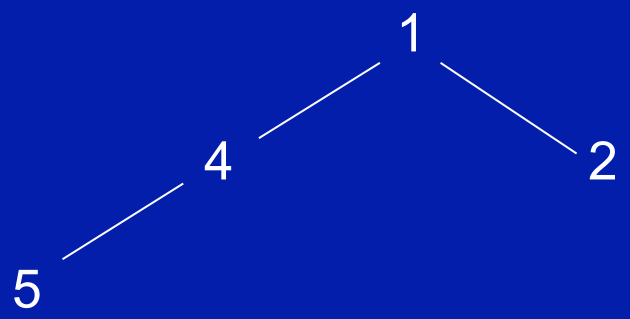
Heapsort

Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



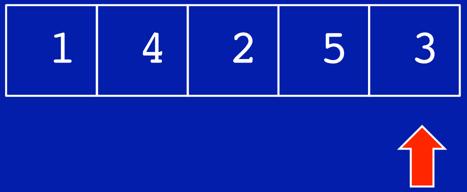


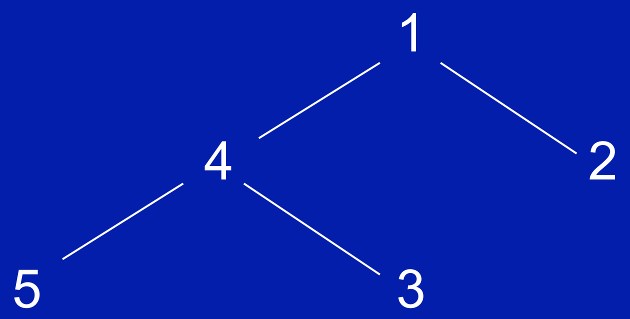
Heapsort

Here’s the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



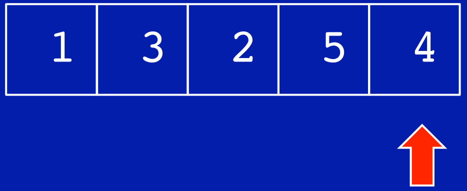


Heapsort

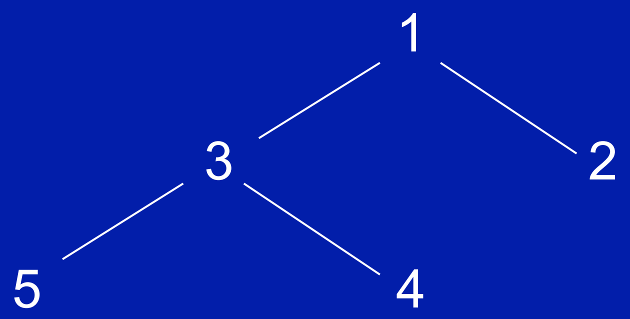
Here's the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:



1	3	2	5	4



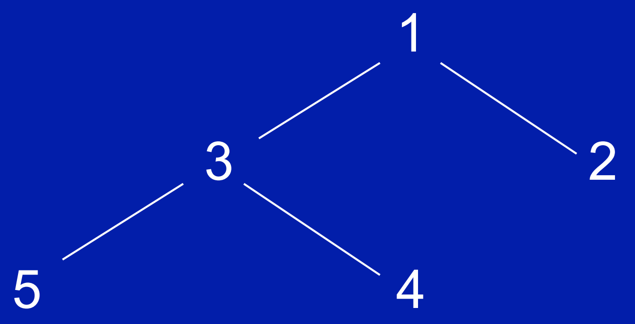
Heapsort

Here’s the heapify component:

```
for each item in the sequence to be sorted
        add the item to the next available position in the
                complete binary tree
        restore the heap property (using ReheapUp)
```

Say we want to sort the sequence 5 2 1 4 3:

1	3	2	5	4



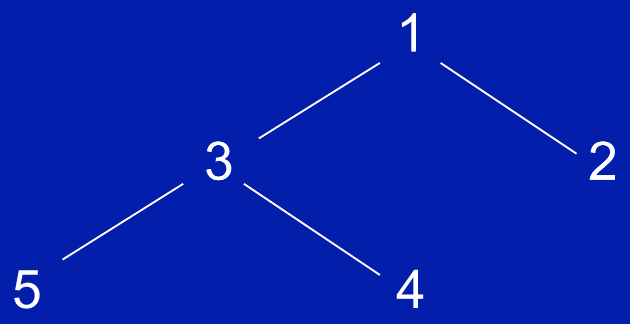
The sequence is heapified

Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

1	3	2	5	4

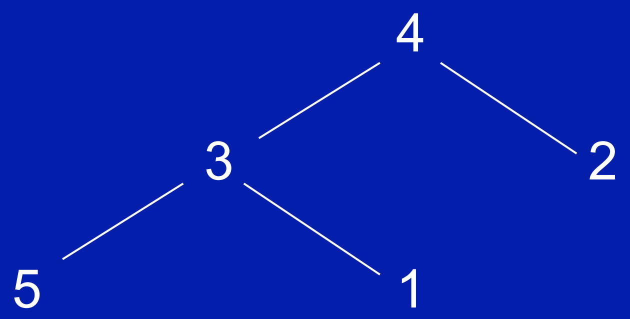


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

4	3	2	5	1

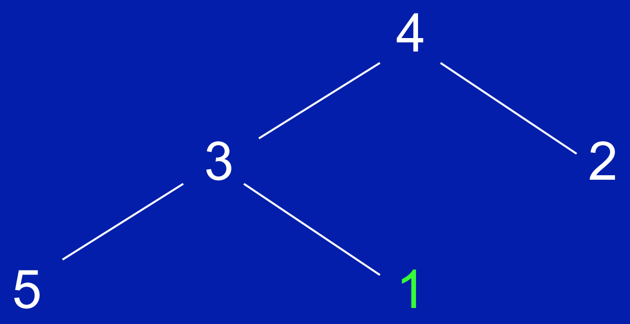


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

4	3	2	5	1

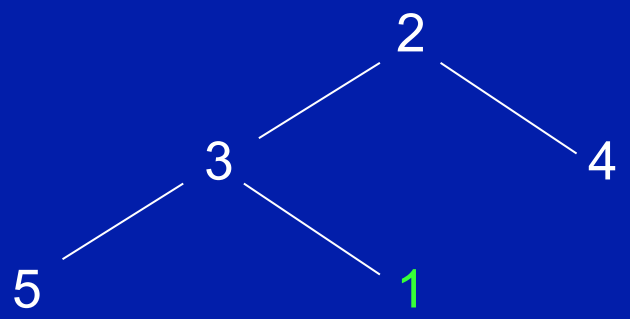


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

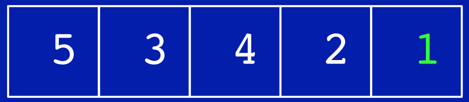
2	3	4	5	1



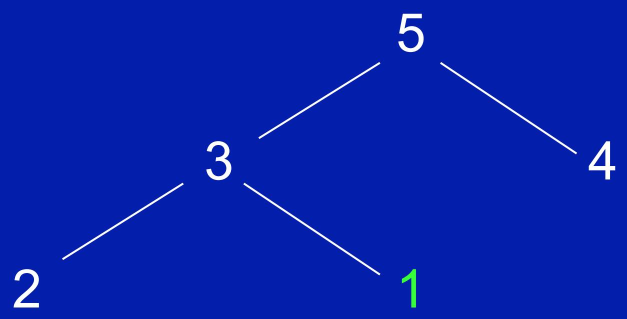
Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



5	3	4	2	1

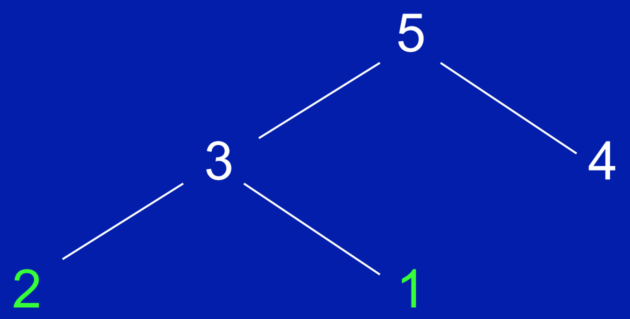


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

5	3	4	2	1

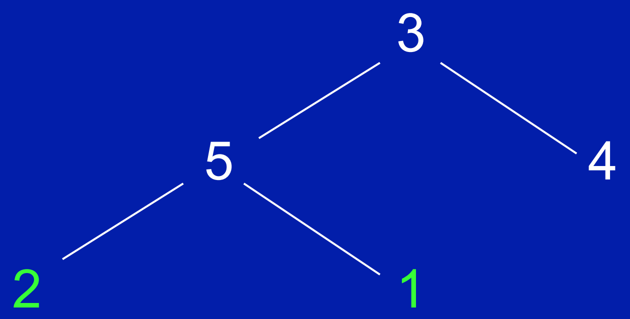


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

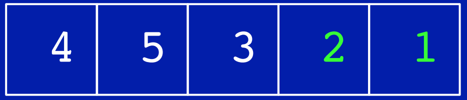
3	5	4	2	1



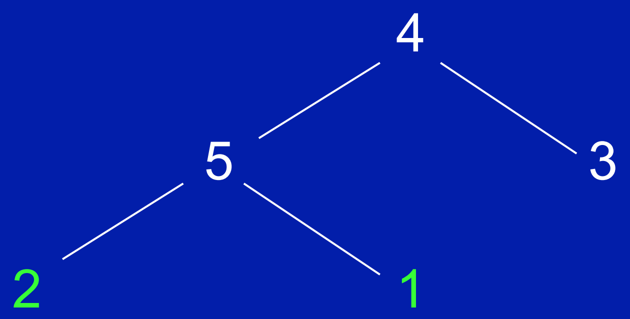
Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



4	5	3	2	1

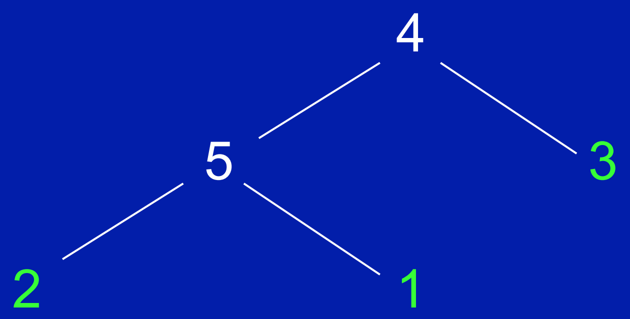


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

4	5	3	2	1

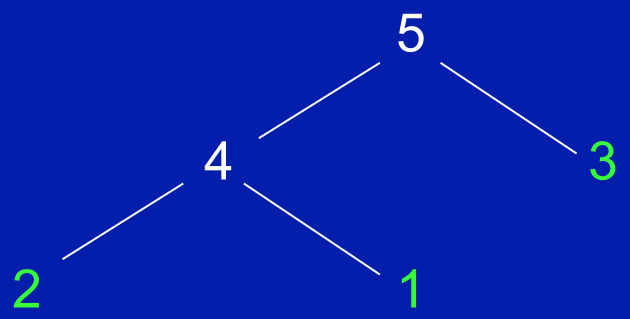


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

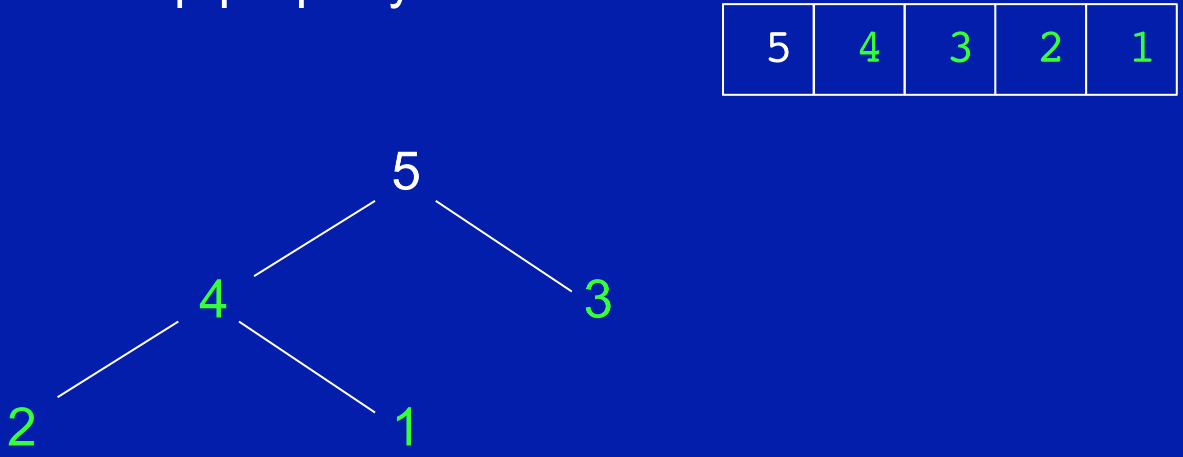
5	4	3	2	1



Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

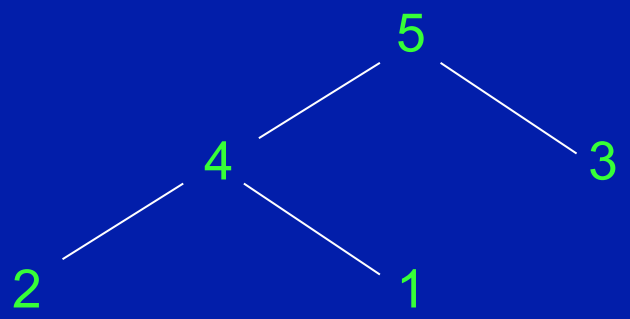


Heapsort

Now for the sorting:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```

5	4	3	2	1

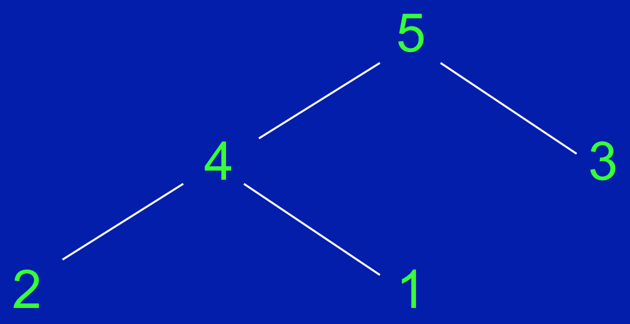


Heapsort

Now it’s sorted, but it’s largest to smallest (reading top to bottom, left to right).

We created a min heap, and we ended up with the sequence sorted from largest to smallest. The same thing will happen when sorting in place in the array.

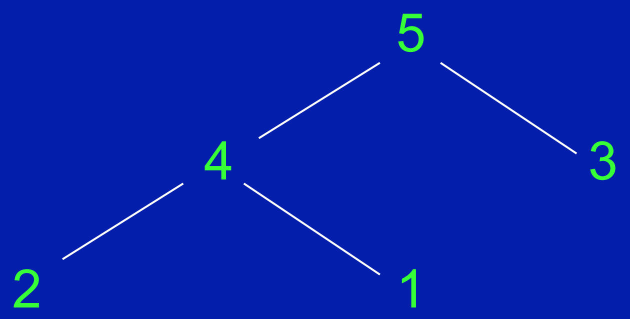
5	4	3	2	1



Heapsort

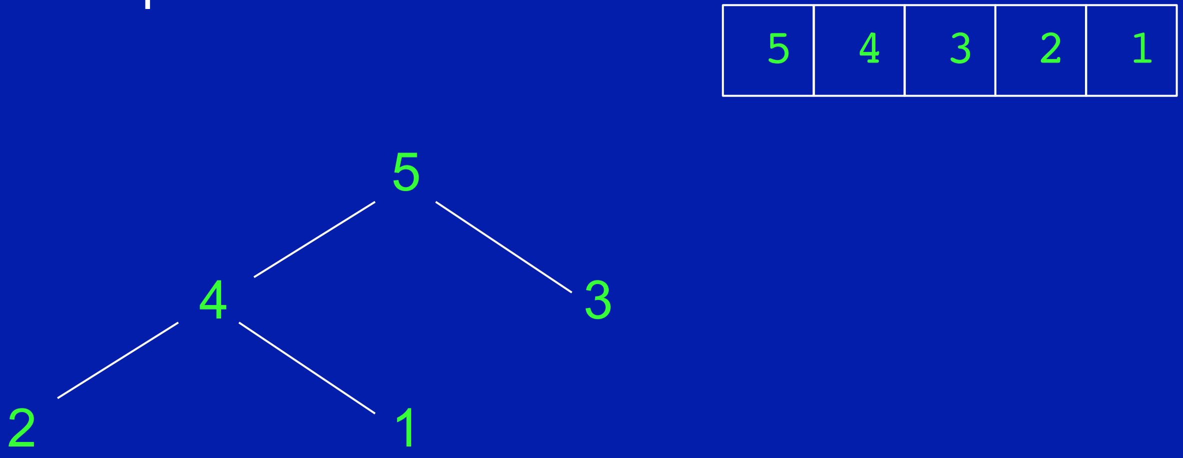
In the array-based heapsort example in the book (which you should study), the original sequence was heapified into a max heap, and the resulting sorted sequence in the array went from smallest to largest. Just something to keep in mind.

5	4	3	2	1



Heapsort analysis

A heap of size n has lg n levels. Building a heap requires finding the correct location for each item in a heap with lg n levels. Because there are n items to insert in the heap and each insert is O(lg n), the time to heapify the original unsorted sequence using ReheapUp or Sift-up is O(n lg n). During sorting, we have n items to remove from the heap, which then is also O(n lg n). Because we can do it all in the orginal array, no extra storage is required.



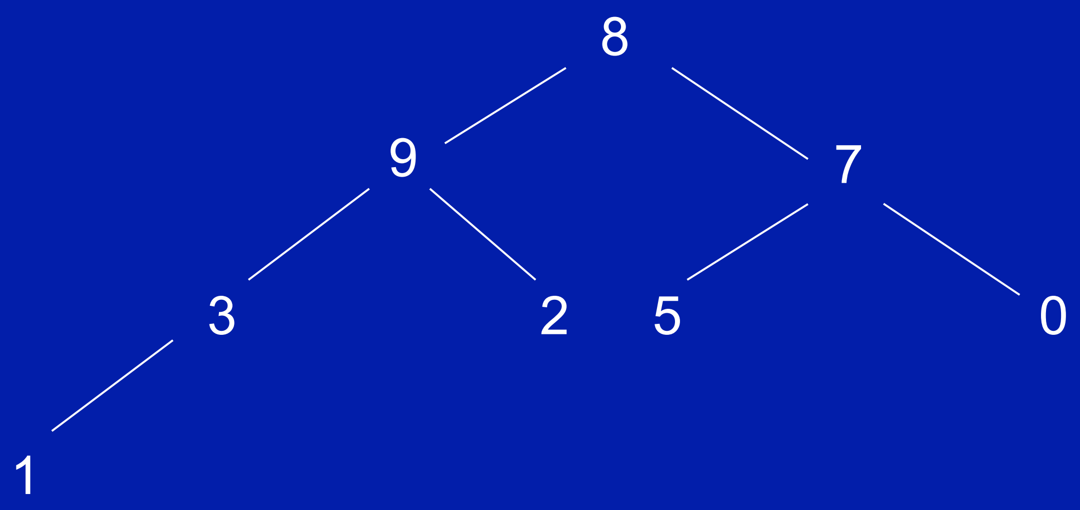
Heapsort redux

The heapify algorithm is the one used in your book (see p. 601). It works fine. It uses repeated insertion into the heap, with the ReheapUp operation applied to every inserted item. In this case, heapify is O(n lg n).

Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

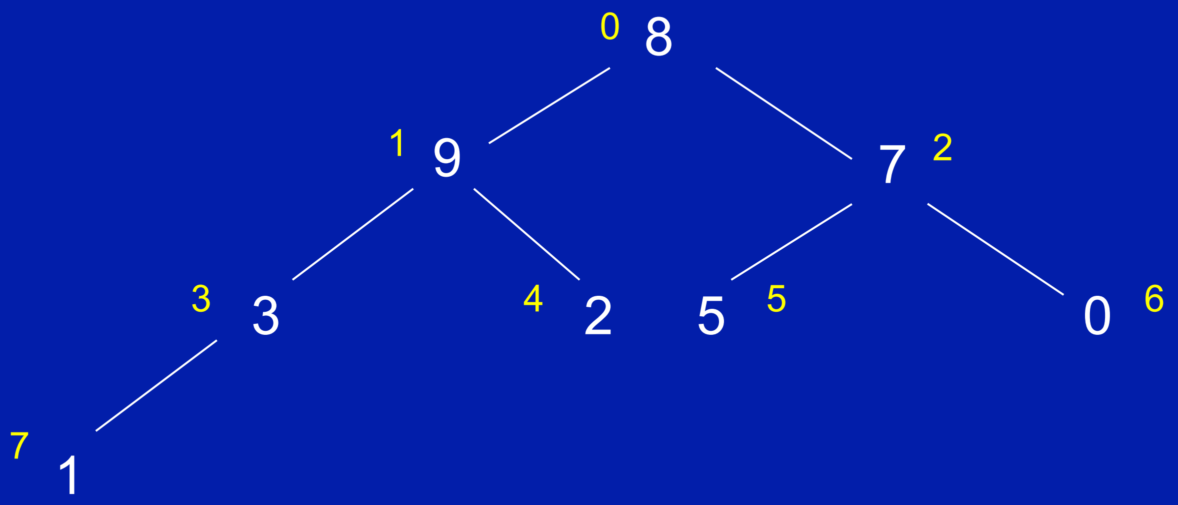
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```

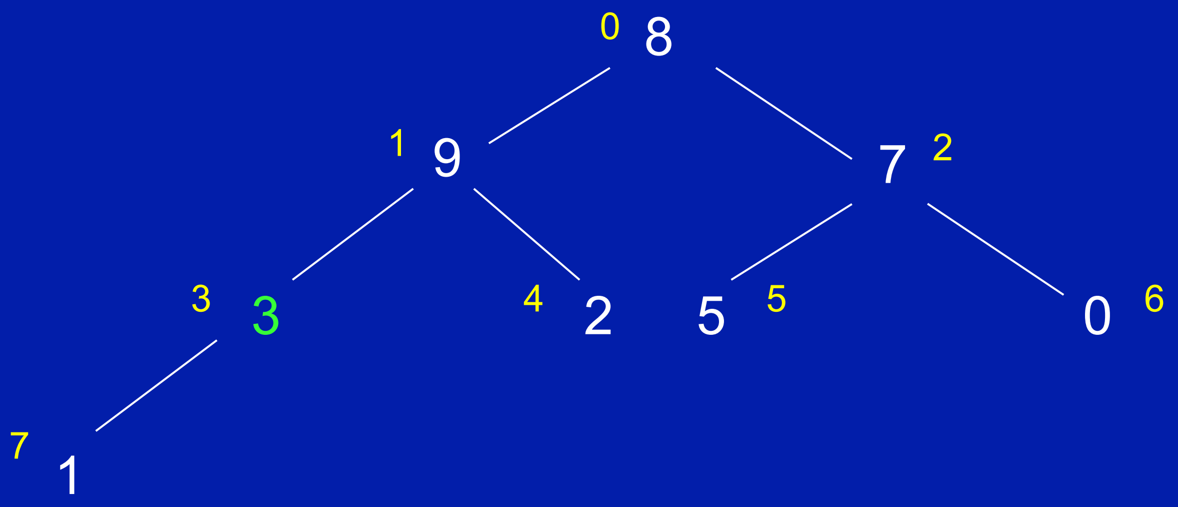


Here are the indexes. Remember that these are the same as the array indexes - heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

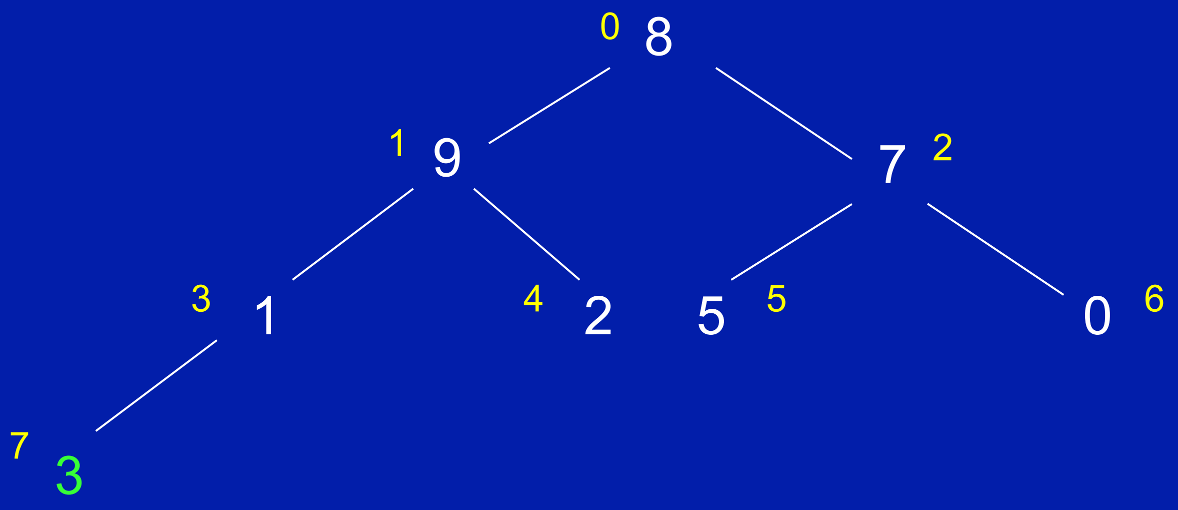
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

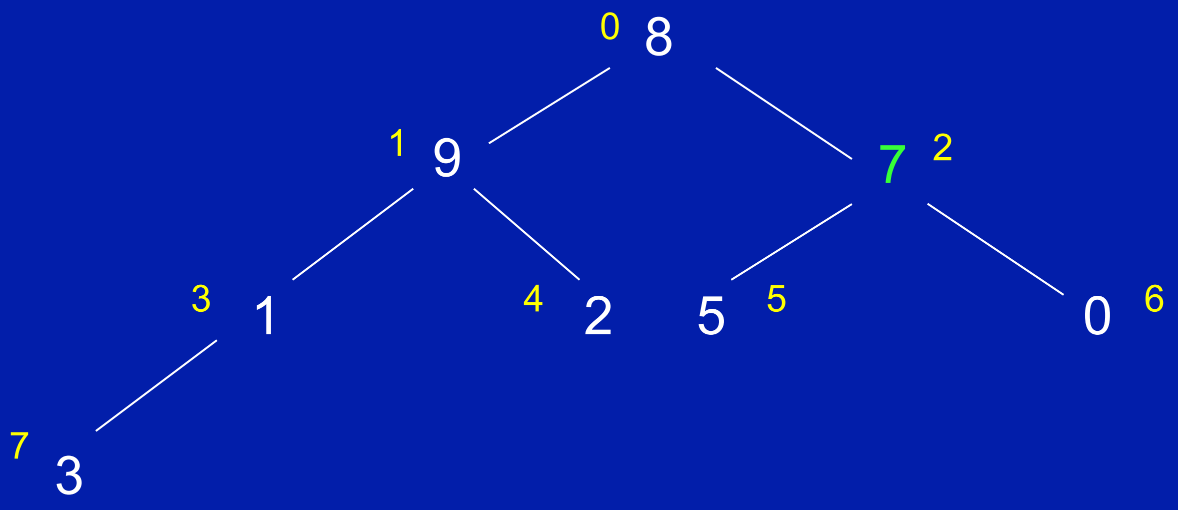
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

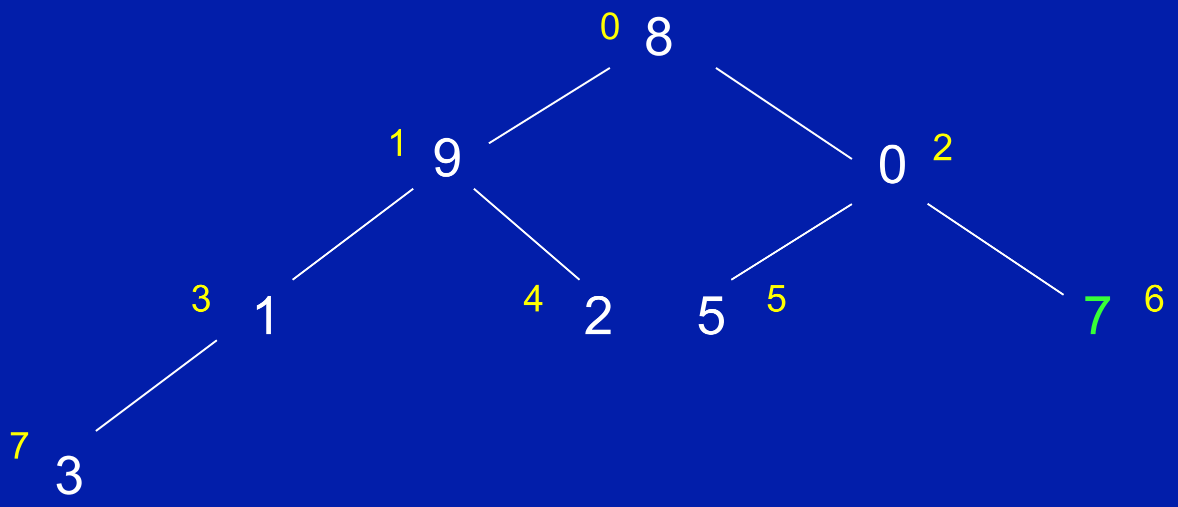
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here’s a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

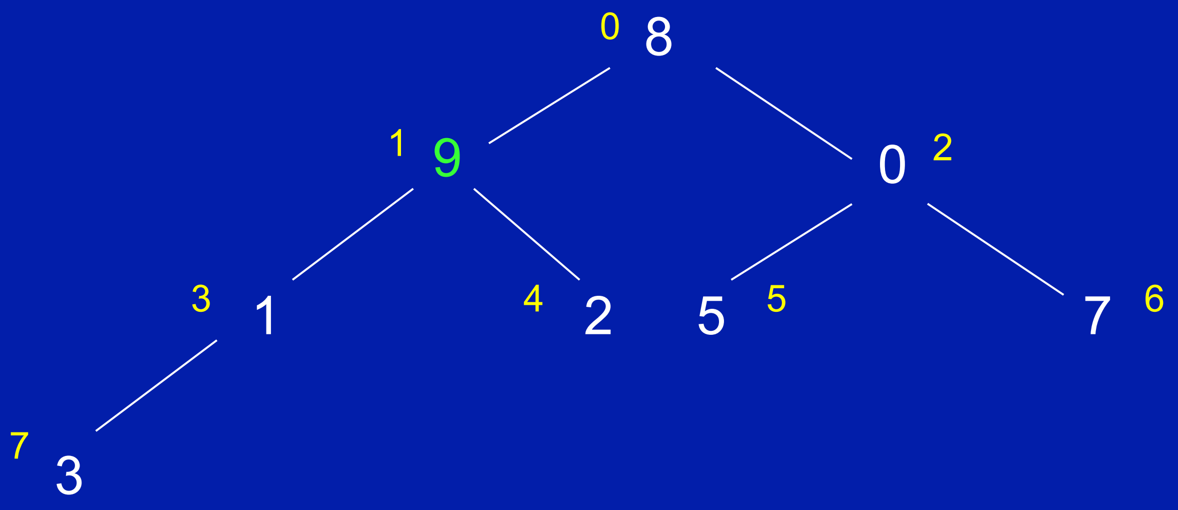
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

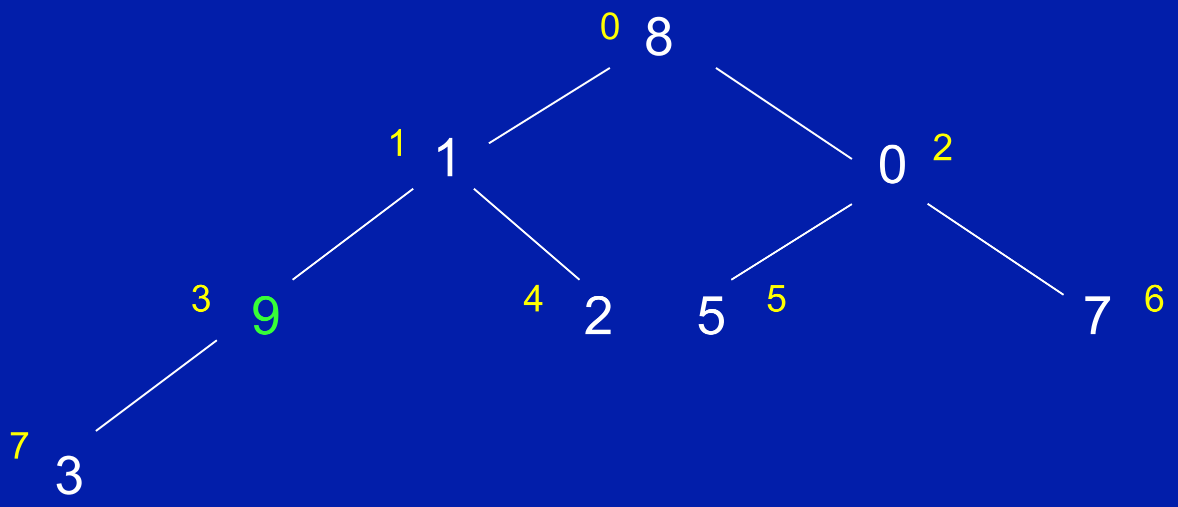
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

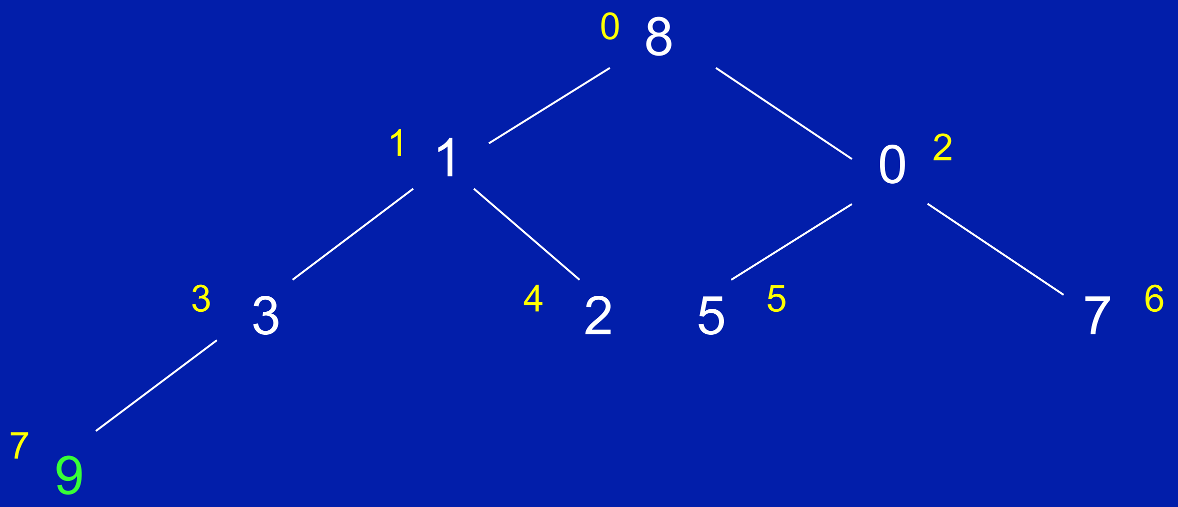
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here’s a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

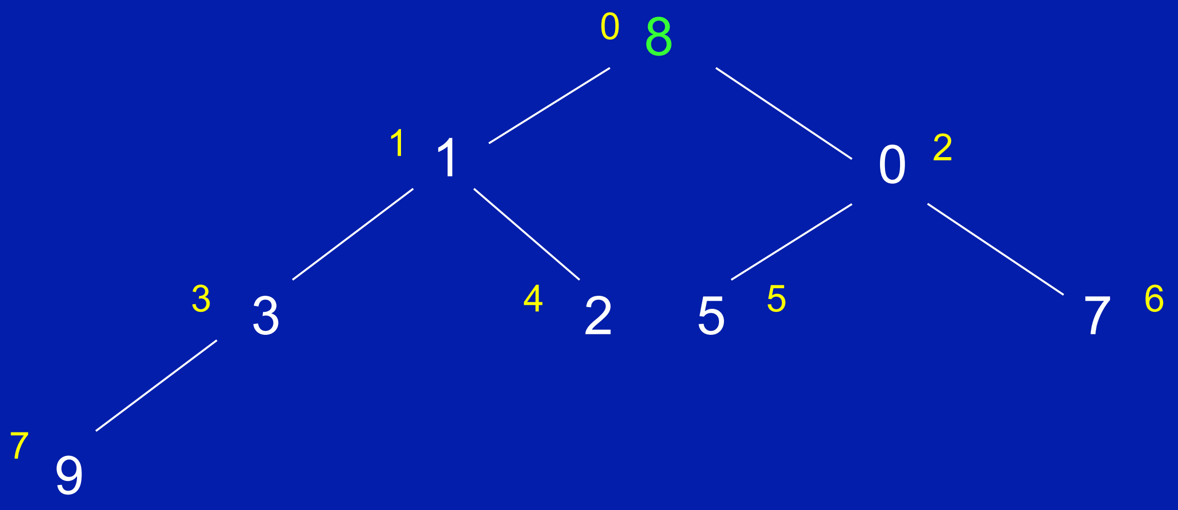
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

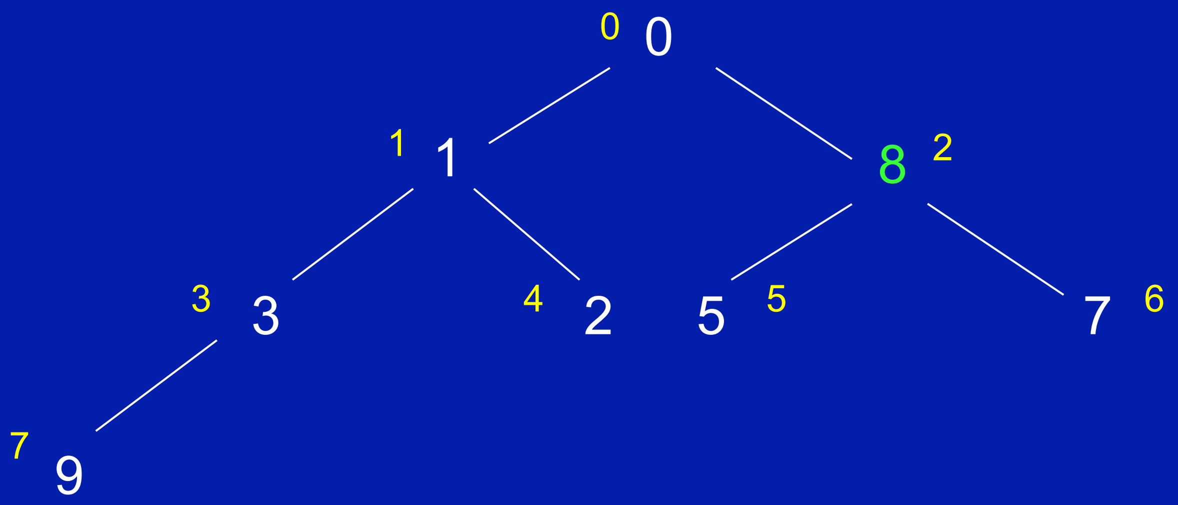
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

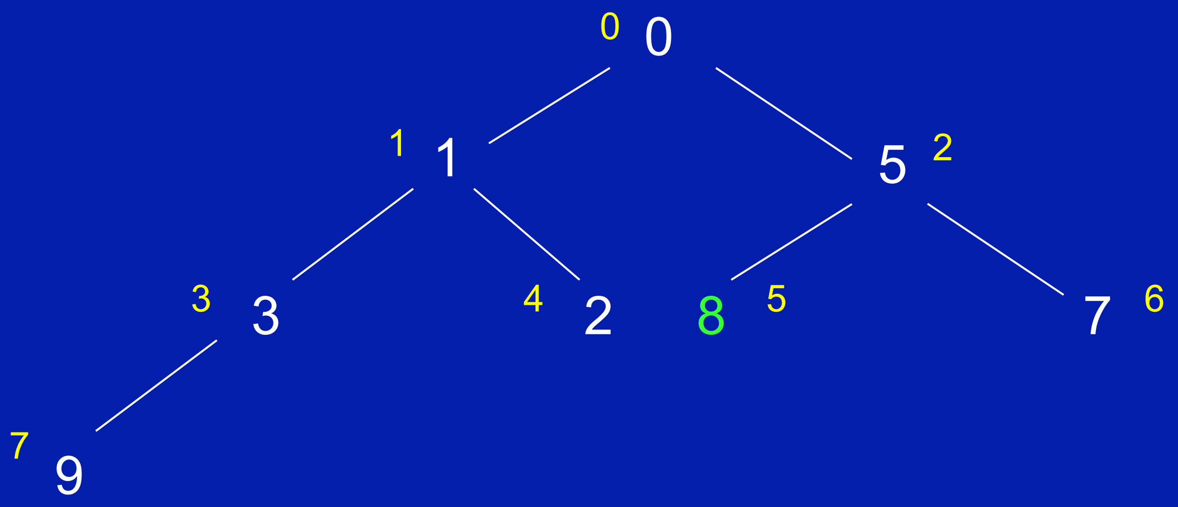
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here’s a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

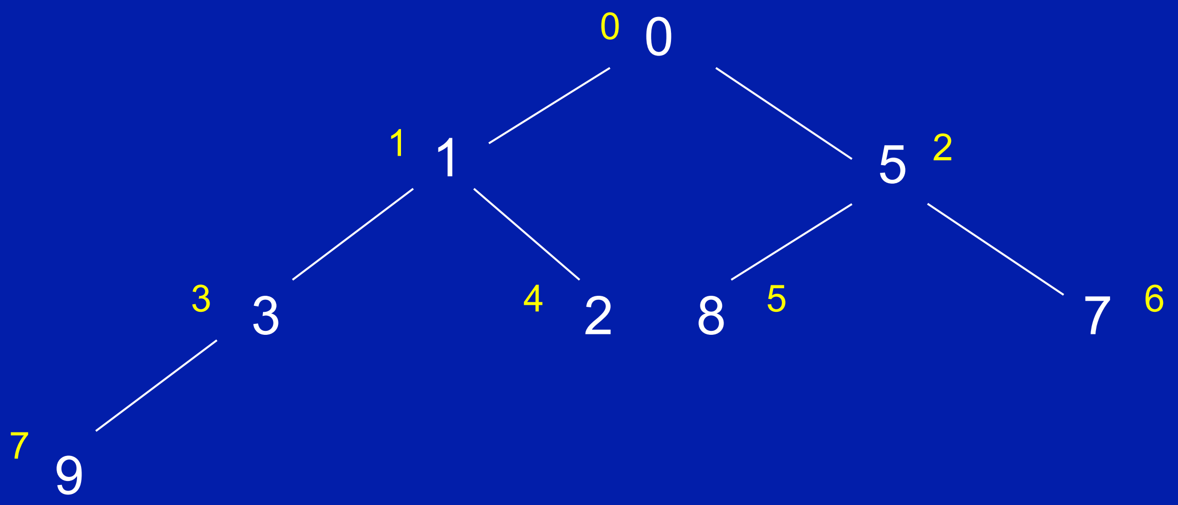
```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia?). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

```
Let index be the subscript of the last parent node in the tree.
While index >= 0
  Perform a ReheapDown operation starting with the node at index
  Decrement index by 1
```



Again, our example is a min heap. So ReheapDown sends the larger numbers down. If we wanted a max heap, we would tweak ReheapDown to send the smaller numbers down.

Heap implementation

8	9	7	3	2	5	0	1
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

8	9	7	1	2	5	0	3
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

8	9	0	1	2	5	7	3
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

8	1	0	9	2	5	7	3
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

8	1	0	3	2	5	7	9
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

0	1	8	3	2	5	7	9
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

0	1	5	3	2	8	7	9
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k – 1) / 2)$

Heap implementation

0	1	5	3	2	8	7	9
0	1	2	3	4	5	6	7

`Done!`

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

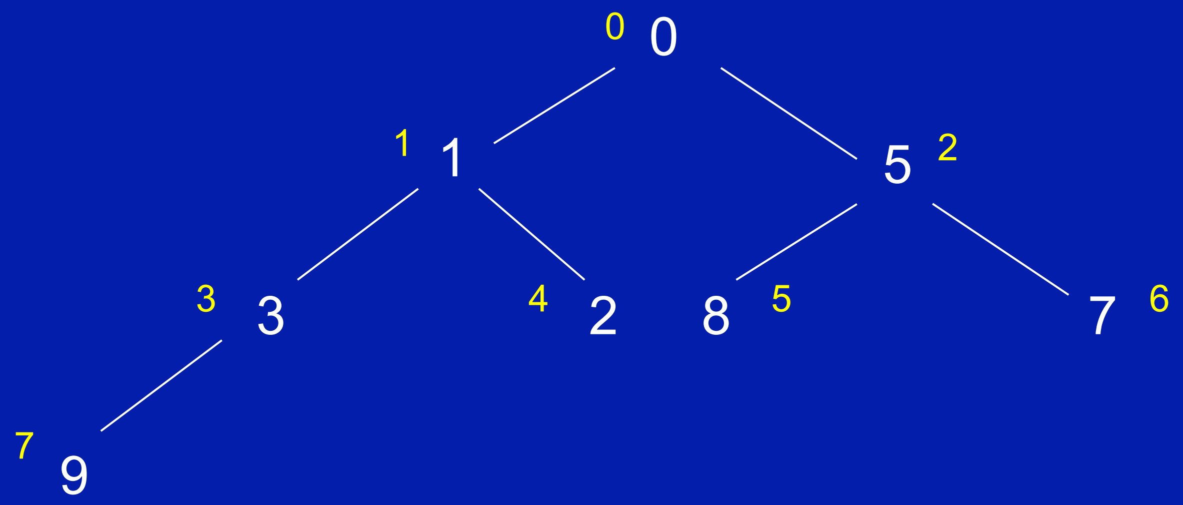
we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heapsort redux

Let’s sort this one now:

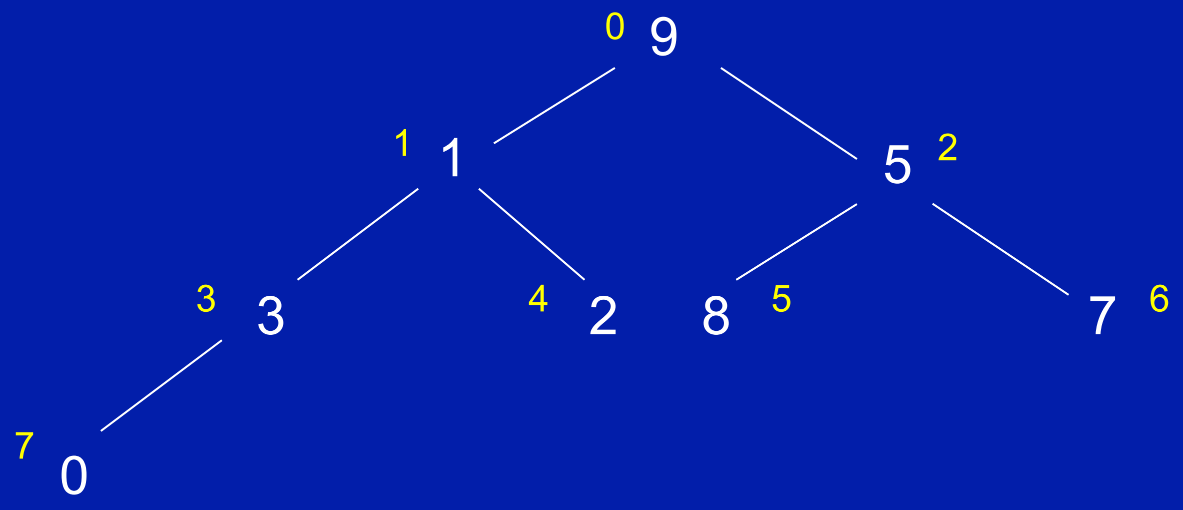
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

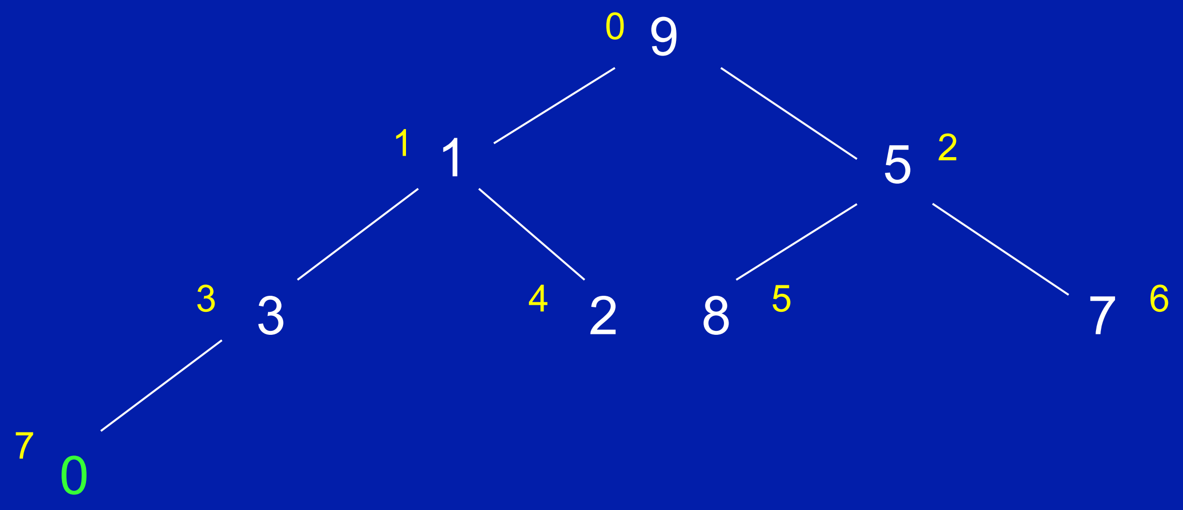
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

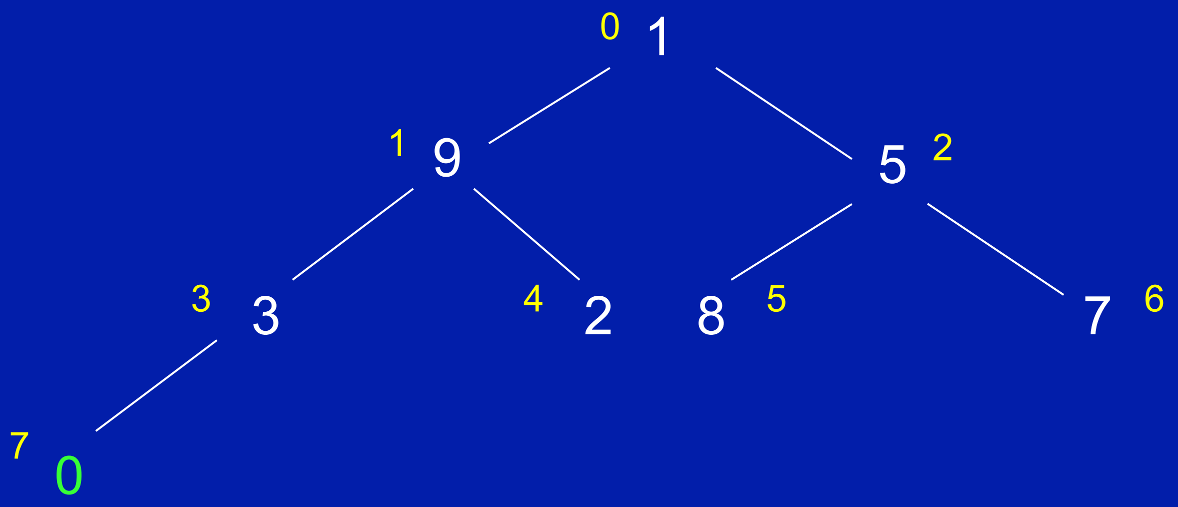
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

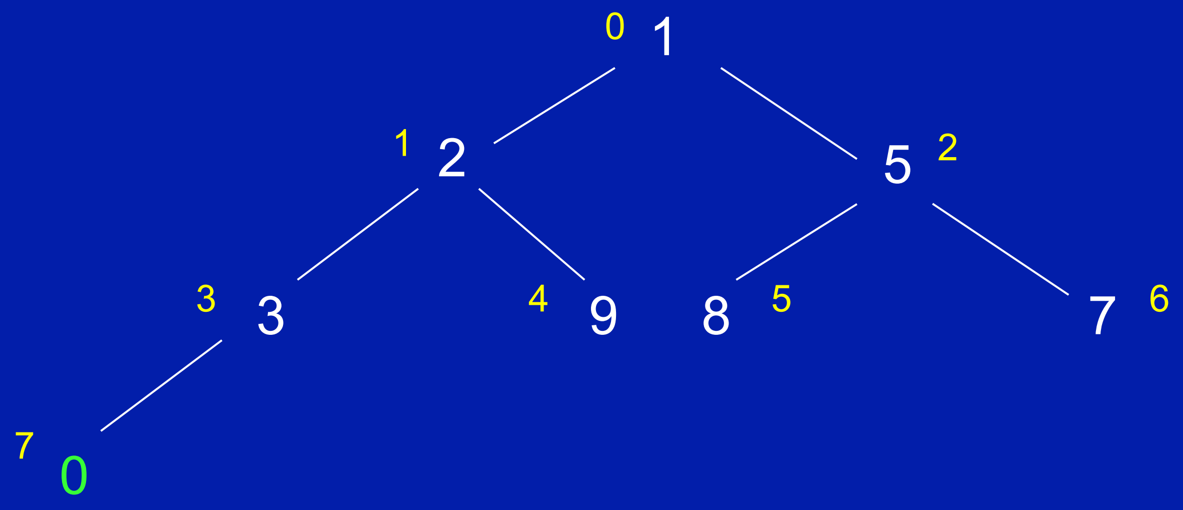
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

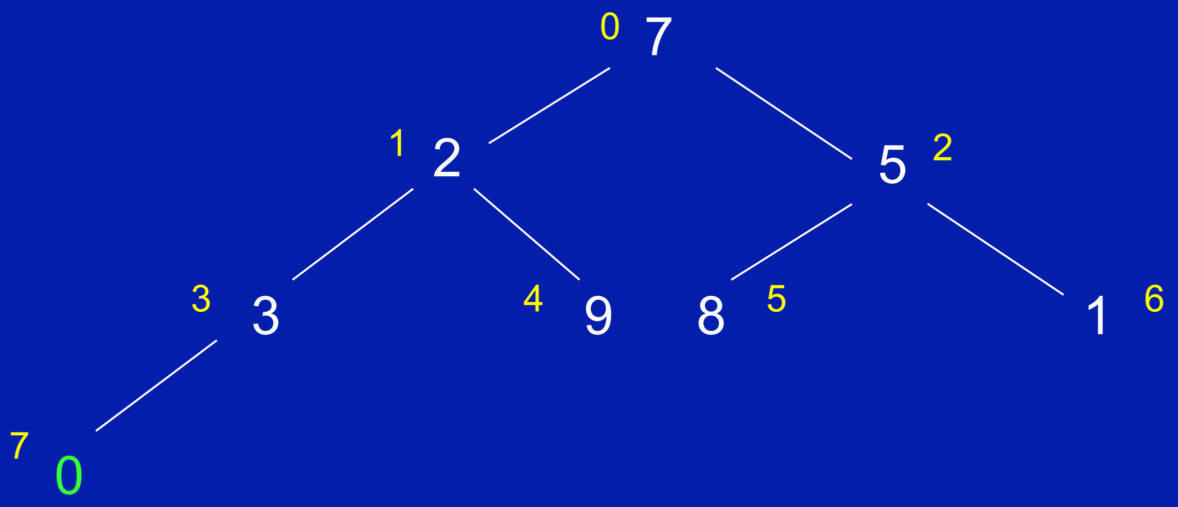
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

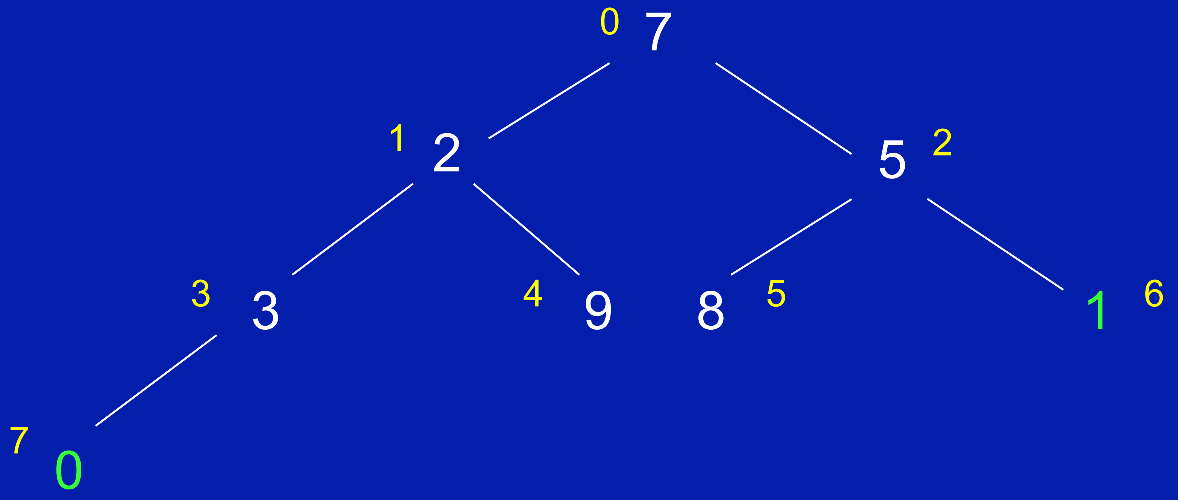
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

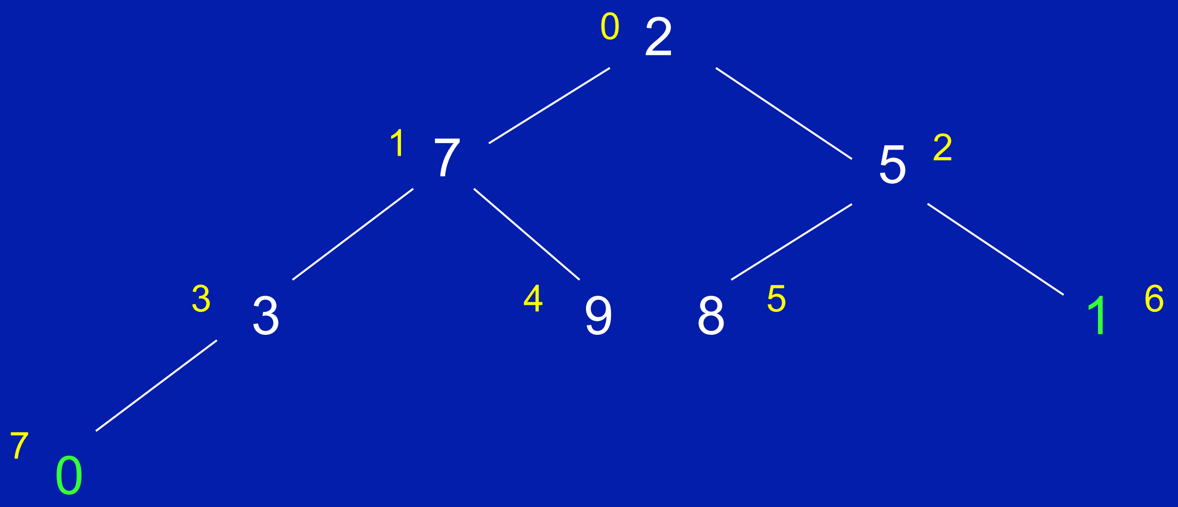
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

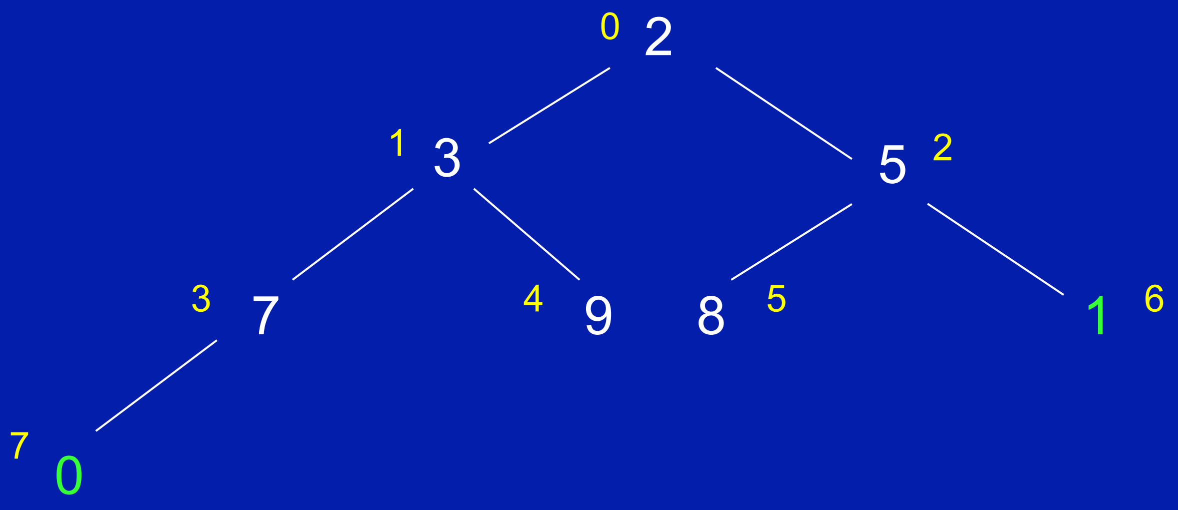
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

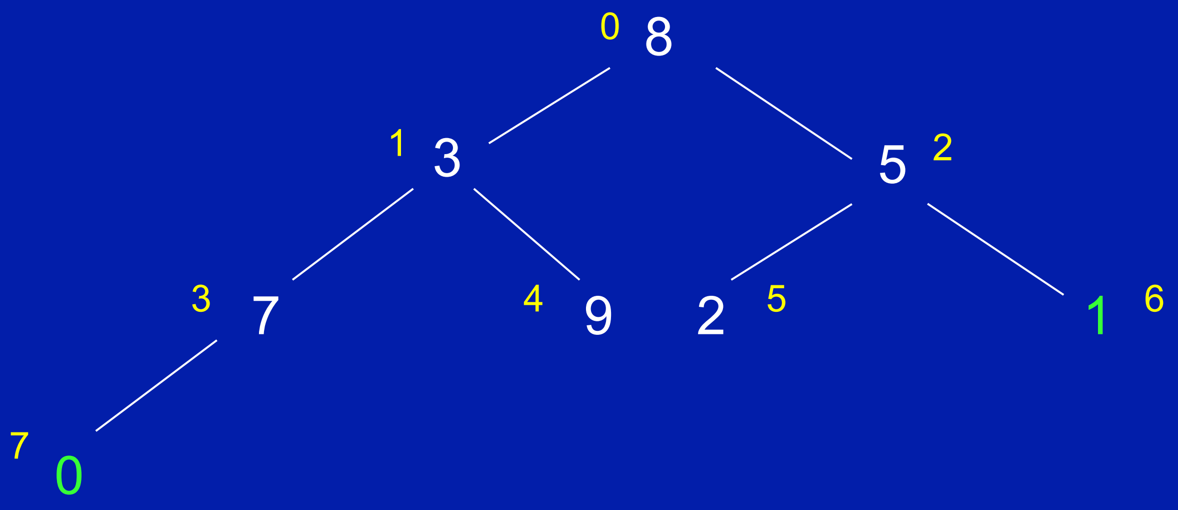
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

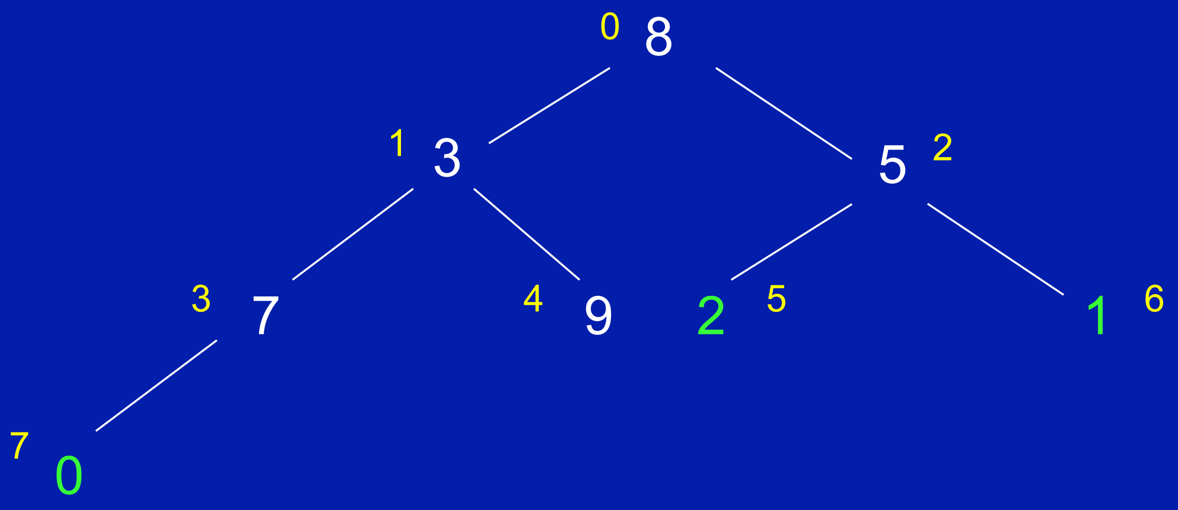
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

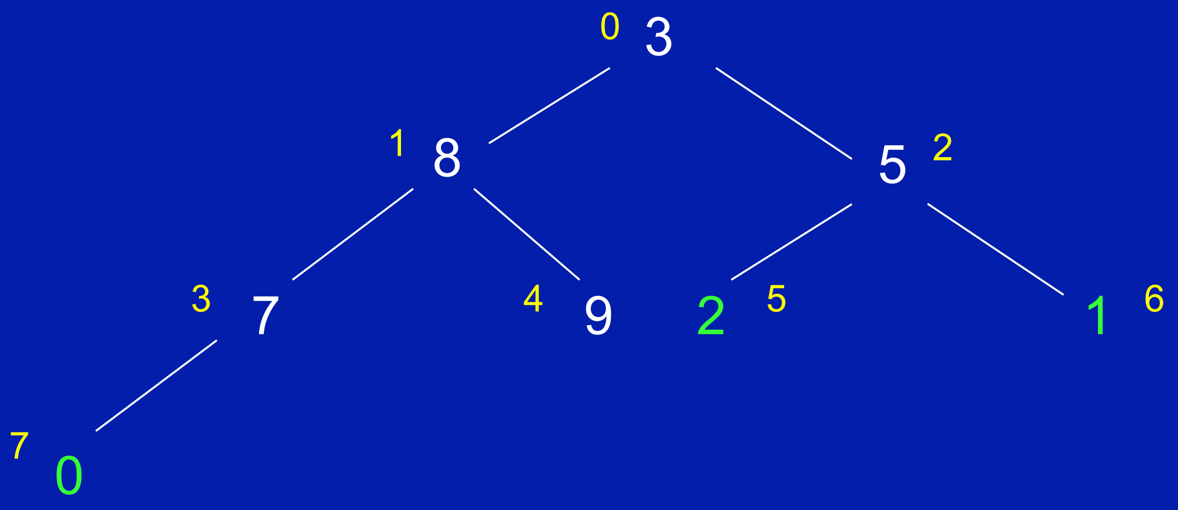
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

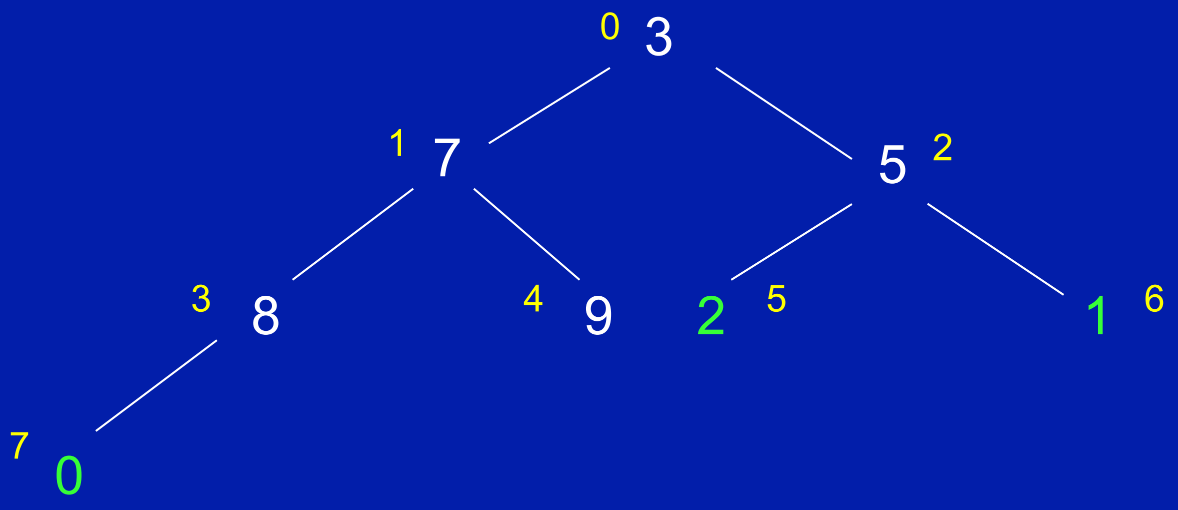
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

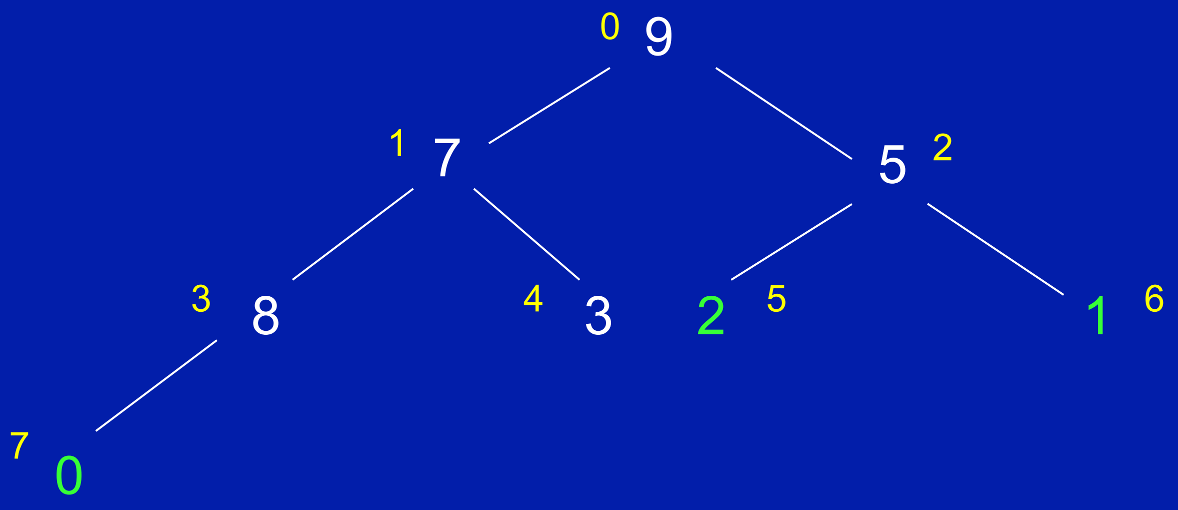
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

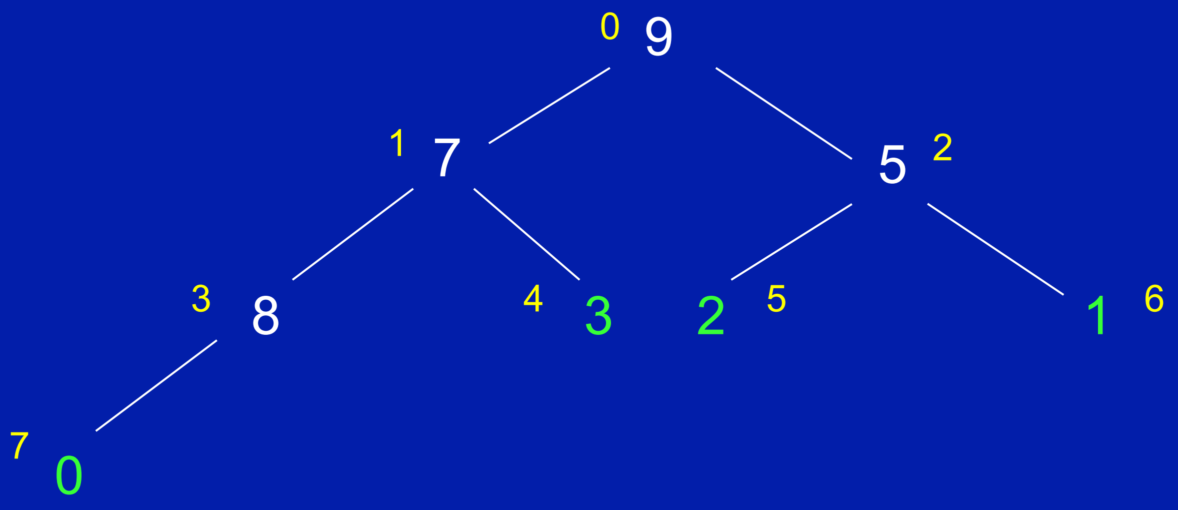
```
while the heap is not empty
          remove the first item from the heap by swapping it
                    with the last item in the heap
          reduce the size of the heap by one
          restore the heap property
```



Heapsort redux

Let’s sort this one now:

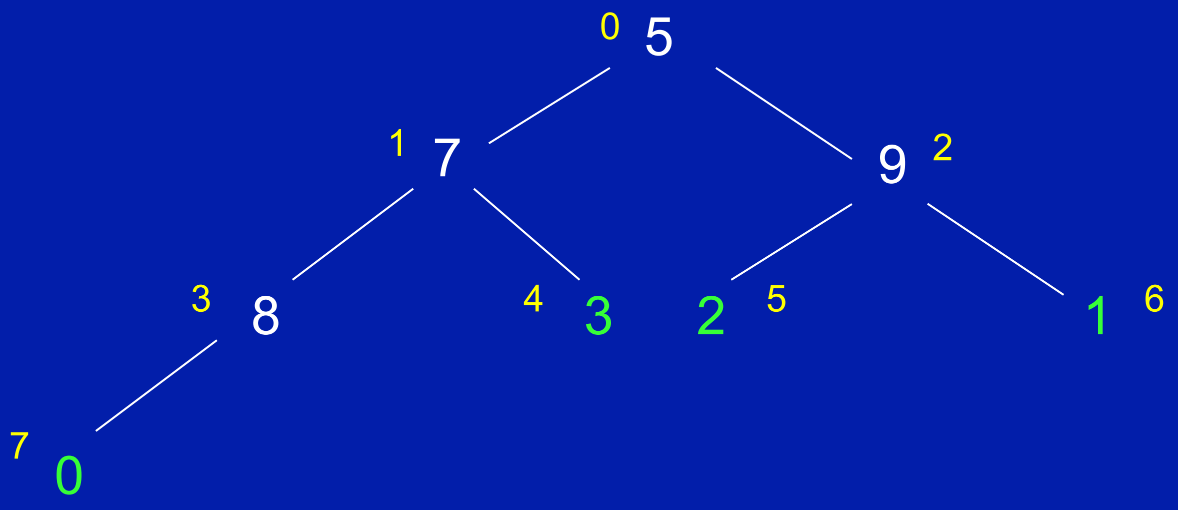
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

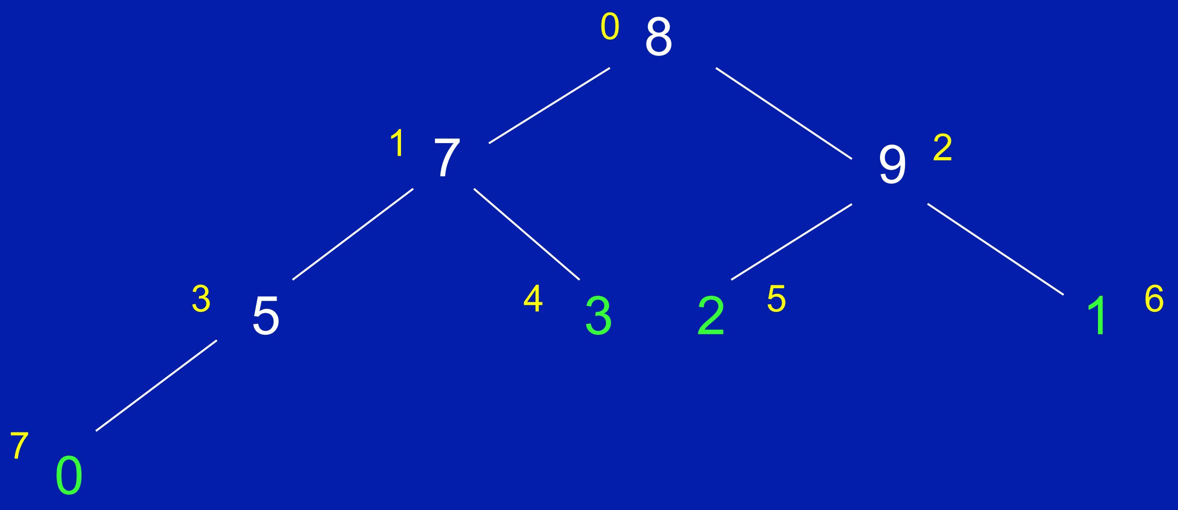
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

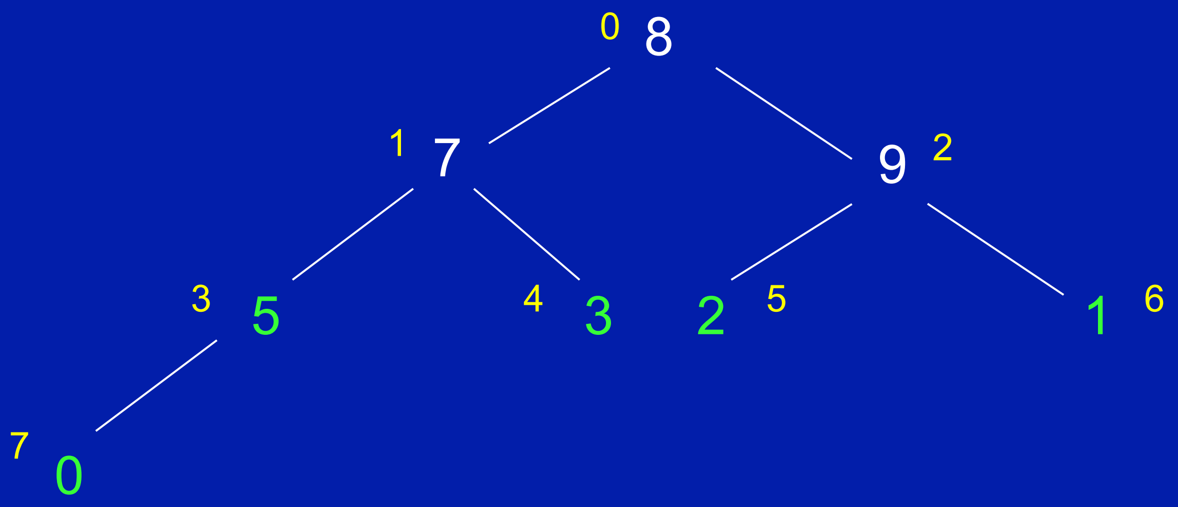
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

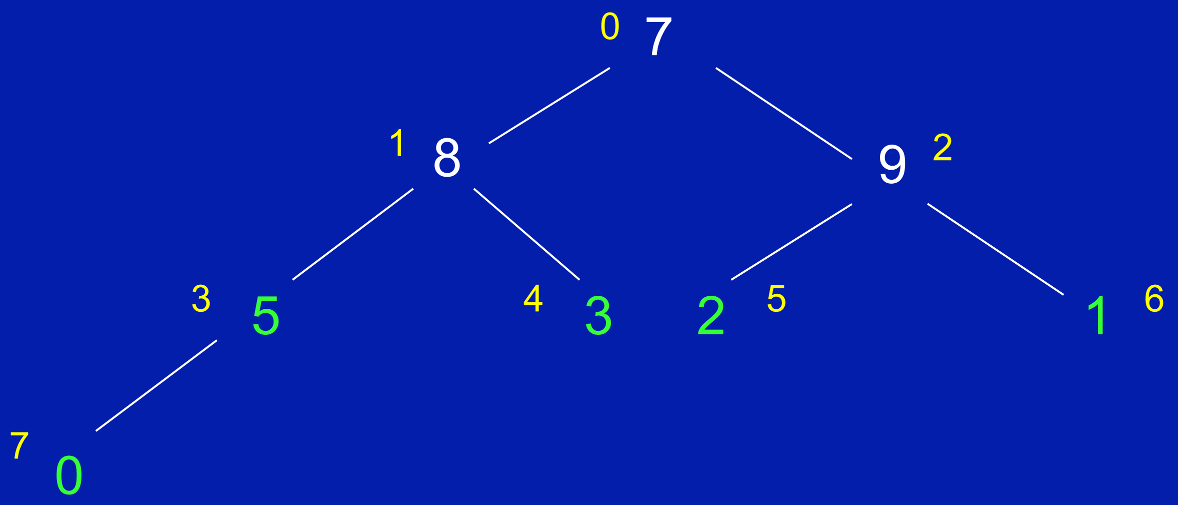
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

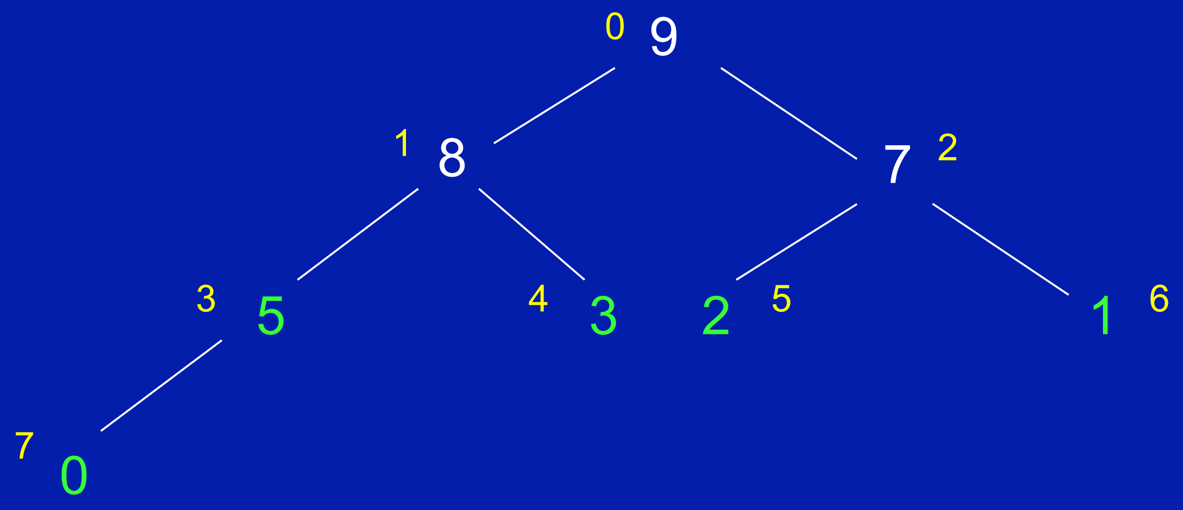
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

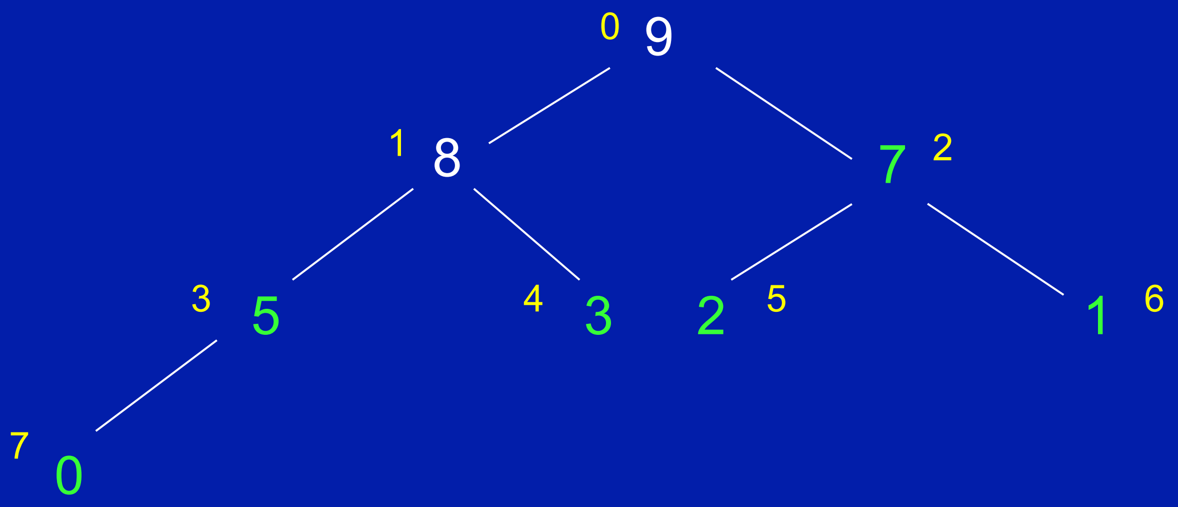
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

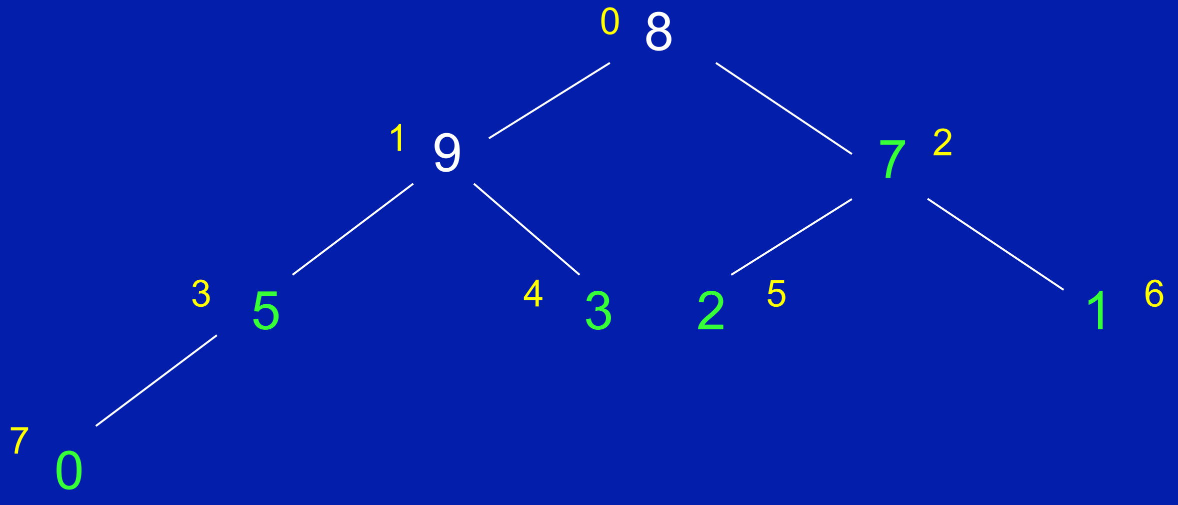
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

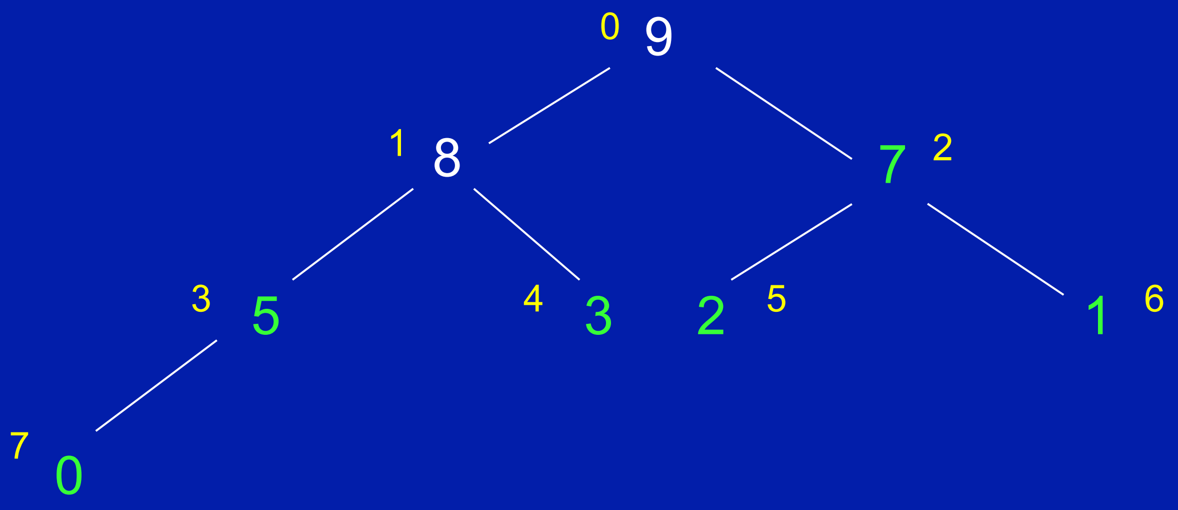
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let’s sort this one now:

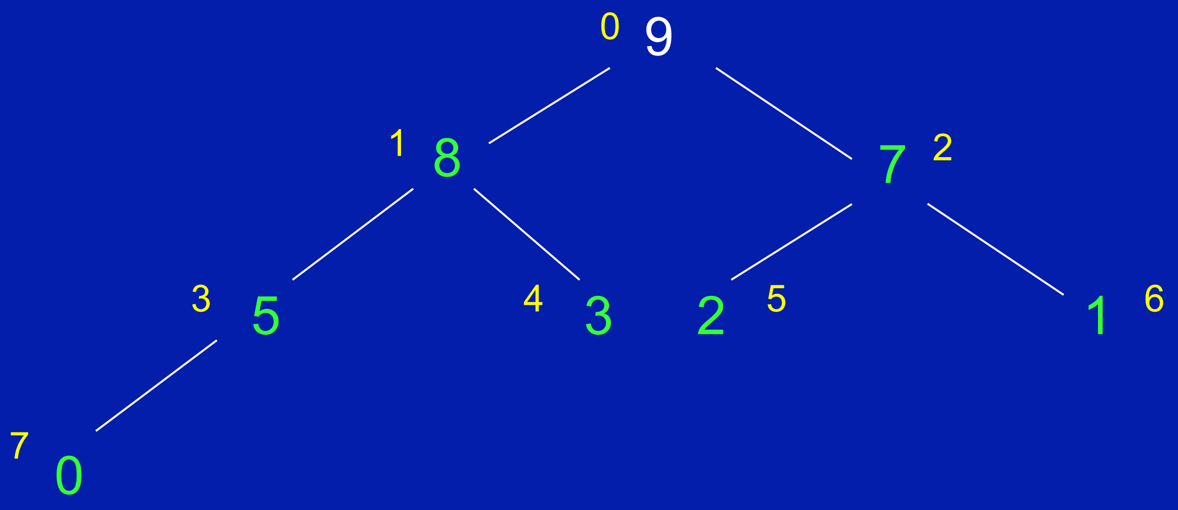
```
while the heap is not empty
          remove the first item from the heap by swapping it
                    with the last item in the heap
          reduce the size of the heap by one
          restore the heap property
```



Heapsort redux

Let's sort this one now:

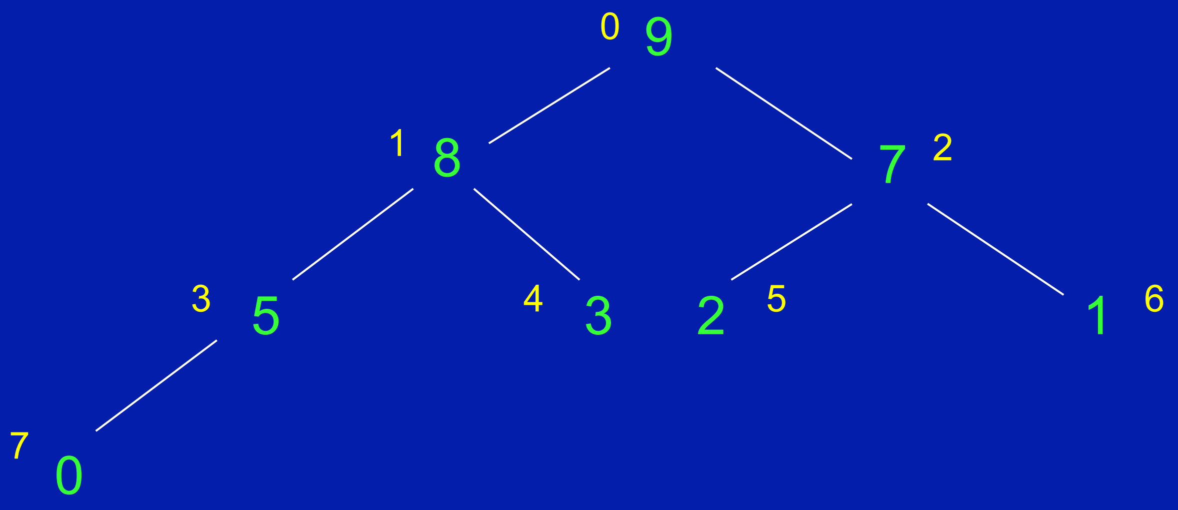
```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heapsort redux

Let's sort this one now:

```
while the heap is not empty
        remove the first item from the heap by swapping it
                with the last item in the heap
        reduce the size of the heap by one
        restore the heap property
```



Heap implementation

0	1	5	3	2	8	7	9
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

9	1	5	3	2	8	7	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

1	9	5	3	2	8	7	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

1	2	5	3	9	8	7	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

7	2	5	3	9	8	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

2	7	5	3	9	8	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

2	3	5	7	9	8	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

8	3	5	7	9	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

3	8	5	7	9	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

3	7	5	8	9	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

9	7	5	8	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

5	7	9	8	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

8	7	9	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

7	8	9	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

9	8	7	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

8	9	7	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Heap implementation

9	8	7	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

9	8	7	5	3	2	1	0
0	1	2	3	4	5	6	7

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k + 1

we can find the right child of k at index: 2k + 2

we can find the parent of k at index: floor((k – 1) / 2)

Heap implementation

9	8	7	5	3	2	1	0
0	1	2	3	4	5	6	7

`Done!`

Again, heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

We can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

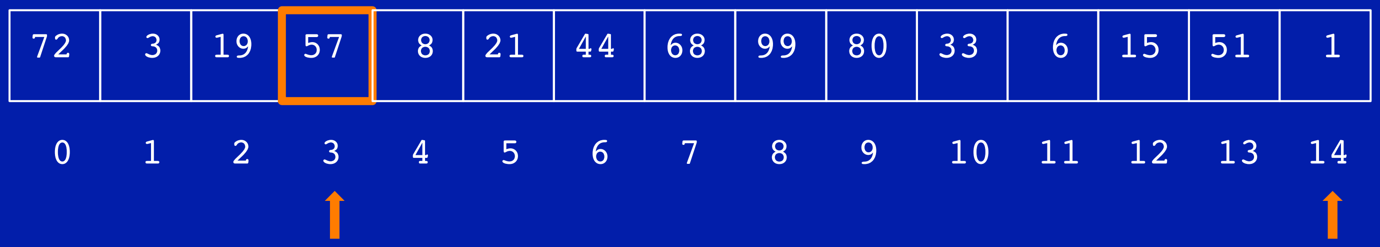
we can find the parent of k at index: $\text{floor}((k - 1) / 2)$

Questions?

Search, revisited

In past weeks, we’ve seen linear search – O(n)...

Linear search with recursion



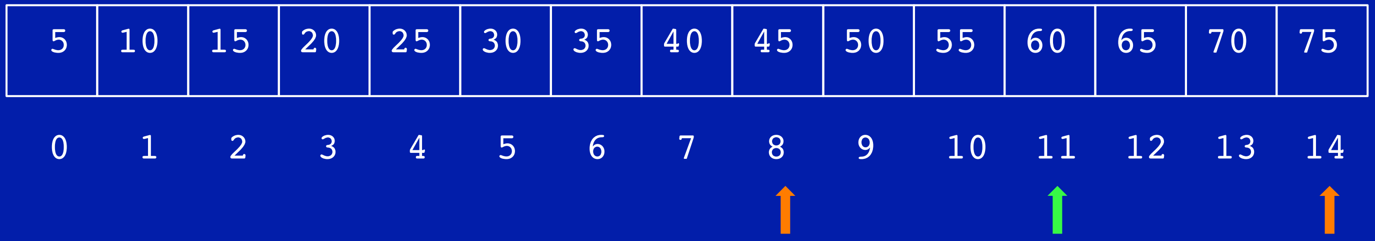
```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
    linSearch(array, 57, 2, 14)
      linSearch(array, 57, 3, 14) // yippee!!
```

Search, revisited

In past weeks, we’ve seen linear search – O(n)...

...and we’ve seen binary search – O(lg n)...

Binary search with iteration



5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Search, revisited

In past weeks, we've seen linear search – O(n)...

...and we've seen binary search – O(lg n)...

The latter gives better time complexity, but it requires sorted data. And both are effectively "trial and error" approaches to solving the search problem.

Wouldn't it be nice if we could get even better performance, without the dependency on sorting?

Search, revisited

Assume we want to store a lot of values that are associated with keys. That is, we have a collection of key-value pairs. The values have no fixed order, and each value is accessed by using the associated key.

A collection of key-value pairs is called a map or association table or a dictionary.

index0		
index1	Key2	Value2
index2		
index3	Key1	Value1
index4		
index5		
index6	Key3	Value3
index7		

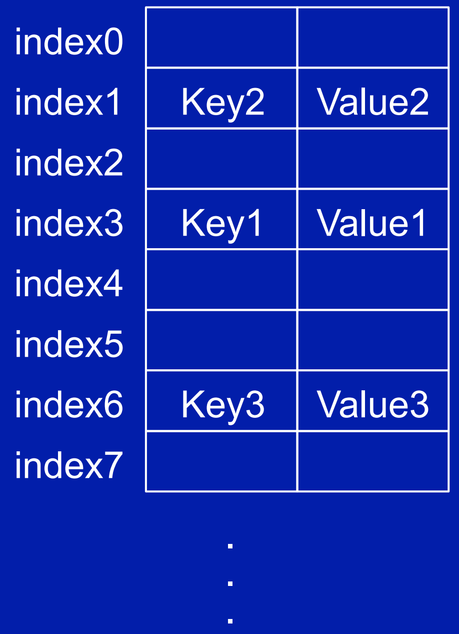
.
.
.

Search, revisited

For example, we could have a table of automobile drivers, where each value is a record containing the driver’s name, address, bad driving points, and a driver’s licence number.

```
struct Driver
{
   string name;
   string address;
   int    points;
   int    licenceNumber;
}
```

Which field could be a suitable search key?



Search, revisited

To place those key-value pairs in a sequential data structure like an array, we need a mapping between the unique keys and individual array indexes.

A very simple solution would be to have the keys and the array indexes be the same (i.e., your driver’s licence number is an index in ICBC’s array).

In other words, if we assign driver’s licence numbers in the range 0 to N-1, then we can store the data for driver k in location k of the array (of size N).

Given a driver’s licence number, we could access the driver’s data in O(1) time!

index0		
index1	Key2	Value2
index2		
index3	Key1	Value1
index4		
index5		
index6	Key3	Value3
index7		

⋮

Search, revisited

But it’s not often convenient to have key and array indexes be the same values.

For example, a credit card number typically has 16 digits, allowing for 10^{16} credit card accounts. That’s way more accounts than there are people on the planet. Each person could have at least 1,000,000 different credit cards, which may not be necessary. And the array with 10^{16} locations won’t fit in your computer anyway.

So it’s really not reasonable to use the credit card number as an index into an array.

Search, revisited

Or perhaps the key isn’t numeric. If you’re storing information about aircraft or radio stations, your keys will be alphanumeric.

Clearly we can’t use ‘C-FIBW’ as an array index.

What do we do now?

Hash functions

The solution to our problem is called hashing.

We define a function that transforms keys into numeric array indices. Such a function is called a hash function. The index is a hash index. The table is a hash table.

Hash functions

The solution to our problem is called hashing.

We define a function that transforms keys into numeric array indices. Such a function is called a hash function. The index is a hash index. The table is a hash table.

Given a key k, our hash function h has to generate an index i in the range of 0 to N-1 where N is the number of locations in the array.

h(Key1) ➔ index3
h(Key2) ➔ index1
h(Key3) ➔ index6

index0		
index1	Key2	Value2
index2		
index3	Key1	Value1
index4		
index5		
index6	Key3	Value3
index7		

⋮

Hash functions

A hash function can be thought of as the composition of two functions:

1. The hash code map:

   h_1: keys ➔ integers

2. The compression map:

   h_2: integers ➔ $[0, N - 1]$

The hash code map is applied to the key first, then the compression map is applied to the result:

$h(x) = h_2(h_1(x))$

Hash functions

In this context, “to hash” means to chop something up or to make a mess of it. The main idea in building a hash function is often to “chop off” some aspects of the key and to use this partial information as the basis for searching.

Two desirable attributes of a hash function:

1. Fast to compute
2. Generate indexes that are distributed throughout the table (array) in an apparently random and uniform way.

Why the first one? Obvious. Why the second one? We’ll talk about that soon.

Strategies for building hash functions

Truncation: Use only part of the key as input to the hash function.

For example, if we want to build a table of student records at UBC, then instead of using all 8 digits of the student number, we could use only the first 5 digits. We would need an array with space for 10^5 student records.

What happens when some students have the first 5 digits of their student number in common?

Strategies for building hash functions

Folding: Partition the key into parts and combine the parts using arithmetic operations.

For example, we could take a 16 digit credit card number and partition it into two 8 digit numbers. Add these two numbers together and we have an index into a table that's quite a bit (i.e., a factor of 10^8) smaller than the 10^{16} locations we were talking about earlier.

Strategies for building hash functions

Modular arithmetic: Use the modulus operator to produce an index in a given range.

For example, suppose a company with 150 employees wants to use each employee's Social Insurance Number as a key. We might use an array of size 200 to leave room for expansion. We could then generate a hash index from the SIN as follows:

```
hashIndex = SIN % 200;
```

Strategies for building hash functions

What about alphanumeric keys?

Suppose we have a table capable of holding 5000 records, and whose keys consist of strings that are 6 characters long. We can apply numeric operations to the ASCII codes of the characters in the string in order to determine an index:

```
int hash(unsigned char * key)
{
   int hashCode = 0;
   int index = 0;
   while(key[index] != `\0')
      hashCode += (int) key[index++];
   return hashCode % 5000;
}
```

Strategies for building hash functions

This is a good beginning, but there’s a problem lurking. Let’s say you use the 256-character extended ASCII code. Are all those characters equally likely to show up in your 6-character keys?

```
int hash(unsigned char * key)
{
   int hashCode = 0;
   int index = 0;
   while(key[index] != `\0’)
      hashCode += (int) key[index++];
   return hashCode % 5000;
}
```

Strategies for building hash functions

To help distribute the indexes generated by the keys, we can do some arithmetic to spread the computed values around:

```
int hash(unsigned char * key)
{
   int hashCode = 0;
   int index = 0;
   while(key[index] != `\0’)
      hashCode = 2 * hashCode + (int) key[index++];
   return hashCode % 5000;
}
```

Strategies for building hash functions

Now, before the % operation, the `hashCode` is in the range 0 to 16065. After applying the modulus operator, we get indexes in the range 0 to 4999 but with better distribution.

```
int hash(unsigned char * key)
{
   int hashCode = 0;
   int index = 0;
   while(key[index] != `\0')
      hashCode = 2 * hashCode + (int) key[index++];
   return hashCode % 5000;
}
```

Strategies for building hash functions

A common, simple, and often effective mathematical expression for basic hash functions has the form:

$$h(k) = (ak + b) \bmod N$$

where a and b are integers. This works even better if N (the array size) is a prime number.

It works even better still if $N \neq r^x \pm y$ where r is the radix of the character set (e.g., 64 characters, 128, 256) and x and y are small integers.

Strategies for building hash functions

A hash table is only as good as its hash function.

If we have n key-value pairs, and we have n locations in our table, and we have a hash function that perfectly maps each key to a unique index in the table, then finding the location of a key-value pair is O(1) time complexity.

At the other extreme, if we have a table with only one location (or “bucket”) to hold all the key-value pairs, and a hash function that maps all keys to the location of that bucket, then finding a key-value pair is O(n).

Reality lies somewhere in between.

Strategies for building hash functions

- Know what your keys will be or
- Study how your keys are distributed
- Try to include all important information in a key in the construction of its hash
- Try to make “neighbouring” keys hash to very different places
- Prune the features used to create the hash until it runs “fast enough” (don’t have the hash function do more computation than is necessary to get the job done)

Strategies for building hash functions

Our dream goal is to build a hash function that assigns a unique digital fingerprint to every piece of data that we might encounter. This isn’t usually possible.

Consider a hash function that, given a key, generates a 32-bit integer. That’s about 4.3 billion unique values.

This function could not generate a unique index for each of the roughly 7 billion people on the planet. If we have 7 billion keys (people) and only 4.3 billion indexes to work with, some people are going to get the same index. Uh oh.

This leads us to...

...formal math?

Let X and Y be finite sets where $|X| > |Y|$. If $f:X \to Y$, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x1 \neq x2$.

The pigeonhole principle

Let X and Y be finite sets where |X| > |Y|. If f:X -> Y, then $f(x_1) = f(x_2)$ for some x_1, $x_2 \in X$, where x1≠ x2.

Informally: If k+1 pigeons fly into k pigeonholes, some pigeonhole must contain at least 2 pigeons.

The pigeonhole principle

In other words, if you have more key-value pairs than places to put them, they’ll have to double (or more) up, even with the best, most perfect hash function you can create.

The pigeonhole principle

Also, even if you have fewer key-value pairs than places to put them, a less-than-perfect hash function may map different keys to the same index.

The pigeonhole principle

When your function maps two different keys to the same index, you have a collision.

The probability of a collision increases as more values are entered into the table.

The von Mises birthday paradox

What are the chances of collisions in a large table? Better than you might think. Here's a simple demonstration to help you see.

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC

The von Mises birthday paradox

What are the chances of collisions in a large table? Better than you might think. Here's a simple demonstration to help you see.

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC

If you have just 23 people in a room, there's a better than 50% chance that two people in the room share a birthday!

The von Mises birthday paradox

That seems counterintuitive, but it can be proven mathematically.

Let’s compute the probability that in a room with 23 people, no two people share a birthday. (Why 23? Because I already know the answer.) Assume 365 different days, and birthdays are equally distributed across all 365 days.

Put one person in the room. The probability of nobody in the room sharing a birthday is 1. Put another person in the room. The probability that person 2 doesn’t share a birthday with person 1 is 364/365. Put another person in the room. The probability that person 3 doesn’t share a birthday with person 1 or person 2 is 363/365.

The von Mises birthday paradox

The addition of each person is an independent event. The probability of all the events occurring is the product of the probabilities of each event occurring. So when we get to 23 people in the room, the probability of nobody sharing a birthday is:

$(365 \times 364 \times 363 \times ... \times 345 \times 344 \times 343) / 365^{23}$

Your calculator will tell you that the result is about 0.4927. The probability that two people in that room do share a birthday then is 1 - 0.4927, or about 0.5073.

The von Mises birthday paradox

In other words, the chances that at least two people in a group of 23 share a birthday is greater than 50%, although there are 365 different birthdays possible!

We may not care about birthdays, but this also says that in a hash table of 365 locations, and with a hash function that maps all possible keys onto 365 indexes, and with 22 key-value pairs entered in the table at random locations, there's a greater than 50% chance of a collision when the 23rd key is hashed (assuming there hasn't been one already).

The von Mises birthday paradox

A rule of thumb for estimating the number of values you need to enter into a hash table before you have a 50% chance of a collision is

sqrt(1.4 * n)

where n is the number of possible hash values that your function can map to.

To put it more succinctly, collisions are inevitable.

Collision resolution

To review, if a hash function h maps two different keys x and y to the same index (i.e., $x \neq y$ and $h(x) = h(y)$), then x and y collide.

A hash function that distributes items randomly will cause collisions, even when the number of items hashed is small.

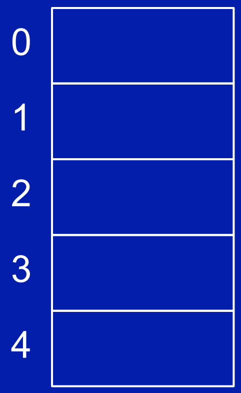
Consequently, we need a means of resolving collisions.

Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

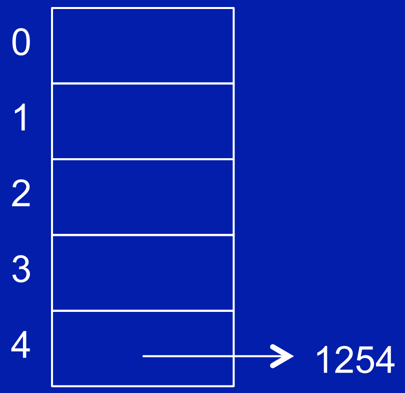


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

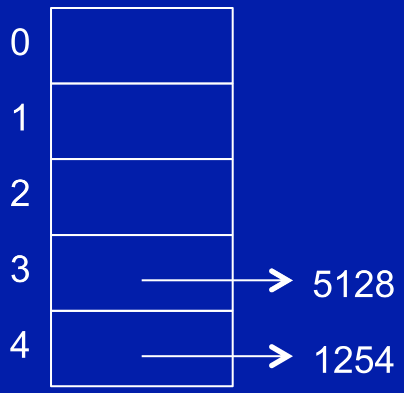


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

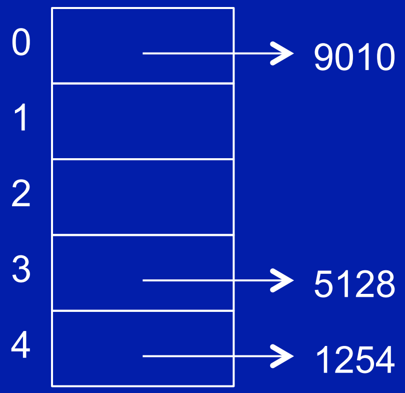


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

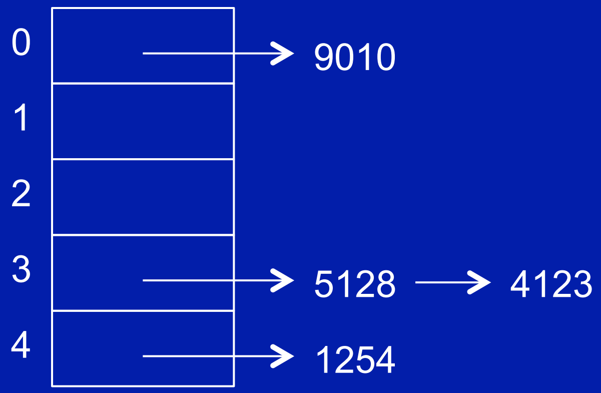


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

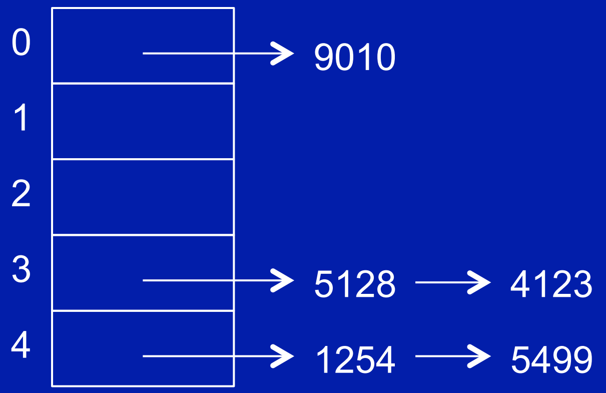


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

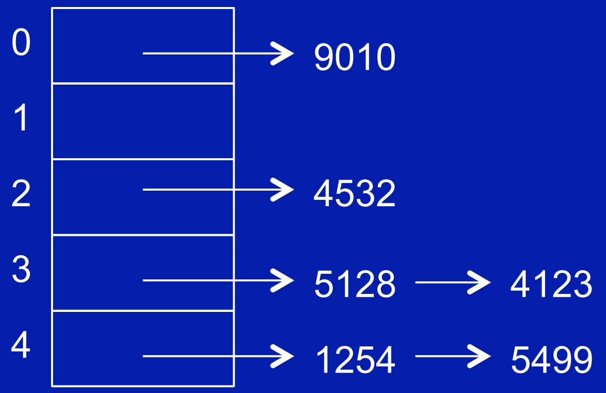


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

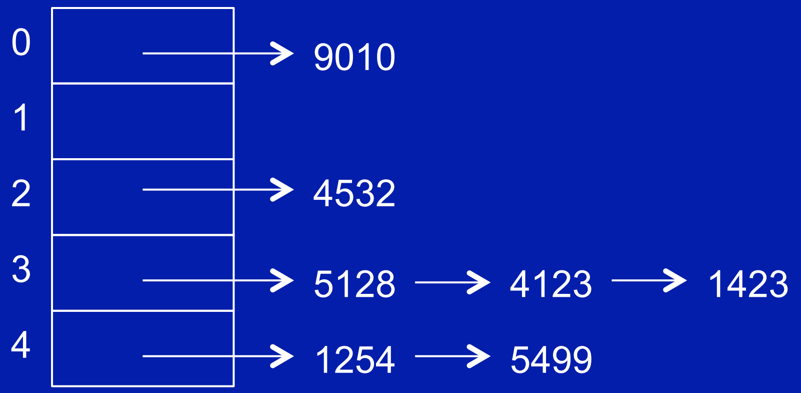


Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

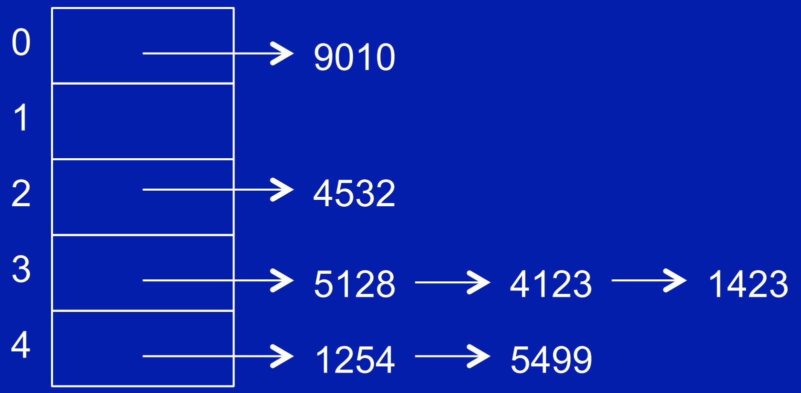
Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236



Chaining

When we search for an item, we apply the hash function to get to its hash index and then perform a linear search of the linked list. If the item is not in the list, then it's not in the table.



Chaining

Advantages:

- “unlimited size”
- easy to program
- deletions are easy

Disadvantages:

- the overhead for pointers can be quite large
- as the table becomes more full, you risk becoming more inefficient (e.g., given table size of N, the worst case searching time can be > N ... how?)

Open addressing

There is no linked list here. We have a hash table of size N containing key-value pairs, and we resolve collisions by trying a table of sequence entries until either the item is found in the table or we reach an empty location. The sequence is called a probe sequence. There are several alternatives...

Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

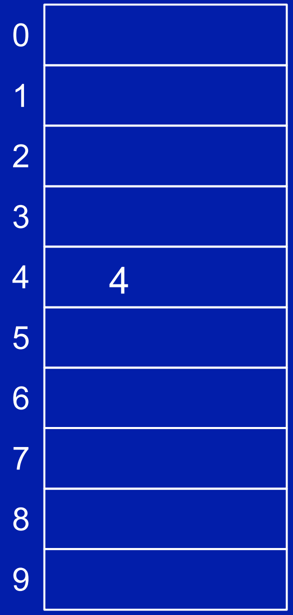
Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5



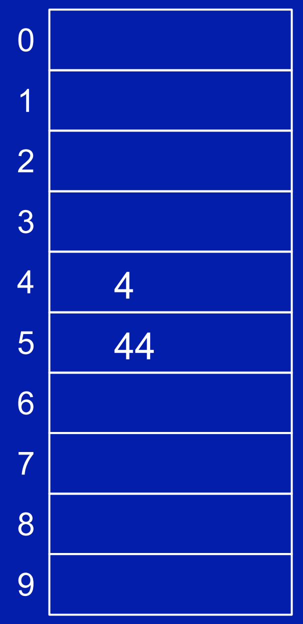
Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5



Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	4
5	44
6	444
7	
8	
9	

Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	4
5	44
6	444
7	6
8	
9	

Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	4
5	44
6	444
7	6
8	5
9	

Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

This is why you must store the values ***and*** the keys.

A draw back is clustering. Adjacent clusters tend to join together to form composite clusters.

0	
1	
2	
3	
4	4
5	44
6	444
7	6
8	5
9	

Open addressing

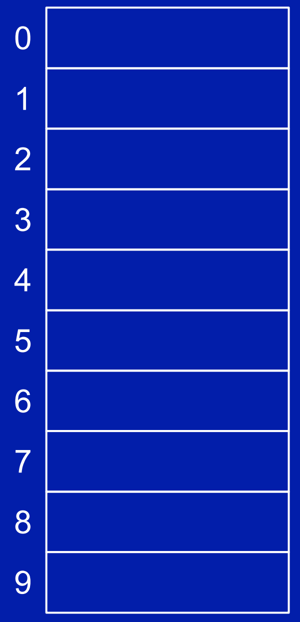
Probing every Rth location:

$h(k), h(k) + R, h(k) + 2R, \ldots, h(k) + (N - 1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose $h(x) = x \bmod 10$, $R = 2$

Hash these keys: 4, 14, 114, 1114, 11114



Open addressing

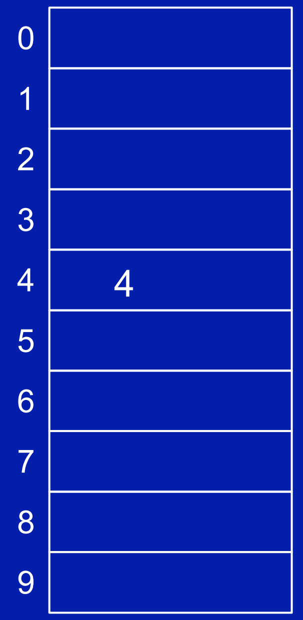
Probing every Rth location:

h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N – 1)R (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose h(x) = x mod 10, R = 2

Hash these keys: 4, 14, 114, 1114, 11114



Open addressing

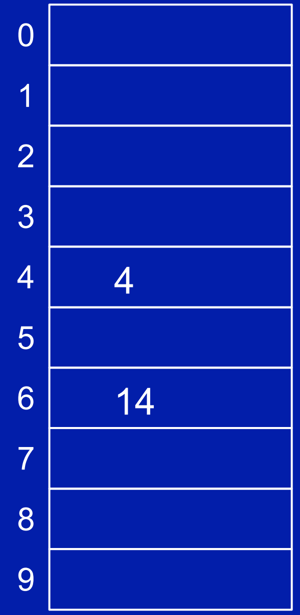
Probing every R^th location:

$h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N – 1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose h(x) = x mod 10, R = 2

Hash these keys: 4, 14, 114, 1114, 11114



Open addressing

Probing every R^{th} location:

$h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N – 1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose $h(x) = x \bmod 10$, $R = 2$

Hash these keys: 4, 14, 114, 1114, 11114

0	
1	
2	
3	
4	4
5	
6	14
7	
8	114
9	

Open addressing

Probing every Rth location:

h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N – 1)R (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose h(x) = x mod 10, R = 2

Hash these keys: 4, 14, 114, 1114, 11114

0	1114
1	
2	
3	
4	4
5	
6	14
7	
8	114
9	

Open addressing

Probing every Rth location:

$h(k), h(k) + R, h(k) + 2R, \ldots, h(k) + (N - 1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose h(x) = x mod 10, R = 2

Hash these keys: 4, 14, 114, 1114, 11114

0	1114
1	
2	11114
3	
4	4
5	
6	14
7	
8	114
9	

Open addressing

Probing every Rth location:

$h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N - 1)R$ (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose $h(x) = x \bmod 10$, $R = 2$

Hash these keys: 4, 14, 114, 1114, 11114

There will still be clustering, but the clusters are not consecutive locations.

0	1114
1	
2	11114
3	
4	4
5	
6	14
7	
8	114
9	

Open addressing

Quadratic probing:

$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Open addressing

Quadratic probing:

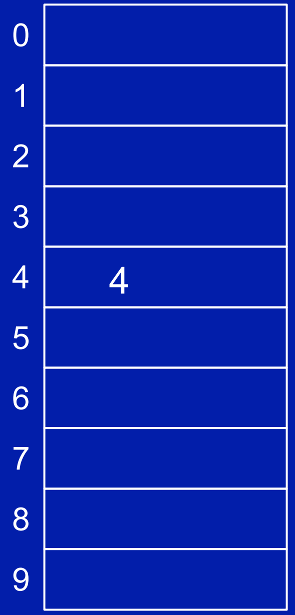
$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5



Open addressing

Quadratic probing:

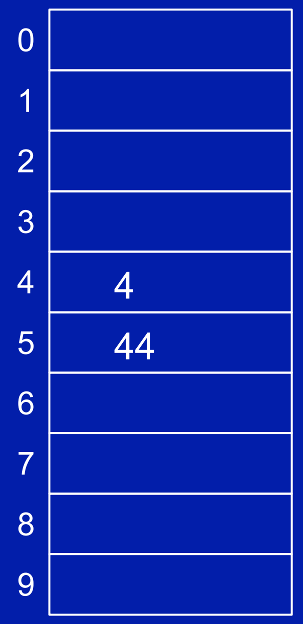
$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5



Open addressing

Quadratic probing:

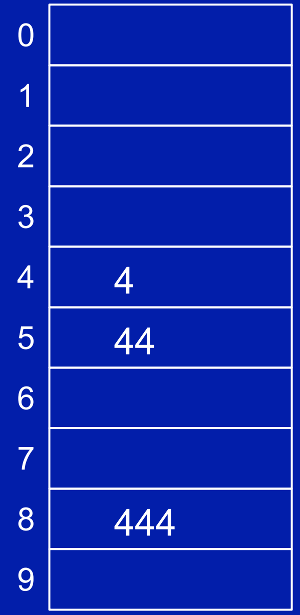
$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5



Open addressing

Quadratic probing:

$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5

0	
1	
2	
3	
4	4
5	44
6	6
7	
8	444
9	

Open addressing

Quadratic probing:

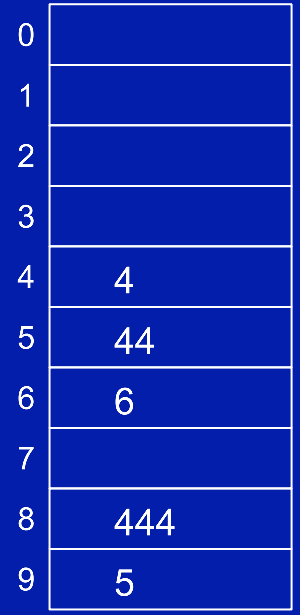
$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N - 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2 \bmod N$ so the probe sequence examines only about half the table.

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5



0	
1	
2	
3	
4	4
5	44
6	6
7	
8	444
9	5

Open addressing

Quadratic probing:

$h(k)$, $h(k) + 1^2$, $h(k) + 2^2$, ..., $h(k) + (N – 1)^2$ (all mod N)

This method avoids consecutive clustering.

Drawback: $i^2 = (N – i)^2$ mod N so the probe sequence examines only about half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5

Some people use:
$h(k)$, $h(k) + 1^2$, $h(k) - 1^2$, $h(k) + 2^2$, $h(k) - 2^2$, ...

0	
1	
2	
3	
4	4
5	44
6	6
7	
8	444
9	5

Open addressing

Pseudo-random probing:

$h(k)$, $h(k) + r_1$, $h(k) + r_2$, ..., $h(k) + r_{N-1}$ (all mod N)

where r1, r1, ..., rN-1 is a sequence of previously-generated pseudo-random numbers.

Drawback: we must store the pseudo-random numbers. Why?

Open addressing

Double hashing:

Up to now, we've seen that two keys that hash to the same location have the same probe sequence. Here's a better solution, assuming the two keys are different.

$h(k), h(k) + 1h_2(k), h(k) + 2h_2(k), \ldots, h(k) + (N\text{-}1)h_2(k)$ (all mod N)

...but if $h_2(k) = 0$, then set $h_2(k) = 1$

A common choice is $h_2(k) = q - (k \bmod q)$ where q is a prime < N

The hope is that if $h(x) = h(y)$, then $h_2(x) \neq h_2(y)$

Open addressing

Disadvantages of open addressing:

•Clusters can run into one another
•The hash table must be at least as large as the number of items hashed, and preferably much larger
•Deletions can be a problem. This will cause problems when searching for an existing key in a possibly long probe sequence. The solution here is to mark the space previously occupied by a key-value pair with a marker indicating that the probe sequence shouldn’t stop at this now-unoccupied location. The marker is called a “tombstone”. For insertion, the tombstone is considered to be an empty location.

Advantages:

•No memory is wasted on pointers

Performance of hashing

Performance depends on:

- the quality of the hash function
- the collision resolution algorithm
- the available space in the hash table

To estimate how full a table is, we calculate the load factor α:

$$\alpha = (\text{number of entries in table}) / (\text{table size})$$

For open addressing, $0 \leq \alpha \leq 1$

For chaining, $0 \leq \alpha < \infty$

Performance of hashing

The legendary Donald Knuth has estimated the expected number of probes for a successful search as a function of the load factor α:

Linear probing:	$\frac{1}{2}(1 + 1/(1-\alpha))$
Quadratic probing or Pseudo-random probing	$(1/\alpha)\ln(1/(1-\alpha))$
Chaining	$1 + \alpha/2$

Knuth will be visiting UBC on June 17. His talk is already full, but you can see live video of the talk at an overflow location.
See https://www.cs.ubc.ca/event/2015/06/conversation-donald-knuth