Implementing a heap using an array

1. Storing a heap binary tree as an array
2. Heap algorithms using array storage
3. Heap sort

1. Storing a heap binary tree as an array

A heap is a binary tree with certain properties.

Assumption: each vertex (node) in the binary tree has a value (a key) associated with it

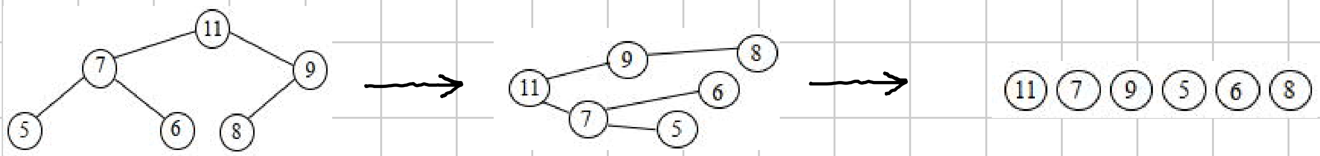
To be a heap a binary tree must have the following properties:

property 1: the parent's key is at least as large as the keys of its children

property 2: all depths, except possibly the largest, have the maximum number of nodes

property 3: at the largest depth, any missing nodes (leaves) are at the right end

From binary tree to array

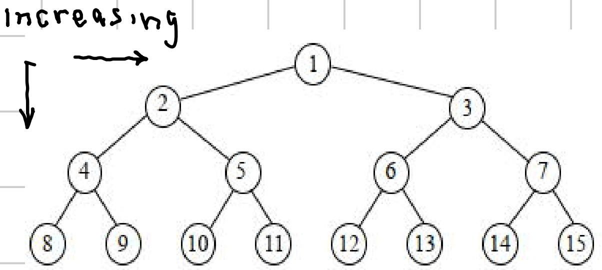


We have lost the pointers to children and parents!

Q: When a binary tree is stored as an array, how can we find the children and/or parent of a given node *i*?

A: We store the nodes in a particular order and use the array indices (locations).

Array indices and locations in binary tree.

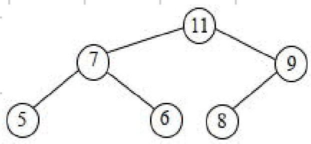


corresponding array

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example

heap as binary tree



corresponding array

11	7	9	5	6	8

For node whose index is x in array:

LEFT(x) = 2x , RIGHT(x) = 2x + 1, PARENT(x) = $\lfloor x/2 \rfloor$

Assumption: heap array is A

NOTES

1. key of root is always A[1]

2. *A.heapsize* is number of elements in heap

3. *A.length* must be at least *A.heapsize*

4. assume functions LEFT, RIGHT and PARENT are available

Example

Consider swapping key of *node1* with its parent key. Assume *A*[*x*] corresponds to *node1*.

Using pointers

```
temp = node1.key
node1.key = node1.parent.key
node1.parent.key = temp
```

Using array

```
temp = A[x]
A[x] = A[PARENT(x)]
A[PARENT(x)] = temp
```

□

Q: Why store a heap as an array?

A: It is more efficient both in terms of running time and storage space to store a heap as an array rather than using a binary tree with pointers.

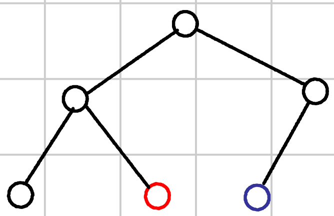
2. Heap algorithms using array storage

HEAP-INSERT: allows insertion of a new node into an existing heap

```
HEAP–INSERT(heapRoot, node)
input a heap whose root node is specified by heapRoot and a
new node that should be added to the heap

if depth height is not yet full of leaves then
  ▷ insert node as the rightmost leaf at depth height
else
  height = height + 1
  ▷ insert node as the first (leftmost) leaf at depth height
end

/* After this insertion, node now has a parent. We allow
node to percolate up in the heap until it finds its
correct location. */
while (node ≠ heapRoot and node.key > node.parent.key)
  SWAP(node, node.parent)
end
```

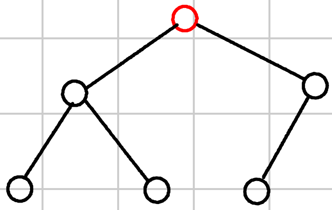


```
HEAP–INSERT(A, key)
input a heap which is stored as an array A and the key of a
new node that should be added to the heap
i = A.heapsize + 1
A[i] = key
A.heapsize = i
while (i > 1 and A[i] > A[PARENT(i)] )
   temp = A[i]
   A[i] = A[PARENT(i)]
   A[PARENT(i)] = temp
   i = PARENT(i)
end
```

HEAPIFY: for a binary tree (or subtree) for which some node *x* lacks property 1, restores this property for all nodes below node *x* thereby making a heap

```
HEAPIFY(heapRoot, node1)
input a heap whose root node is specified by heapRoot and
one particular node (node1) of the heap
/* All nodes in the heap, except for possibly node1,
have the heap properties. We allow node1 to trickle down to
its correct location. */

node2 = NIL
while (node1 ≠ node2)
   node2 = node1
   L = node1.left, R = node1.right
   if (L exists and L.key > node2.key) then
      node2 = L
   end
   if (R exists and R.key > node2.key) then
      node2 = R
   end
   if (node1 ≠ node2) then
     SWAP(node1, node2)
   end
end
```

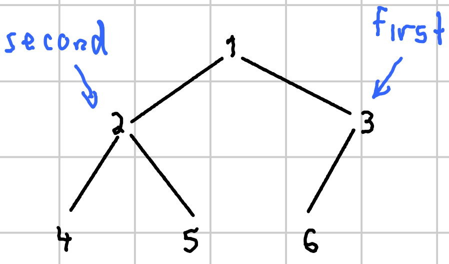


```
HEAPIFY(A, i)
input a heap which is stored as an array A and a location i
whose key may be smaller than its children
j = A.heapsize + 1
while (i ≠ j)
   j = i
   L = LEFT(i), R = RIGHT(i)
   if (L ≤ A.heapsize and A[L] > A[j]) then
      j = L
   end
   if (R ≤ A.heapsize and A[R] > A[j]) then
      j = R
   end
   if (i ≠ j) then
     temp = A[i]
     A[i] = A[j]
     A[j] = temp
   end
end
```

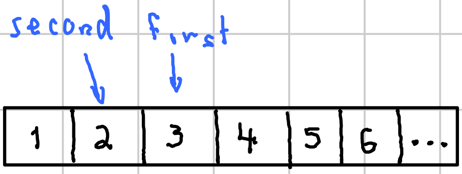
BUILD-HEAP: for a binary tree lacking property 1 at any (and possibly all) nodes, restores this property thereby making a heap

```
BUILDHEAP(heapRoot)
input a binary tree whose root node is specified by heapRoot
/* The input binary tree may lack property 1 at any or all
of its nodes. As such all nodes, except for the current
leaves, must be checked and if necessary modified so that
the final output is a binary tree which is a heap. */

for i from height − 1 to 0
  forEach internal node at depth i from right to left
    HEAPIFY(heapRoot, node)
  end
end
```



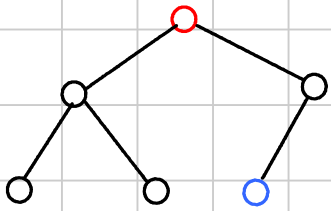

```
BUILDHEAP(A)
input a binary tree which is stored as an array A
/* It is assumed that the entire array is used in storing the
binary tree elements. */
A.heapsize = A.length
for i from ⌊(A.heapsize/2)⌋ to 1
    HEAPIFY(A, i)
end
```



HEAP-EXTRACT-MAX: removes the root from a heap, restores the heap properties and returns the maximum key

```
HEAP-EXTRACT-MAX(heapRoot)
input a heap whose root node is specified by heapRoot
/* The root of the heap is removed and replaced by the
rightmost leaf at the lowest depth. After restoring the
heap properties, the key of the original root is returned. */

max = heapRoot.key
▷ let node be the rightmost leaf at depth height
heapRoot.key = node.key
▷ remove node from the heap
if node was the only node at depth height
  height = height − 1
end
HEAPIFY(heapRoot, heapRoot)
return max
```



```
HEAP-EXTRACT-MAX(A)
input a heap which is stored as an array A

max = A[1]
n = A.heapsize
A[1] = A[n]
A.heapsize = n − 1
HEAPIFY(A, 1)
return max
```

3. Heap sort

A heap can be used to produce a sorted array using procedure HEAPSORT.

```
HEAPSORT(A)
input an array A which contains numbers that have no
    particular order
/* This procedure sorts the numbers in A from smallest to
largest. It is assumed that all locations in A are used. */
BUILDHEAP(A)
n = A.length
for i from n to 2
  temp = A[i]
  A[i] = A[1]
  A[1] = temp
  A.heapsize = A.heapsize − 1
  HEAPIFY(A, 1)
end
```

Example

Starting array: A: $\begin{array}{|c|c|c|c|c|c|c|}\hline 2 & 6 & 4 & 5 & 8 & 11 & 3 \\ \hline\end{array}$

stage	line	computation or new A	
1	5	$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline 11 & 8 & 4 & 5 & 6 & 2 & 3 \\ \hline\end{array}$	
2	6, 7	$n = 7,\ i = 7$	
3	8,9,10,11	$temp = 3,\ A[7] = 11,\ A[1] = 3,\ A.heapsize = 6$	$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline 3 & 8 & 4 & 5 & 6 & 2 & 11 \\ \hline\end{array}$
4	12	$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline 8 & 6 & 4 & 5 & 3 & 2 & 11 \\ \hline\end{array}$	$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline 2 & 6 & 4 & 5 & 3 & 8 & 11 \\ \hline\end{array}$
5	7,8,9,10,11	$i = 6,\ temp = 2,\ A[6] = 8,\ A[1] = 2,\ A.heapsize = 5$	
6	12	$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline 6 & 5 & 4 & 2 & 3 & 8 & 11 \\ \hline\end{array}$	

$\square$

NOTES

1. Heapsort needs no extra storage space.

2. Heapsort's runtime efficiency is the same as Mergesort's runtime efficiency.