Advanced Data Structures and Algorithms

Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology Computer Science Department

2015 – 2016

Heaps and Priority Queues

- Heaps
- Inserting into a Heap
- Removing an Item from a Heap
- Implementing a Heap
- Heap Operation
- Priority Queue
- Implementing a Priority Queue
- Summary

Priority

- **Priority** :- treated as more important than others. "the safety of the country **takes priority over** any other matter“.

- In the expression $2 + 3 \times 4$, the answer is 14 (not 20).

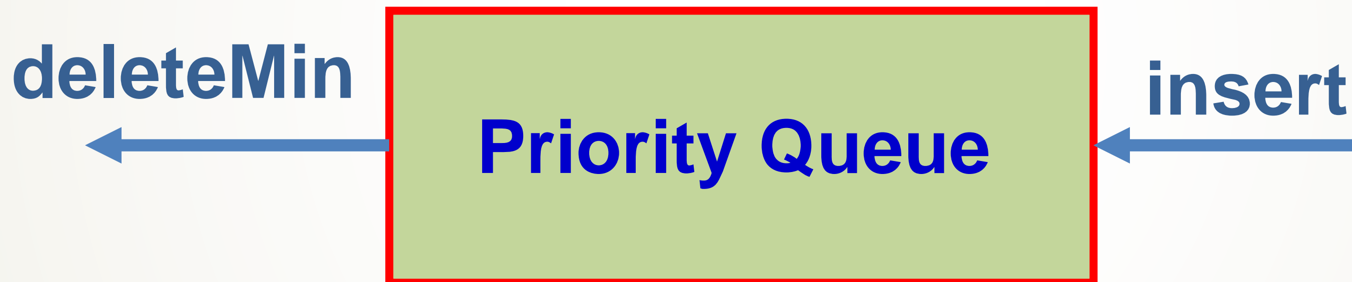
- The highest priority will reside with the lowest array index.

- Because it is complete, a binary heap can be stored compactly as an array of entries.

- A complete binary tree is a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right.

Definition

A priority queue is a data structure that supports two basic operations: insert a new item and remove the minimum item.

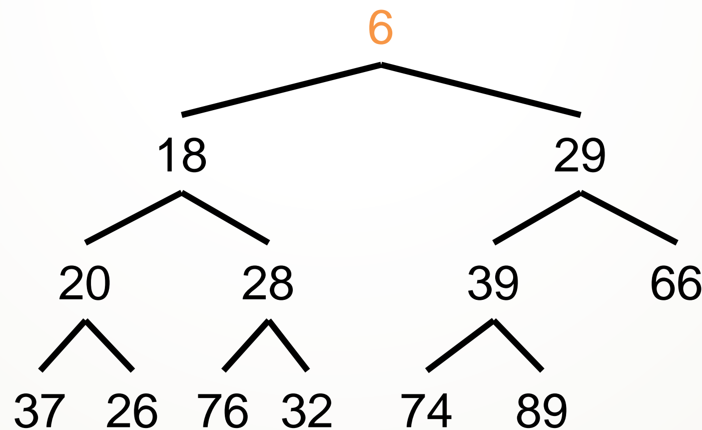


The heap is the classic method used to implement priority queues.

What is a Heaps

A complete binary tree:

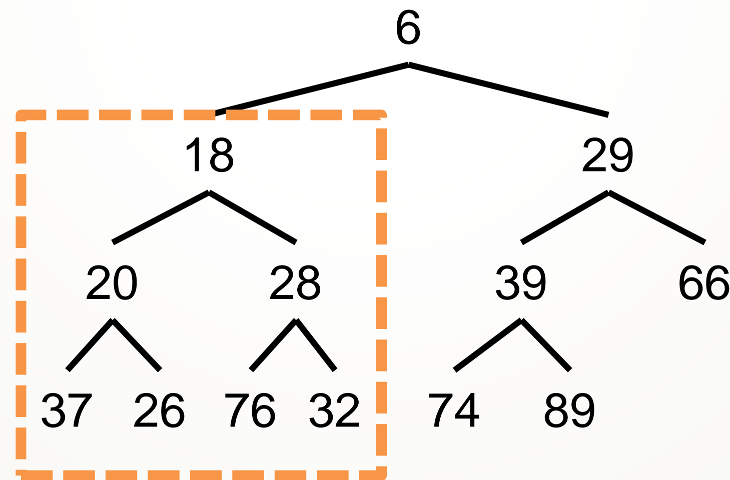
- The value in the root is the smallest in the tree



Heaps

A complete binary tree:

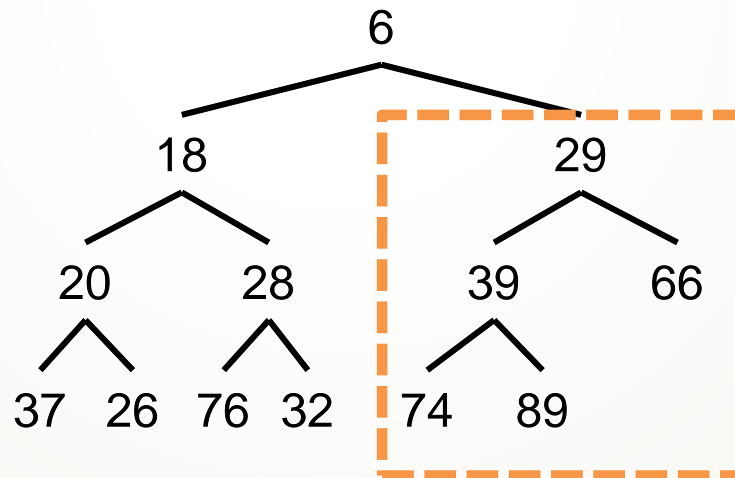
- The value in the root is the smallest in the tree
- Every subtree is a heap



Heaps

A complete binary tree:

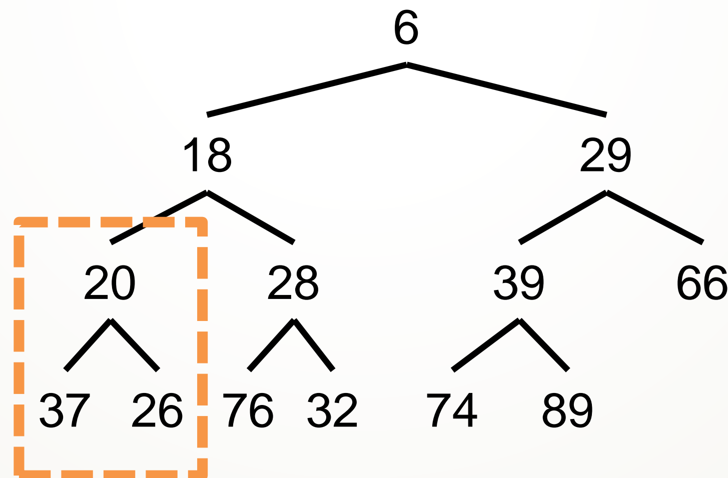
- The value in the root is the smallest in the tree
- Every subtree is a heap



Heaps

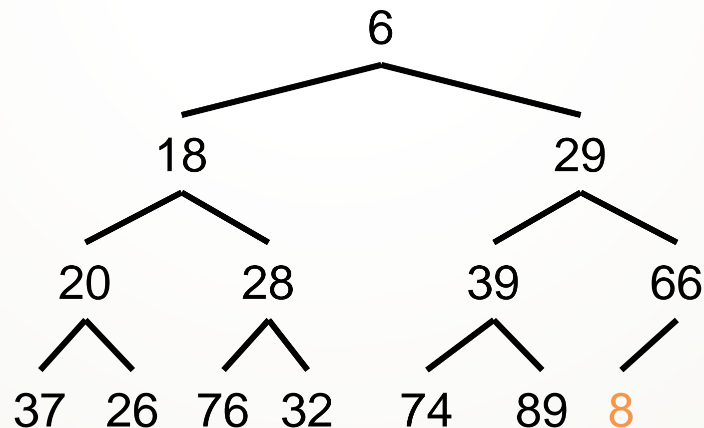
A complete binary tree:

- The value in the root is the smallest in the tree
- Every subtree is a heap



Inserting into a Heap

Insert the new item in the next position **at the bottom of the heap**

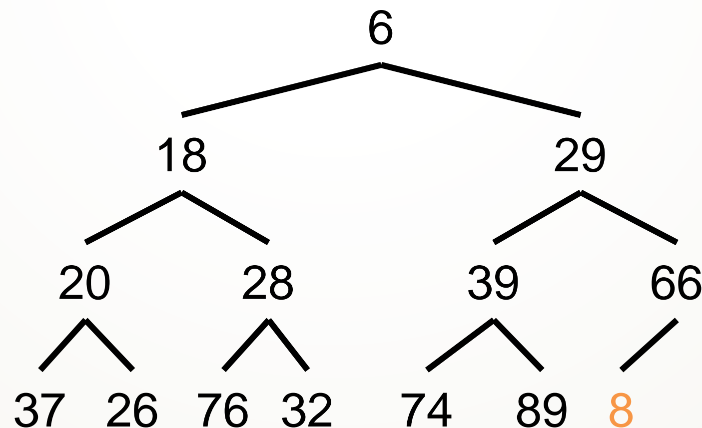


Inserting into a Heap

Insert the new item in the next position at the bottom of the heap

While new item isn't root and new item is smaller than parent

Swap new item and parent

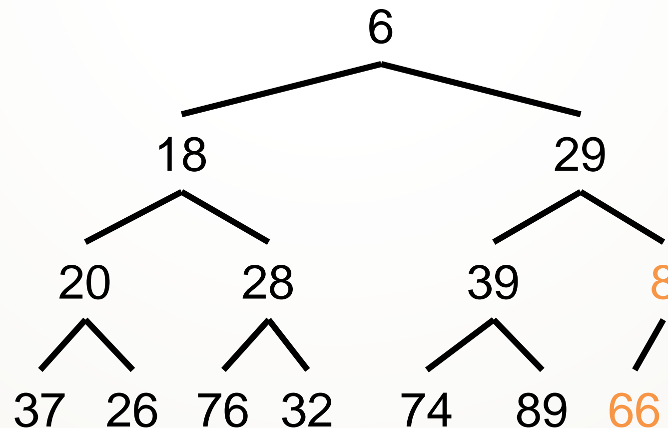


Inserting into a Heap

Insert the new item in the next position at the bottom of the heap

While new item isn't root and new item is smaller than parent

Swap new item and parent

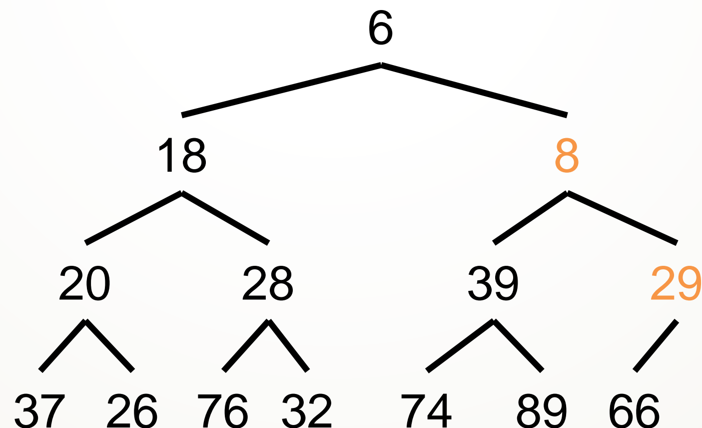


Inserting into a Heap

Insert the new item in the next position at the bottom of the heap

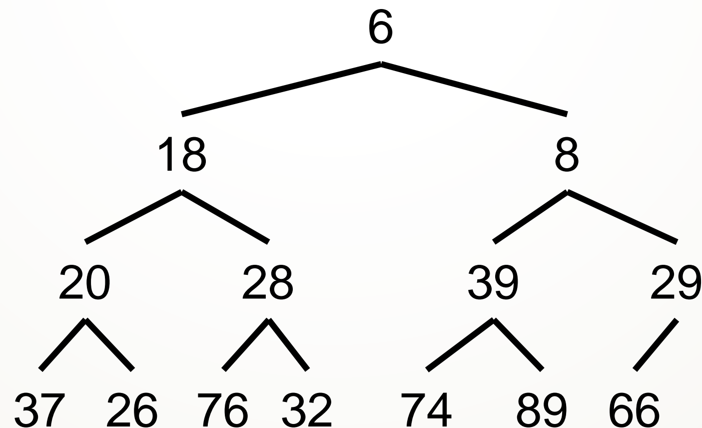
While new item isn't root and new item is smaller than parent

Swap new item and parent



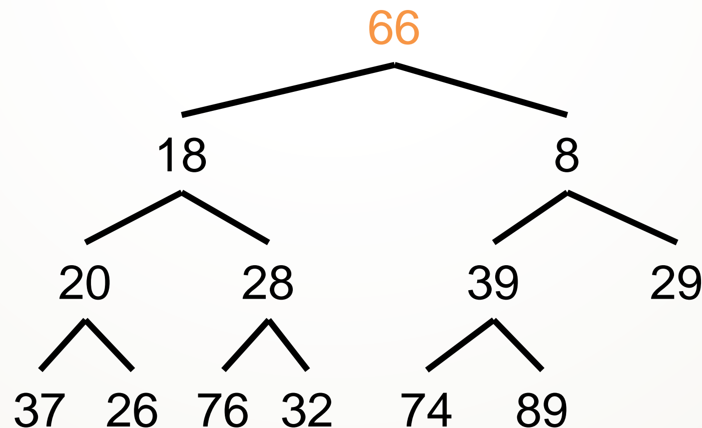
Removing an Item from a Heap

Remove the root by replacing it with **last item in heap** (LIH)



Removing an Item from a Heap

Remove the root by replacing it with last item in heap (LIH)

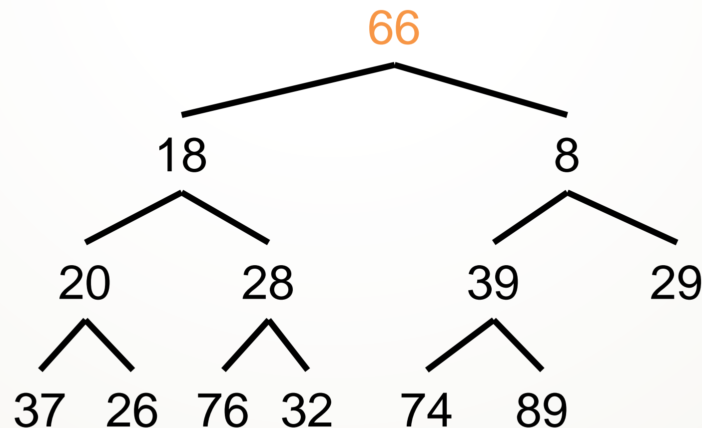


Removing an Item from a Heap

Remove the root by replacing it with last item in heap (LIH)

While item LIH has children and item is larger than either child

Swap item with its smaller child

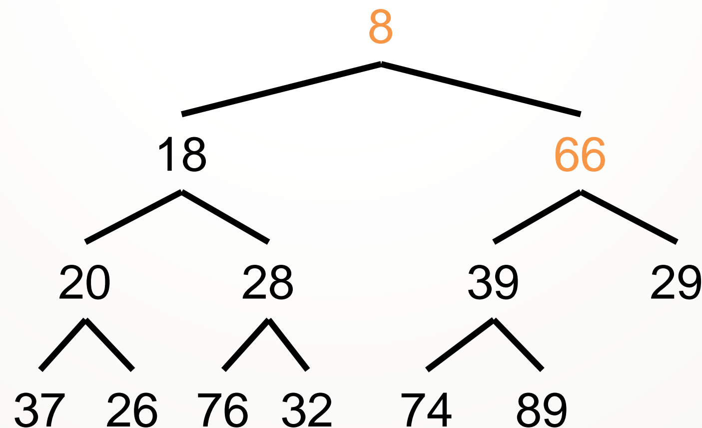


Removing an Item from a Heap

Remove the root by replacing it with last item in heap (LIH)

While item LIH has children and item is larger than either child

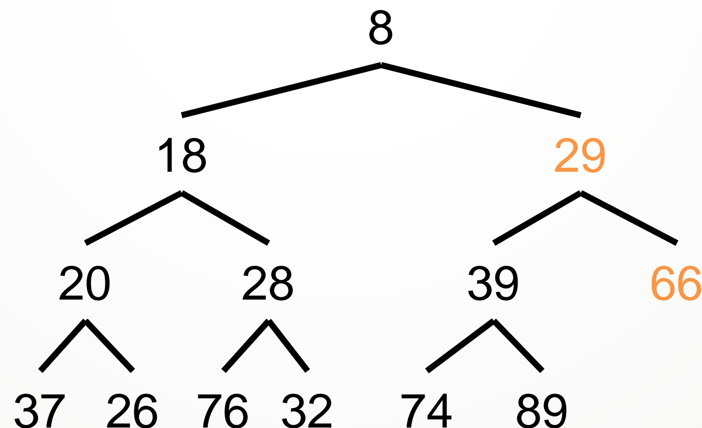
Swap item with its smaller child



Removing an Item from a Heap

Remove the root by replacing it with last item in heap (LIH)

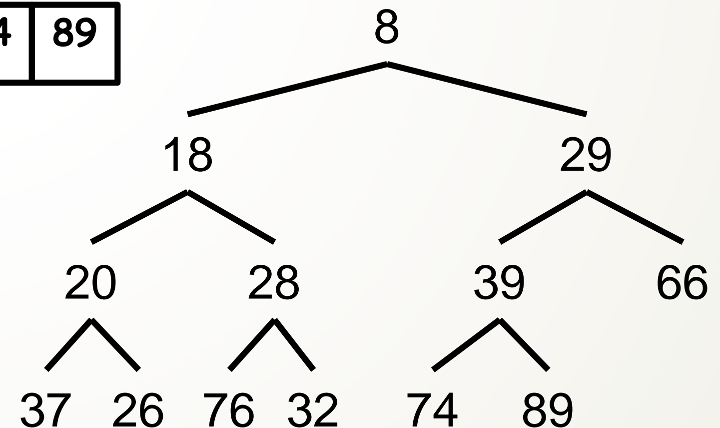
While item LIH has children and item is larger than either child
Swap item with its smaller child



Implementing a Heap

Use an array or ArrayList to hold the data

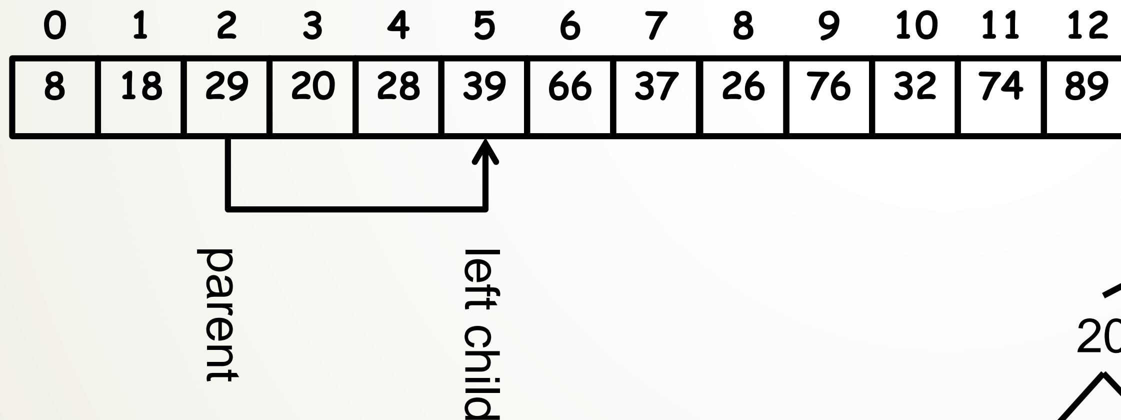
0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

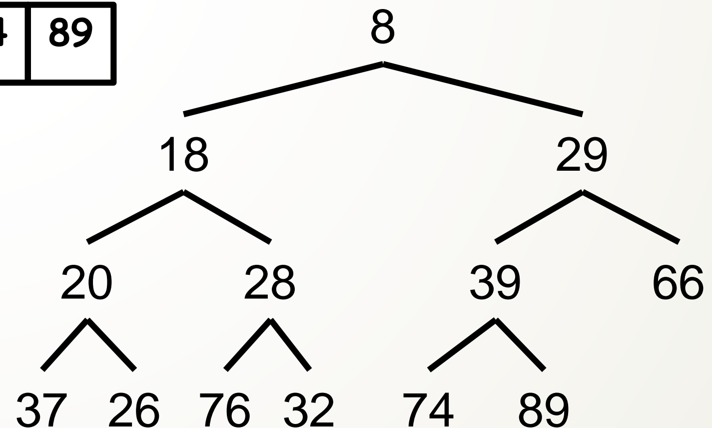


Implementing a Heap

Use an array or ArrayList to hold the data

- Node at ***p***,
  - left child is at $2p + 1$
  - right child is at $2p + 2$



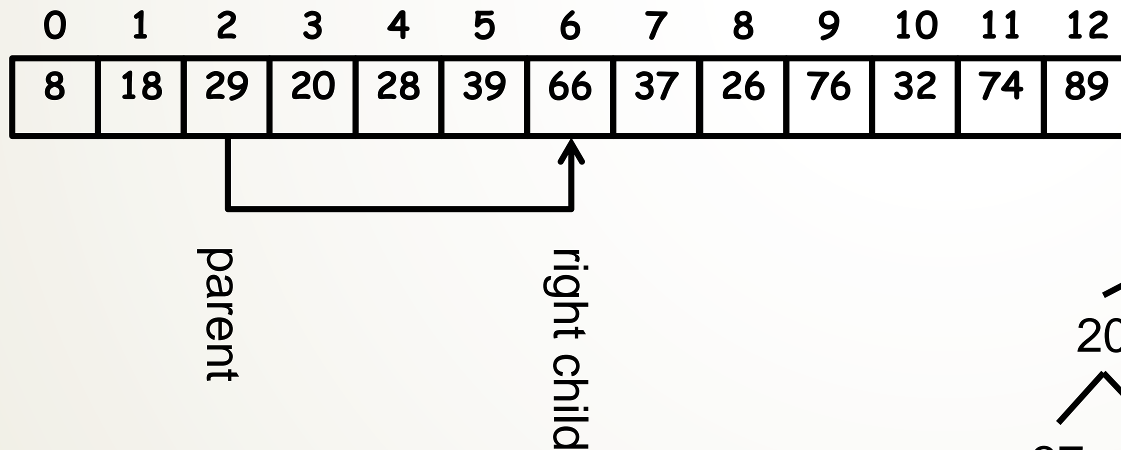


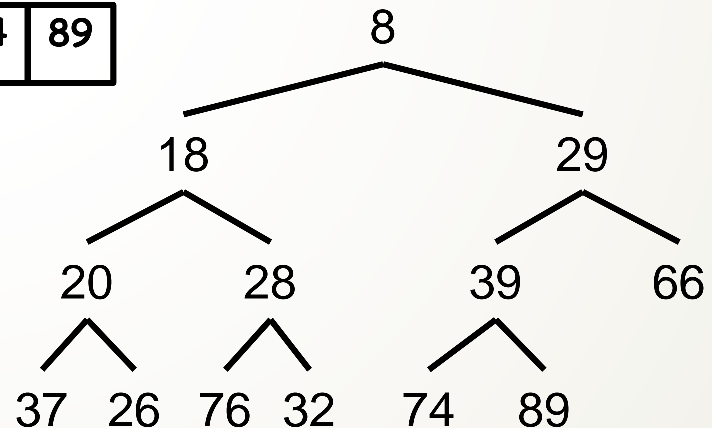
Left Child = 2 * *2* + 1 = 5

Implementing a Heap

Use an array or ArrayList to hold the data

- Node at p,
  - left child is at $2p + 1$
  - right child is at $2p + 2$



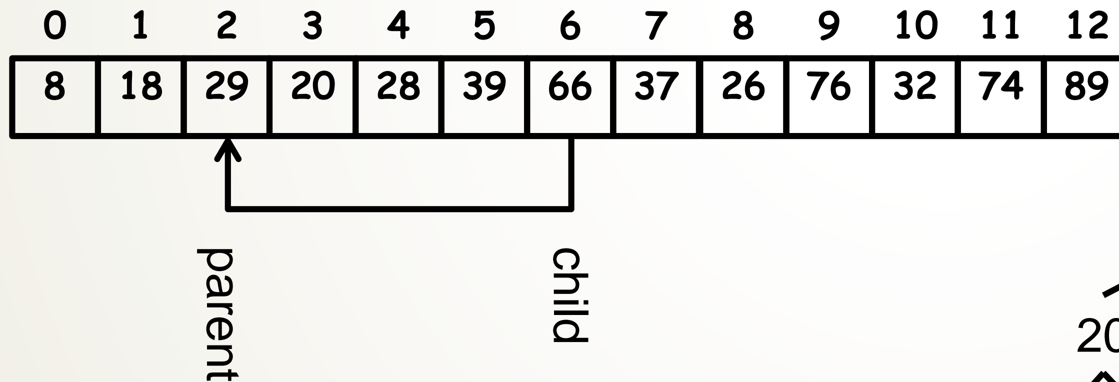


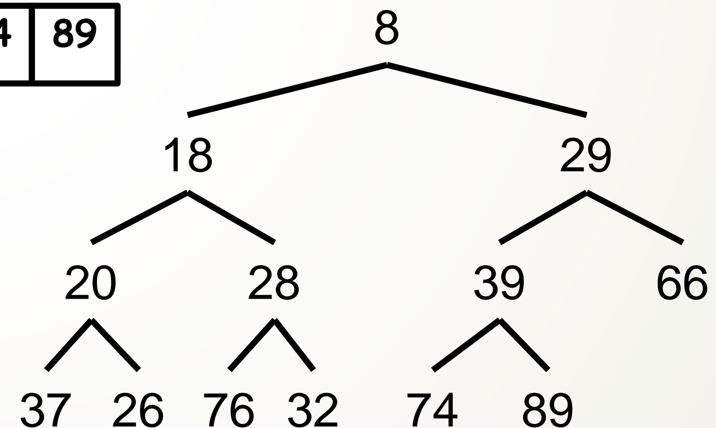
Right Child = 2 * *2* + 2 = 6

Implementing a Heap

Use an array or ArrayList to hold the data

- Node at c,
  - parent is at $(c - 1) / 2$





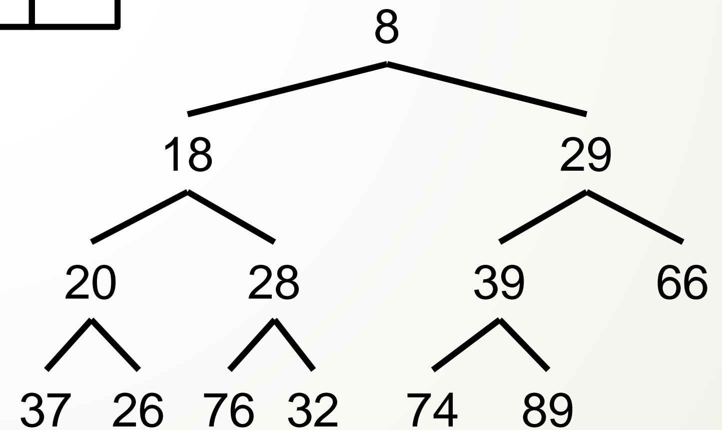
Parent = (*6* – 1) / 2 = 2

Insertion

Insert new item at end

table

0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

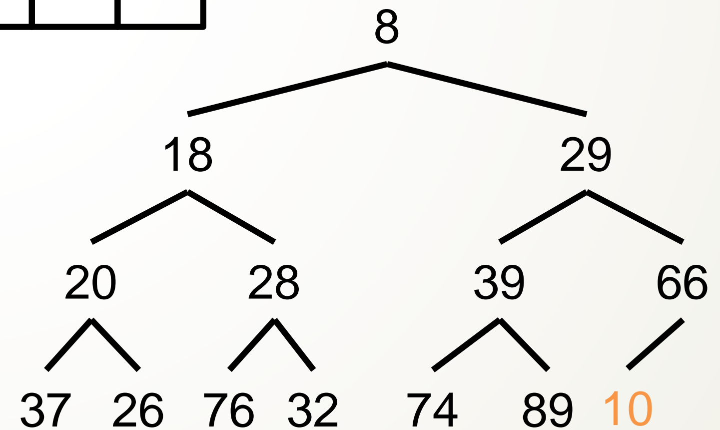


Insertion

Insert new item at end

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	10



Insertion

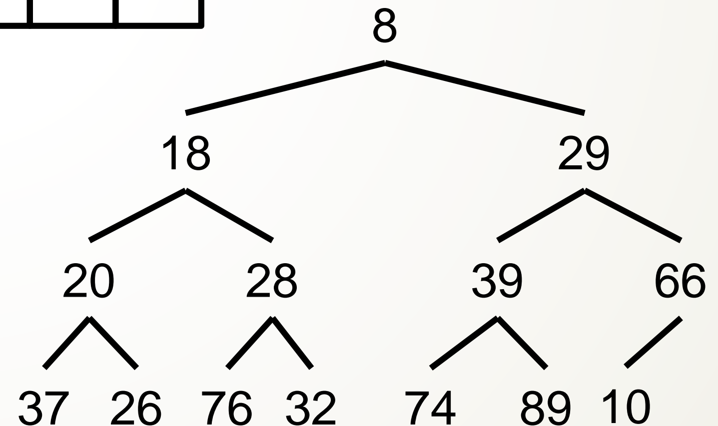
Insert new item at end,

set child to table.size() -1

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	10

```
child = table.size() - 1;  // 13
```



Insertion

Insert new item at end,

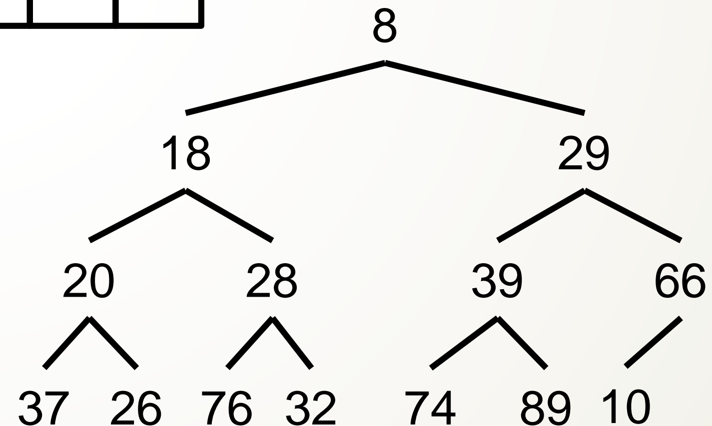
set child to table.size() -1

set parent to (child – 1) / 2

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	10

```
child = table.size() - 1;  // 13
parent = (child -1) / 2;   // 6
```



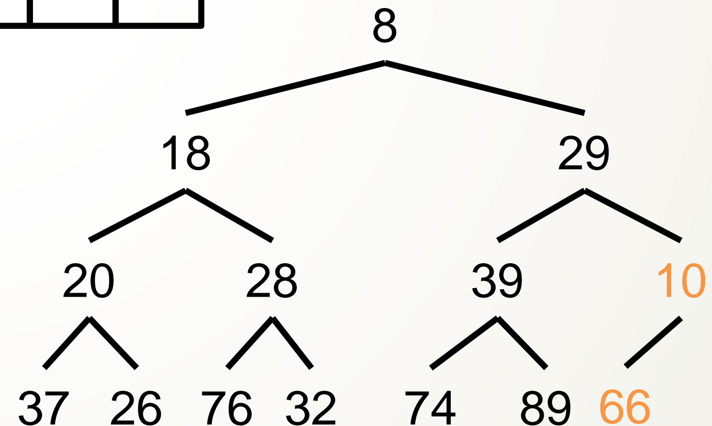
Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	10	37	26	76	32	74	89	66

```
child = table.size() - 1;  // 13
parent = (child -1) / 2;   // 6
```



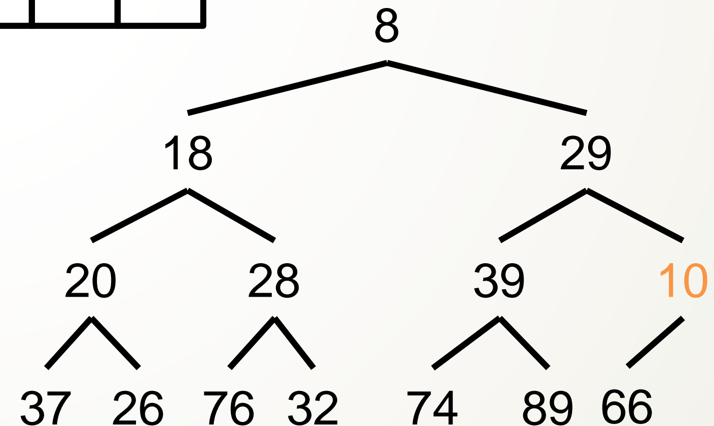
Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]
Set child equal to parent

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	10	37	26	76	32	74	89	66

child = parent; // 6

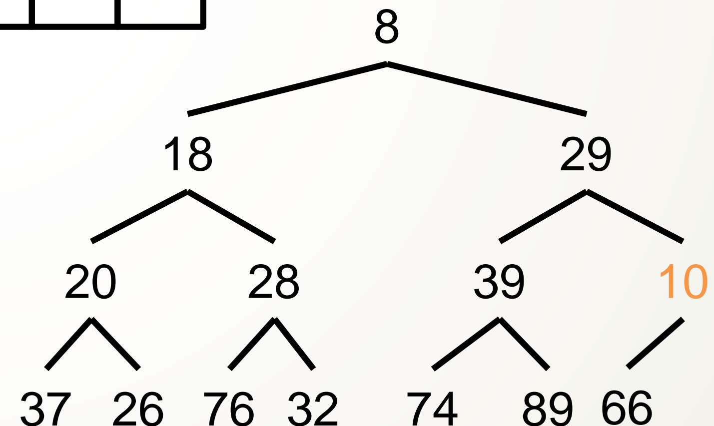


Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]
Set child equal to parent
Set parent equal to (child - 1) / 2

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	10	37	26	76	32	74	89	66




```
child = parent;  // 6
parent = (child -1) / 2;   // 2
```

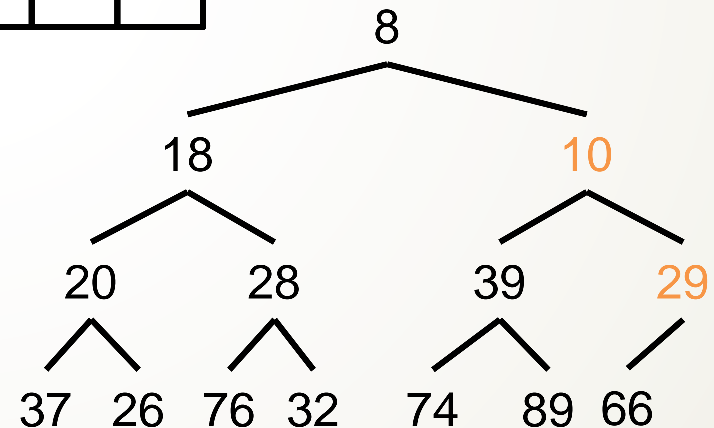
Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	10	20	28	39	29	37	26	76	32	74	89	66

```
child = parent;  // 6
parent = (child -1) / 2;   // 2
```



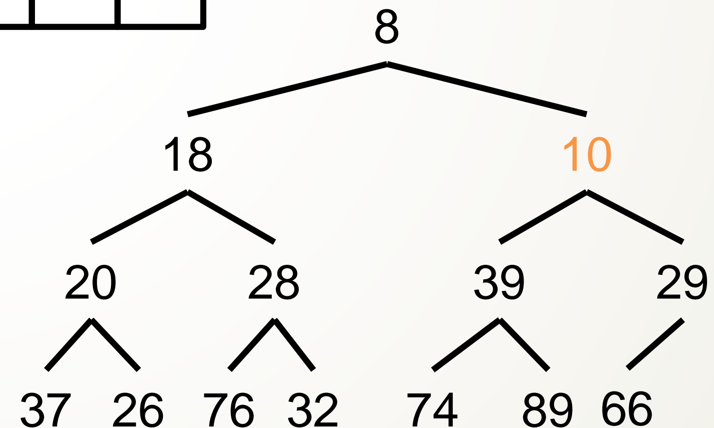
Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]
Set child equal to parent

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	10	20	28	39	29	37	26	76	32	74	89	66

```
child = parent;  // 2
parent = (child -1) / 2;   // 2
```



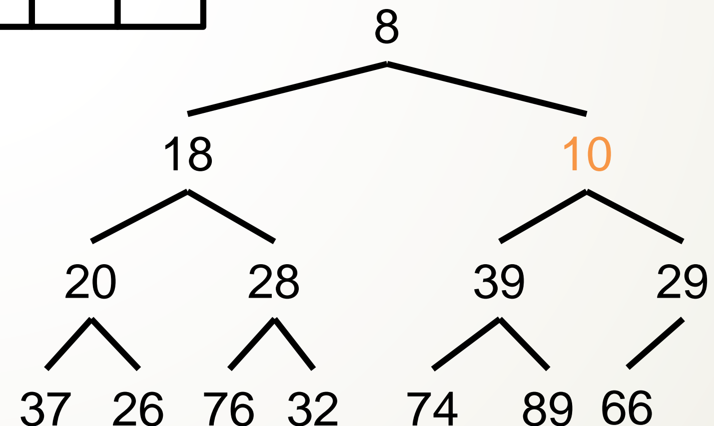
Insertion

while (parent >= 0 and table[parent] > table[child])
Swap table[parent] and table[child]
Set child equal to parent
Set parent equal to (child -1) / 2

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	10	20	28	39	29	37	26	76	32	74	89	66

```
child = parent;  // 2
parent = (child -1) / 2;   // 0
```

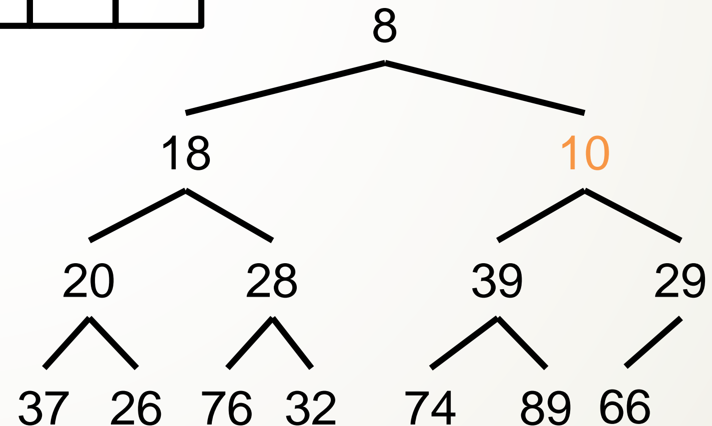


Insertion

while (parent >= 0 and table[parent] > table[child])

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	10	20	28	39	29	37	26	76	32	74	89	66



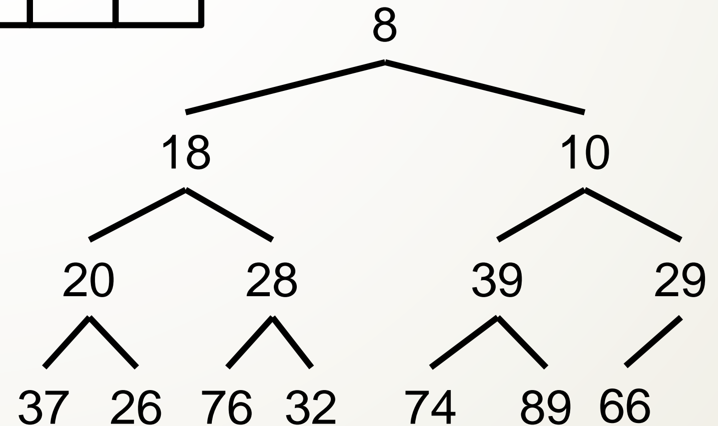

```
child = parent;  // 2
parent = (child -1) / 2;   // 0
```

Removal

Remove root, replace it with last element

table

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	10	20	28	39	29	37	26	76	32	74	89	66

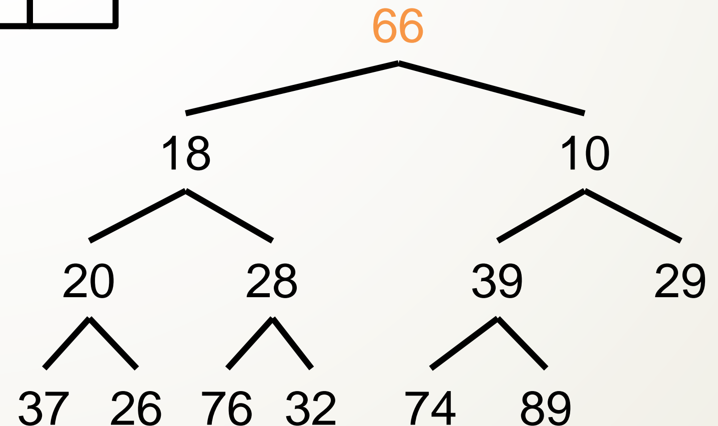


Removal

Remove root, replace it with last element

table

0	1	2	3	4	5	6	7	8	9	10	11	12
66	18	10	20	28	39	29	37	26	76	32	74	89



Removal

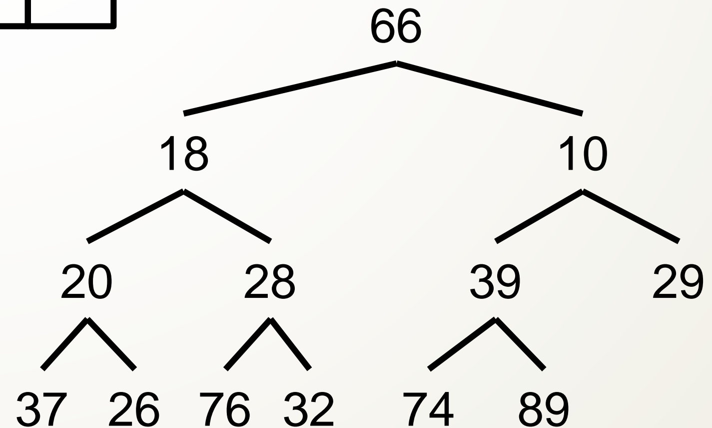
Remove root, replace it with last element

Set parent to 0

table

0	1	2	3	4	5	6	7	8	9	10	11	12
66	18	10	20	28	39	29	37	26	76	32	74	89

```
parent = 0;   // 0
```



Removal

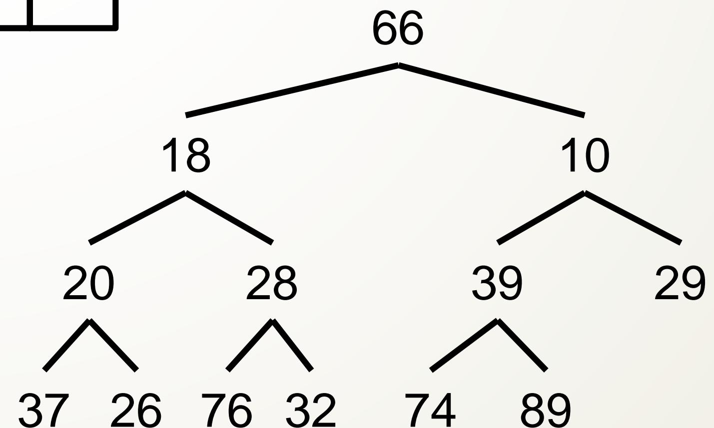
while (true)

Set leftChild to (2 * parent) + 1 and rightChild to leftChild + 1

table

0	1	2	3	4	5	6	7	8	9	10	11	12
66	18	10	20	28	39	29	37	26	76	32	74	89

```
parent = 0;   // 0
leftChild = 2 * parent + 1;   // 1
rightChild = leftChild + 1;   // 2
```



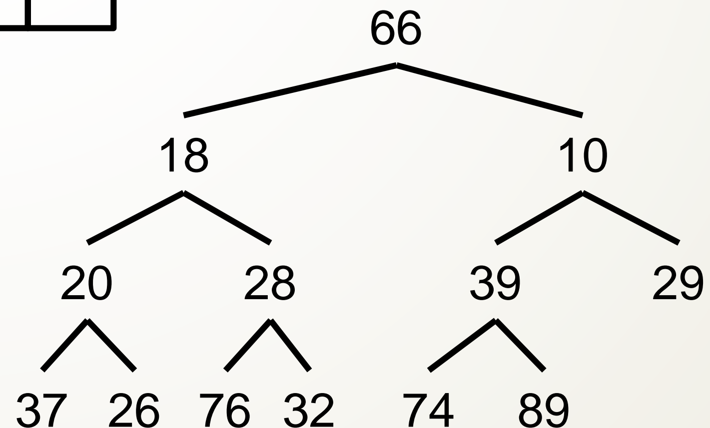
Removal

while (true)
Set leftChild to (2 * parent) + 1 and rightChild to leftChild + 1
if leftChild >= table.size() break out of loop

table

0	1	2	3	4	5	6	7	8	9	10	11	12
66	18	10	20	28	39	29	37	26	76	32	74	89

```
parent = 0;    // 0
leftChild = 2 * parent + 1;    // 1
rightChild = leftChild + 1;    // 2
```



Removal

while (true)

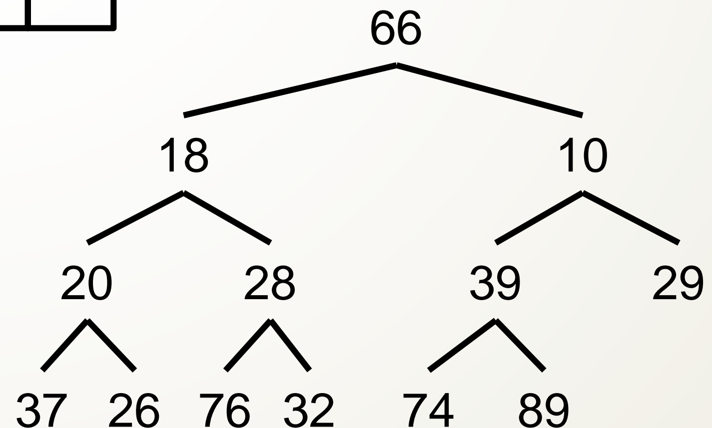
...

Set minChild to index of smaller child

table

0	1	2	3	4	5	6	7	8	9	10	11	12
66	18	10	20	28	39	29	37	26	76	32	74	89

```
parent = 0;   // 0
leftChild = 2 * parent + 1;   // 1
rightChild = leftChild + 1;   // 2
minChild = 2;
```



Removal

while (true)

...

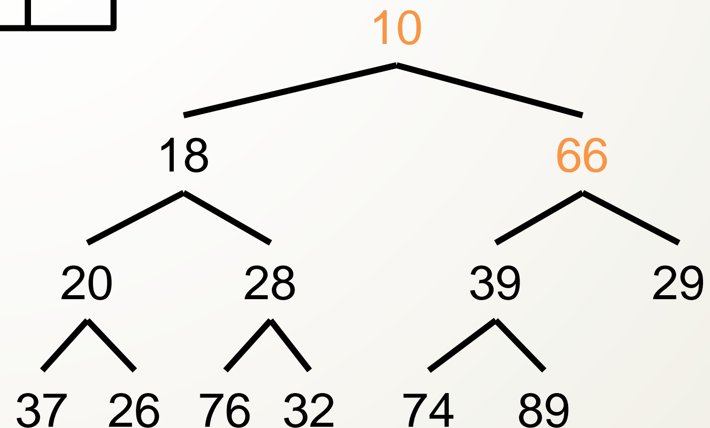
if table[parent] > table[minChild]

Swap table[parent] and table[minChild]

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	66	20	28	39	29	37	26	76	32	74	89

```
parent = 0;   // 0
leftChild = 2 * parent + 1;   // 1
rightChild = leftChild + 1;   // 2
minChild = 2;
```



Removal

while (true)

...

if table[parent] > table[minChild]

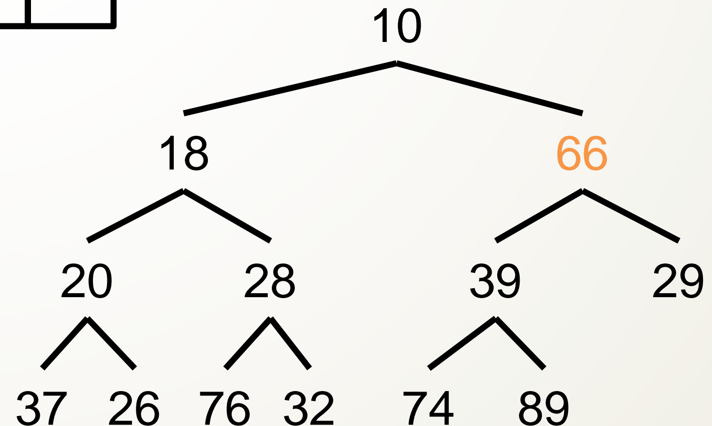
Swap table[parent] and table[minChild]

Set parent to minChild

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	66	20	28	39	29	37	26	76	32	74	89

```
parent = minChild;   // 2
```



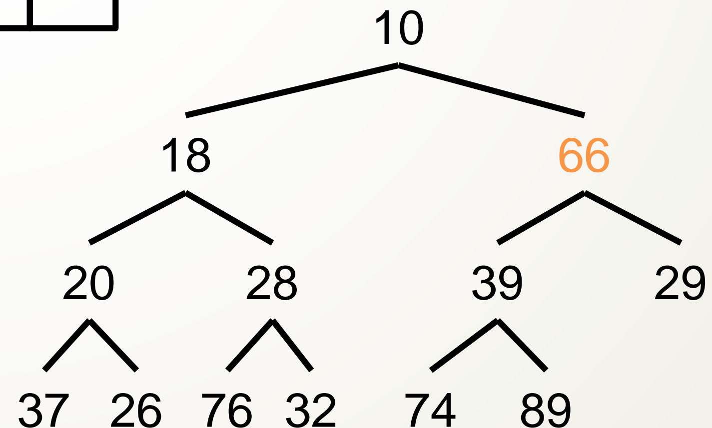
Removal

```
while (true)
        ...
        if table[parent] > table[minChild]
          Swap table[parent] and table[minChild]
          Set parent to minChild
        else
          break out of loop
```

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	66	20	28	39	29	37	26	76	32	74	89

```
parent = minChild;    // 2
```



Removal

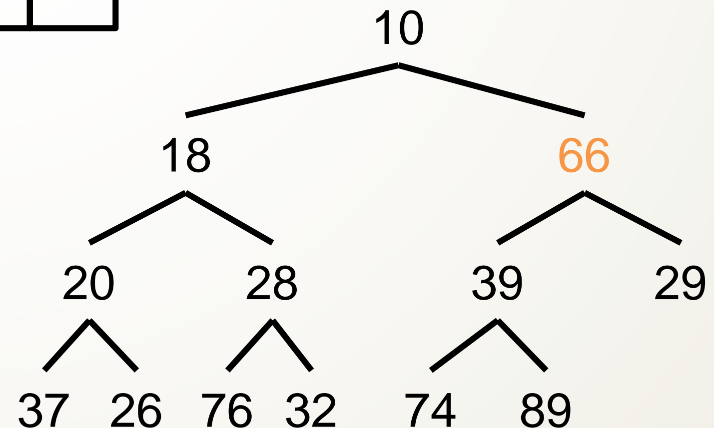
while (true)

...
Set leftChild to (2 * parent) + 1 and rightChild to leftChild + 1
if leftChild >= table.size() break out of loop

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	66	20	28	39	29	37	26	76	32	74	89

```
parent = minChild;   // 2
leftChild = 2 * parent + 1;   // 5
rightChild = leftChild + 1;   // 6
```



Removal

while (true)

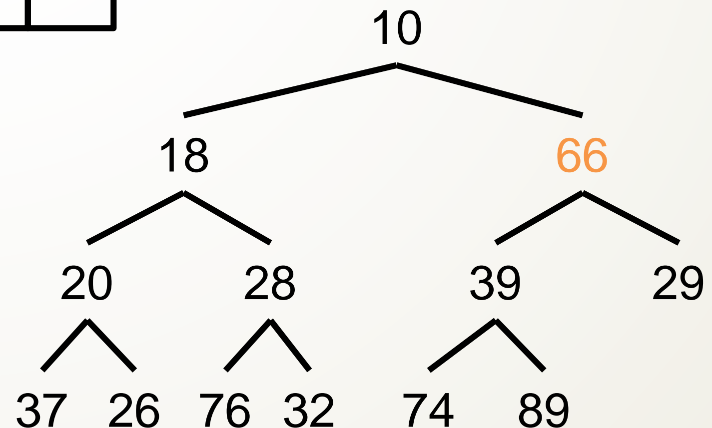
...

Set minChild to index of smaller child

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	66	20	28	39	29	37	26	76	32	74	89

```
parent = minChild;   // 2
leftChild = 2 * parent + 1;   // 5
rightChild = leftChild + 1;   // 6
minChild = 6;
```



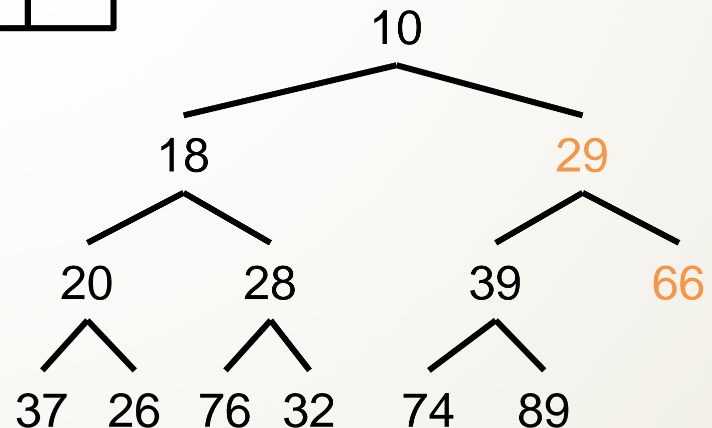
Removal

```
while (true)
        ...
        if table[parent] > table[minChild]
          Swap table[parent] and table[minChild]
```

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	29	20	28	39	66	37	26	76	32	74	89

```
parent = minChild;   // 2
leftChild = 2 * parent + 1;   // 5
rightChild = leftChild + 1;   // 6
minChild = 6;
```



Removal

while (true)

...

if table[parent] > table[minChild]

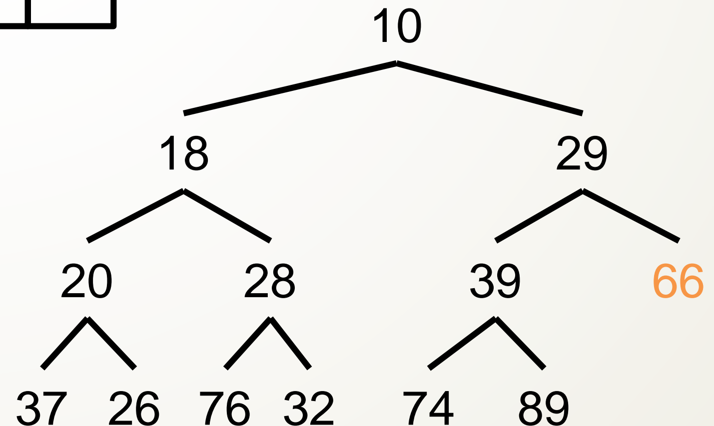
Swap table[parent] and table[minChild]

Set parent to minChild

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	29	20	28	39	66	37	26	76	32	74	89

```
parent = minChild;   // 6
```



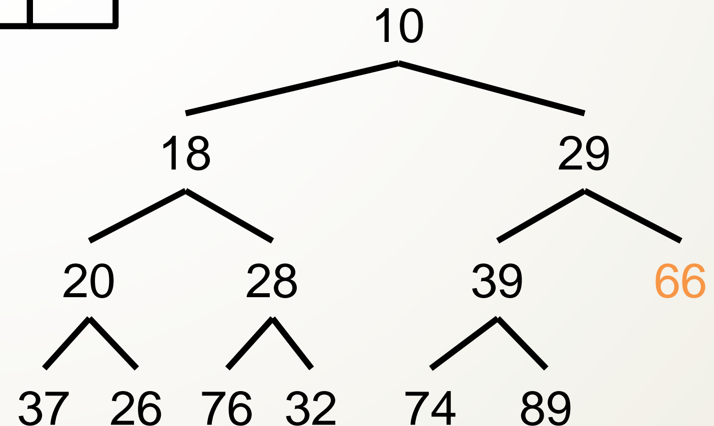
Removal

```
while (true)
    ...
    if table[parent] > table[minChild]
      Swap table[parent] and table[minChild]
      Set parent to minChild
    else
      break out of loop
```

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	29	20	28	39	66	37	26	76	32	74	89

```
parent = minChild;    // 6
```



Removal

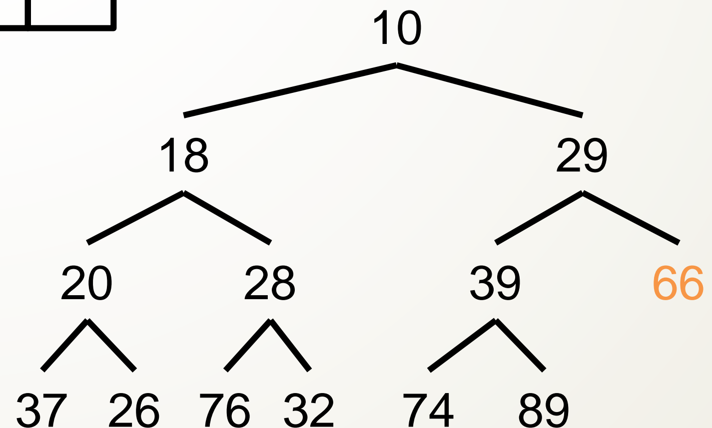
while (true)

> **Set leftChild to (2 * parent) + 1 and rightChild to leftChild + 1**
>
> **if leftChild >= table.size() break out of loop**

table

0	1	2	3	4	5	6	7	8	9	10	11	12
10	18	29	20	28	39	66	37	26	76	32	74	89

```
parent = minChild;   // 6
leftChild = 2 * parent + 1;   // 13
rightChild = leftChild + 1;   // 14
```



Priority Queue

Queues are FIFO

Priority Queue

Queues are FIFO

Priority Queues the position in the queue is determined by a priority

Priority Queue

Queues are FIFO

Priority Queues the position in the queue is determined by a priority

- Each insertion can reorder the queue

Priority Queue

Queues are FIFO

Priority Queues the position in the queue is determined by a priority

- Each insertion can reorder the queue
- The highest/lowest priority item is at the front

Priority Queue

Queues are FIFO

Priority Queues the position in the queue is determined by a priority

- Each insertion can reorder the queue
- The highest/lowest priority item is at the front
- Are not FIFO queues

Summary

Heaps are an efficient way of sorting values

Provide a way of implementing Priority Queues