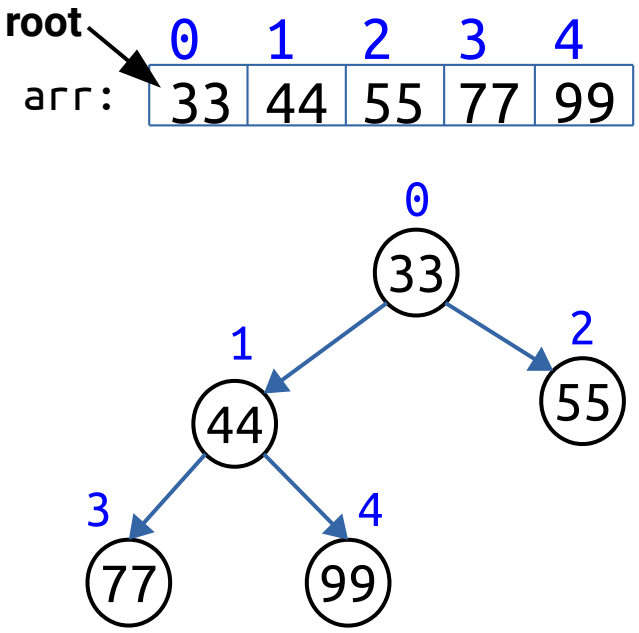
Heaps

A *heap* is a *complete* binary tree.

Max heaps – for any node *C*, if *P* is parent of *C*, *then P*.value ⩾ *C*.value
Min heaps – for any node *C*, if *P* is parent of *C*, *then P*.value ⩽ *C*.value

In this seminar, we will consider **Array-based Min heaps**




```
int leftChild(int i)
{
    return 2*i + 1;
}

int rightChild(int i)
{
    return 2*i + 2;
}
```

Heaps - operations

In this seminar, we will consider the following operations.

1. **heapify** – Re-arrange a heap to ensure the heap property.

2. **buildMinHeap –** Build a heap with an input (possibly unsorted) array.

3. **pop (extractMin) –** Get the root (min value) and remove it from the heap.

4. **heapsort –** sorting an array using a heap.

1. Heapify

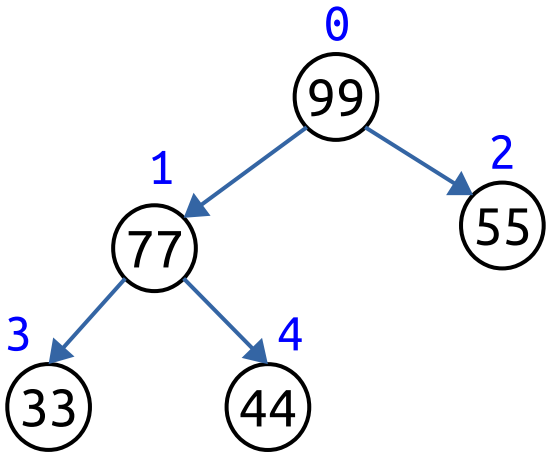
```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

1. Heapify

```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

heapify(arr,i = 1);

	0	1	2	3	4
arr:	99	77	55	33	44

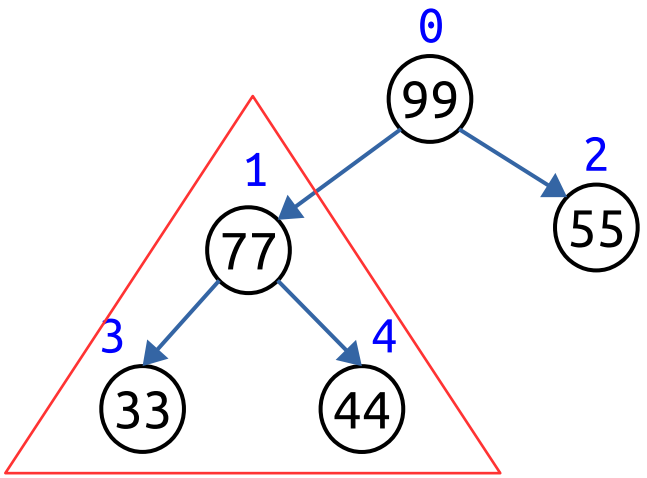


1. Heapify

```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

heapify(arr,i = 1);

	0	1	2	3	4
arr:	99	77	55	33	44

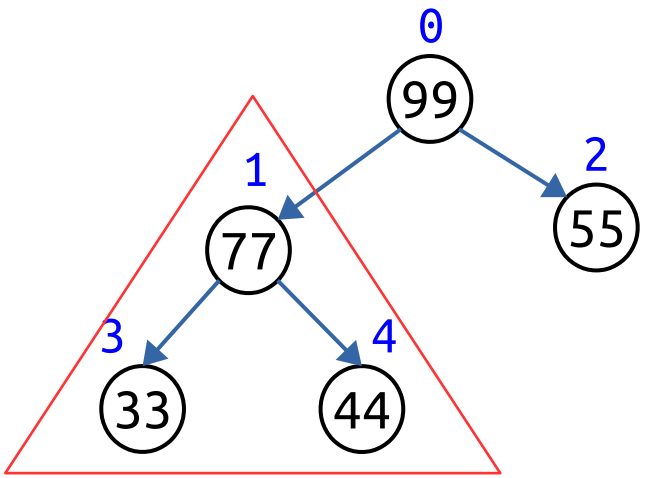


1. Heapify

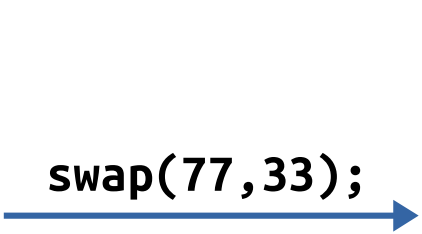
```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

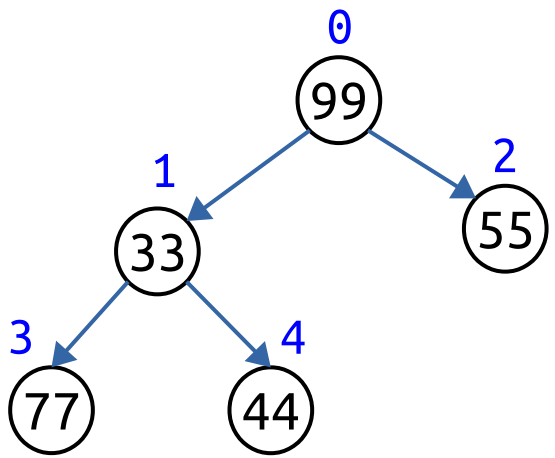
heapify(arr,i = 1);

	0	1	2	3	4
arr:	99	77	55	33	44



swap(77,33);





heapify(arr,i = 3);

	0	1	2	3	4
arr:	99	33	55	77	44

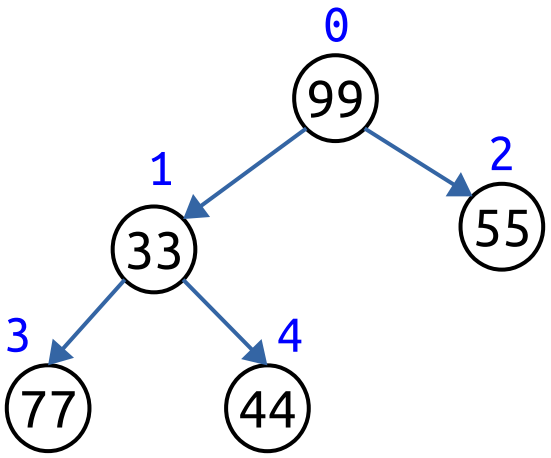
The sub-tree with root node 1 is now a heap ☺

1. Heapify

```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

heapify(arr,i = 0);

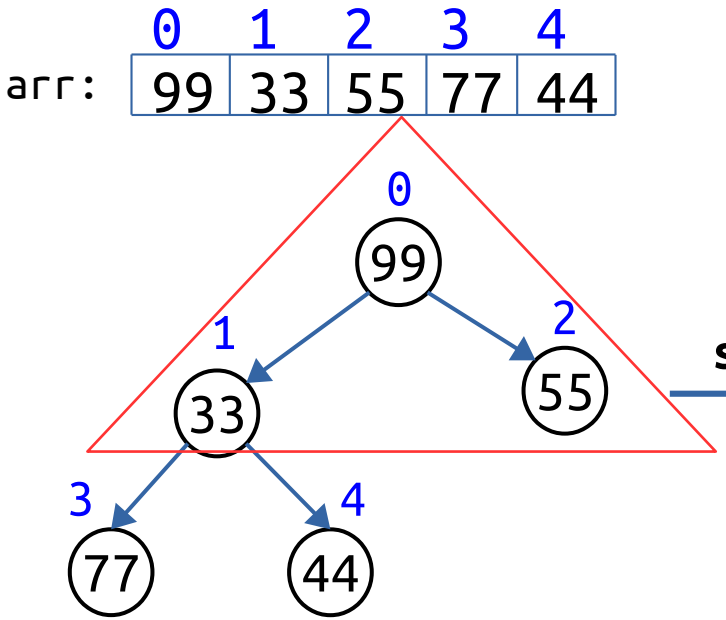
	0	1	2	3	4
arr:	99	33	55	77	44



1. Heapify

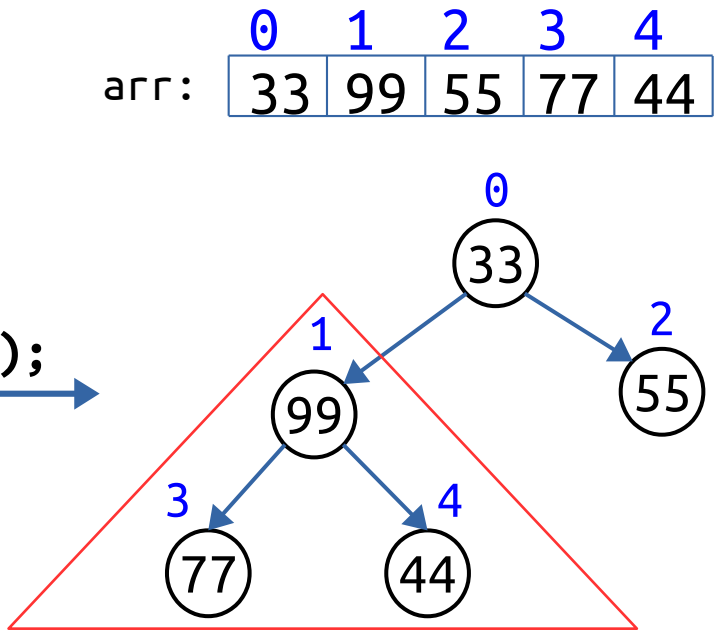
```
1  // re-arranges the "sub-tree" v with root "i".
2
3  void heapify(std::vector<T>& v, int i)
4
5      get l = leftChild(i);
6      get r = rightChild(i);
7
8      note: do not consider "l" or "r" if they are greater than v.size() - 1;
9
10     let "min" be the minimum value between v[l], v[r], v[i];
11
12     if(min is not equal to node i) then // value of a child of "i" is less than "i"
13         swap(v[i], v[min]);             // swap value between the parent and child.
14         heapify(v, min);                // re-arrange the sub-tree whose root is a child of "i"
15     end_if;
16 end_void;
```

heapify(arr,i = 0);

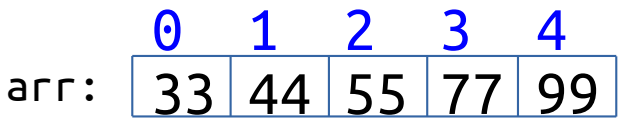


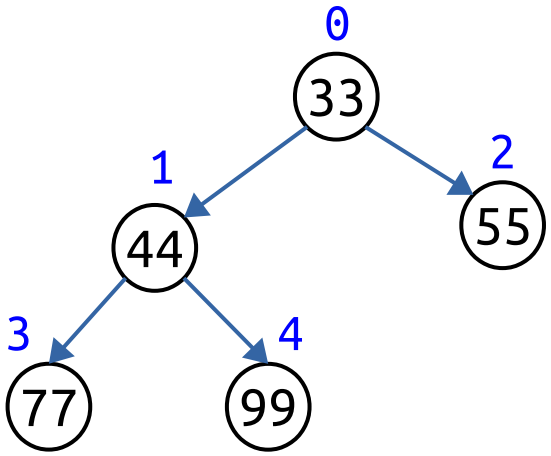
swap(99,33);

heapify(arr,i = 1);



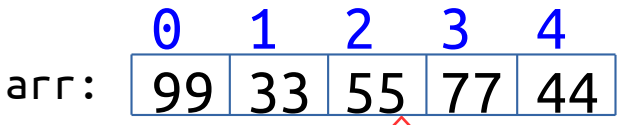
1. Heapify

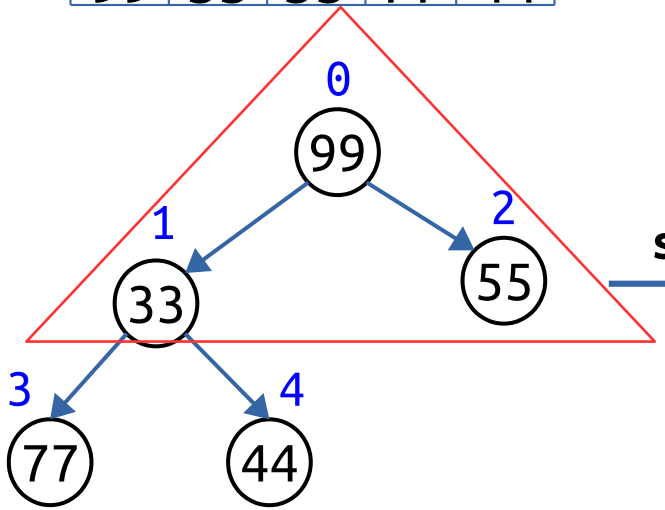





```
swap(99,44);
heapify(arr,i = 4);
```

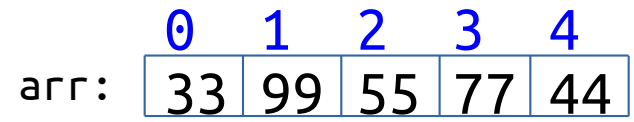
```
heapify(arr,i = 0);
```

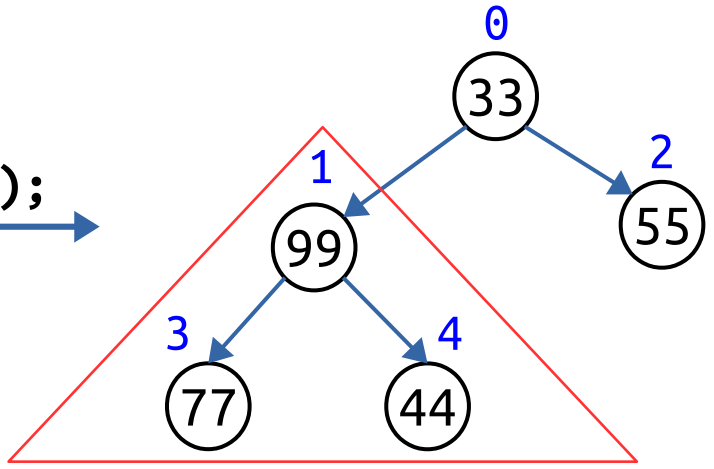





```
swap(99,33);
```

```
heapify(arr,i = 1);
```





Heaps - operations

In this seminar, we will consider the following operations.

1. **heapify** – Re-arrange a heap to ensure the heap property.

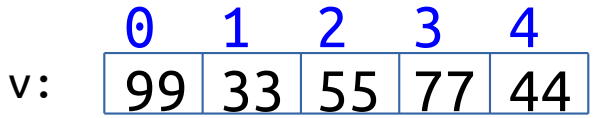
2. **buildMinHeap –** Build a heap with an input (possibly unsorted) array.

3. **pop (extractMin) –** Get the root (min value) and remove it from the heap.

4. **heapsort –** sorting an array using a heap.

2. Build Min Heap

Build a *heap* arr using a vector v as input



The heap has a property that states that nodes in positions `[n/2...n-1]` are leaves. We do not need to *heapify* in leaves of a tree.

2. Build Min Heap

Build a *heap* arr using a vector v as input

	0	1	2	3	4
v:	99	33	55	77	44

The heap has a property that states that
nodes in positions `[n/2...n-1]` are leaves.
We do not need to *heapify* in leaves of a tree,

Heapify sub-trees `[0 , … , (n/2) - 1 ]` from bottom to top.

```
1  // builds heap arr using an input vector v of n elements
2  void buildMinHeap()
3      for(int i = (n/2) - 1; i >= 0; i--)
4          heapify(arr, i);
5      end_for;
6  end_void;
```

Heaps - operations

In this seminar, we will consider the following operations.

1. **heapify** – Re-arrange a heap to ensure the heap property.

2. **buildMinHeap –** Build a heap with an input (possibly unsorted) array.

3. **pop (extractMin) –** Get the root (min value) and remove it from the heap.

4. **heapsort –** sorting an array using a heap.

3. pop() - extractMin

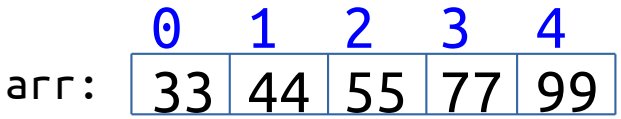
Get the root (min value) and remove it from the heap

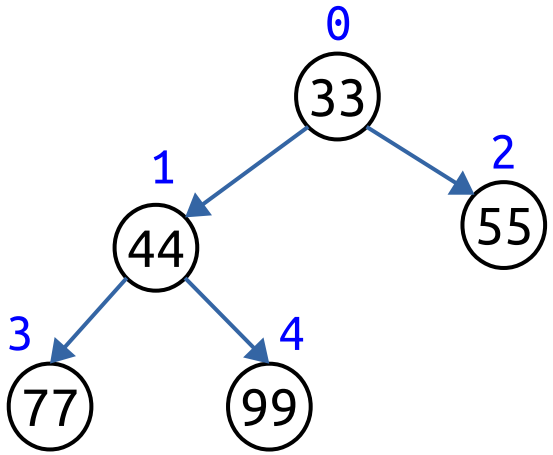
```
1  pop()
2  {
3      tmp <- get a copy of the root;
4
5      erase the root;
6
7      if the heap is not empty then
8          remove the last element in the heap arr and put it in the root.
9          heapify heap arr from position 0.
10     end_if
11
12     return tmp;
13 }
```

3. pop() - extractMin

Get the root (min value) and remove it from the heap

```
1  pop()
2  {
3      tmp <- get a copy of the root;
4
5      erase the root;
6
7      if the heap is not empty then
8          remove the last element in the heap arr and put it in the root.
9          heapify heap arr from position 0.
10     end_if
11
12     return tmp;
13 }
```

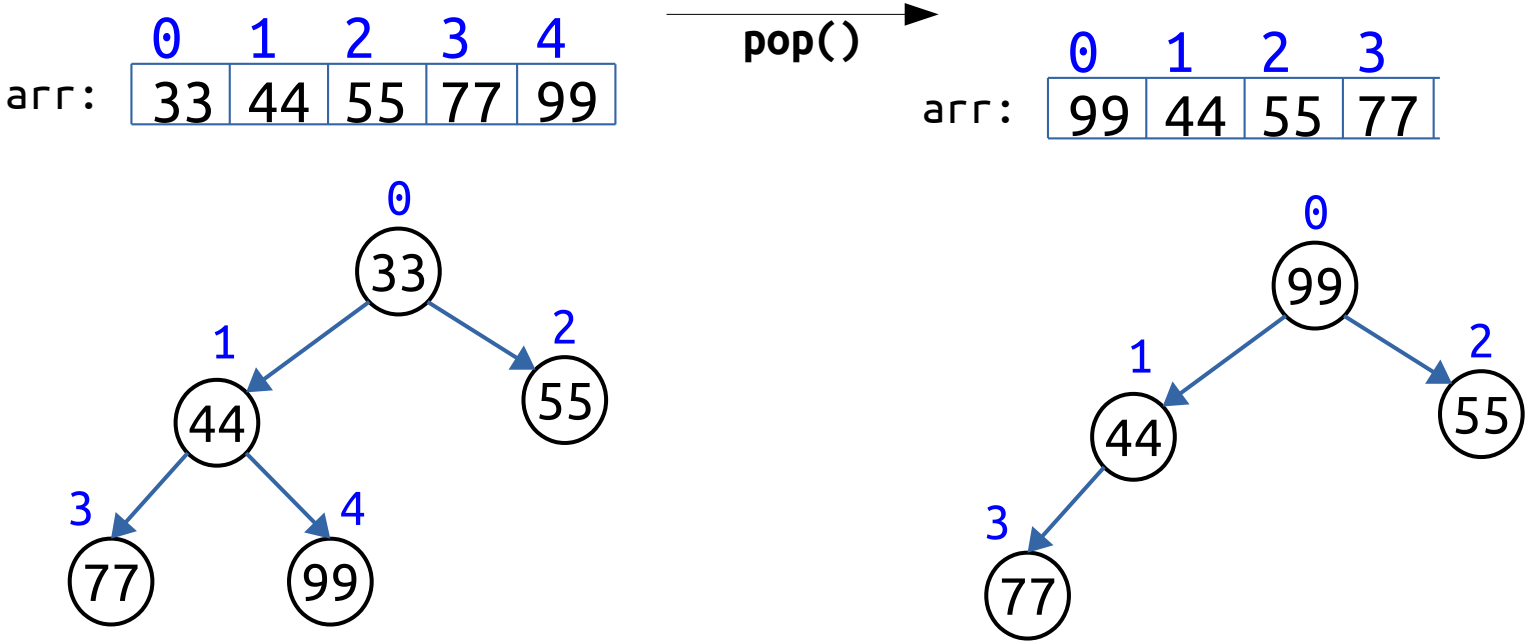




3. pop() - extractMin

Get the root (min value) and remove it from the heap

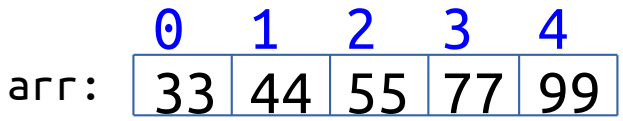
```
1  pop()
2  {
3      tmp <- get a copy of the root;
4
5      erase the root;
6
7      if the heap is not empty then
8          remove the last element in the heap arr and put it in the root.
9          heapify heap arr from position 0.
10     end_if
11
12     return tmp;
13 }
```



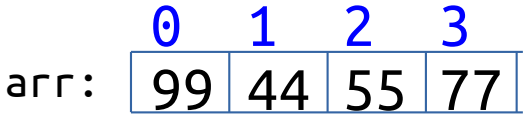
3. pop() - extractMin

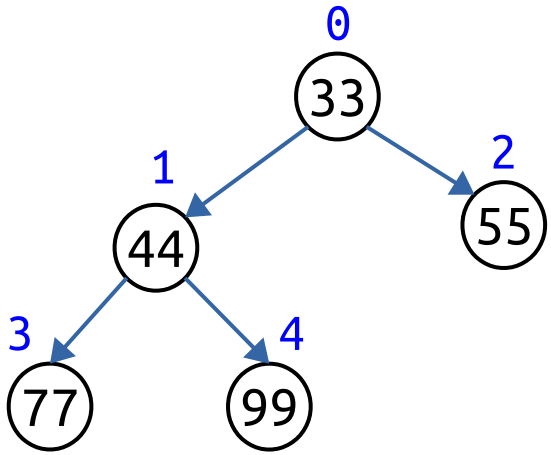
Get the root (min value) and remove it from the heap

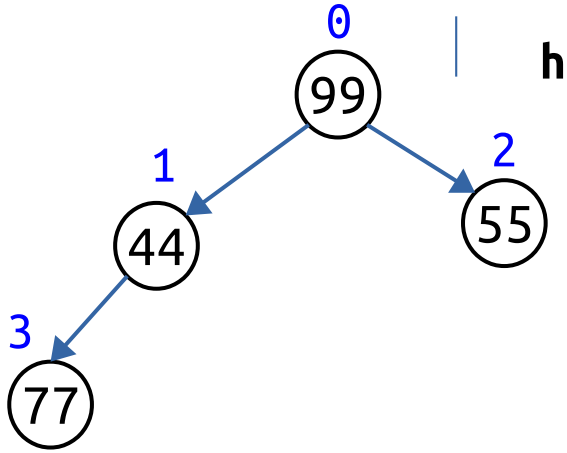
```
1  pop()
2  {
3      tmp <- get a copy of the root;
4
5      erase the root;
6
7      if the heap is not empty then
8          remove the last element in the heap arr and put it in the root.
9          heapify heap arr from position 0.
10     end_if
11
12     return tmp;
13 }
```



pop()





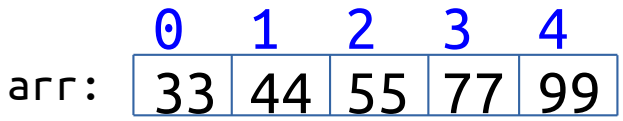



```
heapify(arr,i = 0);
swap(99,44);
heapify(arr,i = 1);
swap(99,77);
heapify(arr,i = 3);
```

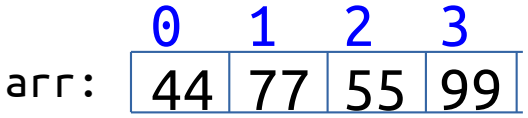
3. pop() - extractMin

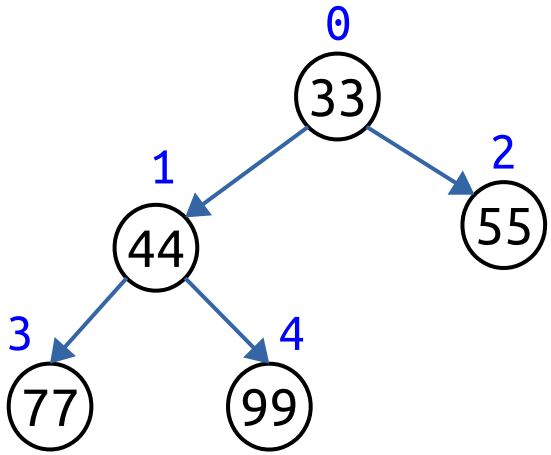
Get the root (min value) and remove it from the heap

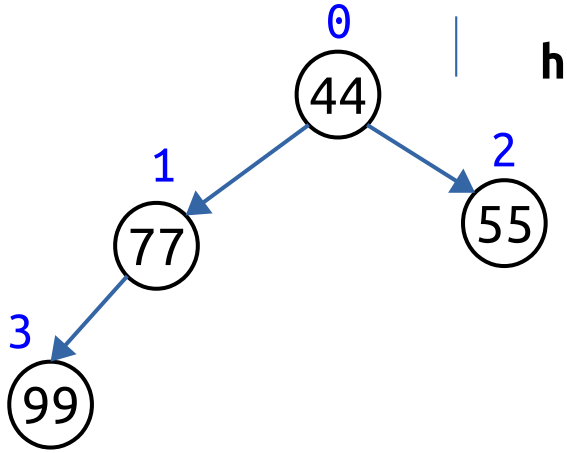
```
1  pop()
2  {
3      tmp <- get a copy of the root;
4
5      erase the root;
6
7      if the heap is not empty then
8          remove the last element in the heap arr and put it in the root.
9          heapify heap arr from position 0.
10     end_if
11
12     return tmp;
13 }
```



pop()








```
heapify(arr,i = 0);
swap(99,44);
heapify(arr,i = 1);
swap(99,77);
heapify(arr,i = 3);
```

4. Heapsort
Sort a vector v using a heap

4. Heapsort

Sort a vector v using a heap

```
1  // sort vector v
2  void heapsort(vector& v)
3  {
4      MinHeap heap(v);
5
6      for(int i = 0; i < v.size(); i++)
7          v[i] = heap.pop();
8  }
```