heap data structure

- complete binary tree
- priority queues (Chapter 9)
- binary heap and min-heap
- maxheap demo
- maxheap implementation
- **heap sort (Chapter 7)**

Heapsort

Basic plan for in-place sort

- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass**:

Heapsort

Basic plan for in-place sort

- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass**:

										N
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

Basic plan for in-place sort

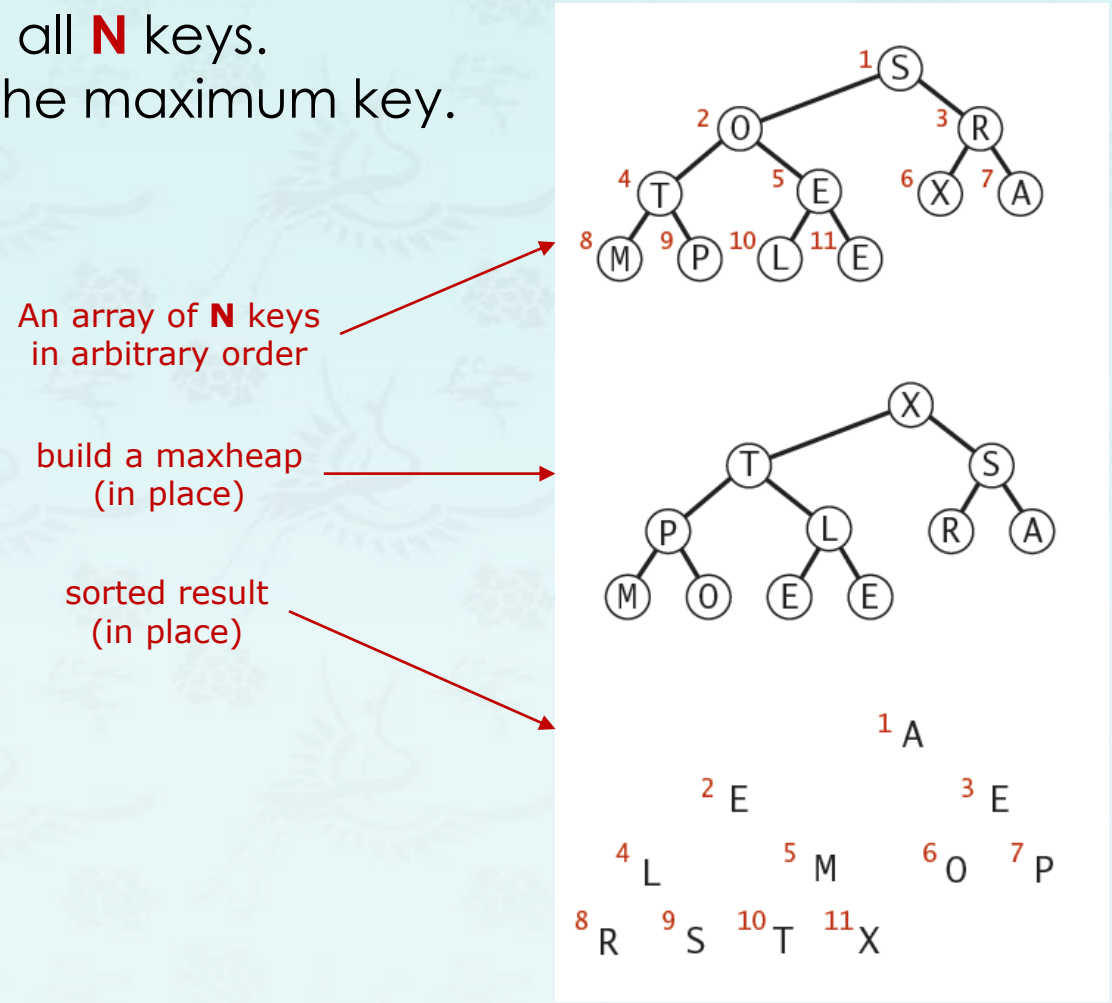
- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass**: Repeatedly remove the maximum key.

										N
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

Basic plan for in-place sort

- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass**: Repeatedly remove the maximum key.



Heapsort

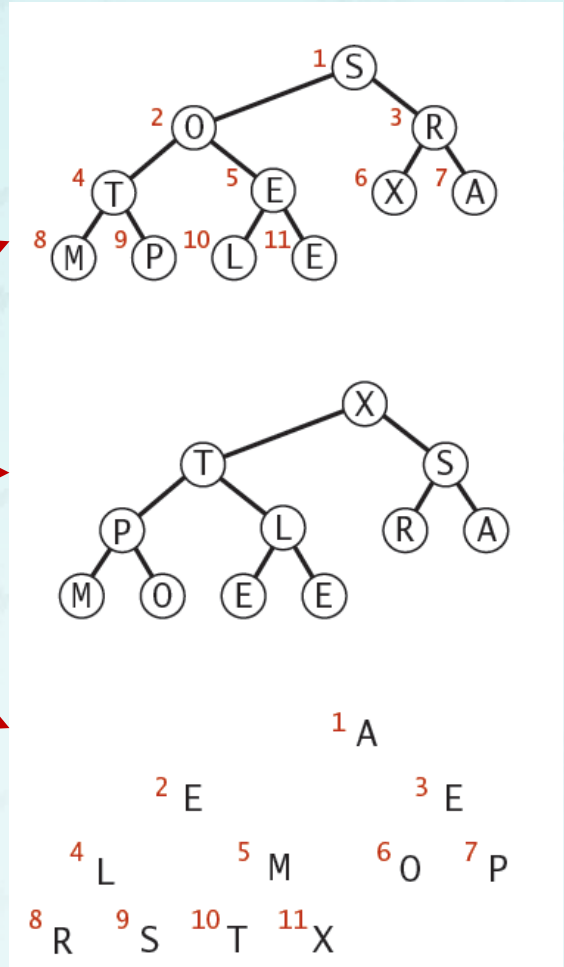
Basic plan for in-place sort

- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass**: Repeatedly remove the maximum key.

An array of **N** keys in arbitrary order

build a maxheap (in place)

sorted result (in place)

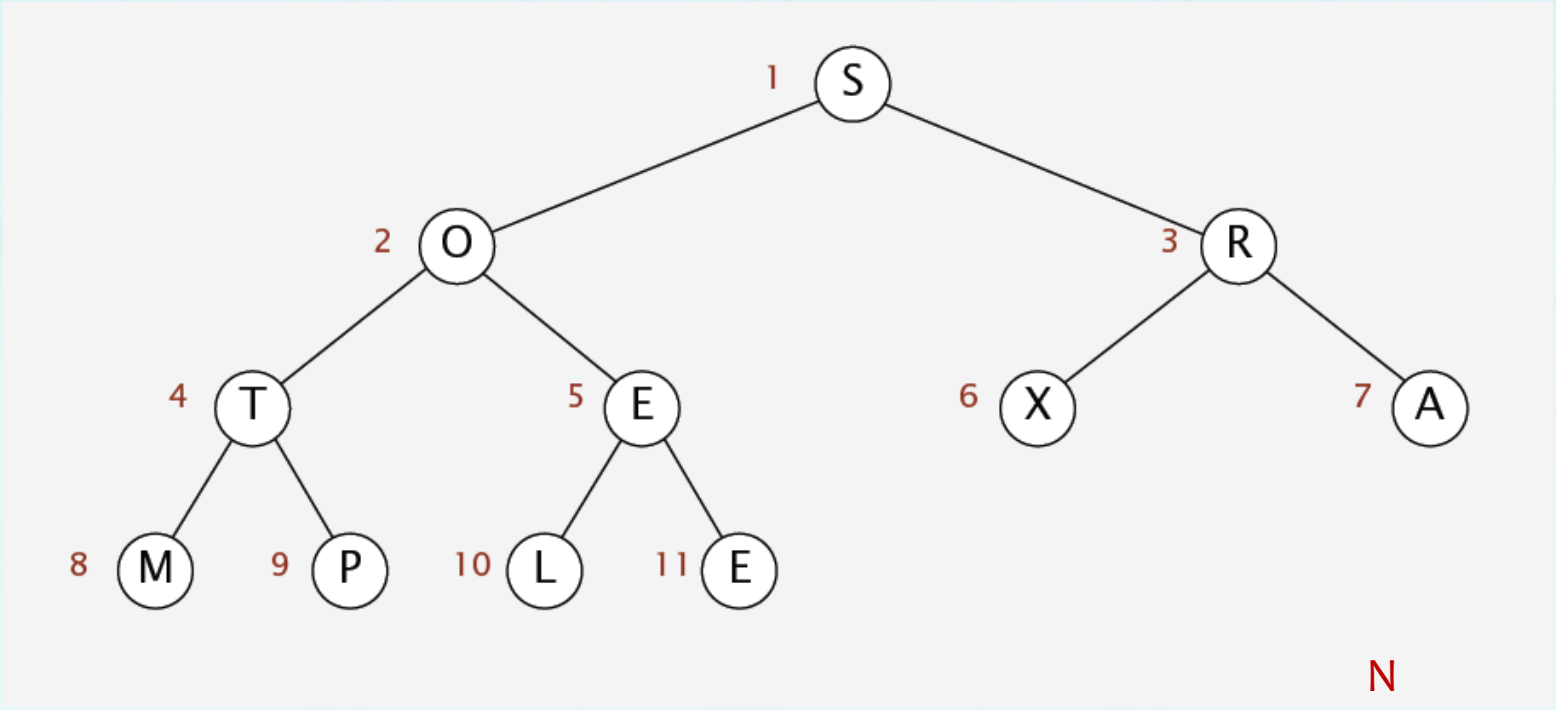


A	E	E	L	N	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

array in arbitrary order



N

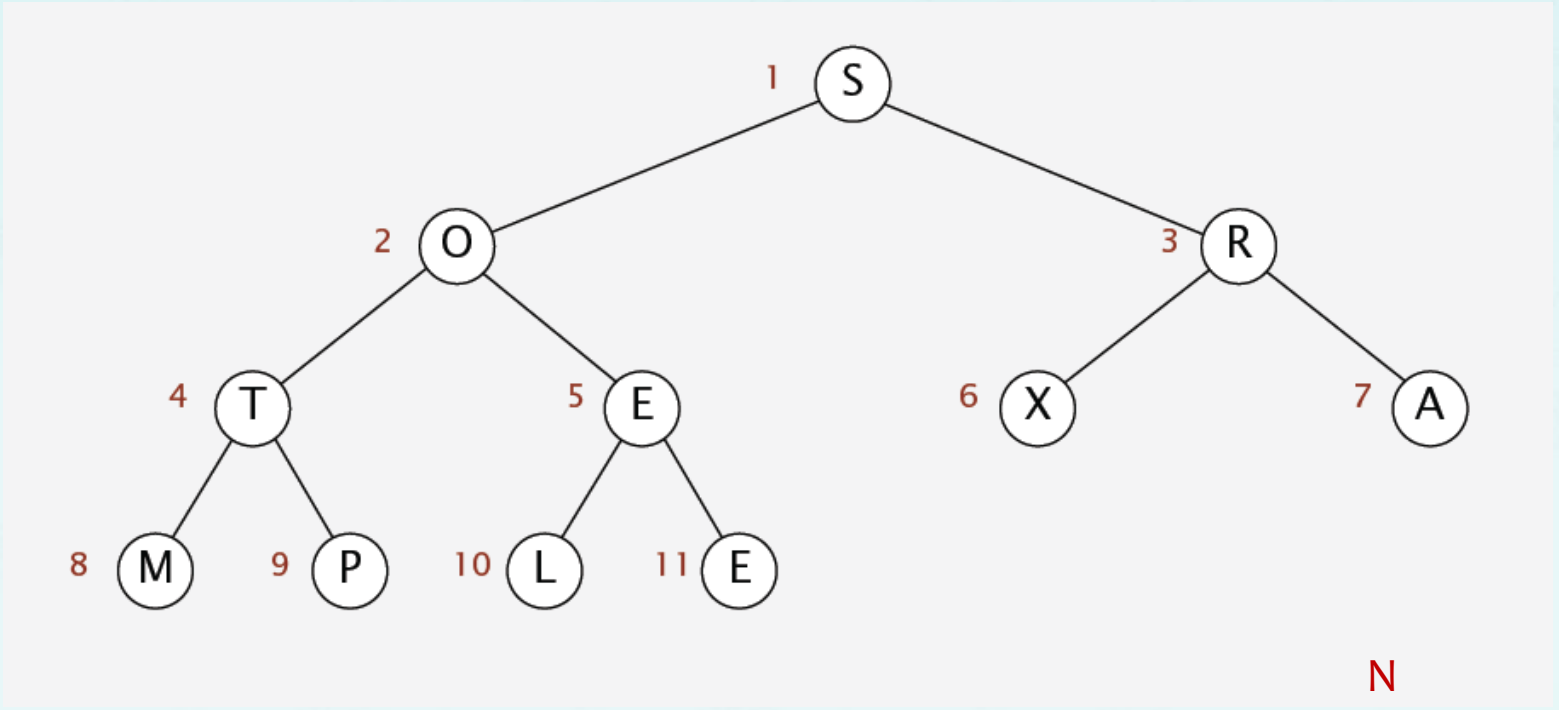
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

Where should we start from?

array in arbitrary order



N

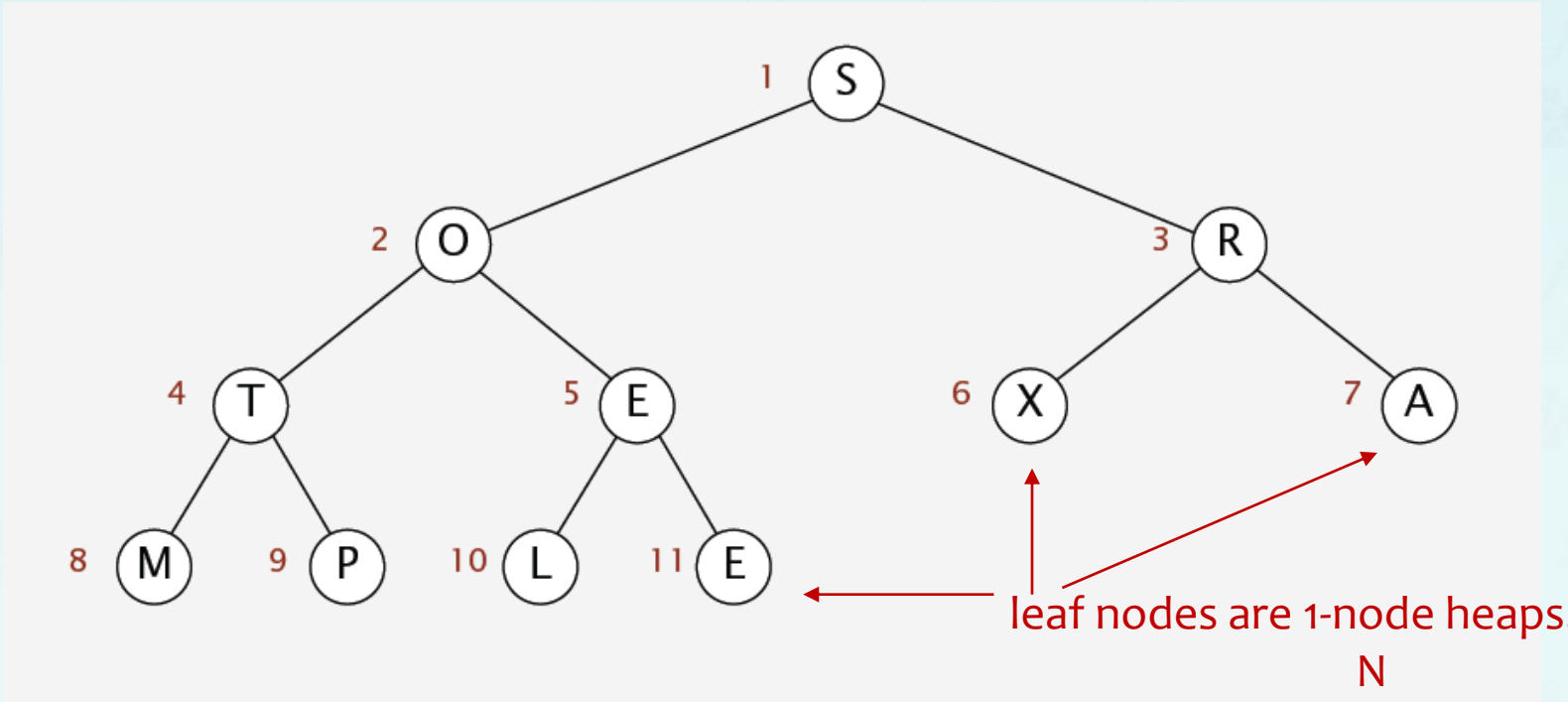
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

Where should we start from?

array in arbitrary order

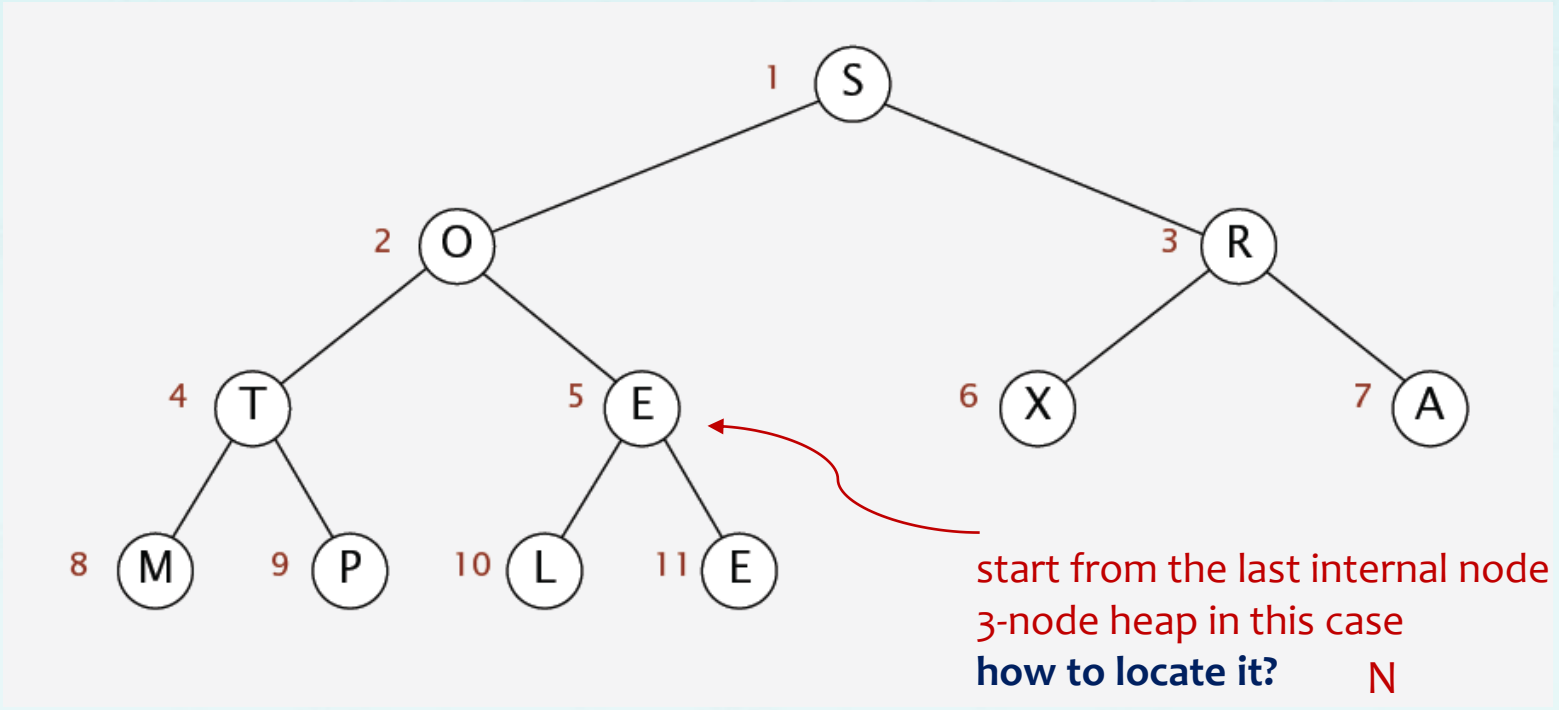


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

array in arbitrary order

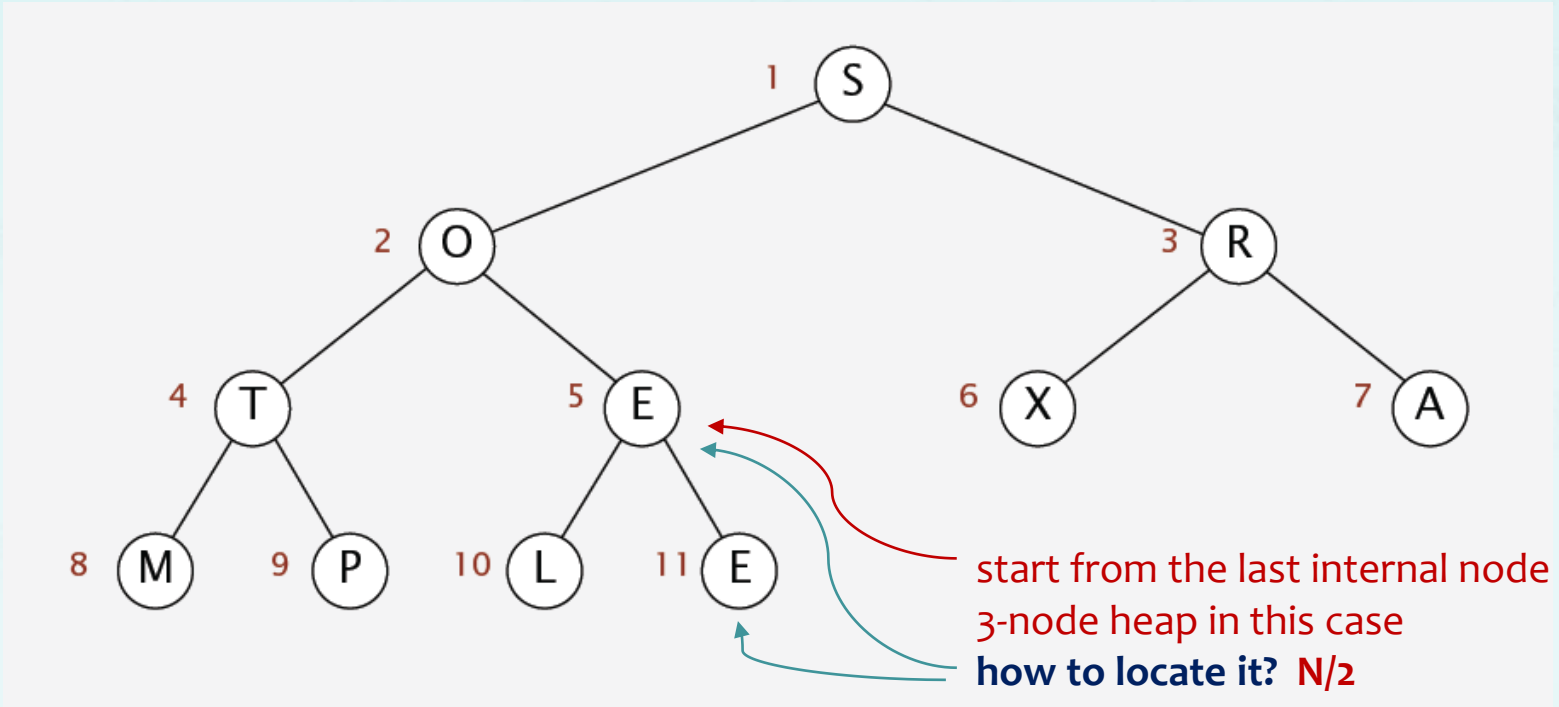


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

array in arbitrary order

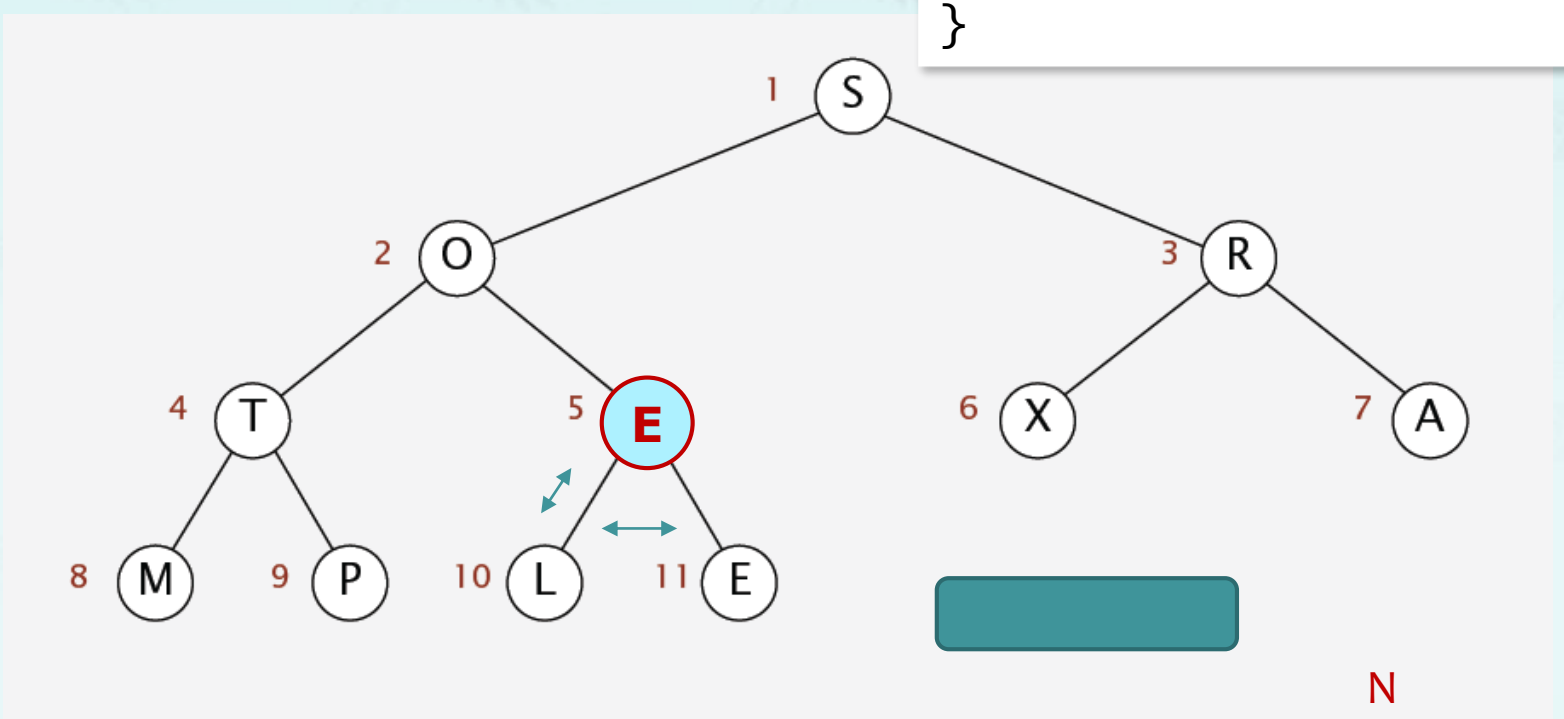


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

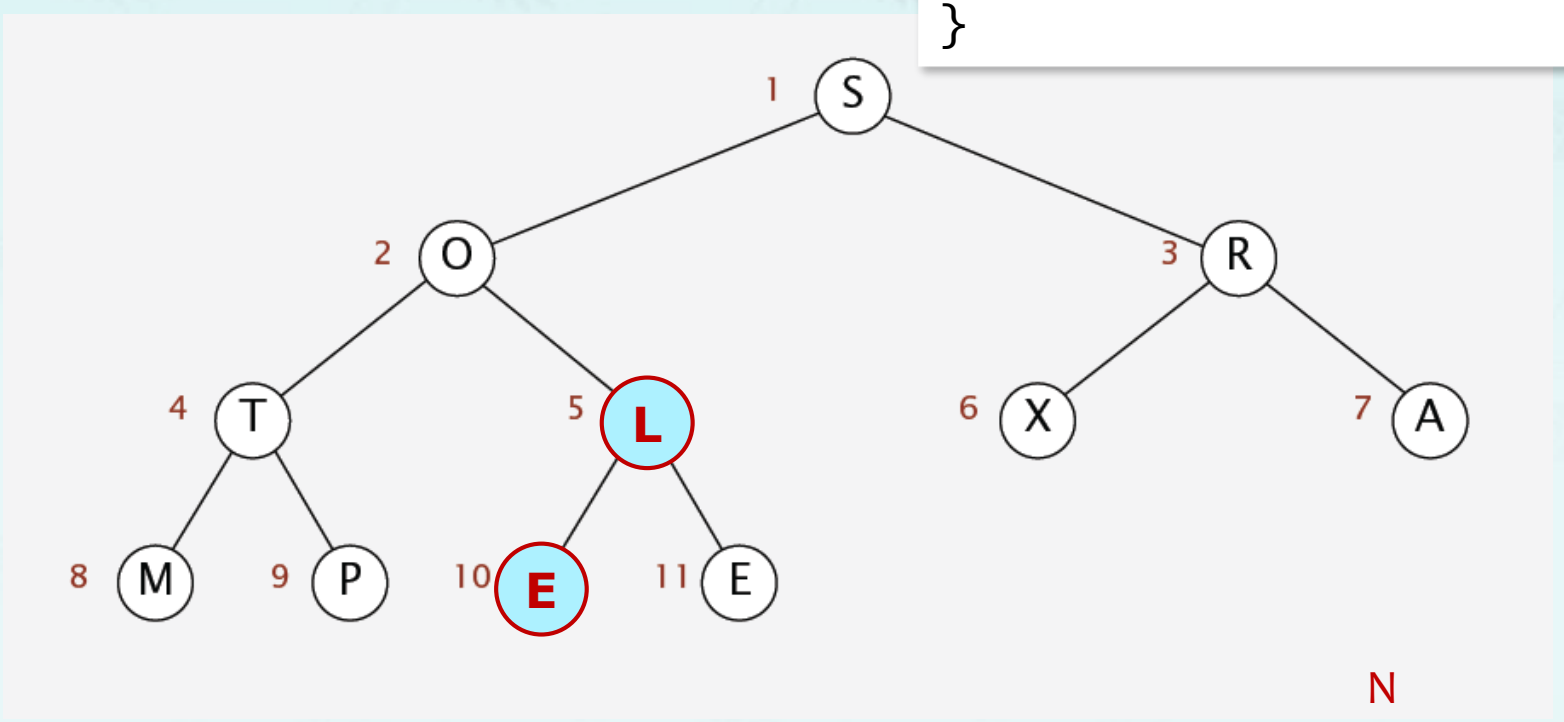


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```



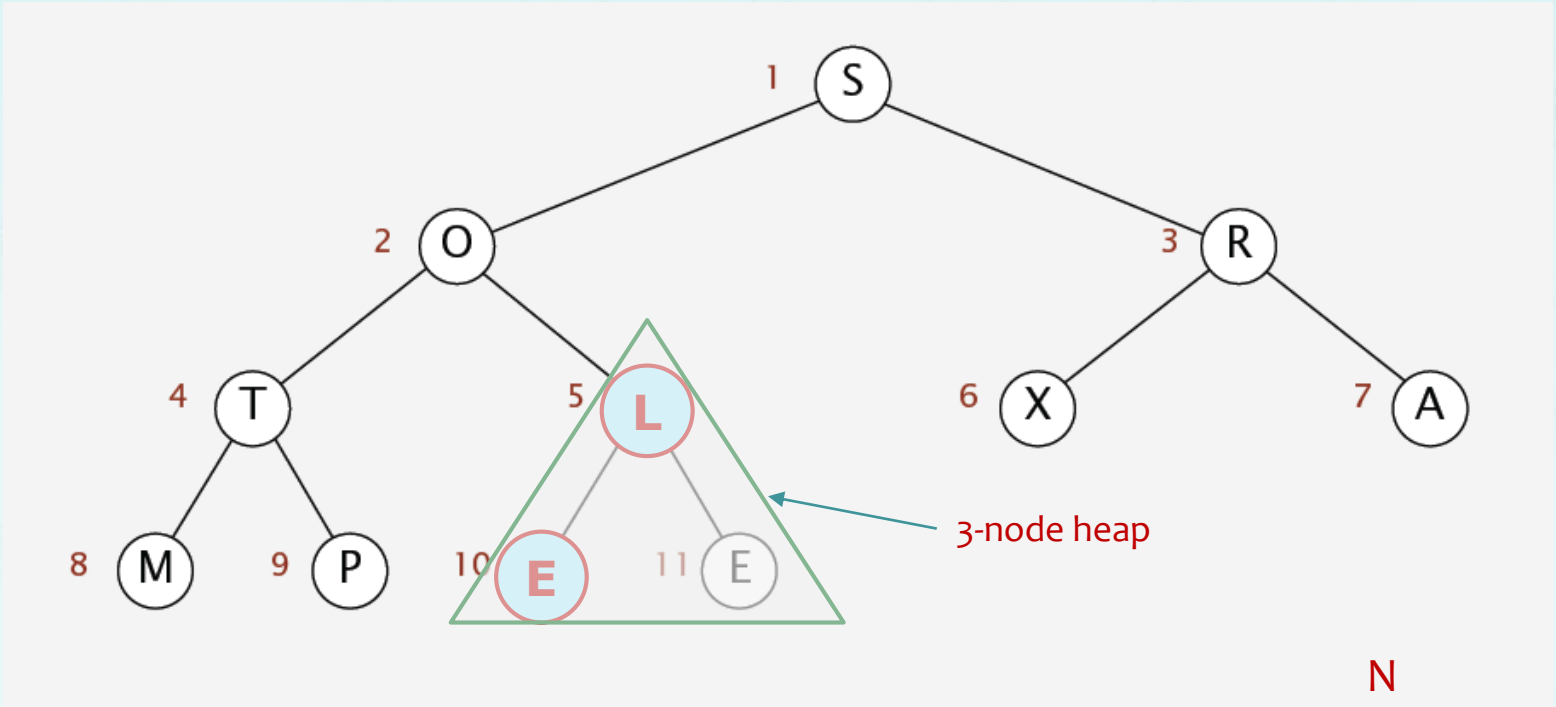
N

S	O	R	T	L	X	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

sink 5



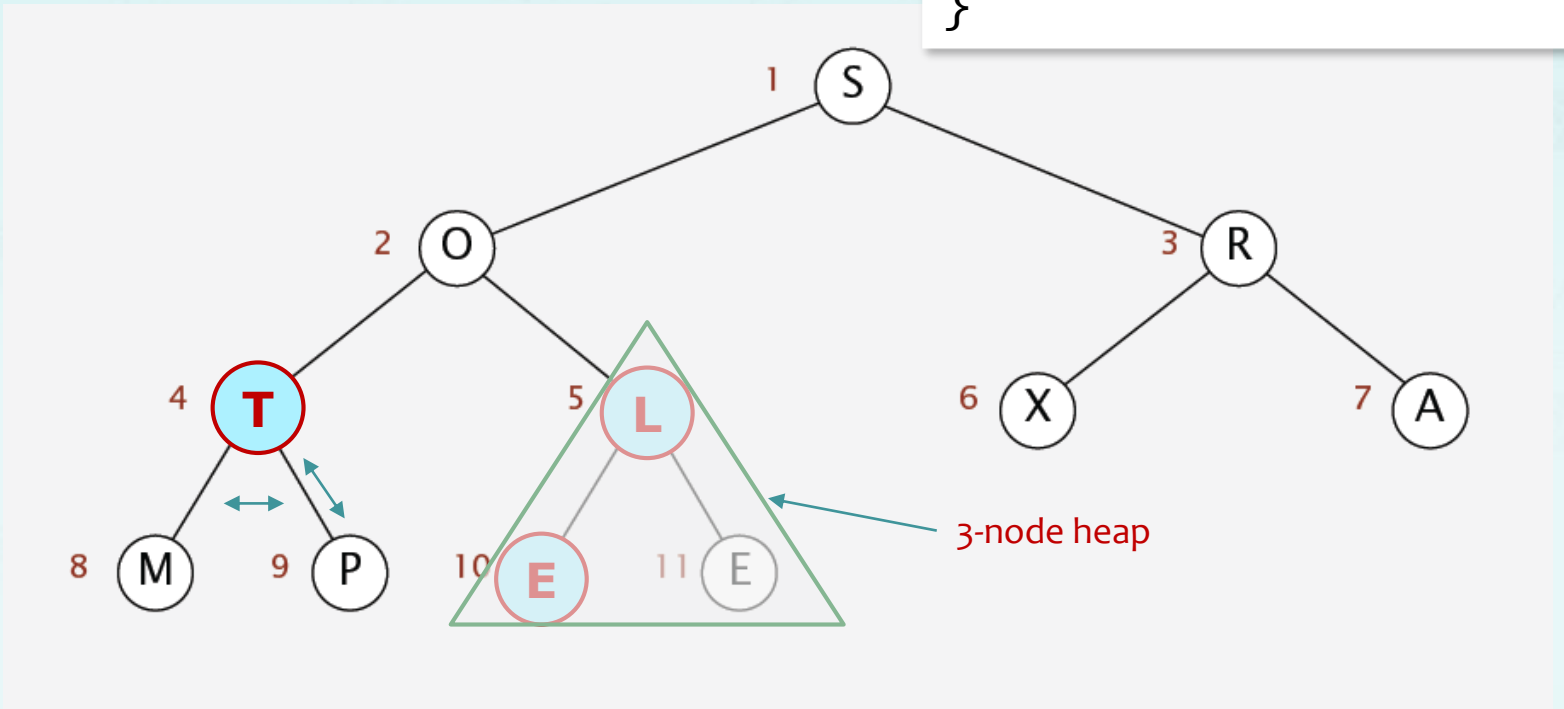
N

S	O	R	T	**L**	X	A	M	P	**E**	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```



S	O	R	T	L	X	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

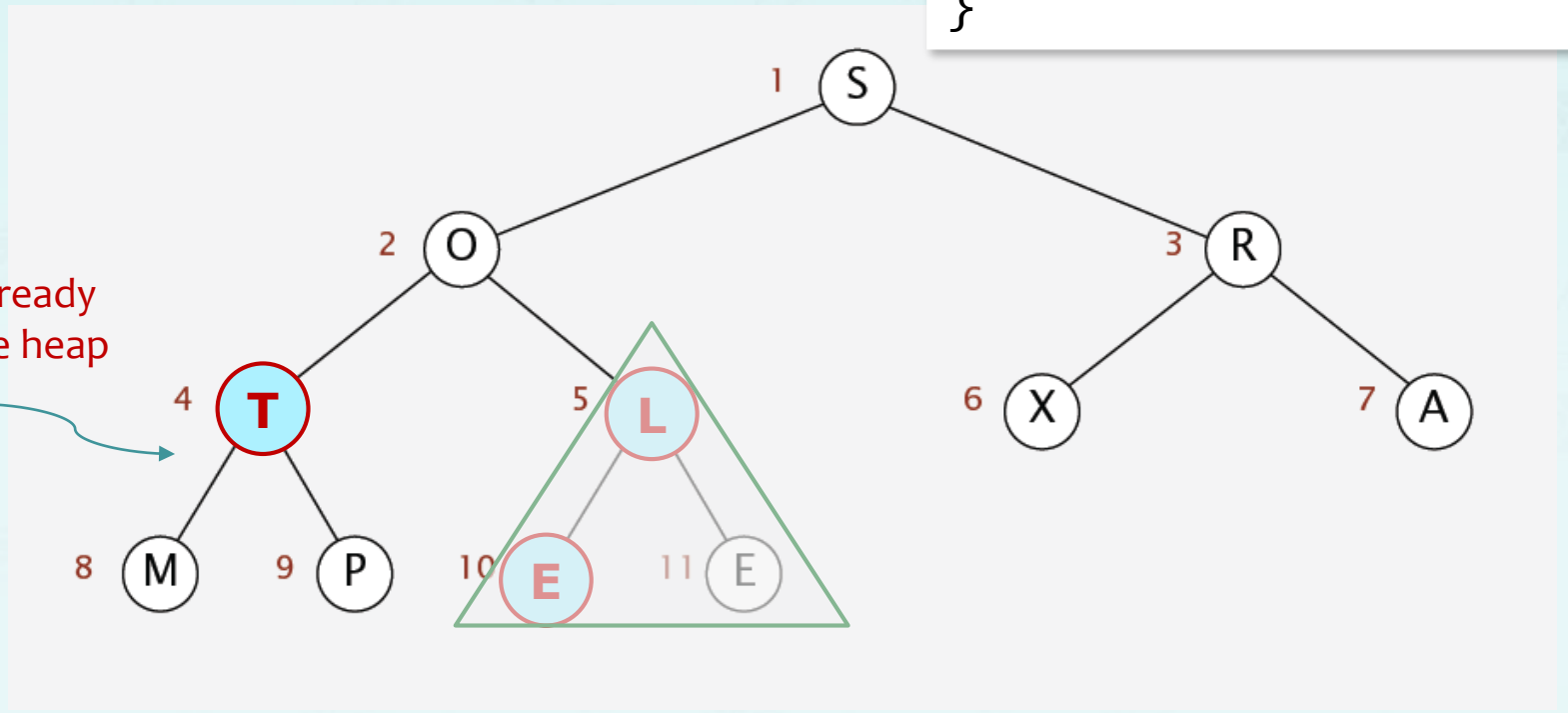
Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

sink 4

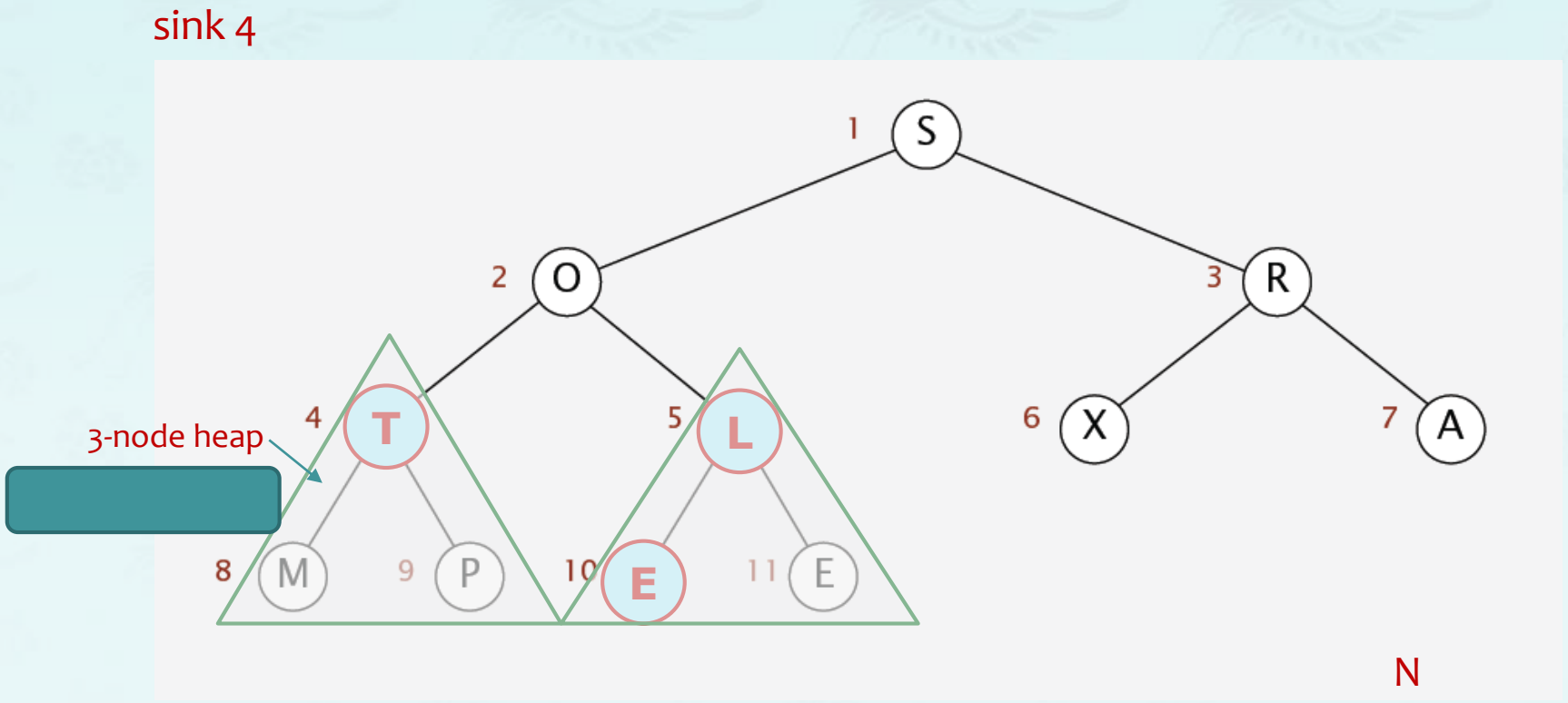
It's already
3-node heap



S	O	R	T	L	X	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

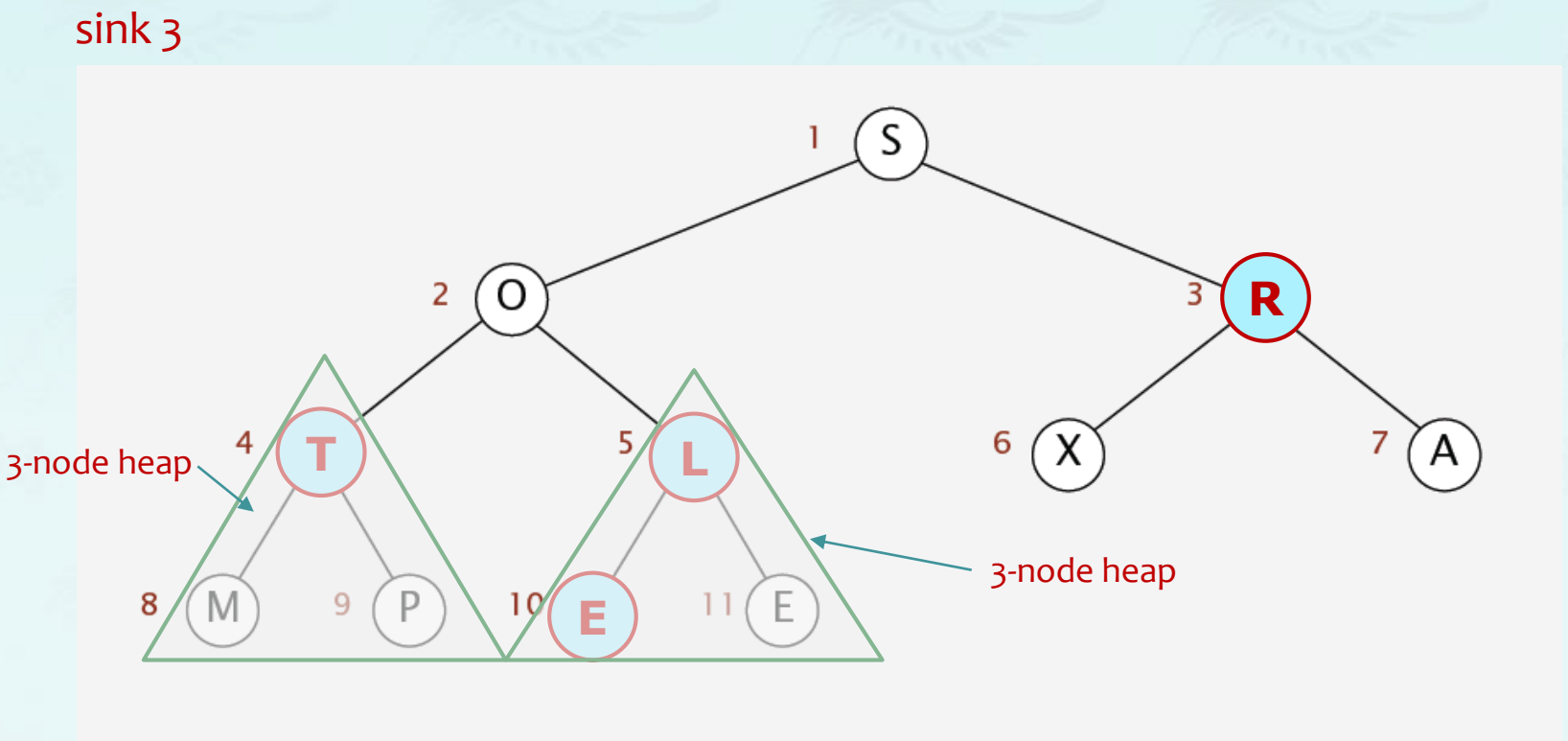
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	R	T	L	X	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



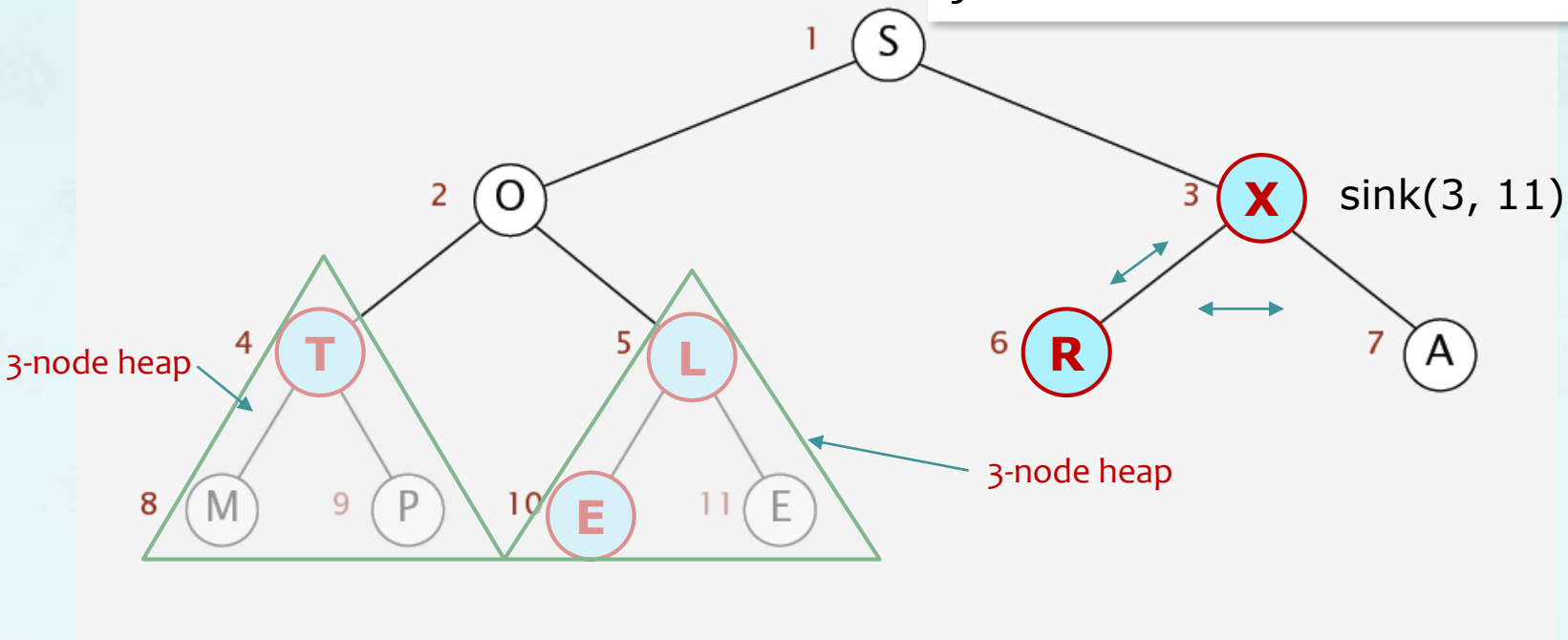
S	O	R	T	L	X	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

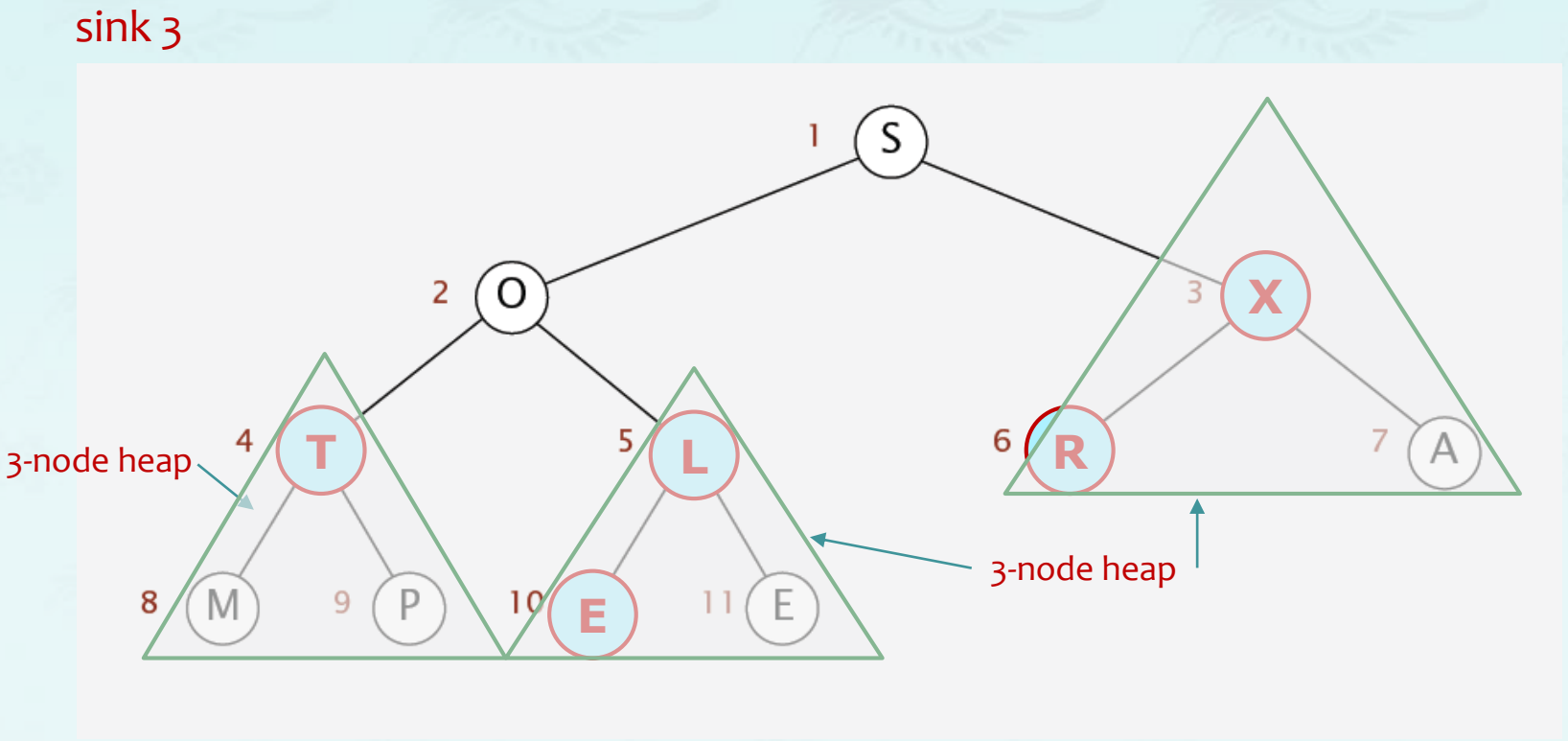
sink 3



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

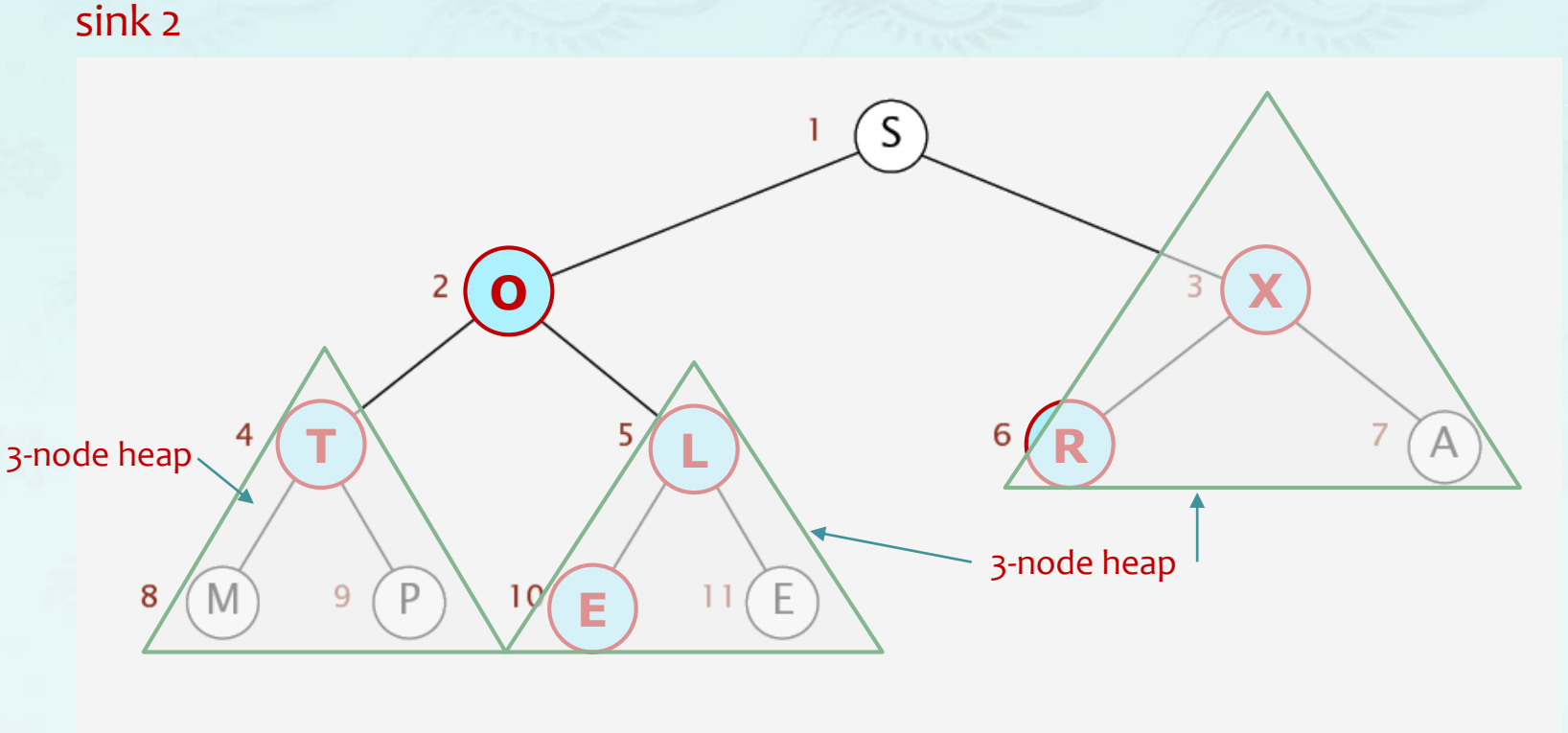
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

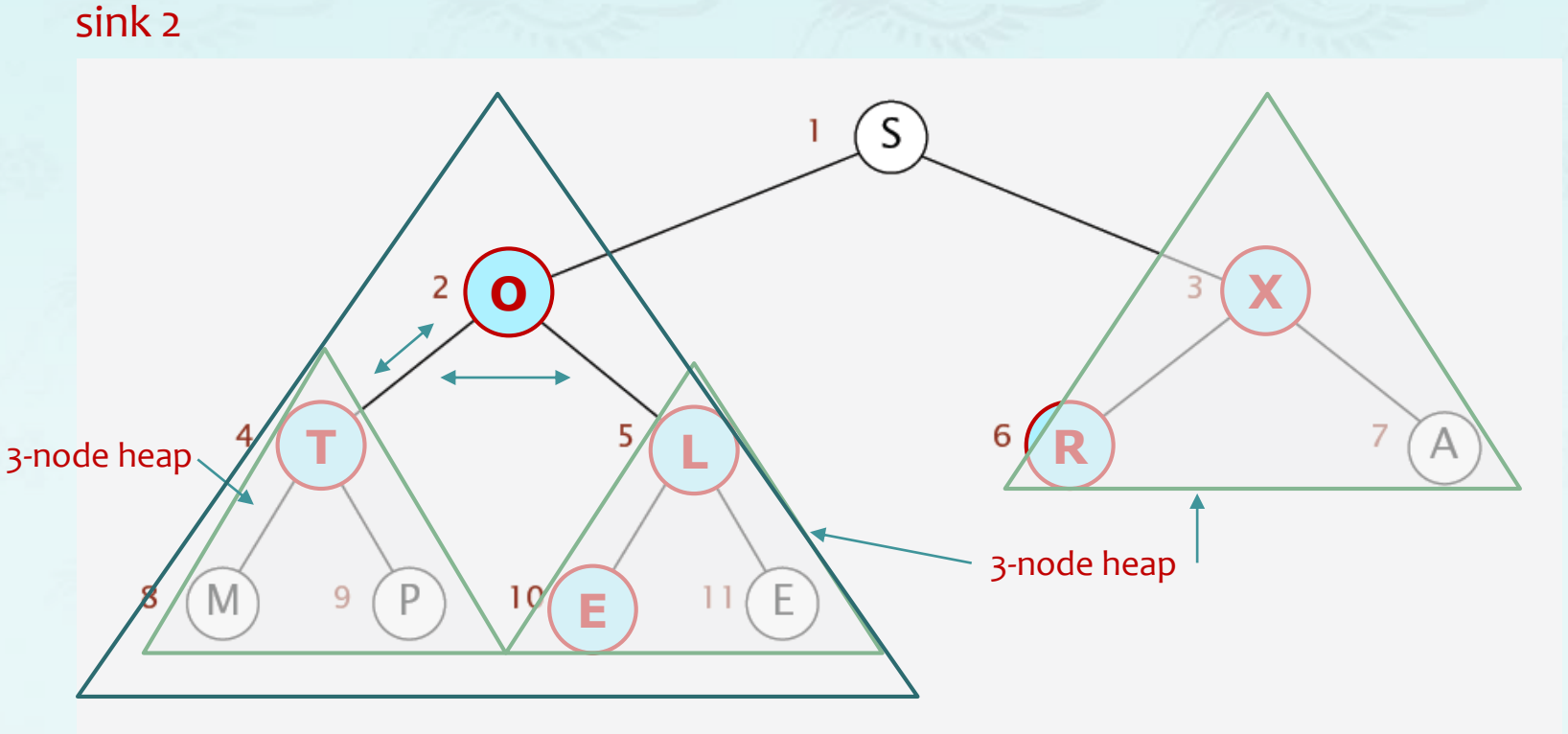
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

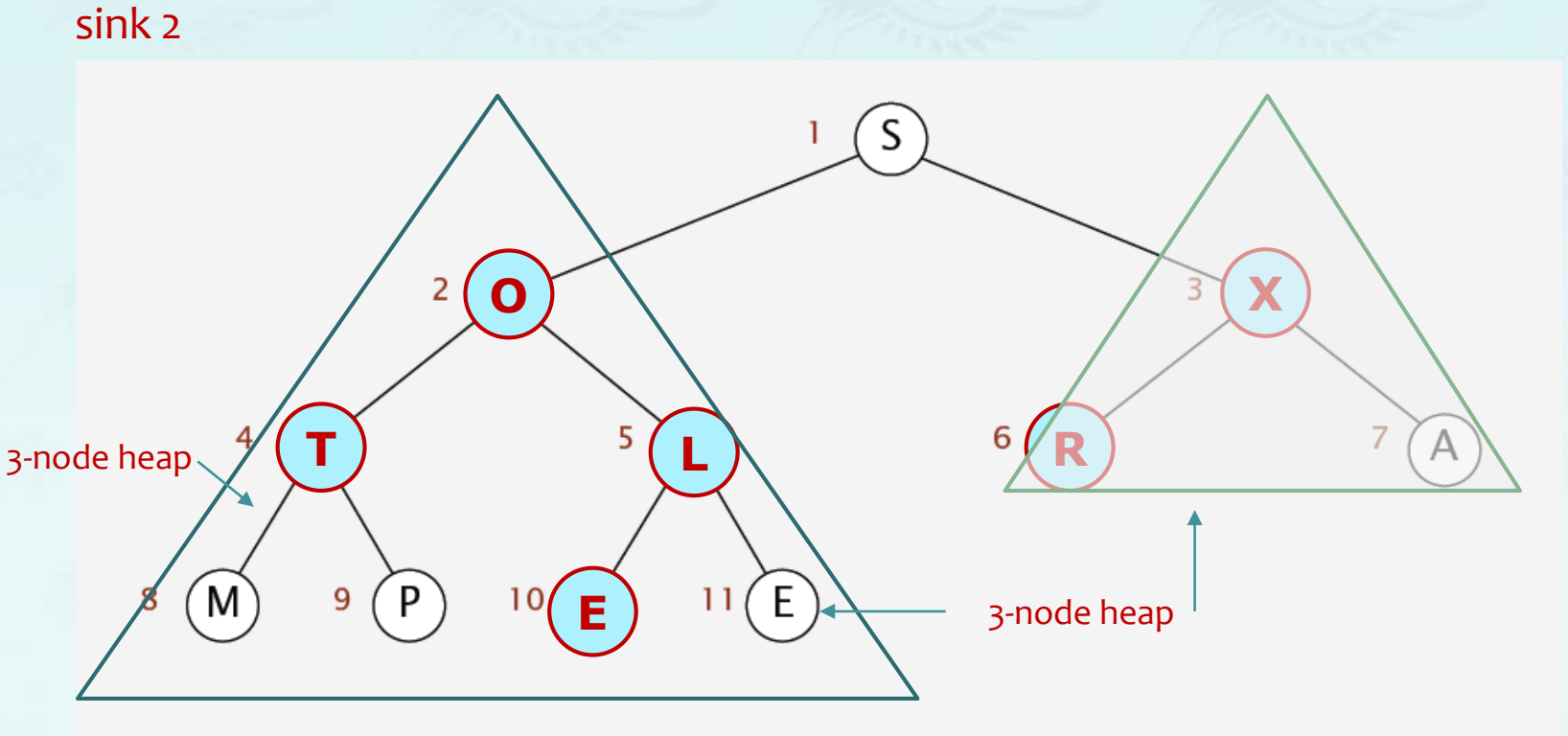
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

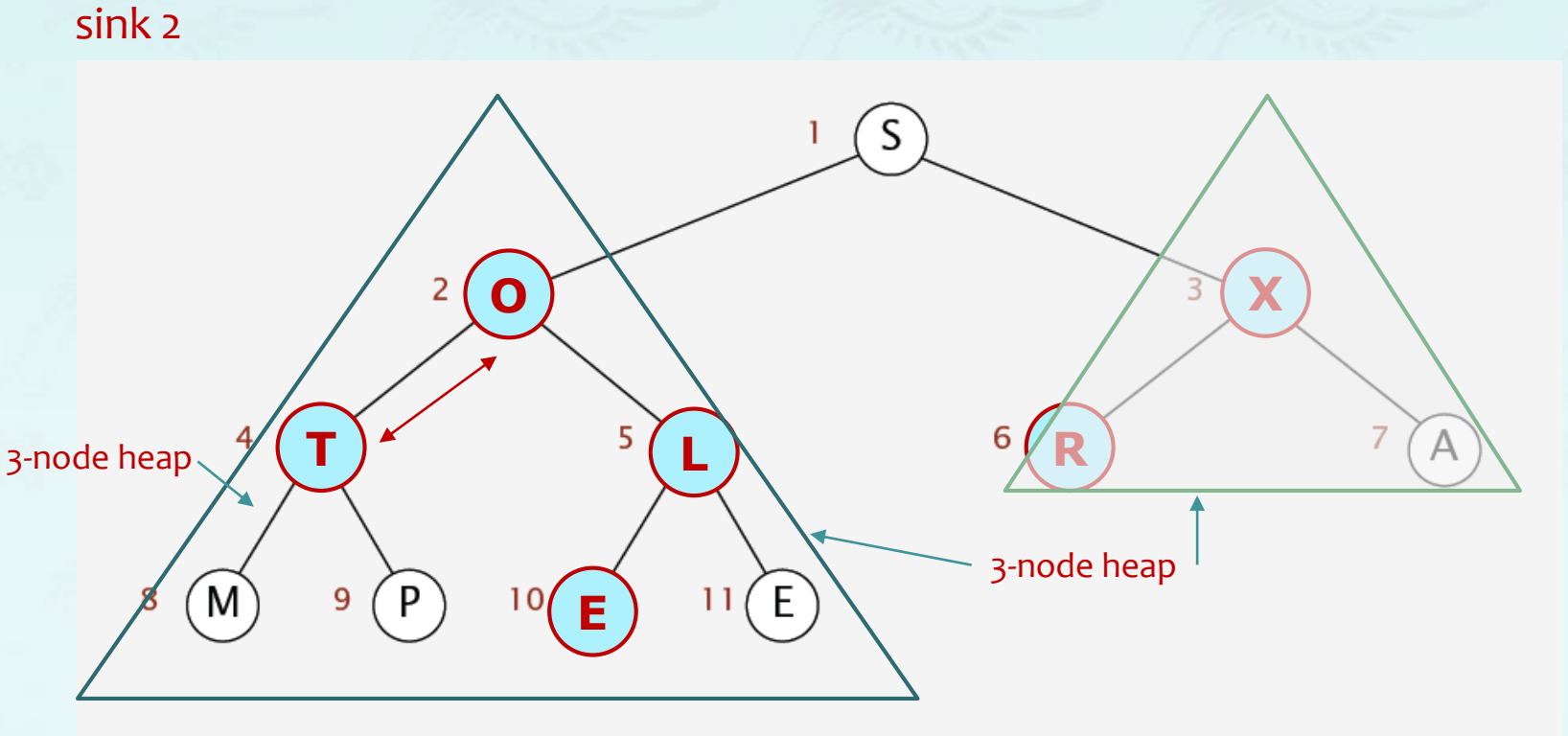
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

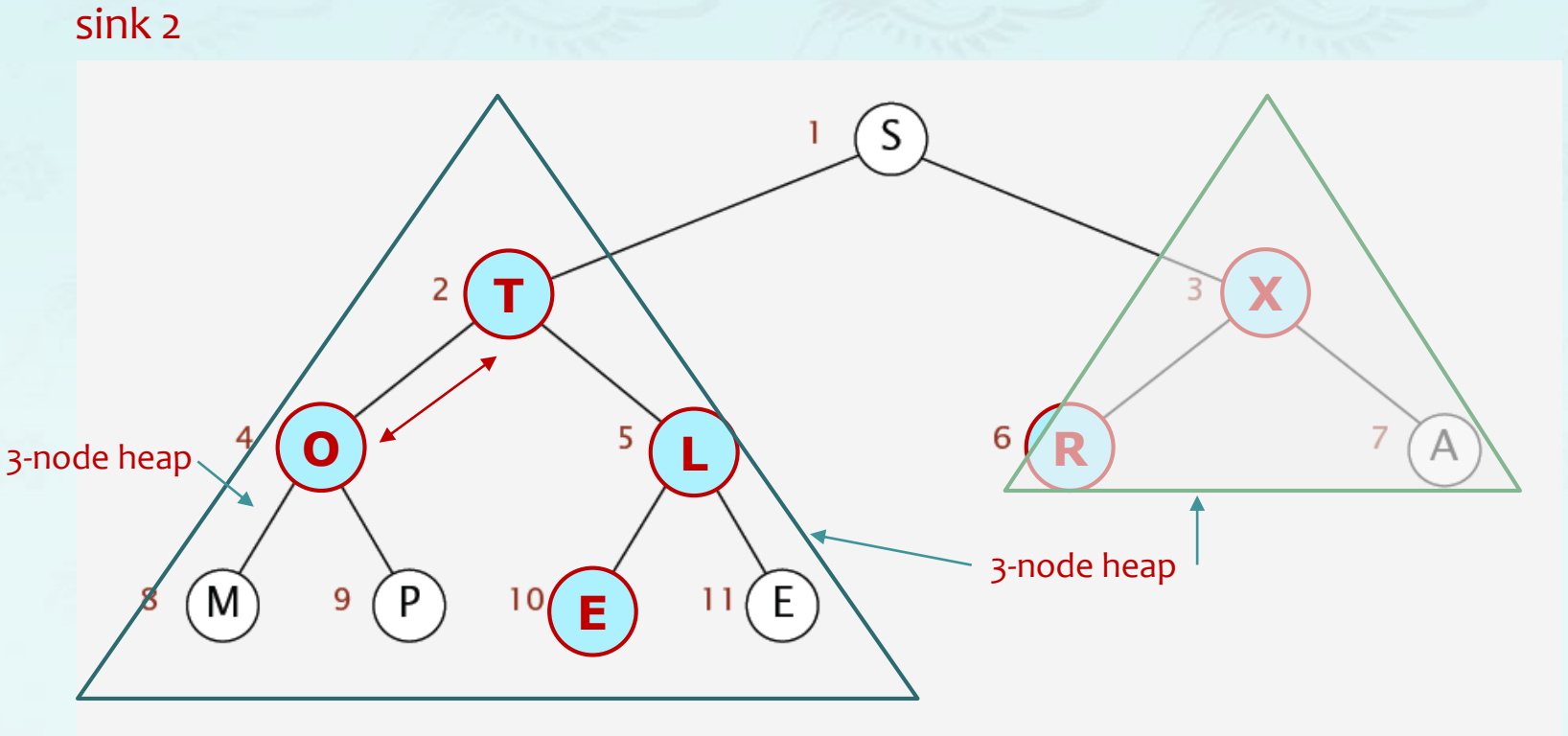
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	O	X	T	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

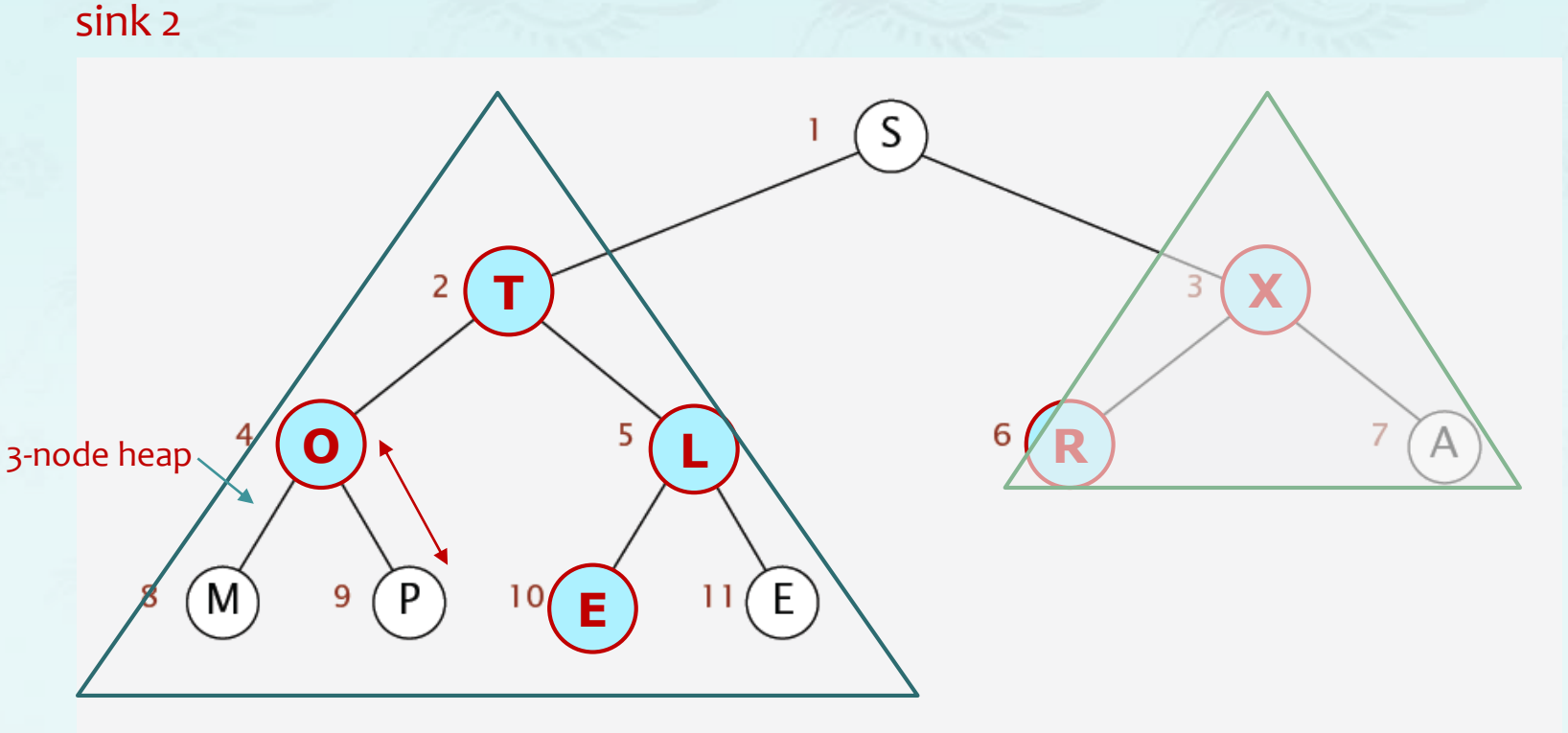
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	O	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

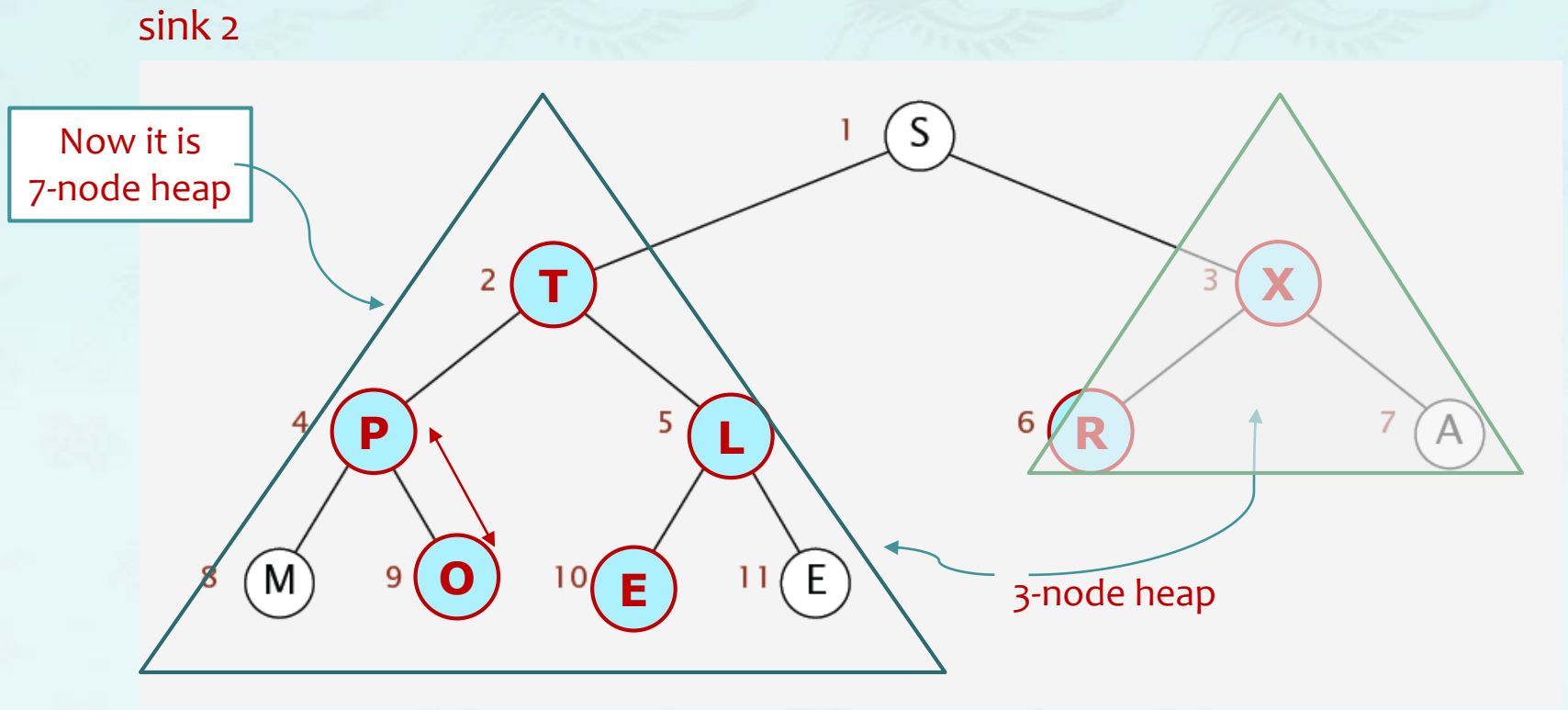
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	O	L	R	A	M	P	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

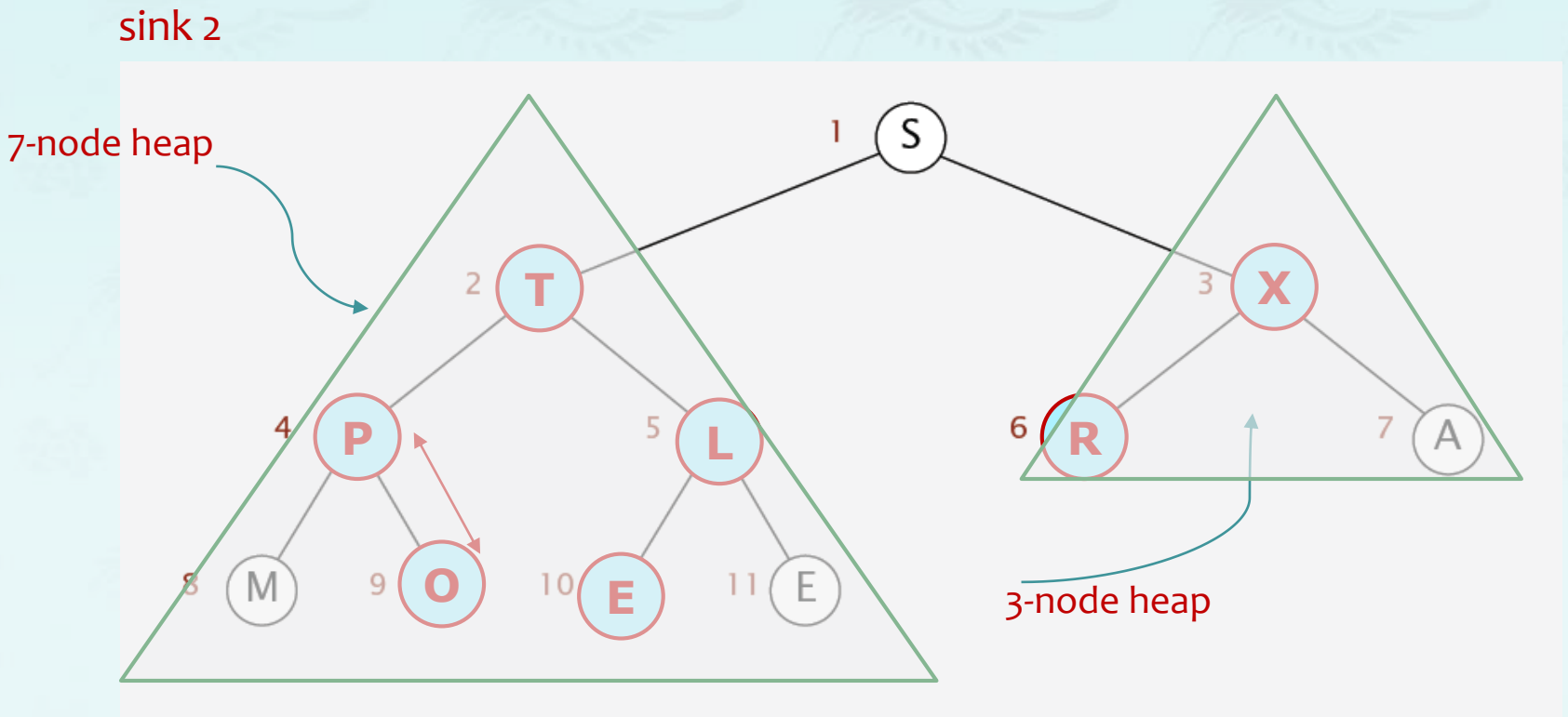
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

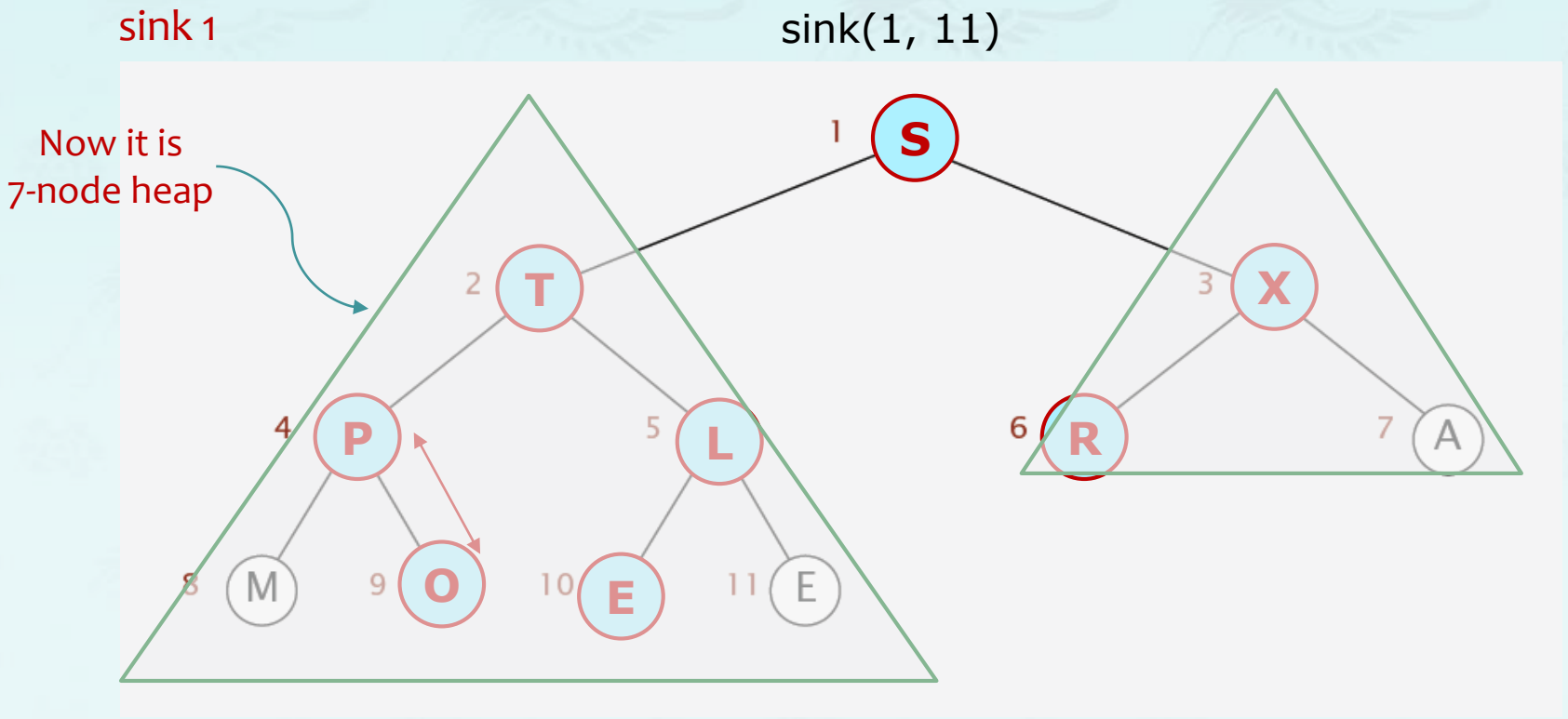
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

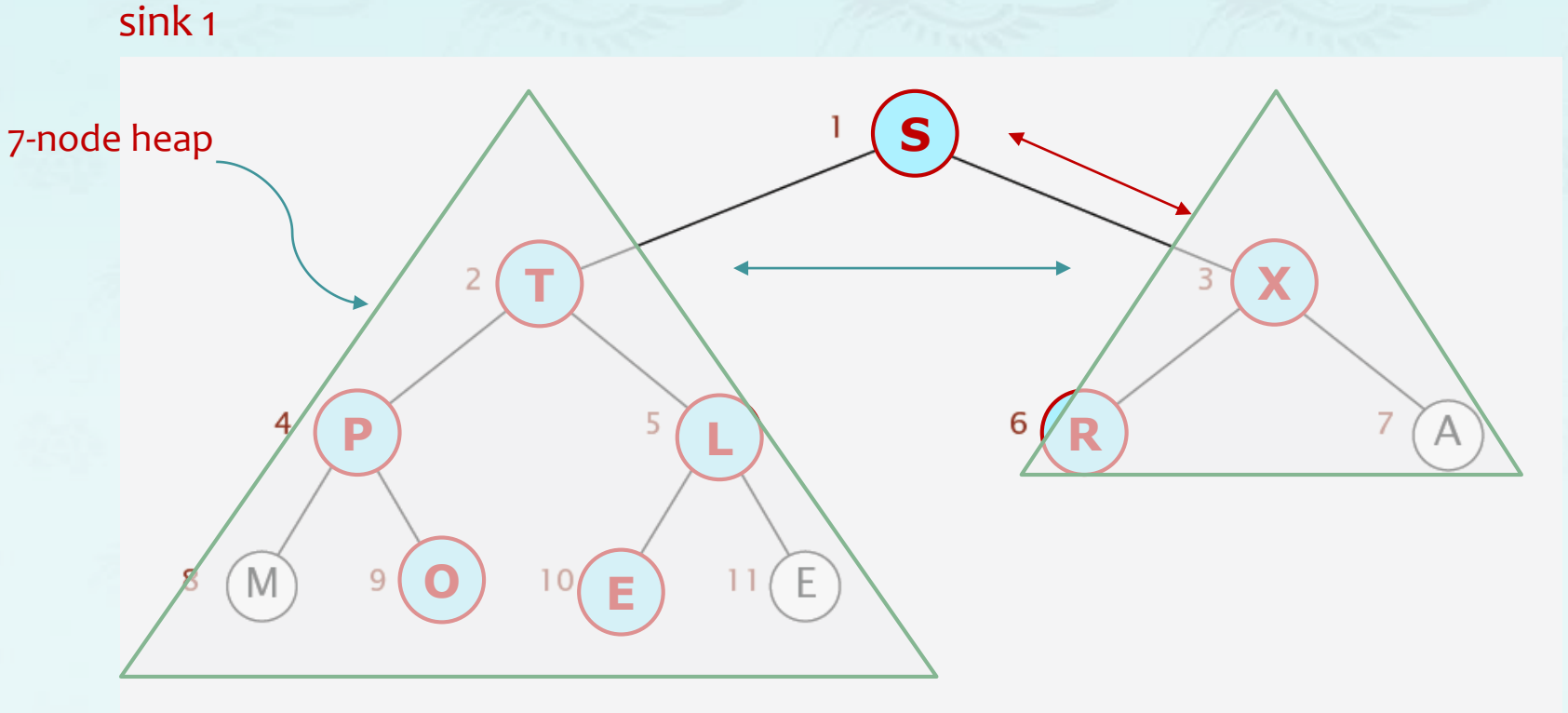
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

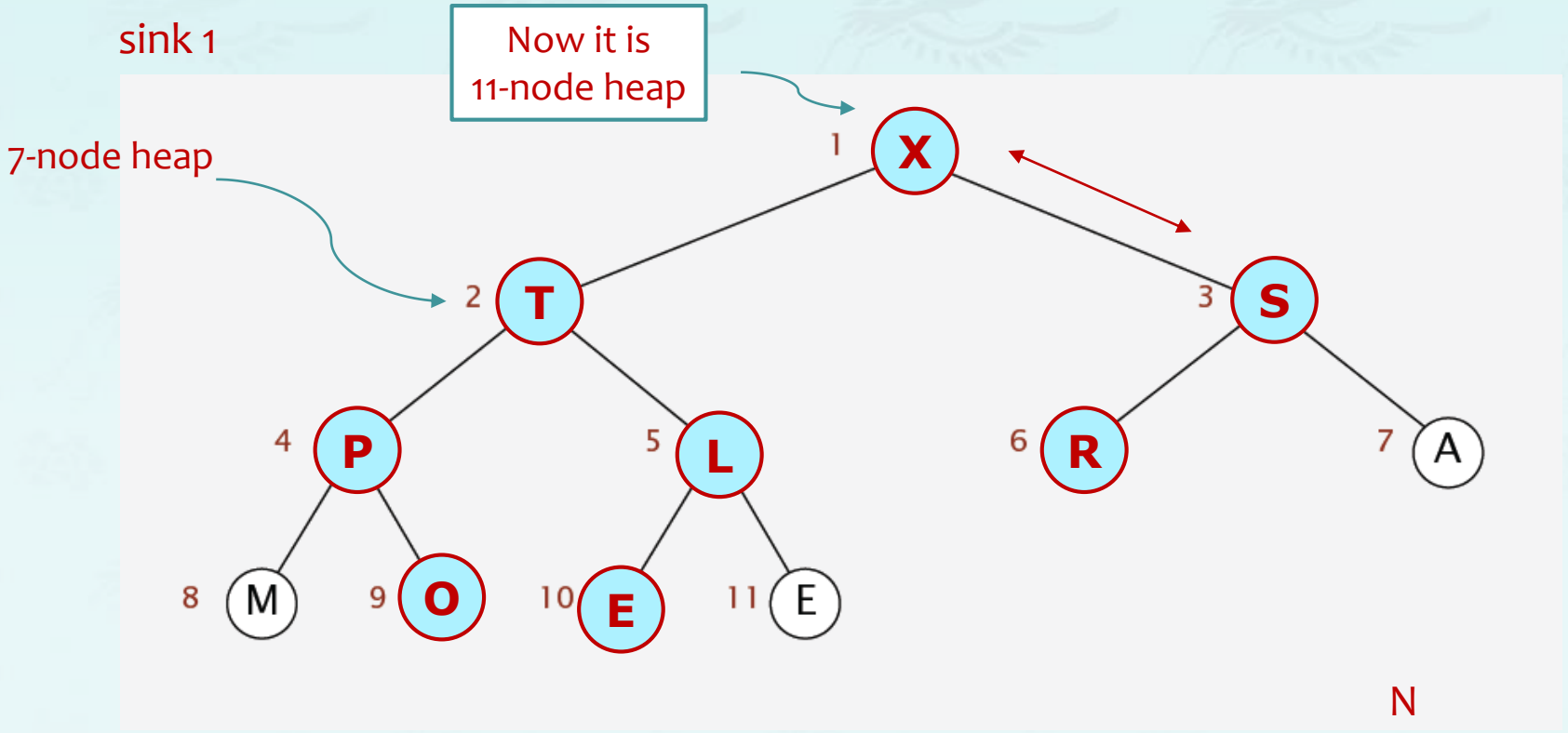
- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



S	T	X	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)

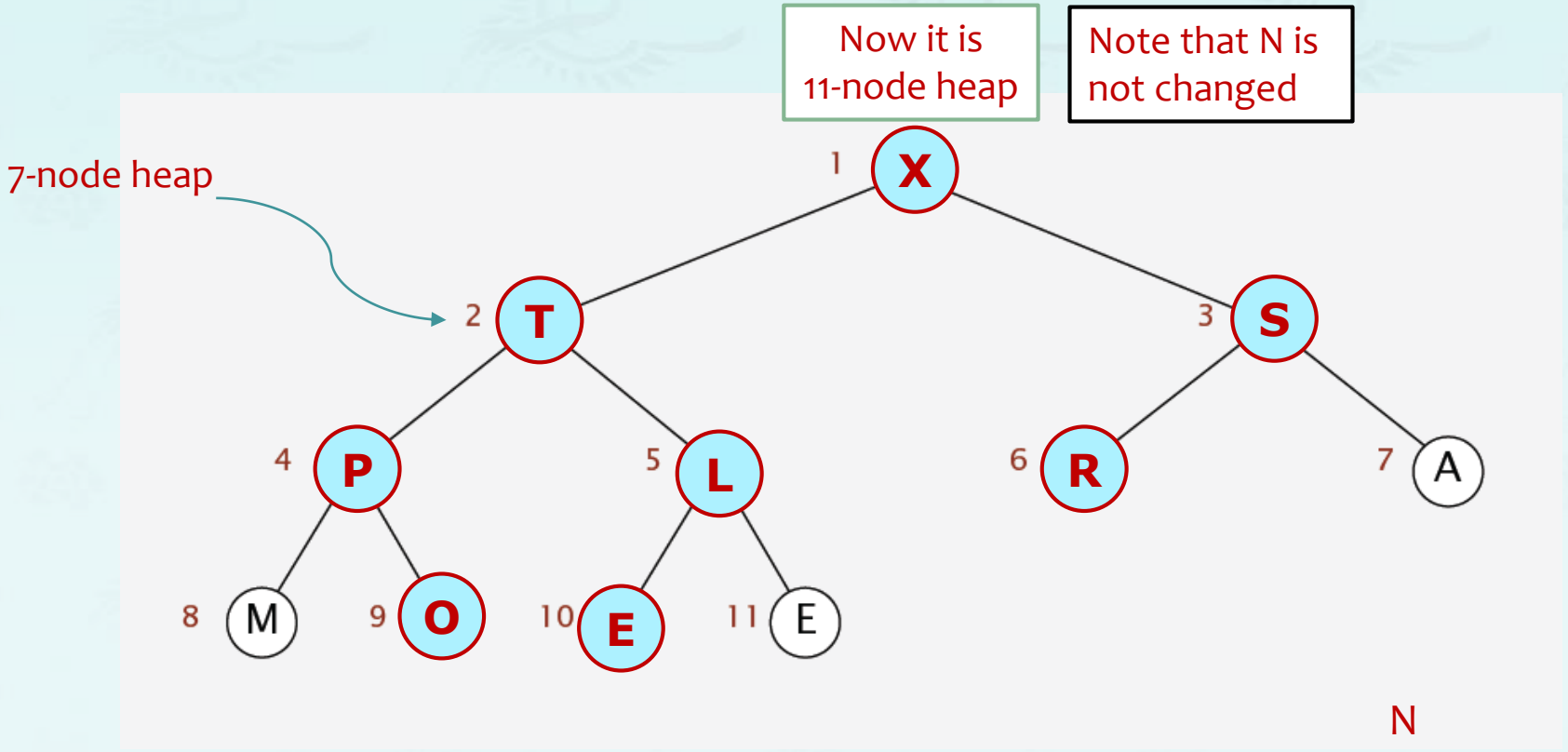


N

S	T	X	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **1st Pass: Heap construction(heapify)**
  Build max heap using bottom-up method.
  (we assume array entries are indexed from 1 to N.)



X	T	S	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

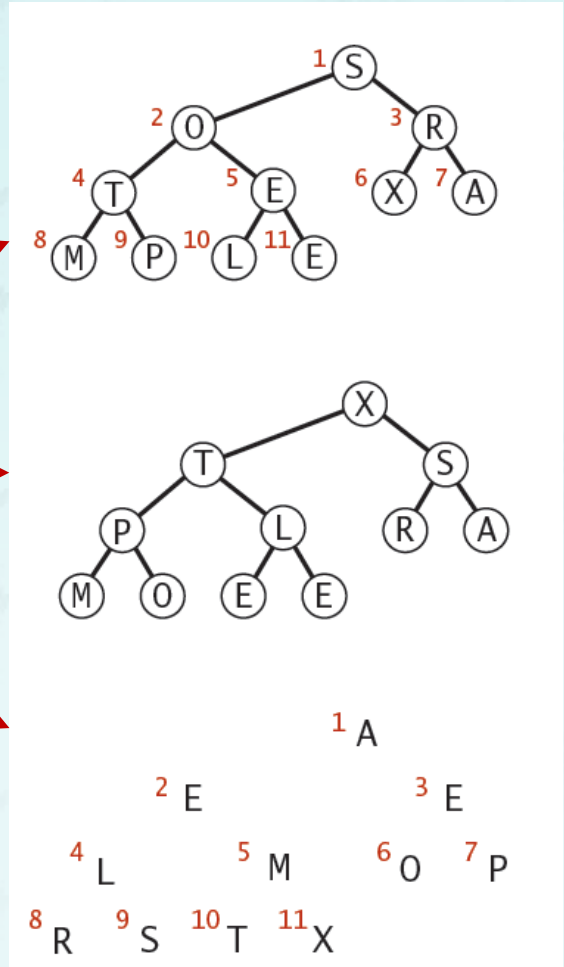
Basic plan for in-place sort

- **1st Pass**: Create maxheap with all **N** keys.
- **2nd Pass: Repeatedly remove the maximum key.**

An array of **N** keys in arbitrary order

1st Pass build a maxheap (in place)

2nd Pass: sorted result (in place)

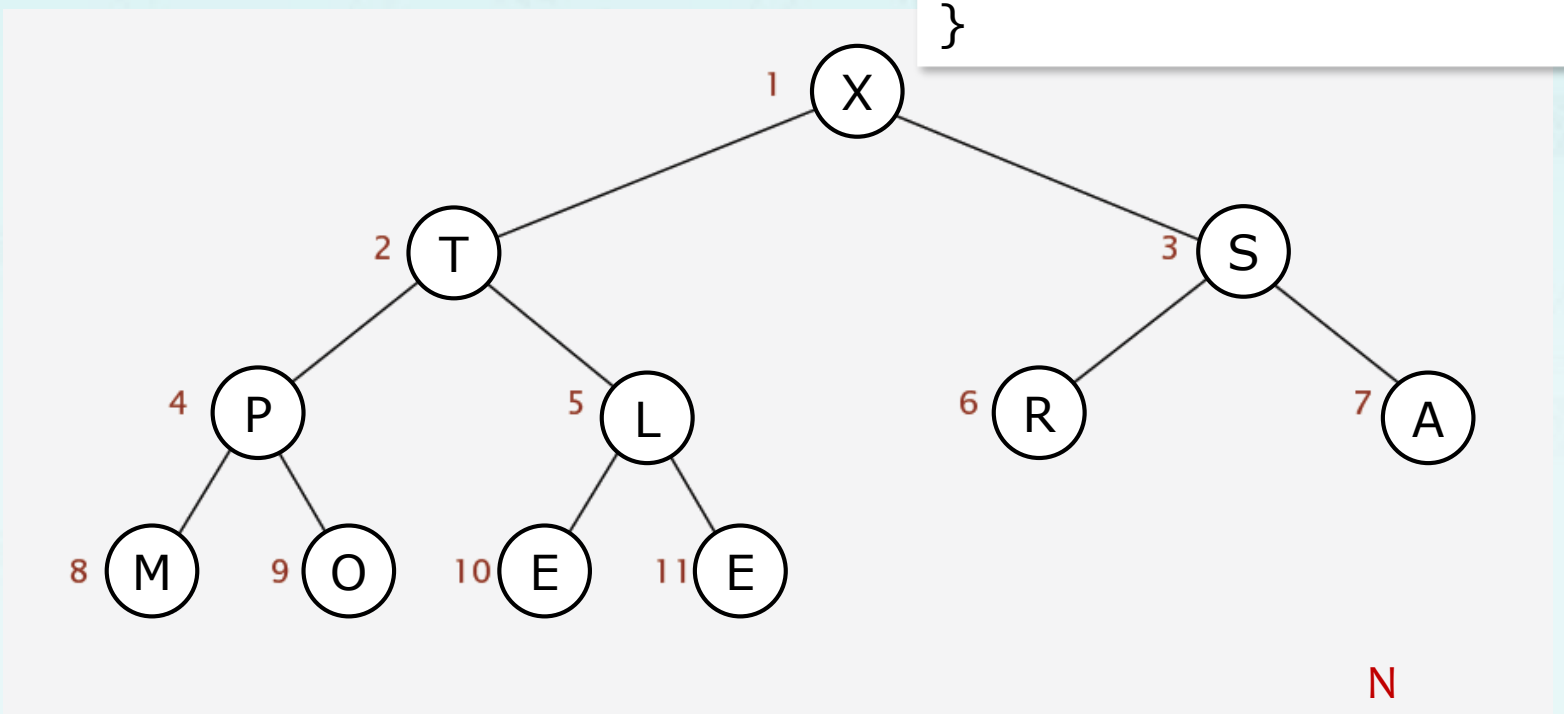


A	E	E	L	N	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

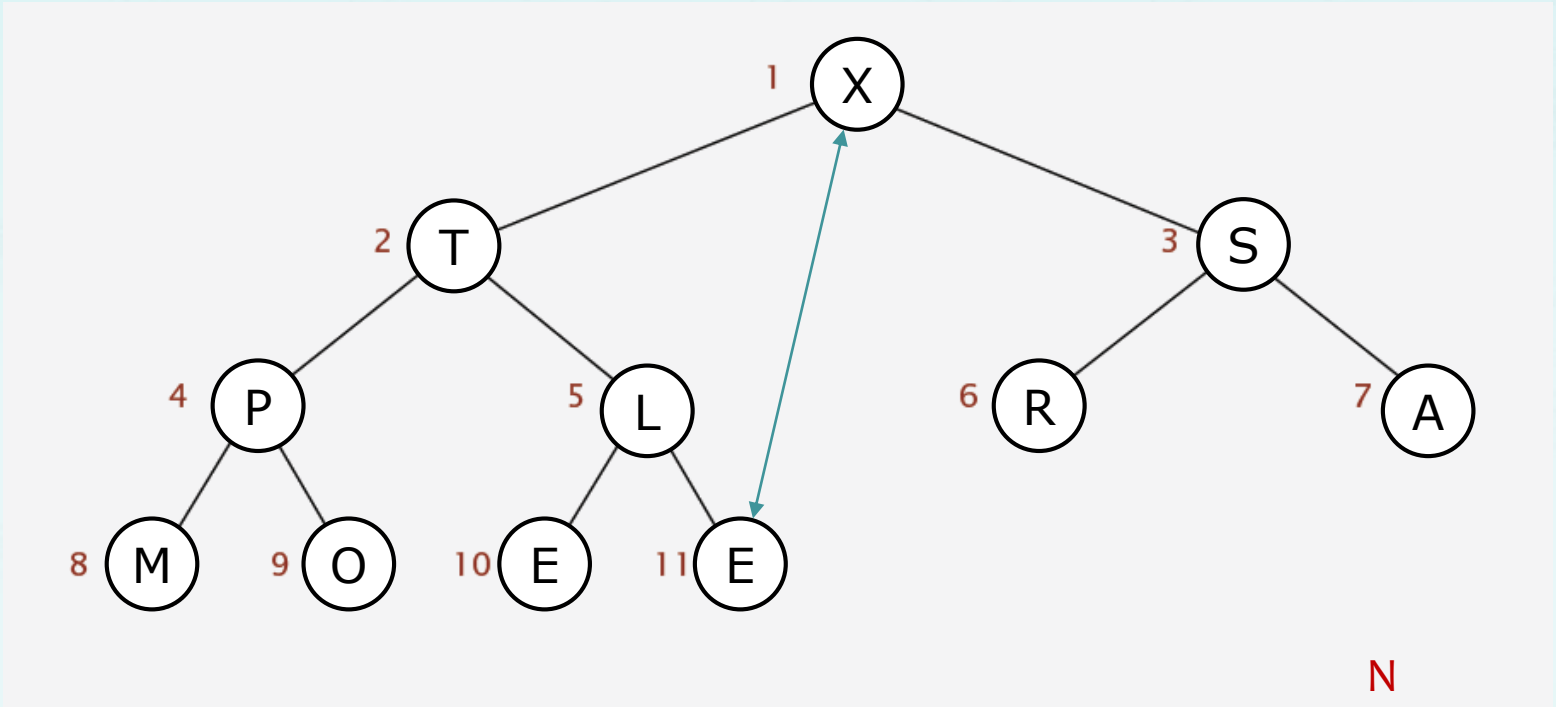


N

X	T	S	P	L	R	A	M	O	E	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

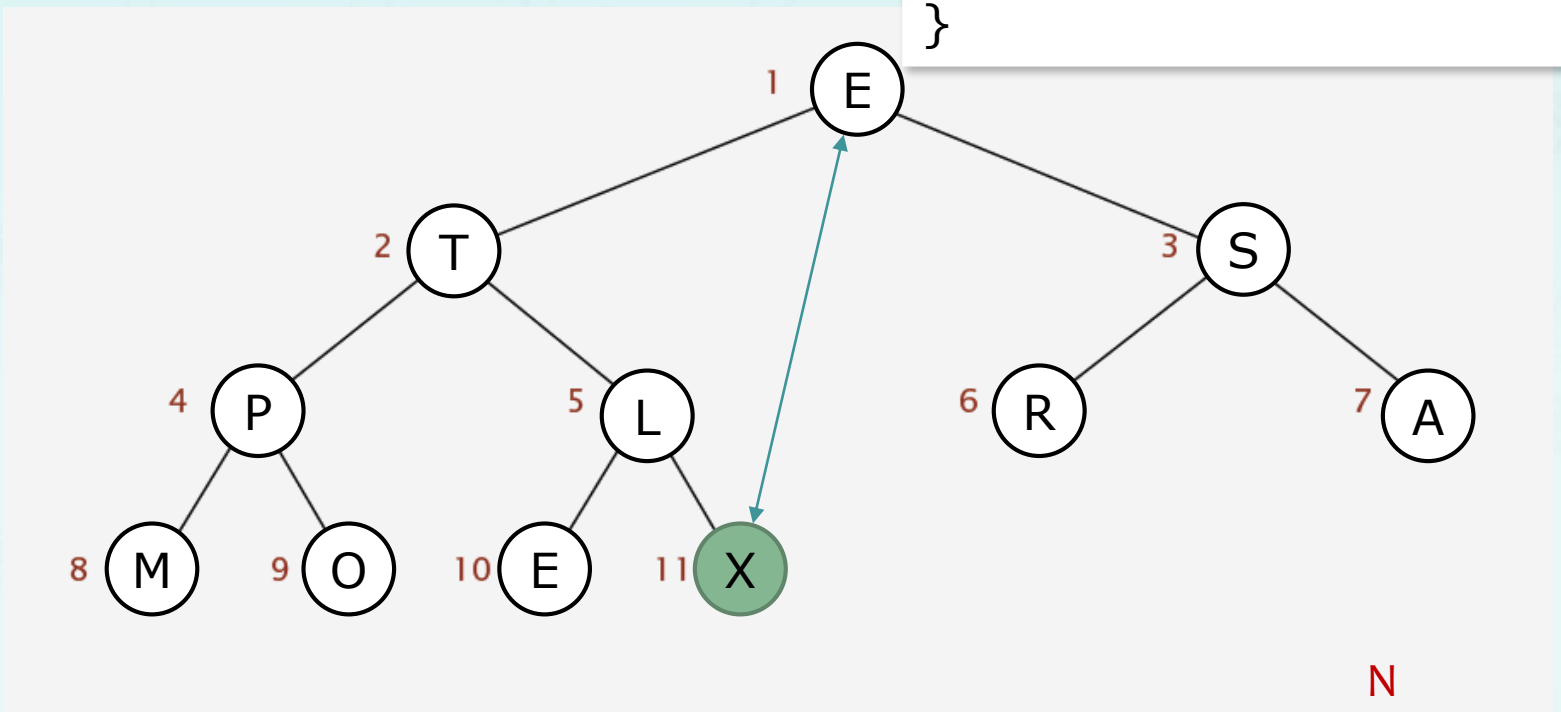


	N
X	1
T	2
S	3
P	4
L	5
R	6
A	7
M	8
O	9
E	10
E	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
void sink(heap h, int k) {
   while (2 * k <= h->N) {
      int j = 2 * k;
      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

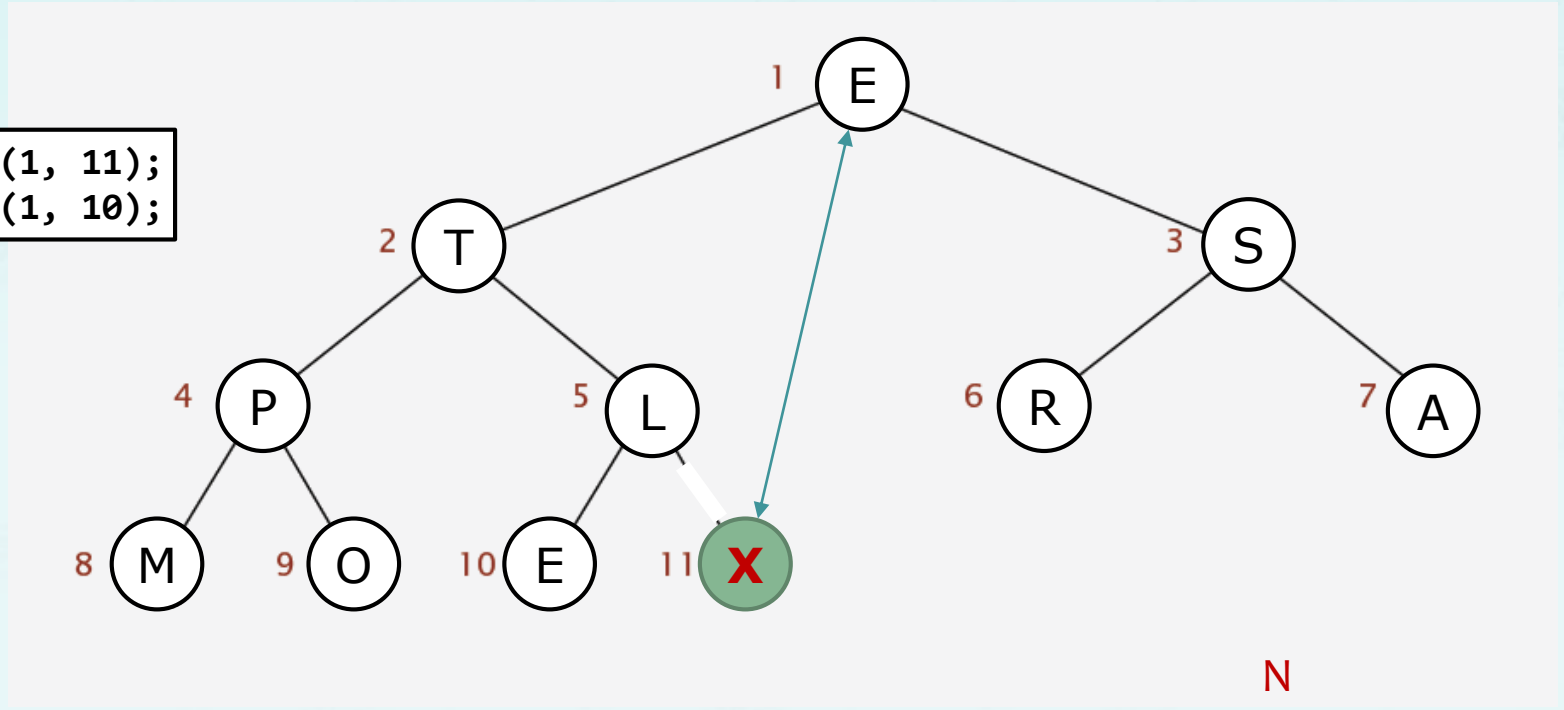


	N
X	1
T	2
S	3
P	4
L	5
R	6
A	7
M	8
O	9
E	10
E	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

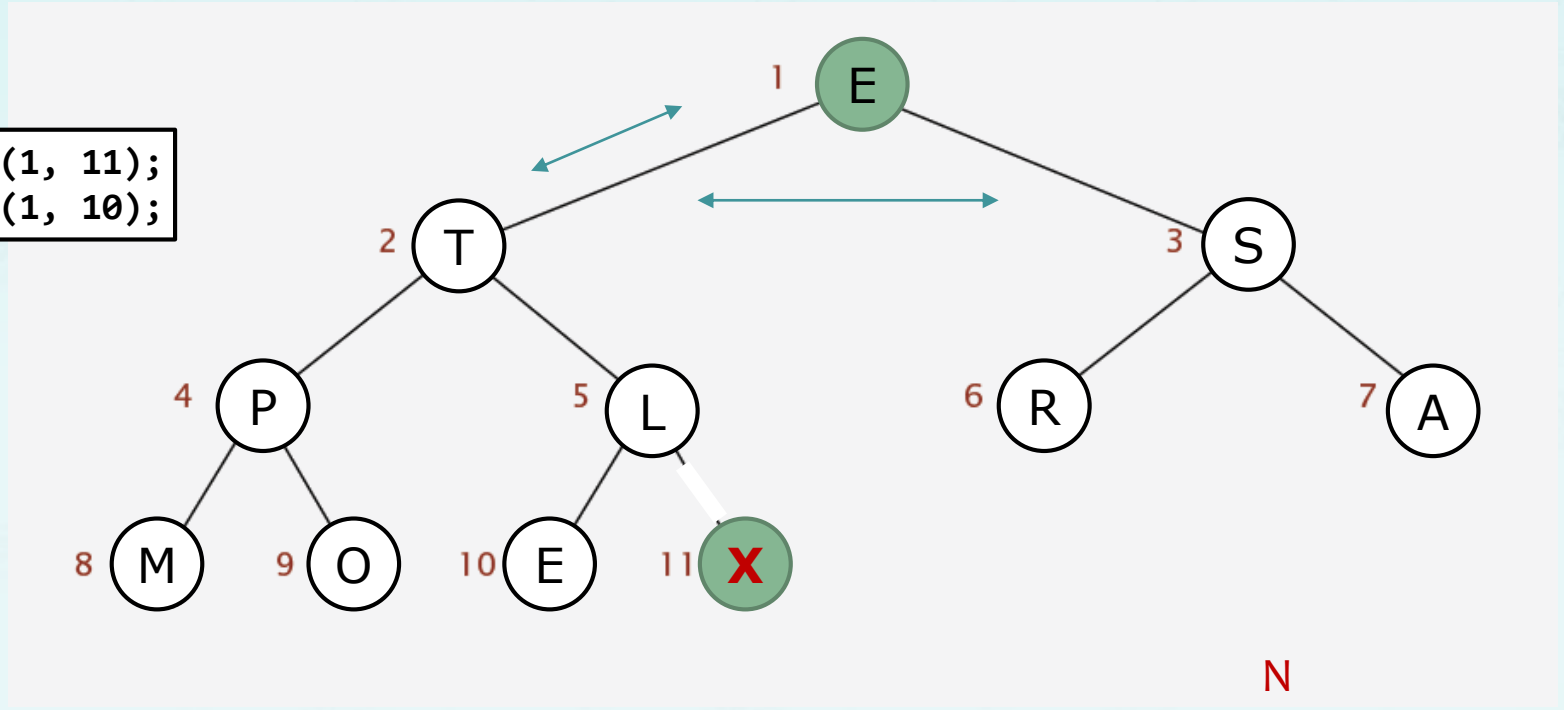


									N	
E	T	S	P	L	R	A	M	O	E	**X**
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

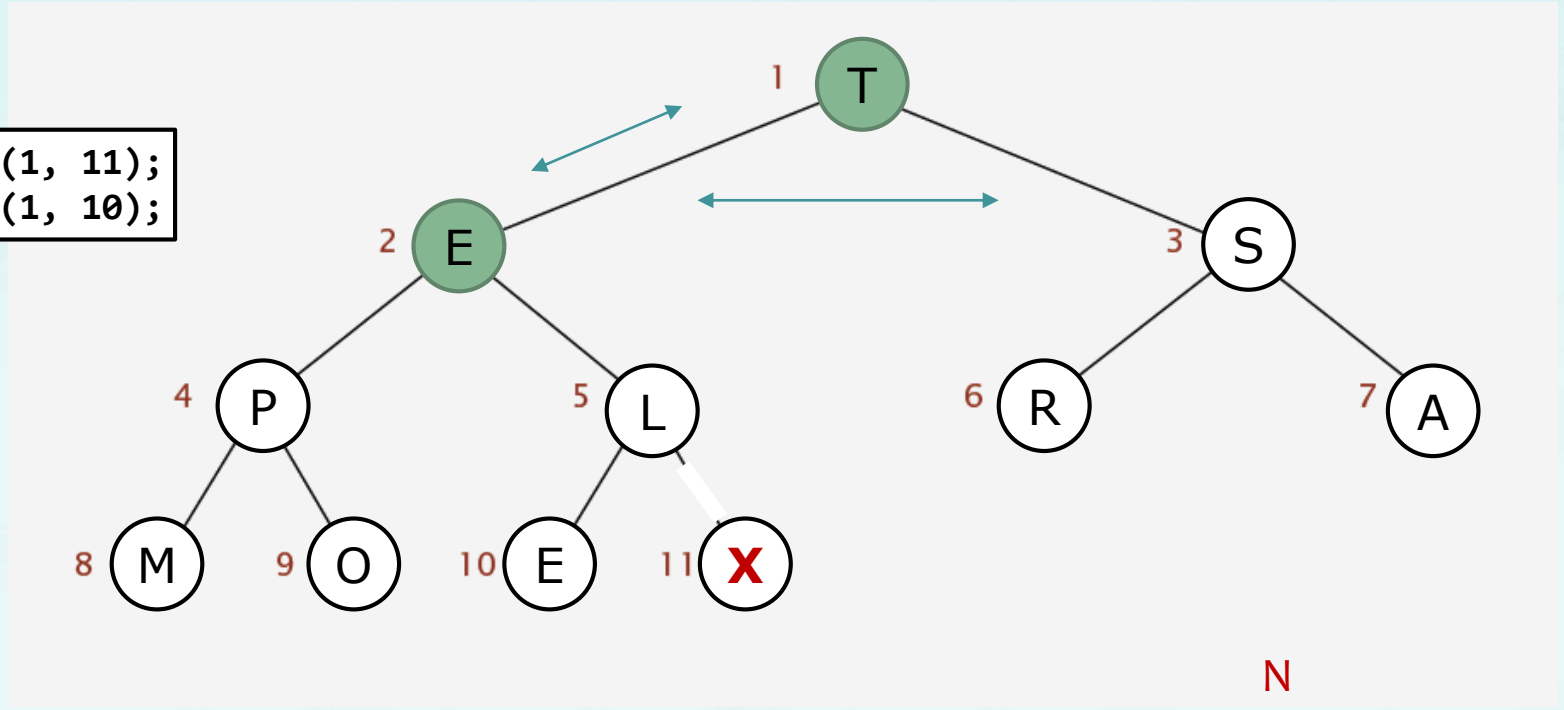


									N	
E	T	S	P	L	R	A	M	O	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

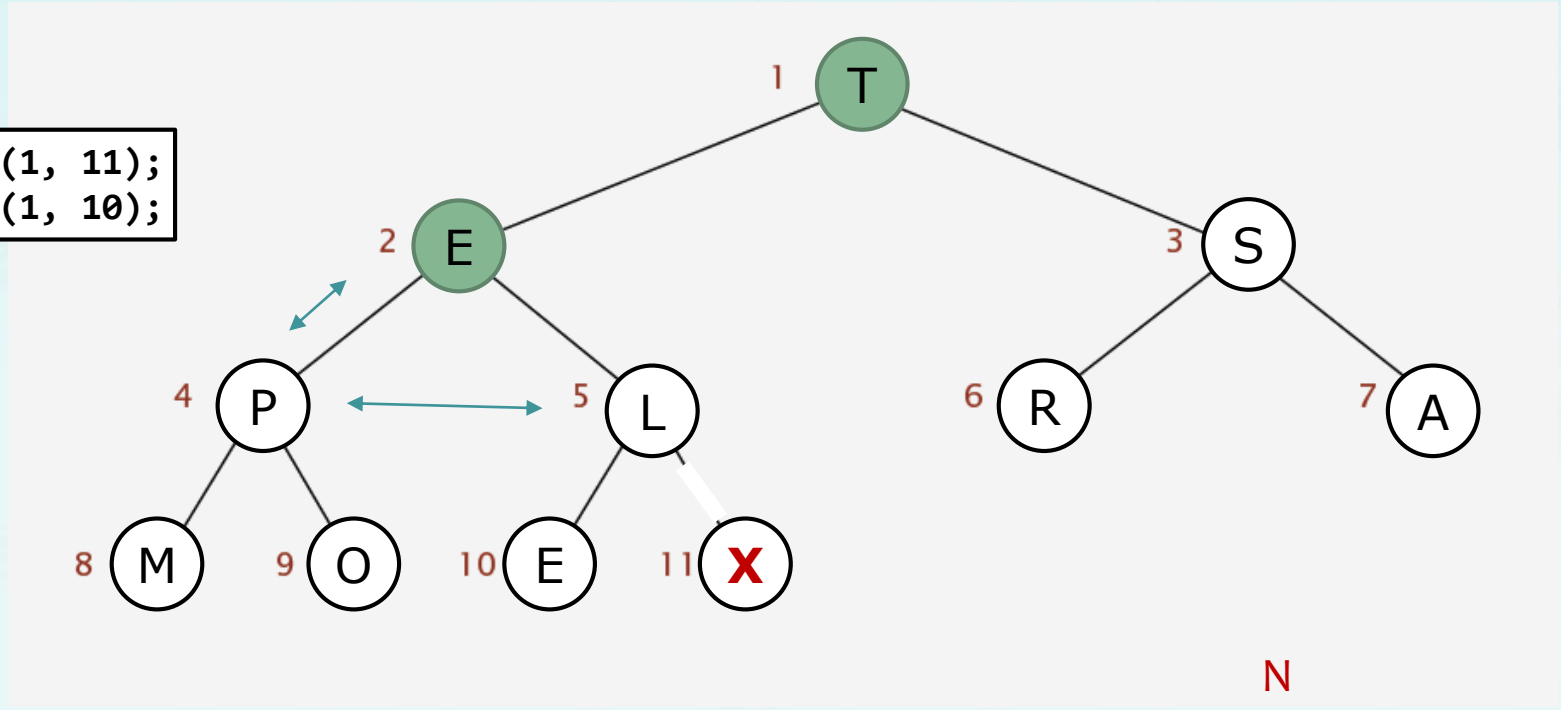


									N	
E	T	S	P	L	R	A	M	O	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

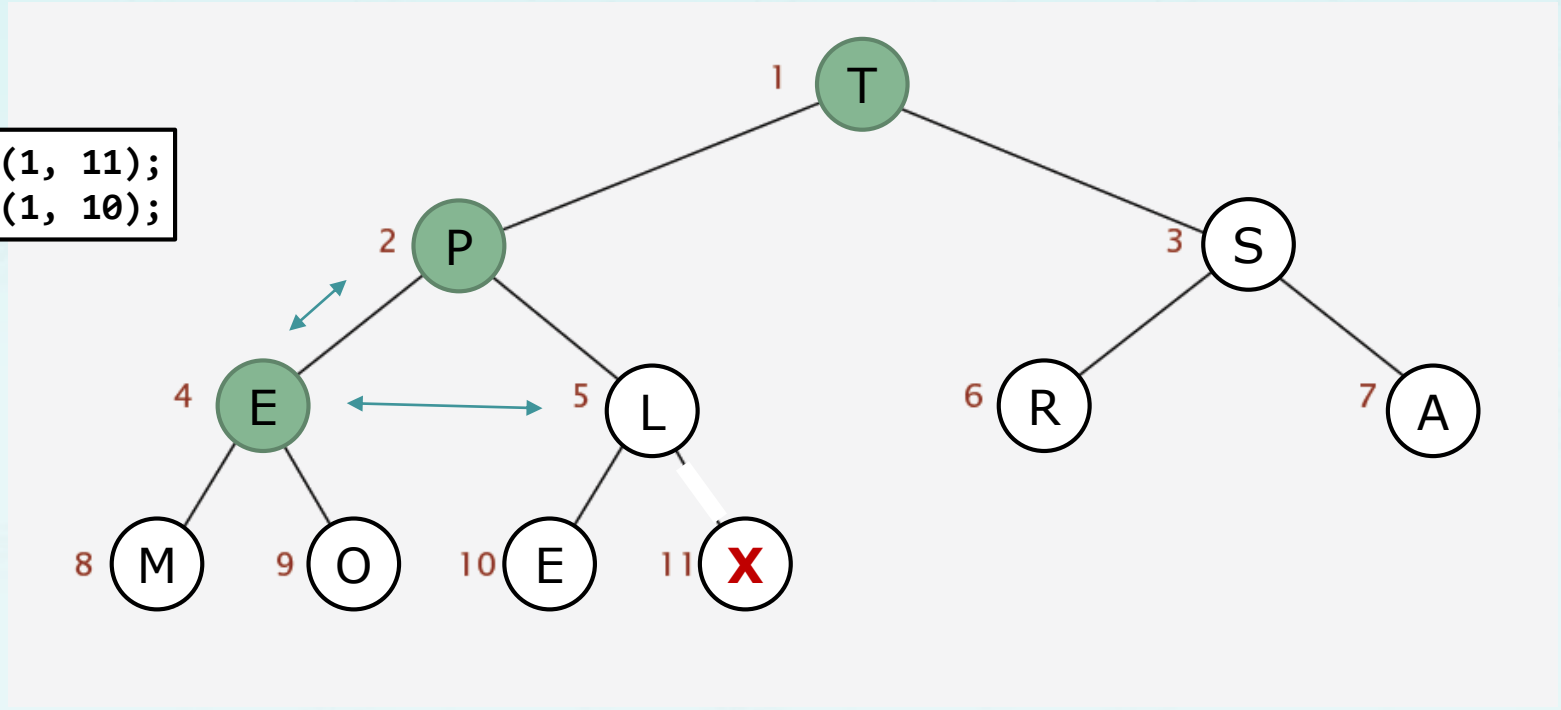


									N	
E	T	S	P	L	R	A	M	O	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

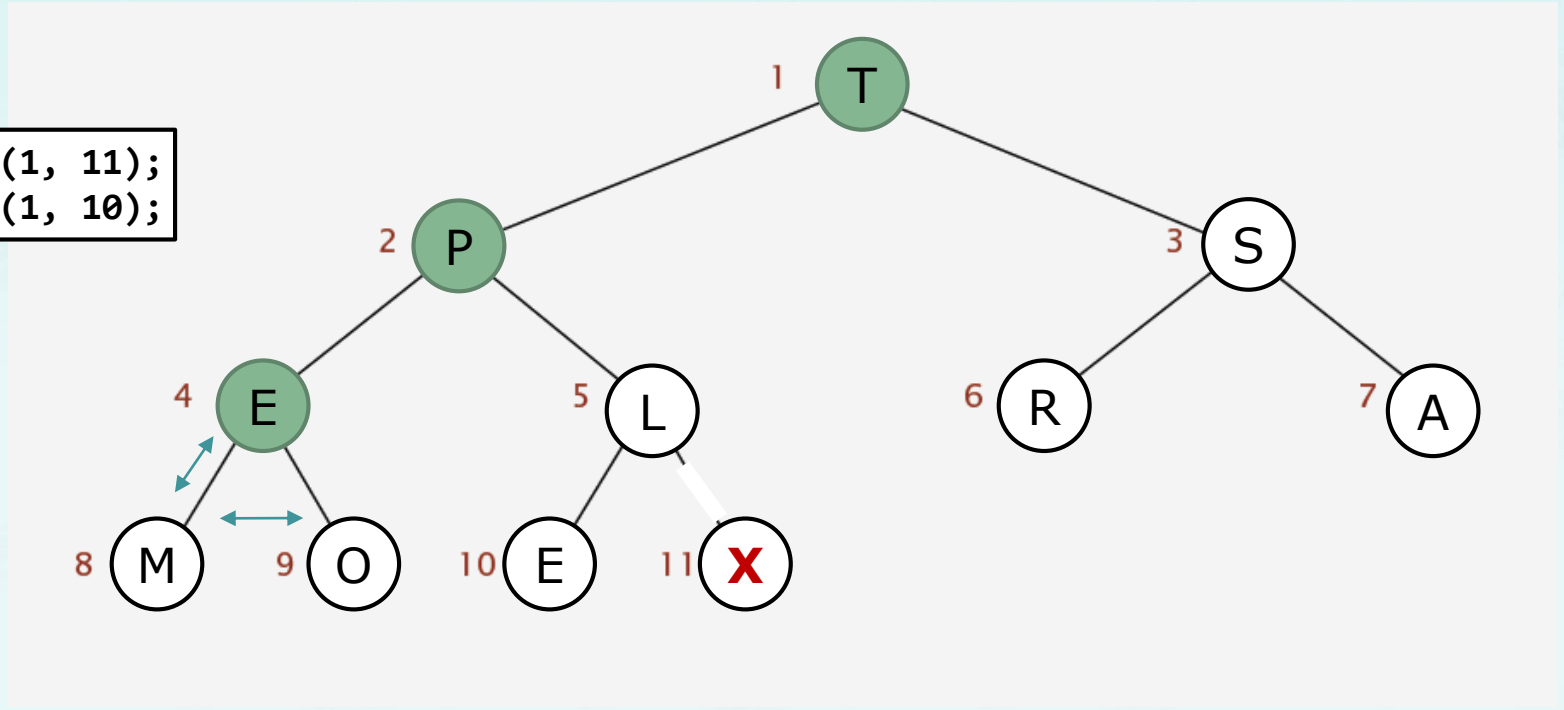


E	T	S	P	L	R	A	M	O	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

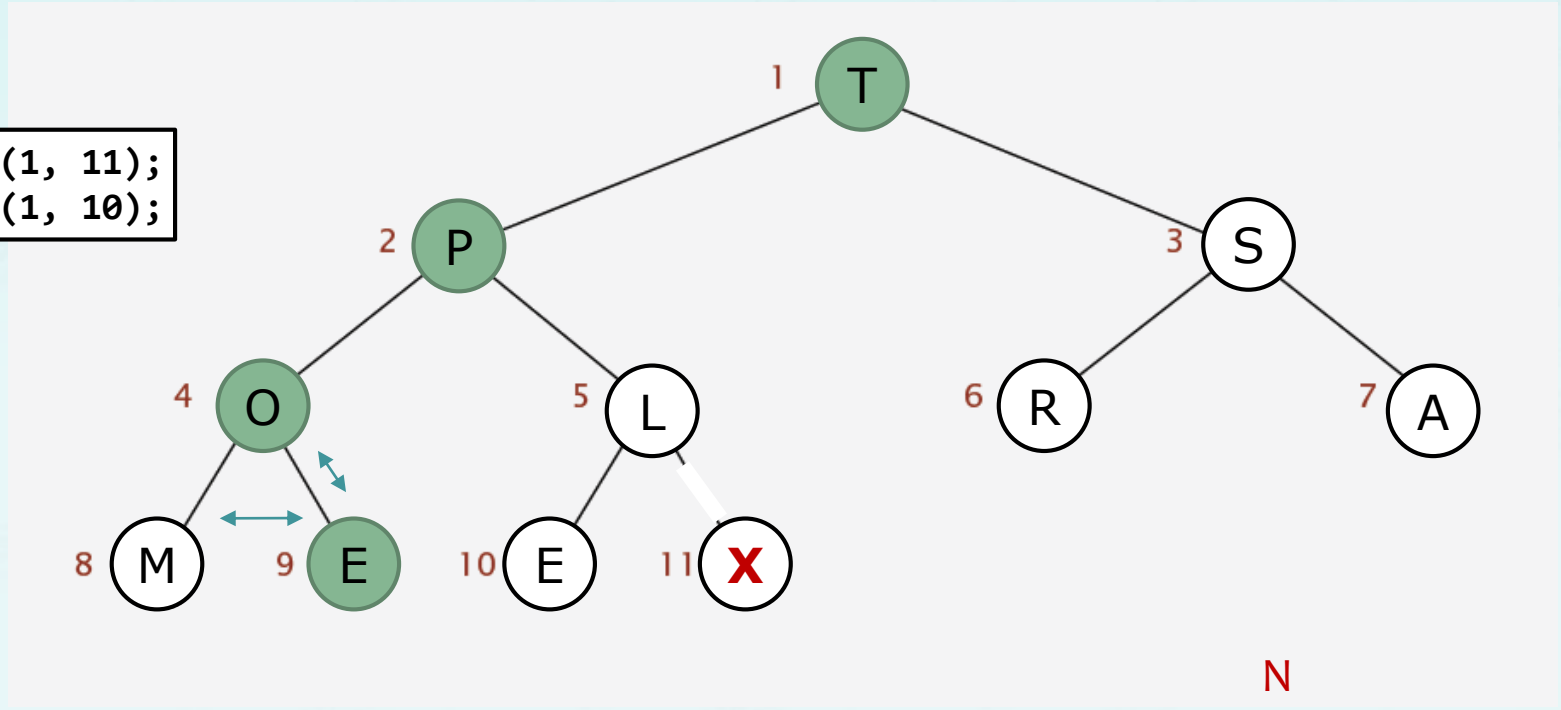


E	T	S	P	L	R	A	M	O	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 11);
sink(1, 10);
```

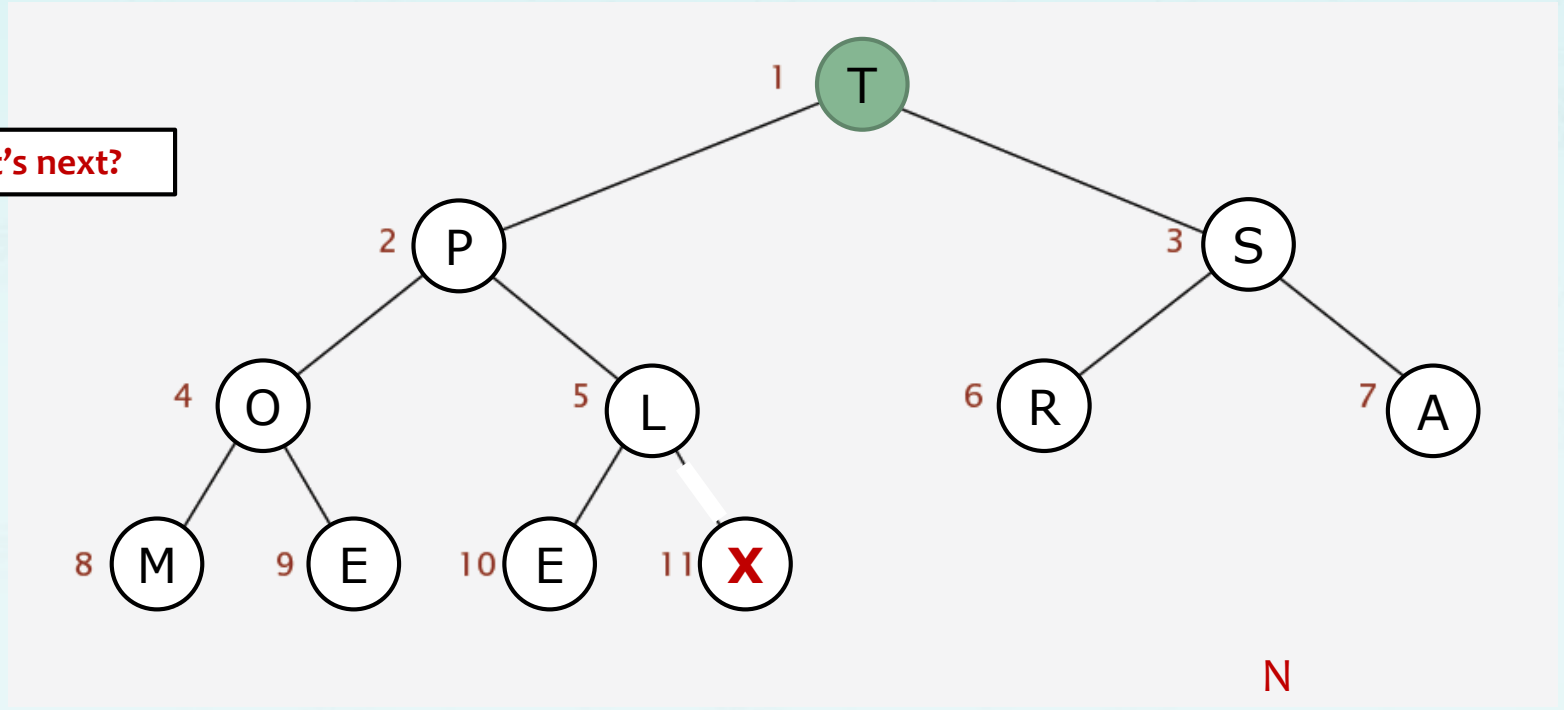


									N	
T	P	S	O	L	R	A	M	E	E	**X**
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

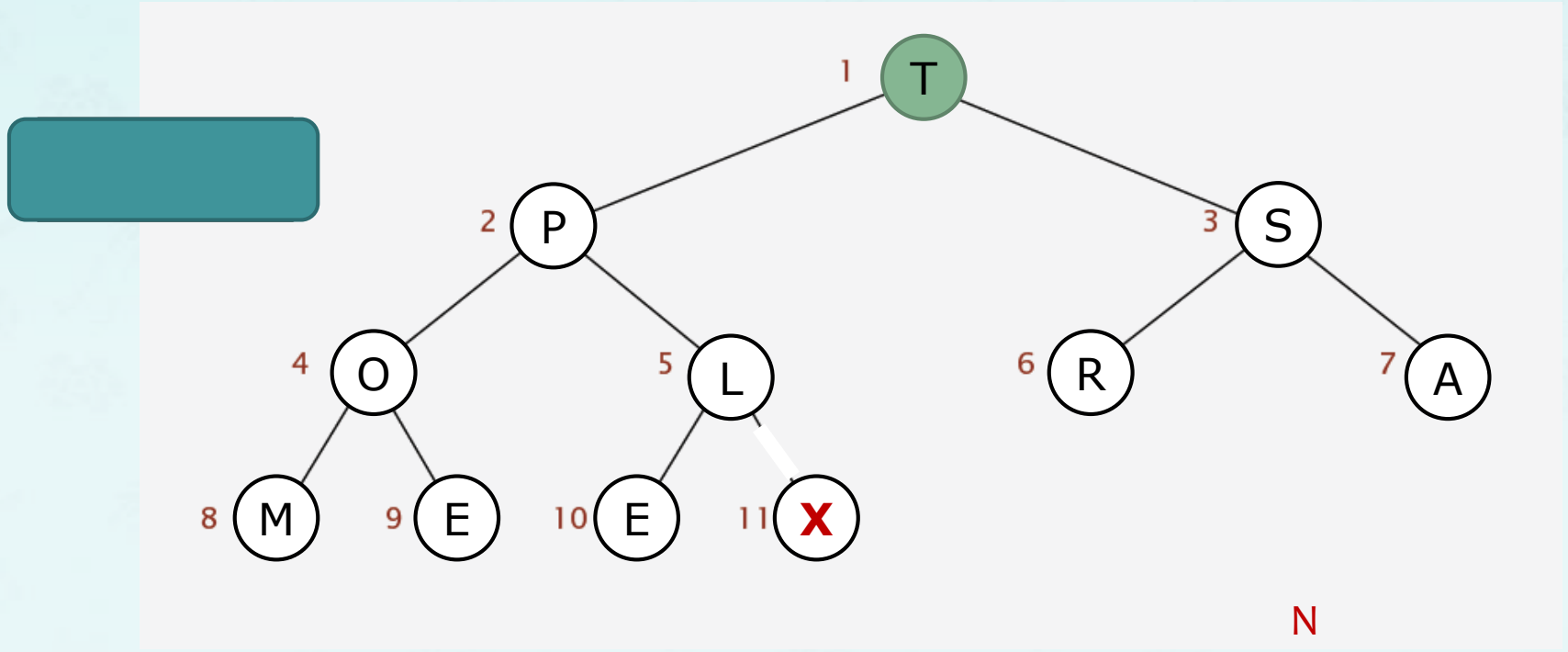
What's next?



									N	
T	P	S	O	L	R	A	M	E	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

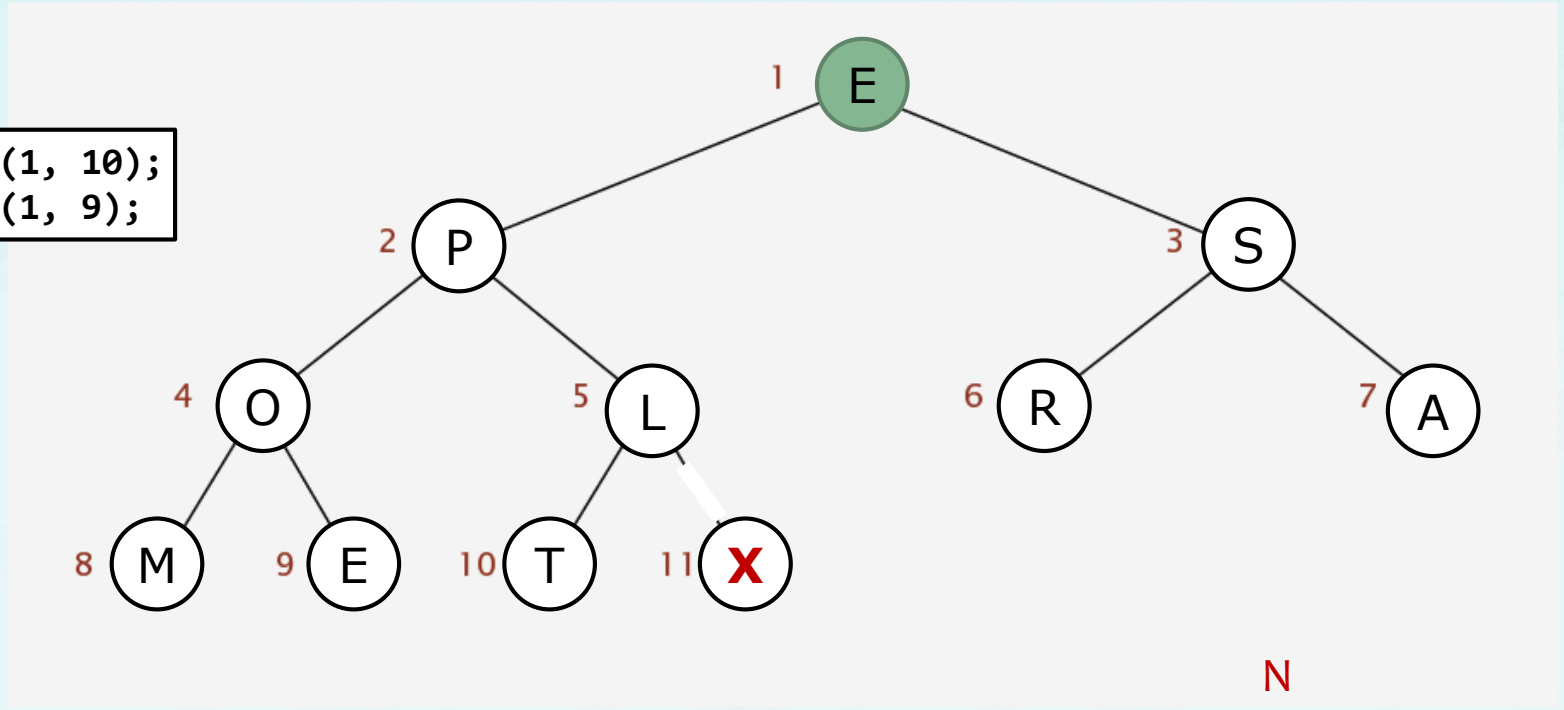


									N	
T	P	S	O	L	R	A	M	E	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 10);
sink(1, 9);
```

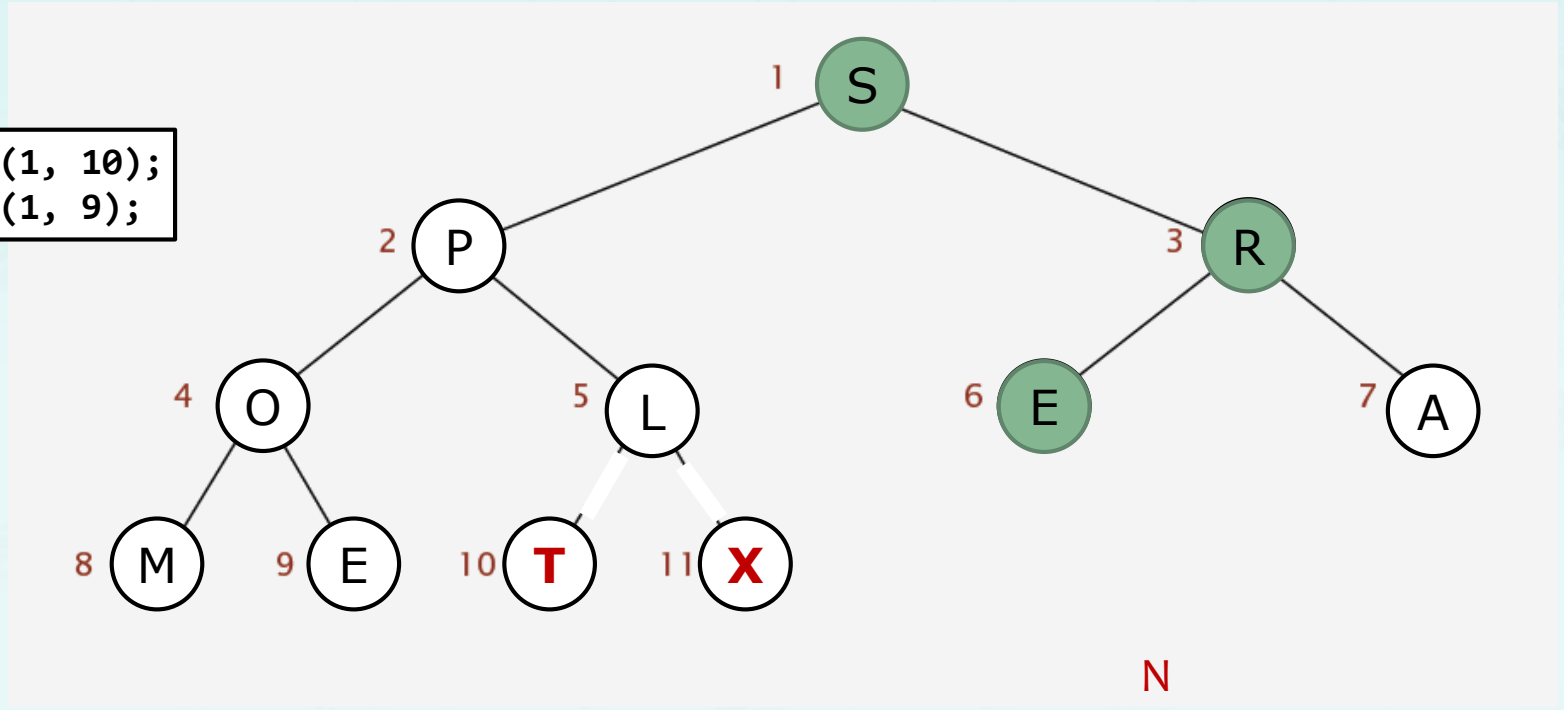


									N	
T	P	S	O	L	R	A	M	E	E	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 10);
sink(1, 9);
```

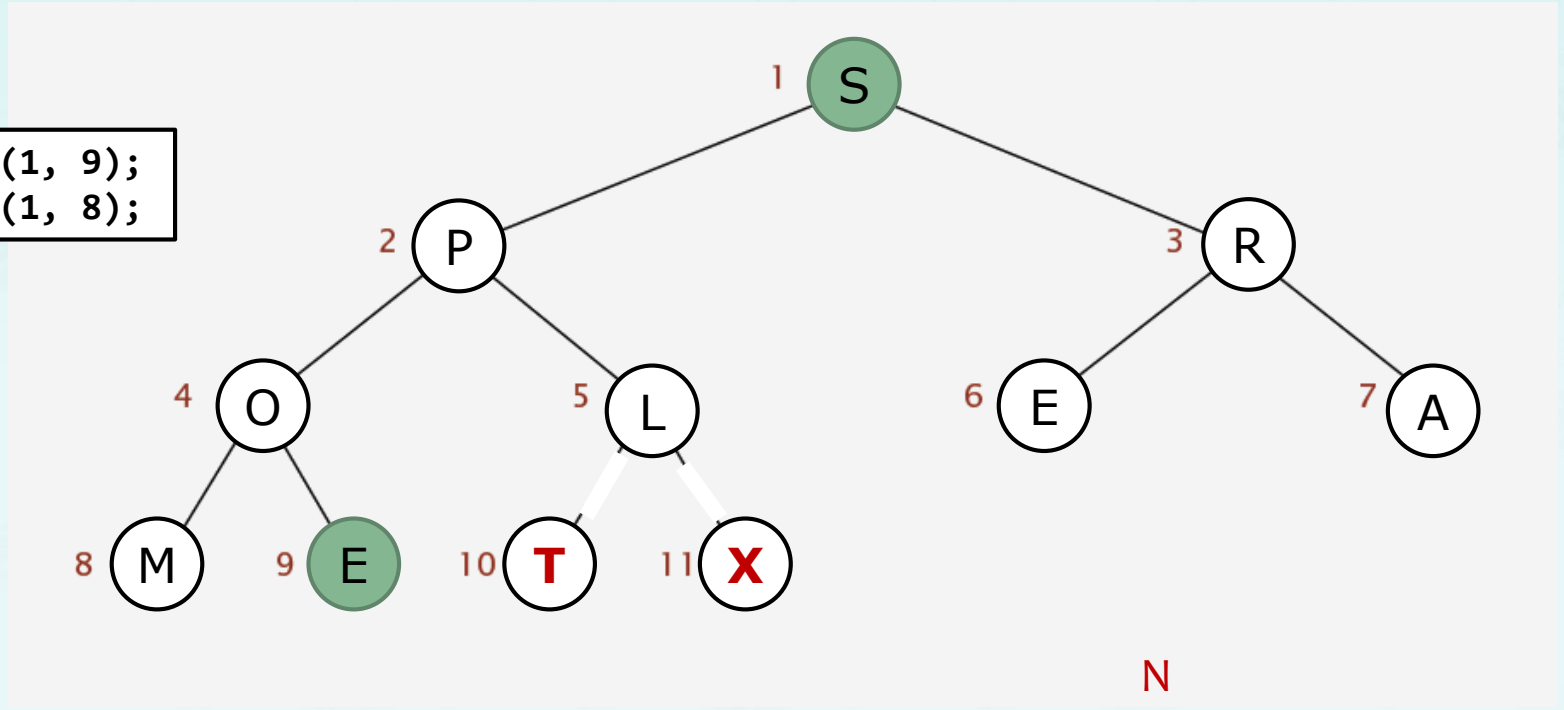


	T	P	S	O	L	R	A	M	E	T	X
	1	2	3	4	5	6	7	8	9	10	11
									N		

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 9);
sink(1, 8);
```



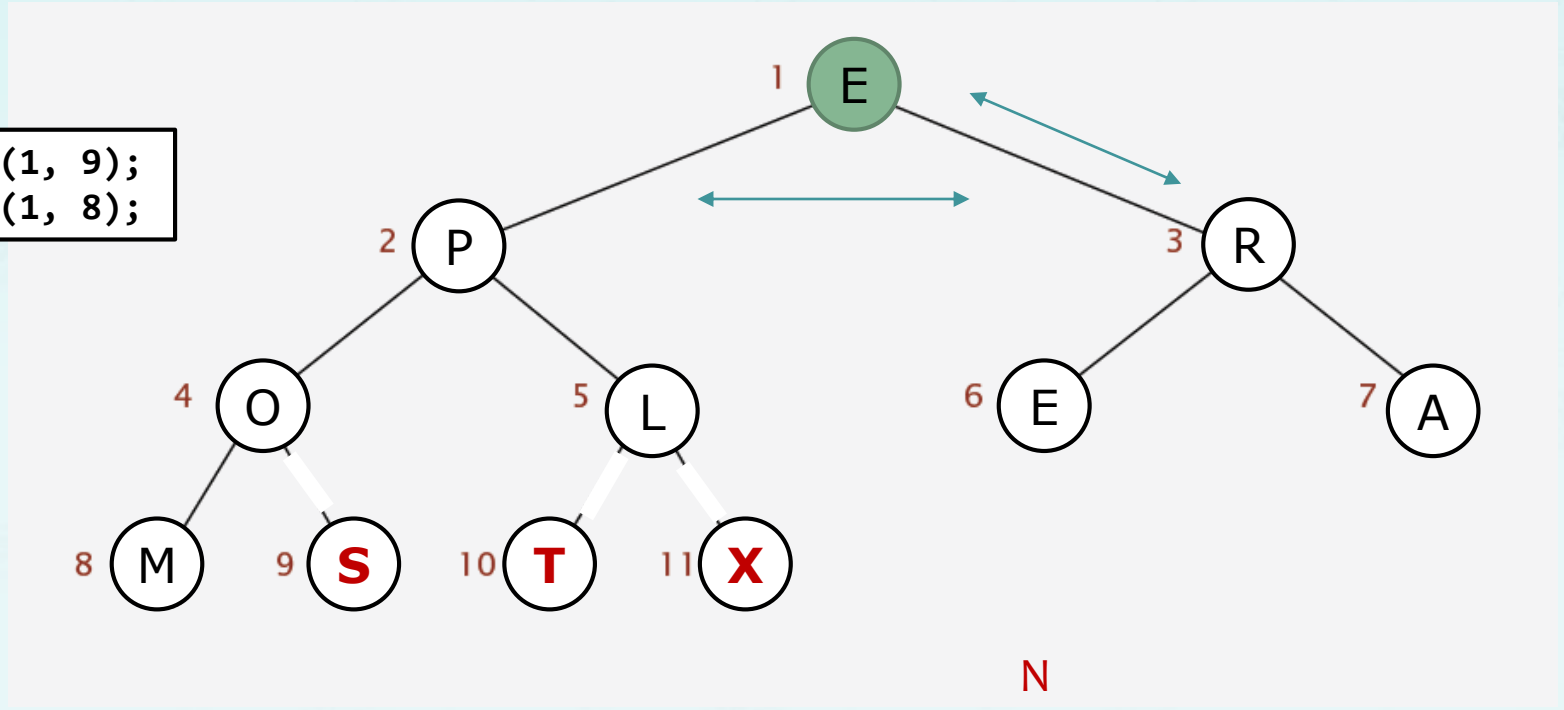
	T	P	S	O	L	R	A	M	E	T	X
	1	2	3	4	5	6	7	8	9	10	11

N

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
swap(1, 9);
sink(1, 8);
```



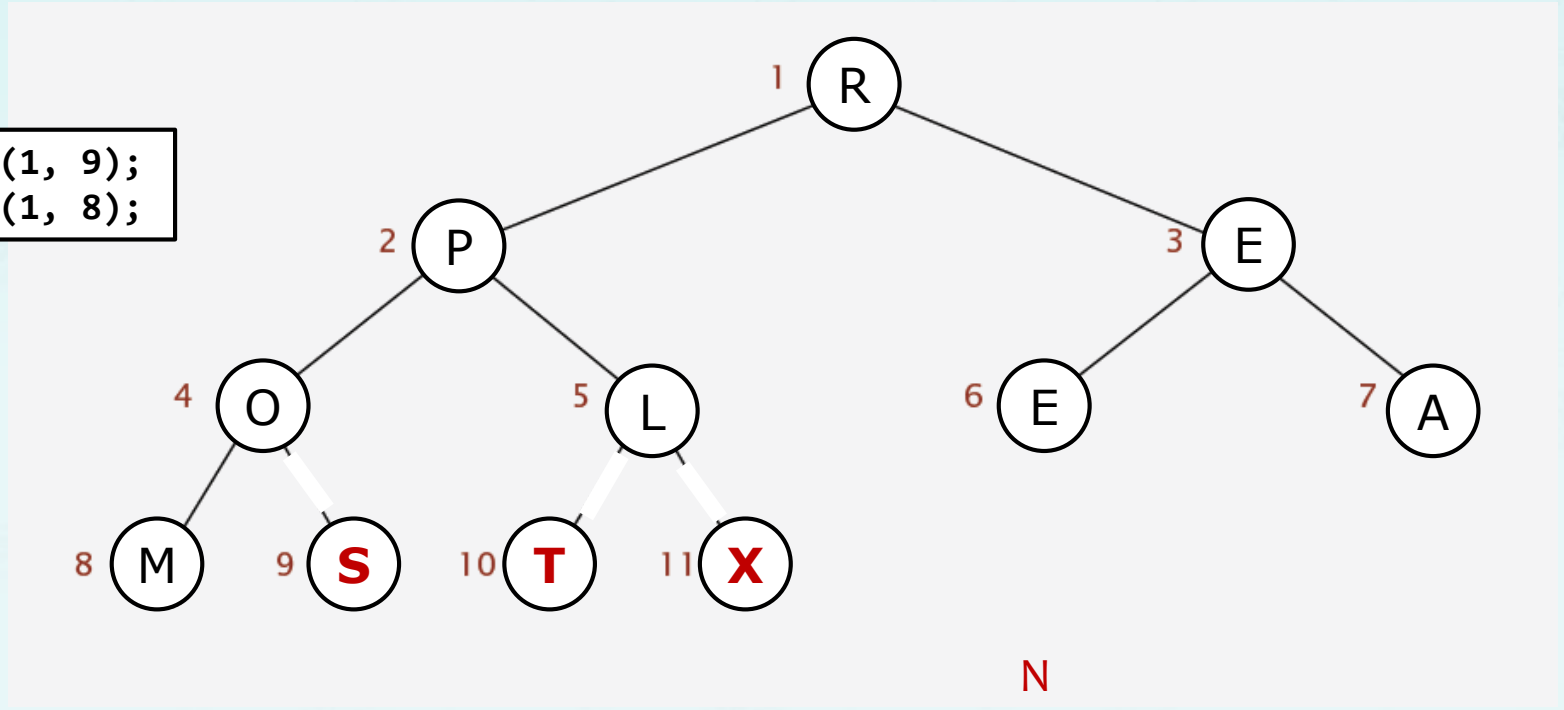
N

E	P	S	O	L	R	A	M	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

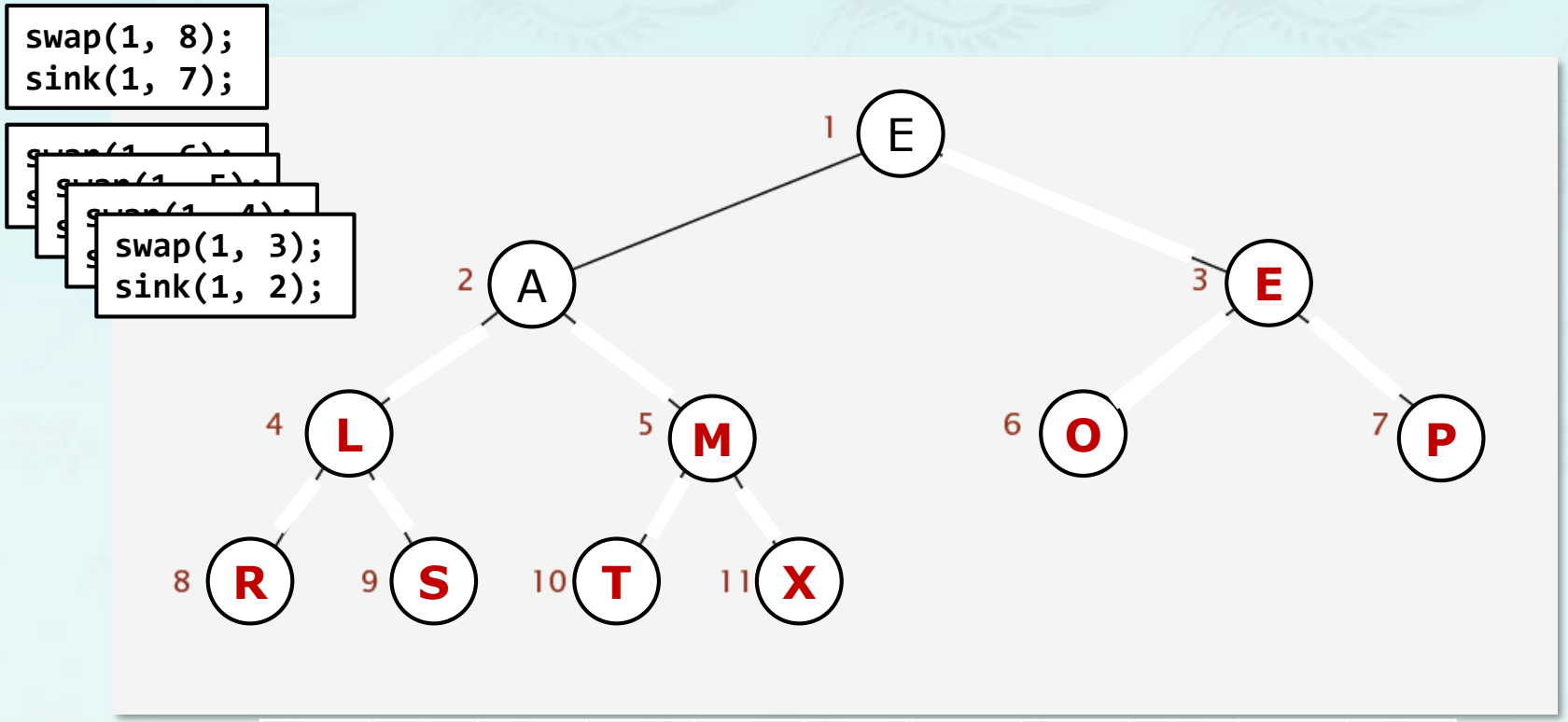
```
swap(1, 9);
sink(1, 8);
```



	N									
R	P	E	O	L	R	A	M	**S**	**T**	**X**
1	2	3	4	5	6	7	8	9	10	11

Heapsort

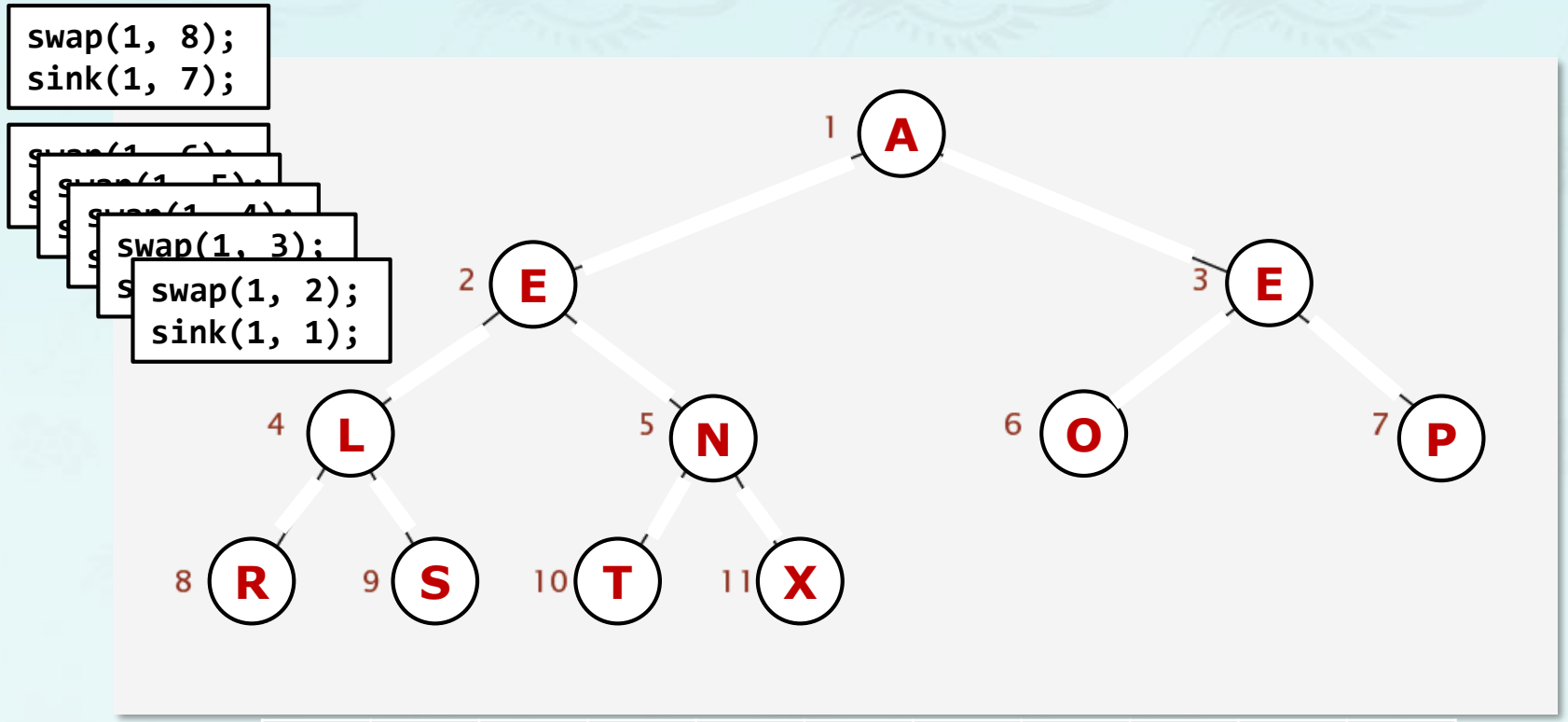
- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out



E	A	E	L	M	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

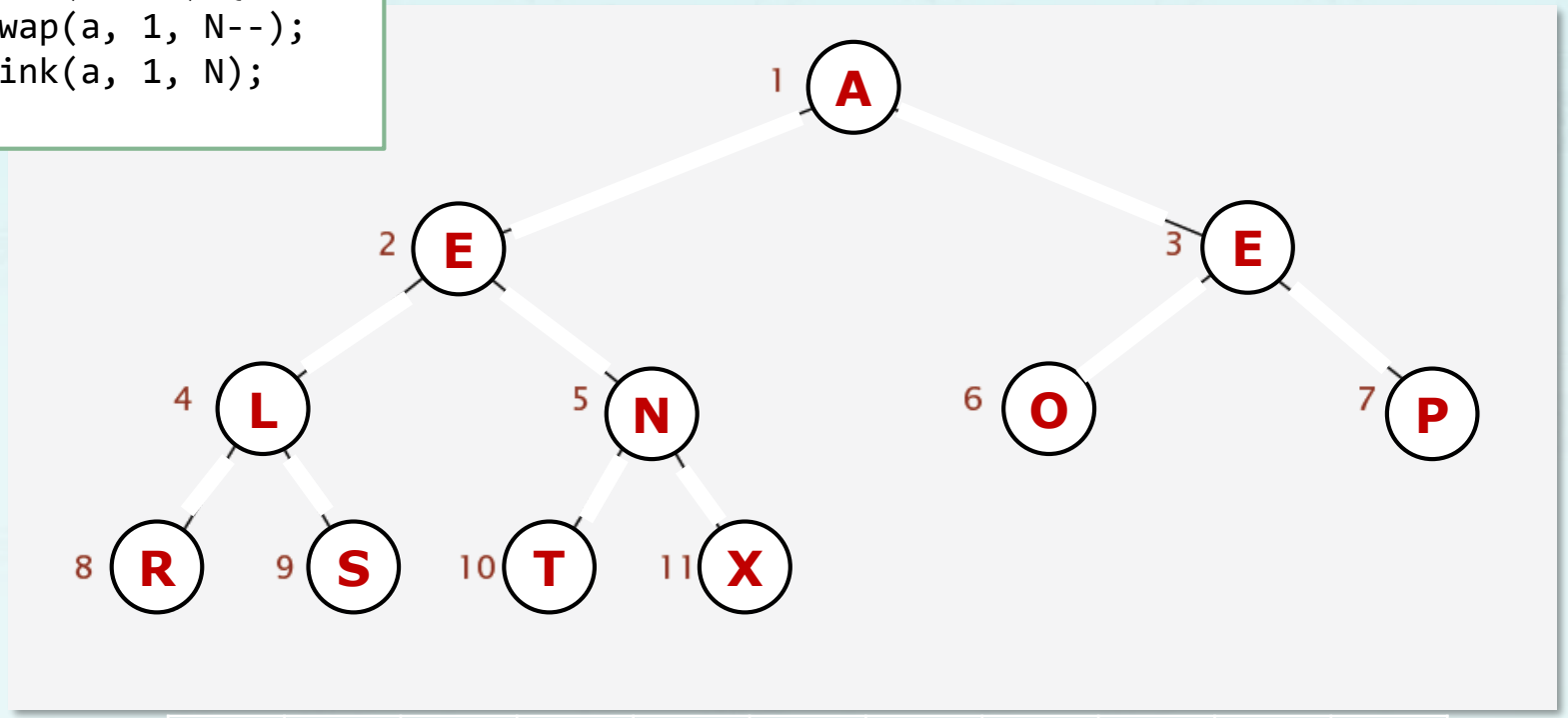


A	E	E	L	N	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort

- **2nd Pass: Repeatedly remove the maximum key.**
  - Remove the maximum, one at a time.
  - Leave them in array, instead of nulling out

```
while (N > 1) {
  swap(a, 1, N--);
  sink(a, 1, N);
}
```



A	E	E	L	N	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heap sort tracing

```
Enter a word to sort: SORTEXAMPLE
     Unsorted: S O R T E X A M P L E
   N=11  k=5: S O R T L X A M P E E
   N=11  k=4: S O R T L X A M P E E
   N=11  k=3: S O X T L R A M P E E
   N=11  k=2: S T X P L R A M O E E
   N=11  k=1: X T S P L R A M O E E
Heap ordered: X T S P L R A M O E E
```

printed in main()

printed in main()

1st path printed in sink()

printed in heapSort()

Heap sort tracing

```
Enter a word to sort: SORTEXAMPLE
     Unsorted: S O R T E X A M P L E
    N=11  k=5: S O R T L X A M P E E
    N=11  k=4: S O R T L X A M P E E
    N=11  k=3: S O X T L R A M P E E
    N=11  k=2: S T X P L R A M O E E
    N=11  k=1: X T S P L R A M O E E
  Heap ordered: X T S P L R A M O E E
    N=10  k=1: T P S O L R A M E E
    N= 9  k=1: S P R O L E A M E
    N= 8  k=1: R P E O L E A M
    N= 7  k=1: P O E M L E A
    N= 6  k=1: O M E A L E
    N= 5  k=1: M L E A E
    N= 4  k=1: L E E A
    N= 3  k=1: E A E
    N= 2  k=1: E A
    N= 1  k=1: A
       Sorted: A E E L M O P R S T X
```

← printed in main()

← printed in main()

1st path printed in sink()

← printed in heapSort()

2nd path printed in sink()

← printed in main()

Heap sort tracing

```
Enter a word to sort: SORTEXAMPLE
     Unsorted: S O R T E X A M P L E
    N=11  k=5: S O R T L X A M P E E
    N=11  k=4: S O R T L X A M P E E
    N=11  k=3: S O X T L R A M P E E
    N=11  k=2: S T X P L R A M O E E
    N=11  k=1: X T S P L R A M O E E
  Heap ordered: X T S P L R A M O E E
    N=10  k=1: T P S O L R A M E E
    N= 9  k=1: S P R O L E A M E
    N= 8  k=1: R P E O L E A M
    N= 7  k=1: P O E M L E A
    N= 6  k=1: O M E A L E
    N= 5  k=1: M L E A E
    N= 4  k=1: L E E A
    N= 3  k=1: E A E
    N= 2  k=1: E A
    N= 1  k=1: A
       Sorted: A E E L M O P R S T X
```

printed in main()

printed in main()

1st path printed in sink()

printed in heapSort()

2nd path printed in sink()

printed in main()

- **NOTE: This implementation does not sort the first element in the array.**
- **NOTE: N=?? k=? .... lines are outputs at the end of each sink()**

heap data structure

- *complete binary tree*
- *priority queues (Chapter 9)*
- *binary heap and min-heap*
- *maxheap demo*
- *maxheap implementation*
- ***heap sort (Chapter 7)***