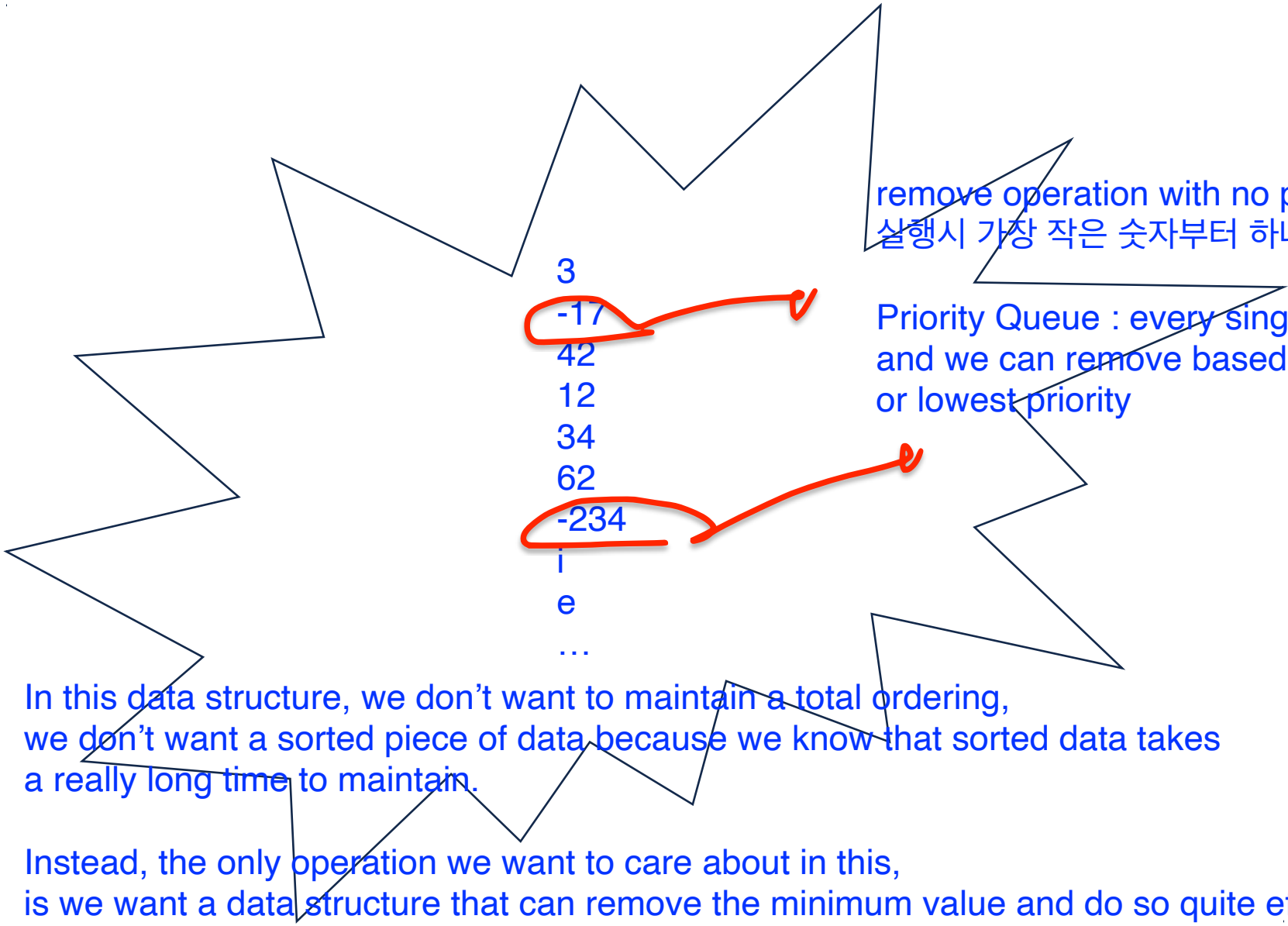
CS 400

Heap - Introduction

ID: 10-01

Priority Queue



remove operation with no parameter
실행시 가장 작은 숫자부터 하나씩 제거

Priority Queue : every single value can have a priority
and we can remove based on the value that has the highest
or lowest priority

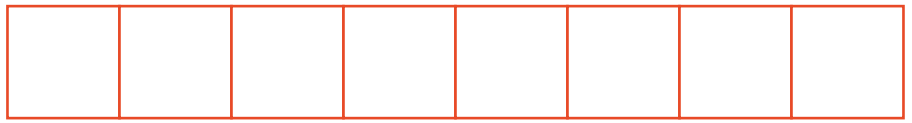
In this data structure, we don't want to maintain a total ordering,
we don't want a sorted piece of data because we know that sorted data takes
a really long time to maintain.

Instead, the only operation we want to care about in this,
is we want a data structure that can remove the minimum value and do so quite efficiently.

Priority Queue Implementation

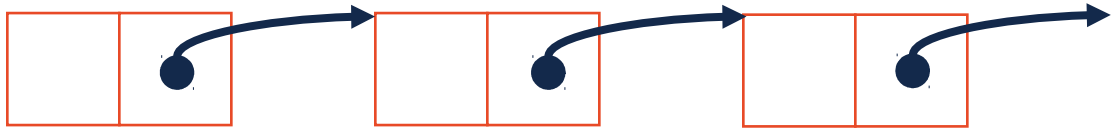
insert	removeMin
greate insertion time,	but terrible remove time
O(1)*	O(n)
O(1)	O(n)
greate remove time,	but terrible insertion time
O(n)	O(1)
O(n)	O(1)

unsorted both the left and the right child nodes

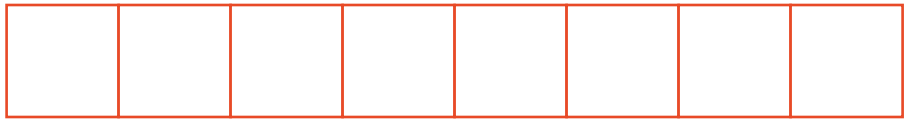


unsorted list

unsorted

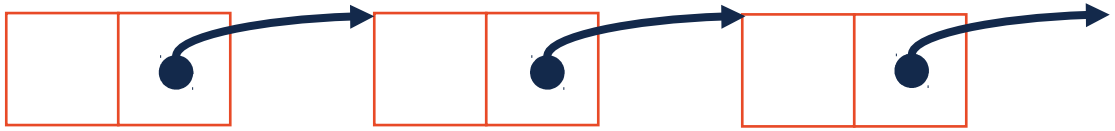


sorted array

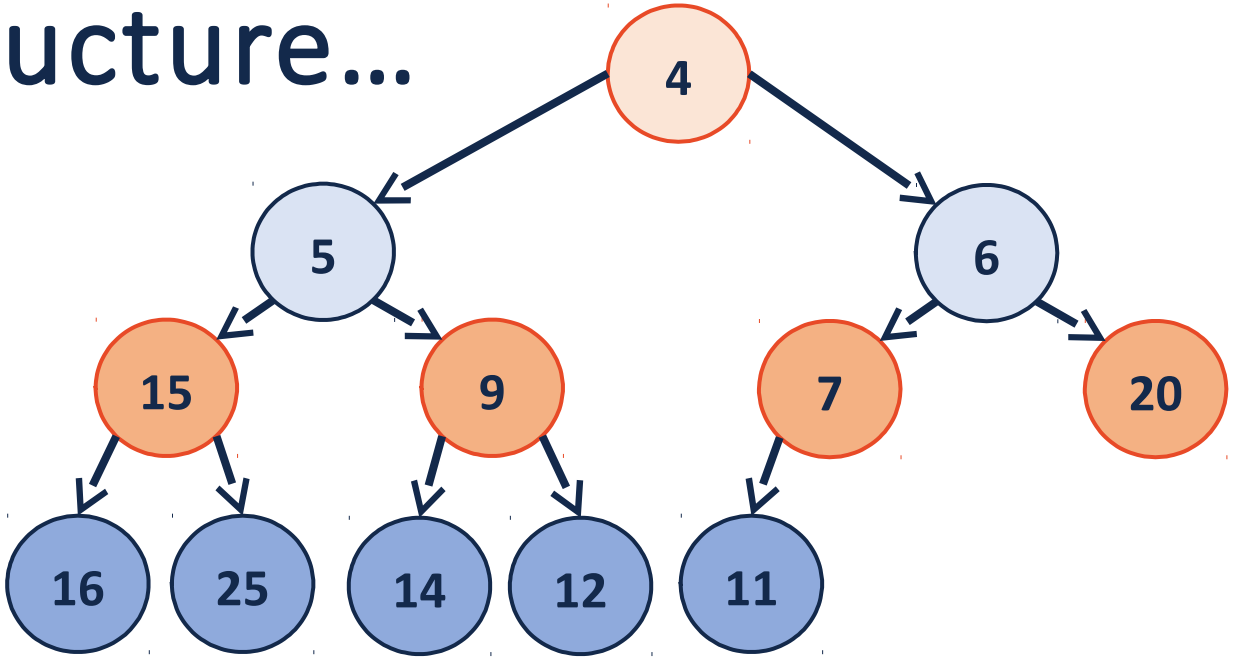


sorted list

sorted



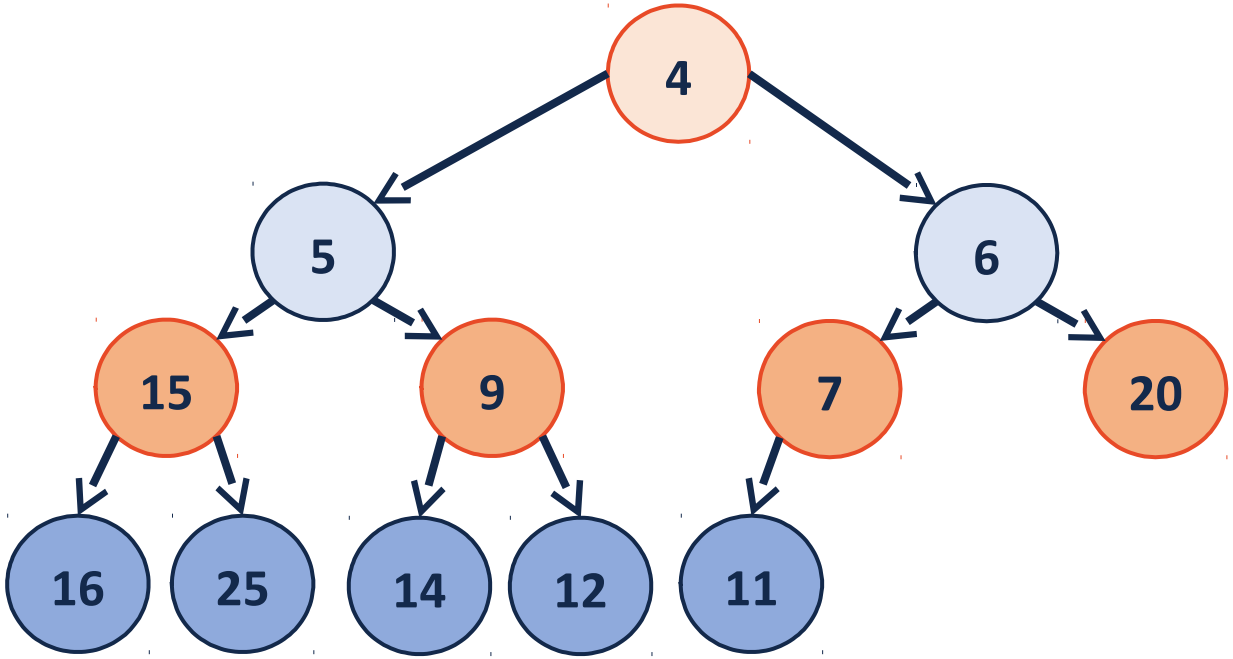
Another possibly structure…



(min)Heap

A complete binary tree T is a min-heap if:

- **T = {}** or
- **T = {r, T_L, T_R}**, where **r** is less than the roots of {T_L, T_R} and {T_L, T_R} are min-heaps.

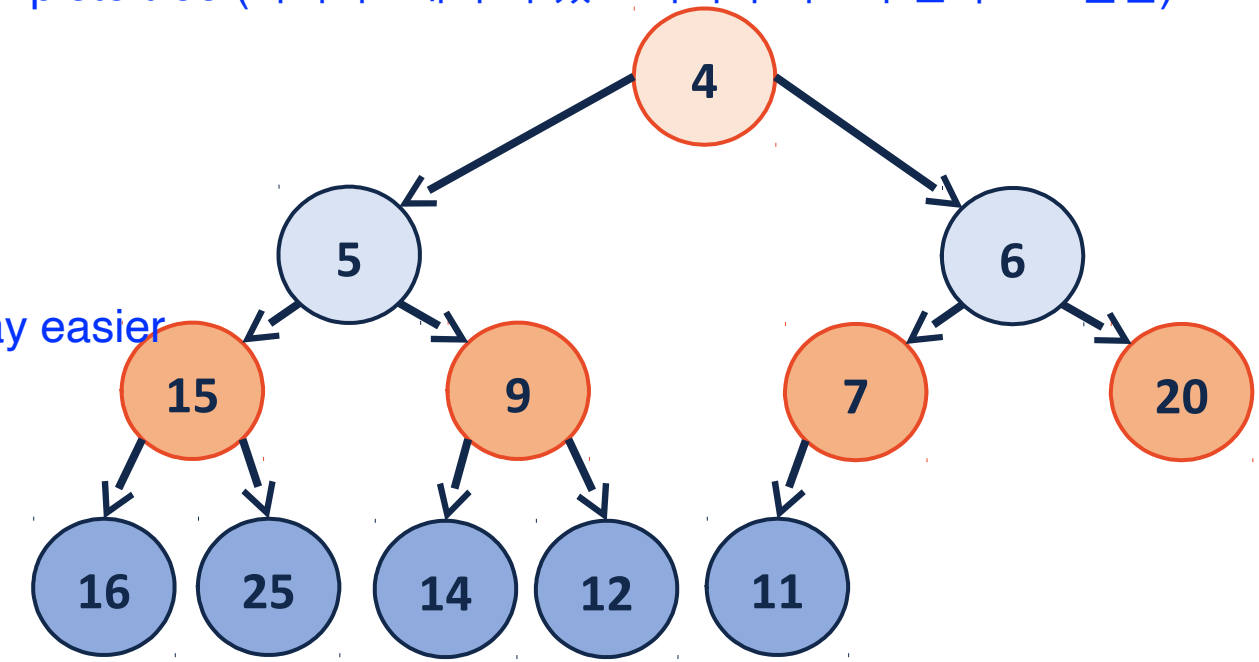


both the left and the right child nodes are larger than the root node. We don't think about siblings, only descendant has to be larger (in a min-heap) or smaller (in a max-heap) than the node itself.

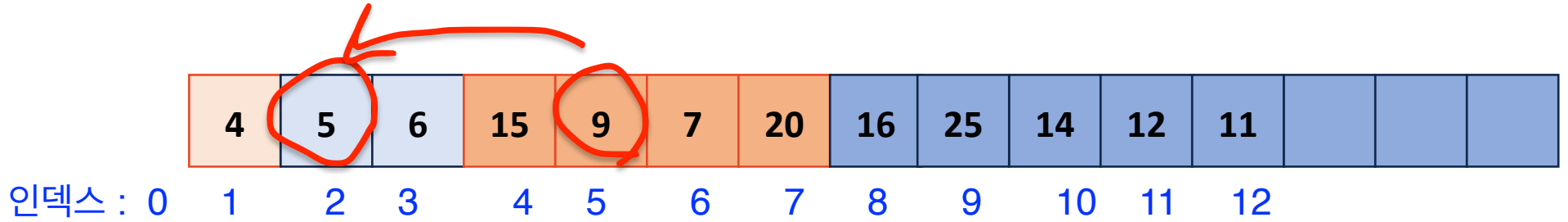
(min)Heap

heap should be a complete tree (자식이 2개씩 다 있고 마지막 리프가 왼쪽으로 쏠림)

the entire memory representation of our heap
is going to be entirely an array
but when we're actually doing the analysis,
we'll draw it as a tree because it's going to be way easier
to think about things in a tree.



9의 부모는 5.
부모노드 인덱스 = 자식노드 인덱스 / 2
5 / 2 = 2
왼쪽자식노드 인덱스 = 부모노드 인덱스 * 2
오른쪽자식노드 인덱스 = 부모노드 인덱스 * 2 + 1



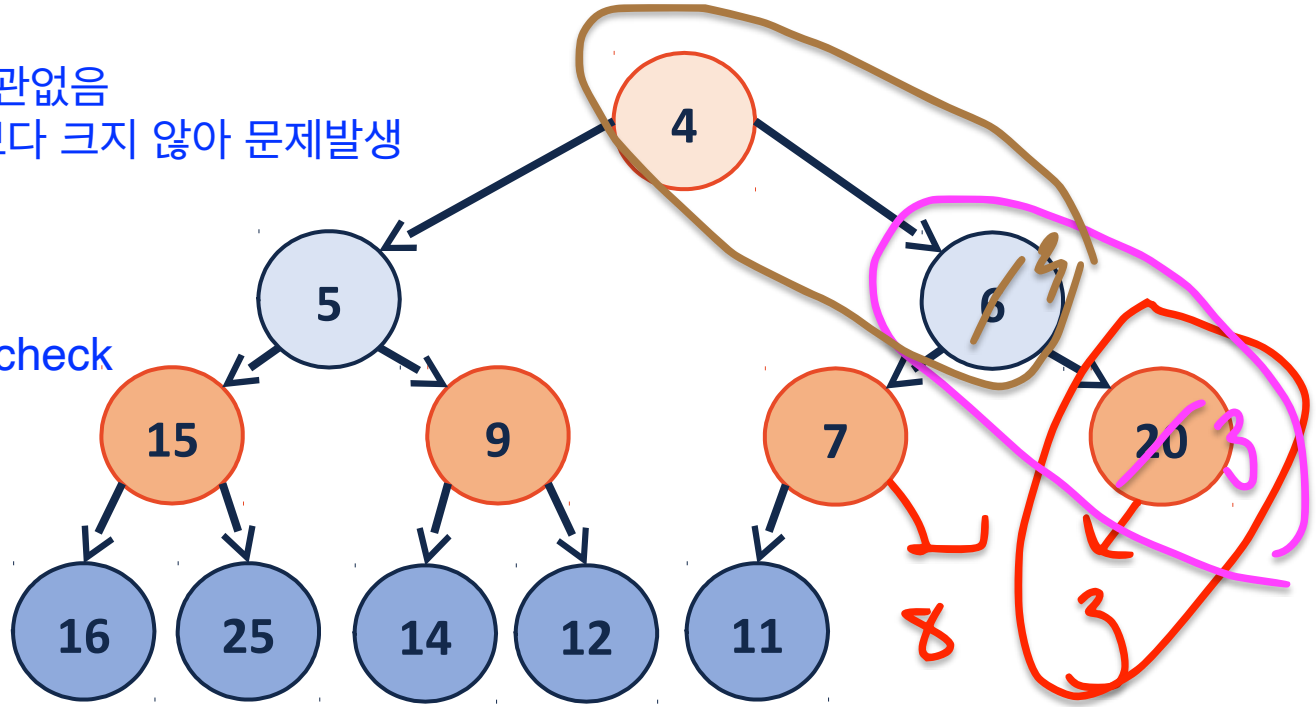
CS 400

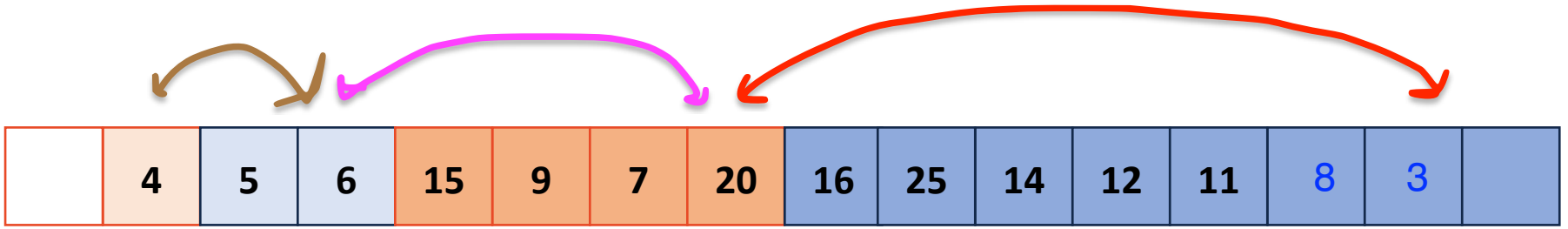
Heap – Insert and removeMin

ID: 10-02

insert

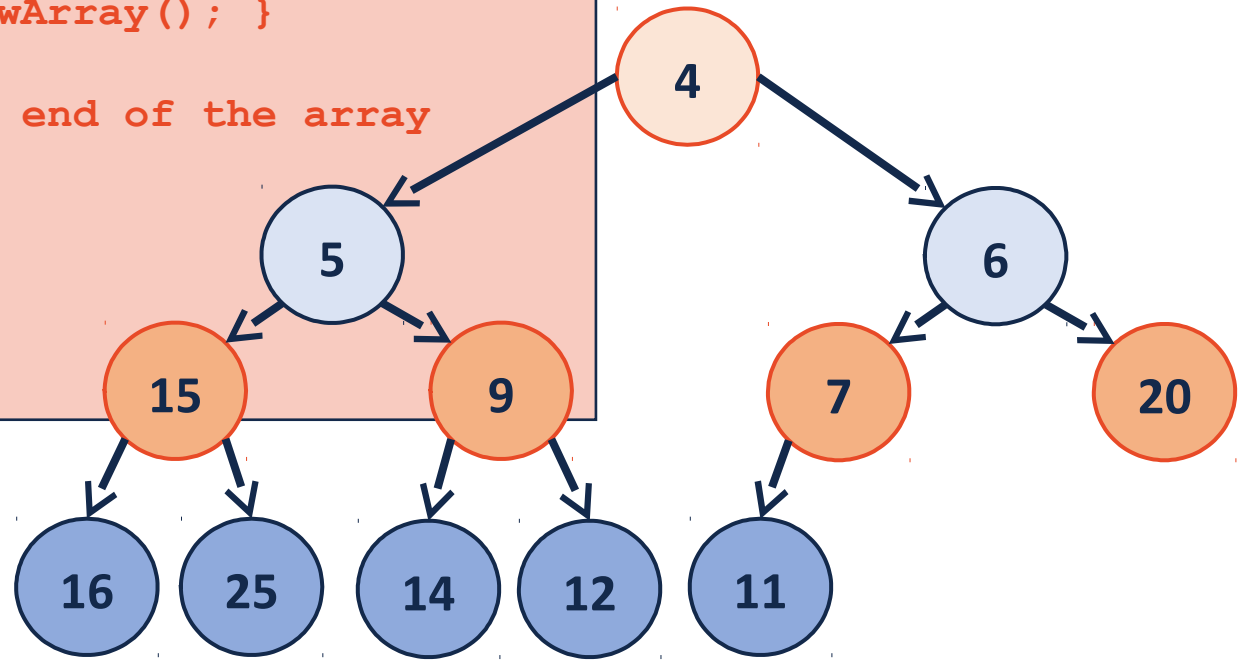
insert(8) => 오른쪽 subtree의 7, 6, 4보다 크니 상관없음
insert(3) => 20의 left child로 들어가게되므로 20보다 크지 않아 문제발생
이때 20과 3의 위치를 바꿔줌
그래도 3이 6보다 작으니 6과 위치를 다시 바꿈
그래도 3이 4보다 작으니 4와 위치를 다시 바꿈
=> keep the maintanance of tree by repeating check

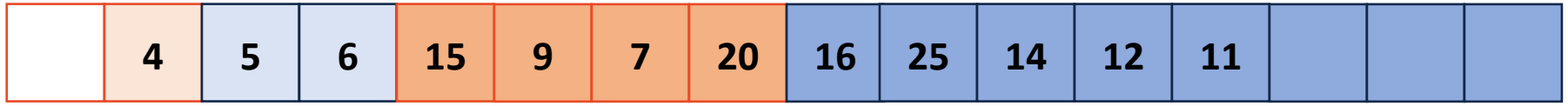




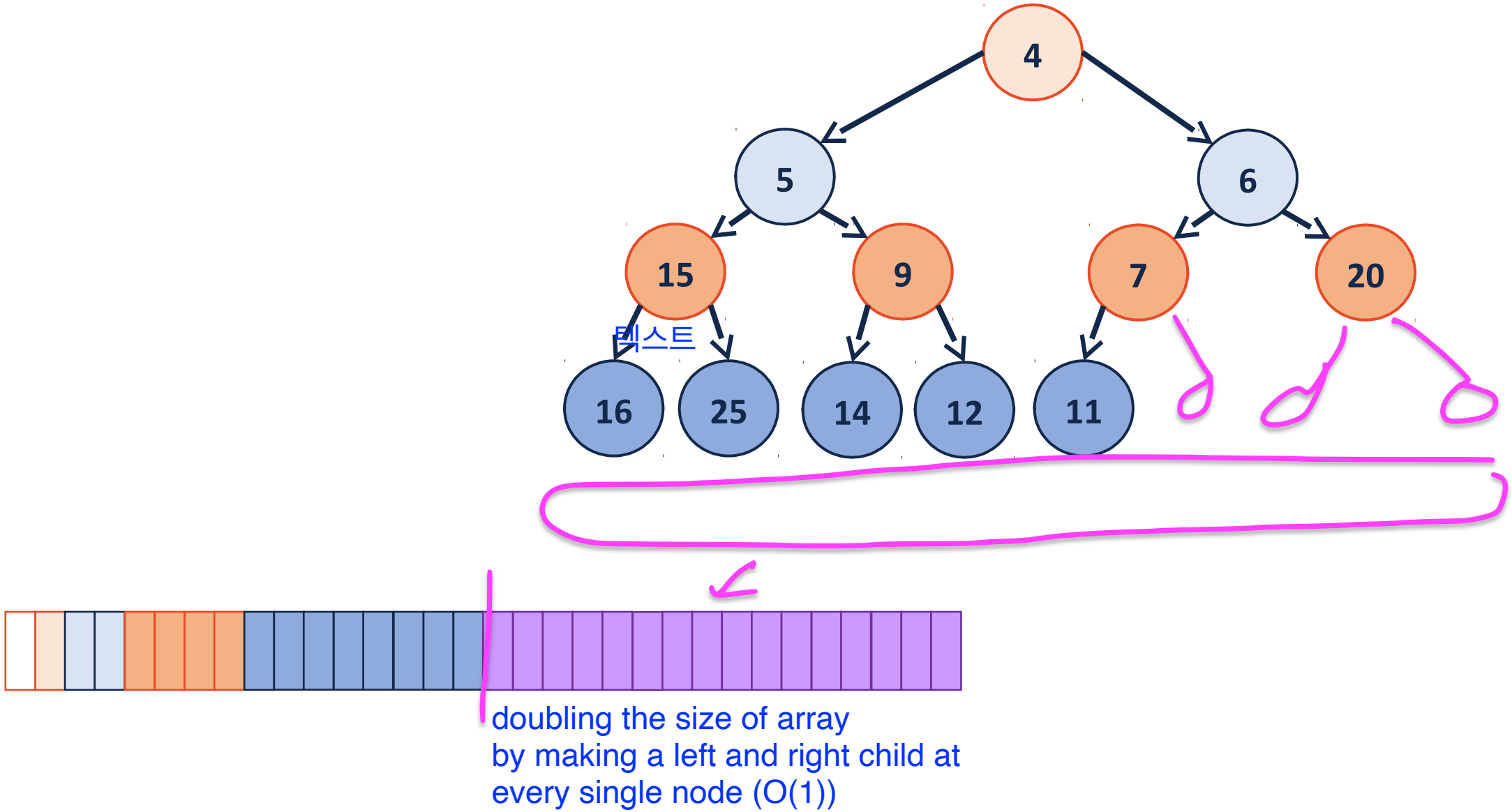
insert

```
 1  template <class T>
 2  void Heap<T>::_insert(const T & key) {
 3    // Check to ensure there's space to insert an element
 4    // ...if not, grow the array
 5    if ( size_ == capacity_ ) { _growArray(); }
 6
 7    // Insert the new element at the end of the array
 8    item_[++size] = key;
 9
10    // Restore the heap property
11    _heapifyUp(size);
12  }
```





growArray


4
5
6
15
9
7
20
16
25
14
12
11
doubling the size of array
by making a left and right child at
every single node (O(1))

insert- heapifyUp

```
1  template <class T>
2  void Heap<T>::_insert(const T & key) {
3    // Check to ensure there's space to insert an element
4    // ...if not, grow the array
5    if ( size_ == capacity_ ) { _growArray(); }
6
7    // Insert the new element at the end of the array
8    item [++size] = key;
9
10   // Restore the heap property
11    _heapifyUp(size);
12 }
```

to ensure the heap property is maintained

```
1  template <class T>
2  void Heap<T>::_heapifyUp( int index ) {
3    if ( index > 1 ) {
4      if ( item_[index] < item_[ parent(index) ] ) {
5        std::swap( item_[index], item_[ parent(index) ] );
6        _heapifyUp( parent(index) );
7      }
8    }
9  }
```

0은 건너뛰고 루트노드가 1

새로 들어온 데이터가 부모 데이터보다 작으면

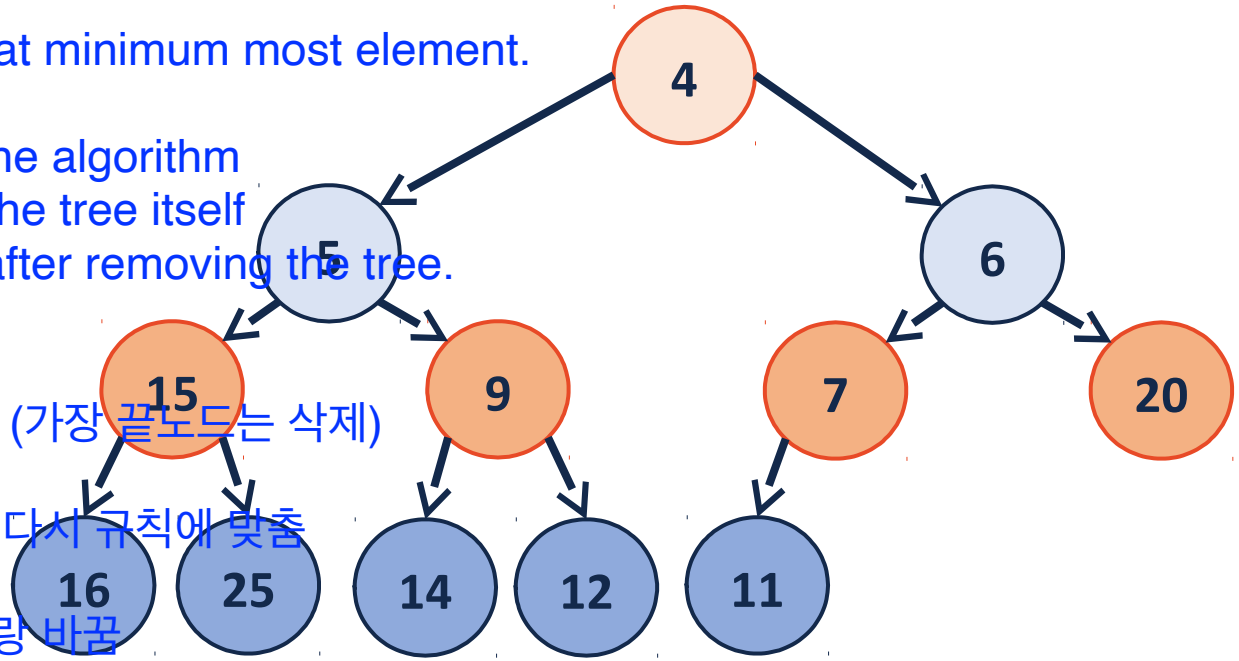
자식과 부모를 바꾸고

새 데이터(이제는 부모노드)로 재귀호출

removeMin

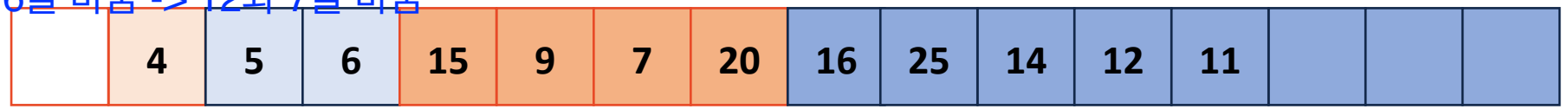
There is no operation on the heap
that we can do removal except for removing that minimum most element.

So to do so, we're going to have to figure out the algorithm
to take the top element of the tree, the root of the tree itself
and actually maintain the heap property even after removing the tree.



< removeMin() >
- 가장 작은값인 루트노드 4와 가장 끝값인 11을 바꿈 (가장 끝노드는 삭제)
(트리구조를 망가트리지 않기위해)
- heapifyDown 으로 루트노트(11) 부터 내려가면서 다시 규칙에 맞춤
(자식노드가 부모노드보다 커야한다)
- heapifyDown : 한층 밑 자식노드중 가장 작은값이랑 바꿈
(11과 5를 바꿈 -> 11과 9를 바꿈)

< removeMin() >
- 루트노드가 된 5를 12와 바꿈 (가장 끝노드는 삭제)
- 12와 6을 바꿈 -> 12와 7을 바꿈

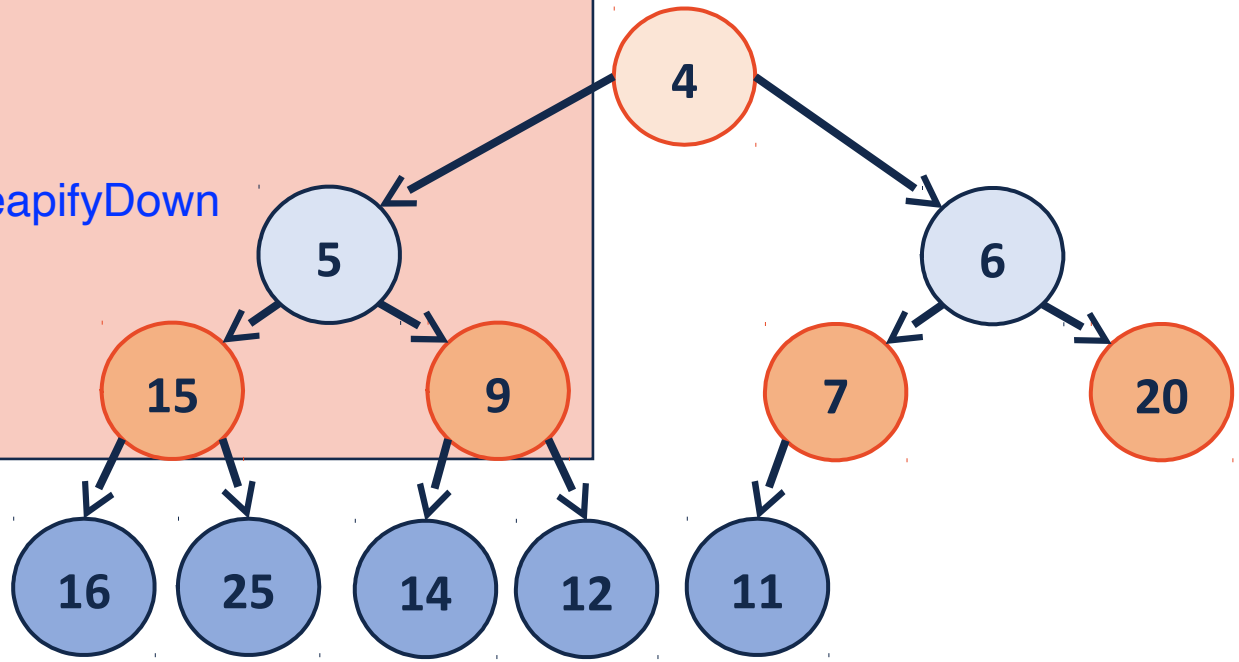


removeMin

```
 1  template <class T>
 2  void Heap<T>::_removeMin() {
 3    // Swap with the last value
 4    T minValue = item_[1];
 5    item_[1] = item_[size_];
 6    size--;
 7
 8    // Restore the heap property
 9    heapifyDown();
10
11    // Return the minimum value
12    return minValue;
13  }
```

swap with last element

heapifyDown



removeMin - heapifyDown

```
 1  template <class T>
 2  void Heap<T>::_removeMin() {
 3    // Swap with the last value
 4    T minValue = item_[1];
 5    item_[1] = item_[size_];
 6    size--;
 7
 8    // Restore the heap property
 9    _heapifyDown();
10
11    // Return the minimum value
12    return minValue;
13  }
```

```
 1  template <class T>
 2  void Heap<T>::_heapifyDown(int index) {
 3    if ( !_isLeaf(index) ) {
 4      T minChildIndex = _minChild(index);
 5      if ( item_[index]  >  item_[minChildIndex] ) {
 6         std::swap( item_[index], item_[minChildIndex] );
 7         _heapifyDown( minChildIndex );
 8      }
 9    }
10  }
```

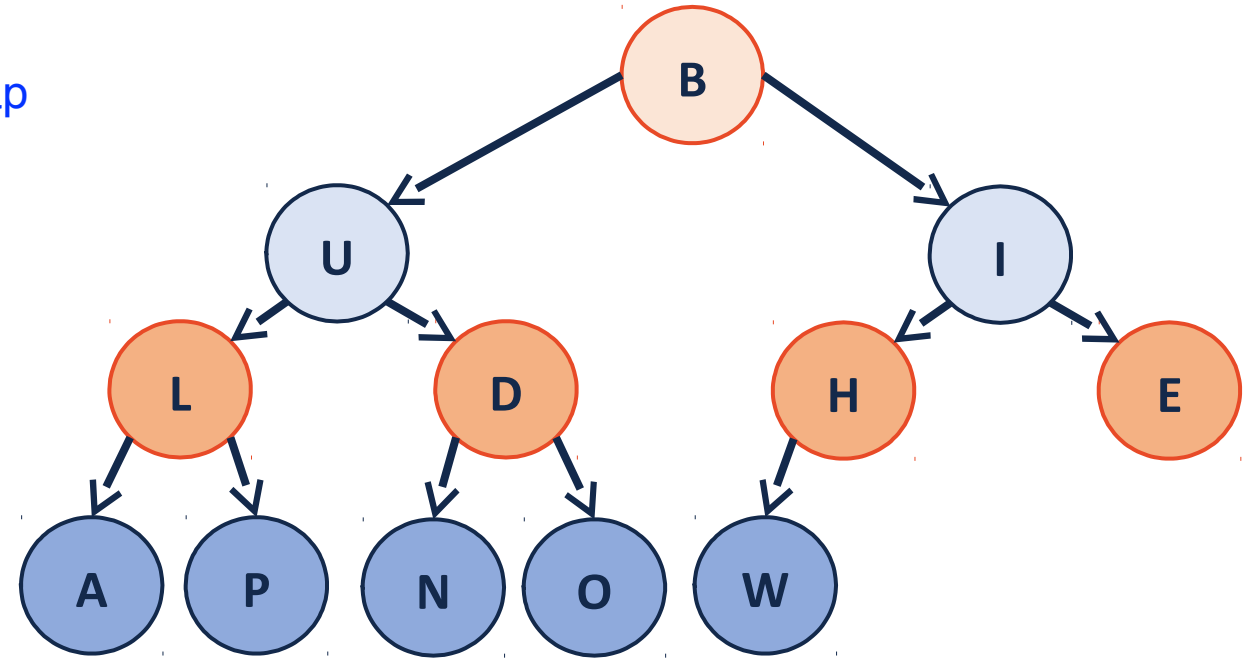
CS 400

Heap – buildHeap

ID: 10-03

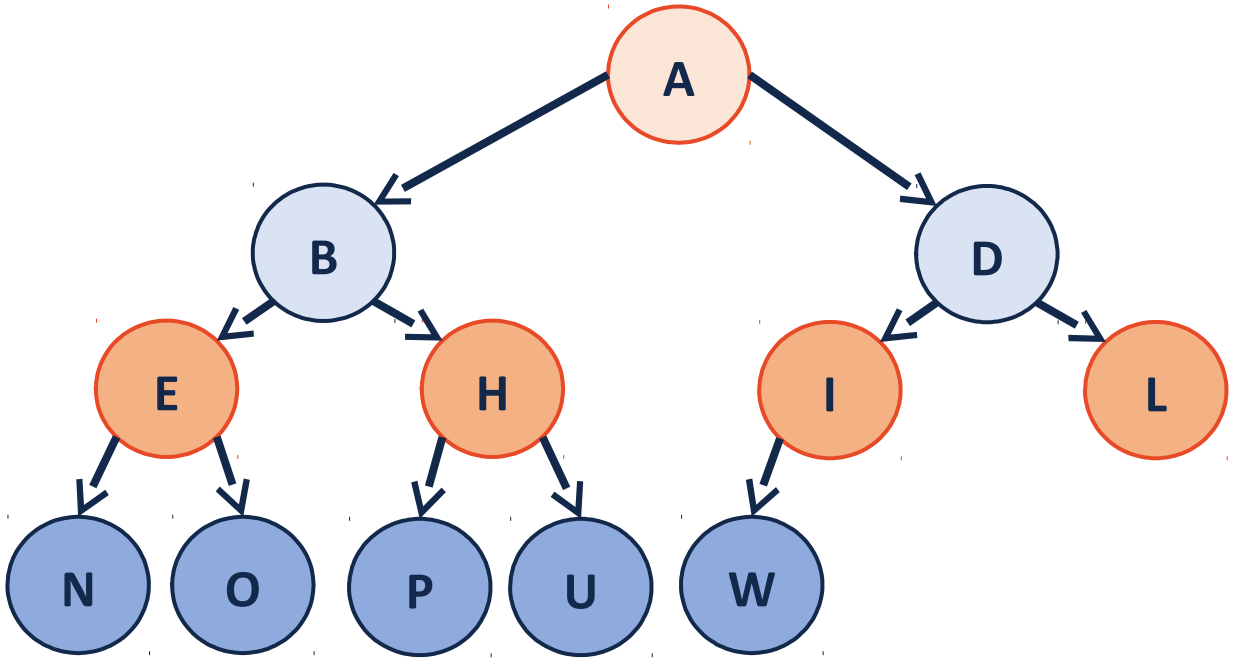
buildHeap

- binary tree랑 비슷한데 heap을 쓰는 이유 :
it's really efficient about actually building a heap from the heap operations we have



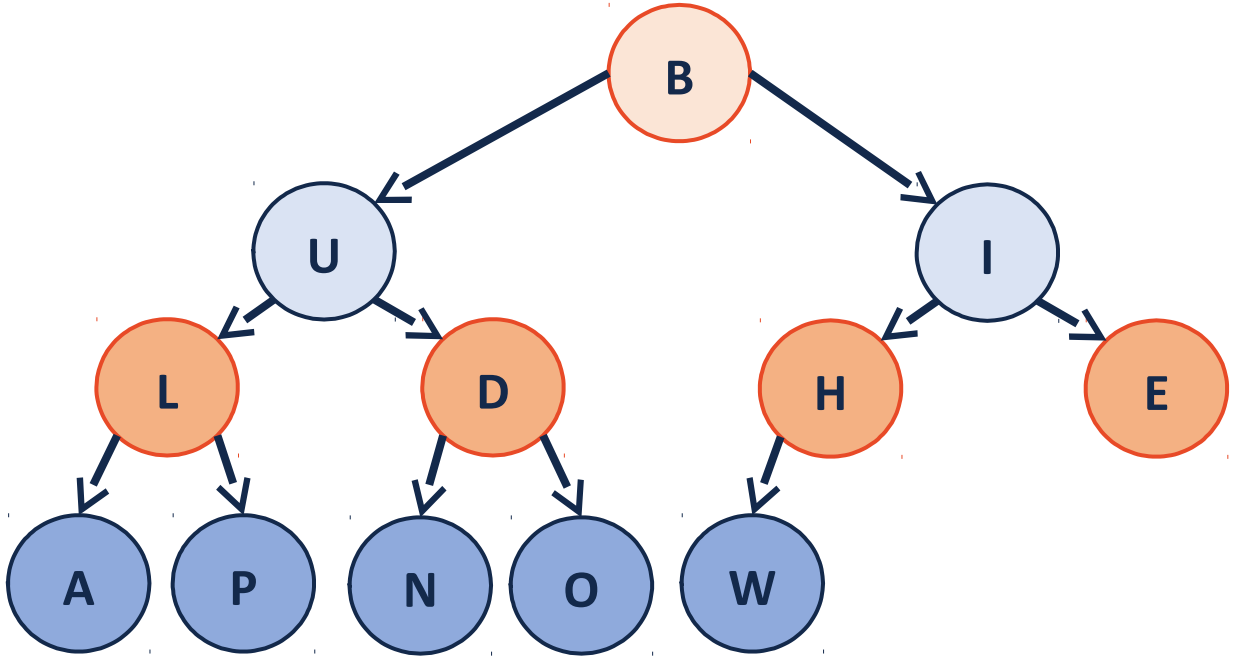
buildHeap – sorted array

	B	U	I	L	D	H	E	A	P	N	O	W			

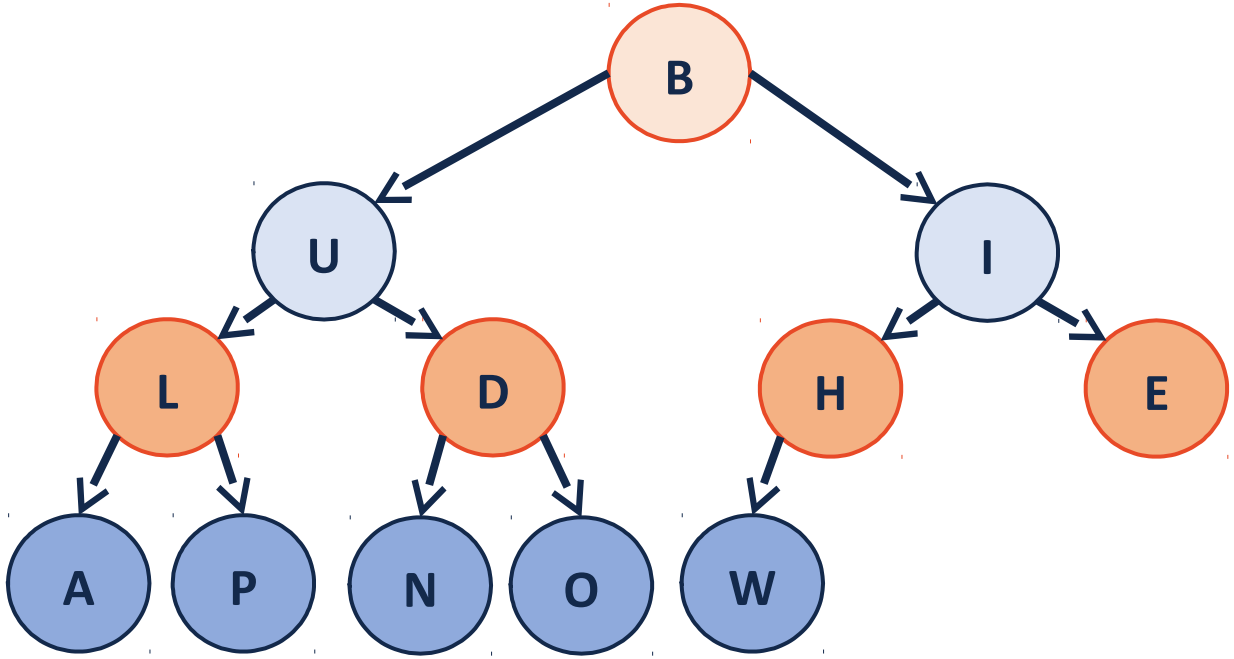


	A	B	D	E	H	I	L	N	O	P	U	W			

buildHeap- heapifyUp



buildHeap- heapifyDown



buildHeap

Key reason that we're care about building heaps : given any sort of data structure, we can build a heap notation of that in just O(logN) time.

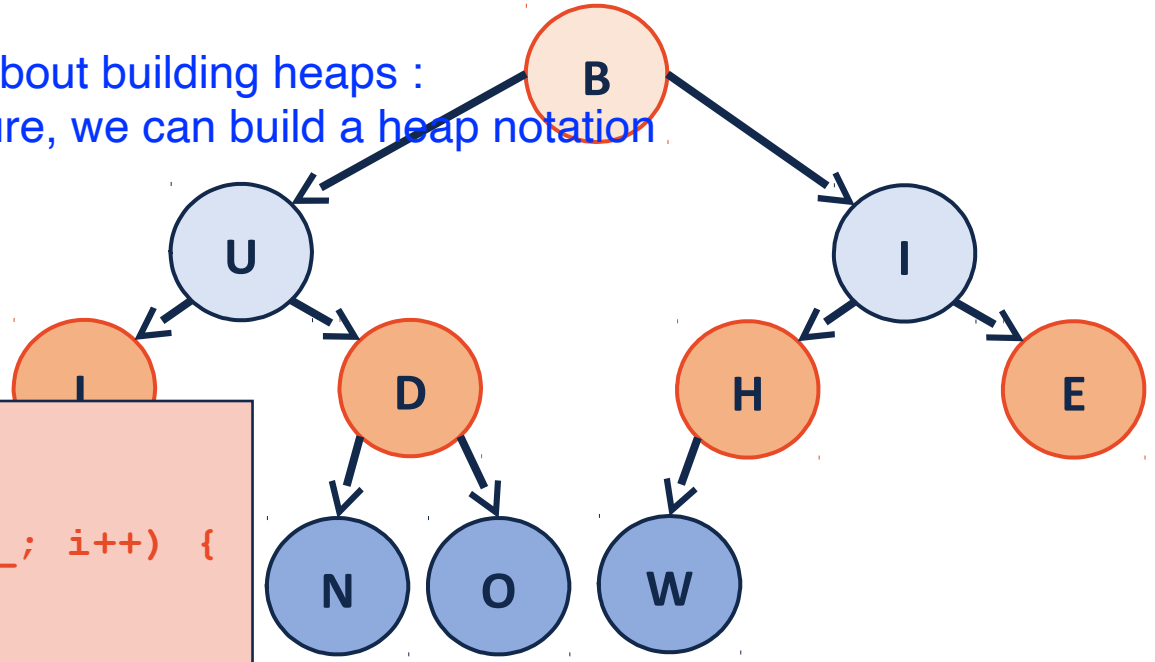
1. Sort the array – it's a heap!
2.

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = 2; i <= size_; i++) {
4      heapifyUp(i);
5    }
6  }
```

3.

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = parent(size); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

By only using heapifyDown operation, we're saying that it doesn't actually matter ~~what's in our very last level of the tree.~~ It's already balanced



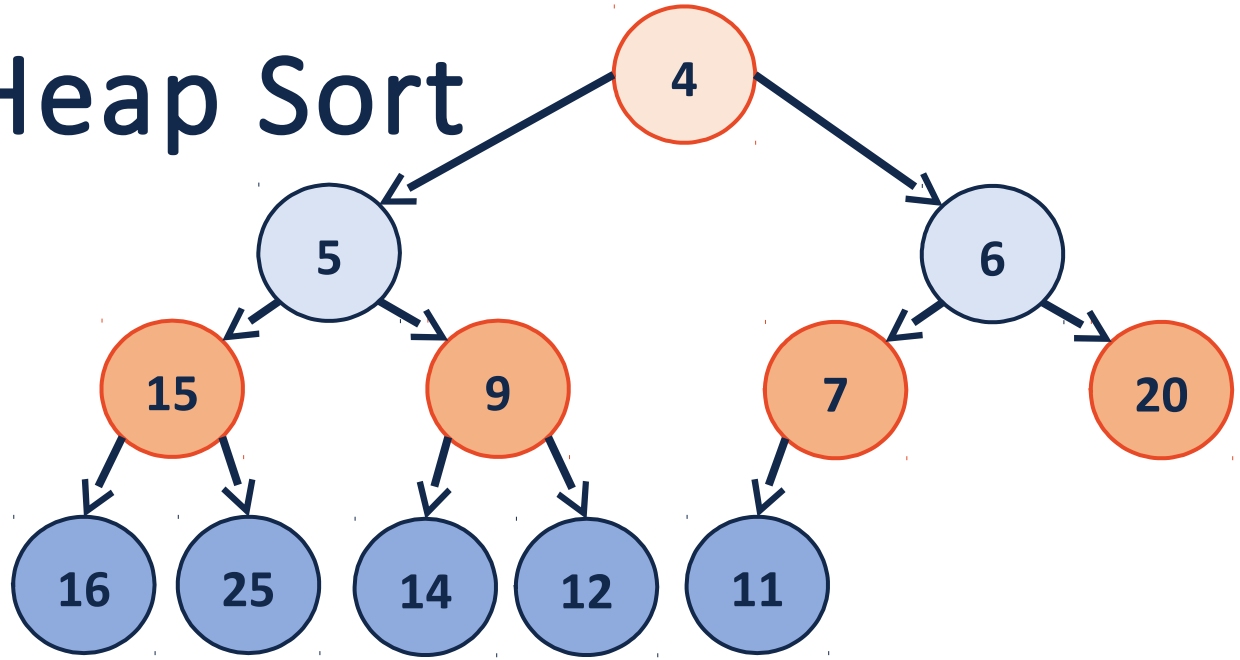
	B	U	I	L	D	H	E	A	P	N	O	W			

CS 400

Heap – Runtime Analysis

ID: 10-04

Heap Sort



1. Build Heap O(n)
2. n * removeMin() O(log(n))
3. swap element to order a list (asc/desc)

Running Time?

worst case : n * log(n)

Why do we care about another sort?