CS 225

Data Structures

March 3 – Quickselect and Priority Queues

G Carl Evans

Power of the lambda

main.cpp

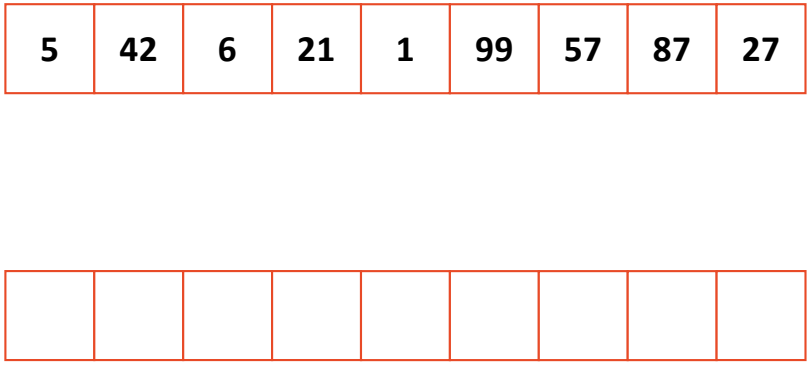
```
29    int big;
30    std::cout << "How big is big? ";
31    std::cin >> big;
32
33    auto isbig = [big](int num) { return (num >= big); };
34
35    std::cout << "There are " << Countif(numbers.begin(), numbers.end(), isbig)
36      << " big numbers" << std::endl;
37  }
38
```

Quickselect Test

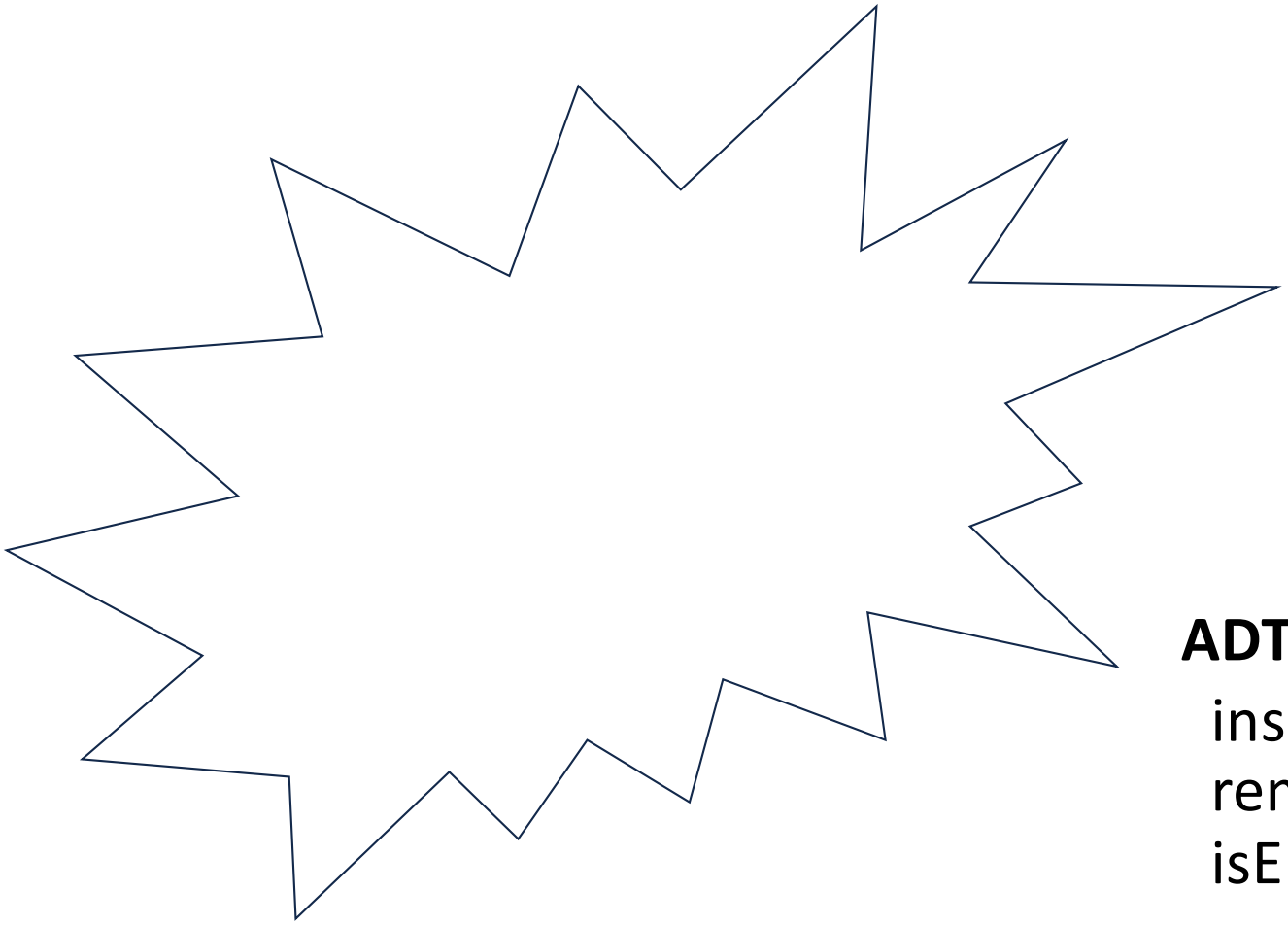
main.cpp

```
84 TEST_CASE("select simple","[weight=1][part=1]") {
85   vector<int>numbers = {5, 42, 6, 21, 1, 99, 57, 87, 27};
86
87   auto cmp = [](int lhs, int rhs){return lhs <= rhs;};
88
89   select(std::begin(numbers), std::end(numbers), std::begin(numbers) + 1, cmp);
90
91   REQUIRE(!isSorted(numbers.begin(), numbers.end(), cmp));
92
93   REQUIRE(numbers[1] == 5);
94 }
```

Quickselect



Secret, Mystery Data Structure

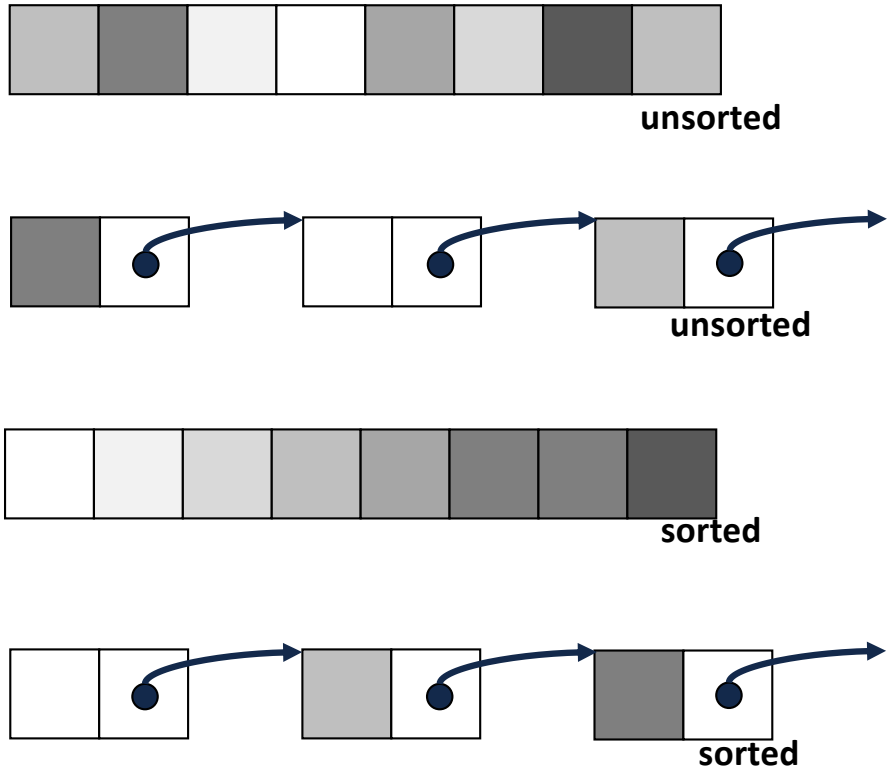


ADT:

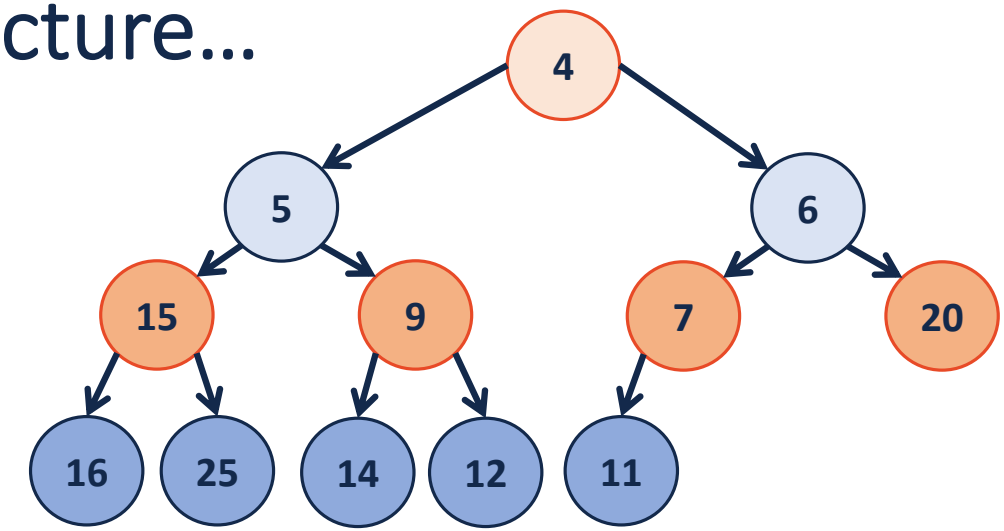
insert
remove
isEmpty

Priority Queue Implementation

insert	removeMin
O(1)	O(n)
O(1)	O(n)
O(n)	O(1)
O(n)	O(1)



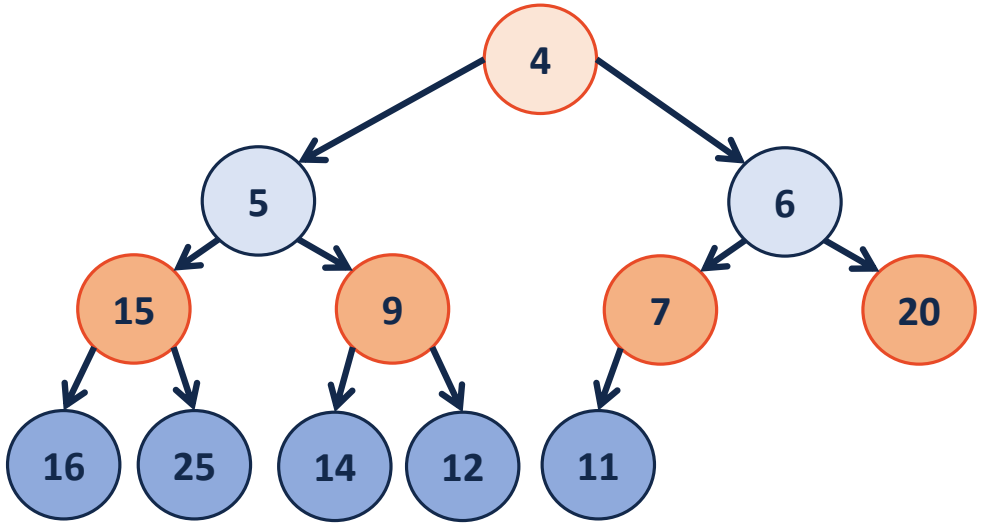
Another possibly structure…



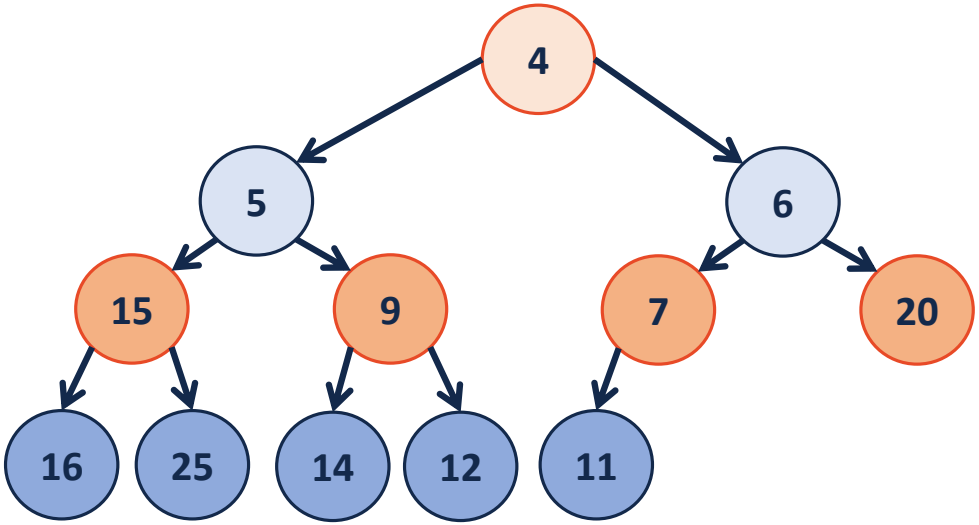
(min)Heap

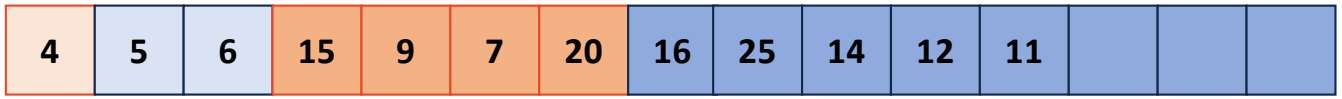
A complete binary tree T is a min-heap if:

- **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.

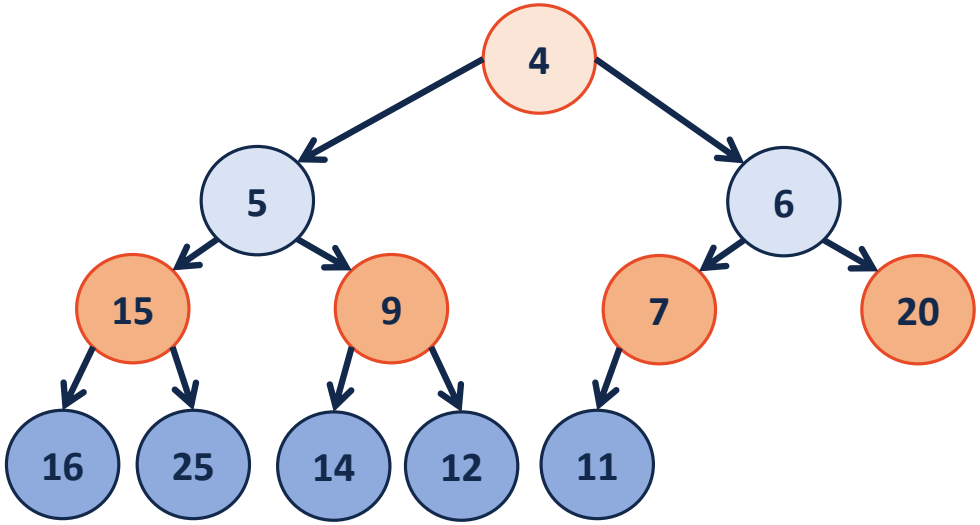


(min)Heap



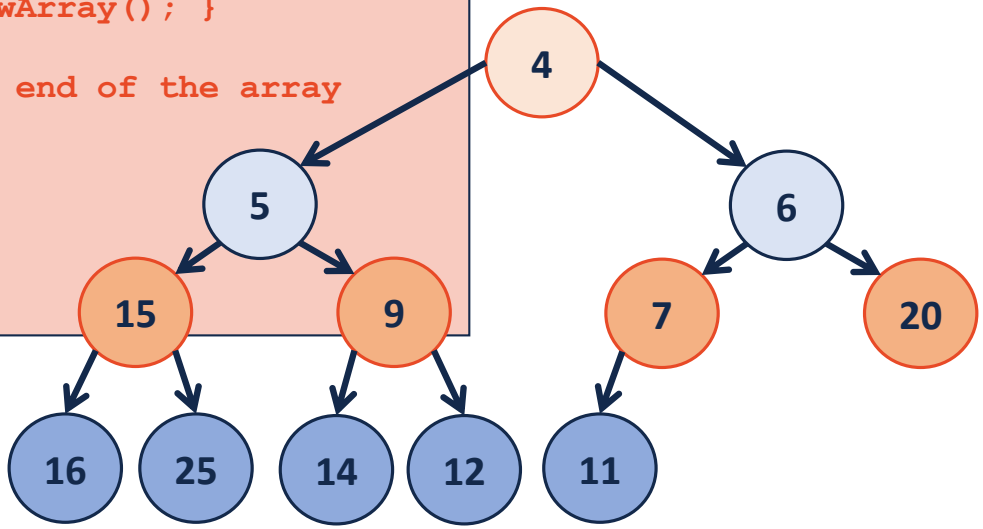


insert

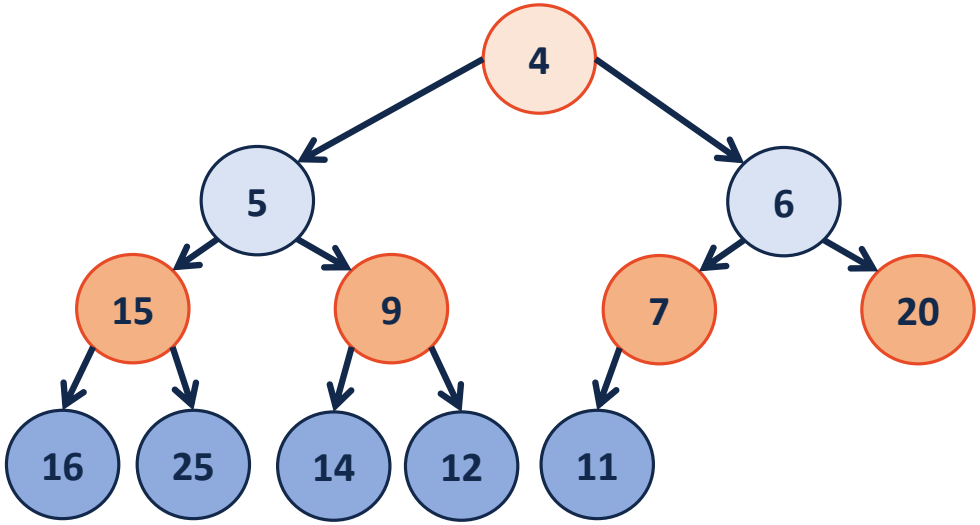


insert

```
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



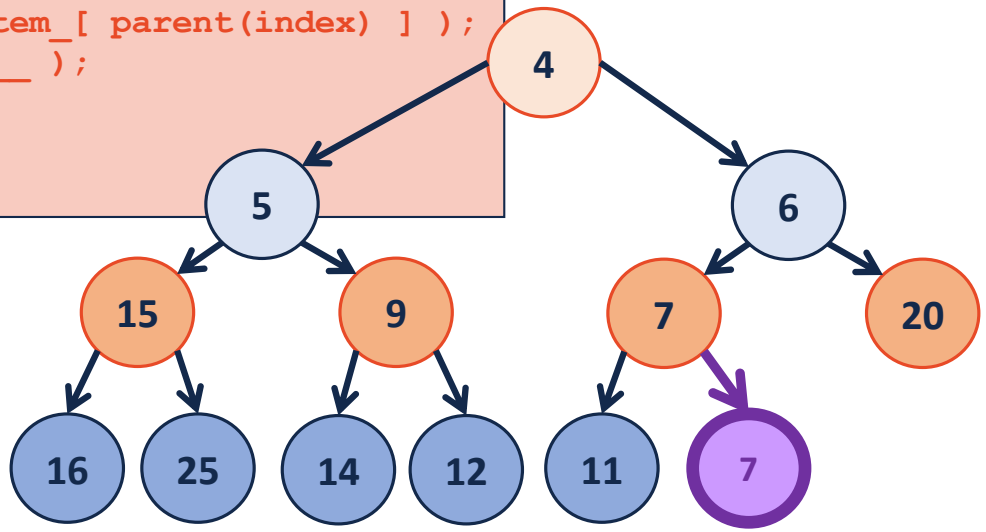
insert - heapifyUp

```
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 }
```

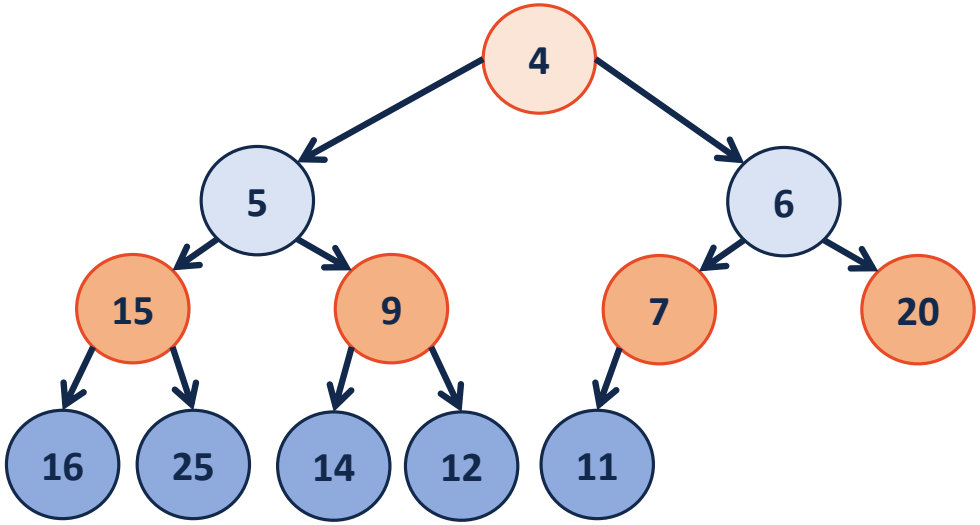
```
1  template <class T>
2  void Heap<T>::_heapifyUp( ____________________ ) {
3    if ( index > __________ ) {
4      if ( item_[index] < item_[ parent(index) ] ) {
5         std::swap( item_[index], item_[ parent(index) ] );
6         _heapifyUp( ___________________ );
7      }
8    }
9  }
```

heapifyUp

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

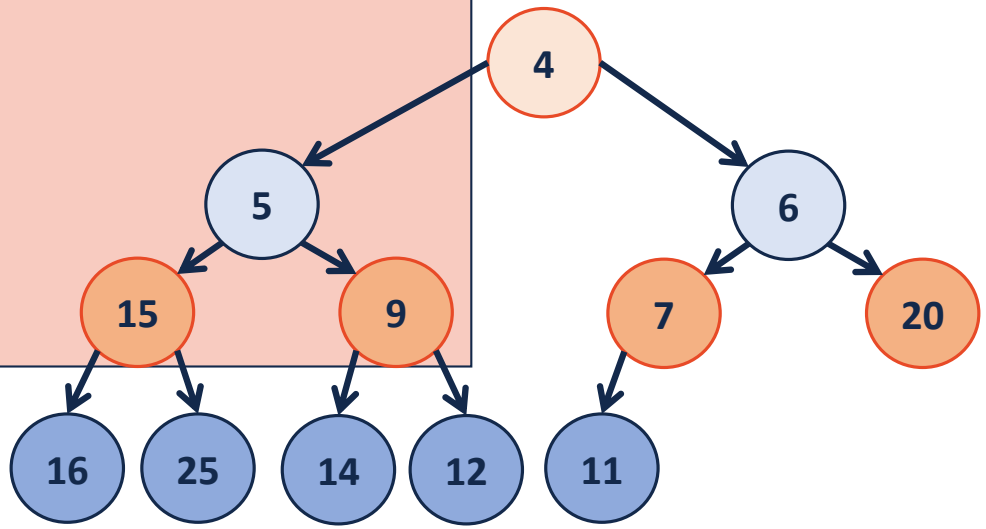


removeMin



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  }
```



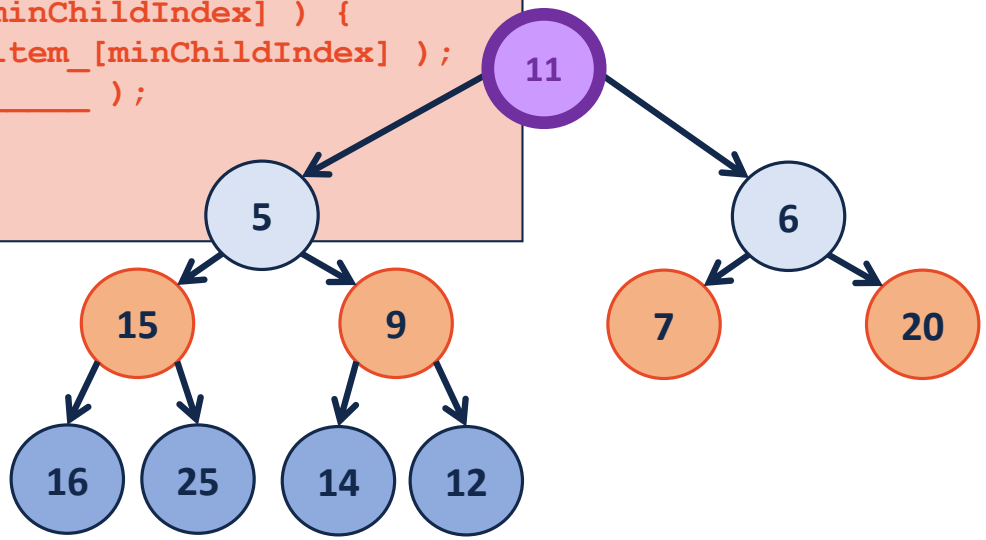
removeMin - heapifyDown

```
 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  }
```

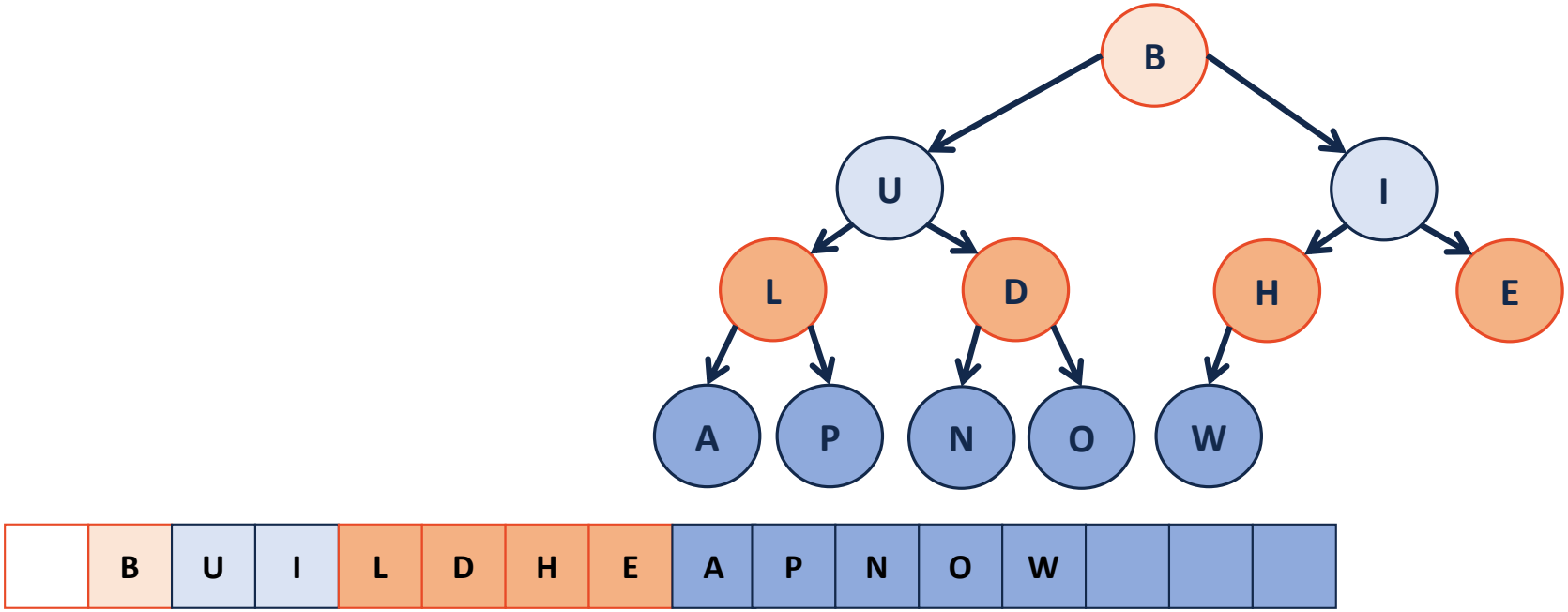
```
 1  template <class T>
 2  void Heap<T>::_heapifyDown(int index) {
 3    if ( !_isLeaf(index) ) {
 4      T minChildIndex = _minChild(index);
 5      if ( item_[index] ___ item_[minChildIndex] ) {
 6         std::swap( item_[index], item_[minChildIndex] );
 7         _heapifyDown( _________________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

removeMin

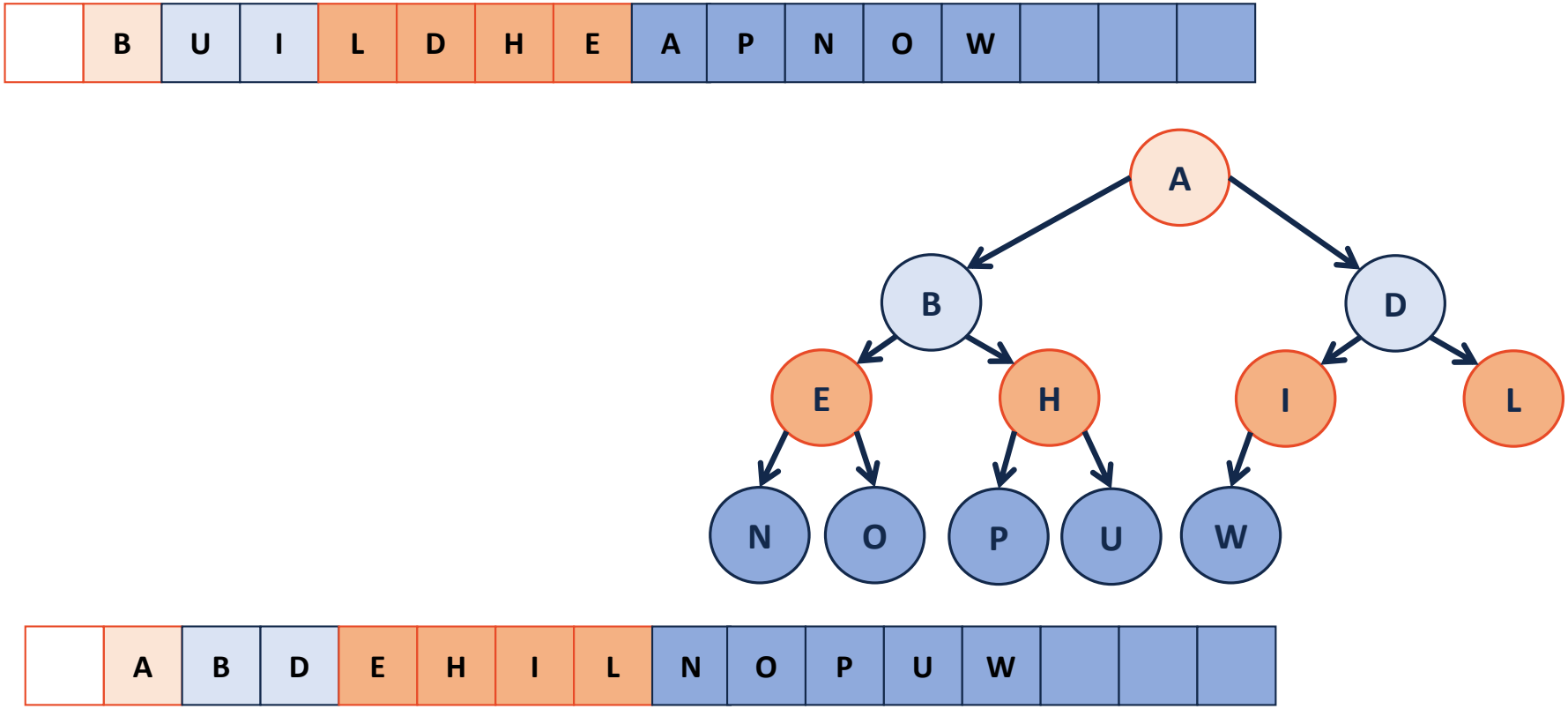
```
1  template <class T>
2  void Heap<T>::_heapifyDown(int index) {
3    if ( !_isLeaf(index) ) {
4      T minChildIndex = _minChild(index);
5      if ( item_[index] ___ item_[minChildIndex] ) {
6         std::swap( item_[index], item_[minChildIndex] );
7         _heapifyDown( ________________ );
8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



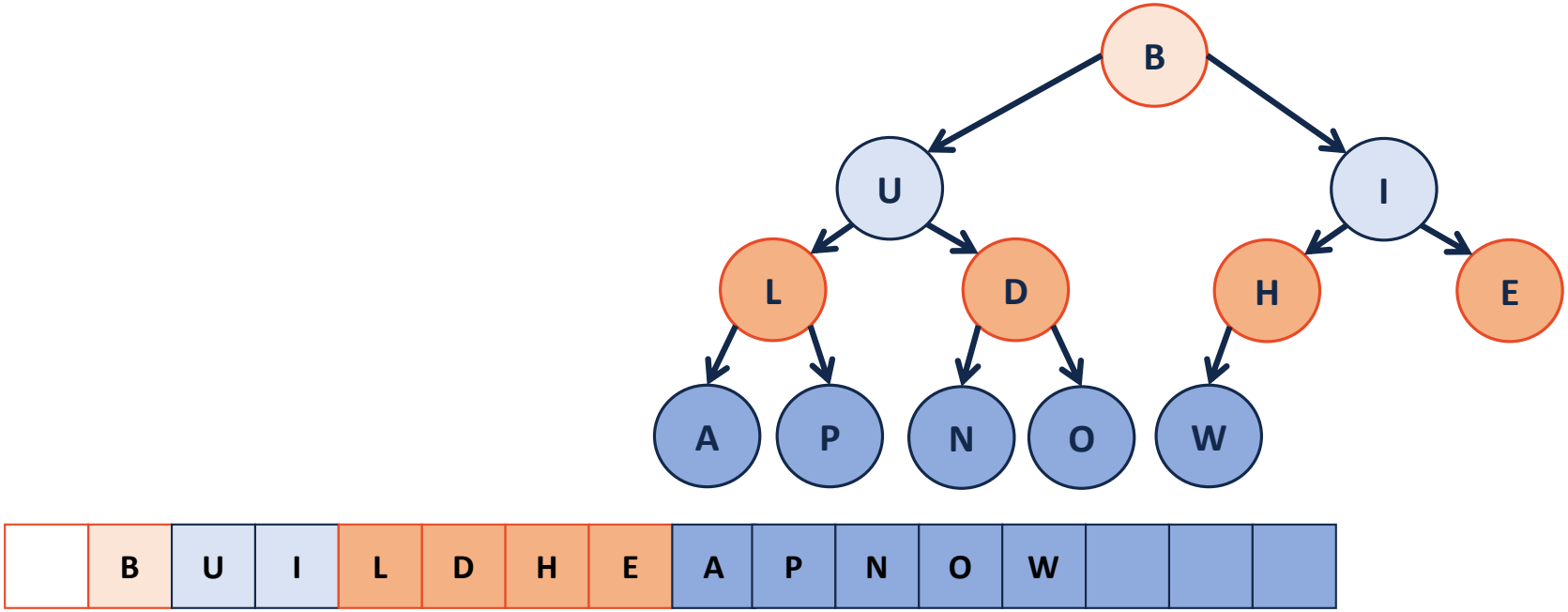
buildHeap



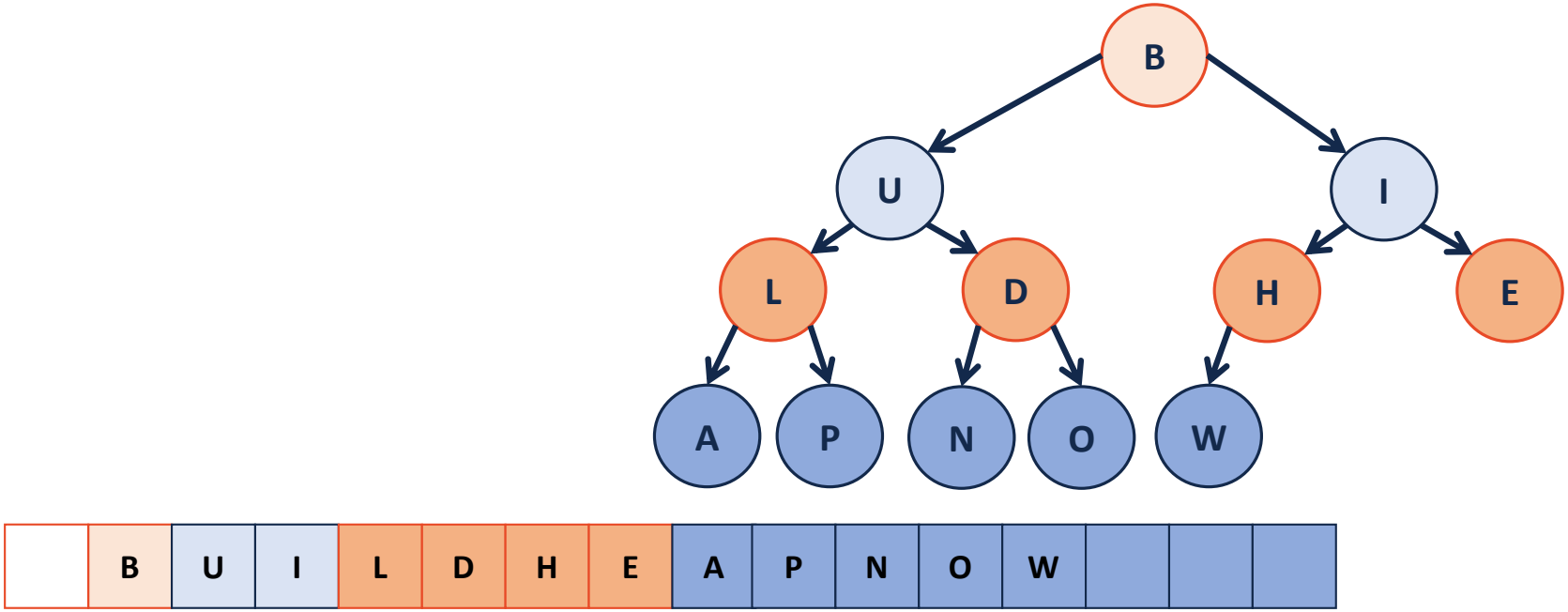
buildHeap – sorted array



buildHeap - heapifyUp

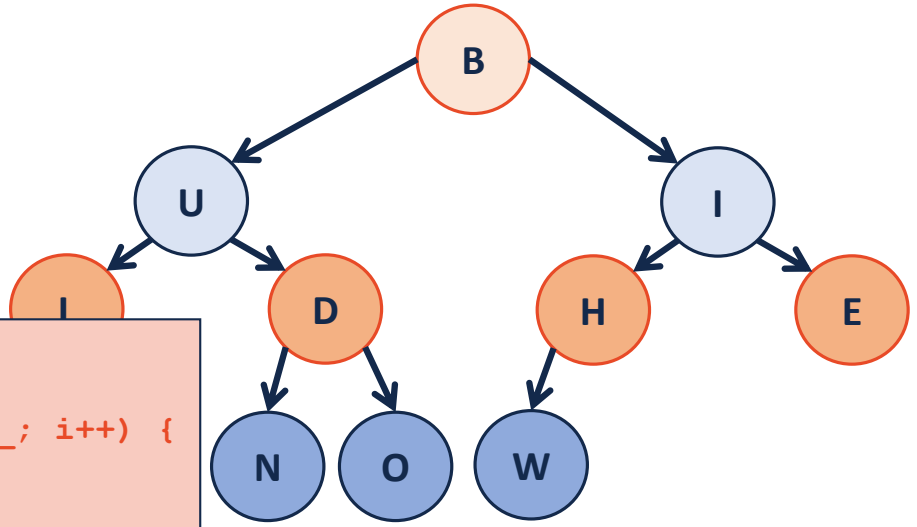


buildHeap - heapifyDown



buildHeap

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  }
```

Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is: _____.

Strategy:

-

-

-

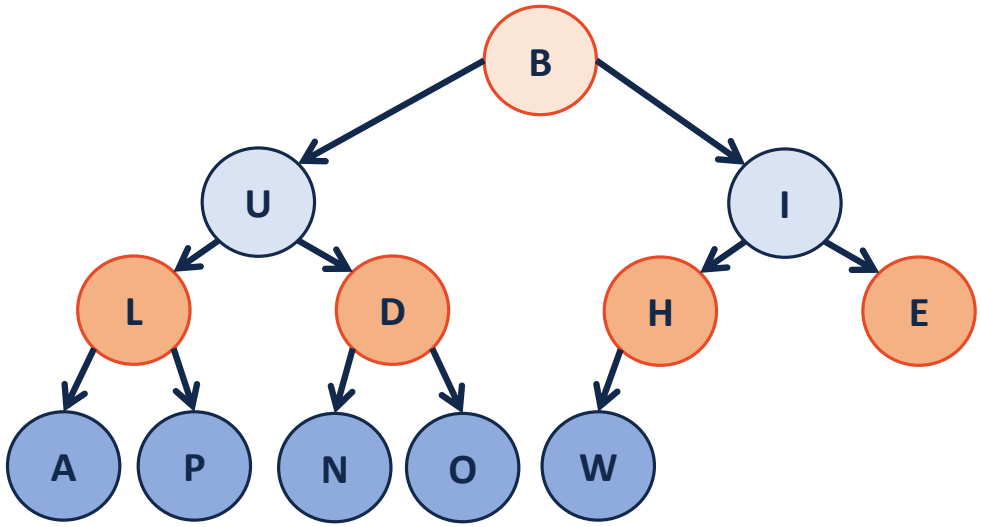
Proving buildHeap Running Time

S(h): Sum of the heights of all nodes in a complete tree of height **h**.

S(0) =

S(1) =

S(h) =



Proving buildHeap Running Time

Proof the recurrence:

Base Case:

General Case:

Proving buildHeap Running Time

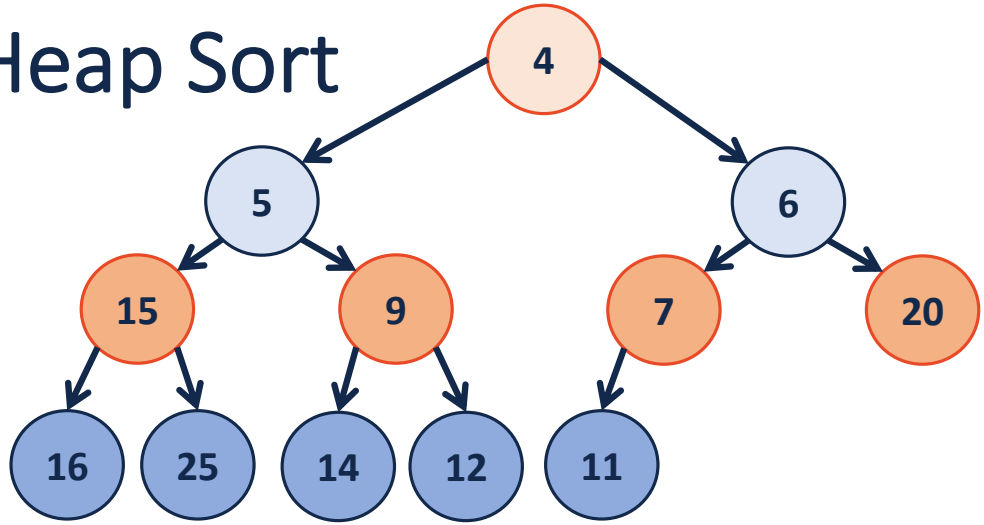
From S(h) to RunningTime(n):

S(h):

Since $h \leq \lg(n)$:

RunningTime(n) ≤

Heap Sort



1.

2.

3.

Running Time?

Why do we care about another sort?