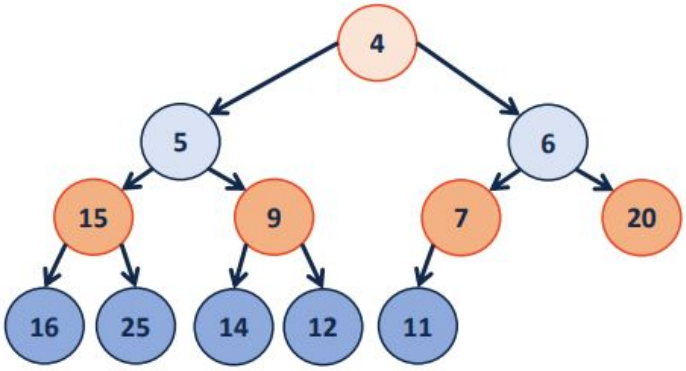
CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

By Wenjie

The minHeap opeartions

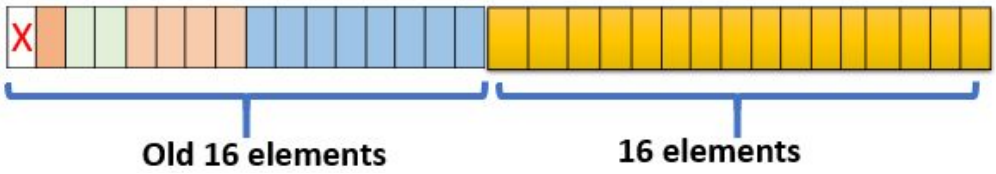
- ADT:
  - `insert()`
  - `remove()`
  - `isEmpty()`



- The visualization is a tree, but the actual implementation will be an array (or vector)
  - Root is at index 1
  - For the node on index i, its
    - Left child is at index $2 * i$
    - Right child is at index $2 * i + 1$
    - Parent is at index $\lfloor i/2 \rfloor$

Insertion

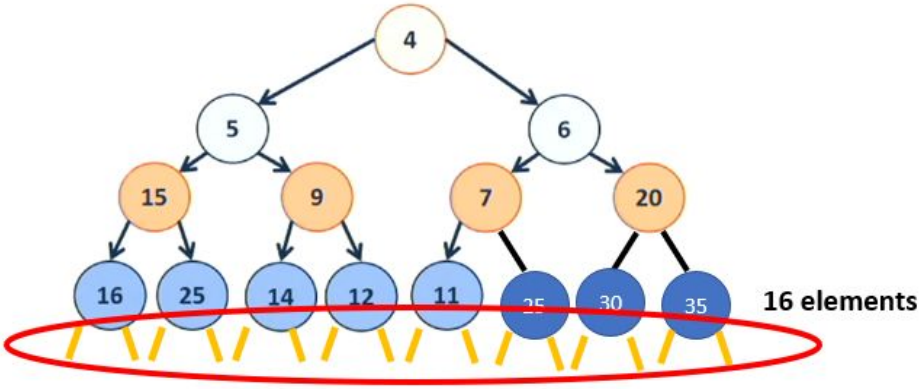
- Check if we still have the array capacity
  - If not, we double the size of the array



  - This is just adding a new layer to the tree

CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

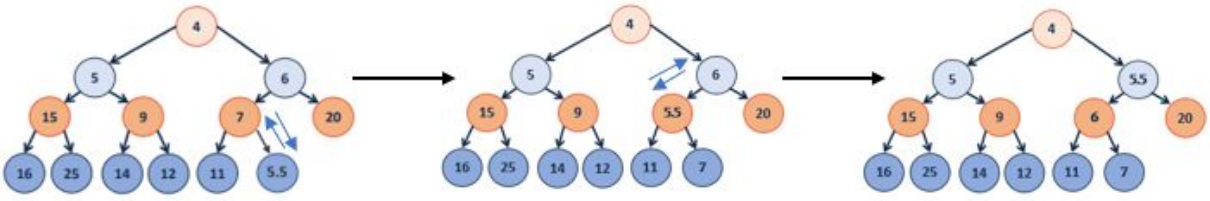
By Wenjie



- Insert the element at the end of the array, and make sure the resulted heap is still a heap (applying heapify-up if needed)

```
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 }
```

Heapify-Up



- Starting from the inserted node, and also assuming the heap is valid everywhere above that inserted node
- If the current element is not the root, and smaller than its parent:
    - Swap the current element with its parent node

CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

By Wenjie

- Continue to applying heapifyUp on the parent node

```
1  template <class T>
2  void Heap<T>::_heapifyUp(unsigned 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  }
```

- Runtime of Insertion operation
  - `growArray()` takes O(1) amortized
  - `insertion` takes O(1)
  - `heapify-up` takes O(h) = O(lg n) since the tree is complete
  - Total runtime: O(lg n)

Remove

- Swap the root with the last element
- Then remove the last element
- Heapify-Down to ensure the heap property is perserved.

```
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 }
```

Heapify-Down

- Starting from the root node with the assumption that both substrees are valid heaps
- If current is not leaf, find the minChild among the two children

CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

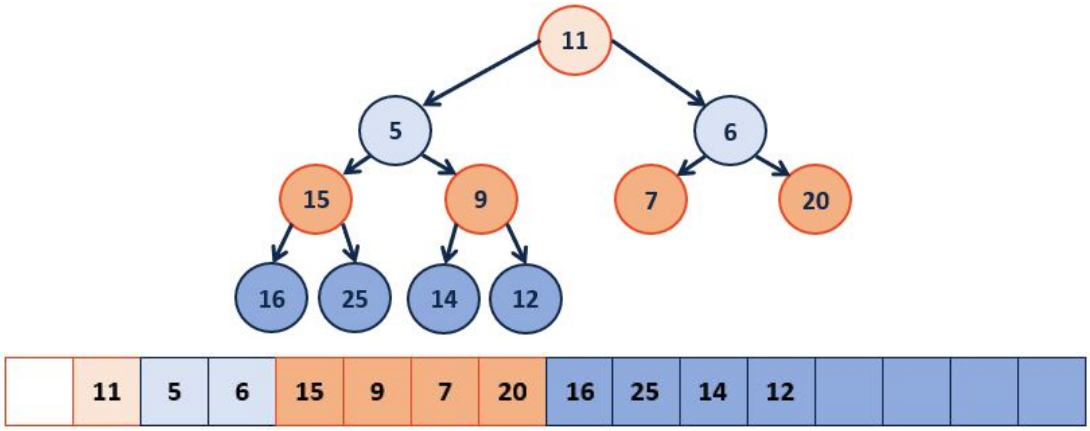
By Wenjie

- Swap the value of minChild and subRoot if needed
- Continue on the minChild node if swap happened

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

- Runtime of `Remove()`
  - swap takes O(1)
  - heapify-down takes O(h) = O(lg n) since the tree is complete
  - **Total runtime: O(log n)**

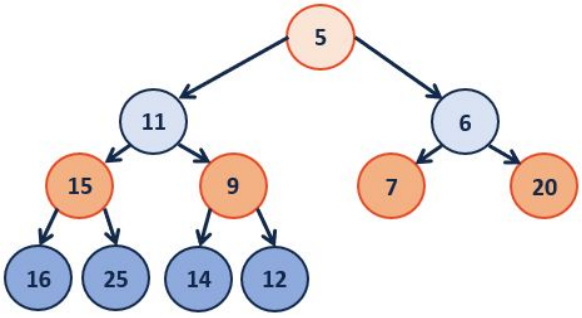
- **Example of HeapifyDown**



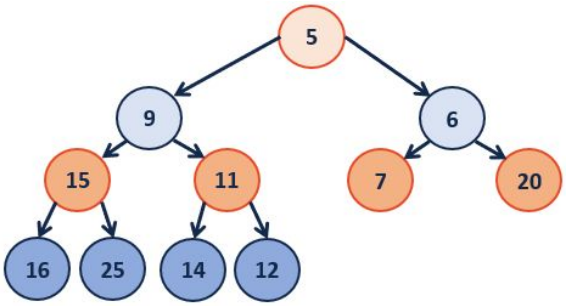
HeapifyDown(1):
minChildIndex = 2;
if(11 > 5) 😊 (true)
swap elements;
heapifyDown(2);

CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

By Wenjie



HeapifyDown(2):
    minChildIndex = 5;
    if(11 > 9) 😊
        swap elements;
        heapifyDown(5);



HeapifyDown(5):
    minChildIndex = 11;
    if(11 > 12) ☹

Done: Heap property restored!

Everything so far can be done using an AVL tree under the same runtime
But the below function, buildHeap, gives Heaps the edge over AVLs

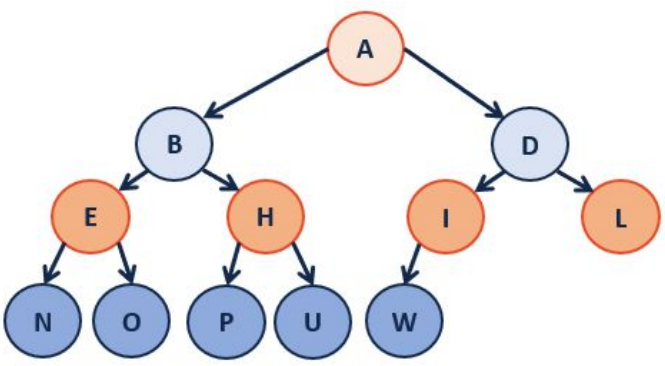
CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

By Wenjie

BuildHeap

- We want to build a heap using a given array:
- **Method 1: sorting**

  - A sorted array is always a heap



  - Runtime: $O(n \log n)$
- **Method 2: heapify-up**
  - Call Heapify-Up on every element from the root

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

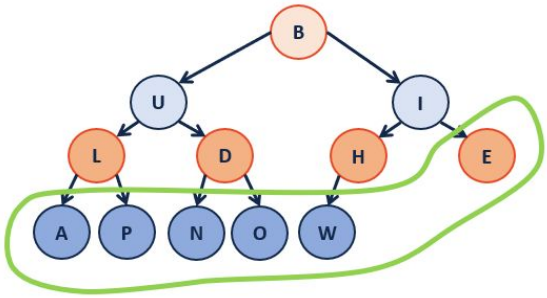
  - Takes $O(\log n)$ for every element, so $O(n \log n)$ in total

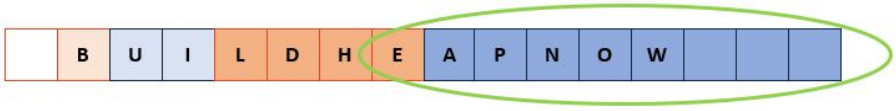
- **Method 3: heapify-down**
  - Call Heapify-Down on every element from the end of the array

CS 225 Spring 2019 :: TA Lecture Notes 3/29 Heaps II

By Wenjie

- Notice that the last level already has the heap property!
- We can start from the second-last level
- In the case below, “H” is the first element that is not a heap






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

- Since heapify-down runs in O(h) time:
  - heapify on “H” takes 1 unit of work
  - heapify on “I” takes 2 units of work
  - heapify on “B” takes 3 units of work
  - In total, we have 1+1+1+2+2 + 3 = 10 units of work, this is just linear to the number of elements

- Then we have the runtime: $O(n)$
  - Proof in next lecture