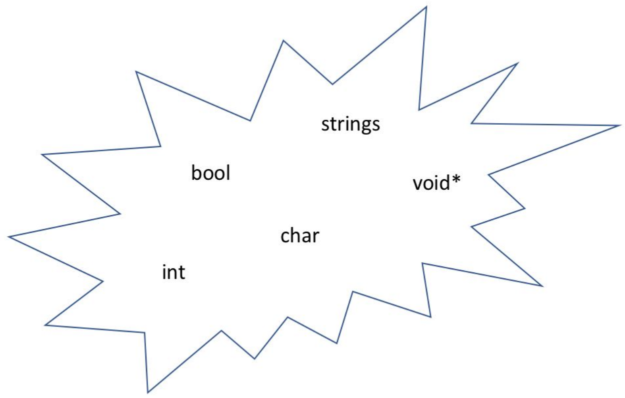
CS 225 Spring 2019 :: TA Lecture Notes
3/27 Heaps

By Wenjie



The Priority Queue/Heap

- ADT:
  - insert
  - remove
  - isEmpty
- Store ordered data
- Operator "<" must be implemented
- Whenever remove is called, the data structure pops out an element with a predetermined property (for example, the smallest element)
  - Just like a Stack/Queue, we cannot tell the structure what it removes
  - Unlike Stack/Queue, the Priority Queue always remove an element with a certain priority (for example, the smallest element)

Implementations

- Possible (bad) implementations of the above ADT and their running times:

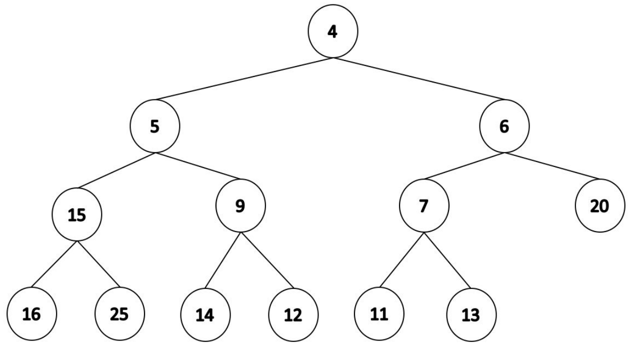
Runtime	insert	removeMin	Total time
Unsorted Array	O(1)*	O(n)	O(n)
Unsorted List	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(1)	O(n)
Sorted List	O(n)	O(1)	O(n)

Further, HashTable is not ordered so not useful. Only thing left is, the **Tree**!

CS 225 Spring 2019 :: TA Lecture Notes
3/27 Heaps

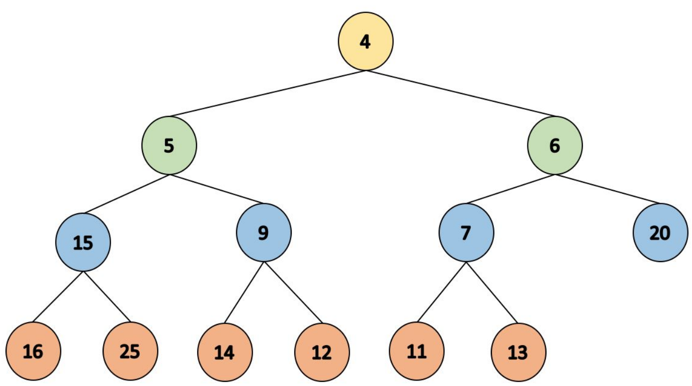
By Wenjie

Tree structure implementation: the (min)Heap



- A binary, complete tree with the smallest element on the root
- Children are larger than their parent
- **Definition of a minHeap:**
  A complete binary tree is a minHeap if
  - **T = {}**, or
  - **T = {r, T_L, T_R}**, where **T_L, T_R** are minHeaps and **r** is greater than their root

- We map the tree into a simpler data structure : **minHeap**.
  - We will map level order tree traversal to an array or vector.
    - We will use trees just for representation.

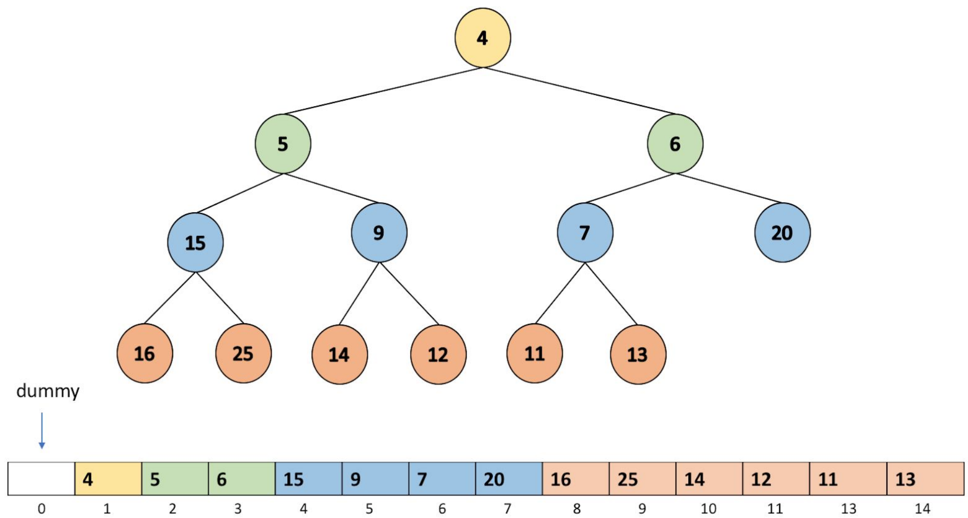


4	5	6	15	9	7	20	16	25	14	12	11	13

- In this case we traverse the array in the following way:

- ■ Left child is at index: 2 * i + 1
- ■ Right child is at index: 2 * i + 2
- ■ Parent is at index: (i - 1) / 2

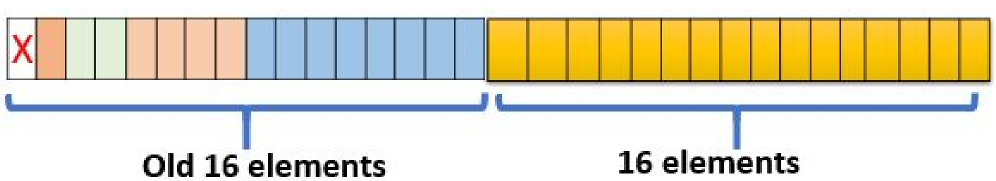
- However, if we want an easier way to compute indices - add a dummy to the beginning of the array to shift the indices by one.



- Now, we can compute indices as follows:
  - ■ Left child is at index: 2 * i
  - ■ Right child is at index: 2 * i + 1
  - ■ Parent is at index: i / 2

Insertion

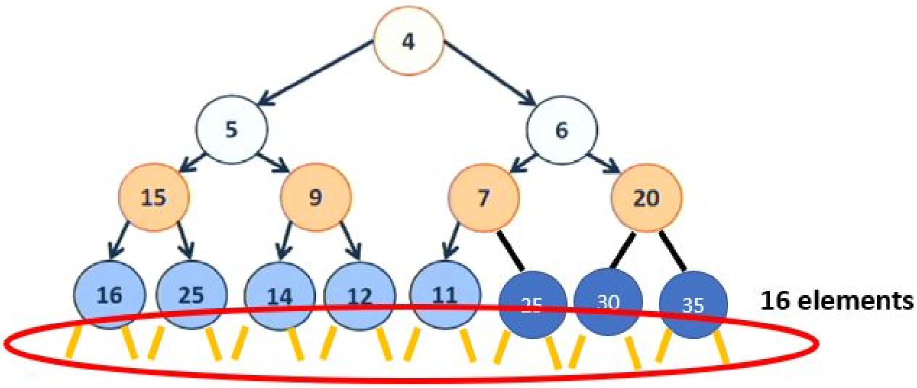
- Check if we still have the array capacity
  - If not, we double the size of the array



  - This is just adding a new layer to the tree

CS 225 Spring 2019 :: TA Lecture Notes
3/27 Heaps

By Wenjie



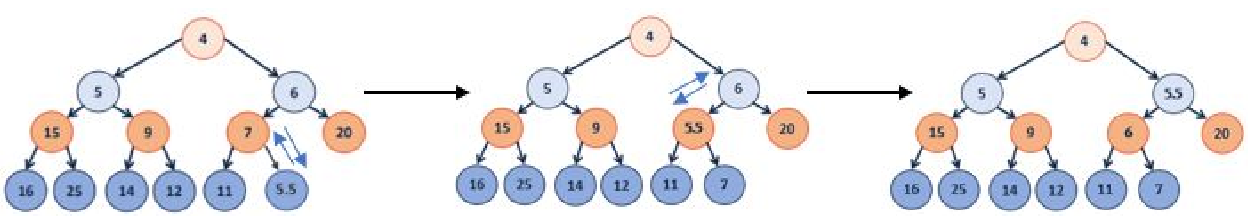
- Insert the element at the end of the array
- Make sure the result is still a heap (heapify-up)

```
template <class T>
void Heap<T>::_insert(const T & key) {
  // Check to ensure there's space to insert an element
  // ...if not, grow the array
  if ( size_ == capacity_ ) { _growArray(); }

  // Insert the new element at the end of the array
  item_[++size] = key;

  // Restore the heap property
  _heapifyUp(size);
}
```

Heapify-Up



- Starts from the inserted node, assumes the heap is valid everywhere above that node
- If the current element is not the root and smaller than its parent

CS 225 Spring 2019 :: TA Lecture Notes
3/27 Heaps

By Wenjie

- Swap the current element and its parent
- Continue on the parent

```
template <class T>
void Heap<T>::_heapifyUp(unsigned index) {
  if ( index > 1 ) {
    if ( item_[index] < item_[ parent(index) ] ) {
       std::swap( item_[index], item_[ parent(index) ] );
       _heapifyUp(parent(index));
    }
  }
}
```

- Runtime of Insertion
  - 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
- Remove the last element
- Heapify-Down to ensure the heap property

```
template <class T>
void Heap<T>::_removeMin() {
  // Swap with the last value
  T minValue = item_[1];
  item_[1] = item_[size_];
  size--;

  // Restore the heap property
  heapifyDown();

  // Return the minimum value
  return minValue;
}
```