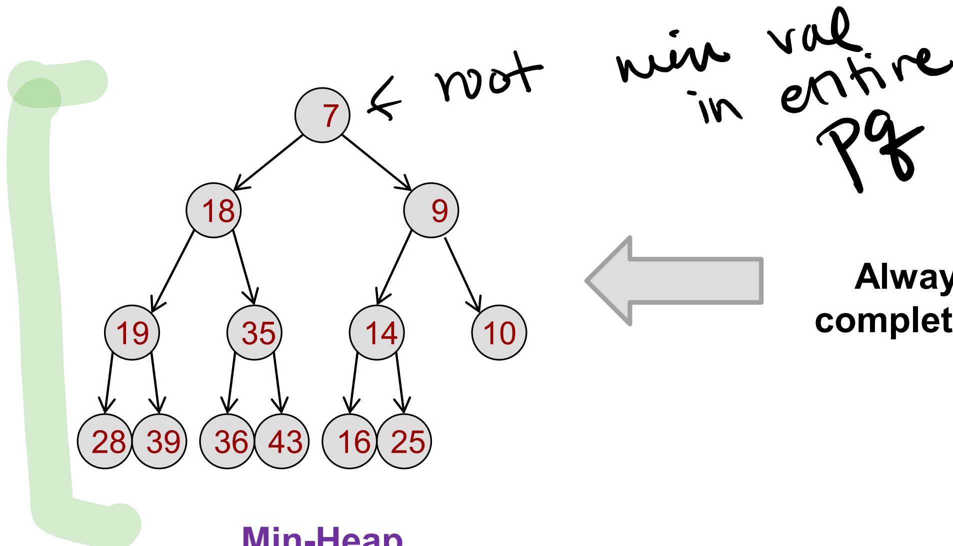
HEAPS

Heap Data Structure

- Provides an efficient implementation for a priority queue
- A ***complete*** binary tree that maintains the **heap property**:
  - **Heap Property**: Every parent is less-than (if min-heap) or greater-than (if max-heap) ***both*** children, but no ordering property between children
- Minimum/Maximum value is always the top element



Always a complete tree

Min-Heap

Heap Operations

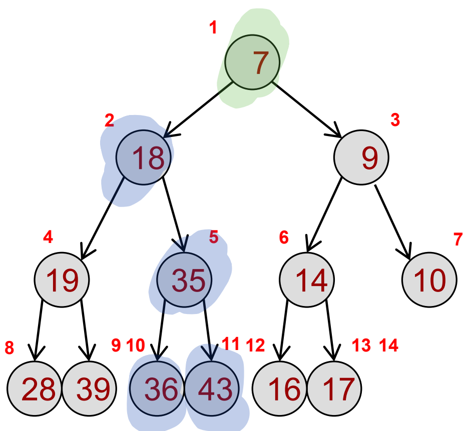
- Push: Add a new item to the heap and modify heap as necessary
- Pop: Remove min/max item and modify heap as necessary
- Top: Returns min/max
- Since heaps are complete binary trees we can use an array/vector as the container

```
template <typename T>
class MinHeap
{ public:
    MinHeap(int init_capacity);
    ~MinHeap()
    void push(const T& item);
    T& top();
    void pop();
    int size() const;
    bool empty() const;
  private:
    // Helper function
    void heapify(int idx);

    vector<T> items_; // or array
}
```

Array/Vector Storage for Heap

- Recall: ~~Full~~ complete binary tree (i.e. only the lowest-level contains empty locations and items added left to right) can be modeled as an array (let's say it starts at index 1) where:
  - Parent(i) = i/2
  - Left_child(p) = 2*p
  - Right_child(p) = 2*p + 1

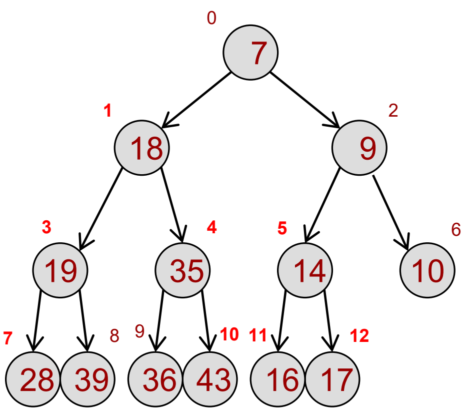


0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

Parent(5) = 5/2 = 2
Left(5) = 2*5 = 10
Right(5) = 2*5+1 = 11

Array/Vector Storage for Heap

- We can also use 0-based indexing
  - Parent(i) = (i-1)/2
  - Left_child(p) = 2*p+1
  - Right_child(p) = 2*p + 2



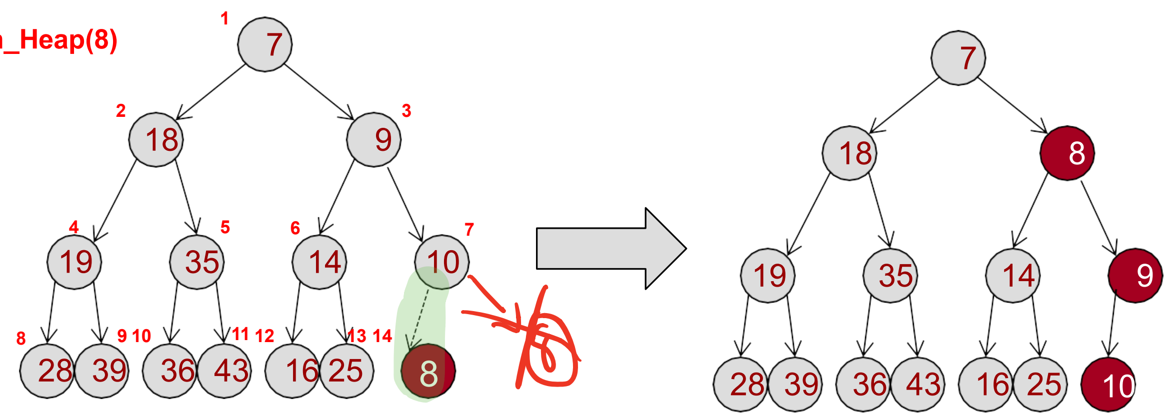
0	1	2	3	4	5	6	7	8	9	10	11	12
7	18	9	19	35	14	10	28	39	36	43	16	17

USC Viterbi School of Engineering

Push Heap/Trickle Up

- Add item to first free location at bottom of tree complete binary tree
- Recursively promote it up while it is less than its parent
  - Remember valid minheap all parents < children…so we need to promote it up until that property is satisfied

Push_Heap(8)



Push Heap

- Add item to first free location at bottom of tree
- Recursively promote it up while it is less than its parent
  - Remember valid minheap all parents < children…so we need to promote it up until that property is satisfied

not exhausted complete binary tree (root is at index 1)

```
void ArrayMinHeap<T>::push(const T& item)
{
  items_.push_back(item);
  trickleUp(items_.size()-1);
}
```

trickleUp(14);

```
void trickleUp(int loc)
{
  // could be implemented recursively
  int parent = loc/2;
  while(parent >= 1 &&
        items_[loc] < items_[parent] )
  {  swap(items_[parent], items_[loc]);
     loc = parent;
     parent = loc/2;
  }
}
```

parent = 7;

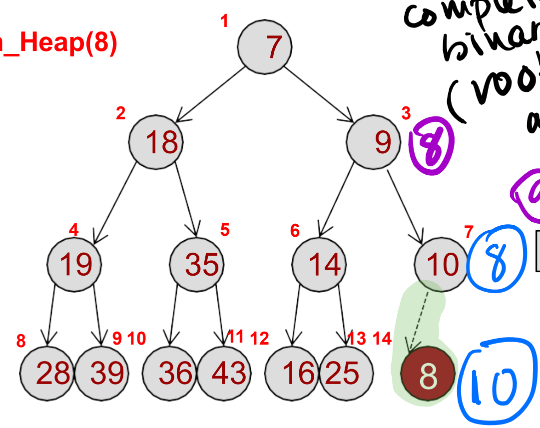
*

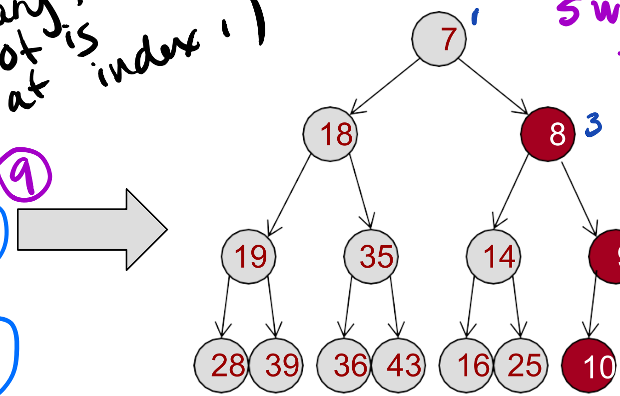
loc = parent; 7; 3;

swap(items_[7], items_[14]);

// parent = 3; 1;

Push_Heap(8)





swap(items_[3], items[7]);

8 > 7 !

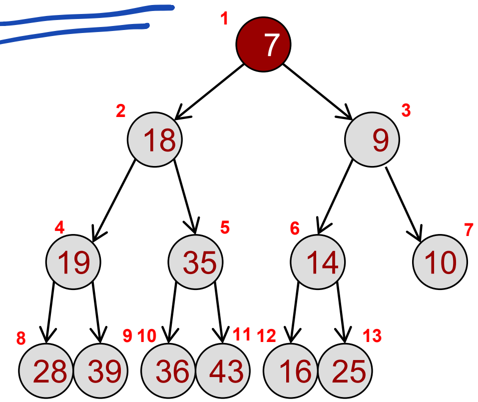
so return restored minheap property

top()

- top() simply needs to return first item

```
T const & MinHeap<T>::top()
{
  if( empty() )
    throw(std::out_of_range());
  return items_[1];
}
```

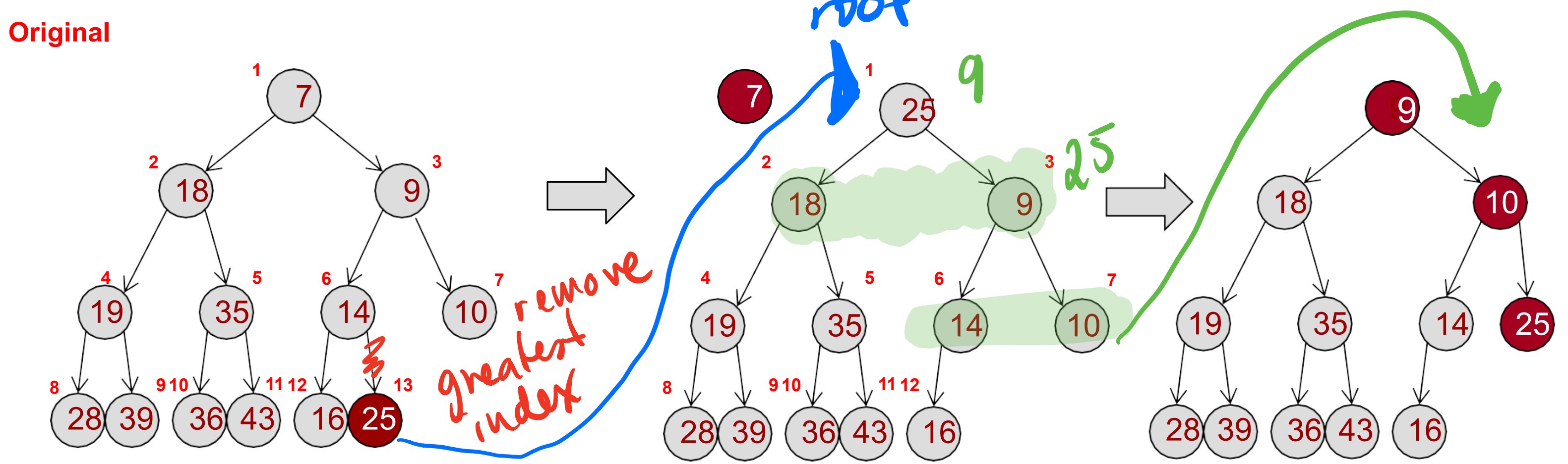
Top() returns 7



Pop Heap/Heapify (TrickleDown)

- Takes last (greatest) node puts it in the top location and then recursively swaps it for the smallest child until it is in its right place

1) make sure we have a complete binary tree
2) overwrite value at greatest index on root
3) restore min heap property



USC Viterbi School of Engineering

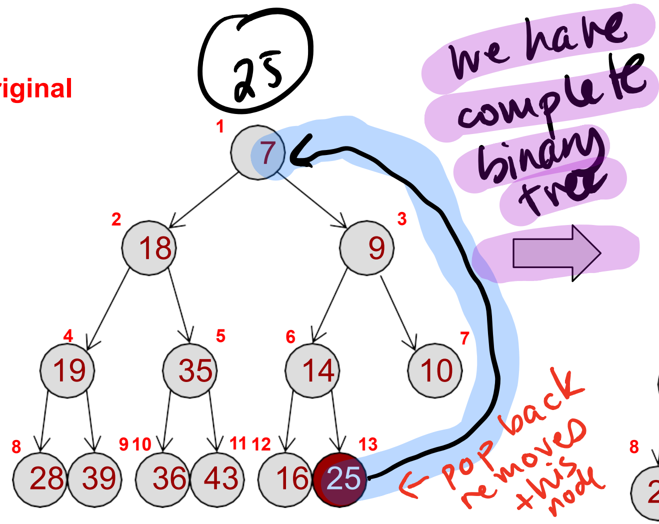
Pop Heap

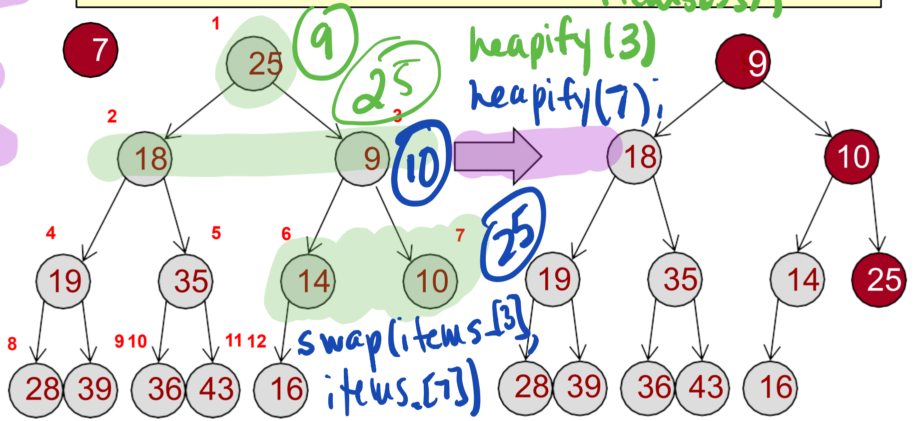
- Takes last (greatest) node puts it in the top location and then recursively swaps it for the smallest child until it is in its right place

```
void ArrayMinHeap<T>::pop()
{ items_[1] = items_.back(); items_.pop_back()
  heapify(1); // a.k.a. trickleDown()
}
```

```
void ArrayMinHeap<T>::heapify(int idx)
{ // multiply idx*2 & compare size
  if(idx == leaf node) return;       2 ; 6
  int smallerChild = 2*idx; // start w/ left
  if(right child exists) {  → 3 ; 7
    int rChild = smallerChild+1;
    if(items_[rChild] < items_[smallerChild])
       smallerChild = rChild;  → 3 ; 7
  } }
  if(items_[idx] > items_[smallerChild]){
       swap(items_[idx], items_[smallerChild]);
       heapify(smallerChild);   swap(items_[1], items_[3]);
} }
```

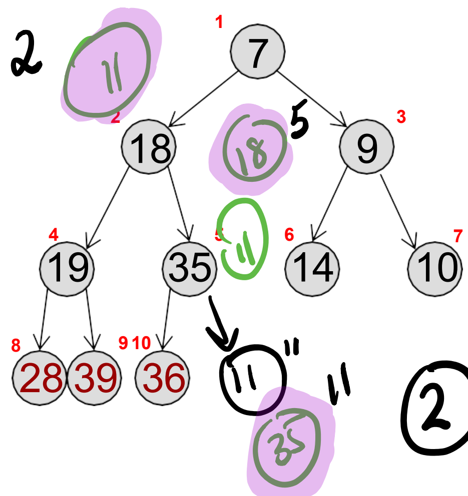
Original





Practice

Push(11)



① add 11 as right most node
→ complete binary tree
min-heap property violated

② items_[11] < items_[5]
11 < 35 → swap

③ items_[5] < items_[2]
11 < 18 → swap

④ items_[2] > items_[1]
11 7
no swap
and stop trickle up

Practice

Items-

0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	~~7~~ 50	18	21	19	35	26	24	28	39	36	43	29	~~50~~

18 50 19 50 28 50

① overwrite items[0] w/ items_[13] remove items_13

② call heapify(1) swap(items_[1], items_-[2])

③ heapify(2) swap(items_[2] items_-[4])

④ heapify(4) swap(items_[4], items[8])

⑤ heapify(8) have leaf node 6 return

complete binary tree

1 50 18

2 19 50

4 28 50

8 50

Pop()