COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Data Structures: Tree

- A binary tree is a an ordered tree where all the nodes have at most two children.
- Each node is either is labeled as either being a left child or a right child.
- A binary tree is proper if each internal node has exactly two children or improper otherwise.
- For any given binary tree T, let:
  - N denote the number of nodes in the T.
  - L denote the number of external nodes (or leaves) in T.
  - I denote the number of internal nodes in T.
  - H denote the height of T. Height of a tree is equal to the height of the root.
- Show that:
  1. $H + 1 \leq N \leq 2^{H+1} - 1$
  2. $1 \leq L \leq 2^{H}$
  3. $H \leq I \leq 2^{H} - 1$
  4. $\log(N + 1) - 1 \leq H \leq N - 1$

- A binary tree is a an ordered tree where all the nodes have at most two children.
- Each node is either is labeled as either being a left child or a right child.
- A binary tree is proper if each internal node has exactly two children or improper otherwise.
- For any given binary tree T, let:
    - N denote the number of nodes in the T.
    - L denote the number of external nodes (or leaves) in T.
    - I denote the number of internal nodes in T.
    - H denote the height of T. Height of a tree is equal to the height of the root.
- Show that:
    1. $H + 1 \leq N \leq 2^{H+1} - 1$
    2. $1 \leq L \leq 2^H$
    3. $H \leq I \leq 2^H - 1$
    4. $\log(N+1) - 1 \leq H \leq N - 1$
    5. The number of edges is equal to $(N - 1)$.

- Here is a high-level java implementation of a Binary Tree.

Code

```
public class Node{
   int value;
   Node leftChild;
   Node rightChild;
   Node parent;
   public Node(){
      leftChild = rightChild = parent = null;
   }
}
public class BinaryTree{
   Node root;
   public BinaryTree(){
      root = null;
   }
   public int computeHeight(Node v){
      //To be written
   }
}
```

- Here is a high-level java implementation of a Binary Tree.

Code

```
public class Node{
   int value;
   Node leftChild;
   Node rightChild;
   Node parent;
   public Node(){
      leftChild = rightChild = parent = null;
   }
}
public class BinaryTree{
   Node root;
   public BinaryTree(){
      root = null;
   }
   public int computeHeight(Node v){
      //To be written
   }
}
   public int computeDepth(Node v){
      //To be written
   }
}
```

- For the context of Binary trees there is another way of tree traversal (other than pre-order and post-order) called in-order traversal.
- Consider the in-order tree traversal method below. This method traverses the tree rooted at node N.

In-order traversal method

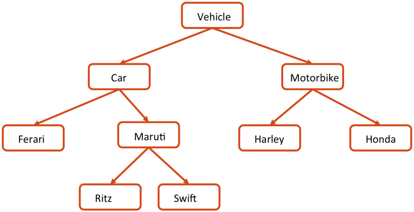
```
public void InOrderTraversal(Node N){
  if(N.leftChild != null)InOrderTraversal(N.leftChild);
  System.out.println(N.value);
  if(N.rightChild != null)InOrderTraversal(N.rightChild);
}
```

- For the context of Binary trees there is another way of tree traversal (other than pre-order and post-order) called in-order traversal.
- Consider the in-order tree traversal method below. This method traverses the tree rooted at node `N`.

In-order traversal method

```
public void InOrderTraversal(Node N){
  if(N.leftChild != null)InOrderTraversal(N.leftChild);
  System.out.println(N.value);
  if(N.rightChild != null)InOrderTraversal(N.rightChild);
}
```

- Question: Produce the output of the above method call when given the root node of the tree below as the input.



Data Structures: Heaps and Priority Queues

- We looked at the Queue ADT which implements a queue that allows adding and removal of elements according to the FIFO principle.
- Such a data structure made sense when automating a queue at a Doctor's office where the FIFO principle is the basic requirement.

- We looked at the Queue ADT which implements a queue that allows adding and removal of elements according to the FIFO principle.
- Such a data structure made sense when automating a queue at a Doctor's office where the FIFO principle is the basic requirement.
- Suppose we want to automate the Prime Minister's office meetings.
  - Each person has a priority (this can be an integer value).
  - We need to add people interested in meeting the PM.
  - The next person to meet the PM should be the person with the highest priority.

- Suppose we want to automate the Prime Minister's office meetings.
  - Each person has a priority (this can be an integer value).
  - We need to add people interested in meeting the PM.
  - The next person to meet the PM should be the person with the highest priority.
- The ADT needed to perform the above task is called a Priority Queue and it supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.

- Priority Queue ADT supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.
- Suppose we implement priority queue using a linked list. What is the running time for each of the following operations:
  - `insert`(k, v):
  - `min()`:
  - `removeMin()`:

- Priority Queue ADT supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.
- Suppose we implement priority queue using a linked list. What is the running time for each of the following operations (the element is added at the head):
  - `insert`(k, v): $O(1)$
  - `min()`: $O(n)$
  - `removeMin()`: $O(n)$

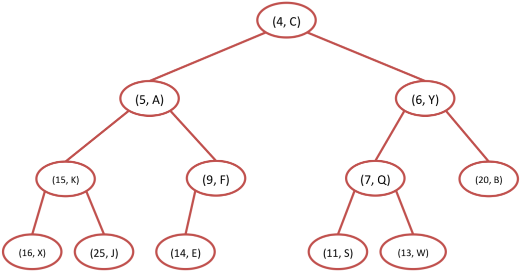
- Priority Queue ADT supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.
- Suppose we implement priority queue using an array. What is the running time for each of the following operations:
  - `insert`(k, v):
  - `min()`:
  - `removeMin()`:

- Priority Queue ADT supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.
- Suppose we implement priority queue using an array. What is the running time for each of the following operations (the new entry is added at the end):
  - `insert`(k, v): $O(1)$
  - `min()`: $O(n)$
  - `removeMin()`: $O(n)$
- How about if we keep the array sorted (in non-increasing order)?

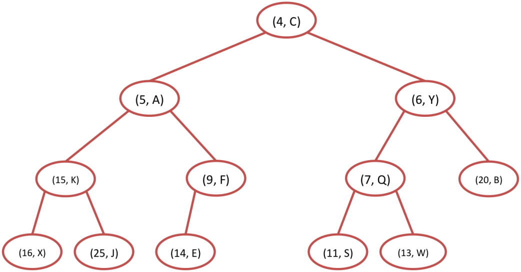
- Priority Queue ADT supports the following operations:
  - `insert`(k, v): Add an entry with key k and value v
  - `min()`: Returns (but does not remove) an entry (k, v) having smallest key; it returns null if there are no entries.
  - `removeMin()`: Removes and returns the entry (k, v) having smallest key; it returns null if there are no entries.
  - `size()`: returns the number of entries.
  - `isEmpty()`: returns true if there are no entries else returns false.
- There is a data structure called heap where the running time of the operations are:
  - `insert`(k, v): $O(\log n)$
  - `min()`: $O(1)$
  - `removeMin()`: $O(\log n)$

- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.

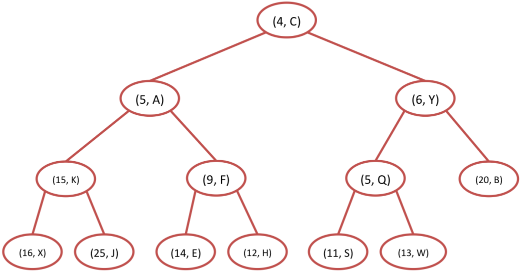
- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap?



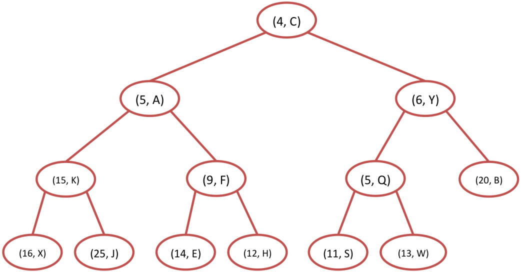
- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap? No



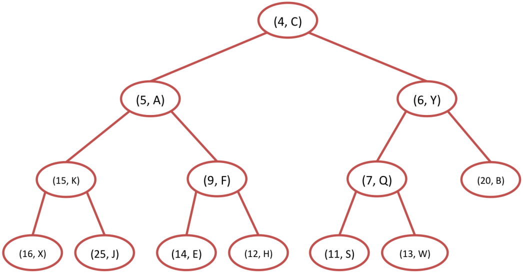
- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap?



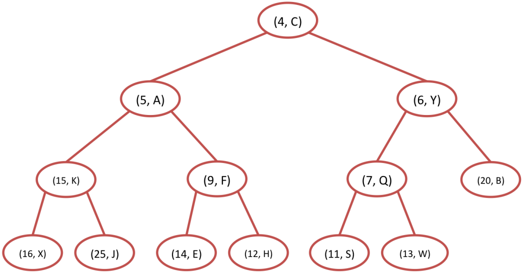
- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap? No



- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap?



- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Is this a min-heap? Yes



- (Minimum) Heap: A (minimum) Heap is a binary tree that stores entries at its nodes and satisfies the following two properties:
  1. Heap-order property: for every node p other than the root node, the key stored at p is greater than equal to the key stored at p's parent.
  2. Complete binary tree property: it is a complete binary tree.
     - Complete binary tree: A binary tree with height h is a complete binary tree iff levels $0, 1, ..., h-1$ have maximum number of nodes possible (i.e., $1, 2, 2^2, ..., 2^{h-1}$) and the remaining nodes at level h reside in the leftmost possible position.
- Question: Show that any heap with n nodes has height $h = \lfloor \log n \rfloor$.

- Consider a pointer based implementation.

Code

```
public class Node{
   int key;
   String value;
   Node leftChild;
   Node rightChild;
   Node parent;
   public Node(){
      leftChild = rightChild = parent = null;
   }
}
public class MinHeap{
   Node root;
   public MinHeap(){
      root = null;
   }
}
```

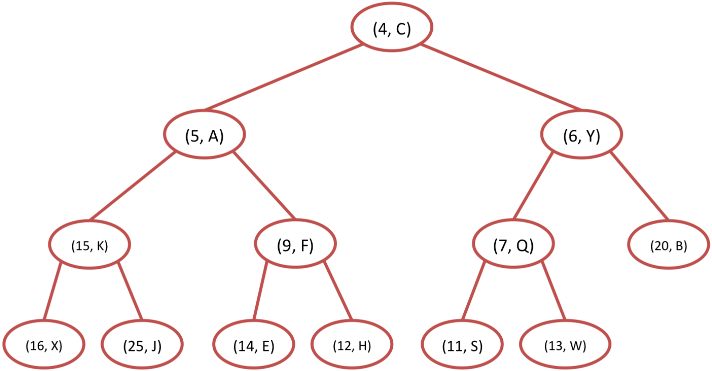
- Consider a pointer based implementation.

Code

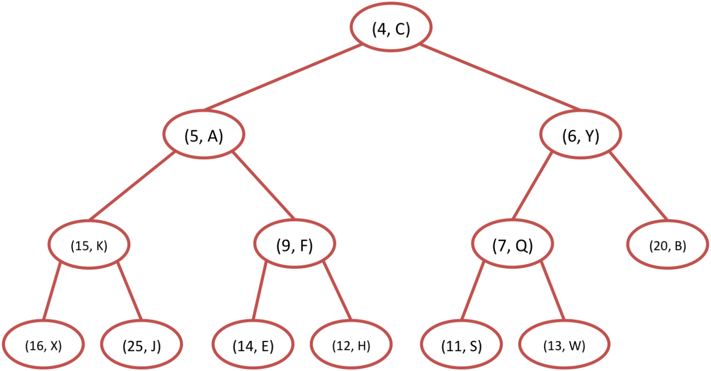
```
public class Node{
   int key;
   String value;
   Node leftChild;
   Node rightChild;
   Node parent;
   public Node(){
      leftChild = rightChild = parent = null;
   }
}
public class MinHeap{
   Node root;
   public MinHeap(){
      root = null;
   }
}
```

- Question: How do we implement `min()`?

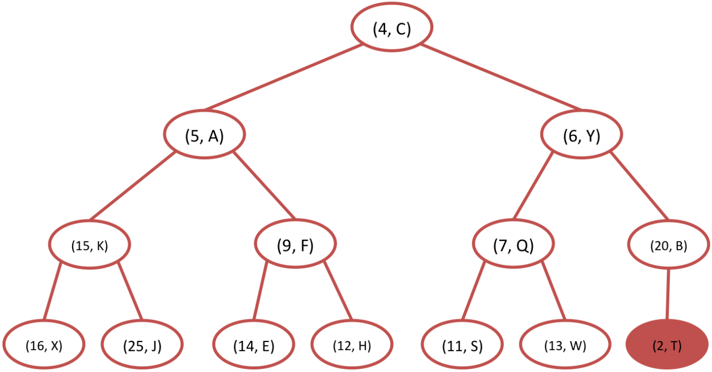
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?



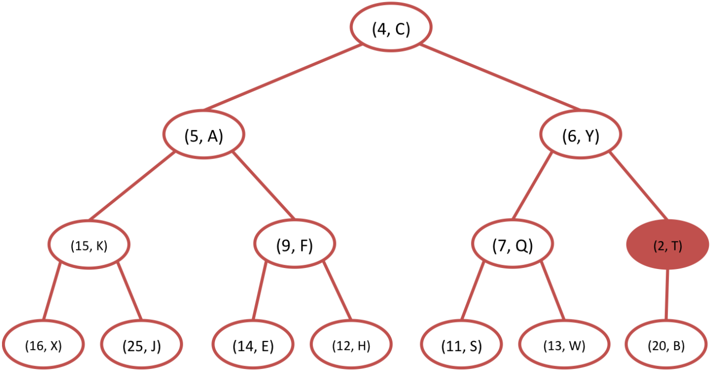
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
  - Suppose we want to insert $(2, T)$ in the heap below.



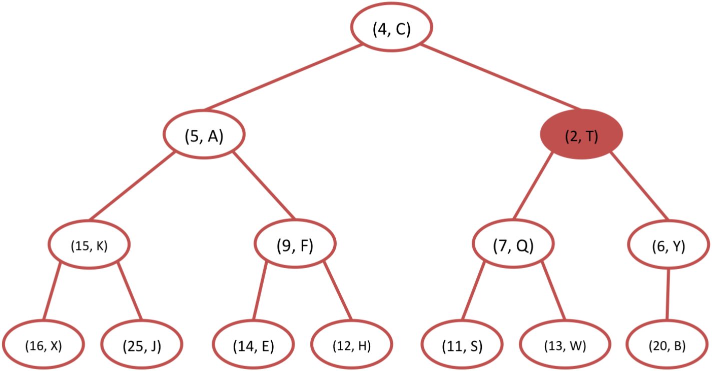
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
  - Consider inserting $(2, T)$ in the heap.



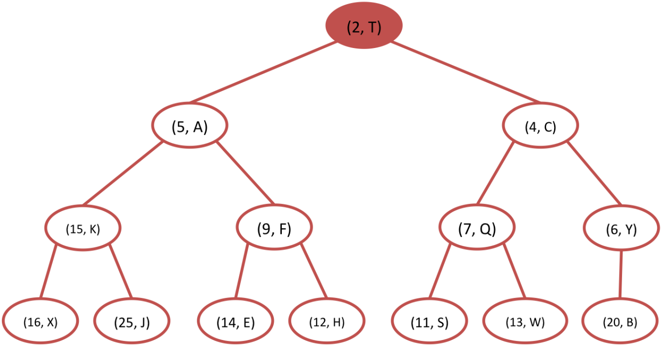
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
  - Consider inserting $(2, T)$ in the heap.



- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
  - Consider inserting $(2, T)$ in the heap.



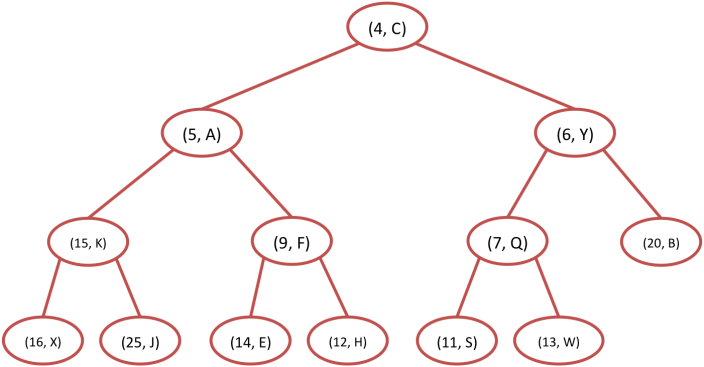
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert(k, v)`?
  - Consider inserting $(2, T)$ in the heap.



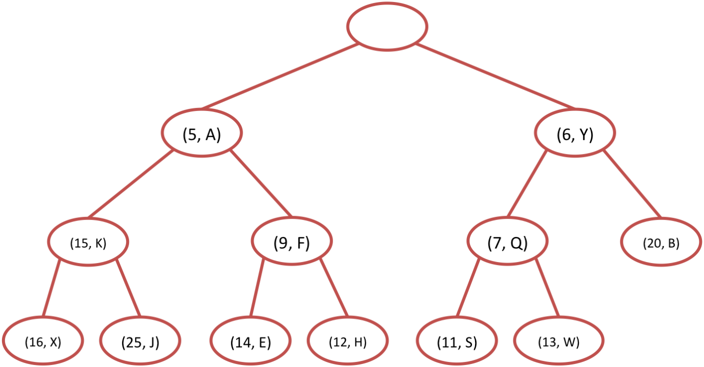
- This process is called up-heap bubbling.

- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
- Question: How do we implement `removeMin()`?

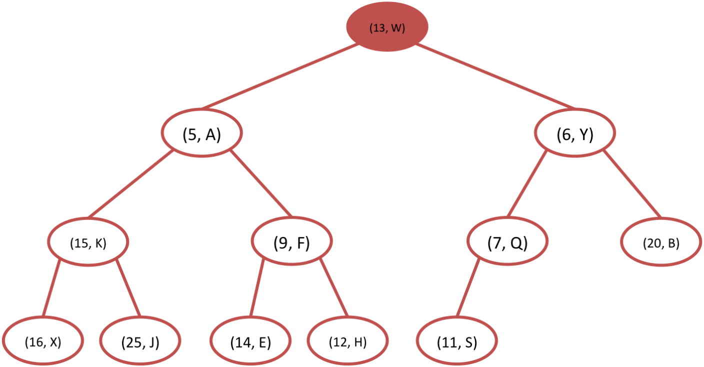
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



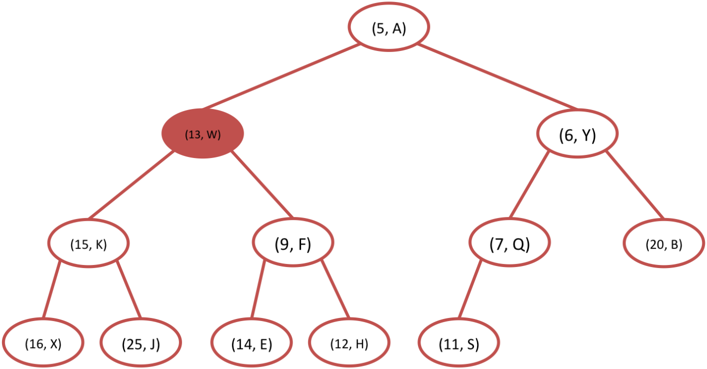
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert(`k, v`)`?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



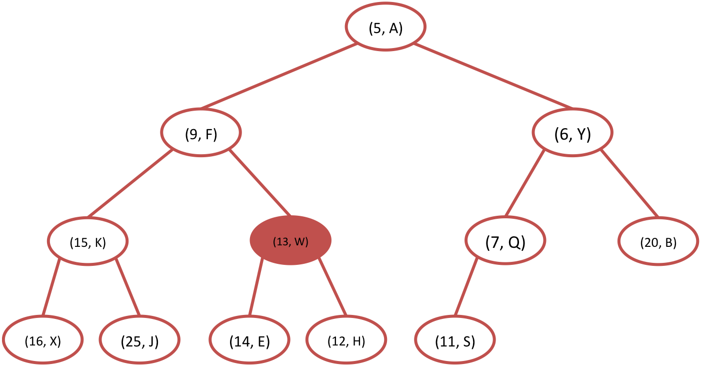
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert(`k, v`)`?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



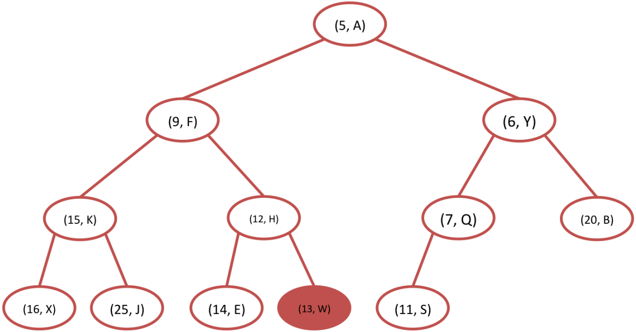
- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert(`k, v`)`?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



- Consider a pointer based implementation.
- Question: How do we implement `min()`?
- Question: How do we implement `insert`(k, v)?
- Question: How do we implement `removeMin()`?
  - Consider removing the entry with the minimum key from the heap.



- This process is called down-heap bubbling.

- Let us write the methods for the pointer based implementation.

Code

```
public class Node{
    int key;
    String value;
    Node leftChild;
    Node rightChild;
    Node parent;
    public Node(){
        leftChild = rightChild = parent = null;
    }
}
public class MinHeap{
    Node root;
    Node lastNode;
    public MinHeap(){
        root = lastNode = null;
    }
}
public String min(){//To be written}
public void insert(int k, String v){//To be written}
public String removeMin(){//To be written}
```

- Let us write the methods for the pointer based implementation.
- What is the running time of each operation:
    - `min()`:
    - `insert(k, v)`:
    - `removeMin()`:

- Let us write the methods for the pointer based implementation.
- What is the running time of each operation:
    - `min()`: $O(1)$
    - `insert(k, v)`: $O(\log n)$
    - `removeMin()`: $O(\log n)$

End