Lecture 11.2 : Priority queue

Introduction

- We introduce the priority queue data type. We will examine some elementary implementations and their limitations before studying an efficient implementation based on the binary heap data structure.
- We looked at the **stack** data type. We can add items to the stack via the `push` operation. We can remove the item most recently added via the `pop` operation.
- We looked the **queue** data type. We can add items to the queue via the `enqueue` operation. We can remove the item least recently added via the `dequeue` operation.
- We now look at the **priority queue** data type. We can add items to the priority queue. We can remove the largest (or smallest) item from the priority queue.

Priority queue methods

- An instance `PQ` of the priority queue abstract data type supports the following methods:
    - `PQ.insert(e)`: Add element `e` to the priority queue `PQ`.
    - `PQ.delMax()`: Remove and return the largest element in the priority queue `PQ`; an error occurs if the priority queue `PQ` is currently empty.
    - `PQ.getMax()`: Return a reference to the largest element in the priority queue `PQ` without removing it; an error occurs if the priority queue `PQ` is currently empty.
    - `PQ.is_empty()`: Return `True` if priority queue `PQ` is empty and `False` otherwise.
    - `PQ.size()`: Return the number of elements in the priority queue `PQ`.

Desired behaviour

- Let’s look at a priority queue in action:

```
from priority_queue import PQ

def main():

    pq = PQ()
    pq.insert('P')
    pq.insert('Q')
    pq.insert('E')
    print(pq.delMax())
    pq.insert('X')
    pq.insert('A')
    pq.insert('M')
    print(pq.delMax())
    print(pq.delMax())
    pq.insert('P')
    pq.insert('L')
    pq.insert('E')
    print(pq.delMax())

if __name__ == '__main__':
    main()
```

```
$ python3 pq_demo.py
Q
X
P
P
```

Elementary implementations

- We could implement the priority queue as an *unordered list*:

```
class UnorderedMaxPQ(object):

    def __init__(self):
        self.l = []

    def insert(self, e):
        self.l.append(e)

    def delMax(self):
        max_e = max(self.l)
        self.l.remove(max_e)
        return max_e
```

```
from unordered import UnorderedMaxPQ

def main():

    upq = UnorderedMaxPQ()
    upq.insert('P')
    upq.insert('Q')
    upq.insert('E')
    print(upq.delMax())
    upq.insert('X')
    upq.insert('A')
    upq.insert('M')
    print(upq.delMax())
    print(upq.delMax())
    upq.insert('P')
    upq.insert('L')
    upq.insert('E')
    print(upq.delMax())

if __name__ == '__main__':
    main()
```

```
$ python3 unordered_demo.py
Q
X
P
P
```

- We could implement the priority queue as an *ordered list*:

```
class OrderedMaxPQ(object):

    def __init__(self):
        self.l = []
    def insert(self, e):
        self.l.append(e)
        self.l.sort()

    def delMax(self):
        return self.l.pop()
```

```
from ordered import OrderedMaxPQ

def main():

    opq = OrderedMaxPQ()
    opq.insert('P')
    opq.insert('Q')
    opq.insert('E')
    print(opq.delMax())
    opq.insert('X')
    opq.insert('A')
    opq.insert('M')
    print(opq.delMax())
    print(opq.delMax())
    opq.insert('P')
    opq.insert('L')
    opq.insert('E')
    print(opq.delMax())

if __name__ == '__main__':
    main()
```

```
$ python3 ordered_demo.py
Q
X
P
P
```

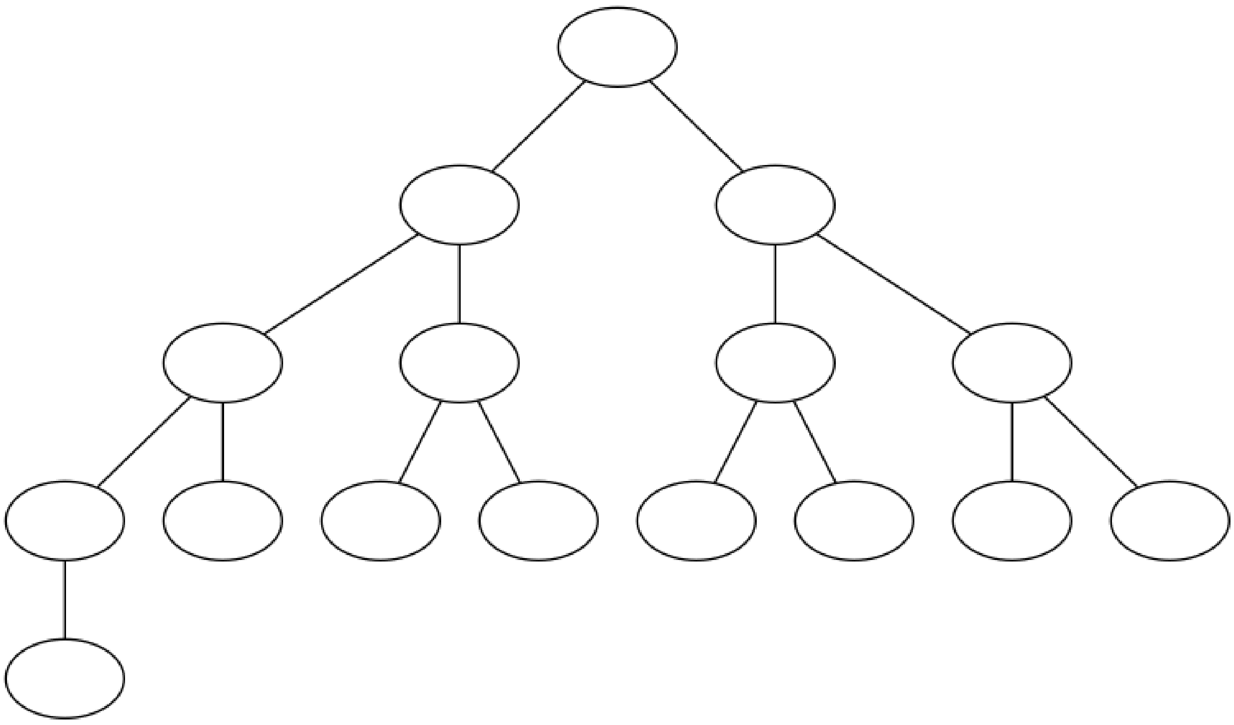
Assessing elementary implementations

- We can see from above that while the unordered list implementation of the priority queue efficiently handles insertion of new items it is inefficient when it comes to removal of the maximum element in the priority queue.
- We can also see from above that while the ordered list implementation of the priority queue efficiently handles removal of the maximum element it is inefficient when it comes to the insertion of new items in the priority queue.
- What we would like is a priority queue implementation that *efficiently* handles both insertion of new elements into the queue *and* the removal of the maximum element from the priority queue.

Binary tree

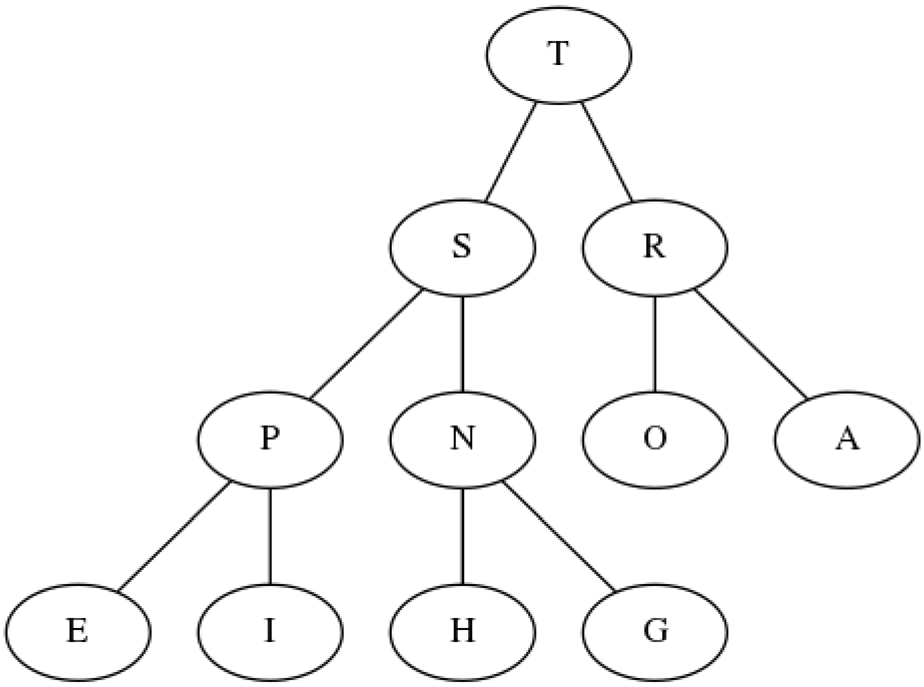
- A binary tree is empty or consists of a node with links to left and right binary trees.

- Here is an example binary tree with 16 nodes:

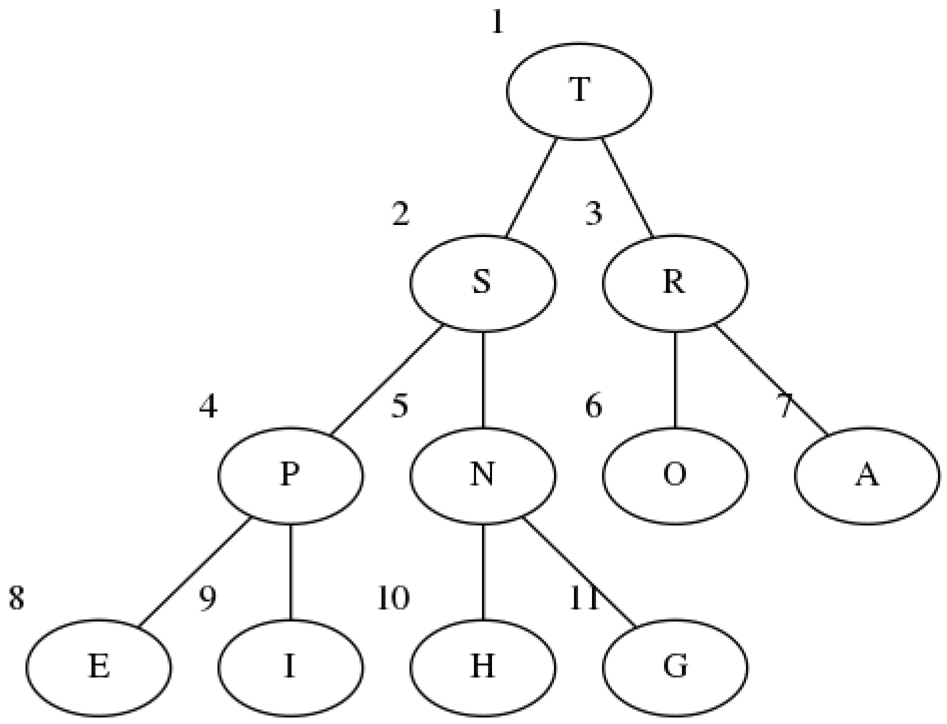


Heap-ordered binary tree

- In a maximum heap-ordered binary tree we store a value inside each node and *a parent node is no smaller than a child node*.
- Here is an example of a maximum heap-ordered binary tree:



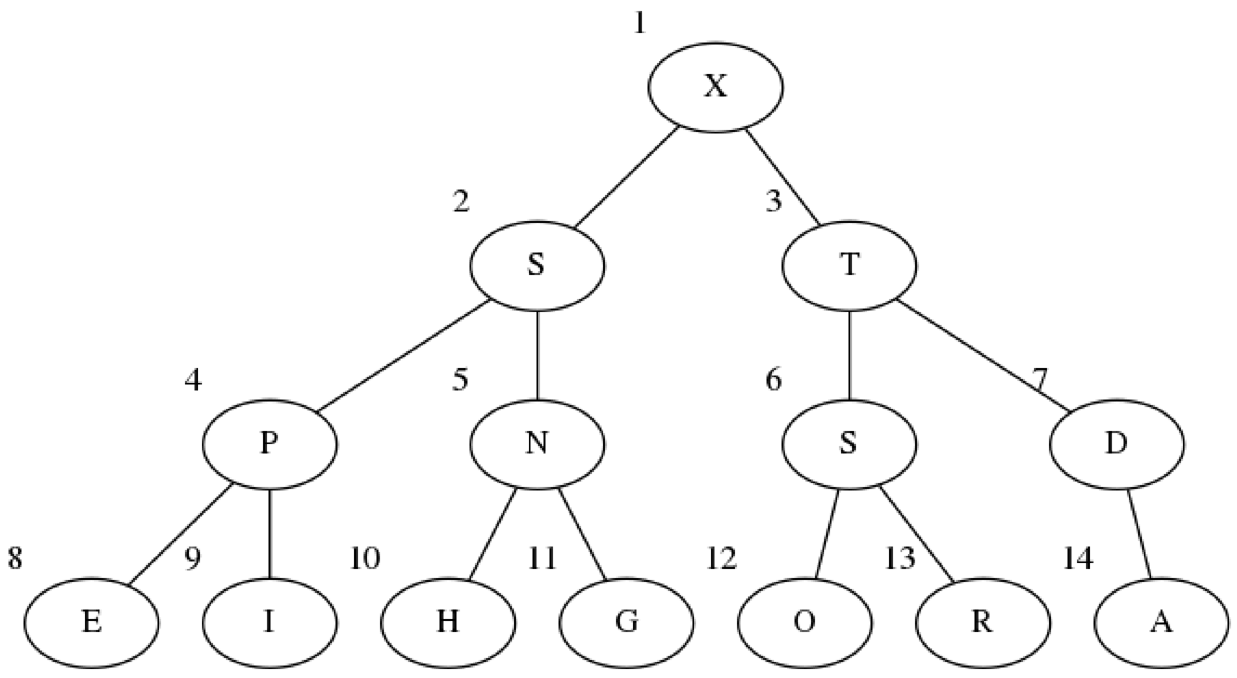
- How are we going to represent this structure in Python? Well, numbering each node might give us some clues:



- Note we start numbering nodes at 1 (rather than the traditional 0) and number nodes in level order.
- Given the above numbering scheme how can we find the children of node N? Simple. The children of node N are node $2 \cdot N$ and node $2 \cdot N + 1$.
- Given the above numbering scheme how can we find the parent of node N? Simple. The parent of node N is node $N//2$.
- For example the children of node 5 are nodes 10 $(2 \cdot 5)$ and 11 $(2 \cdot 5 + 1)$
- For example the parent of node 6 is node 3 $(6//3)$ and the parent of node 7 is node 3 $(7//3)$.
- Thus to locate a parent from a child node or to locate child nodes from a parent node **no explicit links are required**.

Insertion into the heap

- To insert a new node into a heap of size N we add it at location $N + 1$ and then allow it to *swim* up the heap until it finds its correct location.
- The swim procedure is straightforward: *For as long as the parent node is less than the child node, swap the parent and child nodes*.
- In other words we keep swapping the new node with its parent until the heap condition is satisfied i.e. no node should have a value smaller than its children.
- What will our heap look like after inserting *X*, *S* and *D* in that order?



Implementing insertion

- We will implement the heap using a dictionary where the dictionary keys are indices and the dictionary values are the node values in the binary tree. The dictionary is initially empty and the number of nodes is initially zero. Thus our class begins:

```
class PQ(object):

    def __init__(self):
        self.d = {}
        self.N = 0
```

- We will find it handy to have a method that exchanges the contents of two nodes. Let's add it:

```
    # Swap nodes i and j
    def exch(self, i, j):
        self.d[i], self.d[j] = self.d[j], self.d[i]
```

- Let's write the `insert()` method:

```
    # Add a new node to the heap
    def insert(self, v):
        self.N += 1
        self.d[self.N] = v
        self.swim(self.N)
```

- Now let's write the `swim()` method:

```
    # Node k swims up the heap
    def swim(self, k):
        # while not at root and parent < child
        while k > 1 and self.d[k//2] < self.d[k]:
            self.exch(k, k//2)
            k = k//2
```

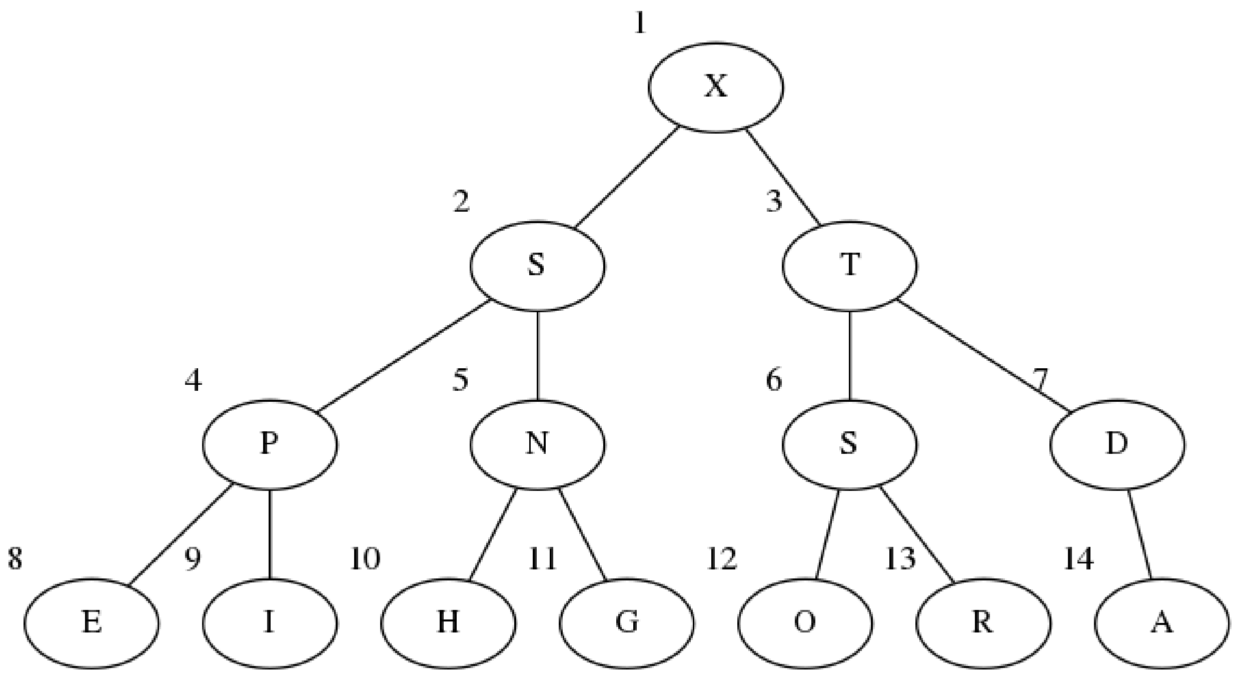
- We can look inside the heap to see how the dictionary is structured:

```
from priority_queue import PQ

def main():

    pq = PQ()
    pq.insert(5)
    pq.insert(6)
    pq.insert(12)
    pq.insert(3)
    pq.insert(15)
    pq.insert(9)
    print(pq.d)

if __name__ == '__main__':
    main()
```

```
$ python3 dict_demo.py
{1: 15, 2: 12, 3: 9, 4: 3, 5: 5, 6: 6}
```

Deletion from the heap

- The `delMax()` method must remove and return the maximum element in the priority queue. In our implementation the maximum element is the root node of the heap.
- However after calling `delMax()` we must ensure that the heap property still holds i.e. *a parent node is no smaller than a child node.*
- To implement `delMax()` we swap node 1 with node N and then allow the new root node to *sink* down the heap until it finds its correct location. (We also decrement N and delete the old root node.)
- The sink operation is straightforward: *For as long as the parent is smaller than one of its children swap it with the larger of its children*. Why swap with the larger child?
- What will this heap look like after we invoke `delMax()`:



Implementing deletion

- We will find it handy to have a method that returns the larger of a node's two children:

```
    # Return the bigger of nodes i and j
    def bigger(self, i, j):
        try:
            return max([i, j], key=self.d.__getitem__)
        except KeyError:
            return i
```

- Let's write the `delMax()` method:

```
    # Remove the max node
    def delMax(self):
        v = self.d[1]
        self.exch(1, self.N)
        del(self.d[self.N])
        self.N -= 1
        self.sink(1)
        return v
```

- Lastly let's write the `sink()` method:

```
    # Node k sinks down the heap
    def sink(self, k):
        # While there is a left child
        while (2*k <= self.N):
            # Index of left child
            j = 2*k
            # Select bigger child
            j = self.bigger(j, j+1)
            # Done if >= both children
            if self.d[k] >= self.d[j]:
                break
            # Swap with larger child
            self.exch(k, j)
            k = j
```

Finishing off the implementation

- Lastly we need to implement the `getMax()`, `is_empty()` and `size()` methods:

```
    # Return reference to max node
    def getMax(self):
        return self.d[1]

    # Is the heap empty
    def is_empty(self):
        return self.size() == 0

    # Size of heap
    def size(self):
        return self.N
```