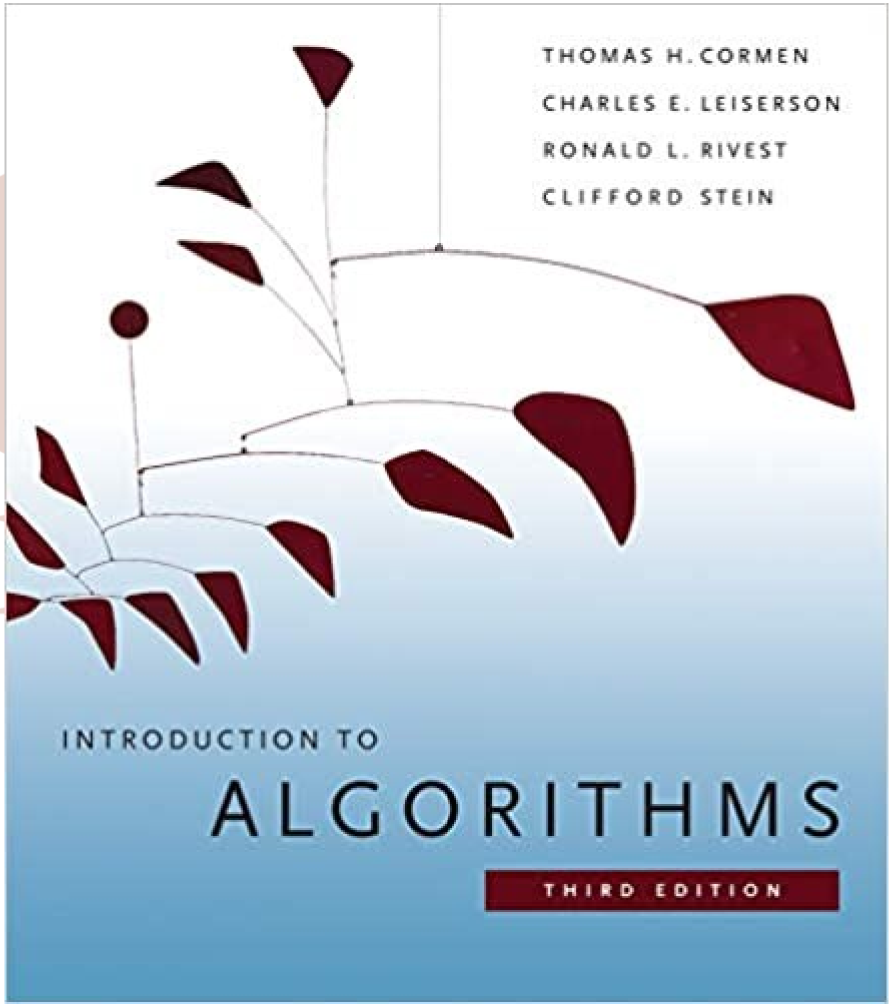
CS 07540 Advanced Design and Analysis of Algorithms

Week 10

- ADT Priority Queue
  - Data Structure as Linked List
  - Data Structure as Binary Heap
  - Data Structure as Binomial Heap (derived from ADT Mergeable Heap)
  - Data Structure as Fibonacci Heap (next class, also derived from ADT Mergeable Heap)



Priority Queue

A priority queue is an ADT which has methods for inserting an item and removing the item with the smallest (or largest) key (priority). The methods for a class **`priorityQueue`** can be just wrappers for an underlying **`Heap`** class, which we already looked at. A priority queue can be implemented in several ways – a heap being one – but a heap is a data structure in its own right. Implementations we will look at include

- Linked Lists
- Binary Heap
- Binomial Heap
- Fibonacci Heap

Let’s start at the beginning…

ADT Queue

Stacks are Last In First Out collections. A queue is a First In First Out collection, a FIFO stack. The first item in is also the first item out. The push and pop operations of a stack could act on different sides of a stack to create a queue. The word ***queue*** is British for line. It is Latin in origin, made it into French in the 16th century, was adopted by the British, and is now immortalized in Computer Science, a mainly American invention.

Queues are appropriate for many real-world situations, for example a request to print a document (printer queue: requests can be entered faster than jobs can be completed). ***Queues*** are a special case of ***Deques*** (double-ended queues), so in Python they can be implemented as *deque* which was designed to have fast appends and pops from both ends. They do support the standard list interface, and we will look at implementation and use, and analyze performance.

ADT Queue Interface

We are going to use the Python Documentation sample implementation for using lists as queues. The interface for queues should include

`createQueue()`	Create an empty(!) queue
`isEmpty()`	Determine whether queue is empty
`enqueue(item)`	Insert a new item to the queue
`dequeue()`	Delete from the queue the item that was added first
`dequeueAll()`	Remove all the items from the queue (clear)
`peek()`	Retrieve from the queue the item that was added first among the still existing items (without removing)

ADT Queue Implementation Using Deque/List

```
from collections import deque

class listQueue:
    def __init__(self):
        self._queue = deque([])

    def isEmpty(self):
        return not self._queue

    def enqueue(self, item):
        self._queue.append(item)

    def dequeue(self):
        return self._queue.popleft()

    def __repr__(self):
        return self._queue.__repr__()

    def __str__(self):
        return self._queue.__str__()
```

ADT Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority value. Elements are served on the basis of their priority. Higher priority elements are served first. Elements with the same priority are served according to their order in the queue. Priorities can be encoded with keys.

A priority queue is a frequently used data structure for

- Dijkstra's shortest path algorithm
- Prim's Minimum Spanning Tree algorithm
- Event-driven simulation
- Huffman encoding
- Heapsort

ADT Priority Queue Interface

`INSERT(PQ, x)`	insert element **`x`** into priority queue
`MIN(PQ)`	return element with the smallest key value
`DEL_MIN(PQ)`	remove and return minimum element
`DECREASE_KEY(PQ, x, k)`	decrease the key value of element **`x`** to **`k`**
`UNION(PQ1, PQ2)`	creates a new priority queue that contains all the elements from **`PQ1`** and **`PQ2`**

A similar interface can be made for a maximum-priority queue.

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap
MAKE-HEAP	1	1	1	1
INSERT	1	Log N	Log N	1
MIN	N	1	Log N	1
DEL_MIN	N	Log N	Log N	Log N
UNION	1	N	Log N	1
DECREASE_KEY	1	Log N	Log N	1
DELETE	N	Log N	Log N	Log N
IS_EMPTY	1	1	1	1

Priority Queue Implementation with Linked List

A linked list is an unordered list of elements, but it could be used to implement a priority queue. In order to attach a priority to a value (or object), we need to store key-value pairs in a linked list, so we have a more granular priority than just list position.

An **INSERT** operation places a new element in the front of the list.

MIN/DELETE_MIN/DELETE operations all need to locate the element first, which results in $O(n)$ worst case run-time (since it is a linked list).

The union operation simply attaches the head of one list to the tail of another list in $O(1)$, but reordering would take $O(n)$. We could trade **MIN** $O(n)$ for **INSERT** $O(1)$ by inserting in order (key-ordered list).

Python Sample Code

```
class Node:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def Insert(self, data, priority=19):
        node = self.head
        self.head = Node(data, priority)
        self.head.next = node
```

```python
# delete by data
# will not work as intended on same data with different priorities
def delete(self, data):
    if self.head is not None:
        if self.head.data == data:
            if self.head.next is not None:
                self.head = self.head.next
            else:
                self.head = None
        else:
            node = self.head
            while node.next is not None:
                if node.next.data == data:
                    if node.next.next is not None:
                        node.next = node.next.next
                    else:
                        node.next = None
                node = node.next
    def __repr__(self):
        representation = "linked list [ "
        node = self.head
        while node is not None:
            representation += node.priority.__repr__()
            if node.next is not None:
                representation += ", "
            node = node.next
        representation += " ]"
        return representation

    def __str__(self):
        representation = "linked list [ "
        node = self.head
        while currentNode is not None:
            representation += f'({node.data.__str__()}, 
node.priority.__str__()})'
            if node.next is not None:
                representation += ", "
            node = node.next
        representation += " ]"
        return representation
```

```
def __iter__(self):
    node = self.head
    while node is not None:
        yield node
        node = node.next

# find minimum in unordered linked list
def Del_Min(self):
    if self.head is None:
        return None
    minNode = self.head
    minPriority = minNode.priority
    node = self.head
    while node.next is not None:
        if node.priority < minPriority:
            minNode = node
            minPriority = minNode.priority
        node = node.next
    data = minNode.data
    # double-linked list would be better
    self.delete(data)
    return data
```

```python
def Decrease_Priority(self, data, k):
    if self.head == None:
        return None
    prevNode = None
    node = self.head
    while node.data != data and node != None:
        prevNode = node
        node = node.next
    if node != None:
        node.priority -= k
```

```
class priorityQueue:
    def __init__(self):
        self.queue = LinkedList()
    def insert(self, data, priority):
        self.queue.Insert(data, priority)
    def Del_Min(self):
        return self.queue.Del_Min()
    def __repr__(self):
        return self.queue.__repr__()
    def __str__(self):
        return self.queue.__str__()
```

We can modify the Node class, insert and delete methods to meet our needs.

More Python Sample Code

```python
def Insert(self, data, priority=19):
    newNode = Node(data, priority)
    if self.head is None:
        self.head = newNode
    elif self.head.priority > newNode.priority:
        newNode.next = self.head
        self.head = newNode
    else:
        tempNode = self.head
        while tempNode.next != None and newNode.priority > tempNode.next.priority:
            tempNode = tempNode.next
        if tempNode.next != None:
            newNode.next = tempNode.next
            tempNode.next = newNode
        else:
            tempNode.next = newNode
```

ADT Mergeable Heap

Recall that a binary heap is a min-heap data structure implemented as an almost complete binary tree – that means it is filled on all levels, except possibly the last, where it is filled from left to right. Binary heaps are a standard way of implementing priority queues. A binary heap of height h (where root has height 0) has between 2^h and $2^{h+1} - 1$ nodes.

Since we have a min-heap, every child is greater than (or equal to) its parent. After an insert or delete operation, the structure may need to be heapified, which results in $O(\log(n))$ run-time. When using the union operation, a complete rebuilt of the heap is required, resulting in $\Theta(n)$ run-time. We have already dealt with these heaps.

If a priority queue does not require a reasonably fast union operation, then Binary Heaps are acceptable (otherwise we need ***mergeable*** heaps). A mergeable heap is another ADT which supports the same set of operations as priority queues. Mergeable heaps could be either MIN or MAX oriented. Examples of mergeable heap data structures include:

- Binomial heap
- Fibonacci heap

Binomial Heap (Data Structure)

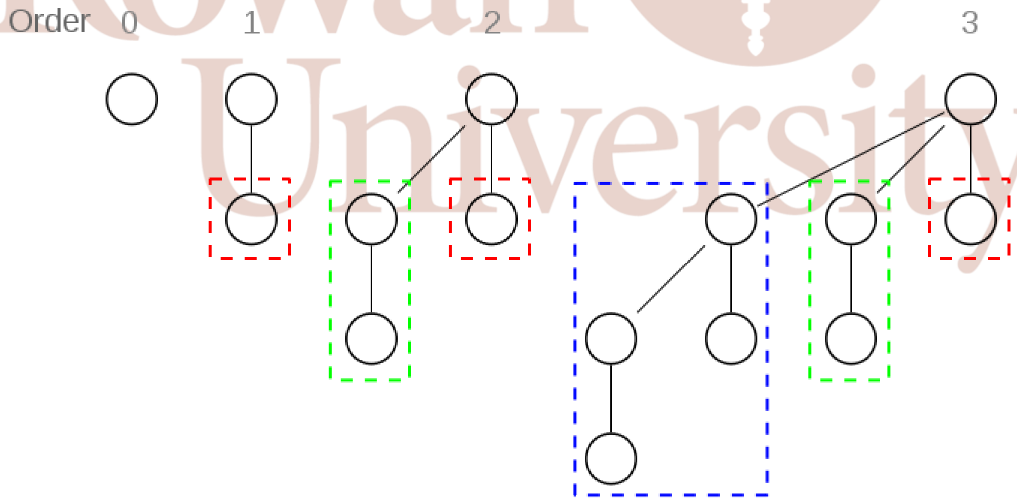
A ***binomial heap*** is not a heap-ordered tree but a *collection* of heap-ordered trees (a ***forest***). Each of the heap-ordered trees is a particular kind of tree, called binomial tree. Binomial trees are defined recursively, as we will see shortly. A binomial heap supports the operations

`MAKE_HEAP()`	build a new heap
`INSERT(H, x)`	insert node x into heap H.
`MIN(H)`	return node with the smallest key value
`DELETE_MIN(H)`	remove and return node with the smallest key value
`UNION(H1, H2)`	merge **`H1`** and **`H2`**, creating a new heap
`DECREASE_KEY(H, x, k)`	decrease **`x.key`** (**`x`** is a node in **`H`**) to **`k`**. (It is assumed that **`k`** $\leq$ **`x.key`**)

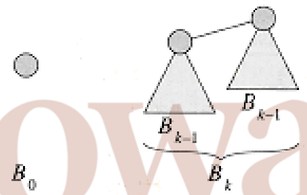
Binomial Trees

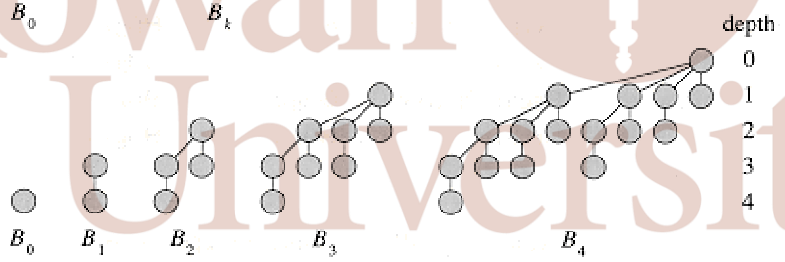
Binomial trees are trees defined recursively as follows:

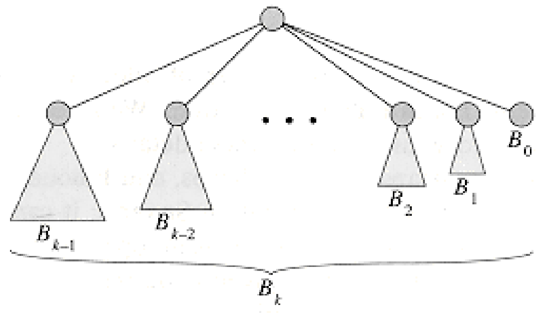
- A binomial tree of order 0 is a single node (size $2^{\text{order}} = 2^0 = 1$).
- A binomial tree of order k has a root whose children are binomial trees of order $k-1, k-2, \ldots, 2, 1, 0$ (in that order)



Every binomial tree must be of size 2^k for some k – no exception! There is at most one binomial tree of a given height k (the order). Binomial tree B_k consists of two binomial trees B_{k-1} linked together, and the root of one is the leftmost child of the root of the other.







There are some mathematical properties which follow for trees B_k of order k.

- The number of nodes is 2^k
- The height is k
- The number of nodes at depth i is $\binom{k}{i}$
- Deleting root yields binomial trees $B_{k-1}, B_{k-2}, \dots, B_0$
- Root has degree k, which is greater than that of any other node
- If the children of the root are numbered from left to right with $k-1$, $k-2$, …, 0 then ith child of root is the root of subtree B_i
- The maximum degree and maximum depth in an n-node binomial tree is $\log(n)$

Proofs by Induction
Number of Nodes

- Base case is $k = 0$ and $B_0 = 1$.
- Inductive hypothesis is $\text{size}(B_{k-1}) = 2^{k-1}$.
- Inductive step:

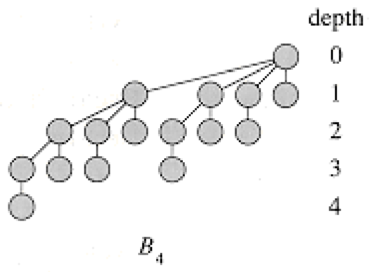
  We are joining two B_{k-1}: $\text{size}(B_k) = 2 \cdot \text{size}(B_{k-1}) = 2 \cdot 2^{k-1} = 2^k$

Height

- Base case is $k = 0$ and height$(B_0) = 0$.
- Inductive hypothesis is height$(B_{k-1}) = k - 1$.
- Inductive step:

  We are joining two B_{k-1} where one copy is a level lower than the other. height$(B_k) =$ height$(B_{k-1}) + 1 = k - 1 + 1 = k$

Number of Nodes at Depth



First an example: B_4 has 16 nodes and 6 are at level 2.

Let $D(k,i)$ denote the number of nodes at depth i in B_k.

Since B_k is two copies of B_{k-1} with one at a lower level, we get a recursive formula: $D(k,i) = D(k-1,i) + D(k-1,i-1)$. Once we have the correct initial values, this will be Pascal's Triangle!

- Base cases are $k = 0, i = 0$ and $k = 1, i = 0,1$.
- Inductive hypothesis is $D(k-1,i) = \binom{k-1}{i}$ and $D(k-1,i-1) = \binom{k-1}{i-1}$.
- Inductive step: recursive formula matches Pascal's Triangle.

A binomial heap is a set of binomial trees satisfying the binomial-heap properties ***heap order*** and ***uniqueness of binomial trees in the heap***.

- Each binomial tree in a set obeys the min-heap property. Every node's key is greater than or equal to the key of its parent.
- For any non-negative integer k there is at most one binomial tree in H whose root has degree k.

Uniqueness of Binomial Trees in the Heap

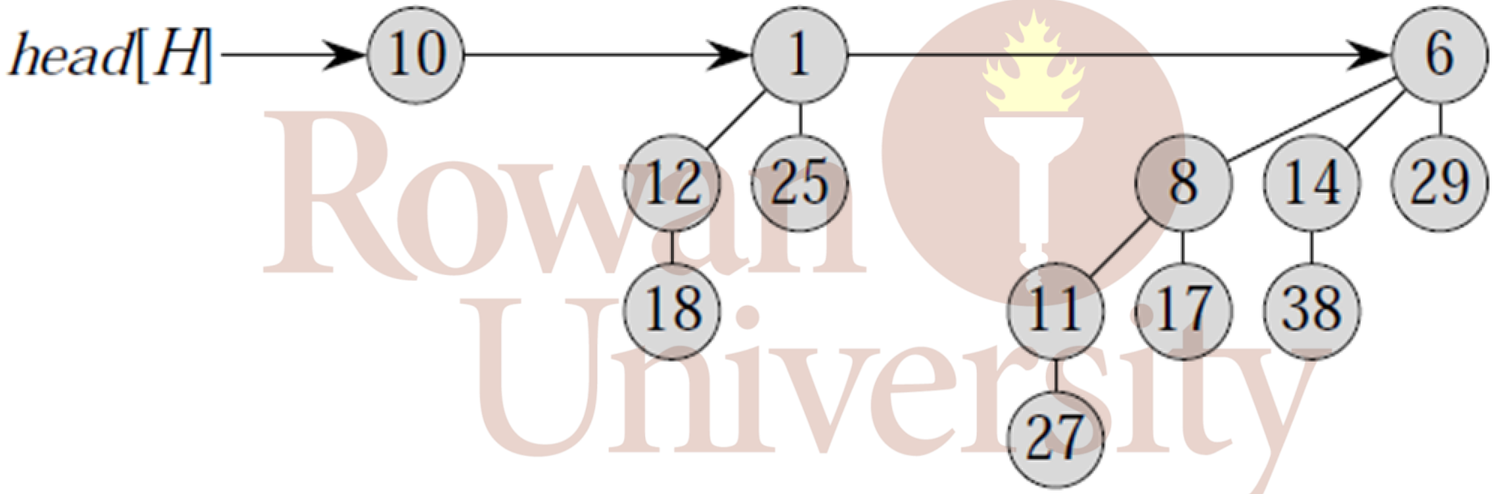
Any n-node binomial heap consists of at most $\lfloor \log(n) \rfloor + 1$ binomial trees. Binomial trees can be represented as a binary number (or characteristic vector) where each B_i that is part of the heap is represented as 1 and those that are not as 0:

$$\langle b_{\lfloor \log(n) \rfloor}, \dots, b_0 \rangle$$

b_i is encoding the presence of B_i.

$$n = \sum_{i=1}^{\lfloor \lg n \rfloor} b_i 2^i$$

For example, $13_{10} = 1101_2$. A binomial heap H consisting of B_3, B_2, and B_0 has $2^3 + 2^2 + 2^0 = 13$ nodes.

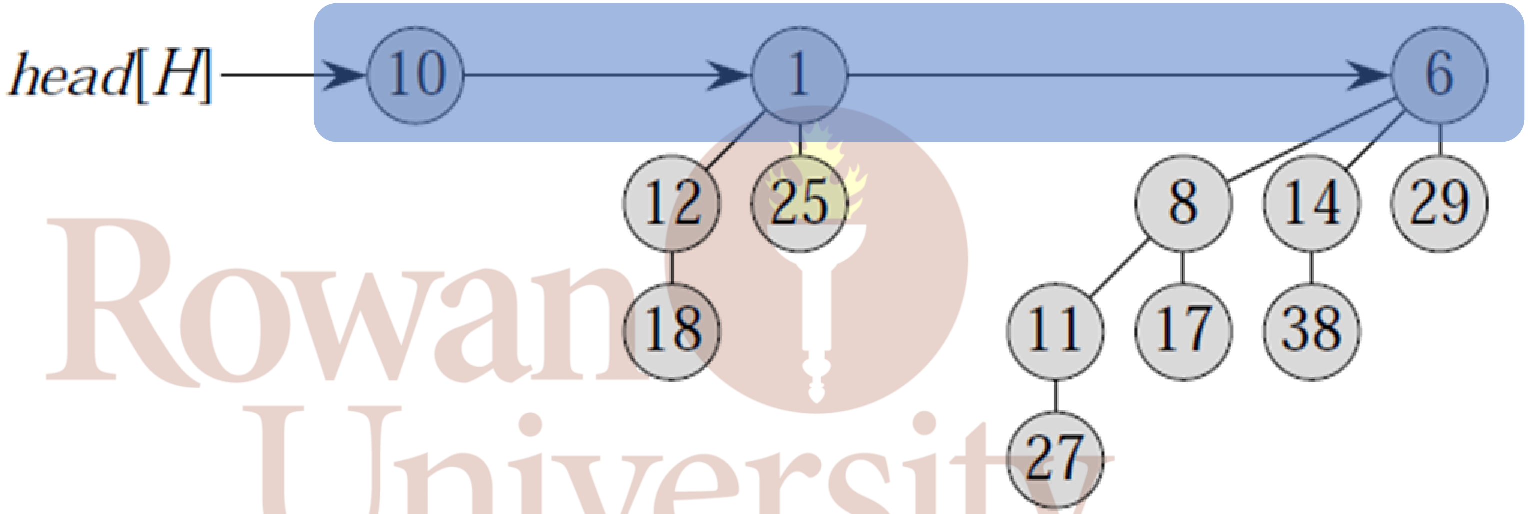


Binomial Heap Implementation

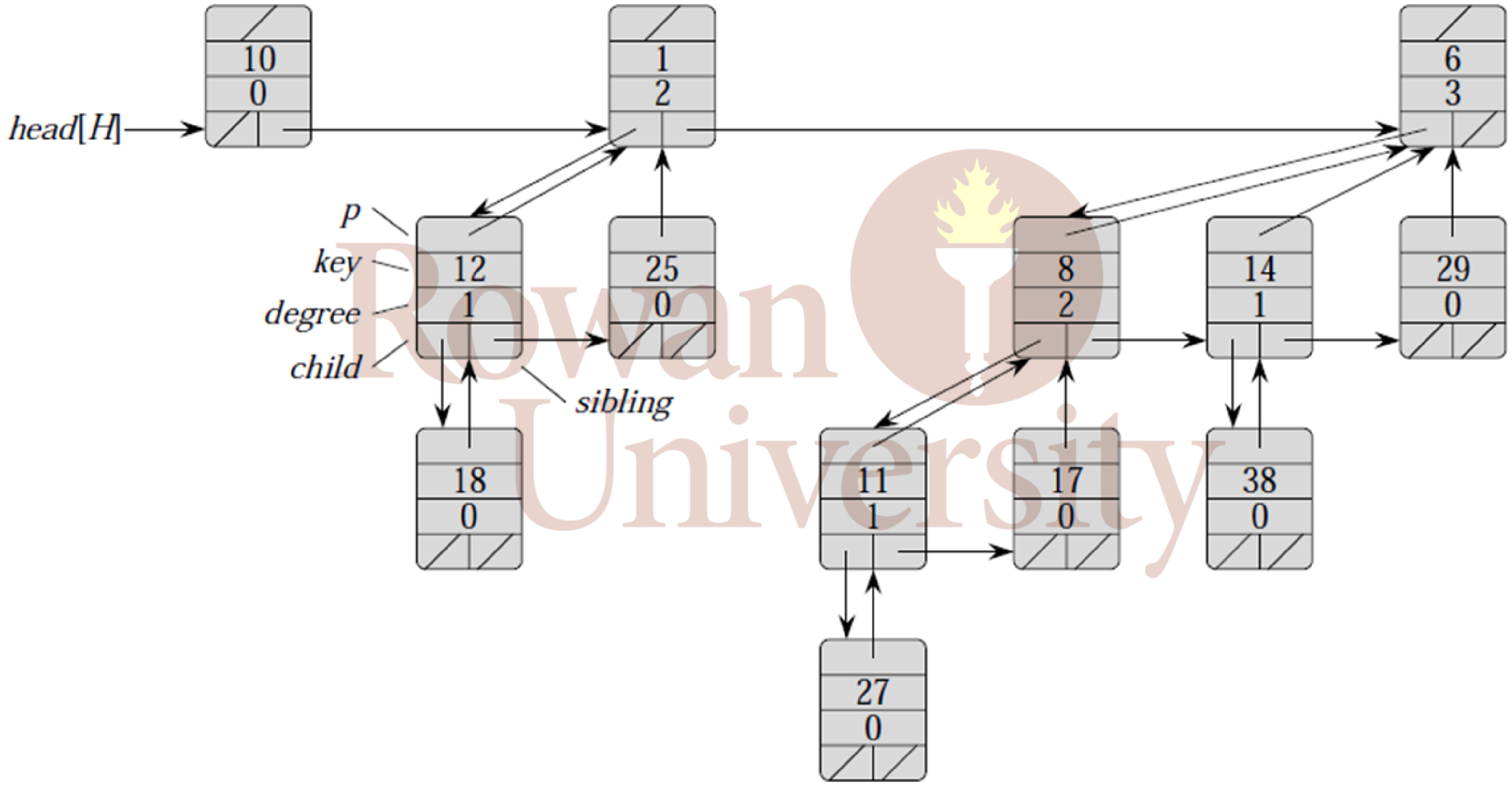
Since binomial trees cannot consolidate themselves without excessively linked data structures for parents and siblings, we will delegate most of the work to a **BINOMIALHEAP** class. The **BINOMIALTREE** will consist of a root node (making it of size at least 2^0) and children which in turn are roots of binomial trees.

```python
class BinomialTree:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.children = []
        self.order = 0

    def append(self, tree):
        if self.order==tree.order:
            # should consolidate
            pass
        self.children.append(tree)
        self.order = self.order + 1

    def __str__(self):
        return f"B({self.order})"

    def __repr__(self):
        return f"B({self.order})"
```

The heap is represented as a list of roots of binomial trees:



Our book (2nd edition) uses a multi-link approach:



Min

We search through the root list until we find the smallest element. Each minimum of a binomial tree is the root of that tree, and the global minimum must be in one of those trees – hence the minimum is the root of one of the binomial trees. The run-time will be $O(\log(n))$ (number of trees).

```
def get_min(self):
    min = None
    for root in self.roots:
        if root.priority < min:
            min = root.priority
    if min is not None:
        return min.data
    else:
        return None

def delete_min(self):
    if self.roots == []:
        return None
    min_tree = self.roots[0]
    for root in self.roots:
        if root.priority < min_tree.priority:
            min_tree = root
    heap = BinomialHeap()
    heap.roots = min_tree.children
    self.roots.remove(min_tree)
    self.merge(heap)
    self.consolidate()
    return min_tree.data
```

Insert

We insert an element as a binomial tree of order 0 (size $2^0 = 1$).

```
def insert(self, data, priority):
    heap = BinomialHeap()
    heap.roots.append(BinomialTree(data, priority))
    self.merge(heap)
```

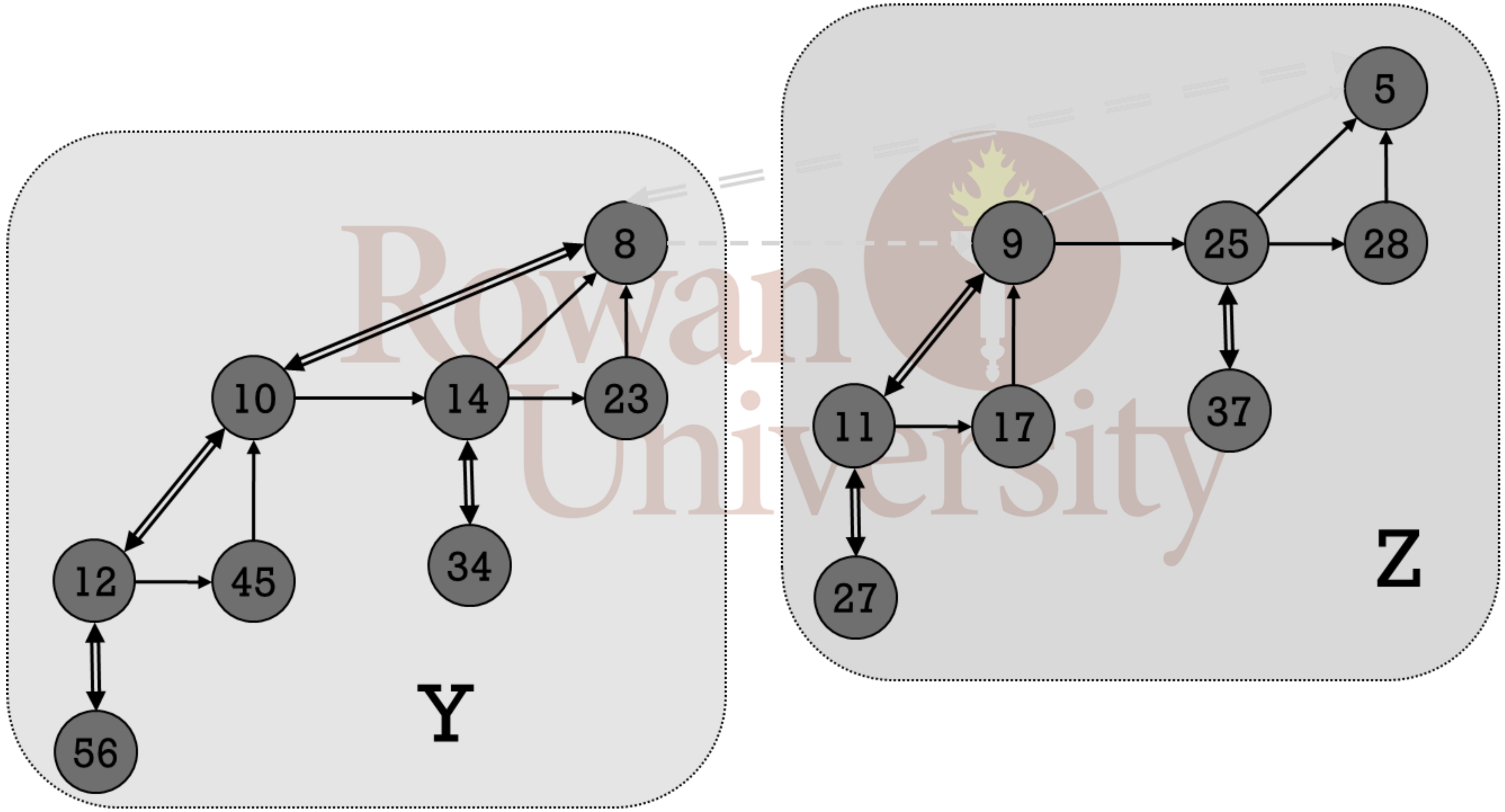
All the work will be in the **Merge** method, which needs to merge trees of the same size (**Link** B_{k-1} with a B_{k-1} into a B_k) and consolidate trees if necessary. Recall that one of the properties of binomial heaps is the uniqueness of binomial tree sizes in its forest.

Linking

We want to link two B_{k-1} to create a B_k while keeping heap order. Since each tree is already min heap ordered, we just need to make the smaller of the two roots the new root (append to the children list), and the other the parent. The new root will have an increased degree (and the new tree double the size).

```
LINK(y, z)
    p[y] = z
    sibling[y] = child[z]
    child[z] = y;
    degree[z] = degree[z] + 1
```

Increase degree of
node with key = 5



Union/Merge

The union of two binomial heaps is analogous to binary addition! Each binomial heap can be represented as binary number $\langle b_n, b_{n-1}, \ldots, b_0 \rangle$ where b_i is either 1 or 0 depending on whether binomial tree B_i is in the heap H. Say we have H_1 with B_0, B_1, B_2 and H_2 with B_0, B_1, B_4.

caries	111
H_1	00111
H_2	10011
sum	11010

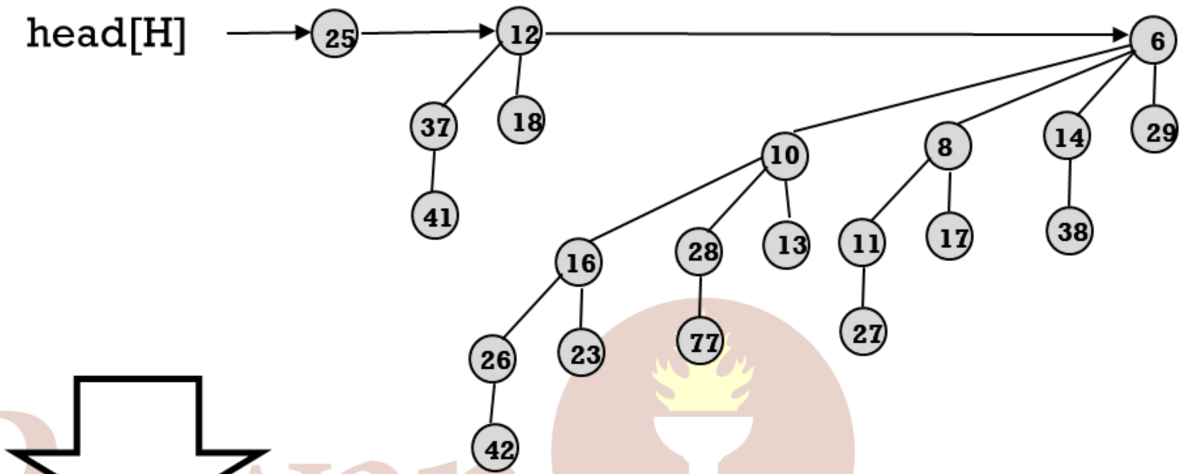
We temporarily have three trees.

```python
def merge(self, heap):
    self.roots.extend(heap.roots)
    self.roots.sort(key=lambda root: root.order)
    if self.roots == []:
        return
    repeat = True
    while(repeat):
        i = 0
        while i < len(self.roots) - 1:
            current = self.roots[i]
            next = self.roots[i + 1]
            if current.order == next.order:
                if (i + 1 < len(self.roots) - 1 and self.roots[i + 2].order == next.order):
                    next_next = self.roots[i + 2]
                    if next.priority < next_next.priority:
                        next.append(next_next)
                        del self.roots[i + 2]
                    else:
                        next_next.append(next)
                        del self.roots[i + 1]
                else:
                    if current.priority < next.priority:
                        current.append(next)
                        del self.roots[i + 1]
                    else:
                        next.append(current)
                        del self.roots[i]
            i = i + 1
        # do we need to consolidate?
        orders = [root.order for root in self.roots]
        repeat = (len(orders) != len(set(orders)))
```

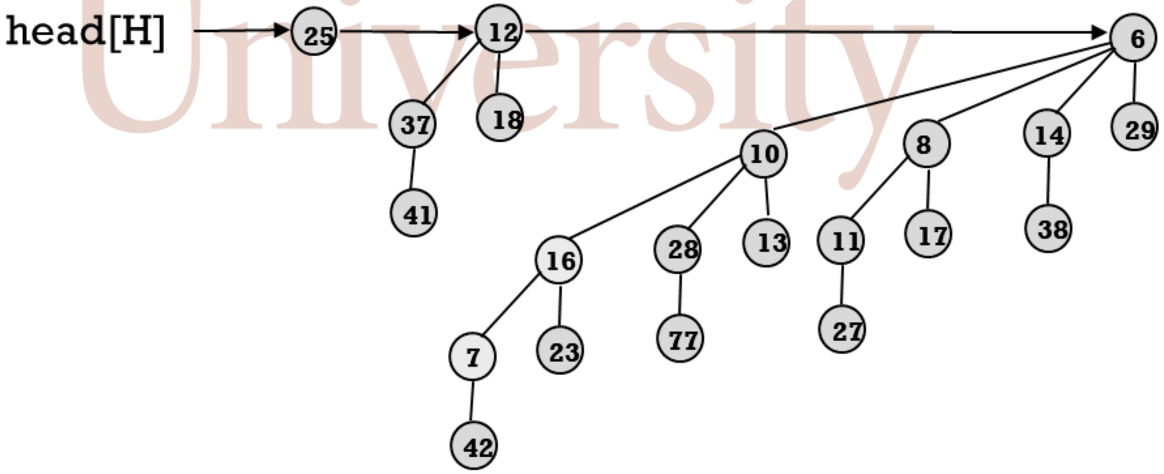
Decrease Key

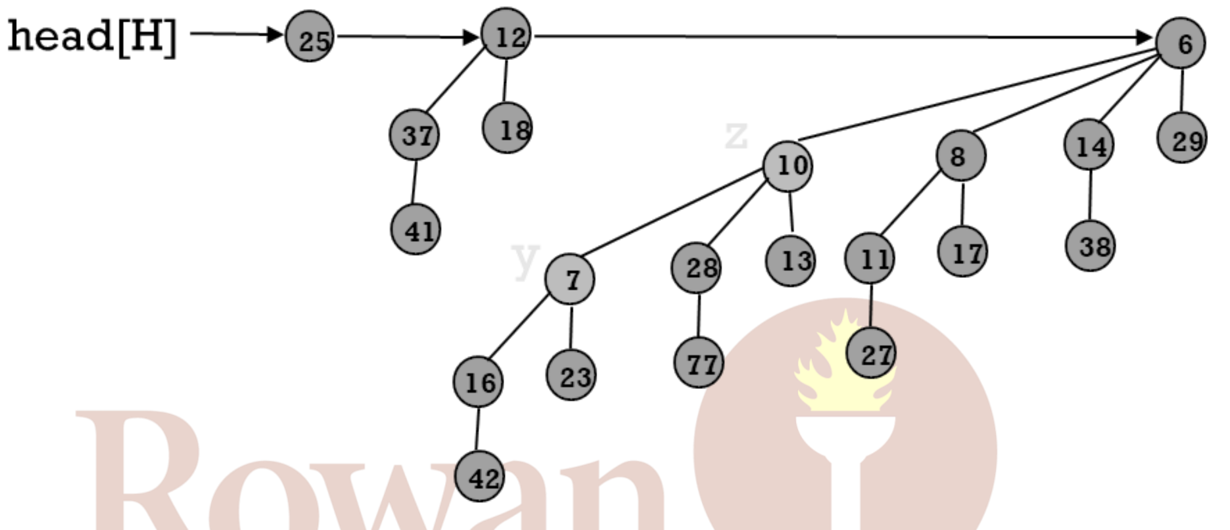
We want to decrease the key of node **x** in binomial heap **H**. Suppose **x** is in binomial tree B_k. Move/bubble node **x** up the tree if **x** is too small. Running time is $O(\log(N))$. It is proportional to the depth of node **x**.

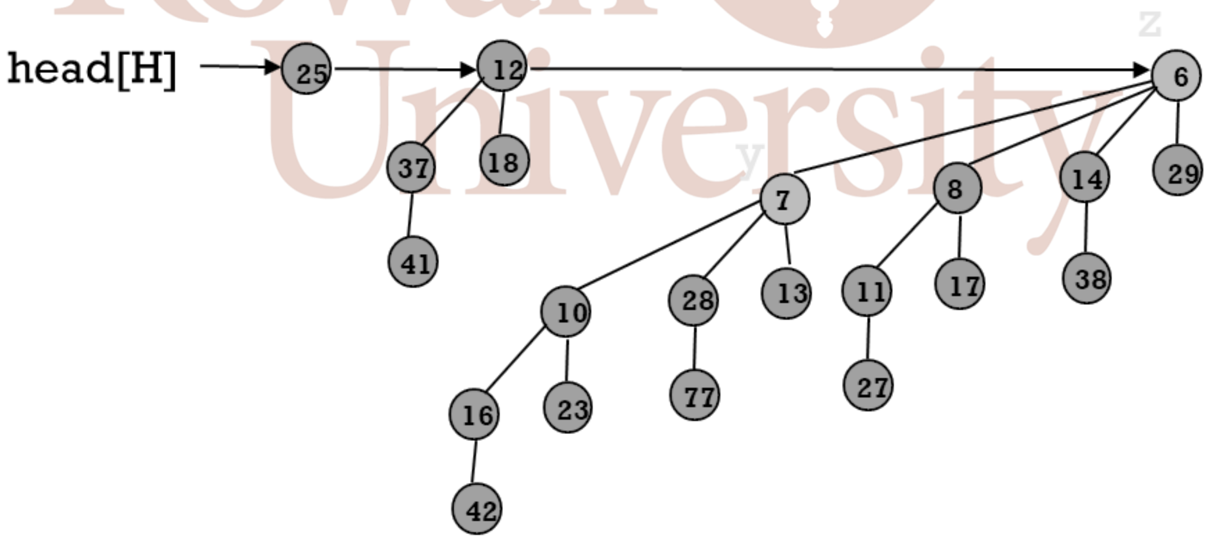
```
Decrease_Key(H, x, k)
   if k > key[x] then "error"
   key[x] = k
   y = x
   z = p[y]
   while (z != NIL && key[y] < key[z])
      swap key[y] and key[z]
      y = z
      z = p[y]
```



Decrease key 26 to 7




head[H]
25
12
6
37
18
41
z
10
8
14
29
y
7
28
13
11
17
38
16
23
77
27
42


head[H]
25
12
6
z
37
18
41
y
7
8
14
29
10
28
13
11
17
38
16
23
77
27
42

Amortized Analysis of Sequence of Inserts

Insert a new node **`x`** into binomial heap **`H`**.

- If $N = \ldots\ldots.0$, then we only need 1 step.
- If $N = \ldots\ldots 01$, then we only need 2 steps.
- If $N = \ldots.011$, then we only need 3 steps.
- If $N = \ldots 0111$, then we only need 4 steps.

Inserting 1 item can take $\Theta(\log(N))$ time.

- If $N \ = \ 11\ldots111$, then $\log_2(N)$ steps.

$$\sum_{n=1}^{N} \frac{n}{2^n} = 2 - \frac{N}{2^N} - \frac{1}{2^{N-1}} \le 2$$

What about average (amortized analysis)?

$$\left({}^{N}\!/_{2}\right) \cdot 1 + \left({}^{N}\!/_{4}\right) \cdot 2 + \left({}^{N}\!/_{8}\right) \cdot 3 + \cdots \le 2N$$

A sequence of N inserts takes $O(N)$ time amortized! You only need to perform union operation under certain conditions, though.

Summary of Priority Queues and Binomial Heaps

- Priority Queue is an ADT
  - Used for Minimal Spanning Tree and Shortest Path Algorithms
  - Often implemented using a heap
- Binomial heap is a data structure which can be used to implement a Priority Queue
  - Improved **MERGE**/**UNION** method ($O(\log(n))$ vs $O(n)$ for binary heaps)